University of Strathclyde
Department of Computer and Information Sciences

Problem Models for Rule Based Planning

by
Alan Lindsay

A thesis presented in fulfilment of the requirements for the degree of

Doctor of Philosophy

2015

Declaration

This thesis is the result of the author's original research. It has been composed by the author and has not been previously submitted for examination which has led to the award of a degree.

Signed:

Date:

Acknowledgements

Firstly, I would like to thank Candice Spencer, and my family: Caroline, Norman and Michael Lindsay, for their love, patience and support.

Thanks to my supervisors, Maria Fox and Derek Long, for their deep knowledge and for providing an excellent example of how research should be conducted.

I acknowledge the support of my friends and colleagues at Strathclyde University. Thanks to Alastair Andrew and Peter Gregory, for being clever rocks. Thanks to Dave Bell, Tommy Thompson, David Pattison and the wider planning group, for a spectrum of ideas and probing debate. I have also been supported in various ways by the teaching and non-teaching staff in the department, especially the system support team. Thanks to the disability service, past and present members, who not only employed me but also became my extended family. And, thanks to the running club for providing a necessary distraction, and a welcome source of *achievable* challenges.

I would like to give special mention to my two examiners, Chris Reed and John Levine, and the convenor, Mark Dunlop, who gave me an engaging and thorough examination. I also acknowledge Ian Ruthven's help in organising my viva. And, of course, thanks to my proof-readers: Peter, Alastair and Julie Porteous (and Dad).

Lastly, thanks to the IVE group at Teesside for putting up with me during the final stages of my write-up.

CONTENTS

List of Figures

List of Tables

Abstract

The effectiveness of rule-based policies as a search control mechanism in planning has been demonstrated in several planners. A key benefit is that a single policy captures the solution to a set of related planning problems. However, it has been observed that a small number of weak rules, common in learned control knowledge, can make a rule system ineffective. As a result, research has focussed on approaches that improve the robustness of exploiting (potentially weak) rules in search.

In this work we examine two aspects that can lead to weak rules: the language that the rules are drawn from and the approach used to learn the rules. The rules are often captured using the predicates and actions of the problem models that the knowledge applies to. However, this language is appropriate for expressing the constraints of the planning world, and will not necessarily include the appropriate words required to express a general solution. We present an approach to automatically invoke language enhancements that are appropriate for the particular aspects of the target problems. These enhancements support rules in problems that include structure interactions, such as graph traversal and block stacking; and optimisation tasks, such as resource management.

There have been several approaches made to learning policies explored in the literature. Learning policies requires a fitness function, which measures the quality of a policy. In previous approaches these have relied on a collection of examples generated by a remote planner. However, we have observed that this leads to weak guidance in domains where global optimisation is required for an optimal solution (such as transportation domains). In these domains we expect good, but not optimal action choices, and this conflicts with the assumption that example states can be accurately explained and ultimately leads to weak rules. Instead of measuring performance from a set of remotely drawn example situations, we propose using progress towards goal instead.

Our approach is evaluated using rule-based policies to control search in problems from the benchmark planning domains. We demonstrate that domain models can be automatically enhanced and that this enhanced language can be exploited by both hand-written and learned policies allowing them to effectively control search. The learning approach is evaluated by learning policies for several of the enhanced domains and it is analysed providing guidance for future work. A key contribution of this work is demonstrating that both hand-written and learned rule-based policies can be used to generate plans that have better quality than domain independent planners. We also learn effective policies for several domains currently untreated in the literature.

CHAPTER 1

INTRODUCTION

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

The Humble Programmer (1972)
EDSGER DIJKSTRA

An important influence on the way that we express ourselves are the languages that we are taught. These languages have been developed for hundreds of years and are functional in allowing us to communicate anything from specific commands to our dreams for the future, all in the context of a complex and dynamic environment.

There are many specialised vocabularies that have been developed that extend this vocabulary in particular areas. Specialised words allow ideas that are too complex or abstract to be expressed in a day to day vocabulary. For example, there are numerous mathematical terms, such as the language of graph theory and in psychology there are specialised vocabularies for expressing states of the brain and patterns of behaviours. These vocabularies have been constructed so that communication in these fields can be carried out concisely and with accuracy.

If we want to communicate a strategy for solving a group of problems then we might rely on words that are specific to the type of problem. For example, a transportation problem involves redistributing packages amongst various locations. A solution to this problem might refer to paths between locations; allocation of drivers; specialist carrying capabilities; or distribution hubs. This vocabulary allows the strategy to be described at a level that is abstracted from the language of a particular problem.

This thesis is concerned with selecting a suitable vocabulary for describing a strategy for a problem. The problems that we investigate are *planning problems* and the strategies are *generalised policies* that allow us to capture the solutions to groups of planning problems.

1.1 Automated planning

Automated Planning is an area of Artificial Intelligence that explores automating the process of planning, a function necessary for intelligent machines. Planning starts with a desire for change in the world. The process of planning is then the deliberation of actions that could be made in the world, based on the expected outcome of the actions and the desired change or goal. Actions are selected and ordered, forming a strategy for achieving the goal.

Humans are (arguably) inherently good at planning. It is seldom the case that we will notice ourselves actively deliberating alternative action sequences and yet we often achieve our final goals efficiently, especially as a task becomes more familiar. This is surprising considering the complex nature of the problems that we often solve.

Consider that you have made your list of Christmas gifts and you want all of the shopping out of the way. The list of gifts provides your goal; to achieve your goal you must consider which shops to visit and in what order. You may decide to head out into the high street and systematically search through every shop until you find the gifts. However, it is likely that you have some idea of where you will find the items on the list. You could form a plan of the shops you wish to visit and order them in such a way as to maximise the chance of getting all of the goods in the shortest period of time.

There are many conflicting factors that we weigh up to select and order the shops to visit. For example, the distance between shops is used to focus on particular groups of shops that are reasonably close and also to make efficient paths between the shops we do choose. However, the shape and size of the gifts might influence this path, as it is often desirable to pick up large and heavy items towards the end of a shopping trip. Also, we might use specific knowledge about the streets, such as how steep the streets are or how likely we are to be stopped by someone doing a questionnaire. All these considerations contribute to our final selection and ordering.

Making a plan like this is made possible through a developed understanding of objects and the environment. This allows us to decide when it is important to be accurate and use all of the details that we know and when it is permissible to reason more abstractly over the relationships in the world. The ability to change the level of context is one of the reasons why we are so effective at planning.

1.1.1 Automating the planning problem

In order to solve a planning problem on a computer, we need two things: a formal description of the world in some language and a planning algorithm. The way a problem is described can have implications on how useful the computed plans will be. This description will provide the computer with its entire view of the world and so it is crucial to provide all of the details that will be necessary for solving the problem. For example, in the Christmas shopping problem, if we describe the shops and the items that they sell, but fail to capture any of the spatial relationships between the shops, then the planning algorithm has no way of knowing that one shop is close to another shop in a particular shopping area. The computed plan could easily have you running back and forth from one side of town to the other.

Describing a planning problem

The Planning Domain Description Language, PDDL (McDermott, 2000), the de-facto standard description language in the field of Automated Planning, can capture many interesting planning problems. One of the key elements of describing a planning problem in this language is expressing the relationships that exist between objects in the world. In PDDL the relationships are represented by logical propositions that are expressed with a predicate symbol and constant symbols. In the previous shopping example a relationship exists between the shops and the items they sell and this might be modelled by the proposition (`sells item shop`) in the planning model, where `sells` is a predicate symbol and `item` and `shop` are both object symbols. The propositions are used to constrain the application of the actions that cause change to the environment. For example, a `buy-item` action could be constrained by propositions such as (`in shopper shop`) and (`sells shop item`).

When describing a planning problem, we must select a language to model the problem. The intention is to choose a language so that the dynamics of the world are captured concisely. This will depend on the expressivity of the modelling language. We observe that this language is also used by the planner to reason about the planning problem. However, there can be a distinction between the appropriate language for expressing the dynamics of a problem and the language for making decisions about appropriate actions.

Solving a planning problem

A naive strategy to solving a planning problem is to start with the initial world and explore any possible sequence of actions that is allowed by the action constraints. However, as the general planning problem is PSPACE-hard (Bylander, 1992), which means that it can be as difficult to solve as any problem that is solvable in polynomial space, this approach quickly becomes infeasible. Several strategies have been investigated that start from the initial state and construct the plan forward until reaching a state that satisfies the goal (e.g., Bonet and Geffner, 1998; Bacchus and Kabanza, 2000); however, they introduce a variety of techniques to prioritise certain action choices over others and in some cases remove action choices entirely. In this work we add to a body of work that uses control knowledge to capture good action choices.

1.2 Control knowledge

Control knowledge is compiled information that provides some form of guidance. It can be used to focus the planner towards certain courses of action (Veloso et al., 1995), to capture solutions to sub-goals (Laird et al., 1986; Newton et al., 2007), or provide an entire strategy for solving the planning problem (Khardon, 1999a; Bacchus and Kabanza, 2000). The TLPLAN system (Bacchus and Kabanza, 2000) has demonstrated that the use of control knowledge in a planning framework is a highly effective approach to planning. The system provides a flexible solution to the planning problem. It was the top performer and indeed one of the winners of the third planning competition (Long and Fox, 2003). It still provides unbeaten performance in terms of time to form plans and also in terms of the quality of the plans it creates for many planning problems.

The control knowledge used by TLPLAN in the competition was constructed by hand. This is expensive as it must be written by someone who intimately understands the dynamics of the domain and understands how to construct an effective control knowledge system. Inspired by the strong performance of TLPLAN, many researchers have investigated the possibilities of learning control knowledge. For these purposes researchers have exploited Machine Learning techniques. Although these approaches are generally computationally expensive, they have been used to solve individual planning problems (Westerberg and Levine, 2000), or improve their fitness (Gerevini et al., 2003). However, an important aspect of the TLPLAN control knowledge is that it encodes control knowledge that is appropriate for the problem distribution of a particular domain (a collection of related problems). This justifies expending the substantial resources often required for Machine Learning approaches.

The key to expressing control knowledge for groups of problems is the language that is used to express it. In particular, using vocabulary that captures the key concepts in the problem precisely. Currently learning technology limits the languages that can be used to learn control knowledge and as a result the correct words cannot be constructed.

1.2.1 A language for learning

In this work we have developed a framework for extending the vocabulary of planning problems. This allows decisions to be captured at an abstraction level that is appropriate for the action choices being made. The levels of abstraction are provided by automatic problem enhancements.

Allowing similar mixed-level reasoning was achieved in control knowledge for TLPLAN. The novelty in our approach is that the vocabulary is enhanced automatically and the control knowledge is expressed in a very limited language. We extend the current rule learning technologies and demonstrate that our vocabulary can be exploited by learned control knowledge.

1.3 Structure of thesis

The work is presented in 10 Chapters. Chapter 2 presents a background of problem modelling and search for planning; as well as control knowledge and learning. The chapter concludes with the statement of thesis. In Chapters 3 and 4, we set up the main framework for the first part of the work. Chapter 3 presents the framework for examining alternative problem models, and how we can use plans for these alternative problem models as plans for the original problem. In Chapter 4, we define the control knowledge representation that we focus on in this work, including observations of its particular limitations. In Chapter 5, the implementation of our framework is presented, we interpret the limitations of the control knowledge representation in terms of our model enhancement approach and define a library of appropriate model enhancements. We conclude this part of the work, in Chapter 6, by evaluating the system using handwritten control knowledge.

The second part of the work (starting in Chapter 7), concerns automating parts of the process. Chapter 7, begins by discussing our approach to automatically invoking appropriate enhancements, from the enhancement library defined in Chapters 5 and 6, for a specific planning domain. The chapter continues with our approach that generates appropriate elements for the library automatically. In Chapter 8, we move onto the problem of learning control knowledge. We present the current approaches and highlight a limitation in the common fitness function[1]. We present our alternative fitness function that tackles these limitations. We conclude the chapter by presenting a fast approach that generates incomplete control knowledge, which can be used for seeding learning. Chapter 9, presents our evaluation of the system. We examine the learned enhancements, the generated seeds and learned control knowledge. The learning approach is examined both from scratch and as a finishing step applied to the seed control knowledge. The contributions, future work and conclusions are presented in Chapter 10.

In the proceeding chapters we will use labelled definitions for named definitions.

[1]The fitness function provides a measure of the *goodness* of control knowledge.

CHAPTER 2

BACKGROUND

Planning is an important aspect of any intelligent agent. A large body of Automated Planning research has grown since the initial stimulus in the sixties (Newell and Simon, 1963). The introduction of regular international planning competitions (IPCs) in 1998 (McDermott, 2000) has been key in bringing the community together; providing inspiration and direction. Since then there has been substantial progress in planning: the development of a series of standardised languages for expressing planning problems (McDermott, 2000; Fox and Long, 2003; Edelkamp and Hoffmann, 2004; Gerevini and Long, 2005; Geffner, 2000); numerous approaches to solving planning problems (Fabiani and Meiller, 2000; Kautz and Selman, 1996; Bonet and Geffner, 1998; Richter et al., 2008; Khardon, 1999a; Culberson and Schaeffer, 1998; Ghallab et al., 2004); theoretic results in both complexity of problem solving (Bylander, 1992; Helmert, 2001) and the computation of heuristics (Hoffmann, 2005; Helmert and Domshlak, 2010; Hoffmann, 2011).

This work investigates learning control knowledge that captures a strategy for solving sets of planning problems. In particular, we rely on previous work in planning, search, control knowledge representation and machine learning. In this chapter we introduce each of these aspects and point out some of the main bodies of research that have led to the work described here. The main analyses of related works are distributed amongst the following chapters. We conclude this chapter with the statement of thesis.

2.1 Modelling

Modelling is the process of capturing the dynamics of the problem space in a formal representation. The planning problem has been modelled in various different ways (PDDL (McDermott et al., 1998), SAS+ (Bäckström and Nebel, 1993), SAT (Kautz and Selman, 1998), or CSP (Gregory et al., 2010)), each with its own properties. A model should capture the important rules of the problem so that a solution to the modelled problem corresponds to a solution to the real problem.

The model gives the planner access to the planning problem. However, the planner might be unable to plan using the presented model. In particular, it might be appropriate to remodel the problem and present the remodelled problem to the planner.

In this section we present some of the models that have been used to represent the planning problem. We then discuss approaches to remodelling. Ghallab et al. (2004) provide a more complete coverage of the adopted approaches of modelling planning problems.

2.1.1 Modelling a planning problem

In this subsection we define a *state transition system*, a general model for planning. We then present the specific proposition based representation adopted in this work. We conclude by discussing some of the research that has investigated the modelling process.

State transition systems

A state transition system (Dean and Wellman, 1991) is a general model for dynamic systems. We use a restricted definition here as there are no events in the systems that we are considering and action application is deterministic.

Definition 2.1.1 *A state transition system is a triple, $\tau = (\mathbb{S}, \mathbb{A}, \gamma)$, where:*

- $\mathbb{S}$ *is a finite set of states;*
- $\mathbb{A}$ *is a finite set of actions;*
- $\gamma : \mathbb{S} \times \mathbb{A} \to \mathbb{S}$ *is a mapping of states and actions to states.*

An example of a state transition system is depicted in Figure 2.1. There are two locations, a single truck and a single package. The truck can move between the two locations along the road between them and pickup or put down the package at its current

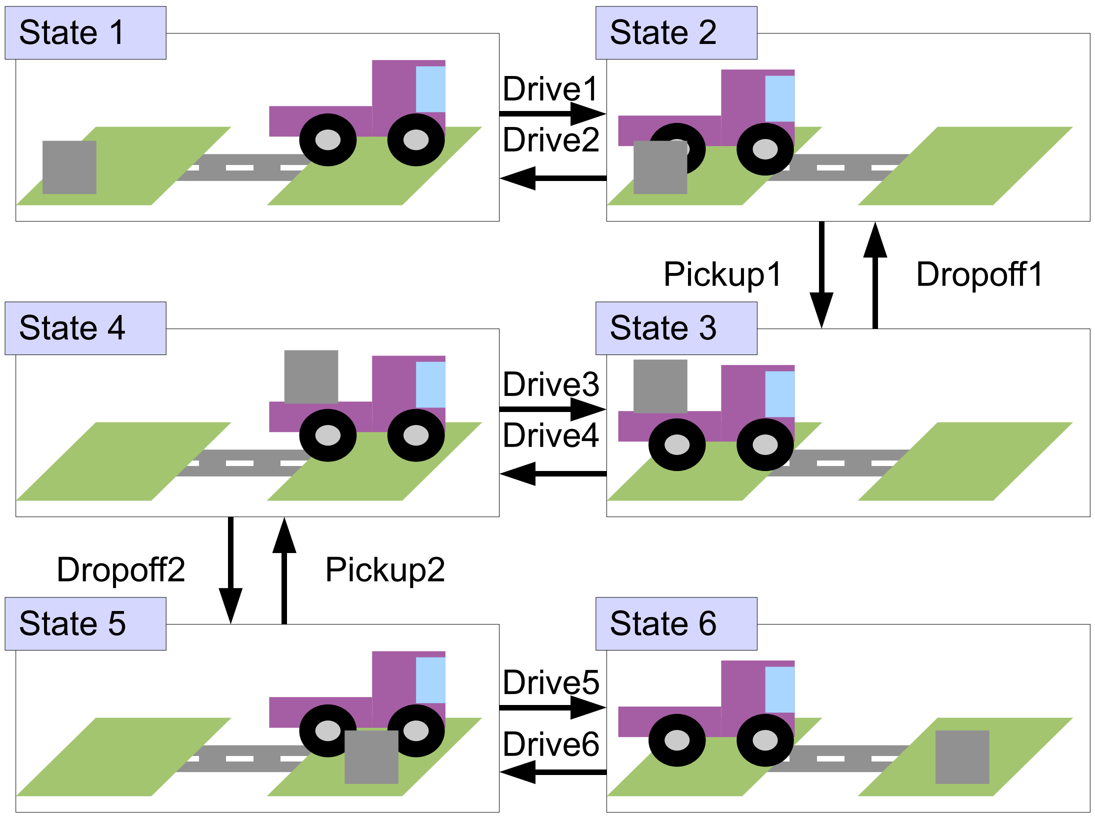


Figure 2.1: A state transition system for a two location transportation problem with one truck and one package.

location. The set of states in this example is $\mathbb{S} = \{$*State*$_1$,..., *State*$_6\}$ and the set of actions is $\mathbb{A} = \{$`Drive`$_1$, ..., `Drive`$_6$, `Pickup`$_1$, `Pickup`$_2$, `Dropoff`$_1$,`Dropoff`$_2\}$. The transition function, γ, is implied by the arcs that the actions describe.

A state system based planning problem

A state transition system, $\tau = (\mathbb{S}, \mathbb{A}, \gamma)$, can be used to capture the environment for a planning problem. The actions in $\mathbb{A}$ represent actions in the planning problem and the states in $\mathbb{S}$ represent the state of the environment. The transitions that can be made in the environment are captured as γ.

Definition 2.1.2 *A planning problem, $\mathbb{P}$, can be described by the triple, $\langle \tau, i, g \rangle$, where τ is the planning environment, $i \in \mathbb{S}$, is the initial state of the environment and g is a goal formula that defines the objectives of the problem.*

The purpose of a planner is to select a sequence of actions that transition from the initial state to a state that satisfies the goal formula.

The state transition system, $\tau_{Transport}$, depicted in Figure 2.1 can be interpreted as a simple planning environment, $\mathbb{P}_{Transport}$. An example of a planning problem for $\tau_{Transport}$ is described by the triple $\langle \tau_{Transport}$, *State*$_1$, package at second location $\rangle$. This is the problem with planning environment captured by $\tau_{Transport}$, with *State*$_1$ as the initial state of τ and the goal of moving the package to the second location. A solution to this problem is to apply the action `Drive`$_1$ in state *State*$_1$, `Pickup`$_1$ in *State*$_2$, `Drive`$_4$ in *State*$_3$ and `Dropoff`$_2$ in *State*$_4$. This changes the state from *State*$_1$ to *State*$_5$ and in this new state the goal of moving the package to the second location is achieved.

Set-theoretic planning

The states of τ can be interpreted with respect to a set of propositions. For example, the proposition that the truck is at `location`$_1$ in the states of $\tau_{Transport}$ is either true or false. If the set of propositions is well chosen then the states of τ can be uniquely identified by the propositions that the state entails. The set of propositions representing each package position: either (`at package`$_1$ `location`$_1$), (`at package`$_1$ `location`$_2$), or (`in package`$_1$ `truck`$_1$) and a proposition for each truck position: (`at truck`$_1$ `location`$_1$) or (`at truck`$_1$ `location`$_2$), uniquely distinguish the states in the transition system, $\tau_{Transport}$.

The actions change the state with the result that the propositions that hold (are true) in the new state are different from those that held in the original state. For example, the action between *State*$_1$ and *State*$_2$ transitions from *State*$_1$, where the proposition that the truck is at `location`$_1$ holds and the proposition that the truck is at `location`$_2$ does not hold, to *State*$_2$, where the proposition that the truck is at `location`$_2$ holds and the proposition of the truck being at `location`$_1$ does not hold.

The states of planning problems can be modelled by a collection of propositions and the actions can be modelled as the add and delete rules that effect the necessary additions and removals from the state to produce the new set of propositions that represent the new state. An action is represented by a triple:

- The name of the action. For example, (`pickup package`$_1$ `truck`$_1$ `location`$_1$), represents the action that puts `package`$_1$ into `truck`$_1$ at `location`$_1$.

- The precondition that determines whether the action is applicable in the current state. This is a set of propositions. The action is only applicable if all the preconditions hold. For example, the applicability of the action, (`pickup package`$_1$ `truck`$_1$ `location`$_1$), may depend on two propositions: (`at truck`$_1$ `location`$_1$) and (`at package`$_1$ `location`$_1$).

- The effects of the action, which are often separated into two sets: the add effects and the delete effects. As their names suggest the add effects are a set of propositions that are added to the current state, and the delete effects are the set of propositions that are removed from the current state. The pickup action might remove the single proposition, (`at package`$_1$ `location`$_1$) and add the proposition, (`in package`$_1$ `truck`$_1$).

We assume that any delete effects are part of the precondition (meaning there are no conditional effects). From a given state there are potentially several actions that could be selected. Applying each of these actions results in a changed state: precisely the state that is represented by the set of propositions obtained by removing the delete effects from the current state and then adding the add effects. A possible PDDL encoding of two $\mathbb{P}_{Transport}$ actions is shown in Listings 2.1 and 2.2.

Listing 2.1: PDDL representation of the `pickup` action

```
(:action pickup
    :parameters (package₁ truck₁ location₁)
    :precondition (and (at package₁ location₁)
        (at truck₁ location₁))
    :effect (and (not (at package₁ location₁))
        (in package₁ truck₁)))
```

Listing 2.2: PDDL representation of the `drive` action

```
(:action drive
    :parameters (truck₁ location₁ location₂)
    :precondition (at truck₁ location₁)
    :effect (and (not (at truck₁ location₁))
        (at truck₁ location₂))
```

The first action puts `package`$_1$ into `truck`$_1$ if they are both at `location`$_1$. The second action moves the truck between `location`$_1$ and `location`$_2$.

The *goal* of a planning problem in the set-theoretic representation is a set of propositions that need to be satisfied. Planning is the task of choosing a sequence of actions that affect the world to transform the initial state into a state that contains the propositions of the goal. Using this representation means that the set of states, $\mathbb{S}$, is implied by all states that are reachable from i by applying any sequence of actions. Therefore, it is not necessary that all the states are expressed up front; this can be important, as the size of the state space can grow quickly.

Planning domains In practice, the description of a planning problem is separated into two parts: the definition of the problem domain that defines the world and its behaviours; and an explanation of the specific problem to be solved within that world. The domain is a tuple, $D = \langle \mathscr{O}, P \rangle$, that defines the set of predicates, P, and operators, $\mathscr{O}$. The predicates define the predicate symbols that can be used in representing the states of the world. The operators are a collection of parameterised actions, which describe the possible behaviours in the world. These are parameterised actions that describe the parameterised sets of predicates that define the precondition and effects, similar to those defined above for propositions. A *ground action* is an operator whose variables have been unified with world constants. In the transportation problem above the predicates would be `at` and `in` and the operators would be `Drive`, `Pickup` and `Drop-off`. The problem model is specified with a tuple, $P = \langle \mathbf{O}, s_i, g \rangle$, with the set of objects, $\mathbf{O}$, and s_i and g as before. The reachable states can be enumerated starting by applying all of the unifications of the operators that have satisfied preconditions to the initial state and repeating this process from each discovered state.

Alternative plan representation

In plan-space planning a plan is a collection of action sequences represented as a set of partially ordered operators with binding constraints. The nodes of the search space in plan-space planning are not states of the world, but partially defined plans. The actions are plan editing operations that affect the partial plans. The planning problem is then the task of selecting appropriate plan modifying operators to transform a partial plan into a complete plan. Control knowledge has been used in approaches to plan-space planning. There are two main differences to planning problems modelled in these representations: the information that is made explicit, and the operations supported by the model. For a complete introduction refer to Ghallab et al. (2004).

Extensions to the language

PDDL (McDermott et al., 1998) is the standard modelling language used in planning and has been developed for over twenty years. It became established after it was used for the first IPC (McDermott, 2000) in 1998. Since then there have been several developments generally coinciding with subsequent IPCs (Fox and Long, 2003; Edelkamp and Hoffmann, 2004; Gerevini and Long, 2005; Geffner, 2000). In this work we use the original PDDL language (which is known as STRIPS, because it was inspired by the language used by a planner, STRIPS (Fikes et al., 1972)).

Hierarchical task networks (HTNs) provide a more expressive language for modelling planning problems (Erol et al., 1996), which models planning problem as a collection of tasks that are identified using a network of hierarchical decompositions. The goal of a problem is to complete a collection of partially ordered tasks. A collection of methods describes how a task is decomposed into a partial order of subtasks. Subtasks are recursively split until the subtask is an instance of an operator, as in the definition above. A solution is a list of instantiated operators that are consistent with the methods and orderings and satisfy the target tasks. A benefit of using an HTN is that it defines an explicit structure that explains how tasks are broken down and in some problems this can make control knowledge easier to interpret. This is supported in the exploitation of an HTN approach to several real-world problems (Nau et al., 2005).

Extensions to PDDL have provided increased expressivity; in particular, they have been focussed so that certain aspects of problems can be modelled more completely. Even so, in Dornhege et al. (2009), it is observed that it can be difficult to achieve the necessary detail in PDDL. They present examples of real world problems where important details must be abstracted in a PDDL model, resulting in a plan that is not directly executable. The language, PDDL/M, developed in Dornhege et al. (2009), closes the gap between the symbolic model and the real problem. PDDL/M computes an enriched state by employing modules that more closely model the real world. The modules are targeted and therefore more efficient and can communicate a view of the real world supporting the planner in making more informed choices.

However, the convention remains to model the physics of the problems, and not advice about solving them (McDermott, 2000). In this work we attempt to establish problem models that provide a rich environment that supports planning. We are not looking to present the solution to parts of the problem, as has been done in Fox and Long (2001); Dornhege et al. (2009), or a complete solution, as in Bacchus and Kabanza (2000); Nau et al. (2003), but to develop the problem model so that more of the implicit behaviours and properties are made explicit. The enhancements that we investigate have no effect on the underlying transition system and are therefore not conventionally modelled, even in richer languages. As such, the ideas developed in this work are transferable to problems expressed in any language.

Domain model acquisition

Another line of research investigates inferring problem models by observing valid action sequences (McCluskey et al., 2009; Cresswell et al., 2009; Cresswell and Gregory, 2011; Mehta et al., 2011). An advantage of using this technology is that a model can be produced without requiring a modelling expert. However, it is likely that there are many different ways of expressing a model that explains the action sequences. Approaches at guiding the model selection have included: user interaction (Mccluskey et al., 2002), providing possible object states (McCluskey et al., 2009), or alternatively a bias can be included in the model learner towards models that explain expected action sequences (Cresswell and Gregory, 2011). However, the systems do not come up with appropriate labels on their own and as more of the task is automated the inferred models become less human-readable. An interesting aspect reported in Cresswell and Gregory (2011), is that control knowledge is implicitly embedded in the domain models induced by their system.

2.1.2 Remodelling

Researchers have been motivated to investigate remodelling for several reasons. These include decomposing the problem into sub-problems and rewording the problem leading to an increase in planner performance.

In Fox and Long (2001) it is observed that planning problems often include hard sub-problems. For example, there are several problems that include variations of the Travelling Salesman Problem; a known NP-hard problem (Garey and Johnson, 1979), meaning it is intractable in general. These sub-problems are expressed as part of the model and most approaches adopt a general search approach. It has been observed that these problems have mature technology developed specifically for solving them (Fox and Long, 2001). This has prompted research into decomposing the planning problem so that appropriate technologies can be applied to hard sub-problems (Fox and Long, 2001; Dornhege et al., 2009).

There are some approaches to planning that reformulate the problem into other representations so the problem can be solved using an alternative approach. For example, the planning problem has been reformulated as a Boolean satisfiability problem (Kautz and Selman, 1996, 1998), and a Model Checking problem (Edelkamp and Helmert, 2001). Although the details of these reformulations will often be tailored to target the solver, the planning problem is equivalent.

Our work is related to approaches that have remodelled the problem model to support a particular planning approach. CONSTANCE (Gregory et al., 2010), is a planner that translates the planning problem into a constraint satisfaction problem (CSP). As part of this translation, the model is abstracted, so that the CSP is not equivalent to the planning problem. The change leads to a problem that can be solved more efficiently by the constraints solver and only a trivial rewording of the solution is required to translate the CSP answer back into a plan. Moreover, various properties are retained, such as plan-length optimality between models. This is similar to the form of remodelling that we examine in this work; however, in CONSTANCE the remodelling is to reduce the redundancy in the search space, whereas in this work it is to support the planner to make better choices.

There are other examples too: during search, areas of the search space may appear equivalent (Section 2.2). This can happen when the heuristic function does not sufficiently discriminate between different states. One solution to this is to reword the problem to provide more information about the states (e.g. Bacchus and Kabanza, 2000; Khardon, 1999a); another is to add bridges that jump to more interesting areas of the search space (e.g. Coles and Smith, 2007). This is the topic of Chapter 3 and the relevant related work is discussed there.

2.2 Search and search control

Searching is the process of exploring a search space in order to find a particular node. The search space is a graph of related nodes. Each node links to a subset of the nodes in the graph. Search starts at one node and progresses out to the node's neighbours. There are various strategies that can be used to explore the search space, such as breadth first, depth first or best first search.

We can model the planning problem as a search problem. Plan-space planning problems can be solved using search directly. Planning starts from a partial-solution and there are a set of operations that generate its neighbourhood. The goal is to find a node where the partial solution is a solution to the problem.

In state-space planning the search space is the planning environment. Each node represents a state. However, the solution is an action sequence that transitions from the initial state to a goal state. This means that the problem is not as simple as finding a solution; we must also remember how we found the solution.

The size of the state-spaces of planning problems often prevents complete exploration. Instead methods are applied to guide search towards promising areas of the state space. Several approaches have been used in planning to achieve search control. We begin this section by presenting some of the standard search techniques. We then augment these approaches so that they are applicable to state-space planning problems. Finally, we introduce some of the methods that are used in planning to provide search control.

2.2.1 The search problem

In this subsection we briefly introduce the general search problem. For a more complete survey of combinatorial search, refer to Aigner (1988) and for an instructive start to algorithms for solving search problems refer to Skiena (2008).

We can encode the search space as a graph and model the search problem as a graph traversal problem. This language is rich enough to capture the search problems that will be encountered in this work. We begin by introducing some useful graph notation. This can be safely skipped over by anyone familiar with the standard formalism.

Graph Theory

Graphs are mathematical structures that represent relationships between objects. Many problems can be represented as graphs and as such there has been much interest in the field of Graph Theory. Efficient algorithms have been developed for solving many of the problems relating to graphs (for example, finding sets of objects that are "connected" through relationships, and selecting short sequences of these relationships that pass particular objects). In this subsection we introduce some of the graph related concepts that are used in this thesis.

Graph representation A graph, G, can be represented by a pair, $G = (V, E)$, where V is a set of vertices and E is a set of edges. Edges are pairs of vertices, (v, w), such that $v \in V$ and $w \in V$. The neighbourhood, Γ, of a vertex, v, is the set of vertices that are connected to v by a single edge: formally, $\Gamma(v) = \{w | (v, w) \in E\}$.

The number of vertices in the graph is called the order of the graph, $n = |V|$, and the size of the graph is defined as the number of edges, $m = |E|$. The density of the graph is the ratio of the number of edges in the graph to the total number of possible edges if every pair of vertices was connected, $\delta(G) = \frac{m}{n \times (n-1)}$. If $\delta(G) = 1$ then all of the possible edges of G are in E, and the graph is called *complete*.

Subgraphs The set of vertices, S (where $S \subseteq V$), defines a subgraph, $G^S = (S, E_S)$, of $G = (V, E)$, such that $E_S \subseteq E$ and $(u, w) \in E_S \implies u \in S \,.\, v \in S$. If $\delta(G^S) = 1$ then G^S is called a *clique* of G.

Paths A path between two vertices u and v is described by the series of edges $(v_0, v_1), \ldots, (v_{k-1}, v_k) \in E$, such that $v_0 = u \,.\, v_k = v$. We use the shorthand $(u, \ldots, v) \in E$ if there is a path between u and v. The *distance* between u and v is the number of edges along the shortest path between u and v. If the edges of the graph have uniform cost then this corresponds to the path with the fewest edges. If there are paths between u and v for all u and v in V then the graph is *connected*.

Graph traversal

The search space can be encoded using vertices to represent nodes and edges to represent the connections between the nodes in the space. A graph *traversal* problem involves *traversing* the graph in order to explore every node. This must be approached systematically so that all connected nodes are discovered. Thus the search will eventually arrive at a satisfying node, or prove that none are reachable from the starting node.

There are two basic approaches to solving this problem: breadth first and depth first search. A single node is selected as the root of the search. When searching breadth first, the search tree expands in layers. The first layer contains just the root vertex. The second layer contains all of the neighbours of the root vertex. In general, the next layer contains all of the vertices that are neighbours of a vertex in the current layer and that have not been discovered in previous layers. Alternatively, depth first search selects one of the neighbours of the root and follows it repeating the process at the next vertex. Once the search finds a vertex with no neighbours, (or all neighbours have been visited already,) then the search backtracks a level. The next edge is traversed and the procedure repeated. Once all of the edges of a vertex have been explored, search backtracks to the previous level. These searches are guaranteed to discover all of the vertices that are connected to the root by a path.

Search control

In some problems it is possible to provide some guidance; directing search to likely areas in the search space. For example, in a particular problem it might be possible to distinguish between the neighbours of the current node and identify those that are more likely to lead to more promising vertices. There are alternative search strategies that can use this form of guidance to organise the way that the graph is explored.

The best first algorithm relies on a heuristic function, $f(x)$, that provides an estimate of the value of a vertex, x. At each step the algorithm selects the vertex, x, with the best value, $f(x)$, and expands its neighbours. The intuition of the algorithm is that we should investigate better scoring nodes first as these should lead towards the target in fewer steps. As nodes with high scores are explored search can return to nodes that were ignored initially. In this way the completeness of best first algorithm is retained.

The best first search can be implemented using two lists: the open and closed lists. The open list is initialised with the root node. A loop begins and the first action is to remove the (heuristically) best node from the open list. If the node has not already been expanded, then it is added to the closed list and is expanded. Its neighbours are discovered and if one of these is the goal node then search stops. Otherwise all neighbours are added to the open list. This is repeated until the list is empty. As we always remove the best node in the open list, the nodes are processed in an order dictated by the heuristic.

Listing 2.3: Pseudo-code for the best first search.

```
def search(startNode) :
  openNodes = new List(); openNodes.add(startNode)
  while (not openList.isEmpty()) :
    node = openList.removeBest(f)
    addToClosedList(node)
    if (alreadyProcessed(node)) :
      for nextNode in expandNode(node) :
        if (goalNode(node)) :
          return node
        openList.add(nextNode)
```

The algorithm is presented in pseudo-code in Listing 2.3. It relies on implementations of several node type specific methods. The `expandNode` method returns all of the neighbours of the node; the method `addToClosedList` updates the closed list to reflect the node that is being processed. The method `goalNode` returns a Boolean indicating if the node is the goal, and finally the `alreadyProcessed` method returns a Boolean indicating whether the node should be processed. This last method will usually be determined by whether the node has been processed already.

Search in planning

State-space planning is a search for a sequence of actions that when applied in order affects the state of the world in such a way that the goal is satisfied. This corresponds to a series of transitions from the initial state to a goal satisfying state in the planning environment. Therefore the search problem is not only to discover a goal state, but also to find a path from the initial state. The best first search algorithm can be extended to solve the planning problem. This can be achieved by including more information in the search nodes. Concretely, a node is a triple: n = (*previousNode*, a, s), such that *previousNode* is the node that generated n, s is the current state, and a is the action that progressed *previousNode*'s state to s. Given a goal node, the plan can be read off backwards by following the previous nodes and collecting the associated actions.

The problem now involves a path as well as a particular node. This impacts on the heuristic functions that are relevant for searching. Instead of simply considering the quality of the node itself, we consider the quality in the context of the number of steps that have been made to find the node. Thus the heuristic is altered to reduce the length of the plan. The A* (Hart et al., 1968) algorithm computes the heuristic, $h(n)$ by adding the number of actions used to get to n, to the estimated number of actions to the goal. If this never over-estimates and $h(n)$ is used in a best first search, then this algorithm is guaranteed to compute the optimal solution while expanding the fewest nodes of the search space (Hart et al., 1968).

Another algorithm, which underlies several of the approaches used in planning, is greedy search. At each node greedy search follows the neighbour with the best heuristic score. The search makes a single probe, always following the best path with respect to h. If it exhausts this branch without finding the goal then it stops and fails.

The greedy approach relies heavily on the selection of an accurate heuristic function. If such a function exists then a solution can be achieved very quickly. If the function is not effective then the outcome will largely depend on the structure of the problem: whether there are many goal states and how many dead ends exist. There are many variations of greedy search that have been designed to combat areas of weakness in the heuristic guidance.

2.2.2 Control strategies in planning

In Chapter 1, we introduced the problem of satisfying a Christmas list. The solution to this problem is a list of steps that would lead to the collection of the items on the shopping list. Simply discovering a state where this is true is not very useful. A key consideration when searching through the possible shop combinations is whether the discovered plan will take a long time to execute.

In a similar way, we are interested in searching for plans with low cost. In the particular model we use in this work cost corresponds to the number of steps in the plan. There is an area of research in Automated Planning that is focussed on finding optimal plans for problems (plans with minimum cost)[1]. Optimal planners are limited in the size of instances that they can solve. In this work we are concerned with good but not necessarily optimal solutions. This allows us to tackle much larger problems.

In this subsection we provide some background of approaches to search control. We introduce abstraction based planning, which is seldom now used, but is particularly relevant to the ideas that support this work. For a more complete discussion of planning approaches refer to Ghallab et al. (2004).

[1]There has been an optimal track in the IPC since 2004 (IPC, 2004)

Domain independent search

Our focus will be on the development in the period since the first IPC. However, it is important to point a key work from before this period, called GRAPHPLAN (Blum and Furst, 1997). GRAPHPLAN introduced a graph based representation for solving planning problems that enabled subsequent important developments in the field. At the time of the first IPC, GRAPHPLAN was the state of the art. As such, several entrants of the first IPC were derived from GRAPHPLAN. STAN (Long and Fox, 1999) used an efficient implementation of the GRAPHPLAN algorithm, along with invariants from the domain analysis tool TIM (Fox and Long, 1998). This work led to HybridSTAN (Fox and Long, 2001), which used an extended domain analysis to decompose the problem model, allowing special treatment of sub-problems. There were two other approaches: the SAT planner, Blackbox (Kautz and Selman, 1998) and HSP (Bonet and Geffner, 1998), a heuristic based planner. HSP introduced the idea of relaxing the problem by removing the delete effects from actions and using the resulting *relaxed* model to compute heuristic estimates. Heuristics based on this model are still in the state of the art now.

The focus of the early IPCs was on being able to solve problems and reducing the time it took for this to be achieved. This encouraged the use of greedy search approaches. The planner FF (Hoffmann and Nebel, 2001) entered the second IPC (Bacchus, 2001)[1]. It used an augmented greedy search that resorted to full search over areas where the heuristic was uninformative. The heuristic itself was an improved version of the HSP heuristic. FF also incorporated an approach at goal ordering (Koehler, 1998) and a filtering technique. The filter involved selecting a subset of the neighbourhood of a node called the *helpful actions*. This is defined below.

Helpful actions The relaxed problem is the problem remodelled so that the actions do not have delete effects. The representation of a plan used in FF defines partially ordered plans. A plan is a list of action sets: $A_0, \ldots, A_n$. The interpretation is that the sets of actions are executed in order, but that the actions in a set can be executed in any order. During heuristic computation FF generates a plan for the relaxed problem. This is called the *relaxed plan*. One derivative of this plan is the helpful action set.

Definition 2.2.1 *The helpful actions are any actions that are applicable in the current state and achieve a precondition of any action in the second set, A_1.*[2]

[1] http://www.cs.toronto.edu/aips2000/

[2] This set includes actions with no effect; intuitively it contains any action that achieves a proposition

The helpful actions set is supposed to capture the actions that are likely to lead to better states. At each node in search FF first computes the helpful action set and evaluates each of these neighbours first. If none of these nodes improve the heuristic value then it might consider the complete neighbourhood of the node.

Further developments There have also been several new heuristic functions. Landmarks (Porteous et al., 2001), are actions that must be in a solution to the problem. They have been exploited in the planner LAMA (Richter et al., 2008). The heuristic is simply a count of the number of landmarks that have not been passed. LAMA uses this landmark heuristic and the FF heuristic, using a round robin strategy. It is part of the current state of the art.

Abstraction based planning

Abstraction based planners, such as ABSTRIPS (Sacerdoti, 1974) and AbNLP (Fox and Long, 1995), define a hierarchy of planning models of increasing abstraction. A plan is constructed for the problem at the most abstract level and this is then used to constrain the planning process in lower levels. The bottom level corresponds to the original model. The intuition behind this form of planning is that there might be some important decisions, which control the shape of the solution and should be made first. Once the main strategy is in place the planner can shift its focus to decisions that will have a more local impact. In ABSTRIPS the abstraction is formed by removing preconditions from the operators. This means that actions in the upper levels can be applied without the correct propositions being established in the state. As the planner focusses on lower levels, more of the preconditions of operators are represented and require to be supported in order to define a valid plan. In AbNLP the operators in higher levels define a set of partially ordered sub-goals; which are unpacked when the planner switches focus to the next lower level of abstraction. Decomposing into sub-goals, rather than operator sets allows more freedom in the selection of actions that are appropriate to the given context. Similar ideas have been examined more recently in Gregory et al. (2011). The hierarchy of models in that work was formed by abstracting from individual propositions to groups of propositions.

that has to be achieved by the actions in A_0.

Hand-tailored planners

The early IPCs also had a special track for hand-tailored planners. This attracted approaches that involved some level of customisation for a particular problem domain. The type of tailoring varied from hand-written planners for TLPLAN (Bacchus and Kabanza, 2000) and TALplanner (Doherty and Kvarnström, 2001), to operator decompositions for SHOP (Nau et al., 2003). These planners demonstrated that using control knowledge is extremely effective.

This success inspired a body of work that investigated using control knowledge in search. The key limitation with the approaches was that the knowledge was hand-written. In fact, this track was discontinued after IPC-3 because it was unclear what was being measured: the approach, or the hand-written control knowledge. The main focus of later research was in control knowledge representations and learning approaches, although exploiting the control knowledge was obviously important. The introduction of a learning track in the sixth IPC (Fern et al., 2011) indicates the level of interest that built around learning control knowledge. Approaches to using control knowledge in search are discussed in Section 2.3.

2.3 Control knowledge

Control knowledge is compiled information that provides some form of guidance. There are various applications of control knowledge in planning including making action selections, improving a heuristic estimate and as a method of extending the problem model.

In this section we introduce several of the forms of control knowledge that have been used with planning systems. Macro actions are one way of representing control knowledge, and another is by using *production rules*. A common aspect of control knowledge is that there is a condition on the application of the knowledge (similar to an action precondition). Ensuring that these conditions can be stated appropriately is a key consideration in this work. We will examine this in detail in Chapter 4. In this section we look at the opportunities for exploiting control knowledge in search. We introduce common representations and how these are combined to form knowledge systems. We also discuss control knowledge acquisition.

2.3.1 Opportunities for exploiting control knowledge in planning

PRODIGY (Carbonell et al., 1991) and SOAR (Rosenbloom et al., 1985) are planners that were used as test beds for investigating the use of control knowledge in search. The intention in these systems was to "open search up" so that search decisions could be observed and controlled. For example, control knowledge could influence whether a regressive or forward-chaining step was made, or impact on the next sub-goal to achieve, or the next action binding to make. Each decision point is an opportunity for exploitation of control knowledge.

The decisions that have been made can provide guidance for future decisions; however, deciding when those decisions are relevant can be challenging. For example, knowledge may only be appropriate in the context that a particular goal is selected.

There were many investigations of exploiting control knowledge carried out using these (and similar) frameworks. For example, learning control knowledge using explanation-based learning (Carbonell et al., 1991), learning action sequences for achieving sub-goals (Laird et al., 1986), and automatically generating abstraction layers (Knoblock, 1990). Although now PRODIGY and SOAR are not competitive with the state of the art, several of the developed ideas have been exploited successfully within modern technology (de la Rosa et al., 2011; Jiménez et al., 2012).

2.3.2 Representing control knowledge for search

In this subsection we survey some of the main knowledge representations used in search. We begin by introducing several general knowledge representations. We then examine how rules systems have been formed and how this has affected the development in the field.

Production rule systems

A production rule is a conditional proposition of the form:

$$\boxed{\text{if } \langle antecedent \rangle \text{ then } \langle consequent \rangle.}$$

The intended meaning is that the consequent is asserted if the antecedent is satisfied. How the antecedent is evaluated and the form of the consequent depends on the type of production rule. Each production rule can be used to capture a single piece of advice or a single fact, and the antecedent dictates when that fact is true or when the advice is applicable. For example, a rule that could be used to capture a guidance heuristic for navigating a maze might be:

$\boxed{\text{if } \langle \textit{ at a junction } \rangle \text{ then } \langle \textit{ choose the leftmost branch } \rangle}$

There are several ways that rules can be used in search, including pruning and selection rules.

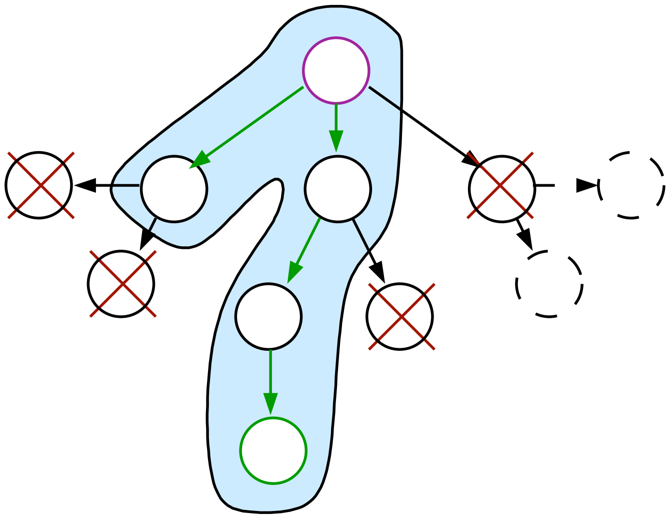


Figure 2.2: The rules remove actions from the neighbouring edges of a node. This reduces the branching factor.

Pruning rules For a particular node and its neighbourhood, a pruning control rule removes nodes from the neighbourhood so that they do not need to be searched. In Figure 2.2 the nodes with the crosses have been pruned from the search. As a result the dotted nodes will not be examined either. Pruning rules have been used in various planners, including PRODIGY and TLPLAN.

These rules have to be constructed carefully as not only will the planner not examine the pruned node, but it might miss the opportunity of examining the nodes that connect to it. If the control knowledge is poor then good nodes can be pruned from the search space and in extreme cases all paths to a goal state might be pruned.

Selection rules An alternative rule interpretation is called a selection rule, illustrated in Figure 2.3. These rules select a single successor and have the effect of pruning all alternatives to the single node that is selected. If the knowledge is of good quality then a depth-first search with no backtracking can be used to lead straight to a goal (Bacchus and Kabanza, 2000; Khardon, 1999a; Martin and Geffner, 2000; Fern et al., 2006).

There are benefits to each interpretation. These rules are appropriate if the decision of what to do next can be made. In some situations this might be difficult and it might

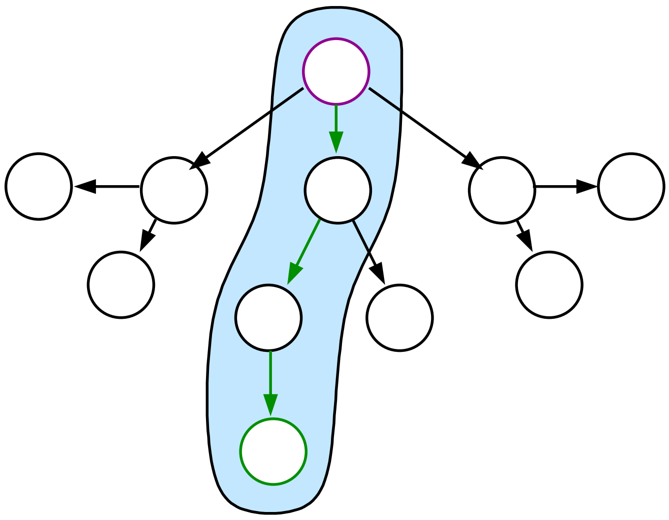


Figure 2.3: The rules select the next action from the neighbouring edges. No search is required.

instead be obvious what not to do. One difference is that pruning rules will remove branches from the search space, whereas selection rules will not.

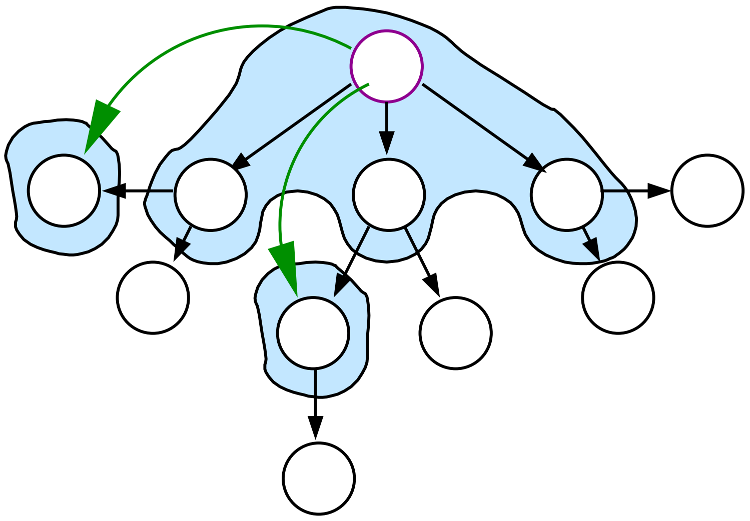


Figure 2.4: Macro actions connect nodes several steps away from the current node.

Macro actions Another form of control knowledge that has been provided to planners to assist search are macro actions. Macro actions are added transitions to the state transition system that correspond to a concatenation of several of the original state space actions. The draw-backs of adding macro actions is that they increase the branching factor and can greatly increase the total number of actions in the state system (Fikes et al., 1972). Planning problems often have very large state systems already and so many approaches, such as pruning rules in TLPLAN, attempt to reduce the choice. However, it has been demonstrated that carefully selected macro actions can provide benefits to search (e.g. Newton et al., 2007; Coles and Smith, 2007).

Control knowledge systems

There are several representations of control knowledge utilised in PRODIGY. The original system supports selection and pruning rules. It also allows ordering rules that allow unsatisfied goals to be ordered. These are selection rules that act over goal choices. The system has been extended in other work to support macro action and other forms of abstraction, as described in Section 2.1. The complete collection of rules and macros forms a control knowledge system. The aim in PRODIGY was that the system could exploit knowledge where available; but it was not expected to be complete.

The use of control knowledge on its own to control search, instead of as a mechanism for improving a planner, was made popular by the performance of TLPLAN (Bacchus and Kabanza, 2000), TALplanner (Kvarnström and Doherty, 2001) and SHOP (Nau et al., 2003) in the early IPCs. One of the key aspects of these approaches was the use of "rich languages" to express the control knowledge. The consequence is that the rules used with these planners can concisely establish the important differences between states and select the correct behaviour. An alternative approach is presented in Khardon (1999a) and implemented in L2ACT, which uses a collection of selection rules in place of a planning strategy. The rule language used to capture the control knowledge in L2ACT is limited.

Learning The reason for the limited language was that the control knowledge used in L2ACT was learned automatically. Subsequent work investigated more expressive languages (Martin and Geffner, 2000; Fern et al., 2006). The main aim was to discover a language that could express useful control knowledge; but limited enough that the learning problem was feasible. The actual languages are discussed in detail in Chapter 4 and approaches to learning control knowledge are used and developed in Chapter 8.

It has been observed that it can be difficult to capture a perfect rule system. In particular, there are often a few weak rules that lead to poor performance. This has led to research that looks at reducing the impact of these weak rules (Yoon et al., 2006;

de la Rosa et al., 2008). This has been achieved in Yoon et al. (2006) by using the rules to correct a weak domain independent heuristic. In de la Rosa et al. (2008) a different form of rule is used that estimates the likelihood of an action being appropriate. These estimates are used to guide a depth first search.

In this work, we take a step back. We observe that an important aspect of capturing rules is the vocabulary that their conditions draw from. We argue that the weaknesses in learned rule systems can often be attributed to the limited vocabulary that the rules are drawn from. The vocabulary can prevent an effective strategy from being constructed, even by hand.

2.4 Learning

Machine Learning is a vast topic that has become increasingly important in many fields. In this study we focus on certain approaches to optimisation and search to motivate the learning discussion later in this work. An optimisation problem over a set (potentially infinite) of candidates, $\mathbf{C}$, is defined in terms of a fitness function. A fitness function maps from candidates to the reals, $\delta : \Pi \mapsto \mathbb{R}$, such that a high value of δ indicates a high fitness. The optimisation problem is the problem of finding a candidate, c, that maximises the value, $\delta(c)$:

$$\mathbf{targetCandidate} = \{c \in \mathbf{C} | \forall c' \ \delta(c) \geq \delta(c')\}.$$

Whether a solution to this problem will be an effective candidate depends on the selection of an appropriate fitness function. If the candidate space is small then it might be feasible to enumerate each possibility and evaluate them in turn, retaining the candidate with the highest fitness. However, this is not always possible; in this section we look at some alternative approaches to this problem.

2.4.1 Local search

Local search is an approach for solving optimisation problems. The approach involves defining one or more neighbourhoods, each defining a set of nodes that are the neighbours for a particular node. Search progresses by selecting a node from one of these sets using one of several methods. For example, selecting the neighbour with the highest fitness is called the hill-climbing approach (comparable to greedy search). Search terminates after a predetermined time, or once no improving step can be made. The success of hill-climbing depends on the selection of the initial candidate and the interaction between the fitness landscape and the neighbourhoods. There are more involved approaches aimed at escaping local minima, including restarting from random nodes, and allowing periods of decreasing fitness.

2.4.2 Genetic algorithms

Genetic algorithms are stochastic search algorithms inspired by the natural reproductive cycle. The approach relies on a population of candidate solutions that are used to seed the construction of the next generation of candidates. One of the key principles at work is the survival of the fittest: better candidates are more likely to be selected to create the next population. The expectation is that the overall solution quality will improve. The measure of a candidate's quality is provided by the fitness function.

An example problem has solution strings of length 4, for example, the strings 1001 and 0011. An example fitness function, $f(x)$, might give a candidate, x, a score by summing the number of ones in the string. The maximum reward in this problem would be $f(1111) = 4$ and the minimum score of $f(0000)$.

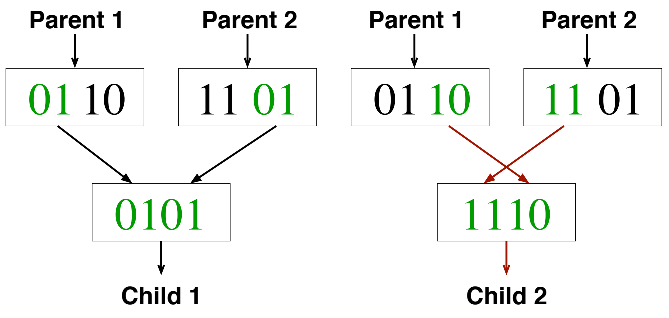
There are three operators that are used in constructing the new population from the old population: selection, crossover and mutation.

- Selection is used to choose individual candidates from the current population for reproduction. The selection strategy will normally bias towards selecting stronger candidates from the current population.
- Crossover is an operation on two candidate solutions that produces two children. The crossover point is selected at random. The first child is constructed by taking the pattern of the first parent up to the crossover point and adding the pattern of the second parent from the crossover point to the end. The second child is constructed with the first part of the second parent and the last part of the first parent. An example of the crossover operator is illustrated in Figure 2.5(a).
- Mutation is an operator that randomly changes single units of a solution. This operator allows search to escape from local optima. An example mutation is illustrated in Figure 2.5(b).

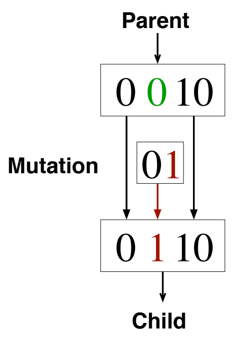
Crossover and Mutation operators are required to form valid solutions.

A simple genetic algorithm can be described by the following steps:

1. Construct the initial population by generating n random solutions.



(a) An example of crossover



(b) An example of mutation

Figure 2.5: Examples of genetic algorithm operators

2. Evaluate the initial population using the fitness function.
3. Continue the following steps until the new population is full.
    - Select a pair of candidates.
    - Apply the crossover operator to the candidates with probability, p_c, (the crossover probability). If crossover is applied then choose a random point in the parents and construct the two offspring. If crossover is not applied then the offspring are exact copies of the parents.
    - Mutate the offspring with probability, p_m (the mutation probability).
    - Evaluate the offspring and place them in the new population.
4. Set the new population as the current population.
5. Jump back to step 3 and repeat until the stopping criteria is met. The stopping criteria may be a threshold fitness, a convergence measure, or a maximum number of generations.

Although the basic algorithm is simple, genetic algorithms have received a lot of attention and have been used successfully in many applications. Crossover allows successful parts of solutions to be passed on into future populations and mutation introduces new parts into the candidates by adding noise. The combination of crossover and mutation help to keep the search out of local minima.

In this work we use an extension of the genetic algorithm called a genetic program; each candidate in a population of a genetic program is a program (a control system in this work), instead of a parameterisation. We rely on more sophisticated mutation and crossover operators that act directly on the control system. These will be defined in Chapter 8.

2.5 Statement of thesis

Control knowledge in planning is often captured using the predicates and actions of the problem models that the knowledge applies to. The thesis that this work defends is that it is possible to learn more effective control knowledge by selecting a richer problem model encoding; that is, an encoding that makes explicit more of the relationships and behaviours of the problem. We will show that this can provide the necessary support for control knowledge expression. We demonstrate that there are certain forms of problem structures where the appropriate model can be selected automatically.

In order to defend this thesis we will take the following steps:

- Develop a framework that allows us to explore alternative problem models.
- Analyse the interaction between the selected control knowledge representation and the problem model, which is the source of the limitations of control knowledge expression. Categorise the aspects of problem models that lead to the limitations being lifted.
- Assess whether the problem models, which lift these limitations, lead to the expression of effective control knowledge, with an aim of demonstrating the feasibility of the approach.
- Investigate the automation of model selection. In particular, we seek a novel method of selecting the model, which will provide the necessary support for exploiting control knowledge in planning.
- Learn and evaluate control knowledge that exploits the selected model and effectively controls search.

CHAPTER 3

A FRAMEWORK FOR EXPLORING PROBLEM MODELS

The process of stating a planning problem typically begins with a real world situation that is abstracted and expressed in a standard language, such as STRIPS. The modeller rejects many behaviours, interactions and concepts that are not required to express the state-transition-system. The convention often followed is to capture what the planner can do, but model no concepts that would give guidance as to what the planner should do. This separates the representation of the model from control knowledge that is relevant to that model (McDermott, 2000).

It is not surprising that planners appear to be sensitive to the model that is presented to them (Hoffmann, 2005; Coles and Smith, 2007; Aler et al., 2000a; Khardon, 1999a; Martin and Geffner, 2000). And as a consequence, many researchers have analysed the benchmark planning models to identify where more information and more choice can be provided to the planner. In heuristic search planning this has predominantly been considered through joining actions together to allow a sequence of actions to be made in a single choice (Iba, 1989; Coles and Smith, 2007; Botea et al., 2005a; Newton et al., 2007). Alternatively, in rule based planning, the focus has been on enriching the states with additional predicates to provide richer concepts for the rule conditions (Khardon, 1999a; Martin and Geffner, 2000).

In this chapter, we present a framework that supports our investigations of the balance between the concepts expressed in a problem model and the concepts that must be constructed over these concepts in order to support control knowledge to make action selection choices effectively. The motivation behind our approach is *not* to solve the planning problem in part or in whole, but, instead, to *support the planner* as it solves the planning problem. This framework will be demonstrated with a concrete implementation in Chapter 5 and a more detailed presentation can be found in Appendix I.

3.1 A chain of language restrictions

An important aspect of a model is the language, Σ, that is accepted, which determines the propositions and actions that can be exploited by the planner. We assume that for a particular planning problem, there exists a maximally rich model, $\mathbb{M}$, which would support any planner to plan effectively. Whereas $\mathbb{M}$ models many actions and propositions, we consider more limited languages, which provide a restricted view of $\mathbb{M}$.

Definition 3.1.1 *The restricted view of* $\mathbb{M}$ *for a language,* Σ*, is denoted* $\mathbb{M}|_{\Sigma}$ *and is a view of* $\mathbb{M}$ *where the part of* $\mathbb{M}$ *that is expressible in the language,* Σ*, is modelled.*

The framework that we define in this chapter establishes a series of enhancement steps from the described model (the model presented to the planner) towards $\mathbb{M}$, a notional model that is ideal for planning. In this work, we focus on enhancements that are appropriate for forward chaining planners. From this perspective, there are two sources of motivation behind an enhancement: to enrich the information provided to the planner when making action choices; and to vary the level of the decisions that are made: both by raising the level of reasoning through abstraction and lowering the level of reasoning through enrichment.

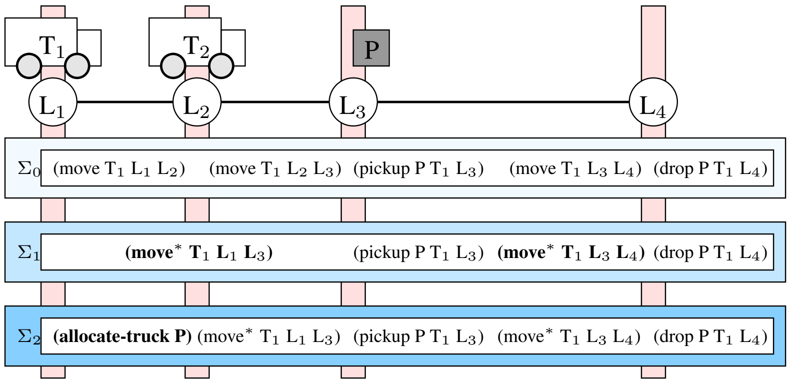

T_1
T_2
P
L_1
L_2
L_3
L_4
Σ_0 (move T_1 L_1 L_2) (move T_1 L_2 L_3) (pickup P T_1 L_3) (move T_1 L_3 L_4) (drop P T_1 L_4)
Σ_1 **(move* T_1 L_1 L_3)** (pickup P T_1 L_3) **(move* T_1 L_3 L_4)** (drop P T_1 L_4)
Σ_2 **(allocate-truck P)** (move* T_1 L_1 L_3) (pickup P T_1 L_3) (move* T_1 L_3 L_4) (drop P T_1 L_4)

Figure 3.1: An example set of language enhancing steps. In the original domain model (Σ_0), trucks move around a road map. In an example enhancement layer (Σ_1), the movement map can be abstracted so that the truck can be moved to any location in a single decision. A further enhancement (Σ_2), provides an explicit representation of allocations of trucks to package deliveries.

In transportation problems, trucks move packages between their initial location and alternative goal locations. Three plans are presented in Figure 3.1 that illustrate a possible sequence of enhancements towards an ideal model for planning in this problem. In the described language, Σ_0, the truck must move several steps to pickup a package. However, from the point of view of a planner it might be more appropriate to choose the destination of the truck, or equivalently, select a single action that moves to that location. The enhanced language, Σ_1, in Figure 3.1, presents an alternative representation of the plan that uses an abstracted move action (`move`*). Of course, in order for the planner to determine the best truck to move towards a package it may make decisions over appropriate assignments of packages to trucks. This could be realised through a set of allocation propositions and actions that make the allocations, as illustrated by the plan in the language, Σ_2 in Figure 3.1. This is an example of making decisions at a lower level.

The example illustrated in Figure 3.1, is an example chain of three steps, from the described model language, $\mathbb{M}|_{\Sigma_0}$, to a richer model, $\mathbb{M}|_{\Sigma_2}$, which captures more of the concepts from $\mathbb{M}$. This chain is generalised by the definition of a *chain of language restrictions*.

Definition 3.1.2 *A chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$, is a collection of languages that are ordered in terms of expressivity, with the most limited language (corresponding to the described model, Σ_0) as the first element of the chain. In particular, this is a chain of languages that exist between the language of the described model and that of $\mathbb{M}$.*

An important factor is that we are ultimately interested in generating a solution to the original described planning problem. It is quite possible to imagine enhancing problem models with many alternative approaches to solving the problem. For example, in transportation problems an alternative to delivering a package by truck is to take a taxi and drop it off; but if this option is not in the described model then it provides no assistance for creating a plan. Moreover it can lead to bridging states that are not connected in the described model. In this work we limit our focus to a subset of chains for which we can make certain guarantees over executability. In particular, we focus on chain steps that provide one of three forms of enhancement steps from Σ_i to Σ_{i+1}:

- Abstract actions (macro actions) that combine one or more actions modelled in Σ_i. For example, the composition of movement actions illustrate in Figure 3.1.

- Enriching (or derived) predicates (any computable Boolean function) that provide an additional interpretation on a state. For example, establishing clusters of locations in a location map.
- Decision predicates and associated decision making actions, which allow decisions to be made explicit in the state. For example, an allocation made between a truck and a package, illustrated in Figure 3.1.

3.2 Co-execution

In this section we present a method of search, called *co-execution*, which is appropriate for using forward chaining search in an enhanced model. The overall approach is illustrated in Figure 3.2. A framework for enhancing the problem model through a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$, to some language, Σ_i, was described in Section 3.1. The planner acts in $\mathbb{M}|_{\Sigma_i}$ allowing it to exploit the enhanced model. The planner can select actions that cannot be directly applied in the described model, $\mathbb{M}|_{\Sigma_0}$. The approach maintains the state of each model concurrently: each selected action is interpreted into an appropriate behaviour for $\mathbb{M}|_{\Sigma_0}$.

In Section 3.1, we presented the three sorts of language enhancements that are possible in our framework. Abstract actions and enriching predicates have been widely used and are each trivial to interpret for the described model. An abstract action, a, such that $s' = \gamma_{\Sigma_i}(s, a)$, can be substituted for any sequence, $a_0, \ldots, a_m$, such that $s' = \gamma_{\Sigma_0}(\ldots(\gamma(s, a_0)\ldots), a_m)$. In the case of enriching predicates, the actions are identical in each model; therefore a plan for some Σ_i is a plan in the described model.

The final case is the decision propositions and sets of connecting actions. The decision propositions are guaranteed to be distinct from previous propositions and therefore these actions have no effect on the state of more restricted models. This leads to actions in the wff(Σ_i) being interpreted as either a sequence of actions (1 or more), or a NO-OP (an action with empty effects and precondition) in wff(Σ_0).

The subset of language restrictions we consider preserve important properties on the states of the co-executed models. We present two theorems here; the proofs can be found in Appendix I.4. The first theorem states that each of the states of the enhanced models is represented by exactly one state in the described model. This means that it is sufficient to maintain a single state in each model. The second theorem states that transitions between states made by an enhanced action can be interpreted as a sequence of actions in the described model.

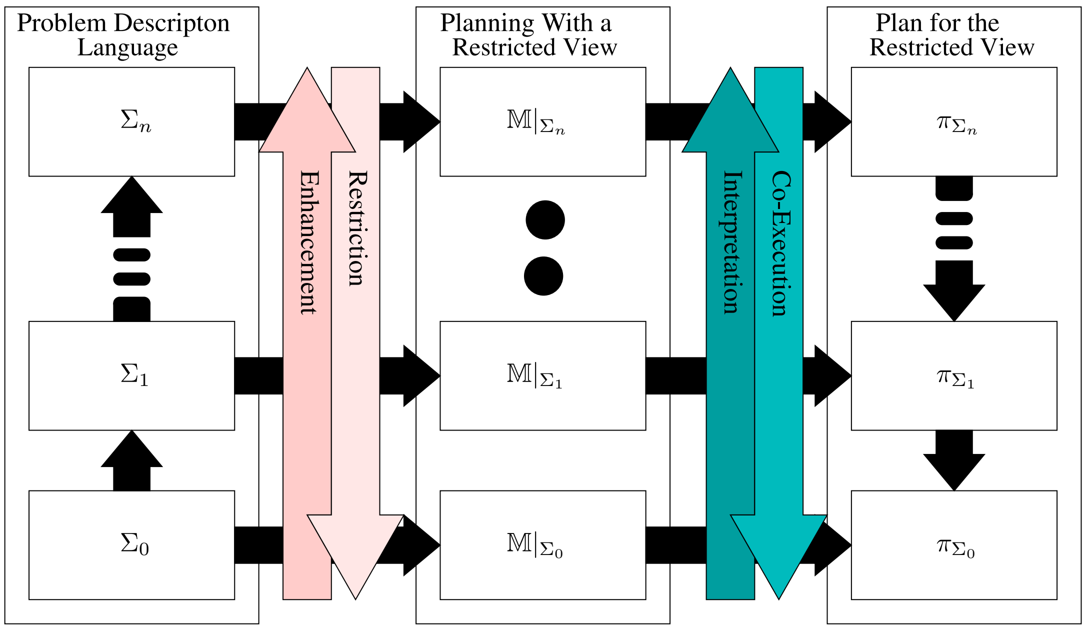


Figure 3.2: We will investigate stepping up from a limited language, Σ_0, to a richer language, Σ_i. This provides an enhanced planning environment. The plan produced is relative to higher level concepts not expressed in Σ_0. The plan in the richer language can be interpreted and used as a plan for the original problem. An effective method of interpreting the plan relies on the execution of both models simultaneously.

These theorems rely on definitions of relatedness between the states and actions of the described and enhanced models. A state in $\mathbb{M}|_{\Sigma_i}$ is represented by a state in $\mathbb{M}|_{\Sigma_0}$ if the states agree on each of the propositions in $\mathbb{M}|_{\Sigma_0}$.

Definition 3.2.1 $(s(\Sigma_i)\mathbf{R}s'(\Sigma_0)) \iff \forall p \in \mathit{wff}(\Sigma_0)\ p \in s' \iff p \in s$

An action, a, is represented by an action sequence, $a'_0, \ldots, a'_m$, if for any states, s_0 and s_1 that a transitions between, $a'_0, \ldots, a'_m$ transitions between the two states, s'_0 and s'_m, which represent s_0 and s_1.

Definition 3.2.2

$$\begin{aligned}
&a(\Sigma_i)\mathbf{R}a'_1(\Sigma_0), \ldots, a'_m(\Sigma_0) \iff \\
&\quad (\forall s_0, s_1 \in \mathit{wff}(\Sigma_i)\ s_1 = \gamma_{\Sigma_i}(s_0, a) \implies \\
&\qquad \forall s'_0 \in \mathit{wff}(\Sigma_0) \\
&\qquad\quad (s_0\mathbf{R}s'_0 \implies s_1\mathbf{R}\gamma_{\Sigma_0}(\ldots(\gamma_{\Sigma_0}(s'_0, a'_1)\ldots), a'_m)))
\end{aligned}$$

Theorem 3.2.1 *For any state, s, expressible in Σ_i, there is a single state, s', expressible in Σ_0, which represents s ($s\mathbf{R}s'$).*

Theorem 3.2.2 *For any action, $a \in$ wff(Σ_i), there exists an (possibly empty) action sequence, $a'_0, \ldots, a'_m \in$ wff(Σ_0), which represents a ($a\mathbf{R}a'_0, \ldots, a'_m$).*

Theorem 3.2.2 ensures that any enhanced action from our framework can be interpreted as an action sequence in the described model and therefore represents a valid sequence of actions for the original problem. As a consequence, the execution of a plan expressed in a language, Σ_i, will be executed as a plan in Σ_0. The architecture that implements co-execution is presented in Section 5.1 and the model presented here is further explored in Appendix I.

3.3 Discussion

In this chapter we have developed a framework for enhancing the problem model. The models explored within this framework share properties with the described problem model, including action-sequence transferral under co-execution. In order to secure this property we have made several restrictive assumptions over the chains of languages that can be explored in the framework. However, this framework consolidates several of the previous approaches to model enhancements, including: macro actions (Iba, 1989; Coles and Smith, 2007; Lindsay, 2012; Botea et al., 2005a; Newton et al., 2007; Gregory et al., 2010), which correspond to abstract actions; state enrichments (Khardon, 1999a; Martin and Geffner, 2000; de la Rosa and McIlraith, 2011); and enhancing the model (Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001), which are comparative to moving along a chain of language restrictions. Enhanced languages have been used to express parts of the problem model (Dornhege et al., 2009; Gregory et al., 2012), so that parts of the model are not represented in the PDDL description. A key difference in these works is that the solution to the problem is expressed for the enhanced model. This means that no interpretation is required. Another approach is to step along a chain in the opposite direction and restrict the view of the problem model (Hoffmann et al., 2004; Fox and Long, 2001). When the problem can be decomposed this can allow the planner to tackle individual sub-problems in isolation, which can lead to more efficient planning. These relationships are discussed further in Section I.5 of the Appendices; we discuss the relationships between control knowledge representations in Chapter 4, implementation of the model enhancements in Chapter 5 and approaches to learning control knowledge in Chapter 8.

CHAPTER 4

CONTROL KNOWLEDGE

Capturing domain specific planners using control knowledge supports effective planning for individual problems (Bacchus and Kabanza, 2000; Winner and Veloso, 2007; Fern et al., 2006). In particular, TLPLAN (Bacchus and Kabanza, 2000) provides an effective framework, in terms of plan quality and planning time. A key benefit is that each planner captures a solution that solves groups of problems. Each planner can solve the problem distribution of a particular planning domain. In Bacchus and Kabanza (2000); Doherty and Kvarnström (2001) each planner is captured by a rule system. However, the languages used to express the rules are rich and they are handwritten.

Ultimately, we are interested in learning the control knowledge; therefore we must limit ourselves to the scope of current learning technologies. Research into using limited rule languages to express the rule systems has been fruitful (Khardon, 1999a; Martin and Geffner, 2000; Fern et al., 2006; de la Rosa and McIlraith, 2011). These languages are not rich enough to express the concepts necessary to support reasoning in certain problems (Xu et al., 2007). However, we observe that the rules are expressed using the vocabulary of the problem model. As the enhancement chain defined in Chapter 3 controls this vocabulary, it also controls the words used to express the rules.

In this chapter we define a series of mappings, called *policies*, that capture plans for progressively larger sets of planning problems. The most general policy is a *generalised policy*, which we will learn in this work. We introduce the rule based representation that we adopt to capture generalised policies and we then compare this representation to other methods from the literature.

4.1 Policies

A policy is a complete mapping from states to actions that is intended to direct execution towards the goal. A simple executive can be used to solve a problem with an appropriate policy: it looks up the action for its current state and applies it, repeating this loop until it reaches a goal state. Planning is then the problem of policy construction.

Definition 4.1.1 *A policy π, is a total map, π: States $\rightarrow$ Actions. A policy is intended to achieve a single goal.*

An appropriate policy will lead the executive by a short path through the state space to a state satisfying the goal that the policy addresses. Application of a policy requires no search and no intelligence on the part of the executive.

Similarly, we can define a partial policy if the mapping is not complete.

Definition 4.1.2 *A partial policy π, is a partial map, π: States $\rightarrow$ Actions. A partial policy is intended to achieve a single goal.*

A plan for a classical planning problem, $\mathbf{P} = \langle \tau, s_{\text{init}}, g \rangle$, can be seen as a partial policy that determines actions for precisely the states on the trajectory from the initial state to the goal.

Definition 4.1.3

$$\begin{aligned}
&\pi \textit{ is a plan for problem, } \mathbf{P}, \iff \\
&\quad \exists a_1, \ldots, a_n, s_0, \ldots, s_n \\
&\qquad \forall i \in [0, \ldots, n-1]\ \pi(s_i) = a_{i+1}\ .\ s_{i+1} = \gamma(s_i, a_{i+1}) \\
&\qquad s_{\textit{init}} = s_0\ .\ s_n \models g
\end{aligned}$$

Under the assumption that action application is deterministic, a plan for a particular problem will always result in the same action sequence. Accordingly, if the initial state is known then we can simply describe a plan as the sequence of actions. Where it is important to distinguish, we will denote a plan, represented as a series of actions, as $< \pi >$.

A usual application for a policy is as a solution to a problem with uncertainty, such as those found in Robotics. In these problems the description of the state can be incomplete and the outcome of an action can be uncertain. In this setting it is not clear which states will be visited during execution and a solution must solve the problem from any states that might be visited.

In this work we reason with a deterministic model of the planning problem. This means a solution for a single problem can be captured in a single action sequence that transforms the initial state to a goal satisfying state. However, in this work we are interested in solving many problems using the same solution.

Martin and Geffner (2000) have defined a more general policy, called a *generalised policy*, that can be used to express a strategy for solving all the problems of a planning domain. In this setting, there is no uncertainty in action application or the state description, but there is uncertainty in the set of objects and in the initial state and goal.

The control knowledge that we learn in this work captures a variation of the *generalised policy*. In this section we describe the generalisation of the policy definition that we use in this work.

4.1.1 Generalised policy

Most of the approaches that learn policies require the entire state space to be enumerated. This becomes impractical for even moderately sized planning problems. Learning policies without enumerating the search space is possible (Boutilier et al., 1999). However, as planning problems are described and solved individually, with changes to the set of objects, the initial state and the goal, a new policy is usually required for each problem. In situations with uncertainty this might be worth the learning cost, however, our target is deterministic classical planning and the learning time cannot be justified.

In this work we do use a learning process that would be too expensive to consider using for learning policies for a single goal. However, we define a variant on a policy that can be used to describe the solutions for many goals. We use the observation that planning problems from the same domain are likely to contain the same sorts of tasks and we use a representation that supports exploiting these similarities in a compact way.

A *generalised* policy is not specific to a particular goal: the action that is selected by a generalised policy is not only dependent on the state, but also on the goal that is to be achieved. It is defined for states and actions that belong to the well formed formula of a language, Σ. In particular, Σ is a language on a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$.

Definition 4.1.4 *A generalised policy π_Σ, for sets of states $\mathbf{S} \in$ wff(Σ) and actions $\mathbf{A} \in$ wff(Σ), is a total map π_Σ: $\mathbf{S} \times$ Goals $\rightarrow \mathbf{A}$.*

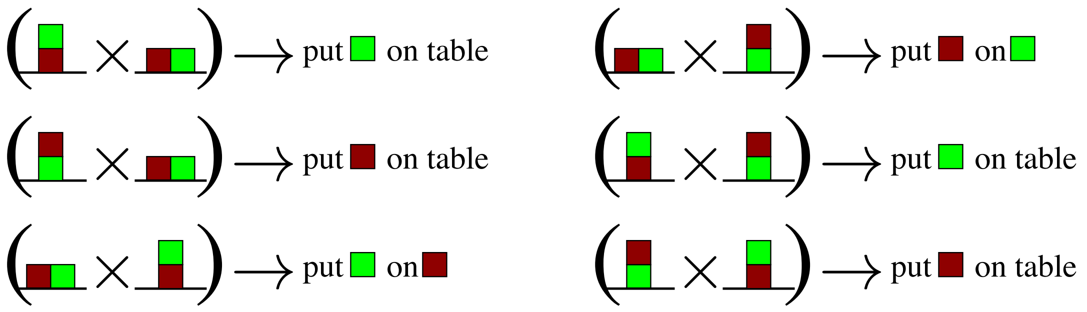


Figure 4.1: An example of a generalised policy for a 2 block Blocksworld problem. The figure illustrates (state,goal) pairs and the action mapped to by the policy.

An example policy is illustrated in Figure 4.1. The pairs of states illustrate the state and goal pairs and the associated action is the action mapped to by the policy. It is important for this work to notice that Σ could be a rich language that expresses an enhanced view of the states in the problem. This generalises definition 4.1.1 to any possible goal expressible in the language.

4.1.2 Domain conventions

In many problems, the structure of initial states and of goals is limited by implicit conventions attached to the domain. For example, in Blocksworld problems no blocks ever start off in two places at the same time, although there is nothing to prevent this in the syntax of the domain description. As a result of this constraint and because of the behaviours of the objects in the domain, there are no reachable states with objects in two places.

Logistics problems provide another example: there are always goals requiring packages to be delivered to specific locations, however, there are no goals for vehicles. This means that goals of packages being in trucks or planes and goals of either of the vehicles being at locations are never posed for this domain. These constraints mean that it is often possible to consider using a partial generalised policy.

Definition 4.1.5 *A partial generalised policy π_Σ, for sets of states* $\mathbf{S} \in$ *wff*(Σ) *and actions* $\mathbf{A} \in$ *wff*(Σ)*, is a partial map π_Σ:* $\mathbf{S} \times$ *Goals* $\rightarrow \mathbf{A}$.

It is not only domain conventions that focus us on particular subsets of the goals. For a particular problem, the goal is independent from the particular language used to model the problem. In particular, we will only solve goals that can be expressed in Σ_0.

This is appropriate because the enhancements are intended to support planning for the described language.

4.1.3 Instantiating the policy

A generalised policy is intended to solve arbitrary problem instances in a planning domain, exploiting explicit knowledge of the goals to direct the actions to achieve them. However, it is not only defined for different initial states and goals in the same state space. A specific problem instance will contain its own collection of particular objects and therefore its own sets of states, goals and actions. A generalised policy for a domain is therefore parameterised by object parameters and it is the appropriate instantiation of these parameters that represents the generalised policy applicable to a particular problem instance.

Definition 4.1.6 *A parameterised partial generalised policy* $\pi_\Sigma[\mathbf{O}]$*, for a particular set of objects,* $\mathbf{O}$*, sets of states* $\mathbf{S}[\mathbf{O}] \in$ *wff*(Σ) *and actions* $\mathbf{A}[\mathbf{O}] \in$ *wff*(Σ)*, is a partial map* $\pi_\Sigma[\mathbf{O}]$*:* $\mathbf{S}[\mathbf{O}] \times \mathbf{G}[\mathbf{O}] \to \mathbf{A}[\mathbf{O}]$.

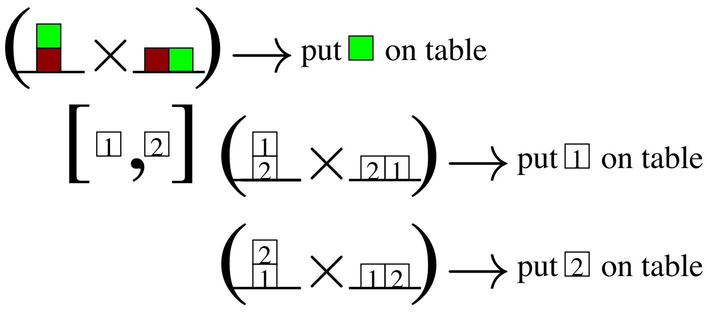


Figure 4.2: An example of instantiating a generalised policy for a particular 2 block Blocksworld problem. In this illustration the coloured blocks represent variables and the numbered blocks provide an example of particular problem constants.

This definition generalises the partial generalised policy to different state spaces. Figure 4.2 illustrates how part of the mapping is bound to constants from a particular problem instance. As these policies are defined for a language, Σ, the solution will capture a solution to the problem in the states and actions in that particular restricted view. In the context of a specific problem this can be used to identify a single action sequence, or a plan. We observed in Chapter 3 that a consequence of Theorem 3.2.2 is that a plan for a language, Σ_i, on a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$, is usable as a plan for Σ_0 using co-execution. In Chapter 3, we considered the language enhancements with respect to a particular problem. Planning domains define a set of related problems that share operators and predicates and are commonly governed by domain conventions. We therefore consider that the selection of an appropriate language is made for the domain and each problem is enhanced using the same steps. The definition is made with respect to an enhanced domain, while in practice each problem is enhanced from the described model individually.

Our definition of a parameterised partial generalised policy generalises the definition of a *generalised policy* in (Khardon, 1999a) to a more general chain of language enhancements. It is *parameterised partial generalised* policies that we learn in this work and we refer to these as policies in the proceeding chapters.

4.2 Computability of the policy mapping

Parameterised partial generalised policies present an attractive formalism: the solution to groups of problems are represented by a single policy. The challenge is how to represent the policy in an efficient structure that can be manipulated by a computer program. In related work (Khardon, 1999a; Martin and Geffner, 2000; Fern et al., 2006; Levine and Humphreys, 2003) the mapping is divided into abstract situations. The argument is that even though there are an infinite number of possible states, they fall into a finite number of abstract situations. The key is that a single course of action is appropriate in each of the concrete states represented by a single abstract situation.

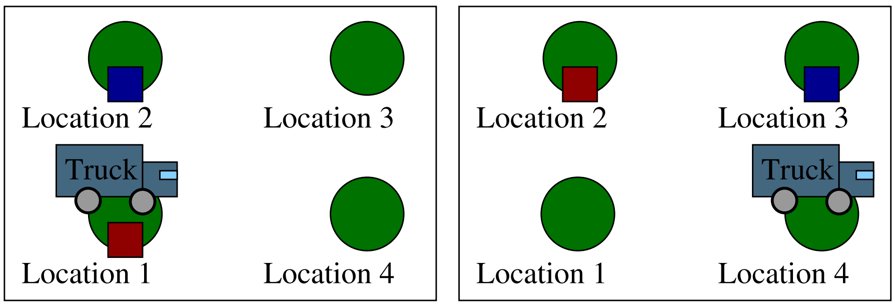
In Khardon (1999a); Martin and Geffner (2000); Fern et al. (2006); Levine and Humphreys (2003), each abstract situation and associated course of action is represented as a production rule. These individual production rules are combined to compute the mapping for the policy. The key issue is capturing the antecedent and consequent of each rule at an appropriate level of abstraction. If the rules are too specific then they will be expensive to exploit (Minton, 1990). If they are too general then the course of action will be ineffective for specific instances. The important point here is capturing control knowledge at the most appropriate level.

In this section we introduce the rule based policy (RBP), our representation for parameterised partial generalised policies. The rule antecedent evaluation context and the antecedent language are defined. In the next section, we compare the representation work to related work.

4.2.1 Policy representation

In this work we follow an approach to grouping the rules that has been applied in similar work (e.g. Khardon, 1999a). The rules are ordered and put in a list. The first rule that has a satisfied antecedent is selected and its consequent is interpreted as the value of the rule system. We will sometimes use the expression *rule firing* to express this selected rule. This ordering establishes a simple resolution mechanism, by placing a priority ordering over the rules the rules earlier in the list get the first chance to fire. This ordering impacts on the interpretation of later antecedents. In particular, the antecedents of rules later in the list are evaluated only when antecedents of earlier rules have not been satisfied.

There are several advantages of the rule representation. Rule systems are intuitive for humans to construct and intuitive for us to interpret the knowledge stored in them. Under certain assumptions, rules can be evaluated efficiently as part of planning. Another advantage is that the execution trace is observable: the reason for an action being applied is given by the fired rule.



(a) Example initial state (b) Example goal

Figure 4.3: An example transportation problem

We use a simple transportation problem, illustrated in Figure 4.3, to support the definition of our representation. The problem involves a single transporter that can move between any two locations in a single move. There are several packages that must be picked up and delivered by the transporter.

One way of defining a policy is as the following list of rules:

1. if the package is not at its goal location then pickup the package.

2. if the truck is at the package's goal location then drop off the package.

3. if the package is not at its goal location then move to the package's location.
4. if the truck contains a package and is not at the package's goal location then move to the package's destination.
5. if the truck is not at its goal location then move the truck there.

Early rules have priority and because of this the transporter will not move until all pickups and drop offs are completed at the current location. Otherwise the rules would have more precise antecedents. For example, before the truck moved to its goal then we should be certain that all packages have been delivered.

In this work we use a context that combines the current state and the propositions that hold in the goal. The antecedent of each rule is a conjunction of two formulas ϕ and ψ. The interpretation is the evaluation of the conjunction $\phi \wedge \mathbf{G}\psi$; ϕ is evaluated in the current state and ψ is evaluated in the goal.

An example drop off rule (rule two in the policy above) has $\phi =$ (at truck Glasgow) $\wedge$ (in spoon truck), and $\psi =$ (at spoon Glasgow). The antecedent is satisfied if the truck is at Glasgow with the spoon in it and the goal is to have the spoon at Glasgow. In general, this provides the context necessary for these formulae to distinguish between every state and goal pair and therefore represent a partial generalised policy.

4.2.2 Relational control knowledge

The definition of a policy in Definition 4.1.6 states that the policy is parameterised by the problem objects. In practice, the efficient representation of a policy requires that parameters be bound by need rather than as a single step *a priori*. This means that, when represented as rules, a policy will be captured by a collection of parameterised rules and a rule will be applied by determining a particular instantiation of the parameters that satisfies the rule condition, $\phi \wedge \mathbf{G}\psi$. There is a single set of variables for the antecedent and consequent of the rule. The formulae can therefore tightly constrain the instantiations of the actions. For example, the move to drop off rule can be expressed in the following way.

```
(:rule MoveBriefcaseToDropoff
    :condition (and (at ?bc ?from) (in ?obj ?bc))
    :goalCondition (and (at ?obj ?to))
    :action movebriefcase ?bc ?from ?to)
```

The formula, ϕ, asserts that a briefcase is at a location and that there is an object in it. ψ asserts that the object has a goal and its location is the same as the destination of the truck. The rule is applicable to a problem state if some briefcase, object and locations exist in the problem's constants so that the unified formulae are satisfied. The rule captures the concept of moving to drop off a package at its goal.

Parameterised rules are far more powerful: instead of referring to individual objects, a rule condition can capture the important relational properties of the objects that lead to a particular action selection. The rule representation presented here is equivalent to those used in L2ACT (Khardon, 1999a) and L2PLAN (Levine and Humphreys, 2003).

We have chosen a limited language for ϕ and ψ. The formulae are simply conjuncts of predicates. This allows us to study the effect of changing the problem model language with limited interaction between the context and the rule language.

4.2.3 Properties of the policy representation

The rule language can capture the mapping of a restricted set of policies. There are two limitations in particular: the antecedents of the rules rely on a finite number of propositions; and there are only a finite number of abstract situations that can be treated by a policy. In this section we define these as the set of *finite representable* propositions and the *finite distinguishing* policies. We will use these properties to motivate the investigation presented in Chapter 5.

Finite representable propositions

Definition 4.2.1 *For a problem model language, Σ, and the rule language presented in this chapter, a proposition, p, can only be modelled if there is a bounding parameter, n, such that when the number of symbols in the formula $\phi \wedge \mathbf{G}\psi$ is bound by n, the proposition can be modelled in any problem. If this does not hold for p then it can only be modelled in a subset of the problems. We call these propositions finite representable.*

For example, in a Transportation problem, using a standard modelling convention, determining whether a package can be reached by a truck in n steps requires $n + 2$ propositions. In a particular problem, a package could be any number of steps away and therefore a bound cannot be placed on n to assert whether a package can be reached by a truck. The reachability proposition is not finite representable in general. A proposition such as `three-steps-away` is finite representable.

In our approach, the symbols in ϕ and ψ are predicates that are determined for a particular domain model. For a particular binding, each predicate is instantiated and represents a specific proposition. This means that we can model propositions that rely on a fixed number of propositions (and therefore finite representable) in the well-formed formulae of Σ and this number does not change with the problem. A similar observation was made in Khardon (1999a).

Finite distinguishing

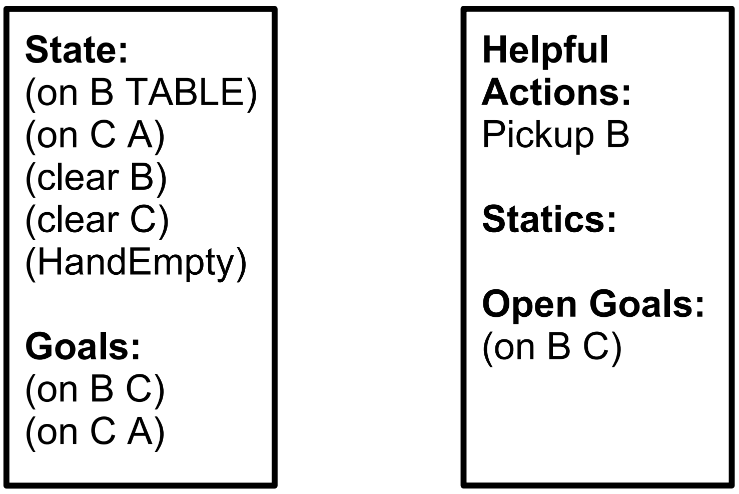
Definition 4.2.2 *For a given problem model language and policy representation, a policy is finite distinguishing if it only distinguishes between a finite number of abstract situations. As a consequence, there is a number, n, that bounds the number of partitions in the infinite number of states. This means that the correct behaviour for a particular state can only be determined using a maximum of n rules.*

In some problems the number of options will increase as the problem size grows in size. As there can be an arbitrary number of options, comparing these alternatives can require an arbitrary number of abstract situations. For example, in a transportation problem, to select the next best place to move requires comparing each of the alternatives, and characterising the differences between these, such as distance, number of packages, contribution to overall goals and so on, may require an arbitrary number of abstract situations.

For a given domain, an RBP represents a policy using a set number of rules, which each define the course of action for an abstract situation. This means that the states are only partitioned into a finite number of partitions and the policy representation is finite distinguishing.

4.3 Evaluation of the rule antecedent

There is a tight relationship between the rule condition language and the vocabulary provided by the context. The context for the rule condition evaluation defines the vocabulary that can be used to construct rule antecedents, or rule conditions as they are more commonly named. The antecedent language defines the way that the words in the context can be combined. As a consequence the context must include building blocks such that all propositions that are necessary for making action selection decisions can be modelled in the rule condition language. In particular, all approaches to capturing generalised policies have included the goal as part of the context (Khardon, 1999a;



(a) State and Goal Context

(b) Helpful Actions Context

Figure 4.4: Two examples of context information for Blocksworld.

Martin and Geffner, 2000; Yoon et al., 2002; Levine and Humphreys, 2003). In this subsection, we discuss how our representation compares with other approaches.

4.3.1 The current state

In the majority of the literature the current state is part of the context (e.g. Khardon, 1999a; Martin and Geffner, 2000; Fern et al., 2006; Levine and Humphreys, 2003). This is intuitive as the decision of the best action to make at the current state will usually have some dependence on observations made over the relationships in the current state.

In ROLLER (de la Rosa et al., 2008) the current state is not part of the context, instead the context includes a subset of the applicable actions called the helpful action set (defined in Chapter 2), the static literals from the problem and the open goals. The helpful actions capture a local view of the relationship between the state and the goal. As such the helpful actions and the static facts are seen as a substitute for the current state. In Figure 4.4 the helpful action context is shown for a Blocksworld problem alongside a more common context composed of the state and goal. The set of statics for Blocksworld problems will provide no contextual advice between problems as it is empty for any problem. Only those goals that are not satisfied in the current state are available for reasoning.

When the state is used in the context the rule condition has to reason directly over the rich problem structures, such as stacks of blocks in a Blocksworld problem. To express these conditions we either rely on concepts being added to the description of the state (Khardon, 1999a), or the rules must be expressed in a rich language (Martin and Geffner, 2000). The use of the helpful action context in ROLLER avoids this issue as an explicit representation of the state is not used. For example, in Blocksworld problems, it is often necessary to reason over a stack of blocks, represented as several `on` relationships. Consider three blocks in a stack `A, B, C`, with `C` at the bottom, as illustrated in Figure 4.5. If the goal is simply to have block `C` clear then we must first remove block, `A`, from block, `B`. In order to realise this requires that the planner understands that block, `A` is on block, `B` and block, `B` is on block, `C` and that block, `B` cannot be moved until it is clear. It is demonstrated that the helpful action context is suitable for certain domains (de la Rosa et al., 2008). However, in domains with resource management such as Zeno Travel and Logistics it is reported that the context is not sufficient for action selection.

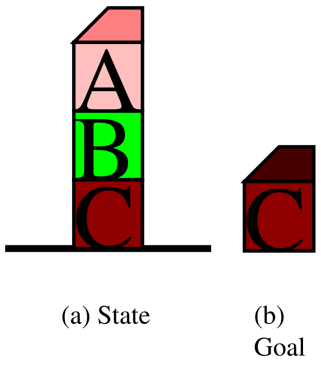


Figure 4.5: Three blocks stacked on top of each other. The goal is to free the bottom block.

4.3.2 Achieved goal context

In Yoon et al. (2002) the context is extended with derived propositions that model the achieved goals. Concretely, an achieved goal proposition holds if the proposition is true in the state and goal. These propositions are particularly useful for constructing recursive propositions, such as the `well-placed` proposition for Blocksworld problems. A block is *well-placed* if it is in its goal position and all blocks underneath it are *well-placed*. Due to the limited language we we have selected, this context can be simulated for a proposition p, by including p in the conjunctions of ϕ and ψ.

4.3.3 Planner specific context

Other contexts have been used to support rule application in related areas. For example, the actions in the relaxed plan along with their add and delete effects are included as part of the context in Yoon et al. (2006). This is appropriate, as the rules are used to improve a relaxed plan heuristic search. In particular, the aim is to compensate for the weaknesses of the heuristic. It is demonstrated that these aspects of the context are important in learning effective rules. This has also been used in Obtuse Wedge (Yoon et al., 2007). One of the configurations of Obtuse Wedge uses an RBP as a probe to generate the neighbours in a best-first search. In Yoon et al. (2008) the impact that this context has on the planner and this is positive in most of the tested domains. Whether it will have a similar impact on direct RBP application is unclear.

The rules in PRODIGY are also used to assist heuristic search. In Section 2.3 we presented the decision points that can exploit control knowledge. The decisions that are made at these decision points are included as part of the context. This allows the choices that have been made to effect future decision making. In our approach the language for communicating decisions is through action selection. However, in Chapter 3 we observed that the ordering of action selections do not accurately reflect the order that decisions are made. In order to make these selections we often have to make choices that are relevant to future action selections. In Chapter 5 we will investigate how recording these decisions in the enhanced model can save repeating the decision process in subsequent steps.

4.3.4 Rule language

A common difference in rule representation is to combine the state and goal antecedent formulae ϕ and ψ. The benefit of this change is that the antecedent is a single formula that can include both goal and state propositions. The predicates of the goal propositions are relabelled so that a distinction can be made between propositions in the state and propositions in the goal. Similar approaches are made for other extensions to the context. However, our rule language restricts the two formula to conjunctions of the propositions so the separation has no effect on expressivity. Similar relabelling is used to include the achieved goal context (Yoon et al., 2002) and relaxed plan context (Yoon et al., 2006).

Several of the state of the art rule learners use a concept language instead of a predicate logic to express their rules. This was motivated by the observation that it is often useful to reason about classes of objects when making action choices (Martin and Geffner, 2000). The concept languages allow certain propositions to be derived from the described state. For example, in the concept language used in Martin and Geffner (2000) the blocks that in the goal are positioned above blocks that are currently well-placed can be expressed as: $(\forall on_g^* \, . \, on_s = on_g)$. It is demonstrated that effective rules expressed in concept languages can be learned (Martin and Geffner, 2000; Yoon et al., 2002).

However, the rules expressed in these languages can be difficult to understand. This can make the control knowledge difficult to interpret and validate. For example, in (Yoon et al., 2006) they are unable to provide any intuitions as to what knowledge their rules describe. Also, there are certain types of propositions that cannot be expressed in the languages (de la Rosa and McIlraith, 2011; Fern et al., 2004). In general, there is a trade off between selecting a concise language and selecting a language that generalises to a large set of domains. Our approach is to identify attributes of problem models that are difficult to express in control knowledge and model them in an enhanced problem model.

We want to investigate how the vocabulary available for expressing rule antecedents influences the policies that we can represent. We have chosen a basic rule language so that our results are not influenced by the use of the vocabulary. It has the benefit that the rules are clear and understandable. It should also be noted that we do not lose any power of expression by using this limited language. This is because that any proposition that can be derived from the language, Σ_i, is expressible in some language, Σ_j, further along a chain of language restrictions.

4.4 Alternative generalised policy representations

There are alternative representations that lead to plans that are applicable in more than a single problem. Of course a single sequence of actions can be a solution to a set of problems that are strongly related. Approaches to solving MDPs generate policies as defined in Definition 4.1.1, which can solve all of the problems that share a goal and object set. Generalised policies have been represented in a program-like language in LoopDISTILL (Winner and Veloso, 2007). In this subsection we discuss the relationship between generalised policies and the plans generated in conformant planning and plan-space planning.

4.4.1 Conformant planning

In conformant planning (e.g. Smith and Weld, 1998) the initial state is not fully defined. This means that there is a set of potential initial states. As actions are applied to the world there are a set of possible states. The problem is to discover a sequence of actions that will achieve the goal for all of the possible initial state configurations.

This is related to the problem of capturing a policy: the problem has a single set of objects and a specified goal and a policy can be used to solve the problem for any initial state configuration. The generality of the problem sets solvable by a conformant plan is equivalent to our original definition of a policy (Definition 4.1.1). In contrast a generalised policy can solve problems with different sets of objects and different goals.

However, in conformant planning the initial state configuration is not known at execution so a single plan has to work for any of the possible configurations. This means that action selection must be done with care. Application of an action must be safe in any of the states that are possible progressions from an initial state. In contrast to this, when using a policy, the initial state must be known at execution so that the action to apply can be looked up in the policy mapping.

Another difference is that in conformant planning the uncertainty in action application is the result of actions with conditional effects combined with the uncertainty in the state, whereas policies can be used with problems that have actions with any form of uncertainty. In this work we focus on deterministic problems.

4.4.2 Partially ordered plan

The plans discovered by plan-space planning are partially constrained (Ghallab et al., 2004). In particular, the actions in the plan might contain variables. These elements are constrained so that any instantiation leads to a valid plan. However, they also add some variability into the plan. This means that the same plan can be applied to several problems, in the sense that more than one binding can be applied to the same plan. However, this is reliant on particular relationships between the initial states and goals of problems.

The main flexibility of partially ordered plans is in the actions selected to solve the problem. Of course, this must be restricted so that the solution is constrained to ensure that any instantiation of the plan is a solution plan. In the common representation a fixed number of operators are selected that will appear in every instantiation of the problem (e.g. Penberthy and Weld, 1992). This forces certain instantiation decisions to be made and thus reduces the generality of the solution. A partial order plan can generate different orderings of actions that cannot be captured by a policy formalism. However, unlike the generalised policy, the number of actions is defined in the plan, and the plan is intended to solve a single problem.

CHAPTER 5

AN ENHANCED PROBLEM MODEL

We have observed that the described model of the planning problem does not always provide the necessary vocabulary for expressing general policies. In Chapter 3 we proposed a rich planning model that would support planning. We defined a chain of language restrictions that bridges the gap between the described model and this ideal planning model. The mapping of our general policy (Chapter 4) is computed using rules that are represented in the language of the problem. This means that as we enhance the language of the problem the mapping computation is made more powerful.

The theoretical model provides a framework and in this chapter we show how this can be used in practice via the selection of meaningful samples of possible chains. An important step in this direction is to interpret the limitations of the RBP representation within the framework of language steps. The limitations lead to poorly specified control knowledge in some domains. We identify the general problem classes that characterise the benchmark domains that are effected. The limitations and problem classes guide the selection of chains and we investigate chains that support RBPs in these domains.

In this chapter, we develop an architecture that supports exploration of the chains defined in Chapter 3: this architecture can be used to co-execute a policy in the enhanced model and the described model. We characterise the planning problems that cannot be treated directly with the RBP. We identify three chain step properties and use the problems to explore each, in order to develop a selection of appropriate planning models.

5.1 Enhancing the problem model

In Chapter 3 we defined the steps in a chain of language restrictions and in this chapter we will consider specific steps that we could make. It is the aim of this section to present our architecture that allows these steps to be realised. More precisely, the approach that we have taken to co-execute an RBP in an enhanced problem model and the described model.

We begin by defining *special purpose solvers*, as these play a key role in enhancing the language and interpreting the richer language in the vocabulary of the original problem model. We continue by presenting the architecture and defining the extension of PDDL that we use to express enhanced domain models.

5.1.1 Special purpose solver

A special purpose solver is an implementation of a step in a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$. Although the functions of solvers are arbitrary they adhere to an interface that allows them to be treated uniformly. The solvers implement general steps that are parameterised by a specific domain and problem. We present our description for a parameterised special purpose solver.

Each solver, $solver_i$, has a set of actions and propositions that it models: $\mathbf{A}_{solver_i}$ and $\mathbf{P}_{solver_i}$. Each solver also has a second set of propositions, s_{solver_i}, that represents the propositions modelled in the current state. Finally solvers implement an `apply` function: $\mathbf{apply}_{solver_i} : \mathbf{A}_{\Sigma_i} \mapsto \mathbf{A}^*_{\Sigma_{i-1}}$ and an interpretation function, $\mathbb{I}^{\Sigma_{i-1} \to \Sigma_i} : \mathbf{S}_{\Sigma_{i-1}} \mapsto \mathbf{null}$. The interpretation function is used to initialise the internal state of the solvers. The `apply` function applies an action to the solver's part of the model and then returns the translation as a finite length action sequence in the previous language in the chain. This can also be the null action ($\perp$) if the change is not built from actions in lower languages.

For convenience we wrap the default behaviour of the described model, $\mathbb{M}|_{\Sigma_0}$, as the base solver, $solver_0$. The solver models the actions and propositions that are modelled in $\mathbb{M}|_{\Sigma_0}$: $\mathbf{A}_{solver_0} = \{a \mid a \in \text{wff}(\Sigma_0)\}$ and $\mathbf{P}_{solver_0} = \{p \mid p \in \text{wff}(\Sigma_0)\}$; the current state, s_{solver_0}, is initialised as s_{init}, the initial state of $\mathbb{M}|_{\Sigma_0}$. The apply function for $solver_0$ simulates the model's transition function and returns $\perp$:

```
def apply_{solver_0}(a) :
  $s_{solver_0} = \gamma_{\Sigma_0}(s_{solver_0}, a)$
  return $\perp$
```

This solver captures the behaviour of $\mathbb{M}|_{\Sigma_0}$.

We can derive the set of modelled actions and propositions after i steps through the chain of language restrictions as the union of the previous sets with the sets modelled by the solver:

$$\mathbf{A}_i = \mathbf{A}_{i-1} \cup \mathbf{A}_{solver_i};$$
$$\mathbf{P}_i = \mathbf{P}_{i-1} \cup \mathbf{P}_{solver_i}.$$

Each solver models its set of propositions and the actions that it defines can effect those propositions. If an action defined further along the chain changes the propositions then this is achieved through the language of this solver. To apply an action, $a(\Sigma_n)$, we apply the following recursive function:

```
def stateChange($a(\Sigma_i)$) :
  $a_0, \ldots, a_n(\Sigma_{i-1}) = \mathbf{apply}_{solver_i}(a(\Sigma_i))$
  if $a_0, \ldots, a_n$ $== \perp$ : return
  for $j = 0 \ldots n$ :
    stateChange($a_j(\Sigma_{i-1})$)
```

This function is passed an action, a, in some language, Σ_i. The associated solver is used to apply the action, with the possible side-effect of creating a single, or a sequence of actions expressible in the language, Σ_{i-1}. The action sequences are taken in turn and each is applied through recursion. This is sufficient when a language step that models an abstract action is formed of actions from the previous language and has no side-effects, which is consistent with the framework developed in Section 3.1. As the actions are composed from previous layers and $solver_0$ always returns $\perp$, the loop will complete if the `apply` functions complete.

The state of the current enhanced model is the union of the partial states modelled by each of the solvers:

$$s_C = \bigcup_{i=0}^{n} s_{solver_i}.$$

So far we have dealt with the progression of states. However, we have not mentioned how the initial state of the solvers is determined. Each solver has an interpretation function, $\mathbb{I}^{\Sigma_{i-1} \to \Sigma_i}$, that takes as input a state in the previous language. The solver uses this state to initialise the propositions that it models. In other words, compute the set s_{solver_i}.

Our chain definition means that enhanced propositions are either richer interpretations of states, or they are commitments to future decisions. The former case results in a one-to-one mapping between states; the latter case implies a state where the commitment has not been made. We observe that it is intuitive to select the state with no commitments.

5.1.2 The architecture

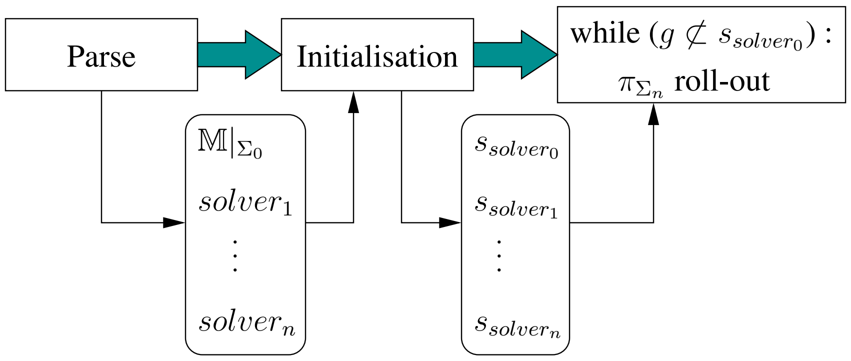


Figure 5.1: The architecture.

In this subsection we present our architecture. The architecture can be split into three phases: parsing the model, the initialisation of the models and policy roll-out, illustrated in Figure 5.1. The model is expressed in terms of special purpose solvers and their parameterisation for a particular domain. The language for expressing the enhanced domain models is defined in subsection 5.1.3. The enhanced domain model and problem models are parsed resulting in a list of solvers: $solver_0, \ldots, solver_n$. The solvers are initialised using the specific problem and then the policy is repeatedly applied until the goal is achieved. In this subsection we explain the initialisation phase and the policy roll-out process.

Initialisation

The initialisation of the system establishes the state of the two models: $\mathbb{M}|_{\Sigma_0}$ and $\mathbb{M}|_{\Sigma_n}$. The process is illustrated in Figure 5.2.

The model $\mathbb{M}|_{\Sigma_0}$ is presented as part of the input to the planner and is used to construct $solver_0$ as described above.

The main aspect of this phase is initialising the solvers' states. This is achieved by incrementally enhancing the state. The i^{th} solver can derive its state from a state in Σ_{i-1} and s_{solver_0} is defined as the initial state. Therefore, we can compute the states of the solvers iteratively:

$$s_{solver_1} = \mathbb{I}^{\Sigma_0 \to \Sigma_1}(s_{solver_0}), \ldots, s_{solver_n} = \mathbb{I}^{\Sigma_{n-1} \to \Sigma_n}(\textstyle\bigcup_{i=0}^{n-1} s_{solver_i}).$$

At each step the formula that was presented for computing the state is used on the set of solvers initialised so far. This state is then interpreted by the next solver, which

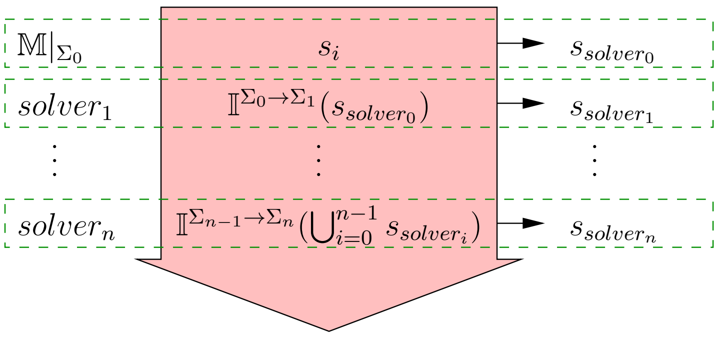

$\mathbb{M}|_{\Sigma_0}$
$solver_1$
$solver_n$
s_i
$\mathbb{I}^{\Sigma_0 \rightarrow \Sigma_1}(s_{solver_0})$
$\mathbb{I}^{\Sigma_{n-1} \rightarrow \Sigma_n}(\bigcup_{i=0}^{n-1} s_{solver_i})$
s_{solver_0}
s_{solver_1}
s_{solver_n}

Figure 5.2: The initialisation of the enhanced model.

initialises its propositions in the context of this state. This process is complete when all solvers are initialised. We now have the initial state in $\mathbb{M}|_{\Sigma_0}$ and can construct the initial state for $\mathbb{M}|_{\Sigma_n}$. We are ready to proceed with co-execution of the models.

Policy roll-out

At this stage we now begin the process of policy roll-out. The process is illustrated in Figure 5.3.

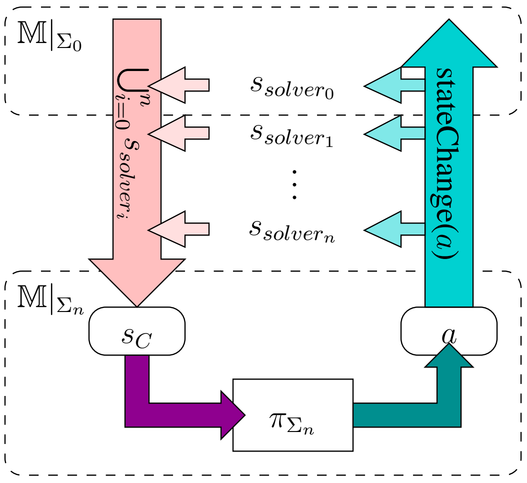


Figure 5.3: The exploitation of solvers to compute the enhanced model.

The current state is computed by combining the partial states of the solvers:

$$s_C = \bigcup_{i=0}^{n} s_{solver_i}$$

The policy, π_{Σ_n}, maps to an action, $a(\Sigma_n)$. This action is then applied to the current state using `stateChange`(a), as defined in subsection 5.1.1. As the `stateChange` function propagates the action through the solvers, the action is interpreted for limited languages. In particular, a solver that models propositions that are effected by the action makes these changes themselves. As such the state of $solver_0$ represents the state of $\mathbb{M}|_{\Sigma_0}$. This means that when $g \subset s_{solver_0}$ the problem is solved.

5.1.3 Enhanced model description

Dornhege et al. (2009) define an extension to the PDDL language called PDDL/M. PDDL/M defines an interface for extending a problem model using special purpose solvers. However, there are several conceptual differences between our work and theirs. This means that we have developed an alternative language for integrating solver based domain extensions.

There are two main difference between our work and the work in Dornhege et al. (2009). The first is that in PDDL/M the solvers model a richer set of propositions and action effects, whereas the solvers in this work model a richer set of propositions and *actions*. This means that we have to extend the language so that we can define richer actions instead of effects.

The second difference is that the solvers that are the focus of Dornhege et al. (2009) introduce new information to the model. They are static procedures that behave largely independently to the propositional state. In contrast, our solvers enhance the information in the problem. They are parameterised and there can be several instances of the same solver. As such the assumptions of static solver methods and a modular separation of the solvers are not appropriate.

In this subsection we define the extension to PDDL that we use to define enhanced models. Our approach is split into two files: the first describes the enhanced domain model; the second is the solver listings file that describes the solvers and modules that provide the implementation of the domain enhancements. An example of these is listed in the Appendix B. We examine how our approach resembles PDDL/M, and how and why it deviates from it.

An extension to PDDL

We extend STRIPS with four declaration constructs: a declaration for each solver; a link to a file defining the solvers; a declaration of the active predicates; and a declaration for each of the active actions. A different approach is used in PDDL/M: a separate modules section is used to define all of the extended language. Our language serves to extend the model using syntax similar to the existing PDDL constructs. The language for defining solvers is detailed in the following part.

Solver definitions There are two parts to the solver declarations. There is a path to the listings file and there is a header for each solver used in the model. In particular, if a solver is relied on to compute an active predicate or action then it should have a solver heading in the model definition.

The path indicating where the solver listings file is located is declared using the `solverListings` tag:

```
(:solverListings SolverListingFolder/DriverlogListings)
```

Solver headers are declared using the solver tag and a type tag; they declare the name and type of the solver. We require both of these as there can be several instances of the same solver. This is an important difference from PDDL/M where there is a single instance of each solver. For example, if there is a road and a path in a problem then we have a single solver and make an instance for each structure. The same problem would require separate solver descriptions in PDDL/M.

The name is the name that is used to reference the solver in following declarations. The type is the type of solver. This is a link to the Java class that the solver is an instance of.

```
(:solver GraphAbstraction0
        :type solvers.solvers.GraphAbstraction)
```

Active predicates play the role of communicating the value of propositions that are modelled by the extended model. The active predicates are defined using an `activePredicates` tag in a manner analogous to predicate definitions in PDDL. Once an active predicate is defined it can be used in the conditions of actions. An example of the syntax for active predicates is:

```
(:activePredicates
    (drive-truck_connected ?from - location ?to - location)
    (walk_connected ?from - location ?to - location)
    ...
)
```

In Dornhege et al. (2009), active predicates are called *condition checkers*. They are defined individually and declare the method that should be called to determine the value of the condition. As our solvers are generated they register the predicates that they determine.

Enhanced-actions are actions that are modelled in the extended model. Our syntax follows PDDL for operators; except we use a separate `activeAction` tag, and the effect is replaced with a reference to the solver that realises the effects of the action. An abstraction of the `drive-truck` operator in Driverlog is presented here:

```
(:activeAction long_drive-truck
    :parameters
    (
        ?truck - truck
        ?loc-from - location
        ?loc-to - location
        ?driver - driver
    )
    :precondition
    (and
        (drive-truck_connected ?loc-from ?loc-to)
        (at ?truck ?loc-from)
        (driving ?driver ?truck)
    )
    :effectApplier GraphAbstraction0
)
```

In PDDL/M there are effect-applicators that model single action effects instead of modelling the application of an action. They have a mixed semantics. Their primary function is to make the prescribed change to the extended state. However, they can also update the numeric fluents. Effect applicators can be used anywhere that an action effect can be applied. However, in PDDL/M they are surrounded by square brackets. An example of an effect-applicator in the PDDL/M syntax is shown below:

```
(:modules
    (putDown ?p - package ?t - truck
    (p0)
    (powerLevel ?o)
    effect putDown@libTrajectory.so)
)
```

After the predicate name and parameters there are values that are set by the solver. In the example the value designated to $p0$ by the solver will become the new value of `powerLevel` in the state. In this example there is only a single value set, but there can be any number. This feedback declaration allows for a clear indication of the values that can be changed by external actions. However, only numeric fluents can be manipulated by the solvers in this way.

Our language is limited to propositions. However, the solvers can change the propositions in the state and therefore could change numeric fluents if they existed. However, due to the chains that we investigate, the changes would have to act on fluents in the enriched model, or be the equivalent of the effect of an action in Σ_0.

In our language we reference one of the defined solver instances. In PDDL/M a function in a C library is referenced (for example, `putDown@libTrajectory.so`).

In both cases the reference indicates where to find the action applier.

A solver instantiation language

The solver instances relied on by a particular enhanced domain model are specified in a separate file called the solver listings file. In this file solvers and the modules that these solvers depend on are described. The first part of the listings provides a header for each module; because the solver headers are part of the domain model, this means that the name and type of each module and solver is declared up front. The following part of the listings file describes each of the modules and solvers.

Module headers are defined in the same way as the solver headers, except using a module tag.

```
(:module StaticGraphModule2
    :type solvers.graphabstraction.StaticGraphModule)
(:module StaticGraph1
    :type solvers.graphabstraction.StaticGraph)
(:module MoveAction0
    :type solvers.graphabstraction.MoveAction)
```

The name is used to reference the module in the descriptions that follow. The type is a reference to java class.

Solver and module descriptions parameterise the solvers and modules using the `solverDescription` and `moduleDescription` tags. Each solver and module has its own collection of parameters or attributes. These parameters take the form of references to modules and descriptions using the module and description tags. The modules provide specific implementations based on the domain properties. For example, the module that specifies the graph implementation will provide the implementation for a static, turn-based or dynamic graph.

```
(:solverDescription GraphAbstraction0
    :Encoding (:module StaticGraphModule2)
)
(:moduleDescription StaticGraphModule2
    :Map (:module StaticGraph1)
    :MoveAction (:module MoveAction0)
    :EnablingPredicates
        (:description (driving ?driver ?truck#))
)
(:moduleDescription StaticGraph1
    :MoveAction (:module MoveAction0)
    :MapPredicates
        (:description (link ?loc-from ?loc-to#))
)
(:moduleDescription MoveAction0
    :MoveAction (:description (drive-truck 0 1 2))
    :Locatedness (:description (at 0 1))
)
```

The solvers in PDDL/M are implemented as C libraries and the reference to solvers are in fact a reference to the C library where the necessary implementation can be found. In Dornhege et al. (2009) a standard interface is defined that must be implemented by the libraries. We insist that attributes are either other modules or strings with correctly nested brackets. We use the reflection API[1] in Java. If a description or module tag exists in the solver listings file then the implementing java class is expected to have appropriate methods for accepting the parameters. A description method is passed the attribute name and the description string. A module method is passed the attribute name and a reference to the module.

This language provides a way of explicitly defining the steps along a chain of language restrictions. In Chapter 7 we explain how these extended models can be generated automatically.

5.2 Structures

Structures are concepts that provide labels for sets of relationships that share certain properties. These relations can exist between arbitrarily many objects. Structures will not necessarily be made explicit in a model: a structure can be implied through a series of transitions and their associated constraints. This leads us to use the term structure interaction (SI) to refer to related behaviours that act over structures; for example, iteration through the nodes of a structure. This definition will be made more precise in later chapters.

In this work, we have focussed on the set of benchmark planning domains used in the planning competitions. However, structures are important concepts in a wider collection of domains: for example, it might be instructive to reason about structures in General Game Playing (Genesereth et al., 2005). Although the discussion here will assume that the planner has control on all changes in the world, the ideas could provide useful concepts in environments where this does not hold; for example, reasoning over sequences of individual moves to predict a future position in a two player game. In the planning benchmarks, the SIs fall into two groups: structure traversing, and

[1]http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/reflect/package-summary.html

structure building. Traversal problems involve moving an object between different locations. Structure building problems involve some notion of attachment of objects and involve reorganising the objects so that they are attached in a different configuration. Structures are particularly important in planning as they constrain the possible interactions with the environment. This can cause dead end states, where the goal cannot be reached, and impact on the number of actions required to transition between states, which increases the importance of individual action choices in the context of overall optimality. A structure can also force the planner to commit to a course of actions many steps before the actions can be applied. In this section we define the traversal and the building problems and present a selection of important derivatives of these that generalise important trends in benchmark problems.

5.2.1 Traversal problems

Traversal problems involve moving an object through a constrained structure. This is particularly important in modelling problems with spatial relationships. In these problems traversers move between different locations on a map. An important aspect is that the location of the traverser constrains the actions that the traverser can perform. For example, in a transportation problem, a traverser picks up packages, but this can only be done at its current location. The set of objects that can be moved by the move action are called the traversers and denoted $\mathbb{T}$. The set of positions that an object can be located are called the locations and are denoted $\mathbb{L}$. We define a function that maps the traversers to their current location: $\textbf{position} : \mathbb{T} \mapsto (\mathbb{L} \cup \perp)$ and refer to this as the location of the traverser.

A key aspect of the traversal problem is the action that moves objects between locations. The move action, (*moveAction* $t\ l\ l'$), has three parameters: the traverser, t, the traverser's current location, l, and the destination of the move, l'. In practice, a move action could have more parameters; we represent these constraints through an accessibility graph. The vertex set of the accessibility graph are the locations in $\mathbb{L}$. The edges of the graph represent the moves that can be made between the locations. We construct this graph by applying move actions from the current state. An edge exists in the graph if there is a move action that connects the locations and exists on a chain of move actions from the current state.

$$
\begin{aligned}
&(l_i, l_{i+1}) \in E \iff \\
&\quad ((\textit{moveAction } t\ l_0\ l_1)\ .\ s_1 = \gamma(s_0, (\textit{moveAction } t\ l_0\ l_1))\ . \\
&\quad \ldots . \\
&\quad (\textit{moveAction } t\ l_i\ l_{i+1})\ .\ s_{i+1} = \gamma(s_i, (\textit{moveAction } t\ l_i\ l_{i+1}))\ . \\
&\quad \ldots)
\end{aligned}
$$

The edges are defined for a particular traverser, t. We therefore have a a set of graphs, $G_s(t) = (V, E)$, for each state, $s \in \mathbf{S}$. This graph identifies the locations that can be reached by the traverser in the current state using the move action.

Transportation problems

A transportation problem is a traversal problem where the traversers are used to service package deliveries. As a convention we relabel traversers that deliver packages as transporters. We extend our language for traversal problems with a set of portables, $\mathbb{P}$. We augment the position function to map from portables or transporters to locations:

$$\textbf{position} : (\mathbb{P} \cup \mathbb{T}) \mapsto (\mathbb{L} \cup \bot).$$

Our model of transportation abstracts several elements found in transportation benchmark problems. A more comprehensive model has been developed in Helmert (2001).

Path opening problems

Path opening problems are traversal problems where a path must be established between the traverser and a specific target. In general establishing a path requires consideration of the problems over general dynamic graphs. However, in this chapter we focus on a particular subset of these problems that we call *turn based* graphs. This is because the path opening problems in the benchmark domains can be explained within this structure. We consider a more general set of problems in Chapter 7.

The traversal graph can be changed by opening actions. A solution requires a series of graph manipulations that establish a path to the target. This process is complicated by constraints on the opening action, such as proximity of the traverser to open a node and nodes locked with keys. The nodes in the graphs can be open or closed; if a node is closed then a traverser cannot traverse to that node. There is a transition graph that defines the edges that can be traversed and there is an underlying graph that defines the potential edges. In particular, if a node is open then the edges in the potential graph are in the transition graph. A static graph to captures the potential accessibility graph for a traverser, t.

Definition 5.2.1 *The graph is static if for any state the edges in the accessibility graph are the same:*

$$\forall s_0, s_1 \in \mathbf{S}\ G_{s_0}(t) = (V_{s_0}, E_{s_0})\ .\ G_{s_1}(t) = (V_{s_1}, E_{s_1})\ .\ E_{s_0} = E_{s_1}.$$

As a consequence the graph can be computed once and used for any state. We use $SG(t)$ as the set of graphs for any state. A turn based graph is represented by a set of static graphs, $SG(t)$, and a mapping from vertices to the Booleans (closed/open), $o_s : \mathbb{L} \mapsto \mathbb{B}$. The interpretation of the mapping is that if $o_s(l) = true$ then in state, s, the node, l, is open, otherwise it is closed. If l is open then for traverser, t, every edge, $(l, l') \in SG(t)$ is in $G_s(t)$.

5.2.2 Structure building problems

Structure building problems involve connecting similar objects together to form a particular structure. A key concept when planning in a structure building problem is whether the current structure is correct. In this part we define the important actions and a graph that represents a state's structure.

A structure building problem is characterised by an attach action, (*attach* o_1 o_2), and a detach action, (*detach* o_1 o_2), each with two parameters: the objects being attached, or the objects being separated. There is a single graph, $G_s = (V, E)$, for every state in this problem. V has a vertex for each attachable object and the edge set has an edge between each attached pair of objects, $(o_0, o_1) \in E$. The aim in these problems is to perform operations on the graph so that it matches a specific graph. In the benchmark problems the structures are defined in goals; however, they could also exist as the precondition to an action. In this work we are interested in structures that are defined in the goal of the problem as these structures might be required to be of arbitrary size. Structures required to satisfy action preconditions have only to be of fixed size. In this work we consider a specific version of structure building problem called a *stacking problem*.

Stacking problem

A stacking problem is a structure building problem that is constrained so that the structures act like stacks. There is the concept of the top object of a stack, which is the only object that can be detached from the stack. An object can only be attached to another object if it is on top of a stack. The initial state is constrained so that the maximum number of incoming edges to each vertex is one and the maximum number of outgoing edges is also one. The objects will be referred to as blocks, as is the case in the Blocksworld domain. It has been observed in Yoon et al. (2002) that an important property in stacking problems is the relationship between the state and the goal. A collection of propositions is defined that represent the intersect of the state and the goal propositions. If there is a path from a clear block down to a block on the table then the block is well-placed.

5.3 Selecting meaningful chain steps

In Chapter 4, we observed that the policy representation was limited to finite representable propositions and a finite number of abstract situations (finite distinguishing). There are problems in planning, as we introduced in the previous section, that require reasoning with problem specific structures, which are naturally constructed from an arbitrary number of relationships. Researchers have investigated the issues of finite representability in certain domains, by extending the rule language with the transitive closure (Martin and Geffner, 2000; Fern et al., 2006). An alternative approach was explored in Khardon (1999a); Levine and Humphreys (2003), where the problem models were enhanced with recursive support predicates. These approaches have been demonstrated on block stacking problems. However, it was observed in de la Rosa and McIlraith (2011) that these features cannot provide guidance between alternative reachable targets. For example, identifying a path between two locations.

In this work we will further the study of using RBPs over arbitrary structures. In this section we interpret the limitations of the RBP representation as categories of chain steps that can lift the limitations. The first two categories: identification and selection between alternatives and optimising the selection between alternatives, are motivated by the limitations we have observed in Chapter 4 and the observations made in the literature (de la Rosa and McIlraith, 2011). We also investigate the effect of varying the level that these enhancements are expressed, in order to understand how the level of reasoning impacts on the support provided to the RBP.

Directed connectivity In order to solve structure based problems the planner must be able to direct search through various SIs, including traversing a graph and stacking blocks. In de la Rosa and McIlraith (2011) it was observed that the current rule languages and evaluation contexts are not sufficient to make individual steps that contribute to a single target. For example, selecting move actions that combine to reach a specific node. Each step is made in isolation and therefore the RBP selects to move towards one of the reachable locations from this state and a different location in the next state. To direct search through SIs, the planner must be able to identify two pieces of information: what are the possible SIs and how each of the possible SIs can be done. This information we call the context of *directed connectivity* for a particular SI. This context provides the options for the planner so that it can choose what should be done. The first category that we explore are chain steps that establish directed connectivity for specific types of SI.

Optimisation Of course, there could be many applicable SIs. The selection between these alternatives can greatly effect the resulting plan length. For example, a transportation problem requires sequences of move actions between locations; to select these effectively requires the careful allocation of packages to trucks. The RBP can only define a finite number of abstract situations, whereas there can be an arbitrary number of structure configurations and each of these can imply a different ordering. This motivates identifying characterising features of the SIs that can be used to distinguish good action choices. The second category we explore are the chains that provide more information to distinguish between alternative SIs.

Level of reasoning In the final category we explore the alternative levels of reasoning. The framework accepts languages that span from languages that model actions that break the planning process into individual decision steps, leading to much longer plans, to languages that model actions that solve the problem in a single action. The level of reasoning is an important aspect in expressing concise control knowledge. We examine alternative chain steps for different levels of reasoning.

In the following sections we will examine each of these types of chain steps and develop alternative models for lifting the limitations of the RBP representation. We will evaluate the enhancements in Chapter 6 and further discuss the properties for each of the alternatives we define.

5.4 Directed connectivity

Directed connectivity is a property that relates properties of a structure to the utility of reasoning with the properties within search. In particular, we say that if a property of a structure supports an RBP to identify the possible SIs on the structure, and action those SIs then the property establishes directed connectivity for that SI. For example, it has been shown that connectivity is sufficient to establish directed connectivity in stacking problems (Martin and Geffner, 2000; Yoon et al., 2002), in terms of the SIs of uncovering specific blocks. This is because any block can be uncovered by unstacking from the top of its stack, which can be identified through connectivity. However, it is not sufficient for determining the appropriate steps to perform graph traversals (de la Rosa and McIlraith, 2011) and its derivative problems. We will typically refer directly to the vocabulary that represents the property in the planning model. In this section we examine directed connectivity in the context of the traversing and structure building problems.

5.4.1 Connectivity

The concept of connectivity is important for identifying the possible SIs. The concept of two nodes being connected in a graph is well defined: for a graph, $G = (V, E)$, u is connected to v if there exists a path from u to v; in other words, two nodes are connected if they are linked by a series of edges. In directed graphs, the series of edges must respect the direction of each edge. For convenience, we use $(v_0, \ldots, v_k) \in E$ to denote that v_0 is connected to v_k.

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the connectivity between pairs of nodes. For a particular state $s \in \text{wff}(\Sigma_i)$ and graph, $G(s) = (V, E)$, we define the set of propositions:

$$\begin{aligned}
&\forall u, v\ (u, \ldots, v) \in E \iff \\
&\qquad (\forall s' \in \text{wff}(\Sigma_{i+1}) \\
&\qquad\quad s'\mathbf{R}s \implies s' \models (\text{connected } u\ v))
\end{aligned}$$

Use of connectivity

The transitive closure is effective for expressing rule systems for stacking problems, as has been demonstrated in Martin and Geffner (2000); Fern et al. (2006).

Definition 5.4.1 *The evaluation of the transitive closure, $R*$, of a binary predicate, R, holds for two arguments, x and y, if there exists a chain of R relationships, in some context, X, that connect x and y.*

The transitive closure can be represented as the connectivity property on the graph with objects as the vertices and an edge for each instantiation of the predicate in the state from the first argument to the second. If we consider a Blocksworld domain where moving a block is achieved in a single action, then we can express a policy using the connectivity propositions using the context of the `on` predicates in the state. We consider the more common detach and attach representation in Appendix H.

```
(:rule PopBadTower1 (A B C D)
    :condition (and (on A B) (clear A) (on* A D) (onTable D))
    :goalCondition (and (on D C))
    :action moveToTable A B)
(:rule PopBadTower2 (A B C D E)
    :condition (and (on A B) (clear A) (on* A D) (on C D))
    :goalCondition (and (on C E) (not (on C D)))
    :action moveToTable A B)
(:rule PopBadTower3 (A B C D E)
    :condition (and (on A B) (clear A) (on* A D) (on C D))
    :goalCondition (and (on E D) (not (on C D)))
    :action moveToTable A B)
(:rule PushGoodTower1 (A B)
    :condition (and (onTable A) (clear A) (clear B))
    :goalCondition (and (on A B))
    :action moveFromTable A B)
(:rule PushGoodTower2 (A B C)
    :condition (and (on A C) (clear A) (clear B))
    :goalCondition (and (on A B))
    :action moveFromBlock A B)
```

In traversal problems connectivity can determine whether a particular traversal is reachable. For example, if a transporter has no path between a package's goal and its goal then it should not be used to service the delivery.

5.4.2 Shortest path

In practice, the concept of connectivity will often fail to provide any guidance for traversing in a graph. For example, if a traverser is to be moved between two locations then it makes sense to move the traverser in the direction of the other location. All of the locations that are in the same component as the target node will model the connected property. The property provides no way to make incremental progress between two points (de la Rosa and McIlraith, 2011), resulting in loops and failed execution.

The RBP requires guidance in expressing movement through a path of nodes. Shortest paths are simple to compute and provide the optimal path between two nodes. The shortest path between two nodes is a list of edges that connects two nodes and has least weight. In this work edges have unit weight and so the shortest path is the path with fewest edges.

Definition 5.4.2 *The shortest path between u and v is defined for a graph, $G = (V, E)$, as:*

$$\mathbf{path-length}(v_0, v_k) = \min_k(v_0, \ldots, v_k) \in E, \textit{ where } v_0 = u \;.\; v_k = v.$$

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the next step on the shortest path between pairs of nodes. For a particular state $s \in \text{wff}(\Sigma_i)$ and graph $G(s) = (V, E)$, we define the set of propositions:

$$\begin{aligned}
&\forall u, v \; (u, \ldots, v) \in E,\; l = \mathbf{path-length}(u, v),\\
&\quad v_0, \ldots, v_k \; (u, v_0, \ldots, v_k, v) \in E,\; s' \in \text{wff}(\Sigma_{i+1})\\
&\qquad (\mathbf{path-length}(v_0, v) + 1 = l \;.\; s'\mathbf{R}s) \implies s' \models (\text{shortestPath } u\; v\; v_0)
\end{aligned}$$

Use of shortest path

The most direct use of the shortest path propositions is moving a traverser through the graph in a traversal problem. In these problems the traversers have destinations and the planner should work to move the traversers towards these. The shortest path propositions, derived from a traverser's accessibility graph, indicate the best next step to take in order to move the traverser from its current location towards its goal. It can be used to define a single rule policy that solves any such traversal problem.

```
(:rule MoveToDestination (Traver Loc1 Loc2 Loc3)
    :condition (and (at Traver Loc1) (shortestPath Loc1 Loc3 Loc2))
    :goalCondition (and (at Traver Loc3))
    :action move Traver Loc1 Loc2)
```

Of course, this is the most basic traversal problem in a wide variety of problems that involve moving traversers to specific locations. For example, in transportation problems a traverser picks up and drops off various packages during a single problem. A traverser might have several target locations to choose from at one time and this approach will not be sufficient.

Guidance with multiple targets breaks down

Theorem 5.4.1 *The shortest path propositions are not sufficient to support control traversal through a graph in the context of multiple targets.*

Proof We present a counter example to demonstrate that the shortest path propositions are not sufficient. The problem is illustrated in Figure 5.4: there is a traverser, `Trav` and four locations, `LocA, LocB, LocC, LocD`. `Trav` starts at `LocB`, which is connected to `LocC` and `LocA`. The final edge connects `LocA` and `LocD`. The problem involves moving to locations that have some property, *aProperty*, and performing an operation; but we simply focus on the movement aspect of the problem. As *aProperty* holds for locations `LocC` and `LocD` we have two targets that must be visited by `Trav`.

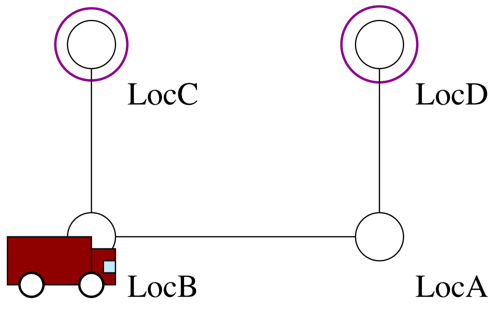


Figure 5.4: Traverser with two targets marked in purple.

We can use a similar rule to move traversers towards target locations.

```
(:rule MoveToDestination (?traver ?loc1 ?loc2 ?loc3)
    :condition (and (at ?traver ?loc1) (aProperty) ?loc3)
                    (shortestPath ?loc1 ?loc3 ?loc2))
    :action move ?traver ?loc1 ?loc2)
```

We observed in Section 4.2 that several bindings can satisfy a rule in the same state. For example, this rule is satisfied by the bindings: (*Trav,LocB,LocC,LocC*) and (*Trav,LocB,LocD,LocA*), giving rise to the set of actions: {(move *Trav LocB LocC*), (move *Trav LocB LocA*)}. If we assume that we sort actions alphabetically and select the first action then we select (move *Trav LocB LocA*).

In the next state *Trav* is at *LocA*. The rule is satisfied by the bindings: (*Trav,LocA,LocD,LocD*) and (*Trav,LocA,LocC,LocB*), giving rise to the set of actions: {(move *Trav LocA LocD*), (move *Trav LocA LocB*)}. Once sorted the first action is (move *Trav LocA LocB*) that loops back to the original state. As this is a policy computation we are trapped in the loop and execution never ends.

Traversal commitment A chain of move actions can be seen as committing to moving to a specific destination and then carrying out the series of moves required to reach the location. Through making the commitment up front it can be used to orchestrate a sequence of policy applications so that each step contributes to a single path to a chosen target. As this approach makes explicit a specific target, the shortest path propositions can be exploited to move the traverser to the target.

$$\begin{aligned}\forall u\ (\textit{aProperty}\ u) &\iff \\ &\mathbb{M}|_{\Sigma_{i+1}} \models (\text{setTarget}\ t\ u)\end{aligned}$$

The effect of applying the action $(\text{setTarget}\ t\ u)$ is to set a proposition $(\text{target}\ t\ u)$ in the next state.

As we have noted the ordering explicit in the policy map representation might result in a change in focus. As a result we require a mirroring set of actions that remove the target from the state.

$$\begin{aligned}\forall u\ (\text{target}\ t\ u) &\iff \\ &\mathbb{M}|_{\Sigma_{i+1}} \models (\text{removeTarget}\ t\ u)\end{aligned}$$

These actions provide the option to make the decision of the destination of a multi-step move action before the first step is made.

This provides a general approach for several of the problems that exist over graph structures in planning problems. For example, the selection of the next package for a truck to pickup is made explicit and through this commitment the truck can make progress towards a single package.

Macro actions An alternative is to abstract the graph completely by providing *macro* move operations between all pairs of nodes in the graph, with corresponding linking predicates. A single action would then move the traverser to its target. This has the advantage that the selection of the target node is wrapped up in the decision to move to a particular location.

We can define a step from Σ_i to Σ_{i+1} with a set of actions. These action model macro moves between all connected pairs of nodes. For a particular state $s \in \text{wff}(\Sigma_i)$, traverser, t, and associated graph $G_s(t) = (V, E)$, we define the set of action:

$$\begin{aligned} &\forall u, v\ (u, \ldots, v) \in E \iff \\ &\qquad \mathbb{M}|_{\Sigma_{i+1}} \models (\texttt{long-move}\ t\ u\ v) \end{aligned}$$

The connection graph for traversal actions will support the planner making graph traversals in static graphs and parts of dynamic graphs.

5.4.3 Constrained traversal: identifying the relevant closed nodes

In this subsection we look at the turn based graphs that we introduced in Section 5.2. We have demonstrated how the states of the problem model can be enriched with connected predicates that model whether a mover can traverse to a particular node. This can be used to find the current connected component. Moreover, it can define the frontier of blocked neighbours of the current component. The problem is that an antecedent cannot determine the subset of this frontier that exist on a good path to the goal. In particular, if the mover is currently disconnected from an area of the map that it must visit then a strategy is required to open a series of nodes that will establish a path between the mover and its goal.

First blocked node

In several of the benchmark domains, the mover is required to be adjacent to open a node. This is because the maps in problems often represent spatial relationships, where blockages are often freed by the traverser. For example, to open a door you need to be beside it and to unblock a pipe you need to be near the pipe. Of course, there are many possible exceptions such as terminal operated doors. However, in many of these problems the nodes can be unlocked from the mover's current component towards the goal location and this is sufficient for the benchmark domains. We focus on the closed nodes that surround the mover's current connected component.

Definition 5.4.3 *The first blocked function is defined as a map from a function from locations to the Booleans, a graph and two locations, to a set of locations:*

$$\mathbf{FirstBlocked} : \ (\mathbf{G} \times (\mathbb{L} \mapsto \mathbb{B}) \times \mathbb{L} \times \mathbb{L}) \mapsto 2^{\mathbb{L}}.$$

For a path opening problem, $(SG(t), o_s)$, for a traverser, t, and state, s, the locations in the set, $\mathbf{FirstBlocked}(SG(t), o_s, l_0, l_1)$, indicate the locations that should be opened to move t between l_0 and l_1.

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the first blocked nodes of a graph. For a particular state, $s \in \text{wff}(\Sigma_i)$, the associated set of static graphs, SG, and open nodes function, o_s, we define the set of propositions:

$$\begin{aligned}
&\forall t \in \mathbb{T}, l_1 \in \mathbb{L} \quad (l_0 = \mathbf{position}(t) \implies \\
&\quad \forall blocked \in \mathbf{FirstBlocked}(SG(t), o_s, l_0, l_1) \\
&\qquad (\forall s' \in \text{wff}(\Sigma_{i+1}) \\
&\qquad\quad s'\mathbf{R}s \implies s' \models (\text{firstBlocked } t \; l_0 \; l_1 \; blocked)))
\end{aligned}$$

Framework for exploiting metrics The computation of the *first blocked* mapping is not trivial. In this general setting there is no information regarding the cost of opening nodes; it is never easy to define what makes a good and bad plan and therefore which nodes should be accepted as *first blocked* nodes. In place of a definite strategy, we will present a definition of first blocked locations that relies on a metric. The metric is a map from vertex sequences to the integers. We define the set of all vertex sequences:

$$O^{\mathbb{L}} = \{v_0, \ldots | \forall i \; v_i \in \mathbb{L} \; . \; (\forall i, j \; (i \neq j) \implies v_i \neq v_j)\}.$$

Therefore the metric is a mapping, $\mathbf{metric} : O^{\mathbb{L}} \mapsto \mathbb{I}$.

We can define a set of sequences of nodes, *tsequences*, that form a path between l_0 and l_1 and are optimal with respect to the metric function:

$$\begin{aligned}
&\mathbf{tsequences}(G, o_s, l_0, l_1) = \\
&\quad \{v_0, \ldots, v_n | \\
&\qquad \forall u_0, \ldots u_k \\
&\qquad\quad \mathbf{metric}(G, o_s, (l_0, v_0, \ldots, v_n, l_1)) \leq \mathbf{metric}(G, o_s, (l_0, u_0, \ldots, u_k, l_1))\}.
\end{aligned}$$

An implementation of the first blocked function, originally defined in Definition 5.4.3, specifies the closed nodes on a sequence of nodes in *tsequences* that are closest to the mover's current location.

Definition 5.4.4

$$\mathbf{FirstBlocked}(G, o_s, l_0, l_1) =$$
$$\{v_i | v_0, \ldots, v_n \in \mathbf{tsequences}(G, o_s, l_0, l_1)\ .\ \min_i(o_s(v_i) = \textit{false})\}$$

First blocked metrics Two general metrics are defined here. The first is the *underlying path* metric that selects the sequences with the fewest steps in the underlying static (potential) graph. This heuristic is fast to compute and is often effective as sequences are often similarly blocked. However, it can perform badly if the shortest path in the underlying graph is disproportionately blocked.

Definition 5.4.5 $\mathbf{UnderlyingPath}(G, o_s, (v_0, \ldots, v_n)) = \begin{cases} n & \textit{if } v_0, \ldots, v_n \in G \\ \textit{undefined}, & \textit{otherwise} \end{cases}$

A second metric is to select the path with the fewest blockages on it. This approach works well when opening a blocked node requires substantial effort. This heuristic is slower to compute. Of course this heuristic will favour very long paths if they have the least number of blockages.

Definition 5.4.6 $\mathbf{FewestOpenings}(G, o_s, (v_0, \ldots, v_n)) = \begin{cases} |\{v_i | o_s(v_i) = \textit{false}\}| & \textit{if } v_0, \ldots, v_n \in G \\ \textit{undefined}, & \textit{otherwise} \end{cases}$

In Section 7.2, we present a framework that will provide a more general approach to tackling these problems, although at the cost of computation time.

5.4.4 Constrained traversal: interaction between traversers

There are problems where the traversal of objects cannot be considered independently but where important SIs involve the coordination and movement of a collection of traversing objects. The problems combine the movement aspect of traversal problems with the ordering aspect from block stacking problems. Investigations into landmarks (Hoffmann et al., 2004) and goal orderings (Koehler, 1998) generalise the identification of suitable orderings for achieving goals in stacking problems. However, the general problem of ordering sub-goals is as difficult as the planning problem (Koehler, 1998) so a complete solution is infeasible.

Problems with interacting traversers

The $(n^2 - 1)$-Puzzle is a problem that involves an $n \times n$ grid, with $(n^2 - 1)$ tiles and one blank square. A tile can be moved into the blank square, leaving a new blank square behind. The goal is to rearrange the tiles to match a configuration presented in the goal. $(n^2 - 1)$-Puzzles can be framed as constraint satisfaction problems and one possible approach would be to wrap the problem in a solver and use a constraints solver. A scalable heuristic approach has been presented in Parberry (1995), where the problem is solved using a divide and conquer strategy. This presents an alternative to providing the solution to the planner; however, in order to support the planner in choosing the next step to be made several problem specific language enhancements would be necessary.

Sokoban is another grid based puzzle game that involves moving blocks into goal positions by moving a man into an adjacent square and pushing the block from behind. Some of the grid locations are closed, forming rooms and corridors. This problem requires careful ordering and selection of moves as the blocks cannot co-locate with the man or other blocks, making dead ends possible. As with the $(n^2 - 1)$-Puzzle, researchers have sought effective solutions to Sokoban problems in isolation. In Botea et al. (2003) a decomposition approach is developed that first abstracts the grid into corridors and rooms and then uses this level of abstraction to plan in a greatly reduced space. This is an attractive approach as it is modular and an RBP would be an appropriate planner for solving the resulting abstracted problem. However, as with the approach for the (n^2-1)-Puzzle, we would need to develop specific specialised solvers for this domain. This highlights the important challenge in this work: the aim is to empower the planner, while enhancing the model with vocabulary that is inexpensive to model and can be introduced in a general way. In order to achieve directed connectivity in these domains, the shared connectivity of the traversers must be considered. We present a general model for modelling directed connectivity in Section 7.2.

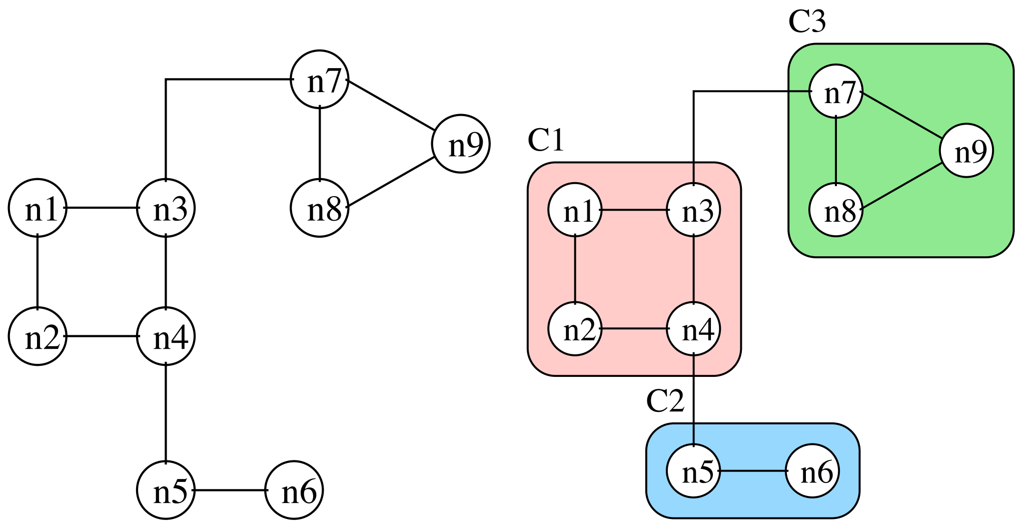
5.5 Optimisation

In Section 5.4, we have developed a collection of chain steps that establish directed connectivity in several problems. These problems commonly involve an optimisation problem that requires these behaviours. For example, a transportation problem requires sequences of move actions between locations; to select these effectively requires the careful allocation of packages to trucks. This means that the planner must choose amongst a number of alternatives that each appear equivalent. In this section we identify chains that provide information that distinguishes between different options. There are two main alternatives that we explore for this problem: to provide the planner with the information so that it can make the choice; and to take the actions selected by the planner and use some method of selecting between them. The aim is to support the planner in making local choices that contribute towards an effective global solution.

5.5.1 Graph relations

As the problems that we are considering are structure based, the optimisation problems that act over those structures are effected by the relationships between the active nodes of the structure; that is, the nodes with objects attached. General graph properties provide global properties of the nodes that can be exploited to improve the solution to a problem. Communication network vulnerability has been analysed using the concept of graph connectivity (Dekker and Colbert, 2004). Introducing a notion of weak nodes/edges in the graph (or high-value nodes/edges from the attacker's point of view) would provide an important information source for a problem that involved constructing a robust communications network. Similarly, centrality provides a valuation of a node's dominance over communication in the network (Freeman, 1977) that can be used to select nodes for sharing resources. More generally, clustering of a graph can be used to break the problem into the smaller parts and tasks can be completed within each cluster. Providing these to the planner provides a new source of distinction between alternative actions.

The potential in exploiting symmetries has also been investigated in planning (Fox and Long, 1999). For example, object symmetries (groups of objects with the same relationships in the initial state and goal) were identified and exploited in STAN (Fox and Long, 1999). The approach used in STAN for symmetry exploitation would translate to exploiting them in the rule matching part of our system and not the solvers. We discuss the implications of not using symmetry detection in our evaluation in Section 6.5. Although we have exploited symmetry in the computation of some of our solvers (for example, the static graph traversal and centrality solvers), we have not investigated providing the information for the planner's use. We would not expect that access to object symmetries would support the planner to find better quality plans.



(a) The graph (b) The clustered graph

Figure 5.5: An example of clustering

Clustering

Divide and conquer is a general approach to algorithm design that involves breaking a problem into a collection of similar problems that can be solved more easily. Through breaking a graph into separate components an algorithm that worked over the whole graph can be used to solve the same problem on a component of the graph, therefore breaking the problem into smaller parts. Of course there may be aspects of the problem that cannot be completely decomposed into components. However, spatial locatedness (with respect to the graph) can provide some indication of the effort required to move between different areas. Therefore we can assume that focussing on the problem within one area before moving on to the next area, will prove an effective strategy in some problems. In addition, the graph components can be used as a general indication of closeness between two nodes.

The partitioning of a graph's nodes into components can be provided by clustering. A graph clustering problem, for a graph, $G(s) = (V, E)$, is a partitioning of the graph into subsets: $C_0, \ldots, C_k$. The intention is that vertices in a particular cluster, C_i, are well connected. For example, the locations in a map can be clustered by locations that are spatially close, or perhaps well connected by road. An example graph and a possible clustering of its nodes are illustrated in Figure 5.5.

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the clusters. For a particular state $s \in \text{wff}(\Sigma_i)$ and graph $G(s) = (V, E)$, with a partitioning of the graph, $C_0, \ldots, C_k$, we define the set of propositions:

$$\begin{aligned}
&\forall u, v\ (\exists C_j\ u \in C_j\ .\ v \in C_j) \\
&\quad (\forall s' \in \text{wff}(\Sigma_{i+1}) \\
&\qquad s'\mathbf{R}s \implies s' \models (\text{sameCluster}\ u\ v))
\end{aligned}$$

We have chosen to link every pair of objects that are in the same cluster together with a binary predicate, `sameCluster`. This allows us to express nodes in a cluster in a single proposition. For example, if we move from *u* to *v* in a traversal problem, the predicate (`sameCluster` *u v*) in a graph traversal action will limit the move to the nodes in the cluster. Another approach would be to add an object for each cluster and then use a binary predicate to link each object with its cluster object. This requires fewer facts to represent and singleton clusters can be represented. However, it involves adding extra objects into the problem and discovering whether two locations are in the same cluster requires two predicates rather than one. Adding new types and objects is out with the scope of our architecture and modelling single ideas in one predicate has benefits for learning rules (Chapter 8).

We can use a standard approach to clustering; switching to more specific techniques when the use of the graph is apparent.

Use of clusters We can demonstrate the use of clusters in the graph based transportation problem introduced above. In the control knowledge that we presented for this domain the picking up and dropping off of packages were largely separate processes. If the truck contained a package for its current location then it would remove it, however, the policy prioritised moving to locations where there were packages requiring collection.

In this example we demonstrate that we can enhance this behaviour, allowing the processes to be interleaved, focussing on picking up and then dropping off in smaller areas. The traverser's accessibilty graph can be clustered and the clusters can be used to concentrate the focus of the traverser to nearby locations. Once the tasks in the cluster have been serviced then the traverser will move on to a different cluster. An example policy is as follows:

1. Pickup misplaced package

2. Drop-off package at destination
3. Move to pickup package in current cluster
4. Move to drop-off package in current cluster
5. Move to pickup package (in another cluster)
6. Move to drop-off package (in another cluster)

Hierarchical clusters The benefits of breaking the problem into areas of the graph can be repeated by generating a hierarchy of clusterings. For example, the divide and conquer approach for transportation problems presented above can be generalised to several levels of abstraction. The limitations on the specialised solvers (cannot introduce objects) and rule system (stateless) mean that this hierarchy must be fixed size. For example, we could identify three levels of abstraction over the graph and a strategy would specify the level of abstraction for each rule. However, this approach would not support a general unfolding of a strategy over larger structures; but, it would support changing the strategy for different abstraction levels.

Centrality

The centrality of a node in a graph is an important factor for several types of problem. In particular, problems that involve sharing resources. For example, in a package routing problem the planner might attempt to reduce the load on central nodes that are likely to have heavy traffic during peak periods. In transportation problems central points are ideal for reallocating packages between trucks. The centrality might also indicate the power of influence; for example, in a social network a central figure might suggest a good person to know in order to gain influence (Freeman, 1977).

Graph centrality is an area of study in graph theory; however, more practical definitions of centrality have been made that examine a node's dominance in a network (Freeman, 1977). The goal is to identify nodes in a graph that have a notion of centrality with respect to the other nodes in the graph. There are certain graphs where the central nodes are obvious, for example in the hub of a wheel, illustrated in Figure 5.6(b). There are of course many graphs that do not suggest such clear examples of centrality. There are various interpretations of what it means to be central to a graph. For example a node could be central within a group of closely connected nodes, however, it is not particularly central to the graph as a whole.

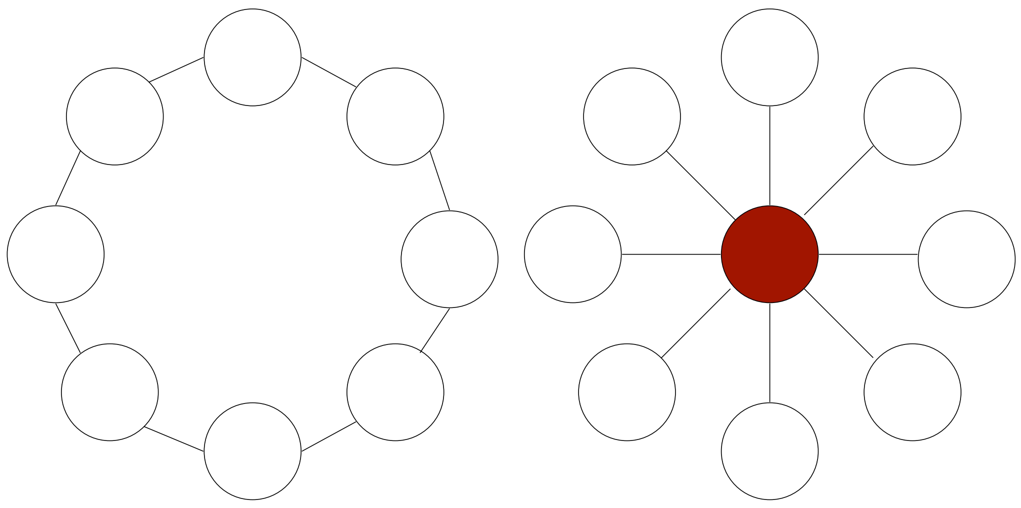

(a) No centre nodes (b) The red node is central to the graph

Figure 5.6: The concept of centrality in a graph

In the literature, it is common to give each node a score that represents its centrality in the graph. This is not appropriate for our rule language. We could retain some of the information by arranging the nodes as an ordered list, representing the list as a graph structure. However, as we have observed, the rule language cannot reason effectively with graph structures. We therefore represent the solution to the centrality problem of a graph, $G = (V, E)$, as a subgraph, $\mathbf{CentralNodes} \subseteq V$.

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the central nodes of a graph. For a particular state $s \in \text{wff}(\Sigma_i)$ and graph $G = (V, E)$, with central nodes, $\mathbf{CentralNodes}$, we define the set of propositions:

$$\begin{aligned}
&\forall u \in \mathbf{CentralNodes} \\
&\quad (\forall s' \in \text{wff}(\Sigma_{i+1}) \\
&\qquad s'\mathbf{R}s \implies s' \models (\text{hub } u))
\end{aligned}$$

Use of graph centrality We might consider using the central locations as hub nodes in a transportation problem. In transportation problems packages are often gathered at hubs so the packages can be redistributed for delivery. This requires rules to pickup the packages, move the trucks to central locations, redistribution of the packages amongst the trucks and then the movement of the trucks to the goal locations of the packages.

However, the central nodes cannot be used directly in this way. If the rule that moves the truck to the hub location is positioned above the rule that moves the truck to the package goal, then as soon as the truck moves away from the hub location the rule that moves it towards the hub will be applicable again. There would be no way of signaling that the stage has been completed.

A binary *sub-goal-at* predicate can be used to link the package with the next position that it should be placed at. The proposition is removed from the state when the package is picked up from that location. This allows a solver to describe a sequence of hub nodes between the package's initial state and its goal destination. This enables exploitation of both local hub locations and global hub locations, as would be expected in a large transportation problem.

1. if the package is not at its goal location then pickup the package.
2. if the truck is at the package's goal location then drop-off the package.
3. if the truck is at the package's current sub-goal location then drop-off the package.
4. if the package is not at its goal location then long move to the package's location.
5. if the truck contains a package and it has a sub-goal location then long move to the package's sub-goal.
6. if the truck contains a package and is not at the package's goal location then long move to the package's destination.
7. if the truck is not at its goal location then long move the truck there.

To make this vocabulary more effective we could inform the hub node selection with a process that can compute package flows through distribution centers. This could employ a sophisticated solver to analyse the package locations and compute an effective strategy. Approaches to hierarchical graph traversal algorithms, such as HPA*, and hierarchical abstraction approaches in planning (Gregory et al., 2011) have demonstrated that planning problems can often be solved quickly at higher levels of abstraction, but that the solutions can be informative for lower levels. The transportation aspect can be abstracted using clustering and a planner can be used to solve the abstracted problem. The sub-goal predicates could then be informed by the locations used for redistributing packages in the plan. This approach was implemented, however, the tested satisficing planners never redistributed the packages, leading to no hub nodes. Even when we tackled the problem's symmetry (which increases at the abstract level) and abstracted the graph as much as possible, the resulting problem could not be solved with an optimal planner in reasonable time. These extensions demonstrate how specialised solvers become increasingly sophisticated as improved performance is required in an isolated sub-problem. However, the main aim in this work is to provide support, rather than making sophisticated specialised solvers.

5.5.2 Local heuristics

An alternative approach to supporting the policy in making selections appropriate to the overall task, is to use heuristics to order the selections that are made by the rule system. In the description of co-execution, in Chapter 3, we use a policy to choose the next action. In practice, we observed in Chapter 4 that the RBP yields a set of rules and a deterministic selection must be made. In the following part of this section we take the view that these actions can be considered equivalent. We may have enhanced the problem model with a series of steps that provide distinguishing vocabulary (as described in the previous subsection), but at a certain point we have stopped and the planner cannot distinguish between two actions. This choice between actions provides an opportunity to exploit heuristic guidance. In the following subsections we move away from supporting the planner in making its choices and instead investigate selecting the best move from the alternatives it presents.

Nearest neighbour

The nearest neighbour heuristic has been exploited to great effect in the rules of TLPLAN and in the traversal solver in HybridSTAN. Given a set of alternatives the nearest neighbour heuristic favours the closest, for some metric. For example, given a collection of move actions for a traverser, this heuristic would pick the location that required fewest steps.

The shortest path solver can include in the state the distance between each pair of points on the graph of each traverser. For example, (`path-length` *?t ?from ?to ?c*), encoding the path length, *c*, for each traverser and location pair. Our current rule representation can only be parameterised from the problem objects and therefore cannot reason with these propositions directly. However, the solver can use this information to compare the alternative choices. In the case of the nearest neighbour, the path lengths of each of the alternative moves is identified and the shortest paths are selected. The communication with the solver is through the set of actions suggested by the planner.

These alternatives are controlled by the planner through action bindings. The solver filters those actions that are not nearest neighbours and presents the remainder as the action selection.

We can exploit the nearest neighbour in the implementation of the `long-move` action; this provides an alternative called the `nearest-long-move` action. Each action has the traverser and the from and to locations as parameters. The solver selects the actions with the fewest steps between the from and to locations. This heuristic selects between actions for different traversers.

Resource management

Resource management involves allocating resources to consumers or users. There are many ways in which a resource can be used: a resource might be able to serve a single consumer, for example, driving a truck; multiple consumers, for example, carrying packages; and use of a resource might destroy it, for example a bomb. The difficulty in making good resource allocation choices is that they depend on several factors; these include constraints, such as a door can only be unlocked by a particular type of key, and efficient use of resources.

Resource management is a general problem and is still an open area of research. A general characterisation and specialised treatment of resources in planning problems have been developed in (Dvorak and Barták, 2010). We have selected a specific situation where resource management is an important aspect of the problem. We demonstrate how the vocabulary can be enhanced with resource management decisions in the transportation problems illustrated in Long and Fox (2002). Given a suitable resource management solver, a similar approach could be adopted in other forms of resource management.

Resource allocation A resource management problem involves sets of resources, $\mathbb{R}$, and consumers, $\mathbb{C}$. A resource allocation can be expressed as a pair: $(r \in \mathbb{R}, c \in \mathbb{C})$. We define a function that maps from a consumer to the resource allocated to the consumer in the state, s: $\mathbf{allocation}_s : \mathbb{R} \mapsto (\mathbb{C} \cup \perp)$.

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the allocation of resources to consumers. For a particular state $s \in \text{wff}(\Sigma_i)$ and function $\mathbf{allocation}_s$, we define the set of propositions:

$$\begin{aligned}
&\forall c \in \mathbb{C}\ \exists r = \mathbf{allocation}_s(c) \iff \\
&\qquad (\forall s' \in \mathrm{wff}(\Sigma_{i+1}) \\
&\qquad\qquad s'\mathbf{R}s \implies s' \models (\text{allocated } r\ c))
\end{aligned}$$

In this work we have experimented with several approaches to allocation and the view that the solver provides of the allocations. We have tested this vocabulary for packages using trucks and for trucks using drivers, over static traversal graphs. As well as the allocated predicate presented above, we defined an alternative step that involved actions that requested an allocation for certain objects. The state then contained a proposition that encoded the allocation so that it was part of the rules' context. This approach has the disadvantage of requiring several rules, but allows the strategy more flexibility over when allocations are made and which consumers require resources.

The static graph means that the number of actions necessary to traverse between nodes is fixed and the resource allocation can be abstracted from the rest of the problem. This problem could be solved as an optimisation problem. In this work we use modifications of a closest resource allocation approach: applying it once from the initial state; and applying it dynamically.

5.5.3 Utilising global heuristics

The approaches above are specific to certain aspects of the problem. However, general purpose heuristics are now mature and effective for planning. Our framework supports exploiting these heuristics, in case the preceding approaches fail to distinguish between alternatives. Our approach to computing the heuristic is to use an estimate for distance to goal in $\mathbb{M}|_{\Sigma_0}$. Each binding of the fired rule leads to an action. We calculate the heuristic estimate for each of these actions and find the minimum value. A deterministic selection is made between the actions that have this minimum score. A benefit of this is that it allows us to harness available domain independent heuristics.

We approximate the distance to goal as the heuristic estimate of the state of $\mathbb{M}|_{\Sigma_0}$:

$$h(s) = h(s'|s\mathbf{R}s').$$

In our experiments we have used the implementation of h^{FF} in the planner, JavaFF.

An important property of this approach is that actions that have no effect in $\mathbb{M}|_{\Sigma_0}$, have no effect on the heuristic estimate. This makes sense as we are interested in reducing the number of steps in the described problem model. Of course the policy is still used to filter the action choices. This means that when search is sitting at a local minimum in the heuristic landscape then we will only enter a loop of making zero cost enriched actions if the policy maps to these actions.

In contrast to our approach, Coles and Smith (2006); Fox and Long (2001); Dornhege et al. (2009) use a heuristic estimate that incorporates estimates from the solvers. A key difference in these works is that the target plan is expressed in the enriched language. This means that taking into account the cost of the enriched actions is important for estimating the quality of the solution.

In this section we have presented three alternative approaches to supporting the planner in distinguishing between alternative options. The first achieves this through providing more information so that the planner can make the choice; while the latter two approaches take the set of possible choices from the planner and choose the best one, based on some metric. In the local approach the decision is made using a heuristic that is specific to the action being applied, while the global approach utilises a general domain-independent heuristic. We compare the performance of these three approaches in an experiment in Chapter 6.

5.6 Level of reasoning

In this section we examine how changing the level of reasoning supported by the modelled vocabulary effects its use with an RBP. Alternative levels of reasoning were the main focus in abstraction based planners (introduced in Section 2.2). In these approaches the layering was used to order the decisions; and selecting an appropriate layering of the model would lead to more efficient planning. In our work, the order of decision making is largely dictated by the forward chaining approach we adopt. All of the levels are presented to the RBP at once and it can select the most appropriate level for the current decision. An important difference, is that when an abstract action is selected in AbNLP (Fox and Long, 1995) or ABSTRIPS (Sacerdoti, 1974), the planner is involved in its refinement through the lower layers. In our approach, the planner delegates the refinement of abstract actions to the specialised solvers. As a consequence, the important consideration regarding the appropriate level of reasoning, is the level of control that the planner has over the instantiation of actions. We first examine chains that provide vocabulary that supports the planner with interpreting the different levels of the problem space. We examine the level of control that the planner can exert over the support provided by the solver. We then look at the level of interaction of the planner and how this effects the control of the planner.

5.6.1 Level of interpretation

Macro actions are a well studied approach that provides an alternative level of action selection. We have presented the long move macro actions that abstracts the underlying graph, performing a series of move actions as a single step. The planner can select the final destination of the move; however, it cannot control how the traverser is moved between the locations. The long move action provides a step at the level of interpreting the problem as a traversal problem. In this section we present two sets of abstract actions that interpret the problem at two higher levels. We examine the effect on the planner's control over the support.

Process level vocabulary

The semantic interpretation of a planning problem identifies various layers that are not expressed in the problem model. For example, an important process during a Blocksworld problem is to uncover a particular block, perhaps as a consequence of it being misplaced. In transportation problems there are similar processes. For example, there are two common processes required for each package: a truck must move to the package's location and then pick it up; and a truck that contains the package must be moved to the package's destination and the package removed. There may also be a reallocation of packages between the trucks that involves an arranged meeting at a central point. The problem model can be enhanced with actions that wrap up the actions that achieve these tasks. The planner can then reason about package delivery in terms of high level concepts. We have focussed on the transportation elements of collect, gather and drop-off.

Collect The purpose of a collect action is to move a truck to the destination of a package and pick it up. The collect action can be constructed using primitive actions from the transportation sub-problem. The collect action will produce a series of move actions that progressively move the truck closer to the package. Once the truck is at the same location as the package, it should then use a pickup action. The action selection can be defined as the recursive application of the following function, until the package is in the truck.

$$\textbf{collect}\ t(\mathbb{T})\ p(\mathbb{P}) \equiv \begin{cases} \text{put } p \text{ in } t \text{ at } l, & \text{if } t \text{ and } p \text{ are at } l \\ \text{move } t \text{ towards } p, & \text{if } p \text{ and } t \text{ are located.} \end{cases}$$

As was the case with the move actions, the number of actions applied by a single collect action will depend on the relationship between the position of the truck and package. This means that the action cannot be parameterised by the nodes that the truck will move through. Moreover we will not be able to guarantee the variables that will be used in the next action as there are two different types of action that could be being applied. We parameterise the collect action with the truck and package parameters. The movement towards the package relies on the long-move action defined previously. The long-move action is parameterised using the current location of the truck and package. The parameters for the pickup action can be bound with the truck and package objects; a unification of the other variables is selected arbitrarily. We can make this vocabulary more powerful by constraining the transporter and package pairs with the allocated propositions from the resource manager and by exploiting the nearest long move actions. The precondition is extended with the predicate (`allocated` *t p*) and the repeated action application becomes:

$$\textbf{collect}\ t(\mathbb{T})\ p(\mathbb{P}) = \begin{cases} \textbf{pickup}\ t\ p\ l, & (\texttt{at}\ t\ l)\ .\ (\texttt{at}\ p\ l) \\ \textbf{nearest-long-move}\ t\ l\ l', & (\texttt{at}\ t\ l)\ .\ (\texttt{at}\ p\ l') \\ \text{end}, & \text{if } p \text{ is in } t. \end{cases}$$

Deliver

We can use a similar approach to capture a delivery action. The truck must contain the package. The action will move the truck towards the destination of the package and then remove the package at its goal. This behaviour is captured in the following function:

$$\textbf{Deliver}\ t(\mathbb{T})\ p(\mathbb{P}) = \begin{cases} \textbf{drop-off}\ t\ p\ l, & (\texttt{at}\ t\ l)\ .\ (\text{goal}(\texttt{at}\ p\ l))\ .\ (\texttt{in}\ p\ t) \\ \textbf{nearest-long-move}\ t\ l\ l', & (\texttt{at}\ t\ l)\ .\ (\text{goal}(\texttt{at}\ p\ l'))\ .\ (\texttt{in}\ p\ t) \\ \text{end}, & \text{if } p \text{ is at its goal.} \end{cases}$$

Gather There is an optional gathering process in transportation problems where the packages are gathered at hub locations and allocated to different transporters. The gather action relies on the package sub-goal-at predicate that we defined in Subsection 5.5.1. If a transporter contains a package with a sub-goal then the gather action will move the truck to the hub location and drop off the package. This can be captured in the function:

$$\textbf{Gather}\ t(\mathbb{T})\ p(\mathbb{P}) = \begin{cases} \textbf{drop-off}\ t\ p\ l, & (\texttt{at}\ t\ l)\ .\ (\texttt{sub-goal-at}\ p\ l)\ .\ (\texttt{in}\ p\ t) \\ \textbf{nearest-long-move}\ t\ l\ l', & (\texttt{at}\ t\ l)\ .\ (\texttt{sub-goal-at}\ p\ l')\ .\ (\texttt{in}\ p\ t) \\ \text{end}, & (\texttt{at}\ p\ l)\ .\ (\texttt{sub-goal-at}\ p\ l) \end{cases}$$

Exploiting the raised level A solution to the transportation problem that exploits this vocabulary can be expressed in three rules:

1. Collect package;
2. Gather package;
3. Deliver package.

This example highlights the impact of abstracting the level of reasoning. The benefit is that we can express a solution to a hard combinatorial problem in three lines. However, we have sacrificed some of the control over how the solution is constructed.

A transportation action

In this subsection we consider raising the level of the vocabulary further. The highest level of interpretation of a problem is a single action that solves the problem. This action provides a direct solution to a hard problem. The computation of the vocabulary is, of course, as complex as solving the planning problem (which is what it is doing). In Blocksworld this action would unstack and re-stack blocks to match the goal state and in a transportation problem unrolling the action would involve a series of package deliveries that achieved all the goals. Of course, these problems can be embedded within a larger context. For example, the Depot domain combines both transportation and structure building sub-problems. Therefore it is interesting to consider this step and the control it provides to the policy when solving such a problem.

We use the example of a transportation step. The transportation step can solve a transportation problem with a single step and a single rule:

1. MakeTransportationStep.

This abstract action provides the option for the policy to solve the sub-problem or do something else. This presents several issues: the definition of the transportation problem; and the interactions between this problem and other parts of the problem. If the problem contains a perfectly isolated transportation problem, with the package destinations in the goal, and little interaction with other parts of the system then this action might be a useful support. However, consider a path opening problem that included doors that were locked by $(n^2 - 1)$-Puzzles. Although the $(n^2 - 1)$-Puzzle can be solved in complete isolation, the requirement of the door being open is a result of the larger context and would not be directly recognisable in the context of a (n^2-1)-Puzzle solver. From the path opening perspective, the $(n^2 - 1)$-Puzzle interferes with the solver's solution to the problem and as a result the solver would fail to model the vocabulary; or solve the $(n^2 - 1)$-Puzzle in a general and inefficient manner.

One of the advantages of the single transportation action is that we can exploit special solving technologies without the solution being skewed by the ordering requirements of the policy representation. On the rare occasion that the problem can be effectively decomposed into isolated sub-problems this form of high-level action provides an efficient method of combining solutions for several solvers. However, as the level of an action is made more abstract, the RBP is given less control over the specific actions that make up the solution. In cases where the sub-problems cannot be decomposed, a lower-level vocabulary provides a more appropriate context that supports the RBP in selecting the next action.

5.6.2 Level of planner interaction

Macro actions provide a raised level of reasoning that can allow the planner to make decisions at an appropriate level. The problem with macro actions is that the policy is not applied at the intermediate nodes in the path. One of the benefits of the RBP is that because it is stateless, it has no previous aims and as a result it can react to opportunities. In some cases, a useful action could be applied at an intermediate state that will be visited during a macro application. As macro actions are applied in a single step this opportunity is lost. For example, the current state in a transportation problem might bind with a rule for moving a truck towards a misplaced package. As the macro action is applied the truck is located at a set of locations. If there is a package in the truck that has the goal of one of these positions then it would make sense for the truck to deliver the package before moving. In particular, a rule that delivers the package might be higher priority than the rule that was moving the truck towards the pickup location. However, the sequence of actions is applied without further use of the policy and the truck must return to the location to deliver the package. We have shown in this section two steps that provide the planner with even less control. In this subsection we present an alternative approach to making these chain steps that provides the raised level for action selection, while maintaining the reactive properties of the rule system.

Step by step macro application

The step by step application approach (SbS) is an alternative interpretation of a macro action. When an abstract action is mapped to by the policy, the action is replaced by the first action in the abstract action sequence. This effects a single move in the direction indicated by the abstract action. The policy can then be reapplied at all of the intermediate nodes, typically regenerating the abstract action to carry along the same path, but being given the opportunity to exploit an opportunity when it arises. This builds on our observation that it is useful to reason about moving to targets, but weakens our previous approach so that the planner does not commit to actually reaching the target. Instead the planner makes a loose commitment to complete a task and then makes steps towards completing the task. However, if a higher priority rule becomes applicable then we might not return to complete the task. This incremental approach to abstract action application provides the benefit of the raised level for reasoning provided by the abstract actions, while allowing the priority of the rule system to dictate the decision made at every state of the plan.

In Subsection 5.4.2 we presented vocabulary for selecting the target node in a graph traversal. The SbS provides a general approach for selecting the desired target and it reduces the number of bookkeeping actions necessary. The intention behind several applications of compatible macro actions is interpreted as a request to continue unrolling the macro action and the specific target itself remains implicit. On the first application an action sequence is identified and the first of that sequence is applied. Subsequent applications of the abstract action continue applying the actions from this sequence. The selection of a different action breaks the weak commitment and the approach will be restarted.

A specific chain step that can be made with this application approach is to combine the abstract graph traversal action with the nearest-neighbour heuristic described in 5.5. The heuristic selects an abstract action that corresponds to the shortest path on the underlying graph. As a step is made in the direction of the target then even if there were more than one action with equal distance then this will not be the case for the second application. We call these actions nearest long move actions. In Subsection 4.2.1, we presented a policy for a simple transportation problem. This can be extended to problems where deliveries are made over graph structures. We can use the nearest long move actions to capture a policy for this domain.

1. if the package is not at its goal location then pickup the package.
2. if the truck is at the package's goal location then drop off the package.
3. if the package is not at its goal location then nearest long move to the package's location.
4. if the truck contains a package and is not at the package's goal location then nearest long move to the package's destination.
5. if the truck is not at its goal location then nearest long move the truck there.

The use of the SbS with this policy means that each long move is unrolled step by step. For example, if a truck is at a location with no package tasks, then it will be moved to pickup a misplaced package (if one exists). A step will be made in the direction of the nearest misplaced package and the policy mapping will be computed for the new state. If the truck contains a package and passes its goal then the package will be dropped off, as that rule is higher priority than the moving rule.

The weak commitment implicit in the SbS effects the way that some of the vocabulary defined in this chapter can be used. For example, traversing to a node within a cluster cannot be handled with the shortest-path vocabulary. Appendix C describes these issues in more detail.

In this section we have explored several chain steps that provide differing levels of control to the planner. The SbS is a method that places some control back with the planner. A key aspect, is that the control knowledge captured by the policy can be exploited at each plan state. The SbS combines the raised level, provided through macro application, with the reactive aspects of the control knowledge representation.

5.7 Conclusion

In Chapter 3 a general framework was defined that can be used to support a decision making planner through search. The specific planning approach used in this work and its main properties were presented in Chapter 4. We have identified a collection of problems that are not currently dealt with in the literature and explored the space of chain steps that address these problems. In this chapter we have explored the space in three dimensions, which span directed connectivity, optimisation, and level of reasoning. We have identified concrete chain steps that are relevant to the planner. In Chapter 6 we will conduct several experiments to analyse the impact of the approach. However, at this stage there are several observations that we can make with respect to enhancing the models of problems. The main contributions of this chapter are summarised here.

- Arbitrary length chain of move actions. This will be generalised in Chapter 7;
- An associated derived predicate: enables reasoning through an arbitrary sequence of move actions;
- Decision predicates: making choices explicit up front so that simple rules can act effectively within optimisation problems;
- A framework that provides two levels of heuristic guidance for supporting a rule-system in action selection;
- SbSs (Subsection 5.4.2): that combine selecting a target with making a single move in that direction. They combine the efficient priority based decision process of generalised policies with the benefit of the raised reasoning level of macro actions.

During the project, we investigated using chains with set and unset actions as this seemed to be an intuitive model for controlling commitment predicates. For example, we modelled an `allocate-resource` action, which made an allocation for the specified consumer and a proposition that represented the allocation being added to the enriched state. However, when we progressed on to learning the policies, we found that the rule learners struggled to discover sets of rules that exhibit the coordination necessary to exploit the vocabulary. In certain realisations of these concepts, the propositions change implicitly between states; therefore making explicit propositions in the state requires careful management in the solvers. An alternative is to model the vocabulary using a set of derived predicates and evaluate them as needed. This approach led to a simplification in the implementation of the solvers.

The raised language of macro actions is appropriate and effective for making decisions over arbitrary sized structures. There are situations where it can be guaranteed that if the first step of a macro action is made then the subsequent steps should be made as well, such as tunnel macro actions (Coles and Smith, 2008). However, in general planning at this abstract level impacts on the rule system's control over the plan and can result in less efficient plans. We have presented the SbS as an alternative that does not force the planner to fully commit to an action sequence. We have combined the benefits of the lifted level of reasoning, while continuing to exploit the benefits of a reactive planning approach.

Our system supports a three step approach for binding the rules of an RBP: first the rule system is queried and identifies a set of actions that are indistinguishable based on the control knowledge; second, a local heuristic specific to the solver is used to narrow down the choice; and third, a global heuristic is used to select the best alternative for the problem as a whole. The ideal situation would be that the model expressed a view that allowed the rules to distinguish more carefully between cases.

The chain of language restriction steps explored in this chapter have helped us to develop effective strategies for several types of domain and this will be demonstrated in Chapter 6.

CHAPTER 6

RESULTS CONCERNING THE ENHANCED LANGUAGE

In this chapter we investigate the framework that we have developed and the support provided by the chains presented in Chapter 5. The question that we seek to answer is:

> Does the enhanced model that we have developed in the previous chapters provide the necessary vocabulary so that RBPs expressed in a limited language can control search through problems with SIs?

We begin by exploring each type of chain step to analyse those most appropriate for enhancing the domain model. We assess the use of directed connectivity, optimisation, heuristics and the improvement in quality provided by the SbS. This analysis feeds into an analysis of the appropriateness of the vocabulary, with respect to how generally it can be applied and, in the appendices we explore the quality of the generated plans (Section E.1), and the effectiveness of combining words from different solvers (Section E.2). In Appendix E.3 we compare our control knowledge directly with the control knowledge used with TLPLAN. In this chapter the rule systems have been defined by hand and only provide an indication of the potential quality of a policy given the language. We begin with an overview of the setup, the problem sets, the planners and the solvers used in the following investigation.

6.1 Investigations in our framework

Our main strategy for analysing our framework is to use it in a variety of planning situations. In particular, we use our framework to solve problems from the standard benchmark problem sets and compare the number of problems solved, the time taken to generate these solutions and the quality of the solutions with the results for other approaches. In this section we detail the setup used for our empirical analyses.

6.1.1 Experimental setup

Our experiments are performed on an Intel Core i5-2500 CPU, clock speed: 3.30GHz. The CPU time was capped at 30 minutes in line with common practice in the IPC. In addition, the maximum memory usage was set at 3Gb.

The quality and time results are plotted on line graphs or presented in a table. For the graphs, a line is plotted for each planner or policy. This indicates its score in terms of quality or time. Quality is reported using the number of steps required to solve the problem. The time graphs plot the length of time that was required to solve the problem. These lines are plotted on logarithmic-scaled graphs.

We continue in this subsection by introducing the planners, the specialised solvers and the problems that we use in our experiments. The discussion on the functionality of the solvers was presented in Chapter 5.

Planners

The planners used in this section include several configurations of our system as well as domain independent planners. We use our architecture to execute the policies. Each policy is used to generate a set of actions and a selection strategy then identifies a single action that is applied to the state. If the set is empty then a backup strategy is used to propose an action. The policies can be found in Appendices D.1, D.2 and D.3. We compare our approach to two domain independent planners, LAMA (Richter et al., 2008) and FF (Hoffmann and Nebel, 2001). These planners are considered in the state of the art and have performed well over the years in the IPCs.

We present the general properties of the planners, such as the version and/or configuration used here. More specific details will be presented with the appropriate part of the investigation.

Lama The version of Lama that won the 2008 IPC (Fern et al., 2008). We configured the planner to return the first solution. This is interesting because we are comparing to policies that direct the executive through the search space with little search.

FF The planner MetricFF. This planner is based on an efficient implementation of the FF planning system.

JavaFF A Java implementation of FF created as a teaching tool (Coles et al., 2008). We have developed part of our system in the JavaFF framework and it is informative to plot its performance.

RBP We have implemented the framework presented in Chapter 5, which allows an enhanced problem model to be defined and used in planning. We have used the JavaFF (Coles et al., 2008) code base, with new search options for applying policies in search. An extension that connects TIM (Fox and Long, 1998) was implemented using the JNI API[1]. The implementation of our architecture uses direct links between a solver and the delegates that are required in its computation. For example, the resource management solver requests evaluations of `path-length` predicates from the graph abstraction solver directly, rather than these being enumerated in the state. The domain analysis required to identify dependent types can use the existence of solvers as part of the criteria for determining a generic type. For example, the existence of a resource management aspect to the problem is predicated on the existence of a traversal generic type. The relationship between the solvers is therefore explicit and exploited. This aspect of the work is further explained in Section 7.1.

The inputs to this system are a settings file, which provides the location of the enhanced domain model, the problem listing and an RBP, as well as the parameters for the planner. The default configuration does not use a heuristic. Testing whether a rule is applicable to a state is a difficult problem in itself. The positive rule conditions, for both goal and state, are bound for each matching predicate in the goal or state respectively. This generates the possible sets for some variables. During this process we select the predicate with the fewest unseen variables. We found that this is more effective than statically determining the predicate order based on example states. The selection strategy between the set of actions mapped to by the RBP is to sort the action strings alphanumerically and select the first action. If the policy is undefined then the backup strategy gathers all applicable actions and uses the same selection strategy. The

[1]http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

intention is that the free-search can move the search to a state that has a mapping in the policy. We use cycle-checking, which prevents search from revisiting states. Any alterations to these settings used are described in the appropriate sections.

Specialised solvers

In Chapter 5 we have presented various chain steps. Our policies have been expressed using the vocabulary in these enhanced models. We describe the specialised solvers in our experiments.

Graph abstraction The graph abstraction solver is used to abstract over chains of move traverser actions. For each graph, the solver enhances the problem model in two ways. A connected predicate is added using the naming convention (*moveActionName*_`connected` *?t* - $\mathbb{T}$ *?from-loc* - $\mathbb{L}$ *?to-loc* - $\mathbb{L}$). An instantiation of this predicate holds for any traverser and locations where the traverser can move between the two locations. The predicate is implemented as a derived predicate. The dynamic solver is limited to model this proposition where *?from-loc* is the traverser's current location. An operator is added using the naming convention (*moveActionName*_`move` *?t* - $\mathbb{T}$ *?from-loc* - $\mathbb{L}$ *?to-loc* - $\mathbb{L}$). This action is applicable for parameters if the connected predicate holds for the parameters and the traverser is at location, *?from-to*. We have implemented an extension to the Floyd-Warshall's all pairs shortest path algorithm (Floyd, 1962), which computes the shortest paths between nodes. If the graph is static then the edges are computed by evaluating the static formula in the initial state of the problem. This is not a reachability analysis, as it analyses each potential edge. As a result the solver can deal with traverser's leaving the graph and returning in a different place, as discussed in Long and Fox (2002). This is computed once before search. For dynamic graphs a reachability expansion is made using the traversal action. In turn based graphs, this is recomputed whenever an opening action has been applied since the last invocation of the language, whereas for other dynamic graphs it is refreshed at each state. This distinction had little impact on planning time in practice.

This solver implements the following optional functionality:

- Nearest blocked location: A predicate of the form (`nearest-blocked` *?from-loc* - $\mathbb{L}$ *?to-loc* - $\mathbb{L}$ *?nearestBlocked* - $\mathbb{L}$) that holds for the nearest blocked location between two locations. The static predicates are identified from the move action operator (as described in Fox and Long (2001)). The path between *?from-loc* and *?to-loc* is stepped through and the first blocked location is identified.

- Clusters: A predicate of the form (`ClusteredPair_GraphAbstractionID` *?loc*1 - $\mathbb{L}$ *?loc*2 - $\mathbb{L}$) that holds of pairs of locations in the same cluster. An action is modelled, (`NavigateCluster_GraphAbstractionID` *?t* - $\mathbb{T}$ *?from-loc* - $\mathbb{L}$ *?to-loc* - $\mathbb{L}$) that moves the traverser towards *?to-loc* on a path through the cluster, as described in Appendix C. This action is only applicable for locations that are in the same cluster. The clustering approach builds hierarchical layers by selecting map nodes and abstracting them as a single node in the next layer (as described in Gregory et al. (2011)).

- Hubs: A predicate of the form (`nextSubtask_solverCount` *?consumer ?resource ?node*) that holds for the next drop off position for the consumer with the resource. The betweenness of a node is a count of the number of shortest paths that the node is on (Bavelas, 1950). We select a single node with the highest betweenness score and use it as the hub.

Resource management The resource management solver makes resource allocations and provides the results in a derived predicate

(`Bound_GraphAbstractionInterfaceID` *?c* - $\mathbb{C}$ *?r* - $\mathbb{R}$).

The predicate holds if the resource has been allocated to the consumer. The implemented solver selects the nearest traverser for a given consumer (either package in transportation, or truck in for driven traverser problems). The location of the traversers are identified using the locatedness predicate in the state. Portables can be connected to locations and transporters through a chain of static propositions, as is the case in Depots and Gripper.

`Well-placed` predicate This solver controls the functionality of the `well-placed` predicate. The naming convention for the predicate is

(`well_placed_stackedPredicateName` *?b* - $\mathbb{B}$).

The predicate holds for a block if it is *well-placed.*

Located key door selector This solver selects the next node in the graph that should be opened. The result in this computation is provided in a predicate

(`doorToOpen_SolverID` *?loc* - $\mathbb{L}$).

The relaxed model is created and a relaxed plan is generated from the initial state. We use the problem goal in this experiment, though the vocabulary could be generalised, to allow a sub-goal to be used to control the computation of the vocabulary. The solver identifies the next door that is opened in the relaxed plan and uses this to evaluate the predicate. The solver analyses the actions that have been applied since its last application and if these are relevant (for example, the pickup action) then the solver compares them to the relaxed plan. If at some point the actions are not consistent then a new relaxed plan is generated.

Not equal The not equal predicate is a solver that provides functionality for a single predicate. The naming convention is $(!= \ ?o1 \text{ - } \mathbb{O} \ ?o2 \text{ - } \mathbb{O})$. The predicate holds for two objects that are not the same.

Problem sets

We have selected a collection of the domains from previous planning competitions that have interesting structures. Where possible we have used problem sets from the IPCs. However, in some cases, there were not enough problems, or problems of big enough sizes. In these cases we have attempted to use the competition generators. There are two cases where we have adapted the generator. The domains are described in Appendix A.

Blocksworld We have generated a larger range of problem using the generator from the 2000 IPC (Bacchus, 2000). These problems range from 10 to 90 blocks problems in steps of 10 and 5 problems of size where generated.

Driverlog The 20 problems generated for the 2002 IPC (Long and Fox, 2003).

Goldminer The 30 target problems generated for the first learning track of the IPC (Fern et al., 2008). We have adapted the domain to remove implicit preconditions and so that the robot is an object. For a given problem, the applicable actions are equivalent in any reachable state in these models.

Logistics The 28 problems from track 1 of the second planning competition (Bacchus, 2000). Problem 18 is unsolvable.

Gripper A selection of generated problems from 300 to 775 balls. All of the balls start in one room and must be moved to the other room.

Grid A set of 20 problems created using an updated version of the competition generator. The generator was altered to produce typed problems and problems that require the robot to be at a specific position (rather than the keys). This changes these problems from transportation problems to traversal problems.

Depots The 22 problems generated for the 2002 IPC (Long and Fox, 2003).

6.2 Directed connectivity

It has been observed that certain concepts of structures cannot be expressed in our rule language and other languages used in learning systems. We have analysed the domains and identified certain SIs that are necessary for a reasoned action selection. We have developed chain steps that introduce the necessary vocabulary into the problem models, supporting an RBP in reasoning about the possible SIs and the steps necessary to act towards a specific SI. We present two experiments to support this work. The first examines whether the vocabulary can be exploited to form RBPs that capture effective control for problems with SIs. The second experiment substitutes the presented framework with heuristic guidance. This simulates approaches that have learned control knowledge to improve heuristic guidance. We investigate whether our framework can be replaced with the guidance of a heuristic.

6.2.1 An analysis of the use of concepts of directed connectivity

We focus on key domains from each of the main forms of SI we have examined in Chapter 5. In this subsection we analyse the performance of RBPs on problems with SIs. In particular, problems that the policy could not provide guidance in without the developed framework.

Setup

We use the setup as described in Section 6.1. The three representative domains are: Blocksworld, Driverlog and Goldminer, providing examples of stacking, transportation and traversal problems. For each domain we have generated solver listing and enhanced domain model files, as described in Section 5.1. We have handwritten an

RBP for each domain, which exploits the vocabulary modelled in an enhanced domain. Directed connectivity is established for these domains through the use of specialised solvers: in Blocksworld, the `well-placed` predicate supports reasoning over correct sequences for stack construction; in Driverlog, a graph abstraction solver supports reasoning over graph traversals; and in Goldminer, a graph solver with the optional `nearest-blocked` predicate supports reasoning about opening paths. Each problem was solved using co-execution and the enhanced model.

On start-up the enhanced domain is constructed. The solver listing and enhanced domain model files are parsed; the enhanced domain model file details the required solvers; and the solver listing file details the parameters (Section 5.1). For example, in Driverlog, the enhanced model includes a graph abstraction solver and this is parameterised specifically for the domain, with properties such as the map is static and that `at` acts as the locatedness predicate. The appropriate solvers are instantiated and parameterised and then their initialisation functions are used to generate the corresponding initial state in the enhanced model. This state can then be used for matching with the rules of the RBP. For example, in the Driverlog RBP, the *move to pickup misplaced package* rule relies on the `long_drive-truck` action. This vocabulary supports directed connectivity for traversing trucks in the road map. The policy maps to an action and the appropriate solver is used to apply its effect on the enhanced state, or interpret the action for a lower language. The resulting action sequence is followed through the solvers until it has no effect. We have solved each of the problems, using the enhanced domains to test our system, and using the original domains with the domain independent planners.

Expectation of results

There are two aspects of the vocabulary provided in directed connectivity steps. The first is the reachability of certain SIs, which is important in determining what the next course of action should be. For example in Goldminer, the robot should move straight to the gold and pick it up if its path is clear. This relies on determining whether the gold can be reached. The second aspect provides a mechanism for moving towards the achievement of a specified SI. We demonstrate in Subsection 6.2.2 that even with a heuristic, selecting the correct actions to perform an SI is not always possible, and is improbable through blind search. If our policies can solve as many problems as the domain independent planner in a particular domain then we have successfully demonstrated that our policy can effectively control search in that domain. As these problems all have underlying structures then we will have demonstrated the exploitation of our enhanced model.

Results

A compacted version of the results of the experiment are presented in Table 6.1. The plan quality, in plan steps; the time taken, in seconds; and the number of problems solved, are plotted for FF, LAMA and the handwritten policy for the Blocksworld, Driverlog and Goldminer domains. The counts are summed over all solved instances. Where a planner covers more instances for a domain this entry is made bold; otherwise the best quality and time score are indicated in bold.

Table 6.1: Quality (Q), Time (T) and Coverage (C) results for FF (FF), Lama (L) and Handwritten (H)

Domain	FF_Q	FF_T	FF_C	L_Q	L_T	L_C	H_Q	H_T	H_C
Blocksworld	290	0.32	7	8766	966.46	39	7140	27.411	**45**
Driverlog	617	19.04	17	1120	45.76	20	**934**	**9.88**	20
Goldminer	642	403.76	25	824	519.46	28	814	22.187	**30**

The results in Table 6.1 demonstrate that our approach leads to complete coverage in the three domains. In Blocksworld and Goldminer we solve more problems than the domain independent planners. These results confirm that the use of directed connectivity is an important step in developing a language for expressing effective control knowledge. The quality of the solutions compares favourably in each domain.

6.2.2 Heuristically guided structure interactions

In this section we consider an alternative approach to controlling search over structures. In many planning algorithms, domain independent heuristics are key to generating plans. It is not surprising that previous approaches to directing search over graphs have delegated the task to the general purpose heuristic (Yoon et al., 2006; de la Rosa et al., 2008). It is interesting to investigate replacing our specialised vocabulary by allowing the decisions to be delegated to a heuristic. The framework that we developed in Subsection 5.1.2 supports using a heuristic to make the selection from the actions that the rules suggest. Our approach is to develop partially bound policies and rely on the heuristic selection process to choose from between these actions. In this section we present our approach to investigating it in our framework; and the results of our investigation.

Delegating to the heuristic

In this subsection we present our approach that delegates aspects of search control to the heuristic. First, we consider creating appropriate policies and we then discuss how the heuristic is used to select the action.

We can partially bind the rules with the information that we can express in the rule language. Partially bound rules will bind to a larger set of actions. A key property is whether the condition language is sufficient to determine the appropriate action. This property depends on the particular problem.

If the rule language is not sufficient to determine the appropriateness of the operator then it might fire when its operator is not appropriate for the current state. For example, in a Grid problem a robot could hold a key and be at the same location as another key. A proposition that determines whether the key in the hand is appropriate to progress towards the goal cannot be represented in the language. Similarly, it cannot represent a proposition to determine the appropriateness of the key at the current location, or in fact any other key in the map. This means that there are several distinct operators and not enough information to distinguish between. However, our rule representation insists on a single rule firing at each state and therefore a single operator. As a result one of these operators must be ordered first and will be fired in states where it is not appropriate.

The partially bound rule is used to generate a set of actions. Our architecture compares these actions using a heuristic. We follow other work by adopting the relaxed plan graph heuristic as the comparator (Fern et al., 2006; de la Rosa et al., 2008; de la Rosa and McIlraith, 2011). In this way the selection of the binding of the operator is delegated to the heuristic. An important limitation is that the set of actions is limited to the operator of the fired rule. In particular, the selected action is the action in the set generated from the fired rule that has the highest heuristic value. If there is not a useful action in the generated set then search will not progress. Therefore it is important that the rule language is sufficient to determine the correct operator.

The policies

In this section we look to define policies with partially bound parameters. We have decided to investigate this in the Driverlog and Goldminer domains. These domains are appropriate because each has an underlying structure and to some extent the rule language is sufficient to determine the correct course of action. Another benefit of these domains is that the structure interaction in each problem is different.

In this subsection we describe some of the important design decisions for the rules in the policy. Some of the rules can be kept because they have no interaction, or trivial interaction with the map. Other rules will have more involved interaction. We first identify the specific classes of relationships in the state that suggest that an operator is appropriate. We then use the operator as the rule operator. The final step is to partially bind any parameters if a suitable binding can be determined. We discuss our policies individually; the policies are provided in Appendix D.1.

Driverlog In Driverlog problems there are several stages: allocating drivers to trucks; picking up packages; dropping off packages; driving trucks home; walking drivers home. Of course some of these stages can be interleaved, such as picking and dropping off packages. Each of these stages requires traversal of a graph. However, as we have demonstrated we cannot construct the necessary antecedent directly from the state description.

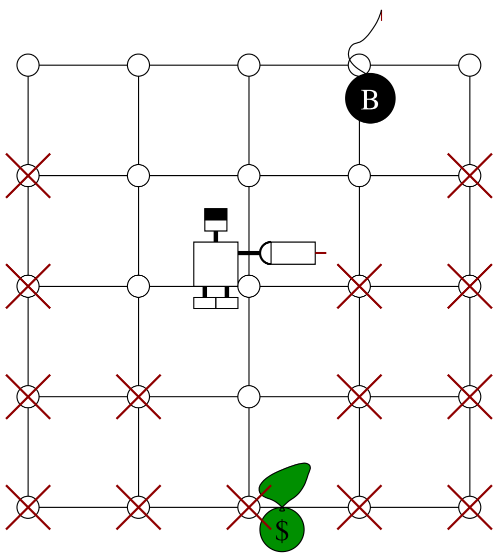
However, we can recognise that we want to walk a driver or drive a truck and present all of the possible walks or drives as options. For example, the fact that there are packages to be delivered can be identified. In this context we can assert that a truck should be moved. We cannot provide any guidance to prioritise any particular truck, or move in any particular direction. We then rely on a general heuristic to make sensible choices.

```
(:rule Drive_misplaced_package
   :parameters ( ?driver - driver ?l - location ?obj - obj
      ?to - location ?truck - truck ?from - location)
   :stateCondition (and (at ?truck ?from) (at ?obj ?l)
      (driving ?driver ?truck)
      (link ?from ?to) )
   :goalCondition (and (not (at ?obj ?l)) )
   :action (drive-truck ?truck ?from ?to ?driver)
)
(:rule Drive_package_in_truck
   :parameters ( ?driver - driver ?l - location ?obj - obj
      ?to - location ?truck - truck ?from - location)
   :stateCondition (and (at ?truck ?from) (in ?obj ?truck)
      (driving ?driver ?truck)
      (link ?from ?to) )
   :goalCondition (and (at ?obj ?l) )
   :action (drive-truck ?truck ?from ?to ?driver)
)
```

Figure 6.1: The rule conditions insist that a particular condition holds in the state, such as a package being misplaced. The desired operator is selected; however, the associated action is only partially guarded, or unguarded.

In the case of dropping off packages, we can do slightly better. We know the trucks that have packages in them and so can limit the heuristic to consider moving only these trucks. In this way we provide as much help as we can; however, we understand that we are in no position to understand how to reason over the structure. Figure 6.1 shows the rules for picking up and dropping off packages. The important aspect of these rules is that the variables used for establishing a particular class in the state are mostly separated from the variables used for the rule action precondition.

Goldminer The Goldminer problems have a similar series of identifiable stages: get laser; shoot through to one away from the gold; get bomb; blow up gold square; pickup gold. However, the shoot through to one away from the gold stage requires interweaving of fire laser and move actions. The class indicating the correct use of these rules is the same for both of these rules. This introduces an ordering problem. The key issue is that the preconditions of each action will often be true after the first application.



```
(:rule Clear_rock
  :parameters ( ?l1 - loc ?l - laser ?l2 - loc
    ?l3 - loc ?r - robot)
  :stateCondition (and (at ?r ?l1) (rock-at ?l3)
    (at-gold ?l2) (holds ?l ?r)
    (connected ?l1 ?l3) )
  :goalCondition (and (holds-gold ?r) )
  :action (fire-laser ?r ?l ?l1 ?l3)
)
(:rule Move_with_laser
  :parameters ( ?l1 - loc ?l - laser ?l2 - loc
    ?l3 - loc ?r - robot)
  :stateCondition (and (at ?r ?l1) (at-gold ?l2)
    (holds ?l ?r) (connected ?l1 ?l3)
    (clear ?l3) )
  :goalCondition (and (holds-gold ?r) )
  :action (move ?r ?l1 ?l3)
)
```

Figure 6.2: The robot has to shoot all of the surrounding rocks before it can move to the next location.

For example, if the robot blows up the rock at a location, then there will often be another location next to the robot with rock at it. If the robot moves between two locations then there is often another location that is clear to move into (for example, the location it just moved from). If the move action is ordered first then the robot will enter a loop moving through a cycle of locations. The fire action will never be considered. If the fire action is ordered first then the robot will continue to blow up rock around it until all rock is gone, or an earlier rule fires. This means that we will create a three wide path towards the gold, this is illustrated in Figure 6.2. Although not particularly efficient, it allows the potential for a successful execution.

The investigation

We have handwritten partially-bound policies for the Driverlog and Goldminer domains. In this subsection we investigate whether the relaxed plan heuristic compensates for the weakness in the control knowledge. In particular, can the heuristic provide effective guidance over structures. Our experiment uses the partially-bound policies to guide search on the benchmark problem sets. There are two aspects that we use to inform our conclusions. The first is whether the problems are solved by this configuration. The second is the quality of the plans; in particular, the quality of the plan steps during interactions with structures.

The experiments are set up as we described for the expressivity results. For purposes of comparison we have plotted the results for several policy configurations and JavaFF. JavaFF is included because the heuristic computation relies on the JavaFF system. It is therefore an appropriate planner to use in comparison. We present our findings in the Driverlog and Goldminer domains.

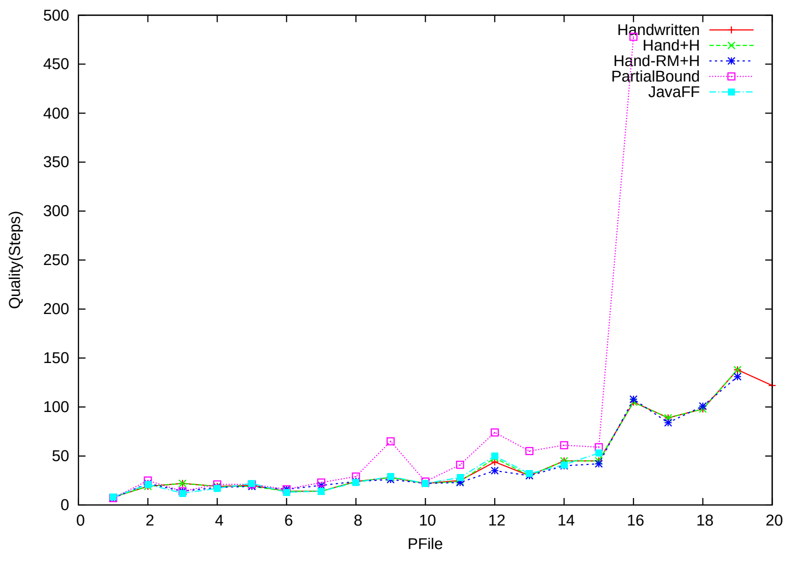


Figure 6.3: Quality results for the partially bound policy on Driverlog problems

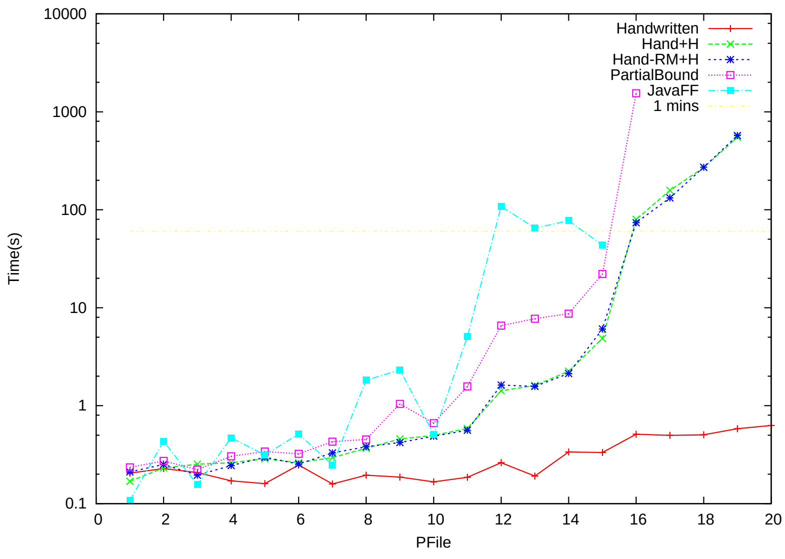


Figure 6.4: Time results for the partially bound policy on Driverlog problems

Driverlog The results of the experiment are presented for quality in Figure 6.3 and time in Figure 6.4. We have plotted results for several policies:

- Handwritten: the handwritten policy as described above.
- Hand+H: This is the same as Handwritten except we use the heuristic to order the actions. This plot demonstrates the effect on time that using the heuristic to order the actions.
- Hand-RM+H: This is the Hand+H, except we do not use the resource management solver to make resource allocations. This plot demonstrates the quality of the plan if resource allocations are delegated to the heuristic.
- PartialBound: The policy described in the text above.

As can be seen from the results, the heuristic guidance succeeds to solve problems in the Driverlog problems. However, the plans are longer and take longer to compute. It appears that the planner is not receiving the same level of guidance over structures. In plotting these policies we can understand where the decrease in quality and increase in time are coming from. It is expected that the time should increase for two reasons. Firstly, because the rules bind to more actions there are more heuristic computations; and secondly, because the plan length is longer.

We have plotted the handwritten policy without the resource management; this makes the execution reliant on the heuristic providing the resource allocation. The difference between this policy and the partially bound policy is the use of vocabulary for reasoning over graphs, provided through the graph abstraction solver. The fact that the partially bound policy computes longer plans suggests that the paths that are generated are worse than the paths generated by the specialised solver.

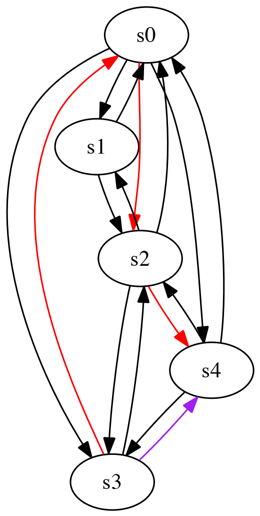


Figure 6.5: The topology of the link predicate for problem 9. The partially bound policy generates the path between s3 to s4 illustrated in red. An alternative optimal path is drawn in purple.

The quality results indicate that the general heuristic is not providing the same quality of control as the specialised solver provides. In particular, the paths that the traversers are led through can be sub-optimal. For example, in the solution to problem 9, the generated plan moves a traverser through three steps when the locations were connected; the path is illustrated in Figure 6.5.

Goldminer The results of the experiment are presented for quality in Figure 6.6 and time in Figure 6.7. We have plotted results for several policies:

- Handwritten: the handwritten policy as described above.
- Hand+H: This is the same as Handwritten except we use the heuristic to order the actions. This plot demonstrates the effect on time that using the heuristic to order the actions.
- PartialBound: The policy described in the text above.

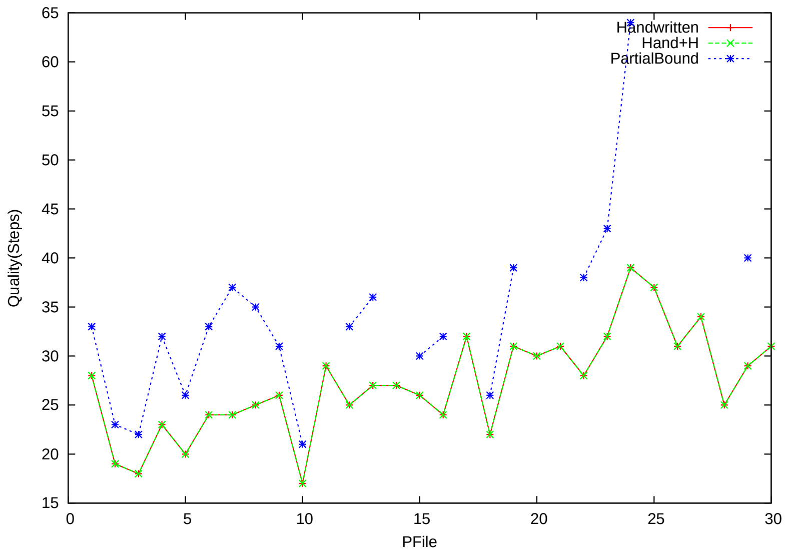


Figure 6.6: Quality results for the partially bound policy on Goldminer problems

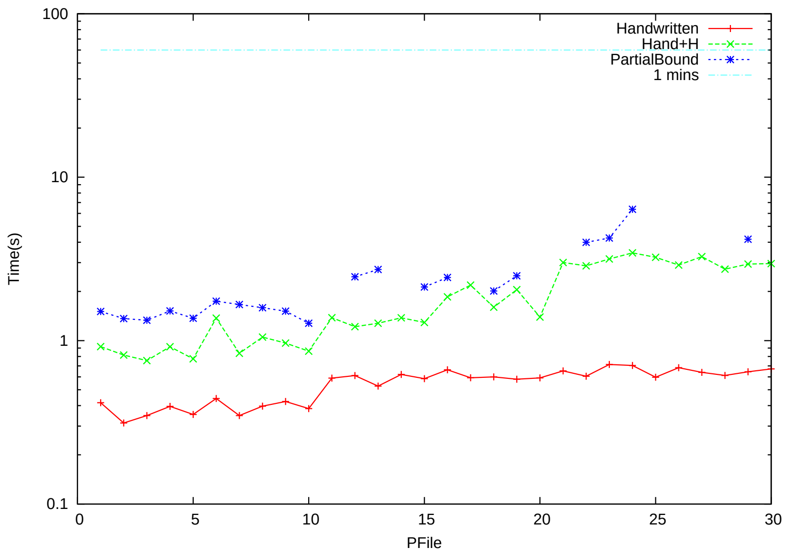


Figure 6.7: Time results for the partially bound policy on Goldminer problems

The graphs show that the partially bound policy fails on several Goldminer problems. Where the execution manages to solve the problem the solutions are longer and take longer to generate. The increased solution length is partly due to the partially bound policy allowing a three-wide path to be made. However, this is not the only cause: the path back to pick up the bomb revisits all the locations created during the journey to the gold.

Both the increased journey back to the bomb and the cause of failed search is due to the choice of heuristic. This is made clear by the failure of JavaFF in solving any problems. This is because the relaxed plan heuristic ignores delete effects. This means that the relaxed plan will move to the gold square using the laser (or bomb) and pick up the gold. Firing the laser at the gold square does not destroy the gold in the relaxed problem. At any state along this path, returning to pick up the bomb makes no sense at all. This results in a fight between the heuristic and the cycle detection. The heuristic promotes moving back towards the gold in any state.

The problem is increased by the three wide path that has been made with the laser during the shoot through to one away from the gold stage. The robot is moved to locations that are as close to the gold as possible and this can result in the search getting stuck.

6.2.3 Conclusion

In this section we have presented two empirical analyses that support establishing directed connectivity in problems with SIs. We have provided evidence that the enhanced vocabulary provides effective support to the RBP, leading to effective planning in SI problems. We have investigated replacing the special purpose solvers by delegating the decisions to a general heuristic. This is not the first time that a general heuristic has been used to provide control where determining the appropriate interaction with a structure cannot be realised in the rule language (Yoon et al., 2006; de la Rosa et al., 2008; de la Rosa and McIlraith, 2011). The reported plan qualities of these approaches are poor for problems with maps. We have supported this by showing that our solvers greatly improve on the solutions found using the relaxed plan graph heuristic.

6.3 Optimisation

In this section we test the two approaches to optimisation that were presented in Section 5.5. First of all we analyse the impact of introducing graph properties to support the rule system in distinguishing between alternatives. We then investigate the three tier optimisation framework that allows two types of heuristic to be exploited. The main focus in this section is on plan quality. We anticipate that providing more information can be exploited in strategies to improve quality and that using heuristic over arbitrary choice will result in better quality solutions as well.

6.3.1 Supporting the policy in making comparisons

In Section 5.5 we discussed various graph properties that could provide useful information for solving planning problems. We use a similar setup to the previous experiment, and introduce a clustering solver and a centrality solver. The information they provide can be used to support strategies that require sharing resources, or exploiting closeness. We investigate their impact in problems from the Driverlog domain.

Setup

We have used the setup presented above to compare three different strategies. We have selected the Driverlog domain for this experiment as the problems feature issues of resource allocation and graph traversal. We use the problem set from the 2002 IPC. We have hand-written three policies for the domain that exploit the available features. The first policy captures a basic handwritten strategy that favours picking up packages over delivering them. The second strategy utilises clusters. The graph is clustered and these clusters are used as zones. The strategy favours picking up packages and then dropping them off within its current zone. It will then move a truck to pickup a package in another zone. There is nothing to ensure that these zones will be close. The final strategy uses a hub node. The graph clusters are used so that the use of hubs is decided at an abstract level. If the path from a package position through the hub is not much more than the path directly to its goal then that package will be routed through the hub. For those packages that are passing through the hub, the solver also computes an appropriate truck for delivering packages to the hub. In each strategy the long move action is used and the actions are applied as macro actions.

Expectations

The basic strategy moves towards misplaced packages using macro actions. Resource management and the selection of the next target are products of the rule bindings and the alpha-numeric ordering. This should lead to long plans; the lack of sophistication may result in shorter planning time. The clustering approach breaks down the problem; this means that pickups and drop-offs are made for nearby locations together. Of course, packages might be found later that must be delivered to locations already visited and the trucks might retrace their steps. The use of hubs will result in more structured plans. However, the improvement in quality will depend on whether the cost of transferring packages is compensated for by more efficient allocation during the delivery and collection tasks.

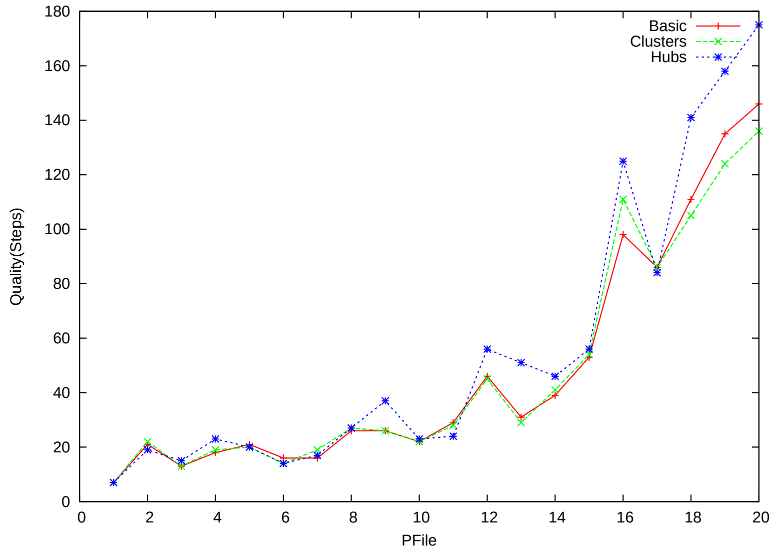


Figure 6.8: Quality results for the basic strategy and the strategies exploiting clusters and a hub node

Results The results, presented in Figures 6.8 and 6.9 plot a basic strategy against two more sophisticated strategies, which introduce a more structured approach to solving the problem. The graphs suggest that the use of the hub node does not lead to shorter plans and that basic strategy can be used to compute plans more quickly.

The more sophisticated strategies do not result in largely different plan quality. The use of a single hub location seems to make the strategy much worse. The benchmark Driverlog problems have densely connected truck and driver graphs (Gregory and Lindsay, 2007). This means that the impact of making random choices over the next package to select, or the best truck to drive, is greatly reduced. In fact the quality of the plans for the basic strategy only start to lose in comparison with the TLPLAN quality

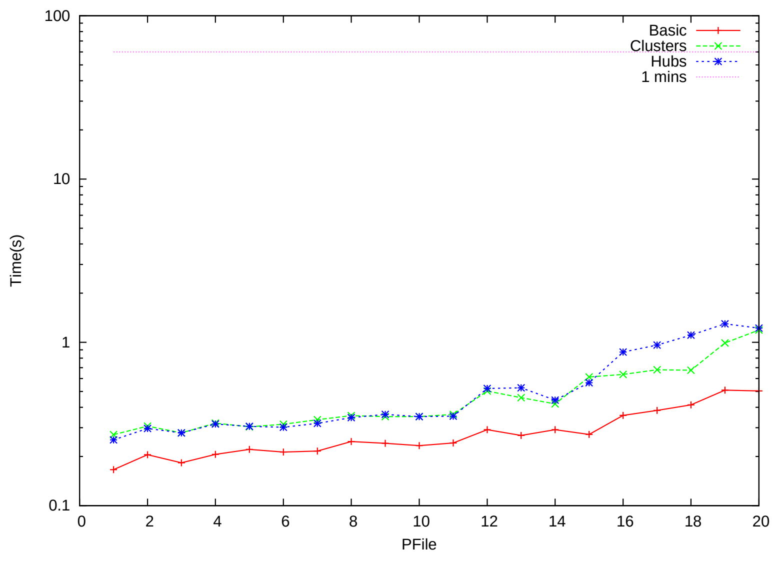


Figure 6.9: Time results for the basic strategy and the strategies exploiting clusters and a hub node

in the last few problems. It also suggests that the added cost of bringing a package to a central location and swapping it is not worthwhile. The increased time can partly be explained by an external process that is used to compute the weakly connected components of a graph. There are also more rules with more predicates, increasing binding utility.

There are several aspects of this experiment that could be contributing to the weak performance. The clustering solver was quite naive. The approach to clustering was to grow clusters with their neighbours until the clusters were a certain size. This could have resulted in weak clusters. We only identify a single hub and use a multiplying factor to bias towards using the hub node. Another factor has been the strategies that we developed that used the features; perhaps another strategy would have worked.

The conclusion of this experiment is that including extra words into the model will be reflected by better performance. We would need to develop our solvers further in order to confirm the benefit of clusters and hubs in this domain. However, it is an important result that we have made information that we thought would be useful explicit in the problem model and it appears this is not the case. In the case of directed connectivity we can identify the behaviours that are necessary to compute a plan; but, identifying propositions that could lead to a more effective strategy requires much more thought and effort. We will therefore focus on using heuristics for processes of optimisation.

6.3.2 Global and local solver heuristics

In Section 5.5 we developed a tiered approach for exploiting heuristics. There are two important roles that heuristics play. The knowledge engineer will select some view of the model, $\mathbb{M}|_{\Sigma_i}$ and any decisions that cannot be biased with the vocabulary expressible in this language is left to a deterministic decision process. In this subsection we analyse whether exploiting heuristics leads to improved selection in these cases.

Setup

The same framework is used as above. We analyse the performance of four configurations on problems from the Driverlog, Goldminer and Grid domains. For each domain we use the similar policies, however, the settings are changed. The settings are as follows: no heuristics (Basic); solver heuristics (+Lh), which are nearest neighbour and resource management; global heuristic (+Gh), which was the relaxed planning graph heuristic in JavaFF; and both heuristics (+LGh). In Driverlog problems the strategy uses resource management to allocate trucks to packages and drivers to trucks; nearest neighbour heuristic determines the next allocated package to pick up and if there are none then it selects the next to be dropped off. In Goldminer and Grid problems, moving in the graph uses the nearest neighbour heuristic. To switch on and off the Resource Management aspect of the Driverlog RBP requires two separate solutions. This is because the heuristic is encoded as a derived predicate and therefore features in the rule conditions. These predicates are the only difference in the policies used.

Expectations

We have shown above that the approach with no heuristics is effective in Driverlog. We are therefore more conservative with our expectations in the quality improvement made by the heuristics. The solver heuristics adopted in Driverlog are appropriate and should provide some assistance. The relaxed plan heuristic is not particularly effective in the Goldminer domain, as discussed in Section 6.2.2 and we would expect few alternatives to be provided by the rule system. The solver used for Grid, is based on the relaxed plan already and this consistency could lead to better quality plans. In terms of time, the JavaFF heuristic is expensive to initialise and compute. This initialisation is required for the solver used in Grid. The nearest neighbour and resource management heuristics are cheap to compute in these domains.

Results

Table 6.2: Quality (Q), Time (T) and Coverage (C) results for Basic (B), solver heuristics (Lh), global heuristic (Gh) and combined heuristics (LGh)

Domain	B_Q	B_T	B_C	Lh_Q	Lh_T	Lh_C	Gh_Q	Gh_T	Gh_C	LGh_Q	LGh_T	LGh_C
Driverlog	960	10.044	20	930	**9.661**	20	953	61.811	20	**912**	42.656	20
Goldminer	**814**	24.61	30	**814**	**24.573**	30	**814**	41.603	30	**814**	41.203	30
Grid	289	131.745	20	267	**125.891**	20	261	191.642	20	**259**	182.19	20

The results, presented in Table 6.2, suggest that the combination of both heuristic types lead to the best quality plans over each of the problem sets. In Driverlog it appears that the local heuristics have more impact, whereas in Grid the global heuristic is more effective. The quality in the Goldminer problems are the same; this is because the nearest closed node heuristic used to establish directed connectivity identifies the best path to the target. In Driverlog (Figure F.15 in Appendix F.2), we expected that the difference in quality between the solutions would increase as the graph size increased. However, this trend is not apparent in the results. The graphs for Driverlog and Grid suggest that most of the difference between the different approaches is made over a small number of problems. For example, in Grid there are two problems where the basic solution have considerably worse quality. The reason for this in Driverlog, could be due to the densely connected graphs; whereas in Grid it might be because there are few situations where two keys of the same shape can be reached in the same state.

The use of the JavaFF heuristic leads to longer planning times; however, more efficient implementations of the heuristic exist. These results confirm that heuristics can usefully select between the alternatives mapped to by the policy. The complete graphs can be found in Appendix F.2.

6.3.3 Conclusion

The analyses in this section have examined the problem of controlling global optimality through local choices. Using properties of the interactions in the model that provide more distinction between states is an important option. However, identifying appropriate properties is an involved process, which requires an understanding of the domain, specific problem distribution and RBP representation. It is therefore more appropriate in the context of a single domain, when specific insights can be exploited. The consistency of local heuristics with global optimality is not certain. However, the heuristics that we have selected in the analysis contributed to the planner finding better plans. We conclude that using an RBP to filter the successor actions and using heuristics to select between the alternatives is an effective approach for planning.

6.4 Level of reasoning

The level of reasoning is a key aspect of this work. However, our investigation into process and full solution level actions in Subsection 5.6.1 concluded that the RBP can provide less guidance in how the plans are constructed. Examining how these behaviours can interact could be an interesting line of research, in drawing together specialised solvers that are already developed. However, the RBP was given very little control in order to bring strategies together. In this section we focus on the step by step macro application approach, which provides the RBP with a greater level of control and could be used to bring together these process like solvers.

6.4.1 Step by step macro application

In this subsection we compare the SbS with the macro application approach. The motivation behind the SbS was to allow the RBP's priority ordering to be maintained at each plan step. In this section we analyse the impact that this approach has on the plan quality and planning time. Our main aim is to understand the trade-off between the reduced number of policy mappings incurred when applying macro actions in a single step and the quality gain through exploiting opportunities facilitated by using SbS.

Setup

We use the framework that we have developed above. We solve the problems from the Driverlog, Goldminer and Grid domains using the same policies in each configuration. In the first configuration we adopt the SbS and in the second we use the macro application approach (Macro). The policies are the same as those used for Section 6.5 and include solver heuristics. It should be noted that more care is required when constructing rules that select macros using SbS. The rule system will be evaluated on each intermediate step and during this path some of the initial conditions might not hold. For example, in the case of a pickup-drop macro for a Blocksworld problem a condition for a rule might include the condition (`attached-to` *?A ?B*), but this condition will not match in the state where the block is being held. We have run the Grid experiments three times, as the solver is sensitive to the generated relaxed plan, which can effect the plan quality.

Expectations

We expect that SbS will lead to better quality plans in domains where the forced ordering of rules is an undesired artefact in the control knowledge. For example, in Driverlog, the choice of moving to pick up or drop off a package depends on the situation and cannot be determined up front. In domains where the next best steps can be identified in the rule language then this mediating step is unnecessary and is unlikely to lead to improved quality. We expect that the macro application will generally be faster, as it is likely that the policy mapping will be computed in fewer states. For example, in Goldminer directed connectivity with the nearest neighbour, provide the necessary vocabulary to determine the next target. We would expect that the use of macros will improve the time and have little effect on the quality of the solutions in this domain.

Results

Table 6.3: Quality (Q), Time (T) and Coverage (C) results for Step by step (SbS) and Macro (M) application approaches.

Domain	SbS_Q	SbS_T	SbS_C	M_Q	M_T	M_C
Driverlog	**922**	10.732	20	930	**9.135**	20
Goldminer	**814**	26.235	30	**814**	**23.929**	30
Grid	816	348.77	18	**792**	**259.36**	18

The results of our experiment are consistent with our expectations in Driverlog and Goldminer. The evaluation of the RBP at each state leads SbS to take longer to generate plans for Goldminer and Driverlog. The quality results, presented in Table 6.3, illustrate that the quality of plans for Driverlog problems were better when using SbS. The size of this improvement is quite small. In particular, there is no obvious increase in improvement as the problem size increases (graphs in Appendices, Section F.3). We are only able to drop off packages when a truck that contains a package is located at the package's goal location. The results suggest that this condition is not often satisfied.

The planner has no control over the path selected to move between targets. Of course, as the graph size increases, the chances of passing a package will reduce. Combining the SbS with clusters would extend the opportunities that were exploited.

The results in Grid are more surprising. The time results are consistent with our expectations. However, the improved quality of the solution in the macro settings was not expected. The reason behind this is that our solver generates a relaxed plan at each state and extracts the first target in the plan. As a result, during planning using SbS, the plan might change several times as the robot is moved towards a target. Through applying the macro directly this indecision is prevented. In Section 5.4 we demonstrated that macro actions provided one approach to solving the problem of setting a target for traversal. In conclusion, we have demonstrated that using the SbS can lead to shorter plans. However, the improvement is quite small; moreover the time results when using macro actions are better. We consider that both of these approaches provide strong performance, although note that it can be easier to construct control knowledge when using the macro application approach.

6.5 Analysis of the architecture

In this section we analyse handwritten rule-systems expressed over the vocabularies that were developed in Chapter 5 and using the previous results to guide their development. The main aim is to demonstrate that the framework can solve problems in several domains. If our rule systems can demonstrate effective control across a variety of benchmark domains this will indicate that our framework is a practical approach to planning. We consider how the solvers have been used in combination in Section E.2 of the appendices.

6.5.1 Generality of framework

In this subsection we investigate the generality of our approach. We have not used our approach to solve problems from all available domains. Instead we have selected a range of domains that demonstrate specific aspects of the framework. Five of these domains have traversing or building sub-problems. To these we have added the Logistics domain, which requires the use of a more general language enhancement; and the Gripper domain, which demonstrates a limitation of the rule based policy approach.

We have handwritten policies for each of the seven domains (these are presented in Appendix D.1). Using the approach that we have explained we have used these policies and two state of the art domain independent planners to solve the benchmark sets. To avoid bloating the main text, we present a summation of the results and direct the reader to a complete presentation of the graphs in Appendix F.

Expectation of results

We have proposed that one of the key limitations of our rule language is that it cannot express certain concepts of structures. We have developed an approach to enhancing the problem model with certain words that provide a view of these structures. In this experiment we investigate whether we can exploit these words and express effective control in our limited rule language.

A main focus of research in domain independent planning has been on increasing the coverage of the planners. We use LAMA and FF to compare to our policy. These planners have demonstrated impressive performances in the number of problems they solve over several IPCs. These planners exploit sophisticated heuristics that have been evolved through several generations of research. These planners provide a high bar to compare the coverage of our solutions.

Our aim in this experiment is to compare our policies to these planners in terms of coverage. If our policies can solve as many problems as these planners in a particular domain then we have successfully demonstrated that our policy can effectively control search in that domain. In particular, if that domain has an underlying structure then we have demonstrated the exploitation of our enhanced model.

Results

A compacted version of the results of the experiment are presented in Table 6.4. The plan quality, in plan steps; the time taken, in seconds; and the number of problems solved, are plotted for FF, LAMA and the handwritten policy on a range of domains. The counts are accumulative over all solved instances. Where a planner covers more instances for a domain this entry is made bold; otherwise the best quality and time score are indicated in bold.

The results in Table 6.4 demonstrate that our approach is suitable for a wide range of domains. In general our coverage in the tested domains is very good. In some domains we solve several more problems than the domain independent planners. The domains that we perform best in are those with structures in them. These are the domains that we cannot express control knowledge for without support. This confirms that the language steps that we defined in Chapter 5 are appropriate.

Table 6.4: Quality (Q), Time (T) and Coverage (C) results for FF (FF), Lama (L) and Handwritten (H)

Domain	FF_Q	FF_T	FF_C	L_Q	L_T	L_C	H_Q	H_T	H_C
Blocksworld	290	0.32	7	8766	966.46	39	7140	27.411	**45**
Depots	968	19.46	20	1102	1103.9	20	895	8.194	**22**
Driverlog	617	19.04	17	1120	45.76	20	**934**	**9.88**	20
Goldminer	642	403.76	25	824	519.46	28	814	22.187	**30**
Grid	**288**	**6.64**	20	297	230.54	20	281	123.54	18
Gripper	32240	524.54	20	32240	**101.18**	20	32240	622.024	20
Logistics	1112	**0.02**	27	1119	0.2	27	**1095**	9.384	27

We solve more problems than FF in four domains and more problems than LAMA in three. Grid is the only domain that we solve fewer problems. The performance of our solution in the Grid domain is not surprising. It is the only domain that we allow the instantiation of the action operators. Figure 6.10 demonstrates that JavaFF is not able to complete this operation in the two problems that we do not solve. However, our general coverage and when combined with the speed and quality of the solutions, these results demonstrate that the rules can exploit the necessary information.

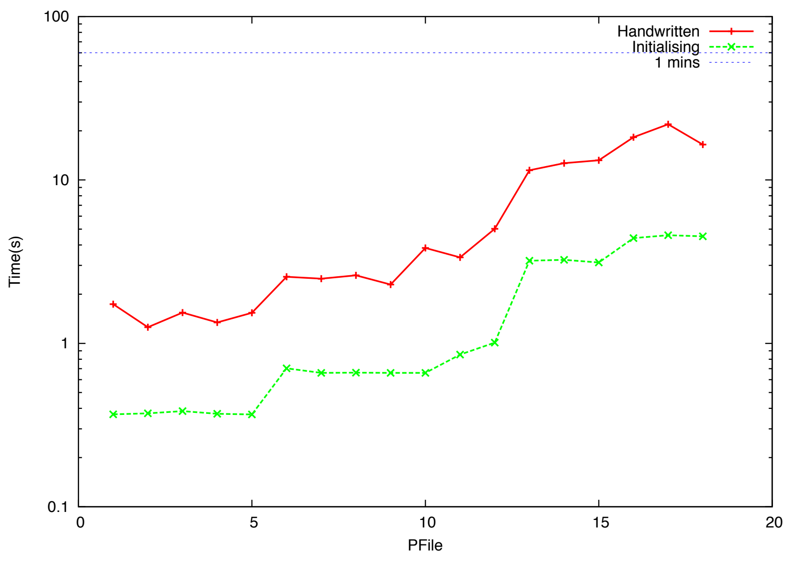


Figure 6.10: Time results for the initialisation of JavaFF and our policy runs on Grid problems

The two domains that we compare least well are Gripper and Logistics. There are no examples of the SIs that we have focussed on in these domains. In Logistics we use a not-equal solver to express control knowledge; in Gripper we do not use any additional information. These domains are included to examine other properties of our architecture.

In most problems the number of combinations of objects that satisfies the rules are relatively small. However, in Gripper there are many symmetric objects. These generate a lot of combinations and during search we generate the same options many times. This is one of the reasons that most modern planners (including FF and LAMA) ground all actions up front, although approaches to detecting and exploiting symmetries have been investigated (Fox and Long, 1999). When generating combinations that satisfy a rule from the state and goal, we aim to generate all possible combinations. This allows us to feed all of the possible actions to an ordering process. In the benchmark problems, instances with very high numbers of symmetric objects (for example, 750) are unique to the Gripper domain. We conclude that our approach is less effective at dealing with symmetry than FF and LAMA prove to be in this domain; however, the problems have very long plans (up to 2325 steps) and it is an important result that our approach generated a solution to every problem. Moreover, our implementation is in Java and not optimised to the level of FF and LAMA.

6.6 Summary

In this chapter we have investigated our framework. We have demonstrated that it supports effective planning with strong results in terms of coverage, speed and quality. In the Appendices, Section E.1, a comparison is made between the quality of our plans and those of TLPLAN and an optimal planner, h^{LM-cut}. The main question that we set out to answer in this chapter was whether control knowledge exploited in our framework could control search in domains with rich structures. We demonstrated that our solution was able to solve more problems than state of the art domain independent planners; that the time of solving the problems was often less; and that the quality of the plans was comparable. These results were gathered for domains with different structures, requiring different SIs. In one sense the results are not surprising: we are comparing hand crafted domain dependent strategies with domain independent techniques. However, we have used a limited language to express our strategies. In particular, our rules are conjuncts of the modelled propositions. As well as this, most of the strategies have not used search. This means that the chains that we have developed in Chapter 5 provide the necessary information for expressing strategies that generalise to the problems of a domain.

We have examined several of the options that we presented in Chapter 5 regarding the vocabulary chains. We investigated SbS and discovered that in problems with structures, the quality of the plans could be improved for an increase in the planning time; however, as the structure size increased, the chance of exploiting an opportunity decreased. The results also confirmed that the use of SbS can require more care when expressing solvers and RBPs. We have observed that improving specialised solvers to improve efficiency is a time consuming process, requiring much trial and error. As an alternative, we investigated the use of heuristics and observed that the quality can be improved using both local and global heuristics, with respect to particular solvers. We have also demonstrated that vocabulary modelling directed connectivity supported the RBPs through SIs. This is an important result and clarifies the appropriate combination of heuristics and language enhancements. In the Appendices, Section E.3, we compare our control system for Driverlog and Blocksworld directly with TLPLAN. The captured control knowledge is similar in Blocksworld, whereas more sophisticated in Driverlog. However, the quality comparison, presented in Section E.1, indicate that the quality of solutions generated are comparable. This suggests the main difference is the efficiency of utilising the control knowledge and illustrates the benefit that alternative encoding methods can provide. In this section we demonstrated that the chains that we have investigated provide some of the features utilised in the TLPLAN solution; for example, level of reasoning and compiled preconditions.

We have investigated our framework using several criteria and it has performed well in each of our experiments. We conclude that the presented framework supports effective control in domains with rich structures. We observe that we should model vocabulary at as low a level of reasoning that still establishes directed connectivity, to maximise the level of control the planner has on the vocabulary. In Chapter 7 we investigate the problem of invoking an appropriate vocabulary automatically. As part of this we investigate identifying new chain steps automatically and the results in this chapter lead us to focus this study to vocabulary that establishes directed connectivity for particular SIs.

CHAPTER 7

AUTOMATING MODEL ENHANCEMENT

In Chapter 5 we proposed several chains of language restrictions that accept actions and propositions that are effective for expressing general policies for problems with specific structures. We have developed a modelling language that allows us to define enhanced models. However, this requires each domain to be examined and the appropriate language enhancements to be identified. This is time consuming to do by hand.

In Section 5.3 we identified SIs as an underlying cause of the limitations of our planning approach. We observed in Chapter 6 that enhancing the problem model with vocabulary modelling directed connectivity could support RBP execution during SIs. In this chapter we investigate whether the appropriate problem model can be selected automatically. The first part is to automatically determine whether a particular SI exists in a problem. We use the technique developed in Fox and Long (2001), which relies on capturing the criteria that determines the existence of SIs. We demonstrate that this approach can be automatically extended to invoke appropriate language enhancements, reducing the effort for exploiting the language in the future. This aspect of the work is reported in Lindsay et al. (2008, 2009).

The solvers that we used for the experiments in Chapter 6 are heavily parameterised. The extension to invoke language enhancements involves carefully interrogating the problem model to identify the appropriate values for the parameters. A problem of more consequence is that the solutions required for each parameter set can be subtly different. The second task is therefore to automatically generate solvers that model an appropriate vocabulary for establishing directed connectivity in a particular domain. We present a general model for expressing SIs that generalises the directed connectivity solvers for transportation, path opening and stacking problems that were defined in Chapter 5. We then present the problem of parameter selection for its enclosing solver and explain how the solver is parameterised for traversal and stacking problems. This aspect of the work is reported in Lindsay (2012).

In this chapter we investigate automating the tasks of solver selection and solver generation.

7.1 Automatic enhanced PDDL generation

We have presented a language for enhanced problem models. This is in the form of a more general description model that explains the behaviour of a group of problems. Each time we consider a new domain model the appropriate solvers must be identified and those solvers must be parameterised. This is time consuming to do by hand and requires understanding of the domain and solvers.

Domain analysis has been developed that uncovers properties of groups of planning problems (Fox and Long, 1998, 2001). In HybridSTAN (Fox and Long, 2001), behaviours called *generic types* are defined across planning domains, providing a new semantic layer for comparing object function in planning problems. This approach has been exploited to invoke appropriate heuristics (Fox and Long, 2001) and control knowledge (Murray, 2002; Murray et al., 2003) We apply the approach used in HybridSTAN to select appropriate domain models automatically.

In this section we describe the mechanism that supports our mapping from a domain and problem model to solvers and finally an enhanced problem model. We begin by discussing the search space; we introduce the domain analysis technology that we exploit; and then we present how the mappings are realised.

7.1.1 Search space

The search space that we explore in this section is the space of chains of language restrictions, $\Sigma_0, \dots, \Sigma_n$ (Definition 3.1.2). In particular, we are interested in selecting those features or chain steps that best support expressing concise control knowledge for action selection, which is related to the problem of feature selection (Blum and Langley, 1997; Kohavi and John, 1997; Guyon, 2003). In de la Rosa and McIlraith (2011) a subset of the chains are defined explicitly and explored in a beam search.

In this section we limit our focus on individual chain steps defined by the solvers we have implemented and define the appropriate model as the collection of the appropriate chain steps.

7.1.2 Domain analysis

TIM is a domain analysis tool that uncovers certain implicit properties of a domain model. This analysis can be extended to identify generic sub-problems (Fox and Long, 2001), such as transportation and resource management. For a full explanation of how this analysis is carried out please refer to Fox and Long (1998, 2001).

Properties in TIM

TIM uncovers a collection of finite state machines that define how the objects in a problem can move between different sets of abstracted relationships. This is achieved by analysing the effect of each parameter of an operator to establish how the operator changes the relationships of the object that instantiates the parameter. The objects are then partitioned into types based on the parameters that they could bind with and therefore the relationships they can pass between. Although the TIM analysis is created from a single benchmark problem, it is common that the identified structures are appropriate for a large number of the benchmark problems. A key aspect of TIM is that it establishes an abstracted layer that captures the generic behaviours of the objects, while also identifying various invariants that are frequently the result of domain conventions.

A property, (`p`,`i`) is a pair, with predicate, `p`, and integer, `i`, that represents one of an object's relationships. For a proposition of an object, `p` is the predicate of the proposition and `i` is the parameter position of the object in that proposition. Properties are grouped into property states, which capture the properties that are held together. A property space is a set of states with the rules that govern the state changes for a particular object behaviour. For example, in Figure 7.1, the package property space illustrates that packages can transition between `in` and `at` relationships. The space also has a list of objects that belong to the space (for example, the set of all packages in the problem). An object can exist in more than one property space and each membership can be interpreted as a behaviour. An important aspect is that a proposition of an object that has an associated property in a property space, is closed under the rules of that space.

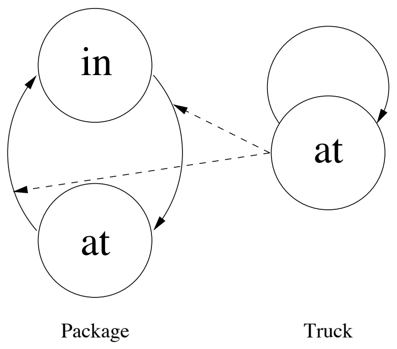


Figure 7.1: Trucks transition between different located properties and packages transition between being related to locations and trucks.

Two example property spaces are illustrated in Figure 7.1. Packages can transition from being `in` trucks to being `at` locations. Moreover the illustrated property space has the interpretation that a package must be either at exactly one location or in exactly one truck. In the Briefcase domain, the `at` and `in` properties have index 0 (`at`0). Trucks can transition from being `at` one location to being `at` another location. The truck property space captures the invariant that a truck will always be at exactly one location. These invariant are typical of transportation domains, however, they are commonly not explicit in the model. Therefore domain analysis provides a method of making explicit some of the domain conventions and also uncovering the various behaviours encoded in the model.

Generic types

The analysis in TIM partitions the potential transitions of an object into different behaviours (Long and Fox, 2002). This behavioural level has provided a useful language for describing generic behaviours that exist in several domains (Long and Fox, 2002). A key benefit of this level of analysis is that certain sub-problems can be identified at the generic behaviour level and specialised solutions can be exploited at the specific instance level. For example, through improving weak heuristics (Coles and Smith, 2006), or inducing specialised heuristics (Fox and Long, 2001) or control knowledge (Murray, 2002; Murray et al., 2003).

The key to the success of these approaches is that there is a clear description of when the solution is appropriate for the current problem. This description is captured by a set of criteria, called a fingerprint, which establishes the requirements of the generic type (Long and Fox, 2002). A problem model that satisfies the criteria exhibits the associated generic type. Having a well defined fingerprint is particularly important in Fox and Long (2001), where the problem model is decomposed and Murray (2002); Murray et al. (2003), where pruning rules are used: each of these approaches can cause the problem to become unsolvable if applied inappropriately. In Coles and Smith (2006), inappropriate use will result in a weaker heuristic. In our approach it could result in enhancements to the problem that are not useful; however, using the vocabulary is decided by the control knowledge engineer, or through a learning process, which is informed by performance. We do not explore here whether relaxing the definition of a fingerprint leads to wider utility of the solvers.

The transportation problem fingerprint In Long and Fox (2002), a network of roles related to the transportation problem are identified. The graph traversal behaviour underlies the transportation behaviours. A graph traversing behaviour is characterised by a property space with a singleton state and a single self-transition on that state (illustrated in Figure 7.1). This is an object that transitions between different instantiations of the same predicate. The objects that belong to this space are labelled traversers (these are referred to as mobiles in Long and Fox (2002)) and these play a role in path opening and transportation problems. The single property represents the locatedness predicate that relates the objects of the property space to the set of locations. The variations of these relationships are explored in Long and Fox (2002).

A transportation problem requires a traverser that acts as the transporter. Certain objects, called transportable objects, can be attached to this traverser and moved to other locations (perhaps attached through a chain of static propositions, such as those representing an arm). Transportable objects are characterised by a property space with two states that are linked to each other in both directions (illustrated in Figure 7.1). One state includes a property that links the object to a traverser and the other state has a property that links the object to a location. The actions associated with the links between the states must require that a traverser is located at the location that is being picked up from, or dropped off to. These objects are called portables.

We have used fingerprints for traversal problems and stacking problems. These were used to identify the appropriate language for the experiments in Section 6.5. The analysis in Chapter 6 required various parameterisations of the solvers, such as whether the graph is static or dynamic can be identified from the domain (Long and Fox, 2000). However, we also introduced several solvers whose appropriateness for a domain will depend on the specific problem distribution. For example, whether identifying hubs or clusters is an important concept in the particular domain. We have associated these with a fingerprint, so that they are provided as alternative vocabulary (for example, clustering is invoked when a transportation fingerprint matches). The reason is that these elements are related to optimising the solution, which requires identifying properties that are specific to the relationships in individual problems, rather than general structural aspects that indicate properties of directed connectivity. In Section G.2 of the appendices, we will examine a specific form of solver parameterisation that uses an empirical method reliant on training data that could be extended to identify a wider range of parameter values. This could be used to determine the appropriateness of clustering by identifying whether the graphs in the training data were suitable for clustering. A more general method would be to provide the vocabulary and let a rule learner select those that are appropriate in practice. We will explore learning RBPs in Chapter 8.

7.1.3 Generic types to the enhanced domain model

In Chapter 5 we defined language enhancements for several sub-problems, including traversal problems and transportation problems. We have demonstrated in this section that these sub-problems can be identified using fingerprints. Our approach is to identify generic behaviours and, where they exist, invoke the associated vocabulary. For example, if there is a transportation sub-problem identified then the problem model is enhanced with the vocabulary that we have selected for transportation problems. In this subsection we explain how the solvers are invoked and explain how we write out the generated problem as a solver listings file and an enhanced domain model file. Thus completing the loop and facilitating automatic generation of a suitable model language.

Parameterising the solvers

We defined solvers in Subsection 5.1.1 as a collection of functions. However, in practice the computation of those functions relies on a collection of parameters. This allows the use of solvers for many different instantiations of the generic behaviour. For example, we have implemented a single graph traversal solver. This solver is parameterised by a particular graph module that is selected based on whether the graph is static, dynamic or turn-based (Section 5.2). Identification of static graphs is presented in Long and Fox (2000). Turn-based graphs are identified by examining the move action: if a proposition that is not a locatedness proposition is changed by the move action and that proposition enables future move actions then the graph is not turned based. For example, a domain where fuel is consumed during traversal is not turned based.

The identification of sub-problems is used to invoke the relevant specialised solvers. The identification process also includes identifying the correct parameterisation of the solver for the particular problem. In the solvers that we have constructed, each required parameter is identifiable with a simple extension of the programs that identify the fingerprints. This is intuitive: a fingerprint distinguishes between problems that the language is suitable for and those that it is not. In order to determine this it requires to analyse the aspects of the problem that the solver works with. For example, the algorithm used to establish a traversing object uncovers the relevant parameters of the located predicate and move action that correspond to the traverser and location arguments. These are precisely the facts necessary for parameterising the move action module that is required by the graph abstraction solver. An extract of the solver listings file for the Driverlog domain is presented in Listings 7.1.

Listing 7.1: Extract from solver listings output for Driverlog domain

```
(:module MoveAction0
  :type solvers.encoding.graphabstraction.moveaction.MoveAction
)
(:moduleDescription MoveAction0
  :MoveAction (:description (drive-truck 0 1 2))
  :Locatedness (:description (at 0 1))
)
```

Writing out solvers

The enhanced domain model requires two files to be created: the enhanced domain model and the solvers listings file. The syntax of these files has been described in Subsection 5.1.3. The enhanced model is partly created using the original domain model. The domain predicates, types and action listings are the same. We query each solver instance for its active predicates and active actions. Each predicate gets an entry in the active predicates list and a new active action entry is made for each action. The solvers that we have constructed have functions for creating appropriate action and predicate names. For example, in Driverlog, the base action drive-truck has a corresponding active action called `long_drive-truck`. The associated condition is called `drive-truck_connected`.

The solvers listings file is built by querying each solver instance for a list of the description strings and modules that parameterise it. Any modules that are depended on are then recursively queried for their descriptions and module dependencies. Suitable entries are made with a tag that describes the parameter and a string. For a description, the string is the description string. For modules, the name of the module is used instead. As we explained in 5.1.3, the type of the solver or module is the Java class that implements its behaviour. The description and module entries combine to fully parameterise the solvers.

7.1.4 Conclusion

In this section we defined a search space for an appropriate problem modelling language. Our approach to selecting chains from this space is based on identifying specific patterns called fingerprints in the problem model. Each language enhancement has an associated fingerprint. If the problem satisfies the fingerprint then the problem structure suggests that the associated enhancing step will be appropriate. An advantage of using a formal approach to matching features is that all labelling is ignored that might otherwise lead to the domain expert failing to recognise the underlying structure (Long and Fox, 2000). For example, the Mystery domain from the 2002 IPC (Long and Fox, 2003) encodes a transportation problem using alternative labels: the locatedness predicate is `craves` and the traverser argument has type `emotion` and the locations have type `food`. The domain satisfies the traversal fingerprint and therefore the traversal vocabulary is invoked.

This approach could be used with several existing approaches to introduce the required model enhancements, such as the `well-placed` predicate, which was manually selected in Khardon (1999a); Levine and Humphreys (2003). An aspect of future work might investigate using a similar approach to determine an appropriate language bias for approaches that rely on rich rule languages, such as Martin and Geffner (2000); Fern et al. (2006), or for those that explore the set of language restriction chains, such as de la Rosa and McIlraith (2011).

We have presented an approach for generating enhanced domain models automatically. The approach is suited to the structure based enhancements that we have investigated in this work. If a fingerprint is matched in a problem then the associated solver can be used with the problem. The information necessary to parameterise the solver is extracted at this matching stage. This process means that a domain expert is not required to select the appropriate solvers and parameters.

7.2 A general model for generating structure interactions

In Chapter 5 we developed several chain steps and in Chapter 6 we demonstrated that these steps were appropriate for providing directed connectivity in several planning problems with underlying structures. During the process of developing these solvers we observed that domains with apparently similar structures required different solutions. For example, in Goldminer problems directed connectivity was established through identifying the nearest closed node in the direction of a target, whereas in Grid problems the solver calculated a route through the locked doors to the goal. The generic type hierarchy in Long and Fox (2002) identifies fourteen types for Transportation problems. A single problem can incorporate several of these types contributing to a large number of possible interacting types. In order to provide the necessary support for the RBP we have developed several solutions to cover different combinations of types from this hierarchy (Chapter5). In general this process requires a knowledge engineer to do a lot of work. The solvers can be parameterised in any way and can be computed in any way. We require a more structured approach if we are going to generate specialised solutions. Our focus in the remainder of this chapter is to reduce the effort of enhancing the problem model.

We return to two observations that were made in Chapters 5 and 6: that SIs underlie the limitations of the rule language and that we should focus on providing concepts of directed connectivity. Combining these observations we conclude that the next step is to develop a model that provides a general approach for defining directed connectivity for SI. An SI is a sequence of actions that act on successive nodes of a structure, contributing towards a single target. We observe that the actions that are applied at each node are often the same or slightly changed. In this section we use this observation to define a *bag of macros*, similar to the sets of macros defined in Botea et al. (2007). We then identify an alternative exploration technique that is more appropriate for use with an RBP.

7.2.1 Arbitrary length macro actions (ALMAs)

Macro actions can encode important subsequences that combine to form a complete behaviour. For example, in Blocksworld problems an `unstack` and `stack` combination can be modelled as a single `move-block` macro action. The problems that were identified in Section 5.2 require that similar tasks are performed on an arbitrary set of the nodes of a structure. For example, in traversing a graph we might move between connected nodes; and in Blocksworld we might uncover a block by repeatedly unstacking blocks that are sitting on top of it. Our model for generating SIs combines these ideas, by generating similar sequences of actions over the nodes in a structure. We first formalise the macro operator and then present our ALMA model.

Macro operator

In Section 2.3 we defined macro actions, which allow the planner to consider a sequence of related actions as a single unit. This can allow the planner to direct search to the goal with fewer decisions. A macro operator is a sequence of operators:

Definition 7.2.1 *A macro operator, mop, is a set of variables,* $\mathbf{V}_{mop} = \{v_0, \ldots, v_m\}$*; and an ordered list of operators,* $op_0, \ldots, op_n$ *over the variables in* $\mathbf{V}_{mop}$.

A substitution, $\Theta = \{c_0 \leftarrow v_0, \ldots, c_n \leftarrow v_n\}, \forall\, j \in [0, \ldots, n]\ v_j \in V_{mop}$, replaces the variables in the macro with problem constants. Application of a macro operator is valid for a given substitution and state, if the action sequence after substitution can be applied to the state. We can define the set of valid bindings to the variables of a macro operator, *mop*, for a given problem and from a specific state, s_0.

Definition 7.2.2

$$\begin{aligned} &\mathbf{MacroBindings}_{mop}(s_0) = \{\Theta | \\ &\quad \exists s_1, \ldots, s_{n+1}, a_0 = \Theta(op_0), \ldots, a_n = \Theta(op_n) \\ &\quad\quad \forall i \in [0, \ldots, n]\ s_{i+1} = \gamma(s_i, a_i)\} \end{aligned}$$

The set of valid actions is constructed by making the substitutions to the macro operator.

Definition 7.2.3

$$\begin{aligned} &\mathbf{MacroActions}_{mop}(s_0) = \{a_0, \ldots, a_n | \\ &\quad \exists \Theta \in \mathbf{MacroBindings}_{mop}(s_0),\ \forall i \in [0, \ldots, n]\ \Theta(op_i) = a_i\} \end{aligned}$$

Bags of macro operators

The use of macro actions is effective because in many problems short sequences of actions combine to perform a single task. Analysis of the plans generated for problems over structures indicate that when acting on a structure we often perform similar tasks on each of a series of nodes. Although the precise details of these tasks might vary, the target of each task is the same, such as moving an object to an adjacent node with the target of moving the object to a particular node.

We gather the behaviours that are relevant for a particular SI together into collections. We represent each of the behaviours as a macro action and group the macros relevant for a particular SI into a set. As an example, consider the target of moving a robot in Goldminer from its current location, $\mathtt{l}_1$, to a different location, $\mathtt{l}_2$. In this example, we assume that the robot is carrying a laser and that the gold is not important. At each individual node we can focus on three possible moves: move the robot into an open square; shoot at soft-rock and move into the square; and shoot at hard-rock and move into the square. Alternative action choices, such as putting the laser down, are not important to the current target. These three moves are each a single task and can be represented by macro operators. We can define a set that gathers these behaviours.

- (`move` *?robby ?l1 ?l2*);
- (`fire-soft` *?robby ?laser ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);
- (`fire-hard` *?robby ?laser ?l1 ?l2*), (`move` *?robby ?l1 ?l2*).

The two fire actions share variables with the subsequent move actions. This means that the operators perform a single task of moving the robot to an adjacent square, perhaps opening the square first. In general we can construct sets of macros that support all of the behaviours that we may require at a particular node. If this is done in the context of a particular target, then we can often reduce the possible action sequences to a useful subset. For example, in the Goldminer problem we did not consider dropping off the laser or picking up the bomb. The fixed length of these macros mean that these sequences are not sufficient to flatten the arbitrary graph structures that we are interested in; however, we are now equipped with a bag of possible behaviours that can be considered at each node. Such a bag of behaviours is a useful building block when considering SIs. The next step is to chain these behaviours together.

Generating SIs from macro bags

Sets of macro operators have been used in previous work (Botea et al., 2007; Newton and Levine, 2010). In these works a single set was defined, which grouped together macros that assisted the planner in reaching the goal. In Botea et al. (2007) no attempt was made to gather macros that would act together, whereas in Newton and Levine (2010) the collaboration of the macros is measured and optimised. In Botea et al. (2007), macro actions are sequenced into arbitrary length action sequences. The motivation is to make as much progress as possible towards the goal while computing as few heuristic estimates as possible. The expansion of the macro bags generates a single greedy action sequence. We also consider linking macros together, however, the motivation behind chaining the actions together is to discover the SIs that can be performed. We therefore consider expanding a much larger number of action sequences.

Focusing application towards specific SIs The set of macro operators provides a collection of action sequences that can be used at each node. In order to establish directed connectivity our aim is to determine the SIs that can be reached by repeatedly applying instantiations of the macro operators in the bag. However, we are interested in focusing this expansion so that it is restricted to a single SI. This means that it is important that we can impose constraints on variable bindings from one macro operator to another. For example, in order to move an object between nodes in traversal problems the moving object parameter of the move actions can be bound to the same object. We therefore introduce a language for joining the macro operators together.

We define a set of binding constraints, B_{mop_0,mop_1}, between each pair of macro operators. This set is appropriate for a macro operator, mop_1, sequenced directly after a macro action, mop_0. The constraints take the form $(v = v')$, $v \in \mathbf{V}_{mop_0}$ and $v' \in \mathbf{V}_{mop_1}$. The interpretation of the constraint $(v = v')$, is that v and v' are unified with the same constant. We define a bag of macro operators as a set of macros that are related through binding constraints.

Definition 7.2.4 *A bag of macro operators is a set:* $\mathbf{macroBag} = mop_0, \ldots, mop_n$ *over distinct variable sets:* $\forall i, j \in [0, \ldots, n]\ i \neq j \implies \forall v \in \mathbf{V}_{mop_i}\ v \notin \mathbf{V}_{mop_j}$, *and the binding constraints,* B_{mop_i,mop_j}*, between each pair of macro operators.*

Our example set of macro actions for nodes of a Goldminer problem can be restricted to focus them towards a single sequence. The binding constraints would enforce that the moves were acted on the same robot and that the destination location of a move action matched the start of the next move. We call this bag of actions the **fireMoveBag** and will refer back to it later.

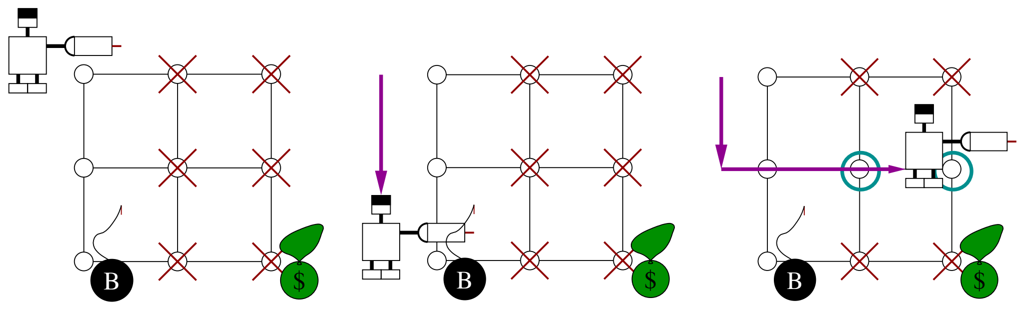
Enumerating the SIs We can think of exploring sequences of macro actions as calculating the connectivity of a graph. The nodes are states and the edges are macro actions. The edge weights are a count of the number of actions in the macro. The connectivity from an initial node can be computed using a best first search (Chapter 2): using a count of the number of actions used so far and choosing the lowest count next. The nodes are tuples: Node = (state, mop, Θ, previous, depth), where mop is the current macro operator; Θ is the binding for the macro; previous is the previous node; and depth is the number of actions in sequence. A node is expanded by applying all macro actions that can be instantiated from the macro operators in the bag of macro operators, **macroBag**. The pseudo-code for this method is presented in Figure 7.2.

```
def expandNode(n) :
    nextNodes = Set<Node>()
    for $mop = op_0, \ldots, op_m \in \mathbf{macroBag}$ :
        for $\Theta$ in $\mathbf{MacroBindings}_{mop}(n.state)$ :
            $a_0 = \Theta(op_0), \ldots, a_m = \Theta(op_m)$
            state = n.state($a_0, \ldots, a_m$)
            depth = n.depth + (m + 1)
            nextNode = Node(state, $mop$, $\Theta$, n, depth)
            if ($\forall (v_0 = v_1) \in B_{n.mop,mop}\ n.\Theta(v_0) = \Theta(v_1)$) :
                nextNodes.add(nextNode)
    return nextNodes
```

Figure 7.2: Pseudo-code for the expand node method of a best first search.

For each binding the macro action is computed and the resultant state is found by applying the action sequence to the current node's state. The depth is found by adding the current node's depth to the length of the macro. Finally, the link between the previous node's substitution and the new node's substitution is validated against the binding constraints. Each pair of variables that are made equal in the binding constraints should unify with the same constant in the substitutions. If this is the case the node is returned.

The states that can be explored will depend on the actions in the bag. An example Goldminer problem state is illustrated in Figure 7.3(a). We can use the move operator as a single step macro operator and define a bag, **moveOnlyBag**, with the move macro as its only contents. The explored states are all states that can be reached using sequences of move actions. This is the states that are connected to the robot's current location by a chain of open locations. Figure 7.3(b) illustrates a state found through a sequence of two move actions. If we allow the macros in the bag **fireMoveBag** (defined above) then we can open locations as well as move into them. This means that exploration reaches states with the robot at nearly all of the locations. The only exception is the square with the gold at it. This square cannot be traversed to as the bag does not include the required `fire-laser` action. Figure 7.3(c) illustrates a state found



(a) Current state (b) Discovered state using **moveOnlyBag** (c) Discovered state using **fireMoveBag**

Figure 7.3: Discovered states using different bags of macro operators.

through interleaving fire and move actions.

When used as intended this search will generate all of the possible SIs for the specified bag of behaviours. The search is started with a single node, n_0 = (s_0, null, null, null, 0). The search terminates when the open list is empty and at the end of search the set of SIs are captured by the tuples in the closed list.

7.2.2 The ALMA directed connectivity solver

This exploration can be wrapped up in a specialised solver, enhancing the problem model with the concepts of directed connectivity for a particular SI. Each specialised solver models a connected predicate and an operator; the modelling of this vocabulary is controlled by the associated bag of macros. The predicate models the reachability of the target from the current state. Depending on the macro application approach, the operator is instantiated to either the actions necessary to complete the parameterised SI, or the first action towards achieving this target.

The vocabulary is parameterised by variables determined by the type of SI. A mapping is required from a node to the parameterisation of the vocabulary. These parameters define the control that the RBP has on the performed SI. For example, in a traversal problem the important information can be represented by three facts: the moving object, and the locations it moved between. The mapping for a traversal problem identifies these parameters from the search node. This extraction provides a set of targets, each with the form, (moverP,fromP,toP). For traversal actions, we extract this information from the first node, n_1 = (s_1,mop_1,Θ_1,n_0,i_1) and the last node, n_m = (s_m,mop_m,Θ_m,n_{m-1},i_m). We define a target tuple, (mover,from,to) as:

- mover = $\Theta_1(mop_1[-1](moverP))$;
- from = $\Theta_1(mop_1[-1](fromP))$;
- to = $\Theta_m(mop_m[-1](toP))$.

We use square brackets to index into the macro action list and negative numbers to indicate that the indexing is from the end of the list. The consequence of this mapping is that the RBP can identify those locations that are reachable using the bag expansion and move the traverser to one of these locations as required.

In structure building problems the targets will be different. For example, in Blocksworld we can define the macro operator that unstacks a block and puts it on the table. This can be used to unstack blocks that are on top of blocks that do not satisfy the goal. **unearthBlock** is the sequence: (`unstack a b`), (`putdown a`). This can be used in a bag and allows chains of `on` relationships to be broken up. The target for this action might be the block that was uncovered by a detach action (Appendix H).

For a given mapping from nodes to parameters, we define the set $\mathbf{TargetSet}_{mop}(s_0)$ as the set of target tuples derived from every node in the closed list. In our example Goldminer problem the two bags, **moveOnlyBag** and **fireMoveBag** would lead to two different sets of targets. We assume for the problem illustrated above that the robot is called robby and the nodes of the grid are named $\mathtt{l}_{i,j}$, with row i and column j.

The target set for **moveOnlyBag** would be:

- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{1,0}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{2,0}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{0,0}$)

The target set for **fireMoveBag** would be:

- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{1,0}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{0,0}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{1,2}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{0,1}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{1,1}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{2,0}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{0,2}$)
- (robby $\mathtt{l}_{0,0}$ $\mathtt{l}_{2,1}$)

For each of the two bags **moveOnlyBag** and **fireMoveBag** we get a predicate and action. For the initial state used in the continuing example, the predicate **moveOnlyBag**$_{connected}$ and **fireMoveBag**$_{connected}$ would hold for each of the target tuples in the respective sets listed above. Similarly, the actions **moveOnlyBag**$_{move}$ and **fireMoveBag**$_{move}$ with one of these targets would make the appropriate move towards achieving that target, depending on the approach.

It should be noted that to exploit this vocabulary using the SbS the bag must be extended with each suffix of the macros in the bag (at least for the first macro application, although we did not make this distinction). This allows the ALMA to be applicable at each step as the executive steps through the actions.

7.2.3 Target structure interactions

The vocabulary can be parameterised by a fixed and finite number of values. This means that the communication from the RBP will be limited to a finite number of world constants. The search as presented distinguishes between all of the possible arbitrary length action sequences (possibly exponential with respect to a particular problem) as distinct SIs, whereas many of them achieve the same target sets. We are providing alternatives for an RBP that can only characterise the arbitrary length sequences using a fixed number of parameters. We therefore introduce a subset of the possible SIs called target SIs, by making the following assumptions:

- A state is sufficient to distinguish between SIs.
- A state is sufficient to determine the set of macros that can be applied. Or in other words, the binding constraints are equally restrictive between any macros.

The first restriction means that our control over the actions that are used in an SI is restricted to the careful selection of the macros in a bag. We do not control the selection of those macro actions in order to achieve a specific SI. This would have implications for planning with preferences over the plan sequence (Gerevini and Long, 2005) and can also lead to dead-ends in certain domains. For example, in Goldminer the bomb is a resource that can only be used once and is intended to be used to free the square with the gold. If the `detonate-bomb` action is part of a bag that is used for an earlier SI then the bomb might be used, leading to a dead-end. The second restriction adds to the first by making it explicit that if we are not interested in the actions that are used to perform an SI then the expansion of a particular macro action should not lead to different constraints on following actions. This is not required by the approach, but simplifies its description and is sufficient in the explored cases.

In the case of target SIs, we can maintain a limited set of visited tuples. Instead of comparing a node with the nodes in the closed list to determine if we should expand the node, we compare the node's state to those of the visited states.

Listing 7.2: Pseudo-code for the add to already processed method for use with a best first search.

```
def alreadyProcessed(n) :
    for closedNode in closedList :
      if closedNode.state.equals(n.state) :
        return true
    return false
```

The change is reflected in Listing 7.2. Once a state is found at a certain layer of search, we do not expand any new nodes that reach the same state. Under the above restrictions an alternative branch will not allow any different states to be explored. We identify a further property in Section 7.5, where we look for an efficient method of searching for targets.

7.2.4 Discussion of ALMAs

In this section we have presented a directed connectivity solver for a wide range of SIs. A benefit to the ALMA representation is that we can conveniently parameterise the computation of the solver by defining a set of macro actions and their associated binding constraints. For example, if we parameterise the solver with the `drive-truck` action from Driverlog and include a binding between the truck parameters of successive macros then we define a solver that can discover sequences of actions that move any truck in a Driverlog problem between any two positions in the link map. Using the **fireMoveBag** defined above, the same solver expands sequences of move actions that can require `fire-laser` actions to open nodes before moving. This provides a convenient method of defining a powerful abstraction layer for empowering an RBP. In this part, we compare our work to the related works.

Establishing an abstract layer

An ALMA is an abstract operator, similar to those exploited in AbNLP (Fox and Long, 1995). In each case we can set goals that require an arbitrary length action sequence and in each case a finite number of constraints can be set to determine how the next layer is generated. In AbNLP the abstraction hierarchy has multiple layers, whereas ALMAs exploit recursion in a single abstraction layer. We discuss a hierarchical implementation of the ALMA solver in Section G.1 of the appendices. ALMAs exploit SI templates, which provide a sensible method of structuring the expansion and have benefits in terms of filtering as presented in Section 7.5.

The motivation behind the ALMA is similar to the *expansion domain*, provided as part of the TLPLAN planner (Bacchus and Ady, 2003), and also to HTN method decomposition. In TLPLAN, the domain model can be expressed using macro operators that are used during planning. An expansion domain allows the macro operators to be expanded in a separate world, so that the plan can be presented in the original language. This has been used to define arbitrary macro actions for truck and driver graph traversal in the Driverlog control knowledge. Operators and not macros are used in the expansion domain (for example, `walk` and `drive` actions). The motivation behind HTNs is that tasks are naturally broken down into smaller tasks. Establishing the raised level of SIs is similar to defining a method decomposition that recursively expands an SI into a combination of macros and each macro as an application of the domain level operators.

Expansion of the space

An aspect of our rule representation is that in order to determine an applicable rule, the rules that are not applicable must be proven false (no binding in the state and goal). If a condition for a rule includes a predicate modelled by an ALMAs then this can require that each of the possible SIs are identified. In addition, an RBP can use the vocabulary in several rules. One reward of expanding the whole space is that it only needs to happen once per state. In Section 7.5, we examine a property of certain SIs that means that we can decrease the number of states that must be visited to determine the reachable SIs.

In Botea et al. (2007) the macro bag is used to generate a single macro action at each search node. The motivation behind the generation of these action sequences is to make as much headway towards the goal as is possible, cheaply, from a single expansion. A similar idea was explored in Lipovetzky and Geffner (2011), where a probe is generated in the direction of the goal from each state. Our motivation is different. We aim to identify all of the targets that can be reached using the actions in the bag so that the RBP can select the next target. Once this selection is made then the approach of Botea et al. (2007) would be appropriate. In particular, if a cheap method of establishing the available targets is available then this approach could be applied to expand the bags to the specified target. For example, in the Driverlog control knowledge, used with TLPLAN, the transitive closure on the `link` predicate is used to identify all of the locations that the truck could visit.

7.3 Automated vocabulary generation

In this and the following sections we consider the problem of generating a parameterisation of an ALMA solver. The space of targets reachable in the expansion of an ALMA is determined by the set of macro actions. The selection of an appropriate set of macro actions will lead to an ALMA solver that models vocabulary that is suitable for controlling SIs in the target domain. In Botea et al. (2007); Newton and Levine (2010), sets of macro actions were learned and the approach in Botea et al. (2007) is particularly relevant as the macro actions chain together to form arbitrary length macros. In both cases the macros were selected using empirical optimisation, which maximised the macros' performance in collaboration (Newton and Levine, 2010), or independently (Botea et al., 2007). We have used an alternative method, where SIs are identified in example plans and generalised to form the ALMA parameterisation.

An SI is a single sequence of actions that contributes to achieve some target (or subgoal). In this section we consider how to extract these action sequences from example plans. Researchers have analysed sub-goals in planning problems in the context of decompositions (Fox and Long, 2001; Hoffmann et al., 2004; Koehler, 1998; Crosby et al., 2013) and caching solution sequences (Laird et al., 1986; Coles and Smith, 2007). We examine extracting sequences for subtasks, however, without the context of planner search information that is exploited in Laird et al. (1986) and Coles and Smith (2007). An alternative approach for identifying the sequences in plans is proposed.

Our approach is inspired by the macro learning approach used in Botea et al. (2007) and more fully discussed in Botea et al. (2005b). However, we take an alternative approach to extracting macros that exploits the domain analysis techniques presented in Section 7.1. The approach splits the problem into two stages. First a series of sequences, or subplans that relate to a particular SI are extracted from a set of example plans. The second stage processes the subplans and identifies a collection of macro actions. As the input is lists of actions that are relevant for a particular SI, the aim of this stage is to divide the sequence into the collections of actions that are observed at a particular node. A related problem has been investigated for domain model acquisition (e.g. Cresswell and Gregory, 2011), where a collection of plans is characterised and represented as a domain model. Our target representation is a set of macro operators, which lends itself to a simpler inference procedure.

7.3.1 Formulation of the problem

We formulate the problem of generating the reachable set of SIs in terms of the ALMA representation presented in Section 7.2. We demonstrated that we can parameterise the ALMA solver's functionality with bags of macro actions and binding constraints. The approach requires that the connections are made between the particular SI and the ALMA expansion. However, we demonstrate that the connection can be expressed at a level that is appropriate for general problem groups. In particular, we define the connection for traversal problems, which includes transportation and path opening problems. We begin by defining our source of examples, the *dependency graph* and the information that must be provided for each SI type. We then present our approach for extracting sequences of actions that are related to achieving a particular target. The final step is to demonstrate how these sequences can be used to parameterise an ALMA solver.

7.3.2 Example plans

We use example plans to guide the action sequences that we consider. An important step is to reduce the sequences to those that are applicable to states that will exist during execution in conventional problem models. We admit that we cannot hope to make any guarantees that the language enhancements will be appropriate for every problem of a domain. In fact, we do not want to make such a vocabulary. The domain conventions often greatly reduce the possible chains of actions. We want to exploit this to ensure the number of macros remains as small as possible. This is achieved through our use of example plans that provide a sample of expected behaviour. If the plans are representative of the expected problem distribution then we expect our solution to generalise to the problems of the distribution.

In this section we rely on examples in the form of problem plan pairs:

$$\langle(\langle\pi_0\rangle, \mathbf{P}_0), \ldots, (\langle\pi_n\rangle, \mathbf{P}_n)\rangle.$$

An example problem, $\mathbf{P}_{Example}$, is illustrated in Figure 7.4 and an example plan, $\langle\pi_{Example}\rangle$, for the problem is presented in Figure 7.5. We use the action sequence representation for convenience.

The quality of the plans will have an effect on the efficiency of the vocabulary. For example, if the plans use a method of enabling traversing that is less efficient than an alternative then the macro actions will lead through similarly inefficient chains of actions. However, if the plans include both good and bad chains then the proposed

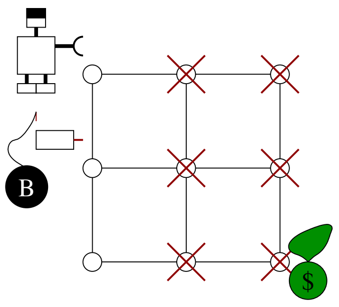


Figure 7.4: $\mathbf{P}_{Example}$, an example problem. The objects are named: robot, `r`; laser, `l`; bomb, `b`; and the location in row, i, and column, j, is called $\texttt{l}_{i,j}$. The goal is to pick up the gold.

1. `move r` $\texttt{l}_{00}$ $\texttt{l}_{10}$
2. `pickup r l` $\texttt{l}_{10}$
3. `move r` $\texttt{l}_{10}$ $\texttt{l}_{20}$
4. `fire r l` $\texttt{l}_{20}$ $\texttt{l}_{21}$
5. `drop r l` $\texttt{l}_{20}$
6. `move r` $\texttt{l}_{20}$ $\texttt{l}_{10}$
7. `pickup r b` $\texttt{l}_{10}$
8. `move r` $\texttt{l}_{10}$ $\texttt{l}_{20}$
9. `move r` $\texttt{l}_{20}$ $\texttt{l}_{21}$
10. `blow r b` $\texttt{l}_{21}$ $\texttt{l}_{22}$
11. `move r` $\texttt{l}_{21}$ $\texttt{l}_{22}$
12. `pickupGold r` $\texttt{l}_{22}$

Figure 7.5: $\langle\pi_{Example}\rangle$, an example plan for $\mathbf{P}_{Example}$.

traversing actions will be similar to the good plans, but the computation of the appropriate sequences will be less efficient as it will expand sequences that include actions from both the good and bad plans. In this work we do not investigate the effect of plan quality on the vocabulary and assume that the plans are optimal.

Dependency graph

Intuitively an action, a, is dependent on a preceding action, a', if the effect of a' allows a to be applied. A dependency graph, or solution graph as it is called in Botea et al. (2005b), generalises this relationship: it describes the dependencies that exist between actions in a specific sequence. In particular, the achievers of an action's preconditions are made explicit. As a consequence of representing the relationship as a graph, we can examine the chains of actions that lead to the action becoming applicable.

A sequence of actions $\langle\pi\rangle = a_0, \ldots, a_n$ is an ordered list of actions that are applicable in a problem model, $\mathbf{P} = \langle \mathbb{M}|_\Sigma, s_i, g\rangle$. The vertex set contains all of the actions (with suitable relabelling where duplicate actions exist in the list): $\mathbf{V} = \{a_j\}$. The edge set connects the action that achieves a proposition with actions that require the proposition to be applicable.

$$\mathbf{E} = \{a_i \rightarrow a_j | \exists p \ \max_i(p \in \text{Adds}(a_i) \ \wedge \ p \in \text{Prec}(a_j) \ \wedge \ i < j)\}.$$

If the proposition holds in s_i, or is not explained by the sequence then there will be no associated edge.

Functions for SI types We assume that SIs can be associated with special actions that actually effect the SI. For example, in a traversal problem, the `move` action moves the traverser. For each of the SI types our approach requires a fingerprint (Section 7.1), a set of *SI action* templates and a set of binding constraints that relate the variables in the templates. There is a slight difference with the binding constraints defined in Section 7.2. In this case we only have a partial structure and therefore the variable constraints are restricted to the variables used in the partial structure. The binding constraints are used to focus the generated language to particular SI sequences. The binding constraints and the *SI actions* are defined at the level of generic types and are therefore applicable to a collection of domains. We assume that the *SI actions* can be identified, as these have been a key aspect of the fingerprints in both of the generic SI problems that we have investigated. We discuss these requirements in Subsection 7.6.1.

The template for traversal domains is the single action template, (`moveAction` *?t ?l1 ?l2*). The binding constraints restricts consecutive templates to move actions on the same traverser and also ensure that the *moved-to* node of one move action is the *move-from* node of the next move action. This template defines traversal problems as sequences of move actions that act on the same traverser.

The templates for the uncover solver for structure building problems defines the template: (`detach` *?b1 ?b2*), (`attach` *?b1*) and the binding constraints accepts two templates where the variable *?b1* is bound to the same value as *?b2* in the proceeding `detach` action. This template defines uncovering problems as sequences of `detach` and `attach` actions that act on the same stack of blocks. An alternative template could be defined for the alternative representation with the atomic move block action. These variations on the representation are identified during the fingerprint identification along with the relevant actions and therefore this information is available to be exploited.

7.3.3 Generating language

We have observed that the graph of nodes that are reachable through isolated SIs can be overly restrictive for planning with an RBP. For example, reachability in the sense of move actions, defined in Section 5.2, is not sufficient. However, there are alternative reachability graphs that incorporate enabling actions that can extend the scope of the reachability analysis. We have introduced three sources that we use to identify the possible ways that an SI can be enabled. Our approach first makes a partitioning of the SI actions and then identifies the non-SI actions that enable the actions in the partitions. This results in a collection of sequences for each type of SI. These sequences can be broken down into a series of individual SI episodes. Each episode fulfills a single SI template with the actions that are required to enable that interaction.

Structure interaction sequences

The first step is to isolate the SI actions of each plan. It is assumed that the actions used to define the templates for an SI are SI actions. Any other action is removed from the plan. The SI actions are then linked together into *template sequences*, which is an ordered set of SI actions that match to a sequence of templates, such that each pair of templates is compatible with the binding constraints. For example, consider a problem with two traversers, `A` and `B` and a package, `p`, and action sequence:

$$(\texttt{move}_A),(\texttt{pickup}_{p,B}),(\texttt{move}_B),(\texttt{move}_A)$$

The first step would be to reduce the sequence to SI templates: $(\texttt{move}_A)$,$(\texttt{move}_B)$, $(\texttt{move}_A)$. A possible template sequence in from this sequence is: $(\texttt{move}_A)$,$(\texttt{move}_A)$, whereas, $(\texttt{move}_A)$,$(\texttt{move}_B)$, is not, because it invalidates the binding constraints.

Definition 7.3.1 *A sequence covering is a partitioning of the SI actions into template sequences that adheres to the following constraints:*

- *Each SI action belongs to exactly one partition.*
- *For every sequence template, $a_{t_0}, \ldots, a_{t_n}$, each of the actions are causally related to the final action, a_{t_n}.*
- *The sequence is consistent with respect to the binding constraints.*
- *There are no partitions that include components that are causally related through another partition.*
- *The partitions are maximal. There are no partitions that could be subsumed by another partition.*

The set of *sequence coverings* defines the alternative ways that the plan could be sequenced given the binding constraints. In both traversal and unstacking problems there is only one partition and therefore this set is easy to compute. In the next stage we assume a partitioning is selected. Each of the coverings could be enumerated and the vocabulary generated for each could be combined. Another approach would be to test the alternative bags using learned control knowledge as a metric (de la Rosa and McIlraith, 2011) and reserve the bag with the best performance. We have not investigated this and leave characterising the structures that would lead to alternative partitionings to future work.

Enablers of structure interaction sequences

The next step is to extend these sequences to discover the actions that enabled them in the example plans. The pseudo-code for our extraction process is presented in Listing 7.3.

Listing 7.3: Pseudo-code for the extract sequences method and the node expansion procedure for the BFS called the stunted dependency tree search.

```
## Inputs:
# $\langle\pi\rangle = a_0, \ldots, a_n$: a plan
## Variables:
# $RCG = (V, E)$: the reversed causal graph for the plan
# visitedIndexes: boolean array for marking plan steps
# $SC = (t_{0,0}, \ldots, t_{0,p}), \ldots, (t_{m,0}, \ldots, t_{m,p})$ is a sequence covering.
## Output:
# sequences: a list of sequences.
def extractSequences($\langle\pi\rangle$,) :
  $RCG$ = createReversedCausalGraph($\langle\pi\rangle$)
  $SC$ = createSequenceCovering($\langle\pi\rangle$, $RCG$)
  orderPartitionsByTheirFinalActions($SC$)
  sequences = []
  for $(t_{k,0}, \ldots, t_{k,p})$ in $SC$ :
    performStuntedDependencyTreeSearch($a_{t_{k,p}}, (t_{k,0}, \ldots, t_{k,p})$)
    $sequence$ = order(gatherAllDiscoveredNodes())
    updateVisitedIndexes($sequence$)
    sequences.add($(t_{k,0}, \ldots, t_{k,p}), sequence$)
  return sequences

def expandNode($n$) :
  neighbours = []
  for $n'$ in $RCG$.getNeighbours($n$) :
    if ! visited[$n'$] :
      if isAnSIAction($n'$)
           & partOfCurrentTemplatePartition($n'$) :
        neighbours.add($n'$)
      else :
        neighbours.add($n'$)
  return neighbours
```

The first stage is to compute a *sequence covering*, as described above. The partitions are ordered by the highest plan index contained in the partition. For each partition a search is carried out from the action in the partition with highest plan index. This searches backwards gathering the enablers for the partition. The discovered nodes are gathered and ordered, the visited nodes are updated and the sequence and its associated *template sequence* are added to a set and ultimately returned.

A stunted breadth first search through the reversed causal graph is used to identify the enablers of the target action. This expansion of a node during this search is pruned in two ways: if an action has already been visited then it is not added as a neighbour; and if the action is an SI action and is not part of the current partition then it is also not considered. The former constraint assigns enablers to a single sequence. This is a heuristic approach that assumes single motivation and was motivated by the observation that if two sequences always share the same enabling sequence in the example plans then we assume that sequences in the future can share the same enabling sequence. The latter constraint acts to break the causal chain into separate sequences of actions for each SI (as defined by the binding constraints and specific partition).

Chunking

ALMA solvers are parameterised by a collection of individual SI episodes. The action sequences that are found using the `extractSequences` function can be broken down into a single SI by breaking the sequence into *chunks*. Each template sequence provides the outline of a specific SI episode. The templates can be used to split the action sequence into a series of *chunks*, each that contain an atomic SI.

The actions in each sequence are ordered using their plan index. The sequence is then partitioned by making a break after the completion of each SI template. For example, the sequence `move`,`move`,`open`,`move`, will be broken down into the three partitions: `move`|`move`|`open`,`move`. The final step is to replace the objects with variables. This is done consistently so that equivalent macros are pruned.

This approach to chunking maintains the ordering observed in the training data. As a result, interweaving between applying actions for sub-goals can lead to actions being partitioned separately from their dependent actions. Consider a domain where the opening of nodes is controlled by a remote terminal that are independent of the traverser's position. Plans for this domain might open several nodes together. Our approach would chunk the opening actions with the first SI episode. This can generate bags that are less susceptible to the optimisation approach that we propose in Section 7.5. A possible improvement would be to extract the partial order implicit in an action sequence and compare the different possible chunks that could be constructed from each linearisation. This would be useful future work, assuming that there are interesting domains that would benefit from it. The presented extraction method is sufficient for the benchmark domains that we have tested.

Chunking a Goldminer plan The Goldminer domain has a single traversal action, `move`, that moves a robot between nodes of a grid. There is also only one robot and therefore a single sequence. The sequence extraction gathers all of the actions except the pickup gold action: all other actions are relevant for moving the robot to the gold square. The sequence can be chunked by splitting it at each move action (the completion of each Traversal template).

- (`move r` l_{00} l_{10})
- (`pickup r l` l_{10}),(`move r` l_{10} l_{20})
- (`fire r l` l_{20} l_{21}),(`drop r l` l_{20}), (`move r` l_{20} l_{10})
- (`pickup r b` l_{10}),(`move r` l_{10} l_{20})
- (`move r` l_{20} l_{21})
- (`detonate r b` l_{21} l_{22}),(`move r` l_{21} l_{22})

Bags

Each of the raised actions is a macro operator. We can reduce these to a set of unique macro operators. This defines a bag that we can use with an ALMA solver.

The above macros collapse into a bag:

- (`move` *?robby ?l1 ?l2*);
- (`pickup` *?robby ?laser ?l1*), (`move` *?robby ?l1 ?l2*);
- (`fire` *?robby ?laser ?l1 ?l2*), (`drop` *?robby ?laser ?l1*), (`move` *?robby ?l1 ?l2*);
- (`blow` *?robby ?bomb ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);

As we use more plans to provide a more complete set of training data the bag will expand. The result over several Goldminer examples is the bag, **allMovesBag**:

- (`move` *?robby ?l1 ?l2*);
- (`fire` *?robby ?laser ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);
- (`fire` *?robby ?laser ?l1 ?l2*), (`drop` *?robby ?laser ?l1*), (`move` *?robby ?l1 ?l2*);
- (`blow` *?robby ?bomb ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);
- (`pickup` *?robby ?laser ?l1*), (`move` *?robby ?l1 ?l2*);
- (`pickup` *?robby ?bomb ?l1*), (`move` *?robby ?l1 ?l2*);
- (`pickup` *?robby ?laser ?l1*), (`fire` *?robby ?laser ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);
- (`pickup` *?robby ?bomb ?l1*), (`blow` *?robby ?bomb ?l1 ?l2*), (`move` *?robby ?l1 ?l2*);
- (`drop` *?robby ?laser ?l1*),(`move` *?robby ?l1 ?l2*).

This bag will lead to a solver that can perform many of the SIs possible in the state space. More generally, this approach can be used to identify a collection of chunks that can be combined to search through the space of SIs.

7.4 Identifying important subsequences

In Section 7.3, bags were generated by considering complete sequences of composable SIs. The result is a comprehensive reachability graph that captures many of the possible SIs. There are two problems with this approach: modelling the propositions and actions is often intractable; and there is little control over the sequence of actions used to perform a specific SI. Closer inspection of the plans indicate that the sequences are not always acting towards the same target. For example, in transportation problems individual portables will each define their own targets. In this section we consider splitting the SIs up to form smaller, more focussed macro bags. The benefit is that it is part of a chain of steps that lead to a language enhancement that can be modelled; the challenge is that we must choose how to split the SIs.

A similar problem has been addressed by researchers investigating approaches for generating HTNs (Hogg et al., 2008; Ilghami et al., 2006; Yang et al., 2007). The first step in Yang et al. (2007) involves partitioning a sequence of actions, and allocating each partition to a task label. In Hogg et al. (2008) and Yang et al. (2007) the task labels are identified for each plan in the training set, whereas in Ilghami et al. (2006) the hierarchy is made explicit in the plan traces.

In this section we present a rule-based process that exploits domain analysis to identify target actions. These rules can be seen as an approach for structuring the training data, providing an alternative to labelling each plan, as has been done for learning HTNs (Hogg et al., 2008; Ilghami et al., 2006; Yang et al., 2007). We use this process to focus the bags so that they are appropriate for achieving a specific type of target. This has two benefits: each bag will have a smaller search space; and the actions will be more appropriate to the target.

7.4.1 Targets

The actions used to perform an SI can depend on the type of target. For example, in Goldminer, the laser is used while opening locations on the path towards the gold, whereas the bomb is used on the gold location. Breaking SI sequences into related parts means that more of the structure of the example plans can be exploited. For example, the example Goldminer plans will only use the bomb on the rocks that cover the gold. If the **allMovesBag** is used then this information is lost and there will be no way of the RBP communicating to the solver that it would like the bomb for a later task. In this subsection we will present the definition and intuition for targets. In the following subsections we present our approach for identifying targets and how those targets are used.

Target actions

Actions that involve an object that interacts with structures can condition on that interaction. An interesting subset of these actions provide the requirement of achieving that interaction. These are the consumers that provide the reason for a particular SI. An example of this, is when a traverser is moved through a series of locations to pick up a package. The traverser must be at the location of the package to pick the package up. We call these actions, which require a specific interaction between an object and a structure, the *target actions*:

Definition 7.4.1 *For a list of actions, $a_0, \ldots, a_n$, the target actions, $\mathbf{targetActions}(a_0, \ldots, a_n)$ are a set of indexes: $i_0, \ldots, i_m$, such that each a_{i_j} is a target action.*

The important aspect of the SI that was performed to enable the target action is called the *target*. For the SIs that we consider in this work, we can define the target as the completing action of an SI template, which supported (or achieved a precondition of) the target action. For example, in a transportation problem a pickup action might be a target action and the location of the package is the target. In this setting the target is a traverser and location pair, with the interpretation that the traverser should be moved to the location. In the following we do not distinguish between the target and the action that achieves it.

7.4.2 Identification of targets

To determine whether a condition that tests an object's current relationship with a structure actually requires that specific relationship, requires some level of interpretation over the action sequence. We present an approach that uses a set of rules to identify the targets within a sequence. Each rule is associated with an SI so that it is activated when appropriate. We focus on the traversal problem; we consider structure building in Appendix H.

Context based targets

The nodes of structures can become significant during planning for various reasons. The node can hold resources that are important for completing the task, for example, the laser in Goldminer, an airport in Logistics, or a particular depot in a Depots problem. This can be an outcome of the problem definition: through an explicit statement in the goal requiring some relationship including the node to be established; or as a consequence of the possible paths that can be made between the initial state and any goal state (for example, landmarks). However, significance can also develop as a dynamic response during planning. In a Transportation problem a transporter might move a package to a location so that it can be picked up by a transporter that is already making deliveries to a similar area. This location becomes an important hub-location as a result of a specific allocation of packages to transporters and because of the relationship between the transporters' relative positions and the package deliveries that are being made.

We have experimented with heuristic approaches of selecting target actions, for example, breaking at an action that adds a proposition that is either a goal, or that is used later in the plan but is not used by actions in the current SI sequence. However, such heuristic approaches are effected by arbitrary orderings in the plan, such as the unnecessary connection or separation of production and consumption propositions. There are many different forms of traversal problem and the appropriate definitions of targets can be different and even contradictory across different problems. There is scope for approximate approaches and this is important future work. In this work, we exploit the existing domain analysis tool, TIM, and establish a mapping from generic types to sequence breaking rules. In this part we formalise the target actions for each of the three types of traversal problems that we have identified.

Graph traversal The main goal in traversal problems is to move an object between different locations. For example, in Goldminer problems the robot should be moved to the location with the gold. There are two more targets that can be important: losing or achieving locatedness. This is when a traverser can be lifted from the graph. Identifying where a traverser loses or acquires its locatedness requires analysing the actions to determine whether the locatedness predicate is only added or removed.

In its most pure sense, a traversal goal is to have the traverser positioned at a specific node in the graph. However, this generalises to moving a traverser to achieve goals in a more general setting. For example, in the Goldminer example above there is no goal for the robot to be at a location. However, the robot must still move to a specific location to pickup the gold and achieve the goal. We want to capture this idea that goals can lead indirectly to target locations.

As a proxy for identifying these implicit sub-goals we use a rule that identifies goal achieving actions that condition on the location of the traverser. These actions are further analysed to determine whether the location and the goal variables share the same variable partition. In theory a richer set of conditions could be identified, however, this identifies the goal achieving actions in the benchmarks.

Transportation In transportation problems the traverser is moved to pickup and drop off packages. Each pickup and drop off is an individual target that requires that the traverser is moved. In some cases, the drop off will be to achieve a goal and is therefore covered by the previous rules. However, there are situations where an object will be dropped off at a non-goal location. For example, in Logistics problems an object could be dropped off at an airport. The pickup and drop off actions are identified as part of the fingerprint analysis. The action sequence is broken at any of these actions.

Path opening Path opening problems involve the traverser moving to locations to open them and moving to pickup objects to enable opening locations. We propose two rules: the first breaks the sequence if a new enabler is picked up; the second breaks the sequence at the end of a sequence of opening actions. We do not break when an enabling object is dropped off.

We define a sequence of opening actions as a series of opening actions that ends when there is a change of enablers. Such a sequence may contain moves and other enabling actions; however, the sequence is broken at the last opening action prior to an enabler being dropped off (or picked up). This provides the potential to generalise over chains of opening actions.

The transportation drop off rule is overwritten if an object is used as an enabler while it is carried. However, the traverser rule is not. This means that the sequence is broken if a package is used as an enabler and then dropped off at its goal; however, it is not broken if it is dropped off at a non-goal location. This is consistent with our assumption of single motivation.

In summary the rules that we use for traversal problems are:

1. Move to goal : a goal achieving action is enabled by move action

2. Lose or gain locatedness

3. Pickup object

4. Drop off package

5. End of an opening episode (where the current enablers were sufficient for the traversal task)

7.4.3 Exploiting targets

The rules are used to alter the *sequence covering* partitioning defined in 7.3. The rules are applied to each step of the plan in the context of each of the partitions. During this process an ordered list of target actions is computed for each partition. The partitions are then split into subsequences that each contribute towards one of these target actions. This process is achieved for a particular target action and associated partition, by searching through a reversed causal graph from the target action until the end of the partition's SI template is found. This action is made the last action in a new partition, splitting the sequence into parts. The sequence extracting method is applied to these smaller partitions. The result is a collection of smaller bags that are appropriate for achieving specific targets.

A worked example

We use the example illustrated in Figure 7.4 to make the application of the rules more clear. The purpose of Goldminer problems is to open up a path so that the robot can pickup the gold. The only action that is not relevant to moving is the final pickup action. There are usually four targets in Goldminer problems: pickup the laser; fire to one location from the gold; pickup the bomb; pickup the gold. The problem in Figure 7.4 can be solved with the following steps:

1. move r l_{00} l_{10}
2. pickup r l l_{10}^{3}
3. move r l_{10} l_{20}
4. fire r l l_{20} l_{21}^{5}
5. drop r l l_{20}
6. move r l_{20} l_{10}
7. pickup r b l_{10}^{3}
8. move r l_{10} l_{20}
9. move r l_{20} l_{21}
10. blow r b l_{21} l_{22}
11. move r l_{21} l_{22}^{1}
12. pickupGold r l_{22}

The target actions are marked in red with the breaking rule index in superscript. Action 5 will be removed from its sequence as it does not enable anything in its sequence.

As the grid grows in size the number of actions between targets increases. For example, the laser might be located four steps away from the robot's starting location. The length of these sequences are determined by the problem instance and are therefore of arbitrary size. For example, Figure 7.6 illustrates the sequence of actions that move the robot to reach a square away from the gold. The general form of the resulting sequence is:

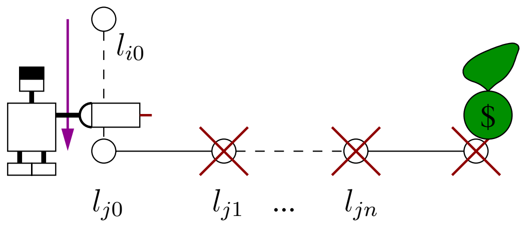


Figure 7.6: The robot moves through a sequence of moves then fire move pairs.

$$\texttt{move}\;\; r\, l_{i0}\, l_{(i+1)0}$$
$$\ldots$$
$$\texttt{move}\;\; r\, l_{(j-1)0}\, l_{j0}$$
$$\texttt{fire}\;\; r\, l\, l_{j0}\, l_{j1}, \texttt{move}\;\; r\, l_{j0}\, l_{j1}$$
$$\ldots$$
$$\texttt{fire}\;\; r\, l\, l_{j(n-1)}\, l_{jn}, \texttt{move}\;\; r\, l_{j(n-1)}\, l_{jn}$$

This sequence is chunked into a single node SI.

$$\begin{gathered}(\texttt{move}\quad r\, l_{i0}\, l_{(i+1)0})\\ \ldots\\ (\texttt{move}\quad r\, l_{(j-1)0}\, l_{j0})\\ (\texttt{fire}\quad r\, l\, l_{j0}\, l_{j1}, \texttt{move}\quad r\, l_{j0}\, l_{j1})\\ \ldots\\ (\texttt{fire}\quad r\, l\, l_{j(n-1)}\, l_{jn}, \texttt{move}\quad r\, l_{j(n-1)}\, l_{jn})\end{gathered}$$

We have not distinguished between `hard` and `soft` rock in the actions above to simplify the presentation. The process derives an equivalent bag to the **fireMoveBag** that we defined in Subsection 7.2:

- (move ?robby ?l1 ?l2);
- (fire-soft ?robby ?laser ?l1 ?l2), (move ?robby ?l1 ?l2);
- (fire-hard ?robby ?laser ?l1 ?l2), (move ?robby ?l1 ?l2).

7.5 Bag expansion pruning

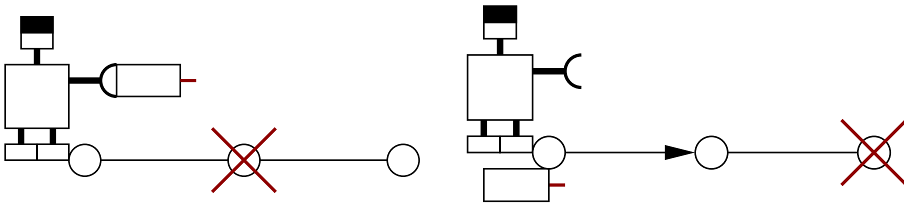
There is an important practical aspect that we have ignored. In particular, to evaluate the reachability of SIs requires the expansion of a search space that can grow exponentially with the nodes in the reachability graph. There are SI that define tractable spaces. For example, traversal in various transportation problems does not require enabling actions. However, in other problems, for example in Goldminer problems, the macros allow us to visit a large number of the states. Exploring this space is intractable even for small grids.

We observe that many of the states that are discovered are equivalent with respect to the parameters set by the RBP. In particular, the main task is to discover the targets that are reachable, not which states are reachable. Therefore, pruning can be used during the expansion, as long as the set of targets that are discovered is the same. In this section we introduce the problem. We propose a solution framework and a criteria that determines whether pruning retains completeness. However, it is not practical to assess the criteria completely and an approximation approach is presented in Section G.2 of the appendices. Similar limitations have been addressed for approximate filtering approaches by repeating the planning process on failure with the filter turned off (as in FF).

7.5.1 Bag significance

The first observation that we make is that there are many accepted templates that are equivalent in terms of the SI that they perform. For example, a traverser may move between two points using many paths. The control afforded to the RBP is limited and cannot characterise the particular SI. We therefore consider that two SIs are equivalent if the targets are the same. An important extension to this is that if we know that two states will lead to the same targets being discovered then only one of those states need to be expanded. We are therefore interested in identify the part of the state that is significant for allowing different SIs. This means that if the significant parts of two states are the same then we only need to extend one of them, reducing the search space.

This is the problem of finding a suitable projection from a state into some reduced space, such that those states that are equivalent under the projection extend to the same targets. This property is naturally dependent on a specific macro bag. We consider a projection as *bag significant*, if the projection is sufficient for any state of a domain.



(a) The robot's position is the only part of the state that is important in determining the locations that can be reached.

(b) With these sequences, the holding laser proposition is important for enabling locations to be reached.

Figure 7.7: Two examples of states in the Goldminer domain.

To illustrate this concept we use the Goldminer examples, illustrated in Figure 7.7. In Figure 7.7(a), the location of the robot is sufficient to determine whether the problem is solvable and solved. To move the robot to the middle location the robot must destroy the rock. If it puts down the laser then the problem is still solvable: the robot must simply pick the laser back up again. Once at the second location, the position of the laser is irrelevant. In this example, a projection to the robot's position would be sufficient to determine the states that are reachable. However, in Figure 7.7(b) when the robot moves to the middle location there are two distinct possibilities. If the laser is in the robot's hand then the rock can be destroyed and the robot can be moved to the right-most location. If the laser has been left behind the goal cannot be achieved. In particular, if search pruned the state with the laser in the robot's hand then we would lose the opportunity to move the robot to the right-most location. Therefore the proposition that models whether the robot holds the laser provides necessary information that splits the two exploration chains[1].

A significance set based exploration

The search needs to be changed in order to perform the search pruning. This is achieved by changing the `alreadyProcessed` function to search in the closed list for the projected state as the current node (Listing 7.4). The pseudo-code relies on the function, `project`, which provides the projection of the state.

Listing 7.4: Pseudo-code for the already processed method in a best first search.

```
def alreadyProcessed(node) :
    sigNState = project(node.state)
    for closedNode in closedList :
      sigCNState = project(closedNode.state)
      if sigCNState.equals(sigNState) :
        return true
    return false
```

The search already pruned states that had been observed previously. These changes mean that we now prune states if we have already observed the projection of the state.

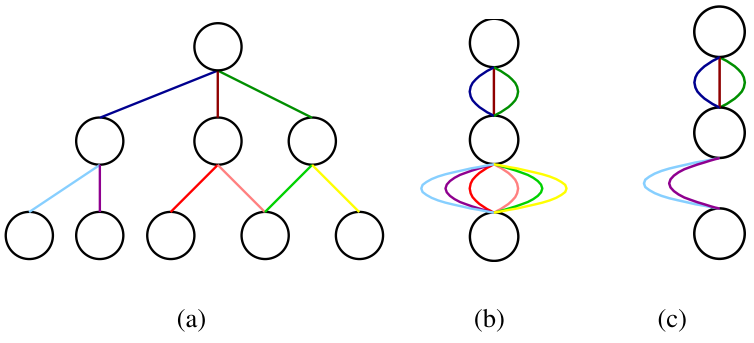


Figure 7.8: Example of bag expansion using 7.8(a) state indexing; 7.8(b) the result when using a projection; and 7.8(c) the expansion when exploring once from each state that is distinct under the projection.

The benefit of using the projection based pruning is demonstrated in Figure 7.8. The reachability graphs defined by ALMAs are potentially large (Figure 7.8(a)). How-

[1]The proposition that determines whether the laser is at the left-location provides just as much information here. However, this proposition will not always be useful.

ever, when that space is projected to the set of propositions that are important in the context then some of the states will be equivalent under the projection (Figure 7.8(b)). We have observed that it is common in SI problems that sets of states extend to reach the same set of targets. We exploit this by expanding a single state from each of the equivalent projected states.

Exploration completeness

We consider that the process is still complete if the projection, q, leads to the same targets being found for any state from any problem in the domain.

Definition 7.5.1 *For a given bag of macros, mop; for the function $\mathbf{TargetSet}_{mop}(s, j))$ that computes the possible targets for the given state and projection, q; and where $\mathbf{TargetSet}_{mop}(s, I)$ is the set of targets using the identity projection, I, then q is bag significant if the following property holds:*

$$\begin{aligned}
&\mathbf{BagSignificant}(q) \iff \\
&\quad \forall \mathbf{P} \in \mathbb{D} \\
&\qquad \forall s \;\; \mathbf{P} \models s \\
&\qquad\quad \forall t \; (t \in \mathbf{TargetSet}_{mop}(s, I) \iff t \in \mathbf{TargetSet}_{mop}(s, q))
\end{aligned}$$

The ideal situation is that any two states that extend to the same set of targets are equivalent under the projection. We have selected to use the projection from states to the target space. This is the minimal target space, as the states must be distinguished by at least the achieved targets.

Definition 7.5.2 *A bag of macros, mop, is target significant if the projection from states to the target space is bag significant.*

7.6 Discussion

In this chapter we have investigated the automatic inference of an appropriate domain model for supporting an RBP in search. Through the use of domain analysis the underlying SI problems can be inferred and appropriate language extensions can be instantiated from a library. In practice we have found that this approach requires various specialised solutions to be implemented, each with a limited scope. We have explored this issue in the context of directed connectivity and proposed a generalised solution for capturing directed connectivity vocabulary. This model (the ALMA) provides a convenient method for expressing domain specific control knowledge. We have developed an approach that parameterises the ALMAs appropriately for a particular domain, using example plans. In this section we discuss our approach, how the problem compares to those tackled in the literature, and we discuss the contribution in terms of previous work.

7.6.1 A summary of the ALMAGen approach

In summary, the definition of an SI requires the following elements:

- A fingerprint that identifies the key elements of the SI (Section 7.1).
- A set of templates that outline the valid SI episodes.
- A set of binding constraints between each pair of templates.
- A set of target identifying rules.

In this chapter we have provided a method of learning the parameters for an ALMA solver. In particular, we have transferred the effort of developing a solution for specific subtypes of SI problems to the problem of defining the elements stated above. We provide a first step towards automating the process of learning specialised solvers. In this subsection we discuss the required elements and the expected effort of providing them.

Defining a fingerprint that makes an exact divide between domains that are suitable for the language enhancement and those that are not could require careful analysis and extensive testing or a formal proof. However, the language extensions are intended to provide the pool of vocabulary that can be used to express the RBP. If the vocabulary is not useful then it will not be used. The advantage of accurately defining the fingerprints is to reduce the size of the vocabulary pool. We will discuss this in Chapter 8.

Analysing the SI is necessary to define a fingerprint and this level of understanding requires knowledge of the common shapes that such interactions will take. Moreover the identification of the important actions is likely to be vital for the fingerprint analysis. This suggests that little added effort will be required to establish the templates and binding constraints. In the two examples of SI that we have investigated there are only one or two direct SI actions.

The last element is the set of rules that identifies the targets. This requires more involvement. We analyse the impact that an under specified rule set will have on the vocabulary in the Chapter 9. The main difference that the rules make is in the time it takes to compute the vocabulary, however, it can lead to dead-ends in problems with certain types of resource. This means that providing a method of identifying rules is rewarding; however, we can hope to provide some useful vocabulary even with a limited set of rules. Trial and error can be used, as the identified targets for the example plans can guide the process. As a result, we have replaced the effort of defining a fully parameterised solver implementation to the problem of identifying target actions. We continue this discussion in Section 10.2.

We observed when implementing specialised solvers that a new domain would often require a new solution, or at least a new feature to our existing solution. This is not surprising because the benchmark planning domains are intended to provide a diverse spread of problems. However, it suggests that a default solution that works in the domains with a particular SI with little extra effort would be a valuable resource. In this section we have presented an approach that provides this general solution. Where efficiency is important for a specific type of SI then a solution can be tailored and linked in with a fingerprint, or the language presented in Section 5.1. In this way, effort is focussed where it is necessary.

7.6.2 Exploiting domain analysis

There are several approaches that have exploited domain analysis. Many of these were efficiency measures used to infer the type structure in problems (Fox and Long, 1998), or state invariants (Fox and Long, 1998; Gerevini and Schubert, 1998). In Hoffmann (2011) the domain is analysed in an attempt to uncover whether the relaxed planning graph will be an effective heuristic. In Armano et al. (2003), the sets of preconditions and add and delete effects of each operators were analysed in order to establish chains of operators that were compatible for forming macro operators. Domain invariants can then be exploited to constrain the variable bindings of these operator chains, leading to effective macro operators (Armano et al., 2005). Our approach exploits a deeper domain analysis; following the approach that was carried out for the planner, HybridSTAN. HybridSTAN uses the analysis to identify appropriate decompositions that can be applied to the problem model. The analysis has been used to select appropriate heuristics (Fox and Long, 2001), improve the estimate of the relaxed plan heuristic (Coles and Smith, 2006) and for selecting appropriate control knowledge Murray

(2002); Murray et al. (2003).

In Botea et al. (2005a), the static propositions of problems are used to uncover structural components. These components are used as support for making macro actions from sequences of actions that are enabled by the component. The intuition is that actions that are enabled by some underlying structure are acting together and can be packaged into a single option. The focus is on small finite sized components and arbitrary graph structures are actively avoided. This approach can lead to macros that capture the behaviour at each node of an SI; however, the planner would be responsible for individually navigating the structure.

7.6.3 Generating abstractions

Our ALMA representation provides a convenient packaging for supporting RBPs in SIs. The reachability graph is defined in terms of macro operations and we provide two views of the structure: the reachable SIs and a greedy next step towards achieving an SI. This establishes an important middle level of reasoning between the local targets and the global RBP strategy. A key aspect is that by including all of the relevant factors in the ALMA solver expansion space, we can solve challenging SI sub-problems. In fact, exploiting ALMAs from an RBP can be interpreted as combining rule based and hierarchical decomposition approaches, as proposed in Bacchus and Kabanza (2000).

In this chapter we have examined the problem of identifying hierarchical structure in plan traces, which is related to problems in the field of grammar induction (De la Higuera, 2010). For example, the identification of target nodes and chunks are related to identifying valid expansions of non-terminal nodes in a grammar. In our approach, we have exploited the rich structures uncovered by the domain analysis and this has provided an alternative solution without the requirement of labelled plan traces. Examining how hierarchies can be generated from the plan traces directly, has been investigated for learning decomposition networks, in planning.

The process of learning a decomposition network can be split into two parts: the identification of the hierarchy of tasks and the learning of appropriate methods for decomposing those tasks. In Ilghami et al. (2006), the methods are learned for a hierarchy of tasks, by building a set of methods incrementally as more task decompositions are observed. This approach relies on hand-tailored training examples that specify the decomposition tree of tasks. In Yang et al. (2007), the training data includes the ordered tasks and the plan trace, however, the mapping between tasks and actions is not explicit. The approach assumes no knowledge of the effects of actions. An approach based on expectation-maximisation is used to cluster the actions around the most appropriate task labels and these clusters are developed to form recursive method networks. This is an alternative approach to sequencing that requires each plan in the training set to be associated with the tasks that are completed by the plan. We provide an alternative, where a set of rules are defined for a general class of problem (such as Transportation problems) and an alternative representation to the labels are generated automatically for each plan. Through using targets and because we do assume that the effects and conditions of actions are explicit, we can extract the actions used to achieve the targets using causality. Our approach for detecting generic types depends on the analysis of the causal relationships between operators and therefore our approach relies on this information. However, within a generative approach to discovering SIs, the use of clustering could be used to determine sequences for use in ALMAs.

In Hogg et al. (2008), action sequences are used to incrementally build a hierarchy of methods. This approach relies on a set of annotated tasks, which corresponds to the description of targets, although we only consider a single layer decomposition. However, it is assumed that the methods will be used to generate the same length sequences in the future: the method preconditions are constructed through regression, similar to approaches for generating preconditions for macro actions. Our precondition cannot be represented in predicate logic as it requires the transitive closure of the expanded graph. The benefit of using regression is that the network is guaranteed to solve the problems that were presented in the training data. In contrast, whether our generated language is useful can depend on the interaction of subtasks.

There has been research investigating program-like languages as a representation for solving sets of problems (Srivastava et al., 2008; Winner and Veloso, 2007). In particular, loop structures support repetition of plan fragments on different object sets. The main focus in these works is on capturing a pattern of behaviour and then applying the pattern with unbounded collections of objects. The approaches used to uncover these patterns are limited. In LoopDISTILL (Winner and Veloso, 2007), the sequence of steps in each loop are identical and loops cannot be nested. For example, if a plan was learned for the simple transportation problem then the plan might loop through picking up a package, moving, then dropping off the package again. However, if the truck moved to a location with two packages it would still only pick up one; if there were no packages then the plan would fail. The set of loops identified in Srivastava et al. (2008) do not necessarily consist of identical steps; however, the conditions are restricted and only a subset of the non-nested loops are captured. These forms of looped behaviour are generalised by the inherent form of recursion in the RBP representation. The main challenge that we have addressed is not the problem of iterating through a series of similar scenarios; instead it is the problem of directing the progression through those iterations so that they contribute to a particular target. For example, moving the truck a step towards a package.

There is a wealth of literature that has investigated generating model abstractions as macro actions (Botea et al., 2007; Minton, 1985; Newton and Levine, 2010; Coles and Smith, 2007; Iba, 1989; Minton, 1985). A common approach is to select macros from a set of example plans, by ranking the candidates after using filtering approaches, measuring the macros by performance and frequency of use (Botea et al., 2005b; Minton, 1985). In Newton and Levine (2010), a genetic algorithm is used to select macros from the candidate space that improve the planning time. The motivation for these approaches is to select macro actions that positively effect the balance between branching factor and search depth and is therefore a general approach for improving the planner's efficiency. In Iba (1989); Coles and Smith (2007), a more specific observation is used to select macro actions. When the planner escapes a local minimum in the heuristic landscape, the planner memorises the sequence that it generated so that the sequence can be attempted in future local minima. We have also used observations to tackle a weakness in our chosen planner, however, we are compensating for a limitation in the planner, instead of improving efficiency through removing redundancies. In Botea et al. (2007) an arbitrary action sequence is generated from a set of macro actions. However, the approach used to generate the macro set is not optimised to discover composable macro actions. The approach in CONSTANCE (Gregory et al., 2010) is to identify a specific relationship between a subset of the actions in a problem and if it exists, generate macro actions to generalise the actions. In this approach the planner is forced to select targets, and therefore the model is not enhanced, but constrained. Whereas we establish a general approach for providing enhancements, in CONSTANCE a single relationship is focussed on, as this leads to a specific method of optimisation. Researchers have also investigated decomposing the problem into components. In Crosby et al. (2013), planning problems are decomposed into private and shared components. Planning in the private components can be conducted cheaply as the actions in these components cannot interfere with other parts of the problem. The shared components require careful coordination as their actions impact on other components. This work shares similar motivations. We are interested in separating the problem into a collection of independent SIs that are managed by the RBP towards achieving the high level goals.

7.6.4 Learning domain vocabulary

In Martin and Geffner (2000); Fern et al. (2006), the process of problem model selection and control knowledge generation is not split into two stages. Rich rule languages are used and useful vocabulary is learned during the process of learning control knowledge. As a consequence the vocabulary is specific to the domain and useful in rule learning. Our approach generates specific vocabulary, however, there is no guarantee that it can be used in learned control knowledge. We investigate this aspect empirically in Chapter 9.

Our rules can represent concepts that cannot be represented in the rich rule languages. Although these concepts could be supported by enhancing the rule language, this will make the learning problem larger. Our framework allows additional language steps to be stored in a library and presented as an option when appropriate. The model is only bloated with extra vocabulary when it is likely that it will be useful for expressing control in the domain.

An alternative approach was investigated in de la Rosa and McIlraith (2011). The approach involves a beam search in the chain of state enriching languages, which only includes additional predicates. A particular language bias was selected, which limits the derived predicates considered. The authors select three operations that define the neighbourhood of their search: the combining of two predicates; the abstraction of a single variable of a predicate; and a transitive closure on a binary predicate with the same typed arguments. The evaluation is carried out by learning a set of control knowledge using the enriched language. As a result, the approach biases the exploration of chains towards the weaknesses of the learning algorithm and the representation. In contrast we have observed common features of a set of domains that suggest certain weaknesses. We provide language enhancements that target those weaknesses directly. Our vocabulary is generated for a particular purpose and therefore can be labelled appropriately for that role, whereas the vocabulary in de la Rosa and McIlraith (2011) is the result of various operations on arbitrary predicates and its function can be difficult to comprehend. These works both share similar aims, but each lead to different language enhancements. An advantage to the approach in de la Rosa and McIlraith (2011) is its general applicability. In contrast, our main advantage is the sophistication of the strategies that are introduced through our language extensions. We leave using these approaches in combination as future work.

The main advantage of our approach over previous work is that the solvers used to enhance the problem models of a domain are selected specifically for the domain. The analysis of the structure leads to a small pool of candidate enhancements.

7.6.5 Conclusion

Our intended application involves using the vocabulary in the expression of RBPs, although we have experimented with exploiting ALMAs in heuristic search (Lindsay, 2012). In many cases, the rule language is expressive enough to select the class of target that should be satisfied next. From the target an appropriate SI can be selected. However, it can be impossible to identify the next interaction in the direction of this target (as was discussed in Chapter 5). This is the reason for exploiting specialised vocabulary. In this context, the main benefit of splitting the interaction into target achieving sequences is that if the interaction is part of a larger problem then the rule system has control at a lower level. The rule system manages the interactions between the aspects of the problems.

In conclusion, we have presented a method of parameterising an ALMA. As a consequence, we can generate appropriate vocabulary for expressing rule based policies in structure based problems. The approach that we have proposed relies on the expansion of a potentially exponential search space. We have proposed a filtering approach that can greatly reduce the number of nodes and edges that must be explored. We believe that the definition of the ALMA is an important contribution to work on decompositions for planning. In particular, the division of planning problems into smaller components and the selection of appropriate control options within each component. There are several limitations in our investigations. We will continue the discussion of these in the future work, Section 10.2.

CHAPTER 8

AUTOMATING POLICY ACQUISITION

The final step in automating the process is to learn the RBPs themselves. Several approaches have been applied to this problem and learned effective policies. However, none of these approaches have learned policies that provide guidance through the SIs that we have looked at in this work. As a consequence learning in these domains has not been fully explored. In examining these problems we have discovered that the class of fitness functions that have been developed to guide learning in previous approaches is inappropriate for learning in these domains.

In Khardon (1999a), it is observed that the states that a forward chaining planner acts on are a product of the decisions made in previous states. However, the training data used in that work, and other related works (Martin and Geffner, 2000; Levine and Humphreys, 2003; Yoon et al., 2002), is composed of a set of predetermined states. This means that the condition of a rule is not only required to capture a correct choice, but also model the behaviour of the planner used to generate the states. We define an alternative fitness function that uses observations of the policy during planning so that its fitness is related to its performance in solving problems.

The approach we present is computationally expensive. The learner applies planning as a black box and uses empirical analysis to direct search. This approach largely ignores the rich structures implicit in the domain model. In Aler et al. (1998), the authors investigate methods of incorporating knowledge into the learning process. The authors observe the opportunity of using general learning approaches to complete and refine existing knowledge. We use one of the presented approaches, which seeds the initial population. There are approaches that investigate explaining action sequences using several different representations (Boutilier et al., 2001; Hogg et al., 2008; Srivastava et al., 2011; Cresswell and Gregory, 2011), and we have used these approaches for inspiration, to develop a process that generates a set of RBPs from a collection of plans. This is realised through an extension of the ALMA generator presented in Chapter 7. These RBPs demonstrate the use of the enhanced vocabulary and provide building blocks for the learner to modify.

In this chapter we begin by presenting the RBP learning approaches. We identify a limitation in the used fitness functions and then present the fitness function we have developed to address this limitation. In the final section we present our RBP generation process.

8.1 Learning policies

Learning rules has interested researchers for many years. We have introduced PRODIGY and SOAR in Section 2.3, and these systems were used as test-beds for learning rules for improving search performance. Other approaches have aimed to learn planners, representing them as RBPs, or HTNs.

There are three main models that have been investigated for learning RBPs. The first approaches build the policy from highest priority rule to lowest, continuing until all of the training examples have been covered by the policy (Khardon, 1999a; Martin and Geffner, 2000; Yoon et al., 2002). In Levine and Humphreys (2003); Galea et al. (2009), learning a policy is modelled as an optimisation problem over the policy space. It has been observed that weak rules in an RBP will greatly impact how effective it is in practice (Xu et al., 2009). More recent approaches have aimed to reduce this impact by incorporating the exploitation of rules within a general search strategy (Yoon et al., 2006; de la Rosa et al., 2008). Recent reviews in the field provide the broader context for our study: in Jiménez et al. (2012), the authors present an overview of planning and learning, while in Kambhampati and Yoon (2010), the authors focus on the use of explanation based learning. In Chapter 7, we analysed the literature surrounding learning macro actions and HTNs. In this section, we focus on work related to learning RBPs.

8.1.1 Learning rules

In Khardon (1999a); Martin and Geffner (2000); Yoon et al. (2002), a learning approach is used that extends Rivest's decision list learner (Rivest, 1987). Rules are selected incrementally from a set of candidates based on how they match with a training set. As the RBP is resolved in priority order the training examples that have been covered by rules can be removed from the training set. This process concludes when the training set is empty and the rules cover all of the examples. In Khardon (1999a) it is shown that a polynomial algorithm exists for learning RBPs, although this only partially transfers: it holds for chains of state enriching languages, and abstracting actions using the SbS.

Rule selection

The candidate rules are drawn from the action and predicate symbols of the domain model and a limited set of variables. The possible predicates and actions can be enumerated by matching their parameters with any possible combination of the variables. Any such predicate, p, generates several possible terms, for example, (`not` (p)) and (`goal` (p)), depending on the rule language (we discussed the rule languages in Chapter 4). In Khardon (1999a); Levine and Humphreys (2003); de la Rosa and McIlraith (2011), the predicates of the domain model are extended with derived predicates. Our approach also extends the action set with actions from the enhanced model.

The approach taken to identify the individual rules of the decision list differs between the approaches. In Khardon (1999a); Martin and Geffner (2000), all of the terms (bounded by a maximum number of variables and predicates) are enumerated for each action and one that covers a subset of the examples correctly is chosen. Of these, the rule that covers most examples is selected. A collection of selection criteria were explored in Khardon (1999b). An alternative approach was explored in Yoon et al. (2002), where instead of enumerating all of the possible terms, the condition is incrementally generated using greedy (or beam) search. Although this approach loses the bounded completeness guarantees brought by enumeration, more complex conditions can be discovered that would be beyond the scope of enumeration. An alternative approach is presented in de la Rosa and McIlraith (2011), where a dedicated library is exploited to learn first order formulae to explain the training data. In Roller (de la Rosa et al., 2008), the problem has been encoded as a set of classification problems. The latter two approaches allow *off-the-shelf* technologies to be exploited.

Training data

The training data used in L2PLAN are sets of m pairs, $\langle \mathbf{P}_i, sc_i \rangle$, with problem, $\mathbf{P}_i = \langle \tau_i, init_i, g_i \rangle$, and scoring function, $sc_i : \mathbf{A} \mapsto \{0, 1, 2\}$, defined for actions applicable in state, $init_i$. Each applicable action is mapped to 0, 1 or 2; this corresponds to the number of steps over optimal that the length of the best resulting plan after choosing the action. We assume that the scoring functions are computed using optimal plans. This distance is capped at two, whereas in principal this score could be infinite. Similar training sets are used in Martin and Geffner (2000); Yoon et al. (2002), except in their formulations no distinction is made between different sub-optimal solutions. In de la Rosa et al. (2008); de la Rosa and McIlraith (2011), the actions are divided into positive examples (the optimal choices) and negative examples (otherwise). In Khardon (1999a), a single plan is generated (the *teacher*'s solution) and the scoring function rewards an action if it matches the plan action.

To make the training set for a domain, a set of problems is obtained using the domain generator. A selection of parameters are used, so that the generated problems represent the problems of the domain, however, the possible parameters must be relatively small so that optimal solutions can be generated. Optimal solutions are expected in most approaches (Martin and Geffner, 2000; Yoon et al., 2002; Levine and Humphreys, 2003); in de la Rosa et al. (2008); de la Rosa and McIlraith (2011) an approximation approach is used to discover near-optimal examples. Each problem, $\mathbf{P} = \langle \tau, i, g \rangle$, is solved and each of the states, s_j, visited along the generated plan forms a new problem, $\mathbf{P}_j = \langle \tau, s_j, g \rangle$, in the training set. For each of the applicable actions, a, in each of the initial states of these problems, the mapping is computed by comparing the plan length from the state after applying the optimal step, with the plan length after applying the applicable action, a.

8.1.2 Learning rule based policies

Various approaches to learning policies have been investigated, including dynamic programming (Boutilier et al., 2001), incremental covering of training data (Khardon, 1999a; Martin and Geffner, 2000), beam search (Yoon et al., 2002), and setting it as an optimisation (Levine and Humphreys, 2003) or classification (de la Rosa et al., 2008) problem. In Levine and Humphreys (2003), the problem of policy learning is posed as an optimisation problem. A solution to this problem has been implemented in L2PLAN. It was reported in Galea et al. (2009) that optimising over rule order leads to RBPs that generate shorter plans than Rivest inspired algorithms. This has motivated our selection of L2PLAN for learning RBPs in this work. We will now overview the L2PLAN approach.

L2PLAN: a policy learner

L2PLAN uses a genetic algorithm and local search hybrid approach to evolve generations of policy populations. We introduced genetic algorithms and local search in 2.4. The initial population is generated randomly by sampling rules from the set of candidates described above for learning rules. Each subsequent population is made by applying genetic program operators to policies in the previous population. Policies are selected from the previous population using a selection process that favours better policies, with respect to a given fitness function. The operators are randomly selected with probabilities set as parameters. The fitness functions in L2PLAN includes evaluating the policy using example states. A restricted local search is applied to each of the candidates in the population. The algorithm pseudo-code is outlined, for a complete presentation refer to Levine and Humphreys (2003); Galea et al. (2009).

```
def L2Plan() :
  population = initialisePopulation
  n = len(population)
  while (continueCondition(population)) :
    newPopulation = []
    for i in 1..n :
      if (random() < crossoverLevel) :
        π = makeCrossoverChild(population)
      else :
        π = makeReplicatedChild(population)
      if (random() < mutationLevel) :
        applyMutationOperator(π)
      newPopulation.add(π)

    applyLocalSearch(newPopulation)
    population = newPopulation
```

Fitness function

The fitness function value for a policy, (or a policy's fitness,) is computed by applying the policy to each of the initial states of the training data problems and looking up the resulting action in the associated scoring function. These scores are inverted and averaged so that the fitness score is between 0 and 1, where 0 indicates weak fitness and 1 is the highest fitness. The fitness of a policy, π, can be computed using the following formula.

$$\delta(\pi) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{1 + sc_i(\pi(\mathbf{P}_i))}$$

The aim of the L2PLAN algorithm is to solve an optimisation problem, as defined in Section 2.4, where the target is to find a candidate, π, that maximises the value, $\delta(\pi)$.

The genetic program operators

In Chapter 2.4, we introduced the crossover, mutation and selection operators for genetic algorithms. L2PLAN also relies on a replicate operator that copies a policy from the previous population to the next population. As the candidates of the populations in L2PLAN are more structured, these operators have been specialised for this application. RBPs have two basic levels: the rule level and the policy level, and each of these is treated separately using a collection of mutator and crossover operators.

The motivation behind the crossover operator is to share partial solutions throughout the population with the expectation that *good blocks* can benefit the population as a whole. Each crossover operator takes two policies, selected from the population with the selection operator and performs crossover using the following approaches:

- Rule list crossover: select a point in each policies' rule list and create two new policies. Each taking the first part of one policy and the second part of the other.
- Rule swap crossover: select a rule from each policy and swap them over.
- Same action condition swap: for two rules with the same associated action, perform crossover on the predicates in the rule condition.

Mutation makes a small change to the solution that is intended to prevent search from getting stuck in local optima. L2PLAN uses 5 mutation operators, split into rule and policy levels. The policy operators are:

- Rule addition/deletion: add or remove a rule from the policy.

The rule operators are:

- Add/remove a predicate from condition;
- Create a new condition for a rule.

The selection of candidates uses Tournament Selection [1], which heavily biases policies with higher fitness. The approach is parameterised by a value t that determines

[1]A generalisation of the approach proposed by Art Wetzel (unpublished) to n-ary tournaments was used. The interested reader can refer to Mitchell (1998).

the size of each tournament. A random sampling with replacement of t candidates is selected and the fittest of these policies is selected. Large values of t will result in a heavy selection bias towards policies with high fitness.

Local search

After a new population has been produced using replicate, crossover and mutation operators, local search is applied to each policy. The local search procedure is parameterised by width and depth parameters that control how completely the approach is applied. A set of neighbourhoods is used, allowing the space to be explored through direct changes to the variables, types, conditions and rules.

In L2PLAN the neighbourhoods are required to generate single neighbours on demand. The width parameter, w, determines the number of neighbours that are generated at each search step. Each of these w candidates is evaluated and the best identified. Search is stopped if none of the w generated neighbours improve the previous score. The depth parameter, d, determines how many levels are explored during a single application of local search, assuming continued improvement. The neighbourhoods that were used in L2PLAN are the rule operators used for mutation.

8.1.3 Integrating rules with search

The negative impact caused by a small number of poorly specified rules has led researchers investigating approaches that reduce their impact. Decision trees can be learned that order the successor states of a depth first search (de la Rosa et al., 2008), or first order formulae with modal operators used to prune successors in a similar setting (de la Rosa and McIlraith, 2011). In Yoon et al. (2006); Xu et al. (2007, 2009), learned rules are used to determine a corrective value that is used to reduce the weaknesses of the h^{FF} heuristic. In Yoon et al. (2006) the rule sets are interpreted as sets of weighted features and linear regression is used to learn the weightings that best explain the difference in heuristic estimate and real value of the distance to goal from the states of a training set. Although we have developed a system that supports utilising control knowledge and domain independent heuristics, we do not experiment with learning supported control knowledge. Instead we focus on learning RBPs that can control search directly.

8.2 Learning rule based policies in optimisation problems

In Khardon (1999a), it was observed that the states that a forward chaining planner acts on are a product of the decisions made in previous states. However, in that work RBPs were learned using state action pairs derived from a remote planner. The authors argue that as the generated policies can often explain the examples then these states are appropriate. Moreover, similar approaches have been used in subsequent research and have demonstrated that the training data is suitable for learning in several domains (Khardon, 1999a; Martin and Geffner, 2000; Yoon et al., 2002; Fern et al., 2006; de la Rosa et al., 2008; de la Rosa and McIlraith, 2011). However, as soon as the

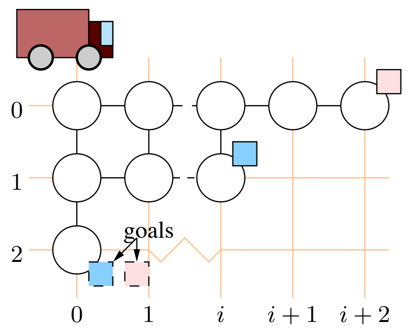


Figure 8.1: A series of transportation problems with any number of pairs of connected nodes between column 1 and i. The pink package is at location $\mathtt{l}_{0,i+2}$ and the blue package is at $\mathtt{l}_{1,i}$. Both should be delivered to $\mathtt{l}_{2,0}$.

learner is presented with a set of examples that cannot be explained with the language bias and parameter restrictions then this approach becomes ineffective. In Chapter 5, we discussed the limitation of RBPs in the context of optimisation problems. It is hardly surprising that learning an RBP that can evaluate the relative benefits of various actions is difficult. Consider the series of transportation problems suggested by Figure 8.1. In these problems the optimal solution involves picking up the pink package then the blue package and delivering them to their goal. Each of the first i training examples derived for this problem would give the highest score for moving towards the pink package and the lowest score for moving in any other direction. This means that a policy that could be used to solve this problem using a slightly longer strategy (picking up the blue package first) would be penalised in each of these states. The control knowledge of TLPLAN used in the Driverlog domain during the 2002 IPC, relied on a nearest neighbour heuristic (and the same strategy was used in HybridSTAN for the transportation specialised solver). In our example, the blue package is the nearest neighbour in each of the first $i + 1$ steps. As a result, the strategy used by TLPLAN would be severely penalised in the first $i + 1$ states of this problem. This strategy contributed towards TLPLAN winning the prize for hand-crafted planner at two IPCs, and the current fitness functions would mark it as a poor strategy.

As a result the learner will often explain moving examples using a move action with an empty condition, as the language bias is leading to penalised scores. It will slowly cover more of the examples by utilising features that happen to isolate a single example in the given problems. For example, a rule might be learned that determines whether there is a package goal and driver goal at the same location as in one example this happens to be a useful package to pickup first. Although this rule set, through a mix of convenient action ordering and over-fitted conditions, might lead to high scores on the training data, the resulting rules will not capture a strategy for solving problems. As soon as the policy is used in a problem from a different distribution it will fail.

One of the problems is that the policy is represented in a limited language that does not necessarily allow it to succinctly explain the examples. The learner attempts to maximise the fitness of the solutions and this leads to over-fitted examples. The landscape of the fitness function is not consistent with our expectations. Instead what we would like is a fitness function that maximised the number of problems that can be solved from start to finish and within this group of solutions look for those that generally lead to shorter plans.

8.3 An appropriate fitness function for RBP learning

In previous work, the posed learning problem determined whether a chosen RBP representation could capture the strategies implicit in a set of training data (Khardon, 1999a; Martin and Geffner, 2000; Yoon et al., 2002; de la Rosa et al., 2008; de la Rosa and McIlraith, 2011). The intention was to determine whether the representation can capture a strategy that leads an executive to the goal. However, when the training set cannot be explained then using an RBP representation becomes ineffective (Xu et al., 2009).

In this section we present a fitness function that evaluates the use of the RBPs in search. The policies are rolled out on a set of problems and their choices at each state are recorded and used to evaluate their performance. In this way the policy is evaluated in the same context that it will be used and the states that it is tested on are a direct result of its strategy. The learning effort is focussed on learning a policy that captures a strategy for solving problems. In this section we detail the features that we think are important in determining success in policy execution; we describe our approach for extracting the features; and present how we have implemented the fitness function.

8.3.1 A fitness function for problem solving

The aim of a fitness function is to map solutions to scores in such a way that the defined partial ordering over the solution space is consistent with an expected ordering. It is therefore our aim to establish what the expected ordering might be in the context of RBPs and the analysis that must be carried out on each RBP in order to determine how it compares with respect to alternatives.

The important factors for a planner are that it can solve problems and that it can solve them in few steps. This introduces two competing ordering criteria:

- If a policy gets closer to solving a problem than another policy then it should be considered a *fitter* policy.
- If a policy gets to a certain distance (steps in an optimal path) from the goal in fewer steps, then it should be considered a *fitter* policy.

There are many more aspects that we could consider, such as planning time. However, we consider these two to be the fundamental considerations for planners. We continue by defining the features and how the features are combined to compute an ordering function. We assume a training set consisting of problems from some distribution; we will discuss how we have generated this below.

Features

Each of the m problems in the training set is of the form, $P_i = \langle init_i, g_i, \Sigma_i \rangle$. A policy, π_j, unrolled through $\omega \times \mathbf{optSoln}(init_i)$ transitions through $\omega \times \mathbf{optSoln}(init_i) + 1$ states, where $\mathbf{optSoln}(s)$ is the optimal plan length from s. Each of these states, s_{ik}, can be posed as a problem, $P_{ijk} = \langle s_{ijk}, g_i, \Sigma_i \rangle$ and solved optimally to find the state's distance to goal, q_{ijk}. It is not expected that a learned policy will solve all problems optimally and we include a parameter, ω, to control the number of steps that the policy is unrolled. This parameter is used as a multiplier on the distance to goal of the initial state of the problem, so that the number of states is related to the size of the problem. The distance from goal measurements provide a trace of how the policy performed on each problem. For a particular problem and policy we use two features: how close the policy managed to get to achieving the goal; and how many steps were made by the policy that made progress towards the goal.

$$\begin{aligned} \mathbf{CP}_{ij} = \text{closestPoint}_{ij} &= \underset{k}{\operatorname{argmin}}\{q_{ijk}\} \\ \text{numberOfImprovingSteps}_{ij} &= |\{(q_{ijk}, q_{ij(k+1)} | q_{ijk} > q_{ij(k+1)})\}| \end{aligned}$$

From these features we derive three measures over all problems: the first two sum the features over the problem sets; the third measure sums the position of the closest point over the problems.

$$\text{closestPointScore}_j = 1 - \frac{1}{n}\sum_i \frac{q_{ij(\mathbf{CP}_{ij})}}{q_{ij0}} \tag{8.1}$$

$$\text{improvingSteps}_j = \frac{1}{n}\sum_i \frac{\text{numberOfImprovingSteps}_{ij}}{\text{MaxSteps}_i} \tag{8.2}$$

$$\text{earliestClosestPoint}_j = 1 - \frac{1}{n}\sum_i \frac{\mathbf{CP}_{ij}}{\omega \times q_{ij0}} \tag{8.3}$$

The first measure promotes policies that move towards the goals of problems. The second measure promotes policies that move towards the goal in more states, with the intention that these policies may be represented by useful rules that are still developing. The third measure supports those policies that achieve their closest state in fewer steps. This should lead to a preference to policies that solve problems with fewer steps. The form of measure (8.3) is appropriate for the way we combine the measures, however, in other cases there might be a more appropriate form.

Combining features

In Aler et al. (2000b), the authors observe that setting weights between different aspects of a fitness function is a challenging issue. The problem of combining measures can be achieved using Pareto Optimality (Deb, 2001), which acts to maintain a population that is spread across the different aspects. However, in this work we use a tiered approach, as in Aler et al. (2000b), that relies on a complete ordering over the measures. To compare two policies the measures are used in order, if the policies are equivalent under the current measure then the next measure is used and so on. If the measure is different then this ordering determines the order between the policies. This heavily biases the comparison between policies to earlier measures.

We would prefer a planner that can solve more problems and in general, we are more interested in planning performance aspects than how the solution is represented. We use closest point and earliest closest point scores as the first and second measures, followed by the number of improving steps. The main priority is to search for a rule that reaches as close to the goal as possible. After coverage the second aim is to find a policy that moves towards the goal efficiently and measure (8.3) is suitable for this. It was expected that the improving steps metric would encourage search to find useful rules and therefore assist the primary aim, however, this was not observed in practice.

Inspired by the *compacity* and *generality* components used in Aler et al. (2001), we added three additional tiers to the fitness function. These are intended to provide more guidance to the learner in terms of the conciseness and application of the knowledge. The following lists the complete fitness criteria used. For two policies the first criteria that is different was used to determine the fitter policy.

- Comparison of closest points over training set
- Comparison of how many steps were required to reach the closest point
- Comparison of the number of improving steps selected by policy
- Comparison of number of extra parameters
- Comparison of number of conditions
- Comparison of number of rules

In each of the final three criteria smaller numbers are considered better.

Problem distribution

As we are unrolling the policy through several steps, we could use problems from the problem generator. A good policy would have to exhibit effective problem solving in order to score a high score. However, a weak policy will be evaluated on a sequence of increasingly irrelevant steps. In fact, the policy will often fail after one or two steps as generated rules are often not applicable to observed states. This approach forces an ordering over how the parts of the solution are learned, as the policy would only be evaluated on initial states.

To reduce this effect, we use a collection of problems that include states encountered during search. The process used is similar to the example generation for L2PLAN.

The first stage is to generate a set of representative problems as described above. Each of these are solved and each of the visited states is turned into a problem. The training set is formed by taking the initial problems and adding a sample of the goal path problems. This training data allows the fitness function to be sensitive to improvements to any part of the strategy.

8.3.2 Implementation

We have implemented the fitness function as an extension to L2PLAN. For each run a folder structure is created with a problem folder for each of the m problems. Within each of these folders there is a policy folder for each of the n policies as well as the problem file associated with that problem folder. For an evaluation of q policies, the policies are written out into the first q policy folders of each of the m problem folders. The policies within a problem folder are unrolled on the associated problem. This is realised as a collection of external commands to the program that was used in Chapter 6 to evaluate the policies. These processes can be run in parallel. For a particular problem and policy the unrolling process outputs the states visited during the unroll into the policy folder. At this stage we read all of the generated states into the main program. This allows us to maintain a cache of visited states. A list of the states that have not been visited is created and h^{LM-cut} is used to solve each of these problems. Once again these can be parallelised, however, as h^{LM-cut} relies on writing temporary files, it is important to run this process from different folders. The resulting output from h^{LM-cut} is parsed and the length of the plan is cached along with the associated state (and goal). This process is illustrated in Figure 8.2.

The main parameters of our system are the training data; the multiplicative factor, ω, which determines the number of steps that the policy is unrolled; and the number of seconds that an external processes was run before it was killed. Maintaining a lookup table of previous solutions has led to far fewer planning steps. There are various measurements that could be stored as information for removing entries from the table, such as the number of iterations since last lookup and how long the plan took to compute. However, during our experiments the cache was manageable.

8.3.3 Discussion

We have introduced a fitness function that addresses several issues in the L2PLAN approach. The fitness function will guide search towards RBPs that direct search towards the goal. In domains with SIs, exploiting the enhanced language is a convenient

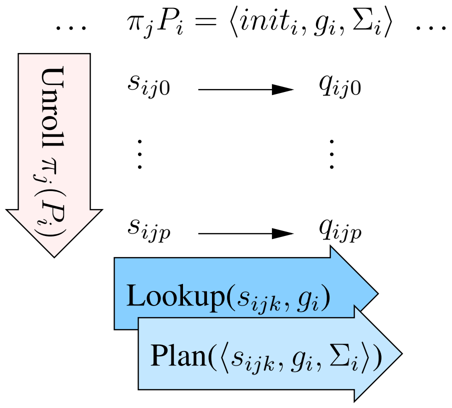

$\ldots \quad \pi_j P_i = \langle init_i, g_i, \Sigma_i \rangle \quad \ldots$
Unroll $\pi_j(P_i)$
$s_{ij0} \rightarrow q_{ij0}$
$s_{ijp} \rightarrow q_{ijp}$
Lookup(s_{ijk}, g_i)
Plan($\langle s_{ijk}, g_i, \Sigma_i \rangle$)

Figure 8.2: For a given policy, π_j and problem, $\mathbf{P}_i$, π_j is unrolled through states, $s_{ij0}, \ldots, s_{ijp}$. The distance to goal of each state is discovered either from the cache, or by running h^{LM-cut} from the state.

method of providing this guidance. Moreover, the addition of control knowledge representation biasing features into the fitness function, as reported Aler et al. (2001), promote using this vocabulary over carefully matching each visited state during rollout (that is, over fitting). In L2PLAN, each of the plans in the training data was created by a planner distinct from the learned RBPs and the biases that underlie the computation of the enriched propositions. This can lead to conflict between the decision made in the modelled vocabulary and the decisions implicit in the example plans. The training data can include other artefacts that are caused by the consistency of the planner that was used to generate the data; similar observations are made in Xu et al. (2009). These aspects can be addressed individually, however, our approach provides a unified solution.

Genetic Algorithms have been used in planning to learn plans (Westerberg and Levine, 2000), control knowledge (Aler et al., 2001), macro actions (Newton and Levine, 2010) and RBPs (Levine and Humphreys, 2003). The fitness function in Westerberg and Levine (2000) is used to learn plans, and attempts to maximise the number of achieved goals and applicable actions in the plan, although similar fitness functions have included other aspects, such as a count of unsupported facts (Gerevini et al., 2003). For macro actions and control rules, the performance in planning has been used to evaluate the knowledge. The key difference in these approaches is that the authors assume a planner up front and seek to improve it. In Aler et al. (2001), the following criteria are used to assess the rule set performance: the number of problems solved by expanding fewer nodes than without the rules; the number of problems solved; total number of nodes. These criteria provide little guidance for learning a complete RBP. As noted above several criteria are included to guide the construction of efficient knowledge. These include reducing the number of variables and the number of rules. We extend our fitness function with similar properties in the evaluation. The criteria in Newton and Levine (2010) uses the product of its measures, rather than using a priority ordering over them. The fitness function is focussed towards how long planning takes, whereas in our approach we assume that an RBP that can solve problems in few steps and is expressed concisely will plan efficiently. As we have observed, the fitness function for learning RBPs uses example states generated by a planner and fails to direct search to effective RBPs in the domains we consider in this work.

The main benefit of our approach is that the policy is evaluated on chains of states that it discovers, rather than on predetermined states biased by a planner. The fitness function guides the selection of policies to those that demonstrate better planning performance. Another approach would be to measure the final state after rolling-out a certain number of steps and this measurement could even be estimated using a heuristic. For example, in Fern et al. (2004), a bounded policy roll-out is used to estimate the fitness function score and if the policy fails to make the goal then a heuristic is used to estimate the remaining distance. However, there is no guarantees that the final point is the closest to the goal, allowing good parts to be missed by the fitness function and introducing inconsistency between its policy ordering. The fitness function developed here is informed by each decision made by the policy and therefore can provide low level guidance. Using a sub-optimal planner or a heuristic estimate at each policy step is potential related work.

The fitness function that we have defined is suitable for searching in solution space. It is not appropriate for guiding search for individual rules, which is required for approaches that incrementally build policies. An alternative approach would be required where the partial policy was evaluated, which would be more computationally expensive. It has been observed that incrementally learning policies with vocabulary that includes abstract actions results in exponential learning time Khardon (1999a), losing the benefits of the Rivest style learning approach (Rivest, 1987). There are alternative search strategies for solving optimisation problems that could exploit our fitness function. For example, local search and simulated annealing are effective approaches to optimisation, which can be less computationally expensive.

8.4 A plan to policy translation

The learning approach is general and is initialised on a collection of solutions that are unlikely to capture much of the required structure. In Aler et al. (1998) the utility of biasing the learning approach with existing knowledge is demonstrated. The authors promote the idea of using evolutionary approaches to complete knowledge rather than fully evolve it. We propose a translation of related plans into an RBP, with the expectation that the RBPs generated in this way will capture certain structures that are exploitable in other planning problems. We first describe the relationships that exist between the RBP and a plan by establishing a mapping between a single action sequence in Σ_0. In the second step we use the enriched model to generalise the approach to a set of plans. We focus on the enhancements provided by the ALMA solver and the resource management solver as demonstrations. We complete the section by considering the limitations of the approach and how these could be addressed.

8.4.1 A translation from plan to policy $(\mathrm{M}|_{\Sigma_0})$

There are fundamental differences between action sequences and rules. The actions in a sequence are linked together through the shared use of world constants and an underlying causal structure. In contrast, each rule is a separate formula, with its own set of variables. The implication of this is that the condition of each rule must establish the relationships of the sets of constants that can unify with its variables, while ensuring that the current state in the world is appropriate for the rule to fire. The advantage of this generality is that several plan steps can be represented by the same rule.

There are two main questions that a rule condition must imply: can the associated action be applied? and should the associated action be applied? Approaches to macro composition provide the solution for encoding an answer to the first question. The latter question requires ensuring that there are incomplete sub-goals in the state. In this subsection we develop a process for translating an individual plan step into a rule.

Training data

In this section we rely on a set of training examples, $\mathbf{TD}$, where each example, $e_i \in \mathbf{TD}$, is a problem and plan pair, $(\mathbf{P_i}(\Sigma_0), \langle\pi_i\rangle(\Sigma_0))$. We also use, a single handed Gripper problem, $\mathbf{P}_{Gripper}$, and solution, $\langle\pi_{Gripper}\rangle$, presented in Figure 8.3 to demonstrate the process. The problem involves one misplaced ball and a robot with one gripper in the destination location, the goal is to move the ball home.

Initial State

- `in ball room1`
- `at herbert room2`
- `free gripper`
- `attached herbert gripper`

Goal

- `in ball room2`

Plan

1. `(move room2 room1 herbert)`
2. `(pickup ball room1 gripper herbert)`
3. `(move room1 room2 herbert)`
4. `(drop ball room2 gripper herbert)`

Figure 8.3: Initial state, goal and solution for a gripper problem.

Testing that the action sequence is modelled by the state

Composing actions is used so that macro actions can be expressed in PDDL (e.g. Newton et al., 2007). However, it can be used to construct a formula that accepts states that will become goal states under the action sequence. These formulae can be trivially generalised by replacing the constants with variables (an approach used to construct macro operators).

The regression formula defines the propositions that hold before an action is applied and is computed by removing the add effects and adding the precondition:

$$\text{rregr}(a, s) = (s \backslash \text{Adds}(a)) \,\cup\, \text{Prec}(a)$$

The propositions that must hold to support the action sequence from action, a_i, are calculated by regressing the actions from the last action back down to the i^{th} action, starting with the goal propositions (the propositions that we know hold in the final state):

$$\phi_i = \text{rregr}(a_i, (\dots \text{rregr}(a_{n-1}, g) \dots)$$

We label this ϕ to distinguish from the set of all propositions modelled by the state.

For example, in the Gripper example, the third set counting back from the end, $\phi_{gripper_1}$, is the set:

{`(in ball room1)(at herbert room1)(attached herbert gripper)`}

These propositions are modelled in a state if-and-only-if the state models the action sequence. In this example, the three actions `(pickup ball room1 gripper` `herbert`), `(move herbert room1 room2)` and `(drop ball room2 gripper herbert)`, rely on exactly the propositions defined by $\phi_{gripper_1}$. Intuitively the condition, ϕ_i, captures the feasibility of using the last i steps of the action sequence to reach the goal.

This can be used to create rules for picking up balls in the Gripper domain. The goal of the problem is used to initialise the formula and the rule for step, i, uses the action that was applied at that step. For example, the third rule can be derived as shown below. The world constants have been generalised by replacing each new world constant with a fresh variable of the same type. Renaming has been done for presentation.

```
(:rule GripperRule_RegressedStateCondition
    :condition (and (in ?ball ?r1) (at ?rob ?r1) (attached ?rob ?gripper))
    :goalCondition (and (in ?ball ?r2))
    :action pickup ?ball ?r1 ?gripper ?rob)
```

Testing that the action sequence should be applied

Although this rule captures a useful set of situations, its condition is too general. In particular, the condition does not capture the reason to apply the action sequence. This explanation comes from propositions that are not modelled in the current state, such as goals that are not already achieved. For example, the rule could pick a ball up that is at its goal already. The important distinction is ensuring that the sub-goals achieved by the sequence are to be achieved in the current state. These propositions can be expressed using negative conditions in the rule language.

Our approach to creating the negated propositions in the condition is to gather a pool of candidate propositions that will be achieved at some point in the later steps of the plan. The propositions are then tested and if they are not modelled in the current state then they are used as negative conditions. From a particular step along the action sequence, the pool of propositions is generated from the remainder of the action sequence. This pool is recursively gathered, adding the add effects and preconditions of each action.

$$\mathbf{Pot}_n = \{g\}$$
$$\mathbf{Pot}_i = \text{Adds}(a_i) \cup \text{Prec}(a_i) \cup \mathbf{Pot}_{i+1}$$

The potential set, $\mathbf{Pot}_{Gripper_1}$, for the first step of the Gripper example is:

{`(in ball room2)(holding ball gripper)(at herbert room2)`
`(attached herbert gripper)(in ball room1)(at herbert room1)`}

Although the regression approach provides an important restricted view that determines the relevant propositions, it is important to use the complete state to evaluate the potential negative conditions. As we have relied on before, the state after j steps is computed as follows:

$$s_j = \gamma(\ldots(\gamma(s_{\text{init}}, a_0), \ldots), a_{j-1})$$

$\mathbf{Pot}_i$, defines the set of propositions that will play a role in subsequent steps of the plan. We test each proposition in the potential set, $\mathbf{Pot}_i$, to determine whether it holds in the current state, s_i. If it is not part of the current state then we consider it an unachieved sub-goal and add it to the negative conditions. It should be noted that if the regressed set was used for determining negative conditions then we could add propositions that held in the current state; this would prevent the generated rule applying to the current state. In the Gripper example, the negative condition is determined by identifying whether the potential propositions ($\mathbf{Pot}_{gripper_1}$) are not in the state ($s_{gripper_1}$).

$$\begin{aligned} \text{negativeCondition}_{gripper_1} &= \mathbf{Pot}_{gripper_1} \backslash s_{gripper_1} \\ &= \{(\texttt{in ball room2})(\texttt{holding ball gripper}) \\ &\quad (\texttt{at herbert room2})\} \end{aligned}$$

The proposition, `in ball room2`, does not hold in the current state. If it did hold then the action sequence that picks the ball up and moves it to the other room would be inappropriate. The negative conditions identify the elements in the state that are still to be achieved.

Maintaining the all-different property

A single collection of variables is generated for each rule by replacing each world constant with a variable. Special treatment is required so that more than one variable cannot be unified with the same constant and change the interpretation of the rule. The problem models are extended with vocabulary to model the not-equal propositions between objects. For each unordered pair of variables, ($?v_1$, $?v_2$), the proposition, ($!=$ $?v_1$ $?v_2$), can be added to the rule condition. In practice we only add propositions for pairs that could unify with the same constant, that is, pairs of objects that have related types. For example, a variable type `ball` and a variable of type `robot` cannot unify with the same objects and the not-equal proposition would be redundant.

Creating a rule

Each rule is constructed by combining the positive propositions and the all-different propositions with the negative propositions and using the plan action and problem goal. We have already given an example of this process above, but here we make the process more concrete.

The state condition of the i^{th} rule is constructed making a positive condition for each of the propositions in ϕ_i and negated conditions for each proposition in `negativeConditions`$_i$ and replacing the constants with the appropriate variables. Similarly, the goal propositions are derived by replacing the constants in the problem goal with variables and the action is derived by replacing the constants in the plan action. The complete set of variables is used to create not-equal predicates that are added to the positive condition.

For example, in the Gripper problem the third rule is derived as follows:

```
(:rule GripperRule1
   :condition (and (in ?ball ?r1) (at ?rob ?r1) (attached ?rob ?gripper)
                (!= ?r1 ?r2)
      (not (in ?ball r2)) (not (holding ?ball ?gripper)) (not (at ?rob ?r2)))
   :goalCondition (and (in ?ball ?r2))
   :action pickup ?ball ?r1 ?gripper ?rob)
```

This rule captures the important aspects necessary to determine whether the pickup action should be used. The negative proposition (`not (holding` *?ball1 ?gripper*)) is redundant [1] because states with the ball in the room are mutex with states with the ball being held. Conditions like this could be detected and removed using a tool that uncovers mutex relationships, such as TIM. Similarly, only one of (`not (at` *?rob ?r2*)), (`!=` *?r1 ?r2*) and (`not (in` *?ball ?r2*)) is necessary. However, the rule condition is satisfied in states where the rule is a useful choice and the redundancy in this example is a product of its simple relationships.

Policy construction

The rule ordering is important because the rules are evaluated in order. We order the rules in reverse order so that the first rule is the rule created from the final plan step. The justification for this is that if we can apply the i^{th} rule in the current state we assume that in the next state the $i + 1^{th}$ rule will be applicable and so on until the goal is achieved. This is similar to the n-stage-to-go value function derived in the MDP

[1] Of course, there are states in the domain that model both propositions, however, not in the state spaces that adhere to the domain conventions.

literature (Boutilier et al., 2001; Gretton and Thiébaux, 2004). The approach learns a set of rules that capture a particular sequence of actions and the propositions of a state that are necessary for the application of the action sequence.

8.4.2 An extended translation ($\mathrm{M}|_{\Sigma_i}$)

In the model presented in Chapter 3 there are three forms that enhancement steps take: an action that abstracts an action sequence; a predicate that provides a richer interpretation; and a set of decision based propositions and the actions that effect them. We develop a process for extending the policy generation process to incorporate each of these enhancement steps.

Listing 8.1: Pseudo-code for policy generation.

```
## Inputs:
# TD: the set of plan, problem pairs (training data)
## Fields:
# planGroup: plans that are equivalent in $\Sigma_i$
# refPlan: one plan from the equivalent group
# planMaps: mappings from each plan to the reference plan
## Output:
# policies: a list of policies
def generatePolicy(TD) :
  planPairs = makeEnhancedPlans(TD)
  compatiblePlanGroups = gatherCompatiblePlans(planPairs)
  policies = []
  for planGroup in compatiblePlanGroups :
    refPlan = getFirstPlan(planGroup)
    planMaps = makeMappingsBetweenPlans(refPlan.$\pi_{\Sigma_i}$, planGroup)
    rules = []
    for $x$ in [len(refPlan.$\pi_{\Sigma_i}$)..1] :
      rules.append(makeRule($x$))
    $\pi_{\Sigma_i}$ = Policy(rules)
    policies.append($\pi_{\Sigma_i}$)
  return policies
```

The pseudo-code for the process is presented in Listing 8.1. In the following parts each of the functions will be explained. The algorithm builds on the approach for individual plans in two ways: the plans are translated into the enhanced language (`makeEnhancedPlans`(*TD*)); and the abstraction steps in enhancing the plan leads to some plans being equivalent in Σ_i. The *equivalent* plans are grouped together, providing a more general context for selecting the condition and used to generate an RBP in Σ_i. We discuss the process for use with the SbS and therefore with the expectation that the policy will be applied at each state. In this subsection we describe how a plan can be rewritten using the enhanced language and how the mappings are established.

In the next subsection we present an extension of the process developed above that implements the `makeRule` function in the listing.

Model enrichment

The first step (`makeEnhancedPlans`(*TD*)) requires that each of the examples in **TD** is enhanced through a chain, $\Sigma_0, \ldots, \Sigma_n$, to a language, Σ_i. This relies on additional behaviours from the special purpose solvers. In particular, the translation of a plan sequence in Σ_j to an enhanced language, Σ_{j+1}. This translation is used to create a plan in Σ_i by incrementally enhancing the plan is it is passed through the chain.

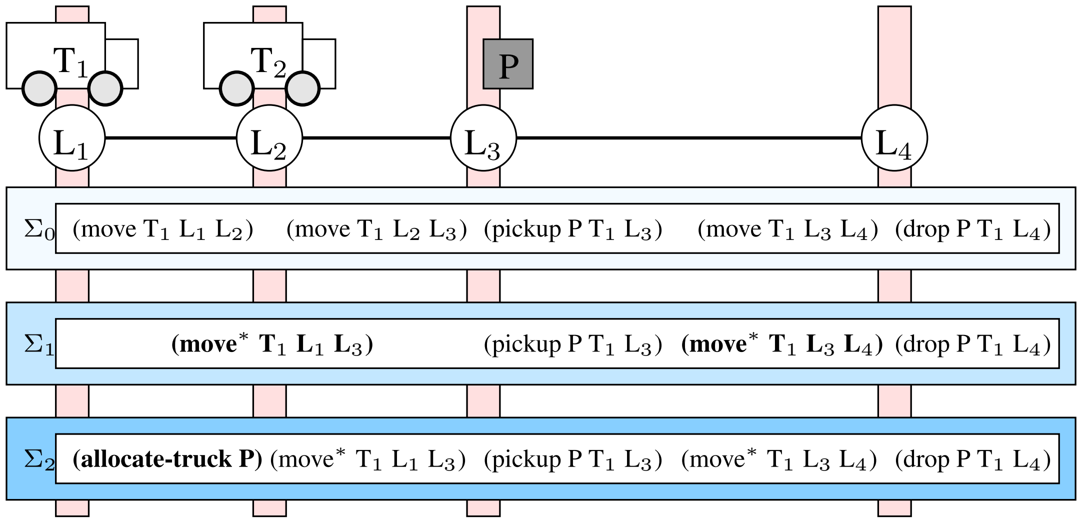

T1
T2
P
L1
L2
L3
L4
Σ0 (move T1 L1 L2) (move T1 L2 L3) (pickup P T1 L3) (move T1 L3 L4) (drop P T1 L4)
Σ1 (move* T1 L1 L3) (pickup P T1 L3) (move* T1 L3 L4) (drop P T1 L4)
Σ2 (allocate-truck P) (move* T1 L1 L3) (pickup P T1 L3) (move* T1 L3 L4) (drop P T1 L4)

Figure 8.4: The enhancements of a plan for a transportation problem. The plan is expressed using the enhanced languages.

In enhancement steps that provide an abstract action, the translated plan will replace the action sequence with the abstract action. For example, in the transportation problem illustrated in Figure 8.4, the step to Σ_1 abstracts the move actions to `long-move` actions (labelled `move`* for presentation). In enrichment steps that provide actions that make decisions then these actions are positioned at points where the decision would have been made to explain the subsequent behaviour. In the example transportation problem, the resource allocation is made at the beginning and leads to `Truck`$_1$ being used to deliver the package. There can be more than one rewording of a plan; however, our ordered rewording leads to a single ordering (that is a solver can only reword actions that are still part of the plan and cannot reword part of an action). We use the notation, $\langle\pi\rangle_{\Sigma_i}$, for a plan, $\langle\pi\rangle$, enhanced through i steps.

We assume that any interaction between an abstract action's action sequence and another action can be explained by the policy, although this is not the case in general. We also make the assumption that if goals are achieved as part of an abstract action then this is ensured by the computation of the action or some property of the problem structure; in particular, the generated rules will not uncover this structure.

Positioning of abstract actions There may be several places that the abstract action can be positioned because of other threads of actions. We order abstract actions by their final action, so that the sequence that finishes first in the plan is positioned earlier. Typically, single actions maintain their position in the enhanced plan. However, if they are ordered within a sequence then they are positioned after the sequence. For example, in a Goldminer plan there is a sequence of moves on the way to pick up the bomb. The laser is dropped during this sequence, but it is not part of the sequence. This action is positioned after the sequence of moves in the resulting plan. This approach is compatible with our rule ordering approach. It will lead to the drop laser rule being positioned before the move sequence, which provides the opportunity for the policy to map to this action during the selection of move steps.

The backbone plan Abstracting actions are underpinned by a sequence of actions that can be expressed in Σ_0; however, the enriched state actions have no associated actions in Σ_0. When a policy, expressed in a language, Σ_i, is unrolled then it will progress through each of the actions in Σ_0 and the enriching actions. For example, to generate the example illustrated in Figure 8.4, a policy would map to the following actions:

$$\texttt{(allocate-truck P)}, \texttt{(move T}_1\texttt{ L}_1\texttt{ L}_2\texttt{)}, \texttt{(move T}_1\texttt{ L}_2\texttt{ L}_3\texttt{)},$$
$$\texttt{(pickup P T}_1\texttt{ L}_3\texttt{)}, \texttt{(move T}_1\texttt{ L}_3\texttt{ L}_4\texttt{)}, \texttt{(drop P T}_1\texttt{ L}_4\texttt{)}$$

We call the plan expressed in this way the plan *backbone* and denote it $\texttt{backbone}(\langle\pi\rangle)$.

An action step's context states The initial state of each training example can be used to generate a corresponding Σ_i state and the *backbone* plan can be simulated from the state using co-execution. We call the sequence of states that are visited during this simulation the *context states*. The *context states* will provide the context for determining the rule conditions. For a plan, $\langle\pi\rangle$, and the associated enhanced plan, $\langle\pi\rangle_{\Sigma_i}$, and backbone, $\texttt{backbone}(\langle\pi\rangle)$, the *context states* of a step, $a \in \langle\pi\rangle_{\Sigma_i}$, are the states that are transitioned from by either a, or an action, $a' \in \texttt{backbone}(\langle\pi\rangle)$

that was part of a sequence that was enhanced leading to a. For example, the context states for (`long-move` $\mathtt{T}_1$ $\mathtt{L}_1$ $\mathtt{T}_3$) will include the states that (`move` $\mathtt{T}_1$ $\mathtt{L}_1$ $\mathtt{L}_2$) and (`move` $\mathtt{T}_1$ $\mathtt{L}_2$ $\mathtt{L}_3$) were applied in, and therefore states with $\mathtt{T}_1$ at $\mathtt{L}_1$ and $\mathtt{L}_2$. The *context states* for an abstract action do not need to be contiguous.

The Σ_i-equivalence property Rewriting the plans in Σ_i can abstract them to a level above the specific plan steps. Although we would expect each training example in **TD** to be unique, they might solve problems that only differ in the specific steps made. The process of rewriting aims to uncover similarities between plans that seemed distinct when expressed in Σ_0.

We can use plans for two transportation problems as an example:

Plan, $\langle\pi_{T1}\rangle(\Sigma_0)$

1. (`move` $\mathtt{T}_1$ $\mathtt{L}_1$ $\mathtt{L}_2$)
2. (`pickup` `P` $\mathtt{T}_1$ $\mathtt{L}_2$)
3. (`move` $\mathtt{T}_1$ $\mathtt{L}_2$ $\mathtt{L}_3$)
4. (`drop` `P` $\mathtt{T}_1$ $\mathtt{L}_3$)

Plan, $\langle\pi_{T2}\rangle(\Sigma_0)$

1. (`move` $\mathtt{T}_2$ $\mathtt{L}_1$ $\mathtt{L}_2$)
2. (`move` $\mathtt{T}_2$ $\mathtt{L}_2$ $\mathtt{L}_3$)
3. (`move` $\mathtt{T}_2$ $\mathtt{L}_3$ $\mathtt{L}_4$)
4. (`pickup` `P` $\mathtt{T}_2$ $\mathtt{L}_4$)
5. (`move` $\mathtt{T}_2$ $\mathtt{L}_4$ $\mathtt{L}_5$)
6. (`move` $\mathtt{T}_2$ $\mathtt{L}_5$ $\mathtt{L}_6$)
7. (`drop` `P` $\mathtt{T}_2$ $\mathtt{L}_6$)

$\langle\pi_{T2}\rangle$ moves several more steps than $\langle\pi_{T1}\rangle$, however, the tasks that are being achieved as part of the plan are the same. This becomes clear when the two plans are rewritten and compared at a level above the specific Σ_0 actions.

$\langle\pi_{T1}\rangle_{\Sigma_1}$	$\langle\pi_{T2}\rangle_{\Sigma_1}$
1. (`long-move` T$_1$ L$_1$ L$_2$)	1. (`long-move` T$_2$ L$_1$ L$_4$)
2. (`pickup` P T$_1$ L$_2$)	2. (`pickup` P T$_2$ L$_4$)
3. (`long-move` T$_1$ L$_2$ L$_3$)	3. (`long-move` T$_2$ L$_4$ L$_6$)
4. (`drop` P T$_1$ L$_3$)	4. (`drop` P T$_2$ L$_6$)

Definition 8.4.1 *Two plans, $\langle\pi_1\rangle_{\Sigma_i}$ and $\langle\pi_2\rangle_{\Sigma_i}$ are equivalent under Σ_i-equivalence if the sequence of action names are the same in each plan (Σ_i), goals are achieved by the same plan index in each plan and there is a one-to-one mapping between the constants that parameterise the actions.*

This definition is used to gather groups of Σ_i-equivalent for the function, `gatherCompatiblePlans`. The output is a partition of the training data.

Creating a mapping between plans The equivalence property is commutative and therefore identifies groups of plans that only differ in the specific steps applied as part of abstract actions. As a result a one-to-one mapping exists between the constants in each pair of reworded plans in a group. This is defined for two reworded plans, $\langle\pi_0\rangle_{\Sigma_i}$ and $\langle\pi_1\rangle_{\Sigma_i}$, as:

$$\begin{aligned} m_{\langle\pi_0\rangle_{\Sigma_i},\langle\pi_1\rangle_{\Sigma_i}}(p_{i_j}) &= q_{i_j}, \text{ where} \\ \langle\pi_0\rangle_{\Sigma_i} &= a_0(p_{0_0},\ldots,p_{0_{m_0}}),\ldots,a_n(p_{n_0},\ldots,p_{n_{m_n}}) \text{ and} \\ \langle\pi_1\rangle_{\Sigma_i} &= a_0(q_{0_0},\ldots,q_{0_{m_0}}),\ldots,a_n(q_{n_0},\ldots,q_{n_{m_n}})^{1} \end{aligned}$$

In our example, the mapping between $\langle\pi_{T1}\rangle_{\Sigma_1}$ and $\langle\pi_{T2}\rangle_{\Sigma_1}$ would capture the following mapping:

$$m_{\langle\pi_{T1}\rangle_{\Sigma_1},\langle\pi_{T2}\rangle_{\Sigma_1}} = \{\mathtt{T}_1 \mapsto \mathtt{T}_2, \mathtt{P} \mapsto \mathtt{P}, \mathtt{L}_1 \mapsto \mathtt{L}_1, \mathtt{L}_2 \mapsto \mathtt{L}_4, \mathtt{L}_3 \mapsto \mathtt{L}_6\}$$

This step computes the function, `makeMappingsBetweenPlans`, in the pseudocode above. The mapping can be used to define the *universe translation* ($\mathbf{UTranslation}$)

[1]In fact, we also include the goals into this mapping. This can matter in problems where goals are already achieved and this part of the state would be otherwise ignored. In these situations we select a mapping that satisfies the sequences and the goal formulae; for our approach this is sufficient.

function that translates a set of propositions from one universe into another. For each proposition, a proposition with the same predicate symbol is created. Each argument of the new proposition is found by mapping the argument in the original proposition. The proposition is added to the translated proposition set if there is a mapping for all of the proposition's arguments. If there is no mapping then the proposition is not included in the state.

$$\begin{aligned}&\mathbf{UTranslation}_{\langle\pi_0\rangle,\langle\pi_1\rangle}(s) = \{\psi(q_0,\ldots,q_n)|\\&\quad\psi(p_0,\ldots,p_n)\in s\ .\\&\quad m_{\langle\pi_0\rangle,\langle\pi_1\rangle}(p_0) = q_0\ .\ \ldots\ .\ m_{\langle\pi_0\rangle,\langle\pi_1\rangle}(p_n) = q_n\}\end{aligned}$$

8.4.3 A rule for a $\langle\pi\rangle_{\Sigma_i}$ plan step

A group of Σ_i-equivalent plans is used to make a list of rules. A rule is created for each step in the enhanced plan. The pseudo-code for the `makeRule` function is presented in Listing 8.2. We first explain how the conditions are generated from a set of *context states* and potential propositions (Pot_i). Then we describe how the set, Pot_i, is generated at each step.

Listing 8.2: Pseudo-code for rule generation.

```
## Inputs:
# $x$ a $\langle\pi\rangle_{\Sigma_i}$ plan step
## Fields:
# planGroup: plans that are equivalent in $\Sigma_i$
# refPlan: one plan from the equivalent group
# planMaps: mappings from each plan to the reference plan
## Output:
# a rule for the plan step
def makeRule($x$) :
    $Pot_x$ = gatherPotentialPropositions($x$, refPlan)
    posCon = identifyPositiveConditions($Pot_x$, $x$)
    negCon = identifyNegativeConditions($Pot_x$, $x$)
    vars = gatherVariables(posCon + negCon)
    allDiff = makeAllDiff(vars)
    action = refPlan.$\pi_{\Sigma_i}$[x]
    return Rule(posCon + allDiff, negCon, action)
```

The rule condition

The motivation is still the same: we aim to identify when the action sequence can be applied and when it should be applied. The context for making the decision has grown in two directions. The propositions used for the potential proposition pool, Pot_i, at step i, include propositions from different language levels. These propositions are gathered using the reference policy and are therefore using its constants. We will discuss how this set is constructed below. We have already defined the set of *context states* for a plan step and these provide the context that we use to validate each of the propositions in Pot_i.

The pseudo-code for the approach to selecting the positive conditions is presented in Listing 8.3. In order to confirm a positive condition the proposition is tested for each of the plans in the group. If each of the plans validates the proposition then it is included as a condition. To evaluate the proposition against a specific plan, the *context states* for the particular plan step are generated for the plan. In the case of the positive condition this is the regressed states, ϕ. The proposition is then evaluated in each of those states. If it does not hold in a state then the proposition is not a positive condition. The negative condition is computed using a similar approach, except the test ensures that the proposition never holds in any of the example states. The computation of ϕ and s, for the enhanced state, is discussed below.

Listing 8.3: Pseudo-code for selecting positive conditions.

```
## Inputs:
# x: a $\langle\pi\rangle_{\Sigma_i}$ plan step
# $Pot_x$: the potential proposition pool
## Fields:
# planGroup: plans that are equivalent in $\Sigma_i$
# refPlan: one plan from the equivalent group
# planMaps: mappings from each plan to the reference plan
## Output:
# positiveCondition: a set of propositions for the positive condition
def identifyPositiveConditions($x$, $Pot_x$) :
  positiveCondition = []
  for $p$ in $Pot_x$ :
    alwaysSeen = True
    for $\langle\pi\rangle \in planGroup$ :
      if not testPropAlwaysTrue($p$, $\langle\pi\rangle$, $x$) :
        alwaysSeen = False
    if alwaysSeen :
      positiveCondition.add($p$)
  return positiveCondition

def testPropAlwaysTrue($p$, $\langle\pi\rangle$, $x$) :
  for $j$ in getContextStateIndexes($\langle\pi\rangle$, $x$) :
      if not $\mathbf{UTranslation}_{\text{refPlan},\langle\pi\rangle}(p)$ in $\phi_j$ :
        return False
  return True
```

Of course, the sets of example states are from a mixture of universes. However, as the plans are Σ_i-equivalent there is a mapping, $\mathbf{UTranslation}_{\pi_0,\pi_1}(\phi_j)$, for the language of π_0 and π_1, between sets of propositions. The proposition is translated into the same universe as the current group plan and the proposition is evaluated in that context. Mapping the proposition is more practical within our implementation, as several of the enhancement steps, including the ALMA predicates, are implemented as derived predicates and are therefore evaluated on demand.

Constructing the sets: s, ϕ and Pot

We rely on the propositions of Σ_0 to provide a background that provides an estimate of the necessary relationships that exist between propositions in richer languages. For example, a truck might move through several locations to arrive at the location of a package. The position of that package provides vital context in constructing the rule condition for moving the truck. Whereas, the intermediary locations, which are abstracted by the richer language, are irrelevant. We draw on a pool of Σ_i-equivalent plans to provide a richer context, isolating a smaller condition, which hopefully include the propositions of more significance to the task. To this background we add the propositions in richer languages. This relies on extending the construction of the sets of propositions, s, ϕ and Pot, to include Σ_0, decision representing, axiomatic and abstract action condition predicates.

The states can be built up in steps as defined in Chapter 5. However, this is not appropriate for the decision predicates, which require some special treatment that we describe below. In this work we do not distinguish between s and ϕ for the enhanced predicates. Each of the potential propositions is evaluated against the enhanced state.

Σ_0 propositions The low level plan is used to gather Σ_0 propositions; however, we remove any propositions that rely on constants that are not in the mapping between plans. This means that we retain as many propositions as possible, while removing the propositions that would prevent the abstract actions from generalising SIs. ϕ is computed in the same way as before.[1]

Abstract action preconditions We treat abstract action preconditions in a similar manner to Σ_0 predicates. A proposition, p, for abstract action, a_k, is added into Pot_j, for $j \leq k$. The preconditions of abstract actions are computed using derived predicates (a formula over the state). To evaluate p the proposition is evaluated in the state in a similar manner to during planning.

[1]This will be used to validate propositions with constants in the mapping, therefore pruning this set is not necessary.

Decision predicates These propositions represent decisions that have been made as part of the action sequence. As a result, the planner will exploit the choice proposition, p, between making the decision, a_D and its removal, a_R, at the point where the decision will play no further role. The proposition should be included into the potential propositions at the point of removal: $p \in Pot_j$, for $j \leq R$. In place of the normal initialisation of the decision predicates, the states in between a_D and a_R, will include the proposition, p. Therefore, the sets ϕ_j and s_j will model p, for $D \leq j \leq R$.

Axioms Axioms are evaluated as a response to the state and are not impacted on by the sequence. As a result, each axiom in Σ_i that uses parameters already in the condition should be included in the sets of potential predicates. This might be overwritten by a particular solver, which selects the appropriate instantiations of the axioms for each state. For example, the `nearest-blocked` predicate relates the local moves to the goal of the move and therefore the furthest node in an opening episode (similar to the rules presented in Section 7.4). For computation of ϕ, the predicate should be evaluated in the state in order to determine whether it should be included as part of the condition.

8.4.4 A worked example

In this subsection we describe how the ALMA generation process, presented in Chapter 7, extends to construct the structures required for policy generation. We then analyse specific rules that are generated by the system.

An ALMA step

The policy generation algorithm can be used with an ALMA language step, with a small extension to the ALMA generation presented in Chapter 7. The same training data set is used to first identify the sequences and generate ALMAs. The chunks, identified during the ALMA generation process, must be recorded, and associated with the ALMA that is generated from them. This then provides a connection between the abstract action and the underlying actions, necessary for the policy generation stage. The original plan actions are replaced by the generated ALMA to define the enhanced plan, $\langle \pi \rangle_{\Sigma_{\mathbf{si}}}$. The *context states* are computed using the original plan actions and grouped around the action that was applied in that state (or ALMA, if the action was in a chunk). The policy generator uses the ALMA action and proposition in the construction of the policy. Due to the computation method of the proposition it requires a slightly modified treatment. However, as the process validates the propositions by evaluating them in the state, this change is isolated to the state sets (Pot_j and s_j)

$\Sigma_{\mathbf{SI}}$ propositions The expansion based computation of the propositions means that SIs can only be evaluated in the current state. This has two implications: an SI that is enabled by part of the subsequent sequence, (for example, the move to goal part of a Goldminer plan is enabled by the clear path towards goal part,) will not be applicable in states before the preparation is complete; and the node that is the current focus of the actor of the SI in the current state, (such as the current node of a traverser) is the only node that can be tested for an SI.

As presented above, for a particular rule, a condition is proposed for each ALMA that occurs later in the plan. Each potential (ALMA) condition is created using the propositions modelled by the ALMA. The actor and the final constants are the same as in the plan sequence; however, the initial constants are modified to be consistent with the current state. For example, in a state of a traverser problem, a traverser, `t`, might be at a location, $\mathtt{l}_1$, and the sequence might move the traverser from $\mathtt{l}_2$ to $\mathtt{l}_3$. In this situation the condition `t` $\mathtt{l}_1$ $\mathtt{l}_3$ will be proposed. In many situations the *from* parameter is redundant and, if it was not modelled, then this would be the behaviour. For example, the stacking problem `unstack` ALMA only references the block to be uncovered and makes not reference to the current configuration. The particular structures of the problem and properties of the ALMA will determine whether this condition remains relevant. The move to pickup gold demonstrates a situation where this condition makes sense: to evaluate the move to pickup the gold condition the solver will attempt to move the robot from its current location to the gold square. If this action is applicable then this action should be selected immediately. Of course, in the states before the path to the gold square is cleared then this action is not applicable and provides the justification for the other actions.

A Goldminer rule

Figure 8.5, presents an overview of the possible plans for Goldminer. The SIs are illustrated with dotted lines indicating that they can cover an arbitrary number of steps. The Goldminer domain is particularly susceptible to our policy generation approach, because the main challenge in the domain is the control of search through a collection of SIs. The following rule was generated from several instantiations of the “move

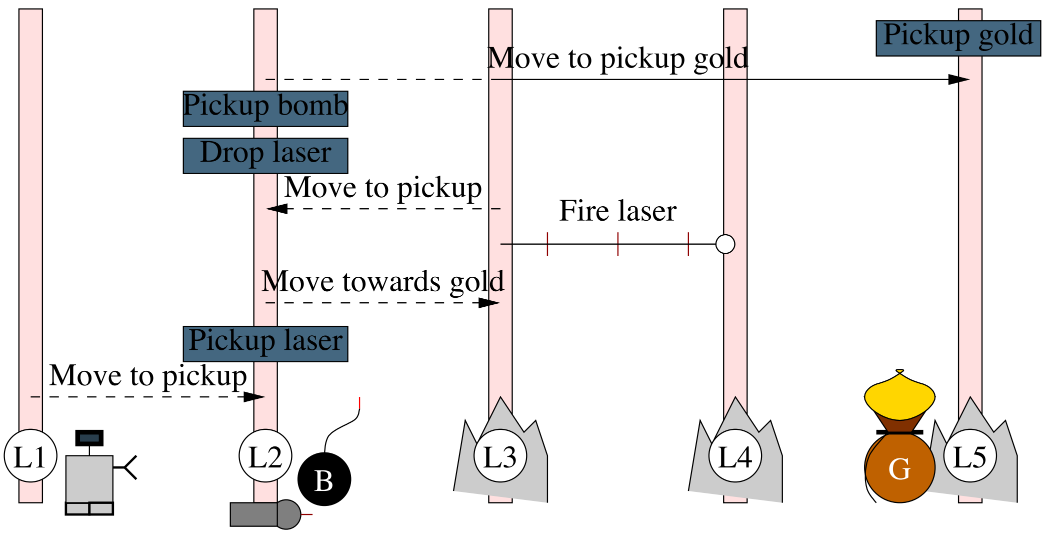


Figure 8.5: An abstraction of Goldminer solutions. Actions are labelled at the locations they act on; the initial state is illustrated at the bottom and the action sequence runs from bottom to top. Dotted lines indicate the action acts on any number of nodes.

towards goal" step. We will use it to demonstrate aspects of the generated policies.

```
(:rule Rule38
  :parameters ( ?f2-3f - loc ?robby - robot ?bomb - bomb ?f0-0f - loc ?laser - laser
      ?f2-1f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and
  ...
  (fire-laser-1-063_connected ?robby ?from_loc ?f2-1f) ; fireMoveBag
  (at-gold ?f2-3f) (holds ?laser ?robby) ; current state
  (connected ?f2-2f ?f2-3f) (connected ?f2-1f ?f2-2f) ; two steps from gold
  (pickup64_connected ?robby ?from_loc ?f0-0f) (at ?bomb ?f0-0f) ; could get bomb
  (not (pickup-gold65_connected ?robby ?from_loc ?f2-3f)) ; cannot reach the gold
  ...)
  :goalCondition (and (holds-gold ?robby) )
  :action (fire-laser-1-063_move robby from_loc f2-1f)
)
```

The predicates `at-gold` and `holds` demonstrate the use of Σ_0 predicates to establish the background of the state. There are many other Σ_0 predicates in the generated rule that have been omitted for presentation. For example, the fact that the bomb is not destroyed, and that the robot is not holding the gold. The rule also demonstrates the enhanced state being used to determine the predicates that are valid. On the path towards the gold, the robot can move to the bomb. However, until the proposition `clear` *?f2-2f* (the robot can move to one step from the gold) then the rule that moves the robot to the bomb will not fire. However, this proposition adds to the context of the situation. If the bomb could not be reached then the intended action sequence would not apply and there would be no benefit of continuing with the encoded strategy.

In the abstracted example illustrated by Figure 8.5, it is evident that there are parts of the problem that are the same in every plan and there are other parts where the specifics vary between plans. For example, there are several dotted lines that indicate that the underlying SIs act on a varying number of nodes and they could also vary in the macros used. In these parts the mapping is only defined for the beginning and end points and therefore the specifics of the underlying structure are left to the appropriate abstract action conditions. Conversely, each plan requires the robot to establish a path in the direction of the gold. In particular, the last step of this path should be two nodes from the gold. The robot will then fire the last shot and go for the bomb. Uncovering this "two-steps-from-gold" property is vital for directing the robot behaviour towards the gold. As the relationships exist between endpoints in SIs, these nodes are part of the mapping; it follows that these connected propositions are part of the condition of any rule where they are preconditions in future actions. The rule presented above captures this property and uses it to guide the opening of nodes in a path towards the goal.

A transportation rule

In transportation problems we have considered the use of allocation actions that determine the transporter that will be used to service a package. We consider an example state from $\langle \pi_{T2} \rangle_{\Sigma_2}$, in between the allocation being set by the allocate action and the pickup action that confirms that it was used as the resource. In this example, the truck is being moved towards this package. A condition that ensures the truck is allocated to the package is included in the condition for rules that moves the truck to pickup the package.

```
(:rule Rule3
  :parameters ( ?package1 - obj ?s4 - location ?s9 - location ?truck1 - truck ?s2 -
      location ?from-location - location)
  :condition (and (unload-truck52_connected ?truck1 ?from_location ?s2)
    (allocated-truck ?truck1 ?package1)
    (at ?package1 ?s4) (at ?truck1 ?from_location) (at ?truck1 ?s9)
    (load-truck51_connected ?truck1 ?from_location ?s4) AllDiff
    ConnectionPredicates (not (at ?package1 ?s2)) (not (at ?truck1 ?s4))
    (not (at ?truck1 ?s2)) (not (in ?package1 ?truck1)) )
  :goalCondition (and (at ?package1 ?s2) )
  :action (load-truck51_move ?truck1 ?from_location ?s4)
)
ConnectionPredicates = (link ?s9 ?s4) (link ?s4 ?s2)
AllDiff = (!= ?s9 ?s2)  (!= ?s9 ?s4) (!= ?s2 ?s4)
```

This rule captures the necessary conditions that determine that the truck should move towards a package: that the package is misplaced, the truck can move to its location and the truck is allocated to the package. There are two limiting factors implicit in this rule: the connection predicates and the specific truck located predicate. The reason for these limitations is that this rule has been learned from one training example and as a result has been over-fitted. We would expect in a training set that there would be examples that require more than a single traversal. If that is the case then there will be several source locations, preventing a single one becoming part of the condition and the source and destination variables are not all connected and therefore the connected predicate cannot become part of the condition.

As a comparison we can use a larger collection of training examples to generate the rule. The resulting rule loses the conditions that limit the application of the abstract actions. In particular, the specific located predicate for the current truck position and the connected predicates are both missing.

```
(:rule Rule3
  :parameters ( ?package1 - obj ?s4 - location ?s13 - location ?truck1 - truck ?
      from_location - location)
  :condition (and (unload-truck52_connected ?truck1 ?from_location ?s13)
    (allocated-truck ?truck1 ?package1)
    (at ?package1 ?s4) (at ?truck1 ?from_location)
    (load-truck51_connected ?truck1 ?from_location ?s4)
    (not (at ?package1 ?s13)) (not (at ?truck1 ?s4)) (not (in ?package1 ?truck1)))
  :goalCondition (and (at ?package1 ?s13) )
  :action (load-truck51_move truck1 from_location s4)
)
AllDiff= (!= ?s13 ?s4)
```

Through using more than one example, we identify the important predicates that always exist and never exist. This allows us to retain the benefits of regression analysis while reducing the effects of over-fitting.

8.4.5 Discussion

In this section we have presented an approach to generating an RBP from a selection of example plans. The key benefit of this approach is that we use regression to extract the reasoning behind action sequences and use a collection of examples to reduce the number of unnecessary propositions used to construct the rule conditions, in an attempt to identify the causal relationships that determine whether the rule is appropriate. The aim is that the policies generated from a small selection of problems will possess characteristic relationships for the domain that can bootstrap the learning process for more general solutions.

Generalising goal hierarchies

The Σ-equivalence mapping that we defined can be relaxed in two ways. The one-to-one mapping between constants means that the plans each use the same collections of objects in the same way. There is an opportunity to allow a surjective mapping between plans, allowing a single constant in one plan to occupy various roles operated by distinct constants in another plan. This would allow more plans to be considered equivalent; however, the equivalence property would be ordered. Instead of the current arbitrary selection of a reference plan, a specific plan with a maximal equivalence set would be used. Part of this development would involve removing the not equal propositions for those variables where there is evidence that the same constant can play both roles.

The other relaxation involves plans that are similar, but miss some stages. For example, the effects happen to have been achieved in the initial state, or as a compulsory side-effect of earlier tasks. These plans could be incorporated into the evidence base for creating rules from longer plans. Each plan could then be associated with a bit string of the length of the complete plan that would indicate for each step whether the plan included that part. A mapping from a particular reference plan would be required for each stage and this plan would be used to generate the potential propositions. The test of whether a proposition is supported would be made in the context of all plans that include the particular plan step. These are two developments that could lead to more general and robust policy generation.

The generated conditions

In the context of the ALMA step, the condition can be seen as two distinct parts: the definition of a background that holds during the interaction that provides the reason for the SI; and a deictic (Agre and Chapman, 1987) representation local to the SI itself that identifies invariants local to the traversing object. The reason for the distinction is as the focus moves through the structure the rule variables active in the SI will bind with several constants whereas variables in the background are expected to bind with the same constants as the policy is unrolled. For example, there could be an object that is attached to a node, but plays no role in an SI under consideration. If it plays a role in the stages after this particular SI then it should be included in the condition as the background to making the SI. Only some of the nodes focussed on during an SI will have this object attached, so it cannot be part of the local SI part of the condition. The properties of nodes that are effected by the unfolding SI should not be singled out. Instead, the rule should characterise the important aspects of the parameters of the ALMA and leave the associated vocabulary to assert properties of the underlying relationships necessary to apply the SI.

The sequence breaking rules that we presented in Section 7.4 play an important role in establishing effective decompositions. If an ALMA was generated from a sequence that achieved several sub-goals then the computation of the vocabulary will not guarantee that these are achieved in subsequent applications. If the sequence is broken at nodes that cannot be explained by the rule language then the RBP will not provide the necessary search control. One example, is the approximation used as part of the ALMA condition. There are potential approaches that could remove or reduce the impact of the restrictions on evaluating ALMA predicates from the current state. In Sebastia et al. (2006), a projection is made of what the state *might* be when a particular sub-problem is being tackled. A similar approach could be used to predict the state that the SI will be made. The bags would be expanded from this state to provide an estimate of their reachability. An alternative is to use the predicate as described above, but reinforce it from other sources. For example, the static graph that underlies the traverse actions can confirm that certain chains are not possible. The rules that govern a package delivery could be reinforced with a condition that tested that there is a path in the static graph between the goal of the package and the truck's goal location. If a static-graph abstraction solver was in the solver chain then the generated policy would exhibit these predicates.

Generating conditions for control knowledge

Researchers have investigated defining conditions for control knowledge in several areas. The approaches of Hogg et al. (2008) for HTN method conditions and Newton et al. (2007); Newton and Levine (2010) for macro operators, use the same regression based approach that we use in establishing whether the sequence can be applied. The approach presented in Ilghami et al. (2006) exploits negative examples to maintain *version spaces* for each method condition for an HTN. Version spaces maintain the possible explanations of the training data given positive and negative examples. Assuming that an appropriate method of extracting negative examples exists, there is some potential for adopting this approach in order to reduce the size of the conditions in our rules.

A related problem has been investigated in the n-stage-to-go heuristic (Boutilier et al., 2001). In this work a quantified goal formula is regressed and used to identify what would need to exist in the state in order for the goal to be reached in n steps. A key restriction in this approach is the first order expansion of the state space grows quickly and the function becomes intractable. This has been addressed in Gretton and Thiébaux (2004), as training data is exploited to restrict the expansion. During this project we investigated using a lifted regression from goal templates in an attempt to generate more general rules. However, the quickly growing search space meant that restricting the expanded sequences using observations from training data was inevitable. In order to simplify the integration with the solvers we decided to start from the sequences (as reported above).

In Srivastava et al. (2011), a generalised plan representation is adopted that can capture plans for arbitrary numbers of objects. Inherent looping and symmetry reduction in our policy representation provide similar compression on plans to the representation used in Srivastava et al. (2011). However, in Srivastava et al. (2010) the authors identify a restricted set of problems, which allows for termination to be guaranteed. Similar guarantees have not been established for our rule representation. However, the policies that we generate can solve a richer set of problems. For example, policies that solve problems over arbitrary graph structures cannot be captured in the representation.

CHAPTER 9

RESULTS FOR LEARNING AND THE ENHANCED LANGUAGE

In this chapter we present evidence that enhanced problem models can be automatically instantiated and that these domain models can be exploited as part of learned control knowledge. Where applicable the experiments in this chapter focus on the ALMA representation: its efficiency, the automatic parameterisation of ALMA solvers, and their exploitation within generated RBPs. The empirical analyses are carried out using a similar setup as was used in Chapter 6, we detail the small changes below. First, we analyse the generated ALMAs, then we present the parameters used for learning and generating RBP and present results for RBPs learned using our approach.

9.1 Invoking a domain model, $\mathbb{M}|_{\Sigma_i}$

We have developed an approach for automatically enhancing a domain model with concepts that relate to its structures. In this section we present the vocabulary that was invoked for a selection of domains.

9.1.1 Invoking an appropriate model from a solver library

We used the automatic invocation of solvers from the solver library for the analysis in Chapter 6 and we summarise the generated vocabularies in this subsection.

In several of the domains an appropriate domain model was invoked for expressing effective RBPs. In Blocksworld and Depots, the stacking aspect was identified and the `well-placed` predicate introduced. In Driverlog, Goldminer, Logistics, Gripper, Grid and Depots the graph traversal generic type was identified and the graph abstraction solver invoked. In Goldminer and Grid the turn-based nature of the map was identified, invoking the nearest blocked location solver. In Grid, the restricted relationship between the opening resource and the opening action was identified and the located key door selector solver was introduced.

In several of the domains, a graph abstraction solver step was made when the model was sufficient for an RBP to effectively control search. The domain conventions mean that, in these domains, the graphs are cliques and therefore connectedness is expressible in a finite number of predicates. An interesting line of future work is to extend the scope of the target significance detection, as a general framework for invoking specialised language. This could be used to further specialise the domain model by parameterising the solvers using observations from training data. For example, through interrogating the graphs in the training data to determine their connectedness. An alternative is to propose the vocabulary and allow a general purpose rule learner to confirm its use in practice. This is the approach that we take and report on below. The RBPs used to generate the results presented in Chapter 6 were expressed for the invoked domain models and the results demonstrate the appropriateness of the vocabulary.

9.1.2 Generating appropriate solvers for directed connectivity

We have proposed an algorithm for automatically specialising the solver for a particular domain. We implemented the algorithm presented in Sections 7.3 and 7.4, making use of certain optimisations that were appropriate for traversal problems. The sequence coverings for SI-templates that each end in a traversal action are unique as distinct traversal actions, or traversal actions controlling a different traverser are distinct sequences. We have generated training data for several domains that contain SIs and have applied our approach. In this subsection we summarise the generated bags and resulting vocabulary. We examine the difference using the algorithm with and without sequence breaking rules, to provide an indication of the importance of breaking the sequences into parts. The generated bags are presented in the appendices in Section E.4, and as an example, we present the results for Driverlog below. We present an analysis of the approach for Structure Building problems in Appendix H.

Driverlog ALMAs

The Driverlog domain is a transportation domain with drivers, trucks and packages. The trucks are driven and problems involve multiple traversers on two maps. The competition problem generator was used to generate 25 problems. These problems had a single truck, driver and package. Each problem had 15 truck locations and a number of joining path locations depending on the generated path map. Goals were generated for the truck, driver and package by the generator, which can include empty goals for objects (this is more likely for the trucks and drivers). The targets identified in Driverlog problems include picking up and dropping off packages and walking drivers to trucks and to their goals.

No rules With no sequence breaking rules each of the agent threads is isolated and the relevant actions for each agent are identified, as described in Section 7.1. The last SI action of each sequence is used as the target and gives its name to the vocabulary. The generated actions were `walk51_move` and `drive-truck50_move`. The bags included the enablers for getting a driver into the truck for driving and getting out of a truck for walking.

- (`drive-truck` *truck0 location1 location2 driver3*)
- (`board-truck` *driver3 truck0 location1*), (`drive-truck` *truck0 location1 location2 driver3*)
- (`walk` *driver0 location1 location2*)
- (`disembark-truck` *driver0 truck3 location1*), (`walk` *driver0 location1 location2*)

Rule based sequencing When using the transportation rules, the bag generation process identifies several bags, one for each of the target types identified. The name of the ALMA is an extension of the target action, with a generic number ID and move attached.

- (`load-truck50_move` *?truck ?loc-from ?loc-to*)
- (`drive-truck51_move` *?truck ?loc-from ?loc-to*)
- (`disembark-truck53_move` *?truck ?loc-from ?loc-to*)

- (`unload-truck54_move` *?truck ?loc-from ?loc-to*)
- (`walk52_move` *?driver ?loc-from ?loc-to*)
- (`board-truck55_move` *?driver ?loc-from ?loc-to*)

The targets for walking were a goal that was achieved by walking (locatedness of the driver) and when the driver got into the truck. Similarly for the driving actions, the truck was driven to a goal, the truck was driven to pickup and drop off packages and the driver disembarked at their goal. The latter target is the result of the driver using a goal-less truck to move to its goal. The actions can be interpreted as actions that move to enable the target action; for example, `move-to-enable-load-truck`.

The generated bags In each case, the associated bags contained a singleton macro. The bag for the truck ALMAs was:

- (`drive-truck` *truck0 location1 location2 driver3*)

The bag for driver ALMAs was:

- (`walk` *driver0 location1 location2*)

These ALMAs model vocabulary that are equivalent to the graph abstraction solvers used in the experiments in Chapter 6, in the sense that the predicate holds for the same arguments and the action will return the same output given the same choice.

The generated ALMA is included in the solver listings file. The listing created for moving a truck to achieve the target of unloading a package is presented in Listing E.1. The “@” symbol is used to separate actions in a chunk and the “#” symbol separates chunks. This has proven a convenient representation when reading the output and when defining ALMAs by hand.

Listing 9.1: Extract from solver listings output for generated ALMA for Driverlog domain

```
(:solverDescription SequenceChainSolver202
  :Graph (:module DynamicGraphModule163)
)
(:moduleDescription DynamicGraphModule163
  :MoveAction (:module MoveAction1)
  :Name (:description (unload-truck202))
  :Sequences
  (:description
    (drive-truck truck0 - truck location1 - location
      location2 - location driver3 - driver@#) )
   :Significance (:description (true))
  :Strategy (:description (false))
)
```

A summary of the generated ALMAs

Table 9.1: A summary of the properties of the generated ALMAs, detailing whether the property holds *with* and *without* sequence breaking rules. The properties are: target significance (Target$_{Sig}$), directed connectivity (DC), and appropriate for expressing a policy (π). In the policy column, *individual target* is used to indicate that solutions for certain goals can be captured. The greyed result is a proposed results for traversing constrained by fuel.

Feature	Target$_{Sig}$	DC	π
Driverlog	with/without	with/without	with/without
Logistics	with/without	without	without
Goldminer	with	with/without	with/without
Grid	with	without	without
Rovers	with/without	with/without	with/without
Fuel traverser	no	with/without	individual targets

Table 9.1 summarises the generated ALMA bags. We have identified three features from each bag generated either using sequence breaking rules (with) or not (without). These are analysed in the following parts. The bags for Grid, Goldminer and Driverlog, which are presented above, demonstrate that the enablers for SIs are identified. The algorithm uncovers the necessary macros so that SIs can be made in the domains tested. In the Logistics domain, the vocabulary over-fits one of the bags. In Logistics problem distributions the truck always starts at a location and in the small problems we used to generate the vocabulary there was only one location in each city. If the package needed moved then it was picked up before the truck was moved. Otherwise the only packages the truck picks up get delivered to the airport. As a result, the `move-to-load` action is computed with a bag that does not include an action to move between an airport and a location, or a location and another location.

- (`drive-truck` *truck0* - `truck` *location1* - `location` *airport2* - `airport` *city3* - `city`)

Of course packages might need to be dropped off at either the airport or the location and therefore the bag for unloading included either direction.

- (`drive-truck` *truck0* - `truck` *location1* - `location` *airport2* - `airport` *city3* - `city`)
- (`drive-truck` *truck0* - `truck` *airport1* - `airport` *location2* - `location` *city3* - `city`)

The difference between the bags demonstrates an advantage of using training data to parameterise the solvers. Implicit control knowledge has also been observed in inferred domain models (Cresswell and Gregory, 2011). In the following parts we present our observations of the results in terms of directed connectivity, target significance and completeness of the vocabulary.

Directed connectivity

Generating ALMA bags with no breaking rules will lead to vocabulary that establishes directed connectivity in domains where the specified SI is appropriate. For example, a single bag was generated for Grid, which included the macros that lead to picking up and swapping keys, required to establish directed connectivity (dropping off a key was never observed).

The resulting bags might not be sufficient to express an effective RBP. This is because each SI is treated in isolation. For example, in a domain with fuel (either as part of the map, or enabling the traverser), a path selected to achieve one particular SI might consume the fuel stored at a location, which is essential for a later SI. Similar observations can be made for other consumable resources where there is a property that demands that SIs are considered together. This can cause dead ends in some domains. The use of breaking rules addresses this problem through focussing the bags to specific targets; however, this is not a complete solution.

If the breaking rules are not appropriate for the particular domain, they can lead to abstract states that can only be partially protected. For example, in the Grid domain, the rules will break the plans into sequences of opening doors with the same key. However, the rule language is not sufficient to characterise these states. In this domain the bags will not establish directed connectivity. Goldminer is another turn-based graph domain. The result when using no sequence breaking rules is similar to Grid. However, in this domain the rules lead to targets that can be explained by the RBP rules. This results in vocabulary that establishes directed connectivity. The algorithm identifies equivalent macro bags for move to destroy hard and soft rock. These macros include the actions necessary to fire at hard or soft rock, if it is there, and move.

- (`fire-laser-1-0` *robot0 laser3 loc1 loc2*); (`move` *robot0 loc1 loc2*)
- (`move` *robot0 loc1 loc2*)
- (`fire-laser-0-1` *robot0 laser3 loc1 loc2*); (`move` *robot0 loc1 loc2*)

This is the **fireMoveBag**, defined in Section 7.2 and is important for tunneling towards the goal in Goldminer.

Target significance

If each individual SI step is isolated then the bag target significant expansion can be used. For example, the **fireMoveBag** moves the robot into the location that it has opened. In contrast, the following macro from the Grid bag opens a door so that it can be traversed through at some later point.

- (`unlock` *robot0 place1 place3 key4 shape5*); (`move` *robot0 place1 place2*)

This is unlikely to lead to target significance, because there will typically be a distinction between the locations that can be opened. There are specific examples where a macro impacts on future actions and is part of a target significant bag. For example, in the no rules setting the `board`;`move` macro sets up any movement. This is a special case where the resource is necessary for all of the moves.

The use of breaking rules can lead to the generation of target significant language. For example, a bag larger than the **allMovesBag** is generated for the Goldminer using no breaking rules. However, the expansion of this bag includes sequences that switch and consume resources. Splitting the sequences up leads to three smaller bags that are each target significant.

- (`detonate-bomb-1` *robot0 bomb3 loc1 loc2*); (`move` *robot0 loc1 loc2*)
- (`move` *robot0 loc1 loc2*)

The `move-to-enable-pickup-gold` bag, presented above, would not always be target significant. However, the example state spaces do not include any examples where the bomb was used unnecessarily. For example, the use of a bomb to achieve the shortest path to a target, where a slightly longer path would have achieved the same target without using the bomb.

Discussion

We have generated bags of macro actions for several domains. The bags generated without the rule based sequence splitting are directed connectivity in each of the tested domains. These bags provide vocabulary that can be used as part of a solution for each of these domains. The intention behind using rules is to split the sequences into sequences that can be explained by target significant bags. This has been demonstrated in the rules for Goldminer and Grid. In Driverlog the rules break the sequences so that `board` and `disembark` actions are targets. This separates the SI from other parts of the strategy; as a result the control knowledge associated with driver allocations is made explicit in the RBP. However, the rules can lead to bags that are not directed connectivity.

The ALMA representation provides an alternative to implementing specialised solutions. In Section 5.4, we presented vocabulary for several types of problems and each of these can be expressed as an ALMA solver. In general, the computation of the ALMA vocabulary is expensive, but we have demonstrated that in some cases the splitting rules provide an alternative way of making the resulting language more efficient. The rules are specified at the generic type level, which allows the developed approach to be used in many domains. In this work we have focussed on a collection of single traverser, single map problems. However, we explore the use of the representation in stacking problems in the Appendices, and in other traverser problems from the literature in the next section.

9.2 Arbitrary length macro actions

We will now investigate the efficiency of the generalised solver representation that we have implemented in the ALMA solver. In this section we will analyse the cost of computing the ALMA vocabulary. This is an important analysis for understanding the trade-off between the ease of expression and its use in solving problems. We also consider the use of the representation for a more general class of problems. In these evaluations the nearest neighbour solver heuristic is used with the graph abstraction and the ALMA solvers.

9.2.1 Vocabulary computation

The ALMA solvers provide a convenient representation for expressing control knowledge. However, our current implementation for computing the vocabulary is expensive. In this subsection we compare the use of the handwritten solvers that we evaluated in Chapter 6, with the use of ALMA solvers. We have selected the Driverlog and Goldminer domains for this analysis and present results for the hierarchical ALMA in Section G.1 of the Appendices.

Setup

The reachability of the move actions are sufficient for reasoning in the Driverlog domain. The same strategy can use the solvers interchangeably, providing a direct comparison of efficiency. In Goldminer the ALMA solvers provide richer reachability analyses and therefore the strategies are slightly different. For example, the strategy can exploit the **fireMoveBag** in order to move towards the gold and does not require the `closest-blocked` predicate. The projection to target space filtering was used in the expansion of the bags and the strategies were handwritten.

Expectations

Our implementation of the ALMA solver tackles a more general problem and its approach is not optimised for particular bags. As a result we would expect an increase in planning time.

Results

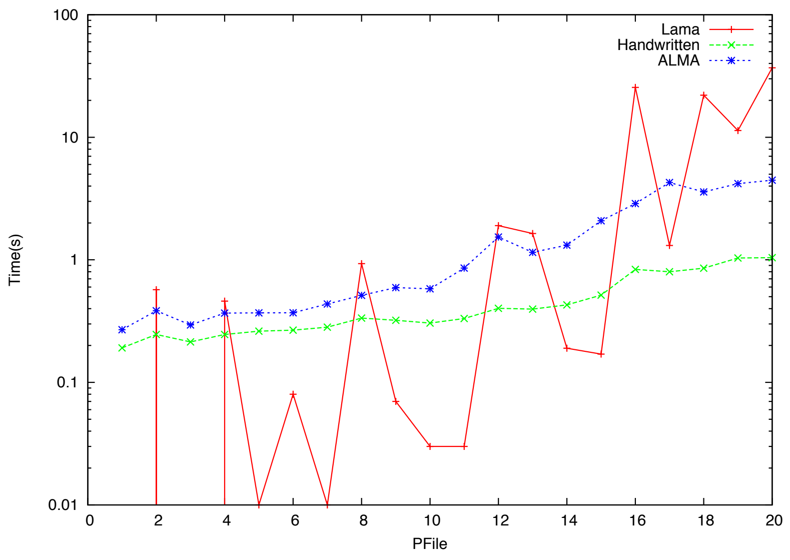


Figure 9.1: Time results for ALMA and handwritten strategies for Driverlog problems.

We have presented the time plots in Figures 9.1 and 9.2; the quality of the plans are similar. In each case the time taken for the ALMA solutions are longer. The plots for Goldminer demonstrate the trade-off provided by the ALMA vocabulary. The bags

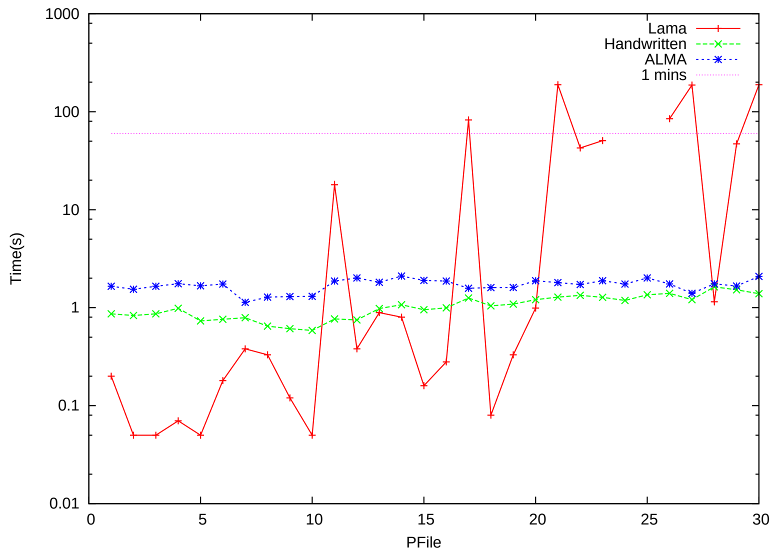


Figure 9.2: Time results for ALMA and handwritten strategies for Goldminer problems.

used by the strategy include opening actions, and therefore extend the reachability analysis from the move action graph. There is an increase in planning time, however, this is compensated for by the reduced effort in defining the solver. The Driverlog plots demonstrate that as the map sizes and the number of traversers increase, the difference between the two approaches increased. This growing difference is not observed in Goldminer after problems 10 and 20, where the grid size increases. However, in Goldminer there are relatively few (one in these experiments) states where bag expansions will cover large parts of the grid. It is promising that although we are using a general solver, all of the problems in Driverlog (including several that are not solvable by FF) are solved in less than 5 seconds and the longest Goldminer time was 2.1 seconds, in a domain where FF only solves 2 problems.

9.2.2 Target and state significance

The target significance filtering technique, presented in Section 7.5, is one approach that we have investigated that can reduce the complexity of the bag expansion problem. This filtering approach expands the ALMA bags under the assumption that alternative states for a specific target can be pruned. An ALMA can define a combinatorial search space leading to a vocabulary that is intractable to compute. We now analyse the impact that using this filtering technique has on the computation time of the vocabulary, using planning time as a proxy. We have included runs using SbS, as this evaluates the policy at each step, providing more evidence of the filtering effect.

Setup

In this analysis the Goldminer and Traverser domains are used. The Bootstrap and Target Goldminer problems from the Learning Track of IPC-08 are used. For the Traverser domain we have generated two training sets. In the first each problem has a single traverser, but the number of locations is increased from 100 to 1500. The other set has 100 locations and the number of traversers is increased from 1 to 50. Each run is given a time-out of 10 minutes.

Expectations

The filtering approach should have little effect on the quality of a solution in these domains. The bags for the Goldminer domain include a bag that can open locations and move into them. The state space of this bag grows very quickly with the state space. This filtering technique is applicable to this bag and therefore we would expect an improvement in the time to solve problems. The move action in the bag of the Traverser problem can only move the traverser between locations. As a result we would not expect much of an improvement.

Results

The time plots for the Goldminer bootstrap problems are presented in Figure 9.3. As we have observed in Chapter 6, the macro application approach leads to a reduction in planning time. There are sharp increases in the state significance plots at problem 15. This coincides with an increase in the grid size from 3×3 to 4×4. This suggests the approach is highly sensitive to the map size, which is in line with our expectations. This is further supported by the fact that the state significance approaches were not able to solve any of the target problems, which have larger grids (5×5 to 7×7). The filtered approach solves each of the target problems in one or two seconds.

The time plots for problems that increase in the locations dimension are presented in Figure 9.4 and for the number of traversers dimension in Figure 9.5. The plots indicate a modest improvement for the filtered approach over unfiltered. There appears to be a slow growth in the cost with increases in locations or traversers. This result is perhaps not surprising, as the bookkeeping required to store visited states and the

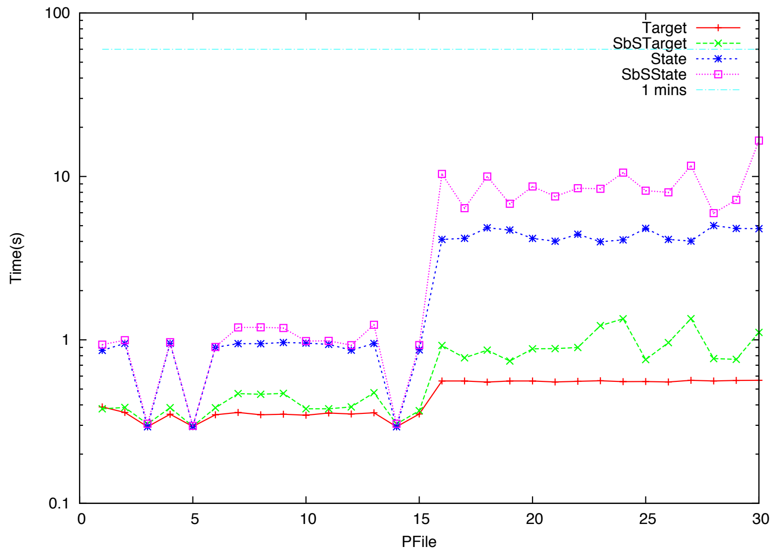


Figure 9.3: Time results for the state and target significance using both step-by-step and macro application approaches for Goldminer bootstrap problems.

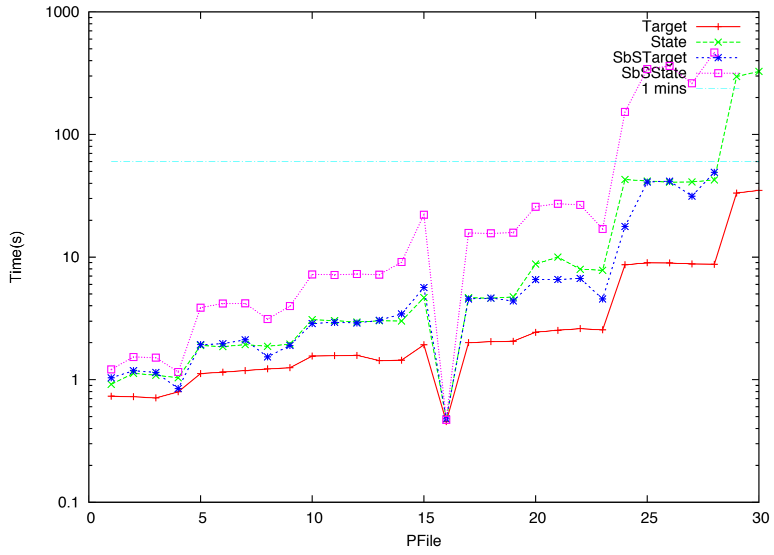


Figure 9.4: Time results for the state and target significance using both step-by-step and macro application approaches for Traverser problems with increasing sizes of maps.

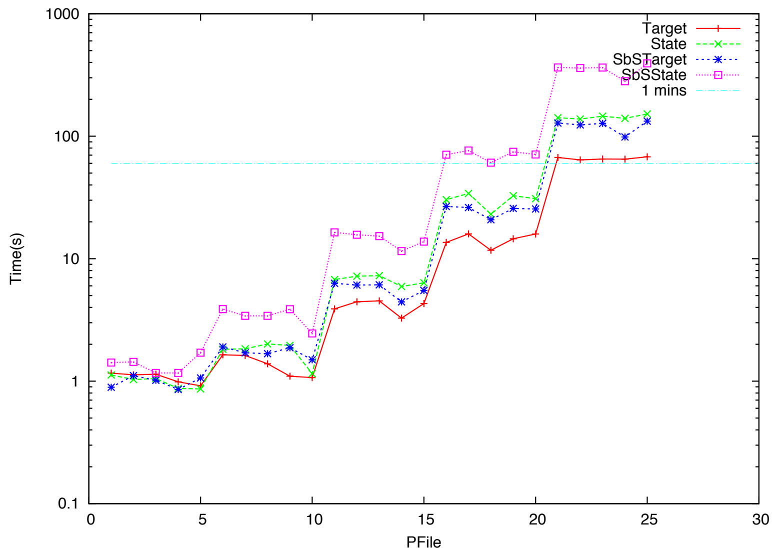


Figure 9.5: Time results for the state and target significance using both step-by-step and macro application approaches for Traverser problems with increasing numbers of traversers.

subsequent comparisons can be performed more efficiently. It can also be observed that the final problems are not solved. This demonstrates the combination of the symmetry issue that we discussed in the context of Gripper problems and the expansion cost of determining the reachability graph for each traverser in every state.

These results support the use of the filtering technique when possible. Although we have presented a complete formula for determining when it can be used, we do not present an approach to evaluating this formula. Our use of this filtering approach is therefore determined through an incomplete process and can therefore lead to an incomplete reachability analysis.

9.2.3 Alternative graph definitions

In this part we discuss how the ALMA representation generalises to other traversal problems. The results are summarised in Table 9.2. In Long and Fox (2002), two generalisations of the traversal problem are noted as not being reducible to a single traverser single map. One of these is traversal problem that involve more than one move actions. The other is where there are more states in the traversal property space, which means that the traverser can escape from being located. There are two variants of

Table 9.2: A summary of the proposed properties for an extended collection of problems. The table records whether certain properties holds *with* and *without* sequence breaking rules. The properties are: target significance (Target$_{Sig}$), directed connectivity (DC), and appropriate for expressing a policy (π). In the policy column, *individual target* is used to indicate that the solution for a specific goal can be captured.

Feature	Target$_{Sig}$	DC	π
Blocking traversers	no	with/without	individual targets
Joining maps	with/without	with/without	with/without
Flying traversers	with/without	with/without	with/without

this: when the traverser must resume the same location (ability of hovering) and when the traverser can resume a different location (ability of flying). We also introduce a third problem, which involves more than one traverser, which we call *interfering traverser* problems. In these problems the map locations are blocked by traversers (and other objects) and coordinating traverser movement is necessary. We will examine each of these generalisation in the context of directed connectivity and target significance.

In interfering traverser problems, the movement of the traverser is dependent on the position of other traversers (for example, in the Airport domain). A more relaxed binding constraints could be used so that sequences of causally related move actions were explored. In particular, we would not constrain the SIs to be chains of the same traverser. The result would be bags that established directed connectivity in domains such as Sokoban, $(n^2 - 1)$-Puzzle, and the Airport domains. However, the vocabulary would not be target significant and would involve expanding the state space to evaluate each bag. Further work is required in order to investigate these problems in the context of the achieved goal context, as these problems can involve dependent goals (for example, squares in $(n^2 - 1)$-Puzzle).

Alternatively, the movement of a traverser could be conducted using several move actions. For example, in the Bulldozer domain (Long and Fox, 2002), the traversers (mobiles) can traverse using the `drive` and `cross` actions. To establish a directed connectivity solver over this network, the move actions that act on each of these structures would be required and the templates and binding constraints would be defined between each move action. The presence of additional states in the traverser's property space can be treated for the cases identified. No special treatment is required for the case where the traverser temporarily loses locatedness and regains it at the same position. If the traverser's locatedness can have changed then the same alteration as for several move actions can be made. The macros will include the actions necessary to change between the different graphs. The identification of these graphs is described in Long and Fox (2002) and this analysis could be used to propose the collection of templates, in the same way as for a single map. The combination of maps does not indicate the loss of target significance. In this way an effective vocabulary would be defined for these domains.

9.2.4 Discussion of the ALMA solver

The main benefit of the ALMA solver is that it greatly reduces the effort required to define SIs solvers. In this section we have demonstrated the use and analysed the efficiency of our ALMA solver. We have demonstrated that ALMAs can be used to support RBPs through SIs in several domains. The difference between the plots for handwritten and ALMAs based solutions provide some indication of the opportunity for improvement, opening up many avenues of future work. We have provided support that our representation is appropriate for expressing SIs vocabulary in many domains. In situations where target significance and directed connectivity are not established, the model can be extended so that heuristic guidance can be exploited within a more general hierarchical framework of the solver (Appendix G.1). However, this framework compensates for a more fundamental issue of the rule language that we have used. A similar effect can be obtained using a disjunctive expression and the delegation of the choice of bindings to the heuristic. This approach provides the most control to the RBP.

9.3 Learning parameters

In this section we present the training data and L2PLAN parameters that were used to learn RBPs. We used a Linux box with 18 Intel Xeon E5 CPUs, clock speed: 2.40GHz. The learning setup and training data are detailed here.

9.3.1 Learner setup

In this work we use a parameter set that has been used in previous experiments using L2PLAN. These parameters are presented in Listing 9.2. The candidate selection approach is aggressive as it is likely to pick from the higher scoring policies. This is enforced through passing the best 5 policies as candidates for the next population. The parameters for crossover and mutation have been established through experimentation in previous work. It was outside the scope of this work to fully explore the learner and its possible parameter sets.

```
GA parameters:
Candidate selection:
    Tournament selection: Best of 50 choices
Population size: 100
Number of replicants: 5
Crossover rate: 0.7
Crossover elitism: false

Mutator parameters:
Mutation rate: 0.1
Rule mutation rate: 0.3

Initialisation parameters:
Min rule conditions: 3
Max rule conditions: 5
```

Listing 9.2: The used settings for L2PLAN

Local search

The general parameters that control the local search from L2PLAN, are the number of neighbours of each policy that are generated, and the number of local search steps that are made (given an improving policy was found). We used similar parameters from previous experiments with L2PLAN; we present these in Listing 9.3.

```
Local search parameters:
Number of candidates: 10
Number of levels: 2
```

Listing 9.3: The used settings for the constrained local search

In previous work with L2PLAN, the neighbourhoods only effected the conditions, however, we have introduced several neighbourhoods, inspired by Aler et al. (2001), that open up the different aspects of the search space.

- Add a single predicate from a rule's condition. This can be a goal predicate, a negated predicate, both or neither.

- Remove a single predicate from a rule's condition.
- Merge two variables. Selects two compatible variables in a rule and merges them. The remaining variable takes the merged variable's place in any predicates.
- Spilt a variable into two. Selects some of the predicates of a variables and replaces the original variable with a new variable.
- Replaces a rule in the policy with a rule from another policy.
- Adds a rule from another policy into a random point in the policy.
- Swaps two rules in policy.
- Lifts the type of a variable if its predicates allow.
- Lifts the type of a variable and deletes predicates to allow.
- Makes the type of a variable more specific.

There is an important relationship between the neighbourhoods and the fitness function and we have found that these neighbours are effective in practice.

Fitness function

We used the fitness function developed in Section 8.2. If the policy does not suggest an action then roll-out fails. Our tests indicated that using a random backup led to worse learner performance. This could be because sequences of random backup actions can add promising sequences on bad policies, guiding the learner to finding states that happen to lead to better random sequences. The time-limit on running the planner and unrolling each policy was set to 10 seconds. We limited roll-out to the length of the optimal plan. However, we only counted abstract actions as a single step, and therefore solutions could have arbitrary more actions. This further promoted using the vocabulary and provided a method that scaled the size of the roll-out with the structure sizes. As we have not experimented with enriching actions in this chapter, this is consistent with evaluating the policy expressed in the enhanced language.

9.3.2 Training data

Sets of problems were generated for each domain. As we discussed in Chapter 8, the problems used to learn had to be small so that h^{LM-cut} could solve them, but exhibit the important domain structures, to provide the opportunity for learned solutions to generalise to larger problems. Following the methodology used for the learning track, we created a small *bootstrap* collection of training data to learn the RBPs and a larger set of *target* problems to validate the learned knowledge. Each of the problems in the bootstrap problems were solved using an optimal planner and the states visited on application of the resulting plans were sampled to form an example size of 50, for Gripper and Traverser and 100 for each of the other domains. The smaller sets were used in the two domains that have less structure to explain.

Blocksworld

The competition generator was used to generate 30 problems.

Driverlog

The competition problem generator was used to generate 30 problems. We used a collection of parameter sets to generate these problems, although all of them were small problems and were quickly solvable with h^{LM-cut}. Goals were generated for the truck, driver and package by the generator, which can include empty goals for objects (this is more likely for the trucks and drivers).

Goldminer

The 30 bootstrap problems from the learning track of the 2008 IPC were used to learn the policies. These problems have a single robot, bomb and laser and small grids (half 3×3 and the other half 4×4).

Gripper

For Gripper we used problems that required between 1 to 8 balls to be moved between rooms.

Structured Briefcase

The Driverlog domain was modified by removing the drivers and the path map and supporting predicates and actions. The Driverlog competition problem generator was changed appropriately, maintaining all other properties. We used this generator to create 30 problems for small parameter sets.

Traverser

The Driverlog domain file was altered by removing drivers, paths and the added driver locations and packages. The resulting problems involved moving the trucks to goal locations on a map. The generator was changed to reflect these changes. 20 problems were generated with 1 or 2 trucks and between 10 and 20 locations.

9.4 Validating $\mathbb{M}|_{\Sigma_i}$

We have used the algorithm to generate several RBPs in two configurations. In the first analysis we have run the learner from a randomly selected initial population. The learned control knowledge was then used to solve the benchmark problems, using the setup described in Chapter 6, except we ran the tests on an Intel Xeon E5 CPU, clock speed: 2.40GHz, and limited the time to 10 minutes, in line with typical practice of the learning competitions. The results provide support that the fitness function is suitable for guiding search towards effective control knowledge. The second analysis seeds the initial population with policies that were generated using the process described in Chapter 8. This has allowed us to discover RBPs in domains with more complex structure. We demonstrate that as the fitness of the population increases, the performance of the corresponding RBPs improves. To provide an overview we have summarised the results in tables. The plots are presented in the Appendices, Section F.4. Although the results in this section provide some evidence that the fitness function is appropriate for guiding search, the main aim is to validate the developed models. In these evaluations the nearest neighbour solver heuristic is used with the graph abstraction and the ALMA solvers.

9.4.1 Learning from randomly initialised populations

We have selected the Blocksworld, Gripper, Structured Briefcase and Traverser domains for testing the learning system. These domains are relatively small, but together they test each of the aspects of the system that have been developed in this work. We use two domains that are similar to domains used in experiments using L2PLAN to validate our fitness function. The Traverser domain requires directed connectivity to move traversers to their goals, which will require the extended vocabulary to be used. The problems from the Structured Briefcase domain have more interesting structures and require optimisation over package deliveries. We have selected this domain as it draws together the requirements of directed connectivity that we have discussed in Chapters 5 and 7, as well as the optimisation over these concepts that we discussed in Chapter 8.

Setup

For each domain L2PLAN was used to generate a policy. We used 50 problems for the Gripper and Traverser runs and 100 problems for the Blocksworld and Structured Briefcase runs, as described in Section 9.3. We used a randomly generated initial population of 100 policies in each case, and used the learner parameters as discussed above. In the case of Blocksworld and Traverser an appropriate domain model was selected from our solver library using domain analysis. A `GraphAbstraction` solver was invoked for the Traverser domain and a `BuildingStructure` solver was invoked for the Blocksworld domain. In the Gripper and Structured Briefcase domains, the language was generated using the bag generation, evaluated in Section 9.1. The bags each contained singleton `move` macro actions (with the appropriate label for the domain). The learning time varied from 1 hour for an optimal policy in the Traverser domain, to almost a week for a Structured Briefcase policy with 99% fitness.

Expectations

The learned policies have all achieved high fitness scores, which means they have already demonstrated the ability to solve the planning problems for problems in the training set. Of course these are small problems and we have used difficult problems to evaluate the RBPs. The policies' performances on the testing sets provides an indication of how well the fitness function is informing search. A likely negative result is if the policies' over fit the training sets then they will provide patchy guidance leading to many backups and probable planning failure. An alternative is that the control knowledge relies on properties of the training set, such as small maps that are more amenable to random search.

Results

Table 9.3: Quality (Q), Time (T) and Coverage (C) results for FF (FF), Lama (L), Handwritten (H) and a learned policy (π_L)

Domain	FF_Q	FF_T	FF_C	L_Q	L_T	L_C	H_Q	H_T	H_C	π_{L_Q}	π_{L_T}	π_{L_C}
Blocksworld	1744	24.86	17	12304	1820.9	39	7140	35.472	**45**	7100	41.682	**45**
Gripper	**32240**	**50.98**	20	**32240**	193.72	20	**32240**	803.954	20	**32240**	930.07	20
SBriefcase	2796	1450.73	15	3810	310.79	20	2717	**71.601**	20	**2586**	452.512	20
Traverser	**1004**	**0.78**	30	791	7.49	28	**1004**	14.576	30	**1004**	14.576	30

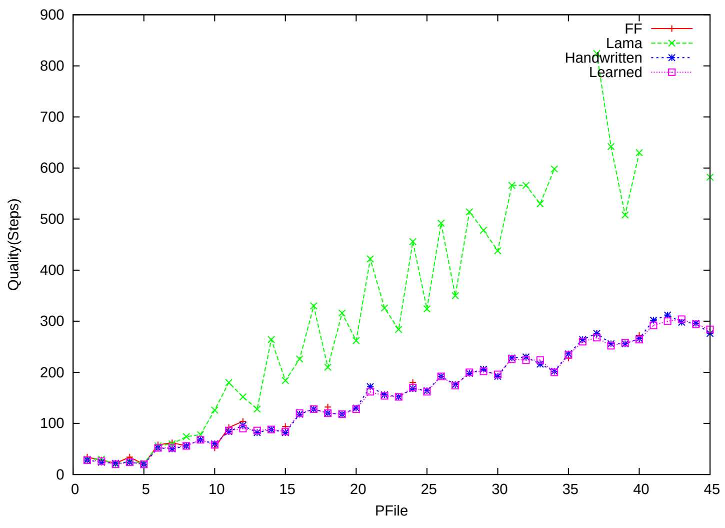


Figure 9.6: Quality results for a learned policy on Blocksworld problems

The results presented in Table 9.3, demonstrate the success of the learned control knowledge to control search in the domains. The learned RBPs solved all of the problems in each of the problem sets, whereas the domain independent planners (LAMA and FF) solved all the problems in two of the four domains. In each of the domains, the lengths of the plans generated by the learned policies were the same or shorter than for both the domain independent planners and the handwritten policies. Figure 9.6, illustrates this for the Blocksworld domain. In terms of time, both the learned and handwritten RBPs perform worse in Gripper and Traverser domains. These are domains with lots of symmetry, which can lead to inefficiency in the rule matching machinery, as was discussed in Chapter 6. The time results in Blocksworld (Figure 9.7) for the RBPs are better than LAMA and more consistent than for FF. In Structured Briefcase the learned RBP, generated plans slower than LAMA; however, the handwritten RBP

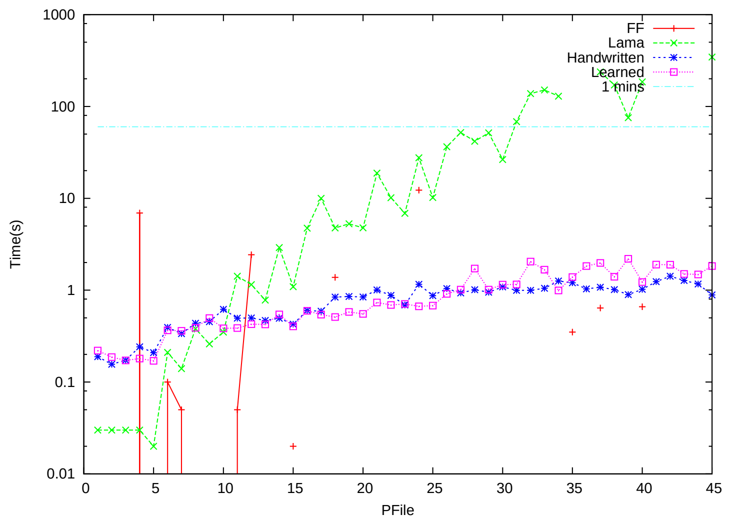


Figure 9.7: Time results for a learned policy on Blocksworld problems

generated plans quicker. The handwritten RBP exploited a more efficient solver, which could be used with the learned policy (without changing the policy).

Traverser is a relatively simple domain, however, the final problem involves moving 50 traversers on a map of 100 locations. This demonstrates the validation of the provided vocabulary by the policy learner and the successful exploitation of the enhanced model in learned control knowledge. The Gripper domain does not rely on any extended vocabulary, however, it demonstrates the learner's capability of discovering solutions with several actions. The results for the Gripper domain (related to the Briefcase domain) and the Blocksworld domain confirm that L2PLAN, using our alternative fitness function, can learn solutions for the domains previously reported (Levine and Humphreys, 2003; Galea et al., 2009). In Structured Briefcase our handwritten policy orders package pickups before package drop-offs. The learned policy has these rules, but also includes more specialised rules that allow drop-offs of packages to locations with other packages at them (no conditions on these packages). The results indicate that this leads to shorter plans on average. Structured Briefcase problems involve optimising over package deliveries. From our experience, similar runs using the L2PLAN fitness function (and many parameter settings) have not learned generalising control knowledge in this domain. For example, we ran 15 runs of the L2PLAN, using the learner parameters, the local search used in this work and the generality and conciseness aspects of the fitness function (as well as the standard L2PLAN fitness function).

We tested the learned policy with the highest fitness value (98%) from all the runs. It solved all of the problems used to make the training set, requiring random backups actions in only two problems. The policy solved none of the problems in the testing set. Learning RBPs that control search over maps directly has not been reported previously. Moreover, the quality of the solutions generated is very high, demonstrating that the selected planning model was appropriate for learning RBPs.

9.4.2 Generated seeds

In the following subsection we seed the learning process with RBPs generated using the process presented in Section 8.4. We limited the enhancements to the ALMA solvers. In this part we summarise our observations of the generated policies. We have generated seeds for Traversal, Gripper, Structured Briefcase, Goldminer and Driverlog, using the same training data sets that we described in Section 9.1. We focus on Structured Briefcase, Goldminer and Driverlog as the aspects of the other domains are contained within these three. There were 6 seed policies generated for Goldminer, 4 for Structured Briefcase and 15 for Driverlog. We have included an example for these domains in Appendix D.4.

Goldminer plans capture a single thread of activity, which moves the robot towards the goal. We have noted above that the generated bags can be used to support an RBP to control search and in Section 8.4 that the important "two-steps-from-gold" proposition is uncovered through the generation process. The rule includes parts that establish aspects of the current state, such as the current locations of the robot and gold: (`at` *?robby ?f2-0f*), (`at-gold` *?f3-3f*). Important requirements of the later sequence are identified, for example, that the bomb is not destroyed, or that the robot does not hold the laser: (`not` (`holds` *?laser ?robby*)) (`not` (`destroyed` *?bomb*)). These include conditions on predicates computed by ALMA solvers: (`not` (`pickup-gold67_connected` *?robby ?f2-0f ?f3-3f*)), where the current location of the robot is the starting point. These examples are extracted from one of the final rules from a generated policy. The complete policy is presented in Subsection D.4.2 of the appendix. The generated policies contain the important propositions and behaviours necessary to solve the problems in Goldminer. In fact, the seed presented in the appendices can solve the problems in the testing set.

The generated policies for Structured Briefcase and Driverlog capture many of the key aspects of the problem. For example, the control knowledge captures behaviours such as moving to pickup a misplaced package, which is a key concept in these domains. We noted that the one-to-one mapping used in the generation process is quite restrictive and it does lead to separating solutions that could be used together. For example, the proposition, (`!=` *?s8 ?s2*), from Rule 14, the `board-truck` rule in the seed policy, requires that the truck's starting point, *?s2*, is different from the package goal, *?s8*. This condition restricts the applicability of the rule with an unnecessary proposition. The rules capture a single delivery, where delivery problems will often require several deliveries to be made. In Structured Briefcase the changes that are required to generalise these rules are small. One change is to move the pickup and drop off actions to have higher priorities. The heuristic to order the rules from the last plan action to the first generates intuitive policies. Some of the first rules of the Driverlog seeds move the driver home, and the condition on this requires a single package to be home. The limited rule language require that this rule is moved to a lower priority than the package deliveries.

The seeds that we generate provide an example of establishing directed connectivity in a domain and can capture a complete solution for the problem. This is particularly effective in domains where the main change is in the specific SIs, such as Goldminer. The process generates effective policies for the Goldminer, Traverser and Gripper domains and examples of vocabulary use in important situations in the Structured Briefcase and Driverlog domains. The rules can be overly restrictive with respect to variable bindings and can fail to generalise over sub-problems (such as multiple package deliveries). These limitations are addressed in the following subsection.

9.4.3 Learning from seeds

We have selected two more challenging domains to evaluate the potential of the approaches that we have developed in this work. Driverlog and Goldminer are challenging domains for both domain independent and domain dependent approaches. In these problems an effective strategy must reason about different SIs and how they integrate. The Driverlog problems involve complex optimisation problems over the SIs. The learned policies are presented in Appendix D.5. We also present the results of using a seeded approach to learn a policy for the Structured Briefcase, so that this can be compared with the randomly initialised approach reported above.

Setup

The policies were generated as described in the previous subsection. For each domain L2PLAN was used to generate a policy. We used 100 example problems for each domain, as described in Sections 8.2 and 9.3.1. We used the seeds in the initial population and filled the rest (up to 100 policies) with randomly generated policies. The learner was parameterised as discussed above. The domain models were generated during the generation of the policies and these models were used for learning. It is interesting to note that the seeded run for Structured Briefcase reached 97% after 8 hours and reached a plateau after 16 hours. The Driverlog run was stopped after ten iterations of plateau with a score of 97% fitness, after 9 days. For Goldminer a policy of 99% fitness was learned in around 40 hours.

Expectations

The previous section supports some confidence in the fitness function and the high fitness scores over the training data indicate that the policies are able to plan in these domains. We have provided some examples of using the vocabulary in the seeds provided. In Goldminer these examples should provide strong support. In Driverlog there are many details that interact and will lead to the seeds being less informative. The main source of uncertainty comes from the small training problems that had to be used to allow the optimal planner to solve the problems within a reasonable time.

Results

Table 9.4: Quality (Q), Time (T) and Coverage (C) results for Lama (L), Handwritten (H) and a policy learned using seeds (π_S)

Domain	FF_C	L_Q	L_T	L_C	H_Q	H_T	H_C	π_{S_Q}	π_{S_T}	π_{S_C}
Driverlog	16	1117	103.22	20	**930**	**9.308**	20	1058	96.016	20
Goldminer	2	884	896.75	28	814	31.37	**30**	826	76.943	**30**
SBriefcase	15	3810	310.79	20	2717	**71.601**	20	**2519**	438.253	20

The results are presented in Table 9.4, along with the coverage figures for FF, and the results for LAMA and the handwritten policies. The results indicate that the policies have provided effective guidance in each of the domains. The learned RBPs solved all of the problems in each of the problem sets, whereas LAMA did not solve all of the Goldminer problems and FF did not solve all of the problems in any of the domains. In each of the domains, the lengths of the plans generated by the learned policies are typically the same or shorter than for LAMA and FF. The quality is better for the handwritten policies in Driverlog and Goldminer and better for the learned policy in Structured Briefcase. The quality results for Driverlog, presented in Figure 9.8,

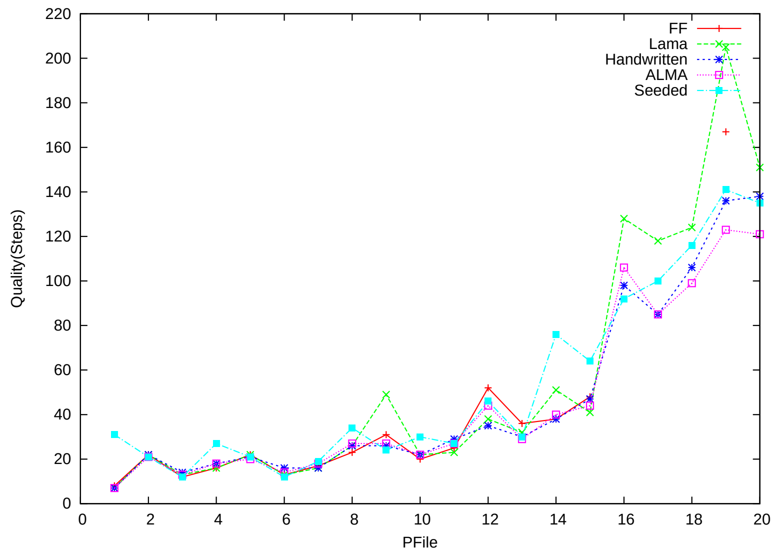


Figure 9.8: Quality results for a learned-from-seeds policy and an ALMA based policy on Driverlog problems

illustrate a small number of longer plans, which are discussed below. In terms of time, the learned RBPs perform better than LAMA in Driverlog and Goldminer. The slow generation of plans in Structured Briefcase is partly due to the use of the ALMA solver. The learned RBPs are slower than the handwritten RBPs; however, these are using more efficient solvers. For example, Figure 9.9 plots the time taken for the handwritten policy using an ALMA solver (labelled ALMA) and this explains part of the difference between the two lines.

There are no previous reports of learning effective unassisted RBPs for Driverlog and Goldminer in the literature. The most related for Goldminer is reported in (de la Rosa and McIlraith, 2011) and learns pruning rules. The authors observe that the approach provides no guidance through the SIs and as a result the quality of the solutions is poor (relative to LAMA). For Driverlog, the use of an RBP as a probe within a best-first search (using the relaxed plan heuristic) was reported in Yoon et al. (2008). The final 5 problems were used as the testing set and their approach solved 4 of the problems. An average plan length of 177 steps was reported, where our average (over the 4 problems) was 112.25, and we are not using a global heuristic to guide our search. Our approach also required considerably less time to generate plans.

In order to achieve these results, the learned policies must capture effective strate-

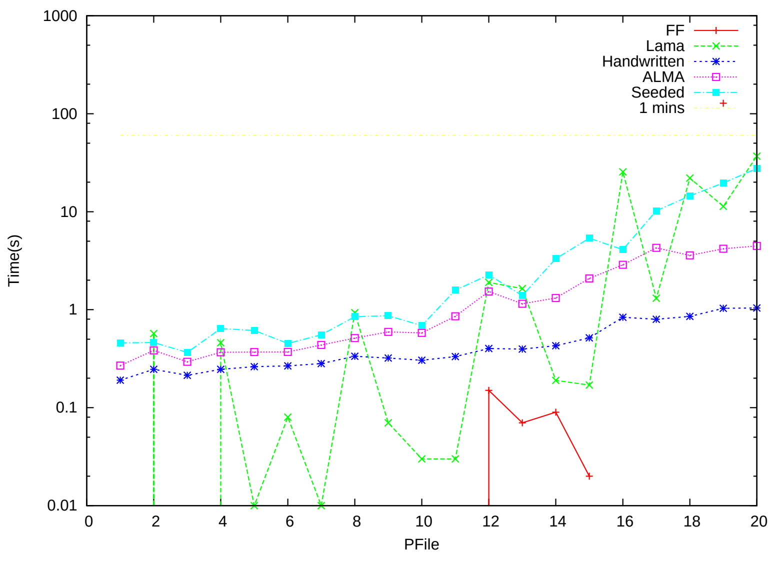


Figure 9.9: Time results for a learned-from-seeds policy and an ALMA based policy on Driverlog problems

gies for these domains. Moreover, as we have discussed, the learned policies must exploit the enhanced domain model. The rule in Listing 9.4, opens a path towards the gold, by establishing the "two steps to gold" proposition:

(`at-gold` *?f0-3f*) (`connected` *?f0-2f ?f0-3f*) (`connected` *?y ?f0-2f*)

And also exploiting the `FireMoveBag_move` action to select the necessary move and open actions to form a path towards the gold.

Listing 9.4: Rule for opening a path to two steps from the gold.

```
(:rule
OpenPathTowardsGold :parameters ( ?x - loc ?bomb - bomb ?y - loc ?f0-2f - loc ?laser
    - laser ?f0-0f - loc ?f0-3f - loc ?r - robot)
  :condition (and (holds ?laser ?r) (at ?bomb ?f0-0f) (not-clear ?f0-2f) (connected ?
      f0-2f ?f0-3f) (no-hard-rock ?f0-2f) (clear ?f0-0f) (at-gold ?f0-3f) (
      no-hard-rock ?f0-3f) (at ?r ?x) (not-clear ?f0-3f) (no-gold ?f0-2f) (connected
      ?y ?f0-2f) (soft-rock-at ?f0-2f) (soft-rock-at ?f0-3f) (not (at ?laser ?y)) (
      not (arm-empty ?r)) (not (destroyed ?bomb)) (not (at ?r ?f0-2f)) (not (clear ?
      f0-2f)) (not (holds ?bomb ?r)) (not (holds-gold ?r)) (not (clear ?f0-3f)) (not
      (no-gold ?f0-3f)) (not (no-soft-rock ?f0-2f)) (not (no-soft-rock ?f0-3f)) (not
      (at ?r ?y)) (not (at ?r ?f0-3f)) (!= ?y ?f0-3f) (!= ?y ?f0-0f) (!= ?f0-2f ?
      f0-0f) (!= ?f0-2f ?f0-3f) (FireMoveBag_connected ?r ?x ?y) (MoveBag_connected ?
      r ?x ?f0-0f) (!= ?f0-0f ?f0-3f) (!= ?y ?f0-2f) (not (BombMoveBag_connected ?r ?
      x ?f0-3f)) )
  :goalCondition (and (holds-gold ?r) )
  :action (FireMoveBag_move ?r ?x ?y)
)
```

Although this rule has a large antecedent, several of these are redundant and could be removed in a post-process.

Through analysing the policies, it can be seen that they contain all of the rules necessary for a complete strategy for solving the problems from the domains. It is important to note that the vocabulary available to the learner was automatically generated and the policy is therefore using the ALMA solver, impacting on the planning time. The plots in the Appendix, Section F.4 can be used to compare the learned policy with the handwritten ALMA solution. The use of the ALMA solver does explain some of the difference in the time results for the learned and handwritten solutions. However, the control knowledge is not perfect. In particular, the Driverlog learned policy results include some points that relied on the random backup action. As an example, in the first Driverlog problem, there are two packages, drivers and trucks, on a map with 3 truck locations. The problem requires that one truck is moved home and a driver is moved to their goal (the packages are at their goals and the other objects do not have goals). The most appropriate rule (from the learned policy) for this situation is presented in Listing 9.5.

Listing 9.5: Rule with irrelevant conditions, for moving the driver to a misplaced truck

```
(:rule walkToBoardMisplacedTruck
  :parameters ( ?driver - driver ?truck1 - truck ?package1 - obj ?s4 - location ?
      loc-from - location ?s14 - location ?loc-to - location ?s13 - location ?s3 -
      location ?s7 - location)
  :condition (and (at ?package1 ?s13) (empty ?truck1) (at ?package1 ?s7) (at ?truck1
      ?loc-to) (at ?driver ?loc-from) (not (in ?package1 ?truck1)) (not (at ?truck1 ?
      s3)) (not (at ?package1 ?s4)) (!= ?s13 ?s4) (!= ?s14 ?loc-to) (!= ?s7 ?loc-to)
      (!= ?loc-to ?s4) (!= ?s3 ?s13) (long_walk_connected ?driver ?loc-from ?loc-to)
      (not (long_drive_connected ?truck1 ?loc-to ?s4)) )
  :goalCondition (and (at ?package1 ?s7) (at ?truck1 ?s14) (not (at ?driver ?loc-to))
      )
  :action (long_walk_move ?driver ?loc-from ?loc-to)
)
```

This rule demonstrates the use of the `long_walk_move` action to abstract the path graph. This rule is appropriate for situations where a truck is misplaced and a driver must be moved to drive it home. However, it has irrelevant conditions. This behaviour is not a common requirement in the small training set problems and therefore this rule might have been over-fitted. In the first state, the only package goal that can bind with (`at` *?package1 ?s7*), is *?loc-to*, conflicting with (`!=` *?s7 ?loc-to*). However, this rule will cover many of the situations where moving towards a misplaced truck is necessary, especially as the problem sizes increase. The number of backups required is presented in Figure 9.12, which demonstrates that the number of backups required in bigger problems is small. This is very promising, as the backups actions are randomly selected and are unlikely to contribute towards the goal in these bigger problems, suggesting that small changes of state are all that is required. This suggests that a larger training set might lead to these rules being more precisely formed.

9.4.4 Fitness and performance

In the previous analyses we have plotted the scores for the policy with the highest fitness that was found during learning. In this subsection we have used some of the RBPs that were generated during learning to demonstrate the relationship between the fitness function value and the performance of the associated RBP. We have used 3 policies, which were the best of a particular population, to solve the test problems. We present the results for Driverlog, as there was more learning for this domain.

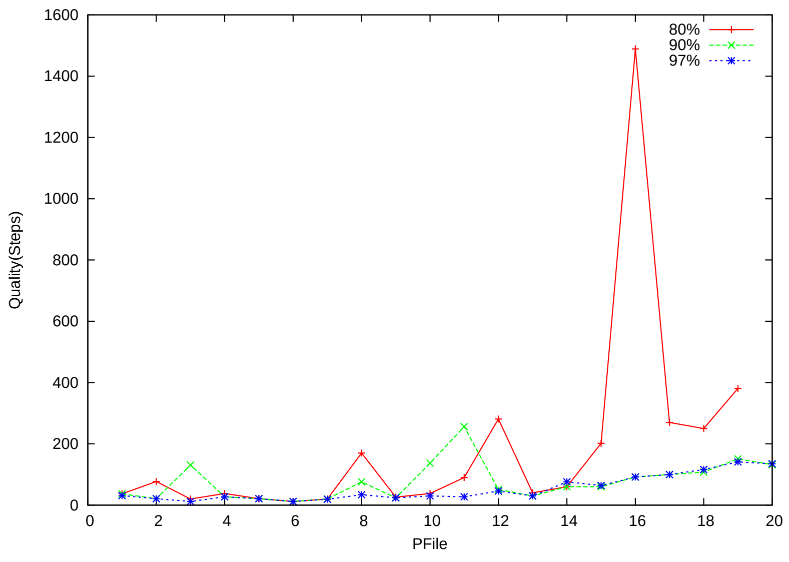


Figure 9.10: Quality results for three policies with different fitness values on Driverlog problems.

Figure 9.10 plots the quality, Figure 9.11 plots the time and Figure 9.12 plots the number of random actions for the policies. The results provide a clear pattern that links higher fitness values with improved planning performance.

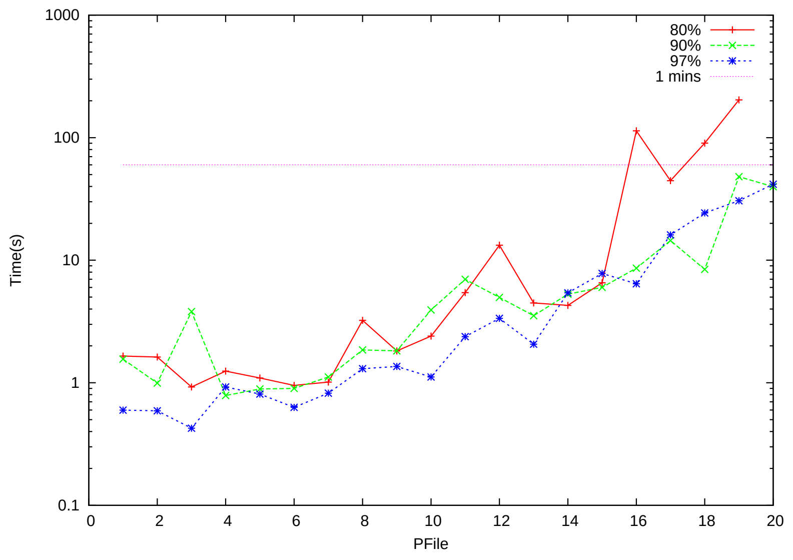


Figure 9.11: Time results for three policies with different fitness values on Driverlog problems.

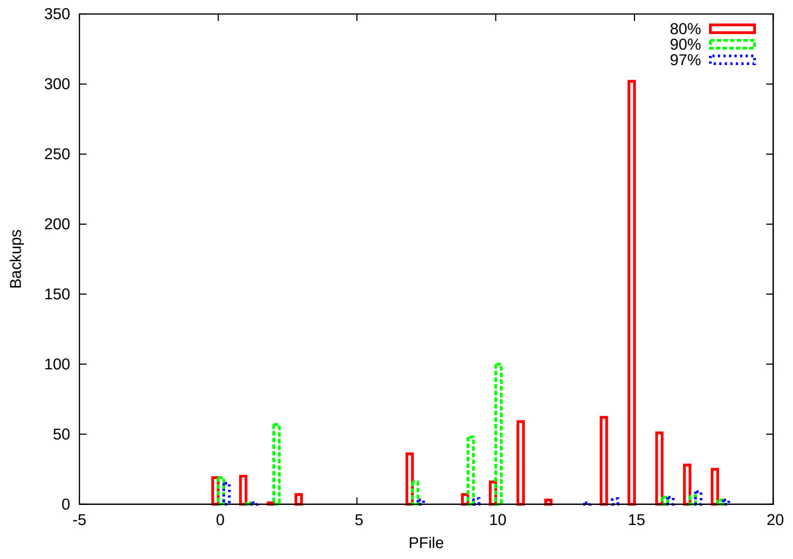


Figure 9.12: Number of random backup actions required for three policies with different fitness values on Driverlog problems.

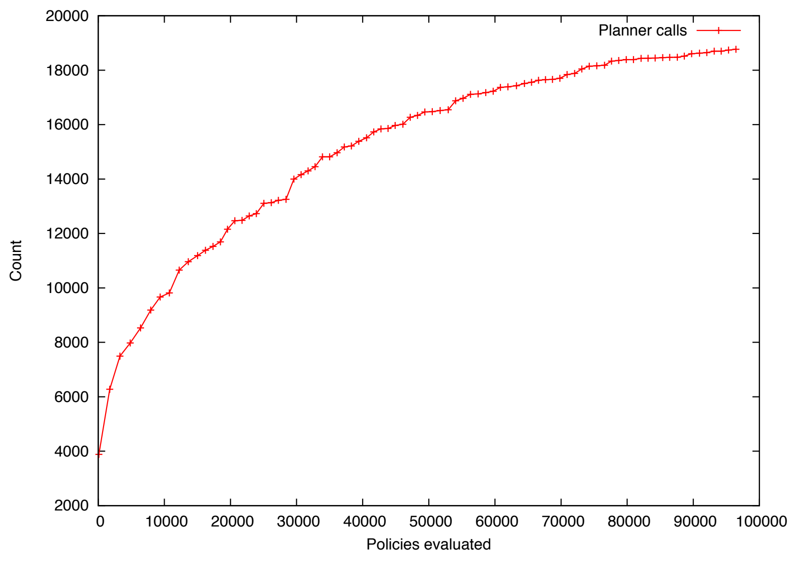


Figure 9.13: The number of h^{LM-cut} calls against policy evaluations during learning of Blocksworld RBPs

9.4.5 Discussion of the learning approach

The learning approach that we have used in this work is expensive. The caching aspect greatly reduces the number of calls to the planner and as a consequence reduces the time taken for each iteration. We have found that the cache size grows quickly and then grows slowly or settles. Figure 9.13, illustrates the relationship between the number of policy evaluations and the number of planner calls during learning the Blocksworld policy. The reducing number of planner calls was exhibited in each learner run. Our main program was run with 1Gb of memory and it did not run out of memory.

The main bottleneck in our approach comes from rolling the policies out. This is demonstrated by plotting the accumulated time used by the planner and policy roll-out after each iteration. This is presented for the Blocksworld run in Figure 9.14. Similar graphs were generated for each learner run and each shared the same relationships. We implemented a framework to unroll policies within the learning system. However, in order to reduce the amount of work required to maintain a learning and planning version of each solver as the rest of the system was developed we determined to write out the policies and compute their mappings in a separate process. This allowed us to use the planning version in each case. While this has greatly increased the flexibility, it has greatly increased the length of time to roll-out a policy.

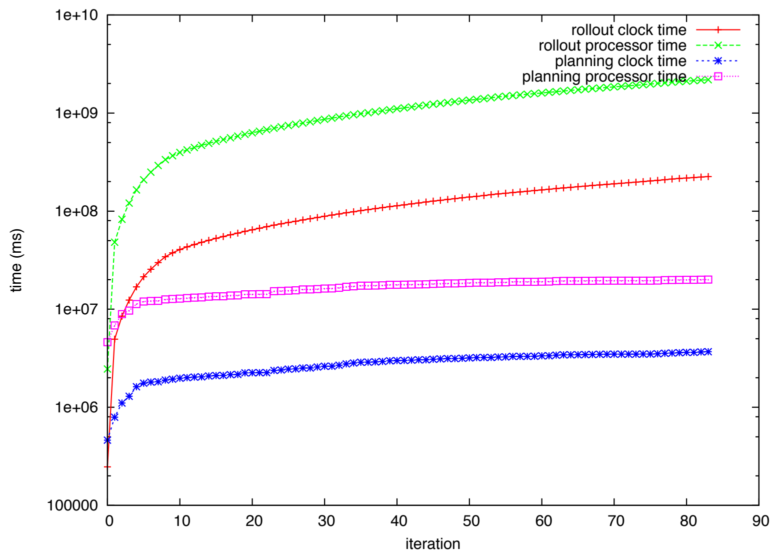


Figure 9.14: The accumulated time for policy roll-out and planning in Blocksworld, for both clock-time and within each thread.

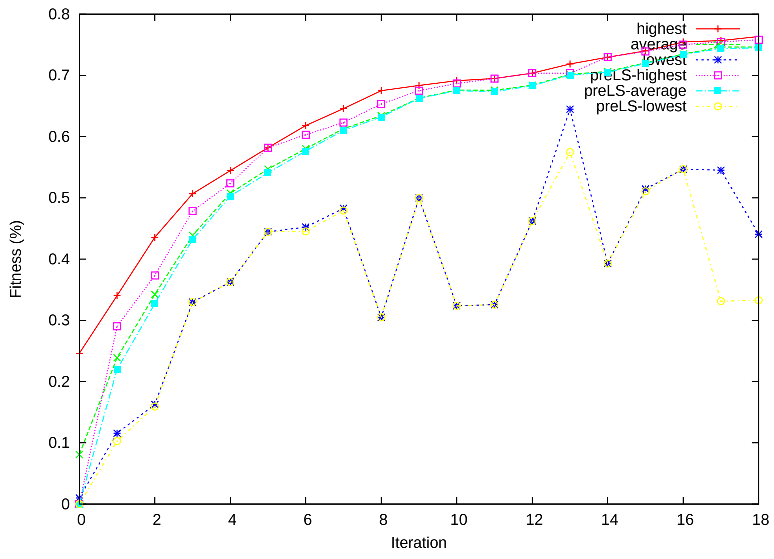


Figure 9.15: Population plots for highest, average and lowest fitness in the first 18 iterations of the Driverlog run, before and after local search.

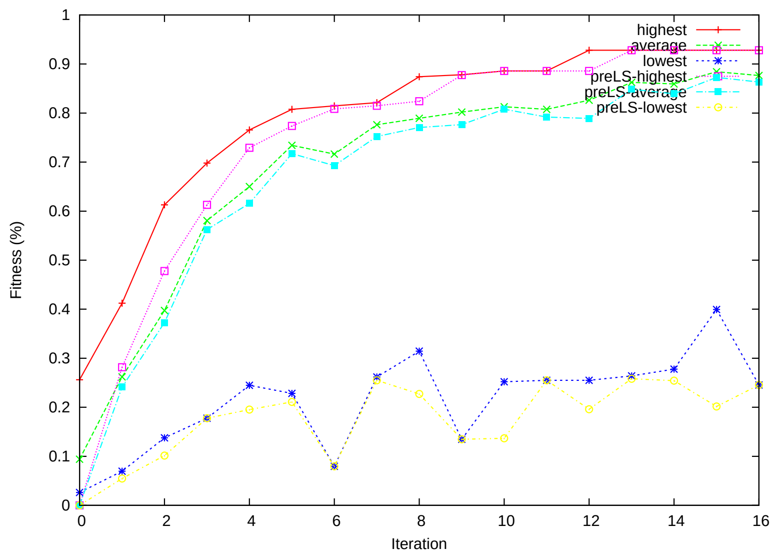


Figure 9.16: Population plots for highest, average and lowest fitness in the first 16 iterations of the unseeded Structured Briefcase run, before and after local search.

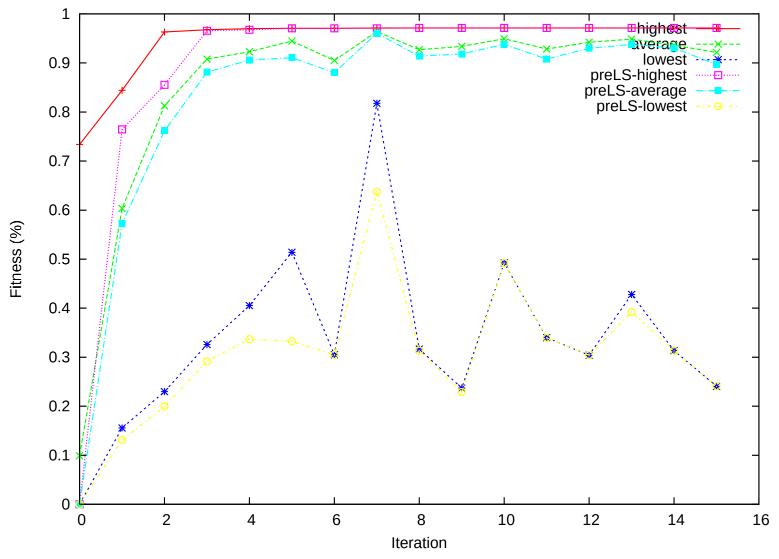


Figure 9.17: Population plots for highest, average and lowest fitness in the seeded Structured Briefcase run, before and after local search.

The plots of the first iterations of search in Driverlog, and for Structured Briefcase starting from a random population and starting from a seeded population are presented in Figures 9.15, 9.16 and 9.17, respectively. Although the unseeded Structured Briefcase run grows quickly, it does not reach the seeded run's iteration 5 score until iteration 63. The first 15 iterations of the seeded run took 29 hours, whereas the same number of iterations took almost 40 hours from fresh. Genetic algorithms are highly stochastic processes and one run is certainly not sufficient to make any hard claims, however, this does suggest that the seeding process has provided a useful starting point in this domain. Future analysis is required to establish whether the seeding process is providing a head start from the random initialisation, or actually impacting on the solution space explored. It is interesting to note that many of the main improvements seem to be made in the local search part of the program. Although it is impossible to interpret the level of interaction between the approaches from this data, it does suggest that a less intensive approach, such as simulated annealing might perform well with the fitness function.

CHAPTER 10

CONTRIBUTIONS, FUTURE WORK AND CONCLUSION

In the preceding chapters we have described the work that was carried out in order to support the thesis. In this chapter we conclude by presenting the contributions of this work; we identify the main areas of future work that have been motivated; and finally present our conclusions.

10.1 Contributions

This thesis has investigated the limitations that prevent learned control knowledge from directing search in planning problems. Our contributions lie in three main areas: problem modelling; control knowledge and its representation; and in planning and search. In this section we identify the main contributions.

10.1.1 Problem modelling

In Chapter 3 we observed that a planning environment that modelled all of the implied relationships between the objects and their behaviours would provide a suitable context for action selection. The definition of this model has provided a conceptual foundation that explains several works seeking an appropriate model for planning, including macro actions and support predicates. We defined a series of language enhancing steps that span the divide between the described problem model and this rich planning model. Although we agree with the view expressed in McDermott (2000), that the planning model should not advise the planner, we have contributed an alternative viewpoint, where we argue that the planning model should be appropriate for planning. To this end we have contributed a formal framework for exploring a general collection of possible models.

In Chapter 5, we analysed the problem of modelling within the context of a specific type of planner: the rule based policy (RBP). As a product of this analysis we identified several categories of model enhancing steps and believe it is likely that similar categories will extend to other planning approaches that reason over the problem states. We identified a library of model enhancing steps and in Chapter 6, we presented the results of an empirical analysis, which compared the alternative enhancing steps from the library. This analysis is instructive for future approaches that look to select planning models for RBPs and other related planning approaches.

We presented an alternative extension of PDDL to PDDL/M (Dornhege et al., 2009), which allows an enhanced domain model to be defined and exploited during planning. We have also extended the domain analysis performed in HybridSTAN to automatically generate an appropriate domain model. The generated models were validated in the evaluations presented in Chapters 6 and 9.

10.1.2 Control knowledge representation and learning

A contribution of this work to control knowledge representation is a general method for describing structure interactions (SIs). This has provided an important enhancement step in our framework, called an arbitrary length macro action (ALMA), which generalises the concept of directed connectivity (Chapter 5). In Chapter 7, we presented an algorithm for automatically parameterising an ALMA solver, to specialise it for the particular domain. The output of the algorithm are bags of macros for particular SI situations in the target domain. These approaches contribute a process for automatically invoking an enhanced domain model and more specifically contribute an method for controlling search through SIs. In Chapter 9, we demonstrated that the generated macro bags were appropriate for supporting RBP execution in an empirical analysis. Through the use of a specialised solver we are able to automatically invoke appropriate middle layers that are important for reasoning in a domain.

We have presented an algorithm for generating RBPs directly from example plans. The generated rules exhibit many of the elements that would be expected in a final solution. Moreover, we have extended the approach to the languages on a chain of language restrictions, providing a new method of generating generalised plans. Our approach establishes a connection between plans and RBPs that can now be developed and strengthened.

During this work, we developed a framework for testing the learning approaches reported in the literature. However, we encountered several limitations in these approaches. We have designed and implemented an alternative learning approach that measures the performance of RBPs directly. We contribute a fitness function that orders strategies based on their performance in controlling search. This is very promising as there are many avenues for making its evaluation more efficient. We have provided some support that genetic programming can be an effective tool for completing control knowledge, in distinction from evolving it from random candidates. We have presented an analysis of learned RBPs and have demonstrated that these policies provide effective control in search. Perhaps the main contribution is that we have demonstrated effective planning of learned RBPs in new domains, and the treatment of domain features that have not been addressed in previous work. In particular, we have demonstrated RBP planning in domains that combine problems of directed connectivity and optimisation.

10.1.3 Planning and search

We have made a theoretical contribution by establishing several properties of policy transference between different encodings of a problem. We observed that there are issues with using a policy for these enhanced models as a policy in the described model. We have defined co-execution, which unrolls the policy in each model simultaneously. In the subset of enhancements we have investigated, this will guarantee that the use of a plan in the rich model can be co-executed as a plan in the described model. This provides a theoretical framework that draws together previous approaches for enhancing problem models in planning, it supports our investigations into enhanced models, and provides the motivation and rational for future work.

In Chapter 5, we developed an architecture, which supports co-executing an RBP in the context of a chain of model enhancing solvers. This contributes an approach for exploiting certain types of specialised solution as part of a solution to the planning problem. It has also contributed a general architecture for developing an appropriate planning model. Within this architecture we exploit ALMAs, and these contribute a novel integration of hierarchical task network (HTN) and classical planning approaches.

Our approach extends the applicability of RBPs in search. This is an important result for the learning community; however, it also contributes to the wider planning community. Tightly specified rule based planners are extremely effective. As the community targets larger problems with increasingly high expectations for plan quality rule based approaches could return to prominence.

10.2 Future work

As is the case with any investigation with an interesting topic, we have uncovered many questions and avenues for future work. The open research opportunities relate to the ALMA representation and the generation of RBPs. In this section we identify some of these avenues for future work.

10.2.1 ALMA

Separating the computation of the ALMA predicate and action could make evaluating the vocabulary more efficient. The approach of Botea et al. (2007) demonstrates the effective expansion of a set of macros given a specific target (the goal). Indeed, there are many approaches to solving this problem (as we can encode it as the planning problem), particularly, there are approaches that guarantee completeness (e.g. Hoffmann and Nebel, 2001). The computation of the predicates requires that the reachability of the space is established, however, it does not require the paths are discovered. We expect that a compact expression of reachability could be more readily found than a compact expression that generates the action sequences themselves. For example, in Driverlog the nodes reachable by the trucks can be expressed with the transitive closure of the `link` predicate and in Goldminer the nodes that are discoverable by the robot with a laser can be expressed by searching through the `link` predicate for nodes that are either closed in specific ways or already open. These formulae would be much faster to evaluate. However, there are several challenges. The reliance on heuristic selection for optimisation problems means that plans for several targets may be required so that the resulting states can be evaluated. The reachability formula might be difficult to express, for example, establishing the reachability of the robot with a bomb requires counting resources.

In the current interpretation, we consider the individual SIs as independent events, whereas in practice there is often more structure involved. For example, in Goldminer plans, the bomb will only ever be used to open the gold square. The chains that we have used to generate the vocabulary exhibit these structures and our selected representation generalises from these particular patterns. We consider that exploiting this structure by using a richer representation, such as using LOCM2 (Cresswell and Gregory, 2011) to generate a biased domain model, could have implications on the evaluation of the vocabulary and also improve its relevance.

In de la Rosa and McIlraith (2011) an analysis of TLPLAN control knowledge has led to a different approach and strong results. Combining the language enhancements developed in our work with a richer rule language and the learned model enhancements developed in de la Rosa and McIlraith (2011) is promising future work. In fact, in de la Rosa and McIlraith (2011) the authors observe that the developed approach is limited in situations that we have addressed in this work. In Appendix G.1 we consider a hierarchical extension to the ALMA solver, which exploits guidance from the relaxed plan. However, this framework constrains the use of the vocabulary, and requires an extra solver and potentially arbitrary levels to make the approach general. An alternative is to allow the RBP to map to the possible actions and allow a heuristic to select the best perceived action. This would be possible in many cases by introducing disjunction into the rule language. Promising future work includes developing our work for use in TLPLAN and combining it with the approach presented in de la Rosa and McIlraith (2011).

The scope of our investigation into automating solver generation is limited. We focus on sub-goals that can be solved largely in isolation. In particular, we focus on sub-goals that can be represented with a finite number of propositions. The use of recursion in the goal has been investigated using derived predicates in Khardon (1999a); Levine and Humphreys (2003) (as well as in this work) and language extensions in Martin and Geffner (2000); Fern et al. (2006). These approaches allow expressing some forms of recursive relations over goals, but there are many that cannot be expressed. We believe that the ALMA architecture can be extended to reason over completing goals. This will only be realised efficiently where target significance is maintained, which is at least sufficient for making good stacks in Blocksworld.

We posed the generation of vocabulary in the context of SIs, which was appropriate in the context of the work. However, in this context we rely on analysis to uncover specific features of a domain model. It would be interesting to pose this problem in a more general setting: in terms of lifted action sequences and sub-goals. In this context we could develop a representation that allowed analysis between different sequences of lifted actions. This could allow us to utilise general learning technologies leading to a more general solution.

10.2.2 Learning RBPs

In Chapter 8, we defined a fitness function that we used to learn RBPs for several domains. It was our expectation that the continual use of an optimal planner would dominate unrolling policies and as a result we settled on a naive coupling of the policy applier. It is likely that basic Software Engineering practices will lead to large efficiency improvements. Our analysis of the fitness function in this work has been limited and there are many properties that could be explored. These include the use of a satisficing planner or heuristic estimate to evaluate the distance from goal, the evaluation of the fitness features to establish the most effective combination, and the inclusion of planning time in the fitness function. In Chapter 9, we observed that the local search aspect of the learner was making many of the improvements. This suggests that, to some extent, the fitness function is able to provide incremental guidance to the learner. If this is the case, then a less expensive learning approach, such as Simulated Annealing (Skiena, 2008), could be applied.

We have presented an approach that generates a RBP from example plans. We have already discussed some approaches for lifting limitations in Chapter 8; however, establishing this connection outlines a more general area for future investigation. The RBP provides an alternative representation for expressing generalised plans. The priority ordering over the rules makes the mapping from plans to rules less direct. However, the representation naturally captures iteration and priority, which can greatly simplify the expression of certain types of strategy. There are also important related works on regression, goal ordering and generating HTNs that can instruct work in this area.

10.3 Conclusion

In this section we will identify the steps taken to satisfy the outline and objective in the statement of thesis. The purpose of this work was to investigate problem modelling with the aim of supporting learned RBPs to effectively control search in planning problems. In Chapter 3, we defined a chain of steps between the described problem model and the ideal model for making action choices, providing the framework for exploring alternative models of a problem. In order to support exploration of an interesting selection of chains we were required to formalise a method of policy execution that we call co-execution, which provides a generalised approach for plan transferral between models on a chain. In Chapter 5, we developed an architecture that provided an implementation for co-executing an RBP. We analysed the relationship between RBPs and the problem model and identified three categories of chain steps: directed connectivity, optimisation, and level of reasoning. We developed a library of model enhancements from these categories and empirically evaluated the use of these steps in planning. We observed that establishing directed connectivity over SIs was important for RBP expression, and using heuristics to compute optimisation steps was more practical than identifying chain steps.

Our approach for automating the process of model selection is composed of two parts: to uncover the SIs in the domain and to generate appropriate solvers for those SIs. Each of the solvers in the library, developed in Chapter 5, was associated with a type of SI. Using domain analysis the SIs in the domain have been uncovered and the appropriate model enhancements from the library selected. In Chapter 6, we demonstrated that the selected models are appropriate for supporting handwritten RBPs in several domains. Moreover, we demonstrate that the model enhancements contribute to faster planning that finds better quality plans, than when domain independent heuristics are used, as in previous approaches. Through developing a novel knowledge representation, called an ALMA, we have demonstrated that model enhancements that establish directed connectivity over SIs can be specialised from plan samples. This has greatly reduced the effort required in defining a general set of model enhancements. This work contributes an approach to combining concepts from HTN planning within a forward chaining, action selection planner. These aspects were described in Chapter 7. In Chapter 9, we have evaluated the practical aspects of our ALMA representation and we have examined properties of the generated model enhancements. The evaluation demonstrates that the enhancements support the policies in effectively guiding search.

Unfortunately, the current learning approaches are not appropriate for generating effective RBPs for the domains that we have examined. We observed a limitation in the fitness functions being used to guide search and have presented an alternative fitness function in Chapter 8, which uses search performance rather than action selection to order the candidates. Our implementation could be largely improved; however, our results are promising and suggest that less intensive search strategy (than genetic algorithms) could be effective. We have investigated automatically generating policies through generalised regression, in order to provide a starting point for the genetic algorithm learning approach. The combination of these approaches was used to learn RBPs that generated high quality plans for Driverlog and Goldminer problems (Chapter 9). These solutions demonstrate the effective use of the enhanced problem models, validating the automatically selected domain models.

We have presented the steps that we have taken in order to support the thesis. We have investigated the relationship between the problem model and control knowledge expressed over it. As part of this investigation we have demonstrated the direct control of learned RBPs in several domains that lie outwith the scope of previous work, fulfilling our objective.

APPENDIX A

PLANNING DOMAINS

In this Appendix we present an overview of the domains used in this work. We will use the problem types defined in Section 5.2. For more information of the origins of the domains, consult the international planning competition (IPC) website (IPC, 2014).

Blocksworld The Blocksworld domain defines stacking problems, which involve a table with unbounded capacity and some stacks of blocks. The goal is to rearrange the stacks into a specific order. In this work we used the four operator variant, which models a hand detaching and attaching the blocks from each other. This domain was used in IPC-2. Identifying whether a stack is consistent with the goal is important for these problems.

Briefcase The Briefcase domain is a simple transportation domain that used ADL language features in PDDL (IPP, 1999). We use a STRIPS version, as used in Levine and Humphreys (2003). The briefcase moves in a fully connected graph redistributing objects. We have extended this domain with a graph structure. We call this the Structured Briefcase domain.

Depots The Depots domain combines transportation and stacking problem aspects. Trucks move between depots carrying blocks. There are cranes at each depot that pick blocks from the trucks and these are then put on one of the stacks at the depot. The goal is to have certain stacks on top of specific crates. The locations are fully connected. This domain was used in the third IPC.

Driverlog Driverlog is a transportation domain that involves delivering packages between locations. The trucks must be driven between locations. The trucks and drivers move on different graphs. The graphs are defined as `link` and `path` propositions, which define the connected edges in the graphs. There can be goals for drivers and trucks. Plans will usually involve moving drivers to trucks, a series of package deliveries, the drivers driving the trucks home and then walking home themselves. This domain was used in the third IPC.

Goldminer The Goldminer domain defines path opening problems. A robot is moved on a grid of locations, which can be blocked by hard and soft rock. One location in the map will have gold, and the goal is for the robot to access the gold. The laser will blow through as many rocks as necessary, but it will also destroy the gold. The bomb can be used once, and will clear rock without effecting the gold. This domain was used in the first learning track of the IPC.

Grid The grid domain is a path opening and transportation domain, which involves a robot on a grid of locations, some with locked doors. The problem involves picking up the appropriate keys to unlock the doors and create a path through the locked doors. Some of the keys have goal locations. This domain was used in the first IPC. We adapted the problem so that it was a path traversal problem with a goal for the robot.

Gripper Gripper problems are transportation problems that involve moving balls between two rooms. The transporter, a robot, has a limited number of arms, imposing a capacity on the number of balls that can be carried. This domain was used in the first IPC.

Logistics In the Logistics domain packages must be delivered between different cities. The cities are connected by aeroplanes, whereas distribution within a city is carried out by trucks. The locations and cities form cliques. This domain was used in the first IPC.

Traverser This domain defines path traversal problems. The locations are connected in a graph network and the goals involve moving traversers to their destinations. This domain was made to provide a simple domain for demonstrating the use of the language features.

APPENDIX B

SOLVER LISTINGS

In this appendix we present an enhanced domain model definition and the associated solver listings generated using domain analysis, as described in Chapters 5 and 7.

B.1 The enhanced domain model

```
(define (domain SBC)
        (:requirements :typing :enhanced-domain-model)
        (:types
                 obj truck - locatable
                 locatable location - object
        )
        (:solverListings SolverListingFolder/SolverListings1400168697047)
        (:solver GraphAbstraction0
                :type solvers.solvers.GraphAbstraction)
        (:solver TransportedObject0
                :type solvers.solvers.TransportedObjectBoundSolver)

        (:predicates
                (link ?x - location ?y - location)
                (in ?obj1 - obj ?obj - truck)
                (at ?obj - locatable ?loc - location)
        )
        (:activePredicates
                (drive-truck_connected ?loc-from - location ?loc-to - location)
                (bound_load-truck ?consumer - obj ?resource - truck)
        )
        (:activeAction long_drive-truck
                :parameters
                (
                        ?truck - truck
                        ?loc-from - location
                        ?loc-to - location
                )
                :precondition
                        (and(drive-truck_connected ?loc-from ?loc-to)
                                (at ?truck ?loc-from))
```

```
                    :effectApplier GraphAbstraction0)

        (:action load-truck ...)
        (:action unload-truck ...)
        (:action drive-truck ...)
)
```

The enhanced domain relies on actions and predicates modelled by two special purpose solvers. The graph abstraction solver models the `long_driver-truck` action, which moves a truck to a target node (or a step in that direction) and the `drive-truck_connected` predicate, which holds for targets that can be reached by the truck. The transported object solver models the (derived) predicate, `bound_load-truck`, which holds between pairs of packages and trucks if the truck is allocated to the package.

B.2 The solver listings file

```
((:module StaticGraphModule2
        :type solvers.encoding.graphabstraction.StaticGraphModule)
(:module StaticGraph1
        :type solvers.encoding.graphabstraction.graphencoding.StaticGraph)
(:module MoveAction0
        :type solvers.encoding.graphabstraction.moveaction.MoveAction)
(:module ConnectingChain8
        :type solvers.encoding.resourcemanagement.ConnectingChainExplorer)
(:module CapacityCounter6
        :type solvers.encoding.resourcemanagement.CapacityCounter)
(:module ConnectingChain7
        :type solvers.encoding.resourcemanagement.ConnectingChainExplorer)
(:module CapacityCounter5
        :type solvers.encoding.resourcemanagement.CapacityCounter)
(:module MovingObjectEntry3
        :type javaff.entries.MovingObjectEntry)

(:solverDescription GraphAbstraction0
        :Encoding (:module StaticGraphModule2)
)
(:moduleDescription StaticGraphModule2
        :Map (:module StaticGraph1)
        :MoveAction (:module MoveAction0)
        :EnablingPredicates (:description ())
)
(:moduleDescription StaticGraph1
        :MoveAction (:module MoveAction0)
        :MapPredicates (:description (link ?loc-from ?loc-to#))
)
(:moduleDescription MoveAction0
        :MoveAction (:description (drive-truck 0 1 2))
        :Locatedness (:description (at 0 1))
)
(:solverDescription TransportedObject0
        :MapConnection (:module ConnectingChain8)
        :ObjectConnection (:module ConnectingChain7)
        :DistanceMeasure (:module MovingObjectEntry3)
        :Deallocation (:description (unload-truck 1 0))
        :Reversed (:description (false))
        :Allocation (:description (load-truck 1 0))
)
(:moduleDescription ConnectingChain8
        :Capacity (:module CapacityCounter6)
        :Chain (:description ())
        :Connection (:description (at 1 0))
)
(:moduleDescription CapacityCounter6
        :Direct (:description (true))
        :Properties (:description ())
        :InfiniteCapacity (:description (true))
)
(:moduleDescription ConnectingChain7
        :Capacity (:module CapacityCounter5)
        :Chain (:description ())
        :Connection (:description (in 1 0))
)
(:moduleDescription CapacityCounter5
        :Direct (:description (true))
        :Properties (:description ())
        :InfiniteCapacity (:description (true))
)
(:moduleDescription MovingObjectEntry3
        :MovingObjectSolver (:module GraphAbstraction0)
)
)
```

The description of the solver includes the definition of various modules that capture the specific mapping for the particular domain. The graph abstraction solver is parameterised by a static graph module, which identifies the move action and the predicates that enable movement (requirements of the traverser). A refinement of this module identifies the conditions on moving (requirements of the map). The move action module identifies the action name its the significant parameters (traverser, from position and destination parameters) as well as the locatedness predicate and its significant parameters.

The transported object solver relies on a graph that is used to locate resources and consumers. The important allocate and (potentially null) deallocate actions are defined (for largely historic reasons) and whether the transported object is in fact the resource, rather than the consumer. Several modules are used to identify the connections between the transported object and the map or traverser and its capacity. A chain of static predicates can link the transported object to the traverser (for example, the hand in Gripper) or the map (for example, the crane in Depots).

APPENDIX C

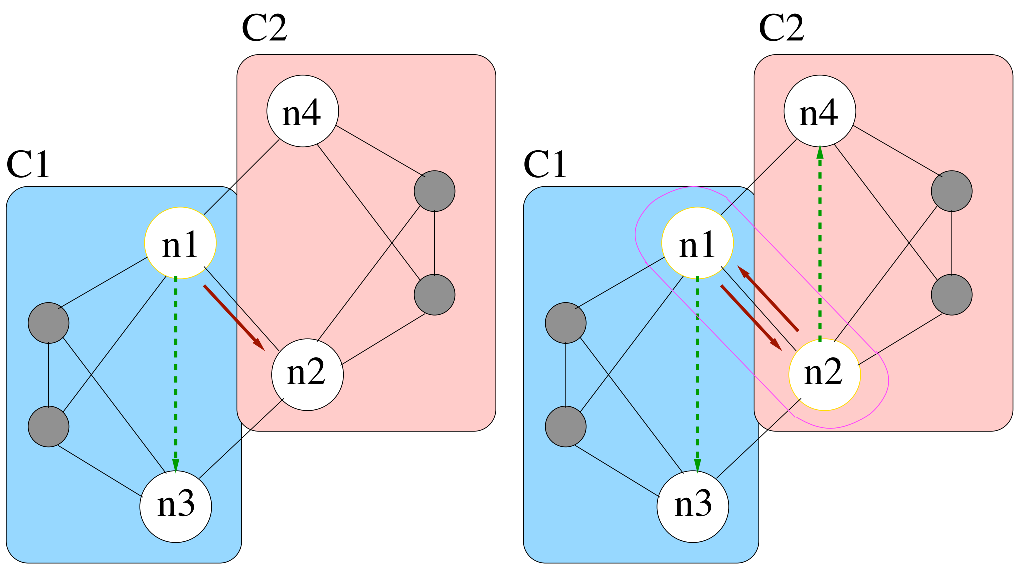
STEP-BY-STEP MACRO APPLICATION VOCABULARY

In this appendix we consider the step by step application approach (SbS) and how the function of vocabulary can be adapted to support this approach. We demonstrate how the original vocabulary is not appropriate and present an alternative chain step.

C.1 Traversing through a cluster

There is a problem when using the SbS and moving between locations in the same cluster. For example, the use of a long move action between two nodes in a cluster will not necessarily stay within the cluster. For example, if the shortest path between the locations includes a location that belongs to a different cluster. The unrolling process supporting the macro move action can also move outside the cluster. However, as the policy is not applied at the intermediary nodes, the specific path chosen is less important. This problem is illustrated in Figure C.1.

The state illustrated in the figure can cause an executive to enter a continuous loop. In this example, the policy will map to an action with the intention of moving the truck from $\mathtt{n}_1$ to $\mathtt{n}_3$ and this is translated into the ground action that moves the truck from $\mathtt{n}_1$ to $\mathtt{n}_2$. In this case the current cluster of the truck's location has now changed and in this new state the policy will map to the action that intends to move the truck from $\mathtt{n}_2$ to $\mathtt{n}_4$. However, a shortest path to this node is through $\mathtt{n}_1$. If the action is translated to a move of the truck from $\mathtt{n}_2$ to $\mathtt{n}_1$ then execution will loop forever. The step by SbS can fall into this trap. However, if there is a shortest path between the locations



(a) The truck is moved out of the cluster on its path to n3

(b) The truck is then moved back to the starting node on its way to n4

Figure C.1: The use of the shortest path solver to select the next step can result in looped execution

that continues in the same cluster then the shortest path propositions can be used with a target node to direct search. This is achieved by guarding the *move-to* location with the `sameCluster` proposition. The problem is that if there is not a shortest path that remains within the cluster then this antecedent will not hold.

The step that we propose is to include an action that allows traversal within a cluster. The `cluster-move` action takes a single step towards the target along a path that is shortest of all paths that remained within the cluster. A language step can be defined in a similar manner as before and the policy can be modified to use these actions.

APPENDIX D

POLICIES

In this appendix several of the policies used in the analyses are presented.

D.1 Handwritten

D.1.1 Blocksworld

```
(define (policy blocksworld_policy)
  (:domain blocksworld)
(:rule stack_on_well_placed
  :parameters  (?ob ?underob - block)
  :condition (and (clear ?underob) (holding ?ob) (well_placed_on ?underob))
  :goalCondition (and (on ?ob ?underob))
  :action (stack ?ob ?underob)
)
(:rule pickup_to_place_well
  :parameters (?ob ?underob - block)
  :condition (and (clear ?ob) (on-table ?ob) (arm-empty) (well_placed_on ?underob)
               (clear ?underob))
  :goalCondition (and (on ?ob ?underob))
  :action (pickup ?ob)
)
(:rule putdown
  :parameters  (?ob - block)
  :condition (holding ?ob)
  :goalCondition (and )
  :action (putdown ?ob)
)
(:rule unstack_from_bad_tower
  :parameters  (?ob ?underob - block)
  :condition (and (on ?ob ?underob) (clear ?ob) (arm-empty) (not(well_placed_on ?ob))
      )
  :goalCondition (and )
  :action (unstack ?ob ?underob)
)
)
```

D.1.2 Depots

```
(define (policy depots_policy)
  (:domain Depot)
  (:rule stack_on_well_placed
    :parameters (?obj1 - crate ?obj2 - surface ?loc - place ?crane - hoist)
    :condition (and (lifting ?crane ?obj1) (at ?crane ?loc) (well_placed_on ?obj2)
                (at ?obj2 ?loc) (clear ?obj2))
    :goalCondition (and (on ?obj1 ?obj2) )
    :action (drop ?crane ?obj1 ?obj2 ?loc)
  )
  (:rule unload_to_stack_well
    :parameters (?obj1 - crate ?obj2 - surface ?truck - truck ?loc - place ?crane -
        hoist)
    :condition (and (in ?obj1 ?truck) (at ?truck ?loc) (at ?crane ?loc) (
        well_placed_on ?obj2)
                (at ?obj2 ?loc) (available ?crane) (clear ?obj2))
    :goalCondition (and (on ?obj1 ?obj2) )
    :action (unload ?crane ?obj1 ?truck ?loc)
  )
  (:rule load_crane_with_misplaced
    :parameters (?obj1 - crate ?obj2 - surface ?truck - truck ?loc - place ?crane -
        hoist)
    :condition (and (at ?truck ?loc) (at ?crane ?loc)
                (at ?obj2 ?loc) (lifting ?crane ?obj1))
    :goalCondition (and (not (on ?obj1 ?obj2)))
    :action (load ?crane ?obj1 ?truck ?loc)
  )
  (:rule lift_from_bad_tower
    :parameters (?obj1 - crate ?obj2 - surface ?truck - truck ?loc - place ?crane -
        hoist)
    :condition (and (at ?truck ?loc) (at ?crane ?loc) (not(well_placed_on ?obj1))
                (at ?obj1 ?loc) (available ?crane) (clear ?obj1) (on ?obj1 ?obj2))
    :goalCondition (and )
    :action (lift ?crane ?obj1 ?obj2 ?loc)
  )
  (:rule move_to_pickup_misplaced
    :parameters (?obj - crate ?truck - truck ?loc1 ?loc2 - place ?crane - hoist)
    :condition (and (at ?truck ?loc2) (not (well_placed_on ?obj))
                (at ?obj ?loc1) (clear ?obj))
    :goalCondition (and )
    :action (drive ?truck ?loc2 ?loc1)
  )
  (:rule move_to_place_well
    :parameters (?obj1 - crate ?obj2 - surface ?truck - truck ?loc1 ?loc2 - place ?
        crane - hoist)
    :condition (and (in ?obj1 ?truck) (at ?truck ?loc2) (well_placed_on ?obj2)
                (at ?obj2 ?loc1) (clear ?obj2))
    :goalCondition (and (on ?obj1 ?obj2) )
    :action (drive ?truck ?loc2 ?loc1)
  )
)
```

D.1.3 Driverlog

```
(define (policy driverlog_policy)
  (:domain driverlog)
  (:rule drop_at_goal
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :condition (and (in ?obj ?truck) (at ?truck ?loc))
    :goalCondition (and (at ?obj ?loc) )
    :action (unload-truck ?obj ?truck ?loc)
  )
```

```
(:rule pickup_misplaced_package
  :parameters (?obj - obj ?truck - truck ?loc - location)
  :condition (and (at ?obj ?loc) (at ?truck ?loc) (Bound_MovingObjectEntry3 ?obj ?
      truck))
  :goalCondition (and (not (at ?obj ?loc)))
  :action (load-truck ?obj ?truck ?loc)
)
;; in the following, the trucks are allocated to packages
(:rule move_to_pickup_misplaced
  :parameters (?truck - truck ?from ?to - location ?driver - driver ?obj - obj)
  :condition (and (at ?truck ?from) (at ?obj ?to) (Bound_MovingObjectEntry3 ?obj ?
      truck) (driving ?driver ?truck) )
  :goalCondition (and (not (at ?obj ?to)))
  :action (long_drive-truck ?truck ?from ?to ?driver)
)
(:rule drive_to_package_goal
  :parameters (?truck - truck ?from ?to - location ?driver - driver ?obj - obj)
  :condition (and (at ?truck ?from) (in ?obj ?truck) (driving ?driver ?truck) )
  :goalCondition (and (at ?obj ?to))
  :action (long_drive-truck ?truck ?from ?to ?driver)
)
(:rule drive_to_truck_goal
  :parameters (?truck - truck ?from ?to - location ?driver - driver)
  :condition (and (at ?truck ?from) (driving ?driver ?truck) )
  :goalCondition (and (at ?truck ?to) (not (at ?truck ?from)))
  :action (long_drive-truck ?truck ?from ?to ?driver)
)
;; in the following, drivers are allocated to trucks
(:rule board_truck_and_misplaced_package
  :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
  :condition (and (at ?driver ?loc) (Bound_MovingObjectEntry7 ?truck ?driver) (at ?
      obj ?l)(Bound_MovingObjectEntry3 ?obj ?truck) (at ?truck ?loc))
  :goalCondition (and (not (at ?obj ?l)))
  :action (board-truck ?driver ?truck ?loc)
)
(:rule board_truck_carrying_package
  :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
  :condition (and (at ?driver ?loc) (at ?truck ?loc)(Bound_MovingObjectEntry7 ?
      truck ?driver) (in ?obj ?truck))
  :goalCondition (and (at ?obj ?l))
  :action (board-truck ?driver ?truck ?loc)
)
(:rule board_misplaced_truck
  :parameters (?driver - driver ?loc ?l - location ?truck - truck)
  :condition (and (at ?driver ?loc) (Bound_MovingObjectEntry7 ?truck ?driver) (at ?
      truck ?loc))
  :goalCondition (and (not (at ?truck ?loc)) (at ?truck ?l))
  :action (board-truck ?driver ?truck ?loc)
)
(:rule walk_to_board_and_misplaced_package
  :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
  :condition (and (at ?driver ?from) (Bound_MovingObjectEntry7 ?truck ?driver) (at
      ?obj ?l)(Bound_MovingObjectEntry3 ?obj ?truck) (at ?truck ?to))
  :goalCondition (and (not (at ?obj ?l)))
  :action (long_walk ?driver ?from ?to)
)
(:rule walk_to_board_truck_carrying_package
  :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
  :condition (and (at ?driver ?from) (Bound_MovingObjectEntry7 ?truck ?driver) (in
      ?obj ?truck)(at ?truck ?to))
  :goalCondition (and (at ?obj ?l))
  :action (long_walk ?driver ?from ?to)
)
(:rule walk_to_board_misplaced_truck
  :parameters (?driver - driver ?from ?to ?l - location ?truck - truck)
  :condition (and (at ?driver ?from) (Bound_MovingObjectEntry7 ?truck ?driver) (at
      ?truck ?to))
```

```
    :goalCondition (and (not (at ?truck ?to)) (at ?truck ?l))
    :action (long_walk ?driver ?from ?to)
  )
  (:rule walk_home
    :parameters (?driver - driver ?from ?to - location)
    :condition (and (at ?driver ?from))
    :goalCondition (and (at ?driver ?to) (not (at ?driver ?from)))
    :action (long_walk ?driver ?from ?to)
  )
  (:rule disembark
    :parameters (?driver - driver ?truck - truck ?loc - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc))
    :goalCondition (and )
    :action (disembark-truck ?driver ?truck ?loc)
)
)
```

D.1.4 Goldminer

```
(define (policy goldminer_policy)
(:domain goldminer)
(:rule pickup_gold
  :parameters (?r - robot ?g - gold ?l1 - loc)
  :condition (and (at ?r ?l1) (at ?g ?l1) (arm-empty ?r))
  :goalCondition (and (holds ?g ?r))
  :action (pickup ?r ?g ?l1)
)
(:rule pickup_if_ready_for_bomb
  :parameters (?r - robot ?b - bomb ?g - gold ?l1 ?l2 - loc)
  :condition (and (blocked ?l1 ?l2) (at ?g ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?r ?l1) (at ?b ?l1) (arm-empty ?r))
  :goalCondition (and (holds ?g ?r))
  :action (pickup ?r ?b ?l1)
)
(:rule pickup_laser_if_gold_blocked
  :parameters (?r - robot ?l - laser ?g - gold ?l1 ?l2 - loc)
  :condition (and (blocked ?l1 ?l2) (at ?g ?l2) (at ?r ?l1) (at ?l ?l1) (arm-empty ?r
      ))
  :goalCondition (and (holds ?g ?r))
  :action (pickup ?r ?l ?l1)
)
(:rule unblock_adjacent_to_goal
  :parameters (?r - robot ?g - gold ?b - bomb ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (connected ?l1 ?l2) (holds ?b ?r))
  :goalCondition (and (holds ?g ?r))
  :action (detonate-bomb ?r ?b ?l1 ?l2)
)
(:rule putdown_laser_to_pickup_bomb
  :parameters (?r - robot ?g - gold ?b - bomb ?l - laser ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (holds ?l ?r) (at ?b ?l1))
  :goalCondition (and (holds ?g ?r))
  :action (putdown ?r ?l ?l1)
)
(:rule fire_laser_towards_bomb
  :parameters (?r - robot ?g - gold ?b - bomb ?l - laser ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (holds ?l ?r) (blocked ?l1 ?l3) (nearest-blocked ?l1 ?l3 ?l4
                 )
             (at ?b ?l3) (connected ?l1 ?l4))
  :goalCondition (and (holds ?g ?r))
  :action (fire-laser ?r ?l ?l1 ?l4)
)
(:rule move_with_laser_towards_bomb
  :parameters (?r - robot ?g - gold ?b - bomb ?l - laser ?l1 ?l2 ?l3 ?l4 ?l5 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (holds ?l ?r) (blocked ?l1 ?l3) (nearest-blocked ?l1 ?l3 ?l4
                  )
             (at ?b ?l3) (connected ?l5 ?l4))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l5)
)
(:rule move_to_pickup_laser
  :parameters (?r - robot ?g - gold ?b - bomb ?l - laser ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (arm-empty ?r) (blocked ?l1 ?l3)
             (at ?b ?l3) (at ?l ?l4))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l4)
)
(:rule move_to_pickup_bomb
  :parameters (?r - robot ?g - gold ?b - bomb ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (at ?b ?l3))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l3)
)
(:rule move_with_bomb_towards_gold
  :parameters (?r - robot ?g - gold ?b - bomb ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l2)
             (at ?g ?l2) (connected ?l3 ?l2) (holds ?b ?r))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l3)
)
;; Make a Path to the Goal
(:rule fire_laser_towards_gold
  :parameters (?r - robot ?g - gold ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l3)
             (at ?g ?l2) (holds ?l ?r) (connected ?l1 ?l3))
  :goalCondition (and (holds ?g ?r))
  :action (fire-laser ?r ?l ?l1 ?l3)
)
(:rule move_to_pickup_laser
  :parameters (?r - robot ?g - gold ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (at ?g ?l2)
             (at ?l ?l3) (arm-empty ?r))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l3)
)
(:rule move_with_laser_towards_gold
  :parameters (?r - robot ?g - gold ?l - laser ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (blocked ?l1 ?l2) (nearest-blocked ?l1 ?l2 ?l3)
             (at ?g ?l2) (holds ?l ?r) (connected ?l4 ?l3))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l4)
)
(:rule move_to_goal
  :parameters (?r - robot ?g - gold ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (at ?g ?l2))
  :goalCondition (and (holds ?g ?r))
  :action (move ?r ?l1 ?l2)
)
)
```

D.1.5 Grid

```
(define (policy grid_policy)
  (:domain grid)
(:rule move_to_goal
  :parameters (?r - robot ?l1 ?l2 - place)
  :condition (and (at ?r ?l1) (move_connected ?r ?l1 ?l2))
  :goalCondition (and (at ?r ?l2))
  :action (long_move ?r ?l1 ?l2)
)
(:rule pickup_fitting_key
  :parameters (?r - robot ?k - key ?l1 ?l2 - place ?s - shape)
  :condition (and (at ?r ?l1) (at ?k ?l1) (arm-empty ?r) (
      doorToOpen_LocatedKeyDoorSelector0 ?l2)
    (key-shape ?k ?s) (lock-shape ?l2 ?s))
  :goalCondition (and )
  :action (pickup ?r ?l1 ?k)
)
(:rule unlock_useful_door
  :parameters (?r - robot ?k - key ?l1 ?l2 - place ?s - shape)
  :condition (and (at ?r ?l1) (holding ?r ?k) (doorToOpen_LocatedKeyDoorSelector0 ?l2
      )
    (key-shape ?k ?s) (lock-shape ?l2 ?s) (conn ?l1 ?l2))
  :goalCondition (and )
  :action (unlock ?r ?l1 ?l2 ?k ?s)
)
(:rule move_to_open_useful_door
  :parameters (?r - robot ?k - key ?l1 ?l2 ?l3 - place ?s - shape)
  :condition (and (at ?r ?l1) (holding ?r ?k) (doorToOpen_LocatedKeyDoorSelector0 ?l2
      )
    (key-shape ?k ?s) (lock-shape ?l2 ?s) (conn ?l3 ?l2))
  :goalCondition (and )
  :action (long_move ?r ?l1 ?l3)
)
(:rule discard_unfitting_key
  :parameters (?r - robot ?k - key ?l1 ?l2 - place ?s - shape)
  :condition (and (at ?r ?l1) (holding ?r ?k) (doorToOpen_LocatedKeyDoorSelector0 ?l2
      )
    (key-shape ?k ?s) (not (lock-shape ?l2 ?s)))
  :goalCondition (and )
  :action (putdown ?r ?l1 ?k)
)
(:rule move_to_pickup_fitting_key
  :parameters (?r - robot ?k - key ?l1 ?l2 ?l3 - place ?s - shape)
  :condition (and (at ?r ?l1) (at ?k ?l3) (doorToOpen_LocatedKeyDoorSelector0 ?l2)
    (key-shape ?k ?s) (lock-shape ?l2 ?s))
  :goalCondition (and )
  :action (long_move ?r ?l1 ?l3)
)
)
```

D.1.6 Logistics

```
(define (policy logisitcs_policy)
(:domain logistics)
(:rule dropoff_at_goal
  :parameters ( ?loc - place ?truck - truck ?pkg - package)
  :condition (and (at ?truck ?loc) (in ?pkg ?truck) )
  :goalCondition (and (at ?pkg ?loc) )
  :action (unload-truck ?pkg ?truck ?loc)
)
(:rule pickup_misplaced_at_location
  :parameters (?place - place ?loc - place ?location - location ?truck - truck ?pkg -
      package ?city - city)
  :condition (and (at ?truck ?loc) (at ?truck ?location) (at ?pkg ?loc))
  :goalCondition (and (at ?pkg ?place)(not (at ?pkg ?loc))) ; seems to make random..
  :action (load-truck ?pkg ?truck ?loc)
```

```
)
(:rule load_local_misplaced
  :parameters (?place ?loc - place ?truck - truck ?pkg - package ?city - city)
  :condition (and (at ?truck ?loc) (at ?pkg ?loc) (in-city ?place ?city) (in-city ?
      loc ?city))
  :goalCondition (and (not (at ?pkg ?loc)) (at ?pkg ?place))
  :action (load-truck ?pkg ?truck ?loc)
)
(:rule pickup_for_other_city
  :parameters (?loc ?place - place ?airplane - airplane ?pkg - package ?city - city)
  :condition (and (at ?pkg ?loc) (at ?airplane ?loc) (in-city ?place ?city) (not (
      in-city ?loc ?city)))
  :goalCondition (and (at ?pkg ?place) )
  :action (load-airplane ?pkg ?airplane ?loc)
)
(:rule dropoff_in_right_city
  :parameters ( ?loc ?place - place ?airplane - airplane ?pkg - package ?city - city)
  :condition (and (in ?pkg ?airplane) (at ?airplane ?loc) (in-city ?loc ?city) (
      in-city ?place ?city))
  :goalCondition (and (at ?pkg ?place))
  :action (unload-airplane ?pkg ?airplane ?loc)
)
(:rule dropoff_at_airport
  :parameters ( ?airport - airport ?loc ?place - place ?truck - truck ?pkg - package
      ?city - city)
  :condition (and (at ?truck ?airport) (at ?truck ?loc) (in ?pkg ?truck) (in-city ?
      place ?city) (not (in-city ?loc ?city)))
  :goalCondition (and (at ?pkg ?place) )
  :action (unload-truck ?pkg ?truck ?loc)
)
(:rule drive_to_pickup_from_location
  :parameters (?place - place ?truck - truck ?city - city ?loc-to - place ?location -
        location ?loc-from - place ?package - package)
  :condition (and (at ?truck ?loc-from) (at ?truck ?location) (in-city ?loc-from ?
      city) (in-city ?loc-to ?city) (at ?package ?loc-to))
  :goalCondition (and (not (at ?package ?loc-to)) (at ?package ?place))
  :action (drive-truck ?truck ?loc-from ?loc-to ?city)
)
(:rule drive_to_local_pickup
  :parameters ( ?truck - truck ?city - city ?loc-to ?place - place ?loc-from - place
      ?package - package)
  :condition (and (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?
      city) (in-city ?place ?city)(at ?package ?loc-to))
  :goalCondition (and (not (at ?package ?loc-to))(at ?package ?place))
  :action (drive-truck ?truck ?loc-from ?loc-to ?city)
)
(:rule drive_to_dropoff_at_airport
  :parameters ( ?truck - truck ?city - city ?loc-to ?place - place ?loc-from - place
      ?package - package ?airport - airport)
  :condition (and (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?
      city) (in ?package ?truck) (not (in-city ?place ?city)) (not (!= ?loc-to ?
      airport)))
  :goalCondition (and (at ?package ?place))
  :action (drive-truck ?truck ?loc-from ?loc-to ?city)
)
(:rule fly_to_pickup
  :parameters (?place - place ?city - city ?loc-from - airport ?loc-to - airport ?
      package - package ?airplane - airplane)
  :condition (and (at ?package ?loc-to) (at ?airplane ?loc-from) (in-city ?place ?
      city) (not (in-city ?loc-to ?city)))
  :goalCondition (and (at ?package ?place) )
  :action (fly-airplane ?airplane ?loc-from ?loc-to)
)
(:rule fly_to_dropoff
  :parameters (?place - place ?city - city ?loc-from - airport ?loc-to - airport ?
      package - package ?airplane - airplane)
  :condition (and (in ?package ?airplane) (at ?airplane ?loc-from) (in-city ?place ?
      city) (in-city ?loc-to ?city))
  :goalCondition (and (at ?package ?place) )
  :action (fly-airplane ?airplane ?loc-from ?loc-to)
)
(:rule drive_to_local_dropoff
  :parameters ( ?truck - truck ?city - city ?loc-to - place ?loc-from - place ?
      package - package)
  :condition (and (at ?truck ?loc-from) (in-city ?loc-from ?city) (in-city ?loc-to ?
      city) (in ?package ?truck))
  :goalCondition (and (at ?package ?loc-to))
  :action (drive-truck ?truck ?loc-from ?loc-to ?city)
)
)
```

D.2 ALMA

D.2.1 Driverlog

```
(define (policy driverlog_policy)
  (:domain driverlog)
  ;; Unload truck at goal
  (:rule driverlog_rule_1
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :condition (and (in ?obj ?truck) (at ?truck ?loc))
    :goalCondition (and (at ?obj ?loc) )
    :action (unload-truck ?obj ?truck ?loc)
  )
  ;; Load misplaced package
  (:rule driverlog_rule_2
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :condition (and (at ?obj ?loc) (at ?truck ?loc) )
    :goalCondition (and (not (at ?obj ?loc)))
    :action (load-truck ?obj ?truck ?loc)
  )
  ;; Drive truck to pickup a misplaced package that it is bound to deliver
  (:rule driverlog_rule_5
    :parameters (?truck - truck ?from ?to - location ?driver - driver ?obj - obj)
    :condition (and (at ?truck ?from) (at ?obj ?to) (drive_connected ?truck ?from ?to
        ) (driving ?driver ?truck) )
    :goalCondition (and (not (at ?obj ?to)))
    :action (drive_move ?truck ?from ?to)
  )
  ;; Drive truck to dropoff a package at its goal location
  (:rule driverlog_rule_6
    :parameters (?truck - truck ?from ?to - location ?driver - driver ?obj - obj)
    :condition (and (at ?truck ?from) (in ?obj ?truck) (driving ?driver ?truck) (
        drive_connected ?truck ?from ?to))
    :goalCondition (and (at ?obj ?to))
    :action (drive_move ?truck ?from ?to)
  )
  ;; Drive truck to its goal destination
  (:rule driverlog_rule_7
    :parameters (?truck - truck ?from ?to - location ?driver - driver)
    :condition (and (at ?truck ?from) (driving ?driver ?truck) (drive_connected ?
        truck ?from ?to))
    :goalCondition (and (at ?truck ?to) (not (at ?truck ?from)))
    :action (drive_move ?truck ?from ?to)
  )
  ;; Board the driver onto the truck if it is bound to drive the truck
  (:rule driverlog_rule_8
    :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (at ?obj ?l)(empty ?truck) (at ?truck ?loc))
    :goalCondition (and (not (at ?obj ?l)))
    :action (board-truck ?driver ?truck ?loc)
  )
  ;; Board the driver onto the truck if it is bound to drive the truck
  (:rule driverlog_rule_8
    :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (at ?truck ?loc)(empty ?truck) (in ?obj ?truck)
        )
    :goalCondition (and (at ?obj ?l))
    :action (board-truck ?driver ?truck ?loc)
  )
  (:rule driverlog_rule_9
    :parameters (?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (empty ?truck) (at ?truck ?loc))
    :goalCondition (and (not (at ?truck ?loc)) (at ?truck ?l))
    :action (board-truck ?driver ?truck ?loc)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_10
    :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
    :condition (and (at ?driver ?from) (empty ?truck)(at ?obj ?l)(at ?truck ?to) (
        walking_connected ?driver ?from ?to))
    :goalCondition (and (not (at ?obj ?l)))
    :action (walking_move ?driver ?from ?to)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_10
    :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
    :condition (and (at ?driver ?from) (empty ?truck)(walking_connected ?driver ?from
         ?to) (in ?obj ?truck)(at ?truck ?to))
    :goalCondition (and (at ?obj ?l))
    :action (walking_move ?driver ?from ?to)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_11
    :parameters (?driver - driver ?from ?to ?l - location ?truck - truck)
    :condition (and (at ?driver ?from) (walking_connected ?driver ?from ?to) (empty ?
        truck) (at ?truck ?to))
    :goalCondition (and (not (at ?truck ?to)) (at ?truck ?l))
    :action (walking_move ?driver ?from ?to)
  )
  ;; walk the driver to its goal location
  (:rule driverlog_rule_15
    :parameters (?driver - driver ?from ?to - location)
    :condition (and (at ?driver ?from) (walking_connected ?driver ?from ?to))
    :goalCondition (and (at ?driver ?to) (not (at ?driver ?from)))
    :action (walking_move ?driver ?from ?to)
  )
  (:rule driverlog_rule_16
    :parameters (?driver - driver ?truck - truck ?loc - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc))
    :goalCondition (and )
    :action (disembark-truck ?driver ?truck ?loc)
)
)
```

D.2.2 Goldminer

```
(define (policy goldminer-policy)
(:domain goldminer)
;; Pickup Holdable
(:rule goldminer_rule_1
  :parameters (?r - robot ?l1 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l1) (arm-empty ?r))
  :goalCondition (and (holds-gold ?r))
  :action (pickup-gold ?r ?l1)
)
(:rule goldminer_rule_2
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 - loc)
  :condition (and (not (open_connected ?r ?l1 ?l2)) (at-gold ?l2) (open_connected ?r
      ?l1 ?l3)
           (connected ?l3 ?l2) (at ?r ?l1) (at ?b ?l1) (arm-empty ?r))
  :goalCondition (and (holds-gold ?r))
  :action (pickup ?r ?b ?l1)
)
(:rule goldminer_rule_3
  :parameters (?r - robot ?l - laser ?l1 ?l2 - loc)
  :condition (and (not (open_connected ?r ?l1 ?l2)) (at-gold ?l2) (at ?r ?l1) (at ?l
      ?l1) (arm-empty ?r))
  :goalCondition (and (holds-gold ?r))
  :action (pickup ?r ?l ?l1)
)
```

```
;; Blockage Adjacent to Goal
(:rule goldminer_rule_4
  :parameters (?r - robot ?b - bomb ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2))
             (at-gold ?l2) (connected ?l1 ?l2) (holds ?b ?r))
  :goalCondition (and (holds-gold ?r))
  :action (detonate-bomb-1 ?r ?b ?l1 ?l2)
)
(:rule goldminer_rule_5
  :parameters (?r - robot ?b - bomb ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (open_connected ?r ?
      l1 ?l3)
             (connected ?l3 ?l2) (at-gold ?l2) (holds ?l ?r) (at ?b ?l1))
  :goalCondition (and (holds-gold ?r))
  :action (putdown ?r ?l ?l1)
)
;; Removed unnecessary rules.
(:rule goldminer_rule_9
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (open_connected ?r ?
      l1 ?l4)
             (connected ?l4 ?l2) (at-gold ?l2) (at ?b ?l3) (open_connected ?r ?l1 ?l3
                 ))
  :goalCondition (and (holds-gold ?r))
  :action (open_move ?r ?l1 ?l3)
)
(:rule goldminer_rule_10
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (open_connected ?r ?
      l1 ?l3)
             (at-gold ?l2) (connected ?l3 ?l2) (holds ?b ?r))
  :goalCondition (and (holds-gold ?r))
  :action (open_move ?r ?l1 ?l3)
)
;; Make a Path to the Goal
(:rule goldminer_rule_11a
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (soft-rock-at ?l3) (
      no-hard-rock ?l3)
             (at-gold ?l2) (holds ?l ?r) (connected ?l1 ?l3) (connected ?l3 ?l2))
  :goalCondition (and (holds-gold ?r))
  :action (fire-laser-0-1 ?r ?l ?l1 ?l3)
)
(:rule goldminer_rule_11b
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (no-soft-rock ?l3) (
      hard-rock-at ?l3)
             (at-gold ?l2) (holds ?l ?r) (connected ?l1 ?l3) (connected ?l3 ?l2))
  :goalCondition (and (holds-gold ?r))
  :action (fire-laser-1-0 ?r ?l ?l1 ?l3)
)
(:rule goldminer_rule_13
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not (open_connected ?r ?l1 ?l2)) (at-gold ?l2)
             (at ?l ?l3) (arm-empty ?r) (open_connected ?r ?l1 ?l3))
  :goalCondition (and (holds-gold ?r))
  :action (open_move ?r ?l1 ?l3)
)
(:rule goldminer_rule_14
  :parameters (?r - robot ?l1 ?l2 ?l3 - loc)
  :condition (and (at-gold ?l2) (connected ?l3 ?l2) (laser_open_connected ?r ?l1 ?l3)
      )
  :goalCondition (and (holds-gold ?r))
  :action (laser_open_move ?r ?l1 ?l3)
)
;; Move to Goal
(:rule goldminer_rule_15
  :parameters (?r - robot ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l2) (open_connected ?r ?l1 ?l2))
  :goalCondition (and (holds-gold ?r))
  :action (open_move ?r ?l1 ?l2)
)
)
```

D.2.3 Grid

```
(define (policy grid_policy)
  (:domain grid)
  (:rule driverlog_rule_1
    :parameters (?t - robot ?l1 ?l2 - place)
    :condition (and (at ?t ?l1) (unlockAndMove_connected ?t ?l1 ?l2))
    :goalCondition (and (at ?t ?l2) )
    :action (unlockAndMove_move ?t ?l1 ?l2)
  )
)
```

D.3 Partially bound rules

D.3.1 Driverlog

```
(define (policy driverlog_policy)
  (:domain driverlog)
  ;; Unload truck at goal
  (:rule driverlog_rule_1
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :condition (and (in ?obj ?truck) (at ?truck ?loc))
    :goalCondition (and (at ?obj ?loc) )
    :action (unload-truck ?obj ?truck ?loc)
  )
  ;; Load misplaced package
  (:rule driverlog_rule_2
    :parameters (?obj - obj ?truck - truck ?loc - location)
    :condition (and (at ?obj ?loc) (at ?truck ?loc) )
    :goalCondition (and (not (at ?obj ?loc)))
    :action (load-truck ?obj ?truck ?loc)
  )
  ;; Drive truck if there is a misplaced package
  (:rule driverlog_rule_5
    :parameters (?truck - truck ?from ?to ?l - location ?driver - driver ?obj - obj)
    :condition (and (at ?truck ?from) (at ?obj ?l) (driving ?driver ?truck) (link ?
        from ?to))
    :goalCondition (and (not (at ?obj ?l)))
    :action (drive-truck ?truck ?from ?to ?driver)
  )
  ;; Drive truck to dropoff a package at its goal location
  (:rule driverlog_rule_6
    :parameters (?truck - truck ?l ?from ?to - location ?driver - driver ?obj - obj)
    :condition (and (at ?truck ?from) (in ?obj ?truck) (driving ?driver ?truck) (link
         ?from ?to))
    :goalCondition (and (at ?obj ?l))
    :action (drive-truck ?truck ?from ?to ?driver)
  )
  ;; Drive truck to its goal destination
  (:rule driverlog_rule_7
    :parameters (?truck - truck ?from ?to ?l - location ?driver - driver)
    :condition (and (at ?truck ?from) (driving ?driver ?truck) (link ?from ?to))
    :goalCondition (and (not (at ?truck ?from)) (at ?truck ?l))
    :action (drive-truck ?truck ?from ?to ?driver)
  )
  ;; Board the driver onto the truck if it is bound to drive the truck
  (:rule driverlog_rule_8
    :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (at ?obj ?l) (at ?truck ?loc))
    :goalCondition (and (not (at ?obj ?l)))
    :action (board-truck ?driver ?truck ?loc)
  )
  ;; Board the driver onto the truck if it is bound to drive the truck
  (:rule driverlog_rule_8
    :parameters (?obj - obj ?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (at ?truck ?loc)(in ?obj ?truck))
    :goalCondition (and (at ?obj ?l))
    :action (board-truck ?driver ?truck ?loc)
  )
  (:rule driverlog_rule_9
    :parameters (?driver - driver ?loc ?l - location ?truck - truck)
    :condition (and (at ?driver ?loc) (at ?truck ?loc))
    :goalCondition (and (not (at ?truck ?loc)) (at ?truck ?l))
    :action (board-truck ?driver ?truck ?loc)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_10
    :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
    :condition (and (at ?driver ?from) (at ?obj ?l) (empty ?truck) (path ?from ?to))
    :goalCondition (and (not (at ?obj ?l)))
    :action (walk ?driver ?from ?to)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_10
    :parameters (?obj - obj ?driver - driver ?from ?to ?l - location ?truck - truck)
    :condition (and (at ?driver ?from) (in ?obj ?truck)(empty ?truck) (path ?from ?to
        ))
    :goalCondition (and (at ?obj ?l))
    :action (walk ?driver ?from ?to)
  )
  (:rule driverlog_rule_16
    :parameters (?obj - obj ?driver - driver ?truck ?t2 - truck ?loc ?l2 - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc) (empty ?t2)
                (in ?obj ?t2))
    :goalCondition (and (at ?obj ?l2))
    :action (disembark-truck ?driver ?truck ?loc)
  )
  (:rule driverlog_rule_16a
    :parameters (?driver - driver ?truck - truck ?loc ?l2 - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc) )
    :goalCondition (and (at ?driver ?l2) (at ?truck ?loc))
    :action (disembark-truck ?driver ?truck ?loc)
  )
  (:rule driverlog_rule_16b
    :parameters (?driver - driver ?truck - truck ?loc ?l2 - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc) )
    :goalCondition (and (at ?driver ?l2))
    :action (disembark-truck ?driver ?truck ?loc)
  )
  ;; Walk the driver to a truck that it is bound to drive
  (:rule driverlog_rule_11
    :parameters (?driver - driver ?from ?to ?l1 ?l2 - location ?truck - truck)
    :condition (and (at ?driver ?from) (empty ?truck) (at ?truck ?l1) (path ?from ?to
        ))
    :goalCondition (and (not (at ?truck ?l1)) (at ?truck ?l2))
    :action (walk ?driver ?from ?to)
  )
  (:rule driverlog_rule_16
    :parameters (?driver - driver ?truck ?t2 - truck ?loc ?l2 ?l3 - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc) (empty ?t2)
                (at ?t2 ?l2))
    :goalCondition (and (at ?t2 ?l3)(not (at ?t2 ?l2)))
    :action (disembark-truck ?driver ?truck ?loc)
  )
  ;; walk the driver to its goal location
  (:rule driverlog_rule_15
    :parameters (?driver - driver ?l ?from ?to - location)
    :condition (and (at ?driver ?from) (path ?from ?to))
    :goalCondition (and (not (at ?driver ?from)) (at ?driver ?l))
    :action (walk ?driver ?from ?to)
  )
  (:rule driverlog_rule_16
    :parameters (?driver - driver ?truck - truck ?loc - location)
    :condition (and (driving ?driver ?truck) (at ?truck ?loc))
    :goalCondition (and )
    :action (disembark-truck ?driver ?truck ?loc)
)
)
```

D.3.2 Goldminer

```
;; This policy relies on a heuristic, by using unguarded move and fire actions.
;; Even a perfect heuristic will make a 3 wide path to the bombable node;
;; this is because the move and fire actions must be ordered and the alternative
;; would be a free walk, ending in repeated state.

(define (policy goldminer-policy)
(:domain goldminer)
(:rule goldminer_rule_1
  :parameters (?r - robot ?l1 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l1) (arm-empty ?r))
  :goalCondition (and (holds-gold ?r))
  :action (pickup-gold ?r ?l1)
)
;; Move to Goal
(:rule goldminer_rule_15
  :parameters (?r - robot ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l3)(clear ?l3) (clear ?l2) (connected ?l1 ?l2
      ))
  :goalCondition (and (holds-gold ?r))
  :action (move ?r ?l1 ?l2)
)
(:rule goldminer_rule_4
  :parameters (?r - robot ?b - bomb ?l1 ?l2 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l2) (connected ?l1 ?l2) (holds ?b ?r) (
      not-clear ?l2) (soft-rock-at ?l2) (no-hard-rock ?l2))
  :goalCondition (and (holds-gold ?r))
  :action (detonate-bomb-1 ?r ?b ?l1 ?l2)
)
(:rule goldminer_rule_10
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l2) (connected ?l3 ?l2) (holds ?b ?r)
    (clear ?l3) (connected ?l1 ?l4) (clear ?l4))
  :goalCondition (and (holds-gold ?r))
  :action (move ?r ?l1 ?l4)
)
(:rule goldminer_rule_2
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 - loc)
  :condition (and (at-gold ?l2) (at ?r ?l1) (at ?b ?l1) (arm-empty ?r) (connected ?l3
      ?l2) (clear ?l3))
  :goalCondition (and (holds-gold ?r))
  :action (pickup ?r ?b ?l1)
)
(:rule goldminer_rule_9
  :parameters (?r - robot ?b - bomb ?l1 ?l2 ?l3 ?l4 ?l5 - loc)
  :condition (and (at ?r ?l1) (connected ?l3 ?l2) (clear ?l3)
              (at-gold ?l2) (at ?b ?l4) (connected ?l1 ?l5) (clear ?l5))
  :goalCondition (and (holds-gold ?r))
  :action (move ?r ?l1 ?l5)
)
(:rule goldminer_rule_5
  :parameters (?r - robot ?b - bomb ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (connected ?l3 ?l2) (clear ?l3)
              (at-gold ?l2) (holds ?l ?r) (at ?b ?l1))
  :goalCondition (and (holds-gold ?r))
  :action (putdown ?r ?l ?l1)
)
;; Make a Path to the Goal - these have to be before the move, or it would move
    forever.
(:rule goldminer_rule_11a
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (soft-rock-at ?l3) (no-hard-rock ?l3) (not-clear ?l3)
              (at-gold ?l2) (holds ?l ?r) (connected ?l1 ?l3))
  :goalCondition (and (holds-gold ?r))
  :action (fire-laser-0-1 ?r ?l ?l1 ?l3)
)
(:rule goldminer_rule_11b
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 - loc)
  :condition (and (at ?r ?l1) (not-clear ?l3) (no-soft-rock ?l3) (hard-rock-at ?l3)
            (at-gold ?l2) (holds ?l ?r) (connected ?l1 ?l3))
  :goalCondition (and (holds-gold ?r))
  :action (fire-laser-1-0 ?r ?l ?l1 ?l3)
)
(:rule goldminer_rule_14
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l4 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l2) (holds ?l ?r) (connected ?l1 ?l4) (clear
      ?l4))
  :goalCondition (and (holds-gold ?r))
  :action (move ?r ?l1 ?l4)
)
(:rule goldminer_rule_3
  :parameters (?r - robot ?l - laser ?l1 ?l2 - loc)
  :condition (and (at-gold ?l2) (at ?r ?l1) (at ?l ?l1) (arm-empty ?r))
  :goalCondition (and (holds-gold ?r))
  :action (pickup ?r ?l ?l1)
)
(:rule goldminer_rule_13
  :parameters (?r - robot ?l - laser ?l1 ?l2 ?l3 ?l4 - loc)
  :condition (and (at ?r ?l1) (at-gold ?l2) (at ?l ?l3) (arm-empty ?r) (connected ?l1
      ?l4) (clear ?l4))
  :goalCondition (and (holds-gold ?r))
  :action (move ?r ?l1 ?l4)
)
)
```

D.4 Generated seeds

In this section we present a selection of the policies generated by our policy generator. Although in each case the vocabulary was generated automatically we have gathered equivalent ALMAs and renamed them.

D.4.1 Driverlog

```
(define (policy Policy_1)
        (:domain driverlog)
(:rule Rule8
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?truck1 - truck ?s8
      - location)
  :condition (and (!= ?s8 ?s3) (at ?truck1 ?s3) (at ?package1 ?s8) (driving ?driver1
      ?truck1) (not (empty ?truck1)) (not (at ?driver1 ?s3)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (disembark-truck driver1 truck1 s3)
)
(:rule Rule9
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?truck1 - truck ?s8
      - location ?from_location - location)
  :condition (and (!= ?s8 ?s3) (at ?package1 ?s8) (driving ?driver1 ?truck1) (at ?
      truck1 ?from_location) (long_drive_connected ?truck1 ?from_location ?s3) (not (
      empty ?truck1)) (not (at ?driver1 ?s3)) (not (at ?truck1 ?s3)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (long_drive_move truck1 from_location s3)
)
(:rule Rule10
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?s8 - location ?
      truck1 - truck)
  :condition (and (at ?truck1 ?s8) (!= ?s8 ?s3) (in ?package1 ?truck1) (driving ?
      driver1 ?truck1) (long_drive_connected ?truck1 ?s8 ?s3) (not (empty ?truck1)) (
      not (at ?package1 ?s8)) (not (at ?driver1 ?s3)) (not (at ?truck1 ?s3)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (unload-truck package1 truck1 s8)
)
(:rule Rule11
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?s8 - location ?
      truck1 - truck ?from_location - location)
  :condition (and (long_drive_connected ?truck1 ?from_location ?s8) (!= ?s8 ?s3) (in
      ?package1 ?truck1) (driving ?driver1 ?truck1) (at ?truck1 ?from_location) (
      long_drive_connected ?truck1 ?from_location ?s3) (not (empty ?truck1)) (not (at
       ?package1 ?s8)) (not (at ?driver1 ?s3)) (not (at ?truck1 ?s8)) (not (at ?
      truck1 ?s3)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (long_drive_move truck1 from_location s8)
)
(:rule Rule12
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?s14 - location ?s8
      - location ?truck1 - truck)
  :condition (and (!= ?s14 ?s3) (long_drive_connected ?truck1 ?s14 ?s8) (at ?package1
       ?s14) (!= ?s8 ?s3) (!= ?s8 ?s14) (at ?truck1 ?s14) (driving ?driver1 ?truck1)
      (long_drive_connected ?truck1 ?s14 ?s3) (not (empty ?truck1)) (not (at ?
      package1 ?s8)) (not (at ?driver1 ?s3)) (not (at ?truck1 ?s8)) (not (at ?truck1
      ?s3)) (not (in ?package1 ?truck1)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (load-truck package1 truck1 s14)
)
(:rule Rule13
  :parameters ( ?driver1 - driver ?package1 - obj ?s3 - location ?s14 - location ?s8
      - location ?truck1 - truck ?from_location - location)
  :condition (and (long_drive_connected ?truck1 ?from_location ?s8) (!= ?s14 ?s3) (at
       ?package1 ?s14) (!= ?s8 ?s3) (driving ?driver1 ?truck1) (at ?truck1 ?
      from_location) (long_drive_connected ?truck1 ?from_location ?s14) (
      long_drive_connected ?truck1 ?from_location ?s3) (!= ?s14 ?s8) (not (empty ?
      truck1)) (not (at ?truck1 ?s14)) (not (at ?package1 ?s8)) (not (at ?driver1 ?s3
      )) (not (at ?truck1 ?s8)) (not (in ?package1 ?truck1)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (long_drive_move truck1 from_location s14)
)
(:rule Rule14
  :parameters ( ?package1 - obj ?driver1 - driver ?s3 - location ?s14 - location ?s8
      - location ?truck1 - truck ?s2 - location)
  :condition (and (empty ?truck1) (at ?driver1 ?s2) (!= ?s14 ?s3) (!= ?s3 ?s2) (at ?
      package1 ?s14) (!= ?s8 ?s3) (!= ?s14 ?s2) (!= ?s8 ?s14) (!= ?s8 ?s2) (at ?
      truck1 ?s2) (not (at ?truck1 ?s14)) (not (driving ?driver1 ?truck1)) (not (at ?
      package1 ?s8)) (not (long_drive_connected ?truck1 ?s2 ?s3)) (not (at ?driver1 ?
      s3)) (not (at ?truck1 ?s8)) (not (long_drive_connected ?truck1 ?s2 ?s14)) (not
      (at ?truck1 ?s3)) (not (in ?package1 ?truck1)) (not (long_drive_connected ?
      truck1 ?s2 ?s8)) )
  :goalCondition (and (at ?package1 ?s8) (at ?driver1 ?s3) )
  :action (board-truck driver1 truck1 s2)
)
)
```

D.4.2 Goldminer

```
(define (policy Policy_4)
  (:domain Goldminer)
(:rule Rule31
  :parameters ( ?robby - robot ?f2-2f - loc)
  :condition (and (at-gold ?f2-2f) (at ?robby ?f2-2f) (arm-empty ?robby) (not (
      no-gold ?f2-2f)) (not (holds-gold ?robby)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (pickup-gold robby f2-2f)
)
(:rule Rule32
  :parameters ( ?robby - robot ?f2-1f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (!= ?f2-2f ?f2-1f) (at ?robby ?from_loc) (bombmove_connected ?robby
       ?from_loc ?f2-2f) (connected ?f2-1f ?f2-2f) (at-gold ?f2-2f) (not (at ?robby ?
      f2-2f)) (not (no-gold ?f2-2f)) (not (holds-gold ?robby)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (bombmove_move robby from_loc f2-2f)
)
(:rule Rule33
  :parameters ( ?robby - robot ?bomb - bomb ?f2-0f - loc ?f1-0f - loc ?f2-1f - loc ?
      f2-2f - loc ?from_loc - loc)
  :condition (and (!= ?f2-2f ?f2-1f) (!= ?f2-1f ?f2-0f) (connected ?f2-1f ?f2-2f) (at
       ?bomb ?f1-0f) (not-clear ?f2-2f) (soft-rock-at ?f2-2f) (!= ?f2-2f ?f1-0f) (
      at-gold ?f2-2f) (clear ?f2-0f) (clear ?f2-1f) (!= ?f2-2f ?f2-0f) (!= ?f1-0f ?
      f2-1f) (arm-empty ?robby) (no-hard-rock ?f2-2f) (at ?robby ?f1-0f) (!= ?f1-0f ?
      f2-0f) (connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-2f)) (not (at ?robby ?
      f2-0f)) (not (bombmove_connected ?robby ?from_loc ?f2-2f)) (not (holds ?bomb ?
      robby)) (not (clear ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (no-gold ?f2-2f))
       (not (destroyed ?bomb)) (not (holds-gold ?robby)) (not (at ?robby ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (pickup robby bomb f1-0f)
)
(:rule Rule34
  :parameters ( ?robby - robot ?bomb - bomb ?f2-0f - loc ?f2-1f - loc ?f1-0f - loc ?
      f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (!= ?f2-2f ?f2-1f) (at ?robby ?from_loc) (
      longmove_connected ?robby ?from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (connected ?
      f2-1f ?f2-2f) (at ?bomb ?f1-0f) (not-clear ?f2-2f) (soft-rock-at ?f2-2f) (!= ?
      f2-2f ?f1-0f) (at-gold ?f2-2f) (clear ?f2-0f) (clear ?f2-1f) (!= ?f2-2f ?f2-0f)
       (!= ?f1-0f ?f2-1f) (arm-empty ?robby) (no-hard-rock ?f2-2f) (!= ?f1-0f ?f2-0f)
       (connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-2f)) (not (bombmove_connected ?
        robby ?from_loc ?f2-2f)) (not (holds ?bomb ?robby)) (not (at ?robby ?f1-0f)) (
        not (clear ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (no-gold ?f2-2f)) (not (
        destroyed ?bomb)) (not (holds-gold ?robby)) (not (at ?robby ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (longmove_move robby from_loc f1-0f)
)
(:rule Rule35
  :parameters ( ?robby - robot ?bomb - bomb ?laser - laser ?f2-0f - loc ?f2-1f - loc
        ?f1-0f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (!= ?f2-2f ?f2-1f) (longmove_connected ?robby ?
        from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (connected ?f2-1f ?f2-2f) (at ?bomb ?f1-0f)
         (not-clear ?f2-2f) (holds ?laser ?robby) (soft-rock-at ?f2-2f) (!= ?f2-2f ?
        f1-0f) (at-gold ?f2-2f) (clear ?f2-0f) (clear ?f2-1f) (!= ?f2-2f ?f2-0f) (at ?
        robby ?f2-0f) (!= ?f1-0f ?f2-1f) (no-hard-rock ?f2-2f) (!= ?f1-0f ?f2-0f) (
        connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-2f)) (not (bombmove_connected ?
        robby ?from_loc ?f2-2f)) (not (holds ?bomb ?robby)) (not (arm-empty ?robby)) (
        not (at ?robby ?f1-0f)) (not (at ?laser ?f2-0f)) (not (clear ?f2-2f)) (not (
        no-gold ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (destroyed ?bomb)) (not (
        holds-gold ?robby)) (not (at ?robby ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (putdown robby laser f2-0f)
)
(:rule Rule36
  :parameters ( ?robby - robot ?bomb - bomb ?laser - laser ?f2-0f - loc ?f2-1f - loc
        ?f1-0f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (not-clear ?f2-1f) (!= ?f2-2f ?f2-1f) (
        longmove_connected ?robby ?from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (at ?bomb ?f1-0f
        ) (connected ?f2-1f ?f2-2f) (not-clear ?f2-2f) (holds ?laser ?robby) (
        soft-rock-at ?f2-2f) (!= ?f2-2f ?f1-0f) (at-gold ?f2-2f) (clear ?f2-0f) (
        soft-rock-at ?f2-1f) (no-gold ?f2-1f) (!= ?f2-2f ?f2-0f) (at ?robby ?f2-0f) (
        no-hard-rock ?f2-1f) (!= ?f1-0f ?f2-1f) (no-hard-rock ?f2-2f) (!= ?f1-0f ?f2-0f
        ) (connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-2f)) (not (bombmove_connected ?
        robby ?from_loc ?f2-2f)) (not (holds ?bomb ?robby)) (not (arm-empty ?robby)) (
        not (at ?robby ?f1-0f)) (not (at ?laser ?f2-0f)) (not (clear ?f2-2f)) (not (
        no-gold ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (destroyed ?bomb)) (not (
        holds-gold ?robby)) (not (at ?robby ?f2-1f)) (not (no-soft-rock ?f2-1f)) (not (
        clear ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (fire-laser-0-1 robby laser f2-0f f2-1f)
)
(:rule Rule37
  :parameters ( ?robby - robot ?bomb - bomb ?laser - laser ?f2-0f - loc ?f2-1f - loc
        ?f1-0f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (not-clear ?f2-1f) (!= ?f2-2f ?f2-1f) (at ?robby ?
        from_loc) (firemove_connected ?robby ?from_loc ?f2-0f) (longmove_connected ?
        robby ?from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (at ?bomb ?f1-0f) (connected ?f2-1f
        ?f2-2f) (not-clear ?f2-2f) (holds ?laser ?robby) (soft-rock-at ?f2-2f) (!= ?
        f2-2f ?f1-0f) (at-gold ?f2-2f) (soft-rock-at ?f2-1f) (no-gold ?f2-1f) (!= ?
        f2-2f ?f2-0f) (no-hard-rock ?f2-1f) (!= ?f1-0f ?f2-1f) (no-hard-rock ?f2-2f)
        (!= ?f1-0f ?f2-0f) (connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-0f)) (not (at
        ?robby ?f2-2f)) (not (bombmove_connected ?robby ?from_loc ?f2-2f)) (not (holds
        ?bomb ?robby)) (not (arm-empty ?robby)) (not (at ?laser ?f2-0f)) (not (clear ?
        f2-2f)) (not (no-gold ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (destroyed ?
        bomb)) (not (holds-gold ?robby)) (not (at ?robby ?f2-1f)) (not (no-soft-rock ?
        f2-1f)) (not (clear ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (firemove_move robby from_loc f2-0f)
)
(:rule Rule38
  :parameters ( ?robby - robot ?bomb - bomb ?laser - laser ?f2-0f - loc ?f2-1f - loc
        ?f1-0f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (not-clear ?f2-1f) (!= ?f2-2f ?f2-1f) (
        longmove_connected ?robby ?from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (at ?bomb ?f1-0f
        ) (connected ?f2-1f ?f2-2f) (not-clear ?f2-2f) (soft-rock-at ?f2-2f) (!= ?f2-2f
         ?f1-0f) (at ?laser ?f1-0f) (at-gold ?f2-2f) (soft-rock-at ?f2-1f) (no-gold ?
        f2-1f) (!= ?f2-2f ?f2-0f) (no-hard-rock ?f2-1f) (!= ?f1-0f ?f2-1f) (arm-empty ?
        robby) (no-hard-rock ?f2-2f) (at ?robby ?f1-0f) (!= ?f1-0f ?f2-0f) (connected ?
        f2-0f ?f2-1f) (not (at ?robby ?f2-0f)) (not (at ?robby ?f2-2f)) (not (
        bombmove_connected ?robby ?from_loc ?f2-2f)) (not (holds ?bomb ?robby)) (not (
        at ?laser ?f2-0f)) (not (clear ?f2-2f)) (not (holds ?laser ?robby)) (not (
        no-gold ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (destroyed ?bomb)) (not (
        holds-gold ?robby)) (not (at ?robby ?f2-1f)) (not (no-soft-rock ?f2-1f)) (not (
        clear ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (pickup robby laser f1-0f)
)
(:rule Rule39
  :parameters ( ?robby - robot ?bomb - bomb ?laser - laser ?f2-0f - loc ?f2-1f - loc
        ?f1-0f - loc ?f2-2f - loc ?from_loc - loc)
  :condition (and (clear ?f1-0f) (not-clear ?f2-1f) (!= ?f2-2f ?f2-1f) (at ?robby ?
        from_loc) (longmove_connected ?robby ?from_loc ?f1-0f) (!= ?f2-1f ?f2-0f) (at ?
        bomb ?f1-0f) (connected ?f2-1f ?f2-2f) (not-clear ?f2-2f) (soft-rock-at ?f2-2f)
         (!= ?f2-2f ?f1-0f) (at ?laser ?f1-0f) (at-gold ?f2-2f) (soft-rock-at ?f2-1f) (
        no-gold ?f2-1f) (longmove_connected ?robby ?from_loc ?f1-0f) (!= ?f2-2f ?f2-0f)
         (no-hard-rock ?f2-1f) (!= ?f1-0f ?f2-1f) (arm-empty ?robby) (no-hard-rock ?
        f2-2f) (!= ?f1-0f ?f2-0f) (connected ?f2-0f ?f2-1f) (not (at ?robby ?f2-2f)) (
        not (bombmove_connected ?robby ?from_loc ?f2-2f)) (not (holds ?bomb ?robby)) (
        not (at ?robby ?f1-0f)) (not (at ?laser ?f2-0f)) (not (clear ?f2-2f)) (not (
        holds ?laser ?robby)) (not (no-gold ?f2-2f)) (not (no-soft-rock ?f2-2f)) (not (
        destroyed ?bomb)) (not (holds-gold ?robby)) (not (at ?robby ?f2-1f)) (not (
        no-soft-rock ?f2-1f)) (not (clear ?f2-1f)) )
  :goalCondition (and (holds-gold ?robby) )
  :action (longmove_move robby from_loc f1-0f)
)
)
```

D.4.3 Structure briefcase

```
(define (policy Policy_3)
  (:domain SBC)
(:rule Rule8
  :parameters ( ?package1 - obj ?truck1 - truck ?s2 - location)
  :condition (and (at ?truck1 ?s2) (in ?package1 ?truck1) (not (at ?package1 ?s2)) )
  :goalCondition (and (at ?package1 ?s2) )
  :action (unload-truck package1 truck1 s2)
)
(:rule Rule9
  :parameters ( ?package1 - obj ?truck1 - truck ?s2 - location ?from_location -
        location)
  :condition (and (at ?truck1 ?from_location) (in ?package1 ?truck1) (
        long_drive_connected ?truck1 ?from_location ?s2) (not (at ?package1 ?s2)) (not
        (at ?truck1 ?s2)) )
  :goalCondition (and (at ?package1 ?s2) )
  :action (long_drive_move truck1 from_location s2)
)
(:rule Rule10
  :parameters ( ?package1 - obj ?s4 - location ?truck1 - truck ?s2 - location)
  :condition (and (at ?package1 ?s4) (at ?truck1 ?s4) (long_drive_connected ?truck1 ?
        s4 ?s2) (!= ?s2 ?s4) (not (at ?package1 ?s2)) (not (at ?truck1 ?s2)) (not (in ?
        package1 ?truck1)) )
  :goalCondition (and (at ?package1 ?s2) )
  :action (load-truck package1 truck1 s4)
)
(:rule Rule11
  :parameters ( ?package1 - obj ?s4 - location ?truck1 - truck ?s2 - location ?
        from_location - location)
  :condition (and (at ?package1 ?s4) (at ?truck1 ?from_location) (
        long_drive_connected ?truck1 ?from_location ?s4) (!= ?s2 ?s4) (
        long_drive_connected ?truck1 ?from_location ?s2) (not (at ?package1 ?s2)) (not
        (at ?truck1 ?s4)) (not (in ?package1 ?truck1)) )
  :goalCondition (and (at ?package1 ?s2) )
  :action (long_drive_move truck1 from_location s4)
)
)
```

D.5 Learned

We present the learned Driverlog and Goldminer policies, as these are the main contributions from this work. Rule names have been changed by hand to assist comprehension.

D.5.1 Driverlog

```
(define (policy a6119)
(:domain driverlog)
(:rule load-misplaced-package
  :parameters ( ?s4 - location ?obj - obj ?truck - truck ?loc - location ?driver1 -
      driver)
  :condition (and (driving ?driver1 ?truck) (at ?obj ?loc) (at ?truck ?loc) (not (in
      ?obj ?truck)) (not (at ?obj ?s4)) (not (at ?truck ?s4)) (!= ?loc ?s4) )
  :goalCondition (and (at ?obj ?s4) )
  :action (load-truck ?obj ?truck ?loc)
)
(:rule disembarkAtDriverGoal-withoutFullConsideration ; partially protected rule
  :parameters ( ?driver - driver ?package1 - obj ?s4 - location ?Add_truck1892 -
      truck ?truck - truck ?loc - location)
  :condition (and (at ?truck ?loc) (driving ?driver ?truck) (at ?package1 ?s4) (not (
      empty ?truck)) (not (at ?driver ?loc)) (not (driving ?driver ?Add_truck1892))
      (!= ?loc ?s4) )
  :goalCondition (and (at ?package1 ?s4) (at ?driver ?loc) (not (at ?truck ?loc)) )
  :action (disembark-truck ?driver ?truck ?loc)
)
(:rule unloadAtGoal
  :parameters ( ?obj - obj ?s3 - location ?truck - truck ?loc - location ?driver1 -
      driver)
  :condition (and (at ?truck ?loc) (in ?obj ?truck) (at ?truck ?loc) (driving ?
      driver1 ?truck) (not (empty ?truck)) (not (at ?obj ?loc)) (!= ?s3 ?loc) )
  :goalCondition (and (at ?obj ?loc) (at ?driver1 ?s3) )
  :action (unload-truck ?obj ?truck ?loc)
)
(:rule moveToDropoff
  :parameters ( ?package1 - obj ?loc-from - location ?loc-to - location ?s3 -
      location ?truck - truck ?driver1 - driver)
  :condition (and (at ?truck ?loc-from) (in ?package1 ?truck) (driving ?driver1 ?
      truck) (not (empty ?truck)) (not (at ?truck ?loc-to)) (not (at ?package1 ?
      loc-to)) (not (at ?driver1 ?s3)) (long_drive_connected ?truck ?loc-from ?s3) (
      long_drive_connected ?truck ?loc-from ?loc-to) (!= ?s3 ?loc-to))
  :goalCondition (and (at ?package1 ?loc-to) (at ?driver1 ?s3) (not (at ?truck ?
      loc-from)) (not (at ?truck ?loc-to)) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule walkDriverToObjectIfPathConnected ; Unlikely, because macro application
  :parameters ( ?driver - driver ?loc-from - location ?loc-to - location ?added_obj1
      - obj)
  :condition (and (at ?added_obj1 ?loc-to) (at ?driver ?loc-from) (path ?loc-from ?
      loc-to) )
  :goalCondition (and )
  :action (walk ?driver ?loc-from ?loc-to)
)
(:rule boardIfUndeliveredPackage
  :parameters ( ?driver - driver ?package1 - obj ?s4 - location ?s13 - location ?
      truck - truck ?s3 - location ?loc - location)
  :condition (and (at ?driver ?loc) (at ?truck ?loc) (empty ?truck) (not (at ?driver
      ?s3)) (not (at ?package1 ?s4)) (!= ?s3 ?s13) (!= ?loc ?s3) (!= ?loc ?s4) (!= ?
      loc ?s13) (!= ?s13 ?s4) )
  :goalCondition (and (at ?driver ?s3) (at ?package1 ?s4) )
  :action (board-truck ?driver ?truck ?loc)
```

```
)
(:rule walkToBoardMisplacedTruck
  :parameters ( ?driver - driver ?truck1 - truck ?package1 - obj ?s4 - location ?
      loc-from - location ?s14 - location ?loc-to - location ?s13 - location ?s3 -
      location ?s7 - location)
  :condition (and (at ?package1 ?s13) (empty ?truck1) (at ?package1 ?s7) (at ?truck1
      ?loc-to) (at ?driver ?loc-from) (not (in ?package1 ?truck1)) (not (at ?truck1 ?
      s3)) (not (at ?package1 ?s4)) (!= ?s13 ?s4) (!= ?s14 ?loc-to) (!= ?s7 ?loc-to)
      (!= ?loc-to ?s4) (!= ?s3 ?s13) (long_walk_connected ?driver ?loc-from ?loc-to)
      (not (long_drive_connected ?truck1 ?loc-to ?s4)) )
  :goalCondition (and (at ?package1 ?s7) (at ?truck1 ?s14) (not (at ?driver ?loc-to))
      )
  :action (long_walk_move ?driver ?loc-from ?loc-to)
)
(:rule unloadAtGoal
  :parameters ( ?obj - obj ?truck - truck ?loc - location)
  :condition (and (at ?truck ?loc) (in ?obj ?truck) )
  :goalCondition (and (at ?obj ?loc) )
  :action (unload-truck ?obj ?truck ?loc)
)
(:rule Not-unifiable
  :parameters ( ?Add_location8252 - location ?Add_location8253 - location ?loc-from -
       location ?loc-to - location ?truck - truck)
  :condition (and (at ?truck ?loc-from) (not (path ?Add_location8252 ?loc-to)) (not (
      path ?loc-from ?loc-to)) (long_drive_connected ?truck ?loc-from ?loc-to) )
  :goalCondition (and (at ?truck ?loc-from) (at ?truck ?Add_location8253) (not (at ?
      truck ?loc-from)) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule DriveToPickupPackage
  :parameters ( ?package1 - obj ?s4 - location ?loc-from - location ?loc-to -
      location ?truck - truck ?s3 - location ?driver1 - driver)
  :condition (and (at ?package1 ?loc-to) (at ?truck ?loc-from) (not (at ?driver1 ?s3)
      ) (not (at ?package1 ?s4)) (!= ?s3 ?s4) (!= ?loc-to ?s4) (!= ?s3 ?loc-to) (
      long_drive_connected ?truck ?loc-from ?loc-to) )
  :goalCondition (and (at ?package1 ?s4) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule Disembark-atDrivergoal ; an example of where version spaces would have helped.
  :parameters ( ?driver - driver ?package1 - obj ?s4 - location ?truck - truck ?loc -
       location)
  :condition (and (at ?truck ?loc) (driving ?driver ?truck) (at ?package1 ?s4) (not (
      empty ?truck)) (not (at ?driver ?loc)) (!= ?loc ?s4) )
  :goalCondition (and (at ?package1 ?s4) (at ?driver ?loc) )
  :action (disembark-truck ?driver ?truck ?loc)
)
(:rule Unloadpackage ; again.
  :parameters ( ?obj - obj ?truck - truck ?loc - location)
  :condition (and (at ?truck ?loc) (in ?obj ?truck) (not (!= ?truck ?truck)) )
  :goalCondition (and (at ?obj ?loc) )
  :action (unload-truck ?obj ?truck ?loc)
)
(:rule moveToDropoff ; again
  :parameters ( ?package1 - obj ?loc-from - location ?loc-to - location ?s3 -
      location ?truck - truck ?driver1 - driver)
  :condition (and (in ?package1 ?truck) (driving ?driver1 ?truck) (at ?truck ?
      loc-from) (not (at ?package1 ?loc-to)) (not (at ?driver1 ?s3)) (
      long_drive_connected ?truck ?loc-from ?loc-to) )
  :goalCondition (and (at ?package1 ?loc-to) (at ?driver1 ?s3) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule moveToDropoff ; again
  :parameters ( ?package1 - obj ?loc-from - location ?loc-to - location ?truck -
      truck ?added_truck0 - truck ?driver1 - driver)
  :condition (and (at ?truck ?loc-from) (driving ?driver1 ?truck) (in ?package1 ?
      truck) (not (empty ?added_truck0)) (not (at ?package1 ?loc-to)) (
      long_drive_connected ?truck ?loc-from ?loc-to) )
  :goalCondition (and (at ?package1 ?loc-to) (not (at ?truck ?loc-to)) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule moveDriverToGoal
  :parameters ( ?Add_truck6600 - truck ?truck1 - truck ?driver - driver ?package1 -
      obj ?loc-from - location ?s14 - location ?loc-to - location ?s7 - location)
  :condition (and (at ?truck1 ?s14) (at ?driver ?loc-from) (at ?driver ?loc-from) (at
       ?package1 ?s7) (not (at ?driver ?loc-to)) (long_walk_connected ?driver ?
      loc-from ?loc-to) (!= ?s14 ?s7) (!= ?s7 ?loc-to) )
  :goalCondition (and (at ?driver ?loc-to) (at ?truck1 ?s14) (at ?package1 ?s7) (at ?
      Add_truck6600 ?loc-from) )
  :action (long_walk_move ?driver ?loc-from ?loc-to)
)
(:rule BoardIfMisplacedDriverAndTruckHasGoal ; not quite..
  :parameters ( ?Add_location670 - location ?driver - driver ?truck - truck ?loc -
      location)
  :condition (and (empty ?truck) (at ?driver ?loc) (at ?truck ?loc) (not (driving ?
      driver ?truck)) )
  :goalCondition (and (at ?truck ?Add_location670) (not (at ?driver ?loc)) )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule moveTruckHome
  :parameters ( ?loc-from - location ?loc-to - location ?truck - truck)
  :condition (and (at ?truck ?loc-from) (long_drive_connected ?truck ?loc-from ?
      loc-to) )
  :goalCondition (and (at ?truck ?loc-to))
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule Not-unifiable
  :parameters ( ?driver - driver ?truck - truck ?loc - location)
  :condition (and (path ?loc ?loc) (link ?loc ?loc) (at ?truck ?loc) (at ?driver ?loc
      ) (empty ?truck) (!= ?truck ?truck) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule Not-unifiable
  :parameters ( ?driver - driver ?truck - truck ?loc - location)
  :condition (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck) (!= ?truck ?truck
      ) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule moveDriverToGoal
  :parameters ( ?truck1 - truck ?driver - driver ?package1 - obj ?loc-from - location
       ?s14 - location ?loc-to - location ?s7 - location)
  :condition (and (at ?truck1 ?s14) (at ?driver ?loc-from) (at ?package1 ?s7) (not (
      at ?driver ?loc-to)) (long_walk_connected ?driver ?loc-from ?loc-to) (!= ?s14 ?
      s7) (!= ?s7 ?loc-to) )
  :goalCondition (and (at ?driver ?loc-to) (at ?truck1 ?s14) (at ?package1 ?s7) )
  :action (long_walk_move ?driver ?loc-from ?loc-to)
)
(:rule driverToADriversGoal
  :parameters ( ?loc-from - location ?Add_driver6968 - driver ?loc-to - location ?
      Add_location5236 - location ?Add_truck5231 - truck ?truck - truck)
  :condition (and (at ?truck ?loc-from) (not (empty ?truck)) (not (path ?loc-from ?
      Add_location5236)) (long_drive_connected ?truck ?loc-from ?loc-to) )
  :goalCondition (and (at ?Add_driver6968 ?loc-to) (not (at ?Add_truck5231 ?loc-to))
      (not (at ?truck ?loc-from)) )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule Not-unifiable
  :parameters ( ?driver - driver ?truck - truck ?loc - location)
  :condition (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck) (!= ?truck ?truck
      ) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
```

```
(:rule Not-unifiable
  :parameters ( ?driver - driver ?truck - truck ?loc - location)
  :condition (and (path ?loc ?loc) (link ?loc ?loc) (at ?truck ?loc) (at ?driver ?loc
      ) (empty ?truck) (!= ?truck ?truck) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule disembarkTruck
  :parameters ( ?added_location1 - location ?driver - driver ?truck - truck ?
      Add_truck13886 - truck ?loc - location)
  :condition (and (at ?truck ?loc) (driving ?driver ?truck) (long_drive_connected ?
      Add_truck13886 ?loc ?added_location1) (not (!= ?truck ?truck)) )
  :goalCondition (and )
  :action (disembark-truck ?driver ?truck ?loc)
)
(:rule Not-unifiable
  :parameters ( ?driver - driver ?truck - truck ?loc - location)
  :condition (and (path ?loc ?loc) (link ?loc ?loc) (at ?truck ?loc) (at ?driver ?loc
      ) (empty ?truck) (!= ?truck ?truck) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule Board
  :parameters ( ?driver - driver ?Add_location12685 - location ?truck - truck ?loc -
      location)
  :condition (and (at ?truck ?loc) (at ?driver ?loc) (empty ?truck) (not (path ?
      Add_location12685 ?loc)) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
(:rule WalkToATruck
  :parameters ( ?truck1 - truck ?driver - driver ?package1 - obj ?s4 - location ?
      loc-from - location ?loc-to - location ?s13 - location ?s3 - location)
  :condition (and (at ?driver ?loc-from) (empty ?truck1) (at ?truck1 ?loc-to) (not (
      at ?truck1 ?s4)) (not (at ?truck1 ?s13)) (not (at ?package1 ?s4)) (not (at ?
      truck1 ?s3)) (long_walk_connected ?driver ?loc-from ?loc-to) (!= ?s3 ?s13) (!=
      ?loc-to ?s3) (!= ?loc-to ?s4) (!= ?loc-to ?s13) (!= ?s13 ?s4) (not (
      long_drive_connected ?truck1 ?loc-to ?s3)) (not (long_drive_connected ?truck1 ?
      loc-to ?s13)) (not (long_drive_connected ?truck1 ?loc-to ?s4)) )
  :goalCondition (and (at ?package1 ?s4) )
  :action (long_walk_move ?driver ?loc-from ?loc-to)
)
(:rule DriveTruck
  :parameters ( ?loc-from - location ?loc-to - location ?truck - truck ?added_truck0
      - truck)
  :condition (and (at ?truck ?loc-from) (not (empty ?added_truck0)) (
      long_drive_connected ?truck ?loc-from ?loc-to))
  :goalCondition (and )
  :action (long_drive_move ?truck ?loc-from ?loc-to)
)
(:rule Board
  :parameters ( ?driver - driver ?Add_location2344 - location ?truck - truck ?loc -
      location)
  :condition (and (empty ?truck) (at ?truck ?loc) (at ?driver ?loc) (not (path ?
      Add_location2344 ?Add_location2344)) (not (driving ?driver ?truck)) )
  :goalCondition (and )
  :action (board-truck ?driver ?truck ?loc)
)
)
```

D.5.2 Goldminer

```
(define (policy a877)
(:domain goldminer)
```

```
(:rule PickupGold
  :parameters ( ?x - loc ?r - robot)
  :condition (and (at-gold ?x) (arm-empty ?r) (at ?r ?x) (not (holds-gold ?r)) (not (
      no-gold ?x)) )
  :goalCondition (and (holds-gold ?r) )
  :action (pickup-gold ?r ?x)
)
(:rule MoveToGoldSquare
  :parameters ( ?x - loc ?y - loc ?f0-2f - loc ?r - robot)
  :condition (and (at ?r ?x) (at-gold ?y) (connected ?f0-2f ?y) (not (no-gold ?y)) (
      not (at ?r ?y)) (not (holds-gold ?r)) (!= ?f0-2f ?y) (BombMoveBag_connected ?r
      ?x ?y))
  :goalCondition (and (holds-gold ?r) )
  :action (BombMoveBag_move ?r ?x ?y)
)
(:rule EmptyHand ; Use protected by previous rule
  :parameters ( ?x - loc ?y - loc ?added_robot0 - robot ?b - bomb ?r - robot)
  :condition (and (arm-empty ?r) (at ?r ?x) (holds ?b ?r) (connected ?x ?y) (
      soft-rock-at ?y) (no-hard-rock ?y) (not-clear ?y) )
  :goalCondition (and (holds-gold ?added_robot0) )
  :action (detonate-bomb-1 ?r ?b ?x ?y)
)
(:rule PickupBomb ; location next to gold is clear
  :parameters ( ?x - loc ?f0-2f - loc ?f0-3f - loc ?h - bomb ?f0-1f - loc ?r - robot)
  :condition (and (at ?h ?x) (at-gold ?f0-3f) (not-clear ?f0-3f) (clear ?f0-1f) (
      clear ?f0-2f) (arm-empty ?r) (at ?r ?x) (no-hard-rock ?f0-3f) (soft-rock-at ?
      f0-3f) (connected ?f0-2f ?f0-3f) (connected ?f0-1f ?f0-2f) (not (no-gold ?f0-3f
      )) (not (holds-gold ?r)) (not (at ?r ?f0-2f)) (not (no-soft-rock ?f0-3f)) (not
      (destroyed ?h)) (not (at ?r ?f0-1f)) (not (holds ?h ?r)) (not (at ?r ?f0-3f)) (
      not (clear ?f0-3f)) (!= ?f0-1f ?x) (!= ?f0-2f ?x) (!= ?f0-2f ?f0-3f) (!= ?x ?
      f0-3f) (!= ?f0-1f ?f0-3f) (!= ?f0-1f ?f0-2f) (not (BombMoveBag_connected ?r ?x
      ?f0-3f)) )
  :goalCondition (and (holds-gold ?r) )
  :action (pickup ?r ?h ?x)
)
(:rule PutdownLaser ; slightly over protected by (not (at ?r ?f0-0f))
  :parameters ( ?x - loc ?bomb - bomb ?f0-2f - loc ?f0-0f - loc ?h - laser ?f0-3f -
      loc ?r - robot)
  :condition (and (at ?r ?x) (holds ?h ?r) (clear ?x) (at-gold ?f0-3f) (clear ?f0-0f)
       (connected ?f0-2f ?f0-3f) (clear ?f0-2f) (soft-rock-at ?f0-3f) (connected ?x ?
      f0-2f) (at ?bomb ?f0-0f) (not-clear ?f0-3f) (no-hard-rock ?f0-3f) (not (clear ?
      f0-3f)) (not (holds ?bomb ?r)) (not (at ?r ?f0-2f)) (not (destroyed ?bomb)) (
      not (at ?h ?x)) (not (no-gold ?f0-3f)) (not (at ?r ?f0-0f)) (not (at ?r ?f0-3f)
      ) (not (holds-gold ?r)) (not (no-soft-rock ?f0-3f)) (not (arm-empty ?r)) (!= ?x
       ?f0-3f) (!= ?x ?f0-2f) (MoveBag_connected ?r ?x ?f0-0f) (!= ?f0-0f ?f0-3f) (!=
       ?f0-2f ?f0-3f) (!= ?x ?f0-0f) (!= ?f0-2f ?f0-0f) (not (BombMoveBag_connected ?
      r ?x ?f0-3f)) )
  :goalCondition (and (holds-gold ?r) )
  :action (putdown ?r ?h ?x)
)
(:rule PickupBomb ; for smaller grids
  :parameters ( ?x - loc ?f0-2f - loc ?h - bomb ?f0-1f - loc ?r - robot)
  :condition (and (no-hard-rock ?f0-2f) (not-clear ?f0-2f) (at ?h ?x) (connected ?
      f0-1f ?f0-2f) (arm-empty ?r) (at-gold ?f0-2f) (soft-rock-at ?f0-2f) (connected
      ?x ?f0-1f) (clear ?f0-1f) (at ?r ?x) (not (at ?r ?f0-2f)) (not (holds-gold ?r))
       (not (no-gold ?f0-2f)) (not (no-soft-rock ?f0-2f)) (not (holds ?h ?r)) (not (
      clear ?f0-2f)) (not (at ?r ?f0-1f)) (not (destroyed ?h)) (!= ?f0-1f ?x) (!= ?
      f0-1f ?f0-2f) (!= ?f0-2f ?x) (not (BombMoveBag_connected ?r ?x ?f0-2f)) )
  :goalCondition (and (holds-gold ?r) )
  :action (pickup ?r ?h ?x)
)
(:rule FireGoldAdjacentSquare
  :parameters ( ?x - loc ?bomb - bomb ?y - loc ?f0-0f - loc ?f0-3f - loc ?l - laser ?
      r - robot)
  :condition (and (connected ?y ?f0-3f) (clear ?x) (no-hard-rock ?f0-3f) (at ?r ?x) (
      holds ?l ?r) (not-clear ?f0-3f) (at ?bomb ?f0-0f) (at-gold ?f0-3f) (no-gold ?y)
       (not-clear ?y) (soft-rock-at ?y) (soft-rock-at ?f0-3f) (clear ?f0-0f) (
        connected ?x ?y) (no-soft-rock ?x) (no-hard-rock ?y) (not (at ?r ?f0-0f)) (not
        (clear ?y)) (not (holds-gold ?r)) (not (clear ?f0-3f)) (not (at ?r ?y)) (not (
        arm-empty ?r)) (not (destroyed ?bomb)) (not (at ?r ?f0-3f)) (not (no-soft-rock
        ?y)) (not (no-gold ?f0-3f)) (not (holds ?bomb ?r)) (not (no-soft-rock ?f0-3f))
        (!= ?y ?f0-3f) (MoveBag_connected ?r ?x ?f0-0f) (!= ?x ?f0-3f) (!= ?x ?f0-0f)
        (!= ?x ?y) (!= ?f0-0f ?f0-3f) (!= ?y ?f0-0f) (not (BombMoveBag_connected ?r ?x
        ?f0-3f)) )
   :goalCondition (and (holds-gold ?r) )
   :action (fire-laser-0-1 ?r ?l ?x ?y)
)
(:rule MoveToPickupBomb
   :parameters ( ?x - loc ?bomb - bomb ?y - loc ?f0-2f - loc ?f0-3f - loc ?f0-1f - loc
         ?r - robot)
   :condition (and (not-clear ?f0-3f) (at ?r ?x) (at ?bomb ?y) (soft-rock-at ?f0-3f) (
        connected ?f0-1f ?f0-2f) (at-gold ?f0-3f) (connected ?f0-2f ?f0-3f) (clear ?
        f0-2f) (arm-empty ?r) (no-hard-rock ?f0-3f) (clear ?y) (clear ?f0-1f) (not (
        destroyed ?bomb)) (not (at ?r ?y)) (not (no-soft-rock ?f0-3f)) (not (clear ?
        f0-3f)) (not (holds ?bomb ?r)) (not (at ?r ?f0-2f)) (not (holds-gold ?r)) (not
        (no-gold ?f0-3f)) (not (at ?r ?f0-3f)) (!= ?f0-1f ?y) (!= ?f0-2f ?y) (!= ?y ?
        f0-3f) (MoveBag_connected ?r ?x ?y) (!= ?f0-1f ?f0-3f) (!= ?f0-2f ?f0-3f) (!= ?
        f0-1f ?f0-2f) (not (BombMoveBag_connected ?r ?x ?f0-3f)) )
   :goalCondition (and (holds-gold ?r) )
   :action (MoveBag_move ?r ?x ?y)
)
(:rule PickupLaser
   :parameters ( ?x - loc ?bomb - bomb ?f0-2f - loc ?h - laser ?f0-3f - loc ?f0-1f -
        loc ?r - robot)
   :condition (and (soft-rock-at ?f0-3f) (at ?bomb ?x) (at ?h ?x) (connected ?f0-1f ?
        f0-2f) (soft-rock-at ?f0-2f) (clear ?x) (not-clear ?f0-2f) (no-hard-rock ?f0-3f
        ) (at ?r ?x) (at-gold ?f0-3f) (connected ?f0-2f ?f0-3f) (arm-empty ?r) (no-gold
         ?f0-2f) (not-clear ?f0-3f) (no-hard-rock ?f0-2f) (not (clear ?f0-3f)) (not (at
         ?r ?f0-1f)) (not (clear ?f0-2f)) (not (holds ?h ?r)) (not (at ?h ?f0-1f)) (not
         (no-gold ?f0-3f)) (not (no-soft-rock ?f0-2f)) (not (holds ?bomb ?r)) (not (at
        ?r ?f0-3f)) (not (holds-gold ?r)) (not (no-soft-rock ?f0-3f)) (not (at ?r ?
        f0-2f)) (not (destroyed ?bomb)) (!= ?f0-2f ?f0-3f) (!= ?f0-1f ?f0-2f) (
        MoveBag_connected ?r ?x ?x) (!= ?f0-2f ?x) (!= ?f0-1f ?x) (!= ?x ?f0-3f) (!= ?
        f0-1f ?f0-3f) (not (BombMoveBag_connected ?r ?x ?f0-3f)) )
   :goalCondition (and (holds-gold ?r) )
   :action (pickup ?r ?h ?x)
)
(:rule OpenPathTowardsGold
   :parameters ( ?x - loc ?bomb - bomb ?y - loc ?f0-2f - loc ?laser - laser ?f0-0f -
        loc ?f0-3f - loc ?r - robot)
   :condition (and (holds ?laser ?r) (at ?bomb ?f0-0f) (not-clear ?f0-2f) (connected ?
        f0-2f ?f0-3f) (no-hard-rock ?f0-2f) (clear ?f0-0f) (at-gold ?f0-3f) (
        no-hard-rock ?f0-3f) (at ?r ?x) (not-clear ?f0-3f) (no-gold ?f0-2f) (connected
        ?y ?f0-2f) (soft-rock-at ?f0-2f) (soft-rock-at ?f0-3f) (not (at ?laser ?y)) (
        not (arm-empty ?r)) (not (destroyed ?bomb)) (not (at ?r ?f0-2f)) (not (clear ?
        f0-2f)) (not (holds ?bomb ?r)) (not (holds-gold ?r)) (not (clear ?f0-3f)) (not
        (no-gold ?f0-3f)) (not (no-soft-rock ?f0-2f)) (not (no-soft-rock ?f0-3f)) (not
        (at ?r ?y)) (not (at ?r ?f0-3f)) (!= ?y ?f0-3f) (!= ?y ?f0-0f) (!= ?f0-2f ?
        f0-0f) (!= ?f0-2f ?f0-3f) (FireMoveBag_connected ?r ?x ?y) (MoveBag_connected ?
        r ?x ?f0-0f) (!= ?f0-0f ?f0-3f) (!= ?y ?f0-2f) (not (BombMoveBag_connected ?r ?
        x ?f0-3f)) )
   :goalCondition (and (holds-gold ?r) )
   :action (FireMoveBag_move ?r ?x ?y)
)
(:rule MoveToPickupLaser
   :parameters ( ?x - loc ?bomb - bomb ?y - loc ?f0-2f - loc ?laser - laser ?f0-3f -
        loc ?f0-1f - loc ?r - robot)
   :condition (and (not-clear ?f0-3f) (connected ?f0-1f ?f0-2f) (not-clear ?f0-2f) (at
         ?r ?x) (no-hard-rock ?f0-2f) (connected ?f0-2f ?f0-3f) (no-gold ?f0-2f) (
        at-gold ?f0-3f) (at ?bomb ?y) (soft-rock-at ?f0-3f) (at ?laser ?y) (clear ?y) (
        soft-rock-at ?f0-2f) (arm-empty ?r) (no-hard-rock ?f0-3f) (not (at ?r ?f0-2f))
        (not (no-gold ?f0-3f)) (not (at ?laser ?f0-1f)) (not (clear ?f0-3f)) (not (
        clear ?f0-2f)) (not (no-soft-rock ?f0-3f)) (not (destroyed ?bomb)) (not (holds
        ?bomb ?r)) (not (no-soft-rock ?f0-2f)) (not (holds ?laser ?r)) (not (at ?r ?y))
        (not (at ?r ?f0-3f)) (not (holds-gold ?r)) (!= ?f0-2f ?y) (MoveBag_connected ?
        r ?x ?y) (!= ?f0-1f ?y) (!= ?y ?f0-3f) (!= ?f0-1f ?f0-2f) (!= ?f0-1f ?f0-3f)
        (!= ?f0-2f ?f0-3f) (not (BombMoveBag_connected ?r ?x ?f0-3f)) )
  :goalCondition (and (holds-gold ?r) )
  :action (MoveBag_move ?r ?x ?y)
)
)
```

APPENDIX E

FURTHER ANALYSIS

In this appendix we present some further analysis of our approach.

E.1 Quality of solutions

In this section we investigate the quality of the solutions that are generated using our approach. We compare the plan quality of the solutions that we computed above against an optimal planner, where it can solve the problem, and TLPLAN.

E.1.1 Analysis of plan length

A common conception of rule-based policy application is that a plan will be generated quickly, but the quality might be poor. However, in Depots, in larger Driverlog, and in particular Blocksworld problems, we solve the problems with better quality than the domain independent planners. These results provide support that the rule-based representation is an appropriate method of harnessing the specialised heuristics.

We use TLPLAN as the comparison planner in these three domains. High quality control knowledge has been developed for TLPLAN and this presents the current upper bound on quality when using control knowledge. Optimal plans have been generated for some of the smaller instances of the problems. We can use this as an indication of how well our policy is solving the problems.

E.1.2 Setup

We have selected three domains: Driverlog, Blocksworld and Depots for this comparison. Driverlog and Blocksworld allow us to observe the use of solvers in problems with different structures and Depots requires reasoning over two types of structure.

We use the same policies that we described above under the same constraints. We have run the optimal planner h^{LM-cut} (Helmert and Domshlak, 2009) on the problems. This gives some indication of the quality of solutions for the smaller problems. As the problems increase in complexity the planner cannot compute a plan in reasonable time. We have used TLPLAN and the control knowledge used in the 2002 IPC as a comparison. The control knowledge used has been well engineered and presents a lower bound on performance.

E.1.3 Expectations

There is no expectation that the solutions we generate are optimal solutions. However, we have demonstrated that the solution quality compares well to domain independent planners. It seems therefore that we are providing relevant information in the enriched model.

The Driverlog and Depots TLPLAN control knowledge systems are quite sophisticated. As a consequence it is likely that we will perform less well in these domains.

E.1.4 Results

The quality results are presented in Figures E.1, E.2 and E.3. The results support our previous findings, demonstrating that our enriched model is proving support for expressing complete strategies.

For Blocksworld problems the results show that our plans are quite close to optimal on the five problems that the optimal planner can solve. The quality results are identical for our solution and TLPLAN.

The results in Driverlog demonstrate that both planners are generating suboptimal plans. However, neither generate bad plans on the runs that the optimal planner can solve the problems. In most of the problems the others planners are quite close, with TLPLAN usually having the better plan when they are not the same. However, the results in this domain are promising as they demonstrate that we are able to express a complete strategy using a limited rule representation.

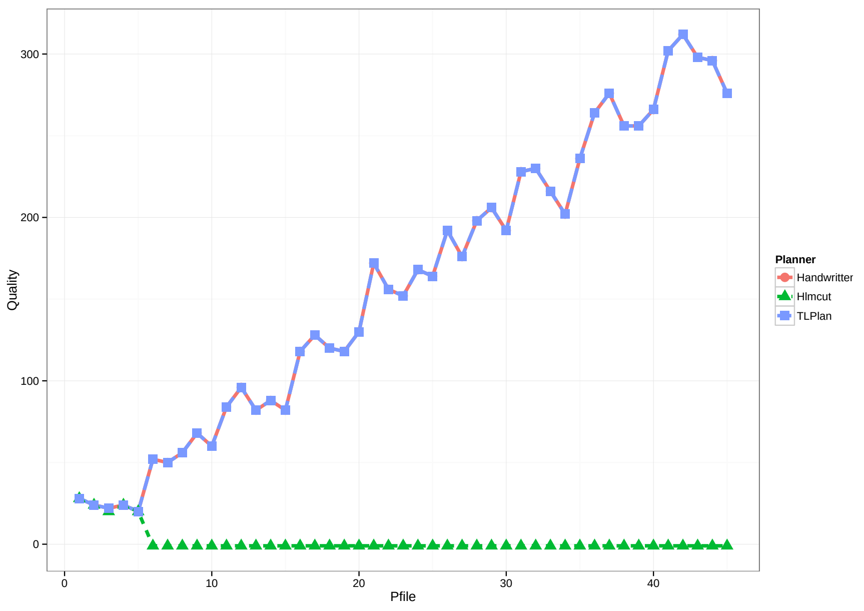


Figure E.1: Quality results for the optimal planner on Blocksworld problems

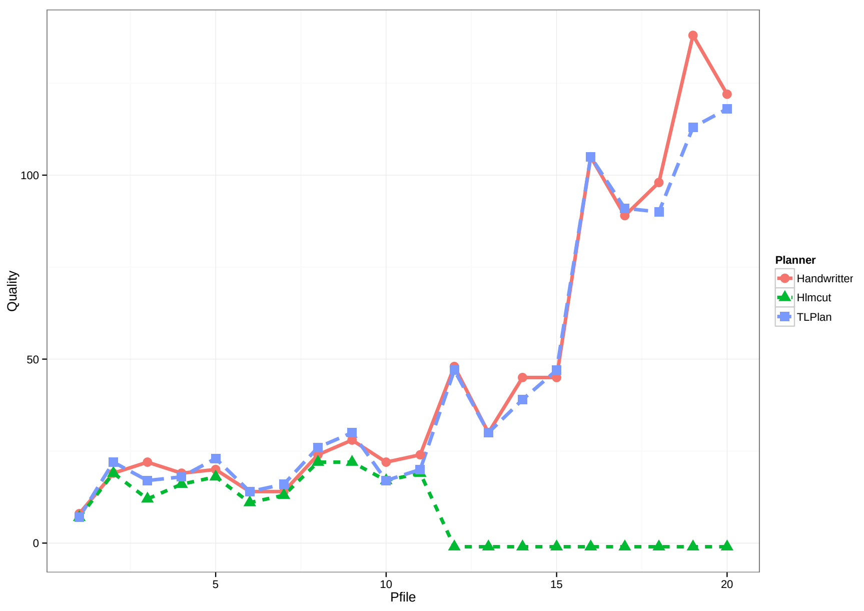


Figure E.2: Quality results for the optimal planner on Driverlog problems

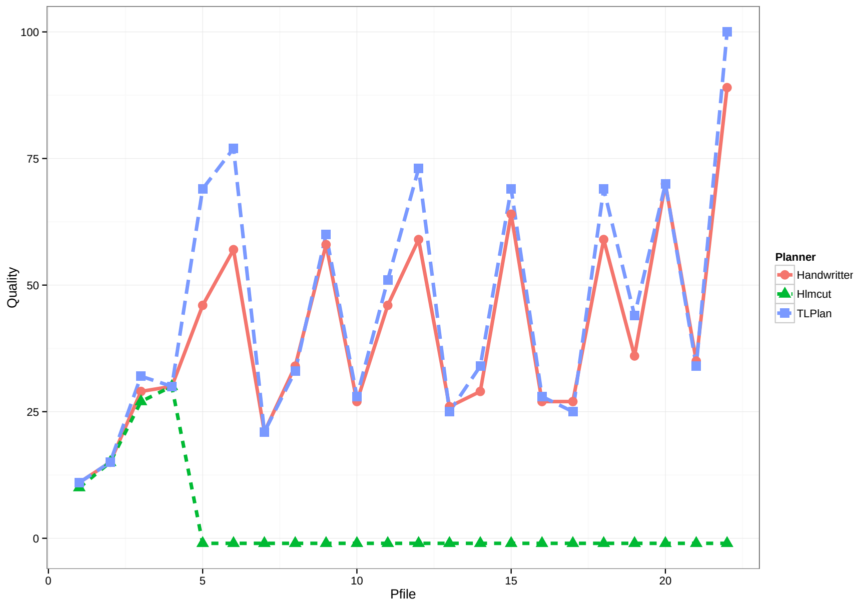


Figure E.3: Quality results for the optimal planner on Depots problems

In Depots our strategy produces shorter plans. This result is unexpected; however, we are also using a handwritten strategy and solvers in these experiments.

We have demonstrated that we can produce quality solutions that are similar in quality to solutions generated by TLPLAN in three domains. This supports the claim that our architecture supports expressing effective control knowledge in a limited language.

E.2 Combining information

In HybridSTAN (Fox and Long, 2001), the use of multiple solvers at the same time would involve a specialised decomposition for the specific combination. An interesting aspect of this work is that the solvers are used to provide information and the planner can choose the information it finds useful. In this subsection, we reflect on the use of multiple information sources in the strategies used in Chapter 6.

E.2.1 Transportation and resource management

The strategy that we used in Driverlog combined a strategy for traversal with a resource management strategy. The results demonstrate that the heuristic for selecting the next location for the truck interacted well with the resource management solver.

We note that the interaction might have been improved if we had made the open transportation tasks more accessible. For example, if we had modelled an (`open-task` *?obj ?truck ?location*) predicate that held of objects that were to be picked up or packages that needed dropped off. This would have provided more information that could have been exploited in the traversal solver.

We observed that a potential problem with our hub and cluster might have been that the solvers had little communication. For example, the allocation of packages was not made in the context of the computed clusters. This suggests that when using two sources of information to tackle a single aspect of the problem, for example, package transportation, then it is important that the information sources provide a consistent view of the world.

E.2.2 Transportation and building

The Depots domain combines a transportation problem with a distributed structure building problem. The transportation aspect of the domain is trivial as the graph is a clique. As a consequence there is no need of a transportation solver. However, if the domain did support an underlying graph structure then a transportation solver could be used to allocate trucks to packages.

However, there are several limitations. Our current transportation solver recognises a package's location if it is connected to a truck graph location by a static structure; for example a crane. In Depots, this does not pose a problem because the package location is made explicit in the model. However, in an alternative encoding this might not be the case. In addition, there are implicit constraints on the goal location of packages. This means that the resource allocation would not have access to the goal location of the packages. This would make its allocations weaker.

In conclusion, our solvers are sufficient to solve problems that link transportation and building structure problems. Our architecture allows the strategies to use the parts of the solver's information that can be modelled for the domain. This is an advantage in the use of solvers as information sources. However, we might not be able to utilise the full strength of the specialised solvers without some extensions. This is not surprising, if the specialised solution is not written to cater for a problem then its information will be less useful.

E.3 TLPlan rules

A main focus of this project has looked at reducing the gap between approaches that learn control knowledge and handwritten methods. The handwritten control knowledge used with TLPLAN is captured in a rich representation language, and encodes sophisticated control strategies. We use a limited rule representation, which is compatible with rule learning technologies (Levine and Humphreys, 2003). It is interesting to compare the sophistication of the knowledge captured in these limited representations with those captured in the rich representations used with TLPLAN. To this end, we make a direct comparison between these control systems.

In this section we look more directly at individual control knowledge systems, with the aim of evaluating the strengths and weaknesses of the architecture that we have developed. We use two case studies in order to provide a more concrete comparison between the approaches. In particular, we want to better understand to what extent we have bridged the gap, what has been achieved in other works and what remains as further work.

E.3.1 Driverlog

The TLPLAN system combines problem remodelling with both early decision making and opportunistic strategies to form an effective system of control. We first describe the control strategy and then compare it with our approach, focusing first on aspects that achieve an efficient system and then on aspects of the strategy.

The Driverlog knowledge systems

We present an overview of the system used in TLPLAN[1] and also the system that we used as the **Handwritten** configuration in the above results.

The Driverlog control system is a complete system for the competition problem generator. The knowledge is compiled into the domain operators. It has been designed to prevent redundancy from taking place. In particular, the system can be used with a depth first search and there is no need to check for repeated states.

We first present the idea behind the conditions that have been added to each action. We do this at a high level to provide an overview of the control system. In the following discussion we will describe some aspects in more detail. Please consult the control knowledge source for more detail.

[1]This system was written by Fahiem Bacchus

It is important to notice that the `walk` action has been replaced and the `drive-truck` action has been complimented by macro actions that move between each pair of connected nodes in the respective maps.

Load-truck ?package ?truck ?loc if *all* of the following:

- there is a driver in the truck
- the package has a goal
- it is not at its goal
- the goal can be reached from its current location

Unload-truck ?package ?truck ?loc if *all* of the following:

- the package has a goal at the current location

Drive-truck ?truck ?from ?to ?driver if *all* of the following:

- there are no packages to put-in or take out
- if *any* of the following:
  - there are packages to deliver
  - all packages are en route and either *?to* is the truck's goal or *?truck* has no goal and *?to* is *?driver*'s goal, or there is a better driver for driving *?truck* at *?to*
- the truck has not just moved

Macro-drive-truck ?truck ?from ?to ?driver is similar but there is not a constraint that there is an edge (*?from, ?to*) in the map graph; instead the constraint is that the nodes are connected. The closest (number of edges to satisfying node) of any satisfied formulae in the disjunction are selected first.

Board-truck ?driver ?truck ?loc if *all* of the following:

- useful to drive *?truck*
- *?driver* is the best driver for the truck

Disembark-truck ?driver ?truck ?loc if *any* of the following:

- if *all* of the following:
  - all packages are home
  - *?truck* is at home
  - *?driver* has a goal
  - *?driver* can walk home
- there is a better driver for the job

Macro-walk ?driver ?from ?to the closest of *any* of the following:

- if *all* of the following:
  - there is a truck at *?to*
  - it is useful to drive the truck
  - *?driver* is the best driver for the truck
- if *all* of the following:
  - all packages are home
  - *?driver* has a goal at *?to*

Driverlog policy strategy Repeatedly apply an instance of the first applicable of the following:

1. Drop-off package at goal
2. Pickup allocated misplaced package
3. Move to pickup allocated misplaced package
4. Move to drop-off package
5. Move truck home
6. Board truck if allocated driver and useful to drive-truck
7. Walk to board truck
8. Walk home
9. Disembark truck

Aspects leading to an efficient system

Although all parts of the knowledge system effect its efficiency there are specific techniques that have been used that can be particularly important. One of these is that the system provides a complete control system. This means that there is no need for search. This is key to the efficiency of this system. We now explore some of the other important aspects.

Generator functions The state has been enriched with a set of domain specific predicates that carefully control the binding of operators. These predicates are used to group together objects that can unify with a particular operator's parameters. This reduces the cost of unification as fewer false combinations are generated.

The binding of variables in TLPLAN is controlled by generator functions. These functions are used to propose the set of possible bindings for the variables. The described predicates can be used as generator functions. For example, in our work we use the positive goal and state predicates in our rules as generators to populate the variables. However, this can be costly as there will be many matchings between the different variable bindings that do not unify.

An alternative is to maintain a raised state that indicates the combinations of objects that could be used to unify with a specific operator's parameters. For example, in the Driverlog system a `can-do-load` *?package ?truck ?loc* predicate is maintained. This predicate holds when `at` *?truck ?loc* and `at` *?package ?loc* both hold. The bookkeeping for these predicates is carefully carried out in each action that can effect them using quantification.

There are several of these parameters; in fact there are only a handful of uses of the predicates in the described model. It would be interesting to investigate the impact of this raised state. Learning these predicates is a potential avenue of future work.

Compiled precondition The control knowledge is not expressed as control rules. Instead it is compiled into the operator schema as additional preconditions. The control is written so that any binding that satisfies the precondition is a useful action. It is important that this condition has clauses for all possible requirements for the operator. If the knowledge is expressed as control rules then states must be generated, matched with rules and pruned. By compiling the knowledge into the action conditions the states are never generated and there is no expensive matching process required.

Our rules use a similar form of compiled knowledge. However, there can be several rules for the same the operator. This means that we can split the different cases over several rules.

A feature of our representation is that we must order each use of an action. This has an impact on the strategies that can be encoded. For example, we must order the two rules move to pickup a package and move to drop off a package. If we prioritise picking up packages then we will not consider actions that move to drop off a package while there are packages that need picked up. In contrast the TLPLAN representation gathers the applicable actions and uses the search strategy as its selection process. This flexibility leads to the better quality of solutions observed in the results. In fact the selection of the next move action is controlled by a specific heuristic that we discuss below.

Level of language Abstracting graphs so that effective expressions can be used to reason about interesting graph nodes is common to both approaches. These ideas are key to each solution. In both approaches the `walk` and `drive` actions are abstracted and allow the intended destination node to be reasoned with. This abstraction acts to support rules that reason about moving to specific targets. This simplifies the rules and allows heuristics to be exploited (discussed below).

We can model these abstractions and use SbS. This means that we can compensate for the strict ordering on our rules. In TLPLAN these abstractions are modelled as macro actions. This is because the rule language allows disjunction, so there is more flexibility in determining the next best move. We have compared the step by step and normal macro application approaches in Chapter 6.

Our definition of ALMAs (Subsection 7.2) generalises the abstraction layer developed in TLPLAN. In this domain a similar abstraction is defined.

Opportunity

The strategy used in the TLPLAN rules uses specific heuristics as part of the `walk` and `drive-truck` macro expansions. The heuristic used is the nearest neighbour ordering that we discussed in Subsection 5.4.2. We have examined the impact that using this strategy has had on our strategy. However, its use in TLPLAN also supports the exploitation of *opportunity*.

An opportunity is an action that is taken when it is convenient. The definition of opportunity used in the system for Driverlog for TLPLAN can be inferred from the control knowledge. An opportunity is defined in terms of a graph, G = (V,E), a current distribution of graph traversers (we reuse the function, **position**) and a similar function for tasks ($\mathbf{taskLoc} : \mathbb{A} \mapsto (\mathbb{L} \cup \bot)$). This can be considered for drivers walking on the path graph and trucks driving on the link graph. An opportunity is a useful action that has a task location that is closer to a traverser than other useful actions.

The use of opportunity contrasts with making decisions up front. The TLPLAN system combines the two strategies; making decisions up front where possible, but leaving some choices to be made if a suitable situation arises. The choices that are made up front are used to filter the actions that are considered useful.

On inspection of the `move` and `walk` macro conditions presented above it is clear that there are several reasons why these actions can be used. Each of the possible bindings to these actions suggest a task that can be performed at the destination. The bindings for the destination parameter are explored by expanding iteratively around the traversers' current locations. If a destination satisfies the condition then the expansion is stopped.

In this system this strategy allows opportunity to determine the order of the package pickups and put downs. It is also used to determine whether drivers are swapped between trucks. In particular, the system makes the decisions that are relatively easy to make and uses opportunity to take care of the rest. The limited rule language (or limited problem model language) means that we must select a specific use of the move action. This acts as a selection process, for example selecting a convenient package to pickup, but does not allow the use of the move to be selected by opportunity. One solution would be to model an open transportation tasks predicate. This predicate would hold for (package, location) pairs if a package had to be picked up or dropped-off at the location. This suggests that there might be utility in including disjunctions in the rule language (this is posed as future work).

We experimented using a limited form of opportunity in Section 5.5. In that experiment we split the location map into clusters. This allows the impact of a strict priority caused by rule ordering in our representation to be weakened. We enforced a strict priority in a localised region. In this way we hoped to gain some of the benefits that opportunistic strategies can bring. Unfortunately, the results did not support this strategy.

E.3.2 Blocksworld

The TLPLAN system for Blocksworld described in Bacchus and Kabanza (2000) uses pruning rules to constrain the explored states. These rules use the `next` modal, which enforces constraints in the next state. We first describe the rules and then compare it with our approach, focusing first on how our strategies differ and then the aspects of the TLPLAN system that lead to a highly efficient system.

The blocksworld knowledge systems

We present an overview of the system described in Bacchus and Kabanza (2000) and also the system that we used as the **Handwritten** configuration in Chapter 6.

The operators are the same as the described domain. The control knowledge is encoded as pruning control rules. This contrasts with the approach that was used for Driverlog. The flexible nature of the TLPLAN language has allowed the control knowledge systems to be developed in the manner that suits the knowledge. However, as we have seen in this chapter, similar coverage can be achieved when using a limited language.

There are several defined predicates that are used in the definition of the rules. A block, x, is a good tower, if it is clear (at the top of a tower) and x and all blocks below x are well-placed. A block, x, is a bad tower if it is not a good tower. A block, x, has a good tower above if either x is clear or the blocks on x are consistent with their goals.

For all clear blocks, x:

- If x is a good tower then in the next state it has a good tower above.
- If x is a bad tower then in the next state there is nothing on it.
- If x is on the table and the goal position of x, y, is a bad tower then in the next state do not hold x.
- If x is a good tower and y is being held and has a goal of being on x then in the next state y is on x.

The first three items are equivalent to the system presented in Bacchus and Kabanza (2000). Steps were taken to improve the efficiency of the rule evaluation.

Blocksworld policy strategy Repeatedly apply an instance of the first applicable of the following:

1. `Stack` x on y if y is `well-placed` and x is `on` y in the goal
2. `Pickup` x if y is `well-placed` and x is `on` y in the goal

3. `Putdown` x if holding x

4. `Unstack` x from y if x is not `well-placed`

Comparison of strategy

The `well-placed` predicate is a powerful concept in this domain. The inclusion of it in our model provides the necessary vocabulary for capturing powerful control knowledge. The main difference in our approach is that the TLPLAN control knowledge prunes states while our control knowledge actively proposes actions.

The pruning rules tightly constrain the actions that can be applied. A good tower cannot be dismantled, a bad tower cannot be added to. A block is not picked off the table if its goal is not clear and if a block is held that has a good tower to be stacked on then it will be put down there. This means that the actions that are left for the planner will lead to a solution.

The strategies are very similar. In each case we allow any removal of blocks off of bad towers and dropping of blocks where their goal is not clear and well-placed. We favour picking up blocks from the table that can be put in on their goals. This means that it is less likely that a block will be picked from a bad tower and put on the table, because its goal is more likely to be prepared. Although this strategy is possible in TLPLAN search, the rules do not enforce it.

Developments that improve efficiency

The control knowledge description has several options for the exact system that is used. For example, the three rule strategy presented in Bacchus and Kabanza (2000) can be used. There are also several levels of optimisation that have been made. For example, it is reported that by factoring the three rule system, 30% can be saved in time and 40% in memory. In addition 10% can be saved in time by using an if then else language feature as a replacement for multiple evaluation of the good tower predicate. The system above is also rearranged to improve its efficiency.

The improvements in efficiency made in this system have come from rearrangements to equivalent formula. It would be interesting to consider rearranging rules that could be used to affect this sort of improvement in a general setting. There are similar issues with the creation of policy rule systems. Including time within the fitness function in the approach presented in Chapter 8, would be an interesting start. The ordering of the rules effects how often certain predicates are evaluated and described predicates can be used redundantly to shield some cases from the more expensive derived predicate evaluation.

An interesting consequence of adding the final rule is that cycles cannot happen. This saves some overhead in the search process. It is possible that this property could be detected automatically. A further step that was taken for the 2002 competition was to compile the knowledge into the preconditions, as was done in Driverlog[1].

E.3.3 Conclusion

In this section we have compared our knowledge systems with those of TLPLAN. In Blocksworld, the strategies used are largely similar. The TLPLAN control knowledge is optimised using alternative language features. In Driverlog we concluded that the rich language meant that the TLPLAN rules were capturing better control knowledge. However, our approach does exhibit comparable quality results, as demonstrated in Section E.1. It would be possible for us to extend the problem model with these features. We have similar aspects of arranging control formula so that it can be evaluated effectively; however, the nature of these are different in each approach.

We conclude that we have developed a framework that allows us to capture effective control strategies. The compromise in limiting the frameworks to facilitate learning is that we cannot express these strategies as efficiently as can be achieved using a richer representation.

E.4 Example output from ALMAgen

We have proposed an algorithm for automatically specialising the solver for a particular domain. We have generated training data for several domains that contain SIs and have applied our approach. In this subsection we present the generated bags and resulting vocabulary. We analyse the bags that are produced when no rules are used, to provide an indication of the importance of breaking the sequences into parts. We present an analysis of the approach for Structure Building problems in Appendix H.

Driverlog

The competition problem generator was used to generate 25 problems. These problems had a single truck, driver and package. Each problem had 15 truck locations and a

[1]Both version of the system were written by Fahiem Bacchus

number of joining path locations depending on the generated path map. Goals were generated for the truck, driver and package by the generator, which can include empty goals for objects (this is more likely for the trucks and drivers).

The targets in Driverlog problems include picking up and dropping off packages and moving to trucks and drivers to their goals.

No Sequencing For no sequencing we isolated each of the agent threads and gathered the relevant actions, as described in Section 7.1. The last SI action of each sequence is used as the target and gives its name to the vocabulary. The generated actions were `walk51_move` and `drive-truck50_move`. The bags included the enablers for getting a driver into the truck for driving and getting out of a truck for walking.

- (`drive-truck` *truck0 location1 location2 driver3*)
- (`board-truck` *driver3 truck0 location1*),(`drive-truck` *truck0 location1 location2 driver3*)
- (`walk` *driver0 location1 location2*)
- (`disembark-truck` *driver0 truck3 location1*),(`walk` *driver0 location1 location2*)

Rule based sequencing When using the transportation rules, the bag generation process identifies several bags, one for each of the target types identified. The name of the ALMA is an extension of the target action, with a generic number ID and move attached.

- (`load-truck50_move` *?truck ?loc-from ?loc-to*)
- (`drive-truck51_move` *?truck ?loc-from ?loc-to*)
- (`disembark-truck53_move` *?truck ?loc-from ?loc-to*)
- (`unload-truck54_move` *?truck ?loc-from ?loc-to*)
- (`walk52_move` *?driver ?loc-from ?loc-to*)
- (`board-truck55_move` *?driver ?loc-from ?loc-to*)

The targets for walking were a goal that was achieved by walking (locatedness of the driver) and when the driver got into the truck. Similarly for the driving actions, the truck was driven to a goal, the truck was driven to pickup and drop-off packages and the driver disembarked at their goal. The latter target is the result of the driver using a goal-less truck to move to its goal.

In each case, the associated bags contained a singleton macro. The bag for truck ALMAs was:

- (`drive-truck` *truck0 location1 location2 driver3*)

The bag for driver ALMAs was:

- (`walk` *driver0 location1 location2*)

These ALMAs model vocabulary that are equivalent to the graph abstraction solvers used in the experiments in Chapter 6, in the sense that the predicate holds for the same arguments and the action will return the same output given the same choice.

The generated ALMA is included in the solver listings file. The listing created for moving a truck to achieve the target of unloading a package is presented in Listing E.1. The "@" symbol is used to separate actions in a chunk and the "#" symbol separates chunks. This has proven a convenient representation when reading the output and when defining ALMAs by hand.

Listing E.1: Extract from solver listings output for generated ALMA for Driverlog domain

```
(:solverDescription SequenceChainSolver202
        :Graph (:module DynamicGraphModule163)
)
(:moduleDescription DynamicGraphModule163
        :MoveAction (:module MoveAction1)
        :Name (:description (unload-truck202))
        :Sequences (:description (drive-truck truck0 - truck location1 - location
            location2 - location driver3 - driver@#))
        :Significance (:description (true))
        :Strategy (:description (false))
)
```

Logistics

Logistics is a domain that requires the use of two maps. The graphs for each move action in the problem sets are cliques, although this is not necessary. We generated a collection of 5 problem using the competition generator, with one or two cities, with a truck in each city. Each city had a location and an airport and there were a small number of packages.

No sequencing The system generated two actions, `fly-airplane50_move` and `drive-truck51_move` that explained the behaviour of the aeroplanes and truck respectively. The associated bags for `fly-airplane50` and `drive-truck51` are as follows:

- (`fly-airplane` *airplane0* - `airplane` *airport1* - `airport` *airport2* - `airport`
- (`drive-truck` *truck0* - `truck` *location1* - `location` *airport2* - `airport` *city3* - `city`)
- (`drive-truck` *truck0* - `truck` *airport1* - `airport` *location2* - `location` *city3* - `city`)

Rule based sequencing

- (`unload-airplane52_move` *?airplane ?loc-from ?loc-to*)
- (`load-truck51_move` *?truck ?loc-from ?loc-to*)
- (`unload-truck50_move` *?truck ?loc-from ?loc-to*)

The associated bag for `unload-airplane52` is the same as for `fly-airplane50` above, and the bag for `unload-truck50` was the same as `drive-truck51`. However, there was a difference between the bags for loading and unloading. The bag for `load-truck51` is:

- (`drive-truck` *truck0* - `truck` *location1* - `location` *airport2* - `airport` *city3* - `city`)

In Logistics problem distributions the truck always starts at a location and in the small problems we used to generate the vocabulary there was only one location. If the package needed moved then it was picked up before the truck was moved. Otherwise the only packages the truck picks up get delivered to the airport. Of course packages might need to be dropped off at either the airport or the location. The difference between the bags demonstrates an advantage of using training data to parameterise the solvers. Implicit control knowledge has also been observed in inferred domain models (Cresswell and Gregory, 2011).

The generated language points out a limitation in our current approach. The problem models from the Logistics domain do not require enhancements to establish directed connectivity. The model is sufficient for an RBP to effectively control search. An interesting line of future work is to extend the scope of the target significance detection, as a general framework for invoking specialised language.

Goldminer

The 30 bootstrap problems from the Learning track were used to generate the macro bags. These problems have a single robot, bomb and laser and small grids (half 3 × 3 and the other half 4 × 4). When generating bags without the sequence splitting rules, the system generates the **allMovesBag**, as defined in Section 7.3, with one extra pair of macros:

- (`pickup` *robot0 laser4 loc1*)(`fire-laser` *robot0 laser4 loc1 loc2*); (`putdown` *robot0 laser4 loc*); (`pickup` *robot0 bomb3 loc1*); (`move` *robot0 loc1 loc2*)

This macro is learned from small examples where only a single laser shot is required.

Rule based sequencing

- (`fire-laser-0-151_move` *?robot ?x ?y*)
- (`fire-laser-1-050_move` *?robot ?x ?y*)
- (`pickup-gold53_move` *?robot ?x ?y*)
- (`pickup52_move` *?robot ?x ?y*)

The bag for picking up the laser or bomb, was a singleton macro. This action is used on two occasions, either when a path has been made to one step from the gold, or at the beginning on the way to pick up the laser.

- (`move` *robot0 loc1 loc2*)

The listings for firing hard and soft rock define equivalent bags. These macros include the actions necessary to fire at hard or soft rock if it is there and move.

- (`fire-laser-1-0` *robot0 laser3 loc1 loc2*); (`move` *robot0 loc1 loc2*)
- (`move` *robot0 loc1 loc2*
- (`fire-laser-0-1` *robot0 laser3 loc1 loc2*); (`move` *robot0 loc1 loc2*)

The bag for picking up gold contained two macros, one that gets rid of the final (gold covering) rock.

- (`detonate-bomb-1` *robot0 bomb3 loc1 loc2*); (`move` *robot0 loc1 loc2*)
- (`move` *robot0 loc1 loc2*)

These macro bags provide exactly the actions that are required so that an RBP can control the robot to the gold. The final bag includes the detonate action, which is appropriate as it is used in order to reach the gold and therefore access a target. However, in the plans, the detonate action is only ever used on the last rock that sits on the gold. This is a result of over-generalising the training data and is a result of our bag exploration approach. In practice we have detected little impact from this and deem its exploration outside of the scope of this project.

Rovers

A collection of 22 problems were generated using the generator from the third IPC. Each problem has one `camera`, five `waypoints`, one `rover-store`, four `modes`, two `objectives` and one `rover`. Without rules, the system generates a single solver called `navigate50` that is parameterised by the singleton macro: (`navigate` *rover0 waypoint1 waypoint2*). In the rule based version, the same bag parameterises five solvers that determine each of the targets identified in the Rovers domain. The added action names are presented.

- (`communicate_image_data53_move` *?rover ?y ?z*)
- (`sample_soil52_move` *?rover ?y ?z*)
- (`communicate_rock_data50_move` *?rover ?y ?z*)
- (`sample_rock54_move` *?rover ?y ?z*)
- (`communicate_soil_data51_move` *?rover ?y ?z*)

APPENDIX F

INDIVIDUAL PRESENTATION OF THE RESULTS

In this appendix we present the graphs from the analyses in Chapters 6 and 9.

F.1 Handwritten RBPs results

Graphs from the evaluation in Chapter 6, of handwritten RBPs for a selection of planning domains. The approaches plotted in each graph are a handwritten RBP (Handwritten), and the planners LAMA and FF.

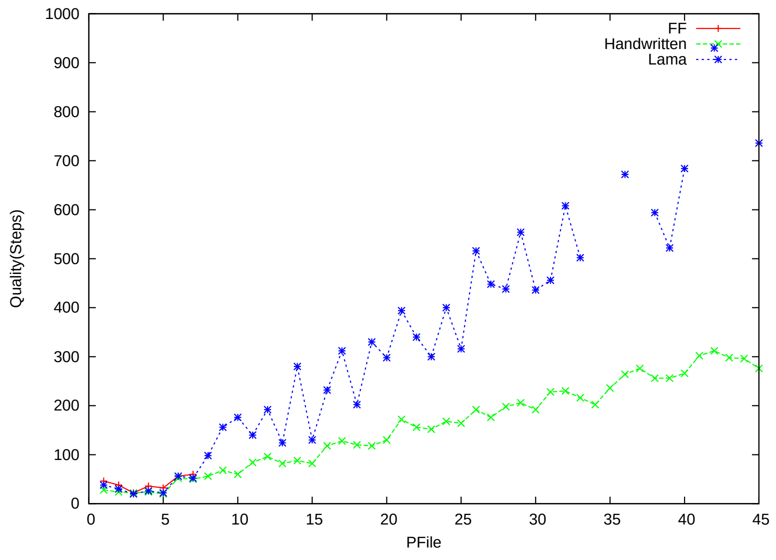


Figure F.1: Quality results for a handwritten policy on Blocksworld problems

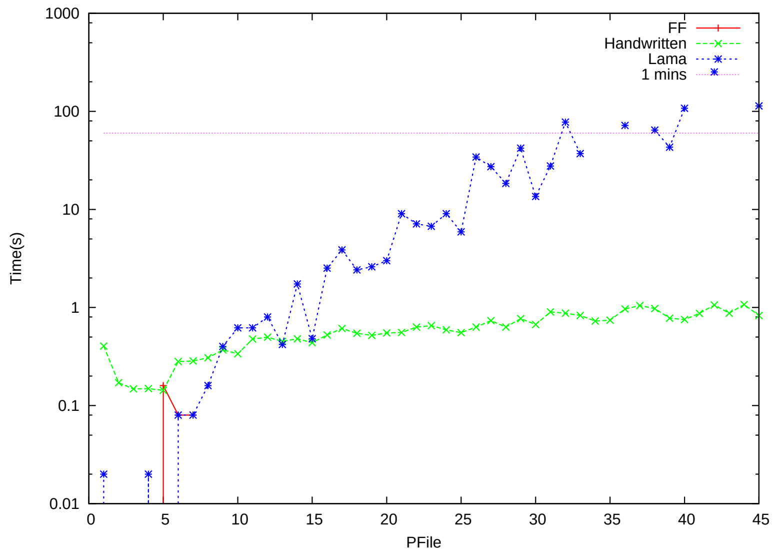


Figure F.2: Time results for a handwritten policy on Blocksworld problems

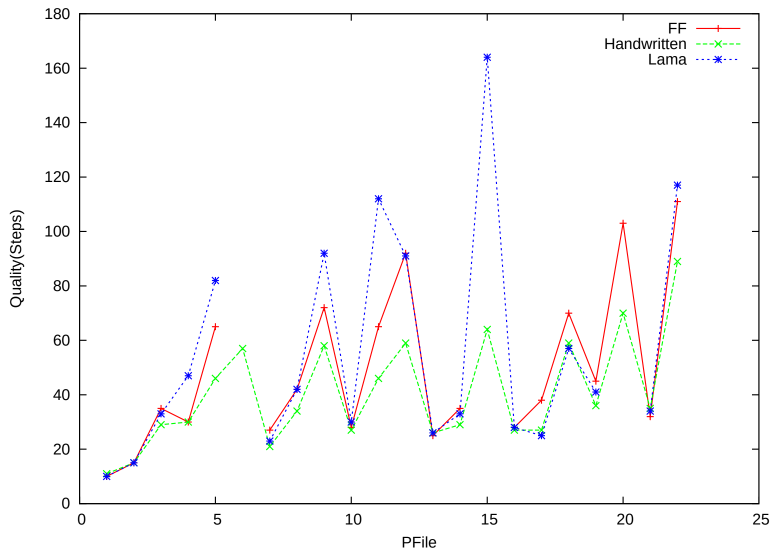


Figure F.3: Quality results for a handwritten policy on Depots problems

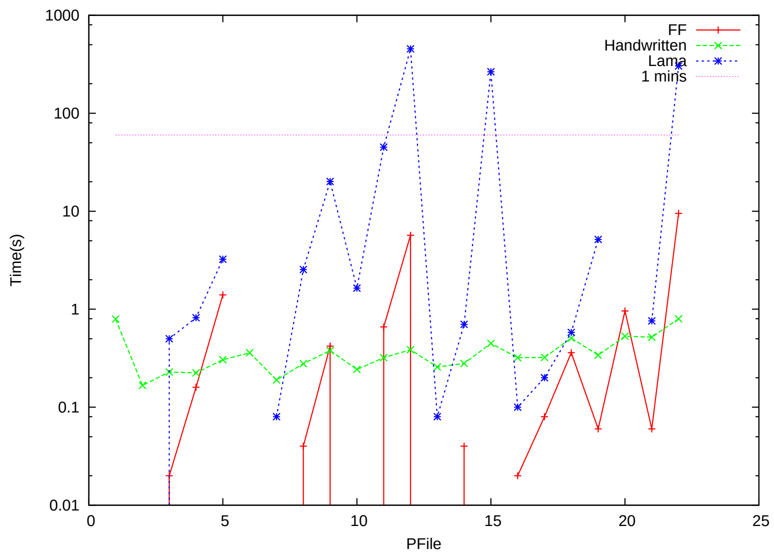


Figure F.4: Time results for a handwritten policy on Depots problems

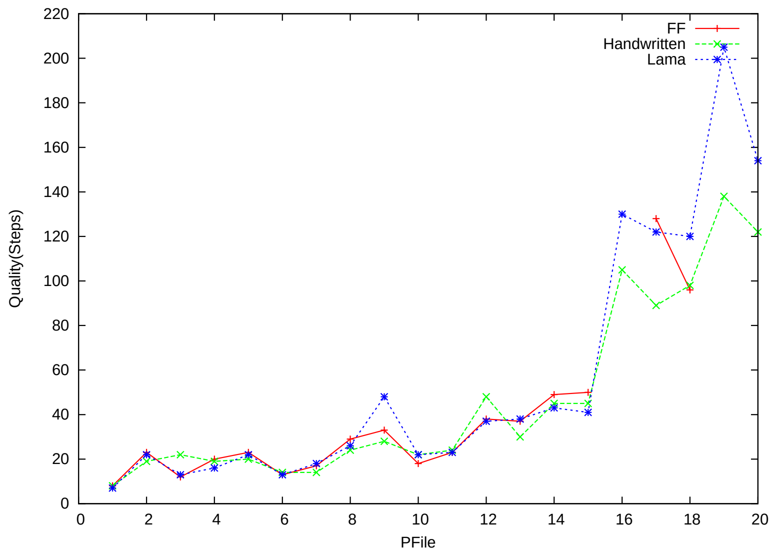


Figure F.5: Quality results for a handwritten policy on Driverlog problems

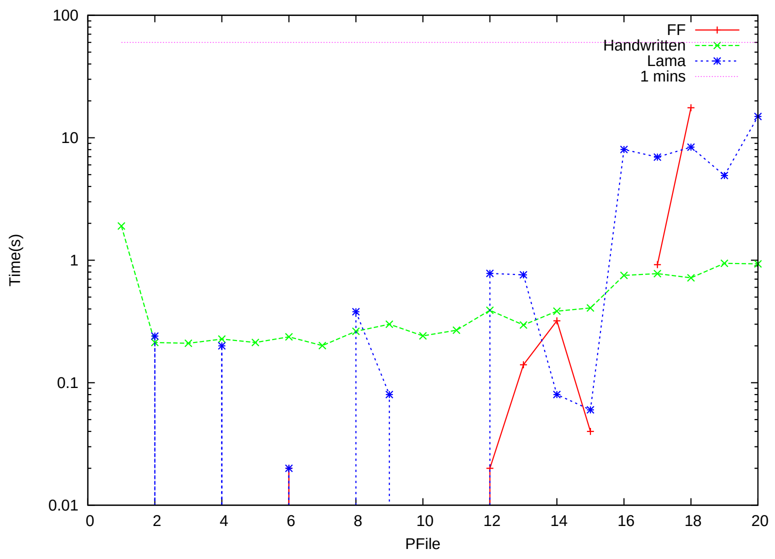


Figure F.6: Time results for a handwritten policy on Driverlog problems

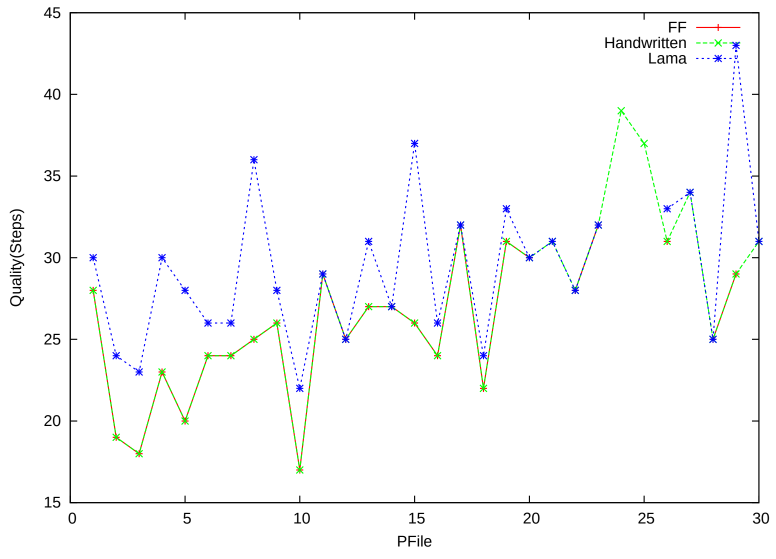


Figure F.7: Quality results for a handwritten policy on Goldminer problems

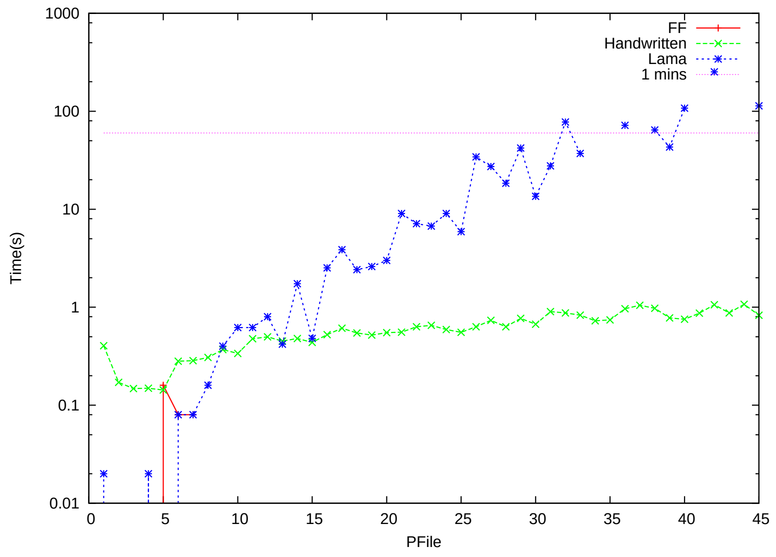


Figure F.8: Time results for a handwritten policy on Goldminer problems

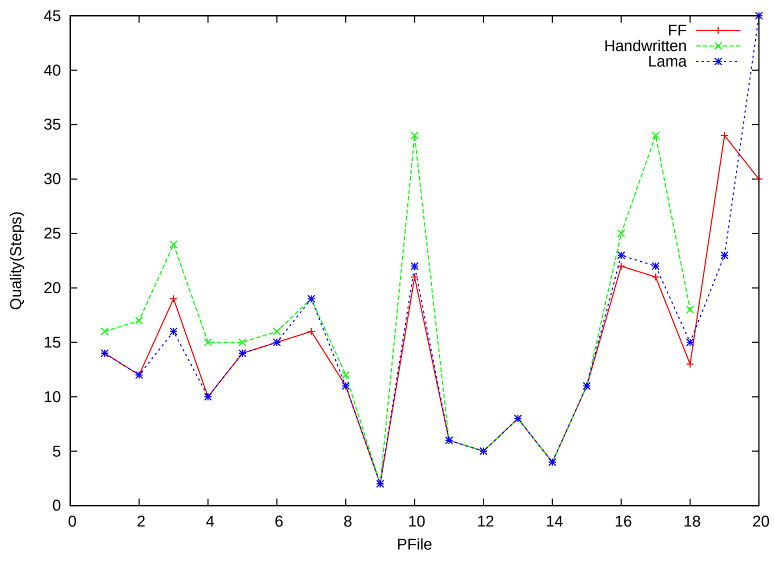


Figure F.9: Quality results for a handwritten policy on Grid problems

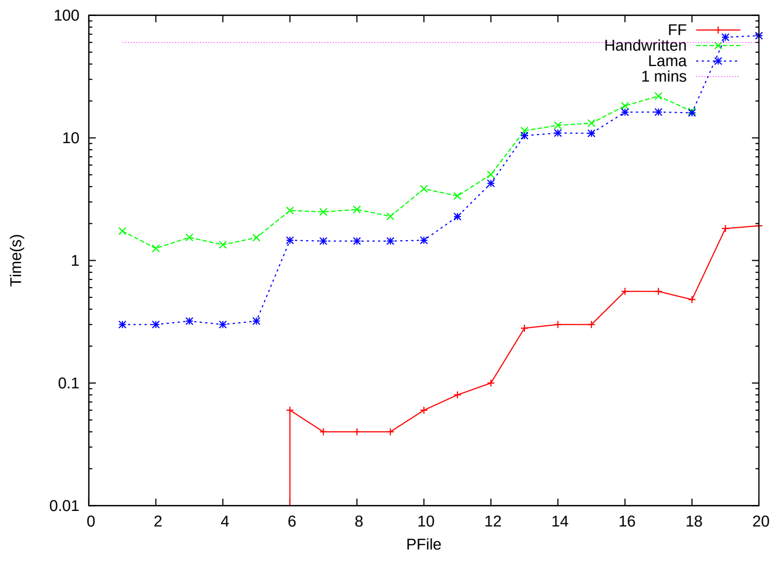


Figure F.10: Time results for a handwritten policy on Grid problems

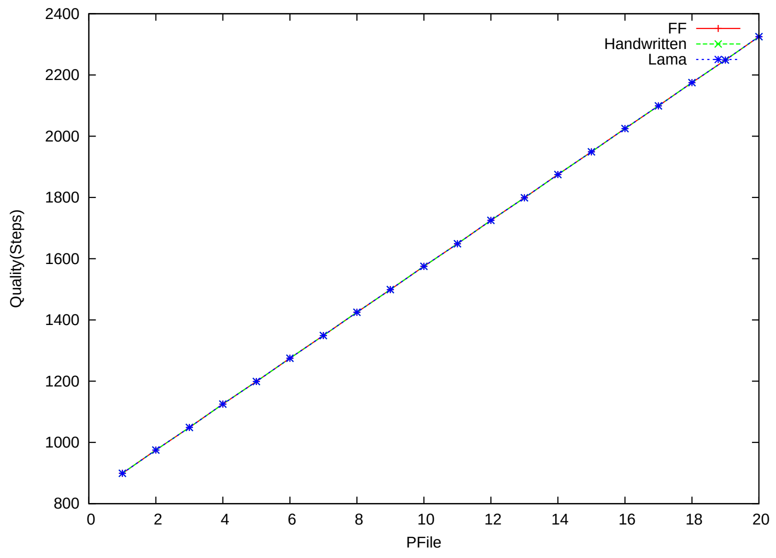


Figure F.11: Quality results for a handwritten policy on Gripper problems

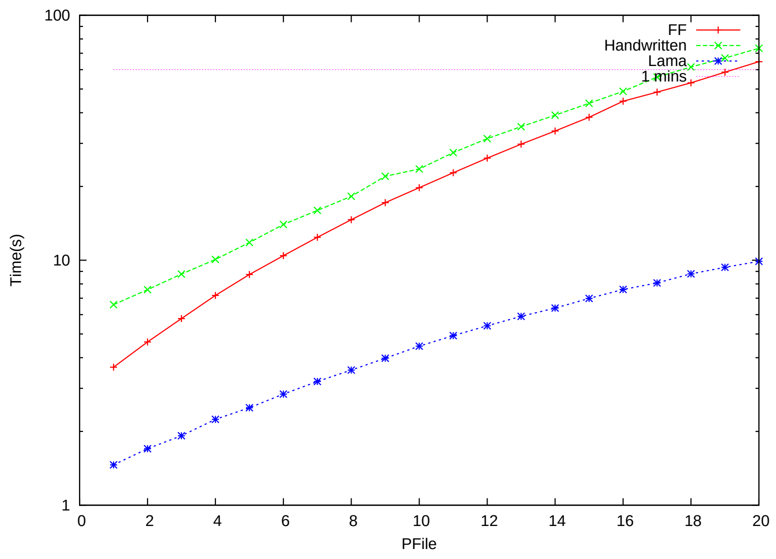


Figure F.12: Time results for a handwritten policy on Gripper problems

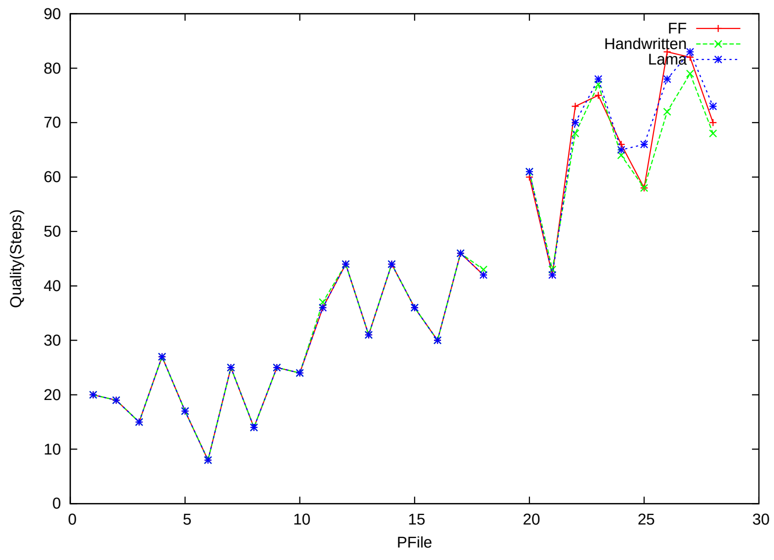


Figure F.13: Quality results for a handwritten policy on Logistics problems

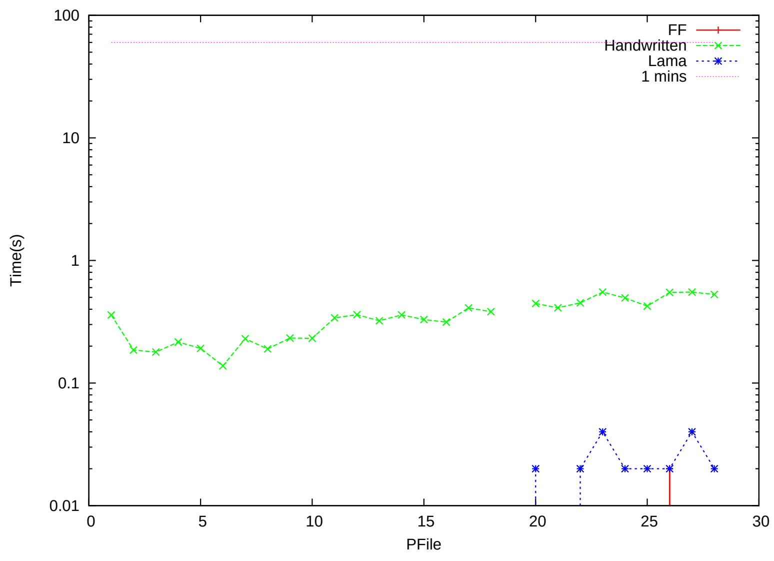


Figure F.14: Time results for a handwritten policy on Logistics problems

F.2 Heuristic guidance

Graphs from experiments in Subsection 6.3.2. Each plot is for no heuristics (Basic), solver heuristics (+Lh), global heuristic (+Gh), and both heuristics (+LGh).

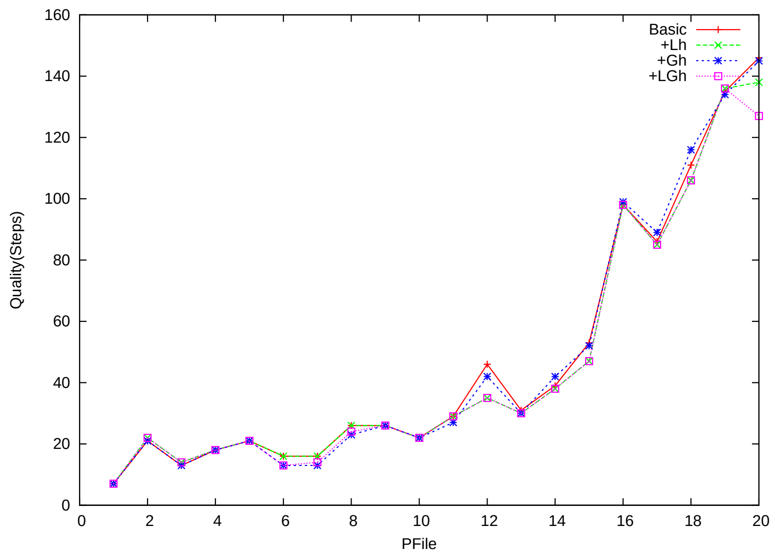


Figure F.15: Quality results for solver (L) and global (G) heuristics on Driverlog problems

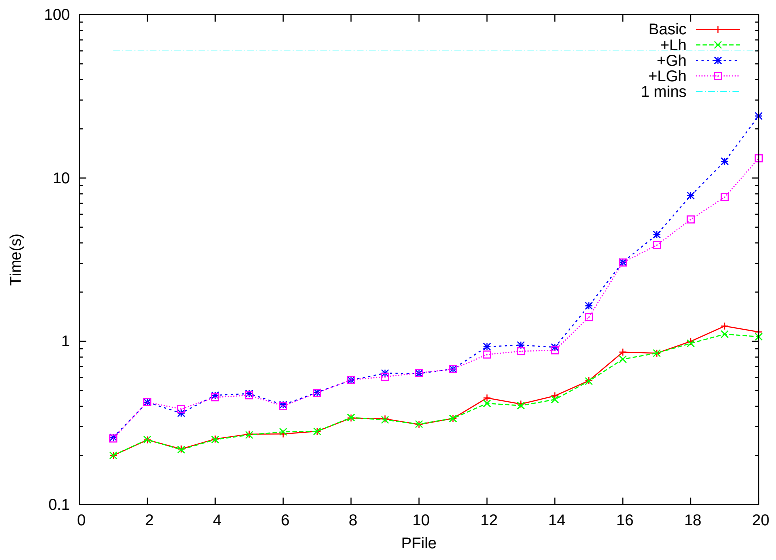


Figure F.16: Time results for solver (L) and global (G) heuristics on Driverlog problems

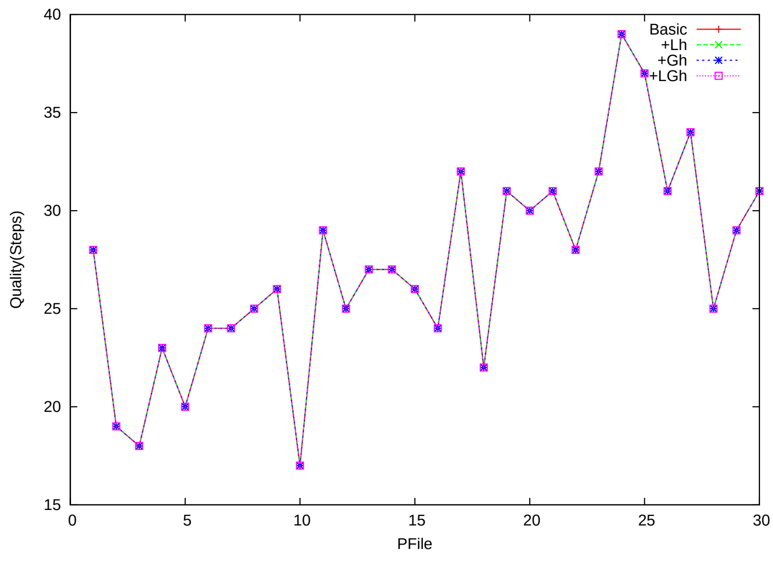


Figure F.17: Quality results for solver (L) and global (G) heuristics on Goldminer problems

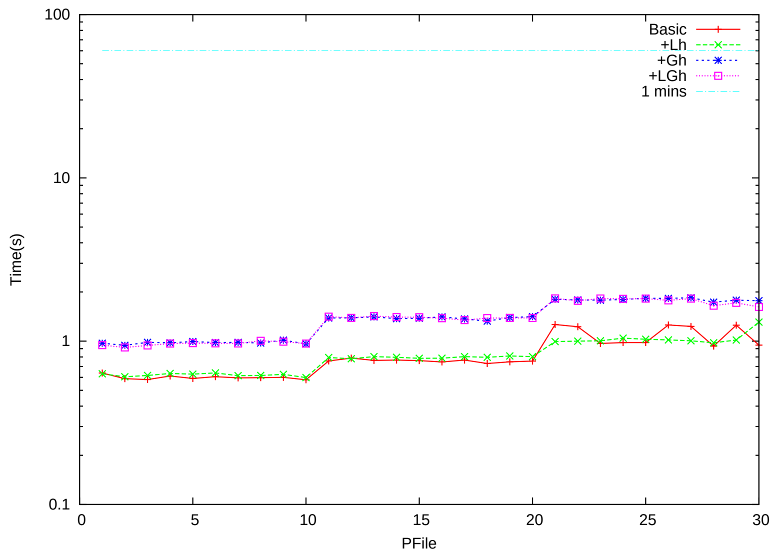


Figure F.18: Time results for for solver (L) and global (G) heuristics on Goldminer problems

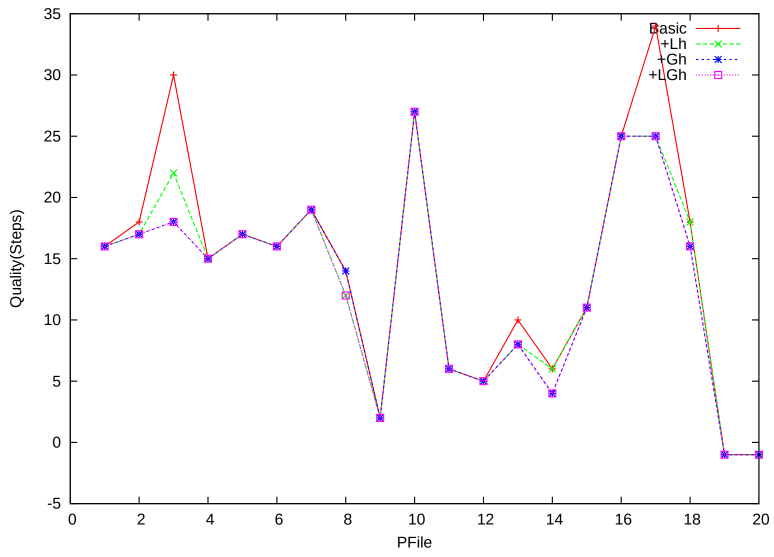


Figure F.19: Quality results for solver (L) and global (G) heuristics on Grid problems

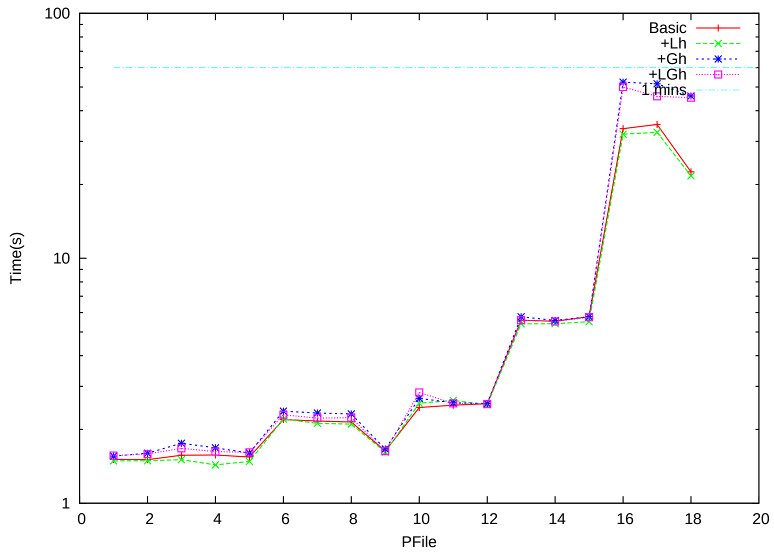


Figure F.20: Time results for solver (L) and global (G) heuristics on Grid problems

F.3 Step by step application

Graphs from experiments in Section 6.4. Each plot is for step by step application (SbS) and macro application (Macro) approaches.

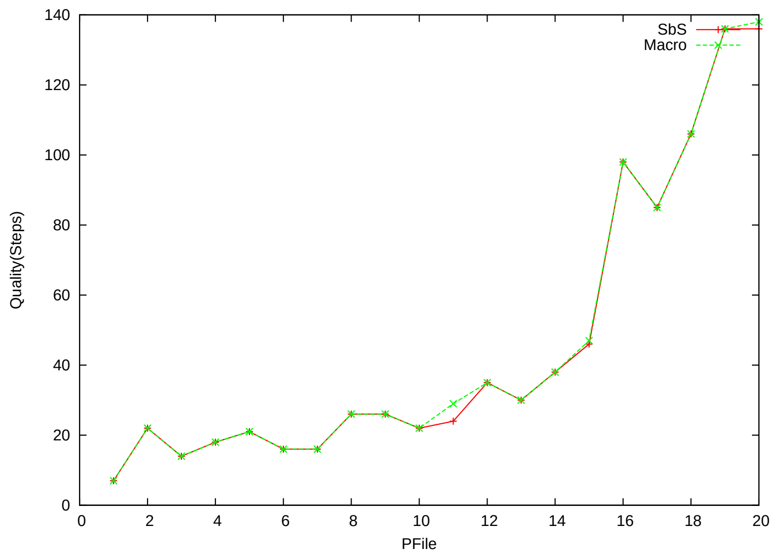


Figure F.21: Quality results for the step by step and macro application approaches on Driverlog problems

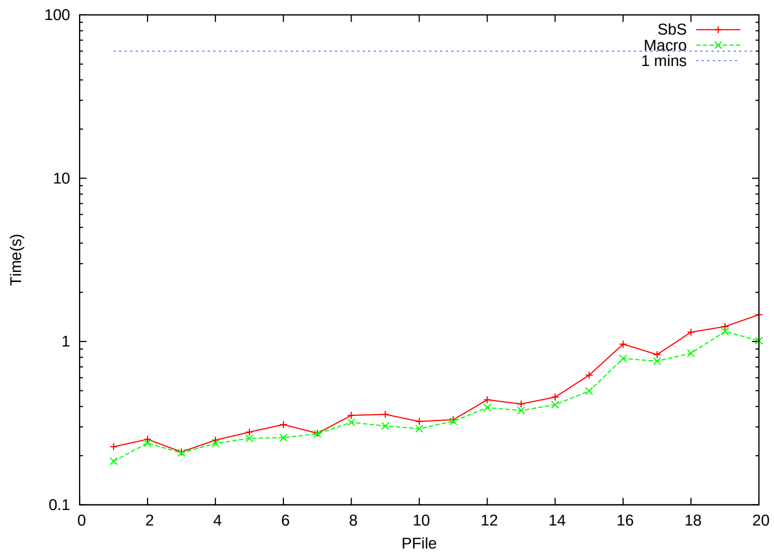


Figure F.22: Time results for the step by step and macro application approaches on Driverlog problems

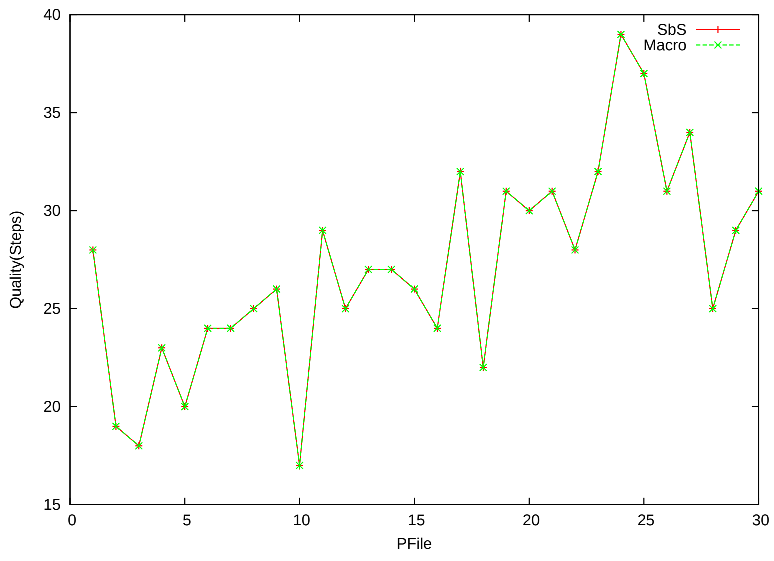


Figure F.23: Quality results for the step by step and macro application approaches on Goldminer problems

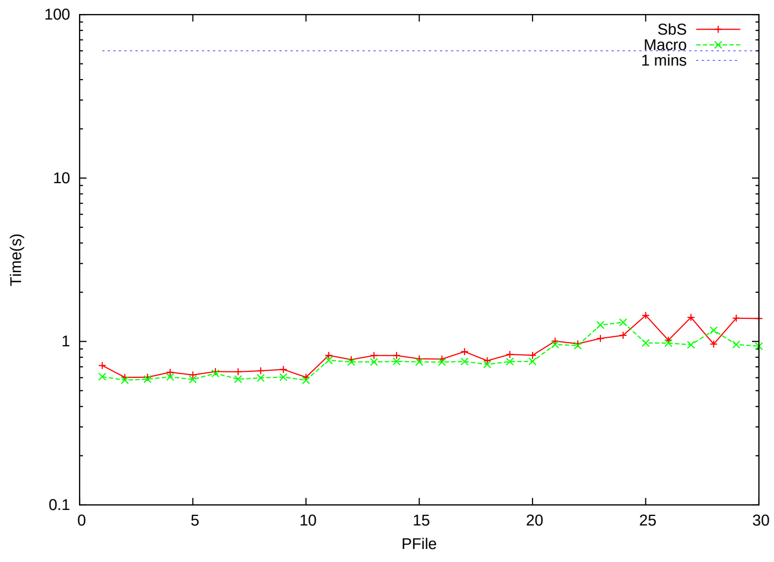


Figure F.24: Time results for the step by step and macro application approaches on Goldminer problems

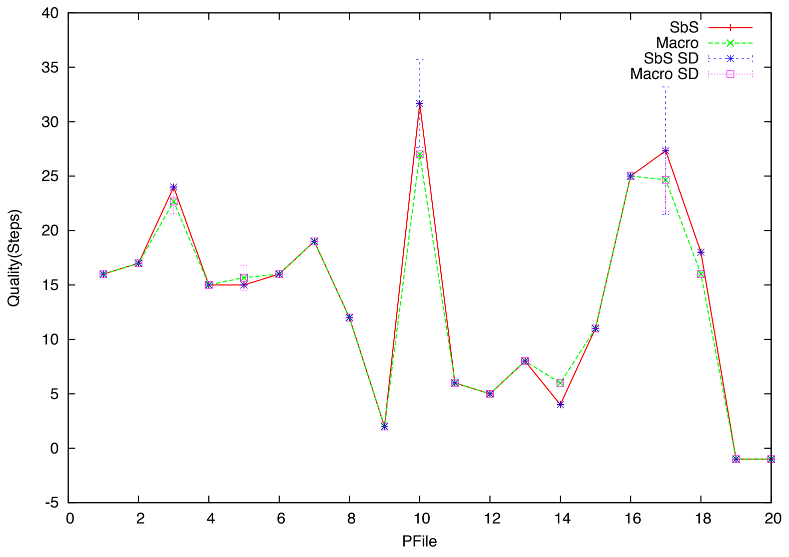


Figure F.25: Quality results for the step by step and macro application approaches on Grid problems. The plots show the means and standard deviations for each problem over 3 runs.

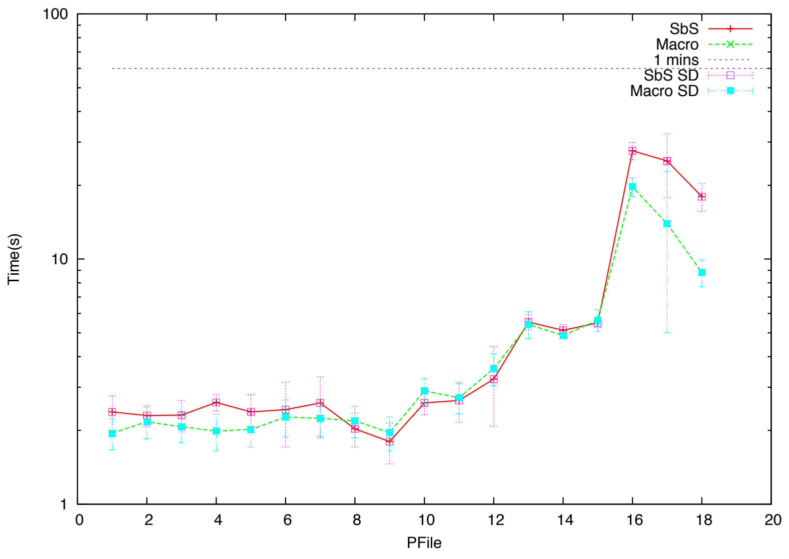


Figure F.26: Time results for the step by step and macro application approaches on Grid problems. The plots show the means and standard deviations for each problem over 3 runs.

F.4 Plots for planning with learned control knowledge

In this section we present the results of planning with learned control knowledge. The runs were restricted to 6Gb of memory and 10 minutes of run time. We present the quality and time plots for each of the domains we have tested. If the control knowledge was learned starting from a randomly initialised population then the plot is labelled `Learned`. In cases were the learning was seeded, the plot is labelled `Seeded`. These plots also include the results for control knowledge that exploits the ALMA solver.

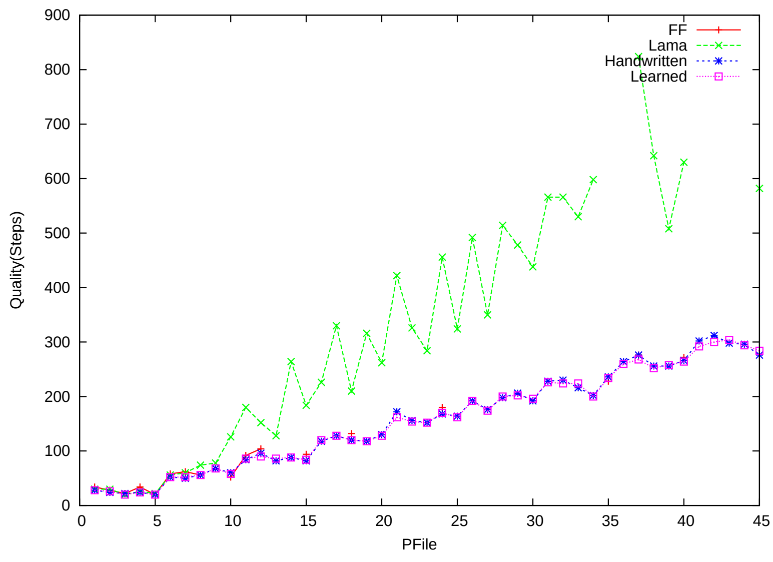


Figure F.27: Quality results for a learned policy on Blocksworld problems

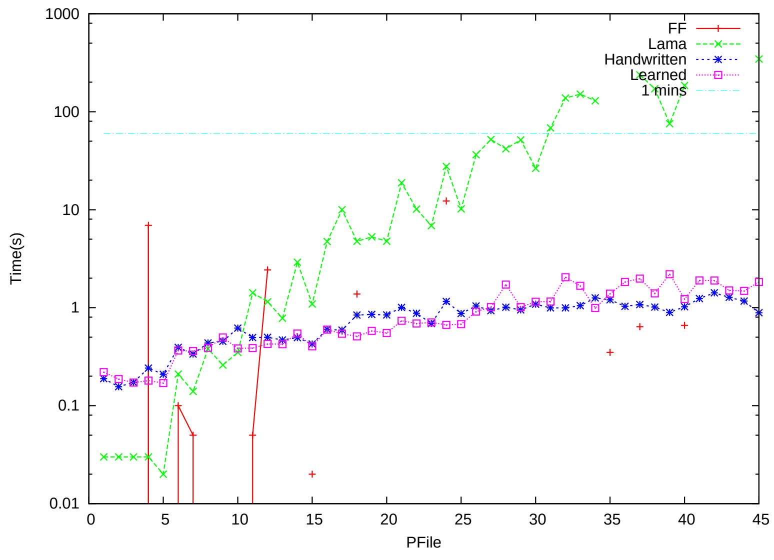


Figure F.28: Time results for a learned policy on Blocksworld problems

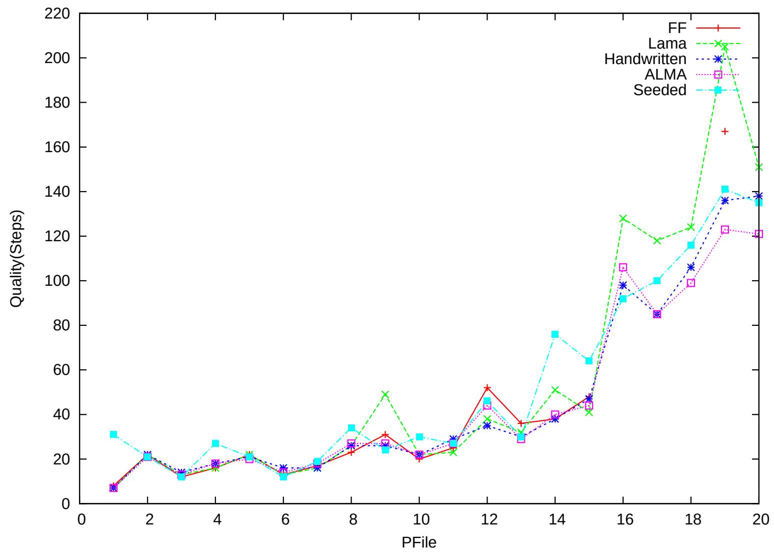


Figure F.29: Quality results for a learned-from-seeds policy and an ALMA based policy on Driverlog problems

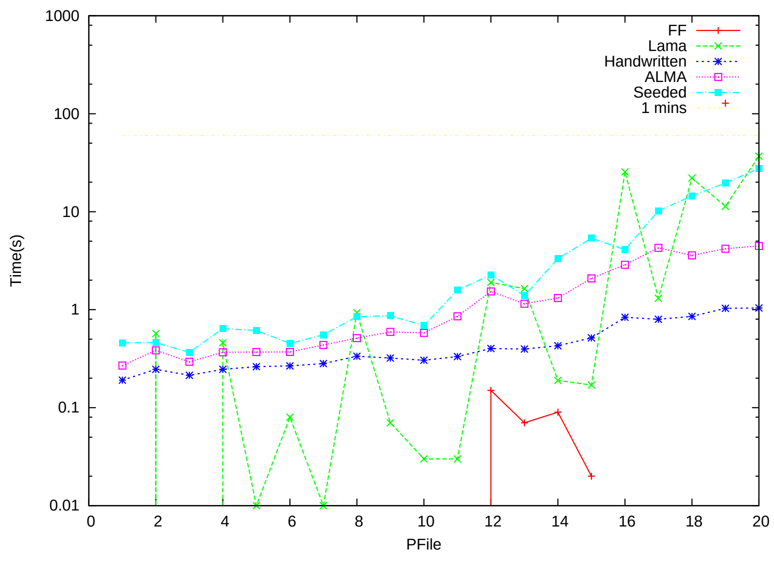


Figure F.30: Time results for a learned-from-seeds policy and an ALMA based policy on Driverlog problems

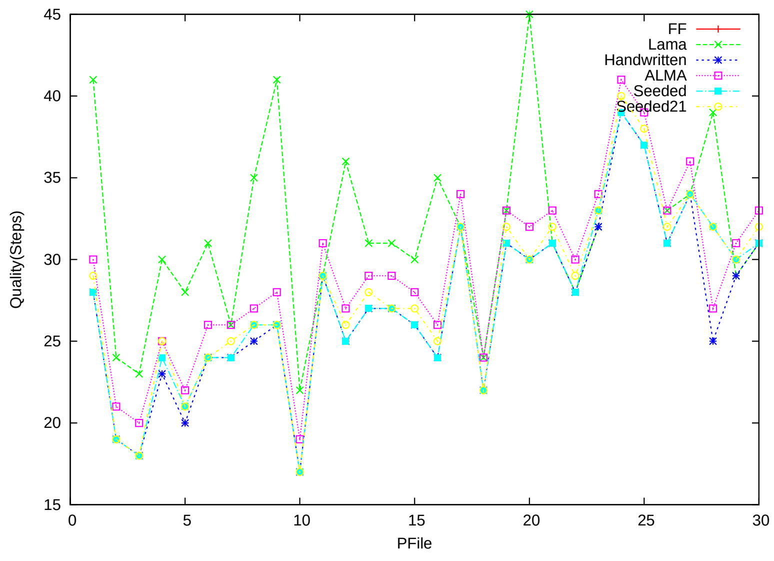


Figure F.31: Quality results for a learned-from-seeds policy and an ALMA based policy on Goldminer problems

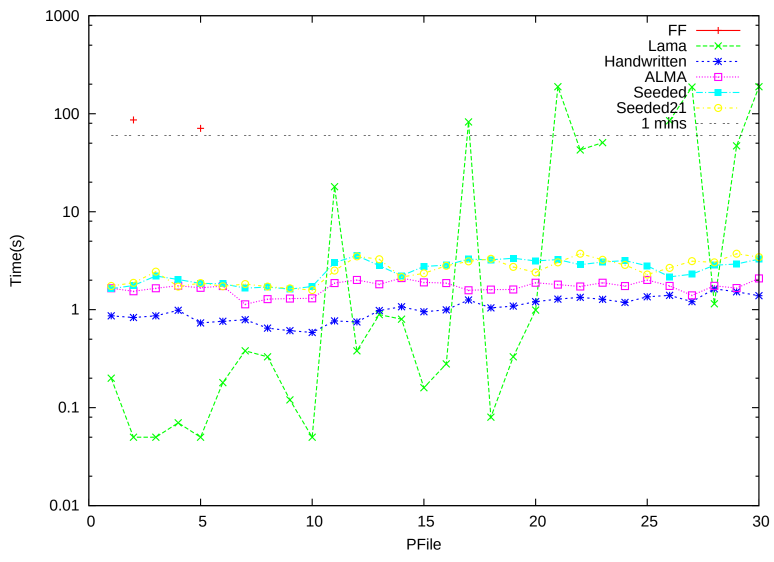


Figure F.32: Time results for a learned-from-seeds policy and an ALMA based policy on Goldminer problems

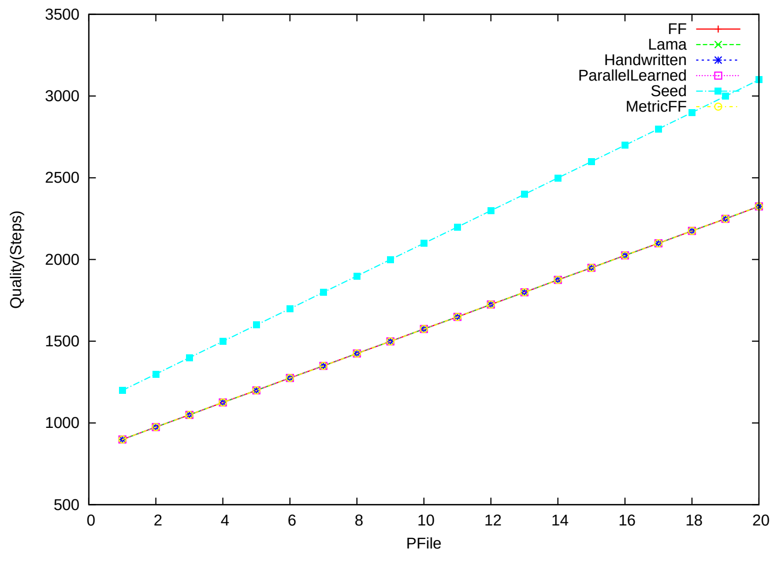


Figure F.33: Quality results for a learned policy and a generated seed on Gripper problems

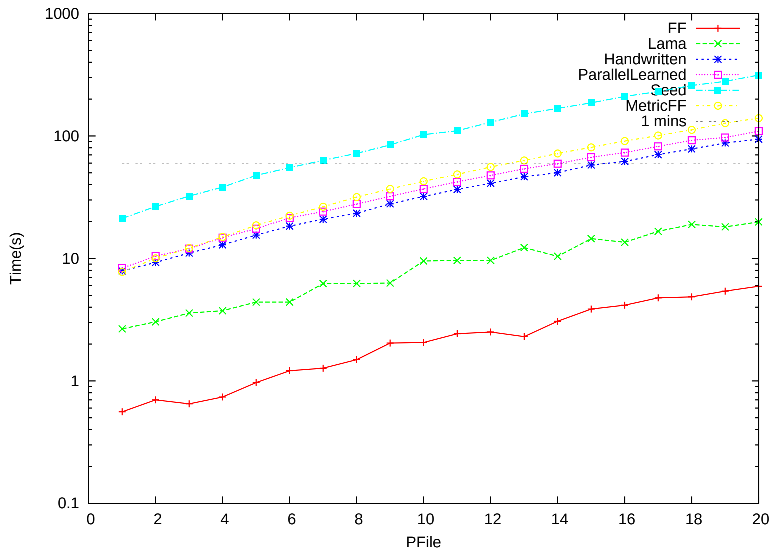


Figure F.34: Time results for a learned policy and a generated seed on Gripper problems

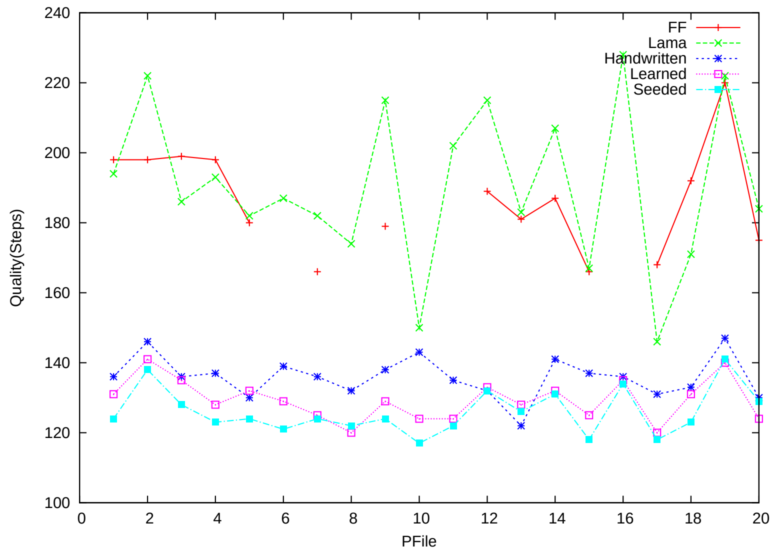


Figure F.35: Quality results for learned and seeded policies on Structured Briefcase problems

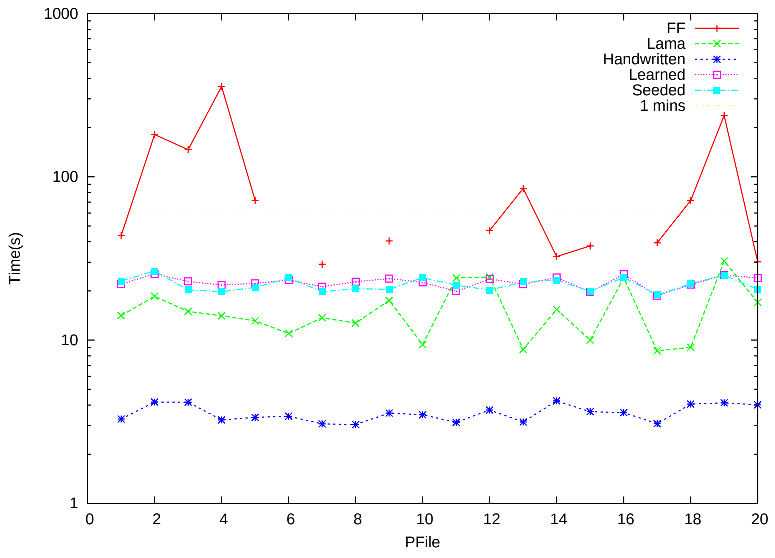


Figure F.36: Time results for learned and seeded policies on Structured Briefcase problems

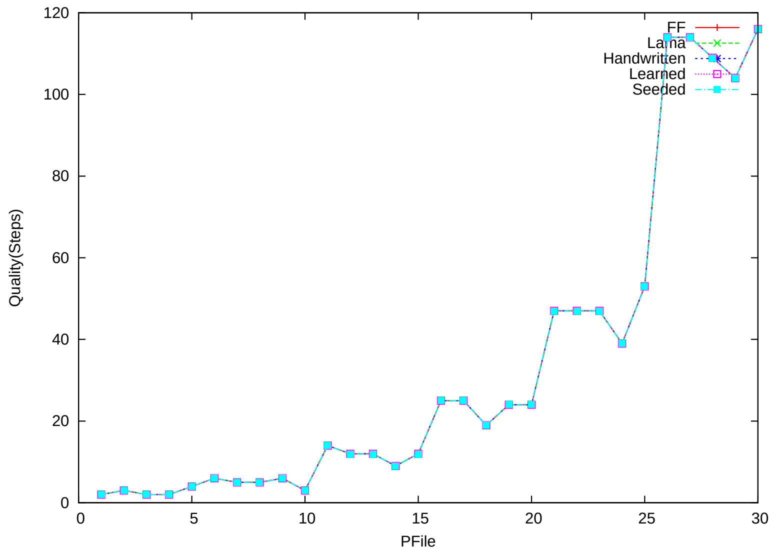


Figure F.37: Quality results for learned and seeded policies on Traverser problems

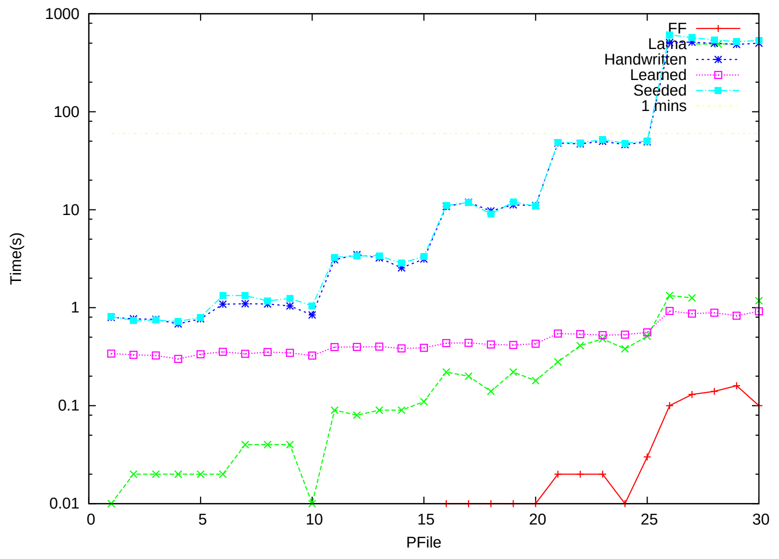


Figure F.38: Time results for learned and seeded policies on Traverser problems

APPENDIX G

ADDITIONAL DEVELOPMENTS FOR ARBITRARY LENGTH MACRO ACTIONS

In this appendix we explore two aspects of ALMAs. We have examined ALMAs as a single layer abstraction, whereas in some cases, the causal structure between sub-goals can be treated more efficiently by exploiting deeper hierarchies. In the first section we present an extension, which supports a further abstraction layer.

In Chapter 7, we observed that the expansion of the search space, defined by an ALMA, involved redundancy and we tackled this by exploiting a projection into a reduced space. In the second section, we consider the problem of detecting when this projection is appropriate for a given ALMA and domain.

G.1 Heuristic guided target selection

The ALMA provides a vocabulary for reasoning about moving to target nodes. However, we can observe in Grid, Sokoban and (n^2-1)-Puzzle domains that targets can exist in a hierarchy; for example, picking up a package; dropping off a key; or moving to the traverser's goal. The next layer of targets might contribute incrementally to reaching the mover's overall target. For example, moving to open a chain of blocked nodes that lead to the mover's target. We could simulate all the sequences in a similar manner to the bags in the ALMA solver. However, this will be a large space that is likely to grow quickly with the size of the problem. We have already tackled this problem in designing our Grid solver for the investigation in Chapter 6. We exploited a heuristic to direct the selection of the next target. Specifically, we use the target SI as the goal and compute the relaxed plan that achieves that goal. We identify the target nodes from the relaxed plan and use these as a back bone for making ALMA expansions.

In this subsection we introduce the target layer. We continue by outlining the approach that is used to generate the next action for a set of possible raised SIs. We conclude by presenting a comparison with the handwritten policy and LAMA.

G.1.1 A template layer

We extend the ALMA expansion by framing its expansion in a set of templates, which resemble the macro actions in the bags of an ALMA. For example, a useful template for the Grid domain can be defined:

$$\boxed{\text{move*,pickup,move*,unlock}}$$

In the situations we consider the targets are dependent and separate the ALMA expansions from other template actions. In the ALMA definition SI templates distinguished the SIs within a macro action. In the raised level we distinguish guided targets, which are actions that will be guided by the relaxed plan. We make the constraints explicit by constructing a template layer that captures the constraints between the guided target and contributing targets. There may not be an instantiation of the list of actions in a template that is applicable. Instead these can be thought of as targets that we will aim search towards. These targets are related to landmarks (Porteous et al., 2001; Hoffmann et al., 2004). However, we do not claim that our targets are necessary actions that any plan will use. Instead, we use these intermediary targets to provide guidance towards the final target. The final outcome is a specialised HTN for the particular SI, with the opportunity of optimising its computation.

A template is a list of lifted actions, $T_{op} = b_0, \ldots, b_m$, with a set of shared variables, $V_{b_0,\ldots,b_m}$. A bag of templates is a set $\mathbf{TargetTemplates} = \{T_{op_0}, \ldots, T_{op_n}\}$.

Expanding templates

The aim is to use the relaxed plan to guide the selection of a chain of targets. We generate a total order relaxed plan by selecting any ordering of the action layers. We then generate the possible bindings of the raised templates that are consistent with the relaxed plan. For consistency we say the bindings of the template are satisfied and the guided actions in the template are represented in the relaxed plan. As a heuristic approximation, we use the first action in the relaxed plan that matches with the final template action.

We use the following approach to iteratively generate the possible template instantiations, from the guided action backwards. The i^{th} template layer is made by extending a template from the previous layer. For a sequence of $i - 1$ in the previous layer, generated from the template, T_{op}, we unroll the template a step back. The next action will be partially bound, due to the shared variables in T_{op}. The next layer consists of all valid instantiations of the loose variables. This process can be repeated.

The sequences are extended from the end of the target template sequences. This layered expansion adds a bias to the search that favours sequences that expand fewer layers. This means that we only unroll the templates if an executable action is not found. The search looks for an action that is closest to the end of a template and within those actions, the action that requires the fewest steps. For example, in the Grid template above, a step that moved towards opening a door would be preferred over a step that would move towards a key.

To determine if a target is applicable, the bag is expanded (once per bag for each state) as was discussed in Chapter 7. The potential targets are identified from the partial binding of the template. The guided action is then tested for applicability in each of the states associated with the potential targets during the ALMA expansion. If the target is applicable then the number of steps used in the expansion is used as a heuristic, selecting the nearest.

In the Grid domain the templates relate the collection of keys with the opening of doors. If a door can be opened then this is the first behaviour that is implemented. If this cannot be achieved then the template will be unrolled and previous behaviour, such as moving the robot to the door, will be applied. In Sokoban and (n^2-1)-Puzzle, the relaxed plan could be used to coordinate the movement of the blocks and tiles. Further work would be required to determine whether the relaxed plan can provide the same guarantees in these domains.

G.1.2 Evaluation

We have implemented this hierarchical approach as a wrapper round the ALMA solver. We present the time plot in Figure G.1. The plot demonstrates that the approach can be used successfully to solve problems in the Grid domain; however, it is not as efficient as the handwritten approach or LAMA. One inefficiency is caused through the expansion of each possible template. There is an opportunity to focus this towards the steps in

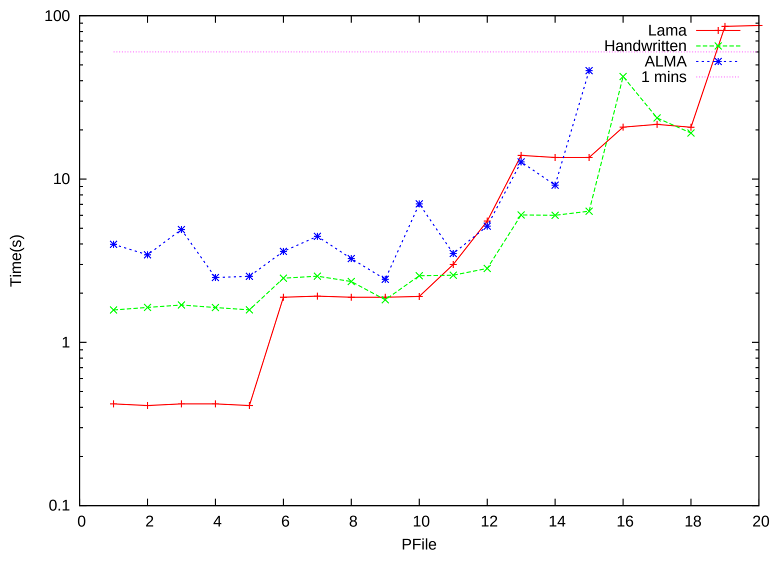


Figure G.1: Time results for ALMA and handwritten strategies for Gird problems.

the relaxed plan. For example, using a directed expansion of the bags, as was used in Botea et al. (2007) (this has been discussed in Chapter 7). A difference between the handwritten solution and the ALMA hierarchy, is that in the handwritten solution we analyse the actions that have been applied to determine whether the reachability could have changed.

In Chapter 5, it was observed that the use of a heuristic was an appropriate alternative for developing specialised solutions for optimising solutions. This has several advantages as the RBP has control over search when possible. An alternative to the approach presented here would be to extend the rule language with disjunctions and allow the RBP to delegate to the heuristic when it could not provide the guidance.

G.2 Automating significance

In this section we consider the problem of determining when it is appropriate to use a projection with an ALMA, or in other words, when a projection is *bag significant*. What we would like to be able to determine is what are the possible sequences of state changes that the sequences could have brought about. In particular, we would like to examine the stages that a group of objects progress through given the sequences. Identifying whether a subset of a state is achievable using a set of actions is as difficult as the planning problem and is therefore equivalent to the problem that we are trying to solve. We observe that we are not interested in parts of the state that are potentially significant; instead, parts of the state that are significant in practice. This has led us to explore an approximate solution.

G.2.1 Target significance

To evaluate the property completely involves every possible problem. We test the significance property using a set, T_P, of n example problems. If these example situations represent the problems of the domain then we expect that a projection that is target significant in these problems will be target significant in every problem. We have selected a single projection to test: the projection to the space of *targets*. This is the minimal target space, as the states must be distinguished by at least the target. If the bags include actions that exchange resources for use at future nodes then this projection will not be target significant. However, it is effective for problems that involve collections of independent local actions on each node. Such as opening a door before entering a room.

Under target significance, expansion of the macro bags requires polynomial time in the number of reachable targets and actions. For each vertex in the reachable targets graph, an expansion of the bags is made by instantiating the bag's macros, if we assume a small constant for the maximum bag size then this is polynomial in the number of actions. Each vertex is only expanded once, and the use of target significance cannot lead to discovering more targets. It is expected that the macros and binding constraints will lead to an expansion of a greatly reduced action set. If target significance is not appropriate then the reached targets could be a smaller set than the targets reachable by the macro bags.

We determine whether the *target* projection is significant for the macro bags over the training examples. If it is then the bag is called *target significant*:

Definition G.2.1

$$\begin{aligned}
&\mathbf{TDTargetSignificant}(T_P, q) \iff \\
&\quad \forall \mathbf{P} = (s_i, g) \in T_P \\
&\qquad \forall t\ (t \in \mathbf{TargetSet}_{mop}(s, I) \iff t \in \mathbf{TargetSet}_{mop}(s, q))
\end{aligned}$$

We have not conducted an exploration of the potential projections, q. Instead we have examined one in particular: the projection to the targets. If this projection is target significant then the state is filtered to indicate the achieved target, otherwise the whole state is used.

We have found that evaluating the target significance of a bag can be an expensive process on medium sized problems (of Goldminer). As a result we use a bounded search. For each example we search using a maximum of m macros applications. If the same set is found using both approaches then we predict that the target is sufficient. This approximation still seems to perform well in practice. We do not evaluate this part of the system in this work.

Appendix H

Enhancing the problem model for structure building problems

In this work we have focussed on the traversing SI as it has received least treatment in the literature. In this appendix we demonstrate the use of our approach in structure building problems.

H.1 Enriching the state with the `well-placed` predicate

In this section we define the `well-placed` predicate and demonstrate how it is exploited in our framework.

H.1.1 Matching a graph in the goal

The common representation of Blocksworld has a hand that moves blocks between stacks. Similarly, blocks are moved using cranes in the Depot domain. This intermediary step means that when using our rule language the connected predicate with the state context is not sufficient to express an RBP. One approach would be to model macro actions that combine pickup and drop off actions and therefore establish a level of reasoning equivalent to the three operator version (Hoffmann, 2005). In Martin and Geffner (2000) a concept language is used and it is demonstrated that control in Blocksworld can be supported with the use of the transitive closure over the `on` predicate in the goal. In Yoon et al. (2002), a combined context (presented in Section 4.2) is used and the transitive closure in this context successfully directs search. An alternative approach is to enhance stacking problem models with the `well-placed` predicate (Bacchus and Kabanza, 2000; Khardon, 1999a; Martin and Geffner, 2000).

The `well-placed` predicate

The `well-placed` predicate can be modelled using the following formulae. For a particular variable, x, the first formula ensures that there are no blocks on top of x. The second formula recursively checks for x and each block under x that it is either not on anything and should not be on anything; or that if it is on a block, then if either block has a goal then it is with the other block.

$$\begin{aligned}&\mathbf{wellPlaced}(x) \iff\\&\quad(\forall y\ \neg(\text{on } y\ x)\ .\ \mathbf{wellPlacedUnder}(x))\end{aligned}$$

$$\begin{aligned}&\mathbf{wellPlacedUnder}(x) \iff\\&\quad[\forall y\ \neg(on\ x\ y)\ .\ \forall y\ \neg(\text{gon } x\ y)] \vee\\&\quad[\exists y\ (\text{on } x\ y)\ .\\&\qquad(\forall\ z(\text{gon } x\ z)\ z = y)\ .\\&\qquad(\forall\ z(\text{gon } z\ y)\ z = x)\ .\\&\qquad\mathbf{wellPlacedUnder}(y)]\end{aligned}$$

Enhancing the language

We can define a step from Σ_i to Σ_{i+1} with a set of propositions. These propositions model the `well-placed` predicate. For a particular state, $s \in \text{wff}(\Sigma_i)$, and graph, $G(s) = (V, E)$, we define the set of propositions:

$$
\begin{aligned}
&\forall u\ \mathbf{wellPlaced}(u)\\
&\quad(\forall s' \in \text{wff}(\Sigma_{i+1})\\
&\qquad s'\mathbf{R}s \implies s' \models (\text{wellPlaced } u))
\end{aligned}
$$

Use of language

This language can be used to express control knowledge for stacking problems. We demonstrate its use in a policy for the Blocksworld problem. A policy for Depot appears in Appendix D.1.

There are four rules necessary to capture a strategy for Blocksworld. The first rule stacks a block on to its goal block if the goal block is well-placed. A block is picked off the table if its goal block is well-placed. The third rule puts a held block on the table. Implicitly this will only happen when its goal is not well-placed: either not clear, or there is a block under it that is not well-placed. The last rule lifts a block from a stack that is not correct.

```
(define (policy blocksworld_policy)
  (:domain blocksworld)
(:rule stack
     :parameters  (?ob ?underob - block)
     :condition (and (clear ?underob) (holding ?ob) (well_placed ?underob))
     :goalCondition (and (on ?ob ?underob))
     :action (stack ?ob ?underob)
)
(:rule pickup
     :parameters (?ob ?underob - block)
     :condition (and (clear ?ob) (on-table ?ob) (arm-empty)
         (well_placed ?underob) (clear ?underob))
     :goalCondition (and (on ?ob ?underob))
     :action (pickup ?ob)
)
(:rule putdown
     :parameters  (?ob - block)
     :condition (holding ?ob)
     :goalCondition (and )
     :action (putdown ?ob)
)
(:rule unstack
     :parameters  (?ob ?underob - block)
     :condition (and (on ?ob ?underob) (clear ?ob) (arm-empty)
         (not(well_placed ?ob)))
     :goalCondition (and )
     :action (unstack ?ob ?underob)
)
)
```

H.2 Arbitrary length macro action case study: structure building

In this section we generate ALMAs for structure building problems. We follow the system that we developed in Chapter 7 for traversal problems. In this appendix we focus on sequences of actions that will clear a particular block in a stack. The first step is to create the mapping between structure building problems and our arbitrary macro action representation. A set of target identifying rules is defined, this was first presented in (Lindsay, 2012); the resulting bags are presented for the Blocksworld domain; we conclude by discussing the use of the language and by discussing the limitations of our SI based ALMA representation in this form of problem.

H.2.1 Problem mapping

The most common structure building problem in planning is the stacking problem. We have limited our study to this form of structure building problem. The structures that are acted on in these problems are stacks and the planner is limited to interacting with the structures from its top element. We assume there is a storage space with sufficient room, for example a table.

A necessary sequence acted on these stacks is uncovering a block from a structure. Uncovering blocks requires a chain of actions that iteratively removes blocks from a structure. Each of these removal steps might require several actions: if the movement of a block is separated into distinct actions, or if the actions need enabled.

The definition of an ALMA requires the macro bag and the bindings between the macros. The macro bag is defined separately for each domain and target type. The other parameters can be generated from the uncovering problem in the context of a structure building problem.

We assume that any macro that forms part of an uncovering solution will detach a block and attach the block somewhere. Therefore the SI template will have the form $\ldots$,(detach o_1 o_2),$\ldots$,(attach o_1 o_3). The binding constraints, B_{mop_0,mop_1}, for macro, m_{op_0}, used directly before macro, m_{op_1}, enforces that ($o_2(\mathbf{V}_{mop_0}) = o_1(\mathbf{V}_{mop_1})$). This means that blocks are removed from the same stack.

The target of these problems is the block that is uncovered. This is the block that was detached from in the macro; o_2, in the previous example.

H.2.2 Targets

The target for an uncovering block problem is that a particular block is free to be picked-up. We assume that there are plans that can be used to extract example sequences. One interpretation of the rule would be that every unstack action satisfies a target: because we lift the block, clearing the block below. However, it is more powerful to identify potentially longer sequences of actions. In particular, we identify where blocks were cleared and then used. We follow a similar process to the one used for structure traversal (in Section 7.4).

The relevant actions are extracted: selecting those actions that enable the removal step actions (for example the pickup and put-on-table actions in Blocksworld), in a similar method as for traversal actions. This leaves a backbone of actions relevant to structure building. We then break this thread up into sequences by identifying when an important block has been cleared. We use a structure to determine whether a block was an important target.

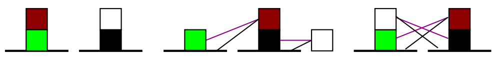


Figure H.1: Example structure, with old-connections in black and below-connections in purple. When white is put on green we examine the stack old-connections and find that red is on black.

The structure updates two distinct sets of connections that record the previous stack of a moved block. When a block is detached from a stack a *below-connection* is made with each block underneath it. When the block is attached to a new stack an *old-connection* is made with its old stack. If a moved block is already connected then we do not update the connections. If it has been attached to an intermediary stack then this could have been achieved using the table. We use this structure to determine whether uncovering a particular block was the target.

We define the following rule to divide the thread of stack interactions into sequences:

1. A block, b, is a target if: b is being attached; b will not be removed; and each of the blocks that have old-connections to b's stack are in the final state not in stacks above blocks that are below-connected to b.

This last property states that if a block was underneath b and will be underneath a block that has been moved from b's stack then it was the reason that b was moved. Therefore unstacking b was not a target. This allows us to break the thread into single sequences that achieve important targets.

Macro bags: the blocksworld domain

We have simulated the process by hand and an example of the bag that might be computed is, **unearthBag**:

- (unstack b1 b2), (putdown b1);
- (unstack b1 b2), (stack b1 b3).

This bag is perhaps more interesting than it might appear at first glance. The sequences that we have derived the language from might include examples of stacking blocks on top of each other. Assuming reasonable sample sequences then blocks will not be put onto a tower that would be dismantled later. Our vocabulary does not capture this important distinction. In allowing the stack action in the bag above we missed the opportunity to restrict the solver's options to sensible sequences.

The consequence of this is that the use of our vocabulary could lead to a cycle. For example, if there are two blocks to be uncovered, $b1$ and $b2$ and each is in a stack: $b1$ is clear and $b2$ has a single block *bp* on top. A solution to uncovering $b2$ is to pickup the block and stack it on top of $b1$. Subsequently a solution to uncovering the newly covered $b1$ is to stack the block back on $b2$, causing the loop. However, we demonstrate in the next section that we can overcome this potential problem in the rules.

H.2.3 Use and limitations of the vocabulary

The system that we have developed for using ALMA comes with the assumption that any expansion of the macro bags that leads to an achieved target is equivalent. This means that the control of the states that are visited during the macro action expansion for a particular target are limited to the selection of the macro actions in the bag and the binding constraints between them. However, if we use SbS, we can use the rules to control the actions that are being executed.

We assume the ALMA that we have developed enhances the problem model with the reachability condition, (`can_unearth` *?b*) and the action, (`unearth` *?b*). The unearth macro action can be used to unstack bad towers. An example of a policy for the Blocksworld domain that unearths blocks that must be clear in the goal is presented here:

```
(define (policy AMA_unearth_block)
(:domain blocksworld)
(:rule unearth_block
  :parameters (?b ?onBlock - block)
  :condition (and (on ?onBlock ?b))
  :goalCondition (and (clear ?b))
  :action (unearth ?b)
)
(:rule discard_block
  :parameters (?b ?held - block)
  :condition (and (holding ?held))
  :goalCondition (and (clear ?b))
  :action (putdown ?held)
)
```

This policy demonstrates that even though the vocabulary might not be perfect for the situation, we will sometimes be able to compensate in the rule system and make use of it nonetheless. We discussed this in the future work Section 10.2.

In the context of the goals set in the benchmark planning problems this action has limited use. In these problems we are often required to create towers of blocks. However, the construction of these towers relies on building on good-towers, which is a recursive property. If the bottom of a stack is made explicit in the goal then we could use rule ordering and iterative composition of towers and form a strategy for constructing good towers. We assume this is modelled by a predicate (`goalBottom` *?b*).

```
(define (policy AMA_unearth_block)
(:domain blocksworld)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; the cases that lead to unearthing a block
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:rule unearth_wrong_above
  :parameters (?b1 ?b2 ?onBlock - block)
  :condition (and (can_unearth ?b1) (on ?onBlock ?b1))
  :goalCondition (and (on ?b2 ?b1) (not (on ?onBlock ?b1)))
  :action (unearth ?b1)
)
(:rule unearth_wrong_below
  :parameters (?b1 ?b2 ?underBlock - block)
  :condition (and (can_unearth ?b1) (on ?b1 ?underBlock))
  :goalCondition (and (on ?b1 ?b2) (not (on ?b1 ?underBlock)))
  :action (unearth ?b1)
)
; once unearthed this block needs remove too
(:rule remove_wrong_below
  :parameters (?b1 ?b2 ?underBlock - block)
  :condition (and (can_unearth ?b1) (on ?b1 ?underBlock))
  :goalCondition (and (on ?b1 ?b2) (not (on ?b1 ?underBlock)))
  :action (unstack ?b1 ?underBlock)
)
(:rule unearth_block
  :parameters (?b ?gonBlock - block)
  :condition (and (can_unearth ?b) (onTable ?b))
  :goalCondition (and (on ?b ?gonBlock))
  :action (unearth ?b)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; the putdown rules during an unearth
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:rule putdown_with_wrong_above
  :parameters (?held ?b1 ?b2 ?onBlock - block)
  :condition (and (can_unearth ?b1) (on ?onBlock ?b1) (holding ?held))
  :goalCondition (and (on ?b2 ?b1) (not (on ?onBlock ?b1)))
  :action (putdown ?held)
)
(:rule putdown_with_wrong_below
  :parameters  (?held ?b1 ?b2 ?underBlock - block)
  :condition (and (can_unearth ?b1) (on ?b1 ?underBlock) (holding ?held))
  :goalCondition (and (on ?b1 ?b2) (not (on ?b1 ?underBlock)))
  :action (putdown ?held)
)
(:rule putdown_with_covered_table_below
  :parameters  (?held ?b ?gonBlock - block)
  :condition (and (can_unearth ?b) (onTable ?b) (holding ?held))
  :goalCondition (and (on ?b ?gonBlock))
  :action (putdown ?held)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; rebuild first layer
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:rule putdown_first_layer
  :parameters (?b1 ?b2 - block)
  :condition (and (onTable ?b2) (clear ?b2) (holding ?b1) (goalBottom ?b2))
  :goalCondition (and (on ?b1 ?b2))
  :action (stack ?b1 ?b2)
)
(:rule pickup_for_first_layer_Block
  :parameters (?b1 ?b2 ?onBlock - block)
  :condition (and (onTable ?b2) (clear ?b2) (on ?b1 ?onBlock)
    (clear ?b1) (goalBottom ?b2))
  :goalCondition (and (on ?b1 ?b2))
  :action (unstack ?b1 ?onBlock)
)
(:rule pickup_for_first_layer_Table
  :parameters (?b1 ?b2 - block)
  :condition (and (onTable ?b2) (clear ?b2) (onTable ?b1)
    (goalBottom ?b2) (clear ?b1))
  :goalCondition (and (on ?b1 ?b2))
  :action (pickup ?b1)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; build next layer
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:rule putdown_next_layer
  :parameters (?partialB1 ?partialB2 ?b - block)
  :condition (and (on ?partialB1 ?partialB2) (clear ?partialB1) (holding ?b))
  :goalCondition (and (on ?partialB1 ?partialB2) (on ?b ?partialB1))
  :action (stack ?b ?partialB1?)
)
(:rule pickup_for_next_layer_Block
  :parameters (?partialB1 ?partialB2 ?b ?onBlock - block)
  :condition (and (on ?partialB1 ?partialB2) (clear ?partialB1)
    (on ?b ?onBlock) (clear ?b))
  :goalCondition (and (on ?partialB1 ?partialB2) (on ?b ?partialB1))
  :action (unstack ?b ?onBlock)
)
(:rule pickup_for_next_layer_Table
  :parameters (?partialB1 ?partialB2 ?b - block)
  :condition (and (on ?partialB1 ?partialB2) (clear ?partialB1)
    (onTable ?b) (clear ?b))
  :goalCondition (and (on ?partialB1 ?partialB2) (on ?b ?partialB1))
  :action (pickup ?b)
)
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; discard block in hand
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(:rule discard_block
  :parameters  (?held - block)
  :condition (and (holding ?held))
  :goalCondition (and )
  :action (putdown ?held)
)
```

This policy does not lead to a particularly efficient strategy. As blocks are unearthed we put the blocks on the table. Only once this process has finished do we allow stacks be rebuilt. In order to allow only good-towers to be built we must control the building of stacks. We achieve this by controlling the construction of stacks of size two, using the `goalBottom` predicate. After this we can build on any tower that is at least a good-tower to the depth of two. The final rule acts as a default that makes the hand free when there is no use of the current block.

In conclusion, we have presented a target type for a structure building problems and demonstrated that we can use ALMA as part of a solution to solving certain types of problem. Once again we have seen that targets that can be solved mostly independently from the rest of the problem work best. However, the rules that use the vocabulary, the macros in the bags and binding constraints, and the rule system allow parameterisation of the vocabulary and provide some flexibility in the computation behind modelling the vocabulary.

An alternative is to evaluate the ALMA expansion in a context with the achieved goal context (Yoon et al., 2002). If the assumptions of target significance held then this would provide a method of implementing a good tower builder. The targets would be paths from the bottom of the tower to the top in the achieved goal context.

APPENDIX I

A CHAIN OF LANGUAGE RESTRICTIONS: FURTHER INTUITION

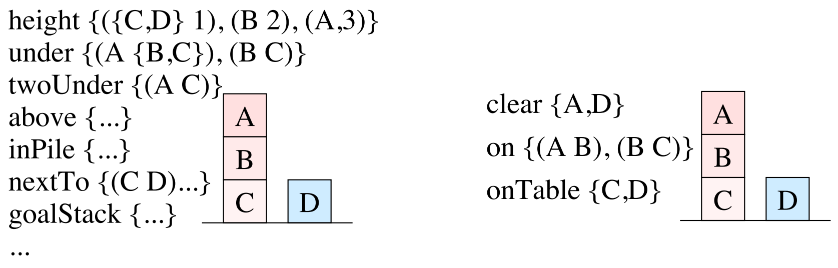
I.1 Vocabulary rich modelling and its restricted views

In Chapter 3 we introduced a notional, vocabulary rich model, $\mathbb{M}$, and its restricted views, $\mathbb{M}|_{\Sigma}$, for a language, Σ. In this section we describe these ideas in more detail, providing more intuition and making explicit more of the relationships between $\mathbb{M}$ and its restricted view.

I.1.1 Vocabulary rich modelling

The vocabulary defined by the model influences the way that the planner reasons and interacts with the model. The Blocksworld planning model typically defines three sets of propositions: the `on` *?a ?b* propositions that exist between two blocks *?a* and *?b* if *?a* sits directly on top of *?b*; `clear` *?a*, that holds if any block, *?a*, is on top of a stack; and `ontable` *?a* that is true of any block, that sits on the table. For example, Subfigure I.1(b) illustrates the modelled propositions in a Blocksworld problem. Blocks can be picked up by a hand and placed onto the table or another block. These propositions allow a concise representation of the valid actions in the problem. However, there are many propositions that are not modelled, such as the `two under` proposition.

It has been shown by Khardon (1999a); Martin and Geffner (2000) that the modelled propositions are an important consideration for certain planning techniques, such as planners that exploit formulae over the propositions modelled by states. For exam-



(a) Rich representation (b) Sufficient representation

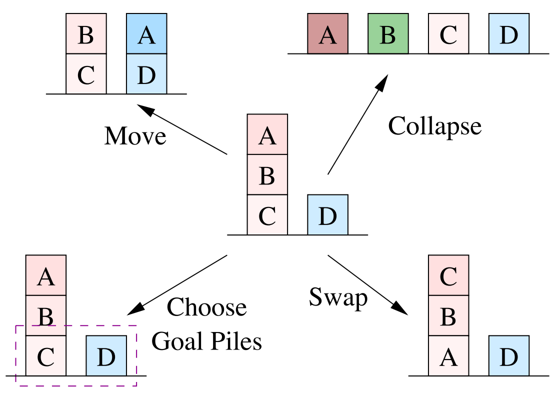
Figure I.1: Propositions for two approaches to modelling a Blocksworld problem.

ple, in Khardon (1999a) it is demonstrated that enhancing the problem model with additional propositions can lead to improved planning performance. The Subfigure I.1(a) illustrates several relationships that could be captured in a rich model of the problem. For example, concepts such as `two under`, or more generally the `somewhere under` propositions. These are basic relationships that hold between objects in the environment. The `somewhere under` *?a ?b* proposition can be useful as it can link a required block, *?b*, with the block, *?a*, at the top of its stack that needs to be moved first.

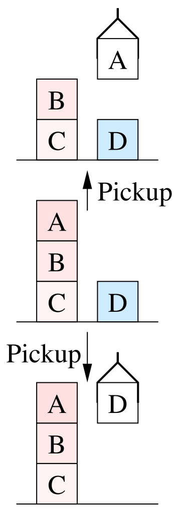
There are many conceptual methods of interaction with these objects; for example, in a Blocksworld problem it might be useful to consider dismantling an entire stack of blocks. However, the presented model defines the vocabulary that dictates how a planner can interact with the objects in the problem. In the benchmark problems the planner is restricted in interaction; only one block can be moved at a time. Even moving a block might involve picking the block up and then putting it down somewhere else, illustrated in Figure I.2(b).

A rich model of the problem could represent conceptual actions on the objects; this would allow the planner to select the level of abstraction that a decision was made at. For example, in an alternative representation of the Blocksworld problem where the arm is not modelled, the relaxed plan heuristic is more informative (Hoffmann, 2005) and the performance of control knowledge based planning can be improved (Aler et al., 2000a). The progression through states introduces many layers of interpretation of the objects in the world; some of these are illustrated in Figure I.2(a). For example, collapsing an entire stack onto the table is a natural extension from pickup and place actions.

We can also consider exactly why a certain behaviour is being carried out, in terms



(a) Rich representation



(b) Sufficient representation

Figure I.2: Actions for two approaches to modelling a Blocksworld problem.

of the individual decisions that led to the choice being made. These decisions are actions, as they manipulate propositions at some conceptual level. For example, it might be useful to decide which stacks will be used for developing the final goal stacks, setting conceptual propositions in the state (I.1(a)). In transportation problems, prior to a truck being moved, the planner will determine a package to collect, or perhaps the entire tour that will be made for redistribution. In a rich model of the planning problem the planner should be able to make decisions at the suitable level, including more "fine-grained" levels of the decisions.

Definition I.1.1 *A rich planning model, $\mathbb{M}$, is a state transition system, $(\mathbb{S}, \mathbb{A}, \gamma)$, such that,*

- *the states in $\mathbb{S}$ model propositions that capture the concepts and relationships that exist between objects or states in the state machine;*
- *the transitions connect the states where a transition exists in some level of interpretation of the model;*
- *$\mathbb{M}$ has a current state, $s_C \in \mathbb{S}$.*

This model provides a rich environment that allows the problem to be interpreted at many levels; in particular, a level that is relevant to the decision being made. The states model all of the propositions that are suggested by the behaviours in the problem. We can observe that the interpretation of the problem as a whole introduces the concept of a solve action. Thus the vocabulary includes a single step plan for any solvable planning problem. The graph defined by the vertex and edge sets $\langle \mathbb{S}, \mathbb{A} \rangle$ is therefore densely connected:

$$\forall s, s' \in \mathbb{S} \; (\exists a_1, \ldots, a_n \; s' = \gamma(\ldots(\gamma(s, a_1)\ldots), a_n) \implies (\exists a \; s' = \gamma(s, a))$$

A planning problem

A planning problem is described by a triple, $\mathbb{P} = \langle \mathbb{M}, i, g \rangle$, where $\mathbb{M} = (\mathbb{S}, \mathbb{A}, \gamma)$ is a concept rich planning environment model, $i \in \mathbb{S}$ is the initial state and g is a goal formula. The current state of $\mathbb{M}$ is initialised to the initial state, $s_C = i$.

I.1.2 A restricted model

The restricted model, $\mathbf{M}$, accepts the sentences of a language, Σ. We define $\mathbf{M}$ as a view of $\mathbb{M}$, such that $\mathbf{M}$ is consistent with $\mathbb{M}$ for any sentence that can be expressed in Σ. In this subsection we formalise this relationship.

Restricting the view of the model

If the language, Σ, cannot express all of the actions and propositions in $\mathbb{M}$ then the planner is given a *restricted view* of $\mathbb{M}$. This will result in the restricted model describing a state transition system, $\mathbf{M} = (\mathbf{S}, \mathbf{A}, \gamma_\Sigma)$, with fewer actions and states (Figure I.3). We require that the restricted model captures all of the states and actions that can be expressed in Σ. Similarly, there should not be extra actions or states that are not in the model. Otherwise consistency will not follow.

States and propositions The restricted model, $\mathbf{M}$, for language, Σ, is defined for propositions expressible in Σ: $\mathbf{M} \models p$, where $p \in \text{wff}(\Sigma)$. This extends to sets of wff(Σ) and hence to states. A state, s, in the language, Σ, represents a state, s', in $\mathbb{M}$, if every proposition that is expressible in Σ and is entailed by s' is also entailed in s. Formally, we say:

$$s'(\mathbb{M})\mathbf{R}s(\Sigma) \iff (\forall p \in \text{wff}(\Sigma) \; s \models p \iff s' \models p).$$

A state in wff(Σ) is modelled by $\mathbf{M}$ if it represents a state that is modelled by $\mathbb{M}$.

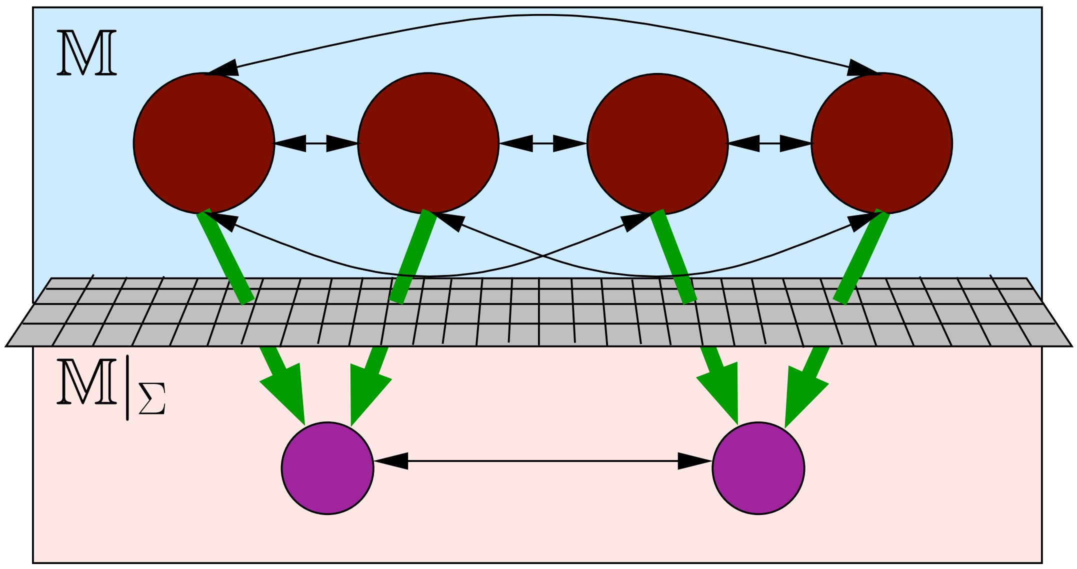


Figure I.3: Capturing $\mathbb{M}$ in a language acts like a filter, retaining the parts of the model that can be expressed in the language and removing the parts that cannot.

$$\mathbf{M} \models s(\Sigma) \iff (\exists s'\ s'\mathbf{R}s \ \&\ \mathbb{M} \models s')$$

A state in $\mathbb{M}$ might entail more propositions than the representing state in the restricted model. As a result the distinction between two states in $\mathbb{M}$ might not be expressible in Σ and they are represented by the same state in wff(Σ). A state in $\mathbb{M}$, however, will be represented by at most a single state in the restricted model. Of course there may be states modelled by $\mathbb{M}$ that are not represented by states in wff(Σ).

Actions As several states in $\mathbb{M}$ can be represented by a single state in wff(Σ), a single action in wff(Σ), between two states in wff(Σ), can represent many actions in $\mathbb{M}$. As the states in $\mathbb{M}$ are richer, there can be actions that have an effect on parts of the state that are not expressible in Σ. In particular, there can be actions that have no visible effect with respect to the state expressible in Σ; as well as actions that transition to states not represented by states in wff(Σ). An action in $\mathbb{M}$ will be represented by at most a single action in wff(Σ).

An action $a \in$ wff(Σ) represents an action $a' \in \mathbb{S}$ if the part of a' expressible in Σ has the same effect on the state as a.

$$\begin{aligned}
&a'(\mathbb{M})\mathbf{R}a(\Sigma) \iff \\
&\quad (\forall s'_0, s'_1 \in \mathbb{S}\ s'_1 = \gamma(s'_0, a') \implies \\
&\quad\quad \forall s_0 \in \text{wff}(\Sigma)\ (s'_0\mathbf{R}s_0 \implies (s'_1\mathbf{R}\gamma_\Sigma(s_0, a))))
\end{aligned}$$

An action in $\text{wff}(\Sigma)$ is entailed by the restricted model if it represents an action in the conceptual model.

$$\mathbf{M} \models a(\Sigma) \iff (\exists a' \in \mathbb{A}(\mathbb{M})\ a'\mathbf{R}a)$$

The restricted view We define the restricted model, $\mathbf{M}$, that accepts the language, Σ, as a restricted view of $\mathbb{M}$. $\mathbf{M}$ is consistent with $\mathbb{M}$ for any sentence expressible in Σ. As we have defined $\mathbf{M} \models e$ for propositions, states and actions, these are all modelled by the restricted view. These definitions ensure that $\mathbf{M}$ captures as much of $\mathbb{M}$ as can be expressed in Σ and that there is nothing captured by $\mathbf{M}$ that is not part of $\mathbb{M}$.

Definition I.1.2 *A Restricted View:* $\mathbb{M}|_\Sigma$*, for some language* Σ*, such that*

$$\forall e\ \ \mathbb{M}|_\Sigma \models e \iff (e \in \mathit{wff}(\Sigma)\ .\ \mathbf{M} \models e)$$

The planning problem We use $\mathbf{P} = \langle \mathbb{M}|_\Sigma, s_{\text{init}}, g\rangle$ to represent the problem for the restricted view for $\mathbb{M}$, for the language Σ. $s_{\text{init}} \in \mathbf{S}$ is the state in $\text{wff}(\Sigma)$ that represents the initial state of the problem, $s_{\text{init}}(\mathbf{S})\mathbf{R}i(\mathbb{P})$, and g is the goal formula.

A language, Σ, can be used to solve a planning problem $\mathbb{P} = \langle \mathbb{M}, s_{\text{init}}, g\rangle$ if the initial state is represented in the states of the restricted model, $\mathbb{M}|_\Sigma$, and if the states that satisfy the goal in $\mathbb{M}$ are represented by states in $\mathbf{M}$ that satisfy the goal of the problem.

Definition I.1.3

$$\begin{aligned}
&\textit{The restricted model } \mathbb{M}|_\Sigma = (\mathbf{S}, \mathbf{A}, \gamma_\Sigma) \textit{ is usable for problem } \mathbb{P} \iff \\
&\quad (\exists s_{\mathit{init}} \in \mathbf{S}\ i\mathbf{R}s_{\mathit{init}})\ .\ (\forall s \in \mathbf{S}, s' \in \mathbb{S}\ (s' \models g\ .\ s'\mathbf{R}s) \implies s \models g)
\end{aligned}$$

I.2 A chain of language restrictions

In definition 3.1.2 we defined a chain of language restrictions as a collection of languages that are ordered in terms of expressivity, with the most limited language as the first element of the chain. In the following we take a more formal approach to defining the chains that we focussed on in this work.

The goal of the chain of language restrictions is to enhance the problem model. As such, all of the states of the model should be represented in the enhanced model. If this is not the case then our view of the model has become more restricted. Also, there is no reason to consider states that are not represented by any of the described model states. These states fall outside the model and any solution that includes them will be impossible to interpret with respect to the restricted model.

Definition I.2.1

$$\begin{aligned}
&\textit{ShapePreserving}_{\Sigma_0 \to \Sigma_1} \iff \\
&\quad \forall s \in \textit{wff}(\Sigma_0)\ \exists s' \in \textit{wff}(\Sigma_1)\ s'\mathbf{R}s \\
&\quad \forall s' \in \textit{wff}(\Sigma_1)\ \exists s \in \textit{wff}(\Sigma_0)\ s'\mathbf{R}s \\
&\quad \forall s_0, s_1, a\ s_1 = \gamma_{\Sigma_0}(s_0, a) \implies \\
&\qquad (\forall s_0'\ s_0'\mathbf{R}s_0 \implies (\exists s_1', a'\ s_1' = \gamma_{\Sigma_1}(s_0', a')\ .\ a'\mathbf{R}a))
\end{aligned}$$

The intuition behind this is that the structure of the described model is important and its shape should be preserved in any restricted model for languages on a chain. In particular, the behaviours of objects in the restricted model should still be possible in the richer models and any new behaviours introduced into the model should not move outside states that are represented by states in the described model.

The concept of *ShapePreserving* goes some way to capture this intuition. However, a chain that is *ShapePreserving* could model an action that bridged two sets of states that are disconnected in the described model. Such actions are not interesting as we cannot interpret them in the described model. We focus our attention on two groups of actions and propositions: those that abstract the model and those that enrich the model (Figure I.4).

As the actions in $\mathbb{M}$ form a densely connected graph connecting the states, it is likely that there will be many actions that would describe shorter paths between the states in $\mathbb{M}|_{\Sigma}$. These actions correspond to abstracting actions that are composed of several of the actions described in the problem model. It should be noted that although these actions have the same effect as several actions and therefore provide an abstract way of interacting with the model, they are added to the existing actions and are therefore an enhancement. We define an abstracting action with respect to a language that the action can be expressed, Σ_1, and a more limited language, Σ_0.

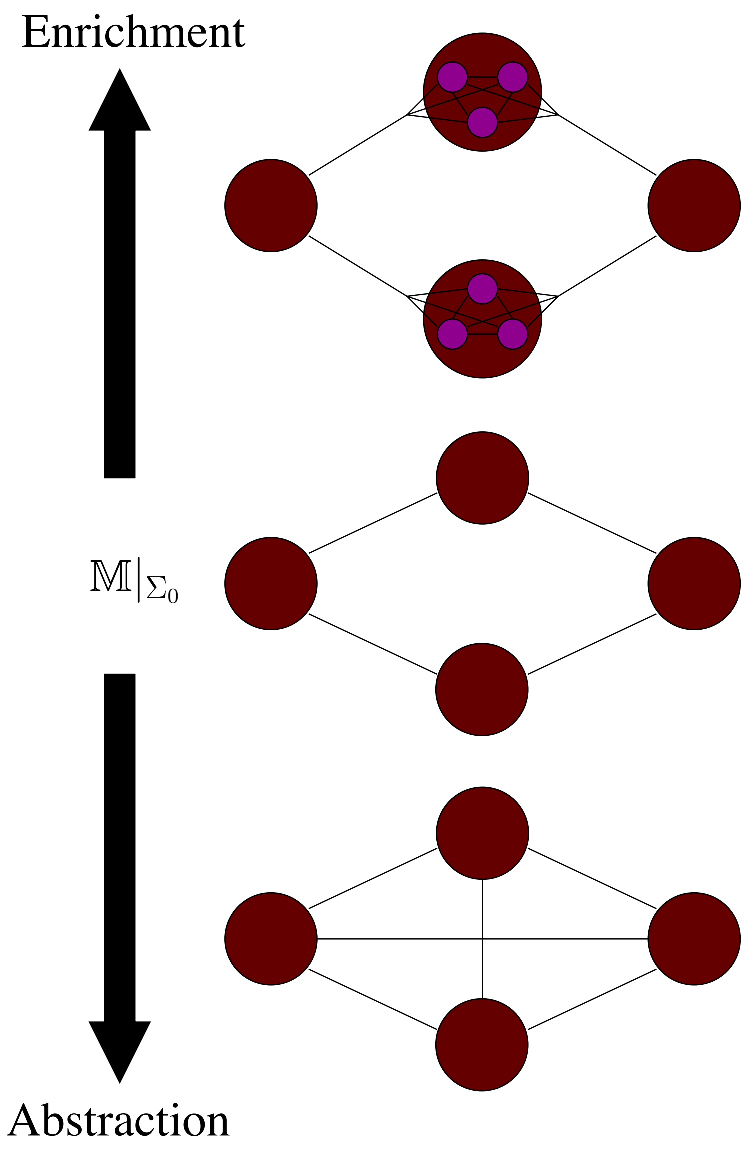


Figure I.4: In relation to the described model $\mathbf{M} = \mathbb{M}|_{\Sigma_0}$, there are two groups of concepts that can be added into the model: concepts that abstract the model and concepts that enhance the model.

Definition I.2.2

$$\begin{aligned}
&\mathit{AbstractingAction}_{\Sigma_0 \to \Sigma_1}(a') \iff \\
&\quad a' \notin \mathit{wff}(\Sigma_0) \\
&\quad (\forall s'_0, s'_1 \in \mathit{wff}(\Sigma_1)\ s'_1 = \gamma_{\Sigma_1}(s'_0, a') \implies \\
&\qquad (\exists a_0, \ldots, a_{n-1}, s_0, s_n \in \mathit{wff}(\Sigma_0) \\
&\qquad\quad s_n = \gamma_{\Sigma_0}(\ldots(\gamma_{\Sigma_0}(s_0, a_0)\ldots)a_{n-1})\ .\ s'_0 \mathbf{R} s_0\ .\ s'_1 \mathbf{R} s_n))
\end{aligned}$$

Adding abstracting actions to the problem model is a well researched topic within Automated Planning. They are typically called macro actions (defined in Section 2.3) and can allow large sections of the search space to be cut through. The usual downside to using abstracting actions is that they can greatly increase the total number of actions in the model and increase the branching factor at each state. The planning approach that we use in this work does not does not ground all of the possible actions prior to search and as a consequence this problem is greatly reduced. We discuss abstracting actions in more detail in Section I.5.

We have observed that some decisions made during planning are made implicitly. One aspect of this work investigates introducing these decisions as explicit propositions in the states of the model. Our framework supports adding actions that change the enriched part of the state and therefore allow these decisions to be made as part of the planning process.

Definition I.2.3

$$\begin{aligned}
&\mathit{EnrichingAction}_{\Sigma_0 \to \Sigma_1}(a') \iff \\
&\quad (\forall s'_0, s'_1 \in \mathit{wff}(\Sigma_1)\ s'_1 = \gamma_{\Sigma_1}(s'_0, a') \implies \\
&\qquad \forall s \in \mathit{wff}(\Sigma_0)\ s'_0 \mathbf{R} s \iff s'_1 \mathbf{R} s)
\end{aligned}$$

These actions transition between states that are represented by the same state in the described language. They break up the decision making into smaller steps by using propositions that are part of the less restricted models.

We combine the property of *ShapePreserving* with the two forms of model enhancements to define a chain of language restrictions.

Definition I.2.4

The languages $\Sigma_0, \ldots, \Sigma_n$, are a chain of language restrictions of the model $\mathbb{M}$, if:

$\mathbb{M}|_{\Sigma_0}, \ldots, \mathbb{M}|_{\Sigma_n}$ *are restricted views of* $\mathbb{M}$

Σ_0 *is usable for* $\mathbb{M}$

for $i = 0, \ldots, n-1$

$\quad$ *ShapePreserving*$_{\Sigma_i \to \Sigma_{i+1}}$

$\quad \forall a \in \textit{wff}(\Sigma_{i+1})$

$\quad\quad (a \in \textit{wff}(\Sigma_i) \lor \textit{EnrichingAction}_{\Sigma_i \to \Sigma_{i+1}}(a) \lor \textit{AbstractingAction}_{\Sigma_i \to \Sigma_{i+1}}(a))$

$\quad (\exists a\, \mathbb{M}|_{\Sigma_{i+1}} \models a \,.\, \mathbb{M}|_{\Sigma_i} \not\models a) \lor (\exists p\, \mathbb{M}|_{\Sigma_i} \not\models p \,.\, \mathbb{M}|_{\Sigma_{i+1}} \models p)$

Each of the languages in a chain of language restrictions, Σ_i, can be used to provide a restricted view of $\mathbb{M}$, $\mathbb{M}|_{\Sigma_i}$. For high values of i this view will be less restricted than for lower values of i. A chain can be seen as a series of steps that can be taken between languages: either downwards from an expressive language to a limited language or upwards from a limited language to an expressive language.

I.3 Policy transferral

In this section we consider how a policy expressed in one model can be used as a policy in another model. This is discussed for policies between the rich conceptual model, $\mathbb{M}$, and a policy for a language, Σ; between two languages on a chain of language restrictions; and between two languages in the case of of co-execution.

I.3.1 Transferring between $\mathbb{M}$ and Σ

As the models have different sets of states and actions it means that a policy that is for the states and actions of $\mathbb{M}$ is not directly applicable as a policy for a syntactic model with states and actions in Σ and similarly for using a policy for Σ as a policy for $\mathbb{M}$. This also means that a plan for one model cannot be used directly as a plan for the other. However, a policy for one model can be interpreted as a policy for the other model. We assume that the restricted model $\mathbb{M}|_\Sigma$ is usable for the problem being solved (Definition I.1.3).

π_{Σ_i} for use with $\mathbb{M}$

The process for interpreting a policy, π_Σ, for the Σ language model, as a policy for $\mathbb{M}$ is quite straightforward. However, π_Σ will only provide a partial policy for $\mathbb{M}$. This is because there are states in $\mathbb{M}$ that are not represented in Σ. For a state, s, in $\mathbb{M}$ there is at most one state, s_0, in the Σ model that represents it. If there is not a state, s_0, then the mapping, $\pi_{\mathbb{M}}(s)$, is undefined. Otherwise, π_Σ maps from s_0 to a single state, s_1. However, it may represent several states in $\mathbb{M}$. There is therefore a set of actions that link from s to each of these states and we define these as the *represented action set* ($\mathbf{RAS}$).

$$\begin{aligned}
&\mathbf{RAS}(s'_0) = \{a'(\mathbb{M})| \\
&\quad \exists s \in \mathbf{S}, a \in \mathbf{A} \\
&\quad\quad s'_0\mathbf{R}s \;.\; a = \pi_\Sigma(s) \;.\; a'\mathbf{R}a \;.\; (\exists s'_1 \; s'_1 = \gamma(s'_0, a'))\}
\end{aligned}$$

These actions are equivalent with respect to the model captured in Σ. Any deterministic selection process, f, can be used to select a single action. This provides the partial mapping: $\pi_{\mathbb{M}}(s) = f(\mathbf{RAS}(s))$.

In the case of plans then it is guaranteed that a plan, π_Σ, is a plan for $\mathbb{M}$. The initial state and at least one goal satisfying state must be expressible in Σ. This relies on $\mathbb{M}|_{\Sigma_i}$ being usable for the problem (Definition I.1.3). The actions described by π_Σ join states that are in wff(Σ). From the definition, the actions in $\mathbf{RAS}(s)$ link to states that are represented by a state in wff(Σ). As the initial state is represented in wff(Σ) this means that execution will not move to a state that is not represented by a state in wff(Σ). If the restricted model was not usable with the problem then there might be a state in $\mathbb{M}$ that is not a goal for the problem, but is represented by a goal state in $\mathbb{M}|_\Sigma$. This could result in the policy providing a mapping for the states along a path to a state represented by a goal state in $\mathbb{M}|_\Sigma$ but then providing no more guidance.

$\pi_{\mathbb{M}}$ for use with $\mathbb{M}|_\Sigma$

The process for interpreting a policy, $\pi_{\mathbb{M}}$, for $\mathbb{M}$, as a policy for the Σ model is slightly more complicated. For a state, s, in Σ there might be several (at least one) states represented by s in $\mathbb{M}$. $\pi_{\mathbb{M}}$ maps each of these states to an action. This describes a set of actions in $\mathbb{A}$ that we label the *abstract action set* ($\mathbf{AAS}$):

$$\mathbf{AAS}(s) = \{a'(\mathbb{M})| \; \exists s' \in \mathbb{S} \; (a' = \pi_{\mathbb{M}}(s') \;.\; s'\mathbf{R}s)\}$$

We then define the *base language action set* ($\mathbf{BAS}$) of actions in wff(Σ) that represent actions in $\mathbf{AAS}(s)$.

$$\mathbf{BAS}(s) = \{a(\Sigma) | \ \exists a' \in \mathbf{AAS}(s) \ a'\mathbf{R}a\}$$

The set $\mathbf{BAS}(s)$ can be empty and even if it is never empty, the interpreted map, $\pi_\Sigma(s) = f(\mathbf{AAS}(s))$ (for some deterministic selection function, f), is not guaranteed to be a policy. Execution can enter loops, and due to the deterministic selection process, the loop will never be left. Intuitively, these loops occur where the policy maps to a state and the distinction between this state and a previous state is not expressible in Σ, leading to a loop.

$$\begin{aligned}
&s_0(\Sigma), \dots, s_n(\Sigma) \text{ loop in } \pi_{\mathbb{M}} \Leftarrow \\
&\quad \forall i \in [0, \dots, n-1], \exists a \in \mathbf{BAS}(s_i) \ s_{i+1} = \gamma(s_i, a) \\
&\quad s_0 = s_n
\end{aligned}$$

We define the Σ-*coherent* property such that if a policy, $\pi_{\mathbb{M}}$, is Σ-*coherent* then it can be interpreted as a policy for a model expressed in Σ.

Definition I.3.1

$$\begin{aligned}
&\pi_{\mathbb{M}} \textit{ is } \Sigma\textit{-coherent} \iff \\
&\quad \forall s \in \textit{wff}(\Sigma) \ \mathbf{BAS}(s) \neq \emptyset \\
&\quad \forall s_0(\Sigma), \dots, s_n(\Sigma) \ s_0, \dots, s_n \textit{ is not a loop in } \pi_{\Sigma_j} \\
&\quad \forall s' \in \mathbb{S} \ s' \models g \implies (\exists s \in \mathbf{S} \ s'\mathbf{R}s)
\end{aligned}$$

There can be goal states in $\mathbb{M}$ that are not represented by states in the restricted view. The final line of Definition I.3.1 ensures this is not the case. An alternative would be to demonstrate that an executive would not be directed towards these states.

This property can be weakened for plans by ensuring that any reachable state (through policy execution) is either the goal or has an action to apply, and that any sub-chain of the states reached on a chain of execution is not a loop.

I.3.2 Transferring between languages

For any two languages Σ_i and Σ_j, $i < j$ from a chain of language restrictions, $\Sigma_0, \dots, \Sigma_n$, we can examine how a policy intended for the model captured in one language can be interpreted to be used with the model captured in the other. The result is similar to transferring policies between the model, $\mathbb{M}$, and a restricted view of the model, $\mathbb{M}|_{\Sigma}$.

π_{Σ_i} for use with $\mathbb{M}|_{\Sigma_j}$

The definition of the restricted chain means that the interpretation of a policy, π_{Σ_i}, for use with a problem, $\mathbf{P} = \langle \mathbb{M}|_{\Sigma_j}, s_{init}, g\rangle$, is straightforward. We can guarantee from the definition of restricted chain that for every state in wff(Σ_i) there is at least one state in wff(Σ_j) that is equivalent or an enrichment of it. Also, for every action, a, in wff(Σ_i) and states, s and s', such that $s' = \gamma_{\Sigma_i}(s, a)$, there is an action in wff(Σ_j) that transitions from a state equivalent or enriching s to a state equivalent or enriching s'.

As the states of $\mathbb{M}|_{\Sigma_i}$ express all of $\mathbb{M}$ that is expressible in Σ_i then there is at most one state corresponding to the initial state, s_{init}. As Σ_i is usable for $\mathbb{M}$ then there is precisely one state, s_0.

We can map this state to an action, $a_1 = \pi_{\Sigma_i}(s_0)$ and find the following state, $s_1 = \gamma_{\Sigma_i}(s_0, a_1)$. The corresponding actions in the richer model are defined by the *represented action set* ($\mathbf{RAS}$). The names are overloaded from definitions above.

$$\begin{aligned}
&\mathbf{RAS}_{\Sigma_0 \to \Sigma_1}(s'_0(\Sigma_1)) = \{a'(\Sigma_1)|\\
&\quad \exists s, a \in \text{wff}(\Sigma_0)\\
&\quad\quad s'_0 \mathbf{R} s_0 \;.\; a = \pi_{\Sigma_0}(s_0) \;.\; a'\mathbf{R}a \;.\; (\exists s'_1 \in \text{wff}(\Sigma_1)\; s'_1 = \gamma(s'_0, a'))\}
\end{aligned}$$

Any deterministic selection process, f, can be used to select an action from this set. And we define the policy for Σ_i as: $\pi_{\Sigma_j}(s) = f(\mathbf{RAS}_{\Sigma_i \to \Sigma_j}(s))$.

π_{Σ_j} for use with $\mathbb{M}|_{\Sigma_i}$

The process of interpreting a policy, π_{Σ_j}, for language, Σ_j, as a policy to be used with the problem, $\mathbf{P} = \langle \mathbb{M}|_{\Sigma_i}, s_{init}, g\rangle$, is more involved, as the policy may map to enhanced actions and have no corresponding action in Σ_i. For a state, s, in Σ_i there might be several (at least one) states that are represented by s in Σ_j. We can define the *abstract action set* ($\mathbf{AAS}$) of actions, using the policy to map each of these states:

$$\mathbf{AAS}_{\Sigma_1 \to \Sigma_0}(s) = \{a'(\Sigma_1)|\; \exists s' \in \text{wff}(\Sigma_1)\; (a' = \pi_{\Sigma_1}(s') \;.\; s'\mathbf{R}s)\}$$

There are two possibilities for each action, a, in the abstract action set: either there is an action that represents a in wff(Σ_i), or there is not.

These actions that make an observable change in the state of $\mathbb{M}|_{\Sigma_i}$, are defined in the *effecting action set* ($\mathbf{EAS}$):

$$\mathbf{EAS}_{\Sigma_1 \to \Sigma_0}(s) = \{a' \in \mathbf{AAS}_{\Sigma_1 \to \Sigma_0}(s) | \exists s' \; (s'\mathbf{R}s \; . \; \gamma_{\Sigma_1}(s', a')\not\mathbf{R}s)\}$$

Similar to above we then define the *base language action set* ($\mathbf{BAS}$), as the actions in wff(Σ) that represent actions in $\mathbf{EAS}_{\Sigma_j \to \Sigma_i}(s)$.

$$\mathbf{BAS}_{\Sigma_1 \to \Sigma_0}(s) = \{a(\Sigma_0) | \; \exists a' \in \mathbf{EAS}_{\Sigma_1 \to \Sigma_0}(s) \; a'\mathbf{R}a\}$$

The result is exactly the same as for interpreting a policy for $\mathbb{M}$, as a policy for $\mathbb{M}|_{\Sigma}$: this set can be empty and there can be loops in the interpreted policy.

I.3.3 Co-execution

In this work we follow a chain of language restrictions from the presented language, Σ_0, to a richer language, Σ_i. This means that we have two models of the problem: $\mathbb{M}|_{\Sigma_0}$ and $\mathbb{M}|_{\Sigma_i}$. If we assume that the state of both models are known then actions can be followed in both models.

Co-execution is the process of executing a policy, π_{Σ_i}, in $\mathbb{M}|_{\Sigma_i}$ and concurrently interpreting the policy as a policy for $\mathbb{M}|_{\Sigma_0}$ and executing the interpreted policy in $\mathbb{M}|_{\Sigma_0}$. If we assume that we co-execute the policies from the current state to the goal then we can interpret a policy for use with $\mathbb{M}|_{\Sigma_i}$ as a complete collection of plans for $\mathbb{M}|_{\Sigma_0}$.

Co-executing $\pi_{\Sigma_i}(s_0)$ in $\mathbb{M}|_{\Sigma_0}$ and $\mathbb{M}|_{\Sigma_i}$

We assume a state, s_0, in the wff(Σ_0) and a state, s'_0, in the wff(Σ_i) as the current state of the models. Previously, the process for mapping state, s_0, began by finding all of the states that were represented by s_0. We can focus on the single action mapped to by $a' = \pi_{\Sigma_i}(s'_0)$. a' can be applied to s'_0 resulting in a new enhanced state, s'_1. There are two possibilities with regards s'_1: it will be represented by s_0 in the restricted model, or it will be represented by another state s_1. In the former case, we do not attempt to translate this as an action, instead we progress $\mathbb{M}|_{\Sigma_i}$ to the new state, s'_1, and use the policy again. This process is guaranteed to lead eventually to a state that is represented by a different state in $\mathbb{M}|_{\Sigma_0}$.

The next state is represented by a state, s_1, in $\mathbb{M}|_{\Sigma_0}$. The languages are related through the chain of language restrictions, this means that there are two possibilities for the relationship between s_0 and s_1. There is either a single action or a sequence of actions (corresponding to an abstracting action) that transitions from s_0 to s_1. For the action, a', mapped to by π_{Σ_i}, we can now define the *action sequence set* (**ASeS**), composed of action sequences that represent this action in Σ_0.

$$\mathbf{ASeS}_{\Sigma_1 \to \Sigma_0} = \{a_1, \ldots, a_n | \ a'\mathbf{R}a_1, \ldots, a_n\}$$

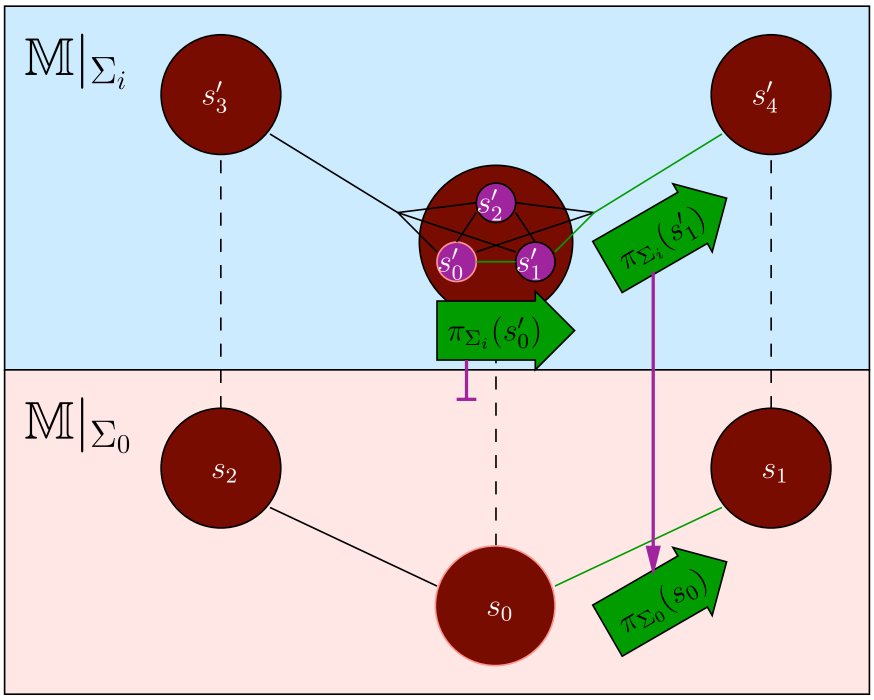


Figure I.5: The process of co-execution relies on the current state of both models being known. There can be several transitions that have no effect at the restricted level, however, progress the enhanced model's state. The enhanced model can translate the action that it applies into an action in the restricted language.

Any of these sequences can be applied; however, it makes sense to select a decision process that minimises the number of actions used to move between the states. Figure I.5 illustrates co-execution for an enriched action.

Co-execution guides $\mathbb{M}|_{\Sigma_0}$ out of loops

We return to consider the previous looping behaviour. Loops occur when a sequence of actions, $a'_1, \ldots, a'_n$, transition the state from s'_0 to s'_1 in $\mathbb{M}|_{\Sigma_i}$ and there is a state, s, in wff(Σ_0) that represents both s'_0 and s'_1. In this case, the part of the state that has been changed cannot be expressed in Σ_0. This is interpreted as a sequence of actions

$a_1, \ldots, a_m$ that represent $a'_1, \ldots, a'_n$ and have no effect on the state of $\mathbb{M}|_{\Sigma_0}$ and so cause a looping behaviour.

When the policy is being co-executed the state, s', of $\mathbb{M}|_{\Sigma_i}$ is known. As the loop is repeating in $\mathbb{M}|_{\Sigma_0}$, the state of $\mathbb{M}|_{\Sigma_i}$ is changing. Eventually, π_{Σ_i} will guide the executive to a state, s'_2, that is not represented by any of the states in the loop and $\mathbb{M}|_{\Sigma_0}$ will safely exit the loop. This is guaranteed as π_{Σ_i} guides an executive to the goal and the chain of language restrictions ensures that the goal cannot be represented by a state in the loop or the problem would be solved on the first loop.

π_{Σ_0}: a complete collection of plans

The previous description of execution fails to define a policy. In fact, we can only interpret π_{Σ_i} as a collection of plans for $\mathbb{M}|_{\Sigma_0}$ if we add some constraints for how it will be used. There are three issues concerning the interpretation. The first is that there can be loops, as we have discussed earlier. The second is that the sequences selected to represent abstracting actions can overlap. In both of these situations a different action may be applied in a state that has already been visited. The third issue is that actions can be represented by action sequences. These three properties violate the policy definition.

We have shown that co-execution will eventually lead the executive out of a loop, however, this relies on different actions being applied for the same state. For a loop in $\mathbb{M}|_{\Sigma_0}$, (s, a_0), (s_1, a_1),...,(s_n, a_n),(s, a_{n+1}), it would be convenient to define the mapping $a_{n+1} = \pi_{\Sigma_0}(s)$. This means that the loop is entirely missed out and therefore does not cause the policy interpretation to map to different actions. Of course this requires loop detection and plan manipulation. However, if using a simple executive then execution is guaranteed to exit loops and this aspect will not prevent the policy solving a problem, but this does not define a policy.

Secondly we consider the sequences of actions that represent abstracting actions. For a chosen sequence, $a_1, \ldots, a_n$, from the set $\mathbf{ASeS}_{\Sigma_1 \to \Sigma_0}$, and current state, s_0, we can store the set of pairs: $\mathbf{StepCache} = \{(s_0, a_1), \ldots, (s_{n-1}, a_n)\}$, such that $s_i = \gamma_{\Sigma_0}(s_{i-1}, a_i)$. We can now define $a = \pi_{\Sigma_0}(s)$, where $(s, a) \in \mathbf{StepCache}$. If the mapping is not defined then we are guaranteed to have completed the sequence (and s will represent the current state of $\mathbb{M}|_{\Sigma_i}$) and we should look up the next action in π_{Σ_i}. Intuitively, this says that the sequence of actions needs to be applied before co-execution can continue, so we break it up and feed the actions to the executive one at a time. This idea relies entirely on the assumption that co-execution traverses from the current states to the goal using only the actions of the policy or the interpreted plan.

Under the presented method of co-execution, for any state, s, the interpretation provides a plan, π_{Σ_0}, that leads an executive to the goal. However, for two plans, $\pi_{\Sigma_{0_{s_0}}}$ and $\pi_{\Sigma_{0_{s_1}}}$ for states, s_0 and s_1, we cannot guarantee that for every state, s, $\pi_{\Sigma_{0_{s_0}}}(s) = \pi_{\Sigma_{0_{s_1}}}(s)$. This is because we might be following sequences of actions that represent different abstracting actions and therefore pass through the state s as part of sequences to different states. This means that our interpretation provides a complete collection of plans but is not a complete policy for $\mathbb{M}|_{\Sigma_0}$. This is a strong property that is sufficient for the purposes of this work.

I.4 Proofs for state and action representation

Theorem 3.2.1 *For any state, s, expressible in Σ_i, there is a single state, s', expressible in Σ_0, which represents s ($s\mathbf{R}s'$).*

Proof This restriction follows from the limited space of enhancements we presented in Section 3.1. This can be demonstrated through an inductive step, by considering an enhancement between languages, Σ_j and Σ_{j+1}, in the case of each of the three possible enhancement steps. The base case is trivial as the chain starts at Σ_0 and all states expressible in Σ_0 can be represented by themselves in Σ_0. The assumption is that every state that is expressible in Σ_j is represented by a state expressible in Σ_0. The added formulae in Σ_{j+1} must belong to one of three cases:

- An abstract action, expressible in Σ_{j+1}, is composed of actions expressible in Σ_j and can therefore only link states expressible in Σ_j and from the inductive assumption, these must be represented by states in Σ_0.

- An enrichment step leads to states in Σ_{j+1} that are enrichments of states in Σ_j: that is, each state, $s \in \text{wff}(\Sigma_{j+1})$ is equivalent to a state $s' \in \text{wff}(\Sigma_j)$ for all of the propositions expressible in Σ_j. If s' is represented by a state, $s'' \in \text{wff}(\Sigma_0)$ then s is represented by s''; and through the inductive assumption this is the case.

- The final case is similar, except there might be a set of states, $s_0, \ldots, s_m \in \text{wff}(\Sigma_{j+1})$, represented by $s' \in \text{wff}(\Sigma_j)$. In this case each state matches in all propositions expressible in Σ_j and each alternative s_k differs in the possible decisions. Also there are actions; however, these actions affect the decision propositions and therefore do not alter the state, with respect to a lower language (that is, the changed propositions are not part of the language).

Theorem 3.2.2 *For any action, $a \in wff(\Sigma_i)$, there exists an (possibly empty) action sequence, $a'_0, \ldots, a'_m \in wff(\Sigma_0)$, which represents a ($a\mathbf{R}a'_0, \ldots, a'_m$).*

Proof We assume that every action, a, and state pair, s_0, s_1, that is expressible in Σ_j is represented by an action sequence, $a'_0, \ldots, a'_m \in \text{wff}(\Sigma_0)$, such that $s'_1 = a'_0, \ldots, a'_m(s'_0)$. The base case holds as all actions in Σ_0 are represented by themselves in Σ_0. The added formulae in Σ_{j+1} must belong to one of three cases:

- An abstract action is equivalent to a sequence of actions, $a''_0, \ldots, a''_k$, in Σ_j and each of these is represented by a sequence of actions in Σ_0 (from the assumption). Therefore the composition of each of these action sequences represents the action.
- Enriching steps preserve the actions set and therefore all actions in Σ_{j+1} are in Σ_j and therefore represented by an action sequence in Σ_0.
- The decision actions' effects involve only propositions not expressible in Σ_j. Therefore, decision actions are represented by a NO-OP (empty sequence) in Σ_j.

I.5 Setting related work within our framework

In Chapter 3 and Appendix I, we have developed a framework for enhancing the problem model. The models explored within this framework share properties with the described problem model, including action-sequence transferral under co-execution. In order to secure this property we have made several restrictive assumptions over the chains of languages that can be explored in the framework. In this subsection we consider approaches that have exploited an enhanced problem model either as part of a planner's strategy, or as the formalism used for planning. In the latter case, we relate the examples from the literature to our model.

I.5.1 Remodelling as a planning approach

Remodelling is a natural strategy for tackling a problem and it has formed a part of many strategies to solving the planning problem. A common approach in problem solving is to construct a relaxed model of the problem and use a solution to the relaxed version of the problem to guide actions taken in the original problem. It can follow that by selecting the relaxation in a sensible way, its solutions might provide useful guidance for the original problem. This approach is part of the strategy adopted in the state of the art domain independent approaches to planning (Hoffmann and Nebel, 2001; Richter et al., 2008; Helmert and Domshlak, 2009). The relaxation is computed by removing the delete effects from each actions. The reformulated problems have the benefit that they can be solved efficiently (Bonet and Geffner, 1998) and it has been demonstrated that they are informative for heuristic search (Hoffmann and Nebel, 2001; Richter et al., 2008; Helmert and Domshlak, 2009). More generally, a collection of relaxed models can be made for the same problem and their solutions can be used in combination to provide an estimate for the original model (Culberson and Schaeffer, 1998). An approach that is related to our framework is presented in Gregory et al. (2011). A hierarchy of abstractions of the model is constructed and information found solving problems higher in the hierarchy is used to inform the search in lower hierarchies. In these approaches the planner is presented with the original planning problem and the solving strategy involves some form of remodelling.

I.5.2 Macro actions

A macro action, a, is a collection of actions, $a_0, \ldots, a_m$ with the interpretation $s' = \gamma(s, a) \iff s' = \gamma(\ldots(\gamma(s, a_0), \ldots), a_m)$, or in other words, the action represents an action sequence. Macro actions can be constructed and provided to the planner as an alternative to the actions in the problem model. These actions are equivalent to abstracting actions in our model. The process of enhancing the problem model with macro actions corresponds to moving along a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$, with the added restriction that the enhancements can only add abstracting actions:

Definition I.5.1

$$\begin{aligned}
&\Sigma_0, \ldots, \Sigma_n \textit{ are a chain of abstraction enhancing languages } \iff \\
&\quad \Sigma_0, \ldots, \Sigma_n \textit{ are a chain of language restrictions} \\
&\quad \forall i \in [0, \ldots, n-1] \\
&\quad\quad \forall a \in \textit{wff}(\Sigma_{i+1}) \\
&\quad\quad\quad (a \in \textit{wff}(\Sigma_i) \vee \textit{AbstractingAction}_{\Sigma_i \rightarrow \Sigma_{i+1}}(a))
\end{aligned}$$

The nature of heuristics means that it is sometimes difficult to evaluate the relative quality of the surrounding states. For example, a state on a plateau, or at a local minimum in a heuristic landscape. This means that a planner cannot rely exclusively on a hill-climbing or greedy algorithm to solve planning problems with heuristic search. One solution to reducing the limitations of the heuristics is to include macro actions in the problem model (e.g. Coles and Smith, 2007). If the actions are constructed sensibly then they can provide a single choice that escapes the plateau in the heuristic landscape (Iba, 1989; Coles and Smith, 2007; Lindsay, 2012). These actions raise the level of planning and allow the planner to step over states where the relative reward cannot be determined. This work takes a more general look at determining the appropriate level for making decisions.

Adding macro actions to the problem model will usually increase the branching factor of the state space, causing an increase in utility cost and more states to examine in each layer. If the actions are helpful then search will have to expand fewer layers as the macro actions will reach the goal using fewer actions. The difficulty in using macros is to select a small number of macro actions that lead to improved performance (Botea et al., 2005a; Newton et al., 2007). Performance improvements will often be a reduction in the time taken to find a solution. In CONSTANCE, macro actions are used to reduce the plan space in a constraints model, by reducing the interweaving of causally unrelated action sequences and abstracting from the specific actions used to transition between states. These reductions lead to a more effective constraint based planner (Gregory et al., 2010). Efficiency of the planner is not the main issue that we address in this work, although matters of efficiency are inevitably important when considering a hard problem like planning.

Pre-process and on-line model extension

In Botea et al. (2005a); Newton et al. (2007) the enhanced view $\mathbb{M}|_{\Sigma_i}$ is constructed upfront and the planner solves the planning problem expressed for the enhanced model. The plan is translated into a solution for $\mathbb{M}|_{\Sigma_0}$ as a post-process. In general, a macro action, a, such that $s' = \gamma_{\Sigma_i}(s, a)$, can be substituted for any sequence, $a_0, \dots, a_m$, such that $s' = \gamma_{\Sigma_0}(\dots(\gamma(s, a_0)\dots), a_m)$.

Another approach to using macro actions is to generate them during planning (Coles and Smith, 2007; Laird et al., 1986; Iba, 1989). We can interpret this as moving through a chain of language restrictions during the planning process. The planner begins with the model captured in Σ_0. During planning a series of models are presented to the planner: the restricted views corresponding to the languages on a chain of language restrictions, $\Sigma_0, \dots, \Sigma_i, \dots$. At a certain point during planning the planner will use the model $\mathbb{M}|_{\Sigma_j}$ to choose the next action to apply. It is limited to the actions that can be expressed in Σ_j. The constructed plan is a valid plan for $\mathbb{M}|_{\Sigma_i}$ as a chain of abstraction enhancements has been followed and any action in Σ_j is in Σ_i. This means that we can translate the solution for $\mathbb{M}|_{\Sigma_i}$, as a solution for $\mathbb{M}|_{\Sigma_0}$ as before.

The use of a post-process to translate the plan for the original problem model is an alternative to our process of co-execution. One advantage of co-execution is the states of both transition systems are known during planning. However, when using macro actions, the chains only enhance the model with abstracting actions, and therefore the models share the same states. As we observed in the previous section, a plan in the enhanced model is not guaranteed to translate to a plan in the base language.

There are two methods of presenting a planner with macro actions. One approach is to compose the actions and include the macros in the problem model that is presented to the planner. The planner uses the actions in the same way as standard actions and a plan in the original language is computed as a post-process. The main benefit to this approach is that the benefits of macro actions can be realised in any planner. The other approach is to build a specialised component into the planner for dealing with the macro actions. In Botea et al. (2005a) it is demonstrated that extending the planner to handle abstracting actions directly can result in a more efficient and effective system. In our approach we rely on a policy and therefore whether the action is a macro action or not is determined by the policy mapping.

I.5.3 Support predicates

The propositions modelled in the state have limited impact on relaxed plan based planners. However, for planners that rely on expressions built from the state and goal formulae, the modelled propositions can be crucial. This observation has already inspired researchers to investigate enhancing the problem model (Khardon, 1999a; Martin and Geffner, 2000; Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001). These approaches to enhancing the model correspond to moving through a chain of language restrictions, $\Sigma_0, \ldots, \Sigma_n$. The main motivation for these approaches came from planners using fully hand-written control knowledge (Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001). This has inspired the community to approach automatically generating the control knowledge. There has been substantial progress in this area (Fern et al., 2006): through development of the language used to express the control knowledge (Martin and Geffner, 2000), discussed in Chapter 4; and improved rule learning strategies (Fern et al., 2006), discussed in Chapter 8. The richer languages that have been developed do not establish comprehensible abstraction layers, resulting in control knowledge that is complicated to understand; however, they allow concepts to be constructed over the described model. We divide the approaches into those that enrich the states (through hand or automatically determined enrichments) and those that enhance the entire model.

Enriching the states

In Khardon (1999a); Martin and Geffner (2000); de la Rosa and McIlraith (2011) the chains considered are restricted: the authors investigate enriching the states, but do not add actions into the model. Instead, richer languages model more propositions. In particular, there is a one-to-one mapping between the states in the described model, $\mathbb{M}|_{\Sigma_0}$, and the states in any of the richer models on the chain, $\mathbb{M}|_{\Sigma_i}$. This is because in these works there are no enriched actions to make fine grained decisions.

Definition I.5.2

$$\begin{aligned}
&\Sigma_0, \ldots, \Sigma_n \textit{ are a chain of state enriching languages } \iff \\
&\quad \Sigma_0, \ldots, \Sigma_n \textit{ are a chain of language restrictions} \\
&\quad \forall i \in [0, \ldots, n-1] \\
&\qquad a \in \textit{wff}(\Sigma_{i+1})\ (a \in \textit{wff}(\Sigma_i)) \\
&\qquad \forall s_0, s_1 \in \textit{wff}(\Sigma_{i+1})\ (\exists s_2 \in \textit{wff}(\Sigma_i)\ s_0 \models s_2\ .\ s_1 \models s_2) \iff s_0 = s_1
\end{aligned}$$

The planner is provided with the same actions as in $\mathbb{M}|_{\Sigma_0}$. This means that the planner generates a plan that is usable for $\mathbb{M}|_{\Sigma_0}$ without translation.

In these works they demonstrate that learning control knowledge relies on richer propositions than are expressed in the described planning models. The authors observe that in certain problems they cannot learn control knowledge that makes effective action choices. After moving through a chain of state enriching languages, the authors demonstrate that once certain propositions are modelled then effective control knowledge can be learned. The key to this is that the states are enriched with propositions that provide the necessary information to make action choices. These works are particularly relevant, as a key motivation for this work is that the planning model does not always express all of the necessary propositions useful for selecting actions.

Enhancing the model

In Bacchus and Kabanza (2000); Doherty and Kvarnström (2001) it is demonstrated that enhancing the model allows effective control knowledge to be expressed. One of the key properties of these systems is that very little search is required, because the control knowledge heavily constrains every choice. The control knowledge used by TLPLAN establishes various levels of abstraction so that decisions can be made at the correct level. A lot of effort goes in to hand-writing the control knowledge, as it is important that the control knowledge covers each of the decisions in problems of the domain. Moreover, a rich knowledge representation language allows knowledge to be represented for the various styles of benchmark planning problem. However, control knowledge can be used to generate a high-quality plan very efficiently (Bacchus and Kabanza, 2000; Doherty and Kvarnström, 2001). We developed a general framework for exploring similar forms of abstraction: the enrichments made in these works can be compared to moving along a chain of language restrictions.

I.5.4 Extra world concepts

In the previous subsections we have presented processes that can be explained with a chain of language restrictions. In Dornhege et al. (2009); Gregory et al. (2012), chains of language restrictions that do not conform to the shape preserving property are investigated. A key difference in these works is that the solution to the problem is expressed for the enhanced model. This means that no interpretation is required.

Definition I.5.3

The languages, $\Sigma_0, \ldots, \Sigma_n$, are a chain of model augmenting
language restrictions of the model $\mathbb{M}$, if:
$\mathbb{M}|_{\Sigma_0}, \ldots, \mathbb{M}|_{\Sigma_n}$ *are restricted views of* $\mathbb{M}$
Σ_0 *is usable for* $\mathbb{M}$
for $i = 1, \ldots, n-1$

$$(\exists a\, \mathbb{M}|_{\Sigma_{i+1}} \models a \,.\, \mathbb{M}|_{\Sigma_i} \not\models a) \vee (\exists p\, \mathbb{M}|_{\Sigma_i} \not\models p \,.\, \mathbb{M}|_{\Sigma_{i+1}} \models p)$$

The motivation for these works is that there are parts of the planning problem that are difficult to express in a standard language. This contrasts with our motivation: that there are parts of the *planner* that are hard to express in a standard language.

The authors extend the model using specialised language and demonstrate the use of heuristic search to solve problems expressed in the richer language.

I.5.5 Decomposition

Breaking a problem into smaller chunks and constructing a solution to the original problem from solutions to the smaller parts is another traditional approach to problem solving. We look at several approaches to planning that decompose the problem in different ways.

Goal ordering is an approach that decomposes the problem by splitting the problem goals into a chain of growing goal subsets. The planner is presented each sub-problem in progression, starting from the final state of the previous run. This decomposition can lead to an exponential complexity reduction (Koehler, 1998). More generally, landmarks are facts that must be true at some point in a plan (Porteous et al., 2001). Various relationships have been derived that imply orderings between pairs of landmarks. An approach to planning is to use landmarks as intermediate goals that the planner must solve before solving the goal. In Hoffmann et al. (2004) it is demonstrated that several problems are broken into easier sub-problems that can be solved quickly by a planner and that this combined solution can be shorter and found faster. However, as some of the used landmark orderings are only approximate and because in the over-head of multiple planner calls, the use of landmarks can also result in worse performance. Each of these approaches is compatible with this work and could be used to improve the performance.

Planning problems can contain challenging optimisation problems as sub-components. In HybridSTAN (Fox and Long, 2001) the problem is decomposed so that appropriate special purpose solutions can be exploited on a collection of these challenging components. At the core is a reduced problem model (the core problem) that is solved by a general purpose heuristic planner. In comparison, this is similar to moving backwards along a chain from a richer language to a less expressive language. A key benefit to this approach is that by exploiting tailored solutions, the planning problem is simpler.

Definition I.5.4

The languages $\Sigma_0, \ldots, \Sigma_{-n}$, are a chain of model simplifying restrictions of $\mathbb{M}$, if:
Σ_{-n} is usable for $\mathbb{M}$
for $i = 1, \ldots, n-1$

$$(\exists a\, \mathbb{M}|_{\Sigma_{i+1}} \models a \,.\, \mathbb{M}|_{\Sigma_i} \not\models a) \vee (\exists p\, \mathbb{M}|_{\Sigma_i} \not\models p \,.\, \mathbb{M}|_{\Sigma_{i+1}} \models p)$$

The core problem is not necessarily shape preserving and the decomposition does not just remove enriching or abstracting actions. For example, when a transportation sub-problem is identified, the transporter's located propositions are removed. In the described problem the transporter is located so that it can pick up a subset of the packages. However, in the core problems the transporter can pick up any package. These actions are not represented by actions in the described problem.

Another way of interpreting this work is as a form of reverse engineering of Dornhege et al. (2009), where the components that should have been defined in a specialised language have been forced into PDDL. In Fox and Long (2001) these components are identified and removed for specialised treatment. The model is then moved through a chain of model augmenting language restrictions and a plan is generated for the enhanced (complete) model. Fox and Long (2001); Dornhege et al. (2009) both use heuristic estimates as the means of communication between the special purpose solvers and the general planner.

An aspect of the approach developed in Fox and Long (2001) is that the components have to separate cleanly from the rest of the model. A fingerprint defines the exact properties that a problem model must exhibit so that a particular special purpose solver is applicable. As soon as the model is slightly different then the problem cannot be decomposed and the specialised solution cannot be applied. In this work we investigate similar specialised solvers; however, instead of decomposing the structures from the model, we enhance the model with information about the structure. In this way, if the structure is not exactly as expected the information could still be provided, although we do not experiment with this in the current work.

BIBLIOGRAPHY

Homepage of IPP, February 1999. URL `http://user.enterpriselab.ch/~takoehle/publications/ipp/ipp.html`.

The 4th international planning competition, 2004. URL `http://ipc.icaps-conference.org`.

The international planning competition, 2014. URL `http://ipc.icaps-conference.org`.

Philip E. Agre and David Chapman. Pengi: an implementation of a theory of activity. In *Proceedings of the sixth National conference on Artificial intelligence*, volume 1, pages 268–272. AAAI Press, 1987. ISBN 0-934613-42-7.

Martin Aigner. *Combinatorial search*. Wiley-Teubner series in computer science. Wiley-Teubner, New York, 1988. ISBN 9783519021094.

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Evolving heuristics for planning. In *Proceedings of Evolutionary Programming VII*, pages 745–754. Springer, 1998.

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Knowledge representation issues in control knowledge learning. In Pat Langley, editor, *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 1–8, Standford, CA, June-July 2000a.

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. GP fitness functions to evolve heuristics for planning. In Martin Middendorf, editor, *Evolutionary Methods for AI Planning*, pages 189–195, Las Vegas, NV (USA), July 2000b.

Ricardo Aler, Daniel Borrajo, and Pedro Isasi. Learning to solve planning problems efficiently by means of genetic programming. *Evolutionary Computation*, 9(4):387–420, 2001.

Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. A parametric hierarchical planner for experimenting abstraction techniques. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, pages 936–941, 2003.

Giuliano Armano, Giancarlo Cherchi, and Eloisa Vargiu. DHG: A system for generating macro-operators from static domain analysis. In *Artificial Intelligence and Applications*, pages 18–23, 2005.

Fahiem Bacchus. The 1st international planning competition, 2000. URL `http://www.cs.toronto.edu/aips2000/`.

Fahiem Bacchus. The AIPS '00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

Fahiem Bacchus and Michael Ady. TLPlan / HPlan-P documentation, 2003. URL `http://www.cs.toronto.edu/tlplan/docs.shtml`.

Fahiem Bacchus and Froduald Kabanza. Using temporal logics to express search control knowledge for planning. *Artificial Intelligence*, 116:123–191, 2000.

Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11:625–655, 1993.

Alex Bavelas. Communication patterns in task-oriented groups. *Journal of the Acoustical Society of America*, 1950.

Avrim Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90(1-2):281–300, 1997.

Avrim L. Blum and Pat Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97:245–271, 1997.

Blai Bonet and Hector Geffner. HSP: Heuristic search planner. In *AIPS-98 Planning Competition*, 1998.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Using abstraction for planning in sokoban. In Jonathan Schaeffer, Martin Müller, and Yngvi Björnsson, editors, *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science*, pages 360–375. Springer Berlin Heidelberg, 2003. ISBN 978-3-540-20545-6.

Adi Botea, Markus Enzenberger, Martin Müller, and Jonathan Schaeffer. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research (JAIR)*, 24:581–621, 2005a.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Learning partial-order macros from solutions. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 231–240, 2005b.

Adi Botea, Martin Müller, and Jonathan Schaeffer. Fast planning with iterative macros. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 1828–1833, 2007.

Craig Boutilier, Richard Dearden, and Moises Goldszmidt. Stochastic dynamic programming with factored representations. *Artificial Intelligence*, 121:2000, 1999.

Craig Boutilier, Raymond Reiter, and Bob Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 690–700. Morgan Kaufmann, 2001.

Tom Bylander. Complexity results for serial decomposability. In *Proceedings of the tenth national conference on Artificial intelligence*, AAAI'92, pages 729–734. AAAI Press, 1992. ISBN 0-262-51063-4.

Jaime Carbonell, Oren Etzioni, Yolanda Gil, Robert Joseph, Craig Knoblock, Steve Minton, and Manuela Veloso. PRODIGY: an integrated architecture for planning and learning. *SIGART Bulletin*, 2(4):51–55, July 1991. doi: 10.1145/122344.122353.

Andrew. I. Coles and Amanda. J. Smith. Generic types and their use in improving the quality of search heuristics. In *Proceedings of the 25th Workshop of the UK Planning and Scheduling Special Interest Group*, PlanSIG 2006, December 2006.

Andrew. I. Coles and Amanda. J. Smith. Marvin: A heuristic search planner with online macro-action learning. *Journal of Artificial Intelligence Research*, 28:119–156, February 2007. ISSN 11076-9757.

Andrew. I. Coles and Amanda. J. Smith. Upwards: The role of analysis in cost optimal SAS+ planning. ICP 2008, Booklet on participating planners, International Conference on Automated Planning and Scheduling, September 2008.

Andrew I. Coles, Maria Fox, Derek Long, and Amanda J. Smith. Teaching forward-chaining planning with javaff. In *Colloquium on Artifical Intelligence Education, Twenty-Third AAAI Conference on Artificial Intelligence*, July 2008.

Stephen Cresswell and Peter Gregory. Generalised domain model acquisition from action traces. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 42–49, June 2011.

Stephen N. Cresswell, Thomas Leo McCluskey, and Margaret M. West. Acquisition of object-centred domain models from planning examples. In Alfonso Gerevini, Adele Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 338–341. AAAI Press, September 2009.

Matt Crosby, Michael Rovatsos, and Ronald P. A. Petrick. Automated agent decomposition for classical planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pages 46–54, June 2013.

Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. In *Computational Intelligence*, 1998.

Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

Tomás de la Rosa and Sheila A. McIlraith. Learning Domain Control Knowledge for TLPlan and Beyond. In *Proceedings of the International Conference on Automated Planning and Scheduling, Workshop on Planning and Learning (PAL)*, 2011.

Tomás de la Rosa, Sergio Jiménez, and Daniel Borrajo. Learning relational decision trees for guiding heuristic planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*. AAAI Press, 2008.

Tomas de la Rosa, Sergio Jiménez, Raquel Fuentetaja, and Daniel Borrajo. Scaling up heuristic planning with relational decision trees. *Journal of Artificial Intelligence Research*, 40:767, 2011.

Thomas Dean and Michael Wellman. *Planning and Control*. Morgan Kaufmann, 1991.

Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

Anthony H. Dekker and Bernard D. Colbert. Network robustness and graph topology. In *Proceedings of the 27th Australasian Conference on Computer Science*, volume 26 of *ACSC '04*, pages 359–368. Australian Computer Society, Inc., 2004.

Patrick Doherty and Jonas Kvarnström. TALplanner: A temporal logic based planner. *AI Magazine*, 22(3):95–102, 2001.

Christian Dornhege, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel. Semantic attachments for domain-independent planning systems. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

Filip Dvorak and Roman Barták. Integrating time and resources into planning. In *Proceedings of the 22nd International Conference on Tools with Artificial Intelligence*, volume 2, pages 71–78. IEEE, 2010.

Stefan Edelkamp and Malte Helmert. MIPS: The model-checking integrated planning system. *AI Magazine*, 22(3):67–72, 2001.

Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of IPC-4. *Proceedings of the International Planning Competition. International Conference on Automated Planning and Scheduling*, 2004.

Kutluhan Erol, James Hendler, and Dana S. Nau. Complexity results for HTN planning. *Annals of Mathematics and Artificial Intelligence*, 18(1):69–93, 1996. ISSN 1012-2443.

Patrick Fabiani and Yannick Meiller. Planning with tokens: an approach between satisfaction and optimisation. In *European Conference on Artificial Intelligence, Workshop on New Results in Planning, Scheduling and Design*, 2000.

Alan Fern, Sungwook Yoon, and Robert Givan. Learning domain-specific control knowledge from random walks. In *Proceedings of the 14th International Conference on Automated Planning and Scheduling*. AAAI Press, 2004.

Alan Fern, Sungwook Yoon, and Robert Givan. Approximate policy iteration with a policy language bias: Solving relational Markov decision processes. *Journal of Artificial Intelligence Research*, 25:75–118, 2006.

Alan Fern, Roni Khardon, and Prasad Tadepalli. International planning competition, 2008. URL `http://ipc.informatik.uni-freiburg.de/`.

Alan Fern, Roni Khardon, and Prasad Tadepalli. The first learning track of the international planning competition. *Machine Learning*, 84(1-2):81–107, July 2011. ISSN 0885-6125.

Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

Robert W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

Maria Fox and Derek Long. Hierarchical planning using abstraction. In *IEE proceedings on Control Theory and Applications*, volume 142, pages 197–210. IET, Institution of Electrical Engineers, 1995.

Maria Fox and Derek Long. The automatic inference of state variables in TIM. *Journal of Artificial Intelligence Research*, 9:367–421, 1998.

Maria Fox and Derek Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of 16th International Joint Conference on Artifical Intelligence*, pages 956–961, 1999.

Maria Fox and Derek Long. Hybrid STAN: Identifying and managing combinatorial sub-problems in planning. In *Proceedings of 17th International Joint Conference on Artifical Intelligence*, pages 445–452. Morgan Kaufmann Publishers, 2001.

Maria Fox and Derek Long. PDDL2.1: An extension of PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.

Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, pages 35–41, 1977.

Michelle Galea, David Humphreys, John Levine, and Henrick Westerberg. Evolutionary-based learning of generalised policies for AI planning domains. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2009.

Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

Hector Geffner. *Functional strips: a more flexible language for planning and problem solving*, pages 187–209. Kluwer Academic Publishers, 2000. ISBN 0-7923-7224-7.

Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3 - the language of the fifth international planning competition. Technical report, University of Brescia, Italy, 2005.

Alfonso Gerevini and Lenhart Schubert. Inferring state constraints for domain-independent planning. In *Proceedings of AAAI/IAAI*, pages 905–912, 1998.

Alfonso Gerevini, Alessandro Saetti, and Ivan Serina. Planning through stochastic local search and temporal action graphs in LPG. *Journal of Artificial Intelligence Research*, 20:239–290, 2003.

Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann, 2004.

Peter Gregory and Alan Lindsay. The dimensions of driverlog. In *Proceedings of the UK planning and scheduling special interest group*, 2007.

Peter Gregory, Derek Long, and Maria Fox. Constraint based planning with composable substate graphs. In Helder Coelho, Rudi Studer, and Michael Wooldridge, editors, *Proceedings of the 19th European Conference on Artificial Intelligence*, volume 215, pages 453–458. IOS Press, August 2010.

Peter Gregory, Derek Long, Craig McNulty, and Susanne M. Murphy. Exploiting path refinement abstraction in domain transition graphs. In *Proceedings of AAAI*, volume 2, pages 971–976. AAAI, 2011.

Peter Gregory, Derek Long, Maria Fox, and J. Christopher Beck. Planning modulo theories: Extending the planning paradigm. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2012.

Charles Gretton and Sylvie Thiébaux. Exploiting first-order regression in inductive policy selection. In *Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence*, UAI'04, 2004.

Isabelle Guyon. An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3:1157–1182, 2003.

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968. ISSN 0536-1567.

Malte Helmert. On the complexity of planning in transportation domains. In *Proceedings of the 6th European Conference on Planning*, pages 349–360. Springer-Verlag, 2001.

Malte Helmert and Carmel Domshlak. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 2009.

Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In Lubos Brim, Stefan Edelkamp, Erik A. Hansen, and Peter Sanders, editors, *Graph Search Engineering*, number 09491 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2010. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

Jörg Hoffmann. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research*, 24:685–758, 2005.

Jörg Hoffmann. Analyzing search topology without running any search: On the connection between causal graphs and h^+. *Journal of Artificial Intelligence Research*, 41:155–229, 2011.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Jörg Hoffmann, Julie Porteous, and Laura Sebastia. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Chad Hogg, Héctor Muñoz-avila, and Ugur Kuter. HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proceedings of the 23rd Conference on Artificial Intelligence*. AAAI Press, 2008.

Glenn A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3:285–317, 1989. ISSN 0885-6125.

Okhtay Ilghami, Dana S Nau, and Hector Munoz-Avila. Learning to do HTN planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 390–393, 2006.

Sergio Jiménez, Tomás de la Rosa, Susana Fernández, Fernando Fernández, and Daniel Borrajo. A review of machine learning for automated planning. *Knowledge Engineering Review Journal*, 27(04):433–467, 2012.

Subbarao Kambhampati and Sungwook Yoon. Explanation-based learning for planning. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 392–396. Springer, 2010. ISBN 978-0-387-30768-8.

Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1194–1201. AAAI Press, 1996.

Henry A. Kautz and Bart Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 181–189, 1998.

Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999a.

Roni Khardon. Learning action strategies for planning domains. Technical report, University of Edinburgh, 1999b.

Craig A. Knoblock. Learning abstraction hierarchies for problem solving. In *Proceedings of the eighth National conference on Artificial intelligence*, volume 2 of *AAAI'90*, pages 923–928. AAAI Press, 1990. ISBN 0-262-51057-X.

Jana Koehler. Solving complex planning tasks through extraction of subproblems. In *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 62–69. AAAI Press, 1998.

Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1):273–324, 1997.

Jonas Kvarnström and Patrick Doherty. A temporal logic based forward chaining planner. *Annals of Mathematics and Artificial Intelligence*, 30:119–169, 2001.

John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine learning*, 1(1):11–46, March 1986. ISSN 0885-6125.

John Levine and David Humphreys. Learning action strategies for planning domains using genetic programming. In *Proceedings of the 4th European Workshop on Scheduling and Timetabling (EvoSTIM 2003)*, 2003.

Alan Lindsay. Macro actions for structures. In *Proceedings of the UK planning and scheduling special interest group*, 2012.

Alan Lindsay, Maria Fox, and Derek Long. Abstracting chains of reasoning. In *Proceedings of the UK planning and scheduling special interest group*, 2008.

Alan Lindsay, Maria Fox, and Derek Long. Lifting the limitations in a rule-based policy language. In *Proceedings of the 22nd International Florida Artificial Research Society Conference*, 2009.

Nir Lipovetzky and Hector Geffner. Searching for plans with carefully designed probes. In *Proceedings of the 21st International Conference on Automated Planning and Scheduling*, 2011.

Derek Long and Maria Fox. Efficient implementation of the plan graph in STAN. *Journal of Artificial Intelligence Research*, 10:87–115, 1999.

Derek Long and Maria Fox. Automatic synthesis and use of generic types in planning. In *Proceedings of Artificial Intelligence Planning and Scheduling Systems*, pages 196–205, 2000.

Derek Long and Maria Fox. *Planning with Generic Types*, chapter 4, pages 103–138. Exploring Artificial Intelligence in the New Millenium. Morgan Kaufmann, 2002.

Derek Long and Maria Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

Mario Martin and Hector Geffner. Learning generalized policies in planning using concept languages. In *Proceedings of the 7th International Conference of Knowledge Representation and Reasoning*, 2000.

Thomas Leo Mccluskey, Nona E. Richardson, and Rosemary M. Simpson. An interactive method for inducing operator descriptions. In *Proceedings of the Sixth*

International Conference on Artificial Intelligence Planning Systems. AAAI Press, 2002.

Thomas Leo McCluskey, Stephen N. Cresswell, Nona E. Richardson, and Margaret M. West. Automated acquisition of action knowledge. In *Proceedings of the International Conference on Agents and Artificial Intelligence (ICAART)*, pages 93–100, January 2009.

Drew McDermott. The 1998 AI planning systems competition. *Artificial Intelligence magazine*, 21(2):35–56, 2000.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL-the planning domain definition language. Technical report, Yale University, 1998.

Neville Mehta, Prasad Tadepalli, and Alan Fern. Autonomous learning of action models for planning. In *Proceedings of the Neural Information Processing Systems*, pages 2465–2473, 2011.

Steven Minton. Selectively generalizing plans for problem-solving. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, volume 1 of *IJCAI'85*, pages 596–599. Morgan Kaufmann Publishers Inc., 1985. ISBN 0-934613-02-8, 978-0-934-61302-6.

Steven Minton. Quantitative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42(2-3):363–391, March 1990. ISSN 0004-3702.

Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.

Luke Murray. Reuse of control knowledge in planning domains. Technical report, University of Durham, 2002.

Luke Murray, Derek Long, and Maria Fox. Automating the use of control information in planning domains. Technical report, University of Durham, 2003.

Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, J. William Murdoch, Dan Wu, and Fusun Yaman. SHOP2: An HTN planning system. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

Dana Nau, Tsz-Chiu Au, Okhtay Ilghami, Ugur Kuter, Dan Wu, Fusun Yaman, Héctor Munoz-Avila, and J. William Murdoch. Applications of SHOP and SHOP2.

Intelligent Systems, IEEE, 20(2):34–41, March 2005. ISSN 1541-1672. doi: 10.1109/MIS.2005.20.

Allen Newell and Herbert A. Simon. GPS, a program that simulates human thought. In E.A Feigenbaum and J. Feldman, editors, *Computers and Thought*, pages 279–293. McGraw-Hill, New York, 1963.

M. A. Hakim Newton and John Levine. Implicit learning of macro-actions for planning. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*, August 2010.

M. A. Hakim Newton, John Levine, Maria Fox, and Derek Long. Learning macro-actions for arbitrary planners and domains. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 07)*, September 2007.

Ian Parberry. A real-time algorithm for the $(n^2 - 1)$-puzzle. *Information Processing Letters*, 56(1):23 – 28, 1995. ISSN 0020-0190.

J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the 3rd International Conference of Principles of Knowledge Representation and Reasoning*, volume 92, pages 103–114. Morgan Kaufmann, 1992.

Julie Porteous, Laura Sebastia, and Jörg Hoffmann. On the extraction, ordering, and usage of landmarks in planning. In *Recent Advances in AI Planning. 6th European Conference on Planning (ECP'01)*, pages 37–48, 2001.

Silvia Richter, Malte Helmert, and Matthias Westphal. Landmarks revisited. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pages 975–982, 2008.

Ronald L. Rivest. Learning decision lists. *Machine Learning*, 2(3):229–246, 1987.

Paul S. Rosenbloom, John E. Laird, John Mcdermott, Allen Newell, and Edmund Orciuch. R1-soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(5):561–569, September 1985. ISSN 0162-8828.

Earl D. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial intelligence*, 5(2):115–135, 1974.

Laura Sebastia, Eva Onaindia, and Eliseo Marzal. Decomposition of planning problems. *AI Communications*, 19(1):49–81, 2006.

Steven Skiena. *The Algorithm Design Manual (2. ed.)*. Springer, 2008.

David E. Smith and Daniel S. Weld. Conformant graphplan. In *Proceedings of the National Conference on Artificial Intelligence*, number 15, pages 889–896, 1998.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Learning generalized plans using abstract counting. In *Proceedings of AAAI*, pages 991–997, 2008.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. Computing applicability conditions for plans with loops. In *Proceedings of the International Conference on Automated Planning and Scheduling*, pages 161–168, 2010.

Siddharth Srivastava, Neil Immerman, and Shlomo Zilberstein. A new representation and associated algorithms for generalized planning. *Artificial Intelligence*, 175(2): 615–647, 2011.

Manuela Veloso, Jaime Carbonell, Alicia Pérez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7:81–120, 1995.

C. Henrik Westerberg and John Levine. Genplan: Combining genetic programming and planning. In *Proceedings of the 19th Workshop of the UK Planning and Scheduling Special Interest Group*, 2000.

Elly Winner and Manuela Veloso. LoopDISTILL: Learning looping domain-specific planners from example plans. In *Proceedings of the International Conference on Automated Planning and Scheduling, Workshop on Artificial Intelligence Planning and Learning*, 2007.

Yuehua Xu, Sungwook Yoon, and Alan Fern. Discriminative learning of beam-search heuristics for planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 2041–2046, 2007.

Yuehua Xu, Alan Fern, and Sungwook Yoon. Learning weighted rule sets for forward search planning. In *Proceedings of the International Conference on Automated Planning and Scheduling, Workshop on Planning and Learning*, 2009.

Qiang Yang, Rong Pan, and Sinno Jialin Pan. Learning recursive HTN-method structures for planning. In *Workshop on Artificial Intelligence Planning and Learning, Proceedings*, 2007.

Sungwook Yoon, Alan Fern, and Robert Givan. Inductive policy selection for first-order MDPs. In *Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence*, pages 568–576. Morgan Kaufmann, 2002.

Sungwook Yoon, Alan Fern, and Robert Givan. Learning heuristic functions from relaxed plans. In *Proceedings of the International Conference on Automated Planning and Scheduling*, 2006.

Sungwook Yoon, Alan Fern, and Robert Givan. Using learned policies in heuristic-search planning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 7, pages 2047–2052, 2007.

Sungwook Yoon, Alan Fern, and Robert Givan. Learning control knowledge for forward search planning. *Journal of Machine Learning Research*, 9:683–718, June 2008. ISSN 1532-4435.