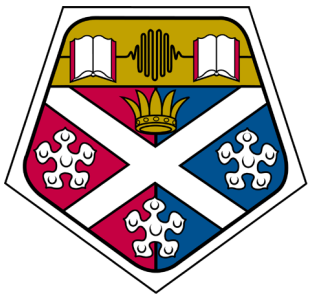
University of Strathclyde
Department of Computer and Information Sciences



Creation of a Goal-Driven and Reactive Agent Architecture

by
Tommy Thompson

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

Doctor in Computer and Information Sciences

2010

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

Signed: .....

Date: .....

Acknowledgements

At the time of writing, I recall the series of events that led to my studying a Doctorate at the University of Strathclyde. A fresh-faced, and energetic Tommy Thompson relished the opportunity to continue studying Artificial Intelligence - a field that had caught his imagination a year or so prior - in a research student position. The steady grant-cheque and future prospect of being addressed as Dr Tommy (Thompson), set my decision in stone. Now, after three and a half years, several thousand litres of tea and an increasing number of grey hairs, a tired, naturally older[1] but still energetic Tommy finds himself writing these words nearing my intended submission date. During this period, I have received guidance and support from many colleagues, friends and family. With these thanks I intend to reach them all, including those who, for reasons of space, could not be mentioned here.

Firsly, I wish to express my gratitude and thanks to my supervisor Dr John Levine. I have had the pleasure of working with Dr Levine since my final year of my honours degree in 2004, and feel that this particular walk in my life could only have happened thanks to his counsel and support. Through John I have discovered many truths in life, notably that one cannot use the term 'Goomba' in a serious scientific text without coming under scrutiny. Despite this, you can still travel to conferences in the far corners of the world provided you develop AI tanks that learn to shoot said Goombas. I have enjoyed the last five and a half years spent in his company, and am indebted not only to the guidance of an effective supervisor, but also a good friend. I hope that in future we may continue our unique, amusing, bizarre, yet very rewarding working relationship.

I also wish to express my gratitude to the staff within the Strathclyde Planning Group and throughout the Computer and Information

[1] By naturally, I mean exactly three and a half years older. Fortunately, while it feels like time flies during a PhD, it is still deceivingly constant and linear.

Sciences department. Having previously studied in this same department as an undergraduate, I have found the transition to research a rewarding and welcome experience.

Special thanks must be made to all my friends in the research area of level 11 (12) of the Livingstone Tower. If anything, this open-area is testimony that only in academic research can so many 'interesting' individuals work in one place and yield strong returns, but do so, somehow, without killing one another. I'd like to especially thank my office mates Alastair, Dave and Alan for the (constant) banter; ranging from Dave's inability to function without internet forums and Alan's hatred of self-service checkouts, to Alastair and his subconscious desire to live life as a benefit-thieving drunkard. All in all, it kept us laughing - either at life or one another - during the hardest of periods. I wish them all the best for the future.

A thank you to the many academics in the Computational Intelligence and Games community who have welcomed this bewildered outsider. I give special mention to Philippa Avery, Georgios Yannakakis and Julian Togelius for their advice and guidance.

I also take this opportunity to thank my many friends outside academia, if anything to prevent the torrent of abuse I'd receive if I did not! To Graeme, Allan, Chris, Pauline, and the rest of the gang for their weekly invasion of my home on a Friday evening, with reminders that my work may one day lead to the downfall of humanity. It's a joke that never wore (too) thin and I hope that one day I can live up to your expectations. Thanks also to Gihan, Tharanga, Swetha, Amit, Ankur and Dian for their continued support. Oh, and of course, a special mention must go to my long-suffering girlfriend Aimie. Her continued support and motivation helped me through many a long day, and even longer night. Thanks babe!

My deepest and warmest thanks go to my family. To my grandmother Margaret, my brothers Darren and Jack and my parents Julie and Tommy. Your continued patience, understanding, support and encouragement through these last eight-plus years has kept me motivated to go one step further time and again.

I dedicate this thesis to my Dad, Tommy Thompson. You have proven to me time and again that dedication and hard work will yield reward. I hope that this thesis will follow in your example.

Abstract

In recent years, the Artificial Intelligence community has embraced computer games as a key platform for research and application. This has ranged from exploring different games for new challenges, to applying trusted AI methodologies in commercial software. However, much of the applied research has focussed on the creation of reactive agents; software designed to adapt to circumstances as they arise while achieving a local, predefined goal. This often leads to AI game players that neither reason about distant goals nor reconsider their actions should events alter the state of the game significantly. Research in the field of Automated Planning and Scheduling specialises in abstract, model-based deliberation that could resolve these issues. Despite this, it has been given little attention in games research to date.

In this thesis, we chronicle an endeavour to integrate model-based deliberative tools found in planning with robust reactive control in a continuous and dynamic game. This is achieved by designing a layered agent architecture that adopts the JavaFF planning system for decision making, while relying on established reactive control to interface with the world. To create our reactive actuators, we apply Evolutionary Algorithms to Artificial Neural Networks; a common approach for agent design. These neural networks are then merged using the Subsumption paradigm, a top-down hierarchy that decomposes control into unique facets of behaviour while dictating order of precedence. Our research explores the challenges in interfacing the planner with a library of reactive controllers, how plan actions are broken down for execution, and issues that arise due to adversarial agents, uncertainty and inaccurate world models. Our end product provides a unique approach to plan monitoring and execution, creating a goal-driven and reactive agent controller. Through the use of robust and decoupled components, our agent is capable of achieving long-term goals while adapting to new situations and reacting accordingly.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

> It is equally wrong to speed a guest who does not want to go, and to keep one back who is eager. You ought to make welcome the present guest, and send forth the one who wishes to go.
>
> Homer, The Odyssey

In every walk of life, we are faced with decisions that make and shape our destiny as it unfolds through our lifetime. Each decision you make changes not only your life, but also the world around us. Changes made, and their impact on the world are dependent on the person making the decision that instigates the change and their status in the social/political structure of modern society. However, it all relies on the same underlying principles, irrespective of the impact or importance of that decision, be that determining whether to get out of bed one morning, or to prepare for international conflict. Ultimately, we rely on our understanding of the world and how lives, our own or those of others, will be affected by that change.

That is not to say that all decision making is the same, since the decision of senior political figures to deploy a military force is driven by the desire to provide security and reassurance to the people of that country - and also slightly less altruistic virtues. Meanwhile, you the reader have probably contemplated what you want for your breakfast/lunch/dinner today. This decision will be driven by

you probably feeling hungry, your preferences in food and your lifestyle habits. However, in the grand scheme of things both of these are incredibly important decisions: since irrespective of how secure your country is, failing to eat will starve you to death. Conversely, we feel it best that government officials never make important decisions on an empty stomach.

But why did you make that choice at that time? Why did we drive that particular road through town? Why do we spend half of our day working in a job? Why am I drinking a cup of tea whilst writing this? Heck, why am I even writing this thesis and why are you reading it? The reasons behind each of these decisions, irrespective of selfish or altruistic reasons, are driven by our desire to complete prescribed tasks that, in time, lead to some form of reward. Such a reward can be explicit and immediate, such as getting home from work 20 minutes earlier, to something long term and implicit like improved health or the respect/love/admiration of friends, family or colleagues. Ultimately, it is about making the right choice at the right time given our knowledge and understanding of the world and how we can improve our lives, and those we care about, as a result. Our ability to reason about the world through thought and deliberation, and instinctively react to situations with such competence and clarity, is what separates us from other creatures on this planet. As such, the ability to replicate these intelligent thought processes for use in industrial/commercial/scientific applications without the need for a human to be present would carry great potential. This led to the creation of Artificial Intelligence (AI), a scientific pursuit to engineer intelligence in a computationally sound form:

> Artificial Intelligence (AI) is the study of intelligent behaviour (in humans, animals and machines) and the attempt to find ways in which such behaviour could be engineered in any type of artefact. (Whitby [2003], pg. 1)

Historically, the idea of "intelligent behaviour" has been strongly debated within the AI community. It has been argued that we should use human intelligence as the example, creating machines that think or act like humans. However, as research in AI has grown, it has encompassed fields as diverse as computer science, discrete mathematics, logic, economics, psychology and even philosophy (Russell and Norvig [1995]). This has weakened the argument to rely on human intelligence as the benchmark, since many of these fields still debate how it actually functions. Hence, a more contemporary interpretation of the AI mandate has arisen that

revolves around the concept of *rationality*, and how it may be applied to a decision maker called an *Agent*. Rationality and agents are the cornerstone of the AI textbook 'Artificial Intelligence: A Modern Approach' by Stuart Russell and Peter Norvig and are described as follows:

> [Rationality is] an ideal concept of intelligence... A system is rational if it does the "right thing", given what it knows. (Russell and Norvig [1995] pg. 1).

> An agent is just something that acts (*agent* comes from the Latin agere, to do)... A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. (Russell and Norvig [1995] pg. 4)

Essentially, if any person or automaton makes the best action in any circumstance, then it can be said to act rationally and will yield optimal returns. Note that it is argued by the authors that rationality is an ideal intelligence, rather than any intelligent behaviour. Whilst humans are intelligent beings, often our decisions are irrational as they are affected by our point of view or perspective[1]. Ironically, rationality is what separates AI systems from their human creators.

So how does an agent act rationally? Do they react to situations as they happen? Do they deliberate on the problems at hand to find an optimal solution to a distant goal? Do they explore the world and behave according to the knowledge they have accrued along the way? In short, the answer to all of these situations is *yes* - provided the best action is taken. Different approaches for rational agents have emerged as a result of the myriad of sub-disciplines in the AI community. Within these disciplines, many provide unique and interesting takes on the reactive and the deliberative agent. However, there is a great divide that separates reaction from deliberation; while the former acts on the knowledge of the present, the latter relies on understanding the world in general and how actions may affect it in the future. In this thesis we are interested in trying to merge these types of behaviour together to create a rational and robust agent for a game environment.

In this introductory chapter we aim to give the reader an insight into the fundamentals of rational agents and the challenges provided by applying them to game environments.

[1] Our personal or religious beliefs, desires or motivations

1.1 Rational Agents

> Computers can do that?!!
>
> Homer, The Simpsons

Here we introduce the key concepts and fundamental models required in order to construct rational and robust AI agents.

1.1.1 Agents and Environments

As previously noted, in modern AI literature the primary decision maker is often referred to as an agent. An agent is often determined by the world it must interact with, called the *Environment*. The agent is responsible for making rational decisions on how to act within this environment. These actions will in most cases instigate change in the environment and possibly the agent itself. These entities interact continually, resulting in a cycle of perception and action by the agent. We can refer to any type of AI system designed for rational decision making within an environment as an agent. Furthermore, an agent may be a hybrid of intelligent systems, with the purpose of becoming more effective in its prescribed task. A task will often be a set of conditions in the world that the agent must achieve, and will often require multiple interactions with the environment to achieve it. These tasks could potentially range from low-level goals, e.g. avoiding a nearby obstacle, to more complex and abstract achievements, e.g. navigating through a building. Irrespective of the task, the agent will aim to maximise utility by completing tasks as efficiently as possible. In order to effectively accomplish prescribed tasks, an agent ideally requires several important components, notably:

- A consistent and accurate model of the environment.
- An understanding of how actions affect the environment.
- A measurement of success and conversely, a measure of failure.

The first two points are the most challenging, since different environments in the real world vary in attributes. For an agent to operate, it must be aware of the qualities of the environment. Once we can characterise an environment, we can begin to address how to model it for problem solving purposes. According to Russell and Norvig [1995], any environment can be conceptualised as follows:

Fully or Partially Observable : *Observability* relates to the amount of information available to the agent relative to the task at hand. If the agent has access to all information relevant to the decision making process, then the environment is *fully observable*. Otherwise we operate under partial observability, where we are not privvy to all factors relevant to our choice of action. A simple example of this is the game of poker: where we are only aware of the cards in our hand and at best an educated guess of what cards other players may be holding.

Deterministic or Stochastic : In more simple circumstances we assume a *Deterministic* world, where our actions will always result in a specific outcome. However in uncertain (*Stochastic*) environments, we must develop an understanding of how likely it is that a future scenario can arise based on a particular action being carried out.

Episodic or Sequential An *Episodic* environment requires that an agent select an action irrespective of the previous episode. Furthermore, the selected action should have no effect on the subsequent episode. Meanwhile *Sequential* environments, such as chess, require the agent to consider the actions it has taken previously and the consequences of future actions.

Static or Dynamic An environment such as chess is considered *Static* as the environment does not change during deliberation. However, a car racing game would be considered *Dynamic* since the world is changing as the player thinks, forcing them to consider actions faster and more frequently.

Discrete or Continuous Such distinction can be made based on the way in which time progresses and the way the environment is modelled. A game of chess would be considered a *Discrete* problem, due to the finite number of board configurations and their distinct separation through players moves. Unless playing with a clock, time has no relevance to this problem. Meanwhile, a racing game would be considered *Continuous* due to the effect time has on decision and progress and the consecutive scenarios that occur as a race progresses.

Single or Multi-Agent We must define which entities exist in the environment and affect the overall problem. However, we must consider whether such an entity has behavioural traits that merit consideration as an agent, rather

than simply as a dynamic element of the environment. They may be an opponent as in a game of chess, or a cooperative agent whom we may need to communicate with.

These attributes define the features of the environment; however they do not necessarily dictate the difficulty of subsequent problems. Naturally, if we are dealing with a non-deterministic environment with imperfect information - acting in a world we know little about and are unsure of how our actions will affect it - then we are (typically) dealing with a very challenging problem. This does not necessarily mean that an environment where we know *everything* leads to trivial problems. Perhaps the simplest and most recognisable of these problems is a Rubik's cube. We can see every part of the cube's surface and know how our immediate actions change that surface; thus, the cube is deterministic and fully observable. As a result, the majority of people who have ever been faced with a Rubik's cube will understand how one works, but how many people can solve it? A slightly more complex example is the game of chess, where we can see all pieces on the board (fully observable) and we know how each piece can be moved (deterministic). However, playing chess well is very difficult. This is due to the sheer number of different moves that can be made throughout a game. Furthermore, it is a multi-agent problem, since an opposing player adds another dimension of complexity to the game. Now we must also consider every move made against our agent and how it may constrain future action selection. This selection of potential actions, referred to in search literature as *branching factor*, is too large for most people to comprehend, even when constrained by an opponent's choices. Therefore, it is often necessary to reduce selection across a set of *useful* actions. While there has been success in creating a chess playing computer system - IBM's *Deep Blue* - this arose thanks to supercomputers that traversed hundreds of thousands of potential actions throughout any given match (Campbell et al. [2002]), with very little informed decision making taking place. Ultimately it is the features of an environment and the decision space they create, that dictates the difficulty of any problems it may present.

Once the rules of the environment have been established, we then require some type of representation or *model* that the computer will understand and process. Modelling an environment and its characteristics is often one of the most time consuming and difficult aspects of problem construction. This is in part due to the wealth of data that can potentially be represented, raising questions of how it

should be represented and how much should actually exist within the model. As a reactive agent, we may only want to consider that which is relevant to our goal and that which we must react to. Meanwhile, for a more deliberative system, we may only be interested in the more abstract changes our existence may have in the world. As such we may ignore much of the finer details in the world as they are irrelevant. These issues amongst others must be addressed once we decide on how we wish to solve a problem within the environment.

Finally, a measurement of success is required in order for us to solve a problem. Like any rational agent, even AI requires a reason to go about its business. An agent requires not only a *Goal* to satisfy, but a means to measure how close it is to reaching that goal. Think of any action you do in your everyday life, it is never carried out merely because you can, but rather because there are goals that need to be satisfied. One works a regular job, eats lunch and attends a gym on a normal day because they have goals to earn money, stop feeling hungry and maintain fitness. Ultimately, we require a means to select and justify actions based on the (perceived) value of their outcome and whether they bring us closer to satisfying our goals. Perhaps unsurprisingly, even AI constructs require the same thing.

Problem Modelling

In research the agent is often assigned a challenging task it must complete, typically by interacting with the environment in order to generate a particular *State*; a set of specific conditions set by the researcher. This relies heavily on not just the model of the world, but how our measurement of success relates to the model. Even from our limited experience, we have observed that this is perhaps one of the most challenging, yet exciting, aspects of research, since we must spend time considering the many facets of the problems we wish to solve. Notably:

- How do we model the problem to store it on the computer?
- Given the model, what kind of decision-making process can then be used to solve the problem?
- Is the measurement of success in the model sufficient? Or is it necessary for us to create a supplementary reward structure specifically for this problem?

As we stated previously, the ‘world’ we wish to consider will carry many characteristics that dictate how we model it. For example, we may model board

games such as chess and draughts[1] in a similar fashion given that they are both board games with similar rules. However we would model backgammon differently as the rules and characteristics of the game differ. One must also consider the level of granularity that must be applied to the model, i.e. how much detail is applied based on the level at which we view it from. Granularity is strongly dependent on what kinds of problems you wish to solve and the kind of decision making we wish to focus our interests upon. This is important as we do not want to take on any unnecessary information in order to solve the problem, nor do we wish to make the problem any more challenging. For example, if we consider an agent that decides how to navigate through a building based on observing directions found on signs to a goal location, do we need to model the mechanics of the leg or wheel being used to move the agent? Do we need to model the contents of each room in the building? Answering these questions is an important aspect of the modelling process and is defined primarily by the scope of the research and also the methodology that we wish to apply. Hence, in circumstances where multiple methodologies are applied within a framework or achitecture, there can potentially be multiple models of the same world tailored to suit each approach.

In order to facilitate an agent's ability to reason, and potentially learn, about the environment, we require a measurement for success. This is achieved through *Reward*, conceptually a numeric value that measures how valuable or desirable a particular state is and the action used to attain it. Naturally, these rewards are structured with respect to the environment and the goal. This can then be tied into a search or learning construct (as we shall see later) to provide a useful metric for performance. Typically, a rational agent's goal is to accrue as much reward as possible.

An example reward structure for chess or draughts would simply dictate winning or losing a match and how close a particular state is to achieving that goal. But you may wonder how this is effective given the large number of moves to win one match. If we were to provide a reward for each subgoal, such as the taking of a piece, then a decision making system may become too wrapped-up in taking specific pieces as effectively as possible, without looking at the bigger picture (Sutton and Barto [1998]). It is important to ensure that rewards are not only easy to quantify but are then propagated through the environment's reward structure effectively to ensure optimal performance.

[1] Also known as checkers.

1.1.2 Solving Problems in the Environment

In the previous section we explained in broad terms the role of an agent in an environment and how it must then create a robust computational model in order to reason about it. However, these models are often dictated by the type of problem a researcher is keen to address. Like many other complex problems in engineering and science, there is no 'silver bullet'[1] to rational decision making and as a result there are a variety of different disciplines within the AI community that address many different problems. It is beyond the scope of this thesis to identify and summarise all of these areas, but we now give a brief account of two areas of interest, notably reactive and deliberative control.

Reactive Control

Reactive actions are an example of common human behaviour and are an area ripe for exploration in AI disciplines. Unlike deliberative action selection, often referred to as *Action Theory* (White [1968]) by philosophers, reactive or reflex actions are often taken from a more biological perspective. Reflex action or simply 'reflex' in humans is considered an involuntary and virtually spontaneous movement of the body as a response to a stimuli (Purves et al. [2004]). These responses range from tendon reflexes to actions within the central nervous system. In fact it has been argued that basic functions such as breathing, digestion and heartbeat are reflex actions. In short, a reaction is a simple response to a series of signals fed from our body's senses.

Our previous work in Thompson [2006, 2005] focussed on low-level decision making for reactive agents; where the agent must consider what actions to make at every discrete time point. Low-level decision problems, like reflex actions, must factor incoming readings from agent sensors and subsequently interact with components in light of this information. Motor control in robots and autonomous vehicles is a fine example of this, where the AI system responsible will be fed data from a variety of sensors. The agent will then be required to interface directly with the motor controls to effectively and immediately respond to the stimuli fed to it. These are often complex and challenging problems that require the agent to handle a variety of inputs and outputs in real-time. Due to the level of granularity and the scope of the data used by reactive agents, they are often

[1] The term often used to signify the one and only solution we will ever require to solve all problems in a given problem area.

constrained to solving local problems with immediate effect. Furthermore, they are incapable of reasoning about larger issues that pertain to the sensor data they can receive. For example, we can create a reactive agent that may learn to navigate an environment using the sensor data. But this data, and the reactive nature of the agent, would be insufficient for finding the optimal path through the environment.

For a reactive control problem, there are two distinct paths we can take. Often these problems can be solved using a hand-written solution created by a human designer, where we simply dictate action based on a certain state. However, in a worst case scenario, there may be a substantial number of similar states with only minor differences that have no defined action. It may be the case that an existing choice would be sufficient for these similar circumstances had the designer accommodated for it.

The human brain is often capable of compensating for these issues, since a sharp mind will recognise the situation regardless and act according to instinct. This occurs since, while the circumstance may differ, the general case remains the same. This ability is referred to as *generalisation* and is a natural process of the human brain. However, to apply it computationally requires structured mathematical models tailored to suit the specific problem.

A popular remedy is to apply a *machine learning* algorithm to *learn* what action should be made in each state. These learning methods often consider how we can *Generalise* action selection by recognising similar scenarios and then attributing a particular action to satisfy them. However, generalising a problem can often be very challenging, as there may be hundreds or even thousands of similar situations that are then placed under the same heading. This issue is often resolved through the use of a *function approximator*, which acts as the decision maker and approximates the best action to take in a given situation by generalising the possible states. A common function approximator is a computational model known as a neural network. Neural networks, when trained correctly, provide a powerful, customisable and computationally cheap tool for creating reactive agents.

The application of neural networks for reactive control is key to our research. We explore the background and effectiveness of neural networks in Chapter 2, followed by exploring our own work in neural net application, in Chapter 3.

Deliberative Reasoning

In contrast, there are also a variety of systems that deal in deliberative reasoning. These decision processes often operate on a symbolic level, resulting in an abstract model of the world they describe, but seldom interface with it directly. A prime example being the research field of Automated Planning (AP), where the system is responsible for making series of high-level decisions to achieve future goals. However, given that an AP system operates on such abstract levels of reasoning, it is incapable of considering lower-level decisions.

Automated Planning is a discipline within the AI community that aims to achieve one of the most fundamental concepts of human intelligence; the deliberation process that dictates action selection and ordering based on our understanding of future outcome. Planning is a resourceful activity when dealing with large, complex and even unfamiliar situations by stripping the problem to the fundamental concepts prior to any reasoning. This process is highly valuable in a variety of different real world scenarios, ranging from everyday situations such as planning your day's trip through the shops to the safety critical and risky situations found in power stations, disaster recovery and autonomous vehicle control (Rajan et al. [2009], Bernard et al. [2000]). Often in the latter situations we can be dealing with large problems that carry a high cost for any action that is carried out, potentially including loss of life, hence we require not just good, feasible plans but we need them quickly. These human faculties are one of our definining intellectual characteristics, hence the drive for automation by modelling this deliberation process computationally.

The variety of research in the AP community seeks to harness the deliberative power that planning can bring and apply within a variety of applications. The potential for tools that give users access to efficient and effective planning resources is certainly attractive. Engineering and logistics often rely on planning based tools to optimise performance, predict overall cost of materials and project lifecycles. While there are a significant number of current applications and success stories, AP as a research field is still in its infancy. Seminal breakthroughs in automated planning led to the use of representations for abstract decision problems, yet there is still a significant amount of work to be carried out in exploring additional forms of complexity, ranging from the control of resources through metrics to the addition of multiple actors and the management of time and scheduling actions within a specified timeframe. Furthermore, while current AP technology works well in

discrete, deterministic, fully-observable, single-agent problems, applying these principles to continuous, non-deterministic and stochastic systems is a vibrant and active research field.

Our interests are in applying the classical - deterministic, discrete, sequential and static - form of planning to a complex, dynamic game. While these are tried and tested methods, there are challenges in employing them for deliberative reasoning in such a challenging environment. These range from how best to model the problem at hand, to the level of detail that is applied to the model. Can we model the environment to a reasonable degree and rely on more granular forms of reasoning to take control? Can we then exploit the power of a planning system to provide alternative solutions if things go awry?

In Chapter 2, we provide a complete breakdown of the classical planning approaches and representations; with a succinct technical background as well as highlights of important research in the community.

1.2 The Challenges of Game-based Research

> If you watch a game, it's fun. If you play at it, it's recreation. If you work at it, it's golf.
>
> Bob Hope

In this thesis, we explore the creation of reactive and deliberative agents in a game environment. At this point we wish to highlight the usefulness of applying research within game or toy environments. We begin by addressing the use of toy domains for problems, followed by the benefits of AI research in games. Finally we explore the impact this has on both AI research as a whole and the video games industry.

Use of Toy Domains & Simulation Historically, toy domains are often necessary to explore and test new algorithms and ideas. Furthermore, these are often developed as a simulation due to the impractical costs of manufacturing and developing systems for the real world. As we will see later in this thesis, we implement some of our research in the EvoTanks domain, a simple game involving tanks that are assigned different tasks. Not only is a simulation of this environment far easier to manipulate and control, it is far more cost effective.

After all, we sincerely doubt any PhD student receives sufficient funding for the use of full-size tanks to have them repeatedly destroy one another.

Furthermore, a simulation of a problem carries many assumptions that benefit the researcher. In these circumstances we remove many of the implementation problems that would challenge the engineer responsible for manufacturing the physical equivalent. This allows the AI researcher to focus on designing the agents' decision making process. This scenario is common in a variety of AI research, since many intelligent systems require test problems to ascertain the effectiveness of the technology, as well as simulation test-beds to visualise how the system would operate when plugged into the intended, no doubt expensive, hardware. This in turn often leads to more effective research, as it permits greater freedom to explore ideas and try additional tests without budget and, in some cases, temporal constraints, since a simulation can operate faster than real-time.

AI & Games Applying AI practices to games is now a popular avenue for research and development. Games are a fantastic test bed to explore the application of AI methods given their complexity and the challenges they present. Typically, these games present an established problem domain with a set of actions a player can apply in an environment to achieve a goal. During play, the player needs to make informed decisions to ensure their progress. These decisions can range from reacting to nearby events, or acting towards achieving tasks across a larger time frame. As they progress further, they are rewarded for their efforts, with an impetus to continue playing to achieve the final goal of the game. As you may have already surmised, these traits sound similar to the AI concepts we have discussed so far in this chapter.

The practicality of applying AI to these problems also applies to video games. In fact, it could be deemed more practical given the larger range of problems video games provide and the fact that any existing game software is essentially a problem simulator; by providing a challenging problem domain that can be interfaced with using the actions available to the human player. By interfacing an AI agent to the software directly, rather than through a keyboard or controller peripheral, we can introduce a range of complex and interesting problems to researchers. This in turn can benefit game developers themselves since to date, the control of cooperative or adversarial characters in video games is often reactive. Furthermore, this reactive behaviour is dictated either using hardcoded behaviours or, at best, simple AI practices. Moreover, there is a lot of room for improvement

in these behaviours, ranging from the creation of more robust reactive control, to the use of deliberative agents that could provide assistance or genuine competition to the player.

Research in Games-based AI Whilst research in games is not novel, in recent years a growing community of AI game-based researchers has emerged. A prime example of this can be found in the IEEE Conference on Computational Intelligence and Games (IEEE CIG); an international academic conference that occurs on an annual basis, with the purpose of disseminating and promoting research in theoretical and applied AI research in games. Furthermore, in the United Kingdom is the Engineering and Physical Sciences Research Council (EPSRC) AI and Games Network, a networking community that convenes several times a year for workshops. The intention of the network is to provide a meeting point for academic game-research enthusiasts and representatives from the UK game industry to come together for discussion and potential collaboration. The benefits of these organisations are twofold. Researchers have become increasingly aware of the benefits of applying AI practices to games, to the point that there are now respected members of AI communities advocating their application (Laird and VanLent [2001]). Such applications range from the ability to construct test problems in established and robust simulators, to being able to observe the performance of these AI constructs as they interact with the human player. Meanwhile, the games industry is now taking an active interest in the potential benefits that AI can bring to their products, from reducing development time to enhancing the overall experience for the player.

1.3 Research Goals

> I'll tell you the problem with the scientific power that you're using here: it didn't require any discipline to attain it. You read what others had done and you took the next step.
>
> Dr Ian Malcolm, Jurassic Park

In this section, we give the reader a clear indication of our research goals, the steps we intended to take in achieving them and how this is reported within the thesis. However, we begin by providing background on our previous research that led to our goals being formulated. Next we discuss the concept we wish to explore and the contributions it will provide to the research community. We then highlight each of the research goals this project tackles, identifying the areas of the thesis that address them.

1.3.1 The Idea...

Our previous work focussed primarily on creating reactive control for low-level execution problems using the aforementioned neural networks described in fuller detail in Chapter 2. We applied these neural networks to a dynamic, continuous game environment called *EvoTanks*, which forces tanks to compete with one another in a small enclosed arena. This was achieved by comparing different forms of evolution-based machine learning algorithms that trained the neural network controllers. These evolutionary algorithms searched for solutions against a defined criteria, with the reactive controllers gradually evolving to satisfy our requirements,

However, after our research detailed in Thompson [2005] and Thompson [2006] was completed, the benefits and drawbacks of the approach had become apparent. These reactive controllers, while capable, were limited in scope in terms of their flexibility and functionality. While each controller was robust and its behaviour effective, they could only solve one simple goal. This reflected much of the work to date in video game AI. Hence we wished to explore how we can *build a reactive controller that required deliberative faculties*; an agent that would need to decide on multiple actions to complete an assigned goal. Such deliberation could readily

be achieved using planning and scheduling methodologies, the challenge then being to find a feasible means to include such high level deliberation while still interacting with a dynamic game in real time.

The fusion of planning and execution is far from a novel concept, with a plethora of research in monitoring and execution. Many practical applications can be found in sensitive and safety-critical situations such as the aforementioned control of autonomous underwater vehicles and spacecraft. However, given the risk and substantial cost of these scenarios, these systems are often monitored by a team of human experts with actual execution being carried out by carefully hand-crafted and rigorously tested controllers. As such, we considered applying our trained reactive controllers in an architecture that allowed them to represent and satisfy the conditions and intentions of actions modelled in a classical planning system. To our knowledge this was a rather fresh yet interesting application of neural networks. Furthermore, it was also an opportunity to highlight how classical planning could be applied to a dynamic and continuous game; an approach that to date had been ignored by the community at large. Finally, this would provide a natural extension to our previous research whilst also exploring the challenges of complete autonomy in plan execution; a synergy between an abstract deliberative process with a collection of low-level reactive controllers that satisfy the planner's intent.

However if we were to apply these neural nets within an agent controller, we had to ensure that they would not only be sufficiently robust, but capable of handling a range of complicated tasks. To date our agents have been effective but did not 'scale-up', i.e. if we made changes to a problem in the EvoTanks game to make it more challenging, then our agents proved incapable of solving it. This inability came from the use of neural networks as a reactive controller and the simple learning algorithms we had applied to date. Hence we wanted to explore methods that would allow us to create and subsequently solve more challenging problems while maintaining a simple yet effective approach.

1.3.2 The Contributions

This work would provide two significant contributions to the research community and established literature. The largest contribution this would give is a powerful application of automated planning and scheduling technologies in gaming. As discussed in the following chapter, little work has been applied to date in applying

automated planning within the computational intelligence and games field. The majority of research to date focusses on the creation of action policies for agents using machine learning algorithms. Typically the end product of these algorithms is an agent that is reactive in nature and can respond to changes in the environment, but cannot deliberate on long-term goals. By applying planning as a deliberative component, this can be realised, provided we still have a significant reactive component that can interface with the environment.

The requirement for reactive control to interact with the world leads to our second significant contribution: notably that our intended agent architecture will merge a planner with reactive controllers. As we previously discussed, the merger of planning with execution is not a novel concept. However, it is typically associated with high integrity systems where there is a high risk of potential losses. This is not the case here, given that we are operating in a simulated environment. As a result, we can explore the potential of applying different control mechanisms for execution. By introducing neural network controllers as the reactive control element, we explore a different approach to plan-driven execution, while also highlighting to the computational intelligence and gaming community that such controllers can be employed in a novel fashion.

The use of the neural networks and the planner combined will also lead to an agent that is not only intelligent, but computationally cheap to and autonomous in execution. While there will be a processing overhead to develop the actual plan of action, the execution will be intelligent and fast provided the neural network controllers are trained sufficiently. This will lead to a complete autonomy in the system from top to bottom, allowing us to craft to intelligent agents that fit specific problem domains.

1.3.3 Research Goals

At this juncture we now state each of the main research goals we aim to achieve in this thesis. These research goals are the key questions we seek to answer as our work develops. Furthermore, we indicate how these will impact our research contributions, the challenges they provide and the areas of the thesis that address them.

- *To create an effective agent controller that incorporates deliberative reasoning based on a discrete, deterministic model in the context of a continuous and dynamic game environment.*

  This goal seeks to address our desire to apply classical planning to games. Naturally there are a series of challenges that we face in creating the necessary models to function within a game environment. To understand how to create the models we must understand how classical planning systems and their representations work. This allows us to understand the power and limitations behind their usage. We explore how classical planning models are constructed in Chapter 2. Furthermore, we introduce a challenging game environment in Chapter 4 and discuss in depth how we create a discrete and deterministic model based upon it.

- *To explore a new evolution-based learning methodology that allows us to train our simple neural networks to solve more challenging problems.*

  As previously noted, our application of evolutionary learning algorithms will not allow us to scale our reactive controllers to more challenging scenarios. Given our intent to use them in a challenging new game environment, we needed to ensure that they were robust and effective in their prescribed task if we wish to attain a reliable autonomy in our architecture. To understand how this learning methodology is created, we begin by providing a breakdown of evolution based methodologies in Chapter 2. In Chapter 3 we explore this goal in detail, discussing the agent design, the learning methodology and an extensive series of tests. The results of this work proved strong enough to merit publication at conference level in Thompson and Levine [2008]. Further work in this area was conducted by final year honours student Fraser Milne under our supervision. This was published the subsequent year in Thompson et al. [2009].

- *To integrate a classical planner with a collection of neural networks to achieve deliberation and reactive reasoning.*

  This goal once achieved would lead to the most important contribution of our work. This requires us to integrate work from the previous two goals within our agent architecture. The work found in Chapter 5 addresses this challenge directly as we create our new agent architecture. In this architecture we build individual components that manage the planning system and the collection of reactive controllers and test the complete system in our new game environment. We then reflect on these results and the benefits, drawbacks and contributions that this new architecture provides in Chapter 6. The work in Chapter 5 has also been published at conference level, as shown in our publication list, found in Appendix G.

- *To build an agent controller that is effective at solving a range of complex problems while comprised of relatively simple and robust components.*

  In order to ensure that we have created a fast and robust system, we sought to ensure that the system is comprised of effective and small components. This design choice can be seen throughout Chapters 3, 4 and 5 as we build our modular planning system.

1.3.4 Challenges

Furthermore, our initial goals will lead to some challenging issues in implementation. Hence we are keen to provide answers to the following questions:

- *How can we scale-up our reactive agent controllers without compromising behaviours established in our previous work?*

  This was our biggest concern when seeking to create our new reactive controllers. We address this concern in Chapter 3 by employing a layered learning approach that is discussed in depth in Chapter 2.

- *How can the reactive neural network controllers represent the planner's actions and how can we associate them?*

  When building our agent architecture, we needed to ensure that the 'gap' that existed between the planning system and neural networks was resolved.

This is tackled in Chapter 5 through the use of a rule based system acting as an intermediate.

- *How do we compensate for factors such as enemy agents or artefacts with quantified characteristics that cannot be modelled in a discrete, deterministic, classical planning system?*

  This challenge is again another factor that needed to be addressed when our agent architecture was created. We address this issue again in Chapter 5 through the use of the rule system.

1.4 Dissemination and Publication of Research

As highlighted in Section 1.3, the major components of our research found in Chapters 3 and 5 of this thesis have been published at international conference level. Given that our research is focussed on applying AI practices to game environments we have submitted our research to the IEEE Symposium on Computational Intelligence and Games. Our publications range from the major contributions of our thesis in Chapters 3 and 5, to side-projects outwith the scope of this thesis. As part of this thesis, we have provided a complete publication list at the time of submission in Appendix G.

Chapter 2

Technical Background & Related Research

> If you steal from one author, it's plagiarism; if you steal from many, it's research.
>
> Wilson Mizner

The research in this thesis crosses a variety of areas within AI. Due to the breadth of work involved, it is important the reader be able to understand a variety of different AI methods, at least on a conceptual level. This chapter provides a technical background to the main research areas explored within the thesis, namely the application of *artificial neural networks* combined with *evolutionary algorithms* to create reactive behaviours and the use of *automated planning and scheduling* for deliberative faculties. Furthermore, we provide an investigation into related research in creating reactive agent controllers through machine learning and advances in plan formulation, monitoring and control.

We begin by exploring the means by which we can create simple reactive agents. This leads on to a formal introduction to artificial neural networks, followed by the design concepts and history of evolutionary computation. These topics highlight how, by combining these methods effectively, reactive control can be achieved. Following this is an exploration into how evolutionary learning can be applied to more complex situations, indicating a desire to 'scale-up' reactive controllers to more challenging environments that require a breadth of functionality.

The second half of this chapter explores deliberative reasoning technology, starting with a breakdown of classical planning approaches and the traditional representations used to model domains and individual problems. This is followed by a brief introduction to a variety of advanced planning techniques, incorporating metric optimisation, temporal planning and scheduling. Finally, to highlight the work to date in merging deliberation with execution, we explore the integration of monitoring and control systems with planning platforms.

2.1 Evolving Artificial Neural Networks

In this section we explore the technical background and research history of applications required to create an effective reactive controller. While we could explore the large variety of techniques available we focus specifically on the combination of Artificial Neural Networks with Evolutionary Algorithms. Both of these technologies have proven to be very popular with researchers, a judgement based on the vast wealth of research available and accessible via journals, conferences and online resources. We endeavour to streamline this background section to provide a succinct and straightforward clarification, focussing on elements important to our research while highlighting other important areas for completeness and potential further reading.

2.1.1 Artificial Neural Networks

> My CPU is a neural-net processor;
> a learning computer.
>
> The Terminator, Terminator 2:
> Judgement Day

An Artificial Neural Network (ANN) is a non-linear mathematical model that processes information through a connectionist approach to computation; a parallel distributed processing methodology that adapts individual units and their connectors to generate interesting and intelligent output signals. Like many of the methodologies explored in this section, ANNs incorporate features of natural biological processes and constructs. The ANN specifically adopts concepts of human neurology by mimicking the structure and functional design of biological neural networks found within the brain. Traditionally an ANN is comprised of a series of processor units called *Neuron*s, interconnected by a series of links called *Synapse*s. A *Neuron* is connected through numerous layers of computation. Typically these are organised in a series of layers that either interact directly with the environment, as a result of being connected to input signals or act as actuators. Otherwise they are 'hidden' within the structure of the network, i.e. they only interact with neighbouring neurons in nearby layers (Russell and Norvig [1995]).

A simple example of an ANN can be found in Figure 2.1, where we see a 3-layer network consisting of nine neurons. Directed edges in this figure represent the synapses as the output signal of each individual neuron is transferred across to

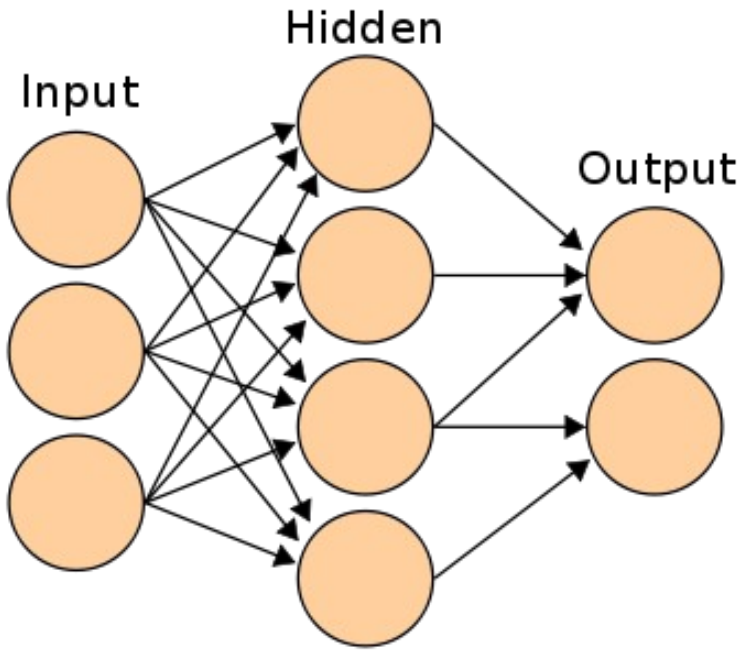


Figure 2.1: A simple example of a layered, feed-forward artificial neural network (ANN). This network is comprised of three layers: an input layer containing three neurons, a hidden layer of four neurons and an output layer of two neurons. Note the directed edges representing the synapses of the network, connecting each individual layer.

the next layer of the network. Connectivity within neural networks traditionally involves layered topologies, where neurons from one layer are only connected to the next layer, hence there are no connections between nodes within the same layer, nor connections that skip ahead to future layers in the network. The two most common forms of network structure are referred to as *feed-forward* and *recurrent* networks. Feed-forward networks can be considered similar to a directed acyclic graph, i.e. all links are unidirectional and there are no cycles within the network. Meanwhile a recurrent network is not constrained in a similar fashion, permitting the formation of arbitrary topologies.

Feed-forward multi-layer neural networks can often be considered as an effective non-linear mathematical model of a desired function, providing a direct mapping of some given set of inputs to a desired set of outputs. As a result, the ANN acts as a function approximator that will, if correctly configured, generate the appropriate output when required. Typically, these networks need to be tailored, or as we shall see in Section 2.1.2, trained, to satisfy this function from an initial (typically incorrect) state (Bishop [2005]). This allows agent designers to configure a network to create a generalised and robust solution without the need for a complex mathematical model by the designer. Perhaps most importantly, this is one method of creating a reactive controller.

For an agent to achieve reactive control, we require that the agent is capable of adapting behaviour with respect to changes in sensor data. Furthermore, we also require these changes to occur quickly so as to ensure that the agent can respond to changes as effectively as possible. This can be achieved by training an ANN with respect to a particular task you wish to achieve. Once it is suitably trained, the generalised solution will provide an adequate action policy based on the incoming sensor data. Also, given the speed at which the output is generated, the network appears to 'react' to changes in the sensor data. This means that an agent driven by the likes of a trained ANN can be relied upon to act as a reactive controller in a dynamic environment.

The reasoning behind the term 'network' in ANN is based on the assumption that the ANN's non-linear function $f(x)$, is comprised of a series of individual functions $g_i(x)$ for each incoming neuron. Furthermore, these can be considered the summation of other functions. This leads to a network structure shown in Figure 2.2, where the final function is dependent on the summation of those in the preceeding layer and so on back to the original input x. The most typical form of value propagation - which we use in this thesis - is a non-linear weighted sum

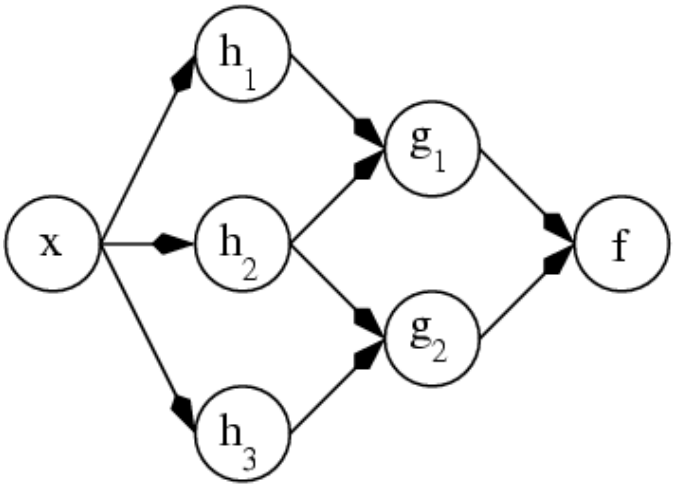


Figure 2.2: A simple dependancy graph within an Artificial Neural Network. Note that each function is comprised of a series of functions in the previous layer.

shown in equation 2.1. This equation computes the output value of a given neuron using the collection of preceding functions weighted against a specific value w, and a predefined function within the neuron itself K, commonly known as a *Transfer Function*.

$$f(x) = K(\sum_i w_i g_i(x)) \tag{2.1}$$

A transfer function defines the output of a neuron given the set of inputs. In ANNs, the transfer function is designed to mimic the action-potential firing within a biological neuron, with typically a binary output, i.e. either the neuron is firing or not. In order to ensure robustness in a single neuron to both increasing input space as well as the potential number of inputs, the transfer function will be normalisable: conforming the potential input to within a defined range (Haykin [2008]). Transfer functions differ based primarily on the designer's preference; however the two most common are the Sigmoidal ($f(x) = \frac{1}{1+e^{-x}}$) and the Hyperbolic Tangent ($\tanh(x) = \frac{\sinh x}{\cosh x} = \frac{e^{2x}+1}{e^{2x}-1}$).

Note that in Equation 2.1 that each of the preceding neuron functions ($g_i(x)$) has a corresponding weight assigned to it ($w_i(x)$). The weight vector $\vec{w} = (w_1, w_2, w_3, \ldots, w_n)$ is the key to modifying and tailoring a given network, allowing for different patterns of behaviour depending on the values within the vector. In the next section we explore the application of evolutionary algorithms and specifically, how these search methodologies can be applied to optimise neural networks for a given problem. Given the non-linear behaviour of ANNs, they

are suitable for modelling complex relationships between input/output sets or patterns of data. This in turn makes ANNs very effective for problems in pattern recognition, function approximation, data classification and sequential decision making.

2.1.2 Evolutionary and Genetic Algorithms

> Mutation: it is the key to our evolution. It has enabled us to evolve from a single-celled organism into the dominant species on the planet. This process is slow, and normally taking thousands and thousands of years. But every few hundred millennia, evolution leaps forward.
>
> Professor Charles Xavier, X-Men

In this section we explore the use of evolutionary and genetic algorithms; a learning process inspired by biological evolution. Throughout we provide a quick account of the history, definitions and terminology used, as well as a brief review of the benefits and drawbacks of such learning algorithms.

Introduction to Evolutionary Computation

Evolutionary Computation (EC) is an iterative, parallel approach for improving the quality of potential solutions to problems where *Candidates*[1] move towards more optimal solutions using a guided random search. As the name would suggest, the method of improvement is dictated by Darwinian principles of evolution. EC emerged from a variety of research by independent computer scientists in the 1950s and 1960s keen to explore the potential of applying evolutionary principles to engineering problems. The field itself was defined through a range of different approaches that, while sharing interests in applying evolutionary principles, presented unique means of implementation. Research by Fogel, Owens and Walsh resulted in the field of *evolutionary programming* (Fogel et al. [1966]), where

[1] potential solutions

evolved solutions were represented through Finite-State Machines (FSMs); a symbolic representation of state and transition. Evolutionary programming, coupled with *evolutionary strategies* and *genetic algorithms*, led to the birth of the EC community and provided the foundation for future research (Mitchell [1996]).

As time has progressed many more practices have been labelled under the EC banner; such as *swarm intelligence*, *artificial life*, *cultural algorithms* and *artifical immune systems*. To retain their identity, the three original practices are commonly referred to in modern literature as Evolutionary Algorithms (EAs): a subset of EC that use selection and modification practices inspired by biological evolution, specifically the concepts of natural selection and reproduction through crossover and mutation. Candidate solutions are considered individual members of a *Population* where they and their environment is defined through a *Fitness Function*; a function that encapsulates the purpose of a candidate and quantifies its optimality with respect to the goals assigned to it. The population improves, or perhaps more appropriately, *evolves*, through generational improvement biased towards higher fitness solutions. Throughout these methodologies, future candidates can be formulated courtesy of modification operators. These operators range from swapping subsections of encodings (crossover) to making minor fluctuations to an existing encoding (mutation). Once more these are inspired by real biological functions. In our research we focus specifically on two methods of learning: evolutionary strategies and genetic algorithms.

The Evolutionary Strategy (ES) was introduced by Ingo Rechenberg in the mid-1960s in the two seminal publications (Rechenberg [1965, 1973]), with further development presented by Hans-Paul Schwefer in Schwefel [1975] that provided an effective optimisation algorithm. ESs are an iterative approach that rely heavily on mutation as a search operator to explore the local solution-space. An evolutionary strategy often utilises problem-dependent representations for search and will operate until some criterion is met that satisfies the designer. Often the point of reference for the search is restricted to a small set of candidates, while selection for advancement runs on a strict ranking of solutions by fitness. Depending on the number of potential mutations permitted, the search process can fluctuate from being an open, parallel search to a greedy random walk. The original ES by Rechenberg was highly aggressive due to the restricted size of the population of candidates. Only one solution was kept as the current point of search, with one mutation per cycle. Should this mutation prove to be a more valuable solution then it replaces the current best, otherwise it is discarded. This is known in

literature as a (1+1) ES. More generally, λ mutations can be generated to compete with the current best, once again with the best solution becoming the new point of reference. This can be found in literature as a (1+λ) ES (Beyer and Schwefel [2002]).

At the time of its conception, the Genetic Algorithm (GA) was a major innovation when compared to other approaches within the ES community. The original genetic algorithm was created by John Holland in the 1960s and developed further by Holland's collaborations with students and colleagues at the University of Michigan until the 1970s. What differentiated GAs from both evolutionary strategies and evolutionary programming was the general formalism that emerged from Holland's research (Mitchell [1996]). Holland's published work found in Holland [1975], presented the GA as an abstraction of the biological evolutionary process, resulting in a theoretical framework for adapting evolutionary mechanisms. Holland's GA focusses on populations of *Chromosome*s; strings of ones and zeros akin to a binary representation. Iterative improvement of a population arises through a form of natural selection, a process that selects candidates for reproduction. The key difference of the GA natural selection process compared to Rechenberg's fitness ranked ES was that fitter solutions would be capable of reproduction *on average*. This allows for poorer candidates to survive in the population as means of ensuring diversity while preventing premature convergence on sub-optimal solutions. The selection process results in a 'parent set' of chromosomes which is then used to create the new population. This is achieved through a mix of crossover, mutation and inversion. Crossover acts as a simple implementation of biological recombination by exchanging subsections of two chromosomes to create new solutions. Mutation modifies values within the string stochastically while inversion simply inverts individual bits in the chromosome.

Since a variety of applied methodologies have emerged within the EA community, continued research has blurred the boundaries that exist between GAs, ES and evolutionary programming (Mitchell [1996]). The term "genetic algorithm" is used to represent a variety of research that differs greatly from the original concept expressed in Holland [1975]. Often researchers rely on a ($\mu + \lambda$) EA; a contemporary evolutionary algorithm that relies on a population of parents (μ)) with a set of mutated offspring (λ), with crossover as an additional operator. While this could be expressed as either a contemporary EA or as an evolutionary strategy, it will frequently be presented in published research as a genetic algorithm (Mitchell [1996]). In this thesis we adopt such flexibility, given that the research

contained here was inspired from the original GA concept but does not strictly adhere to the representation and mechanisms.

Terminology

Here we provide a thorough explanation of the terminology used across a range of EAs. The definitions given in this section are consistent with the majority of existing research and are the basis of our experimentation with EAs throughout this thesis. It must be noted at this point that there are no rigorous definitions for EAs that clearly differentiate them from GAs. For the remainder of this thesis, we will adhere to the terminology shown in Definition 1. In Definition 1, we provide a simple indication of the fundamental components of any evolution-based search algorithm. Furthermore a brief breakdown of the learning process is shown in Algorithm 1.

Definition 1 *A typical evolutionary/genetic algorithm is comprised of the following components:*

- *A Chromosome definition: A concise representation of a point in the search space, thereby creating a potential solution to the problem. The characteristics and requirements of a problem will define the representation applied. Traditionally a chromosome is a string of numbers bound within a strict range. This can range from binary numbers to a more expressive range. Inspired by biological notation, a chromosome is sometimes referred to as the genotype of the solution, given that it is a structured, symbolic representation of the final product.*

- *A Population of Chromosomes: A population must consist of a number of candidate solutions. This allows for a parallel search given that each candidate represents a unique point in the search space. The EA/GA search is responsible for the management and gradual improvement of quality within the population.*

- *A Fitness Function: A simple algorithm that scores the performance of a particular chromosome within the population. This algorithm is required to effectively grade each candidate with respect to a series of strict criteria that define the problem. Naturally, this criteria is domain-dependent; as such, the accuracy of any fitness calculation will also be reliant on the domain.*

In some domains our fitness function may give a 100% accurate measure of solution quality, whereas in other situations it may only give a rough estimate based on the user's demands.

- *Selection: A process that will select chromosomes from the population, denoting them as a parent. Parents are a subset of the population that is responsible for reproducing the candidates for the subsequent generation. The selection process will be guided by the fitness function increasing the probability of selection $p(c)$ for a given chromosome 'c'. Assuming a given fitness function θ is normalised, as $\lim_{\theta(x)} \to 1$ then $\lim_{p(c)} \to 1$.*
- *Variation: Once a parent set has been established, we must create new candidates to be inserted into the population to continue the learning process. Two of the most common forms of variation are established using crossover and mutation.*
  - *Crossover: A function designed to mimic biological crossover of parent genes. This is achieved by swapping sections of the parent chromosomes with one another, creating new candidate solutions from previously successful ones.*
  - *Mutation: A simple method that causes small changes to a given chromosome. This will often be dictated by the chromosome formalism. A simple example in a binary string may be to swap a 0 for a 1 at a given point in the string. Typically mutation may occur with a small probability at each position within the chromosome string.*

The chromosome definition is one of the most important and challenging aspects of EA design. A chromosome must effectively behave as a "blueprint" for the final product (Mitchell [1996]), which can then be understood and later translated for application. As defined, the chromosome exists as the genotype of a given solution, providing a means to recognise gene patterns between individual candidates. This genotype - for the sake of evaluation and later, practical use - must be able to effectively translate to the application it is designed for (the phenotype) and operative effectively. Conversely, the genotype must be designed with sufficient care to ensure that it is capable of representing the intentions of the designer.

The population of candidates can vary depending on the learning model applied. Traditionally we consider a snapshot of an existing population to represent

Algorithm 1: A basic outline of an evolutionary algorithm, where each candidate is assessed in turn, a parent set is created by the selection method, and the population is recreated using the modification operators.

Input: The initial randomised population of chromosomes P and a given fitness function θ

Result: An evolved population that is optimised with respect to the fitness function.

```
1  Evolutionary-Algorithm(P,θ)
2  for i ← 0 to MaxGenerations do
3  |  for j ← 0 to PopulationSize do
4  |  |  candidate ← P_j
5  |  |  evaluate(candidate,θ)
6  |  end
7  |  parents ← selection (P)
8  |  for k ← 0 to PopulationSize do
9  |  |  P_k ← modify (parents)
10 |  end
11 end
12 return best (P)
```

one *generation*. Given that evolutionary learning is an iterative process, we traditionally replace the contents of the population at the end of every generation with new candidates. This is commonly referred to as a generational population model, where future generations contain candidates created using selection and modification operators. A popular alternative method is known as *steady-state* populations, where the set of candidates is relatively fixed, with only a subset of candidates being replaced in each cycle. Hence, when the selection/modification process has created n new solutions, these will often replace the n poorest candidates in the population.

Finally, the fitness function also requires a significant amount of attention, since this is the primary form of assessment. It is important to consider how the fitness criterion is established and what exactly is demanded by the user. An EA is a parallel search that will potentially explore large numbers of possible solutions. If the fitness function is not defined correctly, then incorrect solutions will often emerge as a result. Furthermore, sufficient conditions or constraints must be applied in the fitness function to prevent the search returning what could be considered undesired yet 'correct' solutions[1].

[1] A simple example: We are interested in creating an agent that does not collide with nearby walls, so we devise a fitness function that scores the agent solely on how frequently it collides

Another consideration is 'noise' that may occur in fitness computations, i.e. small errors that may occur in sensor readings and calculations or fluctuations in performance at a task involving some element of chance. Very noisy fitness functions can potentially lead to uncertainty in determining just how 'fit' a given solution is. Thus changes could be made to a chromosome that lead to an improved fitness, however from the designers perspective the improvement is marginal (Beyer [1998]). There is a common misconception that EAs are resistant to noise due to their reliance on Darwinian evolution-based mechanics. However, work found in Rechenberg [1973] showed that increasing noise levels can severely impact the progress rate of a simple (1+1) ES. Two potential yet simple means of noise reduction are discussed in Fitzpatrick and Grefenstette [1988]; increasing the number of evaluations per candidate - thus averaging across numerous measurements while maintaining constant control parameters - and increasing the population size. In Chapter 3 we briefly discuss how we have tackled this problem with the use of a large number of fitness calculations and a modest population size.

In order to move from one generation to the next, we require a series of operators that will create new candidate solutions that move us towards our desired goal. In order to achieve this, we require the selection, crossover and mutation operators shown in Definition 1. Selection is a crucial component of the search process, given that it is responsible for deciding what solutions we should continue to explore given their potential with respect to the fitness function. Once we have made this selection, the crossover and mutation operators help to introduce variation in the population by creating new candidate solutions from the selected set. There are a variety of different selection methods that can be used based on the user's preference. We will now attempt to summarise some of the methods commonly used in research.

Fitness-Proportionate Selection As the name would suggest, fitness-proportionate methods weigh the probability of a chromosome being selected for reproduction based on the fitness it accrued during evaluation. In *rank-based* selection, this selection is dictated by ranking against the absolute fitness values of a solution (Baker [1987]). However, in fitness-proportionate selection, we scale the candidate's fitness against the average fitness of the population. The two most common

with nearby obstacles. Hence, the learning algorithm will find what is naturally the simplest and most effective solution: an agent that doesn't move.

implementations are *roulette wheel* and *stochastic universal sampling* (Mitchell [1996]).

Roulette wheel can be envisioned as the name would suggest, with each candidate solution being visible on a wheel. However the size of each slot on the wheel is proportional to the fitness of a particular chromosome. Hence better solutions have a larger chance of being selected. The parent set is then created by spinning the wheel several times until a sufficient number is selected. Stochastic universal sampling was designed to circumvent the (very) small probability that poor solutions can still fill the entire parent set using roulette wheel (Baker [1987]). In this version, instead of spinning the wheel n times, it is only spun once. However there are now n equally spaced pointers that select all of the parents at once.

Fitness-proportionate methods are often used since they can prevent stagnation and maintain diversity within a population. If all parent candidates have high fitness levels, this can lead to premature convergence of search, since it prevents any real exploration from taking place. Also, once all solutions are strongly similar, then the evolution will grind to a halt. It is in the best interest of the search to have a population of varying fitness.

Tournament Selection Tournament selection is focussed on running small 'tournaments' between a number of chromosomes, with winners being passed on for repopulation. We select a random cluster of solutions based on a specified tournament size n. These n solutions are then ordered based on fitness, and the m candidates (where $m \leq n$) are then selected from the tournament. Traditionally, the best solution is added to set m with probability p. This probability then propagates such that the second best is selected with probability $(p \times (1-p))$ up to $(p \times ((1-p)^n))$ for the poorest solution. The tournaments are then run again and again until we have sufficient numbers of parents selected. In *deterministic* tournament selection - the method we have adopted - once ranking has taken place the best m candidates are always selected. We can reduce the selection pressure - the restricting force of selection that dictates future iterations - in the tournament by increasing the size of the tournament (n) or by increasing the number of solutions (m) retrieved per tournament (Miller and Goldberg).

Elitism Elitism is a rather simple addition to the selection process, typically applied alongside the designer's method of choice. In the seminal publication by De Jong [1975], the selection process is forced to retain n of the best chromo-

somes from the generation. Thus ensuring that the best chromosomes are kept for future generations.

This highlights one of the challenges in evolving solutions to problems: since fitness-proportionate and tournament selection ensure diversity in search, which can lead to not having all of the top solutions in the subsequent generation. In fact we state that having the parent set consist entirely of 'fitter' solutions can cause premature convergence.

So why have two contradictory practices? We feel that it is important to consider the parameters such as population size, number of parents per generation and the number of elite parents selected. Bear in mind we are applying this to an existing selection method. Hence, through elitism we allow the population diversity to maintain through generations but we still want to have some control of how many good solutions are retained.

Crossover and Mutation For the modification process, like other aspects of EAs, there are a variety of different forms of both crossover and mutation. Here we shall provide a quick outline of the canonical form of each with our specific applications explored later in this chapter and again in Chapter 3.

Crossover is perhaps the most defining aspect of the GA yet it is a relatively simple concept in both theory and practice. The basic principle revolves around *schema theory* devised in Holland [1975], where Holland considers each substring of the chromosome as schema or building block. Crossover is then introduced to guide the evolutionary search by building up blocks of correct substrings, resulting in the desired product. In Holland's original GA, future chromosomes were constructed by defining a pivot somewhere within both parents strings which effectively divides the string into two blocks (not necessarily of equal length). Once the pivot is placed, children are created by swapping blocks between chromosomes. This result in two unique offspring that are built from their parents. In Figure 2.3, we see a simple example of Holland's method, typically known as *single-point crossover*. A pivot is defined in the middle of the string (for the sake of the example), followed by a clean swap. When dealing with simple string chromosome representations, it is common to see multiple-point crossover as well as more complex variants.

Mutation is a simpler form of modification, where we make small modifications to the chromosome string. Of course this is strongly dependent on the chromsome representation; while Holland's GA would invert binary digits, different setups

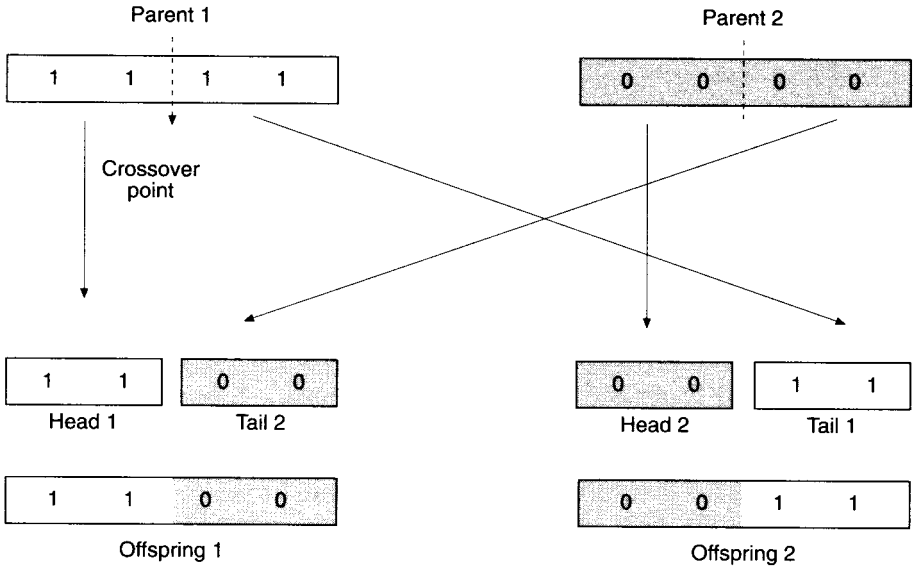


Figure 2.3: An example of single-point crossover from Savic et al. [1995], where a pivot is specified for both strings, followed by a swap to create two new offspring.

will require intelligent and relevant modifiers. Typically mutation can occur under a user-defined probability, it is then a choice of the designer whether each position in the string is mutated or a probability exists for each individual point to be mutated.

It is still open to debate whether mutation is a better approach for modification to crossover or indeed vice versa. A thorough investigation of their capabilities can be found in Spears [1993], where the author attempts to assess the reliability of each method. Spears concludes that while both are potentially very "disruptive" methods[1], he believes that crossover provides a more reliable and effective modification process. Meanwhile, Muhlenbein [1992] contradicts and strongly argues against Spears claims. Arguing that while employing a EA/GA with crossover and mutation is a useful means of parallel search, a traditional ES relying solely on mutation will always outperform it. Essentially, there is a fine line when dealing with crossover and mutation for any GA, where an adequate balance must be reached in order to gain maximum effectiveness.

[1]While it has been argued in the likes of Holland [1975] that crossover is useful in building up good solutions, it is also commonly recognised that both methods can potentially do as much damage to a solution as it can aid it.

Benefits and Drawbacks of Evolutionary Algorithms

The first and perhaps most obvious benefit of applying an EA to a given problem is the parallel search process that filters inadequate solutions. Unlike more traditional search algorithms where we are only assessing a small number of solutions at a given juncture (typically only one), here we now have large numbers of candidates at every step of the search. In fact the number that are explored is dictated only by the researcher responsible. The use of a population not only leads to parallelism in search but it can also then be integrated in execution. If the user is dealing with a computationally expensive task and has the facilities for parallel execution, i.e. a multi-processor computer, this can speed up the search process even further as entire populations take less time to evaluate. EAs are also considered valuable due to their adaptive qualities when searching for solutions.

In the early days of AI, *Expert Systems* emerged by compiling large sets of rules crafted by experts of the respected field. While these systems were good at solving the task prescribed, it was often a very specific problem. Hence the system could not adapt to even mild variations of the problem. In comparison, EAs require only simple models and functions for assessment and improvement. This abstraction ensures that solutions do not converge on a small set of problems. In fact it provides sufficient flexibility to allow for variations of varying complexity to be introduced, relying on the Darwinian paradigm to push towards effective solutions. Of course this flexibility can only be stretched so far, and as a result more contemporary research has explored means to stretch it even further. This relevant research is explored in detail in Section 2.2.

Another key benefit is the abstraction of the agent requirements from the overall domain. Here our genotype only carries that which is relevant to the specific task, allowing us to throw away the excess baggage. Conversely, the translation from genotype to phenotype is often a challenging prospect, where the designer must ensure that the chromosome formulation is sufficiently succinct, yet expressive.

As we previously mentioned, the "survival of the fittest" approach taken by EAs is another key benefit since it means that a solution can easily be found to a given problem, even in more challenging search spaces. This is in part due to the search making no assumptions of the fitness landscape. However this is a double edged sword: as we previously mentioned the fitness function does not require fine-grained criteria to search for solutions. Despite this, sufficient information

must be provided to ensure satisfying results. This can lead to two scenarios; either the search satisfies the stated requirements, but the solution is too 'simple' for your demands, or the search becomes so constrained that it ignores large areas of the potentially useful search space. To ensure diversity and variety we must ensure that a balance is achieved between our demands of the solution and those imposed in the fitness function. Often this will result in intelligent solutions that behave in a manner the designer would not have thought of. Whether this is desirable is of course relevant to the context and, especially in robotics, how acceptable it would be with respect to the domain[1].

Truthfully, the simplicity of the evolution model and process is the greatest benefit and largest drawback, since it is an easy method to begin searching for solutions but is difficult to master. Ultimately, we feel that applying an EA to solve an optimisation problem is less a scientific practice and more of an art form. When conducting our preliminary tests we often see a phase of trial and error as we seek to discover the combination that will guide the search most effectively. One has to consider that there are a large number of parameters or features that have to be maintained; chromosome definition, learning algorithm applied, fitness function, selection method, population/parent size etc. There are a myriad of possibilities where one small decision can lead to drastically poorer results. Continued experimentation will often give a researcher a more well rounded inclination of what will work and what won't. This is not to say that genetic algorithms are a black art, given that several researchers, notably David Goldberg and Fernando Lobo, have made significant contributions in addressing the requirements of EA parameters and proposing some specific methodologies for researchers to employ (Lobo et al. [2007]).

Machine Learning Methods

EAs and EC in general are just some examples of *machine learning* algorithms that exist in the AI field. Machine learning is the class of algorithms that rely on data from the problem domain to gradually improve computer control and decision making through, typically iterative, training. However in deciding on a particular machine learning algorithm, one has to consider whether alternative algorithms are more suited. This can range from the use of supervision in the

[1]For example, an automated car with aggressive cornering and drifting may be acceptable, but a car that drives round a track backwards will be discarded. While it satisfied our fitness requirements, it behaves outwith our defined range of acceptance.

learning process to simply applying different learning models.

The (non) application of a supervised learning algorithm is described in Sutton and Barto [1998] as the contrast between instructive and evaluative feedback. In a supervised learning algorithm we, the user, already know the correct answer to a given situation. Hence the feedback provided to the agent (e.g. fitness function of an EA) by the model is purely instructive. In short, it tells the agent exactly what the action/result should have been based on those inputs. This can lead to a more controlled learning process since the 'feedback' received is far more precise. In fact it can be argued it is no longer feedback, since the signals received are never in the context of the action taken, just whether you were right in that scenario. Furthermore, searching the state-action space is now a non-issue, since we now focus on modifying the features of our solution in order to generate the correct answer the next time a particular scenario occurs. Of course this 'tweaking' is relative to the decision-process being applied, whether it be a decision tree, ANN etc. Conversely, an evaluative feedback algorithm (like most EAs) is often more suited when we do not have instructive feedback available for *every potential scenario that can occur*. In this situation, the feedback evaluates the performance of an agent in that situation. The fitness function in a non-supervised EA is a prime example of evaluative feedback, where we score how well the candidate performed against our criteria. Typically this function will be normalised to provide a smooth fitness space for the search to traverse, giving us a clear indication of which candidate was better than the other. This will then influence the search to look for solutions that hopefully improve on the current best.

One final issue we need to consider is the level of granularity in the evaluation. Training an ANN using an EA to solve a given task is relatively coarse-grained, since the fitness function will score performance against the overall goal (such as navigating an environment to a given point) rather than each individual action. An alternative is to assess the agent at each action it makes, and model the search space around improving each individual step. This is typical of another machine learning paradigm called *reinforcement learning*, where we provide evaluative feedback for *every action made* weighted against a learning rate. Like EAs, there are a multitude of reinforcement learning algorithms, ranging from Monte-Carlo to Temporal Difference Learning (TD). In Monte-Carlo we record the reward of all future-actions from each state to a terminal state[1] and then propagate the reward

[1]Either the goal state, or a state that eliminates the possibility of future exploration.

back through the state-action space, whereas in TD we dictate at what point this value is propagated back. TD(0) refers to the version of Temporal Difference where we propagate back at every step, while Monte-Carlo could be considered TD(n) - where n is the number of actions taken to a terminal state. A common application is the use of TD(λ) where the λ variable allows the user to dictate the depth of reward-propagation. TD(λ) is a common alternative to EAs for agent design, with some debate flaring over which of the two is the most useful method to apply. Work in Lucas [2008] explores the effectiveness of TD in comparison to EAs by assessing the "information rate" of each method, i.e. the number of useful bits of information that can be used by each learning method. The results and argument made by the author show that in certain circumstances, it is far more practical to apply TD to a problem than relying on what is potentially a wasteful EA. This is because TD will harness a lot more of the information throughout the learning process to get to the goal than an evolution approach.

Throughout this thesis we have applied EAs for our learning algorithm for numerous reasons. Firstly as the reader may have realised, an EA has a relatively simple structure and is an easy algorithm to implement. The level of granularity provided by EAs suits our approach, given that we are not too interested in exactly how the agent behaves in order to get to the goal. In fact we have previously highlighted one of the benefits of EAs is the diversity in results that can emerge, which we feel is one of the biggest attractions of the approach. In our previous research we have spent significant time in applying EAs to our agent problems, hence prior to beginning our thesis we already had a substantial amount of existing software that helped us start off smoothly. Furthermore, given that we had yet to apply TD in any of our research, it was still unclear to us whether it would be easy to apply in our problem domains, or whether the diversity in solutions would arise. The idea of a TD-driven approach for our work in the EvoTanks domain (see Section 3.2) is a notion we wish to explore in the near future.

Training Neural Networks with Evolutionary Algorithms

We recall that the ability to modify the behaviour of the ANN relies on the weight vector $\vec{w}$ and to an extent the network topography, since $\vec{w}$ is determined by the ANN's configuration. In order to adapt an ANN to a specific problem, we must find an ideal collection of weights, or even a specific network configuration that, when applied, represents the desired behaviour. However, to explore all

possible configurations of such networking would be a time consuming process.

Applying a GA to search for an optimal network configuration, commonly known as Evolutionary Artificial Neural Networks (EANNs), or simply neuro-evolution is now a seasoned practice amongst AI researchers. EANN can be applied to learn network topologies, transfer functions, input configurations and learning rules. However, the most common application - and the focus of our research - is to evolve the values in the weight vector (Yao [1993]). This is achieved by defining the chromosome representation to be the weight vector of the network. Hence, the evolution now explores the possible combinations of weights that optimise the defined fitness function.

Perhaps the most common EANN application is the field of *evolutionary robotics*, where these principles are applied to the control systems of mobile robots. Typically, the ANN becomes the primary decision making process of the robot and is required to act upon incoming sensor readings to achieve pre-determined goals. By feeding data from the robot sensors to the ANN and with the output neurons corresponding to the actuators, the network will have complete control of the robot once it is correctly trained. To achieve this, we use an EA with a population of weight vectors; which are evaluated by testing them in the robot against a fitness function representing the goals it must achieve. Typically, these evaluations are made in a simulation of the robot's functionality. As the reader may have surmised, this lends to their application in game environments.

Training an ANN is dependent on whether training data is available for analysis. If training data is available, then a *supervised learning* approach is applied, by attempting to improve the accuracy of the network to generate desired output y given input x. This typically requires a back-propagating ANN in order to revise weights based on error calculations. However, if no training data is available then the system will be allowed to search more expansively across all potential solutions. This is a more suitable solution for situations where accruing data regarding the environment may be too difficult to achieve due to time or space constraints (e.g. hand annotating all possible input/outputs for a robot with multiple sensors). In this thesis we do not employ a supervised learning approach, given that it would take a significant amount of time to create all relevant training data. Instead, we rely on the guiding fitness function and assessment-by-evaluation to move our search to more effective solutions.

To modify the chromosome of an EANN there are a variety of methods that can be explored. In our research, we turn to work by Montana and Davis highlighted

in Mitchell [1996] to provide a suitable crossover operator. Given that crossover can be a rather disruptive process, it is best to focus on an implementation that works in harmony with the ANN phenotype. The creatively titled Montana-Davis crossover isolates areas of the chromosome that correspond to the weights of incoming synapses for an arbitrary neuron. This set of weights is swapped with those of a set of similar size, hence we are swapping weights between neurons of similar topology. It is argued by Montana and Davis that this approach proves less disruptive to the parent solutions as it only modifies individual neurons. Our previous experience in applying this method concurs with this argument. For mutation, we apply random 'noise' across the entire chromosome. If a child chromosome is selected for mutation, then the contained genes are visited in sequence. Should the probability of mutation be achieved then an additional value is added to the current gene within the range $-1 \leq x \leq 1$. Where the resulting weight exceeds the imposed range of ± 5 it is then reset to the nearest valid point.

Examples of EANN Application in Game Research

Historically, applying EANNs has proven to be a suitable search method for exploring behavioural spaces. Even in the relatively recent research field of computational intelligence and games there is a significant number of research papers published annually that explore the creation of interesting controllers for small scope problems.

A fine example of iterative development and improvement in games-based control can be found in work in the online 'Asteroids' based space-fighter XPilot. Numerous publications by Matt and Gary Parker focussed initially on weight vector training for fixed topologies, but more recent work found in Parker and Parker [2007] has explored EANN application for both topology and weight vector optimisation. Other work in XPilot, such as Parker and Parker [2008] has explored the evolution of an agent's sensitivity to stimuli, while Parker and Parker [2006] applied a modified GA to operate on distributed systems in an effort to accelerate the learning process. This research has resulted in a small community of XPilot AI developers, culminating in a competition at the CIG 2007 symposium.

Another domain that has generated a vibrant community can be found in the Simulated Car Racing competition. In this domain, developers are challenged to create intelligent reactive controllers for The Open Racing Car Simulator (TORCS); an open-source racing game that provides a programmable interface

for AI drivers. Like XPilot, the car racing competition has been nurtured by iterative improvements and expansions. In this case, significant contributions made by Julian Togelius and Simon Lucas explored different approaches for ANN-driven players. This ranged from applying such learning methods as co-evolution (explored later in this chapter) in Togelius et al. [2007], to comparing the use of an evolved ANN with TD methodologies in Lucas and Togelius [2007]. The Simulated Car Racing community is still highly competitive, with recent competitions at the IEEE Congress on Evolutionary Computation (IEEE CEC) and Genetic and Evolutionary Computation Conference (GECCO) in 2009 and an upcoming event at the CIG 2010 conference.

Another competition domain which has arisen from game-based research is Super Mario, where researchers create reactive controllers to aid the world-renowned plumber in navigating hazardous environments. Whilst a relatively new problem area for research, ANNs have already been considered for application as shown in Togelius et al. [2009]. To date this approach has been moderately successful, but has scope for future improvement.

This is not to say that all research in games has been driven by competition domains. Two notable examples of games research that have received significant acclaim are *NERO* and *Galactic Arms Race*, where in each case the player themselves become involved in the optimisation process. The NERO game, developed at the University of Texas at Austin, is a real-time simulator that allows designers/players to become involved in the learning process by interacting with candidates. This is achieved through a feature-rich interface with the game that allows the user to tweak the algorithm in real-time, coupled with the real-time Neuroevolution of Augmenting Topologies (NEAT) methodology (Stanley et al. [2005]). Meanwhile, Galactic Arms Race developed at the University of Central Florida, is an interesting new multi-player video game that has applied ANNs for content creation. In this domain, variations of ANNs called Compositional Pattern-Producing Networks (CPPNs) are applied in the NEAT algorithm to evolve new types of weapons for players to use, based on their preferences, in real-time (Hastings et al. [2009]).

Further Challenges In each of the aforementioned problem domains, we are dealing with an agent requiring only a small number of actions to respond to incoming stimuli[1]. However, these methods often struggle when more complex dynamics are admitted to the environment or indeed, when the problem is upgraded to a more challenging task.

A simple example of complex dynamics is the game of (Ms) Pac-man, where the trivial - yet socially disturbing - task of navigating through a series of corridors chomping on pills is made all the more challenging thanks to four ghosts haunting the protaganist. Significant time and effort has been spent in exploring the challenges in creating reactive control that can scale to the Pac-man problem, ranging from training a ruleset using an EA in Gallagher and Ryan [2003], Gallagher and Ledwich [2007] to the application of an ANN controller in Lucas [2005]. However it is noted that the majority of this research has been conducted in smaller toy versions of the problem[2]. Lucas's ANN based approach used an input vector of handcrafted data that was selected by the designer. The resulting behaviours were mixed, with adequate behaviour registering in deterministic versions of the game (akin to Pac-man) against a poorer performance in the more complex, non-deterministic circumstances (Ms. Pac-man). It could be argued that one of the most challenging aspects of an ANN-driven approach is trying to generalise the navigation component while considering the ghosts in any circumstance. Results to date suggest that this is an area ripe for exploration which may benefit from a more layered or scaled-up approach.

[1]From our previous examples: incoming fire from enemy ships, turns in the race-track... Goombas.

[2]The irony of a toy version of a game problem does not escape us.

2.2 Scaling-up Behaviours

> What about escalation?... We start carrying semi-automatics, they buy automatics. We start wearing Kevlar, they buy armor-piercing rounds.
>
> Jim Gordon, Batman Begins

As shown in the previous section, there is a variety of exciting and practical applied research in EANNs for games. However, there still remains a significant amount of work to do, notably addressing the concept of adapting to new features that are added to simple problems. These additions either make the original task more challenging, or affect agent behaviour whilst de-coupled from the assigned goal. A simple example of each case could be found in a navigation problem, where adding obstacles makes the task more challenging with respect to the goal, while incoming fire from enemy players adds complexity that is not part of the navigation mandate. In this section, we focus on the range of methodologies applied to scaling-up reactive controllers to address these issues. This, as we previously mentioned, often emerges due to normal methodologies being confined to problems of limited scope. When considering a new learning methodology, ideally the following should be taken into consideration:

- How can we scale our controllers to cope with a larger behavioural search space?
- How do we ensure we achieve the proper behaviour without, unknowingly, pruning large areas of potentially useful search space?
- How do we add new functionality to a controller without compromising the behaviour we have already learned?

Now we shall briefly highlight the breadth of research in addressing these concerns, focussing specifically on the chosen method applied within our research.

2.2.1 Incremental Evolution

Incremental evolution provides a simple yet progressive approach that builds upon previously established methods. Returning to the specification for an EA,

incremental evolution shakes things up by providing additional fitness functions, thus changing the evaluation criteria throughout the learning process. Typically, learning will proceed on an established fitness function until a specified fitness has been reached, at this point the function is swapped out for the next in sequence. The learning process continues as normal but we now have a new fitness function in place. Each fitness function is designed to bring about supplementary behaviour that the designer wishes the agent to harness and can be considered a decomposition of the desired behaviour. The trick of the incremental process is that it can smooth the fitness landscape, having the search navigate unique, and typically simpler, fitness landscapes rather than a larger, more complex one. Work in Gomez and Miikkulainen [1997] gives a strong insight into how incremental evolution can be used for complex behaviours. This is achieved by initial training on simple problems and then subtle modifications are made to increase the complexity of a problem without the population and search becoming lost in the process. This method has proved to be successful provided time is taken for intelligent design, with popular examples such as the NEAT approach mentioned previously (Stanley and Miikkulainen [2002]), to Multi-agent Enforced Sub-Populations (ESP) found in Yong and Miikkulainen [2001].

2.2.2 Modularised Evolution

Another popular extension of the traditional evolution paradigm is *modularised* or *modular evolution*. Here a potential solution is presented from the composition of several individual modules trained in the learning process. These modules are typically complete neural networks themselves, though naturally smaller than a regular stand-alone one, hence each individual ANN must be trained. One of the most important aspects of modular evolution is highlighted in Nolfi [1997] and Calabretta et al. [2000], where it is shown that there is no enforced decomposition for each module, hence each individual component/ANN does not necessarily represent a unique aspect of the complete behaviour. Even a simple navigation example which could be deconstructed as a) basic navigation and b) collision detection and avoidance, may not appear in such a manner in the resulting controllers. Even in the circumstance where there are only two modules, their behaviour may be difficult to classify, yet their combined behaviour would be immediately recognisable.

2.2.3 Co-evolution

Co-evolution is another example of a simple learning methodology that can lead to more effective and robust solutions. Co-evolution's origins can be traced back to the 1980s courtesy of W. Daniel Hillis. Hillis was researching the minimum number of comparisons required for correct sorting networks for any arbitrary size n, an active area that had originally emerged in the 1960s. At the time, research had focussed particularly on networks where $n = 16$, with results of 63 and 60 comparisons made though no proof of optimality was ever provided (Mitchell [1996]). Hillis's initial attempt using a traditional GA with random test cases proved unsuccessful. In questioning the worth of random tests beyond the initial generations, Hillis used an approach where the problems evolved against the solutions and vice versa. This generated a predator-prey dynamic where the prey learns new defences against the predator, while they in turn evolve new means to eliminate their quarry, spiralling into a biological arms race. In this case, the series of solutions acted as predators while the problems acted as prey. This forced the problems to gradually increase in difficulty as the solutions became better in solving them[1](Hillis [1990]). One of the strongest benefits of co-evolution is the ability to avoid the trappings of local maxima within a fitness space, often allowing a strong general solution to emerge. Our previous experience in applying co-evolution found in Thompson [2006] concurs with this argument, where co-evolution applied to our EvoTanks domain resulted in interesting general players that performed better than our previously developed niche players.

2.2.4 Neural Network Ensembles

Neural Network Ensembles (NNE) presented in Hansen and Salamon [1990] have risen in popularity due to their ability to improve ANN generalisation for complex optimisation problems (Sharkey [1996]). As we have previously highlighted, ANNs are an ideal application for generalisation for a variety of problems, particularly in supervised learning where we attempt to reach a compromise that will effectively cover a large set of training data. By creating a generalised solution, in theory it should still prove adequate for data not originally in the data set. In practice this is not always the case, and often the accuracy of the ANN can be called into question. NNEs provide a solution to this problem by essentially clustering

[1]As an aside, this approach gave Hillis a solution of only 61 comparisons, 1 away from the reported best.

a collection of ANNs. Unlike modularised evolution, where each network was part of a whole, the ANNs in an NNE are redundant: each of them provides a solution to the same task, or task component. However each ANN will have been trained using a different learning method to provide a little variety (Shu and Burn [2004]). This setup can lead to an improvement in generalisation and robustness as multiple controllers will be asked to suggest or predict the output of a given input signal. Once all have provided their 'predictions', then the NNE will statistically devise the adequate response. NNEs have been applied in a wide range of challenging optimisation problems such as medical diagnosis, time-series prediction and evolutionary robotics (Sharkey [1999]). Furthermore there are now a variety of defined NNE architectures, with a range of popular methods dependent on the problem. While NNEs are an exciting avenue for researching evolutionary robotics, it has already been noted that they are used primarily for supervised learning methods.

2.2.5 Subsumption

The term *subsumption architecture* was first coined by Rodney Brooks in Brooks [1986] and received much interest throughout a series of publications in the mid-1980s. The principle of subsumption was to provide an alternative form of controller construction to the traditional layered approach commonly used in behaviour-based robotics shown in Figure 2.4. The traditional approach would split modules into control functions that were executed sequentially, only to continue in a cyclic process. Brooks suggested a more modular approach where intelligent behaviours are constructed from subcontrollers tailored to deal with individual facets of the desired behaviour (Figure 2.5). Primarily, it was intended as an alternative to sub-planning execution systems; execution modules that existed beneath control planners that were designed to manage and operate the low-level components of the actuators, as discussed in the somewhat arrogantly titled *'Planning is just a way of avoiding figuring out what to do next'* (Brooks [1987]). Here each controller would operate as an individual reactive perceptron with an intelligent, high-level behaviour emerging from the functional composition of the individual subcontrollers (Brooks [1991, 1992]).

A simple example of a subsumption architecture is shown in Figure 2.6. As we can see the system is broken down into a series of modules that are organised in a top-down order. This order exhibits a hierarchy where higher modules carry

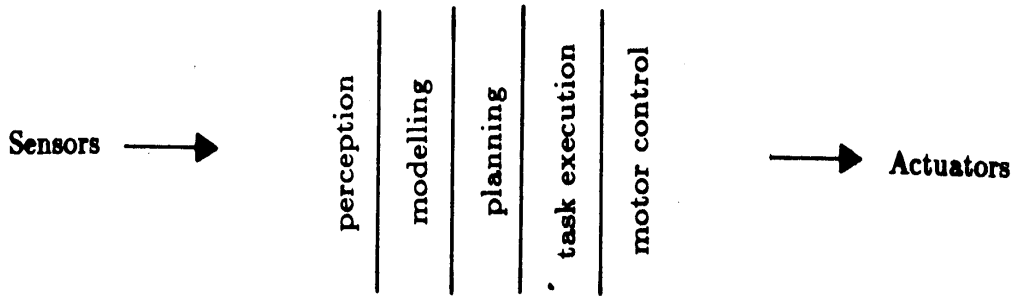


Figure 2.4: The traditional layered decomposition of robot control systems as suggested in Brooks [1986].

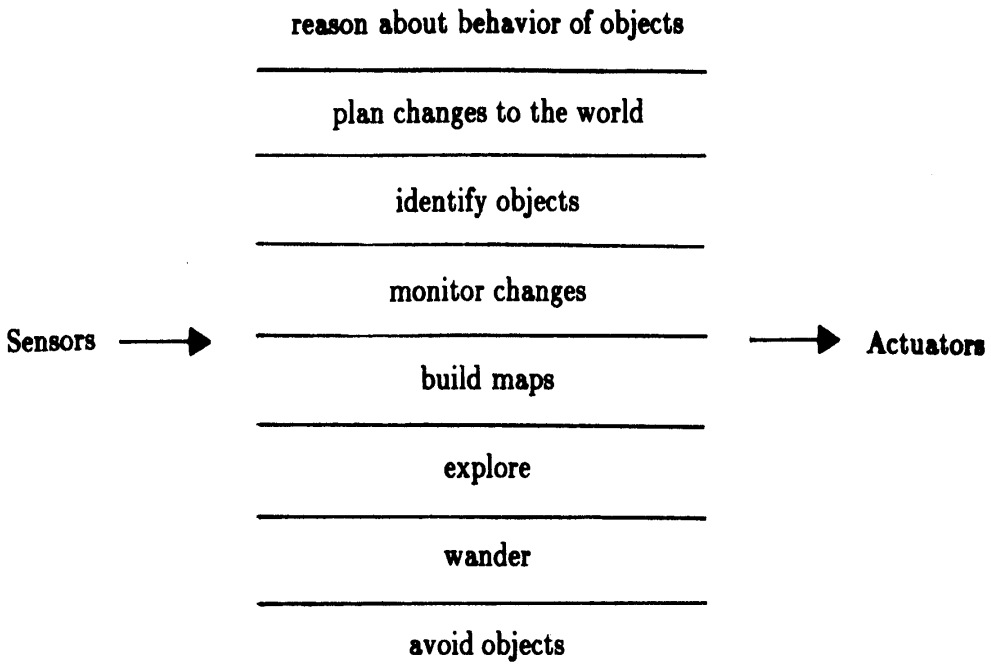


Figure 2.5: A decomposition of the same robot in Figure 2.4, except now it is based on individual behaviours required for the overall behaviour (Brooks [1986]).

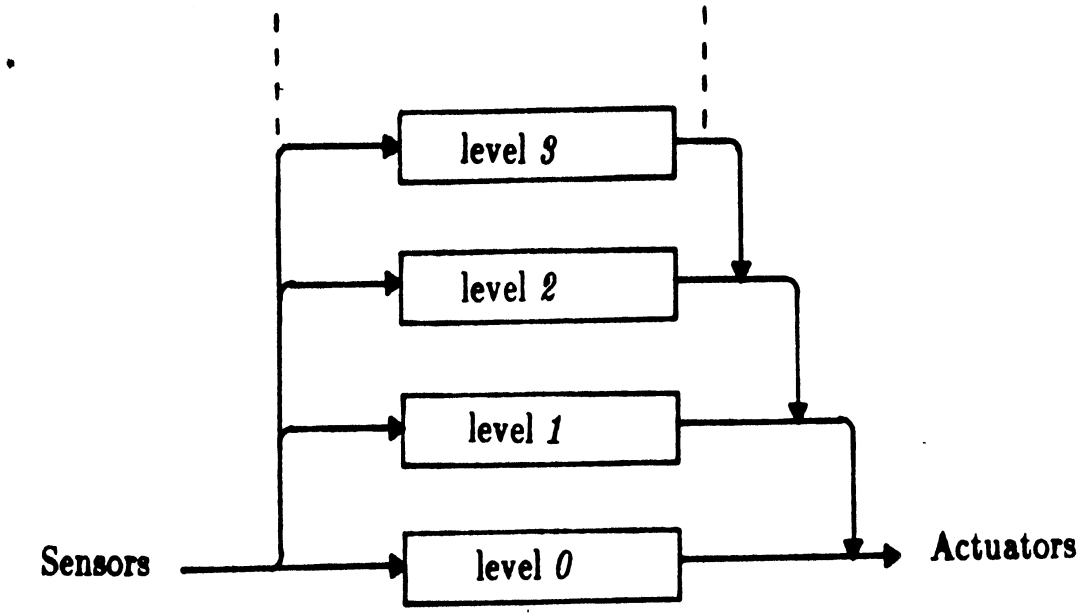


Figure 2.6: A simple subsumption architecture as proposed in Brooks [1986].

a greater priority for execution at run-time. Each module has access to any subset of available inputs from the agent sensors, however whether a particular layer succeeds in sending a signal to the actuators is dictated not by the layer in question but whether any of the n layers *above it* are sending signals.

At every execution-step, the modules are processed in sequence from top to bottom. Should a layer exhibit an output, then it inhibits all layers beneath it from executing. This subsuming behaviour may be used to override individual outputs or entire controllers and is dictated by the designer's preference. The principle behind this feature is that with intuitive design of each subcontroller, the top layers of the architecture would not necessarily need to hardcode particulars of an agent's behaviour; instead they simply know when to place an override command depending on sensor data. The resulting architecture can in theory create more expressive and robust behaviours through the composition of simple components that are trained to override others at the correct time. Brooks envisaged that behaviour construction would emerge from the bottom-up, starting purely in primitive and reactive control, with higher layers composing intelligent behaviour using the controllers beneath it. In recent years this notion has been challenged in the robotics community, with many recognising that situations will occur where a

lower layer must override a higher one. Given that we apply this methodology in our research, we have decided to adhere to a strict high-to-low subsumption in our controller architecture.

The biggest benefits of the subsumption approach are undoubtably the modular approach to behaviour construction and perhaps more importantly the ability to have the controllers read from the sensors and interact directly with the actuators. This means that there is no middleware that constructs a model for the agent to base its decisions on, instead it acts based on real world data. This concept addresses the chief concern of execution with modelled approaches: *we know what needs to be done but how do we go about doing it?*. In Moravec [1983], the author made the observation that, at that time, many model-based architectures required secondary models to compensate for the raw sensor data. Hence the argument made by Brooks to remove the world model, leading to a simple and ultimately very effective approach as shown below:

> ...it often becomes easier to use the world as its own model, and sense the pertinent aspects of the world when it is necessary. This is a good idea as the world really is a rather good model of itself. Continual sensing automatically adds robustness to the system as there is neither a tendency for world model to be out of date, nor are large amounts of computation poured into making sure it is not. Brooks and Flynn [1989] pg.479.

While we would not entirely agree with this statement, it does correlate strongly to our desire to apply an ANN-driven approach. ANNs require continual sensing of the environment in order to ensure robust and effective behaviours for a given problem. A further benefit of a subsumption method is an iterative approach to the development and testing of components within the hierarchy. Again this suited our requirements, since we can then create a layered learning procedure that trains each controller in-turn. This is not to say that the approach does not have drawbacks. Perhaps the most obvious limitation is the number of useful layers that can be applied. In practice we can only really apply a small number of layers atop one another before they conflict with each other too frequently.

These conflicts also impact the flexibility and potential functionality of the architecture, since we cannot have too many controllers with de-coupled goals. Ideally, it is best that we have controllers that facilitate and aid one another. Of course, another drawback is that since we precisely define each controller and how

it operates in conjunction with others, it suffers from poor flexibility at run time. While subsumption has proved to be very popular in the field of behaviour-based robotics, it has received little attention in the evolutionary robotics community. Perhaps the only contribution made to the field prior to our own was the masters dissertation of Julian Togelius in Togelius [2004]. The author made the rather astute observation that applying subsumption to an EA would not only embrace the benefits of both incremental and modular evolution, but also circumvent issues that may detract researchers from applying them. This resulted in a new training method dubbed *layered evolution* where the desired behaviour is broken down into individual ANNs that are trained sequentially. Once a controller has completed training, then the next one is placed on top of it and continues anew. Note that an individual fitness function is present for each unique controller to allow it to achieve its particular mandate. As a result the layered evolution method shifts the focus (fitness function) of training as it progresses like incremental evolution, thus when this shift occurs the controller being trained also changes. This resolves one of the more persistent problems faced by incremental learning, that as the learning criteria shifts the candidate will often 'forget' information it has already learned. The use of multiple ANNs to compose a complete behaviour is similar to a modular evolution approach, however each controller is designed as a specific functional decomposition of the goal. This in turn focusses the training process to specific sub-goals at each layer, drawing similarities to the incremental evolution approach.

2.2.6 Scaled Reactive Control

The methodologies described in this section aim to give the reader a fuller understanding of the variety of methodologies available for scaling reactive control to more complex problem domains. As we shall see in Chapter 3, we embrace the subsumption paradigm for our EANN research and deploy it to improve the reactive control established in our previous work. While these approaches are suitable for more complex domains, there is still no opportunity for deliberation during action selection; in each case we are given means to enhance existing reactive control, either through improved learning algorithms or modular development. Whilst this will improve robustness and functionality we are still reacting to the local environment. We address this lack of deliberation in the following section where we introduce the reader to the field of automated planning.

2.3 Automated Planning

> In preparing for battle I have
> always found that plans are useless,
> but planning is indispensable.
>
> Dwight D. Eisenhower

As previously highlighted in Chapter 1, Automated Planning (AP) is a branch of AI research that focusses on the creation of action sequences to solve tasks. In more general terms it is the ability to model environments and the effect that actions can have within them in a computational structure. By utilising this information, we can create a series of transitions where the world gradually transforms into that which we desire. Resulting plans from AP systems are often applied by intelligent agents such as autonomous robots and unmanned vehicles.

In this technical background we expand upon this brief introduction and explore the the formalisms and methodologies common in Automated Planning technology. We begin by exploring the *Classical Planning* approach; a method suited for single-agent, deterministic, fully observable problem domains. This includes the underlying principles of deterministic planning, the languages used to model problem domains and varying search methodologies. We conclude this section by discussing JavaFF, the classical planner we have chosen to apply in this research.

2.3.1 Classical Planning

Classical Planning refers to a class of planning systems and problems that carry out their search for potential plans under a series of assumptions related to the environment and the fundamental effect that actions will have within it. This is a useful approach to problem solving, since it is common in science to make restrictive assumptions to devise models and techniques to solve the basic problems. This often provides an effective groundwork to scale-up to more complex instances in the future (Ghallab et al. [2004]). In order to achieve this, classical planning relates to the study and exploration of restricted state-transition systems. A formal definition of such a transition system, adapted from Ghallab et al. [2004], is given in Definition 2.

Definition 2 *A restricted state transition system (Σ) used within classical planning is a 3-tuple $\Sigma = (S, A, \gamma)$ where:*

- *$S = \{s_1, s_2, \ldots\}$ is a finite set of States; an abstract representation of the world with respect to the potential tasks that could arise, with each state consisting of a finite set of propositions or atoms that define it.*
- *$A = \{a_1, a_2, \ldots\}$ a finite set of actions; each is a formally defined action that requires some set of conditions to execute (Preconditions) with a correpsonding set of Effects that arise from its completion.*
- *$\gamma : \gamma(s, a) \rightarrow s'$, a state transition function. A function that associates some given state-action tuple $\gamma(s, a)$ with the resulting future state (s') should the action a be committed within state s.*

A restricted state transition system within a classical planning framework must also adhere to a series of assumptions that maintain the consistency and correctness of the system. These assumptions, adapted from Ghallab et al. [2004], are highlighted in Definition 3.

Definition 3 *The restricted state transition system shown in Definition 2 requires a series of restrictive assumptions to maintain a robust and consistent system. The assumptions made with respect to the system Σ are as follows:*

- *Σ contains a finite set of states.*
- *Σ is a fully observable system. The system always has complete knowledge of a given state, with respect to the abstraction used to represent it. Hence no information can be concealed from the planner nor can 'new' information emerge in any given state unless it is the result of a state transition.*
- *Σ is deterministic with respect specifically to the state transition function γ. A deterministic system dictates that should a given action 'a' be applied in state 's' then there is only one possible outcome of applying that action. This is adhered to since the function is a state-action tuple $\gamma(s, a)$ whose only output is one specific future state (s').*
- *No outside factors (known in literature as 'events') can manipulate Σ; only the actions committed by the planning controller affect the state of the system.*

- *A planner utilising Σ only handles restricted goals; a specific listing of state atoms that represent a goal state or are encompassed by a set of goal states.*
- *The resulting solution to the planning problem must be a linearly ordered sequence of actions. This sequence is formulated by the planner's execution. Specific constraints on states to be avoided or specific trajectories through the state-space cannnot be specified. The planning system must be allowed to solve the problem without interference or user-defined constraints.*
- *Actions have no measured duration; the transition from one state to another through a committed action is instantaneous. This is referred to as implicit time and is modelled specifically within γ, since it contains no explicit representation of time taken.*
- *Planning is carried out offline, i.e. the system carries out the complete planning process using the model representation of the environment expressed by Σ prior to execution of any actions. Dynamic changes that occur within the environment during the planning process are ignored until planning is completed.*

The restricted state transition system provided and the list of assumptions it must adhere to are adequate for a classical planning platform. The term classical planning is found in literature to classify planning platforms that focus on deterministic, static, finite and fully-observable environments. In older texts this is often referred to as *STRIPS planning*, a reference to the STRIPS planner and domain representation language that emerged from seminal publications in this area, notably Fikes and Nilsson [1971]. We specifically discuss the STRIPS planner and representation in Section 2.3.2.

Given the definition of the state transition system, we express a planning problem in a similar mannner. A symbolic representation for a classical planning problem is defined in Definition 4.

Definition 4 *A planning problem for a restricted state transition system - $\Sigma = (S, A, \gamma)$ is defined as a triple $P = (\Sigma, s_0, s_g)$ where:*

- *s_0 is the initial state of the problem, a set of atoms and propositions expressing the state from which the search commences.*
- *s_g is the goal state, a set of atoms and propositions that represent the desired future state that the planner must reach.*

The resulting solution to the problem is a sequence of actions $\alpha = (a_1, a_2, \ldots, a_k)$, each corresponding to a particular state visited within the state space $\sigma = (s_0, s_1, \ldots, s_k)$, such that applying an action from α to the corresponding state within σ will result in a future state according to the state transition system $\gamma(s_{n-1}, a_n)$. This leads to a natural progression through the state space such that $s_1 = \gamma(s_0, a_1), s_2 = \gamma(s_1, a_2), \ldots s_k = \gamma(s_{k-1}, a_k)$ where s_k is the goal state.

One of the key benefits of the restricted state transition system is that it can be said to be *Markov* or have the *Markov property*. The Markov property for an agent dictates that once it absorbs a piece of information, it can summarise this data and retain it for future use. Classical planning frameworks adhere to the Markov property since they rely on formal definitions of state and retain any knowledge we have of this state. Should the system move from an arbitrary state *s* to a successor *s'*, then all relevant information is retained and transferred to the successor. This in turn allows us to summarise all relevant information of the world and in time - when approaching planning problems - allows us to summarise past actions and events. Put simply, if we observe any (reachable) state in the state-space, we can create a path to the initial or goal state using the existing propositions and the transformations permitted by actions. Of course, a lot of information is lost along the way, since the agent's continual interaction with the (modelled) environment instigates change, hence facts of the world must be deleted. However, if we are at any state in the middle of the planning problem, given our framework we can then trace our history from the initial state to our current state. This is beneficial in sequential decision making, since systems that exhibit the Markov property allow decision making to ignore previous history, thus eliminating the need for a complex representation to express prior decisions. Furthermore, selecting actions as a function of a Markov state is just as good as a function of a complete action history (Sutton and Barto [1998]).

Given Definitions 2 through 4, a restricted, abstract yet concise planning platform is defined. These are the cornerstones of the classical planning framework and allow researchers to focus on the two most important aspects of planning: representation and search. A language must be defined that can adhere to our restricted state transition system while generalising across a range of potential problem definitions that may be expressed by the researcher. Following this, an effective search algorithm must be deployed that will traverse the state space formulated by our definition language and generate plans.

Given that the research within this thesis focusses on classical planning, we highlight the significant areas of classical representation, followed by a brief breakdown of the planning systems that incorporate these models. We focus specifically on the planning platform used in this research, the *Fast Forward (FF)* planner.

2.3.2 Classical Planning Representations

Representations for planning systems vary depending on the scope of research and the granularity of plans that designers wish to generate. The majority of these representations have been derived from those used in classical planning. Planning representations or languages are by necessity restricted, a design choice that may inhibit the expressiveness of the languages. This in turn restricts the number of possible solutions to a problem. Initial thoughts would suggest this is an undesirable feature, however not only will it restrict the number of potential solution states, it reduces the search space the planner would traverse. This is highly beneficial since we wish to guide the logical inference within the search as effectively as possible without constraining its capabilities. Not only must the language be suitably expressive, but also be pliable to an extent that a planning algorithm can be tailored to process the language and search within its defined constraints.

STRIPS

One could argue that the STanford Research Institute Problem Solver (STRIPS) language is the father of modern planning languages. The STRIPS representation was the language incorporated in the planning program of the same name developed by Richard Fikes and Nils Nilsson (Fikes and Nilsson [1971]). While the STRIPS planner has long since been superseded, the STRIPS language is still a valuable research and teaching resource for problem modelling within planning frameworks, with a number of variants and extensions that have since permutated throughout the planning community (Russell and Norvig [1995], Ghallab et al. [2004]).

Formally, STRIPS is a set-theoretic representation, i.e. it relies on a series of propositional statements that provide an abstract model of the world by stating whether some object or variable carries a specific quality. Naturally these symbols are labels expressed using natural language that the designer can understand. For example, a proposition to represent you are reading this chapter could be

expressed as *ReadingChapter(LiteratureReview)* or *Reading(ThesisChapter2)*. In these examples the propositions have been grounded with a specific value; however in planning we can search for particular values. This use of natural language provides a greater readability than the use of a simple symbolic representation. Given our definitions of the state-transition system and planning problem in Definitions 2 and 4, we do not formally define the states that will arise beyond those that exist at the start and end of the plan. A set-theoretic representation allows us to avoid the trappings of a state-transition representation, where we would formally label each possible state that could emerge (Ghallab et al. [2004]). This would make state recognition an arduous process. Hence the use of natural language provides a greater range of expressiveness, since the state's meaning becomes far more obvious to the casual observer.

To formally define a STRIPS problem, we refer back to Definition 4, where we require an initial and goal state, thus identifying the starting position for our search as well as the situation we wish to reach in order to generate an appropriate plan. To achieve this, a series of grounded propositional statements are provided for both initial and goal state, giving us a clear indication of what is required. Note that extra propositions may arise during planning as actions are applied; however we need only be concerned with the goal state definition. Provided these specific statements are a subset of the final state in the plan, then the planning process has proved successful. These definitions are then coupled with the operators or *actions* that can be utilised in the planning process. STRIPS actions require two specific definitions; *preconditions* that represent the series of conditions that must be true (or not true) in order to execute a particular action, and *postconditions* (typically known as effects in planning literature) that represent the change to the world that arises from execution.

An example STRIPS definition can be found in Figure 2.7. In this simple problem, we require what will ultimately be a 1-step plan to move an agent from location A to an adjacent location B. This is formally defined in the initial and goal state of the problem. Furthermore we provide an action that facilitates the movement between these locations. This action states that in order to move between two locations (X and Y), the agent must be at the initial location and an added constraint that the two locations must be next to one another. The adjacent definition helps us add greater detail to the model and similar definitions can add necessary constraints that more explicitly define the environment being explored. Note in the postconditions of the *move* action, we negate the condition

initial state: at(A), adjacent(A,B)
goal state: at(B)

actions:

move(X, Y)
preconditions: at(X), adjacent(X,Y)
postconditions: not at(X), at(Y)

Figure 2.7: A sample problem defined in the STRIPS language that requires a 1-step plan to move from location A to location B. Note the use of predicates that give natural language explanations of their function leading to easier understanding of the action definitions.

---

that our agent is at the initial location. This is referred to in literature as a *delete effect*, where we must negate a specific statement, and maintain it within a list of atoms that are also considered not true in the current state.

The notion of negative atoms in the STRIPS planning language is rather deceiving. Any atom that is not explicitly described in the current state description is assumed false. This can cause problems since in certain circumstances the user may not have a complete definition of the initial state (typically a circumstance that arises due to a mistake by the designer). The assumption that all other atoms are false may later impede the planning process in circumstances that are not immediately obvious to the designer.

There are several extensions that have been applied to the STRIPS formalism prior to the introduction of more contemporary languages. These range from the introduction of quantified expressions (e.g. a robot can carry n items), existential quantification in goals and the introduction of typed variables.

Planning Domain Description Language (PDDL)

The Planning Domain and Description Language (PDDL) can be considered a natural extension of the classical (STRIPS) representation. PDDL was initially developed by Drew McDermott as an attempt to standardise planning representations throughout the Automated Planning (AP) community (McDermott et al. [1998]). Defining features of the language came in the form of typed variables that constrain the arguments of predicates and actions, negative preconditions, conditional effects and the use of quantification in expressing pre- and post-conditions.

Many of these features were initially suggested as part of the ADL representation found in Pednault [1989]. These features are all achieved while maintaining the Lisp-like structure and style of the STRIPS representation. Once this standard was formed, it would provide the formal representation that would be used at the International Planning Competition (IPC). Since its unveiling in 1998, PDDL has become an active research endeavour in itself, with a variety of publications focussing on new extensions to expand the expressive capabilities of the language. PDDL requires two separate files for execution: a *domain file*, which describes the features of the environment and the actions used to interact within it, coupled with a *problem file*, that simply defines an individual problem instance complete with initial and goal state. We see an example domain definition in Figure 2.8, where we have incorporated the same functionality as the STRIPS example in Figure 2.7. A keen reader will note supplementary information added to the domain definition. As mentioned above, one of the defining features of the language is the introduction of typed variables, unlike STRIPS where the actual identity of a constant is implied from its use. For example, in STRIPS we can identify the location of our agent in the environment as location A by stating *at(A)* (as shown in Figure 2.7). However there is no definition within the problem that states A is a location. Instead we run on the assumption that since it has been used in this predicate, then it must be a location. In PDDL we can go a step further by introducing a type for the constant. Referring back to Figure 2.8, we observe that the concept of a location is now strictly typed. Hence it enforces the constraint that we only apply a location object in the *at* and *adjacent* predicates. This has no effect on the expressive power of the language, however it provides a clearer understanding of the relationships intrinsic to this domain. Similar functionality *can* be achieved in STRIPS, however it is far more concise using these typed variables.

A sample problem instance is shown in Figure 2.9, where we now define the same problem instance as that found in Figure 2.7. Note that we must now define the domain that this problem refers to and explicity define the types of objects used. As we have previously stated, this does not provide any more expressive power for the planning system; however it increases the readability and ease of understanding of the problem. This PDDL problem is relatively easy to read and understand through the use of defined location objects and the predicates applied.

Given the prominence of the PDDL language at the IPC, it is unsurprising to note the wealth of extensions and continued attention it has received within the AP

```
(define (domain sample_domain)
        (:requirements:typing)
        (:types location)

        (:predicates
                (at ?l1 - location)
                (adjacent ?l1 ?l2 - location)
        )

        (:action
                move
                :parameters(
                                ?startLocation - location
                                ?endLocation - location)
                :precondition (and
                                (at ?startLocation)
                                (adjacent ?startLocation ?endLocation))
                :effect (and(not (at ?startLocation)) (at ?endLocation))
        )
)
```

Figure 2.8: A PDDL domain specification akin to the STRIPS example in Figure 2.7, where we need only define the objects in the domain and the applicable actions. Note we introduce a *location* type that as a result constrains the *at* predicate to use only location variables within problem instances. While similar functionality is achievable within STRIPS definitions, PDDL provides a more effective means of modelling this constraint.

```
(define (problem sample_problem)
        (:domain sample_domain)
        (:objects
                locationA - location
                locationB - location
        )

        (:init
                (at locationA)
                (adjacent locationA locationB)
        )

        (:goal (and (at locationB) ))
)
```

Figure 2.9: Code from a PDDL domain problem file. The simple example provided is designed to reflect the problem defined in Figure 2.7. Given the new typed constraints of the domain as shown in Figure 2.8, we must now explicitly define the objects in the environment.

community. One of the first major steps taken with the language was introduced at the 2002 IPC, where the challenge of handling time and numeric resources was proposed to the AP community. To ease this transition, Fox and Long [2003] introduced a series of extensions into PDDL 2.1. This version introduced an explicit model of concurrency into domain definitions: a representation for time that allowed for durative actions to be expressed under certain restricting assumptions. These durative actions also permitted the use of discrete or continuous effects (in a limited capacity), where an effect could be expressed as occurring during or after the completion of the action. It was observed by the authors themselves in Fox and Long [2003] that the expressive capabilities of PDDL 2.1 in fact exceeded the capabilities of planning systems at the time. However it is clear that this unveiling had significant impact on the community, not only due to its integration into future IPC events but also the reaction of fellow researchers. Within the first year of the release of PDDL 2.1, many of Fox and Long's contemporaries within the community, notably Hector Geffner and Drew McDermott responded with opinion pieces to critique and discuss the changes that were introduced.

McDermott's commentary found in McDermott [2003] addressed his concerns over the semantics and syntax of durative actions in PDDL 2.1. An example of this was that functions are permitted within the language; however they could

only incorporate non-numeric arguments.

> A paradigmatic example is (amount-in tank1), which might denote the volume of fuel in tank1. A term such as (object-at-distance 3) is not allowed. Why these restrictions? Because many planners eliminate all variables at the outset of a solution attempt by instantiating terms with all possible combinations of the objects mentioned in the problem statement... (it) means that general functions can't be part of the language. If we had a function $midpoint : Location \times Location \rightarrow Location$ then it would generate an infinite set of terms such as $(midpointloc - a(midpointloc - bloc - a))$. McDermott [2003] pp.145-146.

While McDermott agreed that functions were an important feature to be incorporated into the language, his reservations rested on the restricted syntax and complex semantics that arose in an effort to accommodate contemporary planning systems. Ultimately he feared that PDDL may restrict the progress of the community when future advances in the language were becoming restricted in an effort to provide for existing systems.

Meanwhile Geffner produced a more complimentary response found in Geffner [2003], where his area of discussion focussed on the application of time, resources and concurrency. Notably that a lack of explicit resource robs the language of a simple and clean form of concurrency.

> ... in PDDL 2.1, as in Graphplan, the level of concurrency is defined implicitly in terms of the *syntax* of pre and postconditions. ... In any case, the account of concurrency based on action interference as defined in Graphplan and PDDL 2.1 carries certain implicit assumptions and the question is whether we want to make those assumptions, and whether they are reasonable or not. Geffner [2003] pp.141-142.

A final concern was the intent of PDDL 2.1 to have an influence on research within the planning community, in essence trying to pull the community away from 'toy' domains and into the realm of realistic applications. While Geffner agreed that this kind of progress was important, he sought a cautionary approach to ensure that these toy domains are retained as an important research avenue, the argument being that while they are conceptually simple problems they have not

only proven to be computationally challenging but also provided the community focus.

Despite the reservations of some researchers (not just Geffner and McDermott), PDDL 2.1 has continued to be embraced by the AP community and is perhaps one of the most important additions made to the language. It has provided a highly expressive language that serves as the standard and is more than adequate for creating a range of unique and diverse problem domains.

However this was not the only major contribution presented by Fox and Long, with a latter implementation named PDDL+ in Fox and Long [2006]. This latter iteration was based on the PDDL 2.1 framework and introduced means of modelling autonomous processes that are generated and triggered either by the planning agent or a similar process, addressing an issue found within PDDL 2.1 that the application of continuous durative actions relied on only the planning agent itself to be responsible for any change occurring within the world.

To date three more extensions have been made from the groundwork presented by Fox and Long, with PDDL 2.2 introducing state variables that can be computed as a function of other variables and timed initial literals; facts that will become either true or false at time points known in advance (Edelkamp and Hoffmann [2004]). Meanwhile, PDDL 3.0 allowed the user to refine the plan-solutions through a series of contraints and preferences (Gerevini and Long [2006]).

While these extensions have proven valuable, there is a price to pay for adding expressiveness to the language. Namely, additional time is required in order to compute a plan. In our research, we need the planning process to be as efficient as possible. Therefore, it makes sense to adopt a 'simple' and abstracted model of the world and make the resulting plans feasible through robust execution. Hence, for the domain definitions explored and utilised in Chapters 4 and 5, we are content with using the subset of PDDL 2.1 that is applicable in the JavaFF planner.

2.3.3 Fast-Forward Planner/Java-FF

Here we take a look at the Fast-Forward (FF) planner and JavaFF - a Java implementation of Fast-Forward - since this is the planning platform we have chosen to conduct our research. Given the scope of our research and the manner in which the planner is applied (shown later in Chapter 5), it is not necessary for the reader to have a concrete understanding of the search mechanics employed. However for the sake of completeness, we dedicate time to explore the history and

significance of these systems and a brief overview of the inner workings.

The FF planner is a domain-independent planning system developed by Joerg Hoffmann. Developed in the C programming language, it is capable of handling a range of different planning problems through PDDL. FF has proven to be a highly effective planning system, scoring as the most successful automatic planner in the second International Planning Competition, held at the fifth International Conference on AI Planning and Scheduling (AIPS'00) (Hoffmann [2001]). Future iterations of the system then competed in the 3rd and 5th IPC in the metric and conformant tracks respectively.

Forward Search & Heuristics

FF is inspired by the Heuristic Search Planner (HSP) developed by Blai Bonet and Hector Geffner (Bonet et al. [1997]). HSP relied on the use of the forward-search approach shown in Algorithm 2. Forward-search behaves as the name would suggest, by moving forward from the initial state typically by selecting applicable actions non-deterministically and generating the new state using the transition function. If the resulting state is our desired goal state, then the search can terminate.

One notable difference in the version of forward-search employed in HSP is the use of a guiding heuristic. A *Heuristic* is a function that ranks all successor states based on locally (heuristically) available information about the problem. Hence they operate as a 'rule of thumb', providing a general strategy for solving the problem by guiding the forward search in the best direction. Furthermore, it contains only the list of nodes that have been generated but not visited, hence no nodes will be visited twice in the search. This leads to an *informed search*, where the agent now makes an intelligent decision about where next to explore in the state-space. When we are dealing with search problems we are often interested in finding the shortest path from initial to goal state. Therefore, the heuristic is interested in estimating the shortest path to the goal state. One important aspect of the heuristic is that it will reduce the branching factor of the search-space, i.e. it will reduce the number of possible actions that may be taken since the heuristic will score certain branches very poorly. Conversely, it is important that the heuristic does not overestimate the distance to the goal. It is best if the heuristic estimates a state distance to be less than or equal to the actual distance. If the heuristic never overestimates the path length, then it can be

Algorithm 2: An outline of the forward-search algorithm (Ghallab et al. [2004])

Input: The initial state s_0, a series of operators O and the goal state s_g
Result: The resulting plan π that reaches from s_0 to s_g

```
1  Forward-search(O,s_0,g)
2  s← s_0
3  π ← the empty plan
4  for ∞ do
5  |  if s satisfies g then
6  |  |  return π
7  |  end
8  |  applicable ← {a | a is a ground instance of an operator in O, and
   |  precond(a) is true in s}
9  |  if applicable = ∅ then
10 |  |  return failure
11 |  end
12 |  non-deterministically choose an action a ϵ applicable
13 |  s← γ(s,a)
14 |  π ← π.a
15 end
```

considered *admissible*. An admissible heuristic will ensure that the agent explores all available and valid areas of the state-space, since if the heuristic overestimated, then it can potentially block successor states from being explored.

The FF Heuristic

FF, like HSP, is reliant on a relaxed heuristic that estimates the distance of the current state to the goal. A relaxed heuristic will remove certain assumptions of the problem in order to give a rough approximation of potential plan length and cost. In both FF and HSP this is achieved by refraining from deleting the preconditions of actions after they are selected. While this results in a situation where contradicting facts can occur (e.g. a robot is in location A and B), this relaxation improves performance in complex domains by providing heuristics that, while not admissible, provide useful information about the domain in question (Hoffmann [2001]). The lack of an admissible, relaxed heuristic is due to the search complexity for calculating the length of an optimal, relaxed plan being

NP-hard[1] (Bylander [1994]). Fortunately work found in Bonet et al. [1997] introduced an effective means of approximating the relaxed solution length by trading admissibility for more informative heuristic values. This is achieved by computing weight values for all facts in the relaxed problem based on their distance to the goal, assuming that all facts are independent. FF however removes this assumption and relies on a new heuristic that prunes the potential search space, resulting in a heuristic that for any given state, computes the length of a shortest plan using the relaxed planning task.

Now of course the challenge is how to represent this relaxed plan state space. Since no action preconditions are removed, any state can have a potentially massive number of facts and can often be easily confused with one another to the casual eye. Hence the relaxation heuristic is applied to plan-graph techniques to provide a simple yet highly effective reachability analysis. As a result, the metric is commonly known as the Relaxed Plan Graph (RPG) heuristic.

One final aspect of the RPG heuristic is that it can be applied to find actions that may lead to what could be deemed 'more useful' states when building the plan (Hoffmann and Nebel [2001]). When the search process is conducted, this information is used to prune the successor states that are visible during the search process.

Planning-Graph Techniques

Planning-graph techniques provide a different approach to visualising the state-space we have thus far associated with planning methodologies. In the classical planning definitions we have used throughout, we consider the problem as a connected state-space, with the resulting plans being a sequence of actions that traverse from a starting state to an end state. However, planning-graphs are interested in showing the *Reachability* of a goal in a planning problem, i.e. whether the intended goal state is in fact reachable from the initial state. However reachability in planning problems cannot be computed tractably (Ghallab et al. [2004]), thus plan-graphs rely on a relaxed reachability estimate by suggesting series of layers that express the planning problem. Fact layers indicate all possible propositions based on any combination of actions executed in preceeding states.

[1]NP-hard = non-deterministic polynomial-time hard: a class of computational complexity. A problem is NP-hard if the algorithm that solves it can be translated to solve any previously defined NP problem. To put it simply, an NP-hard problem is "at least as hard as any NP-problem." Weisstein

Hence in the very first fact layer (fact-layer-0), we can only see the propositions that exist at the initial state. The subsequent fact layer is then separated by the first action layer, which shows all possible actions that can be committed based on the propositions found in the preceeding fact layer. The effects of these actions are then added to the subsequent fact layer, showing all possible facts that could exist if any action is taken from the initial state, including taking no action at all. This leads to the situation where conflicting facts exist within the fact layers, therefore mutual exclusion (or mutex) relations must be included in the fact layers. These mutex relations indicate the pairs of facts that cannot possibly exist within a given state, with these relations being propagated across action layers. Once a fact layer exists that contains all propositions that reflect the goal state with *no mutex relations across them*, the graph need no longer expand and the search for a solution can commence.

A simple example of a planning-graph can be seen in Figure 2.10, where we have taken our PDDL domain and problem files shown in Figures 2.8 and 2.9 and modelled the graph to the point the goal is reachable. The initial fact layer *(fact-layer-0)* shows the propositions active at the initial state. From here we see the propositions required (denoted by directed arrows) that activate the 'move' action in *action-layer-1*. This in turn creates the *at(locB)* proposition while deleting *at(locA)*. Therefore the red marker is used to signify a mutex relation, indicating that these two propositions cannot exist together. Given that all propositions of the goal state from Figure 2.9 exist in *fact-layer-1* and no mutex relations hold between them, a search can then begin to create the simple 1-step plan of this example.

The first system that was capable of exploiting these reachability graphs was (perhaps unsurprisingly) GraphPlan (Blum and Furst [1997]). GraphPlan relies on the knowledge that we can compute the reachability graph to generate the facts encompassed by the goal state. Using this it can then traverse the graph *backwards* to reach the initial state and create the sequence of actions that will become our desired plan.

Searching for Solutions

As shown previously, the RPG heuristic provides a value of reachability based upon the relaxation of the problem. However, despite being able to compute this value in polynomial time, the process still proved too costly when assessing large

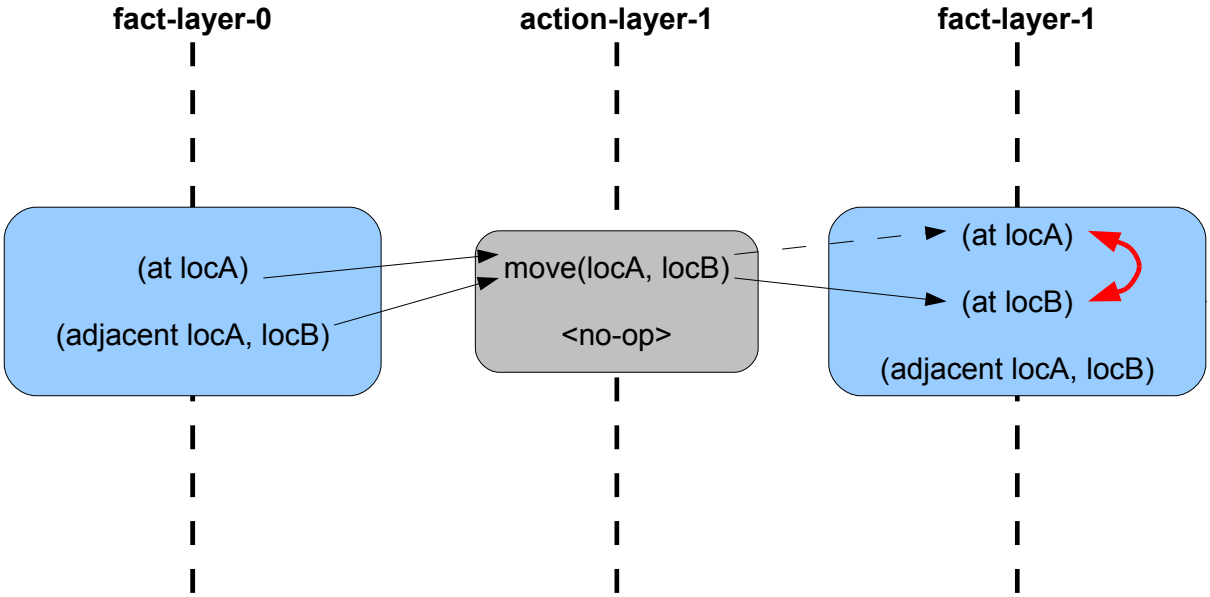


Figure 2.10: A simple example of a planning-graph inspired by the PDDL domain model and problem used in Figures 2.8 and 2.9.

state-spaces. As in HSP, a compromise is made by using a simple local-search algorithm. HSP was reliant on the use of a Hill Climbing (HC) algorithm, a greedy local-search algorithm when applied to planning in its simplest form behaves like the aforementioned forward-search algorithm shown in Algorithm 2, however now the applicable actions are dictated by whether their heuristic value is better than the current best. Meanwhile, FF was based on *enforced* HC; a hillclimber that operates a more strict successor selection policy[1] (Hoffmann [2001]). If we return to Algorithm 2, we will note that an action is selected at random from the set of all applicable actions. This is how the most primitive form of HC operates, provided the successor has a better heuristic value. In enforced hill-climbing however, the action is dictated by finding the successor state with the *best* heuristic estimate, but only provided it is an improvement on the current states value. This selection is achieved using the breadth-first approach, whereby the complete set of potential successors is taken from the current state and are assessed sequentially. As we stated previously, the RPG heuristic is also used to recognise the useful actions for achieving goals in the relaxed problem. When the set of potential successors is created, the only states observed are those that are earmarked by the heuristic,

[1] Also referred to in literature as steepest-ascent hill climbing.

thus preventing the system from visiting potentially attactive yet misleading states. Once any state is found with the better (smaller) heuristic value then it is immediately selected and replaces the current 'best' state.

However, this can lead to a potential situation where there are no successor states whose heuristic value is greater than or equal to the current best, therefore there is no immediate action the algorithm will take. This is commonly known as a *Plateau* in the search space. In the event of reaching a plateau. FF will apply a breadth-first search that checks all successor states at each 'depth' of the search space[1]. Furthermore, this approach removes the 'useful action' constraint from the RPG heuristic. The breadth-first search encounters new states and applies the RPG heuristic to each until a state is found with a better heuristic value. Once a suitable successor is found, then the path to that state, which may be one or more actions, is added to the current plan and the search resumes as normal.

Java-FF Java-FF is as the name implies, an implementation of Hoffmann's FF planner developed in the Java programming language. It was developed by the team of Andrew Coles, Amanda Coles, Maria Fox and Derek Long at the University of Strathclyde. Ultimately the purpose of this was to provide an effective teaching tool for undergraduate students participating in AI practicals (Coles et al. [2008]).

JavaFF replicates the structure of Hoffmann's FF planner using the source code from the previously established temporal planner/scheduler *CRIKEY* (Coles et al. [2009]). In order to successfully replicate the performance of FF, Coles ensured that the forward-chaining enforced HC was applied to the state space while using the RPG heuristic, with the additional breadth-first search method in the event of plateaus. This version does not consider many of the extensions made to FF and focusses on recreating the core functionality while ensuring it can operate as a useful open-source teaching and research tool.

2.3.4 Advanced Planning

In our research we have focussed entirely on classical planning systems and representations. These methods have proved to be suitably adequate for dealing with fully observable, deterministic planning problems. However this is not to say that this is the pinnacle of research within the AP community. In fact it is

[1]For further reading on basic state-space techniques, please consult Russell and Norvig [1995].

merely the tip of the iceberg, as researchers have continued to explore the notion of plan-construction under more constricting circumstances. This ranges from having to consider the overall time taken to execute a plan to plan-creation while not having a good understanding of even what state you are in. For completeness we briefly highlight the range of research beyond classical planning.

While the use of planning systems and algorithms developed by AP researchers has proven effective, there are alternative approaches that can be made to finding solutions. One of the most notable is the use of Boolean/Propositional Satisfiability (SAT) notations, which encode a planning problem as a series of propositional statements. The challenge then being to find a means to satisfy all conditions in the formula. A *satisfiability decision procedure* is then responsible for determining whether the problem is satisfiable. If this proves to be the case, the system extracts a plan based on the assignment process of the decision procedure. This is perhaps best shown in the planner called *Blackbox*, which takes the concept of planning as satisfiability and applies it to STRIPS models in Graphplan. Furthermore, empirical evaluations of Blackbox suggest that existing SAT solvers are capable of solving benchmark planning problems competently (Kautz and Selman [1999]).

Another popular approach is the encoding of a Constraint Satisfaction Problem (CSP), which considers the planning problem as a series of variables that must be given a value within a specified domain. However, as the name would suggest, there are a series of constraints that exist across the variables. These can range from simple boolean constraints (e.g. $A \neq B$ or $A > B$) to the use of real-values (e.g. $A \geq 5$). In the case of boolean constraints, then similar approaches used for SAT problems can be adopted, while more real-valued expressions can be approached from an linear or integer programming perspective. This approach could be argued to be more popular than SAT techniques, with CSPs again carrying a large existing community. Furthermore, they have been heavily adopted in heuristics and search processes for planning algorithms (Ghallab et al. [2004]). Like Blackbox, similar ventures have explored translating planning graphs into CSP problems. In fact it has been argued that the CSP approach outperforms SAT encodings in both time and space complexity (Do and Kambhampati [2001]).

One of the most prominent areas of modern planning research is the application of temporal planning. In this field, the assumption of discrete-time steps made in Definition 3 is broken. Hence each action now has a specified duration, providing another metric that must be reduced. This revelation also impacts the behaviour of preconditions and effects of an action, since we may require a specified situation

to not only hold true at the beginning of an action, but remain throughout the duration. Temporal planning also introduces the notion of scheduling, where we can now assume that a plan may execute as a series of parallel processes rather than a sole series of sequential actions (Ghallab et al. [2004]).

Finally, another important area of planning research breaks more of our classical planning assumption, whereby uncertainty occurs in the planning model. Uncertainty can be introduced through non-determinism in actions and partial observability of the world. In non-deterministic actions, actions may result in one of a number of possible states, while partial observability removes the ability to see all relevant information about a given state. This results in a planner having to rely on a 'belief state', where the system runs on an assumption that it is in a state based on observations from the environment. Managing this state recognition and planning throughout is a challenging area of planning research and receives much attention from our peers in the fields of engineering and robotics (Ghallab et al. [2004]).

2.3.5 Planning Applied to Games

To conclude this section, we highlight some of the most renowned applications of planning in games in recent years. However, to carry out such a research review is rather taxing. This is due to a lack of agreed terminology amongst researchers. There are many examples of applied 'planning' in games that, if observed from the perspective of some planning researchers, would be considered nothing more than the application of an informed search algorithm, such as A*. Planning systems typically differ from informed search algorithms in two key ways; the application of domain knowledge to inform heuristics, and the use of a set-theoretic language such as STRIPS or PDDL. Hence for this review, we have focussed on research that maintains some of these planning concepts.

One of the most renowned examples of planning applied to games in recent years is the work conducted by Jeff Orkin. Orkin, a Ph.D. researcher and former game developer, introduced a new approach to game-AI known as Goal Oriented Action Planning (GOAP). The GOAP approach was introduced in Orkin [2006] as means to create intelligent opponents for the first-person shooter game *F.E.A.R.* developed by Monolith Productions and released on the PC, Playstation 3 and Xbox 360.

The key to the GOAP approach is stated specifically in the title of Orkin

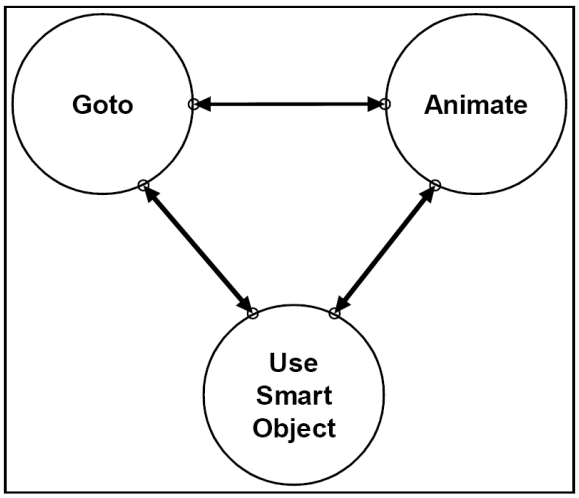


Figure 2.11: The three state FSM used to control the enemy agents in the game F.E.A.R. from Orkin [2006],

---

[2006]: *'Three States and a Plan: The A.I. of F.E.A.R.'*. The three states existed within a FSM shown in Figure 2.11, and the plan was derived using A* to navigate through this FSM in order to achieve a goal. At this juncture this seems wasteful, given that the set of states is so small. However, it was reliant on the FSM's association with the action set. Instead of representing all potential states that could arise from applying the action set, the FSM focusses on the need for an agent to move or commit an animation. The selection of animations at particular states in the FSM would be dependant on them satisfying conditions to be achieved. This was represented courtesy of a representation similar to that of traditional STRIPS, with the search for the correct animations carried out by A* search. If an action could satisfy a current goal, it could then be executed. If the goal is satisfied then another could be provided to the agent and it could continue on until it is, as intended, defeated by the human player.

The GOAP approach arose as an extension of work Monolith incorporated into previous titles, where their enemy agents would have a specific FSM associated with a goal they wanted to achieve. The agent would then execute an action that was applicable in their current state within the FSM until the goal state was achieved. However, the same FSM was applied for the same goal across all classes of agents. This led to the problem where any class-specific behaviour would have to be added as an extension to the FSM for that specific goal. Naturally, this proved unmanageable over time. However, using the GOAP approach with the

reduced state space within the automata, the specific actions any class would carry out were stored in their own action list and could be selected via the 'Animate' or 'Use Smart Object' states. This allowed the designers to create independant action sets that result in effects the planner could recognise. Ultimately, this decouples the action sets from the goals and allows for a greater freedom to add, modify or remove classes of agent from the game without compromising the remainder of the system.

While this was an important breakthrough in AI applications in commercial games, it came with a cost. Given that the game was responsible for managing large numbers of enemy agents in real-time, an intelligent approach was required to manage the agents computation. Given that in a commercial title the CPU cycles are typically devoted to physics, animation and graphics rendering, one needs to consider how AI processes can be introduced. These challenges are discussed in detail in Orkin [2005], where the author explains how the agent architecture can handle this form of "lightweight planning" through the use of distributed processing, caching sensor perceptions to working memory and intelligent garbage collection.

Another researcher who has built an impressive résumé in planning applications for games is Héctor Muñoz-Avila. In Munoz-Avila and Fisher [2004], Muñoz-Avila introduces a new approach for developing agents for the first-person shooter game Unreal Tournament using task-network planning. Hierarchical Task Network Planning (HTN Planning) is an alternative planning approach that represents phases of plans as high-level tasks. These tasks are then decomposed into a series of primitive tasks that correspond to actions the agent can execute, or compound tasks which in turn contain more primitive or compound tasks. Futhermore, there exists a series of ordering relationships between tasks that dictate temporal constraints, allowing for a more expressive formalism than traditional STRIPS-based planning operator representations (Erol and Nau [1995]). The HTN approach was adopted in the authors Java Bot controller which acts as the main client to the Unreal Tournament server. While the Java Bot code would handle events and state modelling, the HTNs were then introduced to satisfy particular strategies that were requested once particular events were recognised.

Muñoz-Avila continued exploring this HTN approach in Hoang and Muñoz-Avila [2005], with the intent to expand the Unreal Tournament bot controller to operate at a more trategic level. In this instance, the HTNs would devise tactics for team-based objective matches such as 'domination'; where a series of points

must be controlled and maintained by the player in order to increase their score. The HTN would delegate compound tasks to individual agents with respect to a particular strategy, and the agents, under the control of a FSM, would then execute primitive actions in the task network. This would result in a plan that satisfied specific temporal constraints between the agents. In testing these HTN managed agents against a team of improved FSM-driven players, the task-network approach was far more successful.

Meanwhile in Lee-Urban et al. [2007], the same design principles behind the HTN approach are explored with the intent to apply them in the context of a Real-Time Strategy (RTS) game. This work sought to expand the Simple Hierarchical Ordered Planner (SHOP) developed by planning researcher Dana Nau (Nau et al. [1999]). SHOP is an HTN planner that specialises in specific HTN planning problems where the subtasks of each network is totally ordered, i.e. the set of actions has a strict linear order of execution. The modified version of the planner, dubbed 'Learn2SHOP', integrated a learning algorithm that acted as a knowledge transfer system. Using the MadRTS engine, RTS problems were defined and a selection of 'skeletal' HTN methods - methods that required further clarification - acted as training samples. By evaluating these training samples by executing them in-game, the learning algorithm would be able to refine the HTN operators to operate generically rather than for specific concrete instances. This work was then evaluated using a transfer-learning task, where it successfully transposed knowledge from one specific problem instance in similar, yet different circumstances.

Further Challenges As can be seen from the research detailed in this section, a lot of time has been spent in integrating planning at either a simple level to automate animation selection, or as a more strategic element. At present one of the larger challenges faced is execution, given that at present, these works are still heavily reliant on hand-craft FSMs in order to execute actions. While these are certainly reliable, this often comes at the cost of extensive development time and testing. If a suitable controller could be learned for all or part of the agents functionality, then this lead time could be reduced. This challenge is addressed in Chapter 5 where we provide a potential solution to this problem.

Another important challenge is the level of difficulty associated with the planning problems these approaches are trying to resolve. Looking back on the work in Orkin [2006], would this prove sufficient for more demanding levels of

functionality? If we wanted our agent to be able to devise solutions to puzzles, could this approach be adopted? Given that no pre-processing of domain information is attempted when using an algorithm such as A*, we hypothesis that such a setup would struggle with more demanding problem domains. In Chapter 4 we introduce our test domain that should prove a challenge even to established planning platforms.

2.4 Layered/Hybrid Architectures

> Sometimes people are layered like that. There's something totally different underneath than what's on the surface. But sometimes, there's a third, even deeper level, and that one is the same as the top surface one. Like with pie.
>
> Joss Whedon

In Section 2.2, we explored a range of the methodologies applied to increase reactive controller scope and robustness. This was followed by our introduction to the field of automated planning in Section 2.3.1 where we defined the theory and practice of deliberation using discrete, abstract and deterministic planning models. However, having defined our deliberative approach, we must find a means to interface this decision making process with reactive control. This leads onto our final background section where we discuss the creation of layered agent architectures.

As their name would suggest, a hybrid or layered architecture describes the use of multiple, functional components with an imposed hierarchy for execution. Concepts introduced in Section 2.2, such as Incremental Evolution, Neural Net Ensembles and Modular Evolution are not hybrid given they rely on one core method. While subsumption can be considered similar to a hybrid architecture in terms of an imposed hierachy, each component still adheres to particular stereotypes and interfaces. With these architectural types we do not face this problem. Instead, time must be spent on how components interact and communicate with one another. In fact, an empirical evaluation of the available forms of agent architecture in Muller [1999] denotes subsumption early on as a reactive architecture: a class of controller only interested in correct and optimal behaviour for situations with a limited amount of information. In contrast, layered architectures are presented as an organised structure combining both reactive and deliberative controllers in an attempt to achieve coherent behaviour as a whole. Muller's observation of the agent architecture community, perhaps unsurprisingly, still holds true today; namely that most agent architectures are designed for autonomous control systems, typically to solve distributed resource allocation or act as an expert cooperative

system.

However, this is a relatively open classification. While there is a substantial amount of research in developing layered architectures, very few sub-disciplines or core methodologies have emerged. At present, it is reminiscent of early AI research, where many researchers attempted different ad-hoc approaches before any standard practices were defined. This too, is readily apparent in the application of planning technologies. In this section, we introduce plan execution and monitoring systems; layered architectures that rely on AP systems for deliberation and problem modelling, whilst utilising defined control models for execution. However, as we shall see, this research ranges from stronger understanding of plan actuators, to expressing greater control over processes at the plan level. At present, the only defining characteristic to unify these approaches is that of an arbitrary planning system which has been installed as the highest level of control. This of course makes our reporting on the subject something of a challenge!

Perhaps one of the most renowned and successful examples of plan and execution integration is NASA's Deep Space 1 (DS1) mission that launched from Cape Canaveral in October of 1998. The DS1 mission was the first in a series of missions by NASA designed to test new technologies that would be strategically deployed in future missions. Amongst an array of hardware and software components the Autonomous Remote Agent (RA) system was also included; an on-board planning and scheduling system designed to take complete control of the spacecraft. The mission successfully completed and was retired in December of 2001 after recording images and data of Comet Borrelly, at which point the engines were shut off, adding to the tonnes of metal currently floating in orbit around the planet (Bernard et al. [2000]). The RA system was developed with the intent of testing a platform for complete autonomous control of space missions. NASA realised that in time spacecraft must be capable of navigation and control with little or no human intervention. This can lead to increased functionality, robustness and flexibility while also reducing the cost needed for ground control (Ghallab et al. [2004]). However, there is of course the signifcantly high risk that millions of dollars of engineering and pre-planning are rendered null and void due to a glitchy AI system. The RA system was designed to circumvent these issues using three main components: a Planner/Scheduler that carried a mission manager, an executive and a mode identification and reconfiguration module. The planner, naturally, formulated a long-term plan for goal achievement, using short-term problems based on the mission profile. Using the executive's model of

the state and the mode identification module it would schedule the actions of the plan accordingly. For the DS1 mission the RA system was tested against three simulated faults while in space; a failed sensor that needed reactivation, a failed sensor that provided false information and an altitude thruster that was 'stuck' in the OFF position. In each circumstance the RA successfully resolved each issue - by reactivating the first sensor, ignoring the second one and switching to a navigation mode that did not require the malfunctioning thruster. This provided strong evidence of a successful planning, fault diagnosis and recovery (Bernard et al. [1999]).

Another example of plan-centric monitoring and execution can be found at the Monterey Bay Aquarium Research Institute (MBARI) in California, USA. The institute focusses on the investigation and discovery of physical, chemical, biological and geological properties within the local coastal front through the use of Autonomous Underwater Vehicles (AUV). This is a scenario that is strongly similar to that found in the DS1 missions, given that an autonomous system is separated from the human users due to challenging and life-threatening environments. Hence an effective control and execution system is required to navigate the murky depths. In Rajan et al. [2009] we are introduced to the T-REX executive; a hybrid executive that carries an onboard temporal planner. This planner is responsible for developing *temporally flexible plans*, a plan that carries a series of flexible intervals and constraints between individual actions/waypoints in execution. This planner is then tied to a series of hand-coded actuators at a vehicle control interface. However, it also passes through extra phases of deliberation in a top-down manner that decompose as well as consider the addition of science missions within the flexible timeframe. Interestingly, the authors state the main reason for this plan-driven approach is to circumvent the shortcomings of subsumption architectures that is clearly articulated in Bellingham et al. [1990] as follows:

> An area where layered control has the potential for increased sophistication is in the mechanism for resolving conflicting commands from behaviors into one output command. As described previously, the [subsumption] output command is generated by the highest priority active behavior. However, a more sophisticated response can be achieved by modifying the method by which the output command is generated.
>
> The current conflict resolution strategy generates responses that

> are particularly inefficient in certain circumstances. An example of inefficiency is demonstrated by the conflict between a high level behavior attempting to command the vehicle to the surface, and a low level behavior attempting to prevent the vehicle from impacting the bottom. If the low level behavior determines that the vehicle is too close to the bottom, it will command the vehicle to a minimum altitude. Only after the vehicle has attained that altitude will the higher level behavior be allowed to control the vehicle. This is inefficient since the high level behavior's command would have satisfied the requirements of the low level in the first place.(Bellingham et al. [1990] pp. 6)

This is an interesting contrast to the comments made when discussing the subsumption architecture and clearly highlights issues in relying on scaled-up reactive control. If we employ too much functionality in the system then we are tied down by conflicting components that will impede performance. This was for us, further fruit for thought.

In other areas of hybrid executives, research focusses more on enhancing the performance of the reactive component in relation to the plan-based approach. An interesting example of this can be found in Stulp and Beetz [2005], where the authors sought to learn performance of plan actuators in order to improve the efficiency of the agent's plan execution. The task at hand is a simple football scenario that would arise in the likes of the RoboCup competition[1], where the agent must acquire a football and turn with it towards the goal. The argument made by the authors is that a traditional approach does not consider the context of action. Hence in the problem example, the authors sought to improve the efficiency of the agent's navigation as it moves towards the ball, turning to the goal and dribbling it into the net. By formulating performance models and subgoal refinement, behaviour can appear less disjointed and more contextual. A second example in this vein explored plan execution behaviours as structured stochastic processes through the use of dynamic Bayesian networks in Infantes et al. [2006]. By gathering and applying knowledge of behaviour capabilities, it allows for better understanding of the agents functionality.

Conversely, research in plan execution architectures has explored execution control from a planning perspective. Research in Gallien et al. [2008] explores the creation of a partial-order-causal-link temporal planner called *IxTeT* based

[1] An international robotics competition whose aim is to develop autonomous football robots to promote research and education in artificial intelligence.

on CSPs for inclusion within a distributed robot control architecture. When executed the IxTeT operates as a temporal executive, maintaining control over continued operation of actions and as a procedural executive expands and refines the actions into commands at a functional level. Meanwhile in Effinger et al. [2005] we are introduced to a planning system dubbed *Kirk* that decompses task-level commands expressed in Reactive Model-Based Programming Language (RMPL) into Qualitative-State plans through the use of Temporal Planning Networks.

Finally, research in Robertson et al. [2006] and Robertson and Williams [2006] reports on advances in rover test-bed systems. The authors note that in complex, critical systems almost every component provides a potential point of failure. This is an unfortunate by-product of the size and complexity of systems required in order to approach and solve these tasks. To provide error proof code is a challenging task due to the sheer size of the number of known (and unknown) cases that can arise. The importance of these papers are three key observations made by the authors:

- Often assumptions made by control software prove to be false during execution.

- Software may be attacked by a hostile agent, seeking to interrupt its execution or may simply be reacting to its presence.

- Continued changes to software often introduces compatibility issues between components.

In order to address these issues, the agent software must be able to recognise and diagnose failures when they occur. To do this effectively, recognition software must be capable of isolating the component where failure has occurred and find an alternative means of execution. However to achieve this we require numerous models to successfully monitor the execution. This ranges from models of component relationships to models of intended behaviour and known errors. Ultimately, an agent or system must be capable of sensing its own state and reasoning about it. These concepts have been taken on board when designing our own agent architecture. During our discussion in Chapter 6 we will return to these observations and highlight how we have addressed these concerns in our own work.

2.5 Summary

In this chapter, we explored the four key areas from whence our research drew for this thesis: evolutionary artificial neural networks, scaled reactive control through subsumption, classical planning and planning execution and monitoring systems.

In Section 2.1 we introduced artificial neural networks and followed the process to create reactive control by tailoring these networks utilising an evolutionary algorithm. Next we explored how this approach can be expanded upon using richer learning methodologies, allowing us to scale our control to more complex problem domains.

To accommodate for any deliberative faculties we wished to be present, we explored the field of classical planning; a research area that focusses on discrete, deterministic decision making using abstract models of the environment. Finally, we explored approaches made in layered architectures to provide an insight into how AP systems have previously been adopted into agent frameworks.

As we observed, the concept of a layered architecture that incorporates deliberation with execution is far from novel. However, our intended approach, to implement classical planning with ANNs for a game environment, is a unique application of such methods. As seen in Section 2.4, these systems require established models of execution for plan-actions. Our approach intends to circumvent these models by providing generalised action-policies supplied by a neural network. We hypothesised this approach would remove many execution concerns, given that we can rely on these reactive controllers providing they have been suitably trained. Furthermore, as is evident from the state of game-based research, applying a classical planning algorithm to direct reactive controllers is an area ripe for exploration.

At this juncture, we have introduced the reader to our field of interest and the relevant technical and research background for our intended application. In the next chapter, we will look at the first phase of research that was conducted for this thesis. This phase was intended to improve our reactive controllers by training layers of artificial neural networks within a subsumption framework. This work will later be adopted in Chapter 5, where we apply the JavaFF planner in an agent architecture to assume control of these reactive faculties.

Chapter 3

Creating Robust Reactive Control

Control, control, you must learn control!

Yoda, The Empire Strikes Back

3.1 Introduction

In this chapter we report on our work in creating reactive controllers that can operate in challenging environments. As we have previously stated this research focusses on developing ANNs that are capable of dealing with a variety of features in the environment and can add new behaviour to accommodate fresh issues that arise. To carry out this research we use our established problem domain called *EvoTanks* (Thompson [2005]) to explore the use of different network configurations. As the reader progresses through this chapter they will be introduced to our subsumption-inspired reactive controller designed to provide a "plug'n'play" approach to behaviour formulation and how it compares with other approaches on prepared test examples.

We begin this chapter by introducing the reader to the EvoTanks environment in Section 3.2, describing the features of the game world, why it proves to be an interesting testing domain and our prior research in the game. Section 3.3 then describes our controller design in detail as well as the learning methodology applied.

Section 3.4 highlights our results in training a variety of different controllers, first individually and then in combinations of up to three layers of control. To assess the effectiveness of this approach we compare it with other methods and discuss the product in Sections 3.5 and 3.6 respectively. We then finish with our concluding remarks in Section 3.7.

3.1.1 Goals

The goals of the research in this chapter are:

- Devise a suitable implementation of the subsumption methodology that would enhance the control of our neural networks without compromising trained behaviours.
- Create a series of controllers that not only test our approach but can effectively achieve what our planning domain requires.
- Devise an effective learning process to train reactive controllers within our subsumption hierarchy as quickly as possible.
- Compare this approach with other approaches to ensure our approach is worth incorporating into future work.

3.2 EvoTanks

> Any problem caused by a tank can be solved by a tank.
>
> Peter Griffin, Family Guy

EvoTanks is a game environment designed initially to develop reactive agents for local tasks using machine learning algorithms. The domain is inspired by the video game *Combat* released for the Atari 2600 in October 1978. Combat is a top-down game where two human players would compete against one another for the highest score, featuring over 27 different gameplay modes, all of which are variations of combat between tanks, bi-planes and jets. The tank game in Combat shown in Figure 3.1 gave control of two tanks to human players with the intent of scoring the highest amount of damage possible within the time limit, with competition taking place in one of three locales; an empty field, a simple or complex maze.

When regarded as an AI problem, it appears challenging but still within the realms of reactive control. EvoTanks is a *deterministic* game world, yet unlike BruceWorld it is also *fully-observable*; giving the agent complete knowledge of the world. This complete observability means that trained EANNs will generalise action selection for different scenarios competently, given that EvoTanks is a problem with relatively small scope. Our choice of ANN application is reinforced given the *continuous* and *dynamic* elements of gameplay. The one outstanding factor here is that we are again dealing with other agents whose actions affect our own, i.e. this is a *multi-agent* problem. This issue, as shown later, is incorporated into the tank's decision making process when necessary.

EvoTanks faithfully mimics the Combat tank game, a Java implementation existing as an enclosed square arena measuring 600 by 600 pixels in size. A standard EvoTanks game (shown in Figure 3.2) includes two agents with the intent of eliminating their opponent as quickly as possible. Each agent is limited to basic movement; forwards, backwards, turn left or right and firing of the cannon. Each agent is capable of activating one or more of these actions at each discrete time-step of the simulation approximately every 15 milliseconds, meaning the agent can for example move forwards, turn left and fire simultaneously. Each movement action is fixed to a predefined distance (2 pixels) or angle (4 degrees) per update. Once any action is committed, there is a minimum delay before that

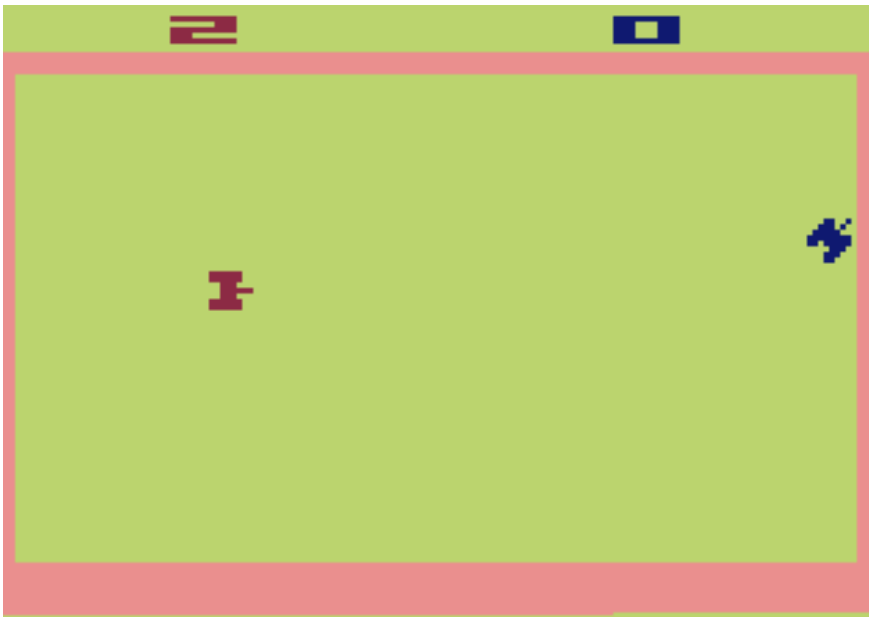


Figure 3.1: A screenshot of the tank mode from *Combat*. The agent's goal is to generate as large a score as possible by damaging the enemy agent within the time limit.

---

same action can be committed again The delay is 1 simulation cycle for movement and turning and 50 cycles for firing the cannon. The cannon is fixed to the front of the tank and carries an unlimited supply of ammunition. Each agent starts with a maximum shield strength of 100 points, and suffers a loss of 25 points for each collision with the environment or if hit by enemy fire. An agent will successfully win a match provided the enemy has lost all shield points within the time limit (1000 time steps), otherwise the match is declared a draw.

EvoTanks provides an interesting environment for the testing and development of trained reactive controllers. The game presents a significant challenge given the need to have knowledge of the local environment as well as the enemy agent. Despite the limited amount of movement, there is still sufficient room for a variety of different strategies to be formulated. To further complicate issues for AI or human players, there is also an array of pre-programmed, scripted players built for the game:

Sitting Duck/Lazy Tank/Random: The Sitting Duck and Lazy Tank are simple and docile agents that do not commit any movement actions throughout a match. While the sitting duck is fairly self explanatory, the only difference

Figure 3.2: A screenshot from the EvoTanks game. The match in progress shows one similar to Figure 3.1, where two agents are attempting to eliminate one another within the time limit.

between it and the lazy tank is that the latter will fire its cannon at every available opportunity. The Random agent will, as expected, commit one or more random actions at every timestep. These agents were originally designed just to provide players a target to focus their behaviour against.

Hunter: An aggressive opponent that turns to face the opponent and moves towards them while firing the cannon. This kamikaze agent ignores all concerns for its own well being and continues onward. This often presents quite a challenge for the player, since the aggressive opponent forces quick thinking to ensure survival.

Sniper: A defensive opponent that tries to maximise the distance between the player and itself while attempting to remove points from the player's shields with long distance shots. Once it has reached a minimum distance from the player (300 pixels) it shall remain in place. This agent provides the opposite challenge from the Hunter, since we are dealing with an enemy that will continue to evade while ensuring a defensive position.

Turret: Perhaps the most challenging opponent due to its fixed position. It maintains some ability from the Hunter; constantly turning to face the opponent and firing whenever capable, however it is incapable of movement. This lack of movement actually reinforces the defensive capabilities of the agent, as the opponent must be able to survive the barrage of incoming fire in order to make a kill-shot.

Each of these opponents were initially designed to facilitate our requirements during our previous research using EvoTanks. While agents are only capable of simple actions, the agent API[1] provides a variety of information that can be used to assist the decision making of the player. This point is further elaborated on in Section 3.3.

3.2.1 Previous EvoTanks Research

Our initial research in EvoTanks was to take our variant of the *Combat* tank-game, and attempt to train a feed-forward, multi-layered ANN that could compete against one of the array of scripted agents shown above. A supervised learning

[1]Application Programming Interface: An interface that provides a range of functionality to allow a software component to interact with another system.

method was not applied due to the lack of training data available for our agents. Given that each agent relied on low-level normalised data, it was deemed plausible that an agent would be able to learn appropriate behaviours from the provided stimuli (Thompson [2005]). The resulting product was promising, with successful agents that learned to out manoeuvre their opponents and develop interesting and varied behaviours.

However one of the main drawbacks was that the evolved agents became niche players: i.e. the agents were, potentially, optimal with respect to the tank they trained against. Hence agents would learn how to eliminate one script competently, but were unable to compete effectively against other agents - even those with minor differences in behaviour. To counteract this, a competitive co-evolution model was applied to the population (Thompson et al. [2007]). Small tournaments comprising two teams of 10 agents would play against one another, the intention being to provide sufficient understanding of a candidate's fitness with respect to the rest of the population by playing against a random sample of other players. Evidence from Thompson [2006] suggested the performance of the learning system was affected by the number of players. Further experimentation indicated that 10 agents was sufficient for the tournament assessment. The co-evolution approach was evaluated against a 1+1 ES where each candidate was assessed against all scripted agents. Results published in Thompson et al. [2007] indicated that the 1+1 ES was more efficient in generating strong behaviours. The ES would often generate more efficient strategies that sought to eliminate the enemy player as quickly as possible. Meanwhile the co-evolution approach generated more diverse strategies, with final fitness values often just below what could be considered an upper-bound generated by the ES. Work in a similar vein can be found in Tan [2008], where a hybrid competitive-cooperative model was applied instead of the simple competitive model shown in Thompson et al. [2007]

3.3 Agent Design

> See first that the design is wise and just: that ascertained, pursue it resolutely; do not for one repulse forego the purpose that you resolved to effect.
>
> William Shakespeare

Throughout the development of our individual networks, we deemed it important that simplicity in the design should remain paramount. Resulting behaviours must be relatively robust and efficient, however as we wished to retain an elegance in the design, we tried to ensure that each component would be small and simple. Provided sufficient effort was made in the design phases, then resulting behaviours would become easier to formulate through our subsumption hierarchy.

In Section 3.3.1 we provide a breakdown of the controller design; how the subsumption is deployed and the parameters that hold across all ANNs. We also provide the full list of controllers that are later explored in detail in Section 3.4.1. This is followed by a breakdown of the learning methodology in Section 3.3.2 that explores how we train these multi-layered controllers.

3.3.1 Controller Design

Our reactive controller is composed of a series of feed-forward, multi-layer ANNs that are placed atop one another within the subsumption hierarchy. Each network is designed to have four layers; inputs, outputs and two intermediate layers of hidden neurons. This topology has persisted since our previous work in Thompson et al. [2007] and given the success enjoyed from this approach, we committed to use the same topology throughout this research. The hidden neurons within the middle layers typically range from three to five per layer, with each neuron using a hyperbolic tangent (tanh) transfer function. This is employed to reduce the initial bias in the network that other transfer functions, such as the sigmoidal approach do not compensate for. By using this approach we constrain the size of each network, with typically less than 30 synapses in each controller. As the weight of each synapse is constrained within a range of ± 5, each chromosome is a series of weight values within this range. This small network topology and

gene range allows for fast training times as shown in Section 3.4. An example subsumption layout composed of three networks is shown in Figure 3.3.

Our subsumption approach dictated that an agent should be able to use any available information about the domain in order to generate an appropriate action. Therefore, any subset of information from the domain could be used in each network. To faciliate this in the EvoTanks environment, we developed the *Oracle*; a separate Java class that keeps up to date information about all entities within the environment. Should we require specific information to be available then it would be easy to add a new method to achieve this. A complete list of available inputs can be found in Table 3.1 with a list of the required inputs and the resulting outputs. Note that each output is *normalised*: a common process that ensures values conform to a specific range using an appropriate scale. As we can see, each piece of resulting data is normalised within the range of $-1 \leq x \leq 1$ or $0 \leq x \leq 1$. Normalising the inputs to the ANN ensures more effective training than using the data without any scaling, since the network has a smaller range of inputs to generalise. The ranges introduced above were selected due to our success in previous EvoTanks experiments. In the case of our EvoTanks controllers, each ANN would typically use two to four inputs from the game. Each input selected was dictated by the goal and function of each controller and is explored further in Section 3.4.

Furthermore, each network within the hierarchy is allowed to provide signals to any outputs of the subsumption controller. In the EvoTanks domain each ANN can have up to three output neurons to commit actions to the subsumption controller. These neurons correspond to forward/backward movement, left/right rotation and firing the cannon. Hence, each individual layer can provide values for any or all of these outputs. Positive output in the movement neuron triggers a 'move forward' action and a 'turn right' action in the rotation output neuron, while negative values for these neurons result in the opposite effect. In order to fire the cannon the fire output value needed to exceed a predefined threshold (≈ 0.01). For both movement and rotation outputs there exists a range of ± 0.05 and ± 0.03 respectively that operate as 'null' outputs, i.e. should the value exist within this range then the agent will not commit an action.

The traditional subsumption approach advocated by Brooks dictated that a top-down order of precedence is required for all controllers within the hierarchy. We have adopted this approach in our version applied throughout this chapter. However, we have taken the liberty of only restricting individual outputs through

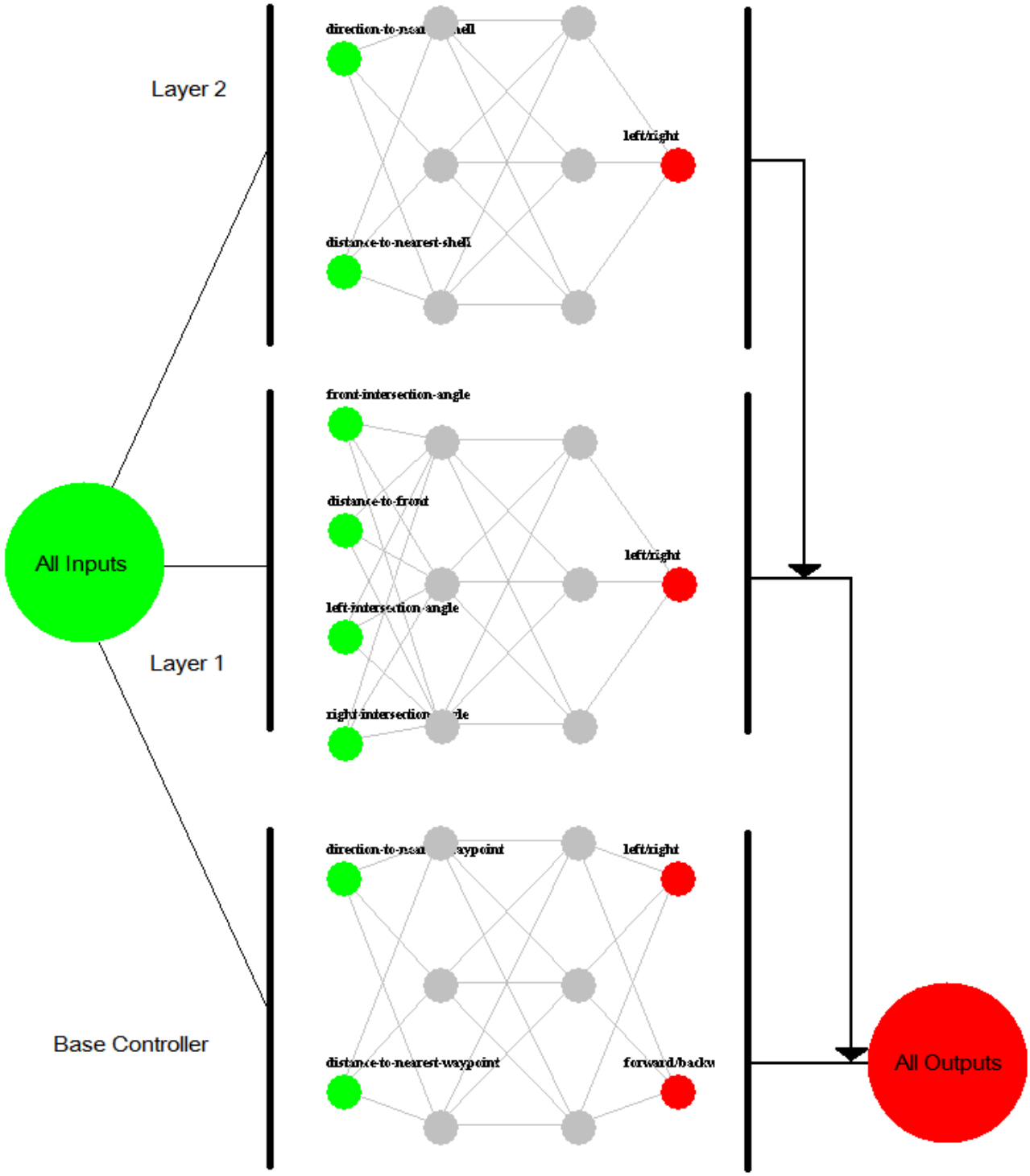


Figure 3.3: An example subsumption architecture with three neural network controllers placed atop one another to dictate a hierarchy of execution. Agents utilise a subset of all available inputs and activate a subset of the outputs.

Table 3.1: The complete list of information available to an EvoTank agent courtesy of the Oracle.

Data	Normalised Output Range
Distance from Point/Target	$0 \leq x \leq 1$
Angle from Point/Target	$-1 \leq x \leq 1$
Enemy Angle to Agent	$-1 \leq x \leq 1$
Distance from Nearest Shell	$0 \leq x \leq 1$
Angle from Nearest Shell	$-1 \leq x \leq 1$
Wall Intersection Angle Ahead	$-1 \leq x \leq 1$
Wall Distance Ahead	$0 \leq x \leq 1$

subsumption rather than complete controllers. As a result, each output of the agent will be executed by the layer with the highest order of precedence that has provided a value for that output. As previously stated, in EvoTanks each controller is able to provide an output for the move, turn and fire commands of the agent. So in the case of our subsumption hierarchy, at each time step in the game the controllers in the hierarchy are queried in descending order to see whether they have a response for the 'move' action. Should a controller specify a signal for an action then this is submitted to the agent to process and all layers beneath it are ignored. If no signal is provided at this layer, then we consult the next controller down in the hierarchy. This continues until we have reached the bottom layer, where if no signal is provided, then that action will not occur on that time step. This process is then repeated to see if the 'turn' and 'fire' outputs will be executed at that time step.

Given the nature of this subsumption process, we hypothesised that placing the core behaviour of a controller at the bottom of the hierarchy was the best approach. This decision actually contradicts the original concept in Brooks [1986], where it is suggested that the goal behaviour be built incrementally as we add more layers to the top of the hierarchy. This would be achieved by adding reactive control at the bottom followed by more abstract and goal specific control at the top. What we wanted to achieve was building the basic functionality of our controller at the very bottom layer and submit 'updates' to the behaviour in the form of supplementary layers.

To reflect these decisions, our controllers were built with one of two possible functions; as either a *base controller*, where the controller is responsible for the basic functionality or as a *layer controller* designed to operate in conjunction

with lower layers. These layer controllers must assist in dealing with information de-coupled from the base controller's goal. Each layer controller would be designed to associate with a feature that could be added to a problem instance. Given the nature of the subsumption hierarchy it was important that layer controllers only activate when they are required. As shown in Section 3.4, our layer controllers only activate when dealing with a specific feature of the environment. This results in only having to train responses to the input and saves having to learn when to ignore inputs. We hypothesised that this would result in faster training times. To explore this approach, a handful of controllers were created which we could test in EvoTanks.

Provided is the list of sub-controllers we built for EvoTanks, with the name of each controller suggesting their function:

Destroy Target (Base): Agent is charged with the elimination of a specified target.

Visit-Waypoint (Base): Agent must drive towards a fixed coordinate in the world.

Grab-Item (Base): Agent must retrieve an item from a position in the world.

Detect-Obstacles (Layer): Sense nearby obstacles and prevent collisions.

Dodge-Shells (Layer): Sense incoming enemy fire and attempt to avoid them.

Once we introduce the learning methodology for our subsumption controller, we will return to these sub-controllers in Section 3.4 and explore the process of how we created effective ANNs to satisfy our requirements.

During execution of the subsumption, layer selection was deliberated across each individual output. Since sub-controllers only provide outputs for actions they require, this gave us a different form of behaviour; since our controllers could learn to interrupt only those actions that were of interest to them. This gave us the freedom to create controllers that did not provide connections to certain outputs, effectively ignoring them. More importantly, it allowed for more than one sub-controller to be in control of the overall behaviour, since an individual layer may want to only override one output and permit the lower layers to proceed as normal, as shown in our 2-layer experiments in Section 3.4.2. At each time-step of the EvoTanks game, all networks within the hierarchy processed

outputs based on the current available data from the Oracle. Once completed, the top-down subsumption approach was applied across all outputs to select the committed action. Therefore movement, rotation and firing were triggered by the network which committed the output while holding the highest position within the hierarchy.

3.3.2 Learning Methodology

Now we had a series of networks placed atop one another within the subsumption framework, our next step was to devise a training approach which would ensure the controllers were trained effectively. Our approach was inspired by the *layered evolution* previously discussed in Togelius [2004]. The overall process is summarised in Algorithm 3.

Algorithm 3: A breakdown of the training process for individual subcontrollers.

Input: A queue of controllers Q of size l, where the tail is the top controller in the hierarchy and vice versa.

Result: A trained hierarchy of ANNs tailored to the Subumption paradigm, R

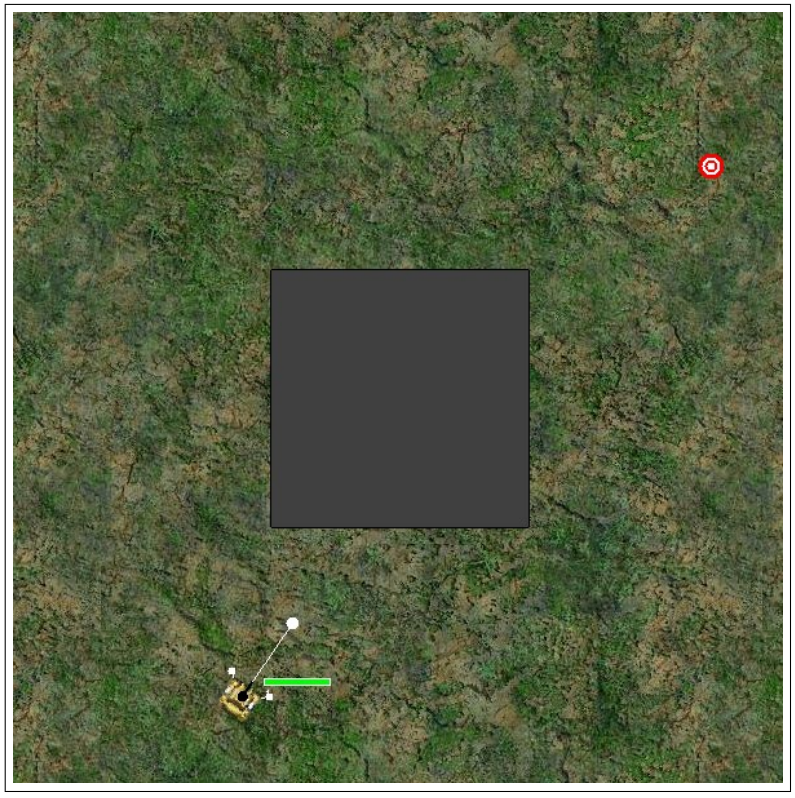
```
1  Assign fitness function for training from Q_0
2  addController(Q_0,R)
3  for i ← 0 to l do
4  |  for j ← 0 to MaxEval do
5  |  |  best_i ← train(Q_i)
6  |  end
7  |  freeze(best_i, Q_i)
8  |  if i < l − 1 then
9  |  |  addFeatureToEnvironment(Q_{i+1})
10 |  |  addController(Q_{i+1},R)
11 |  end
12 end
```

Having decided what our goal controller is, we begin by separating the controller into individual neural networks. The separation process is typically an intuition based decomposition and relies on our understanding of what we wish to achieve. Each ANN is comprised of the same topology and parameters described previously, with the only obvious differences emerging from their function and the required inputs and outputs to achieve them. Subsequently we select the order that they are to be trained, beginning with the base controller which is placed naturally

at the bottom of the hierarchy, then commencing the selected learning process (EA, ES) for a specified number of evaluations using a standard training problem. Once training is concluded for this controller it is *frozen* in place, i.e. it is not trained any further and the behaviour is now fixed. We take the next controller we wish to train and insert it into the subsumption hierarchy above the frozen controller(s) and begin its training. When training the next controller we operate under the same format as before, however we change the problem instance slightly to reflect on the changes necessary to justify the additional control. This cyclic process continues until we have completed training all controllers, resulting in the layered controller shown in Figure 3.3.

An example learning process may be to create a controller that can navigate through an environment and avoid nearby obstacles. The English description would often give us a sufficient idea of how to separate the behaviour, so in this example we detach obstacle avoidance from basic navigation, resulting in two controllers to train, namely *visit-waypoint* and *detect-obstacles*. Again, our description gives us an indication of the basic functionality and the additional control, dictating the order of training. We first place the visit-waypoint controller into the hierarchy and give a simple navigation example for it to train against. Once this phase completes, we freeze the navigation controller and add the obstacle avoidance controller on top. The obstacle controller is next to be trained by using the same number of evaluations. The training instances used in this example are shown in Figure 3.4. In Figure 3.4(a) we see the base controller being trained in a simple environment, prior to the inclusion of obstacles for the avoidance controller in Figure 3.4(b).

(a) Navigation Only



(b) Navigation & Obstacles

Figure 3.4: An example of environment modification to reflect the change of controller being trained. In Figure 3.4(a) we see basic navigation training, followed by the inclusion of an obstacle for the avoidance controller in Figure 3.4(b).

3.4 Implementation & Training

> Training is everything. The peach was once a bitter almond; cauliflower is nothing but cabbage with a college education.
>
> Mark Twain

In this section we explore the creation and training of ANNs designed to facilitate our base and layer sub-controllers for the subsumption controller. We begin by exploring each of the controllers in turn, highlighting the design of each neural network. When examining base controllers, we also provide results from our testing to ensure their validity. We will give the design and layout of layer controllers within Section 3.4.1, however, testing and experimentation is found in Section 3.4.2, where we begin to combine layers of networks to assess their capabilities in more challenging environments.

3.4.1 Individual Sub-Controllers

In this section we provide detail of each individual controller, the inputs used, how they are normalised with respect to the environment and what outputs are subsequently generated. We wish to remind the reader that each individual controller is a standard multi-layer feed-forward artificial neural network. Each neuron utilises a tanh transfer function and there are no bias nodes within any of the networks.

Recall that each individual network can retrieve inputs from the Oracle (see Table 3.1), each of which has been appropriately normalised with respect to the environment. This is required since the range of unscaled data may vary and can become, on occasion, very difficult for a network to learn from. For example, if we feed a distance value into the network then we could potentially be feeding any value from zero to the length of the diagonal of the EvoTanks arena. As a result, we would need to generalise our control policy for any value in the range of $0 \leq x \leq \approx 849$. This would require a significant amount of time to train, since we not only consider all values of x but also any potential combination of inputs involving x. Thus the data must be normalised in a manner that reduces the potential range of inputs as well as smoothens the input space.

To assess the validity of our subcontroller design, we apply the (1+1) ES as the learning algorithm of choice. If we refer back to Section 3.2, this is due to results in previous research found in (Thompson et al. [2007]), where we concluded that the (1+1) ES provides a potential lower-bound on fitness. In the coming sections we provide graphs of performance and the accompanying tables that indicate how well our controllers perform with respect to their fitness functions. Furthermore, we also require a means to assess the statistical significance of these results. Whilst presenting this data indicates how well the agent performs, we must ensure a confidence in these returns in order to validate our findings. This can be achieved utilising a sample size equation with confidence intervals, a simple test found in statistical texts that suggests the number of observations required to constitute a particular assumption with a specified level of confidence. This equation is provided in Equation 3.1.

$$n = \frac{1.96^2\sigma^2}{\alpha^2} \tag{3.1}$$

The sample size calculation above is designed to indicate the number of experimental runs (n) required to attain a 95% confidence, corresponding to a z-distribution of 1.96, that the standard deviation of the sample mean (σ) is within an error bounds (or confidence interval) of 0.1 fitness (α). In the context of our learning experiments, we can take the fitness statistics from a set of results, and calculate how many experiment runs are required to ensure a 95% confidence that the findings can be duplicated within 0.1 of the mean. This is very important when dealing with evolutionary based algorithms since a good result can be the fortunate product of randomly instantiated variables. The confidence interval of 0.1 was considered sufficient for any successful behaviour given the simplicty and urgency of the fitness functions. If the learning algorithm could not consistently generate fitness values within this constraint, then either the heuristic, or the learner was insufficient for the problem. Providing our testing set is sufficient, the resulting sample size would give a strong indication of the robustness of the search process.

Learning Parameters

The learning parameters we apply for the following experiments are provided in Table 3.2, with a summary of the ANN parameters in Table 3.3. Unless otherwise stated, each controller is trained for 50,000 evaluations. One evaluation is

Table 3.2: The standard learning parameters applied across all of our (1+1)ES and Genetic Algorithm experiments.

(1+1) Evolutionary Strategy Parameters				
No. Games Per Candidate	Total No. Evaluations	Probability of Mutation	Mutation Range	Chromosome Weight Range
50	50,000	40%	±1	±5

Table 3.3: A recap of the parameters applied across all of our neural networks during training.

Neural Network Parameters			
Transfer Function	Turn 'Null' Space	Move 'Null' Space	Fire Threshold
tanh	±0.03	±0.05	0.01

considered a complete EvoTanks game that concludes due to the agent completing the task, the tank being destroyed or the max time limit being reached ($T_{game} \equiv T_{max}$). Our previous research in Thompson [2005] and Thompson et al. [2007] provided evidence to suggest that 50 evaluations should be made to assess each candidate in the learning process. Once completed, we take the average fitness across the 50 runs, giving a reasonably accurate measure of fitness.

We apply the mutation methodology introduced in Section 2.1.2, where we introduce random noise within the range of ±1 to each gene in the chromosome with probability p. As we can see from Table 3.2, this will occur with a probability of 40% for each gene. Should the value of a particular gene exceed the range of ±5, then it is reset to the closest extremity.

Lastly, we provide the parameters of the neural network. As previously noted, each neuron in the network runs under a TanH transfer function. Furthermore, we remind the reader of the 'null' ranges for the turn and movement outputs and the threshold required for the fire neuron output to be accepted.

Addressing Fitness Noise As previously discussed in Chapter 2, we must consider the effects of noise on our fitness calculations as we run the learning algorithms. The first way to address this is to ensure each candidate is assessed for a sufficient number of games. An insufficient number of trials may assist a given candidate in sustaining itself within the population thanks to a fortunate

and beneficial initialisation. By taking the average of these 50 games then the noise in individual fitness calculations will be addressed. However, we must also consider the setup of each individual evaluation. If this was left to simple random instantiation, then noise would practically remove any credibility from the fitness of a candidate. As a result, we generate 50 sets of experiment configurations, i.e. starting positions and directions, prior to any learning taking place. When a particular candidate is selected, it will use these 50 experiment configurations across the evaluations. This same set is then used to assess all candidates. This ensures that the agents are assessed under equal terms and prevents further noise being introduced.

Visit-Waypoint

We begin with one of our three base controllers named *visit-waypoint*. As the name would suggest, the controller's function is to move an agent from its current position to some specific (x, y) position (or waypoint) in the world as quickly as possible. The agent only considers the position in the world it is planning on reaching and its actions are based solely on what is needed to reach this waypoint.

The controller relies on two inputs; the distance of the waypoint from the agent and the angle from the front of the agent. This provides two important pieces of information, how far the agent is away from the goal position and what angle it has to turn to face it. For ease of training both of these inputs are normalised (as shown in Table 3.1), the distance variable is normalised with respect to the length of the arena and the angle represented as a polar coordinate from the front of the tank.

The performance of the ANN is assessed by how quickly it reaches the waypoint (T_{game}) with respect to the maximum number of timesteps in an EvoTanks game (T_{max}). We represent this in our fitness function shown in Equation 3.2. In an effort to push the agents to perform better in the learning process, we penalise them for every timestep they fail to reach the waypoint. Thus the longer an agent takes to reach the waypoint then the larger the penalty to the score. Whilst this forces the agent to perform as efficiently as possible, it also ensures that the agent does not become complacent as it can never achieve the maximum score, as it is impossible to reach a waypoint in one timestep. This ensures that the learning process will continue to search for new solutions instead of stagnating. The penalty is scaled so that the maximum penalty is 0.5, our intention being

that the 0.5 is the minimum score that can be attributed to a successful test.

$$F_{succeed} = 1 - ((\frac{0.5}{T_{max}}) \times T_{game}) \tag{3.2}$$

However should the agent fail to reach the waypoint within the time limit then the agent is assessed based on how close it got to reaching the waypoint. In Equation 3.3 we model how close the agent came to reaching the waypoint (D_{Final}) and then scale the reward based on the distance travelled in respect to the initial distance (D_{Start}) from the goal. Bonus fitness is attributed for how closely the agent reaches the waypoint, with a cap at 0.5 since at this point the agent is actually reaching the waypoint and will instead be assessed with Equation 3.2. This in turn provides a rather smooth scale for progression, as a solution achieves greater fitness the closer it reaches the waypoint, followed by a gradual increase past the 0.5 mark as the candidate begins to become more efficient.

$$F_{fail} = 0.5 - ((\frac{0.5}{D_{Start}}) \times D_{Final}) \tag{3.3}$$

The reader may note that Equation 3.3 can lead to situations where the agent receives a negative score. This occurs when the agent moves *further away* from the waypoint compared to its position at the beginning of the test. While we could have rectified this, it provided a clear-cut indication of really poor solutions and we hypothesised that this would assist in the scoring and improvement process.

To assess agents using this ANN, we ran a series of random problem instances. Initially we ran tests using random positioning of the player using the visit-waypoint controller with one static, fixed waypoint in the bottom-right corner of the world. When assessed using both a 1+1 ES and a generational evolutionary model with a population of 200 candidates, this proved highly effective. We continued this phase of experimentation to explore results in moving the waypoint to the remaining corners of the environment culminating at the absolute centre. Once more, using the same search strategies, adequate and capable solutions were discovered. Leading us onward to our final phase of testing, we dictated that both players and destinations be randomly placed throughout the environment.

For our final tests, the agent is placed randomly in an empty EvoTanks arena and the waypoint to be reached is also assigned a random position as shown in Figure 3.5. However, we enforced a rule that the minimum starting distance was 600 pixels to ensure the agent can easily traverse large areas. This is in part due to the normalisation of the distance input once used in earlier EvoTanks research.

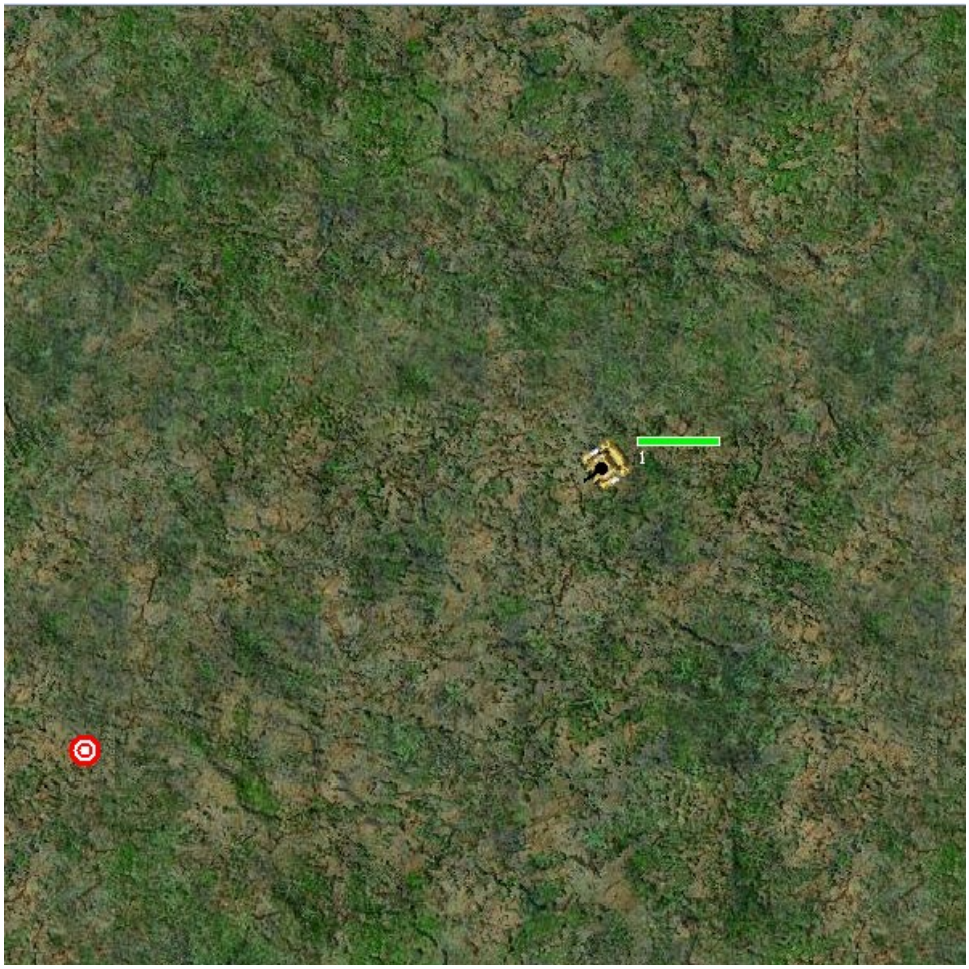


Figure 3.5: An example visit-waypoint test, where a randomly assigned waypoint and agent are placed in the world. The agent must now traverse the world as quickly as possible to the highlighted waypoint marker.

The original normalisation resulted an input range of $-0.5 \leq x \leq 0.5$ rather than the current setup $0 \leq x \leq 1$ (Thompson [2005]). While it proved sufficient for close quarters situations where tanks fought against one another, it proved to be ineffective across long distances as the input would reduce to 0 when the agent approached the centre of the arena. The agent could move no further forward as it was not given a weighted input. Each visit-waypoint network consists of two feed-forward hidden layers of three neurons. Combined with the two inputs and two outputs (movement and rotation), this results in a complete network composed of 10 neurons and 21 weights to be trained in the learning process.

In Figure 3.6 we show learning trends for three experiments using 50,000 evaluations. These results are tied to the data provided in Table 3.4 which

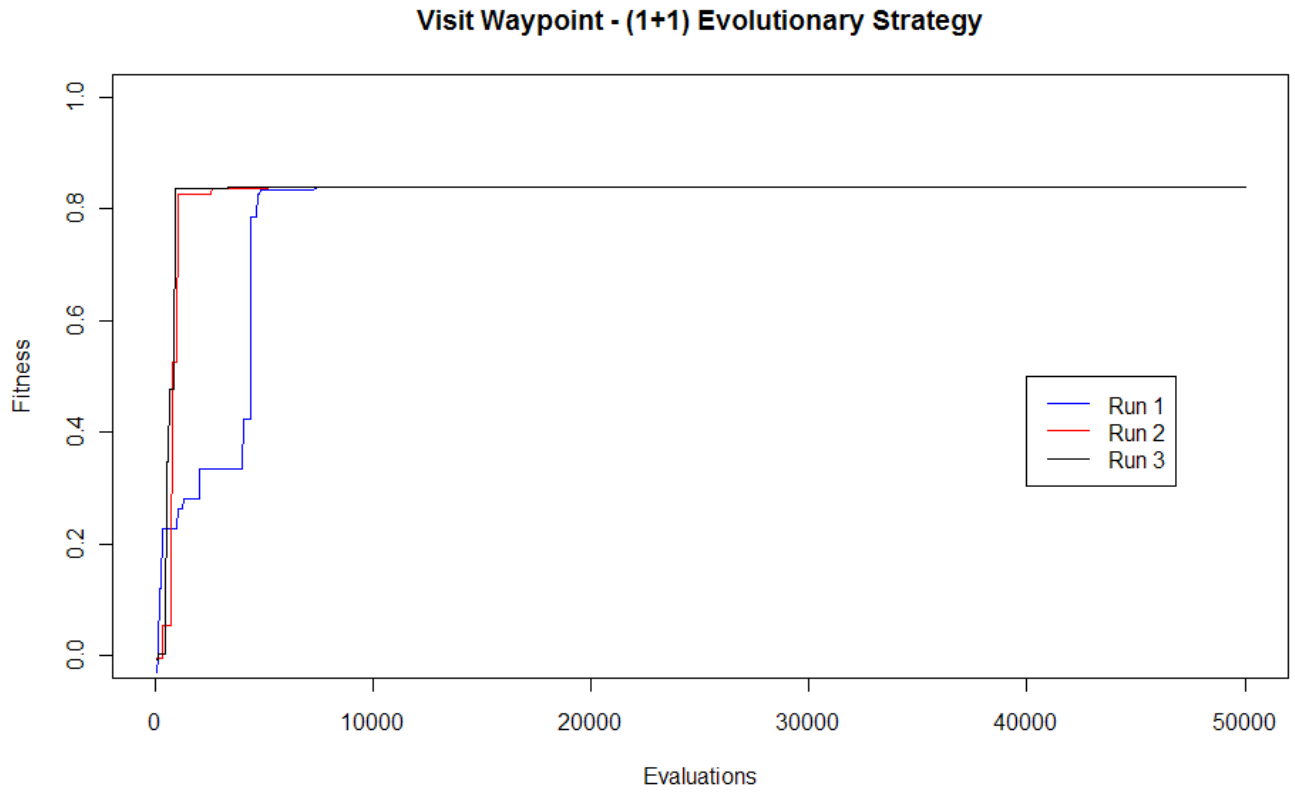


Figure 3.6: Three runs of the visit-waypoint controller using our (1+1) evolutionary strategy. Each agent is given 50 matches to generate their average fitness, with a 1000 separate candidate mutations.

Table 3.4: Statistics gathered from 10 runs of the visit-waypoint controller, including the three runs shown in Figure 3.6. Note the small standard deviation across this set of data and the low sample size requirements to achieve 95% confidence of success within a 0.1 range of the mean.

Statistic	Value
Maximum	0.8412
Minimum	0.8361
Mean	0.8385
Std. Dev.(σ)	0.0013
Sample Size (α = 0.1)	<1

provides a spread of statistics from ten runs of the visit-waypoint experiment. Note that the three runs from Figure 3.6 are part of this data set. The fast learning evident in Figure 3.6 suggests that the controller's design choices were intuitive. Agents learn efficient behaviours very quickly, with the desired behaviours emerging within the first 5000 evaluations. Further evidence in Table 3.4 shows that the best fitness often peaks at around the 0.84 mark. Given the penalty issued for each timestep that the agent fails to reach the goal and the minimum initial distance, this is an impressive result. This suggests a trained agent on average takes only 300 game cycles to reach the target waypoint. These statistics indicate also that there is little variation in the resulting solution's performance. Given that the difference between best and worst solutions is slightly greater that 0.005 and a very low standard deviation it suggests that agents would easily learn efficient performance. Further evidence to support this, is the result from the sample size calculation. Based on our sample data, we only require one run of the learning algorithm to achieve 95% confidence in our returns. This gives significant evidence to suggest that this learning process will almost always generate results akin to the statistics in Table 3.4.

The resulting behaviours from the experiments are directed and efficient. Each agent will often hammer-on the forward action and then turn in the correct direction in order to reduce the angle from the front of the tank to the waypoint. In almost all instances the agent will maintain a constant forward moving action. While we are no doubt highly satisfied with these results, we wondered whether we could in fact improve the performance and the desired controller. Given the behaviour of the resulting neural nets and how quickly the agents learn how to solve the problem, we were curious to see whether agents require both inputs.

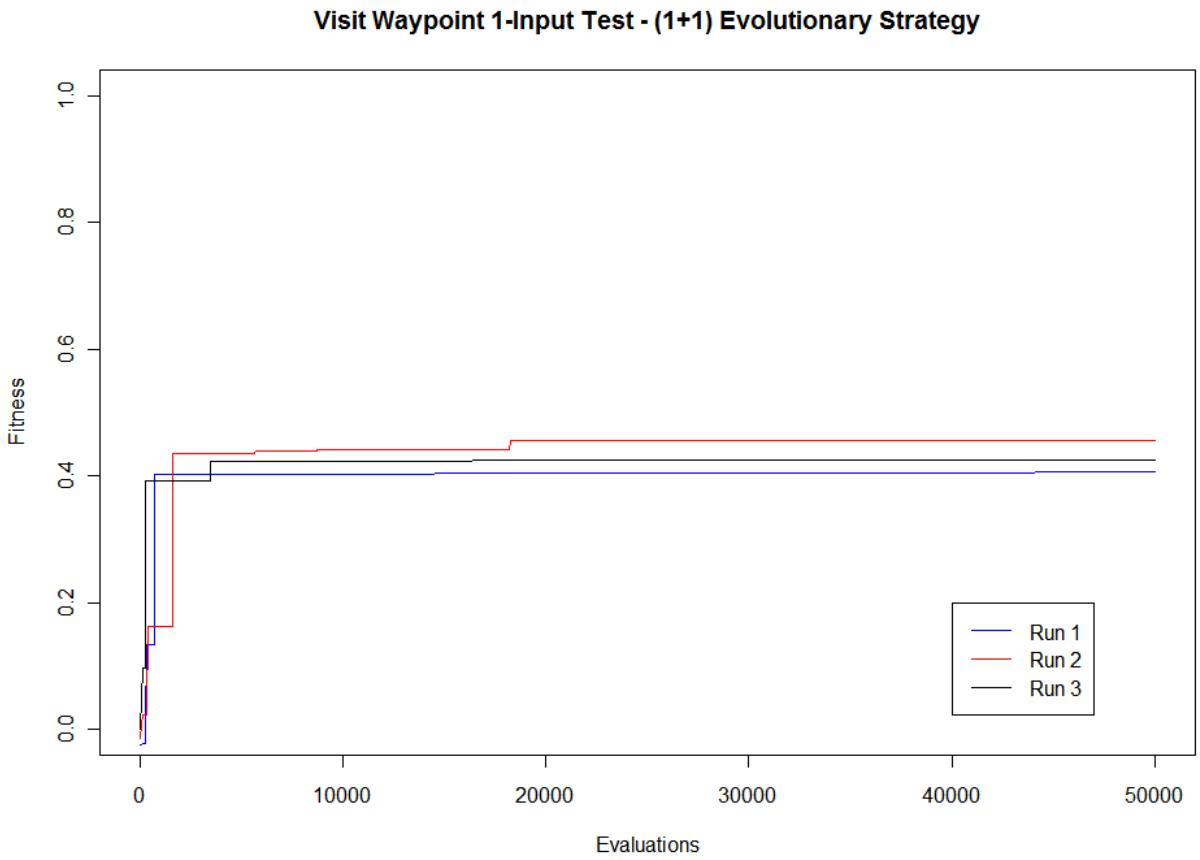


Figure 3.7: A series of experiments running the same problem as that in Figure 3.6, however instead of running with two inputs to the network the agent only runs on the normalised angle input.

The constant forward movement suggested that the distance variable may not be required and since the agent minimises the angle to face the waypoint, it was hypothesised that if we remove the distance input we could achieve similar performance using a smaller network. In Figure 3.7 we show three runs of a reduced network, where the controller runs on only one input, thereby reducing the number of weights to 16. The resulting trends indicate that this is in fact not the case, with a drastic loss in performance across all runs when compared to those provided in Figure 3.6. The resulting behaviours appear to lack focus that perhaps the distance input provided.

However, our resulting agents suffered from a minor setback. In certain situations the agent would become highly proficient in reaching the waypoint by reversing rather than moving forward. This is caused by a lack of information

within the fitness function, since the equation merely states that a candidate must reach the goal location as quickly as possible. This leads to a potentially wasteful search process since we do not collect information on the performance, other than the final result: Did the agent reach the waypoint or not? This highlights one of the issues that can arise in evolutionary learning, since adding new information in the fitness function can potentially prune useful areas of search space. Do we really wish to potentially constrain our search by demanding more within the fitness function? Furthermore, this situation can also arise due to not using a supervised learning process. This is the evaluation vs. instruction tradeoff that is described in Sutton and Barto [1998]. In supervised learning we provide instruction, whereby an agent must change behaviour to mimic the prescribed methodology reflected in the training data. Whereas our approach lends to the evaluation style, where we merely reward an agent based on how well it performs in respect to the fitness criteria (Yannakakis et al. [2007]). Of course it can lead to situations such as this, where an agent successfully achieves our desired goal, but in an undesirable fashion.

We required means to ensure that this 'erroneous' behaviour occurred far less frequently. This was particularly relevant for our 1+1 ES experiments, where this phenomenon was observed regularly. However we were concerned whether changing the fitness function would impact the learning process. Our first approach sought to provide a solution by changing the rules of the EvoTanks game, by simply reducing the reversal speed of a tank, as until this point the agent could move at the same forward speed as it could backwards. Now the agent moved at one fourth speed when reversing. While this proved effective in reducing the number of situations this occurred it did not curb the behaviour. This can be attributed to the learning process, since variations of the reversing behaviour could improve, leading to a local maxima within the search space that a hillclimbing algorithm will traverse to the peak. Our second approach was to used a modified fitness function as shown in Equation 3.4. The modified version placed a strong emphasis on the agent facing directly towards the goal location. In practice this proved to be far more successful, however it changed the search landscape significantly. Now an agent would receive far more reward simply for facing in the right direction due to the weights selected (best fitness values would often reach 0.95). Whilst this proved successful, throughout the remainder of the work we continue to use the original function in Equation 3.2 as reduced speed had decreased the number of reversing incidents to within an acceptable range. This decision was made given

that the new fitness landscape would not be as smooth as the original.

$$F_{succeed} = (1 - ((\frac{0.5}{T_{max}}) \times T_{game}) \times 0.5) + (1 - (\frac{T_{angleFromWaypoint}}{180}) \times 0.5) \quad (3.4)$$

Destroy-Target

Our next base controller is the original premise that our work is based upon shown in our prior research found in Thompson [2005, 2006]. This agent is designed to eliminate an assigned target as quickly and effectively as possible.

This neural network is given information about a specific target, typically another agent, but destructable static objects were also explored, albeit briefly. The *Oracle* provides three inputs; the distance from the agent to the specified target, the angle the agent must turn to face the target and finally the angle that the target must turn to face the agent, assuming the target is a hostile enemy, otherwise it is zero. As in the case with our other controllers, all of our inputs are normalised within a reduced range. Distance inputs are normalised with respect to the diagonal of the arena and the angles are represented as polar coordinates.

This controller is assessed not only by how quickly the agent eliminates the enemy player, but also by how effectively it achieves this. As is shown in Equation 3.5, the agent is assessed primarily by how quickly it eliminates the assigned target. As was the case with the visit-waypoint controller, the agent is penalised for every timestep taken to complete the task to ensure that no stagnation occurs in the learning process since there are potentially (while not necessarily feasible) better solutions to be found. Should the agent fail to eliminate the target then it is assessed according to Equation 3.6. In this instance the candidate is given a bonus for every timestep it survives in the environment. This is designed to smooth the fitness landscape, as this will promote solutions that do not succumb to enemy fire as easily as others. Once more a lower limit of 0.5 is set for successful evaluations, whilst an upper limit of 0.5 is set for failed evaluations. This provides a smooth transition from failing to successful scores.

However, note that this efficiency measurement only comprises 80% of the total fitness. The remaining 20% fitness can be attained from F_{health} shown in Equation 3.7, where the fitness is attributed to the number of health points the agent has retained from the initial maximum (H_{max}), and the number of hit points that have been removed from the target. Thereby a candidate that completely

eliminates the target while not taking any damage itself, will be considered a more efficient solution. In the event that the agent fails to complete the task within the prescribed time limit, then a score of 0.5 is automatically assigned. While we wanted to measure how well the agent could retain shield points, we still felt it was paramount to measure the our fitness primarily on the efficiency measurement. This accounts for why the efficiency measurement covers 80% of the overall fitness.

$$F_{win} = ((1 - ((\frac{0.5}{T_{max}}) \times T_{game})) \times 0.8) + (F_{health} \times 0.2) \tag{3.5}$$

$$F_{lose} = (((\frac{0.5}{T_{max}}) \times T_{game}) \times 0.8) + (F_{Health} \times 0.2) \tag{3.6}$$

$$\begin{aligned} F_{health} = (Agent_{health} \times 0.125) + \\ ((H_{max} - Target_{health}) \times 0.125) \end{aligned} \tag{3.7}$$

To assess candidate solutions during our learning process, we test each ANN against all six of the scripted agents we previously described in this chapter. Each scripted agent was designed to explore different aspects of desired behaviour in our previous EvoTanks projects, therefore we use all six to assess our candidate solution. Whilst our visit-waypoint experiment required only 50 runs of the EvoTanks game to assess our controller, this instance needed to gain a strong mean fitness against each enemy agent. This leads to a need to run against all agents 50 times, resulting in 300 matches played per candidate. Given this requires more matches per candidate, we must also increase the total number of evaluations from 50,000 to 300,000. Our previous research found in Thompson et al. [2007] showed that this approach was successful in finding capable solutions and in fact generated more efficient results than the use of the competetitive co-evolution model. Therefore in this instance we decided to use the best method to achieve the most optimal performance. With respect to the potential noise in the fitness experiments, we prepare 50 sets of starting positions and directions for both the evolving and scripted players. The evaluations will use the same 50 start points and directions while testing across all six enemy scripts.

We provide fitness trends from a sample of three runs of this experiment in Figure 3.8 with statistics across ten runs in Table 3.5. In all of the trial runs

Table 3.5: A series of statistics reflecting on 10 runs of the destroy-target controller. The results provide a strong mean considering the challenge this task presents, which is further highlighted by the maximum and minimym scores attained throughout these runs. However despite this, we remain confident that the agent's performance is relatively robust given the result of the sample size calculation.

Statistic	Value
Maximum	0.7315
Minimum	0.5482
Mean	0.649
Std. Dev.(σ)	0.06882
Sample Size (α = 0.1)	1.82

shown it is evident that the agent learns how to eliminate its enemies rather effectively by clearly surpassing the 0.5 fitness mark in the best cases. However, Table 3.5 suggests that results are often mixed; with final fitness ranging from high performance past the 0.8 mark, to the poorest of our sample runs barely succeeded on average. This range can be attributed to the increasing difficulty of the problem when agents are introduced to the hunter, sniper and turret agents. These scripted agents are often highly aggressive or defensive in strategy and prove difficult even for humans to eliminate in play. This loss of fitness is tied not only to the penalty for time taken, but also since an agent will lose health when fighting the more aggressive opponents. The major flaw in the agent design that has existed since our original research is that we do not provide information on incoming shells for the agent. This gives rise to an assessment of the candidate based on criteria it has no knowledgable of, nor means of correcting. Agents have in the past sought out creative solutions that help to avoid the enemy fire, often at the expense of efficiency. A potential solution is explored later in this chapter when we provide an additional controller that can facilitate the avoidance of enemy fire.

Grab-Item

Our third base controller, dubbed 'Grab-Item' is designed to provide an agent that will navigate through an environment and pick up a specified object. This controller is similar to visit-waypoint given that it must visit a pre-defined destination in the world. However, the additional requirement to pick up an object is what

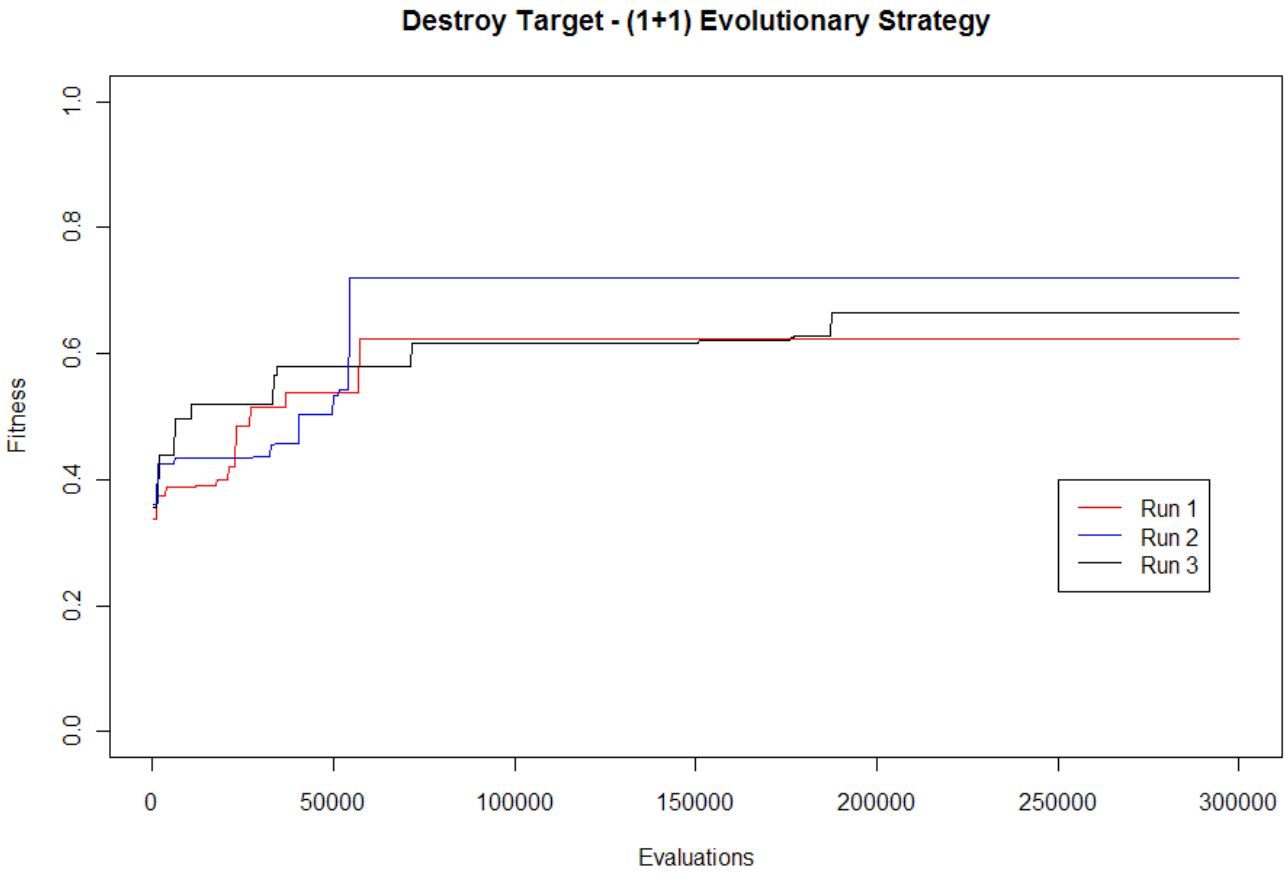


Figure 3.8: A series of destroy-target experiments training against the scripted agents described in Section 3.2. Due to the higher difficulty of the task, the agents are given significantly more matches to train with.

distinguishes it from its predecessor. We wished for the controller to only use the 'grab' actions when necessary, rather than continually attempting to pick-up the item regardless of where it is. This was decided on as in real life an agent that continually moves to pick up an item when it is not within nearby proximity would appear clumsy or ill-informed[1]. In turn this provides a greater challenge and requires a more defined neural network, since the network must be able to recognise the specific sets of data that correspond to being within close proximity of the specified object. Only then should the grab action be committed.

This agent operates using similar inputs to the visit-waypoint controller; the distance to the item and the angle of the item from the front of the agent. Once again the distance input is normalised with the respect to the length of the diagonal of the game arena, and the angle is converted into polar coordinates. This network now has three actions for interfacing with the environment; movement and turn actions corresponding to those we have seen previously and a grab command that - for the sake of this experiment - overrides the fire cannon action. Running on a 2-input, 3-output, multi-layer ANN with two hidden layers of three neurons, this leads to a chromosome length of 24 weights for training.

To evaluate this controller, we once again provide random positions for the tank agent and the item it is charged with retrieving, while retaining the minimum distance of 600 pixels as before. The fitness function for grab-item mimics the visit-waypoint method of assessing success and failure shown in Equations 3.2 and 3.3 respectively. The only notable difference is where the test terminates: visit-waypoint tests end once the agent has reached its destination or the time provided has elapsed. However, grab-item experiments are dictated by two conditions; whether the agent has committed the grab action within the time limit and whether the agent was in close proximity of the item when the grab action occurred. This is made clearer through our pseudocode example of the grab-item update() code shown in Algorithm 4.

Results from running these experiments are provided once again in two formats. Figure 3.9 presents learning trends from three runs of the controller within a 1+1 ES. In our learning trends we note that the learning process is capable of finding effective solutions. However we observe that in this instance there is one specific run where the learning process stalls, but then recovers after 20,000 evaluations. This circumstance arose due to the fitness function penalising the

[1]If of course we can suspend disbelief that a tank trying to pick up an object is perfectly normal.

Algorithm 4: A pseudo code description of the update loop used in the grab-item agent.

```
while T_Current ≤ T_Max do
    if grab then
        if objectClose then
            fitness =FitnessSucceed();
            return fitness;
        end
        else
            fitness =FitnessFail();
            return fitness;
        end
    end
    UpdateNetwork();
    MoveAgent();
    T_Current + +;
end
fitness =FitnessFail();
return fitness;
```

agent for not grabbing the item despite closing the distance as much as possible. This low fitness result was reached by an agent that has learned to move towards the controller, but failed to learn when to pick it up. Due to the nature of the fitness criteria, the agent must initially refrain from any grab actions, otherwise the game terminates and evaluation concludes. However, this can later cause problems such as this where the agent must now learn to use this action at the appropriate time. Fortunately in this circumstance the agent resolves this issue before search terminates, although this situation may not be rectified within the time. This can be observed in Table 3.6, which provides statistical data acros 10 runs of the experiments. While the most successful results score very well, with a maximum performance of 0.841 and an strong average result just over the 0.8 mark, the reader will observe that the standard deviation of the results is significantly higher. This, tied with the results from poorer solutions - which fail to solve the problem - suggests that this is not as robust as previous controllers. Evidence to this effect can be found in the sample size results, indicating we would require approximately four individual runs of this experiment in order to achieve 90% confidence in the mean. This is by no means a disastrous result, but it does however appear clear that extra work is required to improve the robustness of this

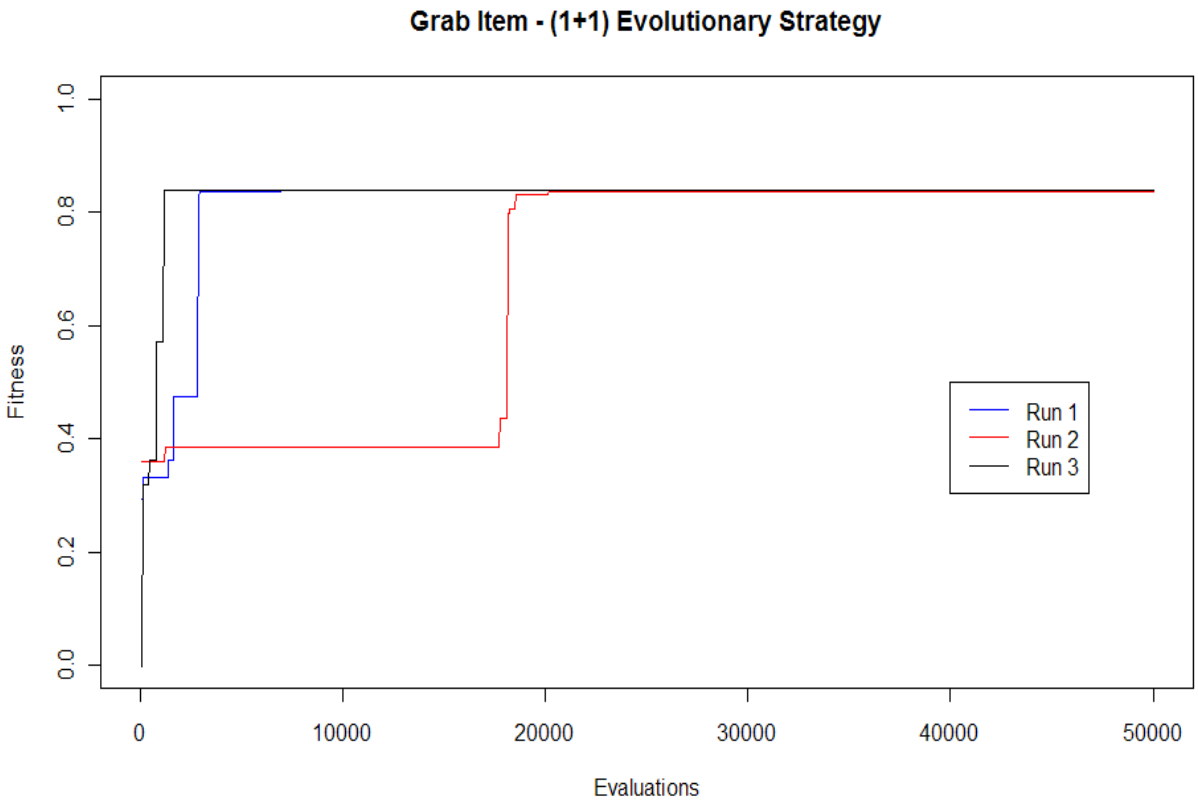


Figure 3.9: A series of experiments running the grab-item controller in a 1+1 Evolutionary Strategy.

controller in training situations. The final behaviours however are useful and we can exploit these in the next chapter.

Avoid-Obstacles

When deciding on what controllers we wanted to make available, we decided it was important to have some functionality that allowed an agent to avoid nearby obstacles within an environment. It was our original intention to build a neural network that could be plugged into our subsumption hierarchy that would allow previously trained behaviours to retain the assumption that no obstacles exist in the world (i.e. we do not need to provide inputs for them), since the new controller would address this issue. However the design of the sensors and how the agent would utilise them was an issue, due to having had no previous experience developing this type of sensor. We decided that whilst the intended product would

Table 3.6: A list of statistics based on 10 runs of the grab-item base controller. These results are interesting, since it appears the agent is not always successful in grabbing the item within the time period. As is noted by the failing minimum score as well as the large standard deviation from the mean. The sample size also indicates there is room for further development in order to increase our confidence in returns.

Statistic	Value
Maximum	0.8410
Minimum	0.5215
Mean	0.807
Std. Dev.(σ)	0.1004
Sample Size (α = 0.1)	3.87

be a controller for the higher layers, initial testing for our designs would be within a base controller. This would determine the effectiveness of certain designs and in addition, visualise their behaviours.

As a result, we created the avoid-obstacle controller, essentially a proof of concept that would allow us to test whether we could develop these sensors to suit our needs. At this juncture we had yet to test the subsumption controllers in any way, therefore we devised avoid-obstacles as a base controller for testing. In time, we took the knowledge accrued from this work and developed the layer controller *detect-obstacles*, with the results from testing given in Section 3.4.2. Given our intentions to develop these sensors for a layer controller, this introduced two problems for our avoid-obstacle ANN. Firstly, we needed to ensure a constant input to the network, and secondly we had to test it in an arena according to a defined fitness function. We now explore how these issues were addressed.

Adapting the Input Sensors for Testing The issue of constant input to the network arose from our adopted ANN design, notably that we do not include bias nodes within the network. Typically, bias nodes provide constant positive or negative input to the network. Without them, the network is dependent solely on the values received from the input nodes. In our intended layer controller, we wished only for the sensors to activate when obstacles were within close vicinity of the tank. However, if we were to test that kind of controller without bias nodes, there would be no input to the network when in an open area. As a result, we needed to find an alternative means of testing that would allow us to test the

sensors while forgoing the option of adding bias nodes. In time, we decided the best option was to design our sensors initially to provide a constant feed to the network.

Therefore in the avoid-obstacle controller, the network is provided two inputs. This sensor reading allows the agent to potentially 'see' as far as the length of the diagonal of the arena, which is sufficient to observe any of the surrounding walls. At every time-step of the simulation, the Oracle calculates the exact point where the sensor collides with a wall in the environment. From this point the Oracle calculates the distance of this point to the tank agent. The second network input is also based on the sensors point of collision, except this time the angle of intersection with the wall is calculated. Once again, each of these inputs are normalised, with the distance normalised with respect to the diagonal of the arena, and the angle within 90 degrees, the largest possible angle of intersection .

Creating a Fitness Function To address the second issue, we created a fitness function that would allow us to measure the performance of a candidate during the learning process. This led to a small stumbling block, since we needed to measure the performance of the agent in a quantitative manner yet all we really sought was a qualititative assessment. Given that this controller would not be used as part of the subsumption framework, nor would it be applied in future experiments, we sought only to assess whether these sensors could achieve our desired behaviour. A natural reaction to this situation is to simply transcribe our desired functionality - do not collide with any nearby obstacles - into a fitness function. However, one has to be aware of the trappings such fitness definitions can cause, since we are dealing with evolutionary search processes that are dictated solely by the parameters fed to it. In such a circumstance where we ask for an agent that does not collide with the nearby environment, the most effective solution is an agent that does not move. So how can one ensure the functionality arises in an effective fitness function?

Our improvised solution was to force exploration of the environment. If an agent is forced to visit portions of the environment and then fill it with obstacles it will satisfy our needs. Our initial intention was to prepare a $n \times n$ grid that would exist beneath the game map. The fitness function to assess this would then require adequate exploration of all grids within the matrix ($GridsPerAxis^2$). If an agent succeeded in visiting all grids, it would simply be penalised by the time taken, while failing solutions would receive fitness dictated by the number of grids

visited. The two resulting functions are found in Equations 3.8 and 3.9.

$$F_{succeed} = 1 - ((\frac{0.5}{T_{max}}) \times T_{game}) \tag{3.8}$$

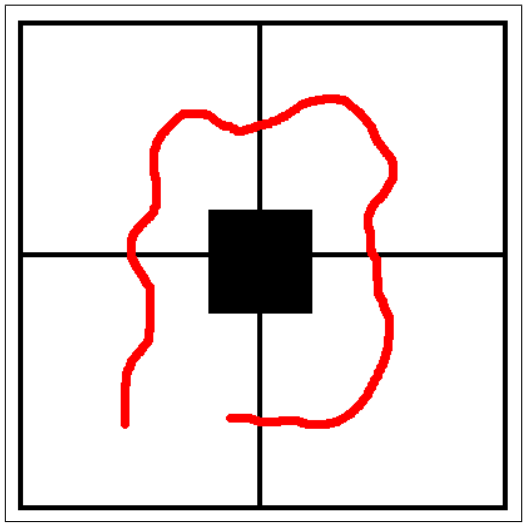
$$F_{lose} = (\frac{0.5}{GridsPerAxis^2}) \times GridsVisited \tag{3.9}$$

Testing In conjunction with this fitness criteria, we provided a series of test environments for the arena; simple arena, halved arena and quarter arena (shown in Figure 3.11). As the name would imply, the latter two divide the game world into halves and quarters that dictate a particular kind of navigation. Our initial consideration was to create more complex maps and in-turn increase the complexity of the exploration grids. However we hypothesised that this would ultimately lead to failure. If we consider that our agents are not 'aware' of the grid, then it would become increasingly difficult to visit each cell. This hypothesis can be surmised from Figure 3.10, where we give example paths of how to navigate increasingly larger grids.

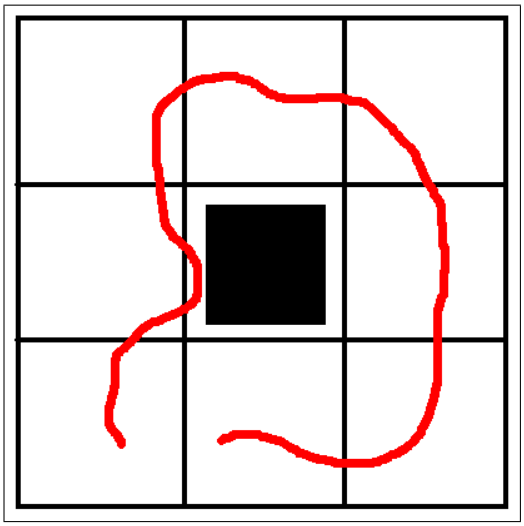
To verify this hypothesis we quickly tested our (1+1)ES on a 3×3 grid. As shown in Table 3.7, our concerns were justified. Furthermore, the resulting behaviours show little performance and give little indication as to whether the sensors are successful, leading to our tests being conducted using 2×2 grids. The results in Table 3.8 show that the agent was capable of manoeuvering throughout the environment without colliding with obstacles. Whilst the fitness results from the quarter-arena do not look too promising, the actual behaviours were more rewarding. In the first two arenas, the agent would be placed at random and would then learn to traverse the environment. In the later examples, the agent would often become trapped in one quarter of the arena as the response to the nearby walls would be too sensitive to allow for a stable path to be maintained. Interestingly, applying neural nets from the first two experiments in the quartered arena proved capable of navigating irrespective of their starting position. Having observed these behaviours, our confidence determined we could proceed.

Detect Obstacles

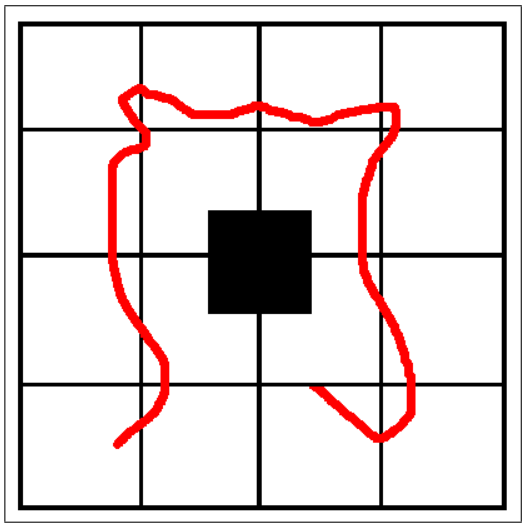
The results from the avoid-obstacle experiments provided sufficient evidence that obstacle detection and avoidance using our sensor setup could be achieved. Whilst the biggest issue with the avoid-obstacle results was the agents' inability to



(a) 2×2



(b) 3×3

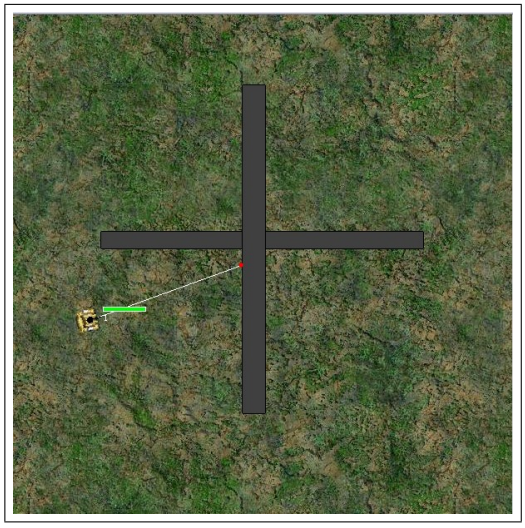


(c) 4×4

Figure 3.10: Example paths that we predict the agent would need to traverse in order to complete exploration grids of sizes 2×2 (a), 3×3 (b) and 4×4 (c) in the simple-arena. Note that as the number of grids per axis increases, the path required will become more complex. This would prove difficult for a simple fitness function to model succinctly.

(a) Simple Arena

(b) Halved Arena



(c) Quarter Arena

Figure 3.11: The three arenas used for the training phase of the avoid obstacle controllers. Each places the obstacles to separate the environment in different ways.

Table 3.7: The average results of testing the avoid-obstacle controller using a (1+1)ES on a 3×3 exploration grids (three runs per map). As predicted, the agents perform poorly.

Arena	Average Fitness (Std. Dev.)
Simple-Arena	0.409 ($\pm$0.0102)
Halved-Arena	0.388 ($\pm$0.0445)
Quarter-Arena	0.347 ($\pm$0.077)

Table 3.8: The average scores from agents attempting to navigate through a 2×2 arena using the avoid-obstacle controller. While highly successful in the first two tests, agents did not fare well in the quarter-arena.

Arena	Average Fitness (Std. Dev)
Simple-Arena	0.776 ($\pm$0.027)
Halved-Arena	0.693 ($\pm$0.016)
Quarter-Arena	0.363 ($\pm$0.217)

navigate complex environments, this was no longer a concern, since it is expected that our agents' navigation will be dictated by any of the first three controllers described in this chapter. The goal of an obstacle detection controller is simply to recognise potential collisions and move the agent away from them. Now that we have discerned that the avoid-obstacle setup can recognise and avoid obstacles, to an extent, the next step is to reduce and improve the setup to work as a layer controller within the subsumption framework.

The controller is once again a feed-forward multi-layer network, with 2 mid-layers of 3 neurons. As the agent is responsible solely for navigation, we only allow it control of the rotation since we assume it will continue moving courtesy of the lower layers. The controller contains the largest number of input sensors, with four different sensors designed to assist the agent in obstacle avoidance and recognition. The first two inputs are based on the main sensor within the avoid-obstacle controller. The agent relies on a forward sensor that unlike our previous design only stretches 75 pixels ahead. We tested using a range of distances from 50 pixels to 150, but we found that in practice a distance of 75 was reasonably adequate. This sensor once again feeds two inputs; the distance to the intersecting wall - normalised with respect to the arena such that the closer the obstacle is, the stronger the signal - and the angle that the agent intersects with the wall.

Figure 3.12: A tank approaching an obstacle at angle a.

Normalising the intersecting angle was a challenging prospect that had to be addressed.

In initial tests of the avoid-obstacle controller, we noted that the agent would often navigate around obstacles in one direction. Our first hypothesis was that this occurred as a result of our preliminary normalisation method. Similar to previous angle based inputs, we converted the values into polar coordinates. However, unlike our previous designs we may require the agent to also consider its current trajectory towards the obstacle. By referencing Figure 3.12, we observe that an agent may be moving towards an obstacle with angle a, meanwhile it could also generate the same intersection angle b by approaching the wall on a different trajectory. Our interest was now on how we could exploit this information to suggest the correct action in a given circumstance. A simple approach is to apply basic trigonometric functions to ascertain the gradient or slope of the line from the front of the tank to the edge of the sensor with respect to the surface of the obstacle. The slope describes the 'steepness' or 'grade' of the line from the tank's current position to the sensor. If we have the (x, y) coordinates of each object, then we can simply calculate the slope 'm' as shown in Equation 3.10.

$$m = \frac{\Delta y}{\Delta x} = \frac{y_{sensor} - y_{tank}}{x_{sensor} - x_{tank}} \tag{3.10}$$

The larger the value of this slope, then the steeper the line. Hence a slope parallel to the wall (i.e. of 0 degrees) is considered to have a slope of 0. Whilst a trajectory perpendicular to the obstacle would be deemed to have no slope, this would require an absolutely perfect 90 degrees intersection. Once the slope of the trajectory has been calculated, we can then calculate the actual angle of intersection the agent will make with the wall shown in Figure 3.12. Provided

the intersecting surface has no slope, which is the case in EvoTanks, since all surfaces are either perfectly horizontal of vertical), then we can calculate the intersecting angle with relative ease by taking the inverse or arc tangent of the slope (Equation 3.11).

$$\theta = \arctan(m) \tag{3.11}$$

By employing this method and running some preliminary tests, we came to several conclusions, notably:

- Should the slope (m) be positive, and the obstacle surface be horizontal (i.e. $\Delta y = 0$) then the resulting angle will be positive. This will suggest the agent turn right to ride along the perimeter of the wall in a counter-clockwise direction.
- Conversely, should the slope be negative and the obstacle surface is horizontal, then the angle will be negative. Suggesting the agent should turn left and navigate the obstacle in a clockwise fashion.
- Should the slope intersect a vertical wall surface (i.e. $\Delta x = 0$), we must first calculate the angle as normal for both types of slope, then negate the complementary angle. This will give the correct angle and sign for this situation.

The data from the slope and angle of intersection would allow us to provide useful information to the neural network controller. The angle values were then normalised within the range of $-1 \leq x \leq 1$, where negative values always corresponded to left turns and positive values to right turns. Irrespective of sign, the stronger value would always indicate a greater need to turn. Hence the agent should intend to reduce this value by turning accordingly and then riding parallel to the obstacle surface. where a stronger negative value would arise if the slope was positive to suggest left turns. In the following section we provide experimental results from applying this controller in different navigation challenges.

Dodge-Shells

Our second layer controller was (as suggested by the title) was introduced to address an outstanding issue from our previous research, the incoming fire from enemy agents. As observed earlier in this chapter, the destroy-target uses a

challenging fitness function for the agent. This is the result of having part of the assessment based on how much of the agents' health remains, despite having no knowledge of the incoming shells. This controller sought to address this, allowing agents to generate different behaviours for navigation and combat by exploiting the given information.

Furthermore, the controller relies on a fairly simple topology and setup. The controller is fed only information on the nearest incoming shell. The importance of a shell is dictated by a semi-circle with a radius of 50 pixels that covers the front of the agent. The closest shell within this semi-circle is then determined to be the most relevant. From this shell we gather two inputs for the network; the angle of the shell relative to the tanks current heading, and the distance. Once again these values are normalised in accordance with previous angle and distance inputs. In our initial tests, we had a full circle of visibility rather than a semi-circle. However this often led to circumstances where the agent would react to situations that were unnecessary, such as enemy fire it has previously avoided passing behind it. This controller was subject to a series of varying experiments to address how well it would perform in both navigation and combat circumstances, notably how the information could be used in a navigation task - such as reaching a distant point in the world - or in a combat scenario against an opposing player. For navigation experiments this can easily be achieved by introducing one or more Turret players into a given scenario, forcing the agent to avoid incoming fire. However, when dealing with combat experiments, this is one controller where no changes are required, as we simply shift our controller focus during the subsumption training. In the following section, we explore applying this controller in both of these scenarios and highlight the results and overall performance in testing.

Summary

As shown in Table 3.9, we provide a short summary of each controller, their inputs, outputs and standard topology. Now that our controllers were formally defined and the base controllers suitably tested, our next phase was to insert them into the subsumption framework and explore the performance and adaptability of our additional layer controllers to handle new external stimuli.

Table 3.9: A summary of the ANN topology and input vectors for our sub-controller designs.

Name	Topology	Inputs	Outputs
Visit-Waypoint	2-3-3-2	Distance Direction	Movement Rotation
Destroy-Target	3-3-3-3	Distance Direction(×2)	Movement Rotation Fire
Grab-Item	2-3-3-3	Distance Direction	Movement Rotation Grab
Detect-Obstacles	4-3-3-1	Intersect Angle(×3) Distance	Rotation
Dodge-Shells	2-3-3-1	Distance Direction	Rotation

3.4.2 Layered Controllers

In this section we explore combinations of the aforementioned individual controllers within the subsumption framework. We have introduced our three base controllers - visit-waypoint, destroy-target and grab-item - in detail in the previous section, whilst also giving an indication of the two additional layer controllers - detect-obstacles and dodge-shells - that would be used for handling added complexity within any given environment. We provide now some feedback on how well these ANN designs adapt to changes in the environment, and whether our prescribed learning methodology in Algorithm 3 would prove sufficient for this task.

Visit-Waypoint & Detect-Obstacles

Our first experiment explores the effectiveness of introducing our obstacle avoidance controller with the navigation behaviour provided by the visit-waypoint controller. We observed from the results shown in Table 3.4 and Figure 3.6 that the controller proved more than adequate for visiting randomly instantiated waypoints within a given environment. Now we must assess the performance of the detect-obstacle controller in these obstacle littered scenarios.

To achieve this, we need to decide how to add the obstacles to the environment. A simple - yet as we shall see, rather effective - approach was to maintain

the original experiment parameters for the visit-waypoint experiments shown in Section 3.4.1. We position a waypoint randomly in the environment, provided it is at least 600 pixels from the agent. In addition we place a fixed, static obstacle in the arena. This is a fixed 200×200 obstacle that sits exactly in the centre of the arena as shown in Figure 3.15.

Naturally, we need to ensure variety in training to prevent niche behaviours from emerging. So the reader may question the use of a single, fixed obstacle. However, if we consider the nature of the inputs for the detect-obstacle as discussed in Section 3.4.1 (and in fact all inputs we provide to our agents), the data is egocentric, i.e. all data is relative to the agent utilising it. Hence the position of the wall in the arena is irrelevant, rather, it is where the wall is from the agent. Provided the position of the agent and the goal are generated at random with the obstacle always impeding the path, it ensures the agent will always have to navigate around it. Furthermore, since we are dealing with random positioning, it means the traversal and avoidance of the obstacle will differ in each instance, hence the inputs and more importantly the experience for the agent will differ in each evaluation. This effectively achieves the same results as moving the wall with minimal effort.

We conducted ten tests of the (1+1) ES using our layered learning approach (Algorithm 3). Each layer was given 50,000 games to achieve the best possible fitness. We provide learning trends of three experiments in Figure 3.13 and the table of statistics in Table 3.10. As can be seen the agent quickly learns how to visit the randomly assigned waypoints in the environment. Once we reach the midpoint in the graph, the point where the first controller is frozen, we introduce the detect-obstacle controller for training. The first point to note is that the agent quickly corrects any issues with the obstacle and learns an effective strategy for passing them. However, there is a drop in overall fitness that may at first glance appear disappointing. However, the agent is still under the learning criteria of the original fitness function, receiving a penalty for the time taken to reach the goal location. Given that the agent now has to navigate around this fixed obstacle, the average time to complete an evaluation will increase with the penalty in each run also advancing. Hence, we should expect a small reduction in the best performance.

Referring to the statistics in Table 3.10 we see an average of 0.756, which is a modest decrease from the 0.84 average found in the visit-waypoint experiments. The standard deviation and sample size accrued indicate that we can be confident

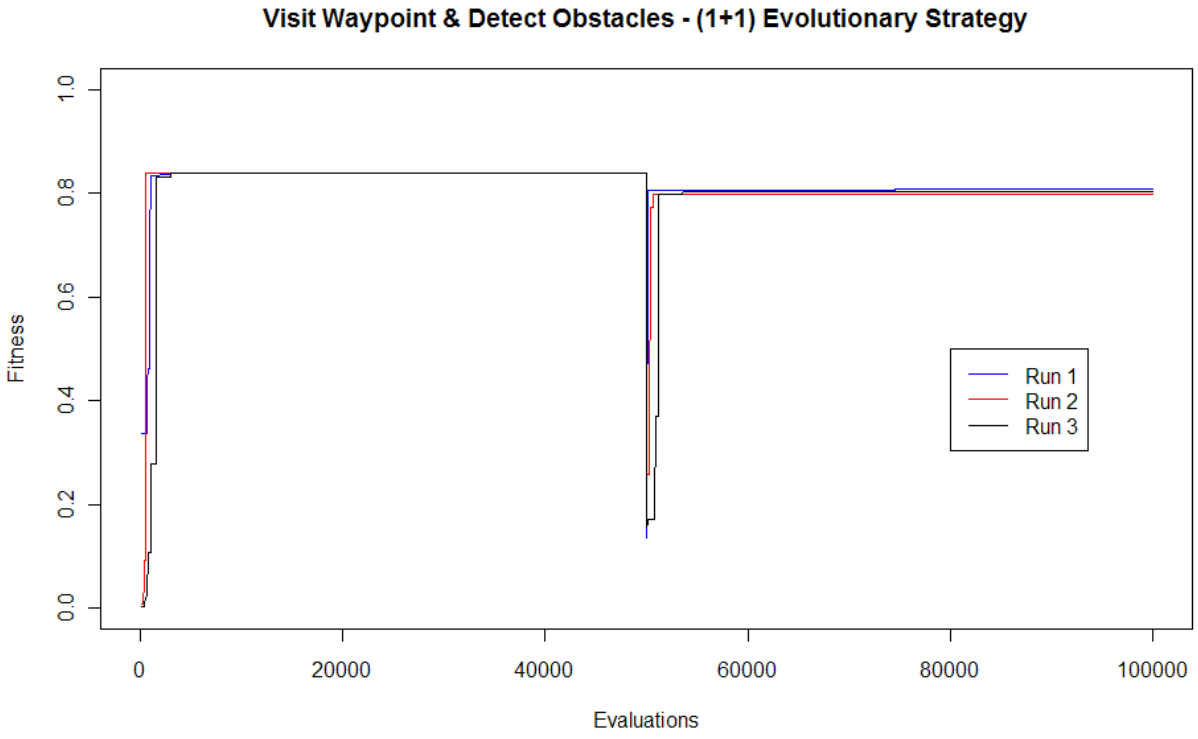


Figure 3.13: The result of training against the visit-waypoint criteria firstly in an empty environment and then in a cluttered environment. It is important to note that after the sudden drop in fitness in the centre of each plot, the subcontroller being trained shifts to the next layer of the architecture for training.

in the results attained. The final behaviours rely heavily on the sensors provided, as expected. The agents will exhibit a 'wall-hugging' behaviour when in the vicinity of obstacles. Once a sensor has established there is an obstacle nearby, the agent will follow the perimeter until it has passed the wall and the path to the goal is clear. Quality in this behaviour can vary, since we will see instances where the agent will either traverse the perimeter smoothly or will rock back and forth in a unrefined manner. Nevertheless these controllers prove to be fairly robust, since they are able to clear different obstacle-filled environments in other tests after this initial training, and again in Chapter 5.

There was however one small setback; each controller would *always* traverse a given obstacle counter-clockwise. We hypothesised at the time that this would not be an issue and correcting the normalisation previously applied to the input is all that would be necessary. However, we soon learned this was not the case. We spent an exorbitant amount of time on different normalisation approaches such as simple inversion of previous values to scaling to a smaller range. We even explored removing the normalisation entirely or simply providing a specific value ($+1$ or -1) as input to trigger the desired action. Irrespective of our efforts, successful experiments would always find counter-clockwise traversal the best choice. Often solutions would explore the notion of travelling clockwise but would often fail to complete the task in this fashion. After much consideration, we are still uncertain as to why this approach - which we believed to be a simple and plausible one - would not yield a behaviour to our standards, much to our frustration! We hypothesise that the network may propagate a strong positive signal with respect to the turn action. This in turn would lead to our best solutions, where they would always travel counter-clockwise around an object competently and efficiently. Like any parent, we are frustrated with how stubborn our creations have been, but we cannot argue with the effectiveness of the resulting behaviour.

Destroy-Target & Dodge-Shells

Our second layered experiment explores the application of the dodge-shell controller. For this experiment we sought to add this functionality to the destroy-target controller. In this experiment, we explore whether providing this desired information through the subsumption layer leads to improvement.

Once again we need to set the modifications to the environment for this experiment to take place, however in this instance no changes are required! Given

Table 3.10: A list of statistics based on 10 runs of the visit-waypoint & detect-obstacle SNA controller.

Statistic	Value
Maximum	0.8078
Minimum	0.6305
Mean	0.756
Std. Dev.(σ)	0.069
Sample Size (α = 0.1)	1.828

that we first train the destroy-target controller, there will already be an enemy agent that is shooting our agent. Hence, the necessary stimuli for the new controller is already provided. In the event that this stimuli was not already present, we would introduce a Turret for the experiment to ensure there is incoming enemy fire.

We now provide results based on running this 2-tier controller. In Figure 3.14 we provide an indication of the learning trends from three arbitrary runs of this experiment, while in Table 3.11 we provide a statistical breakdown of ten runs of the experiment. Note that this experiment only permits 300,000 evaluations for the destroy-target phase as a before, with 600,000 permitted overall. The learning trends shown in Figure 3.14 show promise, as we note that the best fitness of the agent actually *improves* after we introduce the additional layer. Unlike the previous 2-layer experiment the agent has actually improved in performance because it is now avoiding the incoming enemy fire better than it had previously done, thereby scoring better in the fitness function. Whilst the difference in fitness between training phases is not large, it leads to significantly different playing strategies.

We observed that the players would pay attention to incoming fire and attempt to avoid it as it approached the enemy. When dealing with enemies in close quarters we see two strains; one which will continue to dodge the fire at the expense of a potential attack, or simply ignore fire at close range and develop a *Hunter*-like behaviour - attacking the enemy relentlessly until only one is left standing. Meanwhile these new faculties appeared beneficial to agents that would attempt to pick off the enemy from a distance, since they could maintain a defensive position and only occasionally shift to avoid fire, subsequently returning to a fixed position.

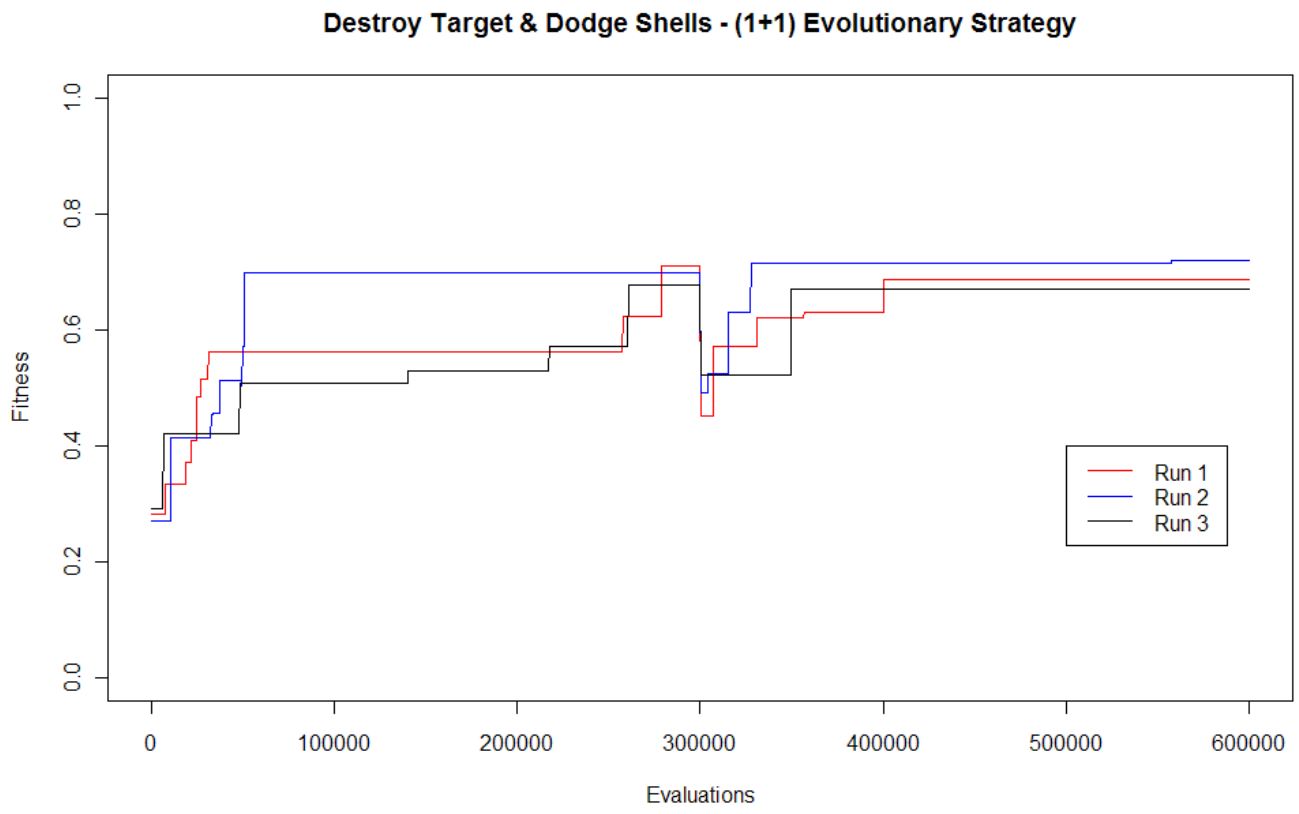


Figure 3.14: The result of training against the destroy-target criteria firstly against the agents with the base controller and then with shell dodging faculties.

Table 3.11: Statistics from 10 runs of a 2-layer architecture containing the destroy-target controller at base level and the dodge-shells network as a layer controller. Results overall prove promising, with high final fitness that almost reaches the same levels as those shown in Table 3.5. The low minimum fitness is also of interest, as it is slightly higher than that found in Table 3.5.

Statistic	Value
Maximum	0.72
Minimum	0.571
Mean	0.646
Std. Dev.(σ)	0.0479
Sample Size ($\alpha = 0.1$)	0.883

Statistically, we note that the additional controller can provide a small boost to the final score. As we can see from the minimum fitness in this experiment, as well as the original destroy-target tests in Table 3.5, this is a challenging test for the agents to surpass, however it appears that the supplementary layer can improve final scores, as well as impact behaviour. In closing, this allows us to add the information as a separate layer to the controller without increasing the number of neurons/weights in the base ANN. The difference in mean fitness between the standalone controller (Table 3.5) and the additonal layer experiments in (Table 3.11) are marginal across the 10 individual runs, with the single-layer approach performing better on average by 0.003 fitness. Agents tend to 'shuffle' when in the presence of enemy fire; quickly navigating away from a shell and then returning control to the lower layers. We have considered the possibility that while the new behaviours are more efficient in terms of health retained, they take longer due to the evasive shuffle. This would result in the time taken increasing, and hence the penalty for increased time taken may well have had an effect on the mean fitness recorded.

3-Layer Controller: Visit-Waypoint, Detect-Obstacle & Dodge-Shells

We have shown that we can combine two controllers using the subsumption hierarchy. Our next step is now to combine the maximum number of three controllers into one subsumption controller. For this experiment we focus on the navigation problem shown in Figure 3.15, where the agent must avoid obstacles and incoming enemy fire, thus requiring the visit-waypoint, detect-obstacle and dodge-shells controllers.

The training is similar to the previous 2-layer navigation task, however after the detect-obstacle controller has completed, it too is frozen to allow for the dodge-shell controller to be trained. Given that we are now training the dodge-shell control, we require two turret agents that are placed within close proximity of the waypoint. This would prove challenging for any agent learning this task, as the turrets are arguably the most difficult script to overcome due to their strong firing accuracy and offensive play.

Once again we conduct ten runs of the experiment, each controller being given 50,000 evaluations for training. We provide learning trends from three arbitrary runs in Figure 3.16. We again we see a gradual decline in best fitness seen previously in Figure 3.13 with the first two phases similar to our previous

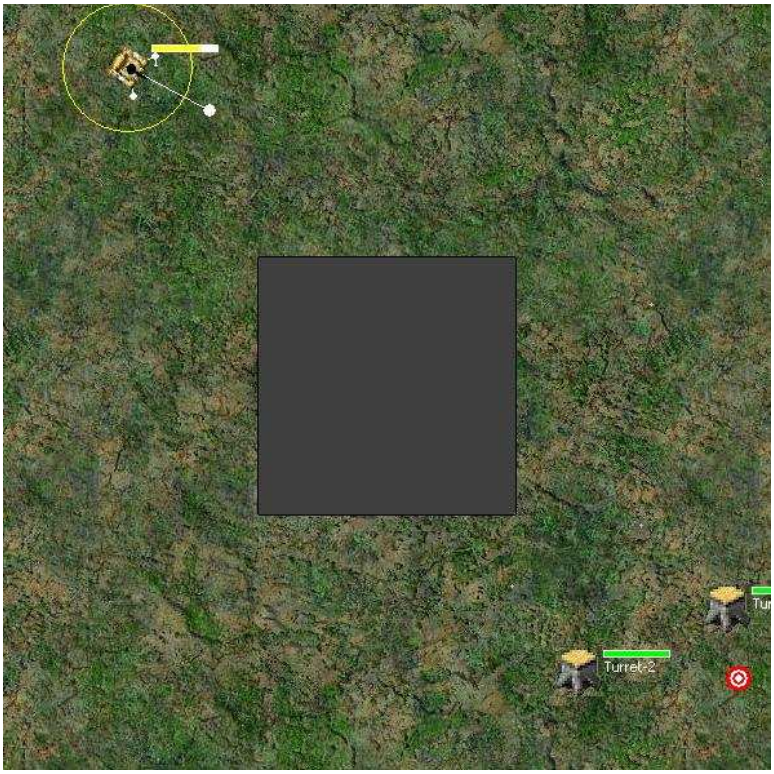


Figure 3.15: The training scenario for our three layer controller, requiring obstacle avoidance and shell-dodging capabilities.

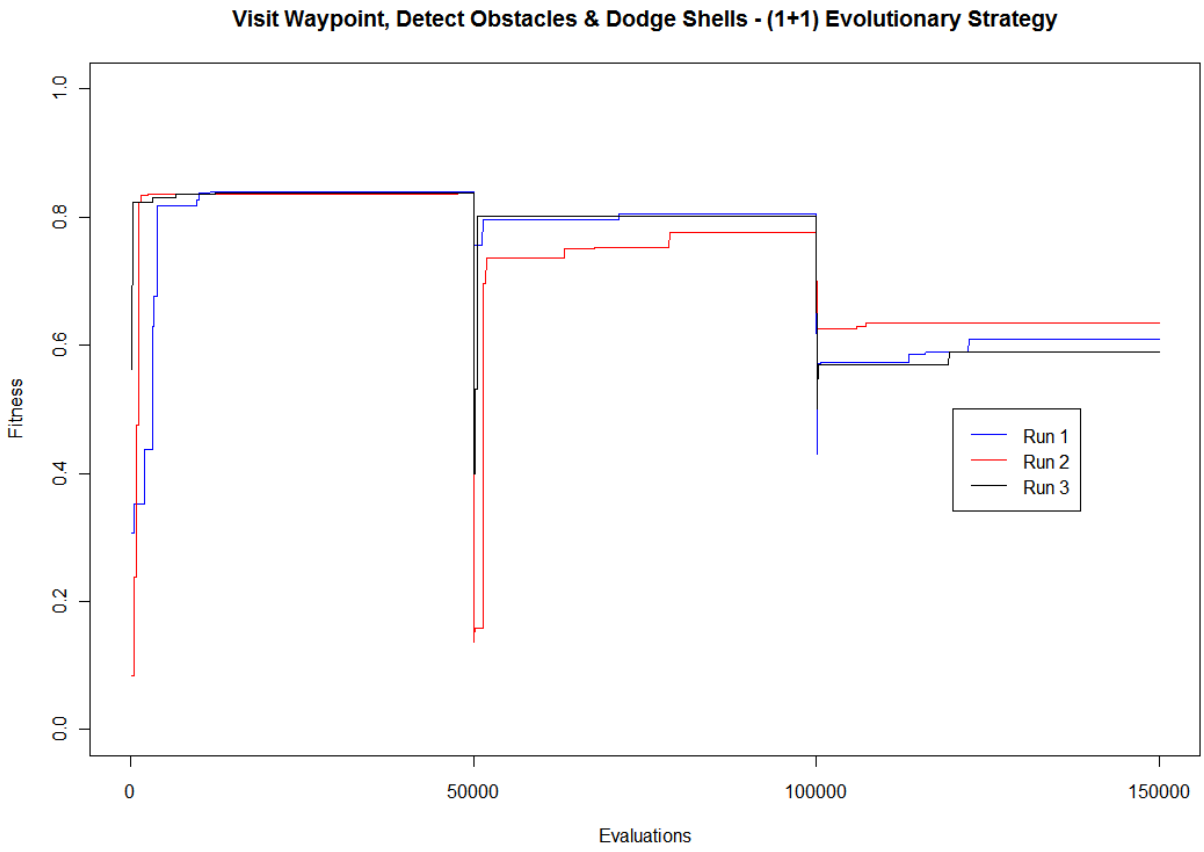


Figure 3.16: A breakdown of running the 3-tier navigation problem using the evolutionary strategy. The performance in the first two learning phases are similar to that shown in Figure 3.13, however we see the agent learn to compensate for the added complexity of the incoming enemy fire in the final training segement.

Table 3.12: Statistics of 10 runs on the 3-layer subsumption experiment. We see a range of results that suggest the agent can continue to perform without significant loss of overall fitness.

Statistic	Value
Maximum	0.642
Minimum	0.589
Mean	0.618
Std. Dev.(σ)	0.0176
Sample Size (α = 0.1)	0.119

results. We observe there is a larger decrease in fitness between the second and third phases of training. This can be accounted for with two important facts: firstly the fitness function's time penalty will again be an important factor, since the agents will be taking even longer than before to reach the target, given that they are having to avoid incoming enemy fire. Secondly, while the resulting behaviours prove successful, there is still a chance that agents may on occasion be overwhelmed and destroyed by the turrets. While the shuffle action is an effective strategy, it is not always successful. Nevertheless this proves to be a success, since the agent is able to learn how to avoid the oncoming enemy fire as it moves towards the goal. Further improvement could be made by providing a greater understanding of the trajectory of an incoming shell or even the position of the agent that fired the shot. Trajectories could potentially be modelled using a recurrent ANN that feeds the previous timesteps outputs back to the network for processing. The statistical data from these experiments is shown in Table 3.12, and we are satisfied with these final results. The mean fitness is relatively strong considering the additional challenges, with on average a drop in fitness of less than 0.15. In game cycles this means that the agent now takes, approximately, an extra 300 cycles on average to solve the task. Our sample size and relatively low standard deviation also provides high confidence in these results.

It becomes clearer through this navigation problem that the fitness of the final product is strongly dependent on the results of the very first controller in the architecture. This effectively 'caps' fitness with an upper bound, and any subsequent controllers will be challenged to ensure the drop-off is not substantial. Interestingly it also has an effect on a population based learning approach. In Figures 3.17 and 3.18 we provide learning trends from our EA approach. Here we give a population of 200 candidates 10 generations to search for high fitness

candidates (again 50 games per candidate). First we can observe that the best results are once again strong, with best fitness slightly above the mean from Table 3.12. However it is the average population fitness that is the focus here. Note that while the best solutions are typically slightly below the ES approach, the average fitness is significantly poorer with a large amount of deviation from the mean. Thankfully as we can see, the average does improve as the learning process progresses. However we see two interesting events during the learning process. Firstly, as we shift the focus to the next controller, we see a dramatic increase in average fitness. This is because the entire population now shares the best controller from the previous phase as the base control. Now that additional layers are being trained, we have - in contrast to the upper bound explained earlier - a lower bound on fitness that is introduced by retaining vital information. In both instances we see gradual improvement within the populations and as we enter the final phase we see another interesting development, when training the final layer the standard deviation of the population tightens significantly. This results in convergence in the population as well as little difference between the population mean and the best solution. This occurs not only due to the introduction of another frozen controller - increasing the lower-bound - it also is due to the nature of the final layer. This layer is required to avoid incoming enemy fire where necessary, otherwise relinquish control to the pre-established layers. Furthermore, it has little ability to impede or modify previously trained behaviour. This often means that behaviour and fitness for the most part will be relatively similar, hence the dramatically reduced standard deviation within the population.

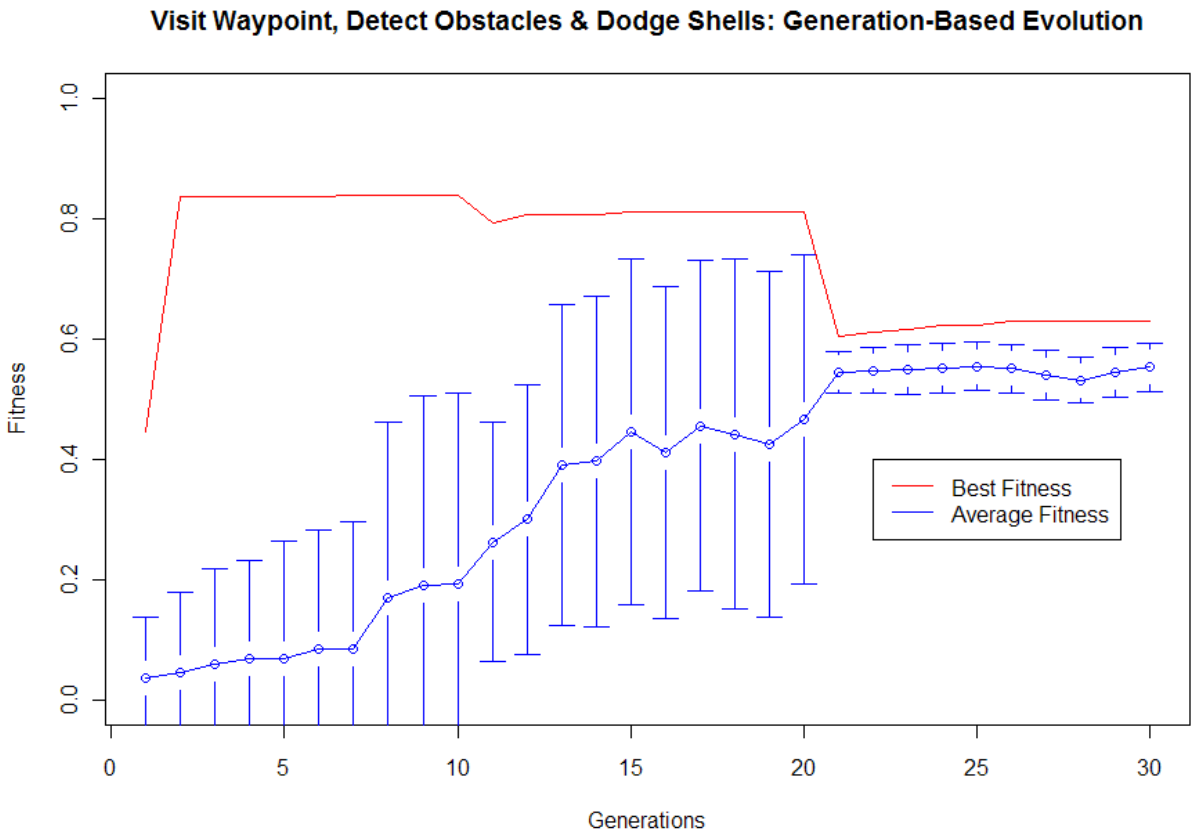


Figure 3.17: A sample run of our population-based evolution algorithms on the 3-tier experiments. The evolution while successful does not provide the same high scores as the evolutionary strategy in Figure 3.16.

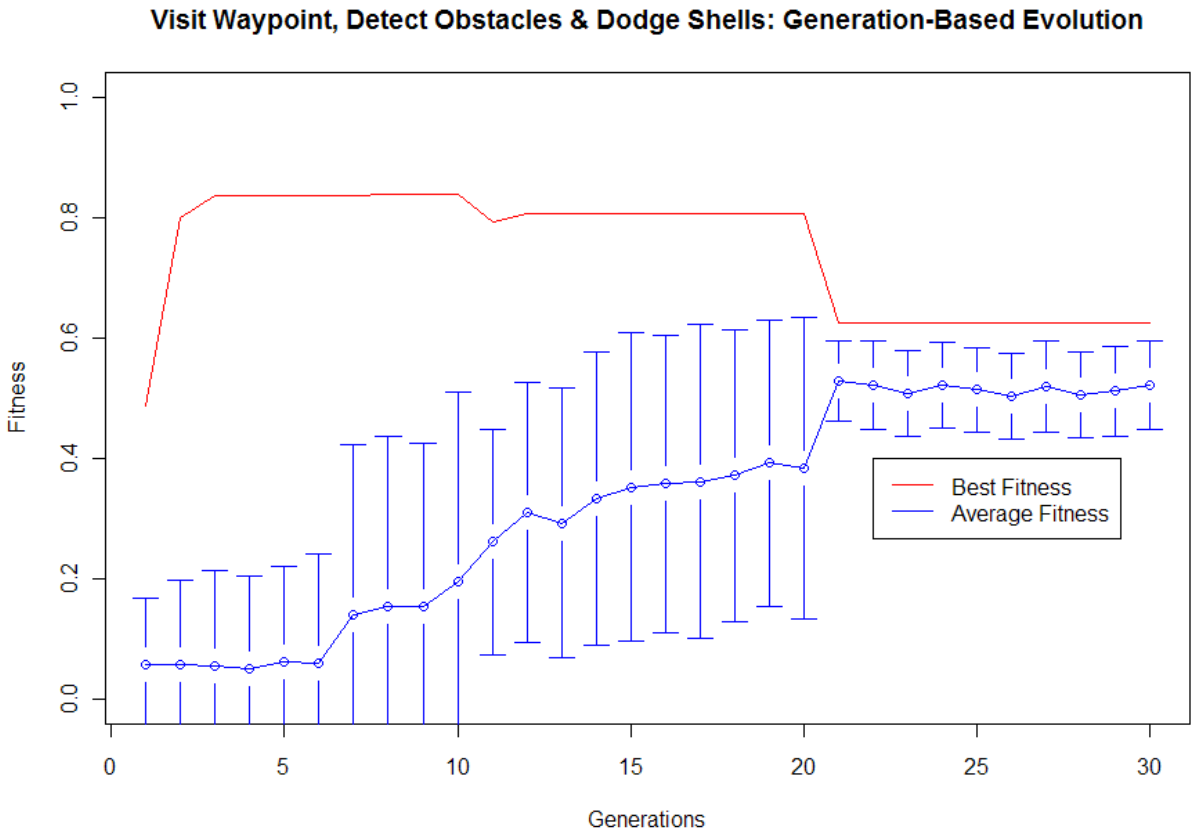


Figure 3.18: A second sample run of the evolution process on the 3-tier experiment. The results prove similar to those shown in Figure 3.17. The average fitness trends show the population gradually improves as training progresses, in part due to the learning process, but also due to the addition of new controllers as training shifts focus.

3.5 Comparative Results

> Facts are stubborn things, but statistics are more pliable.
>
> Mark Twain

So far in this chapter, we have provided a breakdown of each of our individual controllers created in the EvoTanks environment for the subsumption framework. Our results from the previous section provided sufficient evidence to suggest that the approach we have described can effectively combine behaviours using a simple, staged learning process. This process allows us to add additional complexity to the game environment and then compensate for it using the subsumption approach.

In this section we will address how well this approach performs in comparison to more traditional learning approaches. To achieve this, we focussed primarily on using traditional, monolithic neuroevolution methods. This was achieved by training a standard, feed-forward ANN to take in all available inputs relevant to the task at hand. Given that this would lead to a far larger number of inputs than previous, more neurons were provided to the hidden layers. Given the fully connected approach we have maintained within our networks, we decided to provide 10 neurons per hidden layer. This in turn increases the number of dimensions available to explore the behavioural search space, since we provide many more synapses between layers to retain the fully connected topology. Whilst this could potentially prove to be a drawback for the agent, given the far larger search space to explore, we were confident that adequate results would be generated by this size of ANN.

Next we had to provide a suitable scenario for comparison. It was decided that the 3-layer task - visit-waypoint, detect-obstacle, dodge-shell - would prove a challenging prospect for all parties. As shown previously in figure 3.15, this requires the agents to reach points in the world whilst navigating around obstacles and avoiding enemy fire. Not only is this challenging in terms of the environmental complexity, but also having to contend with eight inputs from the environment. While an input vector of this size is far from uncommon in neuroevolution research, it was the driving factor behind the increase in neural network sizes explained previously, ultimately leading to a complete neural network with a total of 210 synapses through the fully connected topology.

We conducted two experiments for the comparison. The first approach was

handed the complete problem for each evaluation. Therefore in every EvoTanks game a random waypoint was assigned within an obstacle-filled arena with enemy agents and the player had to reach the waypoint as usual. The second method was fed the problem in stages, akin to the staged learning applied in Algorithm 3 using the same milestones to instigate change to the environment.

At this time we did consider whether applying incremental or modular approaches for comparison would prove a worthwhile venture, given that a subsumption approach could be argued to carry strengths from both approaches. However, we felt that any attempt to apply an incremental learning algorithm would contradict with the concept underlying our staged learning process. This is largely due to the necessity in changing fitness function during an incremental learning algorithm. To achieve this, we would need to create an additional fitness function for each of the layer controllers, and then find an appropriate weighting between each fitness in order to maintain a smooth landscape. However, this would contradict the approach we have taken throughout. This is also the reason why we did not compare these results against Togelius' approach, as we do not wish to assess individual layers under a unique criteria. The underlying concept of the approach was to build upon the initial behaviour without the necessity of additional functions. Furthermore, a modular approach was not considered for experimentation. While we were under time constraints during development, we concede that a modular experimentation could be explored. However we consider it to be *too* modular, since it operates on a neuron by neuron bases. This approach while modular seeks to retain an identifiable functionality within each individual network, that which a traditional modular approach cannot provide.

For the experiment, each method was given 150,000 evaluations (3000 evaluations at 50 games each) to complete the learning process. This is the same number of evaluations given to the 3-layer experiment discussed previously, hence ensuring a level playing field. In the incremental environment experiment, we must make changes to the complexity of the game state at the same points in our subsumption experiment. Hence at the 50,000 and 100,000 evaluations, we instigate change in the world. Each experiment is a (1+1) ES to ensure a fair comparison between all three experiments.

To provide an effective and valuable measurment of comparison. We employ the Student t-test to assess the statistical difference and potential significance between these approaches. The t-test is calculated as shown in Equation 3.12 below:

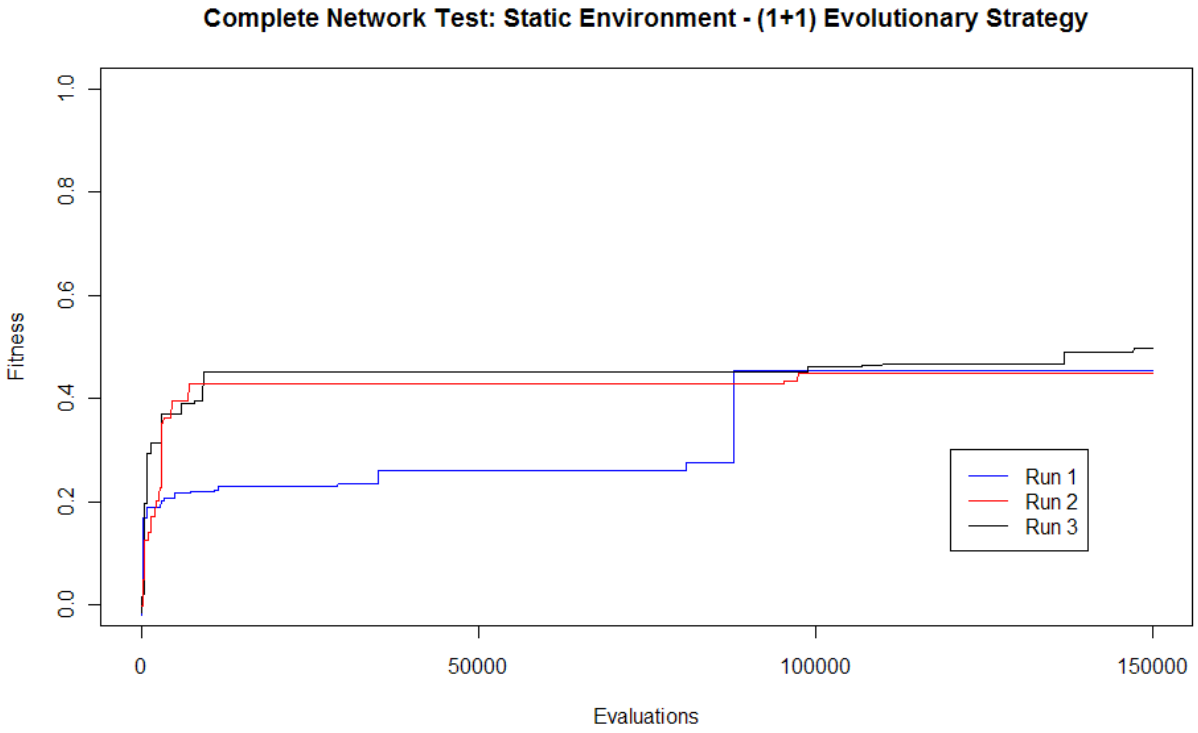


Figure 3.19: A graph of three runs on the complete neural network approach with a static, complete environment. In each instance the agents fail to complete the task, with the agent failing to pass beyond the 0.5 in all instances.

$$t = \frac{\bar{x_\alpha} - \bar{x_\beta}}{\sqrt{\frac{var_\alpha}{n_\alpha} + \frac{var_\beta}{n_\beta}}} \tag{3.12}$$

The t-test is designed to assess two datasets α and β by calculating the difference between group means ($\bar{x_\alpha}$ and $\bar{x_\beta}$) against the variability of the samples (the standard error of the mean difference). The resulting t-value can then be utilised to assess whether this ratio is statistically significant. Each experiment is conducted ten times, and based on this information we then conducted the t-test.

Our calculations are based on the initial results of the 3-layer experiment in Table 3.12 and the resulting data from our comparative experiments. The complete network test results are found in Table 3.13 and the incremental evironment test results in Table 3.14. Furthermore, we provide learning trends of three arbitrary runs from these results in Figures 3.19 and 3.20.

The results for the complete network approach prove interesting, as can

Table 3.13: A list of statistics based on 10 runs of the complete controller training on the static environment. The maximum and mean suggest that the agent can perform reasonably in some instances. However the minimum score recorded and large standard deviation tell a less promising tale.

Statistic	Value
Maximum	0.5032
Minimum	0.4481
Mean	0.47
Std. Dev.(σ)	0.0188
Sample Size (α = 0.1)	0.1356

Table 3.14: A list of statistics based on 10 runs of the complete controller training on the incremental environment. The results do not prove as effective as those found in the other experiment in Table 3.13.

Statistic	Value
Maximum	0.4636
Minimum	0.376
Mean	0.4152
Std. Dev.(σ)	0.02918
Sample Size (α = 0.1)	0.3272

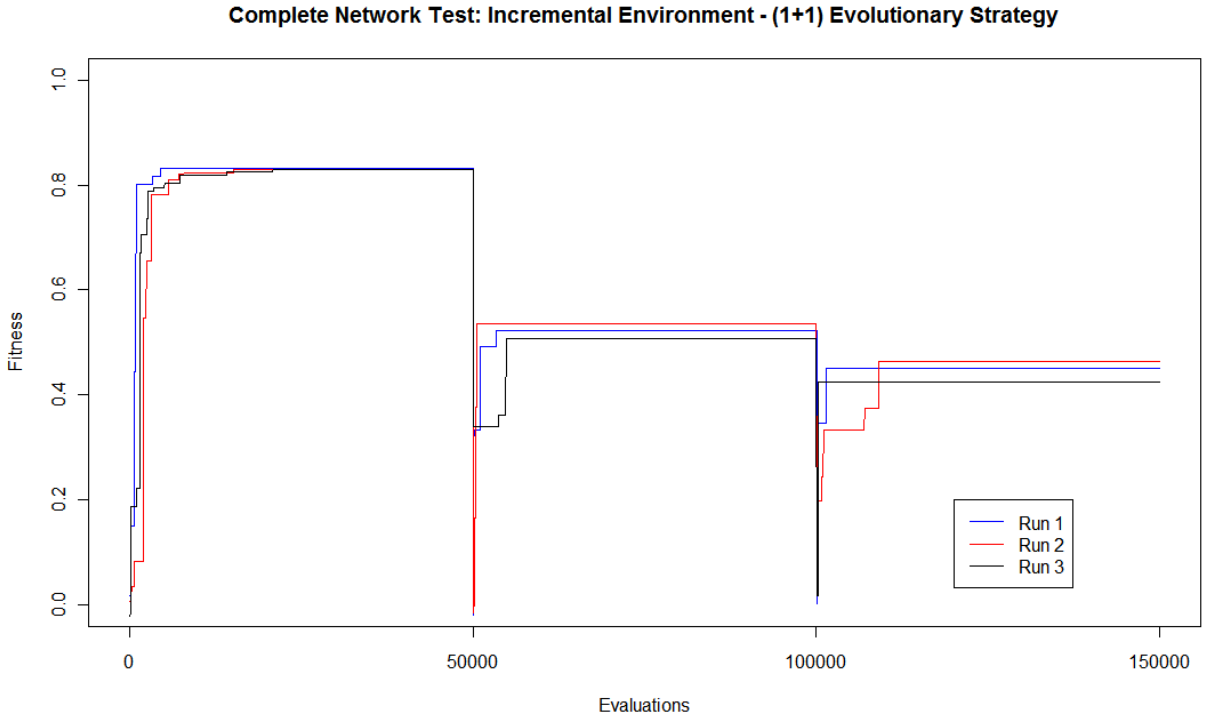


Figure 3.20: A graph of three runs of the complete neural network approach with an incremental environment. In each instance the agent's performance decreases below establised norms (found in Figure 3.16). Given that the subsumption approach succeeded on average while the complete network scores below the 0.5 watermark.

Table 3.15: A breakdown of the results of our t-test calculations.

Statistic	Value
3-Layer Mean	0.618
Complete Static Mean	0.47
Complete Increment Mean	0.415
3-Layer vs Complete Static t-value	2.46
3-Layer vs Complete Increment t-value	2.97
Complete Static vs Complete Increment t-value	0.386
p-value ($\alpha = 0.05$, $df = 18$)	2.101
p-value ($\alpha = 0.01$, $df = 18$)	2.880

be seen from Table 3.13 and Figure 3.19. The three arbitrary learning trends would suggest that this approach is ill-suited to the problem, with the mean fitness significantly below the 0.5 success threshold. Furthermore, by applying an incremental approach the learning process does not succeed on average. The learning trends in Figure 3.20 are similar to those previously shown in our 3-layer experiment (Figure 3.16). However, the drop in fitness between training phases is too steep and the agent fails to recover from it. This is corroborated by the statistics in Table 3.14 which shows the average fitness well beneath the 0.5 threshold.

Once again we provide the sample size results for each of the experiments. Our confidence in the results of the incremental environment are very strong, given that it requires only one run in order to achieve a 90% confidence in the mean and standard deviation. However, we require one extra run for the static environment approach, nonetheless we have high confidence in the 10 runs executed. This sample size from the complete run indicates that this approach can succeed given the confidence we have in the mean and the standard deviation. While this is far from an optimal solution it does indicate that it is feasible and could potentially be explored using different network topologies.

Next we provide the resulting t-values from all three experiments in Table 3.15. The results prove are satisfying, with the t-values exceeding the threshold of the p-value at the 5% level for the 3-layer approach being compared to both the static and incremental approaches. In this instance, we disprove the null hypothesis that there is no difference in performance. This statistically proves that the 3-layer approach to this problem is more effective by comparison. Even more encouraging is the t-value of the comparison between the 3-layer approach and the incremental

environment, indicating that we also disprove the null hypothesis at the 1% level. However, it is interesting to note that while there are significant differences in the means and standard deviation, the t-test cannot statistically verify a difference in performance between the 3-layer approach and the static environment method at the 1% level. This could be attributed to the difference in mean fitness being less than 0.2 for these approaches. We have hypothesised that given the continued performance of the 3-layer approach, we could prove statistical significance at the 1% level using a greater number of runs. Finally, given the minor difference in mean and variance, there is no evidence to suggest that the complete-static approach is more effective than the incremental method.

3.6 Discussion of Results

The results in Sections 3.4 and 3.5 provide strong evidence that our subsumption approach for creating robust actuators for our environment is effective. We now take this opportunity to discuss the significance of the results published, the methodology applied, the potential benefits and drawbacks incurred from the approach. In chapter 6, we return to these remarks and discuss the impact of this work in conjunction with the agent framework introduced in Chapter 5 and reflect on the thesis as a whole. For now however, we focus specifically on the research reported in this chapter.

The resulting hierarchical framework provides a simple yet effective approach to allow for new functionality to be introduced into a reactive controller. Hence we have a means to adapt to new features in an environment without compromising any pre-trained functionality of our controllers. It is important at this juncture to reflect on the similarities and differences to the layered evolution approach presented by Togelius. The most notable difference is the use of a sole fitness function in order to train multiple layers of the hierarchy, which is perhaps the strongest advantage that our approach provides. This is a truly interesting learning mechanism, given that the learning process can find good solutions using a heuristic which, in the case of layer controllers, does not provide any useful information relative to the controller itself. In section 3.4 we presented findings that indicate the controller will learn how to circumvent the added complexity in the environment and will continue to complete the original assigned task, despite circumstances where the agent would have failed had it not been for the controllers adapting to change. In order to train any additional behaviour, we

only require a suitable fitness function for the base control, and properly designed layer controllers to characterise our desired functionality.

In terms of controller construction, the reader may have observed already that none of these fitness functions are particularly complex. The fitness functions hardly constrain the controller requirements and in-turn the search process. There are a few restructions on how the agent should perform, as is indicated by some of the interesting behaviours we generated, but there are certainly high expectations. This is one of the benefits of EAs, since we can take a 'hands-off' approach to the learning process; by providing a simple learning heuristic tied to a purely evaluation-based iterative improvement paradigm.

However, it is important that controller design is given appropriate care and attention, since sensor and sub-controllers must complement the desired functionality in a way to make our goals achievable. Each of the individual neural networks is relatively small in size when compared to contemporary research, where we often require hundreds of synapses to achieve results. As we have seen throughout this chapter, the average controller carries less than 30 synapses. This, tied to the results from our learning process indicate two important findings; the controllers are relatively easy to learn and they can, in most instances, be trained fairly quickly.

The results of the most successful comparative measure - the complete static neural network - provide an interesting reflection on the 3-layer experiment due to the time taken and the overall quality of results. We observe that 150,000 evaluations are provided for this approach and it fails to yield as effective results, while the subsumption trains three controllers with potentially wasted evaluations, as can be seen in the learning trends of some instances, where the agent only requires 20% of the evaluation time to reach a near optimal fitness plateau.

Ultimately the greatest opportunity this approach presents is the ability to create intelligent behaviour through a 'plug 'n' play' approach, i.e. we can take resulting chromosomes from our experiments and put them together into different configurations. Furthermore, we assume at this juncture we can safely use controllers that were trained in one experiment with another problem domain. Whilst the data is dependent on the experiment they were trained in, we are confident that the ANNs have generalised the control policies successfully. In future, further study may be required, but at this point in the research we felt it safe to continue and awaited the results from inserting these controllers within our architecture discussed in the subsequent chapter.

Upon completion we reflected on the overall approach and ultimately how much it diversifies from the original methodology by Togelius. After consideration, we felt that each is an individual entity that can be explored for different uses. The work conducted in Togelius [2004] provides a thorough empirical investigation into utilising the subsumption concept through machine learning. In our work we have taken the original concept and sought to provide evidence that we can streamline the approach while ensuring applicability in different domains. The original concept provides a more expressive formalism, where more expert knowledge can be applied to refine the resulting controllers. The level of detail that can be given to each layer can then be determined by the designer, where we can see specifically crafted behaviours emerging as a result of Togelius's work. While this is certainly a strong piece of work and an applicable solution to the problem, we argue that it is not always convenient or viable in many situations and this is where our approach finds its audience. Our methodology removes the necessity for extra fitness functions for each controller, relying on EA methods to bypass the necessity of expert knowledge. This freedom allows for robust agents that are constructed modularly. The designer can dictate the facets of an agents behaviour by the selection of subcontrollers instead of through a complex fitness formula. As a result, our approach provides an alternative and we reflect on this by including Table 3.16: an amended version of the table found in Togelius [2004]. Our approach can be considered as a modular approach reliant upon functional decomposition of desired behaviour. However despite our sole fitness function, the problem continues to evolve akin to methods in incremental evolution.

This is not say our approach does not suffer from drawbacks. One such drawback is applying neural networks in such a fashion. We are not stating at this point that ANNs are a poor choice for this approach, but we wish to highlight that significant time must be taken to ensure each controller is appropriately designed. In the case of the layer controllers, we must ensure that this does not interfere with base performance when not required, by only providing local, temporary input. Meanwhile, the base controller must always provide input, either from sensors or bias nodes, to ensure consistent behaviour. Ultimately, poor design will lead to conflicts between controllers and ineffective results, hence the series of tweaks and modifications we have described throughout this chapter.

Whilst not necessarily a drawback, one issue we wish to raise at this juncture is the necessity to understand how the desired goal behaviour can be functionally decomposed into individual ANNs. The user must decompose their goal behaviour

Table 3.16: An amended version of the table found as Figure 1 in Togelius [2004], in which the author breaks down the 4 different approaches to evolutionary robotics, with our approach providing a new 5th approach.

	One Layer	Many Layers	Many Functionally Different Layers
One Fitness Function	Monolithic Evolution	Modularised Evolution	Our Approach
Many Fitness Functions	Incremental Evolution	N/A	Layered Evolution

into individual layers according to their own expert knowledge of both the environment and their requirements. Furthermore, they must then develop agent sensors based on this understanding.

Ultimately, there is a limit to the capabilities of applied neural networks for agents either in simulated or real-world situations, and to an extent this is one of the main arguments of this thesis. The work conducted in this phase of the research was intent on exploring a different means to enhance the expressiveness and capabilities of multi-layer perceptrons for execution in low-level problems. We cannot expect highly complex behaviours to emerge from this approach, but we would expect robust and effective control to be trained that ensure a high confidence in their execution. Once we have these solutions, they can then be exploited in our agent framework capable of reasoning beyond the scope of EA/ANN approaches.

3.7 Summary

To conclude we return to our goals highlighted at the opening of this chapter, with the intent of discussing how our goals and concerns have been addressed.

The goals of this chapter were stated as follows:

- *Devise a suitable implementation of the subsumption methodology that would enhance the control of our neural networks without compromising trained behaviours.*

  Our controller design (section 3.3.1), provides a simple yet surprisingly effective means to build control in a hierarchical fashion. The controllers are built atop one another within a subsumption hierarchy, allowing us to create more expressive control without compromising individual behaviours.

- *Create a series of controllers that not only test our approach but can effectively achieve what our planning domain requires.*

  The individual base and layer controllers shown in section 3.4.1 provide a range of different behaviours: from navigation and combat to obstacle avoidance and evasion. While these do not cover the complete range of actions within the Nakatomi domain description, they provide a range of control for many of the more challenging actions in the domain model. Furthermore, the additional layer control helps to remove much of the low-level uncertainty that the environment creates, allowing the high-level decision process to ignore it, assuming that our controller should succeed in execution.

- *Devise an effective learning process to train the controllers within our subsumption hierarchy as quickly as possible.*

  Our learning methodology in section 3.3.2, inspired by the *layered evolution* approach by Togelius provides a simple yet effective training process. This methodology uses a sole fitness function to train a series of different controllers that address unique aspects of the desired functionality. Results of training 2-layer and 3-layer controllers (section 3.4) show that this is a fast and reasonably robust approach for training controllers either through a (1+1) evolutionary strategy or a more traditional EA based approach.

- *Compare against other approaches to ensure our approach is worth incorporating into future work.*

  In section 3.5 we compare against more traditional monolithic evolutionary approaches. The the two comparative measures provided different results, with one succeeding in the task on average. Statistically however, we find a significant difference in our performance to these measures.

3.7.1 Further Work

While we have carried a significant body of work in this chapter, there are still ideas we wish to address as a matter of interest. To date we have conducted research into specific ideas, while others have been left for future work. Here we briefly highlight work conducted outside of the scope of the thesis general direction as well as potential future work we may wish to carry out.

Expanding the Control-Set We came to the conclusion that while this work was valid and results proved effective, the domain was too small for us to explore a wide range of behaviour. As such it would be interesting to explore much more diverse and challenging environments that require many more controllers than EvoTanks. This would allow us to ascertain how many layers we can combine in one hierarchy before control becomes too demanding.

Learning the Hierarchy Automatically Having a larger set of subcontrollers, one challenge that would arise is deciding in what order they be placed in the subsumption hierarchy. At present they are added based on our own intuition and, given the small set of subcontrollers, this was not really an issue. Adapting the learning algorithm to explore which are the best combinations of controllers would prove an interesting future venture.

Remove the Functional Decomposition Another idea we have considered is again based on modifying the training algorithm. Instead of being given the pre-determined layers and the defined order, could we take the modular evolution approach and create layers that act effectively as a whole but individually do not necessarily carry a functional behaviour? While this would remove the 'plug'n'play' functionality, it may provide assistance in dealing with larger and more complex problems.

Explore Different Learning Methods One idea we have considered is whether performance would differ if applied to a different learning algorithm. Perhaps the most obvious choice would be to apply the TD(λ) approach as discussed in Chapter 2.1. The approach made famous in Tesauro [1994] would change our learning algorithm to rely on one sole back-propagating ANN that would use the TD algorithm to explore the environment and update the network weights as it progressed, continually fine-tuning the network until we have reached a sufficient level of performance. To date, we have begun to explore how to apply TD learning to Algorithm 3, however work was still in its early stages at the time of submission of this thesis.

Exploring the Subsumption Further...

Once we had completed this work, several questions were left regarding the best means for utilising the subsumption concept. This led to two direct questions we wished to explore:

- Would iterative re-training of the layers lead to more successful results?
- Can we combine base controllers into the subsumption to create behaviours that rely on more than one behaviour, thus creating a behaviour with more than one type of functionality?

Work exploring these questions was conducted by Fraser Milne under our supervision. The concluding results were later published in Thompson et al. [2009], indicating:

Iterative re-training of layers can improve performance: By re-training lower layers while still having layers above causes the lower layers to change their behaviour with respect to the overall hierarchy. In a visit-waypoint/detect-obstacles hierarchy, re-training the bottom layer improves the overall performance. We surmise that the lower-layer is no longer simply navigating to the point, instead it is now navigating to a point with the assumption that an unknown control (i.e. the higher layers) can deal with the obstacles. Statistics often indicated that results would improve or become more robust in time.

Merging base controllers is feasible: Tests in combining visit-waypoint and destroy-target layers into one hierarchy indicates that we can combine

base subcontrollers. Given the constant activity of these subcontrollers we hypothesised that the higher layer would simply maintain control throughout. However results in Thompson et al. [2009] disprove this.

3.7.2 Closing Remarks

In this chapter, we have chronicled our efforts to build robust reactive control for local scope problems. The results of our training experiments in Section 3.4 indicate that our streamlined version of the layered evolution approach yields robust and adaptable behaviours that enhance agent control with relative ease. We are confident that these reactive controllers will be sufficient for their intended application.

In the next chapter, we introduce a new problem domain that will require not only the controllers described in this chapter, but also the use of the JavaFF planner. We introduce the planning model that will be used to solve problems in this domain and explore how these correspond to our reactive controller.

Chapter 4

Creating the Problem Domain

> If knowledge can create problems, it is not through ignorance that we can solve them.
>
> Isaac Asimov

4.1 Introduction

In this chapter we present our problem domain entitled *BruceWorld*. The simulated world represented by BruceWorld is designed for an agent requiring deliberative reasoning coupled with reactive control in order to succeed, as it should be, given that we designed it with our research interests in mind. That said, it is still a valid game similar in gameplay to traditional video games. After a thorough introduction to BruceWorld and the features of the environment in Section 4.2, we progress to our modelling process. In Section 4.3 we introduce our abstract model of the BruceWorld game dubbed *Nakatomi*. Nakatomi is a PDDL representation that models the features and interactions of the environment for our planning approach. Furthermore, whilst the actions an agent may select from Nakatomi are modelled succinctly, there are still issues pertaining to execution and they will require the effective and robust reactive controllers from Chapter 3 to achieve them. We highlight how these controllers we have created correspond with the actions with those in the planning model.

4.1.1 Goals

In this chapter we shall:

- Introduce BruceWorld, a partially observable, deterministic, sequential, dynamic and continuous game with multiple agents. We explore the features of interactivity within the environment, the adversarial elements of the world and the range of goals that can be assigned to a playing agent.
- Present the Nakatomi domain, the PDDL representation of the BruceWorld game, describing to the reader how elements of the BruceWorld game are modelled or otherwise.
- Discuss the benefits and disadvantages of this modelling approach, and how our design decisions have an impact on the remainder of the research conducted throughout this thesis.

4.2 The BruceWorld Game Environment

> Yippee-ki-yay!
>
> John McClane, Die Hard

BruceWorld is a game environment designed specifically to provide an interesting problem domain to test and evaluate reactive controllers, such as trained ANNs. However, the problem scope is such that a deliberative reasoning process such as planning is a necessity. BruceWorld was developed using the Java programming language to ensure an ease of development and portability, courtesy of the object-oriented approach of the language. The setting of BruceWorld is confined to the interior of an office building where the agent may be assigned goals that require navigation through a series of connected rooms to aid hostages found within the environment[1]. A simple problem example can be seen in Figure 4.1.

4.2.1 Problem Definition

At this juncture it is important that the reader has a clear understanding of the problem, the entities that act within it, the actions and interactions that can take place and important features of the gameplay. We shall take time to describe in finer detail each aspect of the BruceWorld game.

The Environment: As previously noted, the environment is a simple representation of an office building. Each given problem consists of one or more *rooms* filled with office clutter that act as obstacles. Pairs of rooms can be connected in three ways; either via a small *corridor*, a *doorway* or an *air vent*.

- A corridor is the simplest and easiest way to move between rooms. Providing the corridor is not *blocked*, any agent can move through it with relative ease.
- A doorway acts similarly to a corridor but a door can potentially block the path of an agent. If a door is open, then the agent can act as normal; however, if a door is closed then we assume that it is locked. In order to open a locked door, an agent must find the corresponding *switch*. Each switch is pressure-sensitive and requires an agent to stand on

[1] The idea is inspired by the movie *Die Hard*, hence the name is derived from the lead actor Bruce Willis.

top of it. Whilst pressed, the door will remain open. However, if it is released the door will then close again. An agent must therefore remain atop the switch for the doorway to be accessible.

- An airvent is another means to move between rooms, requiring the agent to crawl through it to reach another location.

Characters: There are three types of character that can exist in any problem, notably *Bruce*, *hostages* and *terrorists*.

- **Bruce** is the protaganist of our game and is the only agent that any player (be it human or AI) can control. Bruce can move between rooms in any manner he decides and can carry a *First-Aid Kit* on his person at any given time should he find one in the world. He also wields a gun that can be used to remove any opposition.
- **Hostages** are more docile than Bruce and can only move around via corridors and doorways and can also step on switches for doors. However, they will not do these of their own volition, requiring Bruce to delegate such actions to them. While Bruce can dictate the actions of a hostage, their communication is via a one-way radio and no messages can be sent back. A further constraint is that the limit of their ability to act is based on their emotional state. A hostage can be found in one of five states:

  Calm: In the calm state, a hostage will act as normal.

  Uneasy: An uneasy hostage will not carry out any action, unless they have been trapped in a locked room and the door opens. At this point the agent will run through the doorway and then remain frozen. However, Bruce can slap a hostage which subsequently calms them down.

  Delirious: A delirious hostage acts the same as an uneasy hostage, however Bruce cannot revert them to a calm state. Another option is to knock them out, rendering them unconscious.

  Unconscious: An unconscious hostage is (unsurprisingly) incapable of any action and cannot be revived. However Bruce can carry unconscious hostages one at a time.

Injured: If a hostage is injured, Bruce must heal them prior to their moving. This can be done using a First-Aid Kit which will return them to a state of calm.

- **Terrorists** are aggressive agents that remain within a given room. They will often move around the room on a simple patrol and continue to do so until they spot Bruce. At this point they will attack aggressively until either they or Bruce are defeated. Note that they will not attack Hostages.

Threats: Not only do Terrorists represent a potential threat to Bruce, there may also be *Bombs* littered throughout the environment that can detonate at any moment. If Bruce or any hostage is caught within the vicinity of a bombs blast radius, it will kill them instantly. The potential threat of a bomb is determined by the remaining length of fuse and the blast yield it generates. This information can be obtained by Bruce upon seeing a bomb in a room, allowing him to defuse a bomb provided it has not detonated.

Uncertainty: When a problem map is presented to Bruce, he relies entirely on the description provided. However, while the physical layout of the world will not differ, aspects may be hidden from his view until he enters specific rooms. For example, objects may not be in the location they were thought to be, hostages may be in different mental states and additional terrorists or bombs may be in rooms that we were not initially aware of. The 'real' state of a room and the characters in it only become apparent when Bruce enters said room.

In a given BruceWorld map, Bruce can be assigned a variety of goals, ranging from basic navigation through the environment to acquiring first aid kits or rescuing hostages. Furthermore, goals can be assigned that dictate the final position or state of a hostage and may require a variety of actions to be conducted in order to achieve them.

Next we reflect on the environment descriptors from Russell and Norvig [1995] highlighted in Chapter 1, providing an explanation behind our choices.

BruceWorld is *Partially Observable*: As stated in the problem definition, Bruce does not have a complete understanding of the world state when the game begins. Terrorist and bombs are hidden in future locations, while the

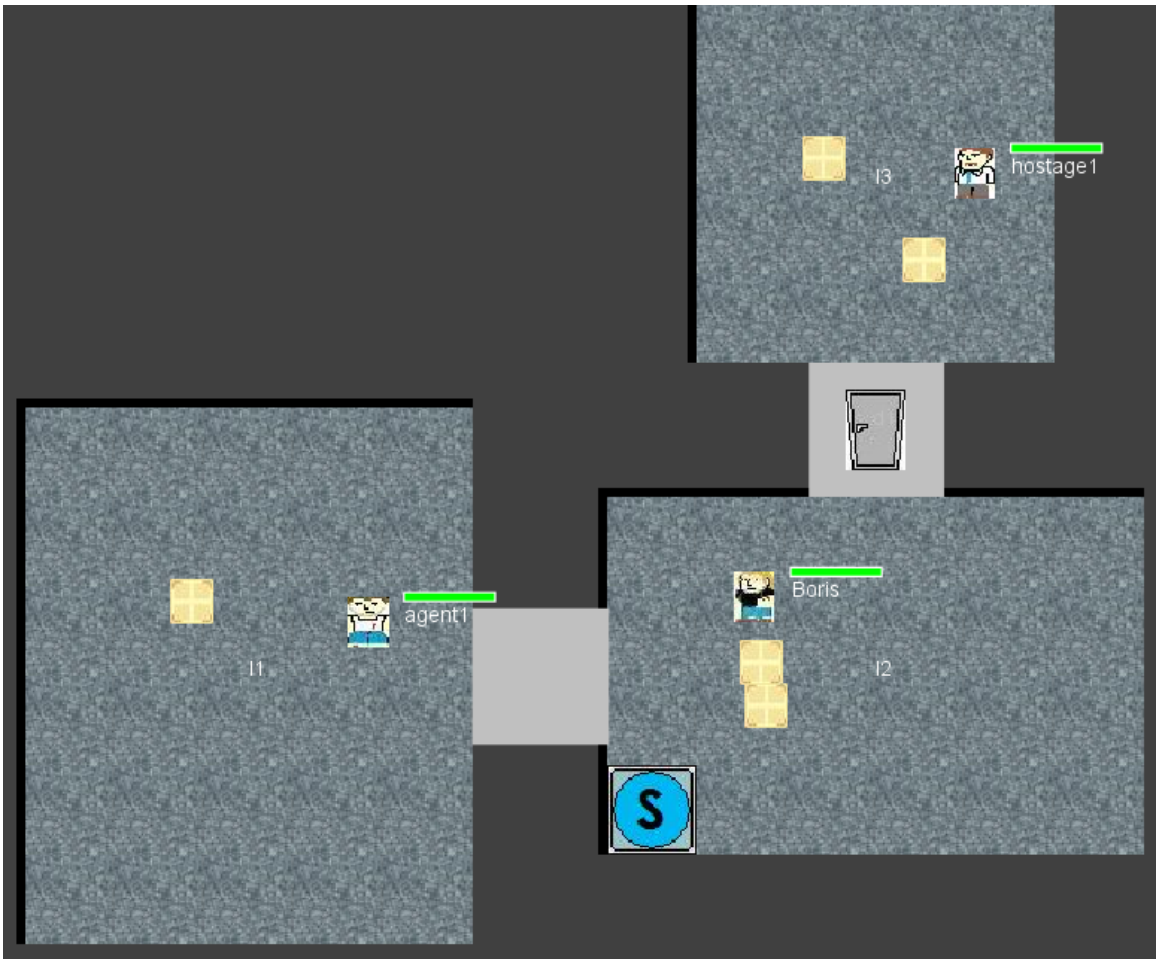


Figure 4.1: A screenshot from the BruceWorld game. The game in progress challenges *agent1* (Bruce) to rescue a hostage (*hostage1*) from behind a locked door. However an enemy terrorist named *Boris* awaits Bruce in the next room, and obstacles/clutter exist in the rooms (though very little in this example) that the agent must navigate around.

assumed state of hostages could be called into question. As Bruce explores more of the environment, many of these concerns are resolved as he can now see the situation in that particular location. Interestingly, once Bruce has visited all locations in a given map, the problem becomes fully observable.

BruceWorld is *Deterministic*: Despite the partial observability, any action made by an agent will achieve the desired state. Unless it is stated that a particular path is blocked or an agent is unfit to travel, then any movement throughout the map will succeed. Furthermore, any interaction with other agents or artefacts will succeed as determined. Given the nature of the game, it would not be difficult to change many actions to become non-deterministic. A simple example could be that slapping a hostage may have no effect, or result in an unforeseen state.

BruceWorld is *Sequential*: Bruce, or any other agent, must consider its actions with respect to the intended future goal and those that have preceeded it. For example, if Bruce is navigating through a series of locations to a specific room, he must remember where he is headed, and where he has came from, in order to progress to the goal.

BruceWorld is *Dynamic*: As any agent executes an action, other events occur throughout the environment, such as agents moving around locations to the dwindling fuses on bombs. This is readily apparent when Bruce fights terrorists, as each agent must consider its actions quickly and frequently to succeed.

BruceWorld is *Continuous*: Given the dynamic changes that occur in the environment and the effect time has on bombs, we consider the game continuous. This presents further challenges for our intended approach, given that classical planning is typically applied in static and discrete problems.

BruceWorld is a *Multi-Agent* problem: As we can see in the problem definition, there are three types of agent in the game: Bruce, hostages and terrorists. Bruce is reponsible not only for his own survival, but that of the hostages and may need to suggest actions for them to carry out. Furthermore, Bruce must also consider the actions of terrorists and how they may impede his progess.

Table 4.1: A summary of all types of entity that exist within the BruceWorld game.

Locations	Connectors	Agents	artefacts
Room	Corridor	Bruce	First Aid Kit
	Doorway	Terrorist	Switch
	Air-Vent	Hostage	Crates
			Bombs

A summary of all types of entity that exist within the BruceWorld game can be found in Table 4.1 for reference.

4.2.2 Game Mechanics

Each agent is capable of basic movement throughout the environment; forwards, backwards, turn left or right. This movement is always relative to the agent and based on their current heading. Agents are capable of activating one or more of these actions at each discrete time-step of the game. Movement is fixed to a predefined distance of 2 pixels for Bruce and 1.75 for terrorists and hostages, with turning set at 4 degrees at each update of the execution cycle. Once any navigation action is committed, there is a minimum delay of 1 cycle before that same action can be carried out again. Each agent, unless predetermined as injured, starts with the maximum 100 health points. Hostages and terrorists will suffer a loss of 25 points for each collision with the environment or if attacked by a weapon. Meanwhile, Bruce is capable of taking up to six hits before death, meaning a loss of 17 points in either case. Bruce and Terrorists can damage other agents through attack actions by either using bare fists or weaponry. Bruce carries a gun that may be fired at a distance but only in the direction he is facing, whilst Terrorists can carry guns or knives that can only be used at close range. Guns cause 25 points of damage, while knives deduct 10 and 5 points for bare-fisted attacks. There is a 50 cycle delay before any weapon can be used again. A given bomb will carry a fuse with a minimum length of 5 and a maximum of 10. The fuse of a bomb will decay by one unit every 400 in-game cycles.

4.2.3 Challenge

Now that we have explained the game, a pertinent question is whether it is challenging to play. The game relies on two aspects; moving the player around

the environment and eliminating opponents, and problem solving. In the latter case, we have to consider how certain issues can be resolved, such as how to enter particular rooms given they are locked and whether we can reach bombs prior to detonation.

For a human player, the game proves challenging in large problem maps. In this case the player needs to devise a plan of action and quickly execute it for fear of bombs killing Bruce or any hostages. There is also the challenge of eliminating enemy terrorists while avoiding the enemy fire. Despite this, any seasoned video game player should be able to solve these problems comfortably once they understand the control-scheme and mechanics of BruceWorld.

For a computer-based player, the challenge is significantly harder. While we are dealing with achieving assigned goals, we must also consider other factors, such as active bombs and enemy agents. As a result, any software agent must somehow factor this into the decision making. This reiterates points made in Section 2.1.2, where we discussed how the Pac-Man games introduced new dynamics that prohibit typical EANN-driven reactive control from solving problems. This problem moves one step further than Pac-Man, by forcing the agent to consider actions from a different perspective. In Pac-Man we are dealing with one enclosed maze, but in BruceWorld we may need to consider our navigation from a more abstract level. If moving to a particular location, which may be n locations away from us, we must consider how to model that space so Bruce can reach it safely. While Pac-Man could only 'chomp' pills or ghosts, Bruce can move, shoot, grab items, step on or off switches, slap, punch and carry hostages. This larger action set has a significant impact on decision making and problem modelling. Finally, given the range of actions that can be executed and the possible states that can emerge, problem maps can provide the player with a range of different goals. While a human can adapt to these changes, it will have significant impact on any AI agent. Perhaps most importantly, it will prohibit any one reactive controller from being able to perform competently, given that it needs to be able to solve any potential set of goals that may arise, which is simply beyond the scope of reactive control.

For an automated player to be able to solve a variety of different BruceWorld maps, it needs the ability to deliberate; to consider how to achieve each goal presented to it and devise a series of actions that will achieve them. It needs a robust means of executing these actions, since they will operate in any situation that is deemed necessary, i.e. they are *generalised*. Finally, it must also consider outside factors that may potentially impede progress, such as terrorists and bombs.

We have addressed the issues of robust execution previously in Chapter 3 and will later explore how they can be managed deliberatively in Chapter 5.

4.2.4 Existing Problem Domains

Now that we have established our problem domain, the reader may question why we have chosen to create a new domain rather than use of the existing domains in the computational intelligence and games field.

Previously in Chapter 2, we discussed a range of research being conducted in games domains. These problem domains included X-Pilot, Ms. Pacman, Super Mario AI, the TORCS racing simulator and Unreal Tournament. There are a large number of problem domains that are available to researchers for application. Outside of the examples named above, there are also a range of strategy games that have a competitive track, including DEFCON, the open source strategy game ORTS and Starcraft.

While these are established competition tracks, at the time of our research we were not confident in our ability to build an architecture that could compete with established competitors. As a result, we decided to create a new domain that carried characteristics that reflected our research goals. As such, we have taken the dynamic, continuous elements of games such as X-Pilot and Mario and created a reactive control problem. Meanwhile, the game also requires numerous actions to be planned ahead and subsequently executed. This relies on the same strategic thinking that has been employed in the Unreal Tournament research chronicled in Chapter 2.

It is our intent that, provided the results in Chapter 5 proved satisfactory, that we would be able to transfer the resulting agent architecture to other domains. This is discussed in Chapter 6 as we reflect on the research conducted across Chapters 3 and 5.

4.3 The Nakatomi Domain Model

> Models are to be used, not believed.
>
> Henry Theil

Once the BruceWorld problem was defined, we required a model for the planner to understand this complex environment. It was important to ensure that whilst this model was sufficiently expressive, it retained a certain amount of abstraction. While we could model the game with a large amount of granularity, it would have a negative impact on JavaFF's performance. As such, it was important we followed several rules:

1. The model requires to be adequately expressive to allow us to represent any potential BruceWorld game map.
2. The model must be sufficiently abstract to ensure the planner's search process is still tractable.
3. The model must conform to the subset of PDDL used by JavaFF.

At this juncture, we relied on the assumption that if an abstract and expressive model was provided we could resolve any problem presented. Moreover, execution of any action modelled in the domain should be applicable using some form of reactive or scripted control. This assumption plays a key part in our layered architecture explored in Chapter 5.

Having taken these points into consideration, we created the *Nakatomi* domain; a PDDL representation of BruceWorld[1]. This approach was taken in order to permit agents to reason about possible plans to resolve problems. However, given the nature of plan models and the abstraction required to ensure ease of deliberation, decisions are made to exclude or reduce detail in certain areas. Throughout this section we highlight the approach taken to model the BruceWorld game and explore how these decisions affect our controller design.

[1]Nakatomi is also taken from the film *Die Hard*, and is the name of the company whose office building the protaganist finds himself trapped in throughout the story.

4.3.1 Domain Description

The environment and all actors within it are represented as a series of defined types. For ease of reading we refer to the definitions previously suggested in Chapter 2 and inspired by representations defined in Ghallab et al. [2004] as well as the PDDL language.

In section 4.2 we provided a definition of the BruceWorld game. One of the defining aspects is the range of different objects within the game: from the three types of agent; Bruce, hostages and terrorists, to locations, switches and first aid kits. As previously discussed in Chapter 2, one expressive component of the PDDL language is the ability to define typed objects. In the following section, we introduce several type hierarchies for modelling components of the BruceWorld game.

Domain Types

Any object from the BruceWorld game we wish to express within the Nakatomi domain definition must adhere to a specific type. For our purposes we have expressed an 'object' supertype, which permits inclusion within one of the following sets:

- location - Corresponding to any room or vent in the environment. Rooms and vents are labelled under their specific subtype but can also be referred to using the location supertype.
- artefact - Any bomb, aid kit or switch is a member of the artefact supertype, with specific types expressed for each individual item.
- person - Bruce and hostages are expressed within the person supertype. This permits specific actions for each type as well as general actions that either type can utilise. Note that terrorist agents are not expressed under this supertype.
- door - As the name suggests, expresses door objects.

This is expressed within the domain file (see Appendix A) as shown below:

```
(define (domain nakatomi)
        (:requirements :typing)
        (:types location door person artefact − object
                                    room vent − location
                                    switch aidkit bomb − artefact
                                    agent hostage − person
        )
        .
        .
        .
```

Note that doors within the BruceWorld game are merely a subtype of the general object type. Since doors do not adhere to any of the characteristics implied by each category (it is neither an individual, an artefact to be interacted with, nor an agent), it was decided to represent it as an individual type. Furthermore, the terrorist agent is completely omitted from the domain definition. This design choice was enforced since terrorists cannot be accurately modelled within PDDL 2.1, since (as previously highlighted in Section 2.3.2) the language focusses all change in the environment specifically at the plan-executing agent. Therefore, no change in the environment triggered by the enemy, even something as simple as moving between locations, could be modelled in the language. While this could potentially be achieved using PDDL+, such an integration would add a large amount of complexity to the model. Furthermore, the few planners that can handle PDDL+ could not generate solutions in the small timeframe we have in mind.

Next, we define the range of attributes that can be assigned to give descriptions of the current state of the world. As previously discussed in Section 2.3.2, it is important that all propositional statements of the environment retain an ease of understanding through the use of natural language. Subsequently, we explore a variety of simple yet effective descriptors that can highlight the gameplay features described in Section 4.2.

Domain Predicates

Objects can be described using a variety of predicate objects as follows. Each predicate carries a typed argument, hence only objects of that specific type are permitted to use it. Based on the definition of the game, we can make certain assumptions as part of the model.

(at ?obj1 − person ?l − location)

Any person can be described as existing within a specific location.

(in ?a − artefact ?r − room)

Any artefact in the problem can be described as lying in a room in the environment.

(on ?obj1 − person ?s − switch)

Any agent type can be described as standing atop a switch object.

(corridor ?x ?y − room)

This identifies that there is a connection between two rooms via a corridor. However this constraint only ensures there is a connection from x to y. In order to make a corridor bi-directional, we must also provide a predicate to indicate that the opposite direction is also valid.

(blocked ?x ?y − room)

This predicate indicates that the path between these two rooms is blocked and cannot be traversed. Again, this is uni-directional.

(doorway ?x ?y − room ?d − door)

As the name would suggest, this indicates that there is a doorway that connects, room x and y. We also specify the actual door object that is in place between these two rooms. This is required, in conjunction with the variety of door predicates that follow. Furthermore, this predicate must operate in the same fashion as the

'corridor' predicate, i.e. one statement only ensures a uni-directional link between rooms. For bi-directional connectivity two statements must be made to indicate the ability to travel from x to y and then vice versa.

(ventilated ?x − room ?y − vent)

Indicates a room is connected to an airvent. Unlike the 'corridor' and 'doorway' predicates, we are corresponding to an actual object, since vents are described as unique objects within the environment. This also prevents the need to ensure bi-directional links exist as we are dealing with two unique object types.

(controls ?s − switch ?d − door)
(open ?d − door)
(closed ?d − door)

The first predicate pairs a particular switch object with a corresponding door, thus providing an easy correlation that can be used in their opening. The remaining predicates simply provide an indication of the current state of a given door.

(calm ?h − hostage)
(uneasy ?h − hostage)
(delirious ?h − hostage)
(injured ?h − hostage)
(unconscious ?h − hostage)

These predicates are a series of hostage descriptors that represent the range of different states these agents can exist in. These predicates are mutually exclusive in practice, i.e. only one of these descriptors holds true in any given state of a hostage agent. The series of actions for interacting with hostages (shown later in Section 4.3.1) give an indication of how a hostage can move from one state to another in accordance with the game definition.

(free ?a − agent)
(carrying ?p − agent ?h − hostage)
(holding ?p − agent ?k − aidkit)

As part of the BruceWorld game, we require the Bruce agent to be able to carry items around the world, either 'aidkit' artefacts or hostage agents. The 'free' predicate ensures that the agent can only carry one hostage (as shown through its application in Section 4.3.1).

(armed ?b − bomb)
(disarmed ?b −bomb)

Finally, we have the predicates that indicate the state of a bomb object.

These predicates provide a suitable description of the game world in a classical planning representation. There are features of the environment previously described in Section 4.2 that are not included in this representation (a subject that we continue to raise throughout this chapter, as well as the potential benefits it can provide us).

Next, we present the range of actions that can be applied within the domain and the key to expressing the nature of the game world. The basis of these actions are modelled again on our description of the game and the logical decisions that a player would make during play.

Domain Actions

Finally, we introduce the series of actions that can be executed within the Nakatomi domain. For each type of action, we provide the complete PDDL listing with precondition and effects. Any similar, yet marginally different actions are listed only with their action signature. The formal definitions of all actions can be found in the complete PDDL domain file found in Appendix A.

Move Actions We begin with the move actions that move agents between rooms, starting with the WALK-AGENT-THROUGH-CORRIDOR action:

```
(:action WALK−AGENT−THROUGH−CORRIDOR
  :parameters
   (?agent − agent
    ?room−from − room
    ?room−to − room)
  :precondition
   (and (at ?agent ?room−from) (corridor ?room−from ?room−to))
  :effect
   (and (not (at ?agent ?room−from)) (at ?agent ?room−to))
)
```

Similar move actions in the model are as follows:

WALK−AGENT−THROUGH−DOORWAY(?person,?room−from,?/
room−to,?door)
WALK−HOSTAGE−THROUGH−CORRIDOR(?hostage,?room−from,?/
room−to)
WALK−HOSTAGE−THROUGH−DOORWAY(?hostage,?room−from,?/
room−to,?door)
WALK−UNEASY−HOSTAGE−THROUGH−DOORWAY(?hostage,?/
room−from,?room−to,?door)
WALK−DELIRIOUS−HOSTAGE−THROUGH−DOORWAY(?hostage,?/
room−from,?room−to,?door)

These move actions are designed to express an agent's ability to navigate through the environment, whether it be through a corridor or a doorway. Note that there are additional actions designed specifically for hostages. This is expressed

not only through the name but also by restricting the type of the first parameter. These are in place to prevent a hostage from moving around certain areas of the environment whilst in an incapacitated state. Note the use of actions for hostages in uneasy and delirious states; while we would have preferred to simply use a negative precondition, as is permitted in PDDL 2.1, sadly the JavaFF parser - much to our frustration - does not recognise them.

Each of these actions relies on an agent navigating from one particular area within the world to another. In each case we need to ensure that the agent moves through the corridor or doorway effectively. Naturally, our visit-waypoint reactive controller is well suited for this problem. This controller, combined with the detect-obstacle controller in a subsumption hierarchy will be employed to execute these actions in Chapter 5.

Switch Actions These actions allow us to move on or off switches that control doors. This is conceptualised in STEP-ON-SWITCH;

```
(:action STEP−ON−SWITCH
  :parameters
   (?agent − agent
    ?switch−room − room
    ?switch − switch
    ?door − door)
  :precondition
   (and (at ?agent ?switch−room) (in ?switch ?switch−room)
   (closed ?door) (controls ?switch ?door))
  :effect
   (and (open ?door) (not(closed ?door))
   (on ?agent ?switch) (not (at ?agent ?switch−room)))
)
```

Similar switch actions are as follows:

STEP−OFF−SWITCH(?agent,?switch,?switch−room,?door)
HOSTAGE−STEP−ON−SWITCH(?hostage,?switch,?switch−room,?door)
HOSTAGE−STEP−OFF−SWITCH(?hostage,?switch,?switch−room,?door)

These actions introduce switch objects and the effect they have on doors in the environment. Stepping on a switch will result in the corresponding door opening,

while the opposite can be said for stepping off the switch. Again we have specific actions for hostages and agents to ensure behaviour is consistent and accurate.

While these actions are responsible for stepping on or off switches, the mechanics of movement required to achieve these operates the same as normal navigation. As a result, we employ the visit-waypoint and detect-obstacle subsumption hierarchy similar to the move actions. The only difference being the destinations visited and how we decide on the point in the world we wish to reach. These issues are discussed in Section 5.3.4.

Vent Actions We have two actions for manoeuvring through air vents, this is shown in the following two PDDL blocks:

```
(:action CRAWL−IN−VENT
  :parameters
  (?agent − agent
   ?room−from − room
   ?vent−to − vent)
  :precondition
   (and (at ?agent ?room−from) (ventilated ?room−from ?vent−to))
  :effect
   (and (not (at ?agent ?room−from)) (at ?agent ?vent−to))
)

(:action CRAWL−OUT−VENT
  :parameters
  (?agent − agent
   ?vent−from − vent
   ?room−to − room)
  :precondition
   (and (at ?agent ?vent−from) (ventilated ?room−to ?vent−from))
  :effect
   (and (not (at ?agent ?vent−from)) (at ?agent ?room−to))
)
```

These actions model an agents' ability to move through airvents into different rooms in the environment. Note there is an explicit constraint that only Bruce (type *agent*) can move through vents, forcing hostages to move through the world

using more conventional means. Once again, these are simple navigation actions which our reactive controllers can facilitate.

Bomb Actions We have only one action associated with bombs, allowing Bruce to defuse it providing it is still active. While this action was deemed necessary in preliminary tests, we later remove this action from the model as discussed in Section 5.4.3.

```
(:action DEFUSE−BOMB
  :parameters
  (?agent − agent
   ?bomb − bomb
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (in ?bomb ?current−room) (armed ?bomb))
  :effect
   (and (disarmed ?bomb) (not (armed ?bomb)))
 )
```

Hostage Actions Next we have a series of actions designed to interact with the hostage.

```
(:action SLAP−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (at ?hostage ?current−room) (uneasy ?hostage))
  :effect
   (and (not (uneasy ?hostage)) (calm ?hostage))
)
```

Other hostage actions include:

KNOCK−OUT−HOSTAGE(?agent,?hostage,?current−room)
PICK−UP−HOSTAGE(?agent,?hostage,?current−room)
PUT−DOWN−HOSTAGE(?agent,?hostage,?current−room)

As we can see, these actions change the state of the hostage in question. Slapping an uneasy hostage will induce calm and allow them to apply delegated actions. Meanwhile, if a hostage is delirious they must be knocked out, rendering them immobile and unconcscious. This in turn necessitates actions allowing for them to be picked up by our Bruce agent and carried to wherever is deemed necessary.

With the exception of PICK-UP-HOSTAGE, we do not require any reactive controllers for these actions. Instead we can facilitate the intended effects by coding the action into a simple script which only requires the agents involved to execute. Meanwhile, the PICK-UP-HOSTAGE action requires the grab-item reactive controller, with the added subsumption layer to deal with obstacles. As previously stated, this is discussed in detail in Section 5.3.4.

Other Actions Our final three actions deal with aidkits and blocked corridors:

```
(:action PICK−UP−FIRST−AID−KIT
  :parameters
  (?agent − agent
   ?aidkit − aidkit
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (in ?aidkit ?current−room))
  :effect
   (and (holding ?agent ?aidkit) (not(in ?aidkit ?current−room)))
)
(:action PATCH−UP−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?aidkit − aidkit
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (holding ?agent ?aidkit)
   (at ?hostage ?current−room) (injured ?hostage))
  :effect
   (and (not(holding ?agent ?aidkit)) (not(injured ?hostage)) (calm ?hostage))
)
(:action CLEAR−RUBBLE
  :parameters
  (?agent1 − agent
   ?agent2 − agent
   ?current−room − room
   ?blocked−room − room)
  :precondition
   (and (at ?agent1 ?current−room) (at ?agent2 ?blocked−room)
   (blocked ?current−room ?blocked−room))
  :effect
   (and (not(blocked ?current−room ?blocked−room))
   (corridor ?current−room ?blocked−room)
   (corridor ?blocked−room ?current−room))
)
```

The first two actions deal with the use of first-aid kits in the BruceWorld game. One is a simple pick-up action which again uses the grab-item and detect-obstacle controller, whilst the other is the application of the aidkit to a specific injured hostage that can be achieved via a simple script. Our final action clears blocked corridors and does not have a corresponding controller, instead it is merely scripted. Note that in this instance there must be two agents available in order to execute it. One agent must be on either side of the blocked corridor, hence the need to specify each room.

4.3.2 Benefits of a Plan-Model Approach

One of the most notable aspects of our PDDL model is the abstraction applied to different aspects in the game. This ranges from abstracting the environment to only consider an agent's general location, to the omission of the Terrorists from the model. This is to ensure simplicity remains paramount whilst trying to solve problems, but also makes large sequences of low-level actions far easier to comprehend given the high-level interpretation attached to specific sequences (e.g. moving between locations). A further benefit of modelling the BruceWorld game within a strict representation such as PDDL is it aids in enforcing specific constraints on behaviour due to the reliance on typed actions and the state-transition system. This allows us to reinforce constraints made by the game's design to maintain an accurate portrayal of the limits imposed, without necessarily relying on them in the game-engine. An anecdote that testifies to this is that during testing of the game and our architecture we came to the realisation that no code had been applied to the engine to ensure that the agent could not carry more than one hostage at a time - a simple mistake! However, the action of carrying a hostage between rooms negates the *(free agent1)* proposition in the plan-model, hence our agent could in fact only carry one hostage at a time as we explicitly stated that was a constraint of the behaviour. Therefore, the addition of the constraint in our model reinforced the original rules of the world, and provided a small anecdote showing the benefits it can provide!

4.4 The Research Challenges Ahead...

> A lot of people say, 'Well, I like a challenge.' I don't like challenges. Life is tough enough without any challenges.
>
> Jackie Gleason

4.4.1 How Do We Solve BruceWorld?

BruceWorld as a problem domain suits our needs given that it provides an environment which neither of our research interests could solve with ease. In fact it succeeds in existing as both a planning problem and a challenging reactive control task. While time could certainly be spent exploring how we could devise reactive control to solve this problem, we suspect that the solution would not be as flexible as our own findings. Reactive control for this problem would need to extend across multiple 'actions' of the PDDL model. This could perhaps be achieved by developing a policy for control using TD(λ). However, can we rely on this policy for different configurations of a particular problem? More interestingly, what happens if we change the problem slightly? Or if the previously mentioned uncertainty is in effect, and what we initially considered to be the problem is no longer the case? Do we need to develop an entirely new policy or controller now in order to solve it? In fact, how can we recognise that a change in action is necessary during runtime?

Conversely, approaching this problem from a planning perspective utilising our defined model would be feasible using scripts for all actions. However, the time taken in creating these scripts may be wasteful, furthermore can they be relied upon in all possible circumstances that arise in the game? Assuming no complex scripts other than basic movement were permitted, what level of granularity would be required to solve even the most basic of problems?

These are questions that highlight the real difficulty this somewhat simple game presents for both methodologies. However, applying planning with reactive control would circumvent many of these problems by relying on the benefits of each approach. The tasks that can be assigned to the player often require a series of different actions to be executed in sequence. Given this conjunction of individual actions it would seem necessary that a deliberative decision process is applied.

However, there are issues in executing each of these actions, dealing with the packing crates that litter the environment, to the terrorists and bombs that can interfere with execution. This provides a fantastic opportunity to apply low-level control to make intelligent generalised actuators that perform irrespective of the specific plan-action being executed. Yet, there are still other issues that need to be addressed caused by the use of the plan model and the 'gap' of knowledge between the low-level controllers and the plan model.

4.4.2 Introducing Reactive Actions

The use of a virtual environment for our problem allows us to make a series of assumptions of the world in the game engine, such as the physics of the environment and the removal of unnecessary factors. Meanwhile, further assumptions are built within the plan-model. What is important is that these assumptions are correctly understood to a point where it will not impede our intended agent architecture. The assumptions made in the Nakatomi model, while necessary, will lead to problems once we attempt to execute actions within the game world.

For example, how do we deal with terrorists from this planning perspective? As the reader will have noted, no PDDL types or actions are associated with terrorists and the destroy-target controller. This results in the terrorist not existing in the Nakatomi model: an assumption that makes the planning process much easier. However, if we have no means of modelling the terrorist at present, how can this controller be selected for execution when needed? This also applies to the the defusal of bombs. How do we decide that a bomb needs defused at a given point in time?

Another consideration is whether additional actions need be made in the case of actions where the PDDL pre-conditions have been satisfied but the action cannot be executed. An example of this is the SLAP-HOSTAGE action, where the PDDL action dictates that the hostage and agent must be in the same room as one another. While this is perfectly valid, the level of granularity applied does not allow us to consider whether the agent and hostage are physically next to another, which is essential for this action to be permitted.

For each of these cases, we must provide means to address them during the execution of our PDDL plan. As a result we provide means to introduce extra actions for execution, denoted as *reactive actions*, that respond to changes in the environment that the planner does not model and is not factored by the ANN

controllers from the previous chapter. This is achieved courtesy of a rule-based system that denotes whether extra actions are required in specific circumstances. This issue is described in greater detail in Section 5.3.3.

4.4.3 Applying the Reactive Controllers

As indicated during Section 4.3.1, our intent was to associate particular PDDL actions with the reactive controllers found in Chapter 3. Given their performance in testing, they will prove adequate in dealing with any challenges with regard to executing the actions we have prescribed in the plan. It was our intention that these reactive controllers will solve one specific task, with the assumption that the correct combination will result in problem resolution. Given we can provide a series of good-quality, generalised controllers then our planning assumptions of a simple environment can be maintained, since the issue of ensuring effective control can be passed down to the reactive controllers. Although issues still remain in dealing with potential uncertainty or discrepencies in the plan-model, we will address them in detail in Chapter 5.

4.5 Summary

In this chapter, we have introduced the BruceWorld game, a partially observable, deterministic, sequential, dynamic and continuous game with multiple agents. We discussed at length the features of the environment as described in Chapter 1, the adversarial elements that may impede a player's progress and the goals that can be assigned to the player. We followed this with a brief discussion on the challenge it presents to a human player, and more importantly, the increased difficulty for any attempts of AI control.

Next, we introduced the Nakatomi domain, an abstract PDDL model of the BruceWorld game that we intend to apply in our layered architecture. We described in detail the types, predicates and actions that are used to conceptualise BruceWorld in such an abstract representation and how our reactive controllers correspond with sets of PDDL actions.

We concluded with a discussion of the merits, drawbacks and challenges presented by the Nakatomi model. Specifically, we discussed how these design decisions impact on the research chronicled in this thesis.

Next in Chapter 5, we introduce our layered architecture, installing our reactive

controllers from Chapter 3 beneath the JavaFF planner. With the BruceWorld problem domain now fully defined, we subsequently address how these unique approaches resolve the challenges our domain presents to them. As we shall see, there are many issues that arise when merging these approaches together. We explore how our agent architecture is designed to compensate for these issues, and devise a lengthy evaluation process to ensure that the agent is robust and adaptable as possible.

Chapter 5

Creating a Plan-Driven Agent Architecture

> You know what I've noticed? Nobody panics when things go "according to plan." Even if the plan is horrifying!
>
> The Joker, The Dark Knight

5.1 Introduction

In this chapter we provide a thorough account of the design and testing of our plan-driven agent architecture. Previously in Chapter 4, we gave an account of the BruceWorld problem environment and the essential role it played within testing phases. In this chapter we explore the design features necessary for an agent to be able to interact with the BruceWorld environment and solve a variety of problems. To achieve this, we introduced the Realtime Executive for Automated Plans using Evolutionary Robotics (REAPER). This chapter documents our incorporation of the FF planner for deliberation and how we circumvent issues that arise from using a discrete planner within a continuous, dynamic game environment. Furthermore, we explore how our evolved reactive controllers from Chapter 3 are integrated within the architecture and our delegation of plan actions selected from the Nakatomi PDDL domain.

5.1.1 Goals

The goals of this chapter are as follows:

- Introduce the *PDDLWorldBuilder*: a java framework designed to parse PDDL problem instances and translate them into environments for testing. Furthermore, we explore how the BruceWorld parser translates particular features of the Nakatomi domain into actual BruceWorld test instances.
- Introduce the REAPER architecture, proposing a potential solution to the BruceWorld game. We provide in-depth detail of the three main components of our agent; the *Plan Manager*, *Rule Controller* and *Controller Library*.
- Implement a series of tests to assess the effectiveness of the approach and highlight any areas of significance for discussion.

5.2 PDDL World Builder

One of the important aspects of creating a plan-driven architecture is we must ensure that the plan-model is sufficiently accurate. As highlighted in Brooks and Flynn [1989], inaccurate models lead to excessive processing overheads in an attempt to retain accuracy. For the purposes of testing we wanted to ensure the plan-model for any given problem run was 100% accurate (unless we dictated otherwise, a feature we address later in this chapter). Furthermore, we needed also to create the problem instances for testing, since at first we relied on constructing problem files by hand. The solution to our problems came in the form of the *PDDL World Builder*, a simple framework that would parse PDDL problem files for the Nakatomi domain and construct the BruceWorld games based on the initial state. This provided us with two benefits; the automation would remove the need to handcraft problems for the BruceWorld game, as we could build them from the existing PDDL files, which were easier and faster to construct. Secondly, it would ensure consistency between the plan-model and environment at the very beginning of testing, given the problem is directly mapped from the supporting planning files.

To create our problem instances, the system requires only the PDDL problem and domain file. While it then adheres to the provided problem description there is still room for variety in each test. We previously highlighted in Chapter 4 that while the PDDL model provides a suitable abstraction of the problem, it also

strips away many of the physical and geographical attributes. For example, if we look at the Nakatomi problem file shown in Figure 5.1, we have two agents; Bruce and a hostage who needs to be rescued and moved to location *l1*. Given that the model operates at a high level of abstraction, it removes many of the assumptions made about the layout of the world. Hence, in this example, whilst we have two locations connected by a doorway, we have no concept of orientation or physical dimensions. Where is location *l1* with respect to *l2*? It could in the simplest case be north, south, east or west of *l2*. In fact, we are not even aware of the dimensions of each location. Moreover, where exactly are the agents within the locations? Where in *l1* is the switch for the door? In this rather simple example we can highlight several aspects of the physical environment that are left ambiguous as a result of the PDDL model. While we can easily picture this problem in our minds, it does not necessarily represent one specific problem instance. In fact, this PDDL model represents a series of unique problems, that when viewed from this level of abstraction look exactly the same. The ambiguity found in the problem files is exploited by the PDDL World Builder in order to instil variety in the test problems.

While this variety is irrelevant from a planning perspective, it is important from an execution level. When testing our agent, we must ensure that our reactive controllers are capable of executing plans in a variety of situations. Handcrafting specific problems to test our agents would not be sufficient as there may be circumstances our reactive controllers are not tested against. An automated process that generates unique versions of each problem helps to address these concerns, allowing us to create a variety of different physical environments whilst retaining accuracy of the planning models.

As highlighted in Section 4.3, many features of the BruceWorld game are not expressed within the Nakatomi domain description. We need to consider whether there are terrorists or bombs in the environment and whether there are obstacles in each of the rooms. Such features provide further diversification for each problem, allowing us to test our agent further. While adding obstacles to the environment is merely an issue for reactive controllers to consider, threats to our agent differ since we must then change or pause our plan of action in order to deal with them. This would require new actions being inserted into our plan, a challenge we address later in Section 5.3.

A simple pseudocode of the process is shown in Algorithm 5, where we begin by generating each object in the PDDL problem file. Each object definition is a

```
(define (problem FLEXIBILITY−EXAMPLE)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        l1 − room
        l2 − room
        s1 − switch
        d1 − door
        )

        (:init
        (at agent1 l1)
        (at hostage1 l2)
        (in s1 l1)
        (doorway l1 l2 d1)
        (doorway l2 l1 d1)
        (controls s1 d1)
        (closed d1)
        (calm hostage1)
        )

        (:goal (and
        (at agent1 l1)
        (at hostage1 l1)
        ))
)
```

Figure 5.1: A simple PDDL problem modelled from the Nakatomi domain. The abstraction used at the planning level leaves much of the physical description open to conjecture.

tuple $(name, type)$, hence it is quite easy to recognise each individual object in the world and generate the corresponding in-game object. Each in-game object is instantiated assuming it is a basic instance of that type. Hence it represents this object under the assumption no predicates have made any explicit statements regarding its current state. We also define the physical characteristics of the environment as objects are created. Thus when genertng a location we give it an arbitrary size, with each axis within the range of 200 to 350 pixels.

Algorithm 5: A breakdown of the process taken to convert the PDDL problem files into BruceWorld problem instances.

Input: Nakatomi problem file p
Result: BruceWorld problem definition d

```
1  foreach object: p do
2  |  generateObject(object,d)
3  |  assignUncertaintyProbability(d)
4  |  generateThreats(d)
5  end
6  foreach predicate: p do
7  |  object ←findObject(predicate,d)
8  |  initialiseObject(object,predicate,d)
9  end
10 startLocation ← findStartLocation(d)
11 sortGeographicalPositioning(startLocation,d)
```

Next we parse the predicates of the initial state in the problem file. Given that each predicate carries several arguments, we structure them so the first argument is always the object we are describing. This makes the parsing process easier, as can be seen in Figure 5.1, where the *at* predicate is always followed immediately by the agent we are describing. In the case of a corridor or doorway we state the start location first, followed by the desination. In order to retain a similar layout, the door described in any *doorway* predicate is always provided last. To apply these predicates we find the object we have created in the game and then apply the specific predicate to that object.

During the predicate parsing phase, we also consider the physical layout of the problem. When defining ventilation, corridors or doorways between locations we detail references to the connecting corridor, door or air vent java objects in addition to an indication of whether this object is to the north, south, east or west of the location. This not only helps reinforce the connectivity as dictated by the predicates but also prepares each object for placement.

Once all in-game objects have been modified to represent the current state of the game world, the next phase is to sort the geographical positioning of the level. Beginning with the starting position of Bruce, we place the location itself, Bruce and any accompanying agents or artefacts within the space. Once completed, we assemble a list of all adjoining locations. Provided these locations are not already positioned, we site them within the world keeping the previous location in memory as a point of reference. Once this final phase is completed, the BruceWorld problem is complete. Once loaded into the game, our agent can then begin deliberation from the initial state of the problem.

One additional feature in the world builder is the ability to modify the original problem to make it more challenging. Given that the world builder can already generate variants of the same problem, we considered the possibility of adding or modifying constraints. Reflecting on our game definition in Chapter 4, the introduction of Terrorists and bombs can greatly change the flow of execution when trying to solve a problem. Hence we focussed on creating a simple (yet useful) means to add them into existing problems. We refer to this as introducing *threats*, since they provide additional challenges or issues that must be addressed during execution. A second approach was to inject *uncertainty* into the problem. We wanted to have a simple modifier that makes subtle changes to the initial state without compromising the problem. Hence when hostages are created there is a (user defined) probability that their initial state will change.

To add threats to the environment, we rely on a defined probability distributed across all locations in the problem that suggests the odds of a bomb or terrorist (or both) emerging in a given location. During the predicate parsing phase we consider this possibility for each location. If the probability is satisfied then we generate a bomb and/or a terrorist agent. The bomb is created with a random initial state (in terms of blast yield and fuse length); however the fuse length is long enough to ensure that the bomb does not detonate prior to Bruce being able to defuse it (if needed). Moreover, the terrorist is placed at random within the room with a randomly assigned weapon provided.

Modifying the problem was slightly more challenging, given that we were still dealing with a deterministic planning problem. We surmised that Bruce should execute as normal until a situation arose where his plan-model did not reflect the actual state of the game. At this juncture, execution would pause while a new plan of action was generated. Hence we wanted any modifications made to entities to only become apparent when the agent is *in the same room*. That way

the agent is forced to re-plan during execution. However, we were limited by the flexibility in what changes can be made to the problems. We needed to ensure that modifications to the current state did not impede progress in a given problem. Such a case would arise by manipulating switch objects: such as modifying which door the switch controls (if there is more than one (switch,door) pair) or changing the location of the switch. This would necessitate running a validation process to ensure we can still reach/operate the switch and continue as normal. As a result, the only modification we could apply with confidence was to manipulate the state of a hostage, forcing more or less actions from the Bruce agent due to the new state.

5.3 REAPER Controller

> Ted? Don't "Fear the Reaper"!
> [air guitar]
>
> Bill S. Preston Esq, Bill & Ted's Bogus Journey

The product of our research is the REAPER architecture; a hybrid executive that is a preliminary step in deliberative reasoning with reactive control for games. REAPER is designed to allow the integration of a classical planning system with reactive agent controllers. While the use of neural networks dictates the agent's behaviour is primarily reactive, the installed JavaFF planner helps provide means of deliberation. As shown throughout this chapter, the architecture proves itself capable of generating goal-driven, varied yet reactive control.

The REAPER architecture is comprised of three individual components. The three modules are described as follows:

The Plan Manager (PM): Responsible primarily for generating plan-solutions to problems. Once plans are generated, the system is responsible for ensuring consistency between the plan-model and the actual game-world as each action is executed.

The Rule Controller (RC): A rule base whose primary function is to associate reactive controller configurations with specific plan actions. Furthermore, it also runs a second validation process that ensures actions are valid for execution with respect to threats or other geographical issues.

The Controller Library (CL): A controller interface generates the required controllers for a specific action. These controllers range from a collection of pre-trained reactive behaviours to hardcoded scripts.

These modules are designed to exist as de-coupled sections of the architecture. Each has a specified interface that is visible to other parts of the system. It is our intention to allow sufficient flexibility for modifications throughout without impeding overall performance. Next, we explore each of these controllers in detail, highlighting the design of each component and how they assist in generating our plan-driven, reactive agent.

5.3.1 Controller Layout and Execution

The complete layout of the agent architecture is shown in Figure 5.2. Once the executive is queried for action, we begin by accessing the PM, where model consistency is validated and the system either begins planning or continues execution of a pre-processed plan. Once an action is decided upon, the system moves into the RC, where further validation takes place. As indicated later in this chapter, a series of rules is checked to ensure no threatening entities may impede progress. However, the RC's primary function is to select a specific controller for execution. Once decided upon the final phase is initiated by requesting the required actuator from the CL. On retrieval this is passed to the agent for execution.

5.3.2 Plan Manager

The PM houses the JavaFF planning system and is responsible for the deliberative control of the agent. It is responsible for two primary functions - ensuring the plan-model is consistent and, to our knowledge, accurate with respect to the game environment and generate/monitor plans for the execution process.

Model Verification The first of the PM's two responsibilities is paramount: if we do not have an accurate model then we cannot hold any confidence in plans generated by JavaFF. In the provision of a means to verify the current state of the world, we can continue holding an assurance that resulting plans will not only be feasible for execution but also create our desired state in the world.

To begin this verification process we need our own model of the world state and a means to express it in the system. This is a non-issue thanks to the type system used in the JavaFF planner. JavaFF models PDDL propositions from the domain using a *Proposition* class, storing the predicate symbol (descriptor) in addition to a list of *Parameters*, the objects that ground the predicate. For our verification purposes we created a *compare()* method in the Proposition class, providing an easy way to verify whether two Proposition instances represent the same predicate.

Subsequently we need to generate the state from the environment to carry out our model check. Each BruceWorld class representing an in-game object type includes a method *generateStatePredicates()* which will provide all relevant information about that specific object using the JavaFF Proposition type. An example of the code applied to a switch object can be seen in Figure 5.3.

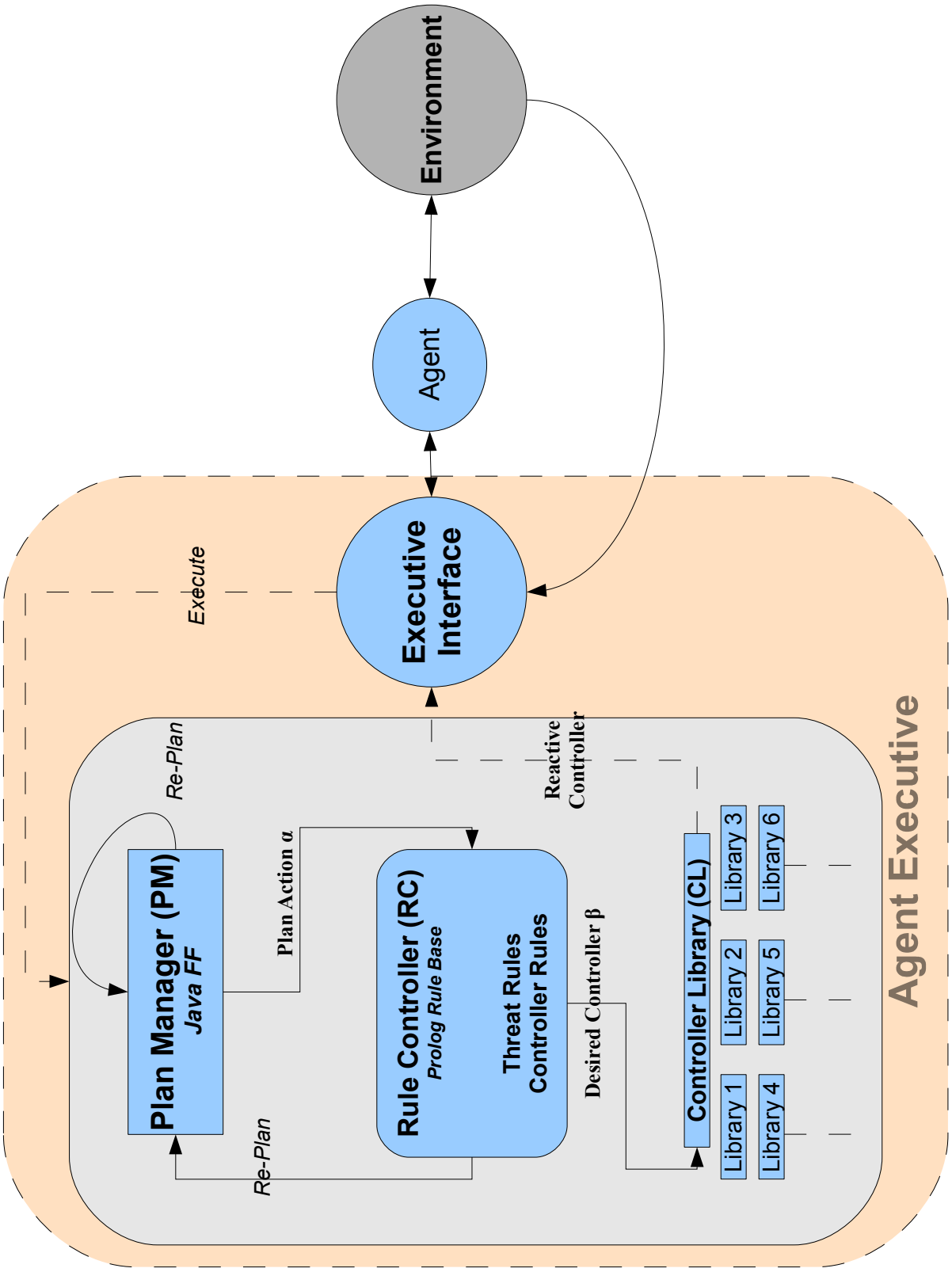


Figure 5.2: A diagram indicating the layout and execution flow within the REAPER framework

```
public List<Proposition> generateStatePredicates(){

        state.clear();
        Proposition in = new Proposition(new PredicateSymbol("in")/
            );
        in.addParameter(new Variable(this.getEntityName()));
        in.addParameter(new Variable(this.getLocation()./
            getEntityName()));
        state.add(in);

        if(BruceWorld.REPLAN){
            Proposition controls = new Proposition(new /
                PredicateSymbol("controls"));
            controls.addParameter(new Variable(this./
                getEntityName()));
            controls.addParameter(new Variable(this./
                connectingDoor.getEntityName()));
            state.add(controls);
        }

        return state;
    }
```

Figure 5.3: An example of the code attached to each in-game entity in the BruceWorld game. This code allows us to represent an object in the PDDL language using the built-in propositions in the JavaFF planner. This example is taken from the switch object for controlling doors. Note the use of two PredicateSymbol objects “in” and “controls”, matching the predicates used for describing switches in the Nakatomi domain file.

The code block in Figure 5.3 shows how we generate the two predicates that represent the switch object. First it provides the 'in' predicate, showing where this switch is and the 'controls' predicate to indicate what door it affects when pushed. As we previously explained, this is achieved by instantiating Proposition objects and then adding the relevant predicate symbols and parameters. Note the conditional block in the code that dictates whether we are in re-plan mode. This is a deliberate design choice dictated by the behaviour of the JavaFF planner. When JavaFF initially reads in the domain and problem files to begin planning, it will identify certain aspects of the problem to be constant. By constant we assume that a particular predicate will remain the same throughout as dictated by its behaviour prescribed by the domain model. Hence, these specific predicates become *grounded*, i.e. they become fixed facts within the problem and are no longer considered open to manipulation during search. In JavaFF, the system does not model grounded predicates within the class that is used to represent individual planning states[1]. This is of great importance during plan monitoring and re-planning, which we will highlight later in this section. With these components in place, model verification became a far more simple process. Using our representation of the current state, we simply compare the Proposition objects stored within the PM to the retrieved collection of Propositions generated by the in-game objects. The Proposition objects stored within the PM belong to our stored model of the environment as part of the plan management process which we will now highlight in detail.

Planning Management When we initialise a given problem instance in the BruceWorld game, we know that the in-game world (or rather our view of it) will directly match the PDDL description in the problem file thanks to the PDDL WorldBuilder software. The next step is to then feed the problem and domain file to JavaFF to create a plan. Once a plan is devised, JavaFF will output the plan in sequence via the standard output, hence it will display on the development environment's console, UNIX terminal, DOS prompt etc. However, we wanted to be able to store the plan in a simple and compact manner within the PM, allowing us to reference it at a later point.

By exploring the JavaFF framework, we became familiar with how the system models and represents states, actions and plans. States are represented in the JavaFF system using the *TemporalMetricState* class, containing all relevant Propo-

[1]This is with the intent of making the planning process easier, since there are less variables to consider.

sitions that model the state predicates. As previously mentioned, this contains all information about the current state with the exception of the grounded predicates. Furthermore, actions are expressed within their own individual class (*TimeStampedAction*), with plans easily represented as a list of these action objects in the correct order. One of the benefits of the TimeStampedAction is that it can be applied to any TemporalMetricState object to generate the successor state according to the restricted state transition system. Hence, we exploit this feature as part of the plan management process. At the beginning of the planning process, the system grounds the problem (since all relevant facts are provided) and from this point we can acquire the initial state. We retrieve this initial state and store it within the plan manager, providing a starting point of reference for the model verification process. Once JavaFF has completed the planning process we retrieve the complete sequence of actions that JavaFF has surmised will attain the goal state. These are then utilised for the actual execution by the executive.

Plan Execution & Re-Planning Having now the initial state and list of plan-actions we can begin actual execution of the plan. The initial state adopted during the planning process is now used as our frame of reference for execution and from hence forward referred to as the current state. Using our model verification process, the Propositions retrieved from the game environment are compared against those found within the current state. Should this prove successful, the next phase is to ensure that the action we wish to apply in this state is applicable. Action applicability is once again a simple check thanks to the built-in functions within JavaFF. After having completed both checks we present the current action as output from the PM to be sent to the remaining systems for verification and execution.

Once a signal has been received to indicate that the current action has been completed, we update our internal model of the environment by applying the TimeStampedAction object of the current action to the current state (TemporalMetricState) object, this process transforms the current state object *into the successor state*. Hence we have applied the state transition function from Definition 2 in our model to create the new current state. We then simply replace the current action with the next action from the queue, which should now be applicable in the current state. We repeat this process until all actions are executed. Once all actions are complete a final check is run to ensure this is in fact the goal state. If this is the case then no further actions are provided from the PM.

However there will of course be circumstances where either the model verification phase fails or the current action is not applicable. To accommodate for this, the system can commit a re-plan. To achieve this, we need to correct any inaccuracies in the state model prior to re-planning. Therefore we query all relevant objects within the game environment to gather all the information required for a new PDDL problem file. This will naturally include terms that may have been grounded during the planning process (thus the conditional flag in Figure 5.3). Initially we sought to insert this information directly into a problem object within the planner, allowing it to immediately run the planning process. However as we began to explore this idea it seemed more viable (read: easier) to simply generate a PDDL problem file that reflected the current state of the game and then load that file in as one would normally. Therefore code was written to generate a PDDL problem file in the local directory that carried all information about the current state. This information is then loaded into the JavaFF planner and planning takes place. Providing a successful plan is found, we then continue execution as normal.

One final aspect is to ensure that the original goal(s) are satisfied. As such, one final feature of the PM is to ensure that all originally assigned goals of the planner are completed prior to declaring that execution has been a success. This is achieved by checking the current state against the grounded goal conditions via a simple function within the JavaFF framework. In the event that not all goals are satisfied, then a re-plan is committed to create a new chain of actions that will satisfy these goals. Otherwise, the system reports that execution has successfully completed.

5.3.3 Rule Controller

The RC is the second phase of deliberation for the executive. Once the PM has completed model verification and any required planning, the next action for execution has been selected. Now we have an action, our next step is to select the corresponding controller - be it scripted or pre-trained - that will facilitate the demands of the PDDL action. Furthermore, it is also designed with the intent of carrying out additional checks against features of the environment that do not exist in the plan-model, notably, hostile elements in the world that cannot be

modelled as part of the planner's model due to its existential point of view[1]. This allows us to introduce extra actions for execution, a point we alluded to at the end of Chapter 4. To achieve this, the system revolves around two rule bases; the *threat rule base* and the *controller rule base*. For each rule base we have a large collection of rules for reference. However, we needed to ensure the rules are easy to model and the lookup process was relatively fast. To facilitate this, we used the Prolog programming language.

Prolog is one of the most prominent and arguably the most popular logic programming language and is used broadly in AI as well as computational linguistics. A logic programming language is a declarative, procedural language that in turn carries a theorem-proving underlay. A user can declare a series of fixed statements and deductive rules that, when used with the theorem-proving system, allows for logical inference to be carried out. While Prolog is capable of these faculties, one particular benefit is it is also very fast. Having previous experience in writing Prolog we felt it would be the best choice to model our relatively simple rules in the RC. The only issue now outstanding was how to integrate Prolog code into our Java-written framework. Fortunately the SWI-Prolog distribution also includes the JPL interface; a bi-directional Java-Prolog framework that allows for Prolog predicate integration in Java as well as Java method calls in Prolog files. Hence we used the JPL interface to allow us to easily code-up our rule base in Prolog and integrate it into the REAPER framework.

Classification One issue that needed to be addressed was how can we translate our real world information into a form that we can then reference from our rule base? If we wish to ascertain which action should be made based on the current state of the environment we need a classification process for resolving the in-game data and labelling it in a manner that represents its value and additionally makes it practical for the Prolog querying process. A simple example found throughout our rules is the notion of distance: certain rules flag different actions based on the approximate distance of our agent to a specific object. However, writing an individual clause for each potential input is costly and wasteful. Thus it is in our best interests to classify this data into particular sets, from whence we can then provide clauses to consider whether an input is within a defined range. But of course how do we define these input ranges? Do we - the designer - decide

[1]Given that a classical planning system observes all change in the world with respect to the executing agent, we cannot express changes of state in other agents.

how relative something is? Or should we use a more structured approach for classification?

We turned to classical set theory to address this problem. Set theory dictates that a value is a member of set provided it is within the constraints of membership - i.e. the value is either a member of the set, or it isn't. Given that we can create different sets with unique membership criteria, it is to be expected that particular values can appear in more than one set. For example, the number 1 is not only a member of the set of positive numbers, but in the sets of odd and prime numbers. Provided we could generate the membership constraints for a collection of data using some informed analysis, we could then rely on these sets to give us information about in-game objects. Returning to our distance example, we could decide whether a particular distance exists in more than one set. If positionally, we are relatively close to an object but still too far away to really do anything with it, then that distance value could be considered a member of both 'close' and 'near' sets, as there will be some overlap between the definition of a close object and a near object.

To achieve classification, we required the constraints of set membership to be defined for the in-game data. We were directed to a simple yet effective tool written by Richard Jensen[1] dubbed *FuzzyGen* (Jensen and Shen [2008]). FuzzyGen generates simple set definitions from a supplied dataset. To accommodate this code, we wrote simple dataset generators that would provide our required sample data; including distances between two objects, blast yields and fuse lengths of bombs and the health of a given agent. Whilst we could have fed information directly from an in-game test, we found it an easier process to write these simple dataset generators, provided we adhered to the constraints of the in-game world that were dictated by the PDDL World Builder. For example, the distance variable would be generated as any distance between the dimensions of the largest possible room (subtracting the bounding boxes of the agents) to the minimum distance between two objects. Once these sets were completed, then the corresponding labels for each set are used as arguments for any clause that relates to in-game information. We provide a breakdown of the resulting sets from Jensen's code in Appendix B.

Once the sets were defined, we implemented a simple membership function within the RC that gathers all relevant information for any rule base query. On

[1]Special thanks to Michelle Galea for her advice and assistance in getting this code running properly.

completion of the data collection, it is checked against our sets and the specific value is replaced one or more corresponding set labels. These label are then used as an argument for the subsequent rule base queries.

Threat Rule Base

The threat rule base is designed to address the hostile elements in the game world. While in BruceWorld we have sought to address hostile entities, this is not a vital component for all problem domains. What is important here is that these rules address issues the planning system cannot. This is due not only to the level of abstraction that is typically applied in planning, but also since deterministic planning does not lend itself to dynamic environments with multiple agents. It is for the latter reason we have explored the creation of a rule base to facilitate their inclusion into our deliberation process.

All threatening entities in the environment must be assigned a *threat level* to indicate whether a specific entity will impede the progress of our agent. There are three threat levels: *low*, *medium* and *high*, with individual clauses applying one of these threat levels dependent upon the circumstance it represents. These are then connected to a series of action rules that dictate which action should be selected based on the threat level.

To clarify, we use bombs as an example; bombs carry numerous attributes that must be considered in assessing their threat level. Naturally, the more damage a bomb can inflict and the time remaining to detonation have a major impact on their threat level. However, our model cannot express the decaying fuse, nor infer the potential damage it represents. Therefore, we require a series of rules that indicate how threatening a particular bomb actually is.

To achieve this, we have a small rule base that dictates a threat level based on the distance of the bomb, the blast yield and remaining fuse length. The block below shows a sample of the clauses and the bomb threat rule found in our rule base.

```
bomb_threat(DIST,YIELD,FUSE,THREAT,ACTION):−
    bomb_threat_level(DIST,YIELD,FUSE,THREAT),
    bomb_threat_action(THREAT,ACTION).


.
.
.
bomb_threat_level(close, high, low, high).
bomb_threat_level(close, high, medium, high).
bomb_threat_level(close, high, high, medium).
.
.
.
bomb_threat_action(high,defuse_bomb).
```

This code block (on the previous page) shows a collection of prolog clauses that specify the threat level of the bomb. We determined this threat level based on combinations of distance (*DIST*), blast yield (*YIELD*) and fuse length (*FUSE*). This threat level can then be supplied as an argument to the threat action rule shown at the top of the code block. This rule combines the threat level clause with the action clause at the bottom. As such we can imply from the high-level language used to describe our clauses, whether an action is required in specific circumstances. For example, if a bomb is close to an agent, with a high blast yield, and medium or low fuse length, it is a significant threat and must be defused.

By replacing any ground fact with a variable we can query the theorem-prover to produce results based on our existing clauses. In this instance, we would run the bomb_threat rule and provide a specific input for *DIST*, *YIELD* and *FUSE* but replace the *ACTION* input with a variable (say X). Hence, once we have a complete rule base that covers all relevant possibilities it then becomes a simple look-up process.

The same principle is also applied to the terrorists as well, however we have a much larger rule base, given that a terrorist can carry different weapons. For the reader's convenience we have provided all threat clauses and rules in Appendix C. If any threat rules fire, then these are issues that the agent must deal with immediately. Terrorist threats take precedence over bomb threats in the BruceWorld game. The reasoning behind this is that an agent could attack a player while they

are in the process of disarming a bomb.

If any of these rules fire, then the architecture responds by halting the current plan-action from executing and introduces the required response to the controller rule base for execution. Should an action selected by the plan manager be ignored, it will have to be considered on the following execution cycle. This functionality allows the agent to react to items in the environment that the planner cannot visualise. As a result, any destroy-target controllers that are employed, or any action that indicates the intent to defuse a bomb (outwith those specified by the planner) is a reactive action introduced by the RC.

Controller Rule Base

While the rule base was used to address issues that were beyond the scope of the planner, the controller rule base is a look-up table for individual actions. We associate actions from either the plan or threat rules with the actual controllers used in the environment. However this is not a simple one-to-one association as there are certain issues that have to be addressed when dealing with specific actions.

To clarify, we refer back to an example given at the end of Chapter 4. In the PDDL actions of the Nakatomi domain (Chapter 4 & Appendix A) we have multiple actions that interact with hostage agents (SLAP-HOSTAGE, KNOCKOUT-HOSTAGE etc.). For each of these actions we state in the PDDL model that the agent must be in the same room as the hostage. This is a correct assumption in terms of the planning-level abstraction, however we must consider real-world practicalities. Namely, if we are to slap/knockout/pickup a hostage, then we must not only be in the same room, but *physically next to the hostage in question*. Hence we now have two functions for the controller rule base to perform: not only must it provide a direct association from PDDL and threat rules to the actual controllers. it must also provide *bridge actions* for specific plan actions. These bridge actions act similar to the reactive actions used in the threat rule base and fix small inconsistencies between models (such as distances to hostages) in execution.

To facilitate this approach we provide two rule sets. The first provides a simple lookup function. These clauses are provided for actions where no issues need be remedied prior to execution. Such an example is a move action between locations, since the reactive controllers we apply will resolve any issues in execution.

Therefore we simply associate a particular action with the corresponding controller to be used in the environment as shown below:

```
quick−lookup(walk−agent−through−corridor,visit_waypoint).
quick−lookup(walk−through−doorway,visit_waypoint).
quick−lookup(walk−hostage−through−corridor,visit_waypoint).
quick−lookup(step−on−switch,visit_waypoint).
quick−lookup(step−off−switch,visit_waypoint).
quick−lookup(crawl−in−vent,visit_waypoint).
quick−lookup(crawl−out−vent,visit_waypoint).
quick−lookup(put−down−hostage,put_down_hostage).
```

The first argument of each of the above rules corresponds to one of the PDDL actions in our Nakatomi domain, while the second argument is one of the controllers available in the CL. In the example we see an association with not only the visit-waypoint base controller but also an extra controller named after the PDDL action 'PUT-DOWN-HOSTAGE'. As we will see in the following section, this is one of a set of hardcoded controllers available for execution.

The second rule set provides clauses for situations where a bridge action may be required. These clauses are dependent on the action and the circumstances required for successful execution. An example can be found on the next page.

```
slap−hostage(close,uneasy,slap−hostage).
slap−hostage(far,uneasy,visit−waypoint).
slap−hostage(near,uneasy,visit−waypoint).
slap−hostage(close,_,re−plan).
```

This is the set of rules in place for the slap-hostage PDDL action. The first variable we consider is the distance of the agent to the hostage. If we are relatively close to the hostage then we are satisfied, however if the agent is further away then we indicate that the agent must use the visit-waypoint controller to visit the hostage prior to slapping them. Furthermore, we have added to the clause that the hostage must be in the uneasy state for execution to take place. The precondition of this action dictates that the hostage must be in an uneasy state prior to being slapped. If any other situation occurs (denoted by the '_' value in the final clause) then we force the execution to halt and re-plan at the PM level.

This is an extra precautionary measure that the PM should be able to recognise beforehand.

In future, we could explore the possibility of the environment being in a state of constant change. As a result, clauses such as these could prove useful during deliberation. For now however, it is simply an error catch that should never be triggered. For further reference, the complete series of controller rules can be found in Appendix D.

Once a controller has been selected then we must retrieve it for actual execution. To do this the result is passed down to the control library.

5.3.4 Controller Library

The third and final component of REAPER is the controller library. The CL is designed to act as an interface for the retrieval of specified controllers that are then used to instigate the changes demanded by the plan or to resolve issues highlighted by the RC. To retrieve a controller, we require three pieces of information:

The Action: Naturally, we need to know what action the agent wishes to execute.

The Actor: The agent that will execute the desired action must be assigned to the retrieved controller.

The Focus: The entity that the action will interact with to achieve a goal.

The focus is a key component of controller construction that directs the actions we try to execute. As the action varies, so does the focus: in navigation situations the destination location will be considered the focus of the controller. In BruceWorld, if we are interacting with a hostage, switch or bomb then that entity is considered the focus of the action. This information is required to ensure that the controller can not only reference the entity for its own purposes (such as the inputs to the ANN hierarchy) but in certain situations be able to apply changes to the focus as part of the action. A simple illustration of this is the slap-hostage action, where we must change the state of the hostage having completed the action.

The CL houses two types of controller for use in the game. Primarily it is a repository for reactive controllers, in this instance our ANNs we explored in Chapter 3. Secondly, it also houses a small collection of hardcoded, scripted behaviours. These scripts are provided to resolve particular actions that we have

not trained ANNs to accomplish. Often these scripts represent PDDL actions that do not exhibit any physical actions we can observe given the level of abstraction and control we have enforced in the BruceWorld game. Rather, they simply script the necessary changes to the current state of the environment to satisfy the post-conditions of the PDDL action. Referring to the slap-hostage action, we do not consider the actual action that takes place here, since movement of the agents' arm is outwith the controls of the game. However we still need to ensure that the resulting state in the hostage still arises. Hence the script implements the changes required to represent the post-condition state.

To provide a simple controller creation scheme, the CL relies on a factory method design pattern; an object-oriented programming design pattern that provides an interface for object creation without specifically referencing the object type to be created. Hence there may be multiple types of object we can create while only using a simple variable to indicate what type we require. In this instance we have resorted to the use of enumerated types in the Java language for identification. An enumerated type allows us to maintain a series of constants or members whilst also providing basic functionality for each member. In this instance we have created an enumeration whose members are all potential actions retrievable from the CL. Each member of the action enum carries the relevant code required to retrieve the desired controller.

Retrieving a Reactive Controller To retrieve one of our pre-trained reactive controllers from the CL we provide the interface with the controller name, actor and focus. Note from the examples shown in the RC (Section 5.3.3) and the complete list of control rules in Appendix D that we only ever request one of our base controllers for retrieval, since we are only interested in the goal we wish to achieve. The additional layer controllers are the concern of the CL, where one of our subsumption layered controllers that is prepared for execution. Referring back to the tasks of the ANNs discussed in Section 3.4.1, we require the relevant information to build each controller. Here we provide a breakdown of how each controller is constructed.

Visit-Waypoint: In this instance, the controller focus is the destination of the agent. By passing this location we then generate a position in the room as the target waypoint. Next we consider whether this is the only destination location we require or the final position in a chain of locations. If we are

in a situation where the agent must walk from one room to another, we provide intermediate waypoints to guide the agent more effectively.

To visit each waypoint in the sequence, we modified the existing visit-waypoint controller to store a queue of waypoints to be reached. There is no need to concern ourselves with the performance of the agent in this instance, as we already have a collection of chromosomes that can effectively reach one waypoint. We simply shift the focus of the ANN inputs once the agent has reached the waypoint at the front of the queue by removing it and focussing on the new head of the queue.

Once our waypoint queue is constructed it is fed into our objective class and we build the ANN controller. We build our standard two-input, two-output, feed-forward network with two hidden layers of three neurons and initialise the weights with a chromosome stored from our experiments. By then attaching a detect-obstacle layer controller through the subsumption setup, this network is also initialised with a pre-trained chromsome's weights. This means it must adhere to the topology we used during those experiments. In this instance we assume the agent will require obstacle avoidance and navigation capabilities due not only to the layout of the world but the additional obstacles in each of the locations. We also assume there will be no hostile agents attacking our agent at this juncture as in accordance with the deliberation process we have shown in this chapter, this action would not be allowed unless we were confident no other agents would interfere.

Grab-Item: In this case, the focus is the artefact we wish to retrieve. Due to the design of the executive, we rely on the assumption that not only is the agent in the same room as the artefact but there are no hostile elements that may interfere with execution. Hence we draw our inputs for the ANN base controller from the artefact object. In progressing, we next add a detect-obstacle layer controller on top to ensure the agent can manoeuvre around obstacles in the location. Once again we set the topologies and initialise the weights of the ANNs based on the chromosomes from our previous experiments.

Destroy-Target: Using the focus of the enemy target, we assign the three inputs used to operate the ANN. Once the weights are initialised with those of an existing chromosome, the addition of a dodge-shell controller on top of the

base layer provides evasive capabilities.

Once the controller is fully constructed, the system stores it in the main class and awaits the signal for execution. While these controllers are reasonably robust, we are aware that there is a possibility they may fail to complete their assigned action. Therefore, when the signal is sent to activate the controller and run it in the game, we set a limit of 2000 in-game time steps to complete the assigned action. This is more than sufficient to execute any given action effectively. If this time limit is exceeded, the game terminates with an error message indicating the reactive controller is at fault.

Retrieving a Scripted Controller The retrieval of scripted behaviours is a relatively simple process. Initially, we request that a reactive controller be retrieved from the controller library. This is the standard procedure for controller retrieval since for the majority of actions the library will return an ANN hierarchy. If no reactive controller is retrieved we know that a scripted controller is available for this action, therefore we retrieve the scripted controller for this action by retrieving the enumerated type for that action. Each enumerated type in the CL stores an execution script if no reactive controller is available. Once we have retrieved the enumerated type we then initialise the scripted action with the relevant actor and focus. In the case of interacting with a hostage, the actor is assigned as the Bruce agent, while the hostage is assigned as the focus of the action. Moreover when executed, it will simply apply the appropriate changes to the world through these entities.

5.4 Testing & Results

> Computers are useless. They can only give you answers.
>
> Pablo Picasso

In this section we now assess the performance of our agent architecture against a series of increasingly challenging tests in the BruceWorld game. We begin by reporting on our initial tests that validated our approach, followed by implementing some small test maps as normal followed by applying threats and uncertainty. To give a clearer indication of how the architecture works in operation, we provide

step-by-step walkthroughs from some of these experiments. Finally, we run a series of more challenging maps to assess performance, followed by comparing the effects of increased threats, maximum uncertainty and the effect of removing model validation and threat detection from the agent.

5.4.1 Initial Tests

Before beginning proper testing, we wanted to assess the performance of each facet of the architecture during construction. This necessitated the testing of five unique scenarios:

1-Step, 0-Sub-Step Problems: Scenarios where the planner would create a solution with only one action to execute. When an action passed from the planner it mapped directly to a controller and executed with no bridge actions. Moving from one room to the next provided a simple test to satisfy our concerns.

1-Step, 1-Sub-Step Problems: A problem where the agent needs to commit a bridge action prior to executing the original action. This situation was easily created by interacting with hostages or defusing a bomb in the same location, given that the agent would have to move over to the object to undertake the desired action.

n-Step, 0-Sub-Step Problem: Committing multiple directly-mapped actions and executing them sequentially. Simple corridor navigation tasks through multiple locations were used for this test.

n-Step, n-Sub-step Problem: These reflect a standard problem instance for the Bruce agent to execute. This was done using some simple tests involving both navigation and object retrieval/interaction.

Hostage Actors: These problems would test the actor function of the system to ensure that actions could also be delegated to the hostages for execution.

These tests were not considered as part of our complete system evaluation, but were merely conducted to assess the basic functionality of the system during construction and are reported to highlight our structured implementation and testing. Once confident that a piece of functionality would operate effectively, we would then run these simple test problems for verification. Once satisfied, we

would proceed to developing and refining the next piece of functionality. Only once all tests were completed did we move onto our basic performance tests.

5.4.2 Basic Performance Tests

For our first series of performance tests we wanted to cover the range of functionality that the architecture provides. We required a set of simple problems where the solutions would require bridge actions, threat resolution and resolving model inaccuracies from uncertainty. A brief description of each test case is given below, with a complete listing of each PDDL problem file found under Appendix E.

BasicTest1 - Figure 5.4: The agent must navigate through a sequence of locations and corridors to reach a destination location.

BasicTest2 - Figure 5.5: A hostage is trapped in a nearby location with an armed bomb. Our agent must defuse the bomb and escort the hostage back to the starting position.

BasicTest3 - Figure 5.6: A hostage is trapped inside a nearby room blocked off by a locked door. The agent must unlock the door and allow the hostage to escape.

BasicTest4 - Figure 5.7: This is similar to BasicTest3 except there are two hostages trapped in separate rooms. Agent must rescue and escort both hostages to the goal location.

BasicTest5 - Figure 5.8: Our agent must escort two hostages to a nearby location. However one hostage is unconscious and the other is in an uneasy mental state.

Each of these problems combined with our definition of the BruceWorld game satisfy our intentions made in Chapter 1 to create tasks that are outwith the scope of the reactive EA-trained ANNs and the deliberatve AP methods. However, theoretically it is plausible that our new agent is capable of solving them.

We focus many of these initial problems on hostage retrieval, since potential uncertainty will have an impact on overall performance. We assessed each problem instance 10 times in the standard mode (i.e. no modifications to the problem) and a further 10 times with the hostile entities and uncertainty added to the model. This was to assess how well our agent performed when faced with elements that may impede progress, or force the agent to rethink the plan of action.

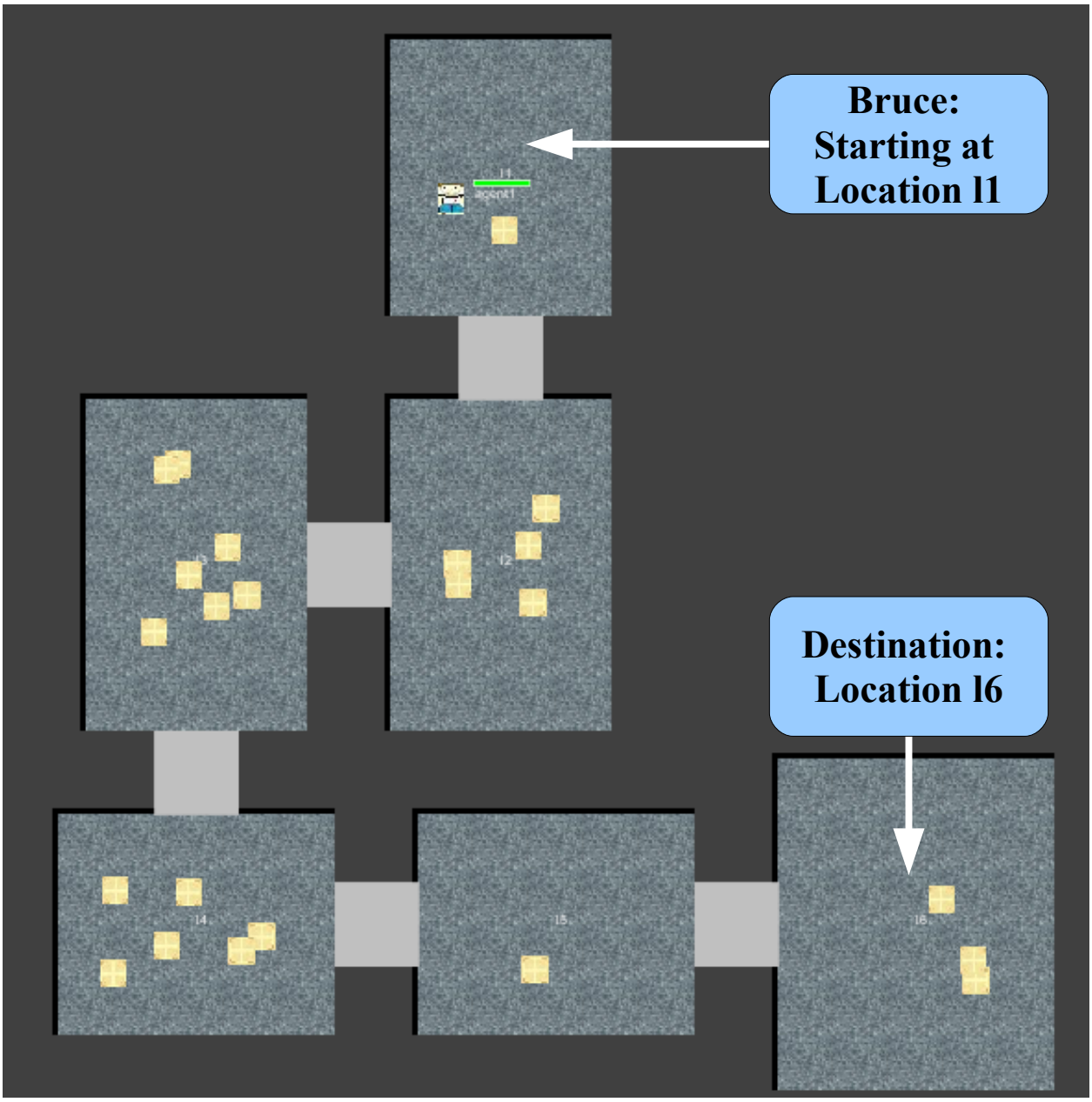


Figure 5.4: One instance of the BasicTest1 problem file in BruceWorld, showing six fully connected rooms that the agent must navigate through.

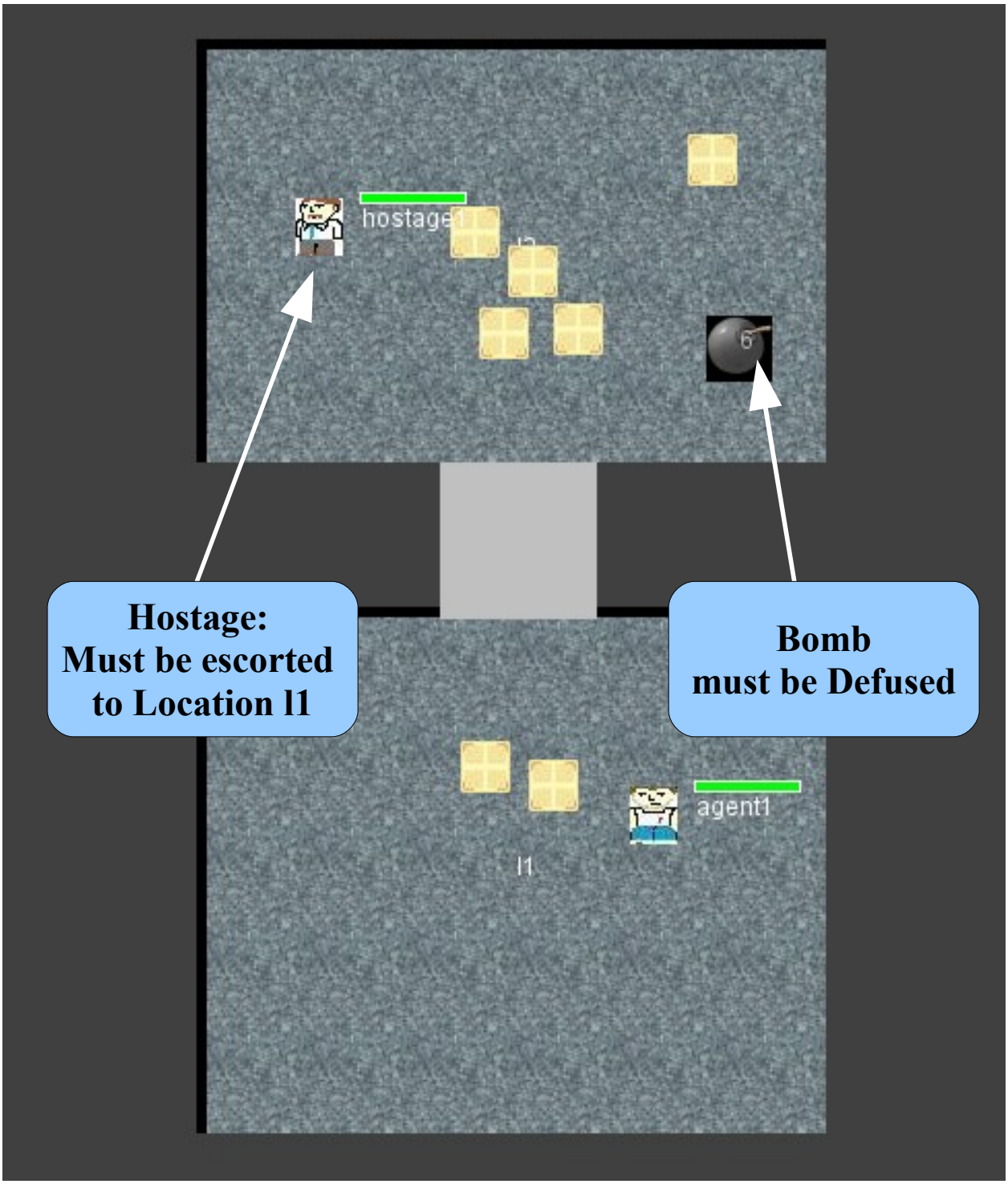


Figure 5.5: A map of the BasicTest2 problem in the BruceWorld game. In this instance, Bruce must escort a hostage between two locations, but not before checking whether the bomb in l2 requires defusing.

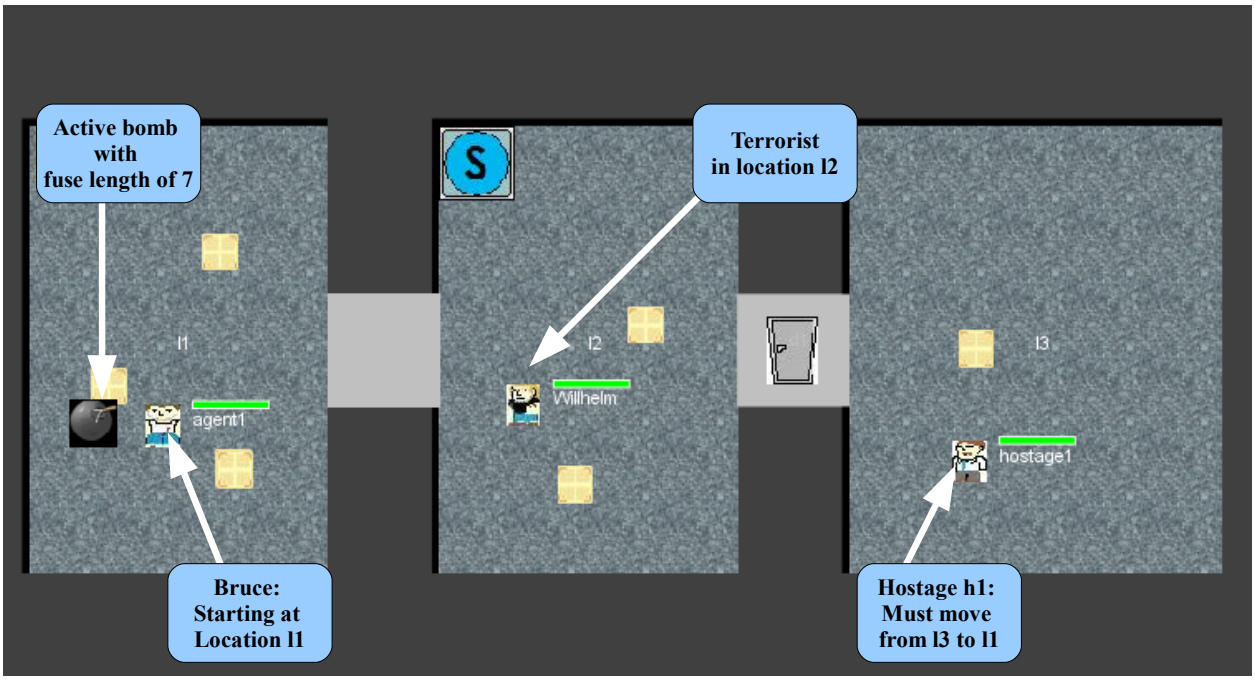


Figure 5.6: A BruceWorld map of the BasicTest3 problem, where Bruce must unlock the door between locations l2 and l3 and guide the hostage back to l1. However in this instance, threats such as a terrorist and an active bomb have been included.

Results

We provide a series of statistics from running each initial problem as normal in Table 5.1 and then with a 100% probability of threats and uncertainty in Table 5.2. After discussing these results, we will then give an execution walkthrough of select instances to clarify the exact behaviour of the agents.

In the case of threats and uncertainty, we have ensured that at least one bomb and one terrorist exist in the problem. Furthermore, if any aspect of the problem can be modified through uncertainty, then at least one change will be made. These results indicate that the agent can solve the majority of the provided test runs. Furthermore, it is also capable of solving these problems for the most part when adversarial agents and uncertainty are added to the problem. One of the most notable changes that happens when threats and uncertainty are added, is the significant increase in the average number of actions per problem. These additional actions are the result of actions deliberated from numerous re-plans and flagged rules. Re-plans may result in additional resolution actions for issues we were not aware of until the agent encountered them, while the rule base will

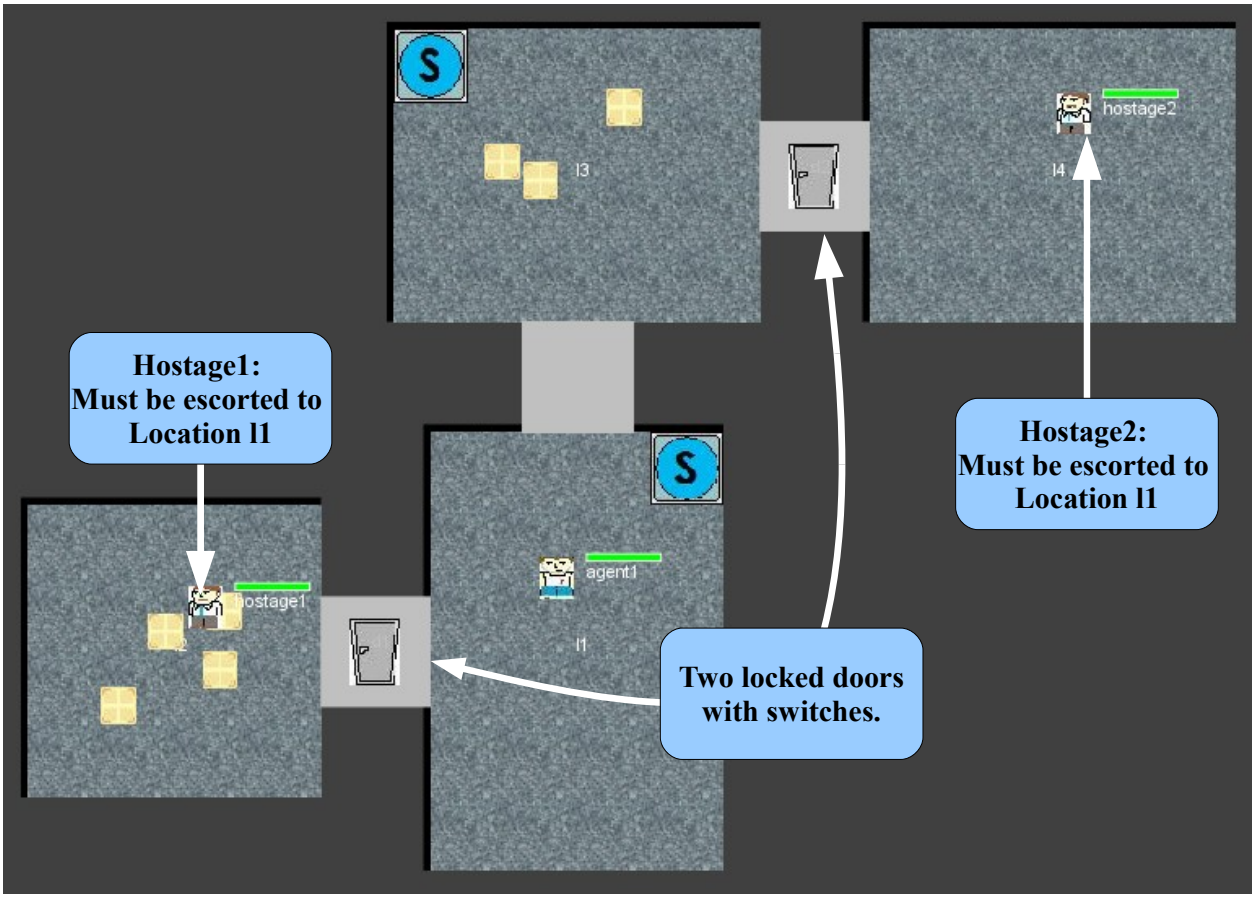


Figure 5.7: A BruceWorld map of the BasicTest4 problem. This instance is very similar to BasicTest3, except now there are two hostages in separate locations.

Table 5.1: Statistics from 10 runs of our basic performance test problems. We provide statistics regarding agent performance, as well as the number of times the system interacts with JavaFF and the Prolog rule base.

Instance	Completed	Initial Plan Length	Average No. Actions	Average PM Runs (Avg. Time)	Average RC Runs (Avg. Time)
BasicTest1	10	5	5	1 (85.7ms)	5 (<1ms)
BasicTest2	10	3	3.78	1 (182.6ms)	3.78 (<1ms)
BasicTest3	9	6	6.78	1 (212.78ms)	6.78 (1.44ms)
BasicTest4	10	8	8	1 (209.3ms)	8 (<1ms)
BasicTest5	10	5	6	1 (177ms)	7.11 (<1ms)

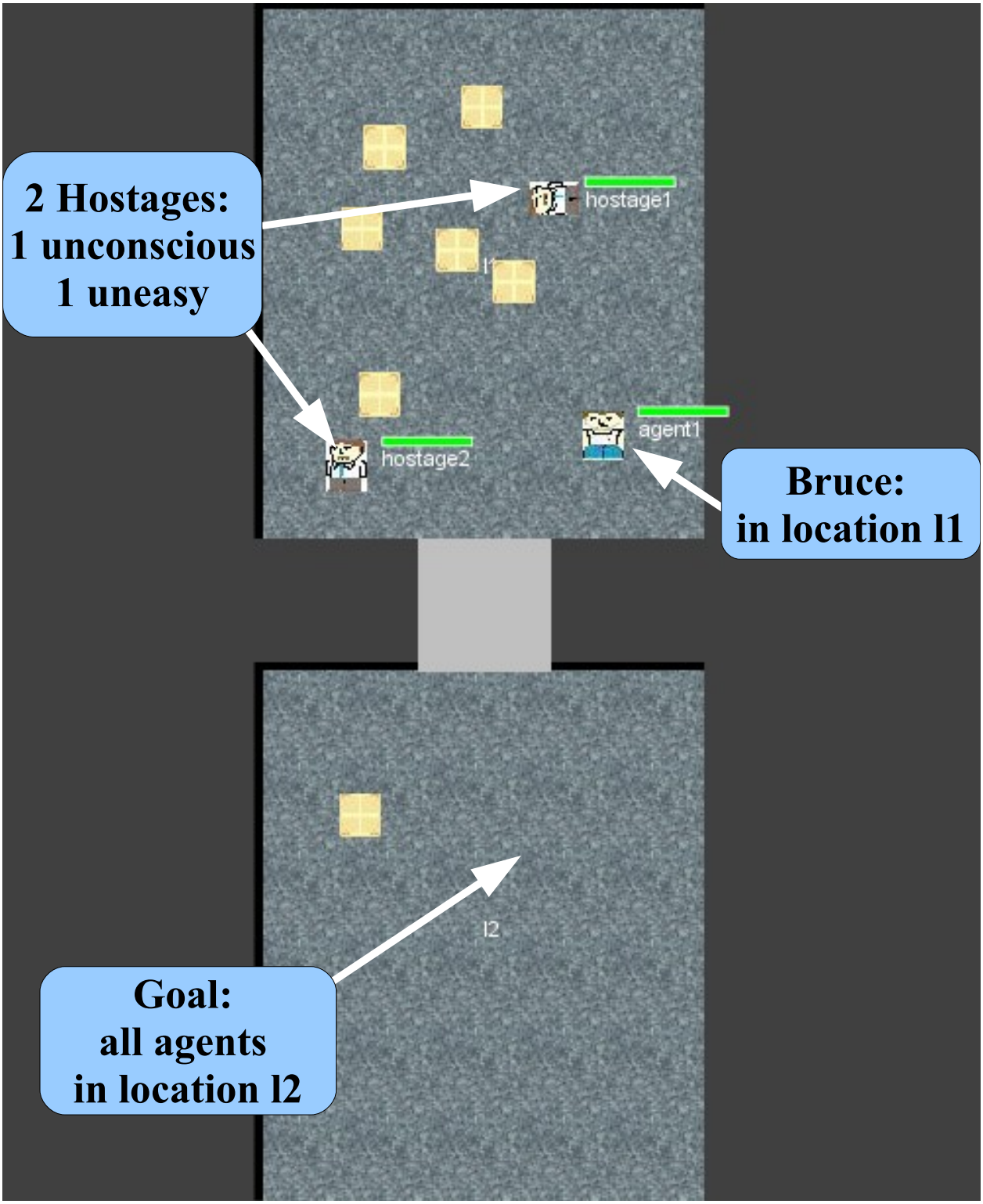


Figure 5.8: An instance of the BasicTest5 problem in BruceWorld, where Bruce must assist two hostages in uneasy and unconscious states to the adjacent location.

Table 5.2: Statistics from 10 runs of each of our basic performance problems when running with threats and uncertainty. This has a considerable impact on the number of actions that are committed, as well as the number of interactions (and the average time taken) with JavaFF and the Prolog rule base.

Instance	Completed	Initial Plan Length	Average No. Actions	Average PM Runs (Avg. Time)	Average RC Runs (Avg. Time)
BasicTest1	8	5	9.86	3.29 (100ms)	21 (1.14ms)
BasicTest2	10	3	10.89	3.11 (30ms)	24.9 (<1ms)
BasicTest3	8	6	12	3.14 (83ms)	15.4(2ms)
BasicTest4	8	8	13.8	3.75(62ms)	14.75(1.13ms)
BasicTest5	8	5	14.86	3.86 (47.4ms)	18.71 (<1ms)

be demanding supplementary actions due to the threats recognised in the RC clauses.

Perhaps unsurprisingly, in the threat/uncertainty runs (Table 5.2) there are a larger number of failed tests. In order to fail a test either a circumstance arose that the system could not generate an appropriate response to, or the agent was eliminated due the threatening elements that were introduced. Given that we were present to observe each test we can safely conclude that the latter circumstance was responsible. In each incompleted test, failure occurred as a result of the destroy-target controller being unable to eliminate a terrorist, or a bomb detonated and killed one of the agents. While we have created capable reactive controllers that are adept at solving tasks, the destroy-target task was always challenging for the agents. Given that we made some real progress with the final controller, it still does not win every match on average. There is one similar instance in Table 5.1 where the agent also failed due to the performance of the reactive controller. This was due to the area it was moving through being heavily cluttered by objects obstructing the route. While typically the agent will navigate around these objects rather competently, in this situation the placement of the obstacles was such that they boxed the agent in. While there was a path that could be navigated through this area, the obstacle sensors found the area too 'hot' to traverse and would often attempt to turn the agent around and backtrack. However given the goal oriented nature of its behaviour, it would then - much to our frustration - turn around and try again. This continued until the permitted time for the the action expired and the test was considered a failure.

System Performance During these initial tests we made some interesting observations of the performance of the planner and prolog querying. These are aspects of the system that we wished to assess for two reasons. Planning is often a time and CPU consuming process, and when dealing with game software, the time taken to carry out this deliberation would become noticeable to the player. Hence we wanted to check to see just how quickly this can be completed on average. When observing the RC performance, we wished to ensure that accessing the Prolog code through the JPL interface would not cause similar bottlenecks in processing.

Returning to the results shown in Table 5.1, we can see the runtimes from JavaFF and the JPL Prolog queries. In the standard problems we observe a significant amount of time is taken for the JavaFF planner to complete the initial plan. The actual time measures 213 milliseconds in the worst instance, given this is somewhat irrelevant on a human level, it is a significant amount of processing time, especially when we take into consideration that game developers wish to commit very few CPU cycles to AI processes per second. Naturally we were concerned about the average time taken when the system requires multiple re-plans to deal with model inaccuracies. Surprisingly, the JavaFF times in Table 5.2 tell a different tale. As we can see, the average time to plan is significantly reduced. We observed during these initial tests that any re-plans would consume very little CPU cycles in comparison to the initial run. The time taken for the initial run can be attributed to loading the planner itself into memory, once this is completed then we can trust the planning process to run much faster.

On a more positive note, when we observe the JPL Prolog queries in both series of tests we note that the querying process is incredibly fast. Often in the standard tests, we note that the number of prolog queries will naturally correlate to the number of actions in the plan since we need to retrieve the related controller. However, when we modify the problem, the number of Prolog queries explodes. This can be attributed to the increased number of actions taken to complete the plan, which can be observed from the difference in the initial number of plan actions to the number of actions executed. Furthermore, there will also be multiple references to the threat rule base to assess whether the threats found in the environment merit recognition and future resolution. Nevertheless the time taken is truly satisfying, given that regardless of the number of queries made, the average time is approximately measured as 1 millisecond. In fact in many circumstances the average time would be less than 1 millisecond, since the

difference in system time could not be measured in nanoseconds.

Agent Behaviours We assessed the performance of the agent based not only on the success in completing a given test problem but by also observing the agent's behaviour. Fortunately as we observed from the tabled results, agents were capable of completing the majority of these test instances. The actual behaviours appeared very focussed and consistent between tests. When hostile elements were added to the problem, the agent would react to and deal with these competently, resolving each issue in the current location in decreasing order of priority. We also observed situations where the agent recognised bombs that would not pose a sufficient threat to merit defusing, hence it would continue the execution of the plan without consideration for the bomb. Agents would immediately recognise inaccuracies in the state model when made apparent and the forced re-plan would be successful in all cases. Hence the model is sufficient given that there were no circumstances where the agent failed to model and re-plan. Once our agents generated a new plan, they would continue without any complications and complete the task (even after multiple re-plans). As noted previously, all failures occurred as a result of the evolved controllers being unable to complete a task or simply timing out. No failures ever occurred at the planning level.

An interesting observation was made when running *BasicTest5* using the threat and uncertainty modifiers where the planner would ocassionally give inaccurate plans to the agent. When the agent recognised that changes had been made to the game due to the uncertainty, the PM would be forced to regenerate the state and commit a re-plan. This re-plan would create a series of actions whose terminal state did not reflect the final state of the problem. The system would continue to execute the actions until the final action was completed. As expected, the PM recognised that the model and final state do not match and another re-plan is committed. At this point the correct sequence to reach the goal state was calculated and execution proceeded as normal. While we were pleased to see our agent compensate for this turn of events, we were concerned as to how this circumstance would arise. A first course of action was to assess the circumstances in the game that would result in this incorrect plan.

To highlight this bizarre occurrence, we refer to the actual problem and the modified instance that led to the erroneous plan. Referring to listing 5.9, the original problem revolved around moving two hostages from location one to the goal location two. Given that there is only a corridor between the two locations

```
(:init
(at agent1 l1)
(free agent1)
(at hostage1 l1)
(at hostage2 l1)
(unconscious hostage1)
(uneasy hostage2)
(corridor l1 l2)
(corridor l2 l1)
)

(:goal (and
(at agent1 l2)
(at hostage1 l2)
(at hostage2 l2)
))
```

Figure 5.9: The original PDDL description of BasicTest5.

this is a relatively simple plan of action once we devise how to deal with each hostage. Due to the initial state of each hostage, this led to a complete plan length of five actions as shown below:

(SLAP−HOSTAGE agent1 hostage2 l1)
(PICK−UP−HOSTAGE agent1 hostage1 l1)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l1 l2)
(WALK−HOSTAGE−THROUGH−CORRIDOR hostage2 l1 l2)
(PUT−DOWN−HOSTAGE agent1 hostage1 l2)

When we changed the problem using our uncertainty modification, a variety of different initial states emerged due to each hostage being assigned the calm, delirious or uneasy state. These new states were the culprit for our odd re-plan, with the simplest example occurring in listing 5.10. In this case, the agent would now have to carry the two hostages into the other room.

Interestingly, JavaFF would often return a plan that would move one agent into the other room but not the other. At first we feared this may be the result of an error in the domain description leading to what is referred to as a 'teleportation problem'; a flaw where two objects of the same type share attributes to the point the planner considers them the same object, such as two trucks (in the DriverLog domain) being in separate locations yet loading a package in one truck means you

```
(:init
(at agent1 l1)
(free agent1)
(at hostage1 l1)
(at hostage2 l1)
(unconscious hostage1)
(unconscious hostage2)
(corridor l1 l2)
(corridor l2 l1)
)
```

Figure 5.10: One of the PDDL states from BasicTest5 that resulted in an erroneous plan.

---

can immediately unload it from the other truck. This however, this did not appear to be the case, and after thoroughly examining the domain model there was no conclusive indication as to why this error occurred. We eventually discovered it was caused by an error in the JavaFF planner itself and not in our system. We explore this issue again in greater detail in Section 5.4.6, where we report on our efforts to resolve this issue after it caused further concern during our advanced tests.

Solution Walkthrough: BasicTest1 (Standard)

While the resulting statistics in Tables 5.1 and 5.2 are satisfactory and give an indication of overall performance, the actual pattern of behaviour may not be immediately obvious to the reader. To conclude this phase of testing, we select three problems and explore exactly how the architecture acts across the whole of the execution. In this section we begin with the easiest problem from this phase of testing: BasicTest1.

BasicTest1 can be found in Appendix E.1, where the agent must navigate through a series of five corridors from the start location *l1* to the goal location *l6*. In this test we have decided to explore this problem without the uncertainty of threats added. An example of this problem in the BruceWorld game can be seen in Figure 5.4. The resulting plan of action from JavaFF is simple: move between each room via the corridors and avoid any of the obstacles that get in our way:

(WALK−AGENT−THROUGH−CORRIDOR agent1 l1 l2)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l2 l3)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l3 l4)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l4 l5)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l5 l6)

Given we have not added threats or uncertainty then the agent need only execute each action from the plan in sequence. We shall now walk through the plan execution, explaining how each component behaves in this instance and ultimately show how the agent reaches the goal state and terminates execution.

1. **Initial State: (at agent1 l1))**

   (a) The Plan Manager is fed the model of the world and the plan is generated.

   (b) The PM first ensures that the model is accurate, which proves successful. At this point the first action (WALK-AGENT-THROUGH-CORRIDOR agent1 l1 l2) is taken from the queue of remaining actions and passed to the Rule Controller.

   (c) The RC first checks for any Terrorists or Bombs in the room. Given there are none it will move to the controller rules and flag one of the quick-lookup clauses. This will return the desired controller: visit-waypoint.

   (d) This is fed to the controller library where the visit-waypoint subcontroller is selected. The start and end locations are parsed from the PDDL action (l1 and l2) and the controller library builds a simple path through the corridor to l2 (as discussed in Section 5.3.4). This path of waypoints is fed to the visit-waypoint subcontroller. It is then added to the base layer of a subsumption hierarchy and a detect-obstacle controller is placed atop and deemed as ready.

   (e) The subsumption controller is fed to the agent and the agent executes the action until the reactive controller indicates the action is completed.

2. **Going for a Stroll: (at agent1 l2/l3/l4/l5)**

   (a) PM will first update the model to coincide with the executed action, then verify the current state is not the goal state. Once verified it will next check to ensure there are still actions remaining in the plan. Since both are valid, execution will continue.

   (b) Steps 1(a) through 1(e) are then repeated since the case will be the same in all states.

3. **Goal State: (at agent1 l6)**

   (a) PM assesses the current state of the world, recognises it as the goal state and terminates execution.

Solution Walkthrough: BasicTest 5 (Standard)

For our second solution walkthrough, we will look at BasicTest5 under standard conditions. The problem (visualised in Figure 5.8) was to move two hostages from one location to another nearby location (again see Appendix E for full PDDL definitions). However, one of the hostages is unconscious, and the other is feeling uneasy. As neither can move to the goal location of their own volition, Bruce is required to intervene. The resulting plan is given below:

(SLAP−HOSTAGE agent1 hostage2 l1)
(PICK−UP−HOSTAGE agent1 hostage1 l1)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l1 l2)
(WALK−HOSTAGE−THROUGH−CORRIDOR hostage2 l1 l2)
(PUT−DOWN−HOSTAGE agent1 hostage1 l2)

The resulting plan is more diverse than the previous one; the agent first slaps the uneasy hostage in order to calm him down, then picks up the unconscious hostage. Now the agent must walk to the goal location l2 with one hostage over his shoulder and the other following behind him. All that remains in order to complete the problem is to put the unconscious hostage on the floor.

1. **Initial State: (at agent1 l1) (unconscious hostage1) (uneasy hostage2)**

   (a) PM is fed the PDDL model of the problem and the plan is generated.

   (b) PM runs a successful model verification to ensure everything is correct, and selects the first action (SLAP-HOSTAGE) to run through the Rule Controller.

   (c) RC will verify no threats exist in the environment and so we can focus on the plan-action. When querying the controller rule base no result will be generated from the quick-lookup clauses. Hence the system must explore clauses specific to the SLAP-HOSTAGE action. The system classifies the distance between the agents and applies it to the slap-hostage clause. Assuming the circumstance in Figure 5.8 for the sake of our walkthrough, the system recognises Bruce is too far from the hostage. Hence a visit-waypoint bridge action is flagged.

   (d) The visit-waypoint flag is sent to the CL, the subsumption controller is provided with the destination within close proximity of hostage2 and the agent executes the action.

2. **Slapping Around: (at agent1 l1) (unconscious hostage1 (uneasy hostage2))**

   (a) After model verification is completed, the agent will recognise that the SLAP-HOSTAGE action has yet to be completed. The PM will flag it for execution.

   (b) This time the RC will approve the action since the additional condition that Bruce be close to the hostage is now satisfied. Subsequently it will send a message for the slap-hostage controller.

   (c) Once this message is received in the CL, the system will recognise there is no subsumption controller for this action. Instead an instance of the agent is provided the necessary scripted actions to execute the slap action.

   (d) The controller is retrieved from the CL and subsequently executed.

3. **Heavy Lifting: (at agent1 l1) (unconscious hostage1) (calm hostage2)**

   (a) The model is updated and verified and the PICK-UP-HOSTAGE action is selected.

   (b) RC runs the quick-lookup clauses and finds the corresponding controller is grab-item and sends a message to the library.

   (c) The library takes information from the current state to plot the course for the grab-item controller. A 2-layer subsumption controller is built with additional detect-obstacle layer for obstacle avoidance.

   (d) The agent executes the controller, moving through the environment to pick up the unconscious hostage.

4. **On the Move: (at agent1 l1) (carrying agent1 hostage1) (calm hostage2)**

   (a) The next two actions (WALK-AGENT-THROUGH-CORRIDOR & WALK-HOSTAGE-THROUGH-CORRIDOR) simply correspond to visit-waypoint controllers. Bruce and hostage2 in turn are fed 2-layer subsumption controllers with waypoint navigation and obstacle avoidance which are executed sequentially.

5. **Baggage to Declare: (at agent1 l2) (carrying agent1 hostage1) (at hostage2 l2)**

   (a) The final action to execute is PUT-DOWN-HOSTAGE.

   (b) Once the model is verified, the RC dictates that the put_down_hostage controller must be retrieved. CL finds a corresponding hardcoded controller and it is fed to the agent for execution.

   (c) Once the action is completed, the plan-model is updated and the goal state validated, execution completed.

Solution Walkthrough: BasicTest3 (Threats and Uncertainty)

For this final walkthrough, we visit BasicTest3. However this time we have ensured that the threats and uncertainty in the world are active. In this instance not only will the problem differ from the model at some point, but there will also be bombs and terrorists present in the game. In Figure 5.6 we see one instance of the problem in BruceWorld. However, unlike our last two walkthroughs, we see there is an active bomb and a terrorist in locations l1 and l2 respectively. Once deliberation starts, the following plan is constructed:

(WALK−AGENT−THROUGH−CORRIDOR agent1 l1 l2)
(STEP−ON−SWITCH agent1 l2 s1 d1)
(WALK−HOSTAGE−THROUGH−DOORWAY hostage1 l3 l2 d1)
(STEP−OFF−SWITCH agent1 s1 l2 d1)
(WALK−HOSTAGE−THROUGH−CORRIDOR hostage1 l2 l1)
(WALK−AGENT−THROUGH−CORRIDOR agent1 l2 l1)

We conduct this final walkthrough through based on the example in Figure 5.6.

1. **Initial State: (at agent1 l1) (at hostage1 l3) (calm hostage1)**

   (a) The plan model is verified for accuracy, however verification fails due to the bomb in location l1. Hence the plan model is reconstructed from the current state of the game and the plan is re-formulated. However, the same plan is constructed since no goals were given in the problem file to defuse a bomb.

   (b) Now that the model is clean and the refreshed plan in is place, the move action is sent to the RC.

(c) RC checks the local environment for any threats. The bomb in location l1 is picked up (at present, the agent cannot see the terrorist in l2) and classifies the distance, fuse length and blast yield against the threat rules. Given that each feature may lie in more than one set, we run all combinations and select the result with the highest threat level. Given the distance is close and the fuse length is medium then irrespective of blast yield the bomb will be flagged with at least a medium threat level. Hence the bomb will need to be defused and the defuse_bomb action is selected as a reactive action.

(d) The defuse_bomb action is passed through the controller rule base, with no quick-lookup result. Instead we run the classified distance through the bridge clauses. At this distance the agent is close enough to defuse the bomb. Hence the defuse_bomb action is sent to the CL.

(e) The hardcoded defuse_bomb controller is retrieved and the action executed.

2. **Moving On: (at agent1 l1) (at hostage1 l3) (calm hostage1)**

(a) The model is checked again, however due to the bomb being disarmed we are once again inaccurate. The model and plan are refreshed to ensure all is correct.

(b) The first plan-action (WALK-AGENT-THROUGH-CORRIDOR) is selected, no threats are detected by the RC and a 2-tier navigation controller is built. The action is executed without any problems.

3. **Fightin' Time: (at agent1 l2) (at hostage1 l3) (calm hostage1)**

(a) The PM approves model accuracy, selects STEP-ON-SWITCH action and passes this action to the RC.

(b) The RC recognises there is a terrorist in the room and assesses the enemy's capabilities using the threat rules. A destroy-target controller is selected as the next action.

(c) The CL selects a destroy-target subcontroller and sets the focus of the controller as the terrorist. An additional dodge-shell controller is added to create a 2-tier subsumption controller.

(d) The agent executes and attacks the terrorist and, with a bit of luck, survives unscathed.

4. **To the Rescue: (at agent1 l2) (at hostage1 l3) (calm hostage1)**

   (a) The PM selects the STEP-ON-SWITCH action again since the plan-model is still valid.

   (b) The RC notes no threats or issues with execution and selects the visit-waypoint controller.

   (c) A 2-tier visit-waypoint/detect-obstacle controller is built and pointed towards the switch. The agent moves to stand on switch.

   (d) Once completed, the model is updated and a visit-waypoint/detect-obstacle controller is sent to the hostage to exit the now unlocked doorway.

   (e) Another simple execution moves the agent back into location l2[1]

5. **Hang on a minute. . . : (at agent1 l2) (at hostage1 l2) (uneasy hostage1)**

   (a) The PM model check proves inaccurate as a result of the uncertainty in the world. Now that Bruce can actually see the hostage he identifies his uneasy condition, hence the state is recreated and a new plan is built. The new plan adds an extra (SLAP-HOSTAGE agent1 hostage1 l2) action to the front of the existing plan.

   (b) The RC notes that there is too large a gap between the two agents and an additional visit-waypoint bridge action is required. The agent moves closer to the agent.

   (c) The model is verified, the rules are satisfied and the SLAP-HOSTAGE action is executed.

6. **Let's Get Out of Here!: (at agent1 l2) (at hostage1 l2) (calm hostage1)**

   (a) The two remaining move actions process as normal with no further complications.

   (b) Goal state is reached and executive terminates.

[1]Note he actually moves back *into* the room. This is because even though in the plan-model a switch is in a room, an agent can only be on a switch or in a room.

Throughout these three walkthroughs, we've seen a variety of different circumstances such as model failure, additional threats, bridge actions that the executive has dealt with accordingly. This gives us a clear indication of how the system operates and how dependent the agent is on each individual component.

5.4.3 Advanced Tests: Setup and Preliminary Results

Up to now we have carried out a series of tests that give us some indication of our agent's capabilities. Furthermore, we have also explored how the architecture solves problems in specific circumstances courtesy of our solution walkthroughs. To complete our testing process, we wanted to carry out a series of tests to assess the flexibility of the system and the maximum level of performance it can attain. In short, we wanted to see just how far we could push the agent before it is incapable of solving the problem. This would be dictated by the problems we assign to it and the level of threat and uncertainty applied. We hypothesised that provided the system continues to operate within memory constraints then the system should be able to solve any problem and handle any uncertainty that is applied, under an assumption that the uncertainty does not result in unsolveable problems. We believed that the greater the threat level, the more likely it is that agents would eventually succumb to the overwhelming odds. Hence, we aim to discover at what level of threat such circumstance arises. A second but equally important test to run, was to highlight the importance of each aspect of the system. While the walkthroughs have shown how each component plays a part in the execution, are we really that reliant on them to solve each problem? If we simply ignore threats or plan-model accuracy can we still solve our test problems? This will help conclude whether our efforts in creating these components were necessary.

In order to conduct these tests, we decided to create a new set of problems. These new problems are more challenging than the tests used previously and littered with more obstacles in the environment than before. Below we give a brief description of each problem, with the complete PDDL definitions found in Appendix F.

AdvancedTest1 - Figure 5.11: The agent must navigate through a series of rooms to a specific location. There Bruce must retrieve a first aid kit to take back to an injured hostage. Once the hostage is healed, another unconscious hostage must also be picked up and escorted to the goal location.

AdvancedTest2 - Figure 5.12: Bruce must crawl from one location to two others to restore an uneasy and an unconscious hostage, carrying one to the goal and then helping the other to escape from a locked room prior to ensuring his own escape.

AdvancedTest3 - Figure 5.13: Bruce and three hostages are trapped in four separate rooms connected via four doorways. Each room has a switch connected to a door, hence the agents must cooperate in order to release each other from their room to the goal location.

AdvancedTest4 - Figure 5.14: A large problem spanning 10 locations. The agent must assist one hostage in getting from one side of the map to the other, while also ensuring that the hostage in the room with him can still reach his goal location next door.

We encountered issues with plan construction when trying to create some of these new problems. Once we had established a given problem file we would test it against the JavaFF planner to ensure that for the simplest case the agent must be able to execute and solve the problem. However JavaFF, specifically the RPG heuristic, would ocassionally struggle with some of the ideas we wanted to introduce. One casualty of this is the *blocked* predicate which restricts movement across specific corridors. To maintain some simplicity in the planning model, we simply denote a path from A to B is blocked, and once the *CLEAR-RUBBLE* action is applied, a corridor is constructed between these two points as an effect. However the JavaFF heuristic will ignore the fact that the path is no longer blocked, hence solutions were never found during any of our preliminary tests. As a result we tried to provide challenging problems whilst ensuring that they are solveable using the RPG heuristic.

Standard Test Results

We begin by providing statistics from 30 runs of each advanced problem file in Table 5.3. In these results we see more instances where the agent fails to satisfy the assigned goals, with AdvancedTest4 proving the most challenging. Our reports from running these experiments indicated that this was due to controller time-out when attempting to execute an action. In each circumstance, this arose from the agent's inability to completely navigate through a heavily cluttered environment prior to the action timeout. From our observations we noted this was due to the

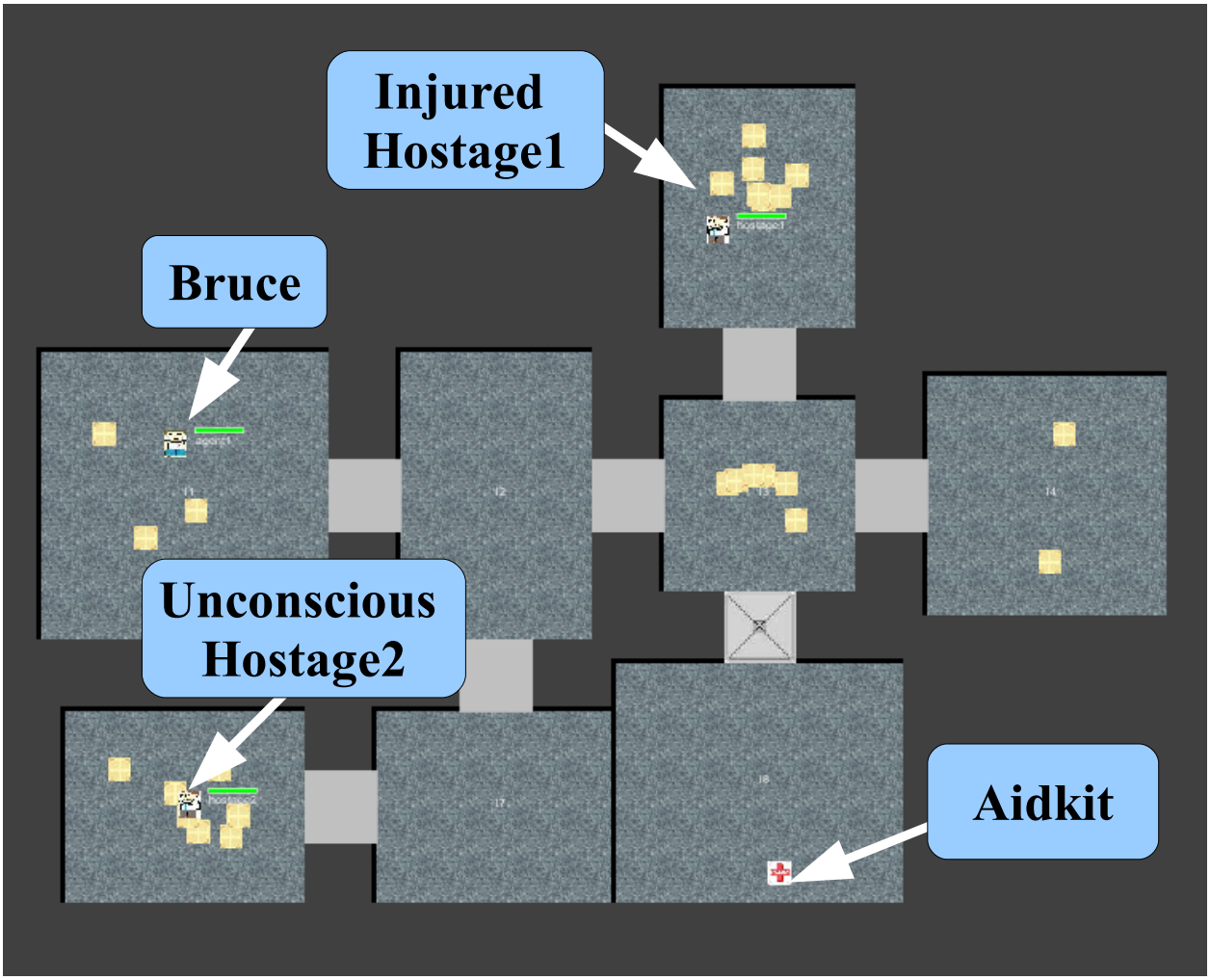


Figure 5.11: One instance of the AdvancedTest1 problem. Here Bruce must move to a vent, crawl through to grab an aidkit to help a hostage and then also deal with the unconscious hostage on the other side of the map.

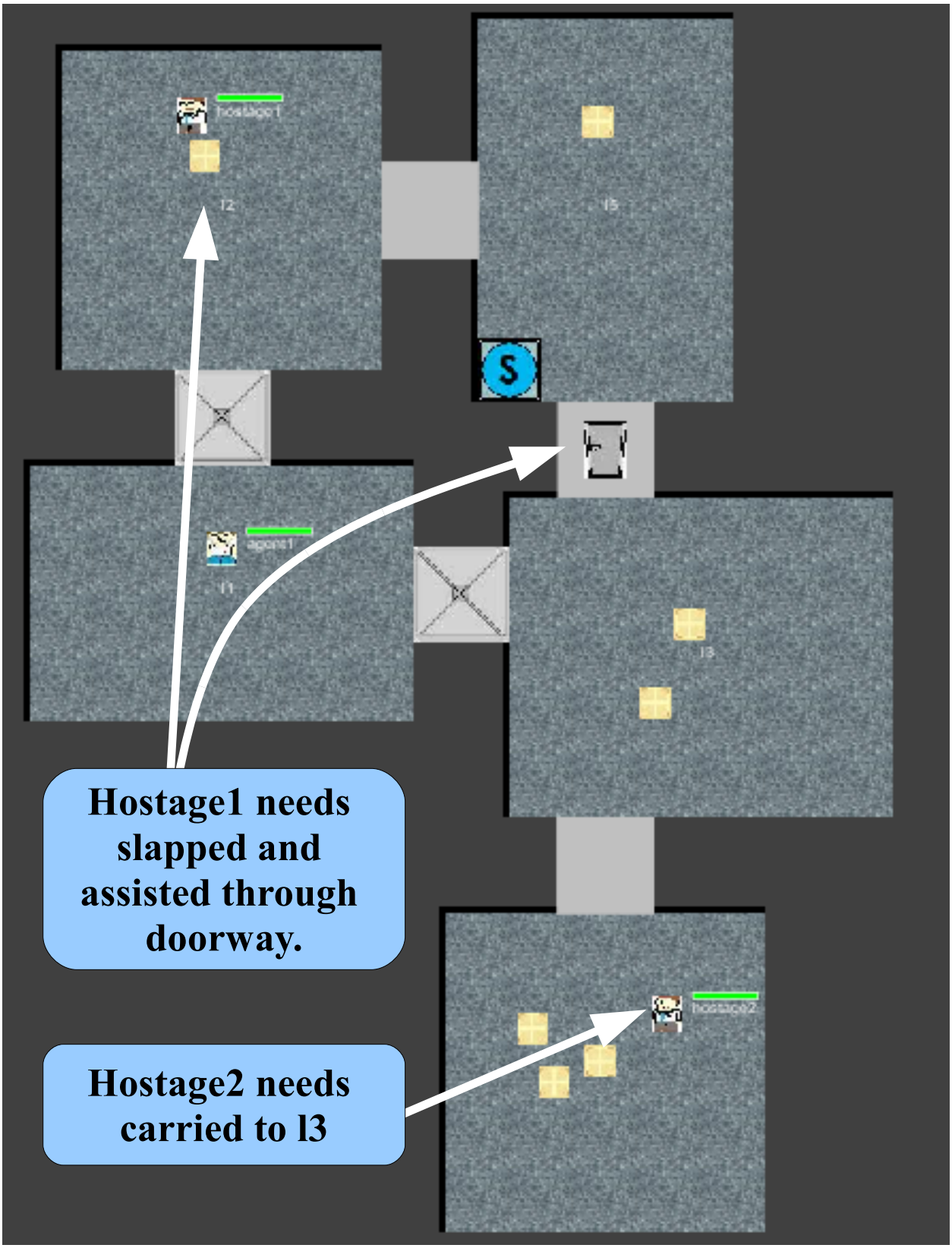


Figure 5.12: In AdvancedTest2, Bruce needs to bring one hostage to their senses before he helps them get out from behind a locked door, then retrieve the unconscious hostage and put them both at the goal location.

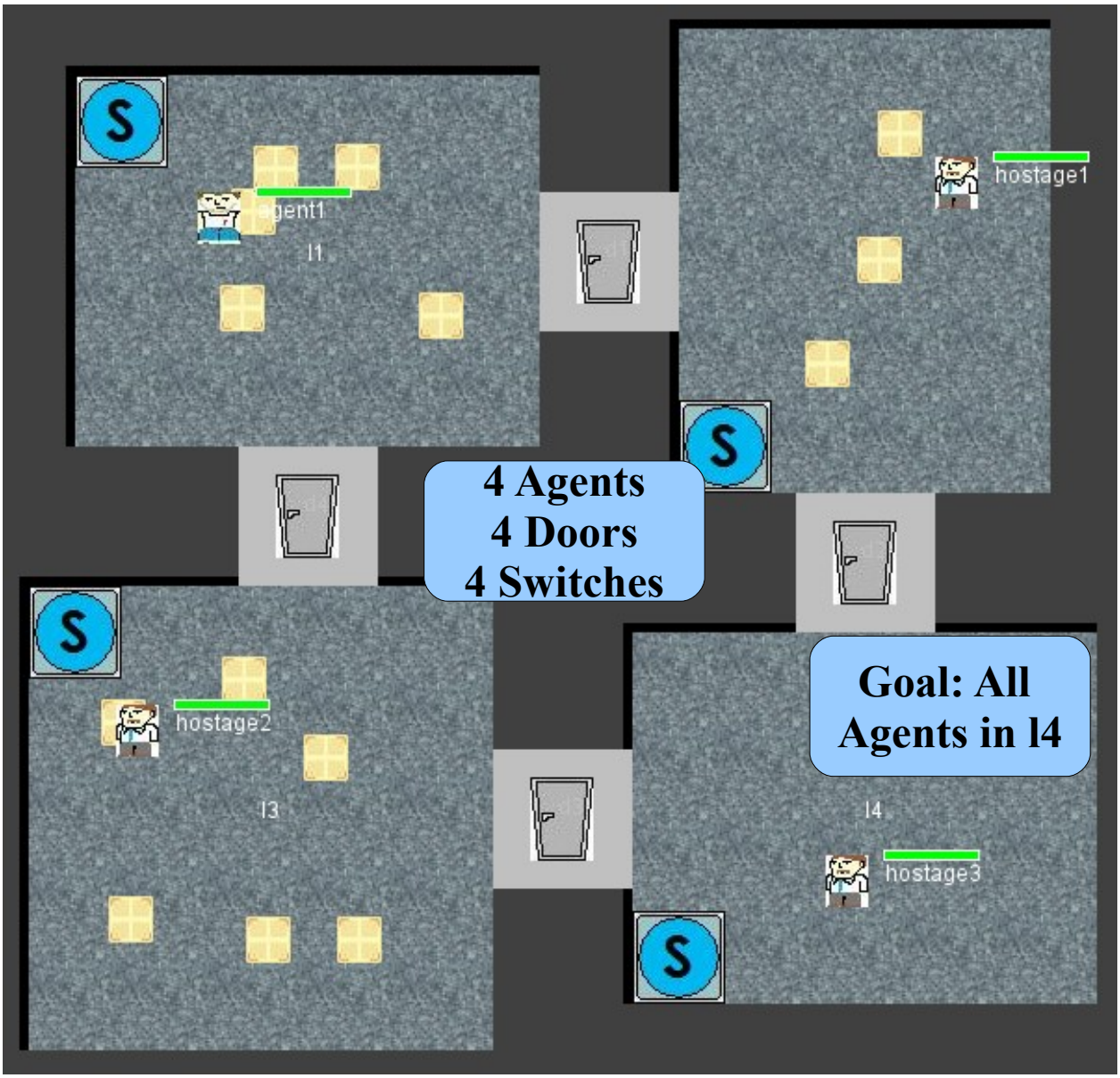


Figure 5.13: The AdvancedTest3 problem is a tricky puzzle since there are four doors to four locations. The agents must work together to get from one room to another.

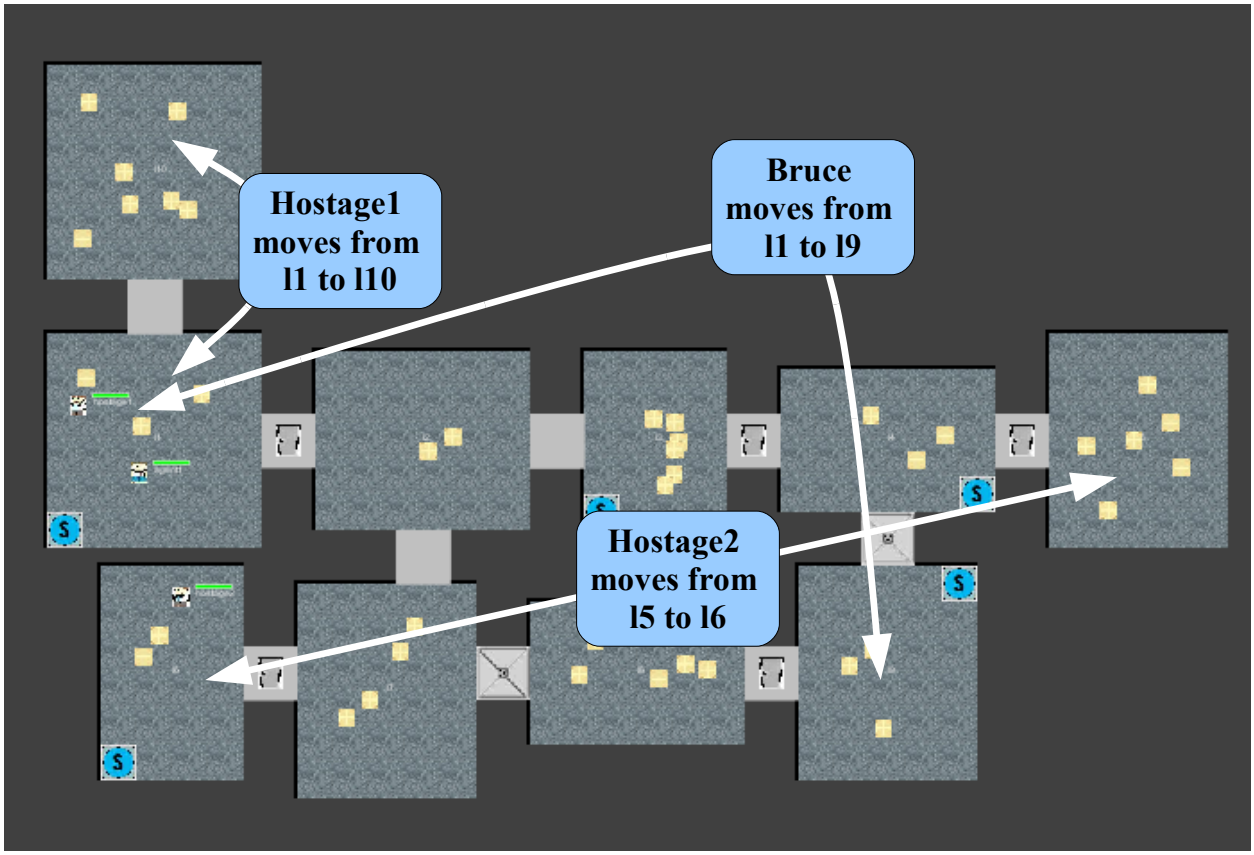


Figure 5.14: The final AdvancedTest file shows Bruce having to navigate across a large range of locations to help get one hostage to the other side. Meanwhile, another hostage needs to move to the next room, however the plan hinges on his cooperation.

Table 5.3: Statistics from 30 runs of our basic performance test problems. Note the average number of actions is taken only from the successfully completed examples.

Instance	Completed	Initial Plan Length	Average No. Actions	Average PM Runs (Avg. Time)	Average RC Runs (Avg. Time)
AdvancedTest1	21	20	22.76	2 (459ms)	22.14 (1.95ms)
AdvancedTest2	20	21	21.85	1 (778ms)	21.9 (2.05ms)
AdvancedTest3	27	10	10	1 (754ms)	10 (4.15ms)
AdvancedTest4	16	25|26	26.81	1.56 (3942ms)	26.56 (1.5ms)

(not particularly long) reach or view of the obstacle avoidance sensor. The random placement of obstacles could lead to situations where the agent would continually react to stimuli and be unable to move competently.

While we provide these results primarily for comparison with later tests, it is interesting to observe two aspects of the planning process. Firstly, the time taken to devise a plan of action is significantly larger than those shown in Table 5.1. As we stated previously, it was our intention to create more challenging problems for the system to solve. Whilst the architecture is successful in devising plans we are now entering large numbers of CPU cycles to achieve success. In the rather taxing AdvancedTest4, note that the average time can now be measured in whole seconds. Keeping focus on AdvancedTest4, we also note that the initial plan length varies between 25 and 26 steps and the Plan Manager may be executed more than once. Similar behaviour can be observed in AdvancedTest1, where the system always created an initial plan of 20 steps but would later re-run the Plan Manager. Investigation into this phenomenon led us to discover that JavaFF was once again at fault. The planner would, in the case of AdvancedTest1 always and AdvancedTest4 often, devise a plan that failed to satisfy all goals. As a result, the system would note at the end of plan execution that a goal was not satisfied and a new plan was constructed to amend this fault, often with the new plan being only one or two actions in length. This once again caused concern and we highlight later in this section the further work conducted to resolve this issue.

For the reader's consideration we provide the individual plan times from the AdvancedTest4 runs in Table 5.4. As shown previously from the average of this sample in Table 5.3, this problem takes a significant amount of time for the agent to solve. Furthermore, we observe that in the event JavaFF constructs a sub-optimal plan of 26 steps, then a re-plan is necessary at a later juncture.

Table 5.4: Records of the total time taken in order to plan solutions for the first 10 runs of AdvancedTest4. We also provide the subsequent replan times when necessary.

Instance	Init. Plan Length	Plan Time 1	Plan Time 2
Run 1	26	5573	101
Run 2	25	6650	n/a
Run 3	26	3659	93
Run 4	25	4417	n/a
Run 5	26	3451	102
Run 6	25	5278	n/a
Run 7	26	3540	130
Run 8	25	6376	n/a
Run 9 (Fail)	26	4271	n/a
Run 10 (Fail)	25	6534	n/a

5.4.4 Advanced Tests: Threats and Uncertainty

Next we test the maximum level of threat and uncertainty that can be applied before our agent is unable to complete the problem. We begin by assessing the performance of each problem with varying threat levels, followed by examining how well the agent reacts to increased modifications to the problem.

Threat Density

In our previous threat experiments we increased the probability of a terrorist or bomb existing somewhere within the problem file. Given there were few locations in these problems this still posed a significant challenge. Here we wish to increase the potential number of threats that exist throughout the problem file. As our advanced tests carry larger numbers of locations, we wished to assess how many threats can be inserted before the agent fails frequently. At this juncture, we hypothesised that the inclusion of large numbers of terrorists would have a significant impact on the results. While handling one or two terrorists per problem may be difficult, placing them throughout all locations was beyond the capabilities of our reactive controllers. Conversely, we were confident that the agent could easily deal with increasing numbers of bombs, provided sufficient time is given to reach them.

In Tables 5.1 and 5.3 we observed that the agent may be unable to complete the task due to a controller failure. This is due to the obstacle detection being unable to handle the randomly instantiated obstacle positioning. However we did not

want this to interfere with our experimental results. Hence in these experiments, any game terminated due to an obstacle controller failure was ignored. Only pertinent failures are reported from these sample runs.

For each threat experiment, we focus on the number of bombs or terrorists that exist in the problem with respect to the number of locations. This is referred to from here as *threat density*. As the threat density increases, the probability of Bruce encountering a threat in each location also increases. We begin by running each advanced problem with bombs included; as the threat density increases, the number of bombs gradually increases until all rooms hold one bomb. We then apply the same principle to terrorists and finally using both bombs and terrorists in the same problem.

Upon commencing our bomb tests, we came to the conclusion that adding them to the domain description may have been a poor design decision. While we were keen to include bombs in the PDDL domain to allow problems that require their defusing, their inclusion as threats often results in inaccuracies in the current plan-model. This is highlighted by our walkthrough of BasicTest3, where the agent had to rebuild the state model after the discovery of a bomb in location *l1*. As a result, the PM was forced to commit a re-plan due to a model inconsistency. While this resolved the model inaccuracy, it had no real impact on the plan, since bomb defusal is considered a problem for the RC. Thus we are commiting numerous re-plans for no real purpose. This was highlighted when we approached AdvancedTest1, where the agent would rebuild the model and re-plan 16 times in the worst case, even though no further actions were included in the plan. Therefore from this point on, we remove the bomb assertions from the PDDL domain file.

Bombs Only For our bomb density experiment, we ran each problem 10 times at threat densities of 0.2, 0.5 and 1.0. This was intended to gradually increase the difficulty of the problem as the tests continued. The results from these experiments are summarised in Table 5.5. As can be seen from these results, success and average progress in these problems decreases - with the exception of AdvancedTest2 - as the number of bombs increases. As bomb density increases, the amount of time spent defusing them also increases. Given that Bruce is preoccupied by disarming an increasing number of bombs, the remaining bombs have a greater chance of exploding. As a result, many of these tests terminated as a result of Bruce reacting to nearby bombs, while a hostage in a distant location was killed

Table 5.5: Results of running each advanced test file 10 times at bomb densities of 0.2, 0.5 and 1.0.

Instance	Bomb Density	Completed	Average Progress
AdvancedTest1	0.2	7	92.4%
AdvancedTest2	0.2	4	71.7%
AdvancedTest3	0.2	10	100%
AdvancedTest4	0.2	5	76.9%
AdvancedTest1	0.5	3	68.6%
AdvancedTest2	0.5	5	76.1%
AdvancedTest3	0.5	10	100%
AdvancedTest4	0.5	4	78.2%
AdvancedTest1	1.0	0	43.2%
AdvancedTest2	1.0	6	79.06%
AdvancedTest3	1.0	5	85%
AdvancedTest4	1.0	3	74.14%

by a bomb merely feet from them. In these cases, the bomb could have been defused by Bruce; however this action was queued behind all plan-actions that led him to the bomb's location *and* defusal of all bombs in previous locations (or they luckily exploded without anyone nearby). If Bruce had a greater knowledge of the world then he would have recognised which bombs carried a greater threat across the scope of the problem and could act accordingly.

Terrorist Only Next we gradually increased the number of terrorists found in each problem. As we stated previously, it was hypothesised that the agents would struggle to complete problems as the number of terrorists increased. Once again, we ran each of the advanced test files for 10 separate runs with threat densities of 0.2, 0.5 and 1.0. The results from these experiments can be found in Table 5.6, where we can see that our concerns were justified. At the lower threat density levels, the agent is capable of eliminating terrorists and continuing on to complete the plan. However, this is not as frequent as we would have liked. Two problems have been highlighted by these experiments; the quality of the reactive controllers and the context in which they are executed.

Our destroy-target reactive controller was designed to eliminate one opponent as efficiently as possible for EvoTanks. While the results in Section 3.4 show that this can be achieved, we must realise that the controller's 'lifespan' was the time in which it executed in EvoTanks. As such, if an agent eliminated the opponent with

Table 5.6: Results of running each advanced test file 10 times at terrorist densities of 0.2, 0.5 and 1.0.

Instance	Terrorist Density	Completed	Average Progress	Average Terrorist Elimination
AdvancedTest1	0.2	5	63.2%	60%
AdvancedTest2	0.2	4	48.7%	60%
AdvancedTest3	0.2	9	93.9%	90%
AdvancedTest4	0.2	4	53.1%	55%
AdvancedTest1	0.5	3	43.6%	50%
AdvancedTest2	0.5	2	37.2%	46.7%
AdvancedTest3	0.5	2	65.83%	35%
AdvancedTest4	0.5	0	37.7%	26%
AdvancedTest1	1.0	0	11.25%	20%
AdvancedTest2	1.0	0	14.4%	20%
AdvancedTest3	1.0	1	47.7%	32.5%
AdvancedTest4	1.0	0	11.7%	11%

only 1 hitpoint remaining, the match would terminate and a fitness value would be assigned based on their performance. However, the next time this behaviour was executed, the health bar was replenished. In BruceWorld, the agent is then expected to continue as normal, with the possibility of finding another enemy to eliminate. However in these circumstances, the playing field is often no longer level since the agent has now less hitpoints than their opponent. To rectify this in the future, we would require a new training scenario that is more relevant to the manner these controllers are executed.

There are further issues relating to the context in which the destroy-target controller is executed. While the controller is only really interested in eliminating the enemy opponent, we may also need to consider other factors beyond the perceived scope of relevance. For example, there were several instances where the execution failed since a hostage was caught in the crossfire between Bruce and a terrorist and killed. This is a different case where the context of the action must be considered in order to prevent undesired results in practical application. Again this could potentially be resolved in future work using BruceWorld-related learning evaluations.

Terrorists & Bombs For our final threat assessment, we run the same experiments again with both terrorists and bombs inserted into the problem. We are

Table 5.7: Results of running each advanced test file 10 times at terrorist and bomb densities of 0.2.

Instance	Density	Completed	Average Progress	Average Terrorist Elimination	Terrorist & Bomb Deaths
AdvancedTest1	0.2	4	50.7%	60%	100%/0%
AdvancedTest2	0.2	6	67.2%	70%	100%/0%
AdvancedTest3	0.2	7	36.56%	70%	100%/0%
AdvancedTest4	0.2	4	54.06%	50%	50%/50%

interested to see how well the agent can cope with these larger problems when dealing with more prevalent threats. For this experiment we decided to run at a threat density of 0.2, since our previous results indicated the probability of success would be low.

Table 5.7 shows the results of this experiment. We are pleased to note while the average progress and number of runs completed has dropped, there is only a modest depreciation between these statistics and those in Tables 5.5 and 5.6. Furthermore, it appears that the inclusion of bombs at this terrorist density level has little impact on the overall performance. In the majority of situations where the agent failed it is a direct result of the terrorist agent(s) killing Bruce. Only in a small number of situations in AdvancedTest4 was this the result of bomb detonation. In some situations Bruce would lose a hitpoint while attempting to navigate through cluttered areas to defuse an active bomb. This worked to the benefit of the terrorists as it meant they would begin any future conflict at an advantage. It is clear from this work that the reactive controllers for eliminating threats would require more extensive testing and refinement for any practical applications.

Uncertainty Testing

For our uncertainty testing, we ran the same advanced test files but now with uncertainty applied to any entity that is susceptible to modification. Hence the state of all hostages in these problems would be affected. When creating these tests, we needed to highlight the effectiveness of the model verification and re-planning modules of the architecture. We hypothesised that, irrespective of the test being ran, the architecture will always solve a problem provided the following

conditions hold true:

- The state-space is still soluble, i.e. irrespective of the changes made by the uncertainty, a solution can still be found by the planner.
- Given the nature of the game, any action dictated by Bruce to a hostage must still be executable after uncertainty is applied.

We believed that the planner will always find a valid, although not necessarily optimal, solution to any changes we made, provided the first assumption holds true. Ultimately, this test would highlight that the architecture is still bound within the constraints of the planner's capabilities. Furthermore, given the egocentric nature of the architecture and the game's constraints on knowledge of the world, we assumed the agent would fail in certain circumstances. In these cases, the agent would not be aware of the state of other entities, given he cannot see them as he is in another room. When these circumstances arise, the agent should fail in execution.

The result of running each advanced test file under such conditions is found in Table 5.8. It's clear in the case of AdvancedTest1 & 2 that irrespective of the changes made to the problem the agent can still solve it through re-planning. In fact in these examples, the changes often resulted in a plan that was shorter than the inital one. This is very common in AdvancedTest2, where changing the states of the two hostages resulted in new solutions. Returning to the screenshot of this test shown in Figure 5.12, we can see there is a hostage trapped behind a locked door with only air vents providing an alternative route. In some of our test runs this agent changed from being uneasy to delirious. The resulting plan that accommodated for this change would knock the hostage out and carry him through the airvents to the goal. Previously, the agent was given no impetus to knock out the hostage, as no provision for such an action was allowed when in an uneasy or calm state. As a result, the agent would take a far longer[1] route to completion. As for the other hostage, if he was not unconscious this would often lead to shorter paths, such as slapping him or not needing to interact with him at all.

Conversely, the agent cannot solve any instance of AdvancedTest3 or AdvancedTest4 when uncertainty is applied, as we expected! These files were built deliberately with the intent to fail during uncertainty execution. Whilst each

[1]Though potentially optimal with respect to the state

Table 5.8: Results of running each advanced test file 10 times with a 100% chance of hostages being effected by uncertainty.

Uncertainty				
Instance	Completed	Init Plan Length	Plan Actions Completed (Mean)	Re-plans (Mean)
AdvancedTest1	10	20	19.4	1.8
AdvancedTest2	10	21	13.7	1.7
AdvancedTest3	0	10	0.9	0
AdvancedTest4	0	25-28	0.33	0.44
Reduced Uncertainty				
AdvancedTest3	1	10	2.2	0.23
AdvancedTest4	10	25-28	28	1.8

of these tests were challenging, they did not lend themself to modification as in doing so Bruce must dictate actions to hostages that can no longer apply them. While certain changes made to the problem render it insolvable, there are a larger class of problems that are solvable from a planning perspective but insolvable at execution due to the game's constraints. If we made changes to the game to allow Bruce to 'see' the state of any hostage before execution then a re-plan could be committed and a solution would be created, provided the state-space was still soluble. However the point here is to highlight that our agent is operating at an individual level within the constraints of the world.

Returning to Table 5.8, we provide a series of extra experiments with *Reduced Uncertainty* for AdvancedTest3 and AdvancedTest4 where we ensure that the state space is not only soluble, but also ensure that each action can be executed under the constraints of the world. In the case of AdvancedTest4 this meant we could only modify the hostage in location l1 since we have no concrete knowledge of the state of hostage *h2*. As we can see in Table 5.8, this meant that the agent could solve the problem effectively.

However, in the case of AdvancedTest3, if we only modified agents that were in full-view from the initial state, then we would have to run the problem with no uncertainty applied! Our compromise was to make select changes to the problem that ensured *a plan could be created* irrespective of the modification. As such, successful execution was then reliant on a specific subset of plans where the agent would not dictate actions to modified hostages until the model discrepancy was rectified. Essentially, having devised plan P from our defined world state w, we

needed P to execute such that it does not conflict with the real world state w'. If a conflict occurred, then the execution would terminate and be considered a failed attempt. However, if during execution our architecture recognised that $w \neq w'$, we could commit our re-plan to generate plan P' and continue as normal. While this was technically feasible, given the conditions classical planners operate under, notably in Definition 3 (Chapter 2) where it's stated that we cannot influence plan creation, the chances of success were slim.

As we can see in Table 5.8, the agent was successful in one instance and the average number of plan actions completed improved. In this one successful instance, the agent was fortunate in not dictating any actions to a hostage that was incapable of executing them (i.e. executing P did not conflict with world w'). Furthermore, as Bruce progressed through the environment, he then discovered the model discrepancy and re-planned. This allowed Bruce to 'fix' any hostage prior to dictating an action to him. Ideally, we would have liked to dictate to the planner the manner in which the agent executed the plan. However, this kind of interference is not permitted and would be rather difficult to incorporate into the existing planner.

5.4.5 Advanced Tests: Feature Removal Testing

In our final tests, we explore how strongly our architecture relies upon the threat detection and plan-model verification components of the Rule Controller and Plan Manager. To achieve this, we run the same combined bomb/terrorist threat experiment with a density of 0.2 shown previously and *turn off* the threat detection code built within the RC. This is followed by running the same uncertainty experiments as in the previous section but without the model validation and re-planning features of the PM. We hypothesised - nay, hoped - that the removal of these key features would lead to poorer results from the agent.

The results of running in a threat-filled environment with the threat detection removed are shown in Table 5.9, while the uncertainty tests are shown in Table 5.10. As we can see from Table 5.9, the removal of threat detection even at a 0.2 density level has an impact on performance when compared to the results in Table 5.7. Prior to execution we considered the possibility that the agent would be able to solve these problems comfortably given that it could simply ignore a lot of what is happening around it. However, this was not necessarily the case. Bruce would often spend time interacting with hostages and switches while, unbeknownst

Table 5.9: Results of running the REAPER architecture without any threat-detection faculties. We run against each advanced test file 10 times at terrorist and bomb densities of 0.2. We see a notable decrease in completed instances when compared to Table 5.7.

Instance	Completed	Average Progress	Average Terrorist Elimination	Terrorist & Bomb Deaths
AdvancedTest1	3	46.74%	0%	71.4%/28.6%
AdvancedTest2	3	60.95%	0%	42.9%/57.1%
AdvancedTest3	7	91%	0%	66.6%/33.3%
AdvancedTest4	3	64%	0%	71.4%/28.6%

to Bruce he is being assaulted by a nearby terrorist. Other - often amusing - situations included leaving a hostage in a room with an active bomb and even escorting them to the said room before handling other affairs. In fact it was seldom the case that Bruce himself would be killed by the bomb, but rather one of the hapless hostages entrusted to his care. This accounts for the increased percentage of bomb deaths in these experiments.

Meanwhile, the uncertainty experiments with plan verification removed, also yielded satisfying results, with no run being able to complete a problem instance. In these circumstances, the agent would ignore the differences in the world and attempt to execute the action the plan had prescribed. Given that in many of these cases the action could either not be executed or simply had no effect (e.g. slapping an unconscious hostage), eventually the game or the architecture would raise an error indicating that execution was infeasible. For the sake of completeness we provide results using the reduced uncertainty setup for AdvancedTest3 and AdvancedTest4. As expected, this yields the same results. This is damning evidence that strongly supports the model verification approach, since the results shown in Table 5.8, while not perfect, yielded far stronger returns.

The difference in results indicates that while Bruce potentially can survive without any threat detection, i.e. he merely focusses entirely on the plan-goal assigned, the success rate decreases. Whereas when the model verification is removed, Bruce is completely incapable of solving problems with uncertainty. This gives substantial credence to our argument that both threat-detection and model verification with re-planning provide greater functionality than without.

Table 5.10: Results of running the REAPER architecture without the model-recognition and re-plan faculties. We run against each advanced test file 10 times with a 100% chance of hostages being effected by uncertainty. This approach has a significant impact on performance when compared to results in Table 5.8.

Uncertainty				
Instance	Completed	Init Plan Length	Plan Actions Completed (Mean)	Re-plans (Mean)
AdvancedTest1	0	20	2	0
AdvancedTest2	0	21	1	0
AdvancedTest3	0	10	0	0
AdvancedTest4	0	25-28	0	0
Reduced Uncertainty				
AdvancedTest3	0	10	1.8	0
AdvancedTest4	0	25-28	0	0

5.4.6 Domain Validation & Plan Discrepenacies

Now that our testing and evaluation was complete, we wished to spend time analysing the discrepancies in the planner's solutions we previously discovered in BasicTest5 and which also occurred periodically in AdvancedTest1 and AdvancedTest4. We previously discussed in our results section that the architecture is capable of handling these invalid plans, due to the discrepancies that arise in the PM's model validation forcing a re-plan. However, it is in our best interests to understand whether the plan-model itself is at fault or whether this is an issue in the JavaFF planner.

Domain Validation We began by examining the Nakatomi PDDL domain file to ensure there are no loopholes or gaps in the model's logic. This was achieved by first re-examining it by hand, which lead to no real insight, followed by the application of VAL, is an automatic validation tool developed by Richard Howey, Maria Fox and Derek Long as part of the 3rd International Planning Competition in 2002. The reader may recall from Section 2.3.2 that the 3rd IPC released PDDL2.1 to the planning community. Thus, VAL was provided to give researchers a tool to validate domain models and the plans that competitors would generate during the IPC competition. Since then, VAL has been extended to incorporate features of later versions of PDDL, notably PDDL3 and PDDL+ (Howey et al. [2004]).

To apply VAL we require the domain file, problem files and a selection of resulting plans. We used our advanced test files coupled with a selection of different solutions generated by JavaFF. Other than an insignificant discrepancy[1] in the domain model, which we quickly corrected, VAL confirmed that the Nakatomi domain model was sound and valid. Furthermore, VAL recognised both valid and invalid plans that JavaFF generated. Note the invalid plans are often those that remove one or more actions that satisfy the final goal in the problem. Hence we were satisfied that our model was not at fault and we turned our attention to the JavaFF planner.

JavaFF Satisfied with the results from VAL, we decided to turn to the JavaFF planner. Our intention was to discover whether the JavaFF planner or the built-in scheduler was at fault during the advanced tests in Section 5.4.3, and more importantly, why this issue persisted.

The JavaFF planner creates a *total-order plan*, i.e. a plan that is defined in sequential order and must be executed as such. Once this plan is devised, the scheduler generates a *time-stamped plan* where each action is given a time stamp to indicate at which stage of execution it can be effected. Often multiple actions will be assigned the same time-stamp, given they are actions that do not conflict with one another or share resources. We begin by verifying that the time-stamped plan is a completely scheduled plan and that no actions are lost during the scheduling process.

We applied AdvancedTest1 and AdvancedTest4 to JavaFF and explored both the total-order and time-stamped plans generated. A simple report that we posted to our terminal indicated the number of actions in each plan, followed by the complete plan being displayed. It is at this juncture we note an anamoly; in both instances, the total-order and time-stamped plan lengths are reported as the same, while the time-stamped plans are missing one or more actions. These missing actions are those that the PM's re-plans always generate. While the time-stamped plan is missing an action, our real concern was the total-order plan giving an incorrect action count.

The plan length is provided courtesy of an action hash set[2] that is accessible via the Plan interface. Upon further examination, this action set is generated from

[1] VAL demanded we change the type of a particular argument in one action.

[2] A hash set, table or map is a data structure that associates data with a key or identifier using a hash function.

a separate list stored within the TotalOrderPlan class. This list is the original solution generated by the planner and contains all actions in total order. Also, when running the scheduler, it takes as input the hash set of the plan-actions. The error in the scheduler arises from the hash set; the key used for the hash function is the action name and parameters. Given that each action is identified in the hashset by its name and parameters, then any future actions with the same name and arguments either overwrite the existing action with that key value or are simply ignored as one already exists. In short, multiple occurrences of the same action are removed from the plan due to the hashset.

Our findings correspond to both the anomolies in BasicTest5 as well as those in AdvancedTest1 and AdvancedTest4. In each of these cases, the missing actions were identical to those that existed earlier in the plan. Hence we are satisfied that the error can be resolved by ignoring the result from the JavaFF scheduler or, perhaps more appropriately, filing a bug-report with the JavaFF developers in our department!

5.5 Summary

To conclude this chapter, we return to the list of goals we presented at the beginning of this chapter. We shall briefly highlight how our research goals have been addressed with the work contained herein. In Chapter 6, we will reflect upon the body of work as a whole, incorporating the work in reactive control from Chapter 3. This will allow us to highlight significant features, contributions, benefits and potential drawbacks of the research in this thesis.

- *Introduce the PDDLWorldBuilder: a java framework designed to parse PDDL problem instances and translate them into environments for testing. Furthermore, we explore how the BruceWorld parser translates particular features of the Nakatomi domain into actual BruceWorld test instances.*

  In section 5.2 we introduce the world builder software designed to translate PDDL problem files into BruceWorld maps. Given that we constructed problem instances from the planning files, there were many aspects of the problem left open to customisation as a result of the level of abstraction. Hence, the world builder software is capable of building BruceWorld levels that vary in physical layout and positioning. This ensures that while each

test instance relies on the problem file, we still create a relatively unique problem for the reactive controllers.

- *Introduce the REAPER architecture, proposing a potential solution to the BruceWorld game. We provide in-depth detail of the three main components of our agent; the Plan Manager, Rule Controller and Controller Library.*

  The REAPER architecture is designed to incorporate our subsumption-layered ANNs introduced in chapter 3 while providing the ability to satisfy distant goals using AP approaches. To encapsulate these within the REAPER agent, we started by creating the PM, a system designed to contain the JavaFF planner. The PM is capable of assimilating a plan-model from the environment to be used for the planning process. Once plans are constructed, the system manages the remaining actions to be executed and ensures model accuracy during execution, forcing a re-plan if necessary. Next we have the CL, an easy to use interface that creates the relevant controller for a given action. Given that we are basing our execution on plan actions, it creates controllers that reflect the actions in the PDDL domain model. The CL constructs subsumption-layered ANNs and grounds them with respect to the current state of the environment. The CL also provides the opportunity to provide scripts for actions if we do not have pre-trained ANN controllers. Lastly, we have the RC that provides a valuable rule base to bridge the gap between the plan-model view and reactive control view. This not only associates PDDL actions with the respective controllers in the CL but also models other issues that are beyond the scope of the plan model such as threatening entities and action preconditions outwith the planner's perspective.

- *Implement a series of tests to assess the effectiveness of the approach and highlight any areas of significance for discussion.*

  In the penultimate section of this chapter, we have provided a series of tests to assess the functionality of the architecture. This begins by extensive tests in varied problems to ensure the ability to execute plan actions directly, add bridge actions for unresolved preconditions and address hostile entities. Next we approached more complex, puzzle problems that required large numbers of actions in one plan. Fortunately, the agent rose to the challenge. Further

experimentation tested the agent's resilience to threats and discovered that whilst capable, the agent is not infallible. For robust execution, we require further work in creating domain-specific reactive controllers. Moreover, the agent, as expected, is constrained by the capabilities of the planner and the environment. Therefore, if there is no solution to the existing task, then the agent will fail to complete a problem. However, we discovered that a REAPER-lite approach, stripped of model checking and threat detection, performs significantly poorer on average.

5.5.1 Closing Remarks

In this chapter we have introduced the agent architecture which we set out to achieve in Chapter 1. This was achieved by building individual components that managed the JavaFF planner, our evolved reactive controllers from Chapter 3 and a series of prolog rule bases to act as the glue between these features. The extensive series of tests that followed indicated that the agent is capable of solving large problems outwith the scope of individual reactive control and can accommodate for changes in the problem on the fly.

In our closing chapter, we take stock of the research reported in this thesis and reflect on how we have achieved the goals set out in Chapter 1. We discuss how the work found both here and Chapter 3 provides a unique approach to introduce deliberative control and highlight the benefits, drawbacks and contributions of the research.

Chapter 6

Discussion & Conclusion

> There is something fascinating about science. One gets such wholesale returns of conjecture out of such a trifling investment of fact.
>
> Mark Twain

In this final chapter, we reflect on our research presented in this thesis. In reaching this point we have already discussed the results of the learning procedure in Chapter 3, however, we have only examined the benefits and drawbacks of the learning approach on its own and not in the context of our overall work. Firstly, we shall examine the design decisions made for our agent and the overall results, highlighting both the benefits and drawbacks of our approach and the contributions it provides. Furthermore, there exists potential for continued work in this project and we wish to highlight those areas that are worthy of exploring in addition to new ideas that have arisen from our time in this project.

6.1 Reflecting on our Research and Results

> The outcome of any serious research can only be to make two questions grow where only one grew before.
>
> Thorstein Veblen

The work presented in this thesis shows an interesting and unique method for creating a high-level of autonomy and control for agents in games. In turn, this work addresses our research goals established at the beginning of this thesis. The executive introduces a hybrid architecture that creates a unique synergy between neural network-driven reactive controllers with the deliberative capabilities of a classical planning system. This achieves long-term deliberation and problem solving at an abstract level whilst the reactive control addresses local, real-time information. We take this opportunity to reflect on our research goals and questions originally highlighted in Chapter 1:

- *To create an effective agent controller that incorporates deliberative reasoning based on a discrete, deterministic model in the context of a continuous and dynamic game environment.*

  The REAPER agent introduced in Chapter 5 is a hybrid architecture that is driven primarily by the JavaFF planner. JavaFF is responsible for the creation of plans and monitoring of their execution in our BruceWorld game. As shown in Chapter 4, BruceWorld is a continuous, dynamic environment with imperfect information that is beyond the capabilities of typical reactive control. Through the use of the Nakatomi PDDL domain model, we can model this problem as a discrete, deterministic problem with perfect information and create solutions to problem instances. This relies on the other components of the system to resolve any issues that are not modelled at the planner's level of abstraction during execution. This resulting controller could potentially be used for creating agents that can interact with the user/player in a game environment or simulation.

- *To explore a new evolution-based learning methodology that allows us to train our simple neural networks to solve more challenging problems.*

In Chapter 3 we introduced our variant of the layered evolution approach created by Togelius. This layered approach to evolutionary learning combined with Brooks' subsumption architecture, creates an incremental, modular, yet simple approach for training a collection of Neural Networks for solving a class of problems. Such a modular approach allows a 'plug'n'play' method, where we can build a library of different solutions to each controller and insert them into the subsumption architecture for execution.

- *To integrate a classical planner with a collection of neural networks to achieve deliberation and reactive reasoning.*

  The Plan Manager component of REAPER integrates the JavaFF and is driven by the actions derived by the planner. To execute these plans, we associate our reactive neural network controllers with particular plan actions. By delegating plan actions to the neural nets, we achieve a high level of autonomy by then interfacing with the environment through real-time reactive control.

- *To build an agent controller that is effective at solving a range of complex problems while comprised of relatively simple and robust components.*

  The design of the reactive controllers in Chapter 3 and the complete agent architecture in Chapter 5 relies on a series of simple, yet robust components. Our neural network designs are small and robust, while REAPER is reliant on a renowned planning system coupled with a clear and understandable plan-domain model. In order to glue these features together, we rely on an expressive but straightforward rule base defined in the Prolog programming language.

Of course in order to achieve these goals, there were a series of challenges that needed to be addressed. Now that the work has been completed, it is now our belief we can address the questions these challenges presented:

- *How can we scale-up our reactive agent controllers without compromising behaviours established in our previous work?*

  Our evolutionary algorithm for training subsumption layered reactive controllers scales our behaviours for more complex and challenging problems

without the risk of previously established behaviours being undermined or 'forgotten'. As was indicated in Togelius [2004], the layered evolution approach combines the benefits of the incremental and modular approach. Our approach can add new features to the environment, then adapt the agent behaviour by including complementary controllers within the hierarchy. This prevents any circumstance where the previously tried and tested behaviours are removed entirely, though they may be superseded by higher-ranked layers. The results shown in Section 3.5 illustrate that this approach works well for our EvoTanks game and statistically outperforms more traditional evolution-based approaches.

- *How can the reactive neural network controllers represent the planner's actions and how can we associate them?*

  The Rule Controller shown in Chapter 5 provides a clean interface between the PDDL model and the reactive controllers provided for the domain. This relies on the assumption we have built reactive controllers that achieve the effects of a PDDL action. The clauses in the RC allow the designer to define which controller(s) will be used in the event of specific plan-actions. This is applied through the Controller Library by grounding the selected reactive controller with respect to the context of the plan action. Once this is achieved, execution can commence.

- *How do we compensate for issues that cannot be modelled in a classical planning system that are also beyond the capabilities of our reactive controllers?*

  The threat rule base found in the Rule Controller allows us to model circumstances that are beyond the planner's scope such as enemy agents. Furthermore, the controller rule base also provides clauses that ensure other action pre-conditions beneath the plan-model's level of granularity are satisfied prior to the execution.

6.1.1 Subsuming Neural Networks

At the end of Chapter 3 we discussed the results from our experiments in Sections 3.4 and 3.5 respectively. Here we briefly highlight the important conclusions made from this phase of the research.

The first, and perhaps most important point, is that the methodology proved to be very effective for the problems assigned to it. The layered training approach

provides a simple tool to build agent behaviours atop one another. Furthermore, the approach carries some novelty in streamlining the learning process, as seen from observing how agents learned to compensate for additional complexity in the problem. In these circumstances, the fitness criteria would not explicitly state addressing these issues as a requirement, instead the performance was dictated by both the basic task to perform and the environment that it was initialised in. This is achieved by exploiting the EA search process, since we need only define the basic constraints of the problem, within reason, and allow the search to do its job. In select scenarios we explored, the additional layers in the subsumption would learn the best form of behaviour relative to the problem irrespective of the simple fitness criteria. This underlying assumption provides a fantastic opportunity for modular development, allowing us to focus - as Brooks initially suggested - on the individual facets of our agents behaviour.

6.1.2 REAPER Architecture

Our series of tests conducted in Section 5.4 were designed to assess the overall performance of our agent and the limitations of functionality. The results from these tests suggest we can rely on our agent to solve reasonably complex problems up to a reasonable level of threat and uncertainty. Of course at present we have only tests within the BruceWorld game to verify performance, but we are confident that the architecture is designed in a manner that can then be applied to different domains. This is a matter that will be discussed further thoughout this chapter.

Reactive, Goal-Driven Agents

The REAPER architecture creates versatile, reactive, goal-driven agents. This agent is capable of long-term deliberation through state-space search to devise a plan of action, further deliberating these plans to low-level reactive controllers that can affect the environment. Furthermore, the agent is able to monitor progress on plans it executes and can rectify issues in real time. Interestingly, the execution monitor introduces a new form of reactive behaviour, where the system responds to the plan-model by attempting rectification through a re-modelling and re-planning process. However, changes to the environment can occur on a level that is beneath the planner's notice. Hence, we have two further levels of modelling present. The Rule Conroller is designed to suggest a course of action should changes arise or where our actual performance is not yet synchronised with the plan-model

(e.g. fixing assumptions of physical positioning in the world prior to an action occurring). Finally, the controllers in the library provide not only the means to effect the actions in the plan but can also resolve other issues in execution, since they have modelled the local environment with respect to their goal. The use of ANNs provides robust reactive control that ensures an action can be completed irrespective of the characteristics of a given scenario provided it adheres to the rules and constraints of the domain. This is due to the ANNs's ability to generalise across a set of scenarios once it has passed through the evolutionary learning process. Furthermore, the use of the subsumption-layered ANNs allows us to introduce new environmental challenges and create behaviours to resolve them. This means we can create controllers that accommodate for each plan-action's basic requirements, allowing addition of extra behaviours to the hierarchy to address context-specific challenges.

One interesting observation to be made from the architecture is the series of assumptions that carry down through the layers of control that ensure robust performance. These assumptions are carried down through each layer of control, allowing them to operate without the need for consistency checks between the individual components. The planning level takes on board all possible information at that level of abstraction and providing the user has developed an appropriate domain model, should encapsulate all potential goals we wish to achieve in our environment. Once we have ensured the model is valid prior to execution, then the assumption that it is valid is carried all the way down through the system. Given the small time-span between plan action being verified to execution actually taking place, this is a safe assumption to make. In the Rule Controller, we also take steps to ensure that the action is safe to execute. However, the clauses do not run checks against the state model. Rather, we assume that the Plan Manager has verified the action is valid and continues as normal. Finally, the controllers in the Controller Library do not consider other aspects of the environment beyond the scope of their functionality; they have no concept or knowledge of the overall plan in progress. We assume at this juncture that there cannot be any issues with our choice of action. Furthermore, these controllers are only designed for one immediate task, since we rely on other controllers in the library to carry out any other required actions. Conversely, the implementation of a planning system using an abstract model also relies heavily on the reactive controllers being able to handle all sorts of immediate threats or issues only visible from that level of scope, such as the layout of environments to obstacles and aggressive opponents.

Design

When approaching the architecture design we were keen to ensure it was compact, robust and simple. When combining different methodologies and practices there is always the concern the system will become entangled in implementation issues and/or become domain specific due to modelling concerns. The architectures' system design ensures that while we may require particular subclasses to define domain-specific instruction, the overall framework remains a distinct system. Furthermore, each of the three main components exists relatively isolated from one another. Although there are certainly domain-specific dependencies between each unit during the actual execution of the agent, significant changes made to each element will have no lasting impact. When we look specifically at how each system operates, we can observe that the provided interfaces turn these components into black-boxes. This means we can manipulate the internal workings of each component without fear of breaking the overall structure. Whilst we are content using the JavaFF planner, given our experience with Java and the open-endedness of the software, there is sufficient flexibility to allow for other planners to be installed into the PM. The greatest advantage this provides will be within the CL where - as we have already seen - a provision for different controllers that can either be scripted, ANNs or otherwise. This provides an opportunity for transference of the executive to other problem domains.

Arguably one of the strongest benefits of our design decisions is it would not be difficult to transfer the agent for use in different problem domains. However, it is important to highlight that this does require a certain amount of domain expertise. This capability is required due to the need for a PDDL domain file, problem definitions, rule clauses for the RC and programmed or trained controllers for the CL. It is entirely feasible that we can apply this executive to more challenging problem domains, since the essential structure of the system will remain intact. If we were to create a build of the system that denies the user access to the architecture itself but allows the creation of domain specific models and controllers, then this software could be made more broadly available.

Reflecting on Related Work

If we take a moment to reflect on the related work documented in Chapter 2, we can see how our direction and potential research impact compares to others.

Brooks & Subsumption Consider the work of Brooks and his subsumption architecture, our layered approach has embraced his vision set out in Brooks [1987]: to create a hierarchical control system that operates underneath a control planner. Furthermore, subsumption was invented to avoid the need for sub-level planning and execution systems, a principle we have adopted. While our original intention was to find a means to integrate EA-trained ANNs with classical planning systems, our final product reflects Brooks' vision; a point only now realised in hindsight. The use of subsumption and the ANN controllers also led to other observations, namely we have realised *the world really is a rather good model of itself.* Despite their limitations and restrictive scope when dealing with larger problems, neural networks - due to their nature as function approximating perceptrons - are a fantastic mechanism for feeding relevant data from the world *in real time* and generating appropriate responses almost immediately. This helps navigate the issue highlighted in Moravec [1983] that secondary models are required for the sensor data. The neural networks model the sensor data with respect to a particular task. Providing sufficient training has taken place, the network should be able to generalise responses across large sets of possible states. Viewed from another perspective, we return to the comments made in Bellingham et al. [1990], specifically, the shortcomings of applied subsumption and why the authors strived to apply a planner-based control system. The authors expressed their concern in providing too much functionality as a layered system with simple conflict resolution dictated by the hierarchical priority. In our work we have circumvented these potential problems by presenting to the user the opportunity to create multiple individual controllers representing the individual plan actions. These controllers are relatively expressive and robust whilst relying on the subsumption paradigm. Hence, we have exploited many of the key benefits that Brooks conveyed while avoiding the trappings highlighted in Bellingham et al. [1990].

Williams & Agent Architectures Finally, we return to arguments by Brian Williams and his colleagues in Robertson et al. [2006]. In Chapter 2, the authors highlighted several issues that can arise when constructing agent architecutre systems. We recap these points and highlight how our executive incorporates measures for these issues below:

- *Often assumptions made by control software prove to be false during execution.*

As shown in Chapter 5, the PM retains a copy of the remaining actions to be executed from the plan, and the current state based on the progression of the said plan. If any inaccuracies exist, we redefine the problem instance based on the game state and run the planner again to ensure our assumptions of the world are once again accurate, before continuing to satisfy our assigned goals. Furthermore, the planning level only carries assumptions of general problem structure and relationships between entities. Should these change, within the constraints of the domain file, we can easily rectify this. Moreover, any assumptions of successful execution rely on the neural network controllers. These reactive controllers model the world directly in real time and are designed using the subsumption to provide flexible, generalised and robust behaviours to execute an action. This compensates for many execution issues created by different problem scenarios, which in-turn allows the planner to retain the assumption that any selected action can be executed successfully.

- *Software may be attacked by a hostile agent, seeking to interrupt its execution or may simply be reacting to its presence.*

  We embraced this concept rather literally by including hostile entities in BruceWorld. These entities exist solely to interupt our agent's execution. We have compensated for this on two levels, firstly with the RC providing supplementary, expert-crafted rules that allow the user to define the behaviour should one of these circumstances arise. Furthermore, the ANN subsumption-controllers can be designed to either directly address these threats, with an RC clause stating this controller should be applied, or add extra functionality to a given behaviour to react (evasively) to these hostile elements whilst focussing on the actual task at hand.

- *Continued changes to software often introduces compatibility issues between components.*

  Each component within the REAPER architecture exists in isolation as an individual component. This gives a certain degree of flexibility, allowing us to modify individual components; such as the planner within the PM and the controllers existing within the CL, without fear of any incompatibility. This is providing we adhere to the principal functionality of each component and any domain-dependencies (e.g. required controllers in the CL). Given

this flexibility, the substituting of different planners can be done to assess performance and explore new modelling approaches. Perhaps, more intriguing is the ability to introduce new reactive controllers for solving individual actions or maintaining a collection of unique yet redundant controllers for each feat.

6.2 Benefits, Drawbacks and Contributions

> The men who create power make an indispensable contribution to the Nation's greatness, but the men who question power make a contribution just as indispensable, especially when that questioning is disinterested, for they determine whether we use power or power uses us.
>
> John F. Kennedy

To conclude discussing our research, we take a moment to highlight, and potentially re-iterate, the benefits, drawbacks and contributions of our work.

6.2.1 Benefits

Layered Autonomy: One of the strongest aspects of REAPER is that once the architecture is provided with all the relevant controllers and domain models, it requires no interaction from the designer and can execute any plan competently. It is for all intents and purposes a simple, expressive and completely autonomous agent platform.

Adapting to New Goals/Problems: Once the controllers have been clearly defined and the plan-model validated, the agent can then solve a variety of different problems providing the complexity is not beyond the planner's present ability to solve. Even so, there is still a significant range in the problem maps we can build in BruceWorld for this problem. Besides, if new

issues arise during execution - as shown from our uncertainty experiments - the agent can resolve these provided the rules of the world and model are maintained.

Customisation: There are a myriad of opportunities within the REAPER framework for extension and customisation. The interfaces between each component provide a black-box view of each component. This allows us to modify them as we see fit, provided the desired output is still generated, implying we can explore the development and application of new REAPER configurations within the framework. This too allows us to dictate which controllers are utilised for specific actions and how to construct them independently from the architecture.

Control and Understanding: The behaviour of the agents and their action selection is dictated by three main components of the architecture; the plan-model, the rule base and the collection of controllers. Consequently, any action taken by the agent can immediately be recognised as either being caused by the planner or the rule base. Moreover, all actions are executed with controllers the user has either crafted by hand or trained. In short this provides us with clear control over the agent's behaviour in any situation while allowing it to retain its autonomy. This allows for a clear cut understanding of what the agent will do in any given situation. No situation should arise where the designer must contemplate why their agent has made a particular action.

Simple Planning with Execution: Unlike many other plan monitoring and execution systems, ours is simple yet robust. We have relied on using small, competent components and combined them to create an architecture that, while not ideal for space or underwater exploration, would be suitable for game or simulator software. The ability to use planning - a powerful tool for complex problem solving - and subsequently monitor the execution of a resulting plan with such a reduced setup is definitely ripe with potential.

Reusable and Stackable Controllers: The use of the controller library and subsumption hierarchy allowed us to create a catalogue of unique controllers for problems, or even subproblems, and combine them in useful and challenging ways. This permits controllers to be reusable and interchangeable, yet can also be placed atop one another to create new useful behaviours. In

time, we could develop vast numbers of redundant controllers that provide a specific functionality (e.g. navigation controllers) that could be transferred to new domains.

6.2.2 Drawbacks

Naturally, there are some drawbacks to our approach and it's important we address these issues and allude to how they may be avoided in future iterations.

Expert-Driven System: Every level of the architecture requires input from the designer and is reliant on domain-specific information. This is most apparent at the planning and rule levels, where we must create a series of domain specific rules and an entire planning-domain model in order for the agent to even consider execution. The designer must have a clear understanding of all elements in a given problem domain and the relationships that they have with one another. One needs to understand the effect an agent can have on an entity and how that can be modelled as a useful action. Furthermore they must also understand more specific conditions outwith the planner's level of control that must be defined as clauses in the RC. Potentially these could be avoided through the use of a rule learning algorithm that could ascertain the conventions of the domain. There are a multitude of existing rule learning systems (including evolutionary rule learning) that, with sufficient modification may prove suitable for this purpose.

Repeated Trial and Error: For someone new to the architecture, it may take some time to develop specific rules and controllers that create the user's desired performance. While we can easily edit and swap elements in these components, a lot of trial and error may arise prior to the agent behaving in the manner we intended.

Controller Development and Training: During the period of our research, we developed REAPER as a natural progression from our subsumption-based reactive control. As such, it was integrated into the architecture, and for the sake of time and ease we used controllers from the EvoTanks game. For a new and different domain it may take some time to create and train these agent controllers, the benefit being that once created they can be used at a later date provided no extra training or modification is required. However, if

a supervised learning method is employed, a significant amount of training data may need to be amassed prior to development.

Time Taken to Construct Plans: As we saw in the advanced tests in Section 5.4.3, the time taken to construct a plan for the most complex instances could reach several seconds in the worst case. This length of time could have a serious impact on performance in many dynamic games where time spent 'thinking' about the problem, thus prohibiting reaction taking place, could result in failure.

Inability to Solve All Prescribed Problems: As was briefly discussed in Section 5.4.3, we were unable to provide solutions to all problems that we explored as a result of the RPG in the JavaFF planner. This could have an impact on other problem domains or creating larger, more complex instances.

6.2.3 Contributions

Finally, our research must make a significant contribution to the existing work in the field. Here we highlight the features and characteristics of our work that constitute its importance.

Classical Planning for Games: This research is - to the best of our knowledge - one of the first attempts to apply classical planning methodologies to a sequential, continuous game environment. Our results from experimentation are, if anything, a proof of concept; that we can apply planning technology to create intelligent, deliberative agents for games. One of the great challenges ahead now lies in exploring how planners can be adapted to these environments more effectively, to ensure that planning can be achieved effectively and quickly without constraining computational resources.

Planning & Neural Networks: Another key contribution is the merger of a classical planner with artificial neural networks. While a merger of planning and execution is far from novel, execution is often handled by a scripted controller. Furthermore, as described in Chapter 1, agents in video games are typically constructed using hardcoded programs or are reactive in nature as a result of a machine learning methodology being applied. As a result, adopting ANN control to realise the actions of the PDDL is an interesting application of established reactive control methodologies. Furthermore, from

a reactive control perspective, we provide an executive that allows us to apply the ANN/subsumption driven control in environments where deliberation is essential, while retaining the advantages the reactive approach provides.

Autonomy in a Plan-Driven Framework: While plan execution and monitoring is far from novel, it is rare for any plan-driven agent to act completely autonomously given the safety-critical situations they are often deployed in. This work provides evidence that we can create customisable, deliberative and reactive autonomy in a plan-driven architecture.

Fast & Cheap Intelligent Agents: Our agent is reliant on several layers of control and deliberation - despite this, processing time and resource consumption is low. The use of reactive control, courtesy of the ANNs gives us a fast, cheap and robust solution to our execution problems. Furthermore, as shown in Chapter 5, the use of the rule clauses for further deliberation is a very fast and clean process. At present our only concern is the planning system; while it is often cheap to solve small problems, there are issues with initialisation overheads and tackling large, complex problem instances.

6.3 Further Work

> Next time, baby.
>
> Jim Rhodes, Iron Man

As is often the case with any thesis, the work submitted raised more questions than it answers. It is necessary therefore to provide ideas for further work that could be explored by either the author or a fellow student. In this instance, there is still a sizeable amount of work available in extending the system to cope with new features or enhance existing functionality. Following is a short summary detailing plausible expansions to the executive.

Planner Expansion One area we would like to explore is the challenge in adding more complexity at the planning level of the architecture. Throughout this research we used a classical planning system due to elegance and simplicity of a discrete, deterministic approach. Now that we have applied such a model, we would be interested in exploring whether we can add new features such as scheduling, metrics or temporal constraints. These could be of use for an agent in

a game reliant on resource acquisition and management, in addition to puzzles or tasks that must be completed within specified time frames, or through the use of concurrent actions. Given the nature of the PM it would not be a significant undertaking to swap-out JavaFF and replace it with another planner - provided sufficient time is taken to integrate it into the PM. The real challenge exists in the execution flow of the system and the added constraints it may carry through to the RC. For example, consider dealing with hostile entities once again while running on a short amount of time and/or limited resources that may be consumed when dealing with the said threat. Are we confident that applying these actions without the planner's approval will not jeopardise plan completion?

The integration of a different planning system we hope would also circumvent the problems we've had in creating complex test problems that are simply beyond JavaFF's capabilities. While FF proved capable for the most part, it is not suited to particular domains and problems as a result of the RPG heuristic (Hoffmann and Nebel [2001]). It would be in our best interests to explore different planning approaches; such as the local-search inspired LPG (Gerevini and Serina [2002]) or through the use of SG-Plan, a planner that focusses on subgoal resolution (Chen et al. [2004]). Planners are also applicable in temporal and metric driven domains, which could be applied to the modelling process.

Prior to submission, we began building a new version of the REAPER executive that could incorporate concurrent actions. This new version carries an updated PM that allows for a series of individual, concurrent plans that arise by scheduling the existing plan. Furthermore, we have expanded the executive to allow for multiple actions to be executed in one action step. However, integrating the scheduling process with the JavaFF planner has proven to be a time consuming process, given that the scheduler in JavaFF needs to be corrected, as indicated in Section 5.4.6. As a direct result of the additional corrections, this extension was not completed in time for our submission, but we hope to complete it in the near future.

Agent Modelling One noted issue we faced throughout this research is the lack of agent modelling in classical planning systems. It would be useful if we could devise a planning system which is able to model the behaviour of other agents/entities as part of the state-space exploration. In fact, what we would prefer is the opportunity to create an agent modelling language that can then be applied to a classical planner. This would provide us with a planning perspective that can observe the effect other agents have on the world, whilst retaining the

elegance of a discrete, deterministic PDDL-modelled approach. This is not to say that agents have never been modelled in planning domains before now. Multi-agent (MA) planning is a research field intent on exploring the possibilities of numerous agents interacting with one another in a planning problem. However, the majority of MA planning assumes that all agents in the problem are altruistic, cooperative and have similar goals, e.g. drivers in a taxi company. As a result, most MA planners construct individual plans for each agent, followed by an extensive scheduling process that determines which actions are executed when and by whom (De Weerdt et al. [2005]). Meanwhile, there has also been work in adversarial planning; where the other agent in the domain is our opponent (Willmott et al. [1998], Applegate et al. [1990]). However, this differs from our intended research, as it has focussed primarily on game-tree style problems where the actions take place in sequence and the two agents are assumed to be 'pure' adversaries, i.e. a reward for one agent is penalty for the other. What we are interested in is the creation of a logic or calculus that can allow us to observe two forms of engagement; *state-triggered* and *action-triggered* actions shown in Figure 6.1. The former occurs in a circumstance where an opponent carries out an action in response to a state our agent has created. Whereas the latter is an action an opponent may carry out having observed our agent execute a specific deed. In either circumstance these may be actions we wish to counteract or exploit for our own benefit. A simple example of this can be found in Real-time Strategy (RTS) games, where the player is often concerned with the acquisition and consumption of resources littered throughout the map, in order to achieve goals. In an RTS domain, the player will typically have agents whose sole purpose is to seek out, harvest and retrieve resources at the base of operations. Using our proposed agent-modelling language, an opposing player may recognise changes in the world state as a result of our player's actions. This may lead to state-triggered actions to counter our strategy; such as attacking our strongholds to potentially destroy our resource refineries, or at least cause significant damage to them, forcing us to spend these resources in maintenance and repairs. Furthermore, if the agent may commit an action-triggered behaviour upon realising we are attempting to mine a particular resource deposit, then it may be able to eliminate our harvesters before any resources are retrieved.

Goal Creation Another interesting area for expansion is the notion of goal-creation. At present we assign the goal that the agent must achieve, and while this

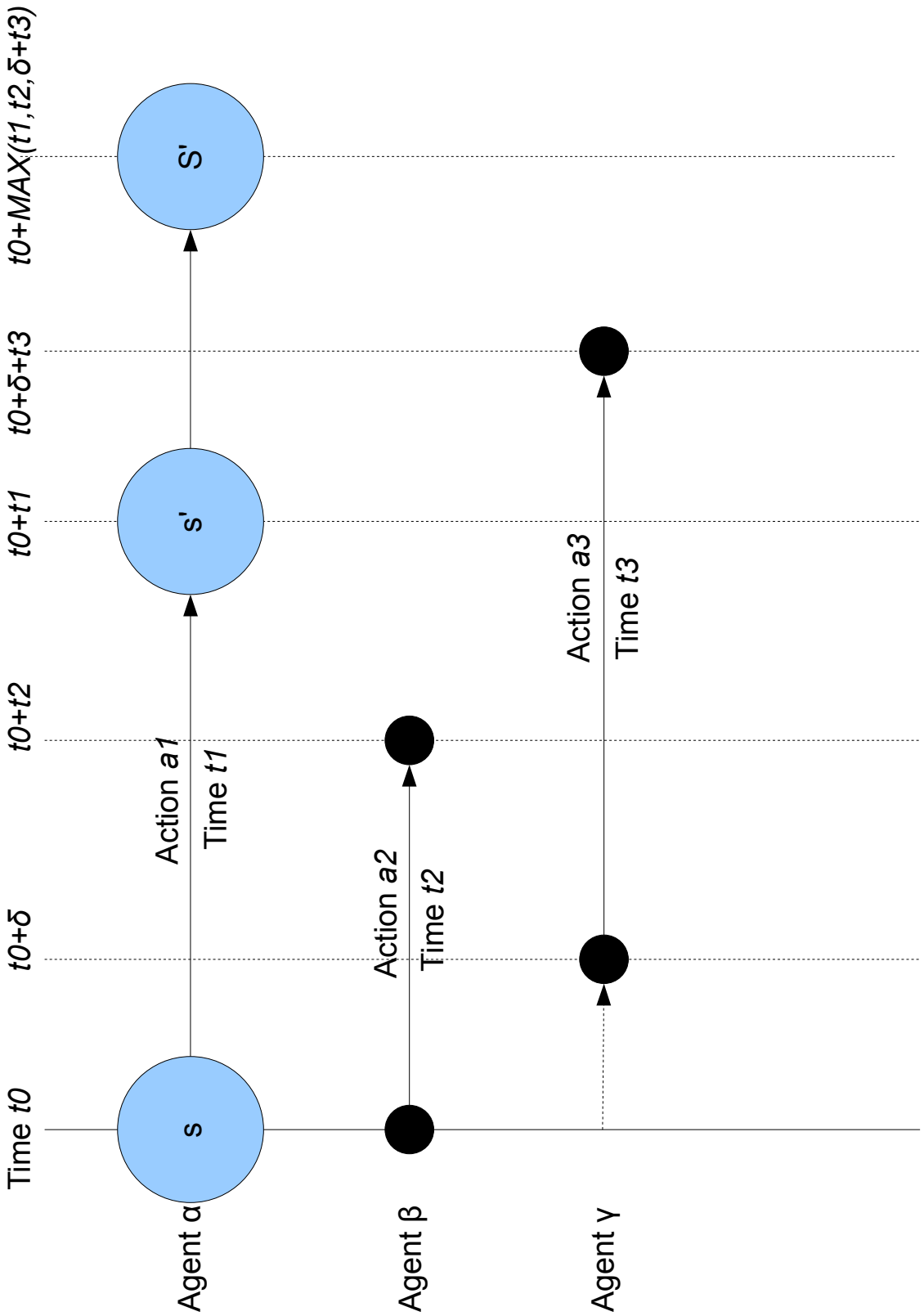


Figure 6.1: A diagram that highlights the possibility of state and action-triggered actions by an opposing agent. This could potentially be modelled within an agent calculus or language and integrated into our classical planning approach.

was suitable for our purposes we would be interested in having these goals generated by the agent itself. Research in real-time goal formulation found in Coddington [2007b] and Coddington [2007a] explores the concept of motivations instilled within an robot-agent. These motivations are instilled by the agents 'belief system' that dictates what is an important goal to satisfy, such as recharging batteries or examining areas of interest. This could help introduce supplementary actions for our agent to consider its own mortality in a hostile situation or even carry out more interesting behaviours as it explores possibilities beyond our original plan-driven mandate.

Automated Rule Generation Finally, the clauses found within the RC are at present hand-coded and expert driven. While this was preferable given our time constraints, it would be of interest to explore how to automate this process. Evolutionary rule generation, such as the work found in Bedingfield and Smith [2003], are designed to take the traditional EA concept and apply it to classification problems. Work in this area could allow for the creation of rules specifically for a given domain. The use of an evolutionary search process could potentially discover new rules that are not only useful but never considered by the designer. Furthermore, the use of random or evolved problem generation could help to create test cases that are beyond those that a human would initially consider, thus preventing the exploitation of holes in the behaviour that the designer did not consider.

6.4 Conclusion

> People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant.
>
> Helen Keller

To conclude, in this thesis we have explored the integration of reactive and deliberative control into a sole AI agent. This has arisen from our intention to expand previous research in reactive control using evolutionary algorithms to train artificial neural networks for small yet challenging game environments. In our previous research we explored the benefits and limitations of the reactive approach and considered the integration of abstract decision making to circumvent the lack of internal state that such function approximators are denied. Hence, our mandate was to explore how far we could push these reactive controllers for such problems, followed by investigating the capabilities of automated planning - a proven deliberative methodology - and explore the feasibility of combining these approaches into an agent framework.

Inspired by the work of Togelius, we explored the approach of layered evolution, where we combine the focus-shifting learning of incremental evolution with the de-centralised approach of modular evolution and organise it within a hierarchical subsumption structure. Experimentation in this learning philosophy was conducted in the EvoTanks environment; a small adversarial domain where tank-based agents faced a series of challenges designed by the user. Our modified layered evolution process proved highly effective, with results indicating it would perform better than traditional approaches. With this phase of work completed, we felt that the capabilities of our reactive control had reached a satisfactory level. Now we could construct effective and robust behaviours that deal with goal-specific tasks whilst compensating for dynamic local stimuli. Should an initial inability to handle these stimuli arise, then we could explore the creation of add-on controllers to facilitate it. However, while these individual neural networks could solve a series of tasks individually, they could not resolve larger problems given our need to shift between different controllers for specific circumstances.

To address this issue, we introduced the REAPER architecture: an executive that managed a library of individual hand-coded or evolved controllers. Controller selection was then dictated by a top-level deliberation process that used the

JavaFF classical planning system. By creating an abstract representation of the problem domain, we could reason how to solve goals by generating plans that could then be executed sequentially using our library of behaviours. This was enhanced with a Prolog-driven rule base system that could address other aspects of the domain beyond either the planner or the evolved neural networks. The resulting product is a goal-driven, yet reactive architecture that achieves a synergy between the abstract deliberation of planning systems, with the reactive control of neural networks. Furthermore, it also creates an agent that is dependent on reactive control to interface with and act within an environment, but also has an internal state that can reason about distant goals and how to achieve them, preventing the agent from being trapped in dead-ends.

We test this framework in BruceWorld; a problem domain of our creation. Extensive testing showed the system is relatively fast at constructing solutions for challenging environments and can delegrate actions correctly to the appropriate reactive controllers. Further experimentation assessed that the agent is capable of compensating for issues outwith the planner's perspective or changes made to the world during execution. However, performance against hostile elements is constrained by the capabilities of the reactive controllers provided. Lastly, we highlighted that a stripped-down version of this architecture that is not reliant on validation and threat-detection performs poorer on average.

In closing, our research has introduced a versatile, robust and interesting new approach for agent controllers that could potentially be applied to game-based agents.

Bibliography

C. Applegate, C. Elsaesser, and J. Sanborn. An architecture for adversarial planning. *IEEE Transactions on Systems, Man and Cybernetics*, 20(1):186–194, 1990.

J.E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application table of contents*, pages 14–21. L. Erlbaum Associates Inc. Hillsdale, NJ, USA, 1987.

S.E. Bedingfield and K.A. Smith. Evolutionary Rule Generation and its Application to Credit Scoring. *Soft Computing in Measurement and Information Acquisition*, pages 262–276, 2003.

J.G. Bellingham, T.R. Consi, R.M. Beaton, and W. Hall. Keeping layered control simple. In *Autonomous Underwater Vehicle Technology, 1990. AUV'90., Proceedings of the (1990) Symposium on*, pages 3–8, 1990.

D. Bernard, G. Dorais, E. Gamble, B. Kanefsky, J. Kurien, G.K. Man, W. Millar, N. Muscettola, P. Nayak, and K. Rajan. Spacecraft autonomy flight experience: The DS1 Remote Agent experiment. In *Proceedings of the AIAA 1999, Albuquerque, NM*. Citeseer, 1999.

D.E. Bernard, EB Gamble, N.F. Rouquette, B. Smith, Y.W. Tung, N. Muscettola, G.A. Dorias, B. Kanefsky, J. Kurien, and W. Millar. Remote Agent Experiment DS1 Technology Validation Report. *Ames Research Center and JPL*, 2000.

H.G Beyer. Evolutionary algorithms in noisy environments: Theoretical issues and guidelines for practice. In *Computer Methods in Applied Mechanics and Engineering*, pages 239–267, 1998.

H.G Beyer and H.P Schwefel. Evolution strategies –a comprehensive introduction. *Natural Computing: an international journal*, 1(1):3–52, 2002.

C.M. Bishop. *Neural networks for pattern recognition*. Oxford Univ Press, 2005.

A.L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

B. Bonet, G. Loerincs, and H. Geffner. A robust and fast action selection mechanism for planning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 714–719, 1997.

R.A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, 1986.

R.A. Brooks. Planning is Just a Way of Avoiding Figuring Out What To Do Next. *MIT Artificial Intelligence Laboratory Working Paper 303*, 1987.

R.A. Brooks. Elephants Don't Play Chess. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back*, pages 3–15, 1991.

R.A. Brooks. Intelligence without Representation. *Foundations of Artificial Intelligence, MIT Press*, pages 139–159, 1992.

R.A. Brooks and A.M. Flynn. Fast, cheap, and out of control: a robot invasion of the solar system. *Journal of the British Interplanetary Society*, 42(10):478–485, 1989.

T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

R. Calabretta, S. Nolfi, D. Parisi, and G.P. Wagner. Duplication of Modules Facilitates the Evolution of Functional Specialization. *Artificial Life*, 6(1):69–84, 2000.

M. Campbell, A.J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2): 57–83, 2002.

Y. Chen, C.W. Hsu, and B.W. Wah. SGPlan: Subgoal partitioning and resolution in planning. *Edelkamp et al.(Edelkamp, Hoffmann, Littman, & Younes, 2004)*, 2004.

A. M. Coddington. Integrating motivations with planning. *Proceedings of the 6th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS07)*, pages 850–852, 2007a.

A. M. Coddington. Motivations as a meta-level component for constraining goal generation. *Proceedings of the First International Workshop on Metareasoning in Agent-Based Systems*, pages 16–30, 2007b.

A. Coles, M. Fox, D. Long, and A. Smith. Teaching Forward-Chaining Planning with Java FF. In *Colloquium on AI Education, Twenty-Third AAAI Conference on Artificial Intelligence*, pages 17–22, 2008.

A. I. Coles, M. Fox, K. Halsey, D. Long, and A. J. Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1–44, January 2009.

K.A. De Jong. *An analysis of the behavior of a class of genetic adaptive systems*. University Microfilms International, 1975.

M. De Weerdt, A. Ter Mors, and C. Witteveen. Multi-agent planning: An introduction to planning and coordination. *Handouts of the European Agent Summer School*, 2005.

M.B. Do and S. Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artificial Intelligence*, 132(2):151–182, 2001.

S. Edelkamp and J. Hoffmann. PDDL2. 2: The language for the classical part of the 4th international planning competition. *4th International Planning Competition (IPC'04), at ICAPS'04*, 2004.

R. Effinger, A. Hofmann, and B. Williams. Progress Towards Task-Level Collaboration between Astronauts and their Robotic Assistants. In *'i-SAIRAS 2005' - The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, volume 603 of *ESA Special Publication*, August 2005.

Hendler-J. Erol, K. and D.S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1123–1123, 1995.

R. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3/4):189–208, 1971.

J. Michael Fitzpatrick and John J. Grefenstette. Genetic algorithms in noisy environments. *Mach. Learn.*, 3(2-3):101–120, 1988.

L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. John Wiley & Sons Inc, 1966.

M. Fox and D. Long. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(2003):61–124, 2003.

M. Fox and D. Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

M. Gallagher and M. Ledwich. Evolving Pac-Man Players: Can We Learn from Raw Input? *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 282–287, 2007.

M. Gallagher and A. Ryan. Learning to play Pac-Man: an evolutionary, rule-based approach. *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, 2003.

M. Gallien, F. Ingrand, and S. Lemai. Robot actions planning and execution control for autonomous exploration rovers. *WS7*, page 33, 2008.

H. Geffner. PDDL 2.1: Representation vs. Computation. *Journal of Artificial Intelligence Research*, 20:139–144, 2003.

A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. In *ICAPS Workshop on Soft Constraints and Preferences in Planning*, 2006.

A. Gerevini and I. Serina. LPG: A planner based on local search for planning graphs with action costs. In *Proc. of the Sixth Int. Conf. on AI Planning and Scheduling*, pages 12–22, 2002.

M. Ghallab, D.S. Nau, and P. Traverso. *Automated Planning: theory and practice*. Morgan Kaufmann Publishers, 2004.

F. Gomez and R. Miikkulainen. Incremental Evolution of Complex General Behavior. *Adaptive Behavior*, 5:317–342, 1997.

L.K. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(10):993–1001, 1990.

E.J. Hastings, R.K. Guha, and K.O. Stanley. Evolving content in the galactic arms race video game. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2009.

S. Haykin. *Neural networks: a comprehensive foundation*. Prentice Hall, 2008.

W.D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D: Nonlinear Phenomena*, 42(1-3):228–234, 1990.

Lee-Urban S. Hoang, H. and H. Muñoz-Avila. Hierarchical plan representations for encoding strategic game ai. In *Proc. Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE-05)*, 2005.

J. Hoffmann. The fast-forward planning system. *AI magazine*, 22(3), 2001.

J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

J.H. Holland. *Adaptation in natural and artificial systems*. MIT Press Cambridge, MA, USA, 1975.

R. Howey, D. Long, and M. Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, 2004.

G. Infantes, F. Ingrand, and M. Ghallab. Learning behaviors models for robot execution control. In *ICAPS*, pages 394–397, 2006.

R. Jensen and Q. Shen. *Computational Intelligence and Feature Selection: Rough and Fuzzy Approaches*. Wiley-IEEE Press, 2008.

H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *International Joint Conference On Artificial Intelligence*, volume 16, pages 318–325. Citeseer, 1999.

J. Laird and M. VanLent. Human-level AI's killer application: Interactive computer games. *AI magazine*, 22(2):15, 2001.

S. Lee-Urban, H. Muñoz-avila, A. Parker, U. Kuter, and D. Nau. Transfer Learning of Hierarchical Task-Network Planning Methods in a Real-Time Strategy Game. In *Proceedings of the ICAPS-07 Workshop on AI Planning and Learning*, 2007.

F.G. Lobo, C.F. Lima, and Z. Michalewicz. *Parameter setting in evolutionary algorithms*. Springer Publishing Company, Incorporated, 2007.

S.M. Lucas. Evolving a neural network location evaluator to play ms. pac-man. *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 203–210, 2005.

S.M. Lucas. Investigating learning rates for evolution and temporal difference learning. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 1–7, 2008.

S.M. Lucas and J. Togelius. Point-to-point car racing: an initial study of evolution versus temporal difference learning. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2007.

D. McDermott. PDDL2. 1: The Art of the Possible? Commentary on Fox and Long. *Journal of Artificial Intelligence Research*, 20:145–148, 2003.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL-the planning domain definition language. *The AIPS-98 Planning Competition Comitee*, 1998.

B.L. Miller and D.E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Urbana*, 51:61801.

M. Mitchell. *An Introduction to Genetic Algorithms*. Bradford Books, 1996.

H.P. Moravec. The Stanford cart and the CMU rover. *Proceedings of the IEEE*, 71(7):872–884, 1983.

H. Muhlenbein. How genetic algorithms really work: I. mutation and hillclimbing. *Parallel problem solving from nature*, 2:15–25, 1992.

J.P. Muller. The right agent (architecture) to do the right thing. *Lecture notes in computer science*, pages 211–226, 1999.

H. Munoz-Avila and T. Fisher. Strategic Planning for Unreal Tournament Bots. In *AAAI Workshop on Challenges in Game AI*. AAAI Press, 2004.

D. Nau, Y. Cao, A. Lotem, and H. Muftoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973, 1999.

S. Nolfi. Using Emergent Modularity to Develop Control Systems for Mobile Robots. *Adaptive Behavior*, 5(3-4):343, 1997.

J. Orkin. Agent Architecture Considerations for Real-Time Planning in Games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*, 2005.

J. Orkin. Three states and a plan: the AI of FEAR. In *Proceedings of the 2006 Game Developers Conference*, volume 12, pages 13–14, 2006.

G.B. Parker and M. Parker. Evolving Parameters for Xpilot Combat Agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2008.

M. Parker and G.B. Parker. Using a Queue Genetic Algorithm to Evolve Xpilot Control Strategies on a Distributed System. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2006.

M. Parker and G.B. Parker. The evolution of multi-layer neural networks for the control of xpilot agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, 2007.

E.P.D. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning table of contents*, pages 324–332, 1989.

D. Purves, GJ Augustine, D. Fitzpatrick, WC Hall, AS LaMantia, JO McNamara, and SM Williams. *Neuroscience Third Edition*. Sinauer Associates, Inc, 2004.

K. Rajan, F. Py, C. McGann, J. Ryan, T. O'Reilly, T. Maughan, and B. Roman. Onboard Adaptive Control of AUVs using Automated Planning and Execution. *International Symposium on Unmanned Untethered Submersible Technology (UUST)*, 2009.

I. Rechenberg. Cybernetic Solution Path of an Experimental Problem,(Royal Aircraft Establishment Translation No. 1122, BF Toms, Trans.). *Farnsborough Hants: Ministery of Aviation, Royal Aircraft Establishment*, 1965.

I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Frommann-Holzboog, 1973.

P. Robertson and B. Williams. Automatic recovery from software failure. *Communications of the ACM*, 49(3):41–47, 2006.

P. Robertson, R.T. Effinger, and B. Williams. Autonomous robust execution of complex robotic missions. In *Proceedings of the Forty-First IEEE Industry Applications Society Annual Conference*, pages 595–604, 2006.

S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall Englewood Cliffs, 1995.

D.A. Savic, G.A. Walters, and J. Knezevic. Optimal opportunistic maintenance policy using genetic algorithms, 1: formulation. *Journal of Quality in Maintenance Engineering*, 1(2):34–49, 1995.

H.P. Schwefel. Evolutionsstrategie und numerische Optimierung. *Technische Universit'at Berlin, Diss*, 1975.

A.J. Sharkey. On Combining Artificial Neural Nets. *Connection Science*, 8(3): 299–314, 1996.

A.J. Sharkey. *Combining Artificial Neural Nets: Ensemble and Modular Multi-Net Systems*. Springer-Verlag New York, 1999.

C. Shu and D.H. Burn. Artificial neural network ensembles and their application in pooled flood frequency analysis. *Water Resources Research*, 40(9), 2004.

W.M. Spears. Crossover or mutation. *Foundations of genetic algorithms*, 2: 221–238, 1993.

K.O. Stanley and R. Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

K.O. Stanley, B.D. Bryant, and R. Miikkulainen. Real-time neuroevolution in the NERO video game. *IEEE Transactions on Evolutionary Computation*, 9(6): 653–668, 2005.

F. Stulp and M. Beetz. Optimized execution of action chains using learned performance models of abstract actions. In *International Joint Conference on Artificial Intelligence*, volume 19, page 1272, 2005.

R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press Cambridge, MA, 1998.

L.C. Tan. Hybrid competitive-cooperative coevolution of decentralized controller in evotanks. Master's thesis, College of Computer Studies, De La Salle University, Manila, 2008.

G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

T. Thompson. Evotanks II. *Final Year Project Report, Department of Computer and Information Sciences, University of Strathclyde, Glasgow, Scotland*, 2005.

T. Thompson. EvoTanks: Co-Evolutionary Development of Game-Playing Agents. *Masters Dissertation, School of Informatics, University of Edinburgh, Scotland*, 2006.

T. Thompson and J. Levine. Scaling-up Behaviours in EvoTanks: Applying Subsumption Principles to Artificial Neural Networks. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 159–166, 2008.

T. Thompson, J. Levine, and G. Hayes. EvoTanks: Co-Evolutionary Development of Game-Playing Agents. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 328–333, 2007.

T. Thompson, F. Milne, A. Andrew, and J. Levine. Improving Control Through Subsumption in the EvoTanks Domain. In *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 363,370, 2009.

J. Togelius. Evolution of a subsumption architecture neurocontroller. *Journal of Intelligent and Fuzzy Systems*, 15(1):15–20, 2004.

J. Togelius, P. Burrow, and S.M. Lucas. Multi-population competitive co-evolution of car racing controllers. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*. Citeseer, 2007.

J. Togelius, S. Karakovskiy, J. Koutnık, and J. Schmidhuber. Super Mario Evolution. *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG)*, pages 156–161, 2009.

E.W. Weisstein. Np-hard problem. `http://mathworld.wolfram.com/NP-HardProblem`.

B. Whitby. *Artificial Intelligence*. The Rosen Publishing Group, 2003.

A.R. White. *The philosophy of action*. Oxford University Press, 1968.

S. Willmott, J. Richardson, A. Bundy, and J. Levine. An adversarial planning approach to Go. *Lecture Notes in Computer Science*, pages 93–112, 1998.

G.N. Yannakakis, J. Levine, and J. Hallam. Emerging Cooperation With Minimal Effort: Rewarding Over Mimicking. *IEEE Transactions on Evolutionary Computation*, 11(3):382–396, 2007.

X. Yao. Evolutionary artificial neural networks. *International Journal of Neural Systems*, 4(3):203–222, 1993.

C.H. Yong and R. Miikkulainen. Cooperative coevolution of multi-agent systems. *University of Texas at Austin, Austin, TX*, 2001.

Glossary

Agent The name given to the core AI system or program responsible for assimilating information from the local (modelled) environment and acting upon it accordingly.

Artificial Intelligence (AI) The branch of computer science focussed on the study and creation of intelligent, rational systems. These systems perceive the world through reduced, mathematically structured models and maximise utility through intelligent deliberation and action.

Artificial Neural Network (ANN) A non-linear mathematical model that mimics the structure and functional aspects of biological neural networks found in the brain. Each neural network consists of layers of *Neuron*s and *Synapse*s to process and propagate information from inputs to outputs.

Automated Planning (AP) The subdiscipline of Artificial Intelligence that focusses on the realisation of action chains through abstract modelling of problems and systems. The subsequent strategies (i.e. plans) to these problems are optimised with respect to domain specific performance metrics.

Candidate One potential solution in a population of chromosomes being evolved for a specific task.

Congress on Evolutionary Computation (CEC) One of the largest and most prominent conferences within the Evolutionary Computation community organised by the IEEE.

Chromosome The term used for the representation of a solution in a Genetic or Evolutionary Algorithm. This term is used given that, like biological chromosomes, it carries the 'DNA' that builds the candidate solution.

Computational Intelligence and Games (CIG) One of several emerging academic conferences that focus on the application of computational intelligence

techniques to game environments as well as applied research in commercial video game products.

Controller Library (CL) A series of scripted and pre-trained ANNs for execution hidden behind a query interface within the REAPER architecture.

Continuous A continuous environment is factored by dynamics such as time progressing and other agents. This has an effect on state modelling, making individual states harder to identify since they are no longer factored solely by our agent. Nonetheless, a continuous environment can still be modelled discretely.

Constraint Satisfaction Problems (CSP) The expression of a particular task or problem as a series of variables that can carry values within a specified domain. The challenge being to assign values to variables while adhering to a series of constraints that dictate the possible solutions that can be found.

Deterministic An environment can be considered deterministic if by applying an action, it only results in one specific outcome.

Discrete A feature often attributed to environment modelling. If the model is discrete, then we are often dealing with a collection of unique finite states that differ due to specific actions made by an agent.

Deep Space 1 (DS1) The first mission of NASA's new millenium program that was chartered for the purposes of testing new technologies for space science programs. The mission launched on October 24th 1998 and retired on December 18th 2001 after successfully taking images from Comet Borrelly.

Dynamic Used to describe an environment which changes frequently, without giving the agent time to consider their action. This often results in more reactive control from the agent.

Evolutionary Algorithm (EA) Perhaps the most common form of Evolutionary Computation. Evolutionary algorithms are parallel, heuristic based search processes that use biological representation and modification functions.

Evolutionary Artificial Neural Network (EANN) The process of applying an Evolutionary Algorithm to an Artificial Neural Network. This can be

used to search for an optimal network configuration, learning rule or more commonly, the values used in the ANN weight vector.

Evolutionary Computation (EC) A subdiscipline of Artificial Intelligence that focusses on solving complex combinatorial problems using an iterative process often inspired by biological evolution.

Effect The resulting changes to the world model as a result of a plan action.

Environment The local surroundings of an agent that it is designed to interact with. These environments can differ in size and scope depending on the problem we are trying to solve.

Episodic An environment that runs in individual epochs. As such, decisions in individual episodes have no effect on one another.

Engineering and Physical Sciences Research Council (EPSRC) A British Research Council that provides government funding to UK universities for research grants and postgraduate study in engineering and physical sciences (mathematics, artificial intelligence and computer science).

Evolutionary Strategy (ES) A search algorithm similar to the Hill Climber that adopts biological representation and search expansion. One of the three founding methodologies of Evolutionary Computation.

Fast Forward (FF) The Fast-Forward planner, a domain-independent planning system developed by Joerg Hoffmann.

Fitness Function The equivalent of a heuristic in an Evolutionary Algorithm. The fitness function is used to assess the quality of an evolved candidate.

Finite-State Machine (FSM) FSMs, also known as Finite-State Automata, are a mathematical abstraction used to represent the behaviour of computer programs. This allows for a designer to symbolically represents states of an agent or program. Furthermore, one can state the conditions necessary to move between states, and the effects actions have that necessitate such transitions.

Genetic Algorithm (GA) One of the principal fields of Evolutionary Computation/Algorithms, GAs are a parallel, heuristic-guided search technique

used to solve complex combinatorial problems. They search for exact or approximate solutions to problems in a multidimensional search space by mimicking traits of Darwinian evolution and biological reproduction.

Genetic and Evolutionary Computation Conference (GECCO) The largest conference in the field of Genetic and Evolutionary Computation. A merger of the International Conference on Genetic Algorithms (ICGA) with the Annual Genetic Programming Conference (GP).

Generalise Generalisation is the basis of deductive inference, whereby a concept is extended to a less specific criteria. In this instance a given concept A is considered to still adhere to this criteria, while other existing yet different concepts, *B*, will also match. For example, animals are a generalisation of birds, since all birds are animals but so are dogs, cats etc. In AI practices, this concepts extends to states, where we must consider whether a series of indvidial scenarios generalise under the same context. If this occurs, then we need only create a policy to act on each generalised state, rather than every possible state in the world..

Goal A set of one or more conditions that must be found in a state to satisfy the planning process. A plan will then traverse the state-space from initial-state to goal-state.

Hill Climbing (HC) A simple yet highly effective local-search algorithm. Hill-climbing begins with a random, typically poor, solution to a given problem and then makes small, incremental changes to the solution by exploring nearby successors in the state-space. Typically, once no further improvement can be made to the solution then the process will terminate. The changes made will be dictated by the selection algorithm applied, which can range from simple (random) changes, to more strict policies for successor selection.

Heuristic A 'rule of thumb' that guides an informed search towards the goal state. Provided the heuristic estimates the distance to the goal correctly, the resulting solutions will border on optimal.

Hierarchical Task Network Planning (HTN Planning) An alternative approach to automated planning that relies on a series of high-level tasks that can be decomposed to primitive tasks/actions. A task network emerges as a

result of constraints that can be enforced across the series of tasks in the hierarchy.

International Conference on Automated Planning and Scheduling (ICAPS) The internationally recognised forum for researchers involved in planning and scheduling technology. ICAPS is the resulting conference from the merger of the International Conference on Artificial Intelligence Planning and Scheduling (AIPS) and the European Conference on Planning (ECP).

International Planning Competition (IPC) The formal planning competition held by the AP community. Traditionally held and organised by researchers involved in the International Conference on Automated Planning and Scheduling (ICAPS).

Neuron A mathematical function that acts as simple represention of a biological neuron. An ANNs typically consists of layers of these artificial neurons. A neuron's output is achieved by passing the summation of inputs through a non-linear transfer function.

Neural Network Ensembles (NNE) Collections of redundant neural networks that are clustered to improve generalisation and robustness of agent behaviour by statistically selecting a response based on each networks outputs.

Observability A feature of modelled environments that impacts on the information available to an agent. If the environment is fully observable, then all information relevant to action selection is available, otherwise, the world is only partially observable.

Planning Domain and Description Language (PDDL) A planning defintion language first devised by Drew McDermott in 1998 as means to standardise planning description languages for competitive purposes (International Planning Competition 1998/2000). This language is now an international standard in planning communities and is a highly active research endeavour, with several notable extensions within the last 10 years.

Plateau A region of search-space where the heuristic values of all successor states is greater than or equal to the best seen so far. Hence there is no immediate 'better' path in sight, which can lead to disastrous circumstances unless concessions are made in the search algorithm.

Plan Manager (PM) The planning segement of the REAPER architecture, responsible for managing and monitoring the planning and execution of agents.

Population A collection of chromosomes that are being evolved as part of an Evolutionary Algorithm.

Precondition A series of conditions that must be in place prior to a plan action being executed.

Autonomous Remote Agent (RA) An onboard planning and scheduling system that was one of the twelve systems tested during NASA's DS1 mission. It is an onboard software system can enable goal-specific control of the spacecraft integrated with a robust fault recovery mechanism.

Rule Controller (RC) A series of rules designed to supplement the Plan Manager within the REAPER architecture as well as provide a correlation between PDDL actions and low-level reactive controllers.

Reachability Reachability in planning problems is whether a given state can be reached from the initial state in the problem. This is a key component of larger planning problems where reachability analysis becomes essential through the use of tree or graph structures.

Realtime Executive for Automated Plans using Evolutionary Robotics (REAPER Our architecture for devising and executing plans within game based environments. The system is comprised of the FF planner, a prolog rule base and a collection of pre-trained neural networks for execution. Each component existing within its own individual and de-coupled component of the architecture.

Reward Typically a numeric value that conceptualises the value of an action. This can be used to direct the action selection process.

Relaxed Plan Graph (RPG) A relaxed plan graph can be found to be similar to traditional plan graphs, however predicates found in the fact layers are never deleted. Despite this, mutually exclusive facts are ignored, allowing all possible actions to be created at a given action layer.

Propositional Satisfiability (SAT) The expression of a problem as a series of boolean statements, with the intent to satisfy all statements as true. Thus proving the problem is satisfiable.

Sequential An environment where action selection is effected not only by the actions that have preceded it, but also by the potential impact they may have in the future.

State A unique configuration of values that represent the situation in an environment.

Static A static environment is one that does not change during deliberation.

Stochastic An attribute of modelled environments. Stochasticity implies that an environment is non-deterministic, such that applying an action may result in one of a handful of different scenarios.

STanford Research Institute Problem Solver (STRIPS) The seminal planning platform developed by Fikes and Nilsson in 1971. While the abbreviation was initially devised to represent the planner, it is more commonly used to represent the set-theoretic language that was utilised within the planner to express problem instances. This is now considered the standard given that the language is still currently used while the STRIPS planner has long since been superseded.

Synapse In a biological nervous system, a synapse is a connection that permits the passing of electrical or chemical signals between cells. In an ANN it represents a weighted connection between *Neuron*s.

Temporal Difference Learning (TD) One of the most common Reinforcement Learning algorithms. The concept is to propagate reward values from a defined depth in the search space across preceding states based on selected actions.

The Open Racing Car Simulator (TORCS) A multi-platform, open-source racing simulator. While a fully functional racing game in its own right, it is also a research platform for AI development. This has resulted in numerous competitions in developing the most efficient AI racing drivers.

Transfer Function The function built within an artificial *Neuron*. This function will absorb all incoming, weighted values and then pass them to the output connection via a mathematical function.

VAL An automatic validation tool for the PDDL language. It is capable of examining plans and domains written in PDDL2, PDDL3 and PDDL+.

Appendix A

Nakatomi Domain Definition

In this appendix we provide the complete code listing for the Nakatomi PDDL domain file. This domain file is described in detail in Chapter 4 and used to model problem instances for our work in Chapter 5. A complete list of problem instances based on this domain file can be found in Appendices E and F.

```
(define (domain nakatomi)
        (:requirements :typing)
        (:types location door person artefact - object
                                room vent - location
                                switch aidkit bomb - artefact
                                agent hostage - person
        )

        (:predicates
                (at ?obj1 - person ?l - location)
                (in ?a - artefact ?r - room)
                (on ?obj1 - person ?s - switch)
                (corridor ?x ?y - room)
                (blocked ?x ?y - room)
                (doorway ?x ?y - room ?d - door)
                (ventilated ?x - room ?y - vent)
                (controls ?s - switch ?d - door)
                (open ?d - door)
                (closed ?d - door)
```

```
                (calm ?h − hostage)
                (uneasy ?h − hostage)
                (delirious ?h − hostage)
                (injured ?h − hostage)
                (unconscious ?h − hostage)
                (free ?a − agent)
                (carrying ?p − agent ?h − hostage)
                (holding ?p − agent ?k − aidkit)
                (armed ?b − bomb)
                (disarmed ?b −bomb)
        )

(:action WALK−AGENT−THROUGH−CORRIDOR
  :parameters
   (?agent − agent
    ?room−from − room
    ?room−to − room)
  :precondition
   (and (at ?agent ?room−from) (corridor ?room−from ?room−to))
  :effect
   (and (not (at ?agent ?room−from)) (at ?agent ?room−to))
)

(:action WALK−AGENT−THROUGH−DOORWAY
  :parameters
   (?agent − agent
    ?room−from − room
    ?room−to − room
    ?door − door)
  :precondition
   (and (at ?agent ?room−from) (doorway ?room−from ?room−to ?door)
   (open ?door))
  :effect
   (and (not (at ?agent ?room−from)) (at ?agent ?room−to))
)
```

```
(:action WALK−HOSTAGE−THROUGH−CORRIDOR
  :parameters
   (?hostage − hostage
    ?room−from − room
    ?room−to − room)
  :precondition
   (and (at ?hostage ?room−from) (corridor ?room−from ?room−to) (calm ?/
       hostage))
  :effect
   (and (not(at ?hostage ?room−from)) (at ?hostage ?room−to))
)

(:action WALK−CALM−HOSTAGE−THROUGH−DOORWAY
  :parameters
   (?hostage − hostage
    ?room−from − room
    ?room−to − room
    ?door − door)
  :precondition
   (and (at ?hostage ?room−from) (doorway ?room−from ?room−to ?door)
   (open ?door) (calm ?hostage))
  :effect
   (and (not(at ?hostage ?room−from)) (at ?hostage ?room−to))
)

(:action WALK−UNEASY−HOSTAGE−THROUGH−DOORWAY
  :parameters
   (?hostage − hostage
    ?room−from − room
    ?room−to − room
    ?door − door)
  :precondition
   (and (at ?hostage ?room−from) (doorway ?room−from ?room−to ?door)
   (open ?door) (uneasy ?hostage))
  :effect
   (and (not(at ?hostage ?room−from)) (at ?hostage ?room−to))
)
```

```
(:action WALK−DELIRIOUS−HOSTAGE−THROUGH−DOORWAY
  :parameters
   (?hostage − hostage
    ?room−from − room
    ?room−to − room
    ?door − door)
  :precondition
   (and (at ?hostage ?room−from) (doorway ?room−from ?room−to ?door)
   (open ?door) (delirious ?hostage))
  :effect
   (and (not(at ?hostage ?room−from)) (at ?hostage ?room−to))
)
```

```
(:action STEP−ON−SWITCH
  :parameters
   (?agent − agent
    ?switch−room − room
    ?switch − switch
    ?door − door)
  :precondition
   (and (at ?agent ?switch−room) (in ?switch ?switch−room)
   (closed ?door) (controls ?switch ?door))
  :effect
   (and (open ?door) (not(closed ?door)) (on ?agent ?switch)
   (not (at ?agent ?switch−room)))
)
```

```
(:action HOSTAGE−STEP−ON−SWITCH
  :parameters
   (?agent − hostage
    ?switch−room − room
    ?switch − switch
    ?door − door)
  :precondition
   (and (calm ?agent) (at ?agent ?switch−room) (in ?switch ?switch−room)
   (closed ?door) (controls ?switch ?door))
  :effect
   (and (open ?door) (not(closed ?door)) (on ?agent ?switch)
   (not (at ?agent ?switch−room)))
)

(:action STEP−OFF−SWITCH
  :parameters
   (?agent − agent
    ?switch − switch
    ?switch−room − room
    ?door − door)
  :precondition
   (and (on ?agent ?switch) (in ?switch ?switch−room)
   (open ?door) (controls ?switch ?door))
  :effect
   (and (not(open ?door)) (closed ?door) (not(on ?agent ?switch))
   (at ?agent ?switch−room))
)
```

```
(:action HOSTAGE−STEP−OFF−SWITCH
  :parameters
   (?agent − hostage
    ?switch − switch
    ?switch−room − room
    ?door − door)
  :precondition
   (and (on ?agent ?switch) (in ?switch ?switch−room) (open ?door)
   (controls ?switch ?door))
  :effect
   (and (not(open ?door)) (closed ?door) (not(on ?agent ?switch))
   (at ?agent ?switch−room))
)


(:action CRAWL−IN−VENT
  :parameters
  (?agent − agent
   ?room−from − room
   ?vent−to − vent)
  :precondition
   (and (at ?agent ?room−from) (ventilated ?room−from ?vent−to))
  :effect
   (and (not (at ?agent ?room−from)) (at ?agent ?vent−to))
)

(:action CRAWL−OUT−VENT
  :parameters
  (?agent − agent
   ?vent−from − vent
   ?room−to − room)
  :precondition
   (and (at ?agent ?vent−from) (ventilated ?room−to ?vent−from))
  :effect
   (and (not (at ?agent ?vent−from)) (at ?agent ?room−to))
)
```

```
(:action DEFUSE−BOMB
  :parameters
  (?agent − agent
   ?bomb − bomb
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (in ?bomb ?current−room) (armed ?bomb))
  :effect
   (and (disarmed ?bomb) (not (armed ?bomb)))
 )

(:action SLAP−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (at ?hostage ?current−room) (uneasy ?hostage))
  :effect
   (and (not (uneasy ?hostage)) (calm ?hostage))
)

(:action KNOCK−OUT−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?current−room − room)
  :precondition
  (and (at ?agent ?current−room) (at ?hostage ?current−room)
  (delirious ?hostage))
  :effect
  (and (not(delirious ?hostage)) (unconscious ?hostage))
)
```

```
(:action PICK−UP−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (free ?agent) (at ?hostage ?current−room)
   (unconscious ?hostage)) :effect
   (and (not (at ?hostage ?current−room)) (not (free ?agent))
   (carrying ?agent ?hostage))
)

(:action PUT−DOWN−HOSTAGE
  :parameters
  (?agent − agent
   ?hostage − hostage
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (carrying ?agent ?hostage))
  :effect
   (and (at ?hostage ?current−room) (free ?agent) (not(carrying ?agent ?hostage)))
)

(:action PICK−UP−FIRST−AID−KIT
  :parameters
  (?agent − agent
   ?aidkit − aidkit
   ?current−room − room)
  :precondition
   (and (at ?agent ?current−room) (in ?aidkit ?current−room))
  :effect
   (and (holding ?agent ?aidkit) (not(in ?aidkit ?current−room)))
)
```

```
(:action PATCH-UP-HOSTAGE
  :parameters
  (?agent - agent
   ?hostage - hostage
   ?aidkit - aidkit
   ?current-room - room)
  :precondition
   (and (at ?agent ?current-room) (holding ?agent ?aidkit)
   (at ?hostage ?current-room) (injured ?hostage))
  :effect
   (and (not(holding ?agent ?aidkit)) (not(injured ?hostage)) (calm ?hostage))
)

(:action CLEAR-RUBBLE
  :parameters
  (?agent1 - agent
   ?agent2 - agent
   ?current-room - room
   ?blocked-room - room)
  :precondition
   (and (at ?agent1 ?current-room) (at ?agent2 ?blocked-room)
   (blocked ?current-room ?blocked-room))
  :effect
   (and (not(blocked ?current-room ?blocked-room))
   (corridor ?current-room ?blocked-room)
   (corridor ?blocked-room ?current-room))
)

)
```

Appendix B

Set Classification Data

In this appendix we produce the data generated from Richard Jensen's automated set creator *FuzzyGen* (Jensen and Shen [2008]). We cover each relevant aspect of the environment that was used as part of the classification process in the Rule Controller of the REAPER architecture (Section 5.3.3 and Appendices C and D). Each set of results produces three sets, a low, middle and high range for the dataset, denoted as a left-shoulder, trapezium and a right-shoulder. By plotting the dataset values against the membership function, we see a trend similar to that shown in Figure B.1.

B.1 Blast Yield

To measure the blast yield sets, we required a range of values that express the potential range of the blast yield. Hence we wrote a simple piece of code that

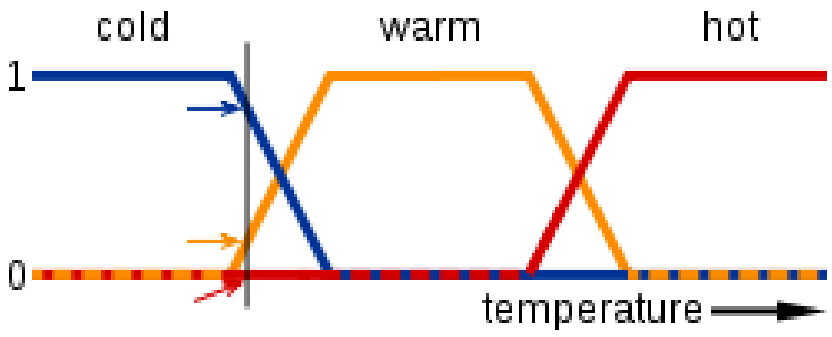


Figure B.1: A typical graph of the resulting set classification.

would randomly generate values within the range of $50 \leq x \leq 300$ pixels. Statistics from the dataset are in Table B.1, followed by the set classification in Table B.2.

Table B.1: Statistics of the blast yield source data.

Statistic	Value
Minimum	50.0
Maximum	299.0
Mean	173.63
Variance	5312.9
Std. Dev.	72.89

Table B.2: Values of the blast yield sets.

Set	Start	Middle	End
Left Shoulder	$-\infty$	100.74	159.05
Trapezium	100.74	159.05, 188.2	246.52
Right Shoulder	188.2	246.52	∞

B.2 Distance

Once again we create a dataset reflecting the minimum and maximum distance that we will measure between two agents in the same room. This is dictated by the potential dimensions of a given room, where the maximum dimension size is 300 pixels and each object is instatiated at bare minimum of 40 pixels from the perimeter. Statistics are in Table B.3 and set data in Table B.4.

Table B.3: Statistics of the distance source data.

Statistic	Value
Minimum	3.0
Maximum	329.0
Mean	147.68
Variance	4839.75
Std. Dev.	69.57

Table B.4: Values of the distance sets.

Set	Start	Middle	End
Left Shoulder	$-\infty$	78.08	133.73
Trapezium	78.08	133.73, 161.56	217.21
Right Shoulder	161.56	217.21	∞

B.3 Fuse Length

This set reflects on the maximum and minimum length of a bomb fuse. The dataset used was based on a fuse being of length $1 \leq x \leq 10$. The statistics from the dataset are provided in Table B.5, followed by the set classifications in Table B.6.

Table B.5: Statistics of the fuse length source data.

Statistic	Value
Minimum	1.0
Maximum	10.0
Mean	5.529
Variance	8.3
Std. Dev.	2.88

Table B.6: Values of the fuse length sets.

Set	Start	Middle	End
Left Shoulder	$-\infty$	2.64	4.95
Trapezium	2.64	4.95, 6.1	8.41
Right Shoulder	6.1	8.41	∞

B.4 Health

Our final set is based on the range of health an agent can have, this is naturally in the range $0 \leq x \leq 100$. Table B.7 provides statistics from the health dataset, while Table B.8 shows the set classification made by FuzzyGen.

Table B.7: Statistics of the health source data.

Statistic	Value
Minimum	1.0
Maximum	100.0
Mean	48.23
Variance	828.5
Std. Dev.	28.7

Table B.8: Values of the health sets.

Set	Start	Middle	End
Left Shoulder	$-\infty$	19.44	42.47
Trapezium	19.44	42.47, 53.98	77.01
Right Shoulder	53.98	77.01	∞

Appendix C

Rule Controller Knowledge Base: Threat Rules

In this appendix we highlight the complete set of rules and clauses that are used as part of the Rule Controller's threat detection process (Section 5.3.3). In short, each rule is designed to express the threat level of a given entity to the Bruce executive in the environment. This threat level is calculated based on various pieces of information from the current state of the game world and is passed through a a fuzzification process to express the numeric values symbolically.

C.1 Agent Threats

We begin with the agent_threat rule, where we check the threat level of a given agent based on an informed classification of its capability to impede our progess. This is determined by the distance it is from our agent and the armament it is carrying. Then using this threat level we then decide on what action (if any) should be taken to resolve this threat.

```
agent_threat(DIST,ARM,THREAT,ACTION):−
    agent_threat_level(DIST,ARM,THREAT),
    agent_threat_action(THREAT,ACTION).
```

Next we explore the clauses used as part of this rule and the knowledge base we have defined. In the following set of clauses, we denote a threat level to different

combinations of distance and armament. Note that in certain examples we use '_' to represent any and all potential inputs for the specific variable.

```
agent_threat_level(close,gun,high).
agent_threat_level(close,knife,medium).
agent_threat_level(close,none,low).
agent_threat_level(near,gun,medium).
agent_threat_level(near,_,low).
agent_threat_level(far,gun,medium).
agent_threat_level(far,_,low).
```

To complete the agent threat knowledge base, we provde a simple set of clauses that dictate if an agent carries at the bare minimum a medium threat level, then we must eliminate it.

```
agent_threat_action(high, destroy_target).
agent_threat_action(medium, destroy_target).
```

C.2 Bomb Threats

Next we look at the threat posed by bombs in the environment. We classify the threat level of a bomb based on three pieces of information; the distance of the bomb to the agent, the blast yield and the remaining fuse on the bomb. Once again we have a simple rule setup for ascertaining the correct action.

```
bomb_threat(DIST,YIELD,FUSE,THREAT,ACTION):-
    bomb_threat_level(DIST,YIELD,FUSE,THREAT),
    bomb_threat_action(THREAT,ACTION).
```

Lastly, on the following page we provide the knowledge base for the bomb threat level and the corresponding action clauses. These are very similar to the agent threat level clauses we saw previously.

```
bomb_threat_level(close, high, low, high).
bomb_threat_level(close, high, medium, high).
bomb_threat_level(close, high, high, medium).
bomb_threat_level(close, medium, low, high).
bomb_threat_level(close, medium, medium, medium).
bomb_threat_level(close, medium, high, medium).
bomb_threat_level(close, low, low, high).
bomb_threat_level(close, low, medium, medium).
bomb_threat_level(close, low, high, low).
bomb_threat_level(near,_,low,high).
bomb_threat_level(near,_,medium,medium).
bomb_threat_level(near,_,high,low).
bomb_threat_level(far,_,_,low).

bomb_threat_action(medium,defuse_bomb).
bomb_threat_action(high,defuse_bomb).
```

Appendix D

Rule Controller Knowledge Base: Controller Rules

In this appendix we give the complete set of controller rules found within the Rule Controller of the REAPER executive. Furthermore we also provide commentary to explain the different rules and how we came to this final collection.

Our controller rules base is designed to allow us to immediately associate facts of the world to corresponding actions that must be taken. Each of these rules is designed to correlate to (one or more) PDDL actions found in the nakatomi.pddl file (Appendix A). The concept being that when we are dealing with an action in the world, we must ensure that we are satisfied with all of the conditions ranging from planner-level down to a lower-level.

The first series of rules on the following page are the quick-lookup rules, where we immediately associate a particular PDDL action with a reactive or hardcoded controller.

```
quick−lookup(walk−agent−through−corridor,visit_waypoint).
quick−lookup(walk−agent−through−doorway,visit_waypoint).
quick−lookup(walk−hostage−through−corridor,visit_waypoint).
quick−lookup(walk−calm−hostage−through−doorway,visit_waypoint).
quick−lookup(walk−uneasy−hostage−through−doorway,visit_waypoint).
quick−lookup(walk−delirious−hostage−through−doorway,visit_waypoint).
quick−lookup(hostage−step−on−switch,visit_waypoint).
quick−lookup(hostage−step−off−switch,visit_waypoint).
quick−lookup(step−on−switch,visit_waypoint).
quick−lookup(step−off−switch,visit_waypoint).
quick−lookup(crawl−in−vent,visit_waypoint).
quick−lookup(crawl−out−vent,visit_waypoint).
quick−lookup(put−down−hostage,put_down_hostage).
quick−lookup(pick−up−first−aid−kit,grab_item).
quick−lookup(pick−up−hostage,grab_item).
```

Next we consider the finer details of each action to ensure that which we refer to as 'low-level preconditions' are satisfied before we proceed further. If they are not satisfied, then we must consider applying *bridge actions* that satisfy these preconditions, allowing us to proceed with the original action. In all of these instances we need to resolve any issues in the spatial relationship between the two objects. Put simply, we need to ensure that while an entity is in the same room as another, it must be physically close to it in order to carry out any sort of interaction. Otherwise, a supplementary action is required in order to satisfy these requirements.

```
slap−hostage(close,uneasy,slap_hostage).
slap−hostage(far,uneasy,visit_waypoint).
slap−hostage(near,uneasy,visit_waypoint).
slap−hostage(close,_,re−plan).

knock−out−hostage(close, delirious, knock_out_hostage).
knock−out−hostage(near, delirious, visit_waypoint).
knock−out−hostage(far, delirious, visit_waypoint).
knock−out−hostage(close, _, re_plan).

patch−up−hostage(close, patch_up_hostage).
patch−up−hostage(near, visit_waypoint).
patch−up−hostage(far, visit_waypoint).

clear−rubble(close, clear_rubble).
clear−rubble(near, visit_waypoint).
clear−rubble(far, visit_waypoint).

defuse−bomb(close, defuse_bomb).
defuse−bomb(near, visit_waypoint).
defuse−bomb(far, visit_waypoint).
```

This shows the complete list of clauses used for the controller rule base. Based on these clauses our agent can execute any action from the PDDL domain description.

Appendix E

BruceWorld Initial Test Files

E.1 BasicTest1

```
(define (problem BASIC−TEST−1)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        l1 − room
        l2 − room
        l3 − room
        l4 − room
        l5 − room
        l6 − room
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (corridor l1 l2)
        (corridor l2 l1)
        (corridor l2 l3)
        (corridor l3 l2)
        (corridor l3 l4)
        (corridor l4 l3)
```

```
        (corridor l4 l5)
        (corridor l5 l4)
        (corridor l5 l6)
        (corridor l6 l5)
        )

        (:goal (and
        (at agent1 l6)
        ))

)
```

E.2 BasicTest2

```
(define (problem BASIC−TEST−2)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        b1 − bomb
        l1 − room
        l2 − room
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l2)
        (unconscious hostage1)
        (corridor l1 l2)
        (corridor l2 l1)
        (in b1 l2)
        (armed b1)
        )

        (:goal (and
        (disarmed b1)

        (at hostage1 l1)
        ))

)
```

E.3 BasicTest3

```
(define (problem BASIC−TEST−3)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        l1 − room
        l2 − room
        l3 − room
        s1 − switch
        d1 − door
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l3)
        (in s1 l2)
        (corridor l1 l2)
        (corridor l2 l1)
        (doorway l2 l3 d1)
        (doorway l3 l2 d1)
        (controls s1 d1)
        (closed d1)
        (calm hostage1)
        )

        (:goal (and
        (at agent1 l1)
        (at hostage1 l1)
        ))

)
```

E.4 BasicTest4

```
(define (problem BASIC−TEST−4)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        hostage2 − hostage
        switch1 − switch
        switch2 − switch
        d1 − door
        d2 − door
        l1 − room
        l2 − room
        l3 − room
        l4 − room
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l2)
        (at hostage2 l4)
        (calm hostage2)
        (calm hostage1)
        (in switch1 l1)
        (in switch2 l3)
        (doorway l1 l2 d1)
        (doorway l2 l1 d1)
        (doorway l3 l4 d2)
        (doorway l4 l3 d2)
        (corridor l1 l3)
        (corridor l3 l1)
        (controls switch1 d1)
        (controls switch2 d2)
        (closed d1)
        (closed d2)
```

```
        )

        (:goal (and
        (at agent1 l3)
        (at hostage1 l3)
        (at hostage2 l3)
        ))

)
```

E.5 BasicTest5

```
(define (problem BASIC−TEST−5)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        hostage2 − hostage
        l1 − room
        l2 − room
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l1)
        (at hostage2 l1)
        (unconscious hostage1)
        (uneasy hostage2)
        (corridor l1 l2)
        (corridor l2 l1)
        )

        (:goal (and
        (at agent1 l2)
        (at hostage1 l2)
        (at hostage2 l2)
        ))

)
```

Appendix F

BruceWorld Advanced Test Files

F.1 AdvancedTest1

```
(define (problem ADVANCED-TEST-1)
        (:domain nakatomi)

        (:objects
        agent1 - agent
        hostage1 - hostage
        hostage2 - hostage
        kit1 - aidkit
        l1 - room
        l2 - room
        l3 - room
        l4 - room
        l5 - room
        l6 - room
        l7 - room
        l8 - room
        v1 - vent
        )

        (:init
        (at agent1 l1)
        (free agent1)
```

```
        (at hostage1 l5)
        (at hostage2 l6)
        (injured hostage1)
        (unconscious hostage2)
        (in kit1 l8)
        (corridor l1 l2)
        (corridor l2 l1)
        (corridor l2 l3)
        (corridor l3 l2)
        (corridor l3 l4)
        (corridor l4 l3)
        (corridor l3 l5)
        (corridor l5 l3)
        (corridor l2 l7)
        (corridor l7 l2)
        (corridor l6 l7)
        (corridor l7 l6)
        (ventilated l3 v1)
        (ventilated l8 v1)
        )

        (:goal (and
        (at agent1 l4)
        (at hostage1 l4)
        (at hostage2 l4)
        ))

)
```

F.2 AdvancedTest2

```
(define (problem ADVANCED-TEST-2)
        (:domain nakatomi)

        (:objects
        agent1 - agent
        hostage1 - hostage
        hostage2 - hostage
        s1 - switch
        d1 - door
        l1 - room
        l2 - room
        l3 - room
        l4 - room
        l5 - room
        v1 - vent
        v2 - vent
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l2)
        (at hostage2 l4)
        (uneasy hostage1)
        (unconscious hostage2)
        (corridor l2 l5)
        (corridor l5 l2)
        (corridor l3 l4)
        (corridor l4 l3)
        (doorway l3 l5 d1)
        (doorway l5 l3 d1)
        (controls s1 d1)
        (closed d1)
        (in s1 l5)
        (ventilated l1 v1)
        (ventilated l3 v1)
```

```
        (ventilated l1 v2)
        (ventilated l2 v2)
        )

        (:goal (and
        (at agent1 l3)
        (at hostage1 l3)
        (at hostage2 l3)
        ))

)
```

F.3 AdvancedTest3

```
(define (problem ADVANCED−TEST−3)
        (:domain nakatomi)

        (:objects
        agent1 − agent
        hostage1 − hostage
        hostage2 − hostage
        hostage3 − hostage
        s1 − switch
        s2 − switch
        s3 − switch
        s4 − switch
        d1 − door
        d2 − door
        d3 − door
        d4 − door
        l1 − room
        l2 − room
        l3 − room
        l4 − room
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l2)
        (at hostage2 l3)
        (at hostage3 l4)
        (calm hostage1)
        (calm hostage2)
        (calm hostage3)
        (doorway l1 l2 d1)
        (doorway l2 l1 d1)
        (doorway l2 l4 d2)
        (doorway l4 l2 d2)
        (doorway l3 l4 d3)
```

```
        (doorway l4 l3 d3)
        (doorway l1 l3 d4)
        (doorway l3 l1 d4)
        (controls s1 d4)
        (closed d1)
        (controls s2 d3)
        (closed d2)
        (controls s3 d1)
        (closed d3)
        (controls s4 d2)
        (closed d4)
        (in s1 l1)
        (in s2 l2)
        (in s3 l3)
        (in s4 l4)
        )

        (:goal (and
        (at agent1 l4)
        (at hostage1 l4)
        (at hostage2 l4)
        (at hostage3 l4)
        ))

)
```

F.4 AdvancedTest4

```
(define (problem ADVANCED-TEST-4)
        (:domain nakatomi)

        (:objects
        agent1 - agent
        hostage1 - hostage
        hostage2 - hostage
        s1 - switch
        s2 - switch
        s3 - switch
        s4 - switch
        s5 - switch
        d1 - door
        d2 - door
        d3 - door
        d4 - door
        d5 - door
        l1 - room
        l2 - room
        l3 - room
        l4 - room
        l5 - room
        l6 - room
        l7 - room
        l8 - room
        l9 - room
        l10 - room
        v1 - vent
        v2 - vent
        )

        (:init
        (at agent1 l1)
        (free agent1)
        (at hostage1 l1)
        (at hostage2 l6)
```

(calm hostage1)
(calm hostage2)
(doorway l1 l2 d1)
(doorway l2 l1 d1)
(corridor l2 l3)
(corridor l3 l2)
(doorway l3 l4 d2)
(doorway l4 l3 d2)
(doorway l4 l5 d3)
(doorway l5 l4 d3)
(doorway l6 l7 d4)
(doorway l7 l6 d4)
(ventilated l7 v1)
(ventilated l8 v1)
(doorway l8 l9 d5)
(doorway l9 l8 d5)
(ventilated l9 v2)
(ventilated l4 v2)
(corridor l2 l7)
(corridor l7 l2)
(corridor l1 l10)
(corridor l10 l1)
(controls s1 d1)
(closed d1)
(controls s2 d2)
(closed d2)
(controls s3 d3)
(closed d3)
(controls s4 d4)
(closed d4)
(controls s5 d5)
(closed d5)
(in s1 l6)
(in s2 l9)
(in s3 l4)
(in s4 l1)
(in s5 l3)
)

```
        (:goal (and
        (at agent1 l9)
        (at hostage1 l10)
        (at hostage2 l5)
        ))

)
```

Appendix G

Publication List

For the examiners consideration, we provide a list of all papers that have been published at conference level during the period of our studies. Each paper has been subjected to peer review and accepted after making appropriate corrections. We also denote the number of known citations made by other researchers where necessary.

- **Real-time Execution of Automated Plans using Evolutionary Robotics** T. Thompson and J. Levine. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games (CIG 2009). September 2009.*
- **Improving Control Through Subsumption in the EvoTanks Domain** T.Thompson, F. Milne, A. Andrew and J. Levine. *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games 2009 (CIG 2009). September 2009.* (1 citation)
- **Scaling-up Behaviours in EvoTanks: Applying Subsumption Principles to Artificial Neural Networks.** T. Thompson and J. Levine. *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games 2008 (CIG 2008). December 2008.* (2 citations)
- **An Evaluation of the Benefits of Look-Ahead in Pac-Man.** T. Thompson, L. McMillan, J. Levine, and A. Andrew. *Proceedings of the IEEE Symposium on Computational Intelligence and Games 2008 (CIG 2008). December 2008.* (2 citations)

- **Evolution of Counter-Strategies: Application of Co-evolution to Texas Hold'em Poker.** T. Thompson, J. Levine, and R. Wotherspoon. *Proceedings of the 2008 IEEE Symposium on Computational Intelligence and Games 2008 (CIG 2008). December 2008.*

- **EvoTanks: Co-Evolutionary Development of Game-Playing Agents.** T. Thompson, J. Levine, and G. Hayes. *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games (CIG 2007). April 2007.* (2 citations)