University of Strathclyde
Department of Computer and Information Sciences

Heuristically Guided Constraint Satisfaction for AI Planning

by
Mark Judge

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

2015

Declaration

Signed:

Date:

Acknowledgements

I would like to express my deep gratitude to Professors Maria Fox and Derek Long for giving me the opportunity to carry out this work. I have benefited greatly from their knowledge and experience.

In particular, I'd like to thank Professor Long for his patient and thoughtful guidance, and for the many useful conversations and suggestions that have allowed me to complete this study.

I'd also like to thank Dr. John Levine and Dr. Alex Coddington. Dr. Levine's teaching aided my understanding of the applications of A.I. Planning. Dr. Coddington's lectures helped me gain a better insight into CSP-based A.I. Planning.

Thanks also to Dr. Peter Gregory for helpful discussions and guidance in the early part of this project. I am also grateful to Dr. Andrew Coles and Dr. Amanda Coles for their excellent teaching.

Many of my colleagues helped along the way, so thanks to all members of the research group, especially Pat, Alastair, Alan, and Wasif.

Publications

The following extended abstracts and publications contain early results and summaries of parts of the work contained in this study.

Chapter 3 / 4:

Mark Judge and Derek Long. Heuristically Guided Constraint Satisfaction for A.I. Planning - in *Proceedings of the 29th Meeting of the UK Planning and Scheduling Special Interest Group*, University of Huddersfield, UK. December, 2011.

Chapter 4:

Mark Judge and Derek Long. A CSP Heuristic for A.I. Planning - in *Proceedings of 20th Automated Reasoning Conference*, School of Computing, University of Dundee, UK. April, 2013.

Chapter 4 / 5:

Mark Judge. Constraint-based Heuristic Guidance for the Solution of Artificial Intelligence Planning Problems - in *Proceedings of International IEEE / EPSRC Workshop on Autonomous Cognitive Robotics*. University of Stirling, UK. March, 2014

Chapter 5:

Mark Judge. Using CSP Meta Variables in AI Planning - in *Proceedings of Joint Automated Reasoning Workshop and Deduktionstreffen*, Vienna Summer of Logic, Vienna, Austria. July 23-24, 2014

Abstract

Standard Planning Domain Definition Language (PDDL) planning problem definitions may be reformulated and solved using alternative techniques. One such paradigm, Constraint Programming (CP), has successfully been used in various planner architectures in recent years.

The efficacy of a given constraint reformulation depends on the encoding method, search technique(s) employed, and the consequent amount of propagation.

Despite advances in this area, constraint based planners have failed to match the performance of other approaches. One reason for this is that the structural information that is implicit in the domain / problem instance description is lost in the reformulation process.

Hence, to achieve better performance, a constraint based planner should have an appropriate encoding and a search procedure informed by the structure of the original problem. This thesis describes a system that aims to improve planner performance by employing such methods.

The planner uses a table-constraint encoding together with a new variable / value ordering heuristic. This ordered, goal oriented guidance reduces the search space and better directs the search focus.

The system also makes use of novel meta variable techniques for goal locking and resource assignment. These improve propagation and prune the search space.

This CP based planning architecture is shown to perform well on a range of problem instances taken from recent International Planning Competitions (IPCs).

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Classical Planning

Planning is a human decision making process which seeks to achieve a given outcome by using a set of predictable operations in sequence. Artificial Intelligence (AI) planning [1] attempts to automate this process. Informally, we can state that automated planning is a sequential decision making technique, the purpose of which is to provide a set of actions that transform a fully specified initial state into a given goal state.

In order to manage the complexity of the real world, AI planning restricts the information described and abstracts much of the unnecessary detail. Classical planning, also known as STRIPS planning [2], requires that the *state space* be finite and fully observable. It is assumed that only specified actions can change a state and that they do so instantaneously, with the resulting state being predictable.

In classical planning, a real-world problem can be expressed as a set of propositional literals. A state is represented by a conjunction of positive literals whilst actions are described in terms of operators that change the value of these literals. Such operator definitions include the action name, a set of preconditions and a set of effects. The preconditions set is a conjunction of literals representing the conditions that must be satisfied before the action can be carried out. Likewise, the effects set contains a list of both positive and negative effects resulting from the execution of the action.

1.1.1 Search

AI planning can be regarded as a search problem. *State-space search* [3] traverses a subset of the state space, choosing actions, in an attempt to find a solution. Each search node represents a world state with the connecting edges equating to transitions between states. Due to the nature of the actions' descriptions, the search for a solution may be conducted in either a forward or a backward direction. The choice of direction can have a major impact on the efficiency of the planning system [4]. Regardless of the direction chosen for state-space search, the resulting solution is a set of actions that is totally ordered.

An alternative approach is *plan-space search* [3]. In this representation a search node is a partially specified plan and the interconnections are operations on plans. Such plan refinement operations include attempts to complete partial plans and adding ordering constraints between actions in a partial plan.

Using either search paradigm without any additional search guidance proves ineffective except on the simplest of problems. The impact of combinatorial explosion[1] means that some form of search assistance, preferably derived from the structure of the original problem, is desirable.

1.1.2 Heuristics

Although it is possible to systematically search the complete search space of a given planning problem, for most interesting problems this approach is not practical. In such cases, where the search space is large, it may be possible to guide the search by constructing a search node scoring function. This *evaluation function* provides estimated information on the relative merits of the nodes reached during search.

The idea behind using an evaluation function or *heuristic* [6] is that, instead of following all possible search paths, priority is given to those paths that appear to be best. That is, the algorithm chooses those paths or nodes that appear to be leading to the solution. It should be noted that such evaluations are estimated [7]. For example, in a simple route finding problem, the "as the crow flies" measurement would guide the search towards the destination. However, this search guidance may fail if, say, it is at some point necessary to travel in the opposite direction in order to join a route that eventually leads to the destination.

Many successful modern planners [8] make use of heuristically guided search, with most employing some form of *relaxation* of the problem to allow construction of the heuristic. A relaxed problem is one in which certain aspects are removed, thereby making a new, less complex problem. Solving the relaxed problem leads to a solution that, in turn, may be used to guide the search for a solution to the original problem.

[1]Classical planning is PSPACE hard [5]

In building the heuristic, there exists a compromise between using an overly relaxed problem, which would be easier to compute, but would less accurately reflect the structure of the original problem; and using a close approximation of the original problem, which would be more difficult and time consuming, but would be a more accurate reflection of the original problem.

1.2 Planning as Constraint Satisfaction

An alternative to using dedicated planning technology is to reformulate the planning problem. This approach allows standard techniques from other areas to be applied to the solution of planning problems. One such approach sees the original planning problem recast as a Constraint Satisfaction Problem (CSP) [9].

The declarative CSP paradigm provides access to enhanced inference or *propagation* machinery and better *pruning* mechanisms [10], techniques not fully exploited by the planning community. In an ideal situation, a solution can be inferred without the use of search. However, for large problems, the process of inferring a solution itself becomes a hard problem, often requiring an exponential number of new constraints. Therefore, various *consistency* [9] methods are used which perform a limited amount of inference. By using these, in combination with search, there is a compromise between the amount of "preprocessing" to limit the search space, and the amount of search in that space.

Even with the use of consistency techniques, the large search space generated for all but the most trivial of planning problems leads to a reduction in the impact gained from propagation of the initial state and goal state constraints. Hence, the problem once again becomes that of traversing a large search space looking for a solution, in this case by systematically assigning values to the problem variables from their respective domains and testing whether each assignment satisfies the constraints imposed.

Many CSP heuristics [11] have been developed to improve the efficiency of CSP search algorithms. Whilst the use of these generic CSP techniques for solving planning problems can be effective on small problems, it has hitherto not matched the performance of other strategies across a wider range of problem instances. This is, in part, due to the fact that the search guidance is not tailored to the structure of the original planning problem.

1.3 Statement of Thesis and Contributions

The thesis proposed in this document is: "*It is possible to solve planning problems more efficiently by better exploiting the inherent constraint inference mechanisms within a given constraint solver. This can be achieved by using the structure of the original planning problem to guide the search for a solution to a CSP reformulation of that planning problem. Such a new, and automatically generated, heuristic may be shown to be effective on a range of domains and problem instances. Additionally, the use of meta-CSP techniques may further increase the amount of propagation achieved and hence provide an additional improvement in solution time. When used in tandem, these techniques will provide a performance increase on standard planning benchmark problems when measured against a CSP reformulation without such guidance.*".

The main contributions of this work are:

- A study of backdoor-like trial variables which, when inserted into the problem space, allow the CSP reformulated planning problem to be solved more efficiently, in terms of run-time and number of backtracks.
- An AI Planning specific goal directed variable and value selection heuristic for guiding the solution process in CSP reformulated planning problems. An implementation of this heuristic, and successful testing of same on a range of benchmark problems.
- The use of CSP meta variable techniques to implement goal-locking in order to make further use of the inherent constraint propagation mechanisms within the heuristically guided CSP solver.
- A novel resource-to-task assignment method, achieved by further application of CSP meta variable methods. The addition of this technique to the heuristically guided solver infrastructure, thereby achieving further benefit from the CSP's inference techniques, and achieving improved run times and backtrack counts.

1.4 Thesis Chapter Description

The following chapter, Chapter 2, provides the necessary background for the work presented in this thesis. Firstly, a broad introduction to AI planning is given which includes a description of two ways of representing classical planning. This is followed by a discussion of CSPs and how these may be used to reformulate planning problems. Finally, the context for this thesis is presented by detailing previous work that makes use of constraint techniques for solving planning problems.

Chapter 3 starts with a description of two CSP encoding methods; *Intensional* and *Extensional*. A comparison of the results of using each to encode planning problems is given. This leads to a discussion of propagation in constraint encodings of planning problems. The chapter then details the first contribution of this work, an empirical study in which selected CSP *trial variables* are inserted into the problem space in order to boost the efficacy of the inherent inference techniques. The results of this study are given and these show the desirability of having known bridging points to break up the search space.

The second contribution of this work, a goal-based CSP heuristic method, is introduced in Chapter 4. This technique makes use of the ideas discussed in the previous chapter to increase the impact of propagation. By using the original planning problem's goals as intermediate deadlines, it is shown that much better use can be made of the CSP propagation machinery. The results of implementing this variable and value ordering heuristic within a fully automated CSP-based planner are given. The system is empirically evaluated over a range of domain / problem instances.

Further contributions of this work are discussed in Chapter 5; the use of meta-CSP techniques to further increase the impact of constraint propagation. The chapter first describes how recently achieved goals are locked by using meta-variables, and then goes on to detail the use of a meta approach for resource to task assignment. The results gained by adding each of these to the prototype CSP system are shown, again by testing the planner on a range of benchmark domains.

Chapter 6 summarises the results provided in the preceding three chapters. Conclusions drawn from these are discussed and recommendations for future work are made.

CHAPTER 2

BACKGROUND AND CONTEXT

2.1 Introduction

Since the work presented in this document deals with using CSPs for planning, it is first necessary to discuss the relevant background to both AI planning and CSPs.

The following section of this chapter provides historical perspective on classical planning and includes a description of several methods of representing planning problems including the de-facto research community standard and an alternative multi-valued representation.

Section 2.3 introduces constraint satisfaction and discusses the different approaches used to find solutions to CSPs. Search and inference techniques are described, as are CSP heuristics.

The steps required to recast a planning problem as a CSP are presented in section 2.4 and the complexity of the CSP encoding is also discussed in this section.

Section 2.5 reviews related constraint based approaches to planning and provides a frame of reference for the current work.

2.2 Classical Planning

Planning may be regarded as a combination of logic and search. The planning problem, described in some logical form, has a given start condition (*Initial state*) and a given target condition (*Goal State*). Predefined *actions* allow movement from the initial state, through a series of possible intermediate states, to arrive at the goal state. It is necessary to search through this state space in order to find the set of actions which leads to the goal condition. This list of actions is the *plan*. Clearly, such a system is centred on the actions and the movement between states that these cause. More formally, this type of interaction can be described using an abstract machine or *state-transition system* [3],[12].

Definition 1 *A state-transition system is a 4-tuple* $\Sigma = (\mathcal{S}, \mathcal{A}, \mathcal{E}, \gamma)$*, where:*

- $\mathcal{S} = \{s_1, s_2, \ldots\}$ *is a finite or recursively enumerable set of states;*
- $\mathcal{A} = \{a_1, a_2, \ldots\}$ *is a finite or recursively enumerable set of actions;*
- $\mathcal{E} = \{e_1, e_2, \ldots\}$ *is a finite or recursively enumerable set of events;*
- $\gamma : \mathcal{S} \times \mathcal{A} \times \mathcal{E} \rightarrow 2^{\mathcal{S}}$ *is a state-transition function.*

Remembering that classical planning, as introduced in Chapter 1, operates in a restricted environment in order to manage complexity.

Definition 2 *For the state-transition system,* Σ*, the classical planning restrictions are:*

- Σ *has a finite set of states;*
- Σ *is fully observable;*
- Σ *is deterministic;*
- Σ *is static unless affected by some action (i.e.* $\mathcal{E} = \emptyset$*);*
- *The planning system handles only restricted goals (i.e. an explicit goal state,* s_g*, (or set of goal states,* $\mathcal{S}_g$*));*
- *Solution plans are finite sets of actions;*
- *Actions have no duration, they are carried out instantaneously;*
- *The planning system, once in operation, is neither concerned with, nor affected by, any changes in* Σ *(i.e.* Σ *plans for the given goals, with the given initial state and is not affected by any change in the real world of which it is a model).*

Applying these restrictions, it is possible to define a *restricted state-transition system.*

Definition 3 *A restricted state-transition system is a 3-tuple* $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$*, where:*

- $\mathcal{S} = \{s_1, s_2, \ldots\}$ *is a finite set of states, each with a finite set of propositions;*
- $\mathcal{A} = \{a_1, a_2, \ldots\}$ *is a finite set of actions;*
- $\gamma : \gamma(s, a) \rightarrow s'$ *is a state-transition function.*

With this description of a restricted state-transition system, $\Sigma = (\mathcal{S}, \mathcal{A}, \gamma)$, classical planning can be defined as follows.

Definition 4 *A planning problem is a triple, $\mathcal{P} = (\Sigma, s_0, g)$ where:*

- *s_0 is the initial state;*
- *g is a set of goal states.*

A solution to $\mathcal{P}$ is a sequence of actions $(a_1, a_2, \ldots, a_k)$ that corresponds to a sequence of state transitions $(s_1, s_2, \ldots, s_k)$ where $s_1 = \gamma(s_0, a_1), \ldots, s_k = \gamma(s_{k-1}, a_k)$, with s_k being a goal state

Although simplified and limited, the restricted state-transition approach provides an easily understood theoretical framework upon which to model classical planning problems. However, in order to build practical problem descriptions and find solutions, some means of representing actions and states must be found, and an appropriate search method must be chosen. Whilst many search techniques have been used for AI planning (discussed further in section 2.2.6), there are relatively few representation schemata for actions and states.

Any such representation schema must be sufficiently expressive to allow for the description of a wide range of interesting problems whilst being compact enough to allow search algorithms to efficiently traverse the problem space. The explicit representation of all potential states in the search space, for anything but the most trivial of problems, is not practical since the number of reachable states is exponential in the size of the problem description: If a problem has n propositional objects, there are 2^n states while, if there are, for example, ternary relations between the objects, the size of the state space increases to 2^{n^3}. It is for this reason that graph search techniques are not appropriate; even for classical planning, the graph would be too large to be constructed explicitly.

Thus, representation schemata for classical planning problems must control the *combinatorial explosion* inherent in the explicit description of such problems. Also, the associated search algorithms should limit the amount of the resulting state space that is actually searched. The following sections introduce a number of representation schemata and search techniques which aim to achieve these goals.

2.2.1 STRIPS

The roots of classical planning can be traced back to the 1960s with early AI work on the General Problem Solver (GPS) [13] and QA3 theorem proving system [14].

A theorem proving system defines the planning problem in terms of logical formulae and uses deductive reasoning to prove the existence of a solution. For example, *Situation Calculus* [15], an expressive first-order language, describes *situations* or states and the actions that, when applied, result in a different situation (change of state). It is necessary to describe both what changes and what does not change following the application of an action. In situation calculus, *Effect axioms* describe the changes and *Frame axioms* describe unchanged objects.

Whilst the number of atoms that change in a situation as a result of an action being performed is relatively small, the number of atoms that do not change (that is, atoms that are unaffected by that particular action) is likely to be much larger. As an example, consider a situation containing a description of a warehouse housing 1,000 packages. Another situation, after a truck has been loaded with ten packages, requires not only the description of the 10 packages now in the truck, but also a description of the 990 packages that remain. Hence, it is clear that under this representation it quickly becomes necessary to have a very large number of frame axioms. This is known as the *frame problem* [15].

The first major dedicated planning system, STRIPS [2], circumvented the frame problem by making the *Strips Assumption*. The assumption made is that any atom not explicitly mentioned in an action's list of effects remains unchanged from one situation to the next. Although AI planning has developed considerably since the introduction of STRIPS, this assumption remains central to the representation and solution of planning problems.

With the development of the STRIPS system came an action representation schema that also proved to be influential and durable. The original system permitted operators (actions) with a list of preconditions, an *add* list, and a *delete* list. It was possible for each of these to contain arbitrary well-formed first-order formulae. Finding a clear semantics for this formulation was difficult [16] and in a revised approach [17] the list of preconditions, add list, and delete list could only contain atoms. It is this version that is referred to as *STRIPS-style planning*.

Formally, a STRIPS planning task may be defined as follows.

Definition 5 *A STRIPS planning task is a triple $\mathcal{P} = \langle \mathcal{A}, \mathcal{I}, \mathcal{G} \rangle$, where:*

- *$\mathcal{A}$ is a set of actions. Each action $o \in \mathcal{A}$ is a triple, $o = \langle pre(o), add(o), del(o) \rangle$, where pre(o), add(o) and del(o) are each sets of atoms representing the preconditions, add, and delete lists respectively. A world state is represented by a set of atoms and an action o is applicable in a state $\mathcal{S}$ if $pre(o) \subseteq \mathcal{S}$. The resulting state $\mathcal{S}' = (\mathcal{S} \cup add(o)) \backslash del(o)$.*
- *$\mathcal{I}$ is the initial state and is represented by a set of atoms.*
- *$\mathcal{G}$ is the partially specified goal state and is represented by a set of atoms.*

A simple example of a STRIPS encoded logistics domain and problem instance is given in Figure 2.1.

Action DRIVE (Truck, Start, End)
  Preconditions: at(Truck,Start), link(Start,End)
  Add: at(Truck,End)
  Delete: at(Truck,Start)

Action LOAD (Package, Truck, Location)
  Preconditions: at(Package,Location), at(Truck,Location)
  Add: in(Package,Truck)
  Delete: at(Package,Location)

Action UNLOAD (Package, Truck, Location)
  Preconditions: in(Package,Truck), at(Truck,Location)
  Add: at(Package,Location)
  Delete: in(Package,Truck)

Initial State:
  at(Truck1,Depot)
  at(Package1,City1)
  link(Depot,City1)
  link(City1,Depot)
  link(Depot,City2)
  link(City2,Depot)

Goal State:
  at(Package1,City2)

Figure 2.1: STRIPS domain and problem definition for a simplified logistics domain.

Important restrictions contained in the STRIPS approach include the stipulation that all literals not contain functions, and that any literals not mentioned in a state's description of the real world are assumed to be false.

The former restriction ensures that the action schema can be transformed (*grounded*) into a finite number of propositional actions. For example, consider the logistics domain example of Figure 2.1, the *DRIVE (Truck, Start, End)* action schema, for a problem instance with five trucks and three locations, would become 45 (i.e. $5 \times 3 \times 3$) propositional actions. Were function symbols to be permitted, an infinite number of actions and states could be formed.

The latter restriction, known as the *Closed World Assumption* [1], can lead to problems if the planning model has been incorrectly designed. Namely, if an atom that is not present in the current state and hence is assumed to be false, is required in the planning process, an erroneous result will occur with respect to the real world. That is, the assumption that the atom's value is false is incorrect. Whilst this is a modelling error, the consequence is a disruption of the planning process.

Although STRIPS and the situation calculus are each based on the state-transition model, the STRIPS representation has been used in many more practical systems. This predominance is largely due to the tractability of STRIPS. In general, the difficulty of reasoning with a representation schema increases as the expressivity increases [18]. By reducing the expressive power of STRIPS, the designers made STRIPS representations of classical planning problems computationally tractable. This contrasts with the situation calculus which makes use of the full expressive power of first-order logic. Deducing a plan in the situation calculus representation is equivalent to theorem-proving in first-order logic, and due to the general intractability of theorem proving in first-order logic [19], it has been shown [20] that it is difficult to efficiently find solutions to planning problems using situation calculus with generic theorem-proving methods. This tradeoff between expressiveness and tractability is a fundamental consideration when choosing a representation system for AI planning.

With the widespread use of STRIPS for classical planning problem representation in the research community came the realisation that, due to its reduced expressivity, the STRIPS schema was inadequate for the representation of some real-world problems.

2.2.2 Action Description Language

The Action Description Language (ADL) [21] augments the STRIPS schema with a number of additional features that allow for the modelling of many more realistic problem domains. This places ADL higher than STRIPS, but lower than the situation calculus, in terms of expressivity.

One of the most important additions is the provision of conditional-effects. These allow a more fine grained control over a given action's effects. For example [22], in a domain containing a spacecraft with controllable engines; when the engines are fired, the consequent change in trajectory depends on the length of time for which the engines are fired, the velocity, mass, orientation and position of the craft at the time of firing as well as the thrust and fuel consumption of the engines. STRIPS operators' effects are unable to reflect such a situation in which an action's effects depend upon the circumstances in which that action is carried out.

Also included in the ADL representation is the facility to deal with both positive and negative literals in state descriptions, quantified variables and disjunction within goal definitions, support for equality (equality predicate is built in), and *typing* of variables. Additionally, ADL operates under the *Open World Assumption* which means that all literals not mentioned in a given state are assumed to be unknown, and not false, as is the case in STRIPS.

2.2.3 Planning Domain Definition Language

In an attempt to standardise the description of planning domains and associated planning problems, the Planning Domain Definition Language (PDDL) [23] was developed. The intention was to encourage researchers to share problem definitions and algorithms, and to allow the performance of different planners to be compared. An additional motivation for this development was the 1st International Planning Competition (IPC) [24], hosted by the Artificial Intelligence Planning Systems Conference (1998). The AI planning research community has embraced PDDL and it has been used at every subsequent IPC.

Using STRIPS style actions, and supporting ADL style conditional effects, PDDL describes what is present in a problem: What predicates there are, the actions and their effects, and the structure of any compound actions. The language is divided into sets of features (*requirements*) with each problem domain specifying which requirements are needed. The most commonly used include *strips*, *adl*, *equality*, and *typing*.

In order to fully describe a problem, two files are generally used. The first, the *domain* file, contains the domain definition and the complete set of action schemata. The second file, the *problem* file, contains a listing of the objects used for a specific problem instance in this domain.

A simple example logistics problem is shown in Figure 2.2. Part of the appropriate domain file is listed in Figure 2.3 and the specific problem file is given in Figure 2.4.

Since its inception, PDDL has been developed and expanded to include many features not required in classical planning. Among the additions included in PDDL 2.1 [25] is the ability to handle actions that take time to complete (*durative actions*), the facility for reasoning about numeric quantities, and the ability to specify an optimisation metric (in addition to plan length). A further iteration, PDDL 2.2 [26], saw the revision of *derived predicates* and the introduction of *timed initial literals*. The first is a mechanism for computing state variables as a function of other variables, and the second a means of introducing facts that become true (or false) at some specified time.

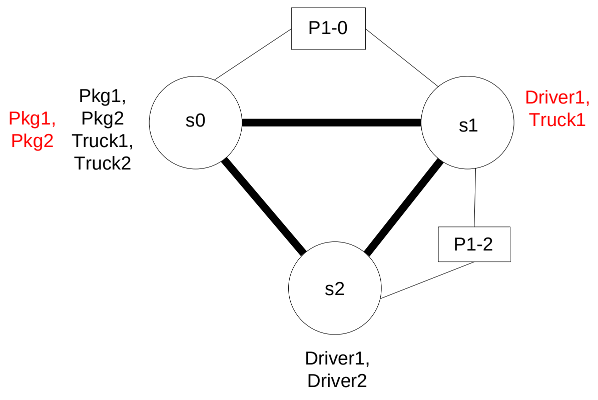


Figure 2.2: Visual description of example logistics problem. (Trucks can only drive on *links* (bold lines), drivers can only walk on *paths* (fine lines). Black text indicates the initial state, red indicates the goal state.)

```
(define (domain driverlog)
   (:requirements :typing :strips)
   (:types location locatable - object    driver truck obj - locatable)
   (:predicates (at ?obj - locatable ?loc - location)    (empty ?v - truck)
                (in ?obj1 - obj ?obj - truck)    (driving ?d - driver ?v - truck)
                (link ?x ?y - location)    (path ?x ?y - location))

(:action LOAD-TRUCK
  :parameters (?obj - obj ?truck - truck ?loc - location)
  :precondition (and (at ?truck ?loc) (at ?obj ?loc))
  :effect (and (not (at ?obj ?loc)) (in ?obj ?truck)))
                              .
                              .
                              .
(:action DRIVE-TRUCK
  :parameters (?truck - truck ?loc-from - location ?loc-to - location
               ?driver - driver)
  :precondition (and (at ?truck ?loc-from) (driving ?driver ?truck)
                (link ?loc-from ?loc-to))
  :effect (and (not (at ?truck ?loc-from)) (at ?truck ?loc-to)))
)
```

Figure 2.3: Partial PDDL domain definition for a logistics domain.

```
(define (problem DLOG-2-2-2)
(:domain driverlog)

(:objects driver1 - driver   driver2 - driver   truck1 - truck   truck2 - truck
          package1 - obj   package2 - obj   s0 - location   s1 - location
          s2 - location   p1-0 - location   p1-2 - location )

(:init (at driver1 s2)   (at driver2 s2)   (at truck1 s0)   (empty truck1)
       (at truck2 s0)   (empty truck2)   (at package1 s0)   (at package2 s0)
       (path s1 p1-0)   (path p1-0 s1)   (path s0 p1-0)   (path p1-0 s0)
       (path s1 p1-2)   (path p1-2 s1)   (path s2 p1-2)   (path p1-2 s2)
       (link s0 s1)   (link s1 s0)   (link s0 s2)   (link s2 s0)   (link s2 s1)
       (link s1 s2))

(:goal (and (at driver1 s1) (at truck1 s1) (at package1 s0) (at package2 s0)))
)
```

Figure 2.4: PDDL problem definition for a logistics domain.

More recently (2006), PDDL 3.0 [27] introduced *preferences* and *soft constraints* in order to improve the language's expressive power in terms of plan quality specification for propositional planning. An even more detailed version of PDDL, PDDL$^+$ [28], specifies five levels of expressive power, with the first three corresponding to PDDL 2.1. The final two levels introduce *events* and *processes*, and allow complex discrete-continuous real-time systems to be modelled.

The steady increase in the expressive power of PDDL has meant a concomitant increase in the amount of resources and reasoning power required to find solutions, with the result that only a small number of planning systems are able to solve problems specified in PDDL$^+$. The work discussed in this thesis relies upon a subset of the expressive power of PDDL 2.1, that which operates under the restrictions of Definition 2.

2.2.4 State Variable Representation of Planning

The Simplified Action Structures (SAS$^+$) [29] formalism is an alternative to the STRIPS based PDDL representation for AI planning. This state variable description is equivalent to STRIPS, in terms of expressivity. There are however two important differences between the two formalisms. Whereas STRIPS makes use of propositional atoms, SAS$^+$ employs *multi-valued variables*. Also, where STRIPS uses add and delete lists (for preconditions and effects), SAS$^+$ has *pre-conditions*, *prevail-conditions* and *post-conditions*.

A SAS$^+$ action's prevail-conditions determine which of the state variables, not included in that action's post-conditions, must hold a given value for that action to be carried out. This contrasts with the SAS$^+$ pre-conditions which contain the required variable and value combinations for state variables that this action does have in its post-conditions. That is, both the pre-conditions and the prevail-conditions of an action must be satisfied in order for the action to be carried out, but only those variables listed in the pre-conditions (and hence post-conditions) are affected by this action. This splitting of the conditions can be useful in a *parallel plan*, where the search is for a sequence of sets of actions, such that any ordering of the actions provides a sequential plan. However, for the purposes of the work in this thesis (*fully ordered sequential planning)*, there is no need to make a distinction between the pre-conditions and the prevail-conditions.

It is possible to translate from a STRIPS style encoding to SAS$^+$ by creating a SAS$^+$ binary variable from each propositional STRIPS variable. Likewise, the SAS$^+$ actions are created from the STRIPS actions; for the preconditions, the SAS$^+$ state variables that correspond to the STRIPS propositional variables must be satisfied (i.e. *true*) for the action to be carried out. Also, the STRIPS add effects correspond to setting the appropriate SAS$^+$ variables to true, and similarly, the STRIPS delete effects correspond to the relevant SAS$^+$ variables being set to false.

Under such a translation scheme, the number of SAS$^+$ variables equals the number of STRIPS variables. It is possible however, with some preprocessing of the STRIPS encoding, to identify implicit non-binary variables. For example, in the problem of Figure 2.2, the propositional approach uses (***at*** *?truck ?location*) to represent the location of a truck. Assuming that a truck can be in only one location at any given time, all of the predicates that describe a truck's location can be compiled into a single SAS$^+$ variable with a multi-valued domain, $truck\_location \in \{Location_1, Location_2 \ldots\}$. An additional value is added to the domain to represent the *undefined* value (used when the truck's location is not defined). In the general case, for each set of *n* predicates describing an object in the propositional scheme, there will be one variable with a domain of *n+1* values in the SAS$^+$ representation.

Stated formally, a SAS$^+$ planning task is as follows.

Definition 6 *A SAS$^+$ planning task is a 4-tuple* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, *where:*

- $\mathcal{V} = \{v_1, v_2, \ldots, v_m\}$ *is a set of state variables. Each variable* $v \in \mathcal{V}$ *has an associated finite domain* $\mathcal{D}_v$*. Implied is an extended domain* $\mathcal{D}_v^+ = \mathcal{D}_v \cup \{\boldsymbol{u}\}$, *where* ***u*** *denotes the undefined value. The total state space is* $\mathcal{S}_v = \mathcal{D}_{v_1} \times \ldots \times$

$\mathcal{D}_{v_m}$, and a partial state space is represented by $\mathcal{S}_v^+ = \mathcal{D}_{v_1}^+ \times \ldots \times D_{v_m}^+$;

- *$\mathcal{O}$ is a set of operators, where an operator is a triple $\langle pre, post, prv \rangle$, with $pre, post, prv \in \mathcal{S}_v^+$ representing pre-conditions, post-conditions, and prevail-conditions respectively. $\mathcal{O}$ has two restrictions:*
  - *for all $v \in \mathcal{V}$, if $pre[v] \neq \boldsymbol{u}$, then $pre[v] \neq post[v] \neq \boldsymbol{u}$;*
  - *for all $v \in \mathcal{V}$, $post[v] = \boldsymbol{u}$ or $prv[v] = \boldsymbol{u}$[1].*
- *$s_0 \in \mathcal{S}_v^+$ is the initial state;*
- *$s_\star \in \mathcal{S}_v^+$ is the goal state.*

Compared to the STRIPS encoding, the SAS$^+$ approach may be considered more concise. That is, a given planning problem can be encoded with many fewer variables, each with a larger range of possible values. This is relevant when considering reformulation of a planning problem as a CSP. Under the constraint based paradigm, it is good modelling practice [11] to use a smaller number of variables, each with a larger number of potential values. It is for this reason that the SAS$^+$ representation of classical planning was used as a starting point for the work described in this thesis document.

Before introducing CSPs in section 2.3, the following two subsections discuss search in classical planning and the use of heuristics to guide that search. This discussion is not intended to be exhaustive, but should allow an understanding of the background and direction of classical planning research.

2.2.5 The Classical Planning Search Space

Some of the early planning systems already introduced, such as GPS and STRIPS, made use of a state-space representation of the planning problem, whereas a number of systems that followed (e.g. NOAH [30], NONLIN [31]) used the concept of plan-space to describe the search landscape. These approaches are discussed in subsections 2.2.5.1 and 2.2.5.2 respectively.

2.2.5.1 State-space

In the state-space approach, a search algorithm operates over a subset of the total state-space. Here, the search for a plan is the search for a path in the graph, Σ, of a state transition system. The nodes of the graph are world states, connecting edges equate to transitions between states (actions), and a solution plan is an action sequence that

[1]States that the prevail condition of an operator must never define a variable which is affected by that operator.

corresponds to a path in the graph from the initial state to the goal state. This search method is considered a *totally ordered* approach since the system only explores linear sequences of actions which originate in either the initial state or goal state, for *forward state-space search* or *backward state-space search* respectively.

Forward state-space search starts in the initial state and chooses one of a number of *applicable* operators (instance of an action). An applicable operator is any operator that has its preconditions satisfied in the current state. Upon application of a given operator, a new state is reached. This successor state is equivalent to the previous state, but with the operator's positive effects added and negative effects removed.

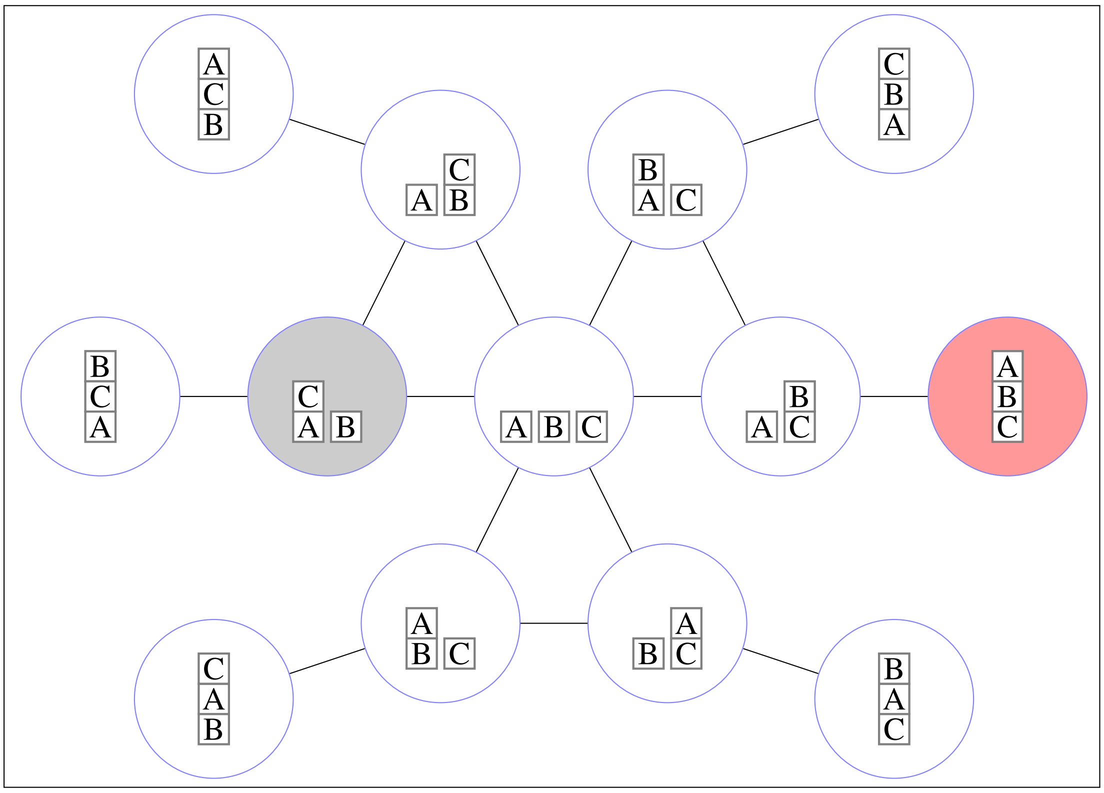


Figure 2.5: State-space representation of a *Blocks* problem.

Figure 2.5 shows the state-space representation for an example problem from a *Blocks* domain [32]. In this domain, a block may be picked up and placed either on the table or onto the top of another block, assuming that neither the block being moved nor the receiving block has anything on top. The initial state is shown in grey, and the goal state in red. Applying forward search from the initial state in this example leads to one of the three nodes connected to the initial state. That is, the system chooses any of the applicable actions available in the initial state. Similarly, the next step leads from the successor node to any of its connected nodes. It is clear that, even in such a small state space, there exists the potential for much blind searching. For example, it is possible for the planning system (without additional guidance or restriction) to *toggle* backwards and forwards between states; thereby, at best, including many pointless actions or, at worst, covering the entire state-space before reaching the goal state.

One advantage of using a simple forward state-space approach is that any of the already well known brute-force search algorithms [1] may be used to choose an action. However, without heuristic guidance such methods very quickly become bogged down by the number of possible action combinations. Another inefficiency is that forward search considers all applicable actions, even when, to a human observer, it is pointless to do so. For example, in Figure 2.5, an action to move block C from block A to block B is applicable in the initial state, but does not help in the search for an optimal plan, which would be: *move C to the table*, *move B onto C*, *move A onto B*.

Searching in the opposite direction, known as backward state-space search, can result in a more efficient search through the state-space. This method sees the algorithm step back through the space, applying actions ”in reverse” in order to reach the initial state. Whereas in forward search all applicable actions are considered, it is possible to only consider *relevant* actions in the backwards approach. An action is considered relevant if it achieves any part of the conjunctive goal description. In addition to the action achieving part of the goal state, it should not undo any other part of it. Such an action is both relevant and *consistent*.

In order to find a plan using the backwards search method, a relevant, consistent action is chosen. The action’s preconditions are then added to the conjunction of goal literals that are, as yet, unsatisfied. Also, the positive effects of the action are removed, thereby forming a new predecessor state to which further action *regression* can be applied. The algorithm terminates when it generates a predecessor state which contains a set of literals that are all true in the initial state.

Referring again to Figure 2.5, backward state-space search, starting in the goal state, requires an action that contributes to the state: [(block A on block B) and (block B on block C)]. The action *move A onto B* satisfies the condition (block A on block B) and hence, it can be chosen. The resulting set of goals (and subgoals) that remain unsatisfied consists of the preconditions of the *move A onto B* action (i.e. block A on the table) and the condition: (block B on block C). This is shown in the only state directly connected to the goal state. Continuing backwards in this manner, choosing the correct actions, leads to an optimum plan: *move C to the table*, *move B onto C*, *move A onto B*. As with forward state-space search, the action choice mechanism can use any brute-force search technique.

Since brute-force search time is $O(b^n)$, where b is the *branching factor* (number of successor nodes at each node) and n is the number of choices made before a solution is found, backward search will be faster than forward search on any given problem. This is due to the pruning of actions that results from starting at the goal state. That is, since the goal state generally contains a small subset of the literals used to describe the initial state (only the facts that are required to be true in the goal are stated, the remainder are omitted and assume a "don't care" value), the value of b will be much reduced, contributing to a lowered search time.

The algorithm associated with the previously introduced STRIPS planning system employed a modified form of backward state-space search. In an attempt to improve search efficiency by pruning the search space, the STRIPS algorithm includes a potential operator in a plan if the current successor node satisfies all of the operator's preconditions. That is, STRIPS commits to such an operator and does not allow backtracking over it. Whilst this does prune the search space, it also renders the STRIPS approach *incomplete* (i.e. the system does not find all possible solutions, including some optimal plans). Another efficiency improving measure, which considerably reduces the branching factor, is the requirement that only the preconditions of the most recent operator added to the plan are considered for processing. Whilst this condition does improve efficiency, it too makes STRIPS incomplete.

By using backward search and solving subgoals independently, STRIPS permits goals that have already been achieved by the unfolding plan to be undone. These goals are referred to as *clobbered* goals, and an additional procedure attempts to repair the plan by re-achieving these goals at the end of the planning process.

The problem shown in Figure 2.5, known as the *Sussman anomaly*, is one of a class of non-linear problems that the STRIPS algorithm is unable to deal with in an optimum way. Regardless of which sub-part of the conjunctive goal STRIPS attempts first, the process of solving the other subgoal always undoes (clobbers) the first.

The realisation that STRIPS style search was incomplete led to the development of both a different type of search-space representation and an alternative method of planning. These are known as plan-space search and *partial-order planning* respectively.

2.2.5.2 Plan-space

In contrast to state-space search, plan-space search is a *partially ordered* technique. Here, the nodes of the graph represent partially complete plans (Figure 2.6), and the edges are plan refinement operators. Such operators are used to modify and augment the partial plans in an attempt to build a plan that satisfies the problem goals. Employing a principle of *least commitment* [33], plan-space systems attempt to add to the plan only those restrictions and bindings that are necessary to achieve the given goals. Whereas in state-space search the final plan is a fixed sequence of actions, in the plan-space approach a "plan" is defined as a collection of planning operators and a set of associated ordering constraints and bindings. This difference allows an executive system, to which the plan is passed, much greater freedom in deciding how the plan is actually carried out.

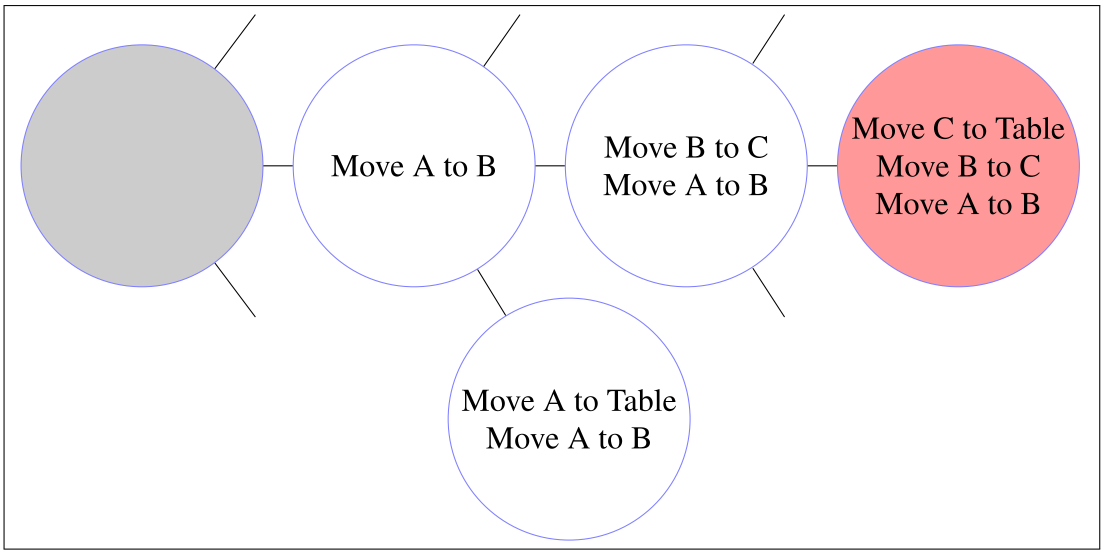


Figure 2.6: Partial plan-space representation of a *Blocks* problem.

Whilst plans in the plan-space paradigm are not as rigidly ordered as those in state-space systems, there remains a requirement for the actions to be placed in an order that prevents destructive interference between them. A number of methods exist for ensuring that the partially ordered plan is as flexible as possible while still achieving the overall problem goal.

The NOAH system, generally considered to be the first planner to use a form of partial-ordering[1], generated separate plans to satisfy each part of a conjunctive goal. These individual plans were then combined to form a partially ordered set of actions to achieve the complete goal. Since achieving one goal potentially undermines the achievement of another (e.g. in non-linear, *Sussman*, type problems), further refinements are necessary. NOAH achieved this by considering the composite plan a potential solution. This potential solution is then analysed for *flaws*, with each systematically resolved in order to arrive at a final ordering.

[1]Actually, NOAH was an HTN planner [3]

The NONLIN system improved on NOAH by adding a backtracking control structure at the top level. This allowed the planner to solve problems on which NOAH failed. NONLIN also introduced further plan refinement operators, including *causal links*.

The TWEAK [34] partial order planner defined a number of useful plan-space concepts and formalised the approach, introducing *promotion* (placing an action before another to avoid conflict), *demotion* (placing an action after another to avoid conflict), and *declobbering*.

UCPOP [35], a sound and complete partial order planner, built on the work of TWEAK and is considered a major contribution to plan-space planning. The system accepts many of the additional features included in ADL, and was the first planner of its type to make use of *lifted* actions. Lifted actions, in contrast to *grounded* actions, are uninstantiated. That is, they contain unbound variables. Lifted actions facilitate the use of *separation* as a conflict resolution mechanism between actions. Separation involves constraining the unbound parameter of an action that may cause a conflict with another action. Such a constraint forces the value of the parameter to be different from that in the potentially conflicting action.

Definition 7 *The UCPOP system represents a partial plan as a quadruple* $\langle \mathcal{S}, \mathcal{B}, \mathcal{O}, \mathcal{L} \rangle$*, where:*

- $\mathcal{S}$ *is a set of steps;*
- $\mathcal{B}$ *is a set of binding constraints on free variables in* $\mathcal{S}$*;*
- $\mathcal{O}$ *is a set of ordering constraints over* $\mathcal{S}$*;*
- $\mathcal{L}$ *is a set of causal links, where a causal link,* $S_P \overset{e,r}{\rightarrow} S_C$*, is a quadruple.* S_P *and* S_C *are pointers to the link's producer action and consumer action respectively.* e *is an effect of the producer,* S_P*, and* r *is a precondition of the consumer,* S_C*. Also,* $\exists\, q \in \theta_e$*, where* θ_e *is the set of effect postconditions, such that* q *unifies with* r*.*

UCPOP's goal-directed search algorithm begins with a dummy plan consisting only of a *start* step and a *goal* step. The start step's effects are equal to the problem's initial conditions, and the goal step's preconditions equate to the problem's goals. The system attempts to supplement this "empty" plan by adding steps, and their associated constraints, until all *open preconditions* are satisfied. There are two search choices in the algorithm.

The first concerns the open preconditions; for each open precondition, the system considers all possible step effects that could satisfy it. One such step is chosen and a causal link is added to the plan in order to record the connection between producer and consumer actions.

The second choice in the algorithm relates to any potential *threats* to the precondition supported by the newly added causal link. The system chooses some means of resolving the threat. Here the options include adding further ordering constraints and posting additional inequality constraints to enforce separation (page 21).

When there are no further unsupported preconditions, and all causal links are in place, and are protected from threats, the algorithm returns the resulting solution "plan".

Whilst plan-space planners have advantages over state-space approaches (e.g. more flexible execution of the partially ordered plan, more explicit structural explanation in the final plan (causal links show which actions support goals and subgoals)), plan-space systems are more complex in terms of nodes in the search space. The operations carried out to refine a partial plan are more time consuming than their state-space equivalents. Hence, without improved search guidance, the partial order systems require considerable amounts of search effort. Earlier systems had attempted to estimate the cost of achieving the problem goal, to use as a heuristic, by counting the number of open preconditions remaining in the plan. This is not always an accurate measure however, since there is not necessarily a one to one relationship between the number of remaining open preconditions and the number of actions required to achieve them. In an attempt to address this problem, more recent partial order planners have employed more advanced heuristics. The following section includes a description of two partial order planning systems that take such an approach (sub-section 2.2.6.2).

2.2.6 Heuristic Search in Classical Planning

Considering AI planning as search relies on the search algorithm traversing the search space in order to return a solution plan. Simple, systematic search algorithms; such as *breadth-first search*, *depth-first search* or *IDDFS* [1] will eventually find a solution. However, such uninformed techniques can potentially visit every node in the space (assuming a finite search space), with the consequent exponential time requirement. By using an informed node evaluation function, $f(n)$, it is possible to highlight the node that appears to be the *best* choice as a successor to the current node, using a given criterion.

A range of such *best-first search* [3] algorithms exist, each with a different method of choosing the successor node, and with various approaches to processing "revisited" states. Generally, the best-first approach makes use of a heuristic function, $h(n)$, which is the estimated length of the shortest path between the current node, n, and a goal node (if the current node is a goal node, $h(n) = 0$).

The *greedy best-first search* [6] method uses the evaluation function $f(n) = h(n)$. That is, the algorithm chooses, as a successor, the node that is estimated to be closest to the goal. By following the lowest $f(n)$ values, a single path is taken to the goal. Thus, as with depth-first search, the solution found may not be optimal. Potentially, this approach is incomplete since the algorithm can proceed along an infinite path, thereby never considering alternative possible (solution) paths. However, when used in AI planning (e.g. [36]), greedy best-first search is complete due to the finite, albeit often very large, number of states. With an accurate heuristic function, this technique typically aims to find not an optimal solution, but a fast, *satisficing* (good enough) one.

The A^* algorithm [37], an augmented form of best-first search, uses an evaluation function that includes both the distance from the starting point to the current node, $g(n)$, and the estimated distance from the current node to the goal node, $h(n)$. Hence, $f(n) = g(n) + h(n)$, is an estimate of the cost of the best solution passing through node n. A^* is guaranteed to find the optimal solution, if the heuristic function, $h(n)$, is *admissible* (does not overestimate). This is achieved by ensuring the heuristic used is *consistent*. That is, the heuristic satisfies the triangle inequality $h(n) \leq c(n, n^{'}) + h(n^{'})$ for all n and $n^{'}$, where $c(n, n^{'})$ is the "cost" of moving from n to $n^{'}$.

Systematic searches, such as A^*, have large memory requirements that can exceed available resources and may cause the search to fail. Alternative *local search* techniques (e.g. *hill-climbing*) [1] do not cover the entire search space systematically, but simply move to the next successor node if that node has a better $f(n)$ value than the current node. By not maintaining a record of alternative branches to search, this method uses little memory, and can often find a solution very quickly. However, progressively following the lowest $f(n)$ value can lead to the search becoming stuck in a *local minima*. In this situation, the algorithm has found the best $f(n)$ value in the search *neighbourhood* and has no way out; all connected nodes have a higher cost (appear further from the goal). Various schemes exist to *restart* the search, including *stochastic hill-climbing* and *random-restart hill-climbing* [1].

One very successful heuristic-based AI planning system [38] employs *enforced hill-climbing* (EHC), which relies on breadth-first search to escape plateaux. When tested on the standard AI planning benchmark problems, EHC exhibits short periods of heuristic-based progress, and longer periods of exhaustive search. Despite EHC being incomplete (FF then invokes the complete search algorithm), when combined with an appropriate heuristic evaluation function, this approach has been shown to be effective over a range of domains [39].

The above *informed search algorithms*, although differing in search implementation, all make use of a heuristic function. For AI planning, a number of heuristic schemes have been developed. These include those based on the *delete relaxation* (e.g. [40]), pattern-databases (e.g. [41]), critical paths (e.g. [42]), and landmarks (e.g. [43]). However, by far the most common are the heuristics that make use of the delete relaxation of the problem. By finding a solution to this relaxed version of the problem, the resulting sequence of actions can be used as a heuristic with which to guide the search for a solution to the real problem (a useful side-effect is that many unhelpful actions are *pruned* from the search space).

The following subsection introduces the use of heuristic search in state-space planning. It includes several examples of heuristic functions used in practical planning systems. Subsection 2.2.6.2 discusses heuristic approaches in plan-space planning and details a number of specific examples.

2.2.6.1 State-space heuristic search

Whilst many of the early planning systems included some form of search guidance, this was usually hand-coded and occasionally problem specific. Two systems, UNPOP [44] and HSP [45], that pioneered the use of relatively simple and automatically generated heuristic functions for state-space search, influenced the direction of planning research for many years.

Both UNPOP and HSP used a relaxation of the original planning problem that ignores the delete effects of all operators. This relaxation is equivalent to assuming that all positive (add) effects of an action remain true forever following the execution of that action. However, this relaxation alone is insufficient to make the problem tractable and an additional relaxation, *subgoal independence*, is assumed. Finding an optimal solution to this form of relaxed problem is computationally (NP) hard [5]. Despite this, the resulting theoretical heuristic value, known as h^+ [38], has been used as a standard by which to measure approximations implemented in practical heuristic based planning systems [46].

The UNPOP planner employed an exhaustive subgoal analysis, which required a re-computation of the complete subgoal hierarchy after each action. This structure was represented by a *greedy regression-match graph*, with actions near the leaves of the graph being feasible and relevant to subgoals of the original goal. An estimate of the number of actions required to achieve each subgoal could then be assigned, with this number being used as a heuristic. In addition, this approach provided a set of candidate actions for the next search step.

HSP[1] used hill-climbing search in combination with two different heuristic functions. The first, an inadmissible heuristic, is calculated from the relaxed problem. The heuristic value for a set of goal atoms is the sum of the costs of achieving the individual goal atoms. This heuristic is known as the *additive heuristic*, h_{add}, [42] and is defined as follows.

Definition 8 *The estimated cost of achieving a set of goals, G, is:*

- $h_{add}(G) \equiv \sum\limits_{g \in G} h(g)$ *where:*

  The cost of achieving a single goal, g, in a state, S, is:

- $$h(g) \equiv \begin{cases} 0 & \textit{if} \quad g \in S \\ \sum\limits_{p \in pre_A} h(p) & \textit{if} \quad \exists A : g \in add_A \\ \infty & \textit{otherwise} \end{cases}$$

The second HSP heuristic is similarly defined, except that the cost of achieving a set of goals is no longer taken to be the sum of the costs of the individual goal atoms, but is now assumed to be equal to the value of the most expensive individual goal atom. This heuristic, known as h_{max}, is admissible and is defined thus:

Definition 9 *The estimated cost of achieving a set of goals, G, is:*

- $h_{max}(G) \equiv \max\limits_{g \in G} h(g)$

Since h_{add} calculates the estimated cost to the goal state by summing the individual goal costs, thereby assuming no positive interaction between goal achieving actions, it always overestimates the total cost (is inadmissible) and is an upper bound estimate of the value of h^+. Hence, h_{add} is often uninformative.

h_{max}, by using the maximum of the cost values of the actions required to reach a goal state, assumes that the other actions will be carried out “in parallel”. Thus, h_{max} is always a lower bound estimate of the value of h^+. This can also be uninformative since, assuming a given heuristic is admissible, it is desirable to have the highest possible valued (more accurate) estimate of the cost of reaching the goal state. One way of achieving a better estimate is to use a technique such as *cost partitioning* [47].

[1]There were several subsequent HSP variants with different search and heuristics techniques.

It is possible to generalise the concept used in the calculation of h_{max} [3]. In definition 9, the distance to the set of goals is defined as the distance to an individual goal atom with the highest cost. An alternative approach is to calculate the distance to the set of goals from the distance to m-tuples of goal atoms, for any m (e.g. pairs of goal atoms, triples of goal atoms, and so on). Hence, $h_{max} = h_m$, with $m = 1$. These heuristics, h_1, h_2,. . . ,h_m, are known as *critical path* heuristics. As the value of m increases, the quality of the heuristic estimate increases, as does the computational overhead (at least $O(n^m)$, for n goal atoms). Thus, small values of m (e.g. $m \leq 2$) are typical [3].

A more complex heuristic is that used in the Fast Forward (FF) planner [38]. The FF heuristic, h_{FF}, is also a delete relaxation based heuristic. However, it relies on the use of a *relaxed planning graph* (RPG). This data structure can be considered a combination of the delete relaxation concept and a *planning graph*, as introduced in the Graphplan[1] planning system [48].

The standard planning graph consists of alternating layers of *facts* (propositions) and *actions*, with the actions at a given layer acting upon the facts at the previous layer in order to produce the facts in the following layer. This graph is a *relaxed reachability graph*, and shows the facts that "could" be true at each layer. That is, it shows, in parallel, several actions taking place in one time step. This is in contrast to a standard reachability graph, in which propositions (facts) at a given node necessarily hold at that node. The planning graph also contains details of which pairs of facts are mutually exclusive (*mutexes*) at any particular layer. Since it provides an incomplete (relaxed) condition for reachability, the planning graph is of polynomial size, and it can be constructed in time polynomial in the size of the problem input. The RPG, as used in FF, is a standard planning graph with the added relaxation that the delete effects of the actions are ignored. FF builds the RPG to the layer at which all goal propositions are reachable (satisfied). The search algorithm then regresses through the graph towards the first fact layer (initial state), constructing a relaxed plan which achieves all of the problem's goals, and recursively their preconditions. It is the length of this relaxed plan that is used as the heuristic estimate, h_{FF}:

Definition 10 *The estimated cost of achieving a set of goals, G, is:*

- $h_{FF}(G) \equiv \sum_{i=0,\ldots,m-1} |O_i|$ *where:*

[1][40] highlights the fact that, although not originally described as such, Graphplan itself can be considered a heuristic search planner with the heuristic function, h_G, encoded in the planning graph.

O_i is the set of actions chosen at time step i (i.e. $\langle O_0, \ldots, O_{m-1}\rangle$ is the relaxed plan from the RPG).

h_{FF} is an inadmissible heuristic since the relaxed plan that is taken from the RPG is not necessarily the shortest. Compared to h_{max}, which often greatly underestimates the distance to the goal, and h_{add}, which can overestimate; h_{FF} may be considered a reasonable compromise.

Other recent heuristics based on the delete relaxation and using (a form of) the RPG include the *cost-sharing heuristic*, h_{cs}, and the *pairwise max heuristic*, h_{pmax} [49]. The first, an admissible alternative to h_{max}, draws on cost-sharing techniques successfully used in probabilistic reasoning. However, the authors show that it has limited value when applied to classical planning benchmark problems. h_{pmax}, by propagating vectors of costs instead of single costs, achieves solutions of a higher quality than both h_{add} and h_{FF} whilst expanding fewer search nodes than h_{max}, and almost matching h_{max} in terms of solution quality.

There are also a number of variations of the h_{FF} heuristic planning system. One such approach, the *FF/additive* ($h_{FF/a}$) heuristic [50], obtains the relaxed plans by means of the additive heuristic instead of using the RPG. A further variant of the $h_{FF/a}$ heuristic, the *Set-additive* (h_{sa}) heuristic [51], is similar, but provides a more accurate estimate due to a more precise method of tracking the positive interactions between action preconditions.

Another system that uses $h_{FF/a}$ is the *local Steiner tree* heuristic (h_{lst}) [52]. In this approach, the authors make use of the fact that delete relaxation heuristics can be considered as instances of computing the *tree of shortest paths* approximation to the Steiner Tree Problem [53]. For h_{lst}, which is inadmissible, a plan is first computed using $h_{FF/a}$, then this plan is iteratively improved using an algorithm that eliminates non-optimal solutions to the Steiner problem. This leads to a less greedy approximation of h^+ which, over a range of benchmark domains, shows encouraging results.

The Fast Downward (FD) planner [54] first converts the planning problem into the SAS$^+$ representation (Definition 6) before calculating its (inadmissible) *causal graph heuristic*, h_{CG}. The causal graph has a node for each of the SAS$^+$ variables and represents the causal dependencies between these. The graph has a directed edge connecting a pair of nodes if either i.) the value of the destination node may be affected by an action that depends on the value of the source node or ii.) both the source and destination node are affected by an action. Formally, this can be defined as follows.

Definition 11 *The causal graph, CG(Π), for a SAS$^+$ planning task, $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$,* *is a digraph $(\mathcal{V}, \mathcal{A})$ containing an edge (u, v) iff $u \neq v$ and there exists an operator $\langle pre, eff \rangle \in \mathcal{O}$ such that eff(v) is defined and either pre(u) or eff(u) are defined.*

In order to relax the problem, cycles in the causal graph can be broken. In this way, FD preserves some of the variable dependencies, whereas making the subgoal independence assumption (e.g. h_{add}) means losing all interaction between goals. The cycles are broken by removing the inbound edges of nodes that have the lowest number of outbound edges. Hence, the edges removed affect only those variables with the least number of other variables dependent upon them. In order to calculate the heuristic value, a set of *domain transition graphs* (DTG) is required.

Definition 12 *For a SAS$^+$ planning task, $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, let $v \in \mathcal{V}$. The domain transition graph, $\mathcal{G}_v$, is a labelled digraph with a set of vertices, $\mathcal{D}$, containing an edge $(d, d^{'})$ iff there exists an operator $\langle pre, eff \rangle$ such that pre$(v) = d$ or pre(v) is undefined, and eff$(d) = d^{'}$. The edge is labelled by pre $|(\mathcal{V} \setminus \{v\})$. For each edge $(d, d^{'})$ with label L, there is a transition of v from d to $d^{'}$ under the condition L.*

The nodes in a given variable's DTG are assigned the values from that variable's range of possible values (its *domain*). The edges of the graph reflect the actions that effect change upon the given variable. These edges have an associated weight, with the weight value being the sum of the costs of that action's preconditions.

FD calculates the heuristic cost of achieving a goal by recursively calculating the costs of achieving the necessary values of the variables upon which the goal achieving action depends. It does this by dividing the problem into a series of local parts, each consisting of one variable and its parents (from subgraphs in the CG). The resulting cost values, taken from the relevant variables' DTGs, are not optimal (since this calculation would be intractable). For those variables in the CG that do not depend on other variables, the heuristic cost is simply the sum of the costs of achieving the required value of the given variable (taken from the transition costs in its own DTG). The total heuristic cost is then the sum of the costs of all of the local parts.

Definition 13 *The causal graph heuristic, $h_{CG}(s)$, is an estimate of the number of operators required to reach the goal from a state, s. This is defined in terms of the estimated costs of changing each variable v in the goal from its value in s, $s(v)$, to its value in the goal, $s_*(v)$:*

- $h_{CG}(s) \equiv \sum\limits_{v \in dom(s_*)} cost_v(s(v), s_*(v))$

An extension of h_{CG}, known as the *context enhanced causal graph heuristic* (h_{cea}) [55], uses a declarative formulation of the CG heuristic. Whereas h_{CG} operates only on acyclic graphs, h_{cea} does not require cycles in the causal graph to be removed, and has been shown to perform very well empirically [55].

Another approach to heuristic state-space search is to use *landmarks* [56]. A landmark is a fact that must be true at some point in every solution of a given planning task.

Definition 14 *For a SAS$^+$ planning task,* $\Pi = \langle \mathcal{V}, \mathcal{O}, s_0, s_\star \rangle$, *let* ϕ *be a propositional logic formula over facts,* F. *Let* $\pi = \langle a_1, \ldots, a_k \rangle$ *be a sequence of actions applicable in the initial state,* s_0, *and* $0 \leq i \leq k$. ϕ *is true at time,* i, *in* π *iff* $s_0[\![\langle a_1, \ldots, a_k \rangle]\!] \models \phi$. ϕ *is first added at time,* i, *in* π *iff* ϕ *is true in* π *at time,* i, *but not true at any time* $j < i$. ϕ *is a landmark of* Π *iff, in each plan for* Π, *it is true at some time.*

Finding landmarks and the orderings between them is a PSPACE-complete problem [57]. However, by working from a known set of landmarks (the facts in the given initial and / or goal states), and using approximations based on the RPG [38], a candidate set may be determined. Filtering these leads to a sound set of landmarks, the ordering of which may not necessarily be sound.

In order to use landmarks as a planning heuristic, one approach (FF_{vX+L}) [57] subdivides the planning task into smaller sub-tasks and searches for plans to satisfy each landmark. This technique requires an additional, final search phase to achieve some of the goal-state goals since some may have been undone during the landmark achievement phase. Whilst this technique does better focus the search, it also leads to plans that are longer than those generated by the base planner (FF) used alone. Further, this approach failed to reach a solution on a number of otherwise solvable problems due to the occurrence of dead-ends. A modified system [58] improved the plan length measure, but increased the amount of time required to find a solution.

An alternative approach, h_{LAMA} [59], which uses landmarks as an inadmissible pseudo-heuristic, used a similar procedure to that above, but extracted additional landmarks and orderings from the planning problem's DTG (Definition 12). By using the landmark information whilst searching for a solution to the original planning task, the authors achieved good results on a number of problems. The heuristic technique used was one which estimates the distance to the goal from a state, s, by counting the number of landmarks, l, not yet satisfied. Where n is the total number of landmarks, m is the number of landmarks that are *accepted*, and k is the number of landmarks that are *required again*; the heuristic estimate (of the number of landmarks still to be achieved) is $\hat{l} \equiv n - m + k$. By combining h_{LAMA} with other heuristics and by making use of *preferred operators* [36][1], the h_{LAMA} results were further improved.

Additional landmark based (admissible) heuristics, h_L and h_{LA} [43], build on the h_{LAMA} technique. These heuristics assign costs to the landmarks, and costs to the actions that achieve them, and then apply a cost-partitioning scheme. Both h_L and h_{LA} show good results in terms of the number of nodes expanded, number of problems solved, and solution time over a range of problem instances [43].

All of the above landmark heuristics rely upon relaxation based landmarks, which give no guarantee as to the completeness of the set of landmarks found. However, viewing the delete-relaxation of a planning problem as an instance of an AND/OR graph [49] allows for the declarative definition of the complete set of *causal landmarks* [60]. A landmark heuristic method [61] making use of this technique improves considerably the accuracy of landmark based, admissible heuristics.

By considering the relationships between the different types of heuristic (p.23 / p.24), it is possible to determine a compilation scheme for translating between classes of heuristic [62]. An admissible heuristic which *dominates* another provides a better heuristic measure. The results show that[2], for example, landmark heuristics dominate additive h_{max} heuristics, and additive critical path heuristics (with $m \leq 2$) strictly dominate both landmark heuristics and additive h_{max} heuristics. From the proofs of these results, the authors obtained a new admissible heuristic, the *landmark cut heuristic* (h_{LM-cut}).

h_{LM-cut} is calculated by applying the reduction from additive h_{max} to landmarks using standard h_{max} with no cost partitioning. This leads to a heuristic value that is based on iteratively computing landmarks that are *cuts* in justification graphs [63]. This is continued until no further non-zero valued cuts can be made. As is the case with other heuristics, h_{LM-cut} may be considered an approximation to the intractable optimal relaxation heuristic, h_+, and empirical results show that it gives very good approximations to h_+. When used within an optimal planning algorithm based on A^*, this heuristic is competitive with the state of the art [63] and is an important result in terms of the scalability of optimal planning.

Having reviewed some recent methods for generating heuristics for state-space search, the following subsection provides an overview of the use of heuristics in plan-space planning. It continues the discussion of partial order planning started in sec-

[1]Preferred operators are operators that improve the heuristic estimate from a given state. FF also makes use of *helpful actions*.

[2]Some restrictions apply.

tion 2.2.5.2, and introduces several systems that integrate state-space techniques into plan-space heuristic search.

2.2.6.2 Plan-space heuristic search

Plan-space planning (section 2.2.5.2) operates in a search space of partial plans, and attempts to find a solution plan by systematically repairing *flaws*[1] in a chosen partial plan. Thus, the search choices in this paradigm are: Which partial plan to select; and, within the selected plan, in which order should the flaws be chosen for repair.

As previously mentioned, one method of choosing which plan to repair is to use the number of open preconditions as a heuristic guide. This estimate is inadmissible and does not accurately reflect the number of actions required due to the interactions (both positive and negative) between plan steps. An improved estimate can be achieved by subtracting from this number the number of open preconditions that are satisfied by literals in the initial state. However, the resulting figure is still not sufficiently accurate.

More recent partial-order planners employ methods initially developed for state-space systems to provide a better heuristic measure. The REPOP system [64] uses such a heuristic for ranking partial plans.

Definition 15 *REPOP represents a partial plan, $\mathcal{P}$, as a five tuple $\langle \mathcal{A}, \mathcal{O}, \mathcal{L}, \mathcal{OC}, \mathcal{UL} \rangle$, where:*

- *$\mathcal{A}$ is a set of ground actions (strictly, $\mathcal{A}$ is a set of steps, with each step mapped to an action instance);*
- *$\mathcal{O}$ is a set of ordering constraints over $\mathcal{A}$;*
- *$\mathcal{L}$ is a set of causal links, where a causal link, $a_i \xrightarrow{p} a_j$, represents a commitment that precondition, p, of action, a_j, is supported by an effect of action, a_i;*
- *$\mathcal{OC}$ is a set of open conditions. An open condition is of the form (p, a), where $p \in Prec(a)$ ($Prec(a)$ is the precondition list of action, a), $a \in \mathcal{A}$, and there is no causal link $b \xrightarrow{p} a \in \mathcal{L}$;*
- *$\mathcal{UL}$ is a set of unsafe (causal) links. A causal link, $a_i \xrightarrow{p} a_j$, is considered unsafe if there exists an action, $a_k \in \mathcal{A}$, such that: $\neg p \in Eff(a_k)$ (where $Eff(a_k)$ is the effects list of action, a_k) and $\mathcal{O} \cup \{a_i \prec a_k \prec a_j\}$ is consistent. In this case, a_k is said to threaten the causal link, $a_i \xrightarrow{p} a_j$;*
- *A solution plan in REPOP is a partial-plan with no flaws. That is, $\mathcal{OC} = 0$ and $\mathcal{UL} = 0$.*

[1]Flaws may be either open-preconditions (a precondition of an action in the partial plan without a causal link) or threats (an action that interferes with an existing causal link).

In order to take into account the interactions between plan steps, the REPOP heuristic makes use of the serial Planning Graph (section 2.2.6.1) [65]. The planning graph is constructed from the initial state, I. Let $lev(p)$ be the index of the level in the planning graph that a proposition, p, first appears, and $lev(S)$ be the index of the first level at which all propositions in S appear (where $S = \{p|(p,a) \in \mathcal{OC}\}$). Let p_s be the proposition in S such that $lev(p_s) = max_{p_i \in S} lev(p_i)$. It is assumed that p_s is possibly the last proposition in S that is achieved during execution. Let a_s be an action in the planning graph that achieves p_s in the level, $lev(p_s)$. p_s can be achieved by adding a_s to the plan. Adding a_s in this way modifies the set of goals still to be achieved to be $S' = S + Prec(a_s) - Eff(a_s)$. The cost of S can be expressed in terms of the cost of a_s and S':

$$\text{cost(S)} \equiv cost(a_s) + cost(S + Prec(a_s) - Eff(a_s)) \qquad (2.1)$$

where $cost(a_S) = 1$ if $a_S \notin \mathcal{A}$ and 0 otherwise. The REPOP relaxation heuristic is then defined as follows.

Definition 16 *The REPOP relaxation heuristic, $h_{relax}(\mathcal{P})$, is:*

- $h_{relax}(\mathcal{P}) = cost(S)$ *where:*
  $S = \{p|(p,a) \in \mathcal{OC}\}$*), and cost*$(S)$ *is computed using equation 2.1;*

REPOP, in addition to using the h_{relax} heuristic, introduced two further improvements on UCPOP, upon which it was based. These were the use of a *reachability analysis* [65] in order to identify further threats to causal links (in addition to the standard threat of an action negating a specific proposition), and the use of disjunctive ordering constraints in order to postpone commitment to an ordering with which to protect threatened links. The combination of these techniques led to REPOP dominating both UCPOP and Graphplan on a number of benchmark planning domains.

Another partial-order planner based on UCPOP is VHPOP [66]. This system uses a form of the h_{add} heuristic adapted for use in plan-space planning. The heuristic is modified to allow potential reuse of actions that are already in a given partial plan.

VHPOP defines a partial plan in a similar way to that used in UCPOP (Definition 7). Working only with grounded actions allows the set of binding constraints on free variables, $\mathcal{B}$, to be ignored. Remembering that subgoals are assumed to be independent (in the calculation of $h_{add)}$), and letting, for a given literal, q, $\mathcal{GA}(q)$ be the set of ground actions having an effect that unifies with q. The cost of the literal, q, can then be defined as follows.

Definition 17 *The estimated cost of achieving a literal, q, is:*

- $$h_{add}(q) \equiv \begin{cases} 0 & \textit{if} \quad q \textit{ unifies with a literal that holds initially} \\ min_{a \in \mathcal{GA}(q)}\, h_{add}(a) & \textit{if} \quad \mathcal{GA}(q) \neq 0 \\ \infty & \textit{otherwise} \end{cases}$$

where the cost of an action, a, is:

- $h_{add}(a) = 1 + h_{add}(Prec(a))$

The additive heuristic for a partial plan, π, can then be defined as follows.

Definition 18 *The additive heuristic cost function for a partial plan, π, is:*

- $h_{add}(\pi) \equiv \sum_{\xrightarrow{q} a_i \in \mathcal{OC}(\pi)} h_{add}(q)$ *where $OC(\pi)$ is the set of open conditions.*

As is the case for state-space planning, $h_{add}(\pi)$ is inadmissible. VHPOP attempts to improve the heuristic estimate by taking into account the positive interactions between action steps. By assuming that an open condition, $\xrightarrow{q} a_i$, could potentially be resolved by linking to an effect of an existing action, a_j, the modified heuristic, $h^r_{add}(\pi)$, allows for action *reuse*. To be able to reuse an existing action's effect requires that the existing action (a_j) is ordered before the action that requires it (a_i). $h^r_{add}(\pi)$ is defined as follows.

Definition 19 *The VHPOP heuristic, $h^r_{add}(\pi)$, is:*

- $$h^r_{add}(\pi) \equiv \sum_{\xrightarrow{q} a_i \in \mathcal{OC}(\pi)} \begin{cases} 0 & \textit{if} \;\; \exists a_j \in \mathcal{A} \textit{ s.t. an effect of } a_j \textit{ unifies with } q \\ & \textit{and } a_i \prec a_j \notin \mathcal{O} \\ h_{add}(q) & \textit{otherwise} \end{cases}$$

$h^r_{add}(\pi)$ can give an over optimistic estimate and is also not admissible, since the cost still relies on the same measure ($h_{add}(q)$) for those open conditions that are not resolved by "reusable" actions. However, $h^r_{add}(\pi)$ was a considerable improvement over previous approaches.

In addition to the modified h_{add} heuristic, VHPOP also introduced a number of flaw selection strategies. By using several instances of the planner running concurrently, each with a different flaw selection strategy, VHPOP outperformed previous partial-order planners over a number of benchmark problem domains.

Following the resurgence of interest in partial order planning stimulated by the development of REPOP and VHPOP, AI planning research in recent years has focussed more on heuristically guided state-space search. However, a recent approach [67] which combines landmarks (a state-space technique) with the plan-space paradigm highlights the continuing research interest in plan space planning.

This section has discussed the use of heuristics in both state-space and plan-space planning. Whilst the use of an accurate heuristic considerably reduces the amount of search required to find a solution, the efficacy of a given heuristic based planner will necessarily be affected by the particular structure of a given planning problem [39] and by the other parts of the search algorithm [68]. For example, Hoffmann [39] identifies several key characteristics of the topology of the respective search spaces of many standard benchmark domain instances; these include *dead ends*, *local minima*, and *benches* (areas where all states have the same heuristic value). Further, Helmert [68] shows that, even under the assumption of an almost perfect heuristic estimator, standard search algorithms such as A^* must necessarily visit an exponential number of search nodes. Hence, any chosen heuristic / search combination is unlikely to be accurate over a very wide range of different problem domains. Consequently, successful planners often incorporate additional search space pruning and search guidance techniques.

Whilst not directly related to the work described in the following chapters of this thesis document, such additional pruning and guidance methods play an important part in many modern planning systems. They include *helpful actions* [38], as used in the FF planner; the *preferred operators*, *multiple heuristics*, and *deferred evaluation* [54] found in the FD system; and *symmetry* detection and exploitation [69]. Further important planning techniques include *hierarchical planning* [70]; *domain analysis* methods (e.g. *control rules* [71] and *machine learning* [72]); *macro actions* [73]; and *generic type inference* [74].

Having completed the background discussion of classical planning, and heuristically guided search in planning, the following section introduces CSPs. CSPs are central to this work since here the planning task is reformulated as a constraint problem before being solved under that paradigm.

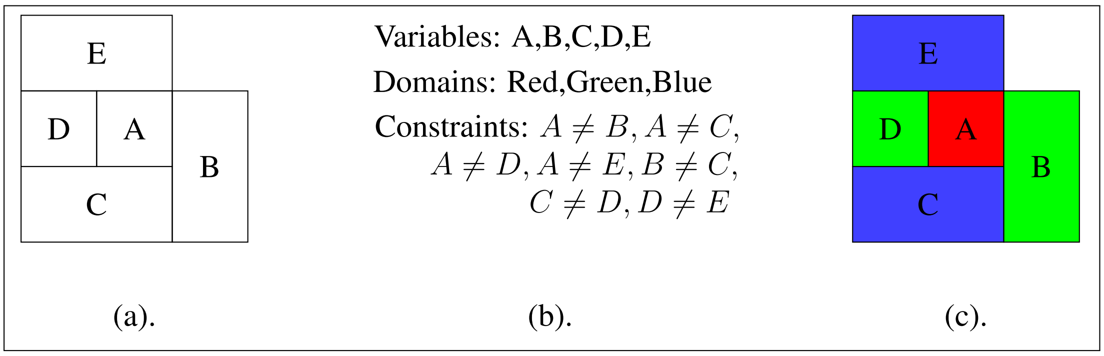
2.3 Constraint Satisfaction Problems

This section of the thesis introduces CSPs. It first defines the basic concepts, introduces CSP modelling, then discusses various solution techniques and heuristic methods.

Constraint programming is a declarative paradigm that has been called the holy grail of programming [75]. Once modelled using a constraint framework, the declarative description of a problem is processed by a constraint solver, with a solution potentially determined by means of inference alone. This allows for higher level modelling and a separation of problem description from the method of solution. CSPs also benefit from a "natural" representation of many real-world problems and, in addition, many computer science problems may be viewed as particular types of CSP (e.g. machine vision [76], scheduling [77], SAT [78]). Consequently, the CSP approach has been used in many complex commercial and academic applications. These include transportation, medical and health services, telecommunications, broadcasting, financial services, electricity distribution, manufacturing processes, and the construction of aeroplanes and spacecraft [79].

Informally, a CSP [80] may be viewed as a set of *variables*, to which are assigned *values* that satisfy given *constraints*. A solution of a CSP is the *assignment* of a value to each of the variables, such that all of the constraints are satisfied. Depending on the requirements of a particular application, the task may simply be to determine whether such a solution exists. Alternatively, it may be necessary to find the actual solution, to find multiple (or all) solutions or to find the best solution according to a given metric.

A classic, simple illustrative example of a CSP is map colouring [9]. In this type of problem, a map showing a number of states or countries requires a different colour for each adjacent region, with the colours taken from a limited range of possibilities. Figure 2.7 (a) illustrates such a map with five regions.



Variables: A,B,C,D,E
Domains: Red,Green,Blue
Constraints: $A \neq B, A \neq C,$
$A \neq D, A \neq E, B \neq C,$
$C \neq D, D \neq E$

(b).

Figure 2.7: Map colouring problem represented as a CSP.

Assuming the map colouring problem of Figure 2.7 (a) has three colours available, the definition of the problem in the form of a CSP is given in Figure 2.7 (b). The variables (A, B, C, D, E) represent the names of the regions, with each variable's range of possible values contained in the domain of that variable $(red, green, blue)$. Hence, each region could possibly be coloured red, green or blue. However, the constraints ($A \neq B, A \neq C, \ldots, D \neq E$) restrict the potential values available to each region. It should be noted that the constraints do not assign any values, they merely state what is permitted (or not, in this case). That is, the constraints state that variable A cannot hold the same value as variable B (region A can not have the same colour as region B) and so on. With the problem stated in this way, a constraint solver would then be able to start assigning values to the variables in order to find a solution which satisfies all of the constraints. One such solution is shown in Figure 2.7 (c). The map colouring problem is an example of the use of binary constraints. That is, all constraints contain only two variables.

A more formal definition of a CSP, as used in this work, is as follows.

Definition 20 *A constraint satisfaction problem, $\mathcal{M}$, is a triple, $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}) \rangle$, where:*

- *$\mathcal{X}$ is a finite set of variables, $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$;*
- *$\mathcal{D}$ is a finite set of domains for those variables, $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$;*
- *$\mathcal{C}$ is a set of constraints, $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$, where each constraint defines a predicate which is a relation over a particular subset of X.*

Early work in the area of constraint satisfaction [76] described the CSP as a *network* of constraints. This network can be represented by a *primal constraint graph* [9]. The graph captures the structure of the problem and can be used as part of the solution process. The constraint graph's nodes represent the CSP variables, and the arcs connect nodes that are within the *scope* of a given constraint. Figure 2.8 shows the primal constraint graph for the map colouring problem of Figure 2.7.

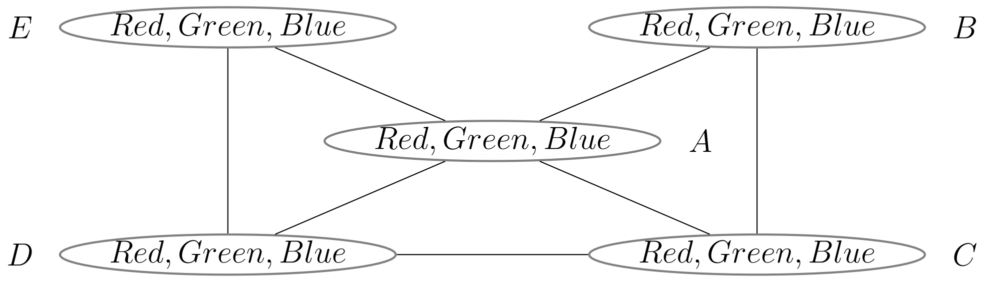


Figure 2.8: Primal constraint graph for the example map colouring problem of Figure 2.7.

Whilst the primal constraint graph is used, and is well-defined, for both binary and non-binary constraints, a more accurate representation for non-binary constraints is the *constraint hypergraph* [10]. In this structure, the nodes again represent the CSP variables, and the *hyperarcs* (regions) connect all variables in each constraint (i.e. the scope of each constraint).

Another important graphical representation of CSPs is the *dual constraint graph* [9]. Here, the nodes represent the scope of the constraints, with a (labelled) arc between nodes showing which variables are common to the connected nodes (constraints).

The dual constraint graph can be used to transform a non-binary constraint network into a particular type of binary network known as the *dual problem* [81]. In the dual problem, the constraints from the primary problem become variables (*c-variables*). The c-variables' domains are the sets of all allowed value combinations as defined by the original set of constraints. Two connected c-variables are linked by the restriction that any shared (original) variables must have the same values. That is, the c-variables are bound by equality constraints. Thus, any CSP containing non-binary constraints can be transformed into a CSP with binary constraints, and hence solved using binary CSP methods[1].

Definition 21 *A CSP is called binary if all of its constraints are either unary or binary (i.e. a given constraint only involves a single variable or a pair of variables respectively).*

In the following sections, it is assumed that the domain of each CSP variable is finite and discrete, and that all constraints are binary.

2.3.1 CSP Modelling

To ensure that a CSP solver achieves the best possible results, the CSP should be formulated according to some basic principles [11]. Thus, constructing an efficient CSP *model* is considered to be an important part of constraint programming [83], [84], [85].

Among the modelling decisions to be made is the choice of variables and domains. That is, which part of the original problem will be represented by the CSP variables and which part will constitute the available values (domains). For example, in the map colouring problem (Figure 2.7), the variables represented the regions and their domain values denoted the colours. The problem could easily have been modelled differently, with the variables representing the colours and their respective domains containing the available regions for colouring. This type of modelling decision depends upon the

[1]In addition to the dual graph translation technique, particular constraint solvers may use other optimised methods to process some types of non-binary constraint [82].

modelling *viewpoint* [86] taken. Since the CSP search space is the set of all possible total (variable / value) assignments and grows exponentially with the problem size ($SearchSpaceSize = DomainSize^{No.ofVariables}$), it is prudent to take steps to reduce the impact that modelling decisions have upon this. Further, it is possible to use more than one viewpoint at the same time, combining these using *channelling constraints* [87] to achieve better performance.

Another important modelling choice is the type of constraint and the means by which this is represented. This not only affects the memory requirements of the CSP definition and operation, but may also have an impact on how much propagation is achieved. Generally, a given constraint solver will have, associated with each type of constraint, a particular propagation algorithm. Therefore, by choosing the appropriate constraint type, the modeller can achieve better results for a given problem. For example, it is possible to use either a series of binary $\neq$ constraints or a global *allDifferent* constraint. Knowledge of the differing constraint propagation behaviours and costs will allow an informed choice to be made.

In terms of the representation scheme for the constraints, two methods are *intensional* and *extensional* [11]. In the former, constraints are described implicitly using relation symbols (e.g. $x_i \neq x_j$). The latter representation requires the constraints to be stated explicitly, often in tabular form, listing the set of permitted / forbidden tuples.

2.3.2 Finding Solutions to CSPs

Having modelled a given problem as a CSP, by choosing a set of CSP variables, their respective domains, and associated constraints, it is then necessary to choose a solution technique. Various strategies have been developed to solve CSPs, and this section of the thesis discusses some of these.

Since one of the aims of the constraint programming paradigm is to separate the problem model from the solving technology, it has been suggested [88] that the constraint based approach can be described thus[1]: Solution = Logic + Control.

Hence, the declarative description (constraint model) of the problem is the *logic*. Added to this is the control function, which allows the solver to infer a solution. As mentioned in section 1.2, in an ideal situation, a CSP can be solved simply by reasoning from what is already known in the declarative model, without recourse to any form of search technique. However, in realistic problems the amount of reasoning (inference) required to solve the CSP by this method can lead to an exponential increase in

[1]This description is derived from work relating to the analysis of algorithms [89].

the number of constraints. Even with the restriction of binary constraints (and finite domains), the full form of this preprocessing is an NP-complete problem [9]. Therefore, any algorithm used could, in the worst case, take time that is exponential in the problem size. To reflect this inherent complexity, a second statement can be made: Control = Reasoning + Search.

This second statement highlights the need for some form of search, in addition to the standard inference techniques used in constraint programming. Therefore, the solution to a constraint formulation of a problem, a *constraint program*, can be considered to be achieved by a combination of **Logic** - the declarative model of the problem; **Reasoning** - the CSP inference / propagation algorithms that reduce the amount of search required; and **Search** - the particular search strategies and heuristics required to find a solution quickly. The formal definition of a solution to a CSP is given below.

Definition 22 *A solution to a CSP, $\mathcal{M}$, is an n-tuple, $\sigma = (v_1, v_2, \ldots, v_n)$ such that $v_i \in d_i$ and the values of the variables $x_i = v_i$, for $1 \leq i \leq n$, meet all of the constraints in $\mathcal{C}$. A CSP is said to be consistent if such a solution, σ, exists.*

The most popular practical approaches to solving CSPs use a limited amount of inference, in combination with search. Inference methods are introduced in the following section, with some basic search techniques presented in section 2.3.2.2. Heuristic methods for determining the search order of variables, and their potential values, are discussed in sections 2.3.2.3 and 2.3.2.4 respectively.

2.3.2.1 Inference Techniques

This section discusses methods for reducing the CSP search space and finding solutions by means of inference. The use of such techniques is central to the CSP paradigm.

In constraint programming, the reasoning part of the solution process is variously referred to as *maintaining consistency*, *domain filtering*, *pruning*, *inference* or *constraint propagation* [11]. The general approach is to preprocess the problem in order to generate a new, more explicit representation of the same problem. This is achieved by adding constraints that are inferred from those in the original problem model. For example, consider a CSP with three variables, A, B, C, each with domains, $Dom = \{Black, White\}$, and with three constraints: (Con_1) $A = B$, (Con_2) $B = C$, and (Con_3) $A \neq C$. Using constraints Con_1 and Con_2 it is possible to infer that $A = C$. This new inferred constraint conflicts with Con_3 ($A \neq C$) and hence, it is possible to reason that this problem has no solution since the set of constraints is *inconsistent*. Such newly inferred constraints may constrain variables that were not previously connected by constraints or may *tighten* existing constraints. Having added the new constraints, the process continues, with the newly inferred constraints being used to infer further constraints, which in turn are added to the model. In this way the inferences made are propagated through the constraint model. This process of adding inferred constraints to the constraint network leads to a more explicit constraint network [9], with the augmented network being *equivalent* to the original problem's constraint network.

Definition 23 *Two constraint networks are equivalent if they are defined on the same set of variables and express the same set of solutions.*

The inferred constraints in the equivalent network(s) can be considered *redundant*. That is, redundant relative to a constraint network if, when the constraint is removed, the set of solutions remains the same.

Definition 24 *A constraint, Con_{ij}, is redundant relative to a constraint network, N', if and only if N' is equivalent to another constraint network, N, when Con_{ij} is removed.*

Since a solution to a CSP is a full consistent assignment (of values to variables) which satisfies all of the relevant constraints, it is possible to make consistent small parts of the constraint network (subnetworks containing i variables) and then extend these to a greater number of variables. The smallest such subnetwork contains a single variable (i.e. one node of the network, $i = 1$), the largest contains i variables, where $i = n$, with n being the total number of variables in the CSP.

For each size of subnetwork there exists a consistency technique[90], [91] which enforces a specific level of consistency (constraint propagation) on the given constraint network. These include *node consistency* for $i = 1$, *arc consistency* for $i = 2$, and *path consistency* for $i = 3$.

Node consistency (NC) is the simplest of the consistency techniques considered here. It operates on single variables and their associated unary constraints (i.e. single nodes in the CSP's constraint network / graph). Where a value in a given variable's domain is inconsistent with a constraint on that variable, NC will remove that value since it will cause a failure in every partial solution in which it is included. For example, a CSP that includes node, X, representing variable X having domain, $d_X = \{1..7\}$, and having a unary constraint, c_X ($X > 4$) is not NC. Removing values 1 to 4 from the domain makes the node consistent. Applying NC to one node achieves *local* NC, whilst applying it to all nodes in a network achieves *global* NC. The global NC algorithm visits each node only once, since unary constraints affect only one variable and local changes cannot have an impact on any other variables.

Arc consistency (AC) removes, from a given variable's domain, those values that are inconsistent with any binary constraints on that variable. In this way, the constraint is made AC by propagating the domain from one variable to the other. This process is repeated in the opposite direction. Thus, an arc, $arc(x_1, x_2)$, is arc consistent if, for every value, d_{1_x}, in the current domain of x_1 which satisfies the constraints on x_1, there exists a value, d_{2_y}, in the current domain of x_2 such that $x_1 = d_{1_x}$ and $x_2 = d_{2_y}$ satisfies the binary constraint between x_1 and x_2. AC can be shown visually using a *matching diagram* (not to be confused with the complete CSP network / graph representation). For example, Figure 2.9 describes the AC of an arc, $arc(X, Y)$, connecting two variables, X and Y, in a CSP network, each with domain, $\{1..3\}$, and having a binary constraint, Con_{XY} $(X < Y)$.

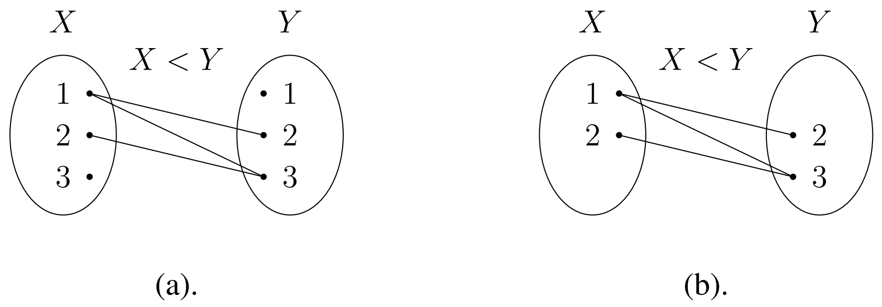


Figure 2.9: Arc consistency matching diagram for part of a CSP containing two variables, X and Y. (a) Initial domains showing that the variables are not arc consistent. (b) Filtered domains showing the variables are now arc consistent.

From Figure 2.9 (a) it is clear that, initially, the domain of variable X contained a value that was not consistent with the constraint $X < Y$ (i.e. 3). Also, the domain of variable Y initially contained an inconsistent value (i.e. 1). Removing each of these satisfies the conditions for AC. Namely, that each value in the domain of either variable has a corresponding value in the domain of the other variable which satisfies the connecting constraint, $X < Y$.

Whereas global NC can be achieved with one pass of the algorithm on each node, global AC requires the AC algorithm to be run more than once per arc. Having reduced the domain of a given variable, x_1, any previously visited arc which includes that variable (e.g. $arc(x_2, x_1)$) must be revisited, since there may be values included in the domain of the other variable (x_2) that are no longer compatible with the remaining values in the pruned domain of x_1.

Various algorithms have been developed to achieve AC. For example, the AC-1 algorithm [90] establishes AC by (re)visiting all arcs until there is no further change in the domain of any of the CSP variables. This approach has complexity, O(bnk^3), where n is the number of variables in the constraint network, k is the size of the largest domain, and b is the number of binary constraints. This complexity measure can be reduced by improving on AC-1. This is possible since it is not necessary to revisit all arcs in the network if only a small subset of the CSP's variables' domains have been reduced.

By using a queue containing the constraints to be processed, AC-3[1] [90] reduces the time complexity of achieving AC as compared to AC-1. Each of the variable pairs included in the CSP's constraints are initially stored in the queue twice, one for each (directional) arc. When processed, each ordered pair is removed, and only replaced in the queue if the second variable's domain has been changed as a result of the processing of adjacent constraints. Hence, under AC-3, each constraint is processed a maximum of $2k$ times. This is due to the reduction by one in the size of the domain of the variables in the scope of a given constraint each time that constraint is replaced in the queue. With, b, binary constraints, each with processing time, O(k^2), the time and space complexity of AC-3 are O(bk^3) and O(b) respectively.

Since AC-3 checks, for each variable involved in a constraint, whether the remaining values in the relevant domains are consistent with that constraint, it is possible that such consistency checks are repeated multiple times. That is, in processing the problem, the algorithm does not record (or learn) anything as it progresses. Hence, some portion of the domain checking is potentially redone. This makes AC-3's time complexity non-optimal [11].

In the worst case, simply verifying the AC of a CSP network requires bk^2 operations. This, therefore, is the best possible performance for any AC algorithm. More recent versions of AC-3 ([92], [93]) achieve this.

The AC-4 algorithm [94] also achieves optimality by operating at the level of variables (*fine grained*) [92]. This is in contrast to the previous AC algorithms which operated on arcs (i.e. constraints) and are thus considered *course grained*. AC-4 carries out an initial preprocessing phase and stores the relevant information in order to prevent unnecessary repetition of effort during the constraint propagation phase. This

[1]AC-2 is an advance on AC-1, but does not use a queue.

preprocessing phase requires all of the constraints to be checked for consistency and, hence, takes bk^2 time, contributing to the time complexity of AC-4 being O(bk^2) . However, considering the best case, AC-1 and AC-3 may perform better than AC-4. That is, in the best case, AC-1 and AC-3 will require only bk operations since the problem may already be AC, but AC-4 will still require bk^2 operations to initialise. Also, the memory requirements of AC-3 can be lower, since the space complexity of AC-4 is O(bk^2) due to the special data structures required for the storage of variables' *supported values*.

AC-5 [95] is a generic AC algorithm parameterised on two procedures. This approach allows AC-3 and AC-4 to be implemented, but also allows for an AC algorithm that runs in O(bk) time. However, this algorithm operates only on *functional* and *monotonic* constraints.

Whilst requiring less memory than AC-4, AC-6 [96] maintains its worst case time complexity. The reduction in space complexity is achieved by not recording all supports for domain members (as in AC-4), but only storing one support for each domain element per constraint. The resulting space complexity is O(bk), and the worst case time complexity remains optimal at O(bk^2).

AC-7 [97], an extension of AC-6, takes advantage of a property common to all binary constraints in order to reduce the number of constraint checks (*cost*) required to achieve AC. Having found a supporting value from the domain of the second variable in a constraint, given a value selected from the domain of the first, AC-7 then exploits *bi-directionality*. That is, for an arc, $arc(x_1, x_2)$, having found a supporting value, $x_2 = d_{2_y}$, for $x_1 = d_{1_x}$, it is possible to infer that $x_1 = d_{1_x}$ is a supporting value for $x_2 = d_{2_y}$. In this way, meta-knowledge is used to infer, instead of compute, support information. Negative support information can also be inferred; having found that $x_2 = d_{2_y}$ does not support $x_1 = d_{1_x}$, it is possible to infer that $x_1 = d_{1_x}$ is not a supporting value for $x_2 = d_{2_y}$. By making use of bi-directionality, AC-7 achieves a space complexity of O(bk), with an optimal upper bound of O(bk^2) for worst case time complexity whilst providing good results on average [97].

Adaptations of the techniques used in AC-7 allow for the use of bi-directionality on the previously discussed course-grained algorithms [98]. The resulting algorithms, AC-3.2 and AC-3.2*, exploit bi-directionality on positive constraint checks partially and fully, respectively. AC-3.3 and AC-3.3* do the same for negative constraint checks. Empirical testing shows that AC-3.3 dominates other AC techniques.

An alternative to the bi-directionality approach of AC-7, as a means of reducing the *cost* of checking constraints for achieving AC, is the technique of ordering the propagation list. One variant sees the *cheapest* arcs visited first, another approach prioritises the arcs that prune the most [99].

Regardless of which of the above AC algorithms is used, each constraint arc connecting two variables is considered separately. That is, for two variables, x_1 and x_2, arcs, $arc(x_1, x_2)$ and $arc(x_2, x_1)$, are each considered. For certain CSPs, those whose CSP network forms a tree, a weaker arc consistency test is sufficient to allow for backtrack-free search. By ordering the variables (i.e. nodes) from the root of the tree to the leaves, and carrying out a *directional arc consistency* (DAC) [9] check on each arc, it is possible to then assign values to the CSP variables in that same order without backtracking over a previous choice. Thus, DAC ensures that, for each value assigned to a given parent node, there exists a consistent value available for all child nodes. Hence, the difference between AC and DAC is that DAC only checks an arc, $arc(x_a, x_b)$, if $a < b$, whereas AC checks both $arc(x_a, x_b)$ and $arc(x_b, x_a)$. Consequently, DAC is less computationally intensive than AC-3, and requires less memory than AC-4. However, the use of DAC depends on the shape of the constraint network and its use is, therefore, domain dependent.

For complex real-world problems containing non-binary CSPs, specific versions of popular AC algorithms have been developed[1], and it is these that are commonly used in commercial CSP solvers [11]. One such approach is the *generalised arc consistency* (GAC) [100] version of the AC-3 algorithm. Where the cardinality of the constraints in real-world problems being processed by GAC is large, the efficiency of the algorithm will be negatively affected. For this reason, more efficient *global constraint* processing can be used. Global constraints (e.g. [101]) make use of specialised, domain-dependent filtering and domain reduction methods.

Although the various AC techniques remove many inconsistent values from the domains of the CSP variables, AC is not sufficient to ensure that an instantiation of the variables from the remaining (pruned) domains will lead to a solution, or even to prove that no solution exists. For example, although the CSP network of Figure 2.10 has been made AC, there is no assignment of values to variables that satisfies all of the constraints. Thus, having made a CSP network AC, there are three possible outcomes: 1. The size of the domain of one of the variables is reduced to zero (becomes empty). This means there can be no solution. 2. The size of the domain for all variables is reduced to one. Therefore, assiging each variable its only available value leads to the

[1]As previously mentioned, it is possible to convert a non-binary CSP to a binary representation and, hence, use binary consistency algorithms. However, in larger problems, this is often not an efficient approach.

only solution. 3. Any other outcome means that there may still be inconsistencies in the network, and that it is impossible to say whether or not there is a solution.

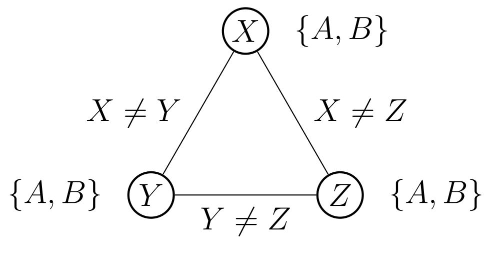


Figure 2.10: An arc consistent constraint network with no solution (No assignment of values to variables will simultaneously satisfy all of the constraints).

As is clear from Figure 2.10, the reason AC is not complete for deciding consistency is that the inferences made are based on single arcs. Using a higher number of arcs would allow more inconsistencies to be removed from the CSP network.

Local path consistency (PC) considers two arcs (i.e. three variables) and requires that any consistent solution to a subnetwork consisting of two variables (an arc) is extendable to any third variable. That is, for every pair of variables, x_1 and x_2, that satisfy their respective connecting constraint, there exists a value for each variable on a path between those variables, such that all constraints along the path are satisfied. A CSP is globally PC if and only if all paths of length two are path consistent [76]. This means that, in contrast to AC which works with pairs of variables (an arc or constraint), PC algorithms operate on triples of variables (i.e. paths of length two).

The algorithms that achieve PC can be seen as extensions of those that achieve AC. PC-1 [90], a brute force approach, is the PC equivalent of AC-1. Likewise with PC-2 [90] and AC-3. Optimal algorithms PC-3 [94] and PC-4 [102] take the same approach as AC-4, preprocessing to produce lists of supported values. The modifications made to AC-4 to give AC-6 are used to improve PC-4, giving PC-5 [103] and PC-6 [104]. Simplified practical implementations of PC-6, known as PC-7 [105] and PC-8 [106], exhibit good empirical results. Bi-directionality, as applied in AC-7, is used to achieve PC in the PC-5++ algorithm [103].

Although PC improves on AC by removing more inconsistencies from the constraint network, it also requires significant computational resources [11]. As compared with AC, the ratio of complexity of the PC algorithm to the reduction in the number of inconsistencies gained from it, is higher. Also, since PC eliminates pairs of values from constraints, there is a requirement to have an extensional representation of those constraints (i.e. in order to remove individual pairs of values, those values must be listed extensionally). A further disadvantage of the PC algorithm is that it can alter the connectivity of the constraint network by adding constraints in order to enforce consistency.

In addition to the increased computational overhead, PC (like AC) is unable to ensure that a complete assignment of values to variables, following completion of a given PC algorithm, will lead to a solution. Likewise, PC is not sufficient to guarantee that there is no solution. For this reason, consideration has been given to consistency techniques that operate on a greater numbers of variables.

K-consistency (and *strong K-consistency*) [107] is a generalisation of the above consistency approaches. A constraint network is K-consistent if, for all assignments of values to variables for $K-1$ variables that satisfy the set of constraints connecting those variables, it is possible to assign a consistent value to a K-th variable. That is, having chosen a value for $K-1$ variables, a value can be chosen for the Kth variable, and the CSP remains consistent (i.e. it becomes K-consistent).

Strong K-consistency requires M-consistency, for all $M \leq K$. Therefore, node consistency is strong K consistency for $K = 1$, arc consistency is strong K consistency for $K = 2$, and so on.

Algorithms that enforce K-consistency for higher values of K increasingly suffer from problems similar in nature to those mentioned above in relation to PC. Further, nothing less than strong K-consistency for $K = n$, where n is the total number of variables in the CSP, allows a solution to be found using inference alone (without the need for search). However, the worst case time complexity for any algorithm achieving K-consistency is exponential in K. For these reasons, although algorithms exist for achieving strong K-consistency with $K > 2$, these are generally not used in practice.

Summarising so far, incomplete inference techniques, such as AC and PC, *may* determine whether a problem is inconsistent. However, this is not, in general, guaranteed. Therefore, having achieved limited consistency in a constraint network, it is normally the case that some form of search will be required to find a solution[1]. Applying a complete inference technique (i.e. K-consistency, for $K = n$, where n is the number of variables in the CSP) *guarantees* that a solution can be found using inference alone. That is, having removed all inconsistent values from the domains of all of the CSP's variables, what remains is a problem in which it is possible to assign all variables any value from their respective domains with the certainty that any such assignment is a

[1]It should be noted that, for some restricted (tractable) classes of CSP, AC and PC *can* find a solution (by inference alone) without the use of search. However, generally this is not the case. This is discussed further at the end of Section 2.3.2.2.

solution to the CSP. However, achieving K-consistency requires time and space that is exponential in K.

In practice therefore, the resource requirements of CSP consistency methods lead to a compromise in the amount of preprocessing (inference) that can be achieved prior to starting to search for a solution to a CSP. Hence, despite the limitations of AC and GAC, these, together with global constraints, are the consistency techniques most used in constraint solver systems. Further, such solver systems sometimes employ an additional, often cost-effective, but weaker notion of consistency known as *bounds consistency* [9]. In this approach, the domain of each variable is bounded by an interval, with only the end-points meeting the AC requirement. Unsupported upper and lower values can be removed until bounds consistency is achieved.

Having introduced inference as a preprocessing technique, the following subsection discusses CSP search. Included in this discussion is consideration of additional uses of inference, both as a consistency check following value assignments to variables, and as a forward looking approach, used prior to value assignments.

2.3.2.2 CSP Search

The previous section discussed inference as a means of reducing the number of inconsistencies in a CSP. Whether or not some form of consistency checking has been carried out, generally there will be a need to search[1] for a solution. This process of attempting to find a solution to a CSP by trial and error can be considered as search through a tree-like structure. A node in the search tree corresponds to the assignment of a value to a given CSP variable. Each branch represents a particular partial assignment of values to variables, whilst a path, from root to leaf, is a complete assignment of values to all variables (Figure 2.11).

Generate and test (GT) [108] assumes no preprocessing and is the most basic (and most computationally expensive) CSP search solution method. GT consists of systematically generating every possible complete variable and value assignment. Once generated, a complete assignment is tested to verify whether it satisfies all of the constraints. If it does, a solution is found and the algorithm terminates. Otherwise, the next complete assignment is checked. For example, Figure 2.11 shows part of the search tree for the map colouring problem of Figure 2.7. The sequence 1,2,3,4,5 shows the path from root to leaf representing the assignment of the value Red to each of the variables A, B, C, D and E (assuming natural variable ordering). GT would assign values

[1]This thesis considers only systematic and complete search. The techniques discussed do not contain any stochastic or incomplete methods.

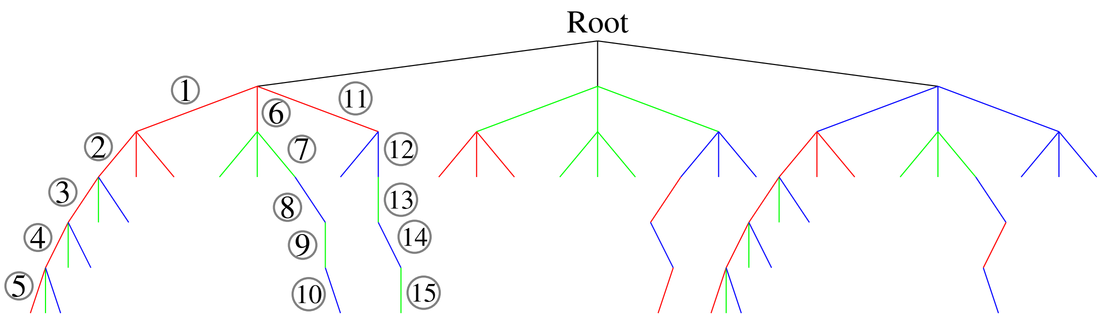


Figure 2.11: Map colouring problem (partial) search tree (branches shown coloured, and many branches left as stubs in order to improve visual clarity).

to all variables, as in this example, before discovering that such an assignment is clearly inconsistent given the constraints. The number of combinations checked under this approach is equal to the size of the Cartesian product of all of the variable domains. The inefficiency in GT, that due to assigning a value to **all** variables **before** checking the validity of the constraints, can be addressed by checking if the relevant constraints are satisfied immediately after each variable in a given constraint's scope is assigned a value. This improvement is incorporated into procedures that use *backtracking* (BT) [109].

The backtracking approach extends a partial solution by assigning values to the CSP variables one at a time. The order in which the variables are chosen for assignment of values may correspond to their numerical ordering in the original definition. Alternatively, a different, static or dynamic, ordering may be used. Likewise for value ordering. Both variable and value ordering are important decisions that have an impact on the amount of search required[1]. Assuming "standard" ordering, the algorithm progresses sequentially from the first variable, attempting to assign a value to each variable in turn. The choice of value for a given variable is then checked for consistency with previously assigned variables which share the scope of any relevant constraints. For example, in the map colouring problem (Figures 2.7 and 2.11), having assigned variable A the value Red and having checked that this did not violate any constraints, the value Red could be assigned to variable B. However, upon checking the constraint $A \neq B$, an inconsistency would immediately be discovered. At this point, the search algorithm would undo the assignment of Red to variable B and attempt to assign another value from its domain, again checking the appropriate constraints after the choice. In this way, backtracking prunes large parts of the search space (in this example, all branches below point 1 in Figure 2.11), whereas GT would have generated all such assignments

[1]Discussed further in sections 2.3.2.3 and 2.3.2.4.

only to find that each was inconsistent in the test phase. Should the backtracking algorithm exhaust all values in the current variable's domain, a *dead-end* is reached.

A dead-end is a branch in which it is impossible to find a value assignment for a given variable, having already assigned values to previous variables higher up the branch. When a dead-end occurs, the algorithm backtracks to the previous variable (i.e. the variable before the one that has no consistent value within its domain) and assigns a different value to it. The algorithm then continues forward to the previously inconsistent variable and attempts to find a consistent value assignment for it. Search continues in this manner until either all required solutions are found or no (more) solutions can be found. This type of backtracking is known as *chronological* backtracking since, when a dead-end is encountered, the algorithm returns to the previous variable in the search order.

An example of backtracking over a dead-end is shown in Figure 2.11 where a possible solution is given by the sequence 6,7,8,9,10. This corresponds to the variable assignments $A = Red, B = Green, C = Blue, D = Green$, and $E = Blue$, as also shown in Figure 2.7. If now the original CSP is modified by adding the constraint $E \neq Blue$ then this solution becomes invalid. However, attempting to solve this new problem, the algorithm will assign all other variables up to the point where it can only assign $Blue$ to variable E (since $A = Red$ and $D = Green$ - see Figure 2.7 (c)). Thus, variable E cannot take the value $Blue$, due to the constraint $E \neq Blue$, nor can it take either of the remaining values of Red or $Green$, since $A \neq E$ and $D \neq E$, and $A = Red$ and $D = Green$ are already assigned. Therefore, the algorithm will backtrack over this dead-end to the previous variable, D, and attempt to assign another value to it. When D is revisited, there is no alternative value remaining in its domain since it is constrained to be different from both A and C, which are already assigned Red and $Blue$ respectively. The backtracking continues in this way until variable B's value is changed to $Blue$ and an alternative solution ($A = Red, B = Blue, C = Green, D = Blue$, and $E = Green$) is found. This is illustrated as path 11,12,13,14,15 in Figure 2.11.

The previous example of backtracking over a dead-end also highlights the importance of the ordering used during CSP variable and value assignment. Had variable E (and its associated constraint, $E \neq Blue$) been processed earlier, a larger portion of the search space would have been pruned earlier. That is, all branches following an $E = Blue$ assignment could be disregarded without instantiating the variables below.

Even without an optimum variable ordering, the pruning benefits of the BT method make it more efficient than the GT approach. Another advantage of BT is that the algorithm requires only a linear amount of space. However, it can, in the worst case, require time that is exponential in the number of variables [110]. One of the reasons for this inefficiency is that the BT algorithm exhibits *thrashing* [111]. This occurs, for example, in the case where a variable at a higher level of the search tree is assigned a value which in turn, due to a connecting constraint, leaves a lower level variable with no consistent value in its domain. The result is that all of the possible value to variable assignments for all of the variables between these levels must be tried before the failure is discovered; the consistency tests fail over and over again for the same reason. That is, the value assigned to the higher level variable will always cause a failure to the lower level variable with which it shares a constraint. Hence, every path through the search tree that connects these two assignments will be visited (pointlessly) to search for a non-existent solution. This situation can be mitigated by checking all of the constraints in the CSP for consistency **before** search starts, and removing from the domains of relevant variables, those values which cause such behaviour. This preprocessing consistency checking approach is that discussed in the previous section. However, due to the necessarily limited amount of consistency checking (e.g. AC), some level of thrashing behaviour will remain.

With the addition of a preprocessing consistency checking step, the standard chronological BT approach can be seen to employ consistency checking in two ways. The first is this preprocessing step before search starts, as discussed in subsection 2.3.2.1. The second is **during** search, after a value has been assigned to a variable. That is, once a value assignment has been made, the algorithm checks whether the current assignment is consistent with previously assigned variables with which the current variable shares a constraint. Improvements can be made to the way BT uses inference during search. This can be achieved in two ways, by using *look-back* techniques [9] and by using *look-ahead* techniques [10].

Look-back methods are concerned with the backwards phase of the search algorithm. The standard BT algorithm backtracks only to the variable immediately prior to the current variable, and also, BT does not "remember" and make use of previous failures (i.e. it backtracks for the same reasons repeatedly, in different parts of the search space). Hence, look-back techniques improve on BT by allowing deeper, more informed backtracking (e.g. *backjumping, backchecking, backmarking*), and by recording (as new constraints) the cause of dead-ends (*constraint recording / no-good learning*).

Backjumping (BJ) [111] differs from BT at the point where backtracking occurs. Whereas BT steps back one level when a dead-end is encountered, BJ analyses the cause of the dead-end, identifying the (previously) assigned variable in the constraint that is causing a conflict with the most recently assigned value (to the current variable). The algorithm then "jumps" back to the conflicting variable and chooses a different value for it. If there are no alternative values for that variable, BJ defaults to standard backtracking. *Conflict directed backjumping* (CBJ) [112] improves on the standard BJ algorithm by recording a *conflict set* of previous variables that failed consistency checks with the current variable. This allows jumps back to a deeper level if necessary.

Backchecking (BC) [113] is used to avoid redundant checking of the current variable's assignment for compatibility with previous variables' value assignments. In this approach, if a combination of value assignments to their respective variables has previously proved incompatible, this is remembered, thus preventing a redundant recheck which would only ascertain the same information.

Backmarking (BM) [113] improves on BC by, in addition to recording the incompatible combinations, remembering those combinations that are successful. Although BM does not reduce the search space as such, it does potentially improve efficiency since the cost of updating a table for each new node visited is constant, and for each node generated, a single table lookup may replace as many as O(c) consistency tests, where c is the number of constraints (the extra space required is O(nk), where n is the number of variables, and k is the number of values).

The second group of look-back techniques are those that "remember" the reasons for a dead-end when one is found. Such constraint-recording or no-good learning methods add new constraints to record subsets of the variable-value combinations causing backtracks. By aiming to speed up the search process, the objective of learning procedures is to identify and record the conflict sets. The benefit gained (by not having to backtrack since a variable-value combination is already known to fail) must be measured against the additional cost incurred as a result of processing an increased number of constraints at each node. Learning methods can be differentiated by the manner in which they find smaller conflict sets.

Deep learning [114] requires that only *minimal* conflict sets[1] are recorded. The process of finding such sets is computationally expensive since it requires deeper analysis. In contrast, *shallow learning* permits the recording of *non-minimal* conflict sets, and takes less time and space. There exist a number of approaches to the implementation of learning algorithms, including *graph-based learning* and *jumpback learning* [9].

[1]Minimal conflict sets are subsets of the conflict set which are in conflict with the current variable's assignment.

Whilst the look-back methods discussed (BJ, BC, BM and learning) all provide improvements, they each operate after the current variable's value has been assigned. That is, after an inconsistent value has been selected, a backward consistency check is carried out. For this reason, look-ahead techniques were developed. These methods attempt to preempt future inconsistencies by carrying out *forward* checks at each node. By looking ahead to uninstantiated *future* variables, and by checking the current variable's tentative value assignment for consistency with the values in the domains of such variables, inconsistent values can be filtered. Where no consistent values in a future variable's domain can be found, the current variable's value is changed since it cannot then be part of the CSP solution. Thus, dead-ends are found sooner and consequently there is a reduction in the overall search effort. Also, because all inconsistent values in the domains of the future variables have been removed, there is no need to check the current variable's assigned value against those of previous variables (i.e. the current variable's domain members have already been checked when previous variables were processed).

There exist several variations of the look-ahead idea, including *real full look-ahead* (RFLA) [115], which repeatedly enforces AC on all unassigned variables after each tentative assignment of a value to the current variable[1], *full look-ahead* (FLA) [113], which enforces AC only once, *partial look-ahead* (PLA) [113], which enforces DAC on the future variables, and *forward-checking* (FC) [113], which applies AC only to pairs of variables containing the current variable and future variables (i.e. those future variables forming pairs with the current variable and connected by a constraint).

The above look-ahead techniques differ in the amount of (forward) consistency checking applied, and can be viewed in terms of degrees of AC that each has [117]. For example, FC is considered to have approximately one quarter the amount of consistency checking of full AC (AC$\frac{1}{4}$). As is the case with using inference as a preprocessing technique, look-ahead methods have associated computational overheads (increases with the amount of consistency checking) and such costs must be balanced against the benefits gained. For many years, based on empirical evaluation [117], FC dominated the other look-ahead techniques. However, further studies of more difficult and larger problems [116], [118], [119] showed MAC and RFLA to be superior. Consequently, despite their theoretical exponential complexity, techniques such as MAC and RFLA are most often used in practical constraint solvers.

Whilst most realistic constraint problems will require search, in addition to the

[1]A further variant of RFLA, known as *maintaining arc consistency* (MAC) [116], carries out AC **after** a domain value is filtered out.

use of inference (before and during search), there are, as mentioned in section 2.3.2.1, classes of tractable CSPs that permit solution by inference alone. Such groups of CSPs, once recognised, can be solved in polynomial time using one of the above incomplete consistency techniques (e.g. AC or PC).

A number of methods have been identified for recognising tractable classes of CSPs. One such makes use of constraint networks with *restricted structure* (e.g. tree networks [120], bounded-width networks [121]). Another method relies on the use of *restricted forms of constraints* (e.g. domain size restrictions [9], restricted relations [122]). Some of these methods are revisited in the following section where they are discussed in the context of structure-guided variable ordering heuristics. Yet another method of identifying tractable classes is a hybrid approach [123]. Recent work [124] attempts to bridge the gap between these tractable classes and the empirical success of theoretically exponential algorithms (e.g. FC, MAC, RFLA).

Unfortunately, complex real-world problems seldom contain such easily solvable tractable classes. Therefore the practical approach remains a combination of inference and search, using techniques such as MAC and RFLA.

This section has discussed backtrack-based search, together with a number of improvements on the basic BT approach. Throughout the discussion it has been assumed that the order in which the variables are chosen is standard. That is, $Variable1$ is assigned a value first, followed by $Variable2$, $Variable3$ and so on. This is the natural ordering of the variables in the original CSP definition. However, since the amount of the overall search space that is actually explored and the amount of backtracking that takes place depend not only on the level of consistency in the constraint network, but also on the variable and value ordering [125], [113], heuristic guides for selecting variables and values would be helpful. The following two sections discuss such heuristic techniques.

2.3.2.3 Variable Selection Heuristics

In CSPs, assuming the use of backtracking type search, the variables will be assigned values in some order. This order can be the natural order as defined in the problem or some other, perhaps more efficient, order. Also, variable ordering may be either *static* or *dynamic*, with dynamically calculated orderings being more informed. Static orderings are determined before search begins and are fixed, whilst a dynamic ordering unfolds during search. The optimal ordering results in the search process visiting the lowest number of nodes of all possible orderings whilst attempting to find a solution (or determining that one does not exist).

Again considering the CSP search space as a tree-like structure (e.g. Figure 2.11), each branching point represents a value to variable assignment. A variable selection heuristic aims to select the "best" choice of variable to be next instantiated at each branching point, with the general idea behind variable ordering being the removal or reduction of backtracking, with a consequent pruning of the search space.

Variable selection heuristics are generally one of two types [11], those that make use of the size of the CSP variables' domains, and those that rely on the structure of the CSP.

An example of the first type, one which employs the *fail-first* (F_aF_i) [1] principle [126], estimates the most constrained variable by selecting for assignment the variable with the smallest number of values in its domain. When implemented in this manner, F_aF_i is known as the *minimum remaining values* (MRV) heuristic, and is defined as follows.

Definition 25 *Let $rem(x|p)$ be the number of values that remain in the domain of variable, x, following propagation of the constraints, given a set, p, of branching constraints.*

Definition 26 *The MRV heuristic, h_{MRV}, chooses the variable, x, that minimises:*

- $rem(x|p)$

MRV can be used with a basic uninformed BT style search. However, combining the heuristic with a FC approach [113] proves more powerful since more information (due to propagation of constraints) is available at each node.

In the F_aF_i approach, the assumption is made that any one of a variable's set of potential values (domain) is equally likely to be part of the CSP solution. Therefore, more values in a variable's domain equates to a higher likelihood of one of them being a successful choice. Conversely, a variable with a small domain has a higher likelihood that one of them is a failure. Hence, F_aF_i attempts to find such a failure early in the search process, thereby avoiding larger amounts of backtracking over previously assigned variables should such a failure be discovered late in the search.

The F_aF_i principle, and the intuition behind it ("to succeed, try first where you are most likely to fail"), have been extensively studied (e.g. [113], [127], [128]), with the conclusions being that a *radical* F_aF_i approach, in which failures are aggressively sought, can actually decrease search efficiency by increasing the branching

[1]When used as a ***constraint*** ordering heuristic, the F_aF_i principle suggests checking first the *constraints* that are most likely to fail.

factor whilst attempting to reduce the branch depth. Empirical results [128] show, however, that finding early failures is important. An alternative failure strategy has been suggested as the basis for a heuristic [129], in which an attempt is made to minimise the number of nodes in each search subtree. In general, two factors have been found [130] to affect the variation in the search efficiency of variable ordering heuristics. These are *immediate failure* and *future failure*.

A potential problem not addressed by the original MRV heuristic is that of breaking ties. That is, when all variables have the same size of domain, for example in the first step of the solution process, the MRV heuristic fails. By adding a tie-breaking step, the *degree heuristic* [131] solves this problem. The degree heuristic first selects the variable with the least number of values in its domain (as MRV), and breaks ties by selecting the unassigned variable with the highest *degree*, where the degree of a variable is the number of constraints in which that variable is involved with at least one other unassigned variable. Both a static and dynamic version of the degree heuristic exist, with the former having each variable's degree calculated once at the start of search, and the latter calculating it dynamically as variables are assigned values. The degree heuristic is also known as the *most constrained variable* (MCV) heuristic.

Another variant of the MRV heuristic [132], which is shown to work well on random problems, divides the domain size of a variable by the degree of that variable, choosing the variable with the lowest resulting value. In a similar approach [133], a weighted degree value is used as divisor. Weights are associated with constraints and initially have the value one. This value is increased each time a constraint is found to be responsible for a dead-end. The sum of the weights of the constraints involving a given variable and at least one other variable then form the divisor (weighted degree value). This technique proves effective over a range of problems.

A number of other heuristics use various factors in combination with the F_aF_i principle. One of these [125] makes use of an element of *constrainedness* (Rho, the solution density (ρ)) [134] to estimate the most constrained variable. The intuition behind this approach is the desire to branch into the subproblem that *maximises* the solution density, ρ. To maximise ρ, the variable, x_i, from the set of uninstantiated variables, is chosen that maximises:

$$\prod_{c \in \mathcal{C} - \mathcal{C}_i} (1 - t_c) \tag{2.2}$$

where $\mathcal{C} - \mathcal{C}_i$ is the set of future constraints that does not contain variable, x_i, and $(1 - t_c)$ is the fraction of assignments that satisfy the constraint, c.

Equation 2.2 estimates the solution density of the subproblem after choosing variable, x_i. Expressed more concisely, and hence involving less computational overhead, the resulting heuristic, h_ρ, can be defined as follows.

Definition 27 *The ρ heuristic, h_ρ, chooses the variable, x, that minimises:*

- $\prod_c (1 - t_c)$*, where:*
  - *c ranges over all constraints that contain variable, x, and at least one other unassigned variable.*
  - *t_c is the fraction of assignments that do not satisfy the constraint, c.*

Hence, h_ρ selects the variable with the most constraints and / or the *tightest* constraints. Thus, the branching is on an estimate of the most constrained variable.

Another heuristic from the same family as h_ρ is the *expected number of solutions* (E(N)) heuristic, $h_{E(N)}$. $h_{E(N)}$ combines elements of h_{MRV} and h_ρ, and branches into the subproblem that maximises the expected number of solutions, E(N). $h_{E(N)}$, is defined as follows.

Definition 28 *The $E(N)$ heuristic, $h_{E(N)}$, chooses the variable, x, that minimises:*

- $rem(x|p) \prod_c (1 - t_c)$*, where:*
  - *c ranges over all constraints that contain variable, x, and at least one other unassigned variable.*
  - *t_c is the fraction of assignments that do not satisfy the constraint, c.*

Thus, definition 28 is a product of the size of the "space" (definition 25) and the probability that a given element in the space is a solution. It is possible to consider another, intuitive, description of $h_{E(N)}$. Assume N is the number of solutions to the current subproblem. If $N = 0$, there are no solutions to the current subproblem, and backtracking will be required. Where $N = 1$, there is one solution to the subproblem. This value will remain constant as long as the search remains on the path leading to this solution, otherwise it will reduce. A potential heuristic is to maximise N. However, N cannot increase in value when variables are assigned values as search descends the search tree. E(N), therefore, is used as an estimate for N, and branching is into the subproblem that maximises E(N). Hence, this is an estimate of the most constrained variable since a loosely constrained variable will reduce N by more $(N/rem(x|p))$ than a tightly constrained variable due to the former having a larger domain of values $(rem(x|p))$.

In addition to $h_{E(N)}$ and h_ρ, the authors propose the *Kappa* heuristic, h_κ [125]. Each of these heuristics makes use of the product term in Definition 28. This can be a limiting factor since there is a need to determine t_c for every constraint, c, during search. Additionally, the estimation of t_c may cause problems with constraints that are not extensionally defined.

By making use of more than one viewpoint, the *Promise* heuristic [135] considers variables and values simultaneously. In this approach, the variable selected is the one with the smallest summed promise values across its domain. Promise for a value is the product of the supporting values taken from the domains of neighbouring, as yet, unassigned variables. The promise heuristic, h_P, is defined as follows.

Definition 29 *The Promise heuristic, h_{Pr}, chooses the variable, x, that minimises:*

- $\sum\limits_{a \in dom(x)} \prod\limits_{y} rem(y|p \cup \{x = a\})$, *where:*
    - *y ranges over all unassigned variables.*

The product term in definition 29 can be considered an upper bound on the number of solutions, having assigned variable, x, a value, a, and thus, the principle of h_{Pr} is to select the most constrained variable. When used in combination with FC, this heuristic is shown [135] to give good results on the n-queens problem.

Further study of the use of multiple viewpoints for branching heuristics [136], tested on a wider range of (permutation) problems, confirms that such an approach can be effective, with this study finding that this technique often performed better than using a stronger propagator. The authors also show that such an approach is consistent with the F_aF_i principle.

A heuristic similar to that of definition 29 is shown [137] to have better results than h_{MRV} on a number of problems, and the authors show that the heuristic is especially useful for selection of the first variable(s) in the ordering. However, it has been shown [130] that this approach does not perform well on random problems.

The second type of CSP variable ordering heuristic, that type which makes use of the structure of the CSP, rely on the fact that a CSP can be represented as a graph. By exploiting the topology of the graph, the need for, and the amount of, backtracking may be reduced. That is, for suitable problems, those exhibiting certain structural features, the use of structure-guided heuristics can bound the worst-case performance of a given BT algorithm. Also, it is possible to record structural *goods* and *no-goods* in order to prune the search space [11].

The *minimal width ordering* (MWO) heuristic [120] makes use of the *width* property of a CSP primal graph. As introduced in Figure 2.8, a primal graph consists of nodes, representing the CSP variables, and connecting arcs, representing constraints between any two variables (nodes).

Definition 30 *With the nodes of a CSP primal graph ordered, the width of an ordering is the maximum number of arcs from any node, x, to nodes coming before x in the ordering. The width of a constraint graph is the minimum width over all orderings.*

The MWO heuristic is a static technique that first gives the variables a minimal width total ordering. The variables are then assigned values in this same order, with the general strategy being to assign values first to those variables that are connected to a higher number of other variables. That is, those variables included in more constraints. It has been shown [120] that, if the level of strong k-consistency is greater than the width of the ordering, then search will be backtrack-free. Also, where the level of strong consistency is greater than the width of the constraint graph, there exists a backtrack-free assignment of values to variables. A less computationally expensive version of MWO, the *max-degree ordering*, approximates MWO by simply ordering the nodes' degrees.

Another static technique, the *minimal bandwidth ordering* heuristic (MBO) [138], preprocesses the CSP variables, giving a total ordering with minimal bandwidth.

Definition 31 *With n nodes in a constraint graph ordered $1, \ldots, n$, the bandwidth of an ordering is the maximum distance between any two nodes in the ordering that are connected by an arc. The bandwidth of a constraint graph is the minimum bandwidth over all orderings.*

The idea behind a minimal bandwidth ordering is to place variables sharing constraints close to each other in the ordering, thus ensuring that, should backtracking occur, the distance backtracked over is small. Tested on graph-colouring problems, MBO gives encouraging results, with the author showing that the search tree size decreases with decreasing size of bandwidth.

An alternative heuristic approach [139] makes use of *stable sets* of variables within the problem in order to recursively divide the CSP primal graph, before solving the resulting subparts.

Definition 32 *A constraint graph can be partitioned into sets of mutually independent variables, in which there is no direct constraint between any pair of variables in the set. Such sets are called stable sets.*

By taking into account the relative independence of certain variables, it is possible to first assign values to the variables that are directly connected by constraints thereby decomposing the problem and separating the remaining variables into disjoint sets. It is then possible to independently assign values to these variables, which are not joined by a constraint relation. By adopting this approach, the authors achieve additive instead of multiplicative behaviour when solving the independent subparts of the problem. Although this technique is a static ordering approach, an improved version [140] employs dynamic variable ordering within the subproblems. Further improvements have been made, with the results applied to certain classes of CSPs [141].

Another heuristic approach that makes use of decomposition of the constraint graph is the *cycle-cutset* technique [142]. By recognising that tree-structured CSPs can be solved quickly and efficiently, the cycle-cutset heuristic aims to identify and instantiate those variables that break cycles in the constraint graph. This set of variables is known as the cycle-cutset. Since finding the smallest cycle-cutset is NP-hard, a modified technique is generally used [143]. In this method, BT search is used with an integrated cycle-cutset algorithm. The variable assignments made by BT are monitored, and when the current set of instantiated variables constitutes a cycle-cutset, the BT search algorithm is replaced by a suitable consistency (tree) algorithm. Hence, either a consistent assignment for all variables is found or it is possible to determine that no such solution exists, at which point backtracking takes place.

Potential limitations of these structure-driven heuristics include the adverse affect of global constraints [11] , and their reliance on a static (or partly static) technique.

This section has discussed various heuristic approaches to variable ordering. The following section continues the discussion of heuristics, but focuses on the selection of values for the CSP variables.

2.3.2.4 Value Selection Heuristics

The previous section discussed variable ordering heuristics. For these, a guiding principle is to fail early in order to limit the impact of backtracking over an unsuccessful choice. In contrast, the strategy for value selection is to choose the value that is most likely to succeed, since the failure of a value will not reduce the size of the search subtree. That is, if a chosen value proves to be a failure when checked, the other values in the variable's domain will still need to be tried. Also, if all possible solutions are required, value ordering becomes less important since all values will be selected and checked anyway. Likewise, if there are no solutions to the CSP. In this case also, all values will be checked in order to determine that no solution exists. However, where a single solution is required, and this solution is required to be found as efficiently as possible, a value ordering heuristic will be useful.

With an ideal heuristic, one that consistently chooses the "correct" value for each variable, a solution may be arrived at with no backtracking. Since this ideal does not exist, a good heuristic is one that selects a value which in turn maximises the number of options for future variables' value assignments. An example of a heuristic that employs this *least constraining value* principle is *min-conflicts* (MC) [118]. MC chooses the value for the current variable that maximises the sum of the remaining variables' domain sizes. It is defined as follows.

Definition 33 *The min-conflicts heuristic, h_{MC}, chooses the value, $a \in dom(x)$, for variable, x, that maximises:*

- $\sum\limits_{y} rem(y|p \cup \{x = a\})$, *where:*
    - *y ranges over all unassigned variables.*

A modified form of h_{MC}, known as the (value) promise heuristic, h_{vPr} [144], instead maximises the product of the remaining variables' domain sizes. This dynamic value ordering heuristic is defined as follows.

Definition 34 *The value-promise heuristic, h_{vPr}, chooses the value, $a \in dom(x)$, for variable, x, that maximises:*

- $\prod\limits_{y} rem(y|p \cup \{x = a\})$, *where:*
    - *y ranges over all unassigned variables.*

It is shown [135] that h_{vPr} provides a more precise measure than h_{MC}, and that h_{vPr} provides the number of *completions* of node, p. The author notes that such completions may be interpreted in two ways. The first is that by choosing the value that maximises the product, the probability of branching into a subproblem that contains a solution is also maximised (assuming that all completions are equally likely to be solutions). The second interpretation is that the completions are an upper bound on the number of subproblem solutions. Empirical results show [135] [144] that h_{vPr} performs well on a range of problems.

An alternative, static, value ordering heuristic [142] makes use of an approximate measure of the number of solutions to each subproblem. This approximation is gained from a relaxed form of the CSP graph. The procedure first removes constraints until the remaining graph is a tree. Then, by counting the number of solutions to that tree, which can be performed in polynomial time, an ordering of the values is constructed, with the highest valued (most solutions) subtree chosen first.

An improvement [145] on the approximation method above makes use of Bayesian networks. This approach provides a less optimistic measure, and empirical results show its usefulness as a search heuristic when attempting to find a single solution.

A further improvement is a dynamic value ordering heuristic [146] that decomposes the CSP into spanning trees and again uses Bayesian networks to calculate probabilistic approximations of the number of solutions based on the current search state. The authors' results show the heuristic to be an improvement over previous techniques when tested on sparsely constrained CSPs. Also, it is shown that this approach can determine, with fewer backtracks, problem instances that are insoluble. However, the authors also show that, as the problem density increases, neither their dynamic heuristic nor previous static methods significantly improve the performance of search.

By adapting ideas from *iterative join-graph propagation* (IJGP), which is another belief propagation algorithm, the IJGP-SC solution counting based heuristic [147] shows good results. The authors test the heuristic on random CSPs, graph colouring problems and quasi-group completion problems, finding that IJGP-SC gives a very focussed search with fewer dead-ends.

All of the above value ordering heuristic methods based on approximating the subtree solution count, whilst providing valuable search guidance, do not provide a relative measure of the complexity or size of the subtrees being considered. Nor do they give an estimate of the associated "cost".

This section and the previous one have discussed CSP value and variable ordering heuristics respectively. The goals of such heuristics are to avoid constraint violation and, if violation occurs, discover it quickly. Variable ordering heuristics generally aim to select the variables that are most constrained, thereby pushing potential failed assignments towards the top of the search tree. This can be visualised as reducing the *bushiness* of the search tree. Value ordering heuristics attempt to choose those values that are more likely to succeed, which has the effect of pushing a solution towards the left of the search tree. Overall, the aim is to find a solution efficiently, with the least possible amount of backtracking search.

In concluding this section, it is worth noting that many of the CSP heuristic experiments in the literature have been conducted on problems that are randomly generated [148]. Such problems generally contain variables with (initially) same sized domains and constraints of equal "difficulty". Additionally, certain heuristics have been deliberately tested only on specific types of problem (e.g. [135],[137],[141]). Real world problems, and AI planning problems, formulated as CSPs often do not have these restricted qualities. Consequently, generic CSP heuristic methods, whilst useful in general, are not tailored to, nor guided by, specific features of the individual given problems.

2.4 Reformulating a Planning Problem as a Constraint Satisfaction Problem

The work described in the later chapters of this thesis involves applying a variable and value ordering heuristic to planning problems reformulated as CSPs. In order to achieve this, the planning problems must first be represented as CSPs[1]. This section describes, as an example of the basic encoding procedure, a *direct encoding* reformulation process. This is the initial encoding used in the work that follows[2].

As previously mentioned, classical planning is PSPACE-hard, with the difficulty coming from the fact that the plan length is not known in advance. However, if a bound is imposed on the plan length, the problem becomes NP [149]. With this bound in place, an NP-complete formalism, such as CSP, can then be used to represent the planning problem. Such a plan length bound, incrementally extended, is the means by which planning as SAT / CSP is typically solved.

It has been noted [11] that it can be difficult to define precisely what is meant when it is stated that a CSP *represents* a particular problem. The definition used here is as follows.

Definition 35 *A CSP,* $\mathcal{M} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$*, represents a problem,* $\mathcal{P}$*, if every solution of* $\mathcal{C}$ *corresponds to a solution of* $\mathcal{P}$*, and every solution of* $\mathcal{P}$ *can be derived from at least one solution to* $\mathcal{C}$*.*

Thus, the set of solutions of the CSP corresponds to the set of plans that achieve the planning problem's goals.

[1]Alternative types of constraint encoding, and modelling choices, are discussed in Section 2.3.1.

[2]Other planners' constraints and an alternative encoding are discussed in Section 2.5 and Chapter 3, respectively.

2.4.1 Direct encoding

In order to encode the bounded planning problem as a CSP, the state variable representation (SAS$^+$) is used. Referring to definition 6 (page 15), the SAS$^+$ bounded planning task, $\mathcal{P} = (\mathcal{V}, \mathcal{O}, s_0, s_\star, k)$ (where k is the plan length bound), can be converted to a CSP encoding, $P^{'}$, in four steps ([150],[151],[3]). These are, defining the CSP variables, encoding the initial state and the goal state, defining the constraints to encode the actions, and encoding the frame axioms. The first step, to encode the CSP variables, is defined as follows.

Definition 36 *For each ground state variable, x_i, of $\mathcal{P}$ ranging over D_i and for $0 \leq j \leq k$, there is a CSP variable of $P^{'}$, x_i^j, whose domain is D_i. Also, for $0 \leq j \leq k-1$, there is a CSP variable, act^j, the domain of which is the set of all possible actions, including a no-operation (noop) action*[1] *that has no preconditions and no effects (i.e. $\forall s$, $\gamma(s$,noop$)$ = s).*

Hence, the variables in the CSP equate to the ground state variables in the planning problem, $\mathcal{P}$, with the addition of a variable to represent the action taken at each step (or level) of the plan $(0, \ldots, k-1)$.

The second step is to define the constraints that encode the initial state, s_0, and the goal state, $s_\star$.

Definition 37 *For each ground state variable, x_i, with value, v_i, in s_0, there is a unary constraint of the corresponding CSP variable for $j = 0$:*

$\{(x_i^0 = v_i)\}$, *and*

for each ground state variable, x_i, with value, v_i, in $s_\star$, there is a unary constraint of the corresponding CSP variable for $j = k$:

$\{(x_i^k = v_i)\}$

The third step converts the planning problem's actions into binary constraints in the CSP, $P^{'}$.

Definition 38 *For each action, a (an instance of an operator, $o \in \mathcal{O}$):*

For each condition of the form $(x_i = v_i)$ in precond(a), there is added to the set, E, the pair:

[1]Readers unfamiliar with "noops" may find it useful to review Section 2.2.6.1, page 26. Also, further useful information can be found in [40] and [48].

(act^j = a, $x_i^j = v_i$)

For each condition of the form ($x_i = v_i$) in precond(a) such that there is no assignment of x_i in effects(a), there is added to the set, E, the pair:

(act^j = a, $x_i^{j+1} = v_i$)

For each assignment, $x_i \leftarrow v_i$ in effects(a), there is added to the set, E, the pair:
(act^j = a, $x_i^{j+1} = v_i$)

Following the above procedure, the set, E, contains all of the pairs of permitted values for the variable, act^j, and for the variables within the actions. The binary constraint on act^j and a variable, x, is then the union of all pairs in E related to act^j and x.

The final step in the reformulation procedure is encoding the frame axioms.

Definition 39 *Frame axioms may be encoded directly into ternary constraints, with tuples containing act^j, an invariant variable, x_i^j, and x_i^{j+1}. Following the procedure in definition 38, the set of all tuples of possible values is first defined and recorded. That is:*

(act^j = a, $x_i^j = v_i$, $x_i^{j+1} = v_i$), for $v_i \in D_i$

A frame axiom constraint is the union of all such tuples relating to the same three variables, act^j, x_i^j, x_i^{j+1}.

The above procedure (definitions 36 to 39) constructs an implicit (intensional) encoding of the constraints. That is, the constraint representation of the planning problem is described in terms of relational symbols. When appropriately coded in software, the resulting procedure automatically generates a (solver specific) constraint encoding from the given definition of a planning problem.

As an illustrative example, consider Figure 2.12 which shows an example SAS$^+$ variable definition from the Driverlog instance 2 planning problem[1]. The PDDL problem definition is reproduced in Figure 2.13 for easy reference.

The variable shown in Figure 2.12, Variable 1 which represents one of the drivers in the problem, can hold one of nine values. When reformulated as a CSP variable, this range (domain) of values takes the form shown in Figure 2.14 (i.e. T[1..24,1]::[0..8]), where T is the CSP solution matrix, which in this case is of size 24 (rows) (see also Figure 2.15).

[1]This domain and full problem set are available at http://ipc02.icaps-conference.org/

```
        var1:
              0: Atom driving(driver2, truck1)
              1: Atom at(driver2, s2)
              2: Atom at(driver2, p0-1)
              3: Atom at(driver2, s0)
              4: Atom at(driver2, s1)
              5: Atom driving(driver2, truck2)
              6: Atom at(driver2, p2-1)
              7: Atom at(driver2, p0-2)
              8: <none of those>
```

Figure 2.12: An example SAS$^+$ variable definition from the Driverlog 2 planning problem.

```
(define (problem DLOG-2-2-3)
(:domain driverlog)
(:objects driver1-driver driver2-driver truck1-truck
truck2-truck package1 - obj package2 - obj package3 - obj
s0 - location s1 - location s2 - location
p0-1 - location p0-2 - location p1-0 - location
p2-1 - location)

 (:init (at driver1 s0) (at driver2 s0) (at truck1 s0)
(empty truck1)   (at truck2 s1) (empty truck2)
(at package1 s2) (at package2 s1)(at package3 s1)
(path s0 p0-1)   (path p0-1 s0) (path s1 p0-1)
(path p0-1 s1)   (path s0 p0-2) (path p0-2 s0)
(path s2 p0-2)   (path p0-2 s2) (path s2 p2-1)
(path p2-1 s2)   (path s1 p2-1) (path p2-1 s1)
(link s0 s2)     (link s2 s0) (link s1 s0)
(link s0 s1)     (link s1 s2) (link s2 s1))

 (:goal (and (at driver1 s1)(at driver2 s1)(at truck1 s2)
(at truck2 s0) (at package1 s0) (at package2 s2)
(at package3 s0)))
)
```

Figure 2.13: PDDL problem definition for Driverlog 2.

Figure 2.15 shows the resulting CSP matrix definition for the nine SAS$^+$ state variables in the example Driverlog 2 problem instance.

Each column (1 to 9) of Figure 2.15 represents a SAS$^+$ variable, with the number of rows (24) being the bounded plan length. Hence, row 1 shows the fully defined

```
T[1..24,1]::[0..8], T[1..24,2]::[0..8],
T[1..24,3]::[0..5], T[1..24,4]::[0..5],
T[1..24,5]::[0..5], T[1..24,6]::[0..3],
T[1..24,7]::[0..3], T[1..24,8]::[0..1],
T[1..24,9]::[0..1]
```

Figure 2.14: CSP (state) variables, with associated domains, for the reformulated Driverlog 2 problem.

```
[3,5,4,0,0,1,2,0,0,
 _,_,_,_,_,_,_,_,_,
 _,_,_,_,_,_,_,_,_,
 _,_,_,_,_,_,_,_,_,
          .
          .
 _,_,_,_,_,_,_,_,_,
 _,_,_,_,_,_,_,_,_,
 4,3,2,3,4,2,0,_,_]
```

Figure 2.15: CSP (state variables) matrix representation of a directly (ECL^{i}PS^e) encoded planning problem.

initial condition, with the final row representing the CSP reformulation of the partially defined goal condition. Each element of the T matrix is therefore constrained by the range of values associated with it, as shown in Figure 2.14. Note that any particular column represents a single CSP variable's values through time. That is, an action acting on a particular variable's value in a given row will change that variable's value in the following time slot (row), assuming the original planning problem action has that variable in its set of effects.

```
Actions[1,1..23]::[1..109]
```

Figure 2.16: CSP (action variables) matrix representation a directly (ECL^{i}PS^e) encoded planning problem.

Figure 2.16 shows the corresponding definition for the CSP variables representing (potential) actions in the resulting solution plan. These are constrained to take only the range of values representing the number of available actions, 109 in the case of this example.

```
 Y is N+1,

(T[N,6]=:=1) and (T[N,2]=:=5) and
(T[Y,2]#=4) and (T[N,8]=:=0) and
(T[Y,8]#=1) and (T[Y,1]#=T[N,1]) and
(T[Y,3]#=T[N,3]) and (T[Y,4]#=T[N,4]) and
(T[Y,5]#=T[N,5]) and (T[Y,6]#=T[N,6]) and
(T[Y,7]#=T[N,7]) and (T[Y,9]#=T[N,9]) and
(Actions[1,N]#=1)
```

Figure 2.17: CSP representation of Preconditions, Effects, and Frame Axioms, of a directly (ECL^{i}PS^e) encoded planning problem.

Figure 2.17 shows a small example of one set of constraints in the final form of the directly encoded (ECL^{i}PS^e) example planning problem (Driverlog 2). This shows, for Action 1, the values of the preconditions (elements of T denoted by N (e.g. T[N,6]=:=1 indicates that one of this action's preconditions is that variable 6 must hold the value 1)), the values of the effects (elements of T denoted by Y), and the frame axioms (e.g. T[Y,3]=T[N,3]).

This (see the example given in Figures 2.14 to 2.17) was the initial method used to construct the base model used for empirical testing in this thesis. This is described further in the next chapter[1].

Having introduced classical planning and CSPs, and having described a procedure for converting planning problems to CSPs, this chapter concludes with a review of related work.

2.5 CSP Planning Research

This section discusses the use of CSP techniques for AI planning. It highlights important developments and reviews work that is relevant and related to this thesis, thereby providing context for the work described in the following chapters.

Using CSPs to solve AI planning problems is not one single technique or procedure. Whilst the planning problem, or part thereof, must be reformulated as a CSP if constraint-based methods are to be used, the type of constraints and the level of integration of these two technologies varies. There are three main ways of using constraint techniques in planning [152] [153]. These are *planning with constraint posting*,

[1]Further details, and a further specific example of this, are given in Section 3.2 on page 80.

planning reformulated as a CSP - with a fixed or incremental horizon, and *planning completely captured within constraint programming - without restricting the horizon*.

The first category only uses constraints to solve parts of the planning problem. That is, a part of the problem is posted to a sub-solver, with the result being returned to the conventional planning machinery for further processing. In this approach, there is limited interaction between the CSP solving and the original planning problem. Hence, although constraint techniques are being used, they are not being used to solve the entire planning problem. Planning systems that employ this model to varying degrees include Molgen [154], Descartes [155] and MACBeth [156].

The second way of using constraints in planning recasts the bounded planning problem as a CSP (e.g. using the procedure described in section 2.4.1) and applies constraint-based solution techniques. The work in this thesis falls into this category. A number of planning systems employing this type of approach are now considered in more detail.

Following the success of systems that automatically reformulated the bounded planning problem as instances of the *satisfiability problem*[1] (SAT) [157], the first attempt at compiling planning problems into CSP encodings was that used in the CPlan planner [158]. The constraint model used in CPlan was hand-coded, with the authors justifying this by emphasising the importance of correct modelling under the CSP paradigm. Each state is modelled by a collection of CSP variables, with the constraints enforcing valid transitions between states. The authors note that for a minimal correct model of a given planning domain they needed only *action constraints* and *state constraints*. Also, that the additional constraints (3-7, below) were necessary to improve the efficiency of the search. The constraints used in CPlan are as follows:

1. Constraints that represent the effects of the actions (action constraints).

2. A set of constraints that enforce how variables within a state must be consistent (state constraints).

3. Constraints (distance constraints) which limit the distance (no. of steps) that a variable travels in order to get from one value to another (The authors give the example of the logistics domain with one package variable. The lower bound on the number of steps required to transfer it from one non-airport location in one city to a non-airport location in another is nine). It is noted that these constraints were found to be very important in terms of reducing the search space in the test domains.

4. Encoding of constraints that break symmetries on the values that variables can take (symmetric values constraints). The example given is again in the logistics do-

[1]SAT is a particular form of CSP, having binary domains and non-binary constraints.

main; with two package variables, the planes in their domains are often symmetric. If there is (or isn't) a solution with a particular assignment of planes to packages then there is (or isn't) an alternative with the planes swapped around. Like distance constraints, the symmetric value constraints were found to be important for reducing the search space.

5. Action choice constraints restrict which actions can be chosen in each state. This type of constraint seeks to remove choices between similar sequences of actions that will result in equivalent outcomes (e.g. Two packages at an airport that are bound for the same destination may either be picked up together and transported or could be transported separately). The constraint restricts the choice to just one suitable action.

6. In order to stipulate limits on resources, capacity constraints may be used. Obviously this type of constraint is only relevant in domains where there are restrictions on the available resources.

7. Encoding possible restrictions on the original domains of the CSP variables, domain constraints. (An example being where, in the logistics domain, a package is to be picked up and delivered within a single city. In this case, no airports or other cities are required and can be removed from that variable's domain).

To find a solution plan in CPlan, a lower bound on the plan length is used to construct a CSP of that length. With the variables representing the initial state and goal state assigned values, backtracking search with GAC and CBJ is applied. Using a dynamic variable ordering (MCV heuristic), the CSP variables are instantiated. If no solution with the bounded plan length is found, the size of the CSP is incrementally increased and the process repeats until a solution is found or it can be shown that no solution exists (or some time limit is reached). CPlan is sound and complete, with the planner's performance shown to be better than the state of the art (at the time) on a number of IPC planning domains, with its only major drawback being that it required manual encoding.

CSP techniques have also been used in conjunction with planning graphs (see section 2.2.6.1), recognising that the backward search phase of the Graphplan planner can be viewed as a form of CSP and improved with CSP-derived techniques [159] [160]. The CSP-based planner, GP-CSP [161], extends this idea by converting Graphplan's planning graph structure into a standard CSP and solving it using regular CSP methods. In this approach, the planning graph search is first interpreted as a *dynamic CSP* [1]

[1]A DCSP differs from a standard CSP only in that it includes a set of *activity constraints* for the variables. With an initial subset of the DCSP variables active, the objective, as the solution process progresses, is to find a consistent assignment for the variables that satisfies both the constraints of the problem and the *activity constraints*.

(DCSP) [162]. The DCSP variables represent the facts at each level of the planning graph, with the domains of those variables representing the actions (at previous levels) that support the facts. The constraints used in GP-CSP are as follows:

1. Action mutex constraints prevent mutually exclusive actions being allowed. However, such constraints must be modelled indirectly due to the actions being modelled as values rather than as variables. That is, if two actions, a_1 and a_2, are mutex, then for every pair of facts, f_{11} and f_{12} where a_1 is one of the possible supporting actions for f_{11} and a_2 is one of the possible supporting actions for f_{12}, there is a constraint of the form $\neg(f_{11} = a_1 \wedge f_{12} = a_2)$.

2. Fact mutex constraints in the planning graph are modelled as constraints that prevent the simultaneous activation of two fact propositions. For example, if two facts, f_{11} and f_{12}, are mutex, the following constraint is required: $\neg(Active(f_{11}) \wedge Active(f_{12}))$.

3. Supporting a fact, f, which is active at a given level, with an action, a, at the previous level sets as active all of the facts in the level prior to the action's level. In this way, subgoal activation constraints are implicitly specified by the action preconditions.

The DCSP is then converted to a standard CSP, with the DCSP's activity constraints being compiled into standard constraints by introducing a *null* ($\perp$) value which is added to the domain of each of the standard variables. An inactive DCSP variable is then modelled as a CSP variable with the value $\perp$. Hence, the constraint that a variable, x_1, be active is modelled as $x_1 \neq \perp$. Once constructed, the CSP is solved using backtracking search with GAC and CBJ, the same solving approach as CPlan. The authors show that GP-CSP was competitive with state of the art SAT solvers and with Graphplan. Improving the way the action mutex constraints were encoded (by reducing the actual number of fact constraints used to record mutex actions) led to a significant reduction in memory use and solution time. Further improvements to the CSP solver, including the addition of *explanation based learning* (EBL), *level-based variable ordering*, *random restart search*, *distance-based variable ordering*, *min-conflict value ordering*, and *backwards mutex constraints* proved partly successful, with only the first three techniques providing significant performance improvements.

In contrast to GP-CSP, the Csp-Plan planner [163] does not rely on the planning graph in order to automatically formulate the planning problem's constraints. Instead, the system makes use of CSP-based techniques that similarly exploit the structure of the planning problem. By first generating a base CSP and then applying a number of transformations, which are similar to CSP consistency techniques, the constructed model contains all of the mutex information found in the planning graph (and more).

Encoding a Graphplan style k-step plan as a CSP requires $k + 1$ sets of boolean propositional variables, P_i^s, and k sets of boolean action variables, Act_j^s, where s ranges from 0 to k for propositions and 0 to $k-1$ for actions, and i and j range over the number of distinct propositions and action instances in the original planning problem respectively (P_i^s indicates that proposition P_i is true at Graphplan step s. Act_j^s indicates that action Act_j was carried out at step s). The base constraints used in Csp-Plan are as follows:

1. Unary initial state and goal state constraints.

2. Precondition constraints ensuring that an action, Act_j^s (equivalent to being at planning graph level s), cannot be true unless its preconditions are true. That is, $Act_j^s \rightarrow Pre(Act_j)^s$, where $Pre(Act_j)^s$ is action Act_j^s's precondition.

3. Successor state constraints ensure that a state following the current one can only contain a proposition that is true if that proposition was made true by an action or if it was already true and no action changed its value. That is, the successor state axiom constraints are of the form:

$$P_i^s \quad \leftrightarrow \bigvee_{Act_j \in Create(P_i)} Act_j^{s-1} \quad \vee \quad (P_i^{s-1} \wedge \bigwedge_{Act_j \in Delete(P_i)} \neg Act_j^{s-1}),$$

where $Create(P_i)$ and $Delete(P_i)$ are the set of actions that create and delete P_i respectively.

4. Concurrent action constraints prevent two actions occurring at the same time. Csp-Plan uses Graphplan style concurrency, which asserts that two actions cannot be true simultaneously if they interfere with each other.

5. Non-null step constraints ensure that, at every plan step, at least one action variable must be true.

Once the Csp-Plan base CSP has been constructed, CSP techniques are used to generate the binary mutex constraints achieved in the planning graph approach. This generalises the mutex computation process, and can lead to a situation where the CSP contains more mutex information than that achieved by the planning graph. Further reductions and transformations (e.g. symbolic reduction of single-valued variables and the addition of sequence constraints) make the CSP easier to solve. Csp-Plan employs a solver that allows for the use of different dynamic variable ordering heuristics and methods for recording no-goods. Empirical testing shows that the best performance was gained using GAC and CBJ with a specially tailored heuristic. Over a range of IPC domains, Csp-Plan outperformed both the original Graphplan planner and GP-CSP.

Recent work [164] has seen the reformulation of three of the above approaches to CSP-based encoding of planning problems. The direct encoding, the GP-CSP encoding and the CSP-Plan encoding have each been recast using an extensional representation (table constraints). Testing, on a series of IPC planning problem domains, has shown that the table constraint encoding of the CSP-Plan approach (using successor state axioms) dominates the others. It was for this reason that the SeP CSP planner [150] made use of this encoding.

SeP, in common with the other CSP planners above, is an optimal planner. That is, it finds the shortest length of plan that satisfies the goal (lowest cost plans, assuming uniform action cost). Using a multi-valued, state variable (SAS^{+}) description of the planning problem, and starting with a CSP of size one (i.e. with just one action slot), the size of the CSP is increased incrementally if no solution is found with the current plan length. In this way, an optimal solution is found since the planner guarantees that there is no plan satisfying all of the goals at previous levels. The constraints used in SeP are as follows:

1. Precondition constraints ensuring that an action, Act_i^s (i.e. Act_i at level s of the CSP), cannot be true unless its preconditions are true. That is, $Act_i^s \rightarrow Pre(Act_i)^s$, where $Pre(Act_i)^s$ are action Act_i^s's preconditions. These precondition constraints can be combined into one table constraint, with each row of the table representing an action from the original planning problem. The constraint's scope includes action variables, Act_i^s, and state variables, V_i^s. Hence, each row explicitly states, for a given action, the required value for each variable present in the preconditions of that action.

2. Successor state constraints combine effect constraints and frame axioms, and ensure that a state following the current one can contain a variable assignment if and only if that variable was assigned its value by an action at the current level or if the variable previously held that value and no action has changed it. That is, the successor state constraints are of the form:

$$V_i^s = val \quad \leftrightarrow \bigvee_{Act_j \in SetVal(V_i)} Act_j^{s-1} \quad \vee \quad (V_i^{s-1} = val \wedge \bigwedge_{Act_j \in NotInc(V_i)} Act_j^{s-1}),$$

where $SetVal(V_i)$ is the set of actions that assign the variable V_i the value, val, and $NotInc(V_i)$ is the set of actions not affecting variable, V_i. These successor state constraints are then represented as ternary table constraints (Act_j^{s-1},V_i^{s-1},V_i^s), with one table per state variable.

3. Dominance constraints are used to break plan-permutation symmetry [1] [166]. Such constraints depend on the concept of non-interfering actions and are not required to represent the planning problem, but instead are added to reduce the search space.

[1]SeP's designers have also proposed a CSP-encoded *parallel* planner, PaP, which achieves a similar result [165].

Two non-interfering actions can be ordered arbitrarily with no adverse effect on the plan. Thus, if one ordering results in a failed plan, the other ordering will do too. Likewise for a successful plan. Hence, only one ordering need be investigated. For two non-interfering actions, Act_1 and Act_2, a dominance constraint is added to the CSP such that for all pairs of successive levels (states) of the CSP, s and $s+1$:

$$Act_1^s \;\rightarrow\; \neg Act_2^{s+1}$$

The dominance constraints are then represented as table constraints, with the prohibited orderings being extensionally defined.

In addition to the above constraints, SeP enhances the constraint model with the application of a number of techniques. These include *lifting*, *singleton consistency*, and *no-good learning*. The system instantiates the action variables (fixed variable ordering) using a regression search approach. Testing on a range of IPC domains showed that the best results were gained using all of the enhancements in combination with the SeP constraint model. On smaller problems, however, the overhead associated with the enhanced consistency technique outweighed the potential benefits.

Constance [167], described by the authors in terms of a subset of *timelines* [168], takes a similar basic approach to that of SeP, augmenting the constraint model with further dominance constraints derived from macro-actions. These macros are determined by identifying sequences of actions that are composable under common prevail conditions. The authors make use of *composable substate graphs*, and by finding all of the strongly connected components of these structures (substate groups), are able to identify the set of composable action macros. Recognising that each action in an optimal plan must contribute to a goal (or precondition of a future action), the authors make use of the idea of *dependent actions* (such an action, Act_2, is dependent on an action, Act_1, if at least one of the effects of Act_1 is a precondition of Act_2). Concluding that, given a cost-optimal plan of minimum length, $p = a_1, \ldots, a_n$, containing two actions, a_i and a_j $(i \leq j)$, from the same substate group, c, there must exist an action, a_d, in the sequence, $a_{i+1}, \ldots, a_{j-1}$, such that a_d is dependent on a_i, it is possible to impose a rule stating that, after a transition has been made, then no further transition in the same substate group is possible until at least one of the effects of the transition has been used. Following some experimentation, this was translated into constraints that prevent actions from the same substate group being allowed in consecutive action slots.

Constance provides two search strategies, forwards and backwards, with no real differences found between them when empirically tested on instances of IPC problems. The results show that Constance performs well on problems with large substate groups, such as transportation type domains with large composable substate graphs generated from the underlying maps. The authors note that, despite Constance's good performance relative to other CSP planners (e.g. SeP) on many problems, the performance of constraint planners, in general, remains below that of the state of the art (non-CSP). However, on a small set of problem domains, the performance of Constance comes close to that of one of the best (at IPC) heuristic based planners.

Drawing on ideas from the SAT planner, SASE [169], the Transition Constraints for Parallel Planning (TCPP) system [170] makes use of the DTG information in the SAS$^+$ representation of planning problems. This parallel planner employs four different types of constraint: initial state, goal state, negative, and transition constraints. The first two are self explanatory, with there being two forms of negative constraint: mutex constraints and parallelism constraints. The former encodes mutually exclusive variable / value pairs, the latter those actions that are inconsistent due to shared variables in preconditions and effects with conflicting values. The transition constraints are generated from the set of DTGs for a given planning problem instance. For every DTG (i.e. SAS$^+$ variable), a transition constraint is generated, with this capturing all possible ways of changing the SAS$^+$ variable's value. The DTG edges contain the conditions (of other SAS$^+$ variables' values) that need to be satisfied in order for the transition between values of the current variable's values to occur. For variables not appearing on the current edge, the associated values can be considered *don't cares* for the given transition.

By using an efficient solver mechanism, and an extensional constraint representation, TCPP capitalises on these *don't cares* by using *wild cards* in the associated cells in the table (constraint). In this way, a more efficient encoding is generated, one for which such specialised propagation algorithms produce impressive results on a range of benchmark domains.

The third way to use constraint techniques in planning, completely capturing planning within constraint programming - without restricting the horizon, has been successfully applied using a number of methods. Among these are the use of *structural constraints*, employing *constraint networks on timelines*, using *temporal intervals and attributes*, and the *combination of partial order planning and CSPs*. Although not directly related to the work in this thesis, a brief overview of these techniques is given below.

Whereas several of the above constraint-based planning systems make use of standard CSP formulations (i.e. "static" CSPs, with completely predefined variables and constraints) and setup the solution structure, the Excalibur system [171] reformulates planning as a *structural constraint satisfaction problem* (SCSP) [172]. In this approach, only the types of constraint, together with their possible connections, are given. That is, the constraint graph is not made explicit, with structural constraints specifying the only restrictions on the types of graph permitted. Thus, the planning problem, reformulated as a SCSP, can be solved without explicitly defining the solution structure.

The varying structure in SCSPs is accommodated by the inclusion of *extensible constraints* as well as conventional *non-extensible constraints*, with the former allowing a non-fixed number of variables in a constraint as the search progresses. *Object constraints* are also included to provide structural context information. In contrast to standard CSPs, where the variables are assigned values and this alone generates the search space, SCSPs use *productions* to build constraint graphs. These, in addition to the variable assignments, contribute to the search space. However, since this search space is potentially infinite, domain-specific search knowledge is very important.

The constraint networks on timelines modelling technique (CNT) [173], [174] is a general constraint-based approach used to describe discrete event dynamic systems. When used for planning, it allows a problem to be described as a form of dynamic CSP (see above description of GP-CSP), in which the set of variables and constraints is not fixed. Instead, *dimension variables* are used to represent the, initially unknown, number of steps in the solution plan. A dimension variable's domain may be infinite. Whereas classical planning relies on descriptions of actions, CNT is based on *attributes* and *time steps*. A *timeline* is a set of synchronised attributes, each of which share the same sequence of steps and temporal position. Thus, a CNT is a set of timelines, with each timeline containing a dimension variable and one attribute variable for each attribute at each time step, and a set of constraints on the timeline variables. There is no restriction on the type or number of variables in the scope of such constraints.

CNT has been used successfully to model real-world planning and scheduling problems, including mission management for unmanned air vehicles (UAV) and satellites [168]. The authors note, however, that although the CNT paradigm offers enhanced complex modelling flexibility, this is also the source of one of its weaknesses. That is, modelling a complex problem relies on a human operator choosing between various alternative designs, the relative merits of which are not easily understood.

The Europa planner [175] is another system that completely captures the planning problem in a constraint-based paradigm. Motivated by work on complex concurrent systems, including spacecraft projects, Europa subdivides such systems into components and subsystems. These are referred to as *attributes*, with each attribute representing a concurrent thread. The attribute describes the thread's history as a sequence of states and activities. An *interval* describes a state or activity over time. Finally, a series of *planning domain constraints* determine the legal interactions between attributes. A plan under this paradigm is a mapping of each attribute instance to a sequence of intervals. To be a valid plan, this mapping must also satisfy the planning domain constraints.

The overall framework used in Europa, known as *constraint-based attribute and interval planning* (CAIP), has subsequently been developed into a class library and tool set [176] for constructing and analysing constraint-based temporal planners. Although this modelling paradigm offers an expressive and flexible set of techniques, the authors highlight the need to improve the system's inference capabilities and to improve search, especially by making use of heuristic guidance.

The final constraint-based planner considered here, which completely captures planning within constraint programming, is CPT [177]. CPT is an optimal (temporal) planner that combines the use of lower bounds with a partial-order causal link (POCL) style branching scheme. The lower bounds are calculated using a modified form of the heuristic, h_m, with powerful pruning rules for reduction of the search space implemented as constraints. A key feature of CPT is that it is able to reason about precedences, action supports and causal links involving actions that are not in the current partial plan. By doing this, the system is able to prune supports of actions that are not yet in the plan, exclude actions from the plan, and detect failures earlier. Empirical testing on a range of problem domains showed that CPT was faster than the state of the art. Further, the authors show that on the TOWER-n problem (stacking n blocks in order, with block 1 on top), a solution can be found by using inference alone with no search, thus highlighting the power of their reasoning techniques.

Augmenting CPT with additional techniques, including landmark derived constraints and distance constraints (as used in CPlan), led to the development of eCPT [178]. This variant was shown to be able to solve many instances of IPC problems with no search, again using only inference. These results show the efficacy of the combination of partial-order branching techniques and constraint based inference methods.

The rationale behind CPT and eCPT, as with all constraint based planning, was to reduce the amount of blind search and increase the amount of inference. This is the motivation for the work in this thesis; a desire to make better use of the inherent inference methods provided by the constraint paradigm, and to achieve, more efficiently, solutions to AI planning problems reformulated as CSPs. This is discussed in the following chapters.

2.6 Chapter Summary

This chapter has introduced AI classical planning and CSPs. Firstly, planning representation schemes were discussed. These included state-transition systems, STRIPS, ADL, and the research community standard, PDDL. An alternative to PDDL, SAS$^+$, was introduced. This method of representing planning problems makes use of multi-valued variables and is a convenient starting point for the encoding of such problems as CSPs.

The classical planning search space was discussed, with the differences between state-space and plan-space highlighted. The principle of heuristic search was explained, with techniques appropriate to each type of search space given.

Having provided an introduction to planning, the constraint programming paradigm was outlined. The importance of modelling with CSPs was detailed, and CSP solution techniques were explained. Inference methods, including arc consistency and path consistency, were discussed. Basic search approaches, such as generate and test, and backtracking, were detailed, as were improvements to backtracking in the form of look-back and look-ahead techniques.

The important topic of heuristic guidance for CSPs was introduced, with many examples of both variable and value heuristics given. These generic heuristics, whilst providing good results on certain problems, are not optimised for CSP reformulations of planning problems as they do not make use of the structure of the original problem.

A direct encoding technique was provided in order to show how a planning problem may be recast as a CSP. This led to a review of recent systems that employ various forms of constraint processing to solve planning problems. These were categorised as those that merely post a part of the planning problem to a CSP sub-solver, those that formulate the planning problem as a CSP with a limited horizon, and those that completely capture the unbounded planning problem as a CSP. Discussion of these systems set the context for the work described in the following chapters.

Whilst the systems described above employ different specific techniques, they share the general principle of attempting to leverage the propagation and pruning mechanisms inherent in CSPs to make more efficient the process of solving planning problems. Some are hand coded and, hence, can be modelled more precisely. Others, by using an encoding based on the planning graph, capture more of the structure of the planning problem and thus achieve improved performance. Likewise, for more recent methods, including those that recognise and make use of planning specific features in order to pre-process the problem.

Regardless of the precise techniques employed, the use of CSPs as a means of solving planning problems aims to capitalise on the inference and informed search methods discussed in this chapter. By doing so, the amount of uninformed backtracking search required can be minimised or, in some cases, a solution found with backtrack-free search. With this in mind, the following chapter describes a study that sets out to better understand the propagation behaviour of CSP encoded planning problems.

CHAPTER 3

PLANNING AND PROPAGATION

3.1 Introduction

This chapter details an empirical study that investigates the propagation behaviour of CSP variables during the solution of CSP encoded planning problems. The aim is to gain a better understanding of the level of inference achieved during the process of finding a solution.

The motivation behind the study is the recognition that constraint based planners do not yet match the performance of their state of the art counterparts[1]. This raises the question: Why not? One answer is that, as discussed in section 2.2.6, many successful non-CSP planners use the structure of the planning problem, by means of a heuristic, to guide the search. Without such heuristic guidance, CSP planners must search "blindly" over a search space that can become extremely large, growing exponentially with the problem size. Another consideration is, as the size of the problem increases, the solution plan length increases. With this growing horizon comes a reduction in the impact gained from the goal-state constraints. That is, having been given the partially specified goal state in the problem definition, a CSP solver will infer new constraints on the variables at the previous level as a result of these values in the goal state (final) level. Similarly with the initial state values, the solver can propagate this known information and infer new constraint information at the following levels. The closer the goal state is to the initial state, the greater the impact gained from the interaction between these two sets of constraints. Hence, as the gap (i.e. length of plan) between initial state and

[1]Using results from the International Planning Competitions (IPC): http://ipc.icaps-conference.org/

goal state increases, the impact of the propagation behaviour, due to the constraints on the variables representing these states in the CSP encoding, decreases.

Thus, there are two areas for consideration: The use of planning problem structure, and the impact of inference / propagation. Noting that the planning problem's structural information is lost in the translation to a CSP encoding, and that such a CSP encoding relies on generic CSP techniques for solution of the problem, the study in this chapter sets out to investigate whether or not it is possible to use inherent (hidden) structural information within the CSP encoded planning problem to increase the impact of the existing propagation techniques in order to effect a faster, more efficient solution. In this context, the structural information being considered is a particular subset of the CSP (state) variables. That is, instead of making use of one of the standard CSP branching techniques (e.g. the most constrained variable, which is known as the Minimum Remaining Values (MRV) heuristic), the idea here is to branch on a known value taken from the problem. Such values are comparable to *Backdoors* [179], with this approach further discussed in section 3.4.

Before discussing the empirical study, section 3.2 provides an overview of the tools and general techniques used in this and the following chapters of the thesis. Next, a brief discussion of the favoured constraint encoding is given in section 3.3. The study itself is detailed in section 3.4, with the results presented in section 3.5. The chapter ends with a short summary of the findings.

3.2 Experimental Setup

This section describes the experimental setup used to conduct all of the empirical work in this thesis. All experiments were carried out on a Kubuntu Hardy Linux system with an Intel (3.4GHz) processor and 2GB of RAM. The programming language used was ECL^{i}PS^e [180].

ECL^{i}PS^e, an open-source Prolog-based language, was designed to allow development of large-scale constraint programming applications, such as systems used by airlines and those used in the telecommunications sector. The extended Prolog system is supplemented with a number of constraint libraries, in addition to interfaces to third-party solvers.

Following the basic constraint programming model, an initial, manually produced, CSP encoding of a simple planning problem was developed. Once validated, this procedure was then generalised and automated as shown in Figure 3.1. The standard PDDL domain and problem description files are first processed by the *translator* subsystem of the FD planner [54] in order to produce a SAS$^+$ encoding of the planning problem. This SAS$^+$ encoding is then passed to a series of Perl[1] scripts to be reformed as a constraint model suitable for compilation in ECL^{i}PS^e.

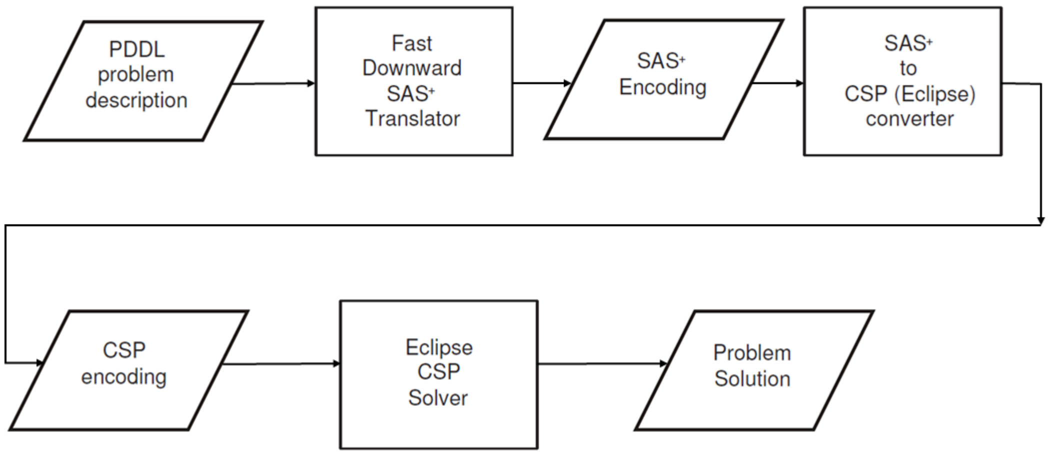


Figure 3.1: Automated PDDL to (ECL^{i}PS^e) CSP conversion process.

The procedure shown in Figure 3.1 makes use of a heuristic estimate to gain an initial seed plan length[2] and was developed with a number of switches and options to allow for experimentation and testing. Where the size of the plan length estimate (and hence the CSP size) was larger than required, the solution plan was completed by padding with no-op actions. No significant impact on performance was observed across the test sets as a result of this.

The raw input to the CSP solver in Figure 3.1 may be visualised, in this work, as a matrix of the form shown in Figure 3.2. This will be referred to as the *problem matrix*. It is important to note that this visualisation does not show the constraints (see section 3.3), but is simply a visual description of the CSP variables and the series of states these represent.

The particular problem represented in Figure 3.2 is the small logistics example introduced in Chapter 2 (see Figure 2.2, page 13). This is problem instance 1 taken from the IPC domain, *Driverlog*[3]. In the problem matrix, a column contains slots for all of the values held by a particular CSP variable during the solution process. That is,

[1]http://perldoc.perl.org

[2]The planner FF[38] was used to generate a plan length estimate, from which the matrix size was taken (no further guidance information was used from this output).

[3]This domain and full problem set are available at http://ipc02.icaps-conference.org/

```
6,0,3,4,1,0,0,0,
_,_,_,_,_,_,_,_,
_,_,_,_,_,_,_,_,
_,_,_,_,_,_,_,_,
_,_,_,_,_,_,_,_,
_,_,_,_,_,_,_,_,
_,_,_,_,_,_,_,_,
_,3,3,4,0,_,_,_
```

Figure 3.2: CSP *problem matrix* for the problem of Figure 2.2, page 13.

row 1, column 1 represents variable 1's value in the initial state, with row 8, column 1 being the slot for the same variable's value in the goal state. Hence, row 1 represents all CSP variables' values in the (fully specified) initial state, and row 8 represents these same variables' values in the (partially specified) goal state. An action operates on the state represented by a given level and produces the CSP state values in the following level. Search progresses down through the matrix, top to bottom, in order to find a solution plan, with this plan represented in the CSP encoding as a list of action numbers instantiated in another array. These two structures (matrix and solution plan array) are connected by the actual constraints used to describe the planning problem's actions (discussed in the next section). A *complete* CSP search routine was used throughout the work in this chapter. That is, a search procedure that explores all alternative choices in the search space. The variables were chosen on the basis of which had the largest number of constraints attached, with the values selected in increasing order from the given domain. This approach was found to be consistently better than the alternatives. Hence, this was the search method chosen for all of the tests in this chapter.

```
6,0,3,4,1,0,0,0,
6,4,3,4,1,0,0,0,                               walkdriver1s2p1_2
6,3,3,4,1,0,0,0,                               walkdriver1p1_2s1
6,2,3,4,1,0,0,0,                               walkdriver1s1p1_0
6,6,3,4,1,0,0,0,                               walkdriver1p1_0s0
6,5,3,4,1,0,1,0,                               board_truckdriver1truck1s0
6,5,3,4,0,0,1,0,                               drive_trucktruck1s0s1driver1
6,3,3,4,0,0,0,0    [5,11,15,20,23,33,45]       disembark_truckdriver1truck1s1

      (a).                 (b).                          (c).
```

Figure 3.3: CSP solver output for the problem of Figure 2.2. (a). *Solution matrix*. (b). Plan action numbers. (c). Action descriptions.

Figure 3.3 shows the solver output for the example problem. Figure 3.3 (a) is the *solution matrix*, Figure 3.3 (b) the solution plan action numbers, and Figure 3.3 (c) is a translation of these back into planning problem action descriptions.

3.3 Constraint Encoding

Before carrying out the study discussed in section 3.4, the initial (ECL^{i}PS^e) constraint encoding used in this work was tested on a number of IPC domains. This encoding was generated from an automated form of the direct encoding procedure described in section 2.4.1. Benchmarking the results of this first test allowed a comparison with the direct encoding[1] used by the CSP based planner, SeP [164]. This comparison is shown in Figure 3.4.

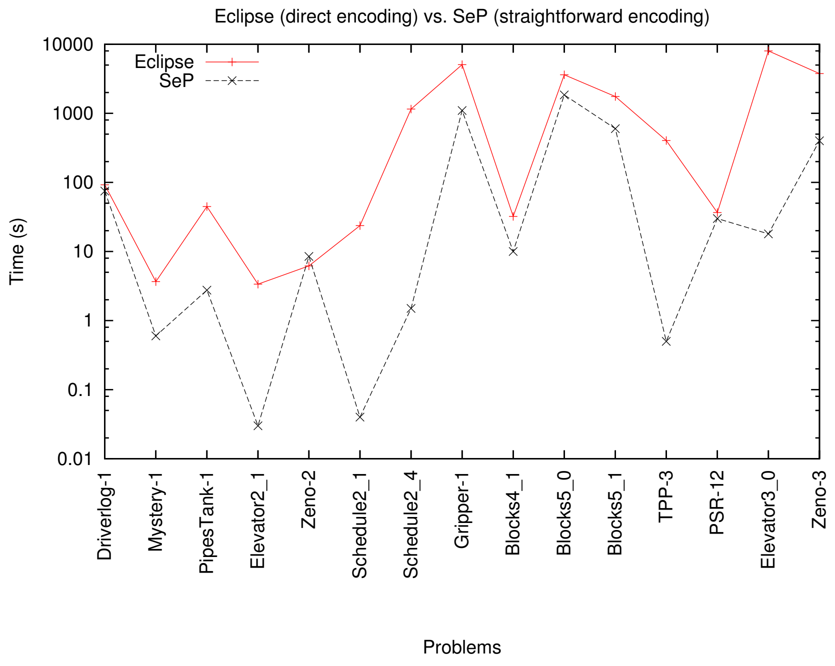


Figure 3.4: Benchmarking the direct encoding against SeP's *straightforward* encoding.

Whilst confirming that the direct encoding was "correct", in the logical sense, the initial testing and comparison with SeP highlighted the fact that the initial direct encoding (*intensional* encoding) provided very little inference during the solution process, as demonstrated by limited domain reduction of the CSP variables via propagation, which was observed during testing. This was due to the nature of the original constraints. That is, these were formulated as a disjunction of conjunctions, with each action definition being a conjunction of the action's preconditions, effects, and number

[1]Referred to as the *straightforward* encoding in SeP.

(see definition 38). The complete set of constraints was then a disjunction of these sets of conjunctions. Figure 3.5 shows part of an example of such a constraint formulation.

Considering the top expression in Figure 3.5, a statement such as $(T[N, 2] =:= 5)$ constrains a particular CSP state variable (represented by element $T[N, 2]$, where N is a time step and 2 is the variable number) to be equal to a certain value, 5 in this case. Likewise for statements containing the alternative equality constraint, $\# =$, (for clarity, $=:=$ is used in statements (in this work) to denote action preconditions, with $\# =$ used in statements representing action effects). That is, a statement such as $(T[Y, 2] \ \# = \ 4)$ constrains the same CSP state variable, variable 2, to be equal to 4 at the **following** time step (represented by Y, where $Y = N + 1$). Thus, these two statements encode, for variable 2, a precondition and an effect of action number 1 ($Actions[1, N]\# = 1$). A statement such as $(T[Y, 5] \ \# = \ T[N, 5])$ is an example of a frame axiom, encoding the fact that, in this example, action 1 constrains variable 5 at time step, Y, to have the same value that it held at the previous time step, N (i.e. action 1 neither requires variable 5 as a precondition nor has any effect acting on it).

```
(      (T[N,6] =:= 1) and (T[N,2] =:= 5 ) and
       (T[Y,2] #= 4) and (T[N,7] =:= 0 ) and
       (T[Y,7] #= 1) and (T[Y,1] #= T[N,1]) and
       (T[Y,3] #= T[N,3]) and (T[Y,4] #= T[N,4]) and
       (T[Y,5] #= T[N,5]) and (T[Y,6] #= T[N,6]) and
       (Actions[1,N] #= 1)
)
or
(      (T[N,6] =:= 1) and (T[N,1] =:= 5 ) and
       (T[Y,1] #= 2) and (T[Y,8] #= 1) and
       (T[Y,2] #= T[N,2]) and (T[Y,3] #= T[N,3])and
       (T[Y,4] #= T[N,4])and (T[Y,5] #= T[N,5])and
       (T[Y,7] #= T[N,7])and (T[Y,6] #= T[N,6]) and
       (Actions[1,N] #= 2)
)
or
.
.
.
```

Figure 3.5: Part of a directly (ECL^{i}PS^e) encoded planning problem.

Thus, the main constraint used in the direct encoding was the equality constraint (in its two, equivalent, forms). For performance reasons[1], such integer equality constraints in ECL^{i}PS^e provide a reduced amount of consistency checking, a type known as *bounds consistency* [11]. Bounds consistency means that the system only computes the consequences for other variables' domains (bounds) when a **bound** of the current variable's domain changes. This contrasts with arc consistency, where any change in the domain of the current variable prompts computation of the consequences for other variables' domains. A simple example illustrates the impact of this reduced form of reasoning. Given two CSP variables, X and Y, each with domain, $Dom = \{1..5\}$, and constraints, $(Con_1)\ X\ \# =\ Y + 1$ and $(Con_2)\ X\ \# \neq\ 3$, bounds consistency will infer $Y = Y\{1..4\}$ and $X = X\{2, 4, 5\}$. That is, the value of $Y = 2$ is not removed from the domain of Y, despite the fact that there is no consistent value for X compatible with it. This limitation[2] means that the constraint solver detects the inconsistency later (i.e. when the constraint is instantiated). This, in turn, means the search tree is larger due to less pruning and, hence, the search for a solution may take longer than, say, if arc consistency was used.

Based on the results of further empirical testing, and on the reported performance of other researchers' reformulated constraint models [164], an extensionally encoded constraint model was considered. In contrast to the intensional representation discussed above, this type of constraint description requires that the permitted set of tuples (variable values) is stated explicitly, generally in tabular form. For example, in Figure 3.5, the frame axiom for variable 1 in the description of action 1 is $(T[Y, 1]\# = T[N, 1])$. In an extensional encoding of this, each of the permitted values would be described explicitly, as shown in Figure 3.6.

Each row in a table constraint of the form shown in Figure 3.6 describes the effect (or frame axiom) of an action on one of the possible values held by a particular variable. In this example, the first figure inside the parentheses represents the action number, the second figure the value held by the variable at time, t, and the third the new value of the variable at time, $t+1$. Hence, in this case, although action 1 does not have an effect which changes the value of variable 1, it is still necessary to state that if the value was, for example, equal to 0 at time, t, then it remains equal to 0 at time, $t + 1$. In this way, for all actions and all variables, all possible values are listed.

Simply using this form of table constraint was found to give little or no improvement over the direct encoding. This is due to the fact that such an encoding mirrors the disjunctive nature of the direct encoding. That is, during constraint setup, constraints

[1]http://eclipseclp.org.

[2]This type of behaviour is typical in commercial constraint solvers [82].

```
Var1(1,0,0).
Var1(1,1,1).
Var1(1,2,2).
Var1(1,3,3).
Var1(1,4,4).
Var1(1,5,5).
Var1(1,6,6).
Var1(1,7,7).
.
.
.
```

Figure 3.6: Part of an extensionally (ECL^{i}PS^e) encoded planning problem.

of the type in Figure 3.6 introduce choice points when called. This contradicts the basic constraint programming paradigm of separating the problem definition from the solution (search) procedure.

The use of the *Propia* library [181] in ECL^{i}PS^e provides a way to better separate the constraint setup and search behaviour when using table constraints. It achieves this by allowing constraints to be formulated without their implementation introducing hidden choice points.

Propia implements *generalised propagation* [182], and its purpose is to transform disjunctive definitions into constraints. By annotating a goal (e.g. a call on the constraints in Figure 3.6) with the *infers* predicate, the *most specific generalisation* of all of the solutions of the goal may be computed, with the information gained being propagated. This is the principle behind Propia. In practice, Propia generally does not actually need to find all solutions, with the amount of information extracted depending on the particular constraint solving library loaded. In this work, the ECL^{i}PS^e finite domain solver, *IC*, is used. This, in combination with the parameter *most* added to the infers predicate, allows for propagation of all of the common information found by the solver. In this way, Propia reduces the variables' domains and ensures that each value in the domain of a given variable is supported by values in the domains of the remaining variables. Hence, assuming all constraint "calls" are defined similarly, Propia enforces arc consistency.

Using the Propia *infers most* annotations, together with table constraints of the form shown in Figure 3.6, a new extensional encoding was developed and benchmarked. The results of this comparison with the original direct encoding are shown in Figure 3.7. Whilst the improvement in performance on the test set gained by using the augmented extensional encoding is significant, it is worth noting that, due to the explicit nature of the table constraints, larger problems may become memory intensive under this representation[1]. Accepting this potential limitation, the table constraint encoding was chosen as the base for all of the work discussed hereafter.

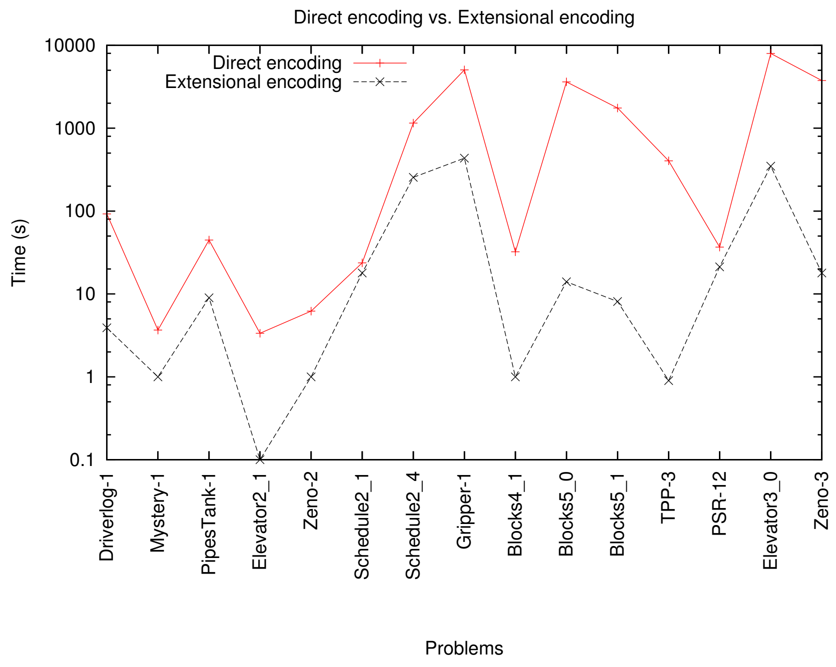


Figure 3.7: Comparison of the direct and extensional encodings.

It should also be noted that the prototype developed for testing the hypothesis in this work does not, at this stage, make use of any additional pruning techniques or performance enhancements. For example, *lifting*, *symmetry breaking*, *no-good learning* [150] and *landmarks* [178] have been used in recent CSP planners. The reasoning behind this is that, with a base system, the results of any testing will be unaffected by potential interactions with such enhancements. The use of *common subexpression elimination* [183] and the addition of *sequence constraints* [184] was investigated during testing of the direct encoding. However, the former improved efficiency (run time) only on certain problem instances, with the additional processing required outweighing the benefits for most problems. The latter (constraints that prevent pairs of inverse actions at consecutive time steps) did provide improvements with very little overhead and, hence, these constraints were included in the base constraint model.

[1]Practical size limits are discussed further in the following chapters, in particular in section 4.4.

3.4 Empirical Study of Trial Backdoor Variables

With the prototype test infrastructure in place, an empirical study was carried out in order to better understand the impact of inference and propagation during the solution of CSP encoded planning problems. This section describes that study.

Backdoors [179] are small sets of variables in combinatorial problems that, once assigned a value, allow the remainder of the problem to be solved in polynomial time. Given a set of backdoor variables, it is possible to restrict the search area by branching only on these variables. However, finding such a set of backdoor variables is, in itself, generally a computationally hard problem [179].

The focus of the study in this chapter is not on the discovery of new backdoor style variables, but instead on the use of these, and on the observation of their impact on the inference and propagation behaviour of a series of CSP encoded planning problems. To achieve this, a previously generated solution matrix will be used to supply known *solution space variables* for testing as potential candidate backdoors. That is, for each of the problems to be tested, a known solution will be used and the variable values found in this solution matrix will systematically be applied to the problem matrix in order to assess the impact of any single solution variable on the solution process. Informally, this approach may be summarised as using the solution to a given problem to solve that same problem, albeit by using the solution one piece at a time. By modifying the system shown in Figure 3.1, the system shown in Figure 3.8 allows each of the variable assignments in the solution matrix to be added, one at a time, to the original problem as an additional constraint. The problem is then "solved" again with the impact of the added variable measured. This variable is then removed, the next one added and so on.

For illustration, a concrete example is shown in Figure 3.9. Figure 3.9 (a) is a visual description of the example problem, with the initial state shown in black text, and the goal state shown in red. A solution to the planning problem is given in Figure 3.9 (b), and the corresponding CSP solution matrix is shown in Figure 3.9 (c). Figure 3.9 (d) shows a translation of CSP variable value to planning problem description for CSP variable two, highlighting in red this variable's value of 2. The test system iterates over the solution matrix, inserting individual values into the problem matrix. For example, row four, column two is CSP variable two with the value, 2. This translates to $driver1$ being placed at location $p1 - 0$ after three steps of the plan (i.e. at row four of the matrix). Supplementing the CSP with this additional information greatly aids solution, as will be shown in the results below, since it is now possible to infer the

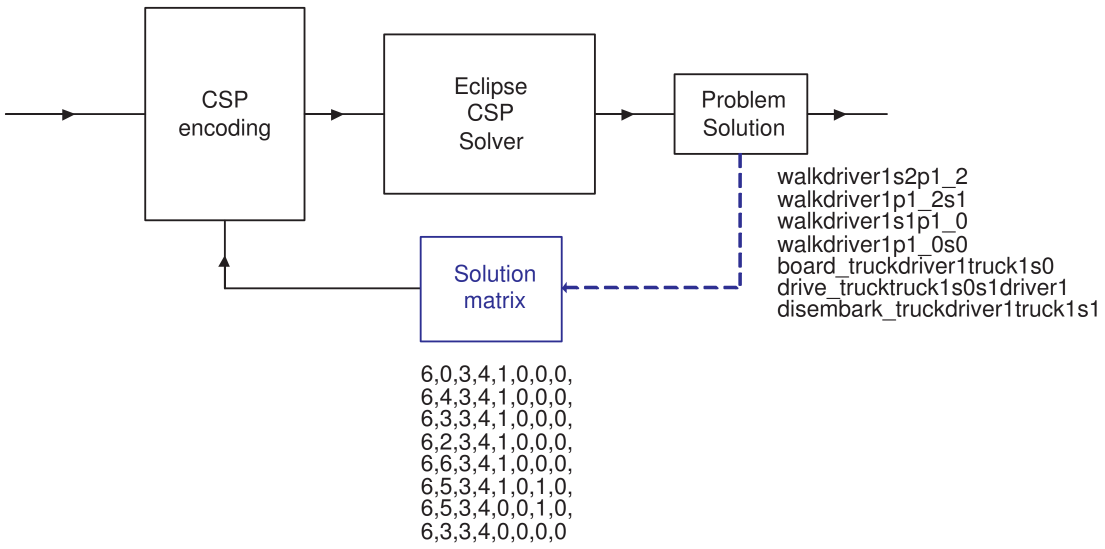


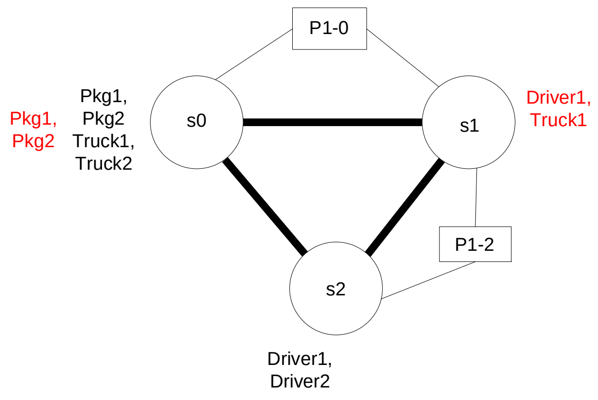
Figure 3.8: Test setup to assess the impact of using backdoor variables.

first three actions. That is, considering Figure 3.9 (a), if $driver1$ is at $p1 - 0$ at time, $t = 4$, the only way this is possible is if the first three actions are: $walkdriver1s2p1_2$, $walkdriver1p1_2s1$, $walkdriver1s1p1_0$. Further, the remainder of the plan can easily be inferred; the goal state constraints force $truck1$ and $driver1$ to be at location, $s1$, at time, $t = 8$. The only way this can be true is if the driver has either walked there or has disembarked from a truck there, and if the truck has been driven there from $S0$ before $t = 8$. Thus, with four actions (steps) available, and with these constraints, the constraint solver has much reduced choice and infers the solution shown in Figure 3.9 (b).

In order to assess the effect that such added variables have, two means of evaluation will be used: Run-time will be measured, with and without the backdoor variable added, and the number of *backtracks* will also be recorded in both cases. The number of backtracks may be considered a “better” metric since run-time can be affected by other factors, although the test environment seeks to minimise these.

The planning domains used for this study will be those used in the benchmarking process. A brief introduction to each is given below, with full descriptions available at the IPC websites[1]. Two additional domains will also be studied. These are extensions of the Driverlog domain and will be discussed in more detail below.

[1]Accessible from links at http://ipc.icaps-conference.org/



(a)

```
walkdriver1s2p1_2
walkdriver1p1_2s1
walkdriver1s1p1_0
walkdriver1p1_0s0
board_truckdriver1truck1s0
drive_trucktruck1s0s1driver1
disembark_truckdriver1truck1s1
```

(b)

```
6,0,3,4,1,0,0,0,
6,4,3,4,1,0,0,0,
6,3,3,4,1,0,0,0,
6,2,3,4,1,0,0,0,
6,6,3,4,1,0,0,0,
6,5,3,4,1,0,1,0,
6,5,3,4,0,0,1,0,
6,3,3,4,0,0,0,0
```

(c)

```
CSPvar2:
  0: Atom at(driver1, s2)
  1: Atom driving(driver1, truck2)
  2: Atom at(driver1, p1-0)
  3: Atom at(driver1, s1)
  4: Atom at(driver1, p1-2)
  5: Atom driving(driver1, truck1)
  6: Atom at(driver1, s0)
  7: <none of those>
```

(d)

Figure 3.9: Driverlog instance 1 backdoor variable example. (a). Visual description of the problem (black text indicates initial state, red indicates goal state). (b). Solution plan (red text indicates position of driver 1 after third action). (c). Corresponding solution matrix (row 4, column 2 highlighted in red shows position and value of CSP variable 2 after third action. (d). SAS^+ CSP encoding showing the meaning of CSP variable 2 holding the value 2.

3.5 Results

This section presents the results of the empirical testing of trial backdoor style variables on a range of classical planning problem instances taken from IPC planning domains. The purpose of this testing is to understand the impact of the additional constraints introduced as a result of adding these variable values to the problem. The thesis being investigated here is that certain constraints, added between the extremes of the initial state and goal state (constraints), will allow better use of the inference mechanisms inherent in the constraint encoding and solver, thereby aiding the solution process and providing a more efficient solution, as measured by both run-time and number of back-tracks.

Referring to Table 3.1, column one shows the names of the problem instances tested. Columns two and three show the standard run-time (RT) and number of backtracks (BT), respectively. Column four lists the trial variables added to the problem matrix for testing (only the three best performing variable / value combinations are shown for each problem instance). The final two columns show, respectively, the resulting RT and BT with the given variable added. The meaning of an entry, such as $(3, 2) = 1$, is that variable 2 is set equal to 1 at time, $t = 3$. That is, after two actions, the value of variable 2 is 1. The two actions are, at this stage, unknown. The idea here is that, by placing such a variable / value combination into the matrix, a form of *stepping stone* will be provided, which will allow increased inference to take place between this constrained value and both the initial state constraints and the goal state constraints. Hence, such variable assignments can be considered to reduce the "distance" between constrained values in the problem, bridging the gap over which blind search will be required. The anticipated result of adding these constraints was that more propagation would occur, leading to fewer backtracks and / or reduced run-time to find a solution.

Domain	Run-time(RT)(secs)	Backtracks(BT)	Var	RT	BT
Driverlog1	3.9	27	$(3,2) = 1$	0.5	0
			$(4,2) = 3$	0.4	0
			$(6,2) = 2$	0.6	0
Mystery1	1	2	$(2,3) = 5$	0.9	0
			$(2,5) = 2$	0.4	0
			$(2,6) = 4$	0.7	0
PipesTank1	9	0	$(2,1) = 3$	8.6	0
			$(2,3) = 4$	9	0
			$(2,7) = 3$	9	0
Elevator2_1	0.1	74	$(4,5) = 0$	0	5
			$(5,5) = 0$	0	6
			$(6,2) = 0$	0	3
Zeno2	1	2	$(3,1) = 2$	0.4	0
			$(4,3) = 0$	0.6	0
			$(5,5) = 2$	0.6	0

Table 3.1: Results of adding selected backdoor variables to the benchmark problems (set 1).

The first problem tested was Driverlog1 (described previously, see Figures 2.2, 2.3, 2.4, and 3.9). The Driverlog domain describes a logistics problem, with trucks being used to deliver packages. Drivers are able to walk between locations in order to access the trucks, and also to move to goal locations, if required, without using trucks (i.e.

when a truck and driver are required to be in different locations in the goal state).

The three entries for Driverlog1 in Table 3.1 are equivalent to placing $driver1$ at locations $s1$, $p1 - 0$, and $s0$ at times 3, 4, and 6 respectively. Intuitively, this should make the problem easier to solve, since forcing the driver to be in a particular place, at a specified time, restricts the actions prior to this state and limits those following it. This follows the same logic as the example described in Figure 3.9 at the end of the previous section. The run-time measures show a considerable decrease, and the number of backtracks required to solve the Driverlog1 problem is reduced to zero upon application of the listed trial variables, indicating that the problem is being solved more efficiently; no backtracking means consistently correct variable assignments and, consequently, a more direct route to a solution. This is to be expected, since, as discussed in Chapter 2, by assigning a given variable the "correct" value, the remaining branches of the search tree, those that represent alternative value assignments for that variable, are pruned.

In order to understand how the additional variables change the CSP representation of the planning problem, two versions of the Driverlog example's problem matrix are given below. The initial CSP problem matrix is shown in Figure 3.10, whilst Figure 3.11 shows the same matrix with a trial variable added (the third from Table 3.1).

6	0	1	2	0	1	0	0	Act_1:	{[1 .. 6, 89]}
{[1, 6]}	{[0, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_2:	{[1 .. 14, 89]}
{[1, 2, 6]}	{[0, 1, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_3:	{[1 .. 18, 89]}
{[1, 2, 4, 6]}	{[0, 1, 3, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_4:	{[1 .. 22, 89]}
{[0 .. 2, 4, 6]}	{[0 .. 3, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_5:	{[1 .. 28, 89]}
{0 .. 6}	{0 .. 6}	{[1, 3, 4]}	{0 .. 2}	0	1	{[0, 1]}	{[0, 1]}	Act_6:	{[1 .. 40, 89]}
{0 .. 6}	{0 .. 6}	{[1, 3, 4]}	{0 .. 2}	{0 .. 2}	{0 .. 2}	{[0, 1]}	{[0, 1]}	Act_7:	{[3 .. 11, 13 .. 19, 21 .. 28, 31 ..]}
{0 .. 6}	{[1, 3 .. 6]}	{[1, 3]}	{[0, 2]}	{0 .. 2}	{0 .. 2}	{[0, 1]}	{[0, 1]}	Act_8:	{[6, 9 .. 11, 13, 14, 17 .. 19, 21, 22, 27, 28 ..]}
{0 .. 6}	1	1	2	2	{0 .. 2}	{[0, 1]}	{[0, 1]}		

Figure 3.10: Driverlog1 standard solution matrix.

Comparing column two in Figure 3.10 to that in Figure 3.11, it is clear to see that, by setting $(6, 2) = 2$, the domain of variable 2 has been reduced at times $t = 3$, $t = 4$, $t = 5$, $t = 7$, and $t = 8$. A consequence of this is that the domains of the action variables (Act_2 to Act_8) are also reduced. That is, by constraining a single CSP state variable to have a specific value at a certain time, not only has the range of values for that variable at previous and future time steps been reduced by propagation, but so too has the range of actions from which a solution plan must be found. This pruning leads to the RT and BT reductions shown for Driverlog1 in Table 3.1.

Whilst recognising the increased propagation shown in Figure 3.11, it is worth noting that the domain reductions in column two do not extend to the CSP state variables in row two, although the domain of action variable, ACT_2, is slightly reduced. Consequently, the domain of the first action slot is not reduced. This limitation does

6	0	1	2	0	1	0	0	Act_1:	{[1 .. 6, 89]}
{[1, 6]}	{[0, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_2:	{[1 .. 11, 13, 14, 89]}
{[1, 2, 6]}	{[1, 6]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_3:	{[1 .. 4, 6 .. 11, 13 .. 15, 17, 18, 89]}
{[1, 2, 4, 6]}	{[1, 3]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_4:	{[1 .. 4, 6 .. 10, 13 .. 15, 17, 18, 20 .. 22, 89]}
{[0 .. 2, 4, 6]}	{[2, 3]}	{[1, 3, 4]}	{0 .. 2}	0	1	0	0	Act_5:	{[1 .. 4, 6 .. 10, 13, 14, 17, 18, 20 .. 22, 26 ..]}
{0 .. 6}	2	{[1, 3, 4]}	{0 .. 2}	0	1	{[0, 1]}	{[0, 1]}	Act_6:	{[1 .. 4, 6 .. 10, 13, 14, 17, 18, 21 .. 28, 31..]}
{0 .. 6}	{2 .. 5}	{[1, 3, 4]}	{0 .. 2}	{0 .. 2}	{0 .. 2}	{[0, 1]}	{[0, 1]}	Act_7:	{[3, 4, 6 .. 10, 13, 14, 17 .. 19,21 .. 28, 31 ..]}
{0 .. 6}	{[1, 3 .. 5]}	{[1, 3]}	{[0, 2]}	{0 .. 2}	{0 .. 2}	{[0, 1]}	{[0, 1]}	Act_8:	{[6, 9, 10, 13, 14, 17 .. 19,21, 22, 27, 28 ..]}
{0 .. 6}	1	1	2	2	{0 .. 2}	{[0, 1]}	{[0, 1]}		

Figure 3.11: Driverlog1 solution matrix with added variable.

not affect the efficacy of the backdoor variables in this problem instance. However, in larger problems, those requiring longer solution plans, the size of the propagation increase relative to the size of the problem will necessarily be reduced. In the matrix this can be viewed as vertical impact (i.e. propagation up and down through time steps above and below the row in which the backdoor variable is assigned a fixed value). That is, in this particular empirical study, the impact of placing a known value in the matrix will depend on how this new constraint interacts with the others in the problem, this being determined by the number of intermediate sub-goals and their associated unsatisfied preconditions. The horizontal impact of a variable assignment will be determined by which other CSP variables are connected to it via constraints (i.e. the encoding of the original planning problem's actions).

It is also important to note that other factors will affect the amount of search carried out. For example, in all areas of the search space, but especially those not reduced by the added inferences, both the variable ordering and the order in which values are selected can have a major impact on the amount of search and backtracking required to find a solution. Also, the structure of the problem will be relevant when assessing search. For example, a problem with many potential alternative wrong search choices (e.g. symmetric ones [69]) will still spend much time exhausting these before reaching a solution.

The Mystery domain is another transportation domain [185] and is described by a planar graph. At each node are vehicles, cargo items and quantities of fuel. Items may be loaded onto vehicles (to their capacity) and the vehicles can travel between nodes. The goal is to move cargo between nodes. However, a vehicle can leave a node only if there is a non-zero amount of fuel there, with that amount decreasing by one unit. In order to disguise the domain, nodes are labelled as foods, cargo items as pains, and vehicles as pleasures. The fuel and capacity measures (pseudo numbers) are encoded as geographical entities, provinces for fuel and planets for capacity. "Just-less" measures are represented by "attacks" and "orbits" predicates for fuel and vehicle capacity respectively. Referring to Table 3.1, the entries for Mystery1 (in order) represent, respectively, $fears(abrasion, rest)$, $locale(pork, quebec)$, and $craves(rest, pork)$ each being true (separately) at time, $t = 2$. Making $fears(abrasion, rest)$ true at time step 2 (meaning the cargo item (abrasion) is in the vehicle (rest) at the earliest possible time) should speed up the solution process since this is the first essential condition for transporting the item to another location. Similarly, setting $locale(pork, quebec)$ true at $t = 2$ (meaning the fuel level is the same as it was in the previous (initial) step) is helpful since this means the vehicle has not moved (i.e. has not used any fuel from this location)) and, hence, another action must have been performed. Assigning $craves(rest, pork)$ at $t = 2$ (meaning that the vehicle is at node $pork$ and therefore has not moved from its initial state location) is also useful since, if it has not moved, another action such as $overcome$ (load) can be carried out. As with Driverlog, the additional inferences made as a result of the extra variables added to the Mystery domain improve the run-time. In terms of backtracks, with the additional propagation gained, the problem is now solved in a backtrack-free manner.

In the Pipesworld (Tankage) domain, the goal is to control the flow of oil products through a network of pipes, obeying various constraints such as product compatibility, tankage restrictions, and (in the most complex domain version) goal deadlines. This domain can be considered to be a transportation domain. However, in contrast to others of this type, moving one object can have a direct impact on the position of another. That is, if one product is pumped into a given pipeline segment, another product may appear at the other end due to the first pushing it through the network. It has been noted [186] that this gives rise to a number of subtle effects. For example, the solutions to certain instances require that the product is pumped, in a cyclic fashion, through a ring of pipe segments. From Table 3.1, the standard results for Pipesworld1 are not improved by the addition of any variables, since the problem is initially already solved with no backtracks, and the run-time is the minimum required to find a solution with this encoding and search method. Interestingly though, and as may be expected, altering the variable ordering and $\setminus$ or the value selection order in the search process for this problem instance leads to a solution containing backtracks (not shown in the results listed). This highlights the relevance of problem (domain and instance) structure; specific problem instances respond differently to different search and variable $\setminus$ value selection techniques (heuristics), with the default experimental test setup search approach being able to solve this problem instance with no backtracks, whereas alternative approaches do not.

The Elevator domain is a model of a simple elevator which services a varying number of floors (f_x) and passengers (p_y), depending on the particular problem instance.

Instance 2_1 contains two passengers, $p0$ and $p1$, and four floors, $f0$ to $f3$. Passenger, $p0$, starts at $f0$, and $p1$ is initially at $f3$. Their respective destinations are $f1$ and $f0$. With the addition of the variables shown in Table 3.1, this problem instance gives an improved or unchanged run-time measure, and exhibits a greatly reduced backtrack count. The variables listed equate, respectively, to $boarded(p1)$ at $t = 4$ and $t = 5$, and $served(p1)$ at $t = 6$. The former indicates that person, $p1$, is in the elevator at time steps 4 and 5, and the latter that person, $p1$, has arrived at the desired destination floor after the fifth action in the plan.

Setting $boarded(p1)$ at time step 4 means that the elevator must have travelled from floor, $f0$ (its initial state condition), to floor, $f3$ ($p1$'s location) and that person, $p1$, has boarded. Potentially, this can be achieved with two actions, up_f0_f3 and $board\_f3\_p1$. However, due to the undirected nature of the forward search, the solver is free to select any applicable action in each state. That is, from the domain of possible values, the solver will select one, limited only by the original constraints and the reductions of the variables' domains made as a result of propagation. In this example, the solver need not select up_f0_f3 and, in fact, it initially chooses up_f0_f1, followed by up_f1_f3 and $board\_f3\_p1$ to satisfy the $boarded(p1)$ condition at time step 4 (imposed by the backdoor variable, $(4, 5) = 0$), added to the matrix). The solver then continues assigning variable values in the rows following time step 4. However, as the solver makes inferences based on both the backdoor variable and the goal state constraints, it is forced to backtrack since the complete problem cannot be solved within the given number of time steps. Thus, in attempting to sequentially complete the rows of the matrix, the solver is forced backwards from the goal state. Backtracking over the sequence up_f0_f1, up_f1_f3 and $board\_f3\_p1$, the solver eventually chooses $board\_f0\_p0$ as the first action (to place person, $p0$, into the elevator) for use later in the plan. Hence, the first three actions are now: $board\_f0\_p0$, up_f0_f3 and $board\_f3\_p1$, which results in both passengers being in the elevator and allows the problem to be solved in the allotted number of steps. This type of backtracking behaviour, resulting in the non-zero backtrack counts for this problem instance, shows that, despite the application of the backdoor variable, there remains choice in the problem; the additional propagation is useful, but it does not eliminate backtracking. It is interesting to note that this particular problem has a small search space (of size 80) and consequently the impact of this backtracking is negligible. However, similar behaviour on a larger problem would be magnified, leading to costly *thrashing* behaviour, as discussed in Section 2.3.2.2.

Considering the remaining elevator2_1 backdoor variables in Table 3.1, setting

$boarded(p1)$ true at step 5 has a similar effect to that just described, except that, additionally, the elevator may also have moved down to $p1$'s destination, $f0$. Setting $served(p1)$ true at time 6 means that person, $p1$, has departed at their destination, thereby again inferring a sequence of actions similar to that described above.

Zeno is the final domain listed in Table 3.1. This problem involves aircraft carrying passengers between various locations. It is possible to travel fast or slow, depending on the mode chosen (fly or $zoom$), with $zoom$ using more fuel than fly. This domain exhibits behaviour similar to the preceding ones when additional variables are added to the CSP solution matrix. For example, setting $in(person1, plane1)$ at time, $t = 4$, implies that one of the first three actions is $board\_person1\_plane1\_city2$ ($person1$'s initial location) and that the others must include either a $zoom\_city1\_city2$ or a fly_city1_city2, since $plane1$'s initial location is $city1$. In this way, despite the original problem solution requiring only a small RT and low BT count, the application of additional variables still provides an improvement in both, leading to a backtrack-free search.

Summarising the first set of results (Table 3.1), it is clear that the addition of backdoor-style variables is useful and leads to increased propagation. On the first two of the five test domains (Driverlog, Mystery), this approach led to backtrack-free search and a reduction in run-time. Although small, these two problems provide a way of easily observing the impact of additional propagation. With respective potential search spaces of size $147,456$ and $361,267,200$, these instances, without a good search strategy, can take longer to solve. Testing on the PipesTankage instance showed that the standard search process alone was finding a solution with no backtracks and, under the current encoding, was doing so in the minimum amount of time. Hence, in this case, the backdoor variables made little or no difference. However, altering the variable and value selection criteria showed that these contribute to the backtrack-free solution result, highlighting the importance of such heuristics. Analysis of the Elevator problem instance showed that, although useful, the backdoor variables do not guarantee backtrack-free search. This highlighted two things: The limited scope of the additional propagation provided by the backdoor variables, and the uninformed nature of the forward search through the CSP. These show that, whilst actions are pruned *locally* around the added variable's row in the matrix, there is no *global* reduction sufficient to provide guaranteed backtrack-free search guidance. The problem from the Zeno domain exhibited reduced run-time and backtrack-free search upon application of the listed trial variables, again confirming the utility of having a fixed value placed in the CSP problem matrix, from which additional inferences may be made.

The results of testing backdoor-style CSP variables on the remaining problem instances in the benchmark set are shown below in Table 3.2.

Domain	Run-time(RT)(secs)	Backtracks(BT)	Var	RT	BT
Schedule2_1	17.9	0	$(2,2)=4$	0.1	0
			$(2,4)=3$	0.4	0
			$(2,3)=0$	0.4	0
Schedule2_4	254.9	20	$(2,1)=1$	1.6	0
			$(2,3)=0$	1.4	0
			$(2,6)=0$	2.1	0
Gripper1	434.9	2,525	$(5,5)=0$	14.1	204
			$(5,3)=0$	10.3	143
			$(6,3)=0$	15.9	239
Blocks4_1	1	91	$(5,3)=2$	0.2	5
			$(6,1)=0$	0.5	5
			$(7,1)=2$	0.3	1
Blocks5_0	14	3,947	$(3,4)=3$	5.3	971
			$(5,5)=6$	0.6	97
			$(9,2)=3$	1.1	140
Blocks5_1	8.1	577	$(5,1)=2$	0.2	8
			$(7,2)=2$	0.4	27
			$(9,4)=5$	0.9	129
TPP3	0.9	3	$(4,1)=3$	0.7	1
			$(5,3)=2$	0.9	1
			$(6,3)=2$	0.9	3
PSR12	21.2	8,719	$(2,1)=3$	6.4	316
			$(5,2)=0$	7.3	731
			$(6,3)=1$	7.8	657
Elevators3_0	348	2,692	$(5,4)=0$	9.4	183
			$(7,5)=0$	1.4	16
			$(9,7)=0$	1.3	16
Zeno3	17.9	97	$(3,1)=2$	9.5	18
			$(4,6)=2$	2	1
			$(5,5)=4$	7.5	4

Table 3.2: Results of adding selected backdoor variables to the benchmark problems (set 2).

The results in Table 3.2 show a similar effect to those problems discussed above. Backtrack-free search is achieved in only two instances (2_1 and 2_4), both from the Schedule domain (the Schedule domain models a machining process in which multiple machines modify objects by, for example, punching holes in them, painting them, and polishing them). Although these instances have a larger number of CSP variables than the problems in Table 3.1, the solution plans are short and, hence, it would be expected that backdoor-style variables, and the associated extra propagation, would lead to backtrack-free search. Instance 2_1 achieves a backtrack-free solution with search alone, and the backdoor variables greatly reduce the run-time. Application of additional variables to instance 2_4 reduces both run-time and number of backtracks.

The Gripper domain is a form of transportation problem consisting of a robot (with two *gripper* hands), two locations, and various numbers of objects (balls) to be moved between locations. The standard solution measures for Gripper1 (Table 3.2) indicate a long run-time and a high number of backtracks. This is, in part, due to the large amount of symmetry in this domain. For example, there is *object symmetry* contained in the choice of which gripper to use (left or right). It does not matter which gripper is chosen for which ball, nor does it matter in which order the balls are transported (*plan symmetry*). However, each of these sets of choices contribute to an explosion in the complexity of the problem. On a given problem, with a bounded plan length, the solver potentially will try all equivalent "wrong" sequences of actions in any particular part of the search space. By applying the variables listed in Table 3.2, which place individual balls at their goal locations at particular time steps, some of these wrong search choices are forced out (see Figures 3.12 and 3.13).

Figure 3.12 shows the problem matrix for Gripper1 with no backdoor-style variable added, whilst Figure 3.13 shows the matrix with variable 5 set equal to zero at time step 5.

```
   3        1        1        1        1        1        0       Act_1:   {[1 .. 9, 35]
{0 .. 4} {0 .. 4} {[1, 2]} {[1, 2]} {[1, 2]} {[1, 2]} {[0, 1]}   Act_2:   {[1 .. 26, 35]
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_3:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_4:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_5:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_6:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_7:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_8:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}   Act_9:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}  Act_10:   {1 .. 35}
{0 .. 4} {0 .. 4} {0 .. 2} {0 .. 2} {0 .. 2} {0 .. 2} {[0, 1]}  Act_11:   {[1, 10, 12, 14, 16, 18, 20, 22, 24, 26, 35]}
{0 .. 4} {0 .. 4}    0        0        0        0     {[0, 1]}
```

Figure 3.12: Gripper1 standard solution matrix.

In contrast to the impact of the backdoor variables shown in Figure 3.11, that shown in Figure 3.13 is much more localised, with the addition of a fixed CSP variable value ($(5, 5) = 0$) resulting in the pruning of the domains of action variables Act_4 and Act_5. This reduction is useful, and accounts for the improved run-time measure and backtrack count shown in Table 3.2. However, there remain large parts of the matrix (search space) to which no pruning is applied, resulting in the non-zero number of backtracks required to solve the problem.

```
    3         1         1         1         1         1         0       Act_1:   {[1 .. 9, 35]
 {0 .. 4}  {0 .. 4}  {[1, 2]}  {[1, 2]}  {[1, 2]}  {[1, 2]}  {[0, 1]}   Act_2:   {[1 .. 26, 35]
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_3:   {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_4:   {[1 .. 3, 6 .. 14, 16, 18 .. 28, 31 .. 35]}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}     0      {0 .. 2}  {[0, 1]}   Act_5:   {[1 .. 3, 6 .. 35]
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_6:   {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_7:   {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_8:   {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_9:   {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_10:  {1 .. 35}
 {0 .. 4}  {0 .. 4}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {0 .. 2}  {[0, 1]}   Act_11:  {[1, 10, 12, 14, 16, 18, 20, 22, 24, 26, 35]}
 {0 .. 4}  {0 .. 4}     0         0         0         0      {[0, 1]}
```

Figure 3.13: Gripper1 solution matrix with added variable.

Blocksworld is a classic benchmark domain and describes the problem of stacking blocks (see Section 2.2.5.1). A block can be on the surface (unlimited table space) or on another block. A robotic arm may lift a block only if there is no other block on top of it. Three instances of the Blocksworld problem are listed in Table 3.2. These are 4_1, 5_0, and 5_1. Each problem shows a considerable reduction in both run-time and number of backtracks upon application of the particular backdoor variables shown. Again, this domain contains a large number of potential unhelpful action combinations, leading to much wasted search effort. As discussed in Chapter 2, the Sussman anomaly can add to this by forcing an "undo" sequence of actions before a solution plan can be found.

The TPP domain describes the problem of buying goods from a market and transporting these back to a depot. Instance 3, as shown in Table 3.2, is solved in a short time with a low number of backtracks using the standard search procedure. Applying backdoor variables maintains or improves this performance. Those that improve performance force goods to be loaded in the truck at specific times which, in a manner similar to previous domains, allows increased inferences which, in turn, provide faster solutions.

PSR is a model of an electrical distribution network and consists of power sources, power lines, circuit breakers, and switches. A problem instance in this domain requires reconfiguration of the network after a fault (e.g. a short circuit) in order to provide power to as many lines as possible whilst ensuring that power is not connected to lines with faults. The actions change the position of switches and circuit breakers in order to reconnect the supply, with the solution plan determining the particular switches, circuit breakers, and an associated ordering. In instance, PSR12, the backdoor variables act in a similar way to those in other domains. For example, the first listed variable for

PSR12 in Table 3.2, $(2, 1) = 3$, is set by only one action and hence, that action must be placed in the first slot (since variable 1 does not hold the value 3 in the initial state). With this set, a series of actions related to circuit breaker, $CB2$, are permitted. Whilst this choice alone does not determine the following plan in any definite way, it does reduce the size of the problem by reducing the search space somewhat. Namely, by determining the first action, it is possible to prune a number of branches in the search tree (representing sequences of actions) which would have as their root, in action slot 1, an action other than that determined by setting $(2, 1) = 3$.

The final two problems listed in Table 3.2 are further problem instances from the Elevator and Zeno domains. Again, these show large increases in performance following the application of trial variables.

Considering all of the results from both Table 3.1 and Table 3.2, it is clear that using this type of structural information (adding backdoor-style trial variables to the CSP problem matrix) leads to increased solver performance on a given problem instance, as measured by reduced run-time and a lower number of backtracks. However, it is also clear that the degree of improvement provided by such individual variables depends on the particular structure of the problem. Relevant structural features include the causal dependencies (between variables), *state invariants* [187], and features such as *almost symmetry* [188] and *symmetry* [69]. As already noted, this planning problem structure, which can be obvious to humans from the PDDL definition, is lost in the translation to a CSP. For example, different types of state invariant [187] can be extracted from the automatically inferred type structure of a PDDL planning domain, with these being used to greatly reduce the number of action instantiations and also to highlight (without search) classes of goals which are unsolvable. Thus, without such structure-based guidance, the CSP solver's variable and value selection (search) policy is not informed by the structure of the original planning problem.

The importance of problem structure is not a surprise since, in (relaxed plan) heuristic-based state-space planning, various structural features have been identified and analysed [39], leading to an understanding of what classes of domains are best suited to this type of solving approach. In terms of the domains used in this (thesis) study, the author [39] found that, under h^+, the local search topology (heuristic cost surface) for Blocksworld, Driverlog, Gripper, Elevators, Zeno, PSR, and Pipesworld had no *dead ends*, while the Mystery and Schedule domains had the potential for dead ends to occur. Domains with no *local minima* include Blockworld, Gripper, and Elevators, while Zeno and Schedule do have minima, but these can be escaped with a maximum (low) number of steps (2 and 5, respectively). Information on which domains contain *benches*, areas where all states have the same heuristic value (under h^+), adds further insight for relaxed plan heuristic planners, since this knowledge could allow for the use of multiple heuristics, with each targeting different domains.

A clear example of the relevance of planning problem structure to the solution of the CSP reformulated problem can be found in the Gripper domain (Table 3.2). Here, there are many alternative (symmetrical) unhelpful action choices available in parts of the solution search space that are outwith the sphere of influence[1] of the trial fixed backdoor variables used in this chapter. In this case, blind search selects from the available values for each variable, and backtracks when such assignments are later found to be inconsistent. Hence, whilst there is an advantage to be gained by using a single backdoor variable, this is a localised gain; the increased inference is propagated vertically in the matrix via domain reduction of the trial variable over a limited range of time steps above and below the fixed value. The lower the number of time steps (actions in the solution plan), the greater the influence of the backdoor variable. Thus, fixing one single backdoor value has less influence as the problem size increases.

In order to further assess the effects of setting backdoor variables, and to better understand the impact on larger problems, further experiments were carried out. The idea here was to investigate a problem larger than those already discussed, one that would allow partitioning of the search space in order to determine the effect on one part of the problem of solving another part of the same problem.

The initial set of problems, that discussed above, was originally chosen since it afforded easy comparison with the SeP planner, but also because the problem instances were all easily solved in a short time (within 30 minutes), making possible the analysis of their propagation behaviour. Larger instances taken from the same domains proved difficult for the solver, with most requiring a run-time measured in hours. Therefore, two new problems were developed for the additional experiments. Each of these problems is a "multiple" of the Driverlog1 problem. The first can be considered as three separate problems in parallel. That is, the problem consists of three times the number of drivers, trucks and packages, with each set being an instance of Driverlog1, as shown in Figure 3.14. The second problem, two instances of Driverlog1 connected in series, is shown in Figure 3.15.

In the first problem (Driverlog1x3Para), the three separate sub-problems have no connection (for example, $Truck3$ can not be used to transport $Driver1$). Therefore, the solution to the overall problem is effectively three solutions, one for each sub-problem. However, there is nothing to determine the ordering of the actions in the

[1]Where this is determined by the causal dependencies between variables.

plan relative to these three sub-problems; all applicable actions are considered in the unguided forward search. Hence, actions operating on the variables representing one sub-problem may be interleaved with those actions operating on the variables representing the others. The combination of the three sub-problems increases the size of the CSP search space from $147,456$ (single Driverlog1 problem) to $3.20618e + 15$ (three Driverlog1 problems).

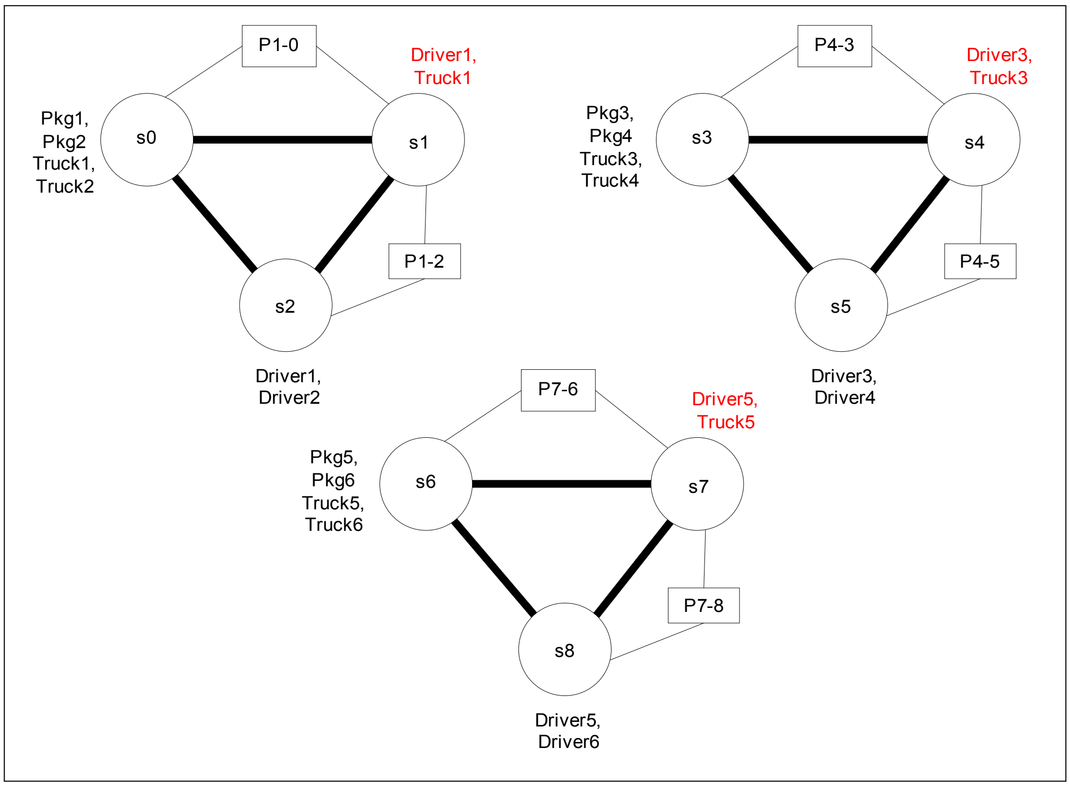


Figure 3.14: Parallel (3x) Driverlog1 problem.

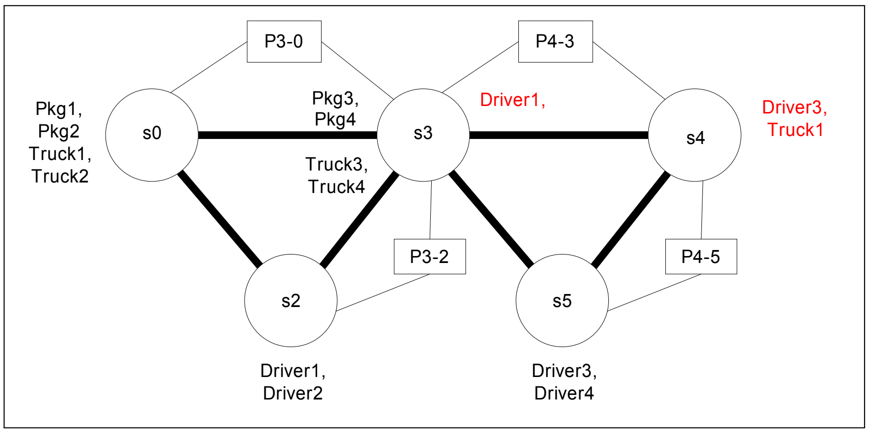


Figure 3.15: Serial (2x) Driverlog1 problem.

As in the first set of experiments, the thesis being investigated here is that, by adding backdoor variables (to any or all sub-problems), increased inferences may be made, with these being propagated, thereby pruning the search space. Considering this idea in regard to the three-way problem invites a further question: Is application of a backdoor variable in one sub-problem sufficient to reduce the search space in that part of the problem by enough to exclude the actions that are used only to solve other parts of the problem. That is, is it possible to partition the problem by adding backdoor variables and, hence, solve the problem more efficiently?

Using a constraint encoding of the problem shown in Figure 3.14, a series of tests was carried out (Table 3.3). Initially, an attempt was made to solve the problem with no additional variables. This was not possible within a time limit of 5 hours. Starting with the first of the three sub-problems (i.e. the first single Driverlog1 part within the three-way problem), the best performing backdoor variable from those listed in Table 3.1 was applied, and the resulting impact on the overall problem solution assessed. Similarly, the following two sub-problems each had their backdoor variable added. These were added individually, with the preceding one removed. Table 3.3 shows that none of these configurations led to a solution within the time limit. Adding all three variables simultaneously also did not lead to a solution within the time limit.

Problem	Variable added	Run-Time (RT)(secs)	BackTracks (BT)
Driverlog1 x 3 (Para)	None	–	–
Driverlog1 x 3 (Para)	Any single sub-problem has a variable added	–	–
Driverlog1 x 3 (Para)	All sub-problems have a variable added	–	–
Driverlog1 x 3 (Para) + Phase Constraints	None	$11,633.2$	$45,467$
Driverlog1 x 3 (Para) + Phase Constraints	1st sub-problem has variable added	$5,863.1$	$30,790$
Driverlog1 x 3 (Para) + Phase Constraints	2nd sub-problem has variable added	$9,719.3$	$31,538$
Driverlog1 x 3 (Para) + Phase Constraints	3rd sub-problem has variable added	$9,100.4$	$28,755$
Driverlog1 x 3 (Para) + Phase Constraints	All sub-problems have a variable added	64	149

Table 3.3: Results of adding backdoor variables to the Driverlog1x3(Para) problem.

Formally subdividing the problem by introducing constraints to focus the search on one sub-problem at a time provided a solution. In this work, these constraints will be referred to as *phase constraints*, reflecting the fact that the problem is being subdivided into phases, with each being solved separately. This additional (hand coded) information being applied to the problem led to the results shown in row four of Table 3.3. Clearly this solution still requires a large number of backtracks and a long run-time. Adding the respective backdoor variables to each phase of the problem provided the improved run-times and backtrack counts given in rows five, six and seven of Table 3.3. Row eight shows the resulting run-time and number of backtracks required when, in addition to the phase constraints, a backdoor variable is added to each phase.

The results displayed in Table 3.3 show that the backdoor variables alone provide insufficient pruning of values in order to render the problem solvable within a reasonable amount of time. However, the combination of phase constraints and backdoor variables does lead to a solution with both low run-time and a low number of backtracks.

In the second problem (Driverlog1x2Ser), shown in Figure 3.15, the two sub-problems are connected. For example, $Truck1$ can (must) be used in both sides of the problem. However, in order to be able to assess the impact of the individual sub-problem's backdoor variables, each sub-problem's drivers are confined to that sub-problem's locations (i.e. $Driver1$ and $Driver2$ are only permitted to be at locations $s0, s2, s3, P3{-}0$, and $P3{-}2$. Likewise for $Driver3$ and $Driver4$ at $s3$, $s4$, $s5$, $P4{-}3$, and $P4 - 5$). To achieve this, an additional predicate was added to the PDDL problem description to allow a particular driver to be restricted to locations within that driver's problem definition.

The CSP encoding of problem, Driverlog1x2Ser, increases the search space size from $147,456$ (single Driverlog1 problem) to $2.0736e + 12$. Table 3.4 contains the results of testing on this problem instance.

The first three rows of Table 3.4 show that no solution was found without the use of phase constraints. As with the previous problem, it was necessary to add these in order to achieve a solution within the time limit (row four). Again, such (manual) constraints focus the search on a subset of the CSP variables (those for one of the sub-problems), whilst effectively forcing the others to remain unchanged for the duration of that search. Rows five and six show the result of adding each sub-problem's individual backdoor variables , and row seven shows the run-time and backtrack count when both sub-problems have their respective backdoor variables added in addition to the phase constraints.

Problem	Variable added	Run-Time (RT)(secs)	BackTracks (BT)
Driverlog1 x 2 (Serial)	None	–	–
Driverlog1 x 2 (Serial)	Either single sub-problem has a variable added	–	–
Driverlog1 x 2 (Serial)	Both sub-problems have a variable added	–	–
Driverlog1 x 2 (Serial) + Phase Constraints	None	$1,714.7$	$6,042$
Driverlog1 x 2 (Serial) + Phase Constraints	1st sub-problem has variable added	$1,448.8$	$4,998$
Driverlog1 x 2 (Serial) + Phase Constraints	2nd sub-problem has variable added	177.6	$1,091$
Driverlog1 x 2 (Serial) + Phase Constraints	Both sub-problems have a variable added	20.2	47

Table 3.4: Results of adding backdoor variables to the Driverlog1x2(Serial) problem.

Summarising the results of Table 3.3 and Table 3.4, it is clear that, as the problem size and search space are increased, the efficacy of the trial backdoor variables decreases. However, upon application of phase constraints, the search space is partitioned, and within this restricted subset of the CSP variables, the extra propagation afforded by the backdoor variables again becomes effective.

3.6 Chapter Summary

The intention in this thesis is to investigate how better use may be made of the constraint paradigm in order to improve the efficiency of solution of constraint encoded planning problems. The work in this chapter has sought to investigate the impact on propagation of using hidden (planning problem) structural information to aid the solution of CSP encodings of such planning problems. The means by which this was achieved was via the introduction of backdoor-style variables that represent this structure. Such variables, in this work, were taken from a known solution to the given problem. Hence, the intention was not to discover these variables, but instead to investigate the result of adding these to the problem matrix.

Having added selected fixed variable values to the problem matrix, increased inference was recorded on a range of problem instances taken from a number of IPC domains. Whilst necessarily small, in order to allow analysis, these problems allowed the general principle to be observed; assigning certain variables specific values leads to a more efficient problem solution, as measured by reduced run-time and a significant decrease in the number of backtracks required.

It was observed that the maximum impact from the backdoor variables was gained on small problems, those with a small horizon (plan length). In these cases, the backdoor variable can be viewed as providing a stepping stone between the constrained values in the initial state and goal state. By reducing the domains of CSP state variables and action variables, the range of available values is trimmed, with a consequent pruning of the search space. That is, less potential for wrong search choices to be made.

Further, it was observed that, in problems with a large number of potentially unhelpful variable value choices (e.g. symmetric values), the sphere of influence of the backdoor variable was reduced. For such problems, as is the case with problems having a larger horizon, the assignment of a backdoor variable remains effective, albeit over a smaller distance relative to the size of the problem.

The final experiments considered problems made deliberately larger, in order to further understand the extent and nature of the impact of using backdoor variables. The first of these showed that, despite simultaneous application of each of the sub-problem's respective best performing backdoor variables, the combined problem remained difficult to solve. However, enforcing a partitioning of the search space, by introducing phase constraints, allowed the sub-problems to benefit from the added inference afforded by application of the fixed variable values. Similarly in the second of the two larger problems, backdoor variables alone did not provide sufficient pruning to allow significantly improved performance. Again, the use of phase constraints introduced sub-problem boundaries, within which the increased propagation, resulting from the insertion of backdoor variables, led to improved efficiency, as measured by reduced run-time and a lower backtrack count.

Considering these findings, it is clear that some form of search guidance (i.e. a heuristic) based on the principles discussed above would lead to more efficient search for a solution to CSP encoded planning problems. Whilst the experiments described in this chapter have used a known solution (the backdoor-style variables) and hand-coded phase constraints in order to reduce the search space, any useful domain independent heuristic would need to discover such aids automatically. Namely, to achieve the increased inferences due to the additional variables, a potential heuristic would require such variables to be made available economically and without the use of a known solution. Further, an automated and systematic procedure to subdivide the problem into "phases" would be necessary. With these requirements in mind, the next chapter discusses an approach that aims to realise such a system.

CHAPTER 4

PROPAGATION FOR DIFFERENT DEADLINES

4.1 Introduction

This chapter continues the theme of the previous one by discussing the use of structural elements from the original planning problem in the solution process of the CSP encoding of that problem. There exist a range of techniques that allow for the use of structural information in planning. Some of these have already been discussed in previous chapters and include: The extraction of *state invariants* [187], [189], [190], the identification of *generic types* [191], the determination of *goal orderings* [192], the revelation of *landmarks* [56], [57], [193] and the use of *backdoors* [194], [195]. Section 2.5 of Chapter 2 discussed how some of these, and similar, methods have been incorporated into CSP based planning. For example, CPlan makes use of manually encoded structural information, GP-CSP encodes structural information via the planning graph, SeP uses symmetry breaking information, Constance relies upon dependent actions, and CPT incorporates landmarks.

The intention in this chapter is to make use of the structure of the planning problem and one of the sources of *leverage*[1] in planning in order to reduce the search space and guide the search choices during assignment of values to variables in the CSP. Hence, this chapter makes use of the ideas discussed in Chapter 3 to develop a planning-specific CSP heuristic. That is, by placing "known" values in the solution matrix and by solving the problem in "phases", a CSP variable and value selection heuristic will

[1]Namely, the imposition of a deadline (sub-plan length).

be described. The first part of the procedure is achieved by subdividing and ordering the goal-state goals, then placing a suitably instantiated variable in the matrix. The second part is achieved by then finding a solution to sub-problems bounded by such variables. This heuristic procedure is detailed in sections 4.2 and 4.3.

The results of applying the heuristic search technique to a range of IPC problem instances[1] are presented in Section 4.4.

4.2 Backward Search and Problem Subdivision

In the previous chapter, the search for value assignments for the CSP variables moved forward from the initial state to the goal state. It is also possible to regress through the problem matrix, from bottom to top, assigning values to variables in reverse order. With no additional guidance, the backward solution process follows a similar pattern to forward search. Namely, values are assigned according to the chosen search scheme until an inconsistency is found, at which time the search procedure backtracks and continues trying alternative values. As discussed, this generally results in thrashing, which can lead to a large number of backtracks and an increase in the time taken to find a solution plan.

Instead of blindly assigning values to variables, a first step toward guiding the search is to consider the search technique employed by one of the early (non CSP) planners, STRIPS, as described in Section 2.2.5.1. Here, each of the subgoals in the goal state is selected in turn, with an operator (action) chosen to satisfy the current one. In this recursive approach, the preconditions of the chosen operator become new subgoals, to be satisfied in the preceding steps of the plan. This technique relies on the assumption of subgoal independence which, as discussed in Chapter 2, can potentially lead to problems in cases where subgoals interact (see description of the Sussman anomaly in Section 2.2.5.1).

Assuming, for the moment, that the subgoals (i.e. individual variables in the final layer of the matrix) in the CSP encoded planning problem are independent and can be ordered appropriately, sequential achievement of these would lead to a completely populated solution matrix and hence, a solution plan. This solution plan would consist of smaller plans, each of which achieves one subgoal. The resulting overall plan will often be suboptimal since there is no opportunity in this approach to interleave actions from multiple sub-plans.

[1]Links to all test domains can be found in Appendix A.

Consider now the situation where, in order to achieve one subgoal, another (previous one) is undone. This is known as clobbering (Section 2.2.5.1). In the context of a CSP, the solver would attempt to re-solve the clobbered sub-problem after solution of the last subgoal in the sequence. That is, an inconsistency would be discovered between a supporting variable's value and the corresponding value of that variable in the initial state. In this way, the initial state's constrained values (goal state in forward search) provide a bound which forces the solver to backtrack over the previous choices, which is likely to increase both the backtrack count and the time required to find a solution. In the case where no ordering of the sub-plans provides a solution, the solver, having exhausted all ordering possibilities via backtracking, would, if so configured, insert additional actions (i.e. extend the bound) in order to re-solve the clobbered subgoal, again increasing the backtrack count and time required to solve the problem.

Thus, with an ordering of the subgoals which guarantees a solution (i.e. no negative interactions), the overall problem can be decomposed and solved by sequentially solving each of the subgoals. Assuming such an ordering, the search will be directed, but will not benefit from any additional inference or propagation. That is, the search, whether backward or forward in direction, is still only constrained in the initial and goal states[1], there is no additional stepping stone allowing extra inferences to be made, as was the case with the backdoor-style variables in the previous chapter.

Hence, whilst the goal focused approach provides a form of search guidance, it does not break up the search space in the way that is required if larger problems are to be tackled. The method described in the following section seeks to address this by introducing a series of intermediate bounds.

4.3 Heuristically Guided Search

In order to construct a heuristically guided search procedure, this section brings together the techniques described above and the methods discussed in Chapter 3. Namely, goal directed search and the use of specific variable values placed in the solution space.

To combat the effects of clobbering whilst using individual subgoals to direct the search process, some form of *goal ordering* is required. This ordering should prevent negative effects (clobbering) arising as a result of achieving a particular goal before another. The use of a causal graph (Section 2.2.6.1) facilitates a useful goal ordering,

[1]These top and bottom bounds obviously move closer as the solution progresses through the matrix.

and this is the technique used in this work. This is described below in Section 4.3.1.

The second part of the combined approach builds on the results of Chapter 3's experiments, and makes use of the idea that a time bound in planning will force out useless actions from the plan, thereby aiding progress towards a solution. Whilst this is essentially what happens (eventually) in a CSP (following thrashing and multiple backtracks), better use can be made of this concept, as shown in Chapter 3. There, known solution information, in the form of a backdoor variable, was fed back in to the problem, creating a fixed point, a known value, within the problem matrix. This intermediate horizon allowed additional inferences to be made, and for these to be propagated up and down through the matrix, thereby pushing out useless actions (i.e. pruning the domains of variables). By recognising that each of the subgoals must be achieved at some point within the solution plan, the technique described in Section 4.3.2 makes use of the idea of intermediate deadlines or horizons to introduce to the solution matrix a series of fixed values.

4.3.1 Goal Ordering

Subgoal ordering has long been recognised as an important factor in AI Planning search, with the GPS system [13] being an early example of its inclusion in a planning system. Since then many goal ordering techniques for both state-space and plan-space planners have been used in planning systems ([32], [30], [196], [197], [198], [199], [200]). Among the differing approaches are techniques which use learning, and those which apply pre-processing to the problem descriptions in order to gain ordering information. An example of the latter approach [192] makes use of the idea that static analysis of the goal structure (found in problem specifications) can reveal orders in which to establish individual goals (which make up conjunctive-goal planning problems). Another such system [201] detects necessary interactions between sets of ground predicates (representing the set of goals in an input planning problem), with the key difference between the two being that the PRECEDE system [192] deals with interactions between lifted goals, which can be identified during compilation.

The goal ordering approach used in this work makes use of the, already available, SAS^+ encoding of the planning problem. Using this representation, it is possible to construct the problem's causal graph (CG) (Definition 11). As discussed in Chapter 2, the CG captures the causal dependencies between the SAS^+ variables. For example, Figure 4.1 shows the CG for problem instance 2 in the Driverlog domain[1]. Each of

[1]This domain and full problem set are available at http://ipc02.icaps-conference.org/

the nodes in Figure 4.1 represent a CSP variable, with a directed edge being present between two nodes if either i.) the value of the destination node may be affected by an action that depends on the value of the source node or ii.) both the source and destination node are affected by an action.

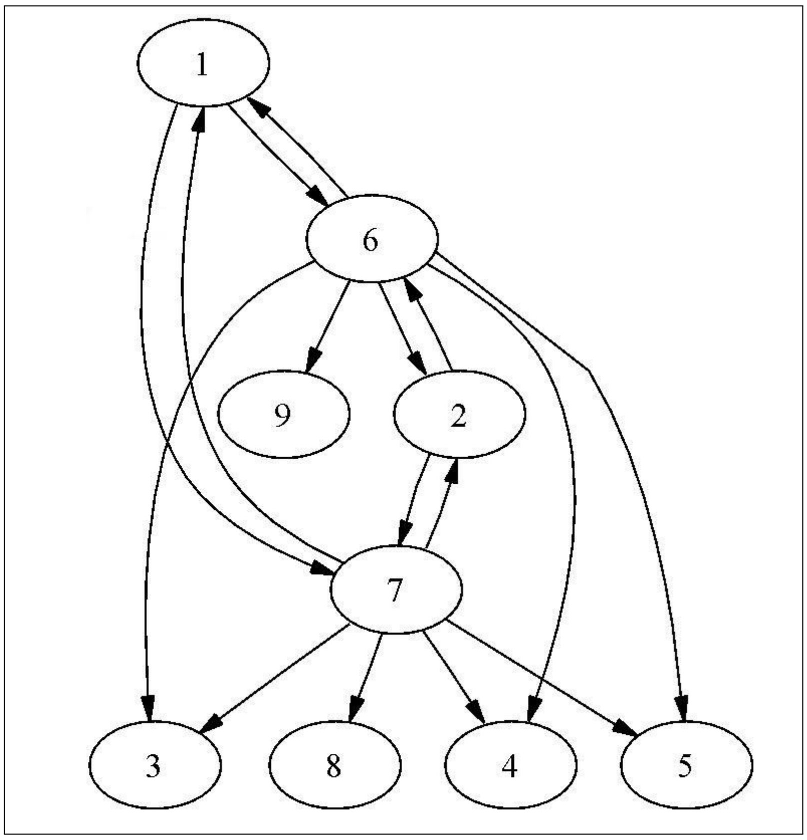


Figure 4.1: Driverlog2 causal graph.

In order to determine a goal ordering from the CG, it is necessary to break the cycles and sort topologically[1]. The cycles are broken by removing the inbound edges of nodes that have the lowest number of outbound edges. Hence, the edges removed affect only those variables with the least number of other variables dependent upon them. In this way, some of the dependencies between the variables are maintained, which is an improvement over a strict subgoal independence assumption. However, some of the dependencies are lost and thus there may arise, for certain non-linear problems, a need to backtrack over the actions chosen as a result of the goal ordering provided by this approach. This is discussed further in the following section.

Figure 4.2 shows the Driverlog2 CG with the cycles removed. Variables 1 and 2 represent drivers; 3, 4, and 5 the packages; 6 and 7 the location of the trucks; with 8

[1]Nodes are ordered such that: If there is a path from u to v, then v appears after u in the ordering.

and 9 being binary variables representing the status of the trucks (empty or not). In this particular example, the goal state specifies the location of both drivers, both trucks, and all three packages. Hence, in the CSP representation there are seven variables assigned values in the goal state, with the remaining two determined by the value held by their respective truck location variables and by the driver location variables (there is, therefore, no need to include these in the goal ordering process).

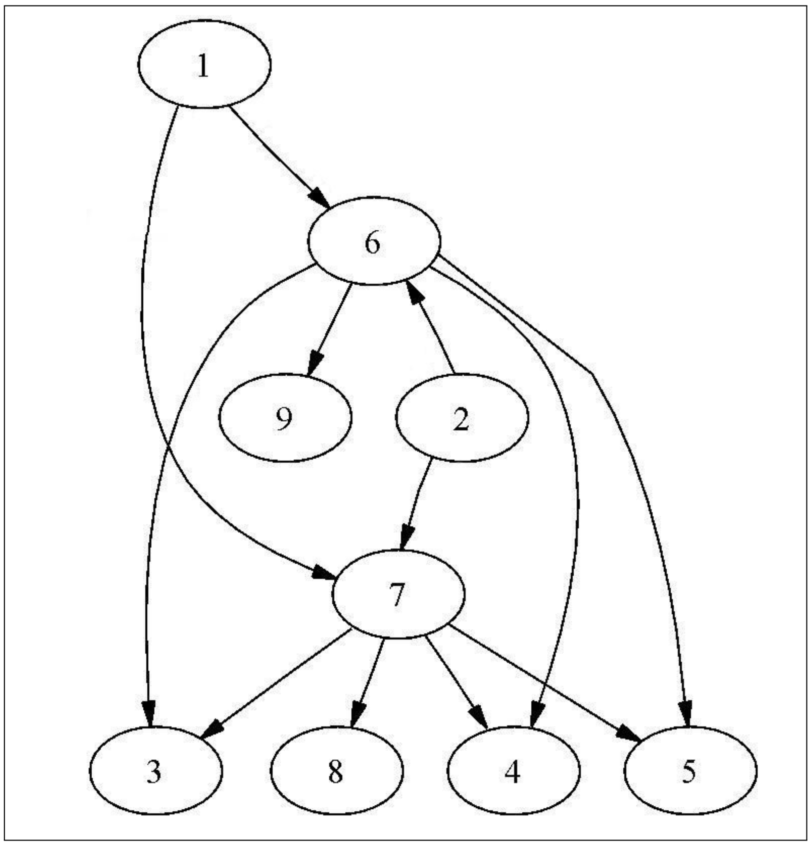


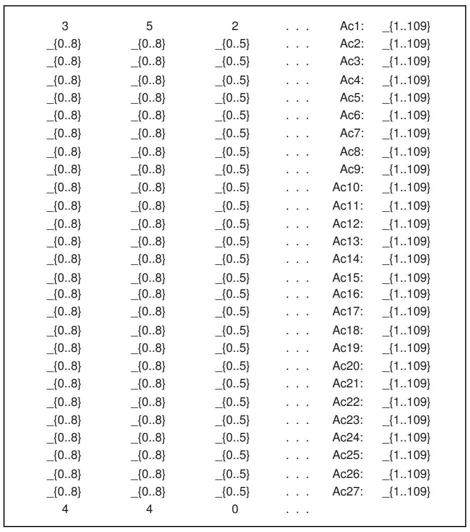
Figure 4.2: Driverlog2 causal graph (cycles removed).

Applying the topological sort to the graph in Figure 4.2, node 1 will be placed before nodes 6 and 7. Node 2 will also be placed before nodes 6 and 7. Node 6 will then be ordered before nodes 3, 4, 5, and 9. Likewise, node 7 will be placed before nodes 3, 4, 5, and 8. This provides a final ordering of: 1, 2, 6, 7, 3, 4, 5, 8, 9. Since variables 8 and 9 do not feature in the goal state, these can be removed. The final ordering is gained by reversing the ordered set, which gives 5, 4, 3, 7, 6, 2, 1. This corresponds to a hierarchy of dependencies, whereby the packages (variables 5, 4, and 3) rely on the trucks (variables 7 and 6), which, in turn, rely on the drivers (variables 1 and 2). Thus, the general ordering of the goals is: Package goals should be achieved first, followed by truck goals, and finally, driver goals. The same ordering technique produces similar results for the other domains in the test set[1].

4.3.2 A Goal Directed CSP Heuristic

The results of the experiments of Chapter 3 showed the efficacy of adding fixed, known values to the CSP problem matrix. These fixed backdoor-style variables were taken from a known solution to the given problem. This section describes a technique which uses this idea, but does not rely on a known solution. Instead, the fixed values are taken from the structure of the problem, in the form of individual variable values found in the goal state. Chapter 3's results also revealed the utility of introducing *phase constraints*. Such constraints focussed the search process on a reduced area of the problem. The technique described below uses this concept by subdividing the problem automatically.

As discussed in the preceding section, it is necessary to order the goal state goals (CSP variables) if these are to be used to construct a solution plan that is made up of individual goal satisfying sub-plans. The approach taken in this section applies to CSPs the idea of solving all goal state goals by subdividing and ordering these goals, and then solving each individually. In order to make use of the idea of gaining extra inference and propagation, the goals guide the search (value assignment) process in the form of a CSP variable and value heuristic. This is coupled with an algorithm which allows these fixed goal values to be placed appropriately in the solution matrix.



3	5	2	. . .	Ac1:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac2:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac3:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac4:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac5:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac6:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac7:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac8:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac9:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac10:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac11:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac12:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac13:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac14:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac15:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac16:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac17:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac18:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac19:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac20:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac21:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac22:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac23:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac24:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac25:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac26:	_{1..109}
_{0..8}	_{0..8}	_{0..5}	. . .	Ac27:	_{1..109}
4	4	0	. . .		

Figure 4.3: Partial solution matrix before constraints applied.

The heuristically guided solution process can be described in terms of the matrix representation introduced in Chapter 3. Referring to Figure 4.3, the partial *solution*

[1]Specific results are described in the results section, Section 4.4.

matrix shows some of the CSP (state) variables (columns 1 to 3) and their associated domains for the Driverlog2 example. The final column shows the action variables, also with their domains. At this stage, only the variables and their domains are represented. The next step is to define and apply the constraints. The constraints represent the grounded actions derived from the planning problem, the means by which the variables change value. Hence, the action in action slot one operates on the variables in the initial state to produce row two in the matrix, and so on.

```
      3           5          2     . . .    Ac1:   _{[1..8,109]}
 _{[0,2,3,7]}  _{[0,1,5,6]}  _{[2,3]}  . . .    Ac2:   _{[1..24,109]}
  _{[0..4,7]}  _{[0,1,3..6]} _{[2,3]}  . . .    Ac3:   _{[1..53,109]}
   _{0..7}      _{0..7}     _{2..4}   . . .    Ac4:   _{[1..72,109]}
   _{0..7}      _{0..7}     _{0..4}   . . .    Ac5:   _{[1..105,109]}
   _{0..7}      _{0..7}     _{0..4}   . . .    Ac6:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .    Ac7:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .    Ac8:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .    Ac9:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac10:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac11:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac12:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac13:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac14:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac15:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac16:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac17:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac18:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac19:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac20:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac21:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac22:   _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac23:   _{[1,2,4..10,13..34,...]}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac24:   _{[5..8,13..21,23,26,...]}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac25:   _{[5,6,13..20,26,27,...]}
  _{[1..4,6]}    _{1..5}     _{0..4}   . . .   Ac26:   _{[5,6,14,16,26,27,...]}
  _{[2,4,6]}    _{[1,2,4]}   _{0..4}   . . .   Ac27:   _{[14,16,56,59,68,...]}
      4           4          0     . . .
```

Figure 4.4: Partial solution matrix after constraints applied.

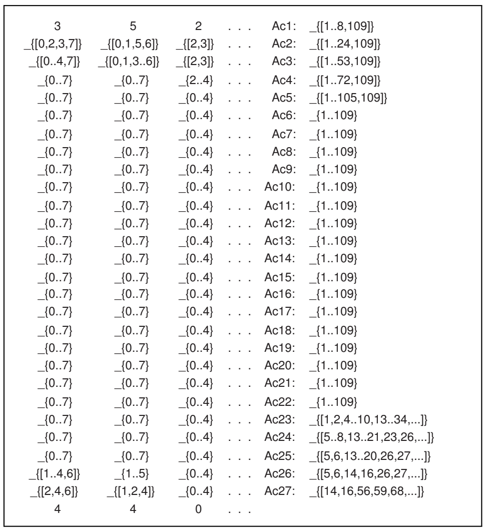
Figure 4.4 shows the matrix following application of the constraints. It can be seen that the constraint solver has used inference and propagation to reduce the domains of the three state variables at layers 1 and 2, and the first two variables at layers 26 and 27. Also, the domains of action slots 1 through 5 and 23 through 27 are reduced. Note should also be made of the reduction by one in the size of all variables' domains. The solver recognises that the SAS$^+$ "none of those" value is never used and simply removes it from the range of available values. Although there has been some constraint propagation, with associated domain reduction, there is not nearly enough to solve the problem; the horizon is too large. The following heuristic attempts to address this.

Using appropriately ordered goals, the first goal is selected and the algorithm (Algorithm 1) regresses up through the solution matrix[1], attempting to find a layer in which the current goal is not satisfied. If such a layer is found, one of a set of *suggested goal achiever actions* is selected and the algorithm moves back down through the matrix until a suitable slot is found for that action. Slot availability is determined

[1]From the "bottom" (uplim) to the "top" (lowlim).

by the domain of permitted actions at a given layer. This, in turn, is determined by propagation of the constraints resulting from the variable assignments made in previous layers. This first part of the procedure is inspired by work [202] in SAT planning.

With an action chosen, and available slot found, the action is placed. This action's preconditions then become new subgoals to be achieved. There now exists a skeleton subplan, bounded at one end by the previous goal's subplan (or initial state) and at the other by the current goal's achieving action. It is now only necessary to find supporting actions for that goal achiever; this is a small, self-contained CSP with a very limited horizon. The solver is able to make better use of inference here due to the propagation of constraints from both ends of the subproblem. Proceeding recursively in this way, blocks of actions are formed into subplans. Upon completion of a subplan, control returns to the highest level and the next goal-state goal is processed.



```
      3            5          2     . . .   Ac1:  _{[1..8,109]}
 _{[0,2,3,7]}  _{[0,1,5,6]}  _{[2,3]}  . . .   Ac2:  _{[1..24,109]}
 _{[0..4,7]}   _{[0,1,3..6]} _{[2,3]}  . . .   Ac3:  _{[1..53,109]}
   _{0..7}      _{0..7}     _{2..4}   . . .   Ac4:  _{[1..72,109]}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac5:  _{[1..105,109]}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac6:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac7:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac8:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .   Ac9:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac10:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac11:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac12:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac13:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac14:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac15:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac16:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac17:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac18:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac19:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac20:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac21:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac22:  _{1..109}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac23:  _{[1,2,4..10,13..34,...]}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac24:  _{[5..8,13..21,23,26,...]}
   _{0..7}      _{0..7}     _{0..4}   . . .  Ac25:  _{[5,6,13..20,26,27,...]}
 _{[1..4,6]}    _{1..5}     _{0..4}   . . .  Ac26:  _{[5,6,14,16,26,27,...]}
 _{[2,4,6]}    _{[1,2,4]}   _{0..4}   . . .  Ac27:  _{[14,16,56,59,68,...]}
      4            4          0     . . .
```

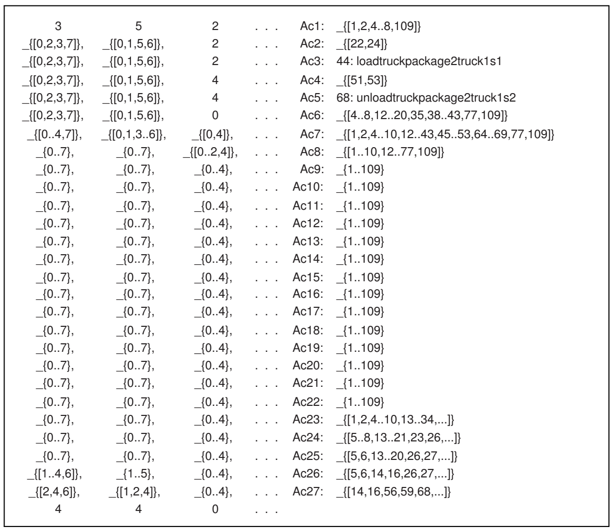
Figure 4.5: Partial solution matrix with constraints applied.

Tracing this procedure through on the matrix shown in Figure 4.5 helps illustrate the operation of the algorithm. The first ordered goal-state goal, variable number 3 (bottom row, column 3), is chosen. The set of suggested actions that achieve the goal value is found, in this case actions 68 and 96. Moving up through the matrix, the first point where the variable is instantiated and not equal to its current value (0) is in the initial state, where it has the value of 2. Proceeding down the matrix now, the domains of the first three action slots do not allow either action 68 or action 96. The first available slot is action slot 4, where action 68 is tried. Propagation of this choice leads to a backtrack, showing that, although there had been some propagation of the initial constraints, there wasn't enough inference to reduce the domain of action slot 4

Algorithm 1: solveMatrix(uplim,lowlim,smax)

```
Input: uplim (Upper limit), lowlim (Lower limit), smax (Solution matrix)
Output: smax (The solution matrix)
goalset ⟵ smax[uplim,(1..numofvars)]
for $s \leftarrow uplim$ to $lowlim$ do
    $q \longleftarrow (s-1)$
    foreach goalvar in goalset do
        if action in layer q then
            if goalvar supported by action in layer q then
                if goalvar is a g-state goal then
                    break;
            else
                if goalvar supported by any row above then
                    if goalvar is a g-state goal then
                        break;
                else
                    $alloc \longleftarrow 0$
                    $acset \longleftarrow getSugActs(goalvar)$
                    foreach $act \in acset$ do
                        findSlotForAction(act,alloc,level)
                        if $alloc == 1$ then
                            $lolev \longleftarrow (level-1)$
                            solveMatrix(level,lolev,smax)
                            break;
        else
            if goalvar supported by any row above then
                if goalvar is a g-state goal then
                    break;
            else
                $alloc \longleftarrow 0$
                $acset \longleftarrow getSugActs(goalvar)$
                foreach $act \in acset$ do
                    findSlotForAction(act,alloc,level)
                    if $alloc == 1$ then
                        $lolev \longleftarrow (level-1)$
                        solveMatrix(level,lolev,smax)
                        break;
```

to exclude action 68. Removing action 68 from slot 4 and continuing down to action slot 5, action 68 is re-placed and propagation occurs again. This leads to the situation shown in Figure 4.6.



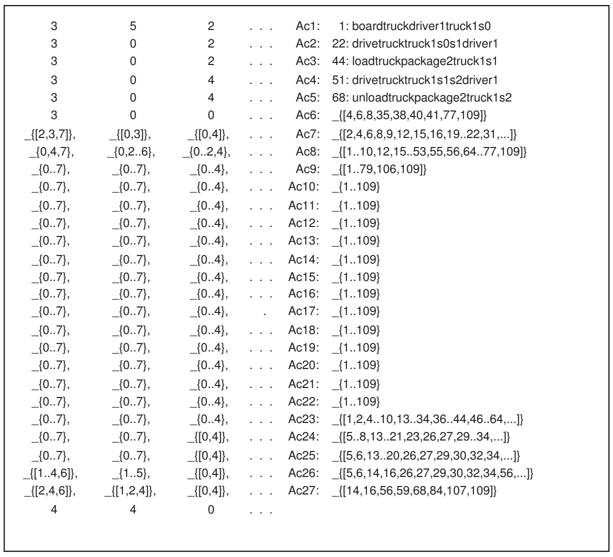
```
      3            5            2        . . .   Ac1:  _{[1,2,4..8,109]}
_{[0,2,3,7]}, _{[0,1,5,6]},     2        . . .   Ac2:  _{[22,24]}
_{[0,2,3,7]}, _{[0,1,5,6]},     2        . . .   Ac3:  44: loadtruckpackage2truck1s1
_{[0,2,3,7]}, _{[0,1,5,6]},     4        . . .   Ac4:  _{[51,53]}
_{[0,2,3,7]}, _{[0,1,5,6]},     4        . . .   Ac5:  68: unloadtruckpackage2truck1s2
_{[0,2,3,7]}, _{[0,1,5,6]},     0        . . .   Ac6:  _{[4..8,12..20,35,38..43,77,109]}
 _{[0..4,7]}, _{[0,1,3..6]},  _{[0,4]},  . . .   Ac7:  _{[1,2,4..10,12..43,45..53,64..69,77,109]}
   _{0..7},      _{0..7},   _{[0..2,4]}, . . .   Ac8:  _{[1..10,12..77,109]}
   _{0..7},      _{0..7},     _{0..4},   . . .   Ac9:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac10:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac11:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac12:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac13:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac14:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac15:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac16:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac17:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac18:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac19:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac20:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac21:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac22:  _{1..109}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac23:  _{[1,2,4..10,13..34,...]}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac24:  _{[5..8,13..21,23,26,...]}
   _{0..7},      _{0..7},     _{0..4},   . . .  Ac25:  _{[5,6,13..20,26,27,...]}
 _{[1..4,6]},    _{1..5},     _{0..4},   . . .  Ac26:  _{[5,6,14,16,26,27,...]}
  _{[2,4,6]},  _{[1,2,4]},    _{0..4},   . . .  Ac27:  _{[14,16,56,59,68,...]}
      4            4            0        . . .
```

Figure 4.6: Partial solution matrix with action placement.

With action 68 in place, the solver has inferred that action 44 must be placed in action slot 3 and has propagated the new constraints arising from these assignments. Continuing the process recursively, the next subgoal, taken from the preconditions of action 68, is chosen. There is now a much smaller search space due to the reduced domains of variables 1 and 2 in the first 5 rows of the matrix. Also, the range of possible values for action slots 1, 2 and 4 is greatly reduced, having been pruned, again by propagation of constraints resulting from the previous action allocations.

The completed subplan for the achievement of the first goal-state goal can be seen in Figure 4.7. It is clear to see that further pruning has occurred in the domains of the first three variables **after** the assignment of the goal-state goal achieving action, action 68, in slot 5. This, together with the reduction of the domains of action slots 6 to 9, aids the choice of actions in the next top-level iteration which constructs the next subplan in order to satisfy the next of the ordered goal-state goals.

This goal-state goal centric algorithm exhibits planning problem, instance-specific, directedness in its guidance. It uses the original planning problem's structure to provide variable and value heuristic direction for the assignment of the CSP variables; it is a planning-specific CSP variable and value selection heuristic. This approach allows the value of the structural information from the planning problem instance to be carried through into the CSP reformulation of that problem. Thus, this structure is no longer



```
     3            5            2       . . .    Ac1:    1: boardtruckdriver1truck1s0
     3            0            2       . . .    Ac2:   22: drivetrucktruck1s0s1driver1
     3            0            2       . . .    Ac3:   44: loadtruckpackage2truck1s1
     3            0            4       . . .    Ac4:   51: drivetrucktruck1s1s2driver1
     3            0            4       . . .    Ac5:   68: unloadtruckpackage2truck1s2
     3            0            0       . . .    Ac6:   _{[4,6,8,35,38,40,41,77,109]}
_{[2,3,7]},   _{[0,3]},    _{[0,4]},   . . .    Ac7:   _{[2,4,6,8,9,12,15,16,19..22,31,...]}
 _{0,4,7},    _{0,2..6},   _{0..2,4},  . . .    Ac8:   _{[1..10,12,15..53,55,56,64..77,109]}
 _{0..7},     _{0..7},     _{0..4},    . . .    Ac9:   _{[1..79,106,109]}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac10:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac11:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac12:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac13:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac14:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac15:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac16:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},      .     Ac17:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac18:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac19:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac20:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac21:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac22:   _{1..109}
 _{0..7},     _{0..7},     _{0..4},    . . .   Ac23:   _{[1,2,4..10,13..34,36..44,46..64,...]}
 _{0..7},     _{0..7},     _{[0,4]},   . . .   Ac24:   _{[5..8,13..21,23,26,27,29..34,...]}
 _{0..7},     _{0..7},     _{[0,4]},   . . .   Ac25:   _{[5,6,13..20,26,27,29,30,32,34,...]}
_{[1..4,6]},  _{1..5},     _{[0,4]},   . . .   Ac26:   _{[5,6,14,16,26,27,29,30,32,34,56,...]}
_{[2,4,6]},   _{[1,2,4]},  _{[0,4]},   . . .   Ac27:   _{[14,16,56,59,68,84,107,109]}
     4            4            0       . . .
```

Figure 4.7: Partial solution matrix with 1st subplan.

lost in the conversion, but is now used to guide the branching process (choosing the next variable, and assigning it a value) in the constraint problem. Further, by inserting the selected variable \value at an appropriate place in the matrix (determined by propagation of existing variable values), this method capitalises on the inherent CSP inference framework by introducing intermediate horizons or deadlines, from which further propagation may be gained. Such deadlines are the mechanism by which the, often very large, CSP search space is subdivided and made tractable.

The efficacy of the goal-directed algorithm is detailed in the following section (Section 4.4) of this chapter.

4.4 Results

This section presents the results gained from using the above described goal-directed heuristic approach on a range of planning problem instances taken from IPC planning domains[1]. This test set is the same as that introduced in Chapter 3 (Section 3.5), with further details and a discussion of each domain's level of complexity available in the planning literature (e.g. [39], [185], [186], [203] and [204]). Figure 4.8 summarises these theoretical complexity measures[2] - the x axis shows the complexity of deciding bounded plan existence, with the y axis indicating the complexity of finding the existence of any plan (unbounded length). It is worth remembering that these are

[1]Links to all test domains can be found in Appendix A.

[2]Results taken from [203], [204] and [205]. Diagram format follows that of [186].

worst-case complexity measures, and that many problem instances from these domains are empirically solvable in polynomial time [186], [205]. Further, it is interesting to note that, among the domains in the bottom left and bottom middle areas of Figure 4.8, both PSR and Schedule require non-obvious decision algorithms [186].

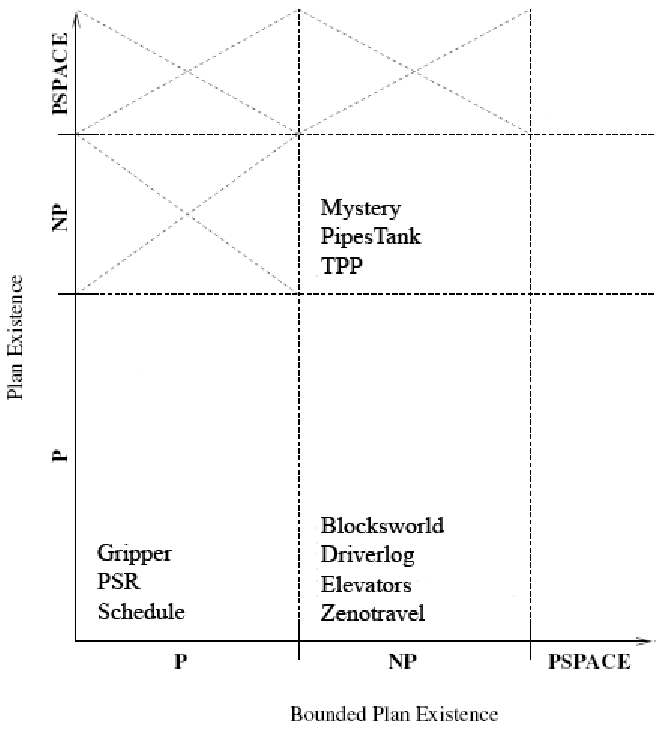


Figure 4.8: Overview of computational complexity results for test set domains.

With the reformulation of each of the problems in the test set as a CSP, it is appropriate to consider the "hardness" of problem instances in this form [206], [207]. Various metrics have been proposed and, intuitively, it would seem that a problem with more constraints would be more difficult to solve. That is, with more constraints on the solution space, fewer combinations of variables' values could feature in a solution (i.e. the solution density will be low). Conversely, with a lower number of constraints, a wider range of combinations of variables' values will lead to solutions. That is, many different paths through the search space will lead to a valid complete assignment of values to variables (i.e. the solution density will be high). Among the metrics used in CSPs (e.g. [208], [209]) are measures of the number of variables, the number of clauses, and various ratios of these. Whilst such metrics give a general idea of a given problem's dimensions, they do not take account of that problem's internal *structure* (e.g. the tightness of the constraints and hence the number and distribution of solutions). Bearing this in mind, a coarse estimate of the “size” of each of the problems in the test set may be gained by considering the number of constraints in each. This includes not only the number of action constraints in the CSP (determined by the number of operators from the planning problem), but also the number of constrained variables in the goal-state (number of goals in the original problem) and the size of the underlying solution matrix. Figure 4.9 shows such a measure for the problems in the test set, as encoded in this work. Table 4.1 provides more detail of each problem’s dimensions.

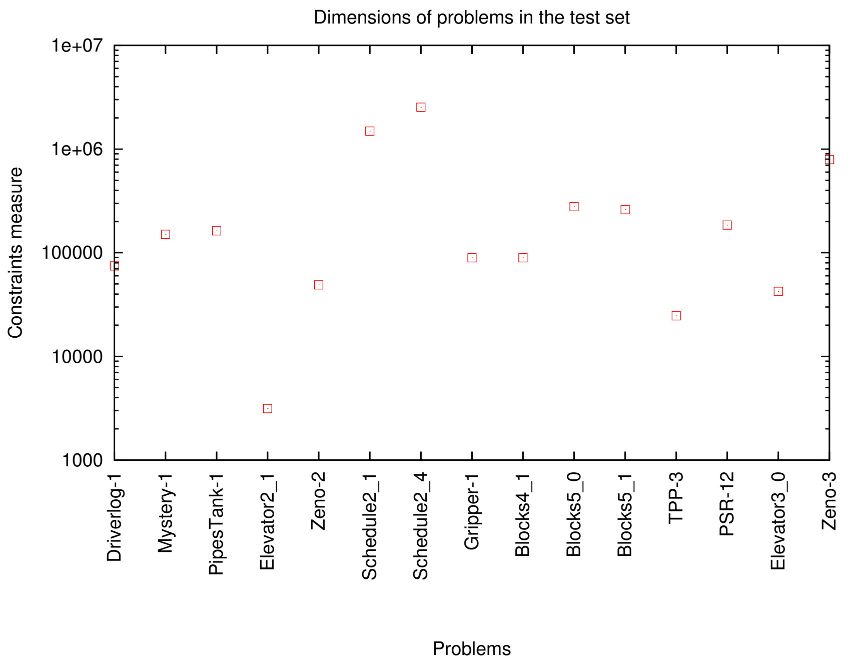


Figure 4.9: Coarse measure of dimensions of the individual test set problem instances. (Number of constraints x Size of the solution matrix).

Although Figure 4.9 gives an indication of the size of each of the problems in the test set[1], and hence gives an idea of the amount of constraint-checking work that may be required, the efficiency of a given solution approach will also depend on other factors. For example, it is known that in CSPs, as in other formulations of combinatorial problems, there exist *phase transitions* [210], [211], [212] between problem instances[2] that are solved (relatively) easily and those that are more difficult. On one side of such a transition lie under-constrained problems, which are soluble. On the other, lie over-constrained problems, which are insoluble (this can be discovered quickly, for example

[1]Details of the size of larger problems in the test set are given in Figures 4.29 to 4.34.
[2]Within a given domain.

by consistency checking). In between these two regions, around the phase transition, problems can be more difficult to solve since they are neither easily solved nor easily identified as being insoluble.

Instance	Number of variables	Number of goals	Size of search space	Number of constraints
Driverlog1	8	4	147,456	2,766
Mystery1	11	1	361,267,200	8,371
PipesTank1	27	4	$2.26e^{+15}$	9,057
Elevator2_1	5	2	80	157
Zeno2	5	2	4,000	1,815
Schedule2_1	128	2	$6.21e^{+43}$	165,898
Schedule2_4	128	2	$4.14e^{+43}$	158,537
Gripper1	7	4	8,748	745
Blocks4_1	9	3	76,832	1,027
Blocks5_0	11	4	2,097,152	2,248
Blocks5_1	11	4	2,097,152	2,248
TPP3	16	3	96,000	342
PSR12	12	6	826,686	882
Elevator3_0	7	3	448	487
Zeno3	8	5	1,327,104	8,352

Table 4.1: Test set problem dimensions.

It has been shown [210] that for many NP problems *order parameters* may be defined, and that around critical values of such parameters the phase transition occurs. This information can then be used to predict the difficulty of a particular problem instance, regardless of the algorithm used to solve it. In addition to those metrics noted above (number of variables, number of clauses, and ratios of these), a measure called *constrainedness* [134], [213] exists. Constrainedness generalises many of the parameters used to measure phase transition behaviour and relies not only on the dimensions of the problem, but also on the expected number of solutions, with this last value determined by the *tightness* of the constraints (i.e. by the proportion of possible values that each constraint restricts). In addition to predicting the location of phase transitions, constrainedness gives insight into why problems at phase transitions are hard to solve. These problems are found to be on a so-called constrainedness *knife-edge* [214], and heuristics that steer the search away from this (e.g. by reducing constrainedness) are found to be effective [125], [134], [214].

Before comparing the empirical results of the performance of this work's planning-specific heuristic procedure (Section 4.3.2) to the performance of a “standard” solution approach[1], it is worth considering how the heuristic guidance operates in terms of reducing constrainedness. As discussed in Section 4.3, the heuristic first orders the problem’s goals using the CG. This has the effect of choosing first the goal variable that depends on the largest number of other variables (e.g. the package variables in the given example). In this way, thinking of a goal-achieving sub plan, the action at its head, which satisfies the actual goal variable’s value, is dependent on the greatest number of variables (i.e. the truck and the driver variables in the example). Hence, it may be considered to be the “most constrained”. By tackling this most constrained part of the problem first, the heuristic is reducing “constrainedness” at the earliest possible time (following the algorithm). It continues to do this for the remaining variables in the hierarchy of dependencies.

In contrast, the standard technique used for comparison[2], which comprised a variable selection method, a value choice procedure and a particular means by which to carry out search, chose the variable with the *largest number of constraints attached*. This provided better results than either a basic *first fail* approach (smallest domain size) or a *most constrained* method (smallest domain, ties broken by choosing the value with most constraints attached). Values for a selected variable were chosen in increasing order from that variable’s domain. This proved more efficient than alternative value selection strategies. Finally, the search technique used was a complete backtracking routine which explored all possible alternative choices[3].

By selecting the variable with the largest number of constraints attached, the standard approach can also be considered, in a way, to be dealing with the “most constrained” part of the problem. However, since this CSP solution approach is applied in a forward direction on the CSP encoded planning problem, row by row, the procedure is considering the “most constrained” variable among the variables currently under consideration (i.e. the current row), not the actual most constrained variable in the planning problem as a whole. This difference highlights how using the planning problem’s structure, in the form of the ordered goal-state goal variables, together with their placement as intermediate horizons within the solution space (matrix), not only guides the variable and value choices, but does so in a way that reduces problem-wide constrainedness.

[1]Detailed below.

[2]Testing, over the range of problems in the set, showed that the following combination of variable and value selection, and search method was the optimum choice to use for comparison, in terms of runtime.

[3]http://eclipseclp.org.

Finally, before analysing the performance of the goal-directed heuristic, it should be remembered (Section 4.2) that a subdivision approach to solving conjunctive goal planning problems is inherently suboptimal since, in the absence of further processing, opportunities to interleave actions, and to capitalise on shared supporting actions, are lost. Thus, it is possible that the solutions found using standard search contain fewer actions than those plans found using the heuristic approach[1].

Considering now application of the heuristic algorithm to the problems in the test set, the purpose here is to measure the potential improvement gained by using the planning-specific, goal-directed guidance as compared to the standard solution approach. As in Chapter 3, both the runtime and the number of backtracks will be measured for each problem instance in the test set. The empirical runtime results are shown in Figure 4.10, with the corresponding measures of number of backtracks per problem given in Figure 4.11. Both measures are also given in Table 4.2.

Instance	Standard RT(secs)	Standard BT	Heuristic RT(secs)	Heuristic BT
Driverlog1	3.9	27	4.8	0
Mystery1	9.7	2	14.7	3
PipesTank1	11	0	10.5	1
Elevator2_1	2.2	74	0.3	0
Zeno2	3.4	2	3.4	1
Schedule2_1	479.7	0	342	0
Schedule2_4	1,041.5	4	664.6	0
Gripper1	169.6	1,667	2.1	8
Blocks4_1	9.7	50	4.3	2
Blocks5_0	510.7	2,282	7,450.2	30,214
Blocks5_1	73.2	339	8	2
TPP3	60.4	2,500	1.5	3
PSR12	16,638.6	143,510	22,390.9	57,034
Elevator3_0	370.1	2,064	1.3	0
Zeno3	64.1	74	11.6	2

Table 4.2: Comparison of standard search with this work's heuristically guided search (runtime and number of backtracks).

[1]A generous initial seed plan length is used to construct the solution matrix for both standard and heuristic methods.

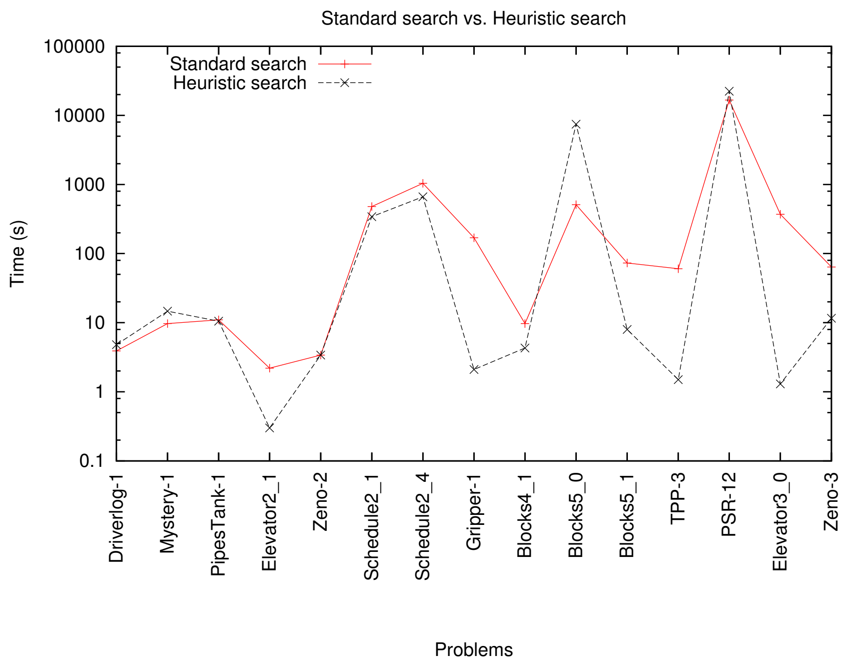


Figure 4.10: Comparison of standard search with heuristically guided search (runtime).

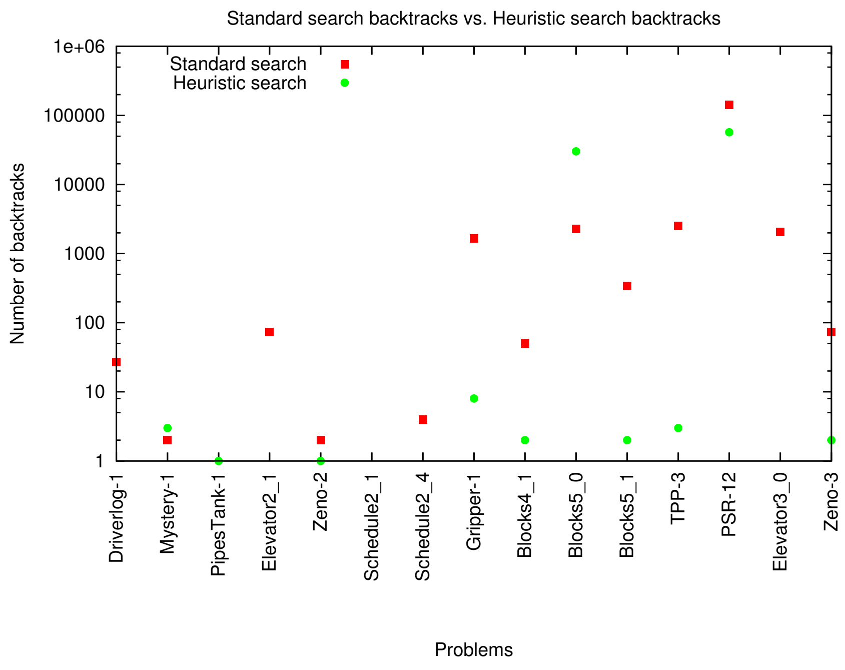


Figure 4.11: Comparison of standard search with heuristically guided search (number of backtracks). Where no value is given, the solution process is backtrack free (i.e. number of backtracks equals zero).

Referring to the results in Table 4.2, Figure 4.10, and Figure 4.11, Driverlog1 is solved in 3.9 seconds using the standard approach. Here, two actions are first selected and placed in action slots 1 and 2. Then, due to the problem shape and lack of propagation, a series of *No-op* actions (Definition 36) are inserted, this continues down to the level at which goal-state constraint propagation provides action slot domain reduction. At this point, the standard search continues assigning values to variables and, despite exhibiting thrashing behaviour, finds a solution with a runtime performance better than that of the heuristic.

Processing the suggested actions and consistency checking are the biggest factors affecting the runtime performance of the heuristic procedure on Driverlog1. Thus, on this problem instance, these can be considered an overhead which, in turn, leads to worse performance, relative to the standard approach. However, although it requires 4.8 seconds to find a solution, the heuristic approach is more efficient, in terms of the number of backtracks, since it finds a solution backtrack-free.

Mystery1 (see Figures 4.10 and 4.11 and Table 4.2), with a larger search space and a higher number of constraints, is also solved faster using the standard technique. Here, the standard search is clearly making "good" choices and arriving at a solution relatively quickly, with a low number of backtracks. The heuristic is observed spending time backtracking over suggested action placement at a given level. That is, the factor that dominates the heuristic runtime measure in this instance is the time spent checking the consistency of an apparently appropriate action. Namely, an action slot's domain indicates, due to the inherent level of inference, that it is safe to place an action. However, when checked and found to be inconsistent, backtracking occurs and the next in the list of suggested actions is tried. In this example, all suggested actions proved to be inappropriate at the given layer, and the following layer of the matrix was then selected, where one such action was successfully placed. This highlights the limitations and trade-offs of limited consistency checking (see Section 2.3.2.1); an action slot's domain may contain particular values which, when actually placed and propagated, prove to be inconsistent, leading to backtracking. With stronger consistency checking this would not occur. However, stronger consistency checking requires additional resources.

The third problem in the test set, PipesTank1, has a larger search space than the previous two instances, with more constraints, and a higher number of variables, each with small domain size. In this instance, the two techniques achieve a similar runtime measure. The choice strategy and inherent inference allows the standard search to run backtrack-free, with a solution returned in 11 seconds. This compares to 10.5 seconds, and just a single backtrack, for the heuristic approach. Here, the heuristic algorithm is more effective than previously, in that over the four goals of the problem, it guides variable and value selection accurately and is not hindered by a significant amount of backtracking.

Elevator2_1 is a very small problem, with a low number of variables, goals, and constraints. In spite of the small size of the problem, the standard search spends most of the runtime (2.2 seconds) backtracking, compared to the heuristic which finds a solution in 0.3 seconds with no backtracking. In this instance the standard method is, as expected, consistently choosing the variable with the most constraints attached which is, due to the structure of the problem, the CSP variable representing the elevator's position. Hence, it can be observed that the backtracking taking place during search represents the assignment and reassignment of values to this variable. That is, continual movement of the elevator for no directed purpose. This contrasts with the goal-focused heuristic which, by first selecting passenger variables, much more efficiently generates correct variable assignments.

Zeno2, another relatively small problem, is solved in 3.4 seconds by both approaches. Here, due to the small problem size, the standard search benefits from propagation of the goal-state constraints, but so too does the heuristic. This leads to the former requiring 2 backtracks, and the latter finding a solution with only 1 backtrack. The backtracking in the heuristic is again the result of placing a suggested action (the only one in this case) for the first goal at a level where it appears appropriate, only for propagation of that assignment to highlight an inconsistency. Another part of the runtime of the heuristic method in this instance is in the placement of the second goal-state goal achieving action. This goal (placing aeroplane 1 at city 2) can be achieved by many (suggested) actions, with each having to be processed for a given level (i.e. tested for membership of the current action slot's domain) [1].

Both Schedule2_1 and Schedule2_4 have a large number of variables, most of which have small domains. This results in a much larger search space. Also, the large number of operators from the original planning problem leads to a higher number of constraints. Whilst each problem instance has only 2 goals, finding a solution using either approach takes a considerable amount of time (see Table 4.2). The standard search method requires only 4 backtracks on Schedule2_4, with none on Schedule2_1, whereas the heuristic solves both backtrack-free. The heuristic technique finds solutions to each faster than the standard method. The dominant factor in these problems

[1]During development, comparison of several ways of carrying out this test found no appreciable runtime difference between them.

is the large number of constraints, and the nature of their encoding; both approaches spend most of the runtime processing these constraints. It is clear that the extensional encoding of the preconditions, effects, and frame axioms required to represent the planning operators expands considerably the overall number of constraints needed and has a negative impact on performance in this domain. This behaviour was anticipated (see Section 3.3), with possible remedies including alternative encodings or the use of a different solver framework.

The eighth problem in the test set (Table 4.2), Gripper1, is solved by the heuristic in 2.1 seconds, with only 8 backtracks. These backtracks result from the above described phenomenon (placing an action in a seemingly appropriate slot), with each of the two suggested actions for each of the four goals causing a single backtrack due to this. In contrast, the standard approach (without any "symmetry breaking") exhibits significant thrashing behaviour, backtracking 1,667 times, and runs for 169.6 seconds before finding a solution. In this domain, the heuristic, due to the goal-focused nature of the algorithm, removes some of the problem's inherent symmetry. That is, by choosing just one goal to achieve, the only decision left relates to the choice of gripper to be used, and this is determined by the natural order of the problem's operator listing.

The next three (Blocksworld) test set instances are, in one sense, more challenging for the heuristic approach. This is due to the multi-valued encoding (SAS$^+$) [36] process. Here, a Blocksworld problem is encoded using a single variable to represent the gripper hand being empty or not, and two variables per block to represent both a block's position and whether or not it is "clear" (has another block on top). The causal graph generated for this type of problem has many connections between the variables (e.g. see Figure 4.12 for the Blocksworld4_1 CG). Comparing Figure 4.12 to Figure 4.13 (Driverlog2's CG), it is clear that Figure 4.12 has many more arcs connecting the state variables, while these problems have the same number of CSP (state) variables. In Figure 4.12, variable 9 (gripper hand) has incoming and outgoing links to all of the other variables (blocks). Variables 5 through 8 have incoming and outgoing connections to all other variables, with variables 1 through 4 being connected in both directions to variables 5 through 8 (and 9). The impact of removing cycles in such a graph is much greater than doing so in, say, Figure 4.13 (or other instances with "local" cycles). That is, the loss of causal dependency information is greater due to the higher number of cycles removed. Hence, since the CSP depends upon this information for goal ordering (Section 4.3.1), the accuracy of the heuristic guidance in such cases is reduced.

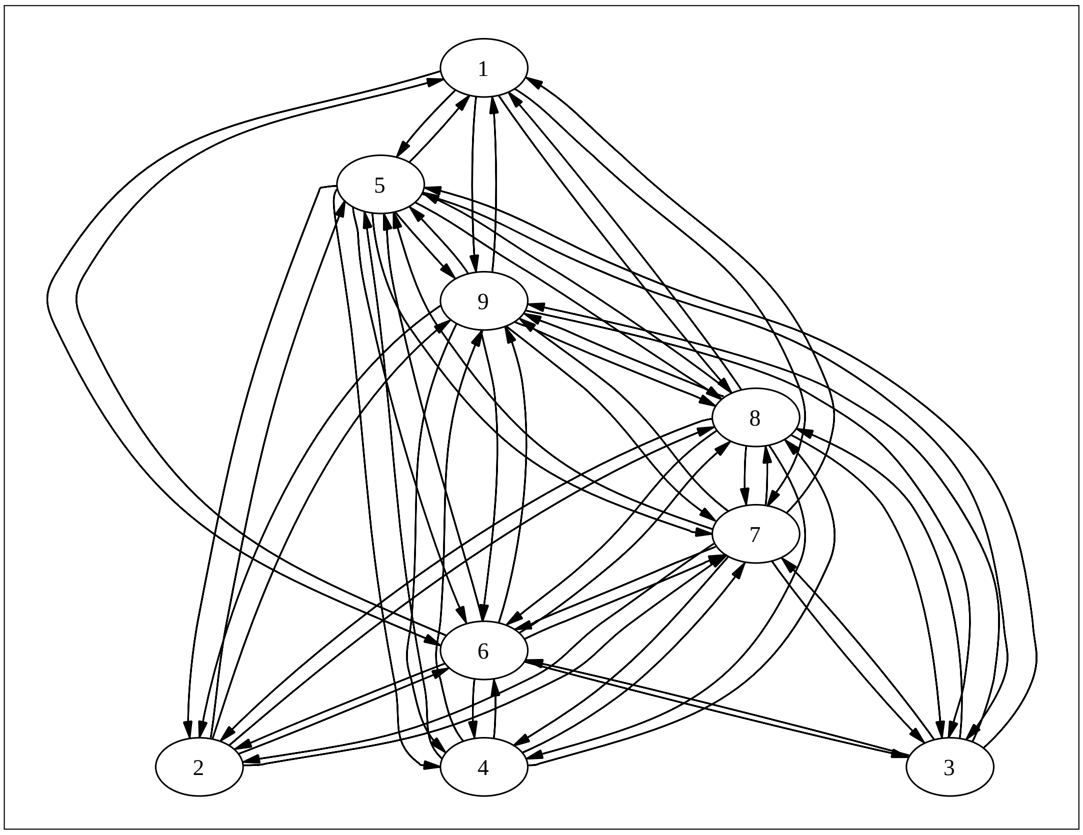


Figure 4.12: Blocksworld4_1 causal graph.

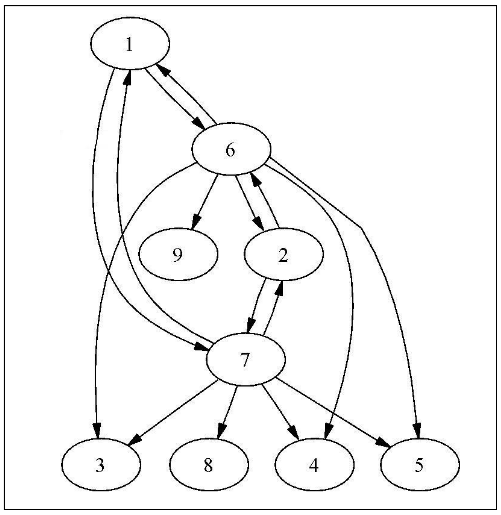


Figure 4.13: Driverlog2 causal graph.

Despite the above limitation, which forces a reliance on the fall-back standard search within the heuristic algorithm, Blocks4_1 and Blocks5_1 are solved by the heuristic method with less backtracking, and in a shorter time, than by the standard search. Thus, there is still a benefit to be gained from goal-based direction. However, Blocks5_0 requires considerably more time, and more backtracking, to find a solution using the heuristic compared to the unguided, standard technique. This is due to Blocks5_0 being an example of the "Sussman" problem (discussed in Section 2.2.5.1). That is, a non-linear problem for which the solution process involves the interleaving of actions required to achieve more than one goal.

The heuristic's poor performance on Blocks5_0 is due to a combination of two features: the backtracking over suggested actions, and the default to standard search when no support actions can be found to populate the subproblem. As discussed above, the necessarily limited level of consistency checking leads to suggested action placement and subsequent removal when an action is found to be inappropriately placed at a given level. The backtracking here is compounded by considerable backtracking during the default to standard search when no solution to a given subproblem is found. In summary, because of the lack of (correct and complete) goal ordering information, the heuristic works towards a (non satisfiable) goal ordering which, in turn, leads to repeated thrashing behaviour, ultimately resulting in a final phase of blind search to "repair" the partial plan; this considerably increases the amount of searching as compared to the standard search technique.

The TPP3 problem instance is a simple transportation style problem, and has the CG shown in Figure 4.14.

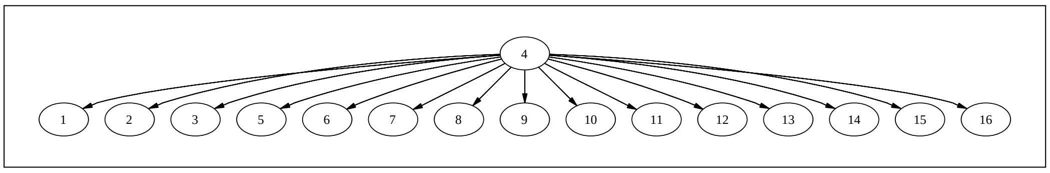


Figure 4.14: TPP3 causal graph.

Since Figure 4.14 has no cycles, no information is lost in processing the CG and, due to the structure of the problem, the only limitation is that goals involving any of the lower level variables (related to the placement of goods) should be considered by the algorithm before those involving variable 4 (representing the single truck's location). Hence, the three "goods" goals in the problem can be solved in any order (since there is no dependency between lower level variables), and the gain here, for the heuristic approach, is in the general goal-driven variable selection rather than the ordering between those lower level variables. The standard search, however, again exhibits significant thrashing. By choosing the "most constrained variable" among those in the given row being considered (the truck), a drive action is chosen. This then leads to the next row's most constrained variable (one representing "goods3") being chosen and assigned a value ("ready to load") which, when propagated, is found to be inappropriate. This process is repeated, until the goal-state constraints force backtracking, which eventually leads to a solution.

Solutions to problems from the PSR domain [215] start with "wait" actions (opens all circuit breakers affected by a fault), and can be solved (optimally) in polynomial time, but require a complex algorithm to do so [186]. The heuristic algorithm performs poorly on PSR12, with this again being due to poor goal ordering information. Because of the domain's interesting structure, the CG (Figure 4.15) contains many cycles, the removal of which reduces the efficacy of the heuristic guidance. Figure 4.15 contains 47 edges, with the corresponding acyclic version having only 28. Further, since the solution plan required to solve the problem contains a sequence of actions which translates into a toggling back and forth of the value of certain CSP variables, the standard goal-focused approach will not provide a (cost effective) result. Namely, for such (toggling) goal values, once they have been achieved, the heuristic approach will consider the next goal, assuming the previous one will not be undone, and require to be redone at a lower level of the matrix. Thus, in a manner similar to the Sussman type of problem, discussed in relation to Blocks5_0 (above), any solution will rely on standard search to "repair" the partial solution found by the heuristic alone. Despite the large amount of time required for the heuristic to find a solution, the number of backtracks remains considerably lower than that of the standard technique.

The final two problem instances from the test set (Table 4.2), Elevator3_0 and Zeno3, are solved more efficiently by the heuristic approach, with solution times of 1.3 seconds and 11.6 seconds respectively. The standard search spends longer on each (370.1 and 60.1 seconds), with 2,064 backtracks for the first and 74 for the second. In contrast, a backtrack-free solution is achieved by the heuristic for Elevator3_0 and that for Zeno3 is found with just 2 backtracks. These results are due to both problems being of the transportation type, with each having a causal graph containing no cycles. Hence, an appropriate goal ordering can easily be found for each.

Before concluding this section with further test results from a wider range of instances taken from the test set domains (and others), it is of interest to discuss the *quality*[1] of the plans produced by both the standard and heuristic search methods. Table 4.3

[1]http://www.plg.inf.uc3m.es/ipc2011-deterministic/CompetitionRules

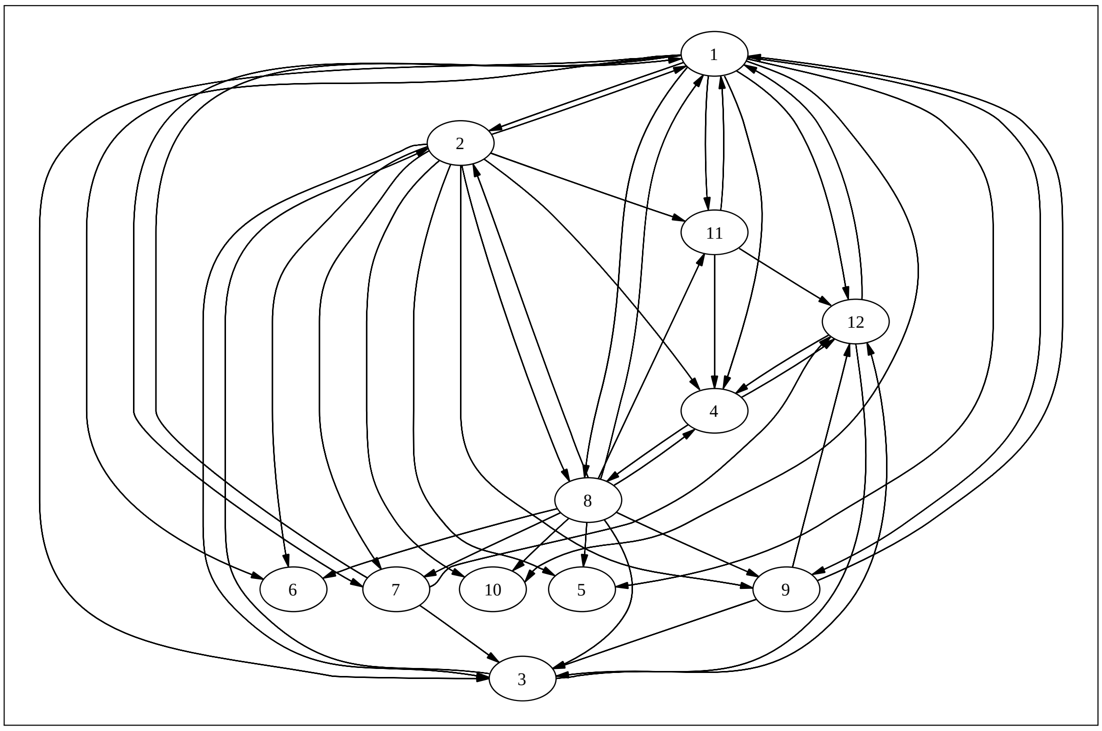


Figure 4.15: PSR12 causal graph.

shows the plan length for each of the instances in the test set, as well as indicating the corresponding optimum plan length.

The plan length measures recorded in Table 4.3 assume a uniform action cost of one. That is, these problem instances do not make use of the *PDDL requirement* `:action-cost`. Thus, as well as aiming to solve the problems in the test set in the least amount of time, in the most efficient way (lowest number of backtracks), the lowest "cost" (i.e. shortest) plan is considered best.

The heuristic produces optimal plans for the first seven instances in Table 4.3, whereas the standard approach only achieves this in two cases. In the Gripper problem instance, the heuristic requires four actions more than the optimum eleven required for the standard search to find a solution. This is to be expected since the single goal focus of the heuristic means that two balls, using both gripper hands, will never be carried simultaneously. Therefore, the heuristic requires four additional "move" actions to achieve all of the goals. For Blocks4_1 and Blocks5_1 both techniques achieve optimal solutions. However, for the Blocks5_0, TPP3, and PSR12 instances, only the standard approach achieves this. For the first and last of these, the heuristic, as discussed above, relies on standard search following a failure to reach a solution, with the final plans including extra actions required to "repair" the initial goal-directed plan. The heuristic's

Instance	Standard Plan Length	Heuristic Plan Length	Optimum Plan Length
Driverlog1	10	7	7
Mystery1	5	5	5
PipesTank1	6	5	5
Elevator2_1	8	7	7
Zeno2	6	6	6
Schedule2_1	4	2	2
Schedule2_4	4	3	3
Gripper1	11	15	11
Blocks4_1	10	10	10
Blocks5_0	12	24	12
Blocks5_1	10	10	10
TPP3	11	15	11
PSR12	12	24	12
Elevator3_0	11	11	10
Zeno3	9	6	6

Table 4.3: Comparison of standard search with heuristically guided search (length of plans produced).

TPP3 solution plan needs four additional actions for exactly the same reason as does the Gripper instance, the single goal direction precludes carrying out a series of *buy goods* and *load goods* actions together followed by a series of *unload goods* actions, separated by a single *drive* action. Instead, each *buy goods*, *load goods*, and *unload goods* series has a dedicated *drive* action. For the final two test instances, Elevator3_0 and Zeno3, the heuristic achieves an optimal solution for the latter, and a near optimal solution for the former, with the additional action in Elevator3_0 again arising from the (single) goal centric nature of the heuristic; here it fails to board a person "on the fly" who is departing at the same destination as another person already in the elevator. Namely, sticking rigidly to the one-goal-at-time guidance prevents such "useful" actions being slotted in to a nascent plan.

Summarising so far (see Figures 4.9, 4.10 and 4.11, and Tables 4.2 and 4.3); on linear problems, where a strong goal ordering can be achieved from the problem's CG, the heuristic technique will find a solution. Such solutions are, for the smaller problems in the test set, optimal or near optimal. For problems where the ordering doesn't force reliance on standard search, but instead merely impacts on the plan length (e.g. Gripper), the (single) goal focus of the heuristic prevents the consideration of actions other than those that directly support the achievement of a given goal (or subgoal). This has the effect of producing a series of subplans which, whilst achieving all of the goal-state goals, does so in an inherently suboptimal way (e.g. each subplan contains a *drive* action instead of one common *drive* action after the subplans, as seen in both Gripper and TPP instances). Where the ordering information gained from the CG is insufficient, and forces the heuristic algorithm to rely on standard search, there is a concomitant increase in both the number of backtracks and runtime required to find a solution (e.g. Blocks5_0 and PSR12). In both examples, there is also an associated increase in the length of the plan produced. Finally, a significant factor is the overall "size" of the problem, as measured by the number of constraints and the magnitude of the solution matrix (see Figure 4.9 and Table 4.1). For the extensional encoding used in this work, in combination with the chosen solver, the dominant factor for certain (linear) problems is the size alone. Clear examples of this can be found in the Schedule domain (see Figures 4.10 and 4.11 and Table 4.2) where, despite there being no (or very small amounts of) backtracking, both the heuristic and the standard search methods require a considerable amount of time to find a solution. This notwithstanding, the (relative) benefits of the heuristic approach remain visible, with a reduced runtime and plan length, and, where possible, a lower backtrack count (Schedule2_4). With this in mind (dominance of the size of the problem), it is potentially interesting to briefly consider this coarse measure of problem size as a predictor of the relative runtimes for the problem instances used in the test set.

By first scaling[1] the measures used in Figure 4.9, then plotting these against the runtime results for both the standard search and the heuristic search (as shown above in Figure 4.10), it is possible to compare (Figure 4.16) a measure of a given problem's size with the time taken to find a solution to that problem.

Considering Figure 4.16, in general a problem's size is a reasonable predictor of the order of the amount of relative effort (time) required to find a heuristically guided solution to that problem. The exceptions to this are those non-linear problems discussed above, Blocks5_0 and PSR12, which rely on a standard-search repair phase.

The results of further testing of the heuristic versus the standard search, on additional test set domain instances[2], are shown below in Figures 4.17 to 4.28.

[1]Dividing by 10,000.

[2]Links to all test domains can be found in Appendix A.

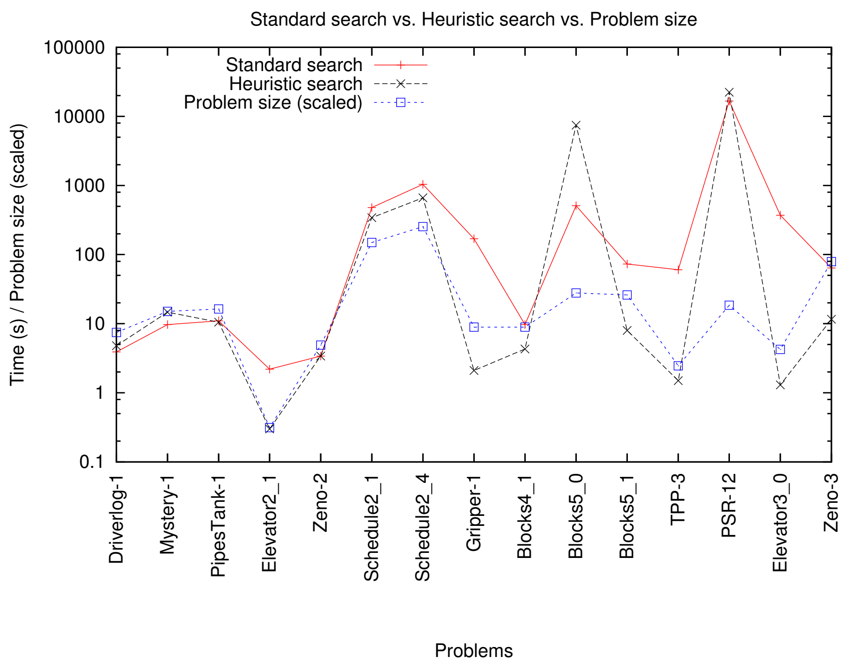


Figure 4.16: Comparison of standard search (runtime), heuristically guided search (runtime), and problem size.

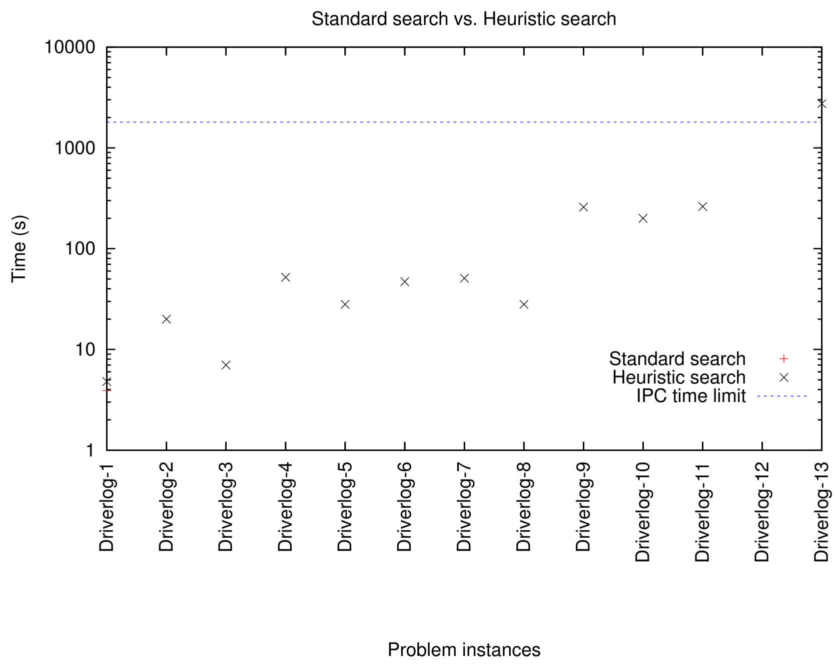


Figure 4.17: Comparison of standard search with heuristically guided search (runtime) for Driverlog instances 1 to 13.

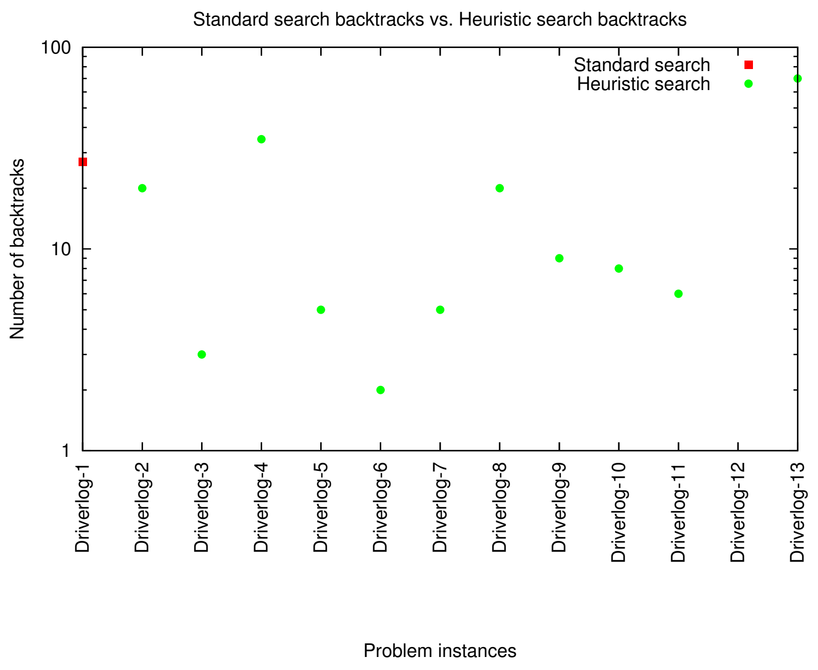


Figure 4.18: Comparison of standard search with heuristically guided search (number of backtracks) for Driverlog instances 1 to 13. Where no value is given, no solution was found (except instance 1 (heuristic) = BT free.)

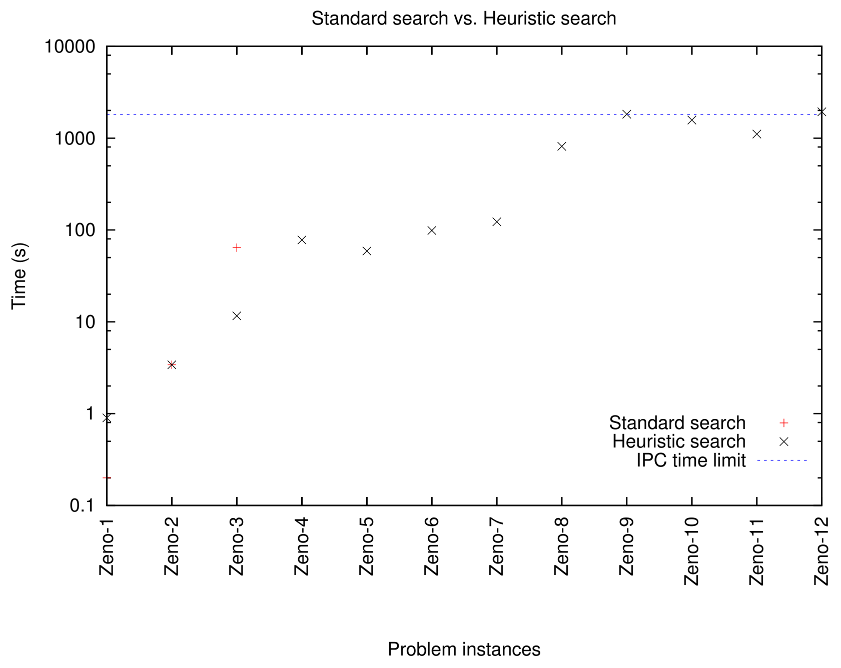


Figure 4.19: Comparison of standard search with heuristically guided search (runtime) for Zeno instances 1 to 12.

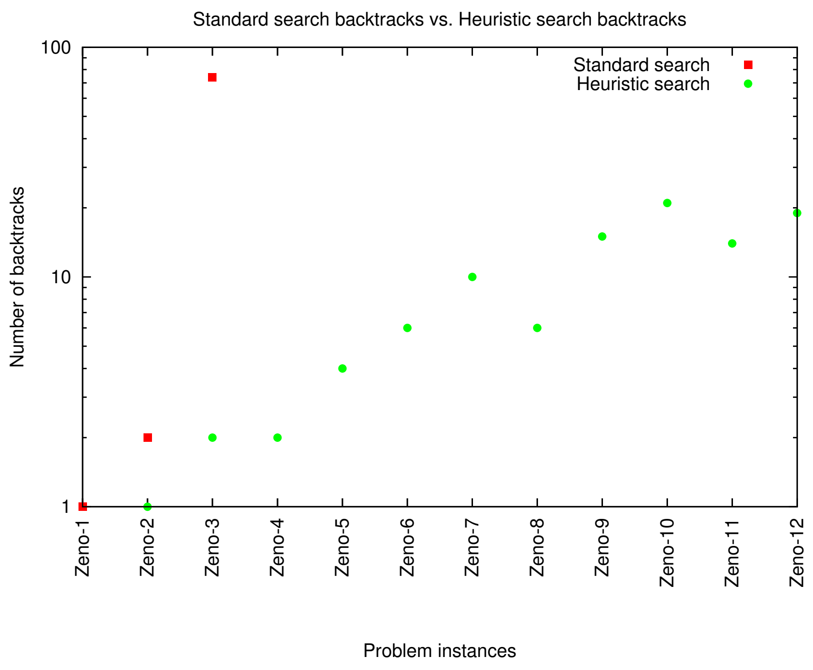


Figure 4.20: Comparison of standard search with heuristically guided search (number of backtracks) for Zeno instances 1 to 12. Where no value for the standard search is given, no solution was found.

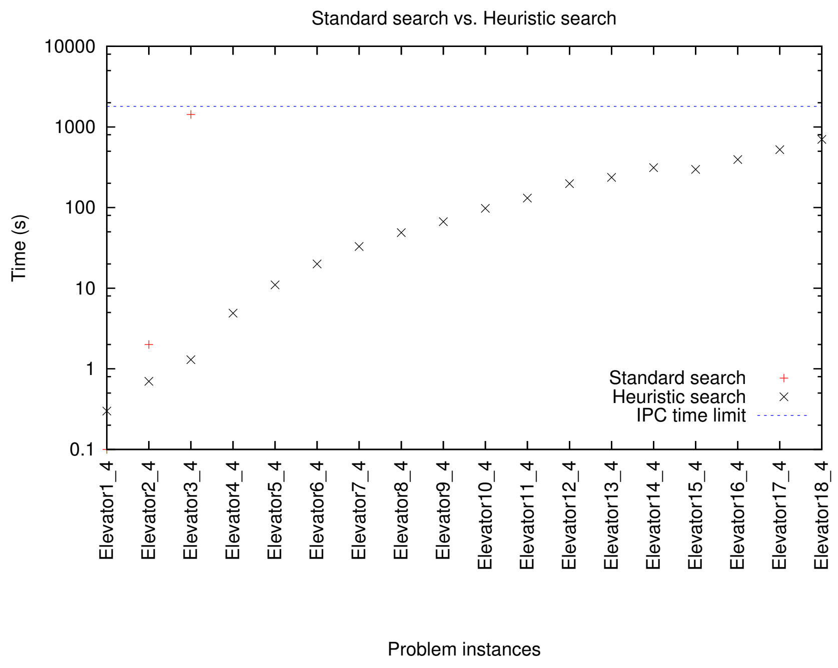


Figure 4.21: Comparison of standard search with heuristically guided search (runtime) for Elevator instances 1 to 18.

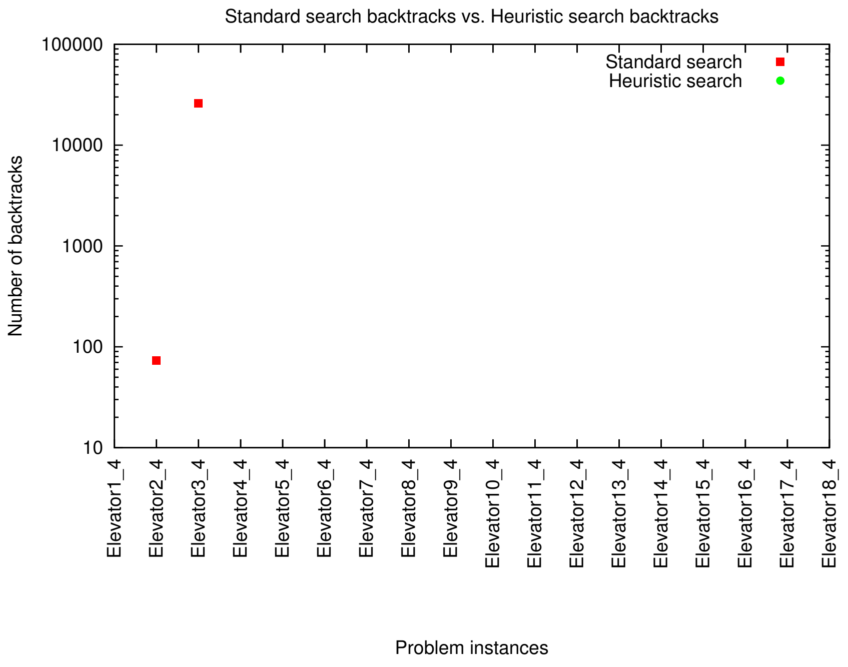


Figure 4.22: Comparison of standard and heuristic search (backtracks) for Elevators 1 to 18. No value for standard search means no solution (except 1_4 = BT free). No value for heuristic search means backtrack-free solution.

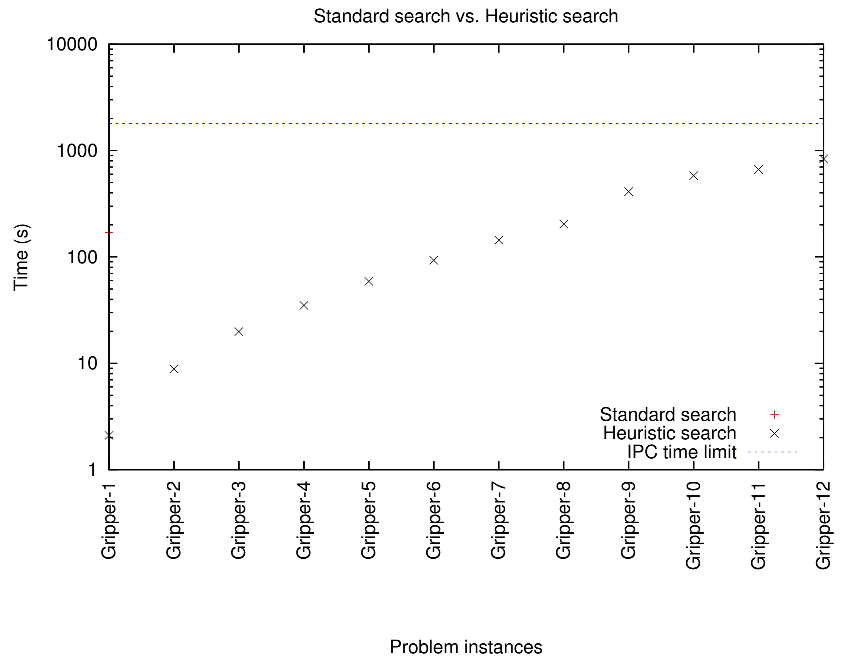


Figure 4.23: Comparison of standard search with heuristically guided search (runtime) for Gripper instances 1 to 12.

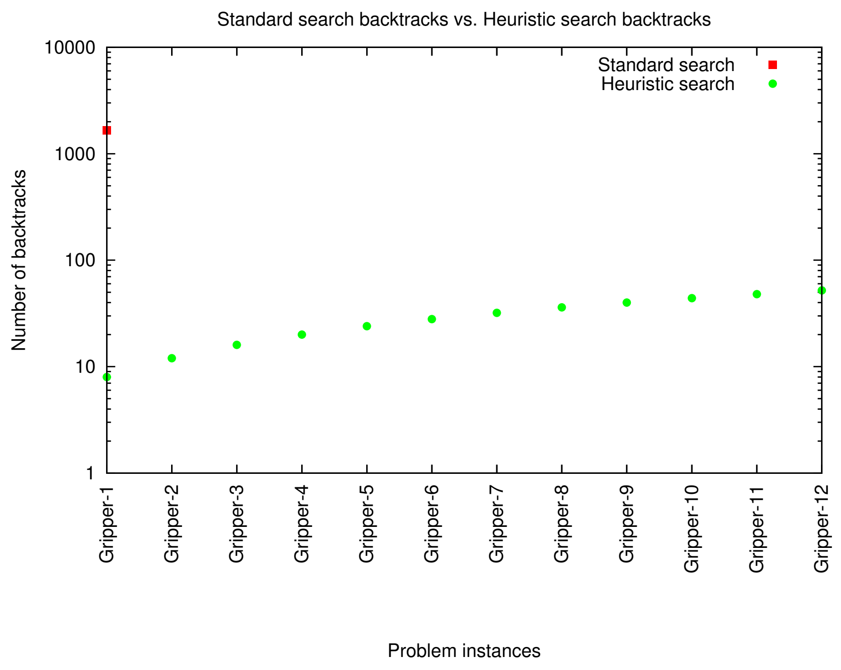


Figure 4.24: Comparison of standard search with heuristically guided search (number of backtracks) for Gripper instances 1 to 12. Where no value for the standard search is given, no solution was found.

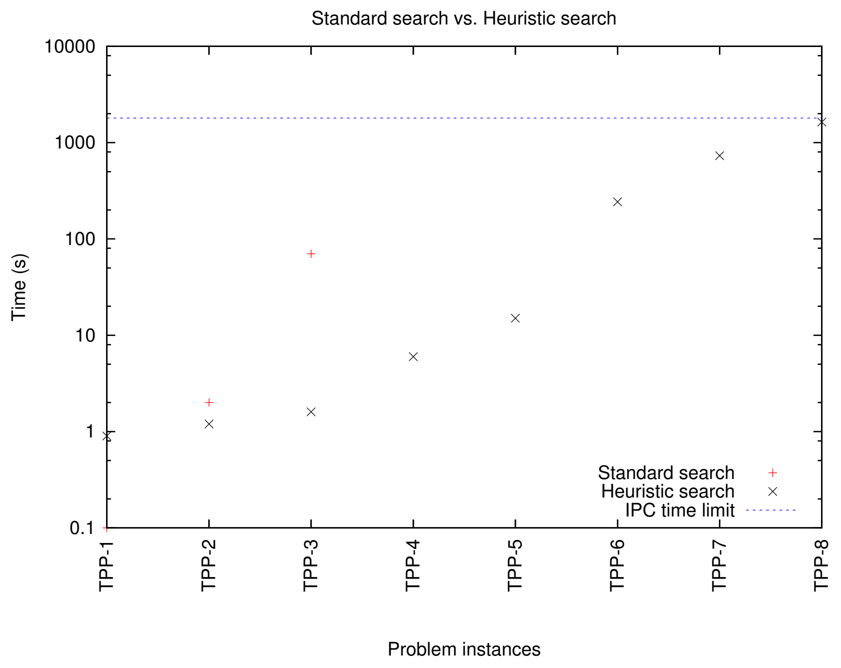


Figure 4.25: Comparison of standard search with heuristically guided search (runtime) for TPP instances 1 to 8.

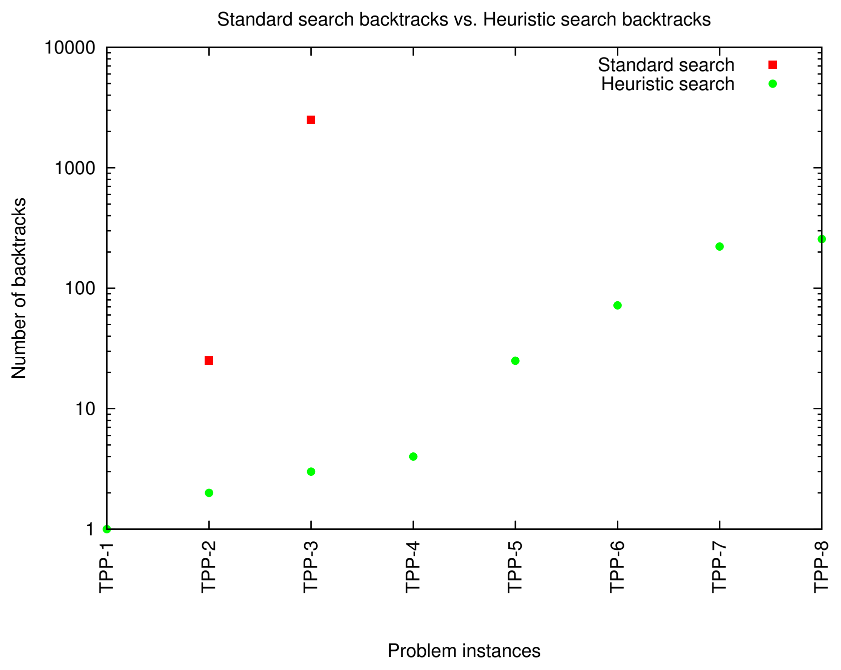


Figure 4.26: Comparison of standard search with heuristically guided search (number of backtracks) for TPP instances 1 to 8. Where no value is given, no solution was found (except instance 1 (standard) = BT free).

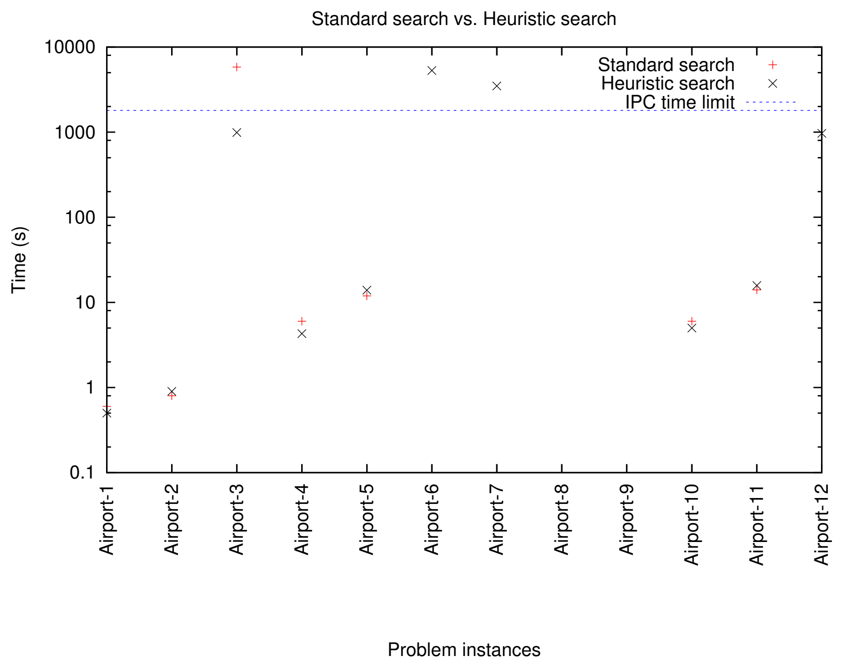


Figure 4.27: Comparison of standard search with heuristically guided search (runtime) for Airport instances 1 to 12.

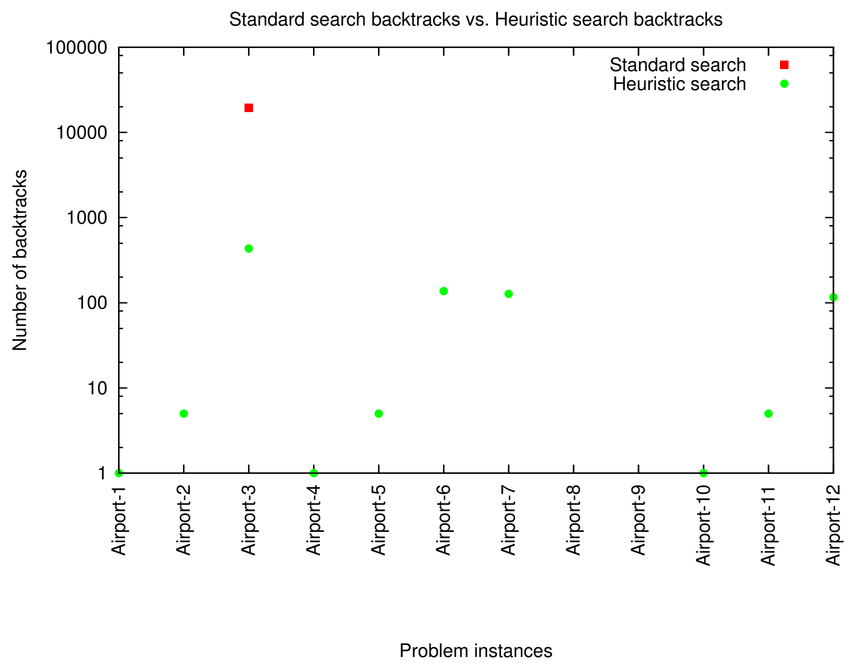


Figure 4.28: Comparison of standard search with heuristically guided search (number of backtracks) for Airport instances 1 to 12. Where no value is given, no solution was found (except instances 1,2,4,5,10 & 11 (std) = BT free).

Figures 4.17 to 4.28 show the results of running both the heuristic and the standard search on further instances from the domains used in the test set[1]. In the Driverlog domain (Figure 4.17), the heuristic solves all but one of the first thirteen instances, failing on instance 12. The standard search solves only the first instance within the time limit[2]. Clearly, the heuristic is able to solve many more instances than the standard search approach; as the problem size increases, propagation of the constrained values from the initial and goal states has less impact, and therefore the standard search has more "space" in which to assign incorrect values, resulting in thrashing behaviour and a consequent failure to find solutions within the time limit. In contrast, the heuristic, by breaking up this larger search space, allows the solver to take advantage of the new sub-plan deadlines, capitalising on the associated propagation of values in order to provide solutions with a low number of backtracks (Figure 4.18). Considering also Figure 4.29, the runtime increases steadily as the instance size increases. Deviations from this, for example Driverlog_4, are a consequence of plan repairs being necessary upon completion of the heuristic search, not due to a deficiency in the goal ordering, but

[1]No further Blocksworld instances are given due to the poor performance on this domain.

[2]Although the IPC time limit is indicated, a further period (almost 3 hours in total) was allowed to determine whether or not a solution was found.

because later (correctly ordered) goals clobber previous ones[1] (e.g. a sub-plan satisfying a *driver* goal makes use of a previously satisfied *truck* goal variable). Driverlog_12 also suffers from this problem, exacerbated by an increased number of locations and goals, which results in no solution being found within the time limit.

The heuristic performs well in the Zeno domain (Figures 4.19 and 4.20), solving instances 1 through 12, compared to the standard search which solves only instances 1 to 3. While the former incurs a limited amount of backtracking throughout the set of instances, the latter has a sharply increasing rate over the few problems solved. There are no additional actions (repairs) in any of the heuristically derived solution plans, with increasing runtime directly related to increased problem size (Figure 4.30).

Elevator instances 1_4 through 18_4 solve in a backtrack-free manner with the heuristic (Figure 4.22), whereas, again, the standard search exhibits a steep rise in the amount of backtracking as the problem size is increased. In terms of runtime (Figures 4.21 and 4.31), the heuristic closely follows the problem size, requiring no plan repairs due to clobbering, with the standard search runtime growing exponentially in the problem size.

Figures 4.23 and 4.24 show, respectively, the runtime and backtrack count for both the heuristic and the standard search techniques on problems from the Gripper domain. The standard approach solves only the first instance within the time limit, with a large amount of backtracking, whereas the heuristic finds solutions to problems 1 through 12, with a steadily rising (low) number of backtracks. The increasing number of backtracks is directly related to the increasing number of goals, with each problem instance exhibiting similar characteristics to those described above for Gripper1. Runtime grows in line with increased problem size, as shown in Figure 4.32, reflecting the increased number of constraints and associated consistency checking required.

The results of testing on problem instances from the TPP domain are shown in Figures 4.25 and 4.26. Here, again, the amount of time required for the standard search to find a solution rises sharply as the problem size increases and, as a consequence, this approach solves only the first three instances within the time limit. The number of backtracks follows a similar pattern, increasing rapidly with increased problem size. In contrast, the heuristic search runtime measure rises steadily with the increase in the size of problem in this domain's test set (Figures 4.25 and 4.33). The number of backtracks, whilst initially low, increases with an increase in problem size, although this number remains very low relative to that for the standard search.

The final set of problems considered here are those in the Airport domain [216].

[1]This is addressed in the following chapter.

This more recent IPC domain was chosen since it, in common with the previous IPC domains, does not make use of the *PDDL requirement* `:action-cost`. The Airport domain models the layout of an airport runway network, with incoming and outgoing aircraft. An important factor here is that a given aeroplane does not impede or endanger the progress of any other. Thus, an aeroplane with running engines effectively locks the runway segments leading to the segment it occupies. The domain is modelled using ADL (Section 2.2.2), with the STRIPS (Section 2.2.1) versions gained by grounding many of the operator parameters. There are a large number of variables in the CSP encodings, leading to large problem matrices and a high number of constraints, each contributing to increased problem "size" (Figure 4.34). The resulting causal graph for a given instance contains many interconnections, and hence the removal of cycles again leads to less accurate goal ordering information.

The runtime and backtrack performance measures, for both the standard search and the heuristic search on Airport instances 1 to 12, are given in Figures 4.27 and 4.28, respectively. The standard search approach finds a solution faster, and with fewer backtracks, than the heuristic in instances 2, 5, and 11. The heuristic is faster on problem instances 1, 3, 4, and 10, and additionally solves instances 6, 7, and 12, whereas the standard search does not. The heuristic backtracks on the first instance due to the placement of a suggested action at an inappropriate slot (propagation of this value subsequently leads to the single backtrack). Despite this, it finds a solution faster than the standard search. On the second instance, the heuristic suffers from the same problem, but with a larger number of actions and constraints, backtracks more, requiring a longer run time than the standard approach. Instances 4 and 10, and 5 and 11, exhibit similar characteristics to instances 1 and 2, respectively. With a larger problem size and increased number of goals, the standard search requires a considerable amount of time to find a solution to problem instance 3, due to a significant amount of backtracking. The performance of the heuristic approach is better, although this suffers as a result of shortcomings in the determination of variable dependencies (CG). Since this problem requires two aeroplanes' positions to be swapped, either ordering of the aeroplane goals means an interleaving of actions that helps facilitate the achievement of both. Therefore, the factor affecting the run-time of the heuristic on this problem is not only the order of the goals, but also the consequent manner in which the solutions of these (sub plan lengths) divide the solution space. The automated ordering achieves a solution in 991 seconds (433 b/t) (Figures 4.27 and 4.28), whereas a manually chosen ordering leads to a solution in 61.8 seconds (55 b/t)[1].

[1]Supports the thesis since manually using problem structure still leads to better solver results.

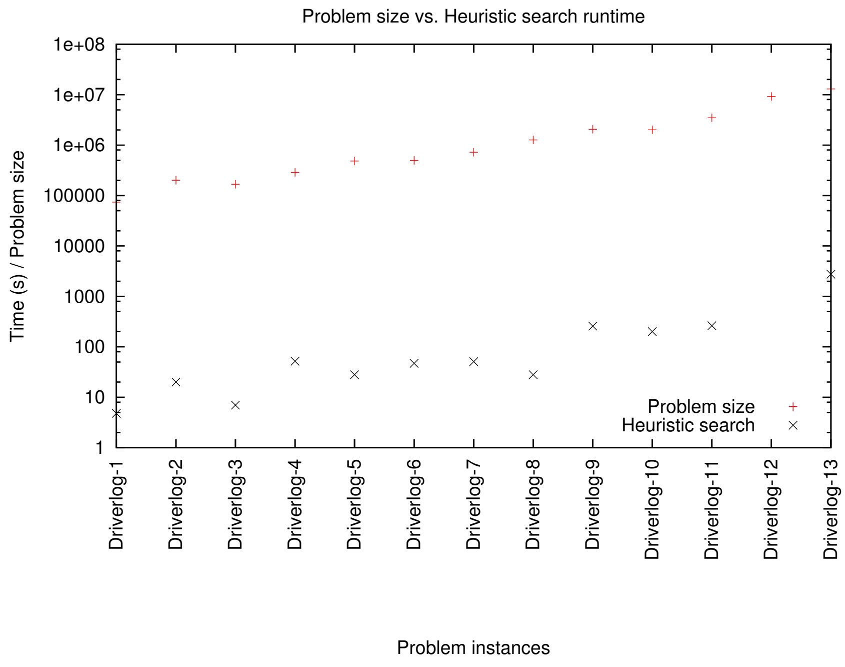


Figure 4.29: Comparison of heuristic (runtime) and problem size for Driverlog instances 1 to 13. Where no value is given, no solution was found.

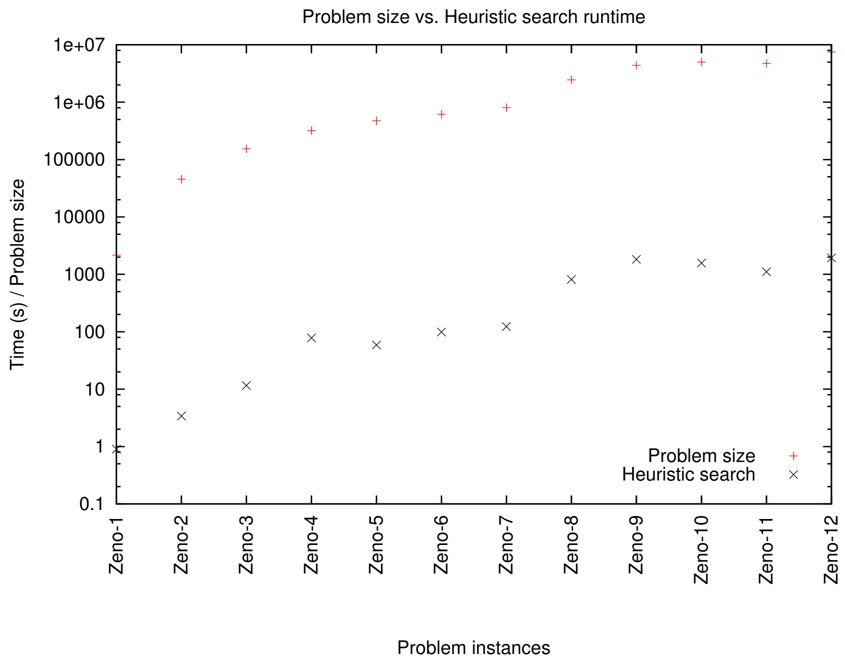


Figure 4.30: Comparison of heuristic (runtime) and problem size for Zeno instances 1 to 12.

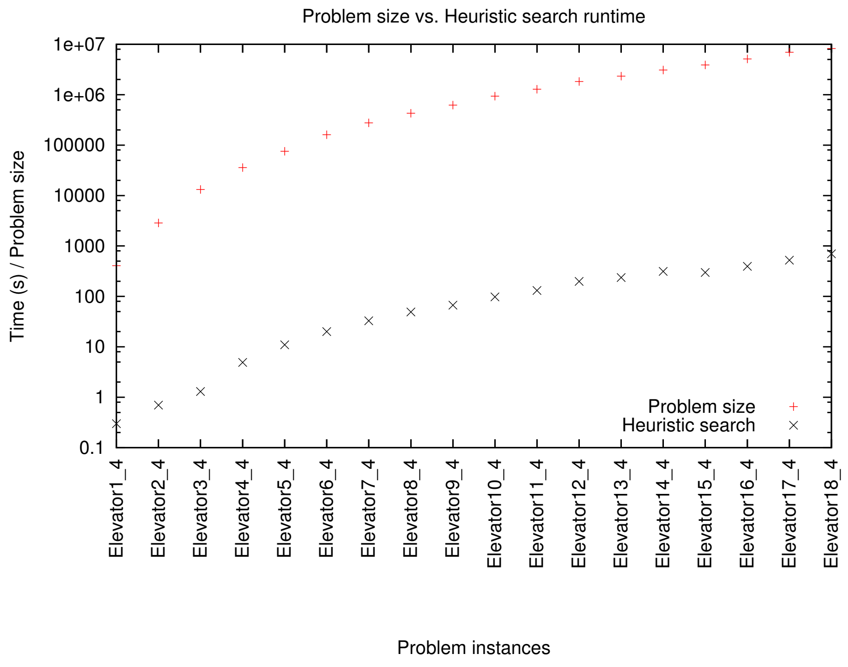


Figure 4.31: Comparison of heuristic (runtime) and problem size for Elevator instances 1 to 18.

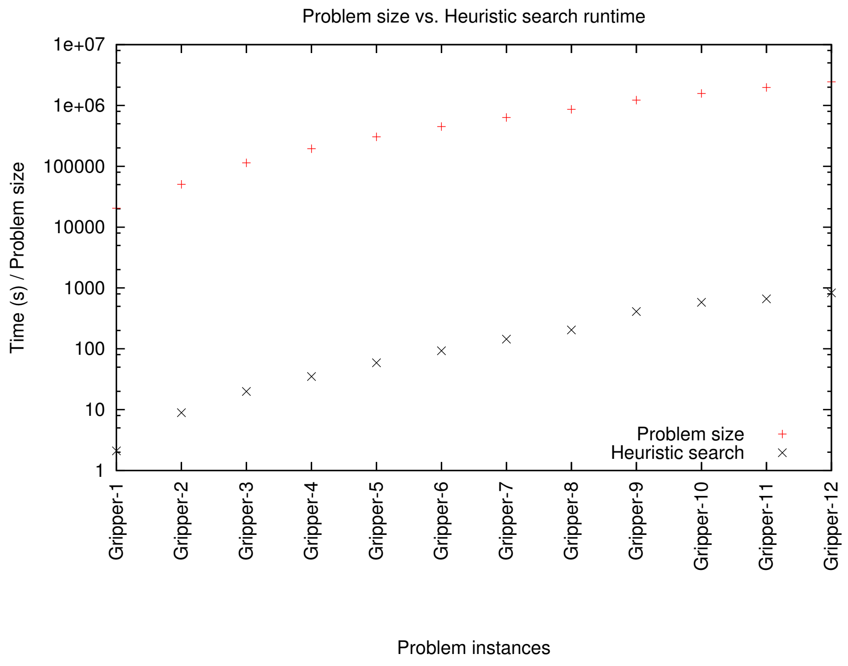


Figure 4.32: Comparison of heuristic (runtime) and problem size for Gripper instances 1 to 12.

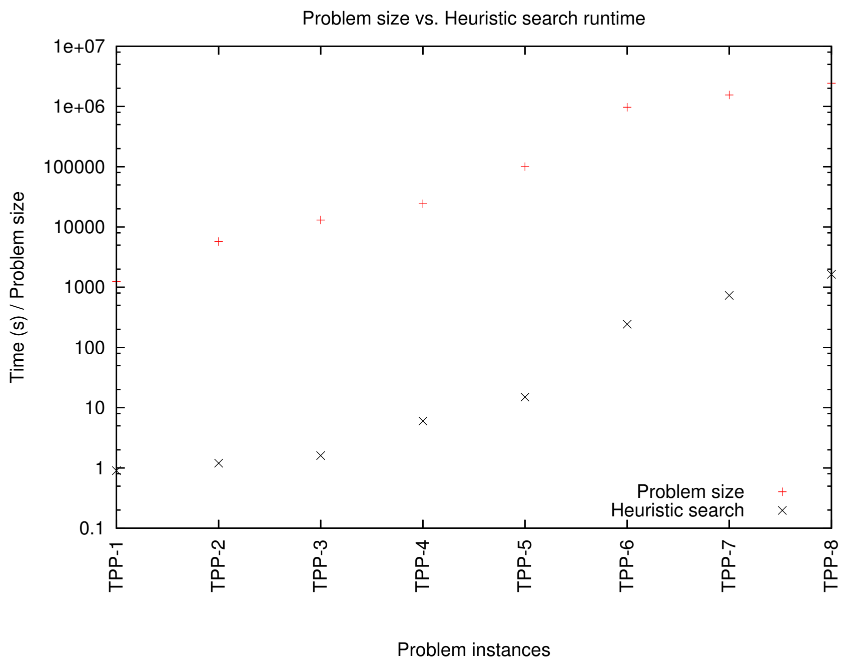


Figure 4.33: Comparison of heuristic (runtime) and problem size for TPP instances 1 to 8.

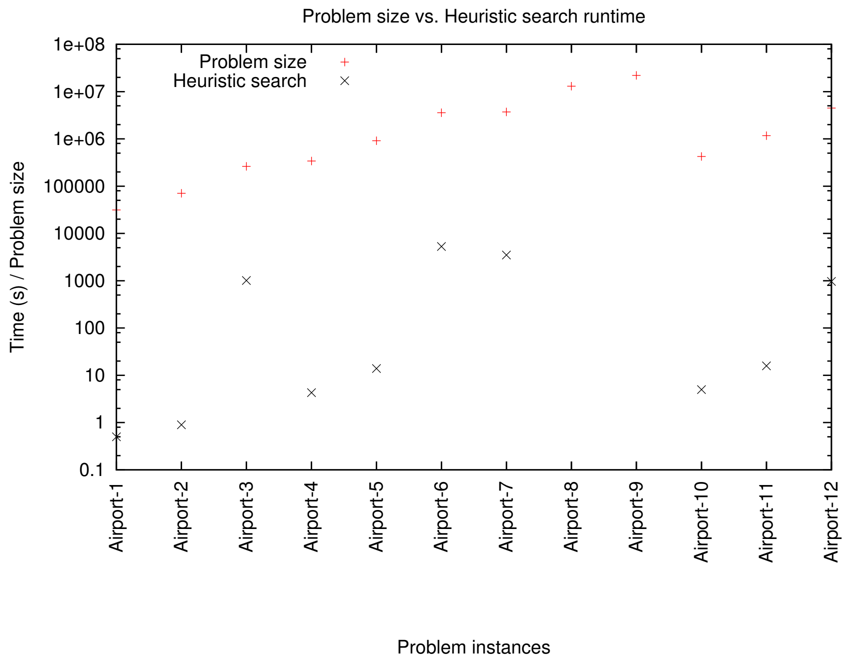


Figure 4.34: Comparison of heuristic (runtime) and problem size for Airport instances 1 to 12. Where no value for heuristic runtime is given, no solution was found.

Summarising, the above experiments sought to determine the runtime and amount of backtracking required to find solutions to further instances from the test set problem domains, using two CSP solution methods. The performance of the variable and value selection heuristic was compared[1] to that of the standard approach. The former used information from the original planning problem in order to guide the search, whereas the latter used generic CSP guidance in the selection of variables and values.

The results presented in Figures 4.17 to 4.28 confirm the findings from the initial test set measurements. That is, given good goal ordering information, the heuristic performs significantly better than the standard approach, as measured by runtime and amount of backtracking required to find a solution. Whereas both the runtime and backtrack count for the standard approach rise sharply after the first few instances in each domain, the heuristic is able to solve many more problems, with each measure rising steadily with the increase in problem size (Figures 4.29 to 4.34). There is a consistent size limitation across the domains, with instances failing to compile when the CSP encoding reaches the upper limit. Also, using this problem size measure as a predictor of relative search effort is useful, and is shown to be accurate across domains, with the few exceptions being related to the goal ordering deficiencies referred to above. This is expected since the coarse size measure used does not take account of CSP problem structure (i.e. *tightness* of constraints), only problem "size" (i.e. dominated by quantity of constraints).

4.5 Chapter Summary

This chapter presented the results of a series of empirical measurements designed to test the efficacy of a new CSP heuristic for the solution of AI Planning problems. These experiments set out to show that it was possible, using information from the original planning problem, to guide the variable and value selection procedure in a CSP encoding of such a problem.

Building on the work discussed in Chapter 3, the automated procedure for determining the goal ordering was presented. Utilising the causal graph structure, it was shown how planning problems may be subdivided, with the information gained (in the form of an ordered series of fixed variable values) being used to provide known values for placement in the CSP solution matrix. An algorithm to place these variables was described, with the effect being the introduction of a series of intermediate horizons.

[1]Where comparison was possible.

Propagation through the solution matrix from these new deadlines allowed the solution process to benefit from one of the sources of leverage in planning. Namely, bounding the length of the sub-plan that supports the action chosen to achieve the newly introduced variable value. That is, the better use of the inherent inference within the CSP encoding, resulting from the introduction of the goal-ordered variables, serves to reduce the domains of the preceding variables in the sub-plan. In this way, the overall planning problem is solved using structural information *implicit in the given planning problem* in combination with *targeted best use of the characteristics of the constraint programming paradigm*. Thus, the planning problem structure is no longer lost in the CSP reformulation, but instead is integrated into the CSP solution process.

Considering the results presented in this chapter in terms of the thesis being investigated (Section 1.3), improved solution times and reduced amounts of backtracking over a range of problem domains show the efficacy of the heuristic guidance procedure[1]. However, there remain a number of areas in which improvements may be made.

One of the limitations highlighted above concerned the clobbering of previously achieved goals by an action required for the achievement of a subsequent goal. In order to prevent goals from being undone later in the plan solution procedure some form of goal "locking" could be used. Setting up a system of meta-variables to represent goal-achieving actions within the CSP encoded planning problem could achieve this. This should also lead to additional inferences being made in the following solution space since a column of fixed values, representing the locked goal, would necessarily reduce the domains of the other CSP state variables, thereby pruning the number of available actions, consequently reducing the overall search space.

Another area in which improvement may be made is in the provision of suggested actions. The goal-directed CSP heuristic considers the suggested actions in the order generated by the automated CSP encoding procedure. For example, in the Driverlog domain, when a package delivery goal is to be achieved, the heuristic considers all possible truck-package-unload combinations. With a predetermined resource allocation strategy, using a meta-CSP to represent the resource to task assignment process, it should be possible to reduce the number of choices available, thereby potentially reducing the time taken to find a solution, the amount of backtracking required or the plan "cost", as measured by number of actions present in the final plan.

The following chapter introduces meta-CSP methods, and discusses how such techniques may contribute to the thesis goal of achieving increased inference in the solution of constraint encoded AI planning problems.

[1]Although, as noted in Chapters 1 and 6, these do not yet match state of the art (non-CSP) planners.

CHAPTER 5

META VARIABLES FOR GOAL LOCKING AND RESOURCE ASSIGNMENT

5.1 Introduction

The previous chapter introduced a new goal-directed CSP variable and value selection heuristic for use on AI planning problems. It was shown that, on a range of IPC planning domains, the heuristic guidance allowed better performance and solution of a wider range of problem instances compared to that of a standard CSP solving approach. This was achieved by using the structural information from the planning problem to guide the solution procedure in the CSP reformulation. In doing so, it was shown that better use could be made of the inherent characteristics of the CSP paradigm for solving certain planning problems. This chapter continues this work, the theme of the thesis, by introducing the use of meta-CSP methods.

The use of such meta-CSP techniques is first discussed in relation to goal-locking (Section 5.2). That is, once the value of a goal variable has been achieved, this value can be fixed until the end of the plan. The thesis being investigated here is that, by locking the satisfied goal variable's value from the layer in which it is first achieved until the end of the plan, there will be a consequent reduction in the domain size of the other CSP state variables (connected to the goal variable by constraints describing the problem's actions) at a given level. Inferences made and propagated as a result of the domain reductions may, in certain instances, lead to improved performance, as measured by run-time and amount of backtracking required to find a solution. The results of using this form of goal-locking on the test set(s)[1] are shown and discussed in Section 5.2.1.

The further use of meta-variables, to represent the resource to task allocation part of a planning problem, is described in Section 5.3. The idea here is to use a meta-CSP to facilitate resource assignment before attempting to solve the CSP. The intention is to test whether such constraints significantly reduce the amount of choice in the problem, in turn reduce the search space, and lead to the realisation of further benefits, gained from the inherent CSP inference mechanisms. The results of adding meta-variable based resource assignment to the heuristic search technique are detailed in Section 5.3.3.

5.2 Goal Locking using CSP Meta Variables

This section first introduces the use of meta-variables in constraint satisfaction problems. It then discusses their use for locking recently achieved goals within the CSP encoded planning problem.

The solution to a CSP (Section 2.3) requires that a value is found for each of the CSP variables, and that such values are taken from the respective domains of those variables. Additionally, the set of values assigned must simultaneously satisfy all of the declared constraints. It is possible, however, that certain values for a given variable are *interchangeable* [217] with no impact on the solution(s).

Definition 40 *A value, b, for a CSP variable, V, is* fully interchangeable *with a value, c, for V iff:*

1. *Every solution to the CSP which contains b remains a solution when c is substituted for b,* and
2. *Every solution to the CSP which contains c remains a solution when b is substituted for c.*

Where it is possible to eliminate interchangeable values in CSPs [218], a clustering of subsets of the original CSP variables may be achieved, with a set of *meta-variables* [219] created.

Definition 41 *A* meta-*CSP of a ground CSP,* X, *consists of variables that correspond to subsets of the variables in* X. *The values of the meta-variables are the solutions of*

[1]Links to all test domains can be found in Appendix A.

the problems induced by the subsets of variables. The constraints between the meta-variables are satisfied when all of the constraints from the original CSP are satisfied.

Applying this concept to a CSP encoded planning problem [220] allows a meta-CSP to be constructed, the variables of which represent the goals of the original planning problem.

Definition 42 *For a planning problem, P = (O,I,G), the meta-CSP is a CSP where* $|X|$ $= |G|$ *and* $D_i = \{\ a \mid g_i \in$ *add(a)*$\}$.

Each variable represents a goal, $g_i \in G$, with the associated domain containing only the actions that achieve g_i. Such actions have g_i in the add list of their effects. With a variable assigned a value in this representation, $\langle x_i, a \rangle$, the meaning is that a is the *final achiever* of g_i. This means that it is not possible for any action appearing later in the plan than a to delete g_i.

With these additional meta-variables added, the heuristic goal-driven procedure becomes that described by Algorithm 2. The effect of the additional constraints is that, once a goal-state goal variable's value is achieved, it is locked until the end of the plan. This is shown in the partial solution matrix below (Figure 5.1) for the example Driverlog problem instance discussed previously (pages 112-114).

Referring to Figure 5.1, the heuristic procedure has suggested an action to satisfy goal variable three (column 3, bottom row). This action, Action 68, has been placed in action slot five, and actions for its supporting sub-plan have been found and placed appropriately. With the consistency of the constraints checked, the value of the variables representing this state variable through the remainder of the matrix are forced to maintain the desired goal-state value, as shown in column 3, with each row set to the goal value of zero.

Comparing Figure 5.1 with the equivalent solution matrix showing the heuristic procedure's first sub-plan without the meta-variable goal locking applied (Figure 5.2), it is clear that, in addition to the maintenance of the goal value, there is a consequent reduction in the domains of the action variables Ac6 through Ac22 due to propagation of the locked goal variable's value.

Depending on the structure of a particular problem instance, the action pruning resulting from the increased inference may lead to a reduction in the time taken, and the amount of backtracking required, to find a solution to the given problem. However, whilst such propagation may aid faster solution, the increased number of constraints

```
       3           5          2     . . .   Ac1:    1: boardtruckdriver1truck1s0
       3           0          2     . . .   Ac2:   22: drivetrucktruck1s0s1driver1
       3           0          2     . . .   Ac3:   44: loadtruckpackage2truck1s1
       3           0          4     . . .   Ac4:   51: drivetrucktruck1s1s2driver1
       3           0          4     . . .   Ac5:   68: unloadtruckpackage2truck1s2
       3           0          0     . . .   Ac6:   _{[4,6,8,35,38,40,41,109]}
  _{[2,3,7]},  _{[0,3]},      0     . . .   Ac7:   _{[2,4,6,8,9,12,15,16,19..22,31,...]}
   _{0,4,7},   _{0,2..6},     0     . . .   Ac8:   _{[1,2,4..10,12,15..43,45..53,55,56,...]}
   _{0..7},    _{0..7},       0     . . .   Ac9:   _{[1,2,4..10,12..43,45..66,70..75,78,79,...]}
   _{0..7},    _{0..7},       0     . . .  Ac10:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac11:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac12:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac13:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac14:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac15:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac16:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac17:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac18:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac19:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac20:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac21:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac22:   {[1,2,4..10,12..43,45..66,70..75,78..80,...]}
   _{0..7},    _{0..7},       0     . . .  Ac23:   _{[1,2,4..10,13..34,...]}
   _{0..7},    _{0..7},       0     . . .  Ac24:   _{[5..8,13..21,23,26,...]}
   _{0..7},    _{0..7},       0     . . .  Ac25:   _{[5,6,13..20,26,27,...]}
  _{[1..4,6]},  _{1..5},      0     . . .  Ac26:   _{[5,6,14,16,26,27,...]}
  _{[2,4,6]},  _{[1,2,4]},    0     . . .  Ac27:   _{[14,16,56,59,68,...]}
       4           4          0     . . .
```

Figure 5.1: Partial solution matrix with goal locking applied.

```
       3           5          2          . . .   Ac1:    1: boardtruckdriver1truck1s0
       3           0          2          . . .   Ac2:   22: drivetrucktruck1s0s1driver1
       3           0          2          . . .   Ac3:   44: loadtruckpackage2truck1s1
       3           0          4          . . .   Ac4:   51: drivetrucktruck1s1s2driver1
       3           0          4          . . .   Ac5:   68: unloadtruckpackage2truck1s2
       3           0          0          . . .   Ac6:   _{[4,6,8,35,38,40,41,77,109]}
  _{[2,3,7]},  _{[0,3]},   _{[0,4]},     . . .   Ac7:   _{[2,4,6,8,9,12,15,16,19..22,31,...]}
   _{0,4,7},   _{0,2..6},  _{0..2,4},    . . .   Ac8:   _{[1..10,12,15..53,55,56,64..77,109]}
   _{0..7},    _{0..7},    _{0..4},      . . .   Ac9:   _{[1..79,106,109]}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac10:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac11:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac12:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac13:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac14:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac15:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac16:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},        .    Ac17:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac18:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac19:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac20:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac21:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac22:   _{1..109}
   _{0..7},    _{0..7},    _{0..4},      . . .  Ac23:   _{[1,2,4..10,13..34,36..44,46..64,...]}
   _{0..7},    _{0..7},    _{[0,4]},     . . .  Ac24:   _{[5..8,13..21,23,26,27,29..34,...]}
   _{0..7},    _{0..7},    _{[0,4]},     . . .  Ac25:   _{[5,6,13..20,26,27,29,30,32,34,...]}
  _{[1..4,6]},  _{1..5},   _{[0,4]},     . . .  Ac26:   _{[5,6,14,16,26,27,29,30,32,34,56,...]}
  _{[2,4,6]},  _{[1,2,4]}, _{[0,4]},     . . .  Ac27:   _{[14,16,56,59,68,84,107,109]}
       4           4          0          . . .
```

Figure 5.2: Partial solution matrix with no meta-variable goal locking.

Algorithm 2: solveMatrixWithLocking(uplim,lowlim,smax)

```
Input: uplim (Upper limit), lowlim (Lower limit), smax (Solution matrix)
Output: smax (The solution matrix)
goalset ⟵ smax[uplim,(1..numofvars)]
for $s \leftarrow uplim$ to $lowlim$ do
    $q \longleftarrow (s - 1)$
    foreach goalvar in goalset do
        if action in layer q then
            if goalvar supported by action in layer q then
                if goalvar is a g-state goal then
                    lockGoalvar(s,g-state)
            else
                if goalvar supported by any row above then
                    if goalvar is a g-state goal then
                        lockGoalvar(s,g-state)
                else
                    $alloc \longleftarrow 0$
                    $acset \longleftarrow getSugActs(goalvar)$
                    foreach $act \in acset$ do
                        findSlotForAction(act,alloc,level)
                        if $alloc == 1$ then
                            $lolev \longleftarrow (level - 1)$
                            solveMatrixWithLocking(level,lolev,smax)
                            break;
        else
            if goalvar supported by any row above then
                if goalvar is a g-state goal then
                    lockGoalvar(s,g-state)
            else
                $alloc \longleftarrow 0$
                $acset \longleftarrow getSugActs(goalvar)$
                foreach $act \in acset$ do
                    findSlotForAction(act,alloc,level)
                    if $alloc == 1$ then
                        $lolev \longleftarrow (level - 1)$
                        solveMatrixWithLocking(level,lolev,smax)
                        break;
```

and associated processing may have a concomitant, and perhaps not insignificant, overhead. Further, the overall impact will depend on the causal dependencies between as yet unassigned CSP variables and those recently locked.

This section has presented a meta-variable based method for locking a goal variable's value once this has been achieved using the previously described CSP variable and value selection heuristic. In the example described, a reduction was seen in the domains of the action variables in the solution matrix, thus indicating that propagation of the goal variable's locked value led to an increased amount of inference.

The following section discusses the results gained when this approach was applied to the test sets used in previous chapters.

5.2.1 Results

Applying the above meta-variable based goal locking technique to the initial test set[1] gives the results shown in Figure 5.3. Generally, with the application of goal locking there is little or no reduction in the run time on this set of, relatively small, problem instances. Also, in the cases where a solution was found, there was no decrease in the amount of backtracking required.

Looking more closely at the results of Figure 5.3, the Driverlog1 instance requires the same amount of time to find a solution both with locking and without. This problem involves moving one truck and one driver from different starting points to the same destination. The goal directed algorithm achieves the truck goal, necessarily using the driver, with just a "disembark" action required to complete the plan. Locking the truck's position in this instance provides no real benefit since, even without locking, the system solves the problem with no backtracking, inferring that the driver is already in the truck at the destination location.

In general (recalling the causal graph structure shown, for example, in Figure 4.2), locking a (goal) variable upon which no future goal variable has a causal dependency, will provide little benefit. This is particularly true in relation to the goal oriented technique used in this work. That is, for example, for a problem instance in which there are package goals, truck goals and driver goals, locking a particular package variable's value (once the goal to deliver it is achieved) will have little impact on other goals, since nothing else depends on that package variable. However, locking the value of a truck or driver variable's value (once a goal is achieved) may have an impact on the remaining solution process, depending on the nature of the remaining goals.

[1]Links to all test domains can be found in Appendix A.

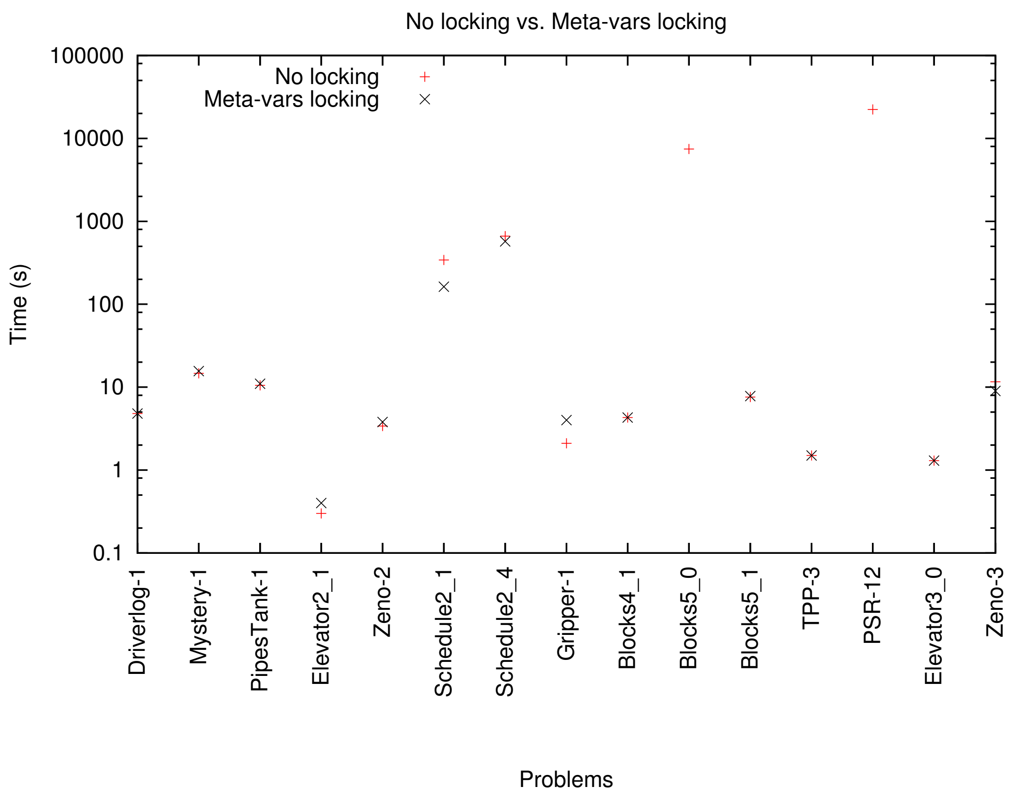


Figure 5.3: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking. Where no value is given, no solution was found.

The following four instances in Figure 5.3 exhibit a small increase in run time with the addition of meta-variable locking, with the number of backtracks remaining the same. The increase in run time is observed to be due to the additional constraints, and the associated extra time taken processing these, with the meta-variable processing being an overhead in these cases.

On both Schedule domain instances (Figure 5.3) there is a reduction in the run time when goal locking is applied. Each of these problems has a larger number of variables and a greater number of actions than the previous instances. The latter of which, in turn, leads to many more action constraints in the CSP encoding. When goal locking is applied, a significant reduction in the size of the domains of the action slot variables is seen. This pruning results in a 47% decrease in the time required to find a solution for Schedule2_1, and a 14% decrease in run time for Schedule2_4. Due to the structure of this problem domain, following locking there is also observed a reduction in the domain size of several of the CSP state variables (e.g. locking the first goal variable, representing an object having been painted blue, prunes future values of other variables, for example those representing that same object later having changed shape or temperature as a result of machining or rolling, since both such operations remove an object's painted surface). Consequently, the overall search space is much reduced, leading to faster run times for these two instances, with the number of backtracks remaining unchanged.

The Gripper_1 instance result given in Figure 5.3 shows that the solution run time increases by almost 50% with the addition of the goal locking constraints. Initially, this may appear counter intuitive, but on further consideration it is not. This is due to the structure of the Gripper domain, which consists of a robot with two gripper arms, each one able to carry a ball, with the goal being the relocation of a set of balls from one room to another. When one goal (a ball placed in the destination room) is locked, there is no advantage gained for the solution of a subsequent goal. That is, the goal-directed heuristic will focus on one single ball goal, achieve this, and then proceed to the next, with the locked, achieved goal having no dependency relationship with subsequent ones. Thus, the balls in the gripper domain are analogous with the parcels in the Driverlog domain, as shown in Figure 5.4, the Gripper_1 causal graph.

Blocks4_1 requires the same amount of time to find a solution with and without meta-variable goal locking applied. In this instance, with the first goal locked, the domains of the action variables are reduced by a small amount. This reduction, however, does not translate into a substantial reduction in the domain size of other CSP (state) variables. Further, as discussed in Chapter 4, the goal ordering information is less well defined in this domain due to the large number of arcs removed from the causal graph when cycles are broken. As a result of this, both approaches (lock / no-lock) require a "repair" procedure in this instance, which forces a backtrack over the first achieved goal, with this being satisfied at the end of the plan using standard search.

In the results discussed in the previous chapter (Table 4.2), Blocks5_0 was seen to solve using the heuristic procedure, but only after a considerable amount of time, and with double the number of actions of the standard search solution. This was due to the requirement in this instance for the interleaving of actions to satisfy subsequent goals being required within the sub-plan of a given goal ("Sussman" style problem). This led to an increase in the amount of backtracking, and run-time, required to find a solution. With the addition of goal locking, and the associated overhead of the extra constraint processing, this situation is exacerbated, with no solution found within the time limit.

Blocks5_1 requires the same amount of time to solve with and without goal locking, here again there is a reduction in the domain sizes of the action variables, and a consequent small reduction in the domains of some CSP (state) variables, but with this

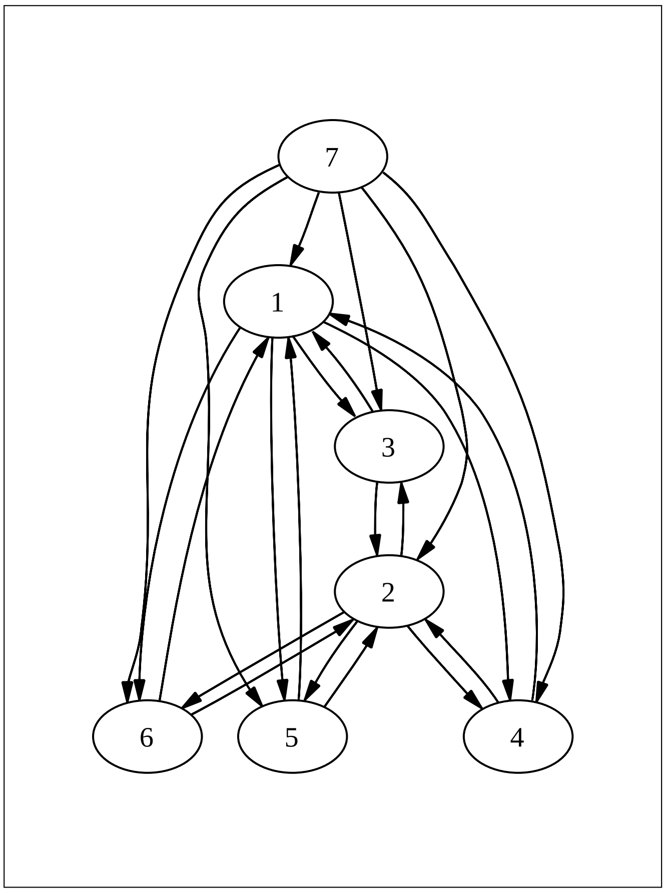


Figure 5.4: Gripper_1 causal graph. Variable 7 represents the robot's position, variables 1 and 2 the left and right arms, respectively, and variables 3 to 6 represent four balls.

leading to no appreciable reduction in run time in this instance.

Comparing locking and non-locking approaches to the solution of TPP3 (Figure 5.3) is interesting. The structure of this particular problem means that a "$STORED$ $(goods_x$, $level1)$" goal (goods bought and taken to the depot), once achieved, cannot be undone (in both the locking and non-locking formulations). This explains the near identical run-time measure and identical backtrack counts; there is nothing further to be gained, in terms of propagation, by locking the recently achieved goal, it is effectively already locked following achievement.

Applying the meta-variable goal locking during the solution of PSR12 prevents a solution being found within the time limit. Considering the findings discussed in Section 4.4, this is not surprising. The lack of accurate goal ordering information and the requirement for some of the goal-state goals to be toggled on and off before finally being set suggests that any procedure that locks these at the first point at which they are achieved will lead to increased backtracking and longer run times. The effect of this, combined with the requirement that the heuristically derived plan be repaired at the end of the procedure, is that PSR12 with meta variable goal locking fails to solve within the time limit.

Elevator3_0 takes 1.3 seconds to solve regardless of whether or not meta-variable goal locking is applied. In this domain there is no way to undo a goal (of the form: $SERVED\ (p_x)$) once it is achieved. Therefore, in a similar manner to the TPP domain, a goal predicate once set cannot be undone. In the CSP formulation this means the fixing of a variable's value, via the inherent inference mechanism, once that goal is achieved by the heuristically guided algorithm, with the additional meta-variable goal locking making no difference since this (latter) procedure merely achieves the same effect. With this is mind, it is possible that across a range of problem instances from this (and similar) domains, the meta variable locking may provide a slight improvement or make no difference on small instances, but actually prove to be an overhead on larger instances in which the additional constraint machinery increases the solution run time.

The result for the final instance in the test set (Figure 5.3) shows that goal locking provides a decrease in the time required to find a solution. This instance, Zeno3, benefits from good goal ordering information during the heuristic phase, derived from the causal graph (Figure 5.5).

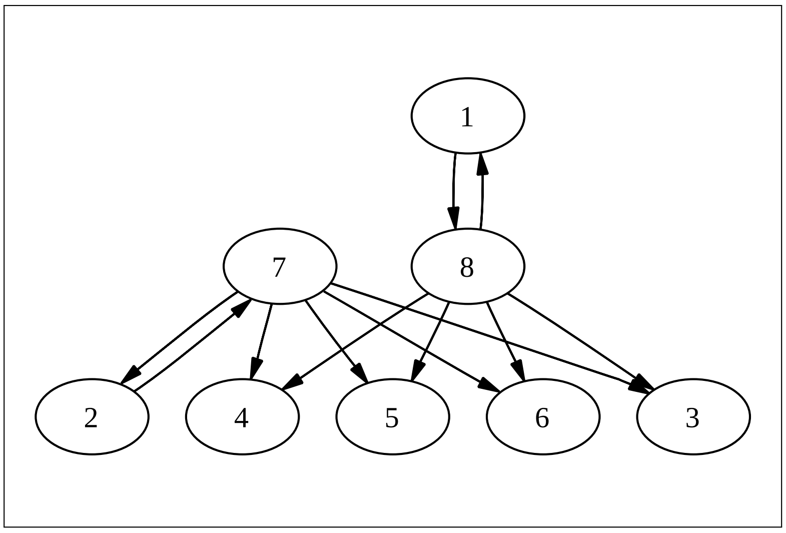


Figure 5.5: Zeno3 causal graph. Variables 1 and 2 represent the fuel level of the aeroplanes, variables 3 to 6 the passengers, and variables 7 and 8 the aeroplanes.

With meta-variable goal locking applied, all of the "people" goal variables are fixed throughout the solution matrix. The difference in this domain (instance) is that there are many goals specified in the goal-state which are already achieved in the initial-state. The combination of goal-locking and the good ordering allows these variables' values to be locked at the earliest possible point, with the result being three columns of fixed values. Another column of fixed values is added for the variable representing the second aeroplane, when it is encountered in the ordering (again, it already holds the desired goal value in the initial-state). The combined effect of this locking, in addition to that achieved as a result of the (heuristically solved) goal achievement locking, and the consequent propagation resulting from both, speeds up the solution process, leading to a 22% reduction in the solution time.

In summary, of the instances included in the initial test set, four of the fifteen exhibit no difference in run time when goal-locking is added, three require less time to find a solution, with the remainder showing an increased run time (including those with no solution). From the above discussion of these results, it is clear that the structure of a given problem has an impact on the efficacy of goal-locking. This is discussed further below, in relation to adding goal-locking constraints to the additional problem instances from the test set domains[1] used in the previous chapter (Figures 5.6 to 5.13).

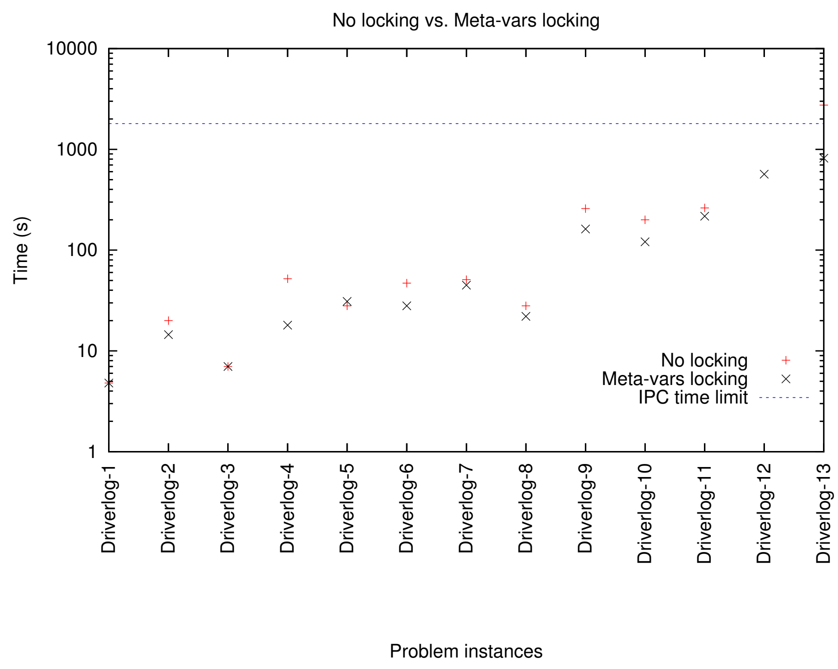


Figure 5.6: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for Driverlog instances 1 to 13.

[1]Links to all test domains can be found in Appendix A.

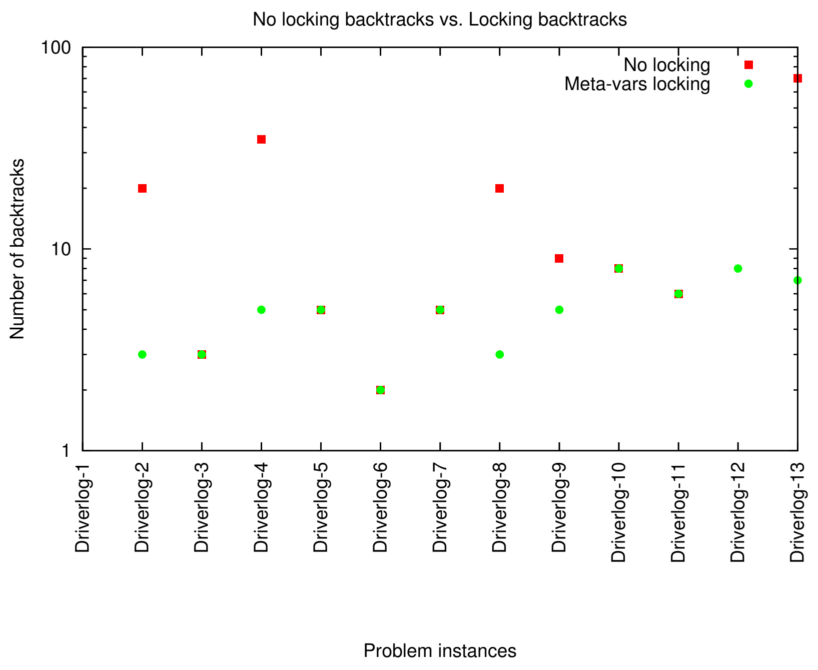


Figure 5.7: Comparison of the heuristically guided search (number of backtracks) with and without meta-variable goal locking for Driverlog instances 1 to 13.

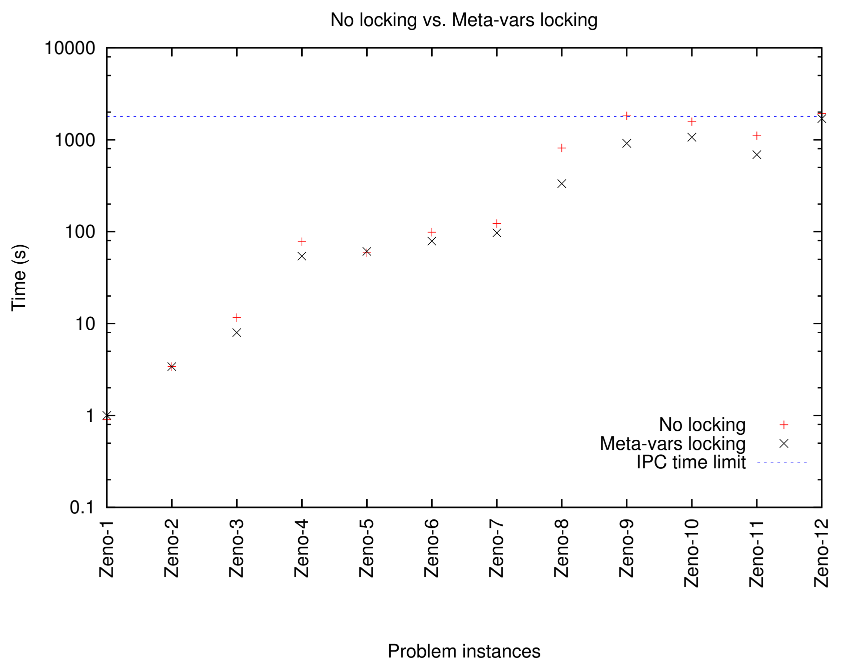


Figure 5.8: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for Zeno instances 1 to 12.

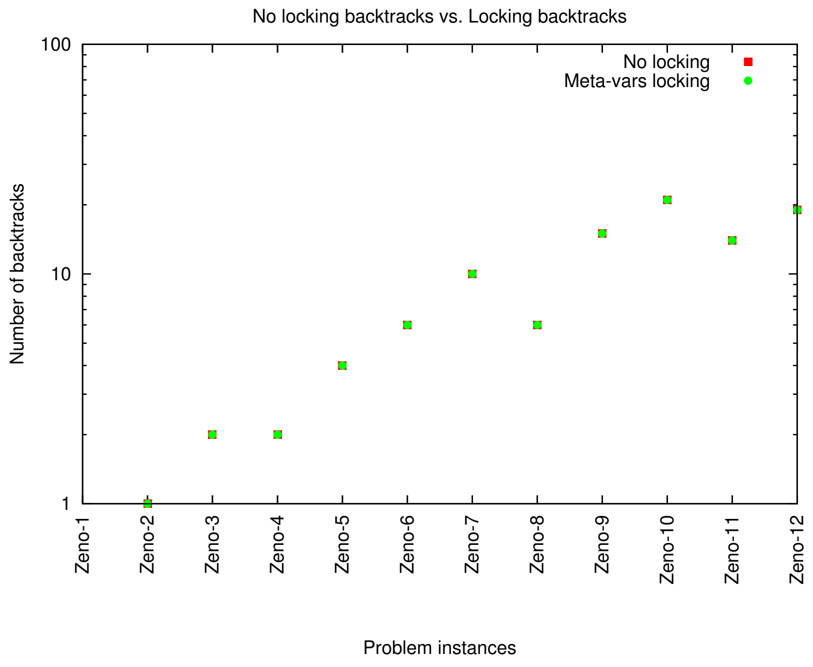


Figure 5.9: Comparison of the heuristically guided search (number of backtracks) with and without meta-variable goal locking for Zeno instances 1 to 12.

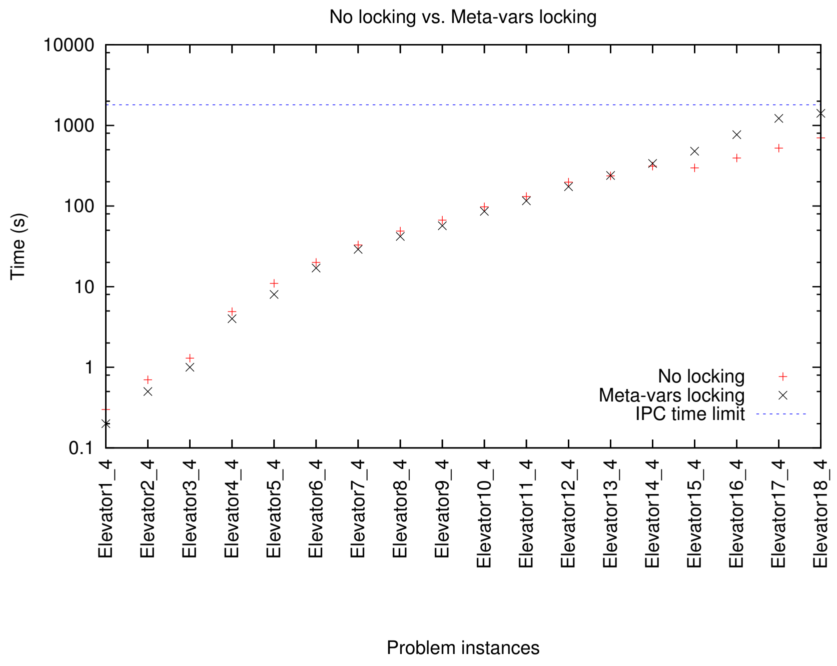


Figure 5.10: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for Elevator instances 1 to 18.

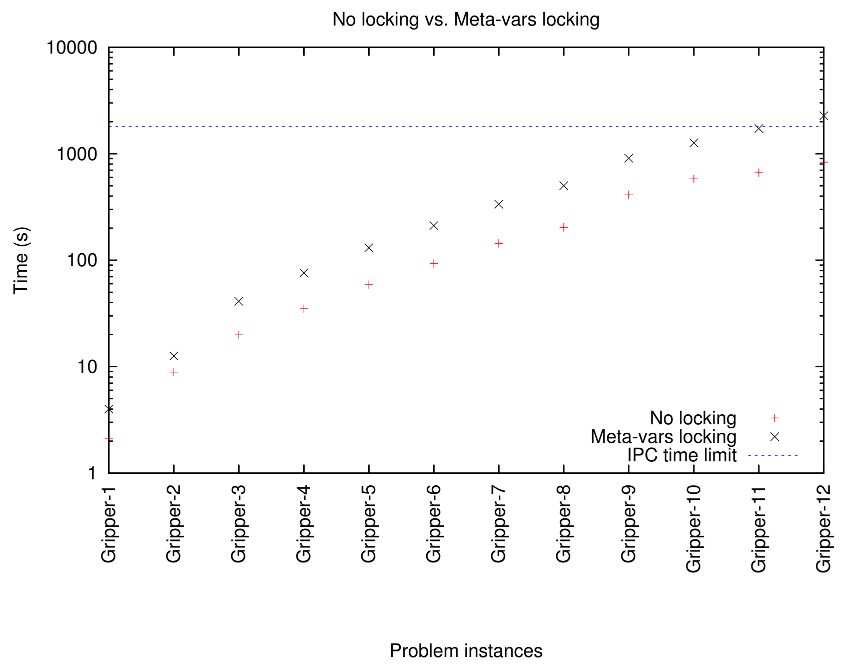


Figure 5.11: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for Gripper instances 1 to 12.

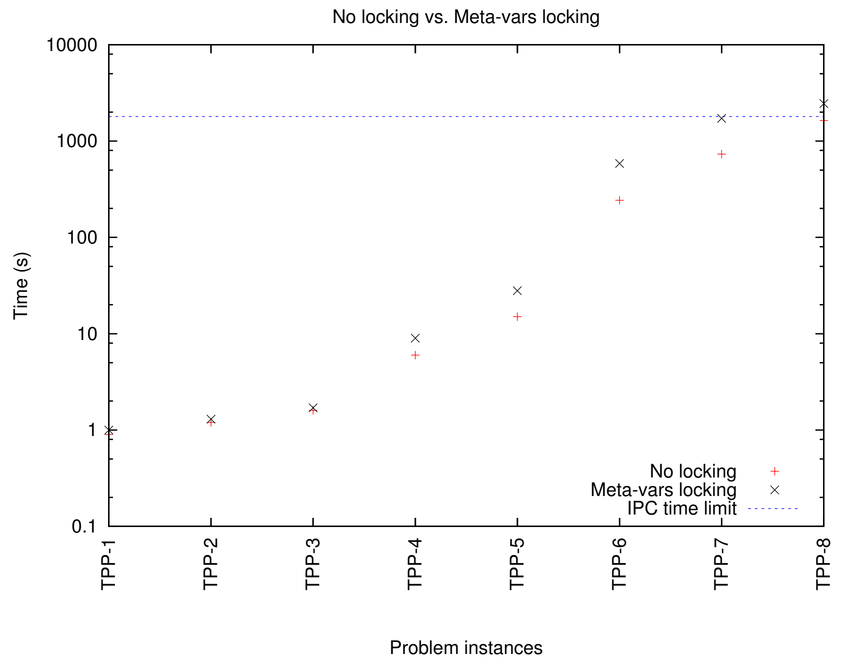


Figure 5.12: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for TPP instances 1 to 8.

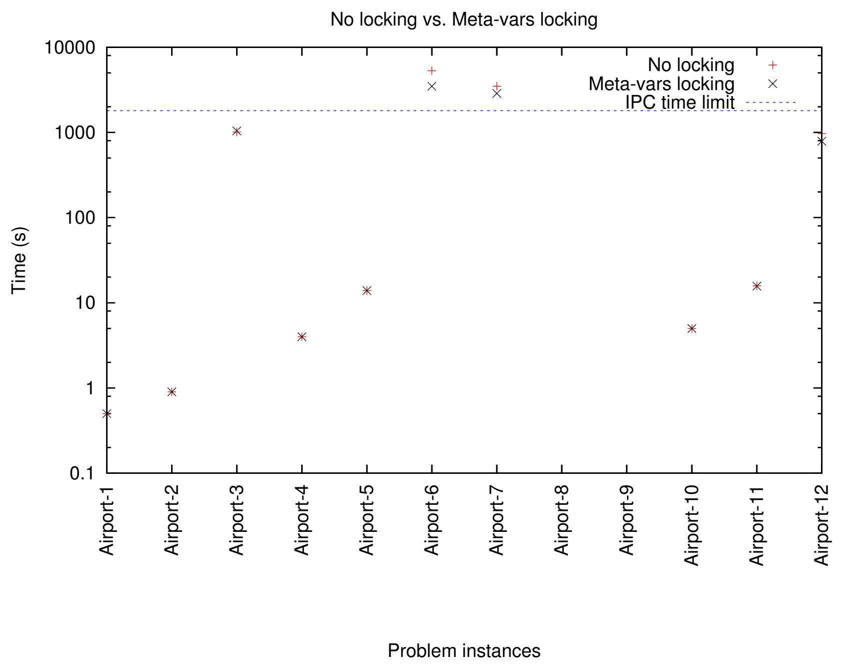


Figure 5.13: Comparison of the heuristically guided search (runtime) with and without meta-variable goal locking for Airport instances 1 to 12.

Figures 5.6 and 5.7 show the result of applying goal-locking to problem instances 1 to 13 in the Driverlog domain. One of the thirteen problems takes longer to solve with the addition of goal-locking, two require the same amount of time, with the remaining ten showing improved run times. Driverlog5, solved faster without locking, requires a "repair" using standard search at the end of the plan. By using the meta-variable goal locking, no repair is necessary since the variable value representing the achieved *truck2* goal (which is undone when using the heuristic with no locking) is fixed until the end of the plan. However, since the repair is a simple one, it is faster to (blind) search for the few actions required (in the non locking version) than it is to process (in the locking version) the required suggested actions, choose one, and complete the sub-plan. Instance 1 is solved in the same time, with and without locking, as previously discussed (Figure 5.3). Similarly for Instance 3, there being no clobbering nor consequent repair(s) necessary. Of the remaining ten instances, problems 2, 4, 6, 8, 9, and 13 required a repair in the non locking approach and benefit from having goals locked, with each instance being solved in less time and with fewer backtracks when locking is applied[1]. Instances 7, 10, and 11, requiring no repair in the non locking ver-

[1]With the exception of instance 6 which has improved run time, but no reduction in backtracking. This is due to the (non locking) repair being an "easy" one which incurred no extra backtracking.

sion still benefit from goal locking, with each requiring less run time to find a solution when locking is applied. There is however, as expected, no improvement in the number of backtracks. No solution to problem 12 was found using the heuristic alone. With the application of goal locking, however, a solution was found, with a low number of backtracks.

Comparing the run time of the heuristic technique, with and without goal locking applied, running on instances 1 to 12 in the Zeno domain (Figures 5.8 and 5.9), nine instances are solved faster (3, 4, 6 - 12), one shows no change (2), with a fractional increase in the remaining two instances (1, 5). When locking is applied to instance 1 there is nothing to be gained. That is, the problem consists of 3 goals (one aeroplane movement goal, two person goals), with two (person goals) already satisfied in the initial state. Thus, the algorithm locks the variables representing the two persons, and then achieves the aeroplane goal. Due to the structure of the problem, locking the person variables has no impact on the aeroplane (see, for example, Figure 5.5), and the time taken to process the additional constraints manifests as a slight overhead. Instance 5, consisting only of person goals, is similarly affected by the structure of the problem domain, in that locking each goal as it is achieved has no impact on any of the others, and again the locking adds very slightly to the run time. Zeno2 requires the same time to solve regardless of whether or not locking is applied. Again, locking the person goals, one of which is satisfied in the initial state, has no effect on the overall solution process (or run time). The remaining nine instances show a reduction in run time when goal locking is applied. Considering that there is no associated change in the amount of backtracking in these instances (Figure 5.9), the goal locking is serving to speed up the consistency checking process. Namely, in the solution matrix, the structure of the problem is such that the locking of the person goals does not prevent (or reduce the number of) backtracks occurring following the inappropriate placement of suggested actions. That is, propagation of the locked values does not prune the domains of the action variables at a given level. However, during completion of the sub plans, which make up the overall solution plan, such columns of fixed values in the matrix will serve to speed up the process of assigning values to the remaining CSP (state) variables at the preceding levels[1].

Figure 5.10 details the run time results gained when goal locking is added to problem instances 1 to 18 in the Elevator domain. The first twelve instances show a decrease in the run time required to find a solution. Instances 13 and above require more

[1]The number of goals is relevant here, with the goal locking being an overhead (as discussed) in instances 1 and 5, but providing some speed-up in instances 3, 4, and 6 - 12.

time to find a solution when goal locking is applied. All instances are solved backtrack-free, regardless of whether or not goal locking is added to the heuristic search procedure. As noted when discussing the initial test set results (i.e. Elevator 3_0 above), the Elevator domain's action schema does not provide a means of undoing a goal of the sort $SERVED\ (p_x)$. Therefore, locked goals would be expected to provide limited, if any, improvement on the non-locked configuration. Also, as previously discussed, it is possible that the addition of goal locking constraints may have an adverse effect on larger problem instances. This is shown to be the case over the range of instances shown in Figure 5.10. Whilst goal locking constraints do improve the run times of the smaller instances, on the larger ones the additional constraint processing is an overhead, leading to an increase in run time.

The results of applying goal locking constraints to instances 1 to 12 of the Gripper domain are shown in Figure 5.11. On all instances there is an increase in the run time required to find a solution when goal locking is used, with no change in the level of backtracking required on any problem instance (see Figure 4.24). As previously mentioned (in relation to Gripper_1, above), the lack of a dependency relationship between balls (goal variables) in this domain means that locking a given ball will have little or no positive impact on any others. In addition to this, the necessarily larger solution matrices[1] required for this domain's problem instances, in conjunction with the single ball (goal) focused nature of the underlying heuristic, exacerbate the overhead caused by adding goal locking meta variable constraints. That is, the larger matrices lead to even more time steps at which the goal variables have to be locked, thereby further increasing the amount of constraint processing and consequent run time.

The run time results for solutions to problem instances 1 to 8 of the TPP domain are shown in Figure 5.12. In this domain each instance is solved faster without goal locking. Here, again, the structure[2] of the problem definition is important. As discussed above (TPP3 in the first test set), the action schema does not provide a method to undo a goal ($STORED\ (goods_x\ , level1)$) once it has been achieved. Hence, with the application of goal locking, there is no advantage to be gained in terms of either a run time improvement or a decrease in the amount of backtracking required to find a solution. The overhead (run time) associated with goal locking is minimal on instances 1 to 3 (Figure 5.12). However, this increases as the problem size increases (instances 4 to 8). The amount of backtracking across the range of TPP problem instances is

[1]Due to the single goal sub plans, which effectively lead to plans approximately double the size of those possible if both grippers were used.

[2]For example, see Figure 4.14.

unchanged from that shown in Figure 4.26.

Problem instances from the Airport domain (Figure 5.13) in which there is only one goal (1, 2, 4, 5, 10, 11) show no change in the amount of run time or backtracking required to find a solution when goal locking is used. In such cases, the backtracking (Figure 4.28) results from the operation of the heuristic, and is therefore the same regardless of whether or not locking is applied. Hence, on such small problems the goal locking meta-variables neither improve performance nor manifest as an overhead. Instances 3, 6, 7, 8, 9, and 12 each have more than one goal. With the exception of instances 8 and 9 (not solved), this set exhibit similar solution characteristics. That is, when goal locking is applied there is an improvement in the run time measure. This is not due to a reduction in the number of backtracks, but comes from improved propagation resulting from the fixed variable values. The exception to this is instance 3 which shows a small increase in run time when locking is used. As previously discussed (Chapter 4), this instance suffers from poor goal ordering in the heuristic solution process, resulting in a larger number of backtracks (Figure 4.28), leading to a longer run time (with and without locking). The added meta-variables result in a 2.7% increase in run time, but no further increase in the amount of backtracking.

Summarising the impact of adding goal locking meta-variables to the instances in the test set domains (Table 5.1), it is clear that there is no negative impact in terms of the amount of backtracking carried out when the extra constraints are added (column 6). Generally, the number of backtracks remains unchanged, with the Driverlog domain being the exception. Here there is a significant decrease in the number of backtracks on several problems (Figure 5.7). This improvement is due to the goal locking mechanism capitalising on the structural dependencies between variables in the problem, highlighting better use of the inherent constraint machinery when guided by the structural information implicit in the planning problem.

Taking run time as a metric (Table 5.1), the impact of adding goal locking meta-variables to the Gripper and TPP domain instances is wholly negative (column 4), for the reasons discussed above; namely that both the structure of these domains and the goal centric nature of the variable / value selection heuristic conspire to make the added meta-variables an overhead. On the Airport domain, only one instance requires more time to solve when the additional variables are used. Here the extra inference achieved as a result of locking a (single) goal is not beneficial, but instead, processing the added constraints increases the solution run time. All other instances in the Airport domain solve in the same time or faster[1] with the meta-variables included. Driverlog

[1]On larger instances, those with an increased number of goals.

Domain	Number of Instances	Lower / same run time	Higher run time	Fewer / same backtracks	More backtracks
Driverlog	13	12	1	13	0
Zeno	12	10	2	12	0
Elevator	18	13	5	18	0
Gripper	12	0	12	12	0
TPP	8	0	8	8	0
Airport	12	11	1	12	0

Table 5.1: Summary of effects of adding goal locking meta-variables to benchmark problems (set 2).

instances solve faster or in the same time when the meta-variables are added, with the exception of instance 5. This problem requires a simple plan "repair", with this being achieved faster using the heuristic plus standard search compared to using the heuristic with locking. Nine of the twelve Zeno instances are solved faster, one shows no change, with the remaining two requiring a very small additional amount of time to solve when the meta-variables are added to their respective CSP models. Goal locking is an overhead on the instances exhibiting increased run time, with the structure of the problems determining that there is no benefit gained from the locked variables, instead the constraint processing adds slightly to the overall solution time. Similarly with the instance that is unchanged, in that locking an achieved goal variable, upon which future goal variables do not depend, has no positive impact on the run time (nor negative impact in this case since the problem is small). Elevator instances 1 to 12 benefit from goal locking, due only to a speed up in the consistency checking[1]. That is, as noted above, once a goal is achieved in this domain, it cannot be undone. Therefore, whilst providing a small improvement in run time, goal locking is of limited value, and in fact becomes an overhead on the larger instances (13 to 18).

The intention in this section was to test the thesis that, by locking a satisfied goal variable's value from the layer in which it is first achieved until the end of the plan, there would be a consequent reduction in the domain size of the other CSP state variables (connected to the goal variable by constraints describing the problem's actions) at a given level, and that inferences made and propagated as a result of the domain reductions would, in certain instances, lead to improved performance, as measured by run-time and amount of backtracking required to find a solution. This concept has been shown to be valid (e.g. Figures 5.6, 5.7 and 5.8) where a good goal ordering can be

[1]Since all instances solved backtrack-free, using the heuristic either with or without goal locking.

found and the structure of the domain allows gains from such a technique (i.e. there are clearly defined and accessible dependency relationships between goal variables). However, it is also clear that, where there is not a good goal ordering and / or there are no dependency relationships between the planning problem's CSP variables, additional inferences, even if made, will not lead to benefits of the same scale in the form of reduced run time or fewer backtracks.

The following section continues this chapter's aim of employing meta-variable techniques to better use the underlying constraint infrastructure to more efficiently solve AI planning problems. In doing so, it continues the attempt to investigate this work's overall thesis. Namely, that it is possible to use the structural information from the original planning problem to guide the search choices in the CSP reformulation of such a problem.

5.3 Meta Variables for Resource to Task Assignment

The previous section described the process of using meta-variables to lock the value of goal variables once these are achieved in a plan. This procedure was an attempt to gain further traction from general CSP techniques directed by information taken from the planning problem. It also served to mitigate the effects of one of the limitations (clobbering) of the CSP heuristic described in Chapter 4 (Section 4.5). The other issue highlighted there was the fact that the CSP variable and value ordering heuristic selects a *suggested action* from a set, the size (and ordering) of which, is reliant upon the uninformed automated CSP encoding procedure.

With further analysis and processing of the planning problem's structural information, it should be possible to reduce the amount of choice during the selection of suggested actions, with selection influenced by prior allocation of resources within the CSP. Thus, considering the planning problem to be one in which there may be a number of *tasks* to be achieved, with some of these requiring *resources*, with a resource to task allocation procedure it would be possible to set which resource is assigned to a given task. This would allow particular resource allocation strategies to be implemented. For example, a uniform resource use allocation policy or a lowest "cost" policy. This section describes a meta-variable based approach which allows such resource allocation, using meta-variables, during the operation of the CSP variable and value selection heuristic described in Chapter 4. The aim here is to further investigate this work's thesis that, by applying structural information from the planning problem during the (CSP) solution process, better use may be made of the underlying CSP machinery, as measured by more efficient plans (in terms of run time, amount of backtracking or plan length).

Section 5.3.1 first describes the resource to task assignment process in general terms, and then discusses how this may be achieved in the CSP based planning framework used in this work. The following section (Section 5.3.2) introduces a particular resource assignment approach. The results of applying this example resource to task policy to the problems in the test sets are detailed in Section 5.3.3.

5.3.1 Assigning Resources to Tasks

Whilst planning is the problem of finding a set of actions which allows progression from an initial state to a goal state, scheduling is the problem of ensuring adequate resources are allocated at the required time in order to allow the selected action sequence to successfully complete the appropriate task(s) and achieve the goal(s). For example, with a single 10 litre bucket, two serial sequences of *fill*, *carry*, and *empty* would be required to fill a 20 litre tank. Here, the bucket may be considered a resource for the task of filling the tank. This notion of resources is similar to the idea of *unary resources* [221], in which a single machine can support only one activity at a given time.

If another bucket were available in the above planning task, and assuming a serial plan (i.e. buckets can not be used simultaneously), then further choice is introduced, with the planner required to find, not only the sequence of actions, but also the particular instantiated resource object. That is, the planner may select the same bucket for each task, or different buckets (in either combination).

If pre-analysis identifies each of the buckets as being a potential resource for the tank filling task, then a meta variable, which encodes the allocation of only one of these to the task, may be added to the CSP before starting the solution process. With this in place, there is no choice, in terms of resource allocation, at each point where an action requiring the resource is needed. Namely, actions are pruned from the search space, forced out by propagation of the meta variable constraints which encode the links between task and resource.

With reference to the planning specific, CSP heuristic discussed in Chapter 4, this resource choice manifests as additional suggested actions, with these treated as being equally appropriate at a given level, with the consequent requirement for tentative placement and subsequent consistency checking. In order to better use the inherent CSP mechanisms employed to solve the CSP encoded planning problem, additional information may be taken from the original planning problem, with this then used to generate a set of meta variables. Such meta variables can encode particular assignments of resource to task, with this information in turn informing the CSP search process by potentially reducing the amount of choice at a given level.

A simple analysis of the CSP encoding of the original planning problem furnishes a set of tasks, the goal state goals. With these identified, it is then possible to find a set of CSP state variables upon which these depend. In this way, from each action, the required resources for each task are identified. Table 5.2 shows the result of such an analysis for problem instance two from the Driverlog domain.

CSP variable	Depends on	Variable meaning
1	1 or 6 or 7	Driver 2
2	2 or 6 or 7	Driver 1
3	6 or 7	Package 2
4	6 or 7	Package 3
5	6 or 7	Package 1
6	1 or 2	Truck 1
7	1 or 2	Truck 2

Table 5.2: Variable dependencies for resource to task allocation (Driverlog 2).

A set of meta variables reflecting a resource allocation policy can then be developed. For example, considering Table 5.2, a meta variable, A, may be assigned to indicate that, for any tasks involving variable 1 as a goal, the only resource available is variable 1. In this problem instance, this means that whenever *driver 2* is required to change position the only "resource" available is, in fact, *driver 2*. Namely, the driver is forced to walk, not make use of a truck. Likewise, another meta variable, B, may be assigned to indicate that, whenever variable 5 is required, it can only make use of the resource, variable 6. Here, the meaning is that whenever *package 1* has to be moved, it may only be transported by *truck 1*. The following subsection introduces an example resource allocation policy, the one used to test the efficacy of the idea of using meta variables to assign resources, and thereby potentially make better use of the CSP solver for the solution of AI planning problems.

5.3.2 A Resource Assignment Policy

Using meta variables for resource to task assignment allows various resource allocation strategies to be employed. These may include, with further analysis, a lowest cost strategy, in which the most cost-effective resources are employed. Alternatively, a policy of even-use could be configured, whereby the available resources are evenly divided between the problem's tasks.

In this section, a simple minimise-resources scheme is used. The intention here is to determine the impact on the CSP solution process of this pre-allocation of resources. This additional planning problem resource information, added to the CSP, should reduce the amount of choice in the problem, generally. More specifically, by removing alternative goal achieving actions (those using different resources), the number of *suggested actions* should be reduced, leading to a potential decrease in run time, a possible reduction in the level of backtracking recorded, and / or a change in plan length.

The minimise-resources meta variables approach first identifies the variable dependencies for resource allocation (e.g. as shown in Table 5.2), then variables (such as variables 1 and 2) which can provide their own resources are allocated. Next, for each subset of the variables having common resources, a single resource is selected. For example, in the Driverlog 2 problem instance, variables 1 and 2 are assigned resources 1 and 2, respectively. Next, variables 3, 4, and 5 are each assigned variable 6. Finally, variables 6 and 7 are both assigned variable 1 as a resource. The meaning of these variable assignments is that *driver 1* and *driver 2* do not use any other resource, all packages rely on *truck 1*, and both *truck 1* and *truck 2* use *driver 2* as a resource (although, only *truck 1* is actually used in this problem). With this allocation of resources, the Driverlog 2 problem exhibits a 31% reduction in the time required to find a solution[1].

Whilst, in the above illustrative example, the application of the simple resource allocation scheme results in an improvement in run time, it does not lead to a reduction in the amount of backtracking. There is, however, a reduction in the number of actions in the solution plan. This highlights the fact that the particular structure of a given problem instance will influence the level of change (if any) in the metrics used in this work (run time, amount of backtracking, and plan length) when the meta variables are included. Reiterating, the intention here is not to find the most efficient plan, instead it is to observe the impact on the CSP solution process of applying resource allocation meta variables. That is, by applying planning problem (resource / task) knowledge, encoded as additional constraints, the aim is to observe the impact on the amount of choice during the operation of the CSP solver. The following section discusses the results obtained when the above resource allocation policy was implemented on the problems in the test sets used throughout this work.

[1]Discussed further in the Results section, and shown in Figure 5.15.

5.3.3 Results

This section discusses the results of applying a "minimise resources" policy to the problems in the test sets[1] used previously. The results from the first test set are given below in Table 5.3 and are shown graphically in Figure 5.14. Table 5.4 shows the plan length for each of the instances in this test set, as well as indicating the corresponding optimum plan length.

Instance	Heuristic + Locking Meta-Var (MV) RT(secs)	Heuristic + Locking MV BT	Heuristic + Locking + Resource MV RT(secs)	Heuristic + Locking + Resource MV BT
Driverlog1	4.8	0	2.8	0
Mystery1	15.6	3	8.9	3
PipesTank1	11	1	10.5	1
Elevator2_1	0.4	0	0.2	0
Zeno2	3.8	1	2.3	1
Schedule2_1	163	0	104	0
Schedule2_4	571.7	0	352	0
Gripper1	4	8	1.9	4
Blocks4_1	4.3	2	4.3	2
Blocks5_0	-	-	-	-
Blocks5_1	7.8	2	7.1	2
TPP3	1.5	3	1.0	3
PSR12	-	-	-	-
Elevator3_0	1.3	0	0.7	0
Zeno3	9	2	5.8	2

Table 5.3: Comparison of heuristic (plus locking) search with heuristic (plus locking and resource allocation meta vars) search (runtime and number of backtracks).

With the minimise resource use strategy in place, Driverlog1's meta variables are assigned values, which in turn restricts the values of the CSP state variables. Here, the goal is for one driver (d1) and one truck (t1) to move location to S1, from S2 and S0, respectively. By allocating driver, d1 (resource), to truck, t1 (task), the minimised resources approach solves the problem in seven steps (optimum), with zero backtracks. The meta-variable allocation of resources leads to a reduced amount of choice, since

[1]Links to all test domains can be found in Appendix A.

all actions using other drivers and trucks are pruned, with a consequent reduction in the amount of consistency checking required. Since the solution plan was already optimal (plan length), with no backtracking required, the only benefit observed is a reduction in run time.

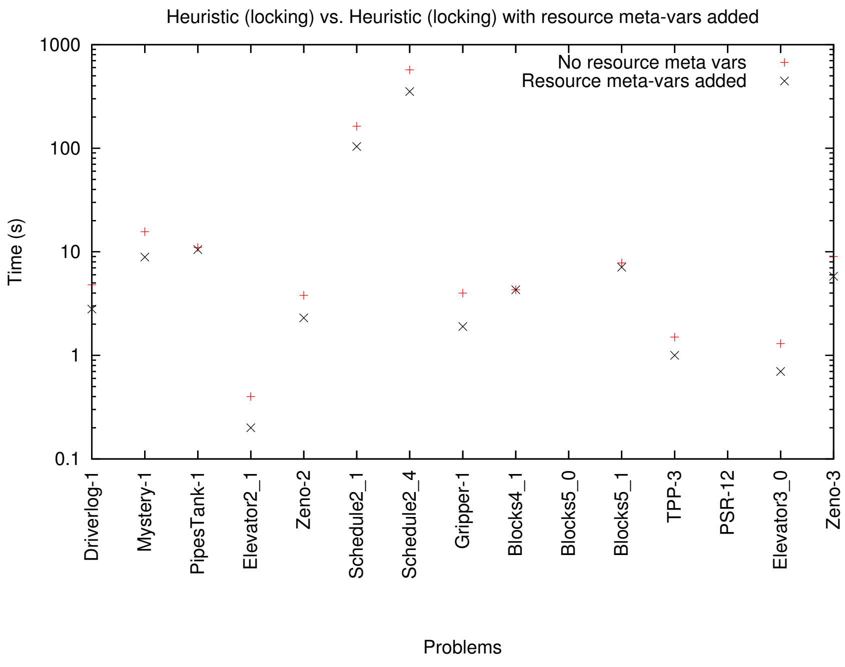


Figure 5.14: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime).

Although disguised, as discussed in Chapter 3, the Mystery domain is a form of transport problem, with the goal in Mystery1 being for a cargo item, “abrasion”, to be placed at a location, “rice”. In the initial state, the cargo item is at location, “pork”. In this problem there is no choice, in terms of variables representing trucks since there is only one truck specified in the problem. However, by assigning a meta-variable linking that truck resource, “rest”, to the cargo item, “abrasion”, a decrease in run time is observed, due to a speed-up in the action selection process.

The meaning of “minimising resources” in the Pipesworld domain is that a limit is placed on which tankage areas and / or pipe segments are used. In the CSP, the effect of assigning meta variables to determine such resource assignment, is that certain goal variables are constrained to rely upon a subset of the other supporting variables, with the result being that certain planning actions (via CSP constraints) are excluded from the search space. Minimising resource use in the PipesTank1 problem instance leads to a plan of length six (not optimal), and a single backtrack, with these achieved faster

Instance	Heuristic + Locking Plan Length	Heuristic + Locking + Res. Alloc. Plan Length	Optimum Plan Length
Driverlog1	7	7	7
Mystery1	5	5	5
PipesTank1	5	6	5
Elevator2_1	7	7	7
Zeno2	6	6	6
Schedule2_1	2	3	2
Schedule2_4	3	3	3
Gripper1	15	15	11
Blocks4_1	14	14	10
Blocks5_0	-	-	12
Blocks5_1	10	10	10
TPP3	15	15	11
PSR12	-	-	12
Elevator3_0	11	11	10
Zeno3	6	6	6

Table 5.4: Comparison (plan length) of heuristic search, with and without resource allocation applied.

than when using the heuristic without resource allocation meta variables. Investigating alternative resource allocations (not shown) led to a plan of the same length, achieved in a similar run time, but with no backtracking.

In the Elevator2_1 problem, minimising resource use provides a speed up in the time taken to find a solution, due only to faster consistency checking resulting from the added meta variables. This small problem has only one action capable of satisfying each goal, and therefore there is no pruning of the number of suggested actions as a result of the resource allocation.

With only one aeroplane (resource) available in the Zeno2 problem, there is no choice regarding which aircraft to use. The single backtrack is a result of incorrectly placing the chosen *disembark* action in action slot three, whereas the first consistent location is slot four. The meta variable pre-assignment of the aeroplane resource does lead to an improvement in run time.

Assigning meta variables to enforce a minimum resources policy in the Schedule2_1 problem instance leads to a reduced run time, no change in the backtrack count (remains at zero), but with an additional action increasing the plan length by one. The run time reduction is observed to be a direct result of action pruning, which reduces the size of the sets of actions suggested for each goal. That is, by pre-assigning a given resource (e.g. the spray painter or the immersion painter) to a particular (painting) task, a large number of actions relating to the other resources are removed. In a domain with such a large number of instantiated suggested actions, all of which may be consistency checked at each level, a significant reduction is welcome. The increase in plan length is due to the need for a *do-time-step* action between the painting actions using the shared resource. This action releases the resource following its use in a *do-immersion-paint* action.

Similarly, Schedule2_4 exhibits reduced run time when the resource allocation meta variables are used. Here again, a significant reduction in the number of suggested actions leads to a faster solution, and a zero backtrack count, with the resulting plan remaining optimal, in terms of plan length.

As discussed in the previous chapter, the plan length of the solutions for problems in the Gripper domain are adversely affected by the single goal focus of the heuristic. However, the application of meta variables for resource assignment, in the form of the allocation of a particular gripper hand, removes some of the action choices (relating to the other hand). In this way, at each level, a reduced number of actions are considered, with a consequent improvement in both run time and backtrack count. The plan length remains unchanged, as expected.

The resources identified in the Blocksworld problem, Blocksworld4_1, are other blocks. That is, for a given block, where that block is to be placed on top of another in the goal state, the second (lower) block is identified as a resource for the first. This is intuitively correct since, without the availability of the lower block's unobstructed upper surface, the current block could not be placed there. With the addition of the resource meta variables, Blocksworld4_1 requires the same run time to find a solution, of the same length, as that using the heuristic alone. The number of backtracks also remains unchanged. In this small problem, this is expected since there is no alternative assignment that will lead to a solution, with no reduction in choice due to pruning of suggested actions. Also, the addition of the meta variables is not significant in terms of adding overhead to the problem solution process.

Blocksworld5_0 again fails to solve within the time limit, for the reasons discussed in Section 5.2.1. Namely, since an interleaving of (subsequent goal-supporting) actions is required, the single goal focused heuristic with goal locking and resource meta variables fails to find a solution, forcing reliance on standard search.

Although Blocksworld5_1 also does not afford any opportunity for significant action pruning as a result of adding meta variables for resource allocation, using these does give a small improvement in run time. The amount of backtracking is unaffected, with the backtracks there are being a result of placing suggested actions in positions initially deemed appropriate (action is within the given slot's domain), but which is subsequently found to be inconsistent. Plan length is not affected, remaining optimal.

Referring again to the TPP3 CG (Figure 4.14), the TPP3 transportation problem does not allow for the reduction of choice via directed allocation of resources, although this instance clearly benefits from the application of resource allocation meta variables by exhibiting a reduced run time. Here again, although there is no pruning of the suggested action sets, faster consistency checking leads to a faster solution, with the backtrack count and plan length remaining the same, the latter again being sub-optimal due to the single goal-focus of the heuristic guidance.

As described in chapters 3 and 4, the PSR domain models an electrical distribution network, consisting of power sources, transmission lines, switches, and circuit breakers. As previously discussed there, the structure of this problem, and the consequent lack of a good goal ordering, combined with the goal-directed heuristic, led to poor performance in the CSP reformulation. Since the initial state is considered unsafe (i.e. a power source connected via circuit breakers and switches to at least one faulty line (load)), there must be, in any solution plan, a *wait* action(s) (opens circuit breakers) and an *open* action(s) (opens dangerous switches) before closing circuit breakers in order to make the circuit safe, reconnecting only those loads not exhibiting a fault. The allocation, and minimisation, of resources here is not necessarily as intuitive as it is in other domains. However, since the *open* and *close* (switch) actions require the state to be safe, the circuit breakers can be considered a resource. In this particular problem instance, the goal state requires switch_5 to be closed, but since there is only one action that achieves this, the use of the meta variables does not provide any reduction in the size of the search space, since both circuit breakers are required. Again, the dominant feature in this instance is the poor goal ordering, and the consequent need for a plan repair, which in turn relies on blind search. Hence, no solution is found within the time limit.

Elevator3_0, as with the previous instance in this domain, is solved faster with the inclusion of resource allocation meta variables. However, again this is due to faster consistency checking, since any choice in the order in which the passengers goals are achieved is already determined by the heuristic, and there is only one elevator resource, which all passengers must use. Hence there is no significant action pruning, but the added constraints do provide a reduction in run time, with both the plan length and amount of backtracking unaffected.

In the Zeno3 problem, the choice of resource centres on which of the available aeroplanes should be used to transport the passengers in order to satisfy the goal-state goals. Using meta variables to assign CSP state variable 8 (aeroplane 1) to all of the person goal variables, an optimal plan is found, with only 2 backtracks. Thus, directing the search with the resource allocation policy allows a plan to be found in less time, with the backtrack count and plan length unchanged.

In summary, thirteen of the fifteen problem instances in the initial test set were solved. The two unsolved instances failed for the reasons described previously. That is, the structure of these problems forces reliance on blind search following the failure of the heuristic with the meta variables added. Of the set solved, none showed an increase in the amount of backtracking required, with all but one being solved in a shorter time when the minimise-resources meta variables were added. The exception being Blocks4_1, which exhibited no change in run time. A single instance, Gripper1, backtracked fifty percent less with the addition of the meta variables. This was due to a reduction in the amount of choice in the solution space, namely a decrease in the number of available gripper arms due to the meta variable resource assignment policy. Considering plan length, only two instances (Schedule2_1, PipesTank1) generated longer plans (one additional action). In the Schedule domain, minimising resource usage by exclusively using an immersion painter required an additional action in order to free that resource between uses (goals). In the PipesTank domain, forcing the use of a specific pipe segment required an additional *pop* action to clear that segment.

Figures 5.15 to 5.25 show the results (run time and backtracks) of applying a minimise resources policy to the problems in the second test set[1]. Solution plan lengths, with and without resource allocation meta variables, are given in Figures 5.26 to 5.31.

With the application of meta variables, all thirteen instances from the Driverlog domain were solved faster (Figure 5.15) than when using no resource allocation policy. Instances 1 to 4, 8, and 13 required the same number of backtracks (Figure 5.16) as their respective non meta variable solutions, whilst instances 6, 7, 10, 11, and 12 generated fewer backtracks when the solution was guided by the meta variables. However, an increased number of backtracks were found in the solutions to problems 5 and 9. Considering plan length (Figure 5.26), instances 4, 5, 7, 9, and 11 have longer plans, instances 1, 3, 10, 12, and 13 have plans of the same length as their respective non meta variable solutions, with problems 2, 6, and 8 requiring plans of a shorter length. Taking all three measures together, it is clear that faster solutions can be found, regardless of

[1]Links to all test domains can be found in Appendix A.

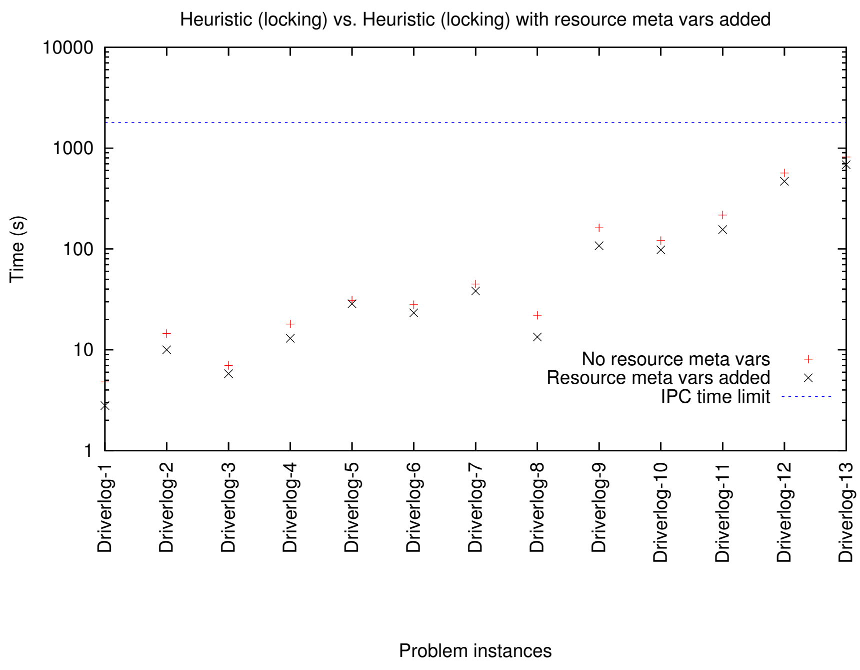


Figure 5.15: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for Driverlog instances 1 to 13.

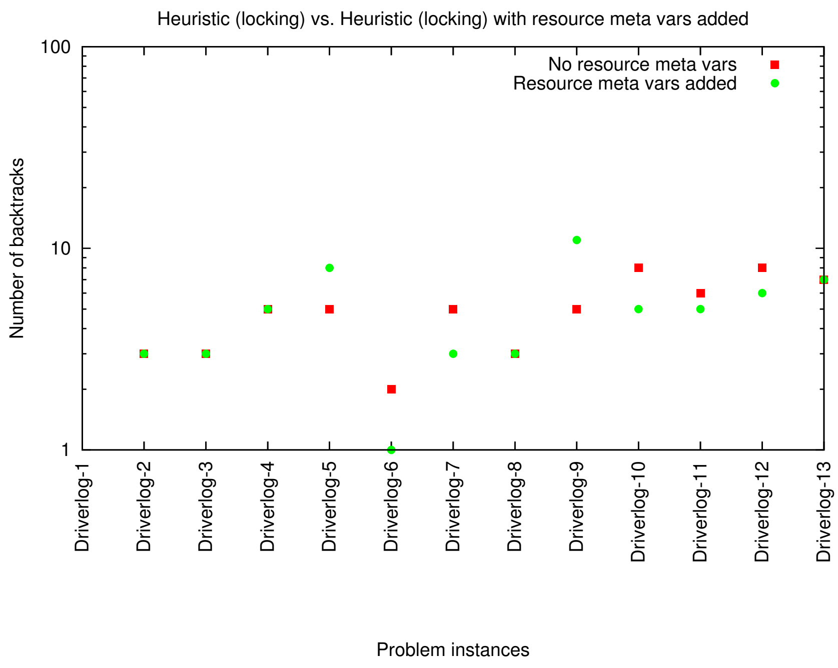


Figure 5.16: Comparison of heuristically guided search with heuristic using resource allocation meta variables (backtracks) for Driverlog instances 1 to 13.

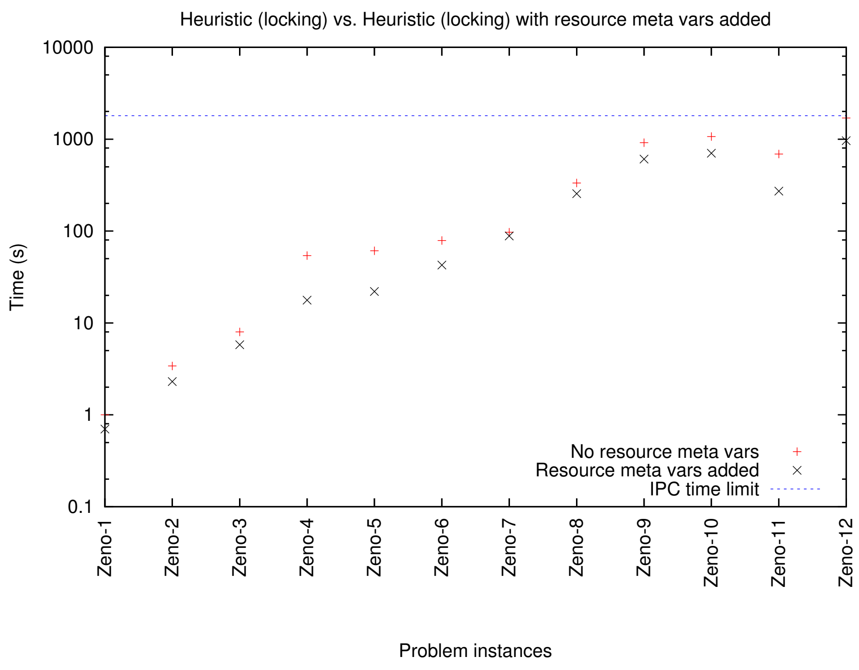


Figure 5.17: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for Zeno instances 1 to 12.

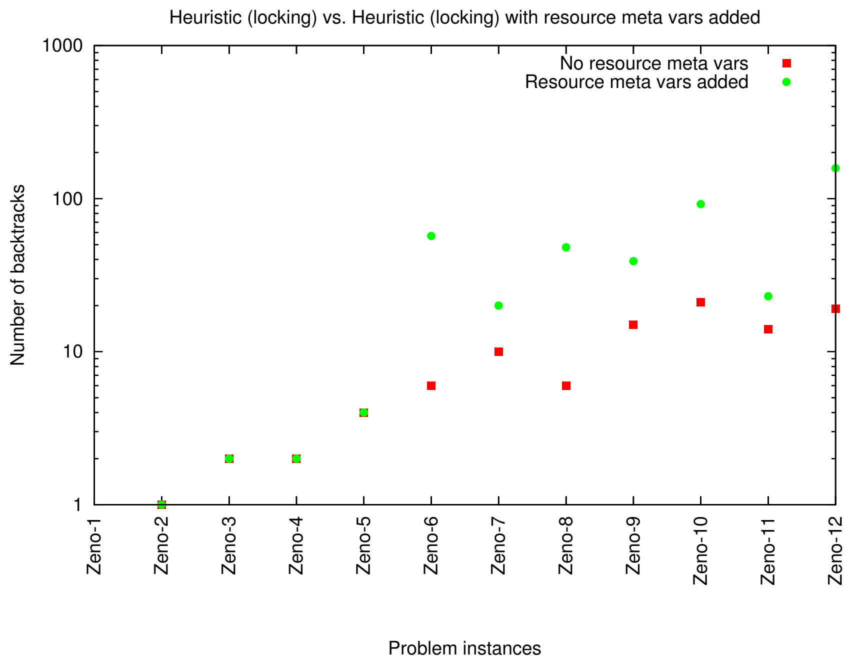


Figure 5.18: Comparison of heuristically guided search with heuristic using resource allocation meta variables (backtracks) for Zeno instances 1 to 12.

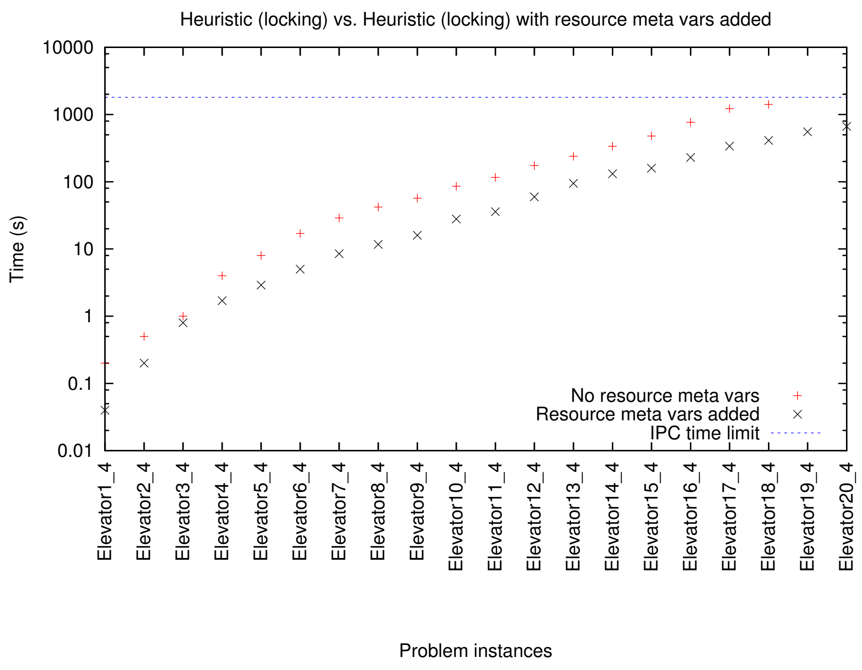


Figure 5.19: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for Elevator instances 1 to 20.

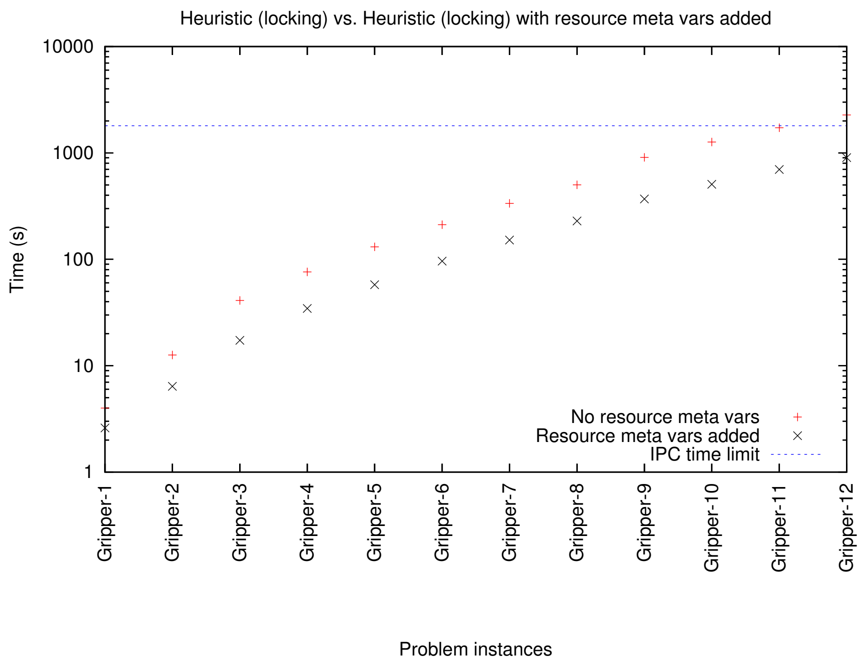


Figure 5.20: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for Gripper instances 1 to 12.

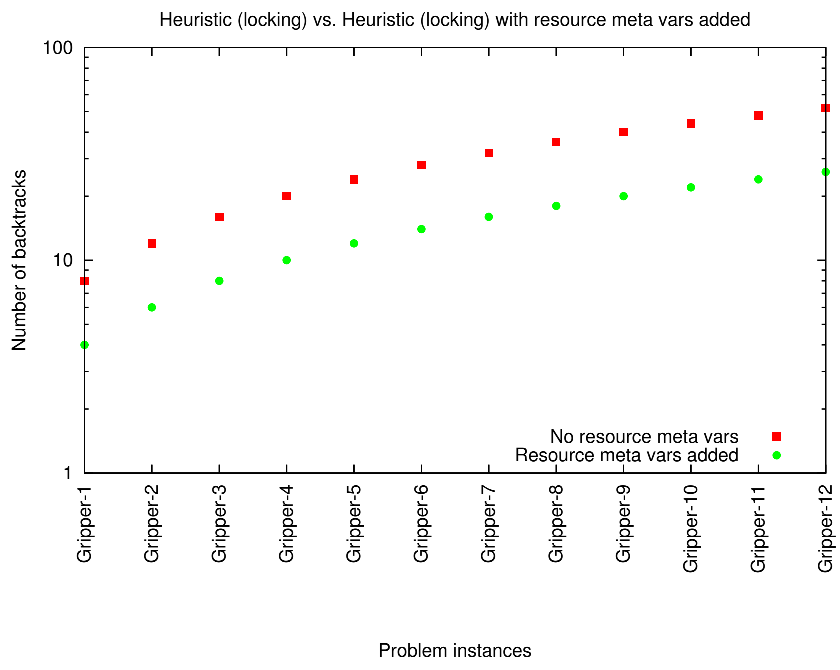


Figure 5.21: Comparison of heuristically guided search with heuristic using resource allocation meta variables (backtracks) for Gripper instances 1 to 12.

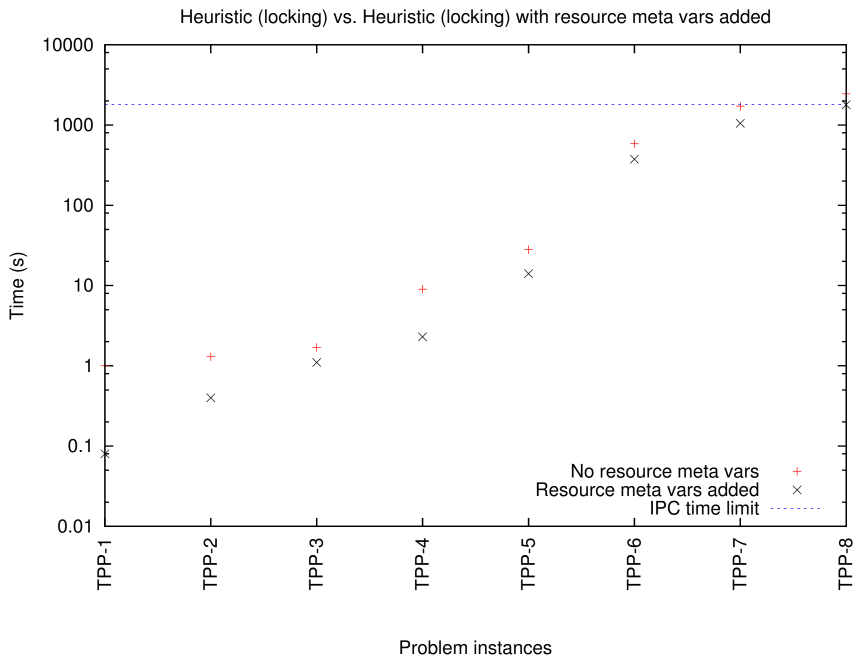


Figure 5.22: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for TPP instances 1 to 8.

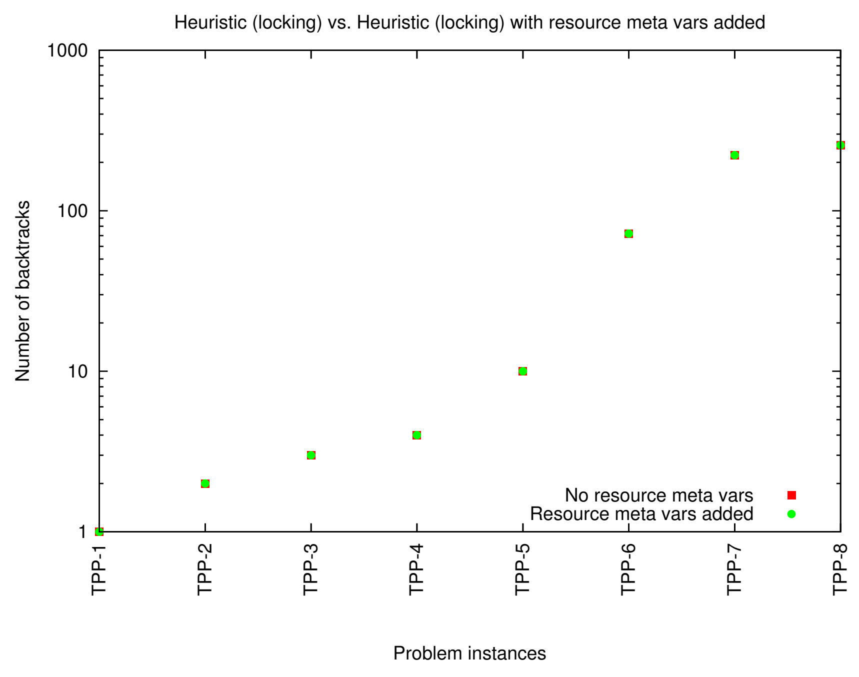


Figure 5.23: Comparison of heuristically guided search with heuristic using resource allocation meta variables (backtracks) for TPP instances 1 to 8.

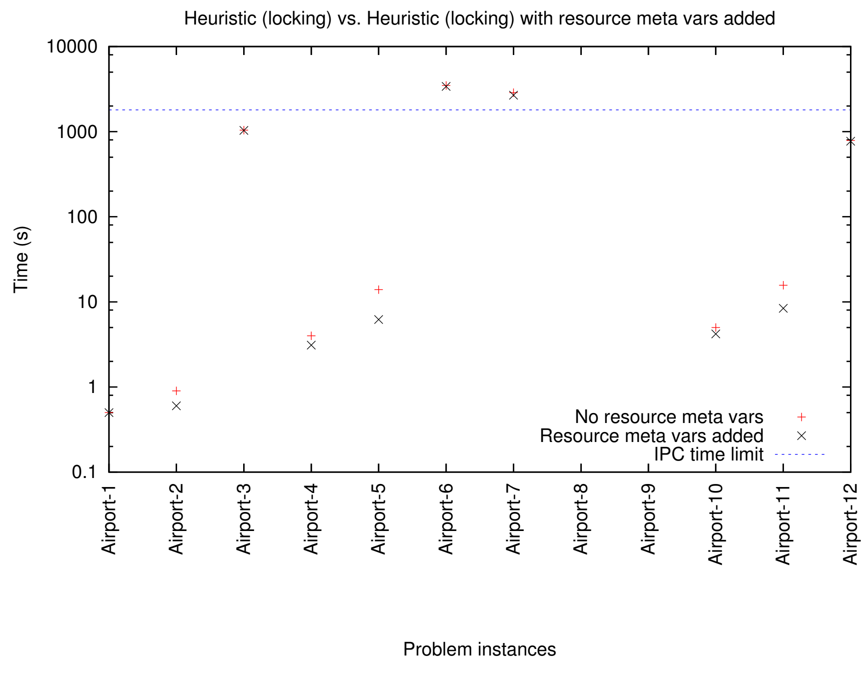


Figure 5.24: Comparison of heuristically guided search with heuristic using resource allocation meta variables (runtime) for Airport instances 1 to 12.

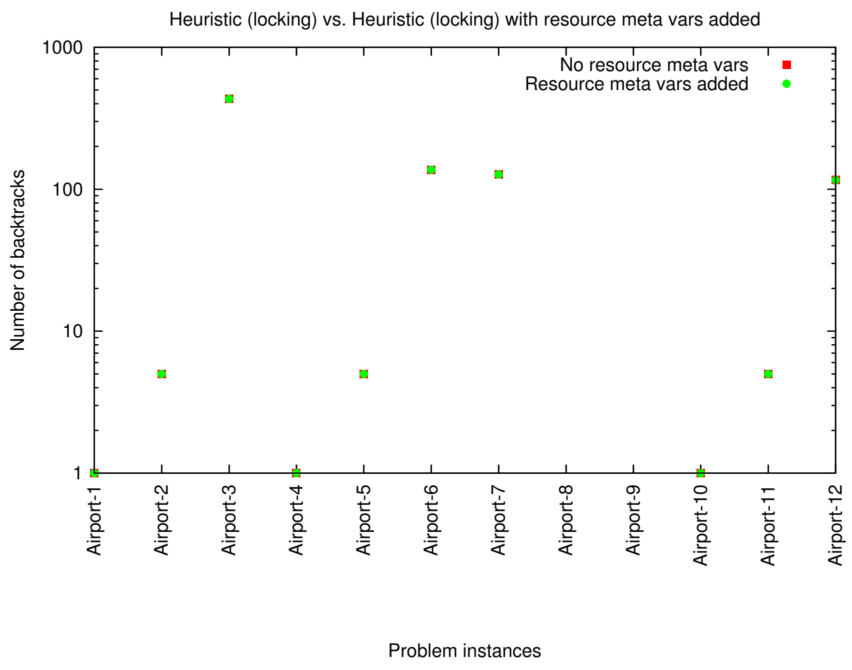


Figure 5.25: Comparison of heuristically guided search with heuristic using resource allocation meta variables (backtracks) for Airport instances 1 to 12.

whether there are more or less backtracks generated, and irrespective of whether the solution plan length is longer or shorter than the non guided equivalent.

The result of adding meta variables to minimise the resource use in the Driverlog domain is that a subset of the suggested actions are used, only those pertaining to the chosen resource. Hence, depending on the structure of the problem instance, the consequence of this reduction in choice may be less backtracking, since a reduced number of actions are chosen, and subsequently rejected as being inconsistent at any given level. However, by selecting actions from this subset, the most efficient plan (in terms of number of actions) may not be achieved since the meaning of reduced resource use here is, for example, lowest number of trucks, and not least distance covered by a particular truck. Thus, it is possible that, having reduced the amount of choice in terms of (sub)goal achieving action, the resulting sub plan is longer, which in addition to generating a longer overall plan, may actually increase the total amount of backtracking. From the set of instances solved, it is clear that regardless of plan length or amount of backtracking, the solutions are generated faster with the inclusion of the meta variables.

Figure 5.17 shows the run time results of using the meta variable guidance on problems 1 to 12 in the Zeno domain, with Figures 5.18 and 5.27 showing the number of

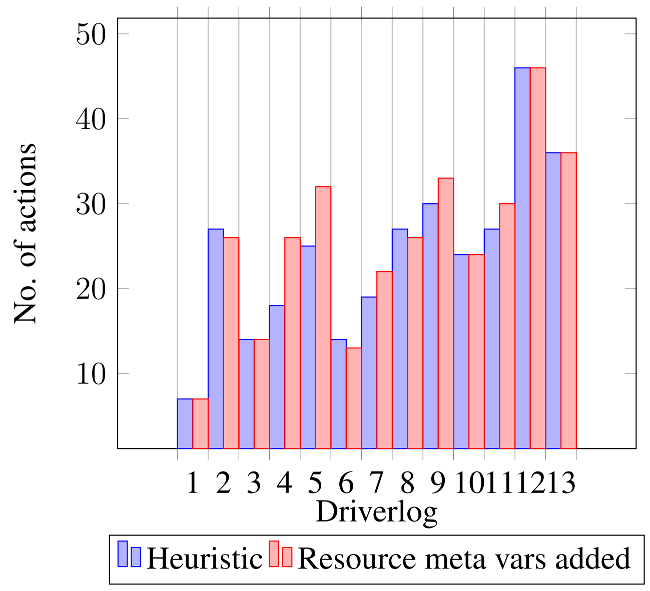


Figure 5.26: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (Driverlog instances 1 to 13).

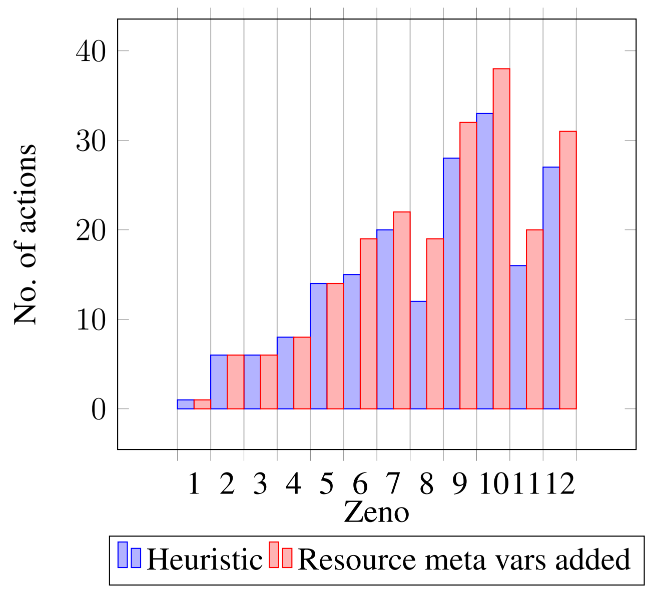


Figure 5.27: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (Zeno instances 1 to 12).

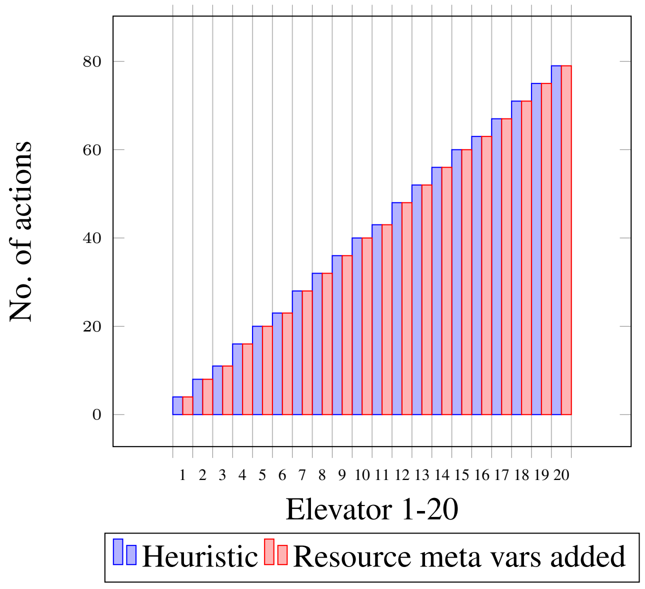


Figure 5.28: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (Elevator instances 1 to 20).

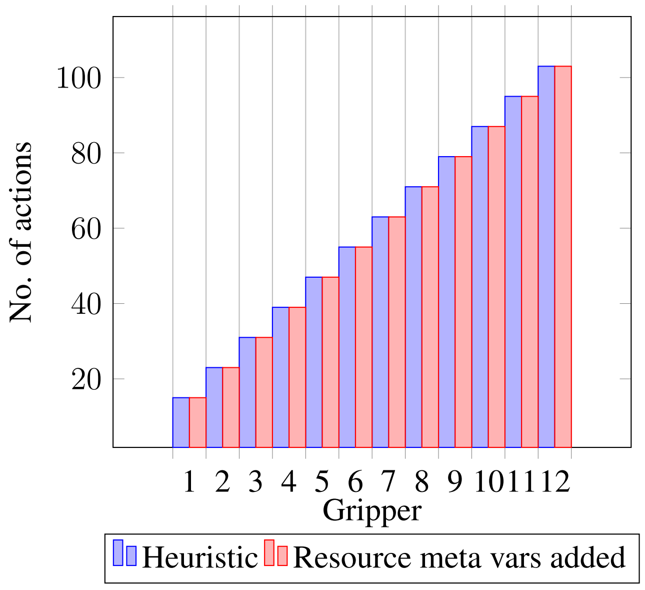


Figure 5.29: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (Gripper instances 1 to 12).

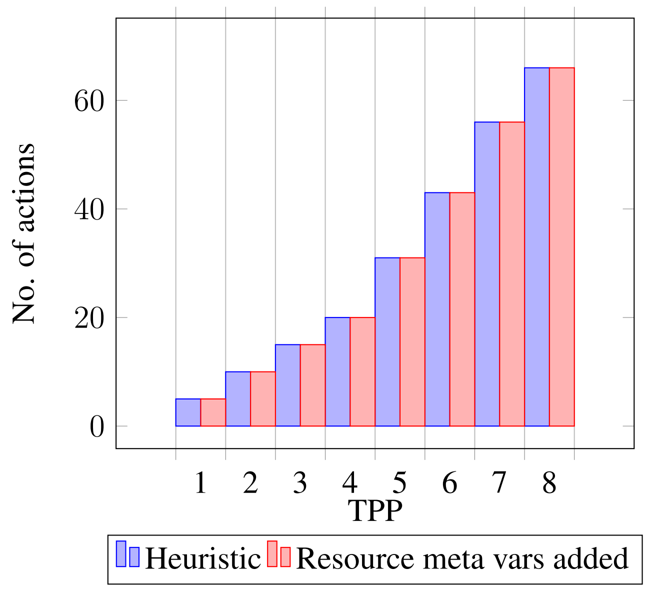


Figure 5.30: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (TPP instances 1 to 8).

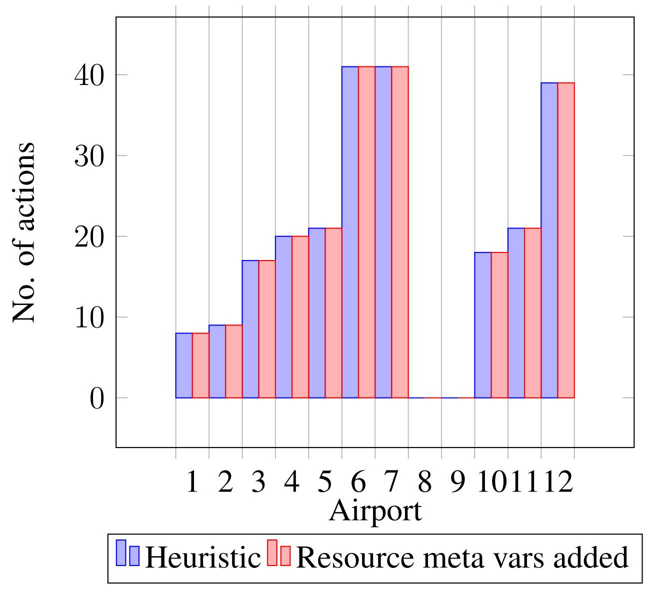


Figure 5.31: Plan length for heuristically guided solutions compared to those for solutions using resource allocation meta variables (Airport instances 1 to 12).

backtracks and plan lengths, respectively. Again, all instances in the test set are solved faster when resource allocation meta variables are added. The level of backtracking is unchanged for Zeno problems 1 to 5, with backtracking increasing for all remaining problems in the set. Similarly, plan lengths for the first five instances are unchanged, with the length of solution plans for problems 6 to 12 increasing with the addition of the meta variables. Taking Zeno6 as an example, the resource allocation meta variables force the use of certain resources (e.g. aeroplane2), with the solver placing a suitable (sub)goal achieving action[1], such as *debarkperson5plane2city1*, in a seemingly appropriate action slot. This action slot can be significantly further from the end of the nascent plan than would be a non (meta variable) guided action's slot. Whilst the (desired) effect of this guidance is a reduction in choice, a side-effect is that, with this action further from the nearest constrained variables (last solved sub plan), it is possible for the sub plan to be almost fully populated before propagation of the variables' values highlights an inconsistency. This is what causes the increased backtracking on this, and remaining, Zeno instances. That is, an action is chosen from the reduced set of suggested actions, it is placed, and the sub plan is partially populated. Due to the larger size of the sub plan, only then is the inconsistency found, with this then requiring many more backtracks than if the incorrect placement had been discovered earlier. Despite this, in terms of run time, the benefits of reducing the amount of choice in the problem still outweigh this overhead.

All instances in the test set from the Elevator domain are solved faster when resource allocation meta variables are added to the CSP (Figure 5.19). There is no change in the amount of backtracking (not shown), with this remaining at zero for all instances. Likewise, the plan lengths (Figure 5.28) remain unchanged. These results are not surprising since the choice in this problem is focused on the order in which elevator users are boarded and serviced. The goal ordering incorporated in the heuristic used in this work determines this, with the additional resource constraints providing a decrease in solution time due to faster consistency checking, since the only "resource" is the (single) elevator.

As discussed in relation to the Gripper instance in the first test set, action pruning provides a significant decrease in both the run time and amount of backtracking (Figures 5.20 and 5.21) required to find a solution in each of the twelve Gripper problems solved using the resource allocation meta variable guidance. The heuristic process is necessarily single goal centric, which means that either of the two available gripper hands can be used for each goal, but also that two will never be used simultaneously.

[1]From the resource allocated reduced set of suggested actions.

Therefore, by automatically selecting only one of these resources via the meta variables, this action symmetry can be removed. The plan length measure (Figure 5.29) for each instance is, as expected, unchanged.

All eight problem instances from the TPP domain are solved faster when meta variables are added to the CSP (Figure 5.22), with no change in the amount of backtracking (Figure 5.23). Instances 1 to 4 have no choice in relation to resource assignment since only one truck (and one market) are specified in each of the problems. The run time improvement is again observed to be due to faster consistency checking throughout the solution process, not directly as a result of action pruning. However the larger TPP instances in the set (5 to 8) do benefit from a reduction in choice, since there are additional resources available in each of these problems. Hence, the addition of meta variables, whilst not reducing the amount of backtracking, does lead to a reduction in the amount of time required to find a solution. Pre-assignment of resources does not affect solution plan length (Figure 5.30) since the goals are solved separately, and the resulting sub plans follow a predictable pattern containing actions for reaching the market from a depot, buying and loading goods, and returning from the market to a depot for unloading.

The final set of problems in the test set come from the Airport domain. Ten of the twelve instances were solved by the resource assignment based meta variable solution process (Figures 5.24 and 5.25), with instances 8 and 9 again not solved (due to CSP size (e.g. see Figure 4.34)). The (relatively) small, less complex, instances (1, 2, 4, 5, 10, 11) exhibit improved run time to find a solution, with the associated backtrack counts and plan lengths (Figure 5.31) unaffected. These last two measures are influenced by the nature of the heuristic, not by the pre-assignment of resources, since there is a low number of both goals and resources in these problems. This leads to no action pruning as a result of the meta variable guidance, but instead a decrease in run time due to faster consistency checking. On the larger or more complex instances (3, 6, 7, 12) there is, again, no action pruning since, due to the structure of the problems, it is not possible to select a resource which removes any actions from the sets of those suggested for the achievement of the goal-state goals. That is, for a given goal variable (action achieving it), all available resources are required. Thus, there is no reduction in choice; if again all actions are applicable, all need to be tested for consistency when suggested at a particular level. As a consequence, any gain in run time resulting from faster consistency checking is reduced by the need to carry out additional action checking at each level. There is, however, a small reduction in run time for instances 3, 6, 7, and 12. The first of these is again hampered by poor goal ordering in the initial heuristic stage.

In summary, the application of additional meta variables to implement a resource assignment policy, prior to the heuristically guided solution of the instances in the test sets used in this work, leads to an improvement in run time in all but two of the eighty five different problems tested. The remaining two problems were solved in the same time with or without additional guidance using meta variables, due to their small size.

The particular resource assignment policy used here was one which minimised the use of a given resource. For example, in problems with multiple vehicles, the minimum number was used. On several instances (e.g. Driverlog), the amount of backtracking was reduced. However, in other cases more backtracking occurred (e.g. Zeno). This highlights the relevance of a given problem instance's structure in relation to the particular resources assigned by the meta variables in the CSP solution space. That is, a beneficial assignment may (by chance) take advantage of the proximity of a resource, thereby solving a problem with less backtracking, and perhaps with fewer actions (e.g. Driverlog instances 2, 6, and 8). Alternatively, the opposite situation is also possible (e.g. Zeno 6 to 12). Here, a less "efficient"[1], allocation of resources leads to more inconsistent variable and value choices, which in turn leads to more backtracking and longer plans (e.g. Driverlog 4, 5, 9, 11). A consistent reduction in the level of backtracks was observed on all instances in the Gripper domain, with this attributed to the combination of the goal-directed heuristic and the action pruning resulting from the resource allocation meta variable guidance. This reduction in choice, of gripper arm in this case, removed some of the symmetry found in this domain. All instances in the Elevator domain were solved faster with the addition of meta variables, due only to faster consistency checking. With the choice of ordering of goals predetermined, and with no choice available in terms of (single) elevator resource, plan lengths and levels of backtracking were unchanged. A reduction in the amount of choice is observed in the larger instances from the TPP domain, with this achieved by pre-assigning particular resources. Although not leading to a reduction in the amount of backtracking, all instances were solved faster with the addition of the meta variables. Finally, the instances in the Airport domain all exhibit faster or unchanged run time when meta variables are used. With no action pruning, the amount of choice in these instances is not reduced, with the decreased run time being a consequence of faster inference, not a reduction in the need for it. On the larger instances in this domain, any time saving is reduced due to the number of actions to be checked at each level of a (large) solution matrix, with this dominating.

[1]In terms of the amount of backtracking carried out.

5.4 Chapter Summary

This chapter introduced the use of meta-CSP methods, first in relation to locking goal variables' values once these had been achieved by the CSP variable and value selection heuristic being used in this work. The further use of meta variables, for resource to task allocation, was then discussed. The results of adding each of these sets of meta variables to the CSP-encoded AI planning problems in the test sets were then presented and analysed.

Using meta variables for goal locking was shown to be useful, in terms of the thesis being investigated in this work. However, this was only the case where a good goal ordering could be found, and where there were clear and accessible dependencies between goal variables. This was reflected in the results, which showed a decrease in run time and number of backtracks in some domains, and no change or an increase in run time in others. Thus, using the goal locking meta variables as a method of transferring planning problem structural information to the CSP encoding and, further, attempting to enhance the amount of inference available in that CSP, was found to be heavily dependent on the structure of the given domain and problem instance. Hence, where there was little or no dependency between variables, locking one of these had little impact on the solution of the others. However, where a dependency relationship existed, and was strong, propagation of the additional constraints resulting from the goal locking did lead to increased inferences, and in turn to faster run times and, in certain cases, reduced amounts of backtracking.

Another way of using meta variables, in order to transfer planning problem structural information to a CSP reformulation of that problem, is via the allocation of resources to tasks. Here, a given problem's resources are first identified. Then, it is possible to implement a particular resource allocation strategy, with the meta variables encoding the relationship between tasks and allocated resources. A simple "minimise resources" policy was used as an example. The idea being investigated was that additional leverage may be gained during the CSP solution process. That is, by pre-allocating resources, the amount of choice (in terms of *suggested actions*) would be reduced, with a consequent reduction in the amount of consistency checking, and potentially backtracking, required to solve the problem. Again, the efficacy (reduction in amount of choice) was linked to the structure of the individual problem instances. However, 96% of the test cases exhibited a decrease in run time when resource allocation meta variables were used. Reduction in the amount of backtracking required was also linked to the structure of a given problem, with instances containing near identical (or symmetrical) resources (e.g. Gripper) benefiting most. Both positive and negative changes in plan length were recorded, with these again depending on each problem's individual structure.

The following chapter draws some conclusions, suggests areas where improvements might be made, and introduces potential areas in which future work may be carried out.

CHAPTER 6

CONCLUSIONS

6.1 Introduction

This chapter, the final one, re-examines the aims of this work, briefly reviewing the methods used and the results achieved. By revisiting the Statement of Thesis, the intention is to align the actual outcomes and results with the thesis proposed in Chapter 1. Potential improvements and alternative approaches are also considered, together with possible directions in which future work may be carried out. The Statement of Thesis is reproduced below for ease of reference:

"*It is possible to solve planning problems more efficiently by better exploiting the inherent constraint inference mechanisms within a given constraint solver. This can be achieved by using the structure of the original planning problem to guide the search for a solution to a CSP reformulation of that planning problem. Such a new, and automatically generated, heuristic may be shown to be effective on a range of domains and problem instances. Additionally, the use of meta-CSP techniques may further increase the amount of propagation achieved and hence provide an additional improvement in solution time. When used in tandem, these techniques will provide a performance increase on standard planning benchmark problems*".

6.2 Discussion of Thesis and Results

This section seeks to consider each part of the Statement of Thesis individually, with the intention of aligning the proposed thesis with the empirical results. Each part of the thesis will be considered in terms of: overall success, limitations, alternative approaches, and potential avenues for further investigation.

The Statement of Thesis (SoT) begins with the assertion that:
"*It is possible to solve planning problems more efficiently by better exploiting the inherent constraint inference mechanisms within a given constraint solver*".

Although general, and qualitative in nature, this statement is useful since it begs a number of questions that helped focus the study. These, in turn, can be used to help quantitatively measure the validity of the thesis; What is the definition of "more efficiently"? What are the "inherent constraint inference mechanisms" and, finally, which "constraint solver" is to be used?

Throughout this work, the performance of the CSP solving process has been measured consistently using run time, number of backtracks and, where appropriate, plan length. Using such recognised measures (e.g. IPC[1] and [164]) can be considered successful since it not only allows comparison with appropriate works in the literature (e.g. [150]), but also facilitates easy comparison of relative measurements (i.e. with and without a proposed improvement). One limitation of these metrics is found in Section 5.3. Here, "amount of choice" (or branching factor) manifest as the number of *suggested actions*, would perhaps be a useful additional measure.

The constraint mechanisms used in this work were described in Chapters 2 and 3, with the particular solver further discussed in Chapter 3. Since the aim of this work was to investigate the use of AI planning problem structural information during the CSP solution process, it could be argued that the details of the inherent constraint inference mechanisms are not relevant. Whilst this may be partly true, it was still necessary to consider these in enough detail in order to be able to determine and understand the baseline measures. That is, by determining the amount of propagation provided by the basic encoding of the existing constraints (planning actions, and initial state and goal state), in addition to achieving an understanding of the standard operation of the solver, it was possible to better understand how to add to this the planning specific guidance. It was then also possible to observe and measure the potential benefits (or otherwise) of these additional constraints and solving algorithms. Limitations of the initial direct encoding, and the associated levels of propagation, were addressed by employing extensional constraint descriptions. In turn, the success of this approach is somewhat limited by the absolute size of the encoding. An alternative encoding(s) (e.g. using MiniZinc[2] or Zinc [222]), potentially in combination with different propagation

[1]International Planning Competitions (IPC): http://ipc.icaps-conference.org/
[2]www.minizinc.org

behaviours, is one avenue for future investigation.

Further, in relation to the inherent CSP techniques and use of the chosen solver, ECL^{i}PS^e, a decision was taken to neither incorporate additional techniques (e.g. symmetry breaking, no-good learning) nor to not make use of any existing constraint-based AI planning system. That is, with the exception of the use of the *translator* subsystem of the FD planner [54] for conversion of PDDL to SAS+, this project was developed entirely by the author, with the decision to use ECL^{i}PS^e influenced by the author's previous experience of a similar system. It is, however, recognised that ECL^{i}PS^e is by no means the most efficient approach [223]. This is perhaps less critical in this work, since the metrics used are employed to measure differences between different techniques implemented on a common base system. This approach can be considered successful in terms of the outputs achieved. That is, the overall system was effective in successful completion of the experiments used to determine the validity of the thesis. However, since the absolute measures are not competitive with the state of the art, the use of ECL^{i}PS^e may be simultaneously considered a limiting factor, a position accepted by the author. Hence, in addition to an alternative encoding, a different solver (e.g. Minion [224]), perhaps encompassing alternative solving approaches, is one direction in which future work may proceed.

The SoT continues with the statement that:

"*This [solving planning problems more efficiently] can be achieved by using the structure of the original planning problem to guide the search for a solution to a CSP reformulation of that planning problem. Such a new, and automatically generated, heuristic may be shown to be effective on a range of domains and problem instances*".

Chapter 3 shows the successful use of structural information from the original planning problem, in the form of backdoor style variables, to improve the run time and reduce the number of backtracks required to find a solution to a range of planning problems. This chapter also shows that, by breaking up the problem using *phase constraints*, further structural information may be employed in order to improve CSP solver performance. Since the intention was not to discover these useful "stepping stone" variables, but only to determine the efficacy of using them in the CSP solution process, a previously populated solution matrix was the means used to supply these. This limitation reduced the number of test instances available, since each problem required a solution using the CSP solver, the result of which could then used to supply the test variables. An alternative approach would be to make use of landmark [56] type structural information, and then guide the CSP variable and value selection with this. Hence, any additional form of structural information, taken from the planning problem, used as guidance in the CSP reformulation is an area for potential further enquiry.

The approach taken in Chapter 4 uses the causal graph (CG) (Definition 11) to supply goal ordering information. These ordered goals are then used as the basis of a CSP variable and value selection heuristic algorithm, which incorporates the principles of problem subdivision and intermediate horizons highlighted in Chapter 3. With the reliance on goal ordering from the causal graph, there comes a limitation in the form of lost variable dependencies. Whilst some of the variable dependencies are maintained, some are lost due to cycle breaking in the graph. An alternative approach may make use of landmarks or use a goal ordering derived from another planner, perhaps one used to generate a seed plan length. Additional further work on the heuristic presented in Chapter 4 could focus on integration with a partial ordering system. That is, by using the goal-state goal achieving actions and tentative sub plans, it may be possible to generate or make use of intermediate partially ordered combinations of sub plans / sets of actions. Another direction might be the investigation of a parallel plan based system, in which the sub plans, subject to resource constraints, could be carried out in parallel.

The next part of the SoT states:
"*Additionally, the use of meta-CSP techniques may further increase the amount of propagation achieved and hence provide an additional improvement in solution time*".

Chapter 5 discussed two separate uses of meta variables, one for goal locking, the other for resource to task assignment. The goal locking meta variable approach was, in general terms, successful. However, particular domains, and individual instances within these, highlighted the connection between problem structure and level of benefit achieved by adding goal locking meta variables. The limitation here is that the amount of increased inference and propagation relies on both a good goal ordering and a well defined and accessible dependency relationship between the planning problem's CSP variables. Where these are not present, the goal locking gives little or no benefit, and in some cases is a significant overhead. Thus, an alternative approach, for possible further investigation, may be to use the variable dependency information to inform the choice of whether or not to employ goal locking. In this way, it could be possible to capitalise on this additional guidance only where it is most likely to be useful.

The use of meta variables for resource to task assignment, also presented in Chapter 5, was more widely beneficial. 83 from a set of 85 problem instances exhibited improved run time, with the remaining two instances having unchanged run time measures (due to their small size). Here, a potential limitation is that the particular resource allocation policy chosen, a simple minimise resources approach, relies on the structure of the problem allowing a solution to be found with the given allocation of resources. For this reason, the amount of backtracking can increase or decrease, with this also being the case in relation to the plan length. Alternative resource allocation policies could be implemented with, for example, one which makes use of the known structural information from the planning problem (via the DTGs (Definition 12)) allowing the most efficient use of resources. This, and other resource allocation schemes, could form the basis of future work in this area.

The SoT concludes with the statement that:
"*When used in tandem, these techniques will provide a performance increase on standard planning benchmark problems*".

From the results presented, the combined techniques do provide a performance increase across the range of problem instances tested. However, it is important to qualify this by considering the limitations discussed above. Overall, this work's thesis may be considered valid, with the techniques used generally appropriate and mostly successful. Whilst this work has been limited to STRIPS based AI planning, and a restricted set of test domains, future work in which, say, temporal constraints and a wider range of resource type were included, would necessarily require revision of many of the techniques used in this work, including better provision of goal ordering and variable dependency data, a more efficient CSP encoding, and an alternative CSP solving infrastructure.

6.3 Context and Final Conclusions

Whilst the results discussed throughout this work have served to validate the thesis proposed, this section places these results in a wider context. It does so by comparing this work's results to those from a number of CSP based planners found in the literature.

As discussed in Chapter 2, an important recent contribution in the area of constraint based AI planning is the planner, SeP [150]. Consideration of SeP's performance allows comparisons to be made with the results achieved in this work.

With a similar experimental setup, SeP may be compared to the work presented in the preceding chapters. Table 6.1 shows the run time and the plan length measures for both planners on those benchmark domains on which both have been tested.

On the lowest numbered instances (Table 6.1) of the benchmark domains, SeP often finds a solution faster than the heuristically guided (including locking and resource allocation meta variables) approach used in this work.

Domain / Instance	SeP RT(Secs)	MetaHeuristic RT(Secs)	SeP PL	MetaHeuristic PL
Driverlog 1	0.1	2.8	7	7
Driverlog 2	1017.6	10	19	26
Driverlog 3	11.6	5.8	12	14
Driverlog 4	-	13	-	26
Driverlog 5	-	28.6	-	25
Driverlog 6	83.9	23.3	11	14
Driverlog 7	-	38.3	-	19
Driverlog 8	-	13.4	-	27
Driverlog 9	-	107.9	-	30
Driverlog 10	-	97.6	-	24
Driverlog 11	-	155.9	-	27
Driverlog 12	-	469.3	-	46
Driverlog 13	-	685.7	-	36
Zeno 1	0.01	0.7	1	1
Zeno 2	0.06	2.3	6	6
Zeno 3	0.3	5.8	6	6
Zeno 4	0.9	17.7	8	8
Zeno 5	153.9	22	11	14
Zeno 6	530.4	42.6	11	15
Zeno 7	-	88.3	-	20
Zeno 8	-	254.8	-	12
Zeno 9	-	607.7	-	28
Zeno 10	-	703.1	-	33
Zeno 11	-	272.1	-	16
Zeno 12	-	961.6	-	27
TPP 1	0.01	0.08	5	5
TPP 2	0.02	0.4	8	10
TPP 3	0.16	1.1	11	15
TPP 4	2.1	2.3	14	20
TPP 5	-	14.1	-	31
TPP 6	-	375.5	-	43
TPP 7	-	1049	-	56
TPP 8	-	1786	-	66
Airport 1	-	0.5	-	8
Airport 2	-	0.6	-	9
Airport 3	8.8	1034	17	17
Airport 4	-	3	-	20
Airport 5	-	6.2	-	21
Airport 6	1736.3	-	41	-
Airport 7	-	-	-	-
Airport 8	-	-	-	-
Airport 9	-	-	-	-
Airport 10	-	4.2	-	18
Airport 11	-	8.3	-	21
Airport 12	1459	773	39	39
Airport 13	1548	-	37	-

Table 6.1: Comparison of results (run time (RT) and plan length (PL)) achieved on benchmark problems for SeP and the heuristically guided CSP planner (with meta variables) used in this work ("-" indicates no solution found within 1800 seconds).

In these cases (e.g. Driverlog 1, Zeno 1 to 4, TPP 1 to 4), the overhead of the heuristic processing and additional constraints dominates. However, the heuristic technique solves many more instances overall, across all domains. Further, where each planner finds a solution for the higher numbered instances, the heuristic is significantly faster (e.g. Driverlog 6, Zeno 5 and 6, Airport 12). In terms of plan length, due to the single-goal focus of the heuristic (as discussed previously), the expectation would be for SeP to produce shorter plans. This is the case on a number of problems (e.g. Driverlog 2, 3, and 6, Zeno 5 and 6, and TPP 2 to 4), but not on others (e.g. Driverlog 1, Zeno 1 to 4, TPP 1, and all instances in the Airport domain), where the planners find exactly the same length of plan.

From the results in Table 6.1, the benefits of using the original planning problem's structural information, to guide the solution process in the CSP reformulation of that problem, is clear. Instead of relying on a standard CSP variable / value choice strategy[1], the approach used in this work branches on planning problem specific variables (and particular values of these). Together with the additional propagation and pruning benefits resulting from the inclusion of the meta variables, the planner used in this work is consistently able to find solutions where SeP does not.

Considering now two further constraint based planners (CPT [177] and Constance [167]), Table 6.2 shows the number of instances solved by these planners compared to the number solved by this work's heuristic-based CSP planner. As noted previously (Chapter 2), CPT uses a modified form of the heuristic, h_m, with a partial-order causal link (POCL) style branching scheme, giving powerful pruning rules for reduction of the search space. Constance employs macros and substate group dominance constraints.

Domain (1st 10 instances)	CPT No. solved	Constance No. solved	MetaHeuristic No. solved
Driverlog	3	6	10
Zeno	4	8	10
Elevator	10	10	10
Gripper	2	2	10
TPP	4	5	8
Airport	6	7	6
Blocks	10	10	5

Table 6.2: Comparison of results (no. solved) achieved on benchmark domains for CPT, Constance, and the heuristically guided CSP planner (with meta variables) used in this work.

[1] As is the case in SeP.

In the Driverlog, Zeno, Gripper, and TPP domains (Table 6.2), this work's CSP planner solves all of the first ten instances, more than either of the other two planners. In the Elevator domain, all three solve all instances, and in the Airport domain, the heuristically guided planner matches the performance of CPT, with both beaten by Constance. Due to the limitations of the goal ordering and variable dependency information in the Blocks domain (Chapters 4 and 5), this work's planner performs less well here. Of particular interest, however, is the success of the heuristic guidance, together with the pruning due to the (resource) meta variables, in the Gripper domain, one in which various types of symmetry appear to cause problems for the other planners.

A recent, and continuing, trend in CSP planning has been the development of parallel planning systems. For example, PaP [165], PaP2 [225] and TCPP [170] all use constraint based systems in order to solve AI planning problems, with the resulting sets of actions forming parallel plans, as opposed to sequential plans, which is the case with this work's planner, and all of the CSP based planners discussed above. Noting that parallel planners are not directly comparable (e.g. in terms of plan length), it is still useful to consider the relative performance of such systems, especially since one of these, TCPP, makes use of the given planning problem's structural information, in the form of the DTGs from the SAS+ encoding. Table 6.3 shows the number of instances, from a range of benchmark domains, that are solved by PaP, PaP2, TCPP, and this work's planner.

Domain	Instance Count	PaP No. solved	PaP2 No. solved	TCPP No. solved	MetaHeuristic No. solved
Driverlog	15	12	13	13	13
Zeno	15	11	12	12	12
Gripper	10	-	2	2	10
TPP	15	8	8	10	8
Airport	15	6	14	14	8
Blocks	35	7	13	32	5

Table 6.3: Comparison of results (no. solved) achieved on benchmark domains for PaP, PaP2, TCPP, and the heuristically guided CSP planner (with meta variables) used in this work.

Considering Table 6.3, the heuristically guided planner with meta variables matches the performance of the state of the art parallel planners on the Driverlog and Zeno problem instances, but lags in the TPP, Airport, and Blocks domains. The size of the CSP encoding starts to affect the heuristic planner on the larger TPP instances, whilst in the Airport domain additional instances were solved, but were excluded since the run time measure exceeded the IPC limit. Again, poor goal ordering, and the lack of good variable dependency data, hampered the heuristic's performance on the Blocks problems. However, the goal-directed heuristic, together with the additional action pruning provided by the resource allocation meta variables, allowed all problem instances from the Gripper domain to solved.

This section has placed into context the work resulting from the investigation of the thesis introduced in Chapter 1. That is, this work's results have been compared to those from a range of state of the art CSP planners. This discussion showed that the heuristically guided planner with meta variables is competitive, or dominant, in a number of domains. The heuristic planner does not, however, match the performance of the best planners on all. Likewise, none of the other CSP planners dominates on all domains. Hence, it is possible that a combination of techniques, including those presented here, may further improve the efficiency of CSP based AI planning.

This combination of CSP techniques, and the integration of such methods with other planning approaches, may also be seen to be benefiting AI planning more generally. For example, two recent pieces of work ([226] and [227]), whilst not directly CSP-based, do overlap conceptually with the techniques used to pursue the thesis described above.

The first of these [226], in common with this work, addresses the fact that structural information from AI planning problems is obscured or lost, even when using more expressive planning languages. By using an alternative logical interpretation of the propositional layers in the RPG (section 2.2.6.1), the authors retain the assumption that the set of state variable values increases monotonically, but not the assumption that all conjunctions of such values are valid, with this captured in a *Constrained Relaxed Planning Graph* (CRPG). Approximating the CRPG, by using a standard CSP, it is shown that, by incorporating extensional encodings (section 3.3), standard CSP consistency techniques and global constraints (section 2.3.2.1), improved heuristics (h_{max} and h_{FF}) can be obtained. Thus, as with the techniques used to pursue the thesis in this project, the approach used [226] capitalises on better use of local constraint propagation to obtain better overall performance.

The second recent technique [227] employs Logic-Based Benders Decomposition (LBBD) [228], which allows a planning problem to be decomposed into a master mixed integer problem (MIP) and at least one sub problem, with the two sharing certain variables. The master problem represents a relaxation of the original problem, whilst the sub problem(s) is used to check for flaws in that relaxation. It is possible to draw direct comparisons between the methods used to pursue the thesis in this work and the techniques used by the authors [227]. For example, the relaxation imposed as a result of using the CG based goal-ordering heuristic here is analogous to their use of an operator count (bounds literals) heuristic. In this thesis, the sub problems effectively "check for flaws" in the heuristic guidance provided by the CG based goal-ordering. That is, if a sub problem is unsolvable, backtracking will occur, and the plan length will be extended, ultimately leading to discovery of problems with the goal ordering heuristic guidance. Extending this to include the meta variables for resource allocation, it can be observed that such shared meta variables can further reduce the solution space, with this being comparable to the cuts in the Benders decomposition. Further, both approaches allow for the use of existing solver technologies (i.e. MIP and CSP), an obvious benefit for AI planning more generally.

In conclusion, the thesis in this work has shown the efficacy of using the structural information, contained within AI planning problems, to enhance the solution process of the CSP reformulation of such problems. The limitations, for example those due to the increasing size of the extensional constraint encoding, could be addressed by incorporating recent advances in this area found in parallel planning (TCPP [170]). Hence, together with those recommendations discussed in Section 6.2, a realistic proposal for future work would be the combination of such a compact representation [170] with the heuristic procedure and meta variable techniques described here. This combination, used in concert with a more efficient solver (e.g. [222]), one that can provide enhanced propagation [229] by capitalising on a particular extensional representation, could allow CSP based AI planning systems to better compete with state of the art heuristic planners.

APPENDIX A

PDDL PLANNING DOMAIN LINKS

PDDL domain and problem files used in this work may found here:

Driverlog, Blocksworld, Elevators & Zeno:

`http://agents.fel.cvut.cz/codmap/`

Mystery & Gripper:

`http://ipc98.icaps-conference.org/`

Airport, Pipesworld & PSR:

`http://idm-lab.org/wiki/icaps/ipc2004/deterministic/`

Schedule:

`http://www.cs.colostate.edu/meps/aips2000data/`
`2000-tests/schedule/domain.pddl`

TPP:

`http://idm-lab.org/wiki/icaps/ipc2006/deterministic/`

BIBLIOGRAPHY

[1] S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards, *Artificial Intelligence - A Modern Approach.* Prentice-Hall, Inc., 3rd ed., 2003.

[2] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," in *Readings in Planning* (J. Allen, J. Hendler, and A. Tate, eds.), pp. 88–97, San Mateo, CA: Kaufmann, 1990.

[3] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice.* San Francisco, CA, USA: Morgan Kaufmann, 2004.

[4] M. Genesereth and N. Nilsson, *Logical Foundations of Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann, 1987.

[5] K. Erol, "Complexity, decidability and undecidability results for domain-independent planning," *Artificial Intelligence*, vol. 76, pp. 75–88, 1995.

[6] J. Pearl, *Heuristics: intelligent search strategies for computer problem solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[7] J. Pearl, "Heuristic search theory: survey of recent results," in *Proceedings of the 7th international joint conference on Artificial intelligence - Volume 1*, (San Francisco, CA, USA), pp. 554–562, Morgan Kaufmann Publishers Inc., 1981.

[8] A. Garci-Olaya, S. Jimenez, and C. Linares Lopez, "International conference on automated planning and scheduling, results of the seventh international planning competition." `http://www.plg.inf.uc3m.es/ipc2011-deterministic/Results?action=AttachFile&do=view&target=ipc2011-booklet.pdf`, 2011.

[9] R. Dechter, *Constraint processing.* Elsevier Morgan Kaufmann, 2003.

[10] E. P. K. Tsang, *Foundations of constraint satisfaction.* Computation in cognitive science, Academic Press, 1993.

[11] F. Rossi, P. van Beek, and T. Walsh, eds., *Handbook of Constraint Programming*. Elsevier Science Inc. New York, NY, USA, first ed., 2006.

[12] T. Dean and M. Wellman, *Planning and control*. Morgan Kaufmann series in representation and reasoning, Morgan Kaufmann, San Francisco, CA, USA, 1991.

[13] A. Newell and H. Simon, "GPS, a program that simulates human thought," in *Computers and Thought* (E. Feigenbaum and J. Feldman, eds.), pp. 279–293, McGraw-Hill, New York, 1963.

[14] C. Green, "Theorem-proving by resolution as a basis for question-answering systems," in *Machine Intelligence* (B. Meltzer and D. Michie, eds.), vol. 4, ch. 11, pp. 183–205, Edinburgh University Press, 1969.

[15] J. McCarthy and P. J. Hayes, "Some philosophical problems from the standpoint of artificial intelligence," in *Machine Intelligence 4* (B. Meltzer, D. Michie, and M. Swann, eds.), pp. 463–502, Edinburgh, Scotland: Edinburgh University Press, 1969.

[16] V. Lifschitz, "On the semantics of strips," in *Readings in Planning* (J. Allen, J. Hendler, and A. Tate, eds.), pp. 523–530, San Mateo, CA: Kaufmann, 1990.

[17] N. J. Nilsson, *Principles of artificial intelligence*. Morgan Kaufmann, San Francisco, CA, USA, 1980.

[18] H. J. Levesque and R. J. Brachman, *A fundamental tradeoff in knowledge representation and reasoning*. Readings in Knowledge Representation, Morgan Kaufmann, 1985.

[19] D. Cohen, *Computability and logic*. Ellis Horwood series in mathematics and its applications, E. Horwood, 1987.

[20] C. Green, "Application of theorem proving to problem solving," in *Proceedings of the First International Joint Conference on Artificial Intelligence (IJCAI-69)*, pp. 219–239, Morgan Kaufmann, 1969.

[21] E. P. D. Pednault, "Adl: Exploring the middle ground between strips and the situation calculus," in *Proceedings of the 1st International Conference on Principles of Knowledge Representation and Reasoning*, pp. 324–332, 1989.

[22] E. P. D. Pednault, “Adl and the state-transition model of action,” *Journal of Logic and Computation*, vol. 4, no. 5, pp. 467–512, 1994.

[23] C. Knoblock, A. Barrett, D. Christianson, M. Friedman, C. Kwok, K. Golden, S. Penberthy, D. E. Smith, Y. Sun, and D. Weld, “Pddl: The planning domain definition language,” *AIPS98 planning committee*, vol. 78, no. 4, pp. 1–27, 1998.

[24] D. Long, J. Koehler, M. Brenner, J. Hoffmann, C. R. Anderson, D. S. Weld, and D. E. Smith, “The aips-98,” *AI Magazine*, vol. 21, no. 2, pp. 13–34, 2000.

[25] M. Fox and D. Long, “Pddl2.1: An extension to pddl for expressing temporal planning domains,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, pp. 61–124, 2003.

[26] S. Edelkamp and J. Hoffmann, “Pddl 2.2: The language for the classical part of the 4th international planning competition,” in *Proceedings of the 4th International Planning Competition (IPC.04), at ICAPS.04*.

[27] A. Gerevini and D. Long, “Plan constraints and preferences in PDDL3,” in *ICAPS Workshop on Soft Constraints and Preferences in Planning*, 2006.

[28] M. Fox and D. Long, “Modelling mixed discrete-continuous domains for planning,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 27, pp. 235–297, 2006.

[29] C. Bäckström and B. Nebel, “Complexity results for sas+ planning,” *Computational Intelligence*, vol. 11, pp. 625–656, 1995.

[30] E. D. Sacerdoti, “The nonlinear nature of plans,” in *Proceedings of the 4th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI’75, (San Francisco, CA, USA), pp. 206–214, Morgan Kaufmann Publishers Inc., 1975.

[31] A. Tate, “Generating project networks,” in *Proceedings of IJCAI-77*, pp. 888–893, 1977.

[32] G. J. Sussman, “A computational model of skill acquisition,” tech. rep., Cambridge, MA, USA, 1973.

[33] D. S. Weld, “An introduction to least commitment planning,” *AI Magazine*, vol. 15, pp. 27–61, 1994.

[34] D. Chapman, “Planning for conjunctive goals,” *Artificial Intelligence*, vol. 32, pp. 333–377, 1987.

[35] J. S. Penberthy and D. S. Weld, *UCPOP: A Sound, Complete, Partial Order Planner for ADL*, pp. 103–114. Morgan Kaufmann, 1992.

[36] M. Helmert, “The Fast Downward Planning System,” *Journal of Artificial Intelligence Research*, vol. 26, pp. 191 – 246, 2006.

[37] P. Hart, N. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths,” *IEEE Transactions on Systems, Science, and Cybernetics*, vol. SSC-4, no. 2, pp. 100–107, 1968.

[38] J. Hoffmann and B. Nebel, “The ff planning system: Fast plan generation through heuristic search,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 14, pp. 253–302, 2001.

[39] J. Hoffmann, “Where ’ignoring delete lists’ works: Local search topology in planning benchmarks,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 24, pp. 685–758, 2005.

[40] B. Bonet and H. Geffner, “Planning as heuristic search,” *Artificial Intelligence*, vol. 129, pp. 5–33, 2001.

[41] S. Edelkamp, “Symbolic pattern databases in heuristic search planning,” in *AIPS* (M. Ghallab, J. Hertzberg, and P. Traverso, eds.), pp. 274–283, AAAI, 2002.

[42] P. Haslum and H. Geffner, “Admissible heuristics for optimal planning,” in Chien *et al.* [230], pp. 140–149.

[43] E. Karpas and C. Domshlak, “Cost-optimal planning with landmarks,” in Boutilier [231], pp. 1728–1733.

[44] D. V. McDermott, “A heuristic estimator for means-ends analysis in planning,” in *AIPS* (B. Drabble, ed.), pp. 142–149, AAAI, 1996.

[45] B. Bonet, G. Loerincs, and H. Geffner, “A robust and fast action selection mechanism for planning,” in *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*, AAAI’97/IAAI’97, pp. 714–719, AAAI Press, 1997.

[46] C. Betz and M. Helmert, "Planning with h^+ in theory and practice," in *KI* (B. Mertsching, M. Hund, and M. Z. Aziz, eds.), vol. 5803 of *Lecture Notes in Computer Science*, pp. 9–16, Springer, 2009.

[47] A. Coles, M. Fox, D. Long, and A. Smith, "Additive-disjunctive heuristics for optimal planning," in Rintanen *et al.* [232], pp. 44–51.

[48] A. Blum and M. L. Furst, "Fast planning through planning graph analysis," in *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pp. 1636–1642, Morgan Kauffmann, 1995.

[49] V. Mirkis and C. Domshlak, "Cost-sharing approximations for h+," in *ICAPS* (M. S. Boddy, M. Fox, and S. Thiébaux, eds.), pp. 240–247, AAAI, 2007.

[50] E. Keyder and H. Geffner, "Heuristics for planning with action costs revisited.," in *ECAI* (M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, eds.), vol. 178 of *Frontiers in Artificial Intelligence and Applications*, pp. 588–592, IOS Press, 2008.

[51] E. Keyder and H. Geffner, "Heuristics for planning with action costs," in *CAEPIA* (D. Borrajo, L. A. Castillo, and J. M. Corchado, eds.), vol. 4788 of *Lecture Notes in Computer Science*, pp. 140–149, Springer, 2007.

[52] E. Keyder and H. Geffner, "Trees of shortest paths vs. steiner trees: Understanding and improving delete relaxation heuristics," in Boutilier [231], pp. 1734–1739.

[53] H. Prömel and A. Steger, *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity*. Advanced Lectures in Mathematics, Vieweg, 2002.

[54] M. Helmert and S. Richter, "Fast Downward - Making Use of Causal Dependencies in the Problem Representation," in *Proceedings of the International Planning Competition 2004*, pp. 41–43, 2004.

[55] M. Helmert and H. Geffner, "Unifying the causal graph and additive heuristics," in Rintanen *et al.* [232], pp. 140–147.

[56] J. Porteous and L. Sebastia, "Extracting and Ordering Landmarks for Planning," in *Proceedings of the Nineteenth Workshop of the UK PLANSIG*, pp. 161–174, 2000.

[57] J. Hoffmann, J. Porteous, and L. Sebastia, "Ordered Landmarks in Planning," *Journal of Artificial Intelligence Research*, vol. 22, pp. 215–278, 2004.

[58] L. Sebastia, E. Onaindia, and E. Marzal, "Decomposition of planning problems," *AI Communications*, vol. 19, pp. 49–81, Jan. 2006.

[59] S. Richter, M. Helmert, and M. Westphal, "Landmarks Revisited," in *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, pp. 975–982, 2008.

[60] L. Zhu and R. Givan, "Landmark extraction via planning graph propagation," in *ICAPS 2003 Doctoral Consortium*, 2003.

[61] E. Keyder, S. Richter, and M. Helmert, "Sound and complete landmarks for and/or graphs," in *ECAI* (H. Coelho, R. Studer, and M. Wooldridge, eds.), vol. 215 of *Frontiers in Artificial Intelligence and Applications*, pp. 335–340, IOS Press, 2010.

[62] M. Helmert, "LM-Cut: Optimal Planning with the Landmark-Cut Heuristic," in *Seventh International Planning Competition (IPC 2011), Deterministic Part*, pp. 103–105, 2011.

[63] M. Helmert and C. Domshlak, "Landmarks, critical paths and abstractions: What's the difference anyway?," in *ICAPS* (A. Gerevini, A. E. Howe, A. Cesta, and I. Refanidis, eds.), AAAI, 2009.

[64] X. Nguyen and S. Kambhampati, "Reviving partial order planning," in *Proceedings of the 17th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'01, (San Francisco, CA, USA), pp. 459–464, Morgan Kaufmann Publishers Inc., 2001.

[65] D. Bryce and S. Kambhampati, "A Tutorial on Planning Graph Based Reachability Heuristics," *AI Magazine*, vol. 28, no. 1, pp. 47–83, 2007.

[66] H. L. S. Younes and R. G. Simmons, "Vhpop: Versatile heuristic partial order planner," *Journal of Artificial Intelligence Research (JAIR)*, vol. 20, pp. 405–430, 2003.

[67] B. Ridder and D. Long, "Integrating landmarks in partial order planners," in *ICAPS 2010 Doctoral Consortium*, 2010.

[68] M. Helmert and G. Röger, "How good is almost perfect?," in *AAAI* (D. Fox and C. P. Gomes, eds.), pp. 944–949, AAAI Press, 2008.

[69] M. Fox and D. Long, "The Detection and Exploitation of Symmetry in Planning Problems," in *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 956–961, 1999.

[70] K. Erol, J. A. Hendler, and D. S. Nau, "Umcp: A sound and complete procedure for hierarchical task-network planning," in *AIPS* (K. J. Hammond, ed.), pp. 249–254, AAAI, 1994.

[71] F. Bacchus and F. Kabanza, "Using temporal logics to express search control knowledge for planning," *Artificial Intelligence*, vol. 116, no. 1-2, pp. 123–191, 2000.

[72] J. Levine and D. Humphreys, "Learning action strategies for planning domains using genetic programming," in *Proceedings of the 2003 international conference on applications of evolutionary computing*, EvoWorkshops'03, (Berlin, Heidelberg), pp. 684–695, Springer-Verlag, 2003.

[73] D. S. Nau, Y. Cao, A. Lotem, and H. Muñoz-Avila, "Shop: Simple hierarchical ordered planner," in *IJCAI* (T. Dean, ed.), pp. 968–975, Morgan Kaufmann, 1999.

[74] D. Long and M. Fox, "Automatic synthesis and use of generic types in planning," in Chien *et al.* [230], pp. 196–205.

[75] E. C. Freuder, "In pursuit of the holy grail," *Constraints*, vol. 2, no. 1, pp. 57–61, 1997.

[76] U. Montanari, "Networks of constraints: Fundamental properties and applications to picture processing," *Information Sciences*, vol. 7, pp. 95–132, 1974.

[77] M. S. Fox, *Constraint-directed search: a case study of job-shop scheduling*. PhD thesis, Pittsburgh, PA, USA, 1983. AAI8406442.

[78] R. Zabih and D. A. McAllester, "A rearrangement search strategy for determining propositional satisfiability," in *AAAI* (H. E. Shrobe, T. M. Mitchell, and R. G. Smith, eds.), pp. 155–160, AAAI Press / The MIT Press, 1988.

[79] H. Simonis, "Constraint applications (online repository)." `http://hsimonis.wordpress.com/`, Nov. 2011.

[80] K. R. Apt, *Principles of constraint programming*. Cambridge University Press, 2003.

[81] F. Rossi, C. Petrie, and V. Dhar, "On the equivalence of constraint satisfaction problems," in *Proceedings of the 9th European Conference on Artificial Intelligence*, pp. 550–556, 1990.

[82] B. M. Smith, "Modelling for Constraint Programming." First International Summer School on Constraint Programming, 2005.

[83] E. C. Freuder, "Modeling: The final frontier," in *The First International Conference and Exhibition on The Practical Application of Constraint Technologies and Logic Programming (PACLP 99)*, (London, UK), 1999.

[84] J.-F. Puget, "Constraint programming next challenge: Simplicity of use," in Wallace [233], pp. 5–8.

[85] R. Bartak, "Effective Modeling with Constraints," in *Lecture Notes in Computer Science, 2005, Volume 3392/2005, 149-165, DOI: 10.1007/11415763_10* (D. et al Seipel, ed.), pp. 149 – 165, Springer-Verlag, 2005.

[86] Y. C. Law and J. H.-M. Lee, "Model induction: A new source of csp model redundancy," in *AAAI/IAAI* (R. Dechter and R. S. Sutton, eds.), pp. 54–60, AAAI Press / The MIT Press, 2002.

[87] B. M. W. Cheng, K. M. F. Choi, J. H. M. Lee, and J. C. K. Wu, "Increasing Constraint Propagation by Redundant Modeling : an Experience Report," *Constraints*, vol. 4, no. 2, pp. 167–192, 1999.

[88] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace, "Eclipse: A tutorial introduction." `http://eclipseclp.org/doc/tutorial.pdf`, Mar. 2011.

[89] R. A. Kowalski, "Algorithm = logic + control," *Communications of the ACM*, vol. 22, no. 7, pp. 424–436, 1979.

[90] A. Mackworth, "Consistency in Networks of Relations," *Artificial Intelligence*, vol. 8, pp. 77–98, 1977.

[91] C. Lecoutre, *Constraint Networks: Techniques and Algorithms*. Wiley-IEEE Press, 2009.

[92] Y. Zhang and R. H. C. Yap, "Making ac-3 an optimal algorithm," in Nebel [234], pp. 316–321.

[93] C. Bessière and J.-C. Régin, "Refining the basic constraint propagation algorithm," in Nebel [234], pp. 309–315.

[94] R. Mohr and T. C. Henderson, "Arc and path consistency revisited," *Artificial Intelligence*, vol. 28, no. 2, pp. 225–233, 1986.

[95] Y. Deville and P. V. Hentenryck, "An efficient arc consistency algorithm for a class of csp problems," in *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pp. 325–330, 1991.

[96] C. Bessière, "Arc-consistency and arc-consistency again," *Artificial Intelligence*, vol. 65, no. 1, pp. 179–190, 1994.

[97] C. Bessière, E. C. Freuder, and J.-C. Régin, "Using inference to reduce arc consistency computation," in *IJCAI (1)* [235], pp. 592–599.

[98] C. Lecoutre, F. Boussemart, and F. Hemery, "Exploiting multidirectionality in coarse-grained arc consistency algorithms," in *CP* (F. Rossi, ed.), vol. 2833 of *Lecture Notes in Computer Science*, pp. 480–494, Springer, 2003.

[99] R. J. Wallace and E. C. Freuder, "Ordering heuristics for arc consistency algorithms," in *Proceedings of 9th Canadian Conference on Artificial Intelligence*, pp. 163–169, 1992.

[100] C. Bessière and J.-C. Régin, "Arc consistency for general constraint networks: Preliminary results," in *IJCAI (1)*, pp. 398–404, Morgan Kaufmann, 1997.

[101] J.-C. Régin, "A filtering algorithm for constraints of difference in csps," in *AAAI* (B. Hayes-Roth and R. E. Korf, eds.), pp. 362–367, AAAI Press / The MIT Press, 1994.

[102] C.-C. Han and C.-H. Lee, "Comments on mohr and henderson's path consistency algorithm," *Artificial Intelligence*, vol. 36, pp. 125–130, 1988.

[103] M. Singh, "Path consistency revisited," in *Proceedings of the Seventh International Conference on Tools with Artificial Intelligence*, TAI '95, (Washington, DC, USA), pp. 318–, IEEE Computer Society, 1995.

[104] A. Chmeiss and P. Jegou, "On path consistency and partial forms," in *Proceedings of the Congress-AFCET RFIA-96*, (Rennes, France), pp. 212–219, 1996.

[105] A. Chmeiss and P. Jégou, "Path-consistency: When space misses time," in Clancey and Weld [236], pp. 196–201.

[106] A. Chmeiss and P. Jégou, "Efficient path-consistency propagation," *International Journal on Artificial Intelligence Tools*, vol. 7, no. 2, pp. 121–142, 1998.

[107] E. C. Freuder, "Synthesizing constraint expressions," *Communications of the ACM*, vol. 21, no. 11, pp. 958–966, 1978.

[108] V. Kumar, "Algorithms for Constraint- Satisfaction Problems: A Survey," *AI Magazine*, pp. 32–44, 1992.

[109] J. R. Bitner and E. M. Reingold, "Backtracking programming techniques," *Communications of the ACM*, vol. 18, no. 11, pp. 651–656, 1975.

[110] A. K. Mackworth and E. C. Freuder, "The complexity of some polynomial network consistency algorithms for constraint satisfaction problems," *Artificial Intelligence.*, vol. 25, no. 1, pp. 65–74, 1985.

[111] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, 1979.

[112] P. Prosser, "Hybrid algorithms for the constraint satisfaction problem," *Computational Intelligence*, vol. 9, pp. 268–299, 1993.

[113] R. M. Haralick and G. L. Elliott, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, no. 3, pp. 263–313, 1980.

[114] R. Dechter, "Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition," *Artificial Intelligence*, vol. 41, no. 3, pp. 273–312, 1990.

[115] B. A. Nadel, "Search in artificial intelligence," ch. Tree search and ARC consistency in constraint satisfaction algorithms, pp. 287–342, London, UK, UK: Springer-Verlag, 1988.

[116] D. Sabin and E. C. Freuder, "Contradicting conventional wisdom in constraint satisfaction," in *ECAI*, pp. 125–129, 1994.

[117] B. A. Nadel, "Constraint satisfaction algorithms," *Computational Intelligence*, vol. 5, no. 3, pp. 188–224, 1989.

[118] D. Frost and R. Dechter, "Look-ahead value ordering for constraint satisfaction problems," in *IJCAI (1)* [235], pp. 572–578.

[119] D. Frost and R. Dechter, "Looking at full looking ahead," in Freuder [237], pp. 539–540.

[120] E. C. Freuder, "A Sufficient Condition for Backtrack Free Search," *Journal of the Association for Computing Machinery*, vol. 29, no. 1, pp. 24 – 32, 1982.

[121] G. Gottlob, N. Leone, and F. Scarcello, "A comparison of structural csp decomposition methods," *Artificial Intelligence*, vol. 124, no. 2, pp. 243–282, 2000.

[122] M. C. Cooper, D. A. Cohen, and P. Jeavons, "Characterising tractable constraints," *Artificial Intelligence*, vol. 65, no. 2, pp. 347–361, 1994.

[123] M. C. Cooper, P. G. Jeavons, and A. Z. Salamon, "Hybrid tractable csps which generalize tree structure," in *ECAI* (M. Ghallab, C. D. Spyropoulos, N. Fakotakis, and N. M. Avouris, eds.), vol. 178 of *Frontiers in Artificial Intelligence and Applications*, pp. 530–534, IOS Press, 2008.

[124] A. E. Mouelhi, P. Jégou, C. Terrioux, and B. Zanuttini, "On the efficiency of backtracking algorithms for binary constraint satisfaction problems," in *ISAIM*, International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, Florida, USA, January 9-11, 2012, 2012.

[125] I. P. Gent, E. MacIntyre, P. Prosser, B. M. Smith, and T. Walsh, "An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem," in Freuder [237], pp. 179–193.

[126] S. W. Golomb and L. D. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, pp. 516–524, Oct. 1965.

[127] B. M. Smith and S. A. Grant, "Trying harder to fail first," in *ECAI*, pp. 249–253, 1998.

[128] J. C. Beck, P. Prosser, and R. J. Wallace, “Trying again to fail-first,” in *CSCLP* (B. Faltings, A. Petcu, F. Fages, and F. Rossi, eds.), vol. 3419 of *Lecture Notes in Computer Science*, pp. 41–55, Springer, 2004.

[129] B. Nudel, “Consistent-labeling problems and their algorithms,” in *AAAI* (D. L. Waltz, ed.), pp. 128–132, AAAI Press, 1982.

[130] R. Wallace, “Factor analytic studies of csp heuristics,” in *CP* (P. van Beek, ed.), vol. 3709 of *Lecture Notes in Computer Science*, pp. 712–726, Springer, 2005.

[131] D. Brélaz, “New methods to color the vertices of a graph,” *Communications of the ACM*, vol. 22, pp. 251–256, 1979.

[132] C. Bessière and J.-C. Régin, “Mac and combined heuristics: Two reasons to forsake fc (and cbj?) on hard problems,” in Freuder [237], pp. 61–75.

[133] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, “Boosting systematic search by weighting constraints,” in *ECAI* (R. L. de Mántaras and L. Saitta, eds.), pp. 146–150, IOS Press, 2004.

[134] I. P. Gent, E. MacIntyre, P. Prosser, and T. Walsh, “The constrainedness of search,” in Clancey and Weld [236], pp. 246–252.

[135] P. A. Geelen, “Dual viewpoint heuristics for binary constraint satisfaction problems,” in *ECAI*, pp. 31–35, 1992.

[136] B. Hnich and T. Walsh, “Why channel? multiple viewpoints for branching heuristics,” in *Proceedings of the 2nd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems: Towards Systematisation and Automation. CP03*, 2003.

[137] P. Refalo, “Impact-based search strategies for constraint programming,” in Wallace [233], pp. 557–571.

[138] R. Zabih, “Some applications of graph bandwidth to constraint satisfaction problems,” in *Proceedings of the eighth National conference on Artificial intelligence - Volume 1*, AAAI’90, pp. 46–51, AAAI Press, 1990.

[139] E. C. Freuder and M. J. Quinn, “Taking advantage of stable sets of variables in constraint satisfaction problems,” in *IJCAI* (A. K. Joshi, ed.), pp. 1076–1078, Morgan Kaufmann, 1985.

[140] J. Huang and A. Darwiche, “A structure-based variable ordering heuristic for sat,” in *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI.03)*, 2003.

[141] W. Li and P. van Beek, “Guiding real-world sat solving with dynamic hypergraph separator decomposition,” in *ICTAI*, pp. 542–548, IEEE Computer Society, 2004.

[142] R. Dechter and J. Pearl, “Network-based heuristics for constraint-satisfaction problems,” *Artificial Intelligence*, vol. 34, no. 1, pp. 1–38, 1987.

[143] R. Dechter, *Constraint Networks (In Encyclopedia of A.I.).* No. 2nd, Wiley and Sons, 2 ed., 1992.

[144] M. Ginsberg, M. Frank, M. Halpin, and M. Torrance, “Search lessons learned from crossword puzzles,” in *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 210–215, 1990.

[145] A. Meisels, S. E. Shimony, and G. Solotorevsky, “Bayes networks for estimating the number of solutions to a csp,” in *Proceedings of the 14th National Conference on AI*, pp. 179–184, 1997.

[146] M. Vernooy and W. S. Havens, “An examination of probabilistic value-ordering heuristics,” in *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence: Advanced Topics in Artificial Intelligence*, AI ’99, (London, UK, UK), pp. 340–352, Springer-Verlag, 1999.

[147] K. Kask, R. Dechter, and V. Gogate, “Counting-based look-ahead schemes for constraint satisfaction,” in Wallace [233], pp. 317–331.

[148] B. M. Smith, “A tutorial on constraint programming,” tech. rep., University of Leeds, 1995.

[149] H. Kautz and B. Selman, “Pushing the envelope: planning, propositional logic, and stochastic search,” in *Proceedings of the 13th national conference on Artificial intelligence - Volume 2*, AAAI’96, pp. 1194–1201, AAAI Press, 1996.

[150] R. Bartak and D. Toropila, “Revisiting Constraint Models for Planning Problems,” in *ISMIS 2009, LNAI 5722* (J. E. A. Raunch, ed.), pp. 582–591, Berlin Heidelberg: Springer-Verlag, 2009.

[151] R. Bartak, M. Salido, and F. Rossi, “Constraint satisfaction techniques in planning and scheduling,” *Journal of Intelligent Manufacturing*, vol. 21, pp. 5–15, 2010.

[152] A. Nareyek, E. C. Freuder, R. Fourer, E. Giunchiglia, R. P. Goldman, H. Kautz, J. Rintanen, and A. Tate, “Constraints and AI Planning,” *IEEE Intelligent Systems*, vol. 20, no. 2, pp. 62–72, 2005.

[153] C. Pralet and G. Verfaillie, “Constraint Networks on Timelines for Planning and Scheduling,” *Journées Francophones de Planification, Décision et Apprentissage pour la conduite de systèmes (JFPDA 2009)*, 2009.

[154] M. Stefik, “Planning with Constraints,” *Artificial Intelligence (An International Journal)*, vol. 16, pp. 111–140, 1981.

[155] D. Joslin and M. E. Pollack, “Passive and Active Decision Postponement in Plan Generation,” in *Proceedings of the Third European Conference on Planning*, pp. 1–15, 1995.

[156] R. P. Goldman, K. Z. Haigh, D. J. Musliner, and M. Pelican, “MACBeth: A Multi-Agent Constraint-Based Planner,” in *Working Notes of the AAAI Workshop on Constraints and AI Planning*, (Austin, Texas), 2000.

[157] H. A. Kautz and B. Selman, “Planning as Satisfiability,” in *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI’92)*, pp. 359–363, 1992.

[158] P. van Beek and X. Chen, “CPlan: A Constraint Programming Approach to Planning,” *AAAI*, vol. 99, pp. 585–590, 1999.

[159] S. Kambhampati, “Improving graphplan’s search with ebl and ddb techniques,” in *Proc. IJCAI-99*, pp. 982–987, Morgan Kaufmann, 1999.

[160] S. Kambhampati, “Planning graph as a (dynamic) csp: Exploiting ebl, ddb and other csp search techniques in graphplan,” *Journal of Artificial Intelligence Research*, vol. 12, pp. 1–34, 2000.

[161] M. B. Do and S. Kambhampati, “Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP,” *Artificial Intelligence*, vol. 132, no. 2, pp. 151–182, 2001.

[162] S. Mittal and B. Falkenhainer, "Dynamic constraint satisfaction problems," in *AAAI* (H. E. Shrobe, T. G. Dietterich, and W. R. Swartout, eds.), pp. 25–32, AAAI Press / The MIT Press, 1990.

[163] A. Lopez and F. Bacchus, "Generalizing GraphPlan by Formulating Planning as a CSP," in *Proceedings of The International Joint Conference on Artificial Intelligence (IJCAI 03)*, pp. 954–960, 2003.

[164] R. Bartak and D. Toropila, "Reformulating Constraint Models for Classical Planning," in *Proceedings of the 21st International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pp. 525–530, AAAI Press, 2008.

[165] R. Bartak, "A Novel Constraint Model for Parallel Planning," in *Proceedings of the Twenty-Fourth International Florida Artificial Intelligence Research Society Conference, May 18-20, 2011, Palm Beach, Florida, USA* (R. C. Murray and P. M. McCarthy, eds.), AAAI Press, 2011.

[166] D. Long and M. Fox, "Plan permutation symmetries as a source of planner inefficiency," in *Proceedings of UK Workshop on Planning and Scheduling*, 2003.

[167] P. Gregory, M. Fox, and D. Long, "Constraint Based Planning with Composable Substate Graphs," in *Proceedings of the Nineteenth European Conference on Artificial Intelligence*, 2010.

[168] G. Verfaillie and C. Pralet, "How to Model Planning and Scheduling Problems using Timelines," in *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS'08)*, 2008.

[169] R. Huang, Y. Chen, and W. Zhang, "A novel transition based encoding scheme for planning as satisfiability," in *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010* (M. Fox and D. Poole, eds.), AAAI Press, 2010.

[170] N. G. Ghooshchi, M. Namazi, M. A. H. Newton, and A. Sattar, "Transition constraints for parallel planning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA.* (B. Bonet and S. Koenig, eds.), pp. 3268–3274, AAAI Press, 2015.

[171] A. Nareyek, "A.i. planning in a constraint programming framework," in *Communication-Based Systems* (G. Hommel, ed.), Kluwer Academic Publishers, 2000.

[172] A. Nareyek, “Structural Constraint Satisfaction,” in *Proceedings of the AAAI Workshop on Configuration*, AAAI Press, 1997.

[173] C. Pralet and G. Verfaillie, “Using Constraint Networks on Timelines to Model and Solve Planning and Scheduling Problems,” in *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS’08)*, no. ICAPS, pp. 272–280, 2008.

[174] A. Cesta and S. Fratini, “The timeline representation framework as a planning and scheduling software development environment,” in *Proceedings of the Twenty Seventh Workshop of the UK Planning and Scheduling Special Interest Group, Edinburgh, UK, December 11-12, 2008*, (Edinburgh, UK), 2008.

[175] J. Frank and A. Jonsson, “Constraint-based Attribute and Interval Planning,” *Constraints*, vol. 8, no. 4, pp. 339–364, 2003.

[176] J. Barreiro, M. Boyce, M. Do, J. Frank, M. Iatauro, T. Kichkaylo, P. Morris, J. Ong, E. Remolina, T. Smith, and D. Smith, “Europa: A platform for ai planning, scheduling, constraint programming, and optimization,” in *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012* (L. McCluskey, B. Williams, J. R. Silva, and B. Bonet, eds.), AAAI, 2012.

[177] V. Vidal and H. Geffner, “Branching and pruning: An optimal temporal pocl planner based on constraint programming,” in *AAAI* (D. L. McGuinness and G. Ferguson, eds.), pp. 570–577, AAAI Press / The MIT Press, 2004.

[178] V. Vidal and H. Geffner, “Branching and pruning: an optimal temporal pocl planner based on constraint programming,” *Artificial Intelligence*, vol. 170, no. 3, pp. 298–335, 2006.

[179] R. Williams, C. Gomes, and B. Selman, “Backdoors To Typical Case Complexity,” in *Proceedings of International Joint Conference on Artificial Intelligence*, pp. 1173–1178, 2003.

[180] J. Schimpf and K. Shen, “Eclipse - from lp to clp,” *CoRR*, vol. abs/1012.4240, 2010.

[181] T. L. Provost and M. Wallace, “Domain independent propagation,” in *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS)*, pp. 1004–1011, June 1992.

[182] T. L. Provost and M. Wallace, “Generalised constraint propagation over the clp scheme,” *Journal of Logic Programming*, vol. 16, pp. 319–359, July 1993.

[183] A. Rendl, I. Miguel, I. P. Gent, and P. Gregory, “Enhancing Constraint Models of Planning Problems by Common Subexpression Elimination,” in *Proceedings of Abstraction, Reformulation and Approximation, International Symposium* (J. C. Beck and V. Bulitko, eds.), no. 1, 2009.

[184] A. Lopez and F. Bacchus, “Generalizing graphplan by formulating planning as a CSP,” *Proceedings of IJCAI 2003*, vol. 03, 2003.

[185] M. Helmert, “On the complexity of planning in transportation domains,” in Cesta and Borrajo [238], pp. 349–360.

[186] J. Hoffmann, S. Edelkamp, S. Thiébaux, R. Englert, F. dos S. Liporace, and S. Trüg, “Engineering benchmarks for planning: the domains used in the deterministic part of ipc-4,” *Journal of Artificial Intelligence Research (JAIR)*, vol. 26, pp. 453–541, 2006.

[187] M. Fox and D. Long, “The Automatic Inference of State Invariants in TIM,” *Journal of Artificial Intelligence Research*, vol. 9, pp. 367–421, 1998.

[188] J. Porteous, D. Long, and M. Fox, “The identification and exploitation of almost symmetry in planning problems.,” in *Proceedings of the Twenty Third UK Planning and Scheduling Special Interest Group (PlanSIG 2004)*, 2004.

[189] A. Gerevini, L. Schubert, and U. Brescia, “Inferring State Constraints for Domain-Independent Planning,” in *Proceedings of the Tenth Conference on Artificial Intelligence*, pp. 905–912, 1998.

[190] U. Scholz, “Extracting State Constraints from PDDL-like Planning Domains,” in *Proceedings of AIPS Workshop*, pp. 43–48, 2000.

[191] D. Long and M. Fox, “Automatic Synthesis and use of Generic Types in Planning,” in *Proceedings of the Eighteenth Workshop of the UK PLANSIG*, 1999.

[192] T. L. McCluskey and J. M. Porteous, “On Extracting Goal Structure from Planning Domain Specifications,” in *Proceedings of the Fourteenth Workshop of the UK PLANSIG*, 2000.

[193] S. Richter and M. Westphal, “The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmark,” *JAIR*, vol. 39, pp. 127 – 177, 2010.

[194] C. P. Gomes, *Backdoors in Combinatorial Problems.* 2005.

[195] P. Gregory, D. Long, and M. Fox, “Backdoors in Planning and Scheduling Problems,” in *Proceedings of International Conference on Automated Planning and Scheduling*, 2006.

[196] R. Waldinger, “Achieving Several Goals Simultaneously,” *Machine Intelligence*, vol. 8, 1977.

[197] J. Irani, K.B. and Cheng, “Subgoal Ordering and Goal Augmentation for Heuristic Problem Solving.,” in *Proceedings of IJCAI.*, pp. 1018–1024., 1987.

[198] M. Drummond and K. Currie, “Goal ordering in partially ordered plans,” in *IJCAI* (N. S. Sridharan, ed.), pp. 960–965, Morgan Kaufmann, 1989.

[199] T. McCluskey and J. Porteous, “Learning heuristics for ordering plan goals through static operator analysis,” in *Methodologies for Intelligent Systems* (Z. Ras and M. Zemankova, eds.), vol. 869 of *Lecture Notes in Computer Science*, pp. 406–415, Springer Berlin / Heidelberg, 1994.

[200] J. Koehler and J. Hoffman, “On Reasonable and Forced Goal Orderings and their Use in an Agenda-Driven Planning Algorithm,” *Journal of Artificial Intelligence Research*, vol. 12, pp. 339–386, 2000.

[201] J. Cheng and K. B. Irani, “Ordering Problem Subgoals,” in *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pp. 931–936, 1987.

[202] J. Rintanen, “Heuristics for planning with SAT,” in *Principles and Practice of Constraint Programming - CP 2010 - Sixteenth International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, vol. 6308 of *Lecture Notes in Computer Science*, Springer, 2010.

[203] M. Helmert, “Complexity results for standard benchmark domains in planning,” *Artificial Intelligence*, vol. 143, no. 2, pp. 219–262, 2003.

[204] M. Helmert, “New complexity results for classical planning benchmarks,” in *ICAPS* (D. Long, S. F. Smith, D. Borrajo, and L. McCluskey, eds.), pp. 52–62, AAAI, 2006.

[205] A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos, “Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners,” *Artificial Intelligence*, vol. 173, no. 5-6, pp. 619–668, 2009.

[206] C. Williams and T. Hogg, “Exploiting the deep structure of constraint problems,” *Artificial Intelligence*, vol. 70, pp. 73–117, 1994.

[207] G. Boukeas, C. Halatsis, V. Zissimopoulos, and P. Stamatopoulos, “Measures of intrinsic hardness for constraint satisfaction problem instances,” in *SOFSEM 2004: Theory and Practice of Computer Science* (P. Emde Boas, J. Pokorn, M. Bielikov, and J. tuller, eds.), vol. 2932 of *Lecture Notes in Computer Science*, pp. 184–195, Springer Berlin Heidelberg, 2004.

[208] E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham, “Understanding random sat: Beyond the clauses-to-variables ratio,” in Wallace [233], pp. 438–452.

[209] K. Smith-Miles and L. Lopes, “Measuring instance difficulty for combinatorial optimization problems,” *Computers Operations Research*, vol. 39, no. 5, pp. 875 – 889, 2012.

[210] P. Cheeseman, B. Kanefsky, and W. M. Taylor, “Where the really hard problems are,” in *Proceedings of the 12th international joint conference on Artificial intelligence - Volume 1*, IJCAI’91, (San Francisco, CA, USA), pp. 331–337, Morgan Kaufmann Publishers Inc., 1991.

[211] P. Prosser, “An empirical study of phase transitions in binary constraint satisfaction problems,” *Artificial Intelligence*, vol. 81, no. 1-2, pp. 81–109, 1996.

[212] B. M. Smith and M. E. Dyer, “Locating the phase transition in binary constraint satisfaction problems,” *Artificial Intelligence*, vol. 81, no. 1-2, pp. 155–181, 1996.

[213] M. A. Salido and F. Barber, “Exploiting the constrainedness in constraint satisfaction problems,” in *Artificial Intelligence: Methodology, Systems, and Applications LNAI 3192*, pp. 126–136, 2004.

[214] T. Walsh, “The constrainedness knife-edge,” in *In Proceedings of the 15th National Conference on AI (AAAI-98*, pp. 406–411, AAAI Press / The MIT Press, 1998.

[215] S. Thiebaux and M.-O. Cordier, "Supply restoration in power distribution systems: a benchmark for planning under uncertainty," in Cesta and Borrajo [238], pp. 85–96.

[216] S. Edelkamp, J. Hoffmann, R. Englert, F. Liporace, S. Thiebaux, and S. Trug, "Towards realistic benchmarks for planning: the domains used in the classical part of IPC-4.," in *Proceedings of the International Planning Competition 2004*, pp. 7–14, 2004.

[217] E. C. Freuder, "Eliminating interchangeable values in constraint satisfaction problems," in *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 1*, AAAI'91, pp. 227–233, AAAI Press, 1991.

[218] R. Weigel and B. Faltings, "Compiling constraint satisfaction problems," *Artificial Intelligence*, vol. 115, pp. 257–287, Dec. 1999.

[219] R. Weigel and B. V. Faltings, "Structuring Techniques for Constraint Satisfaction Problems," in *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*, pp. 418–423, Morgan Kaufmann, 1997.

[220] P. Gregory, D. Long, and M. Fox, "A Meta-CSP Model for Optimal Planning," in *Proceedings of Abstraction, Reformulation and Approximation, Seventh International Symposium*, pp. 200–214, 2007.

[221] F. Dvorak and R. Barták, "Integrating time and resources into planning," in *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 2*, pp. 71–78, 2010.

[222] M. G. de la Banda, K. Marriott, R. Rafeh, and M. Wallace, "The Modelling Language Zinc," in *Principles and Practice of Constraint Programming - CP 2006 - Twelfth International Conference, CP 2006, 2006. Proceedings* (F. Benhamou, ed.), vol. 4204 of *Lecture Notes in Computer Science*, Springer, 2006.

[223] A. J. Fernández and P. M. Hill, "A comparative study of eight constraint programming languages over the boolean and finite domains," *Constraints*, vol. 5, no. 3, pp. 275–301, 2000.

[224] I. P. Gent, C. Jefferson, and I. Miguel, "Minion: A Fast, Scalable, Constraint Solver," in *Proceedings of the Seventeenth European Conference on Artificial Intelligence*, 2006.

[225] R. Barták, "On constraint models for parallel planning: The novel transition scheme," in *Eleventh Scandinavian Conference on Artificial Intelligence, SCAI 2011, Trondheim, Norway, May 24th - 26th, 2011* (A. Kofod-Petersen, F. Heintz, and H. Langseth, eds.), vol. 227 of *Frontiers in Artificial Intelligence and Applications*, pp. 50–59, IOS Press, 2011.

[226] G. Frances and H. Geffner, "Modeling and computation in planning: Better heuristics from more expressive languages," in *Proceedings of the Twenty Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2015.

[227] T. Davies, A. Pearce, P. Stuckey, and N. Lipovetzky, "Sequencing operator counts," in *Proceedings of the Twenty Fifth International Conference on Automated Planning and Scheduling (ICAPS 2015)*, 2015.

[228] J. Hooker, "Logic-based benders decomposition," *Mathematical Programming*, vol. 96, p. 2003, 1995.

[229] P. Nightingale, I. P. Gent, C. Jefferson, and I. Miguel, "Short and long supports for constraint propagation," *CoRR*, vol. abs/1402.0559, 2014.

[230] S. Chien, S. Kambhampati, and C. A. Knoblock, eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO, USA, April 14-17, 2000*, AAAI, 2000.

[231] C. Boutilier, ed., *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, 2009.

[232] J. Rintanen, B. Nebel, J. C. Beck, and E. A. Hansen, eds., *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, AAAI, 2008.

[233] M. Wallace, ed., *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, vol. 3258 of *Lecture Notes in Computer Science*, Springer, 2004.

[234] "Proceedings of the seventeenth international joint conference on artificial intelligence, ijcai 2001, seattle, washington, usa, august 4-10, 2001," in *IJCAI* (B. Nebel, ed.), Morgan Kaufmann, 2001.

[235] *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*, Morgan Kaufmann, 1995.

[236] W. J. Clancey and D. S. Weld, eds., *Proceedings of the Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference, AAAI 96, IAAI 96, Portland, Oregon, August 4-8, 1996, Volume 1*, AAAI Press / The MIT Press, 1996.

[237] E. C. Freuder, ed., *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, vol. 1118 of *Lecture Notes in Computer Science*, Springer, 1996.

[238] A. Cesta and D. Borrajo, eds., *Proceedings of the Sixth European Conference on Planning (ECP-01), Toledo, Spain*, September 2001.