Learning Plan Selection for BDI Agent Systems

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

Dhirendra Singh
B.Eng., B.App.Sc.

School of Computer Science and Information Technology
College of Science Engineering and Health
RMIT University

April 2011

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Dhirendra Singh
School of Computer Science and Information Technology
RMIT University
April 14, 2011

To Sharyn, Rohan, and a little button

Acknowledgments

My work in this thesis would simply not have been possible had it not been for so many people who have supported and helped me everyday. To all of you I am truly indebted.

I have to first thank my supervisors Professor Lin Padgham and Dr. Sebastian Sardina for their invaluable guidance throughout this work. I am deeply grateful for Lin who, in the decade that I have known her, has always gone out of her way to help me in my endeavours. She is solely responsible for instilling the spark that eventually drew me back to research after many years in the industry. Sebastian is a brilliant researcher, who has a wonderful knack for separating sense from nonsense in all our discussions, and who I thoroughly respect. I would also like to extend my gratitude to Dr. Andy Song, Dr. Stèphane Airiau from the University of Amsterdam, and Professor Sandip Sen from the University of Tulsa, for mentoring my various excursions as I tried to establish my final research path.

This work was vastly funded by an RMIT PhD scholarship, in some part by the Australian Research Council grant DP1094627, as well as a top-up PhD scholarship from The Commonwealth Scientific and Industrial Research Organisation (CSIRO), Australia. I am fortunate to have a wonderful supervisor like Dr. Geoff James who has been extremely supportive and encouraging of this collaboration, and has willingly funded all of my many visits to Sydney and Newcastle to engage with the research team.

There have been simply too many friends and fellow students that, willingly or otherwise, have had to endure my babbling over the years. These include Rod and Meghan, Natalie, Leanne and Ivan, Steve, John, Lavindra, Nitin, Iman, and Matthew. You have all helped me in more ways than you know and for that I am truly thankful.

It will be remiss of me not to acknowledge the immense support that I have had from my family and friends who have patiently tolerated my seemingly unending study. Our son Rohan has not known life without daddy working on his thesis. I would like to express my heartfelt gratitude to my parents, who visited all the way from India to help look after Rohan in the first few months, and Mum and Steve who have driven up the several hundred kilometres from Adelaide on many occasions to help support us whenever we asked.

Finally, I would like to thank my beautiful wife Sharyn who has done more for me than I can ever thank her for. She willingly agreed to change her life completely so that I could start my PhD. Had it not been for her encouragement, I would not have taken the first step in this journey. She is my best friend, a true champion, and I love her dearly.

Credits

Portions of the material in this thesis have previously appeared or will appear in the following publications:

- Singh, D., Sardina, S., Padgham, L., and James, G. (2011). Integrating learning into a BDI agent for environments with changing dynamics. In Toby Walsh and Craig Knoblock, editors, *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2525–2530, Barcelona, Spain.

- Singh, D., Sardina, S., and Padgham, L. (2010). Extending BDI plan selection to incorporate learning from experience. In *Robotics and Autonomous Systems (RAS)*, 58:1067–1075.

- Singh, D., Sardina, S., Padgham, L., and Airiau, S. (2010). Learning context conditions for BDI plan selection. In van der Hoek, Kaminka, Lespérance, Luck, and Sen, editors, *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 325–332, Toronto, Canada.

Contents

List of Figures

List of Tables

Abstract

Belief-Desire-Intention (BDI) is a popular agent-oriented programming approach for developing robust computer programs that operate in dynamic environments. These programs contain pre-programmed abstract procedures that capture domain know-how, and work by dynamically applying these procedures, or plans, to different situations that they encounter. Agent programs built using the BDI paradigm, however, do not traditionally do learning, which becomes important if a deployed agent is to be able to adapt to changing situations over time. Our vision is to allow programming of agent systems that are capable of adjusting to ongoing changes in the environment's dynamics in a robust and effective manner.

To this end, in this thesis we develop a framework that can be used by programmers to build adaptable BDI agents that can *improve plan selection* over time by learning from their experiences. These learning agents can dynamically adjust their choice of which plan to select in which situation, based on a growing understanding of what works and a sense of how reliable this understanding is. This reliability is given by a perceived measure of *confidence*, that tries to capture how well-informed the agent's most recent decisions were and how well it knows the most recent situations that it encountered.

An important focus of this work is to make this approach practical. Our framework allows learning to be integrated into BDI programs of reasonable complexity, including those that use recursion and failure recovery mechanisms. We show the usability of the framework in two complete programs: an implementation of the Towers of Hanoi game where recursive solutions must be learnt, and a modular battery system controller where the environment dynamics changes in ways that may require many learning and relearning phases.

Chapter 1

Introduction

There has been a phenomenal growth in the application of computing to solve complex problems in every field of work. The growing complexity of such problems has made the task of writing computer programs increasingly challenging and prone to errors. As a testament to the issue, an exhaustive 2002 study estimated the cost of software errors to the US economy alone, to be up to $59.5 billion annually [Tassey, 2002]. This thesis is essentially about making computer programs more robust and less error-prone. We do this by adding learning, that is, an ability to adapt to changing environments using past experiences, to computer programs written in a high-level programming paradigm called agent-oriented programming.

Intelligent software agent technology is rapidly becoming attractive because it simplifies the development of complex programs that operate in highly dynamic environments, especially when many choices are to be made about appropriate ways for handling different situations. For example, an empirical study has shown that agent technology repeatedly produces substantial savings in time and effort when developing logistics software, with the average programming productivity increase being over 350% [Benfield et al., 2006]. The use of agents in financial trading is also on the increase. A third of all EU and US stock trades in 2006 were driven by automatic programs [Aite Group, 2006]. In 2009, high frequency trading by intelligent computer programs accounted for 73% of all US equity trading volume [Lati, 2009]. In the area of computer generated graphics, one big success story for agent technology is Massive Software, a company that specialises in autonomous agent technology for animating realistic crowd behaviours. The extent to which their work is applied in present day films is remarkable,

and includes such successful ventures as Avatar and The Lord of the Rings trilogy [Massive Software, 2010; Verrier, 2006].

So what is intelligent agent technology? While the term is used loosely in media to classify any *computer software* that acts on behalf of users, the notion has a specific meaning in the context of this thesis. The essence of the idea is an *agent* — an entity that exhibits autonomy of operation, a social ability to interact with others, reactivity to changes in its environment, and pro-activeness in pursuing its objectives. In general terms an agent is analogous to its human counterpart and may be viewed as an intentional system "whose behaviour can be predicted by the method of attributing belief, desires and rational acumen."[Dennett, 1987]

The concept of agency is an abstraction to help us understand and explain the behaviour of complex software. The history of computer program design has been underpinned by many fundamental styles of *programming*, based in different notions of abstraction and driven by necessity of application. Over the decades, there has been a progressive shift in programming paradigms towards higher levels of abstraction to accommodate increasingly complex program design: from assembly programming, to structural and procedural programming, to object-oriented programming, and more recently to agent-based or *agent-oriented programming* [Shoham, 1993] that is the focus of this thesis.

Of particular interest to us is the Belief-Desire-Intention (BDI) model of intelligent agents [Bratman et al., 1988; Cohen and Levesque, 1990; Rao and Georgeff, 1991, 1992, 1995], which is a popular and successful cognitive framework for implementing practical reasoning in computer programs. The BDI model has its roots in philosophy, and is based on Bratman's theory of human practical reasoning [Bratman, 1987] and Dennett's theory of intentional systems [Dennett, 1987]. A BDI agent is characterised by an informational state (*beliefs*) about the world, a motivational state (*desires*) or objectives in the world, and a deliberative state (*intentions*) or commitments that are current in the world. It uses these mental attitudes within a process of *deliberation* to rationalise its actions, much like humans do.

Conceptually, BDI programs achieve their goals and react to different situations by matching pre-programmed abstract recipes, or *plans*, to the situation. For example, a programmer may provide two plans that could be used to get to work. While one plan may involve cycling to work, the other may require catching the bus. Which one of these will be used by the agent will depend on the situation, such as whether the agent

believes it is going to rain or not.

BDI agent systems are particularly suited for dynamic environments where enabling conditions for a course of action can change quickly, causing plans to fail midway. When this happens, BDI agents are able to reassess the situation and try alternative plans of action to achieve their goal rather than giving up and aborting it altogether. This *failure recovery* mechanism is a powerful feature of BDI programs that makes them robust in rapidly changing environments.

The interleaving of deliberative (triggered by the agents goals and beliefs) and reactive (triggered by the external environment) reasoning in BDI agent systems, combined with the failure recovery mechanism, delivers a responsiveness that is well suited to many complex domains [Burmeister et al., 2008; Karim and Heinze, 2005; Rao and Georgeff, 1995]. Several programming languages exist today in the BDI tradition, including AgentSpeak(L) [Rao, 1996], JACK [Busetta et al., 1999], Jason [Bordini et al., 2007], JADEX [Pokahr et al., 2003], GOAL [de Boer et al., 2007], and 3APL [Hindriks et al., 1999], to name a few.

Despite its successes, one of the criticisms of the BDI model is that it does not incorporate the notion of learning, when human intelligence is generally regarded to include elements of reasoning and learning from experience. *It is this relationship between reasoning and learning in practical decision making that we explore in this thesis.* Precisely, our goal is develop a methodology for programming a new generation of BDI agents that are inherently capable of improving their behaviour by learning as they go. In that sense, the question we wish to answer is: "*how does one write BDI intelligent computer programs that continually improve their behaviour by learning from ongoing experiences*."

The issue of combining learning with deliberation for ongoing, or *online*, decision making in BDI programs has not been widely addressed in the literature. Although some work has been done to integrate prior, or *offline*, learning with BDI reasoning [Brusey, 2002; Guerra-Hernández et al., 2005; Lokuge and Alahakoon, 2007; Nguyen and Wobcke, 2006; Riedmiller et al., 2001], these systems do not actively learn once deployed. As such, they do not deal with the issue that, in ongoing learning, acting and learning are interleaved, and the agent must also consider how *reliable* its current learning is when making decisions. There are also examples of learning in hierarchical task network (HTN) planning systems [Erol et al., 1994], a closely related area to BDI programming,

but this work similarly does not deal with online learning issues since HTN planning itself is performed offline. Our work does share concerns with existing work in hierarchical reinforcement learning [Barto and Mahadevan, 2003] where learning and acting are interleaved. That work relies on a formal description of the agent's operating environment and uses a very different language to agent programming, which is where our contribution lies.

The first attempts to integrate online learning in BDI systems were made within our own research group [Airiau et al., 2009; Singh et al., 2010a,b] and this thesis forms part of that project. While this work is a start, it explores only the beginnings of what we consider to be a much wider research agenda for programming practical BDI agents that learn.

Summary of Contributions

The specific contribution of this thesis is a usable framework for integrating machine learning [Mitchell, 1997] with BDI programming. In particular, *our focus is on improving plan selection over time in a BDI agent through the use of ongoing learning*. We test our framework in an empirical setting and describe complete programs to demonstrate how *adaptive* BDI programs that learn can be built using this framework. Overall, our work enables the building of agent software that is more reliable and robust than the current generation of such programs.

A BDI Learning Framework The first contribution of this thesis is a framework that can be used by programmers to build BDI agent systems that are capable of learning. In particular, such learning is focussed on improving plan selection over time based on ongoing experience. The framework allows programmers to write dynamic applicability conditions for plans in the agent's plan library. This means that the decision about when to use a given plan is dependent not only on programmed applicability conditions, but additionally on learnt knowledge about how well the plan actually works in the environment. The machine learning technique used for representing this knowledge in the framework is decision trees [Quinlan, 1986]. The other aspect of the framework is a new probabilistic mechanism that selects from applicable plans in any situation based on their perceived likelihood of success. This way plans that are believed to work well are selected more often. Importantly, the framework is useable in complete BDI programs including those that use BDI failure recovery and recursion.

Dynamic Confidence Measure The second important contribution of our work is a measure for estimating the reliability of ongoing learning. Since our BDI agent is learning and acting in the environment in an interleaved manner, then in order to select sensibly between candidate plans based on learnt knowledge it must also have some sense of how reliable its current learning is. Our proposed *confidence* measure is such a quantitative measure of reliability and can be used to weigh up between selecting what is known (but possibly unreliable) and what is unknown (but possibly better) — commonly known as the exploitation versus exploration dilemma in machine learning literature. The *dynamic* aspect of this measure relates to its ability to continuously adjust according to how the agent's most recent choices are faring: the more that its choices lead to success the higher the confidence and vice versa. Among other things, this means that learning is not a one-off event, in contrast to the usual assumption in machine learning literature, but a dynamic ongoing process.[1] By integrating the dynamic measure in its exploration (plan selection) strategy, our learning agent is able to deal with such changes in the environment that make its existing learning fail, by appropriately promoting new exploration in response to failures on an ongoing basis.

A Practical Approach A third and final consideration in this thesis is to make the task of programming learning BDI agents practical. We succeed in this goal in so far that we show how we build two complete BDI programs using the framework: a BDI agent that learns to solve the Towers of Hanoi puzzle, and a modular battery storage controller that operates in an environment that requires re-learning. The overall task of allowing learning BDI agents to be programmed easily is far from over. In a recent article on his views about the future of machine learning research, Professor Tom Mitchell, author of Machine Learning [Mitchell, 1997], a widely used textbook on the subject, said: *"Can a new generation of computer programming languages directly support writing programs that learn?"* [Mitchell, 2006] Indeed this vision also applies to BDI programming.

[1] In principle our confidence measure supports infinite learning, but there are other important issues that we have not addressed in this work such as how to appropriately maintain what is useful in an infinite stream of experience information. This is indeed an open question also for the machine learning community.

Outline of Thesis

This thesis is organised as follows. In Chapter 2 we present the background information that is required to understand this work. Chapter 3 describes our learning framework that augments plans' applicability conditions with decision trees to be learnt over time, and provides a probabilistic plan selection mechanism that makes use of the ongoing learning. We then show how the framework works for BDI programs that use failure recovery and recursion. Chapter 4 deals with the key issue of reliability of ongoing learning and develops a quantitative measure of confidence that dynamically adapts to changing dynamics of the environment. In Chapter 5 we test the validity of the framework and explore some of the nuances in learning using a suite of synthetic BDI agents and learning tasks. Chapter 6 then demonstrates the use of our learning framework in two complete BDI programs: the Towers of Hanoi puzzle and a modular battery controller. Chapter 7 compares our work in BDI learning to related work in two other fields of research: hierarchical task network (HTN) planning and hierarchical reinforcement learning. Finally, in Chapter 8 we summarise our contributions and discuss areas for future work.

Chapter 2

Background

Computers are ubiquitous. Primarily so because they make our everyday lives simpler by helping us achieve our tasks. In today's digital world, the information processing power of computers is utilised in innumerable ways across all disciplines of life: be it the arts, sports, or sciences. As computing is increasingly applied to newer and more challenging tasks, a parallel effort is always underway to find more effective ways of constructing *programs* that instruct computers to perform such tasks. *Programming*, the art of writing programs, in other words, is ever evolving.

The history of computer control is underpinned by certain fundamental styles of programming, based in different notions of abstraction and driven by necessity of application. The earliest and most direct way of passing sequential control information to computer processors in order to direct them to perform a series of tasks was through a combination of flip flops in hardware. ENIAC, the first general-purpose electronic computer built in the early 1940s was programmed this way. This *hardwired* control, however, was also a rigid and expensive method of programming since any changes to operation required the control circuit to be re-designed and replaced.

Around the same time in the 1940s, John von Neumann published his now famous First Draft [von Neumann, 1945] — incidentally an incomplete document — that described a computer design using the notion of *stored programs*.[1] This document grabbed the attention of Maurice Wilkes, second director of the University of Cambridge Mathemat-

[1] The computer architecture prescribed in "First Draft of a report on the EDVAC" is better known these days as the "von Neumann Architecture."

ical Laboratory, who took upon himself to construct a practical computer inspired by this design. His insights from the design of the resulting EDSAC,[2] and one so-named Whirlwind computer from across the Atlantic at MIT, led Wilkes to eventually develop the *microprogramming* paradigm: a new way of writing hardware control instructions to replace the combinational circuitry of hardwired design prevalent at the time. In this paradigm, a typical microinstruction for a processor consisted of several sequential operations at the hardware level, such as: connect the appropriate input registers to the arithmetic logic unit, perform the desired calculation, then store the result in the desired output register. In that sense, the microcode was specific to the hardware it controlled, and resided in internal memory. Importantly, however, microprogramming allowed processors with complex and extendible *instruction sets* to be designed using much simpler circuits compared to hardwired control.

This separation of programs from hardware concerns was mostly complete by the 1950s, when *assembly programming* became the predominant programming paradigm of choice. By using a processor's published instruction set, a programmer with no prior knowledge of the underlying hardware was now able to successfully write programs for it using assembly language. Through the use of mnemonics to replace machine code and symbolic labels to replace direct memory addresses, assembly programming made the task of writing, understanding, and correcting programs much easier. While still a low-level programming language by current standards, assembly programming was nonetheless considered extremely versatile and powerful as it enabled the use of subroutines and external routine libraries. The Apollo Guidance Computer (AGC) onboard lunar module LM-5 that landed the first humans on the moon in 1969, for instance, was programmed almost entirely in assembly language.

By the 1970s, improvements in computing power and memory saw the development of several *high-level programming languages*,[3] like FORTRAN, COBOL and C [Kernighan et al., 1978]. The paradigms in use in these languages were that of *structured* and *procedural* programming that prescribed the use of structures and procedure calls respectively. This allowed more complex programs to be written by systematically separating out the functionality into routines that could be understood independently.

Later in the 1970s, a new paradigm called *object oriented programming* emerged. The

[2] For his efforts on EDSAC, Wilkes was to later receive the A. M. Turing Award in 1967.

[3] The phrase refers to the high level of abstraction from the underlying computer hardware.

term was first used in the Smalltalk programming language, although Simula 67 is now considered to be the first language to introduce object oriented concepts. The object oriented programming paradigm deals with entities or *objects*, of a certain type or *class*, and the mechanisms of interaction between them. It promotes strong *decoupling* between objects through the use of data and functional *encapsulation*. Furthermore, it allows for appropriate levels of *abstraction* through *inheritance* of classes and using *polymorphism* that allows different objects to appear similar based on a common interface. These features make object oriented programming an extremely valuable paradigm for the development of large scale software, because the separation of concerns allows teams to work on portions of the program in parallel. Object oriented programming dominates large-scale software development methodology in current times. Popular languages in the tradition include Java [Gosling et al., 2005] and C++ [Stroustrup, 1997].

The application of object oriented programming to large-scale systems led to another development that aimed to ease the construction of complex software: that of *agent-oriented programming* [Shoham, 1993]. The new paradigm offered an alternative perspective on computation and communication: a view analogous to society, in which complex objects, called *agents*, co-existed in the system as autonomous entities with cognitive abilities. This new way of conceptualising a program had its advantages in many domains, primarily in robotics and Artificial Intelligence [Russell and Norvig, 2009]. In his founding work, Shoham described an agent as "an entity whose state is viewed as consisting of mental components such as beliefs, capabilities, choices, and commitments," and introduced agent-oriented programming as a "specialization of the object-oriented programming (OOP) paradigm." In broad terms, an agent is analogous to its human counterpart and may be viewed as an entity that exhibits autonomy of operation, a social ability to interact with others, reactivity to changes in its environment, and pro-activeness in pursuing its objectives. Over the past two decades, many agent-oriented languages have been developed including AGENT-0 [Shoham, 1993], AgentSpeak(L) [Rao, 1996] (based on PRS [Georgeff and Ingrand, 1989]), Golog-like languages [De Giacomo et al., 2000, 2009; Levesque et al., 1997], CAN/CANPlan [Sardina et al., 2006; Winikoff et al., 2002], JACK [Busetta et al., 1999], Jason [Bordini et al., 2007], JADEX [Pokahr et al., 2003], GOAL [de Boer et al., 2007; Hindriks et al., 2001], 3APL [Hindriks et al., 1999], 2APL [Dastani, 2008], and GORITE [Rönnquist, 2008]. Of immediate interest and relevance to our work is a particular subgroup be-

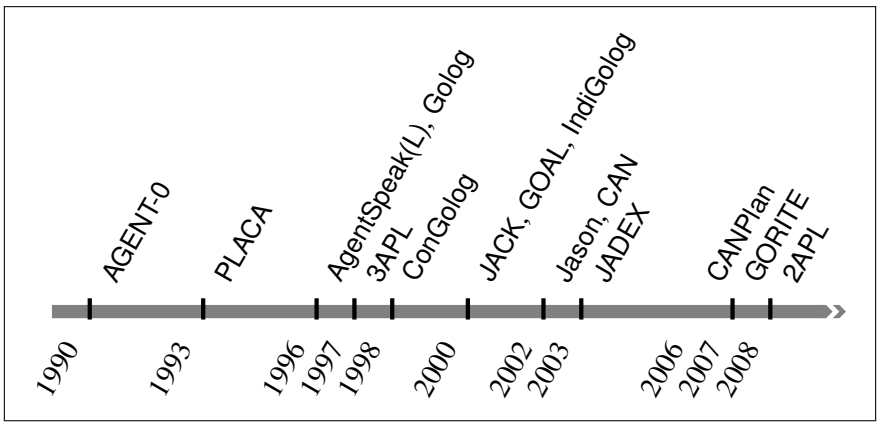


Figure 2.1: A brief history of agent-oriented programming languages.

longing to the Belief-Desire-Intention agent architecture that we will discuss next.

2.1 Belief Desire Intention (BDI) Model of Agency

The Belief-Desire-Intention (BDI) model of agency [Cohen and Levesque, 1990; Rao and Georgeff, 1991, 1992] is a well-studied agent paradigm that has been successfully applied to a wide range of problems over the past two decades [Burmeister et al., 2008; Karim and Heinze, 2005; Rao and Georgeff, 1995]. The BDI model has its roots in the philosophy of mind with Bratman's theory of human practical reasoning [Bratman, 1987; Bratman et al., 1988] and Denett's theory of intentional systems [Dennett, 1987]. Practical reasoning refers to the process of *deliberation* to figure out what a rational agent should do next. As Bratman [1990] puts it: "Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes." At any given time, an agent may hold several *desires* in the world, and only when such desires are committed to do they become *intentions* of the agent. As such, the BDI model is characterised by an informational state (beliefs) about the world, a motivational state (desires) or objectives in the world, and a deliberative state (intentions) or commitments that are current in the world.

Beliefs An agent's beliefs capture what it perceives to be the state of affairs in the world. Beliefs may include factual statements such as 'The ball is under the table.', interpretations such as 'This flower is beautiful.', and statements about mental states

such as 'I feel elated.' Importantly, beliefs are evaluated relative to the agent, unlike knowledge that is irrespective of the observer. Beliefs may or may not represent truth about the world and are free to change over time.

Desires The agent's desires represent the state of affairs that it would like to bring about. This may include wanting to read a book in the park, become a neuroscientist, or eat porridge. Normally an agent will hold several, even conflicting, desires at any one time, and must *deliberate* to decide which ones it wishes to pursue: this subset representing the *goals* of the agent. Bratman draws on the relationship between goals and beliefs to explain rational judgement in [Bratman, 1987]: "an agent adopting a goal to bring about a state of affairs while at the same time believing that the state of affairs is unachievable is acting irrationally; not having a belief about the achievability of the goal (being agnostic about it), however, is not irrational."

Intentions An agent's intentions constitute goals that it is committed to achieving. By commitment a sense of purpose is implied in that the agent must decide how to bring about the intended state of affairs, such as by using a plan of action, or *plan*. The agent may adopt several intentions to pursue, however unlike desires, it will not generally adopt conflicting ones.[4] A plan is simply a reasonable strategy for achieving a goal. Bratman suggests that plans are generally partial and hierarchical since adopted plans only partially specify what is to be done and may invoke other intentions and plans in a hierarchical manner. For instance, a plan to visit the zoo the next day may start with the agent getting organised the night before, and merely holding the intention to get there by noon. Deliberation over how to get there is left for a time closer to departure when, say, the weather conditions are known.

To highlight the difference between an agent's desires and its intentions, Bratman [1990] gives the following example:

> "My desire to play football this afternoon is merely a potential influencer of my conduct this afternoon. It must vie with my other relevant desires [...] before it is settled what I will do. In contrast, once I intend to play basketball this afternoon, the matter is settled: I normally need not continue to weigh the pros and cons. When the afternoon arrives, I will normally just proceed to execute my intentions."

[4] In practical BDI systems it is normally up to the programmer to ensure that new intentions do not conflict with existing ones.

A key message here is that intentions persist once adopted, and the agent makes some attempt to achieve them using a plan of action. For indeed if the agent were to adopt an intention and then drop it without trying, then it would be acting irrationally. This however does not preclude dropping an intention if it is no longer relevant or if achieving it becomes infeasible. Moreover, intentions impact the agent's future course of action which invariably impacts subsequent intentions.

On the whole, the idea behind the BDI paradigm is to *see rational behaviour as the result of the interaction between mental attitudes*.

2.1.1 BDI Formalisms

One of the first computational models to embody Bratman's views on the role of intentions in practical reasoning, was the Intelligent Resource-Bounded Machine Architecture (IRMA) Bratman et al. [1988]. The IRMA model centered on Bratman [1987]'s claim that rational agents tend to focus their practical reasoning on the intentions they have already adopted while bypassing full consideration of options that conflict with those intentions. In other words, intentions provides a *screen of admissibility* for adopting other intentions.

Some aspects of Bratman's theory of practical reasoning were formalised by Cohen and Levesque [1990]. Their work focusses on the role intentions play in maintaining the rational balance between an agent's beliefs, goals and plans. For instance, a rational agent should act on (and not against) its intentions; adopt only those intentions that it believes are achievable; commit to achieving intentions, but not forever; drop those intentions that are believed to have been achieved; and allow for sub-intentions during a plan of action. Although Cohen and Levesque do not explicitly model intentions, they describe them using the concept of a *persistent goal*: an agent adopts a persistent goal that it believes is achievable but not already achieved, and drops it if it is achieved or no longer feasible. However, since these are the only conditions under which the goal may be dropped, the authors label this strong commitment to the goal as fanatical. Plans are also not explicitly defined by this framework, although the authors contend that intentions may loosely speaking be viewed as the contents of plans. This is because the commitments one undertakes with respect to an action in a plan depend on other planned actions, as well as the pre and post conditions brought about by those actions.

A subsequent formalism by Rao and Georgeff [1991] resolved the issue of fanatical commitment by also allowing intentions to be dropped when certain conditions are believed to hold. Further, their formalism expresses intentions as first-class citizens on par with goals and beliefs, unlike the former case where intentions were expressed in terms of goals and beliefs. Another notable shift was in the treatment of outcomes that they modelled as a property of the environment rather than something the agent is free to choose; the agent merely performs actions that it believes will bring about the outcomes. In their framework, the state of the world is captured by belief-accessible worlds, for each of which at any given time there exists a goal-accessible world that is a subset of it, and subsequently an intention-accessible world that is a subset of that. Intuitively, these represent increasingly selective choices about the desire for possible courses of actions.

Rao and Georgeff [1991] provide several axioms for beliefs, goals and intentions. For example, axiom $\mathsf{GOAL}(\alpha) \supset \mathsf{BEL}(\alpha)$ says that if the agent has a goal α, then it must also believe that α is an option, while axiom $\mathsf{INTEND}(\mathsf{does}(a)) \supset \mathsf{does}(a)$ says that if an agent has an intention to perform an action a, then it will perform that action.

While the formalisms by Cohen and Levesque and Rao and Georgeff have a clear semantics, they are unsuitable for practical BDI implementations since they are not efficiently computable [Rao and Georgeff, 1995]. To address this, Rao and Georgeff [1992, 1995] propose an abstract BDI architecture that makes certain simplifying assumptions about the theoretical framework, and models beliefs, goals and intentions as data structures.

Figure 2.2 shows Rao and Georgeff's abstract BDI interpreter [Rao and Georgeff, 1995] that has been the basis for the execution model of such practical systems as PRS [Ingrand et al., 1992], dMARS [d'Inverno et al., 1998], JACK [Busetta et al., 1999] and Jason [Bordini et al., 2007]. Here the global data structures B, G, and I represent the agent's beliefs, goals, and intentions respectively and may be queried and updated as necessary according to the axioms specified in their earlier formalism [Rao and Georgeff, 1991].

The abstract interpreter's processing loop aims to continually resolve a queue of pending events (event-queue) by deliberating over (line 3) and selecting (line 4) appropriate options (opts) for handling those events. Events may be generated externally by the environment, other agents, sensors, or internally to realise belief changes for instance.

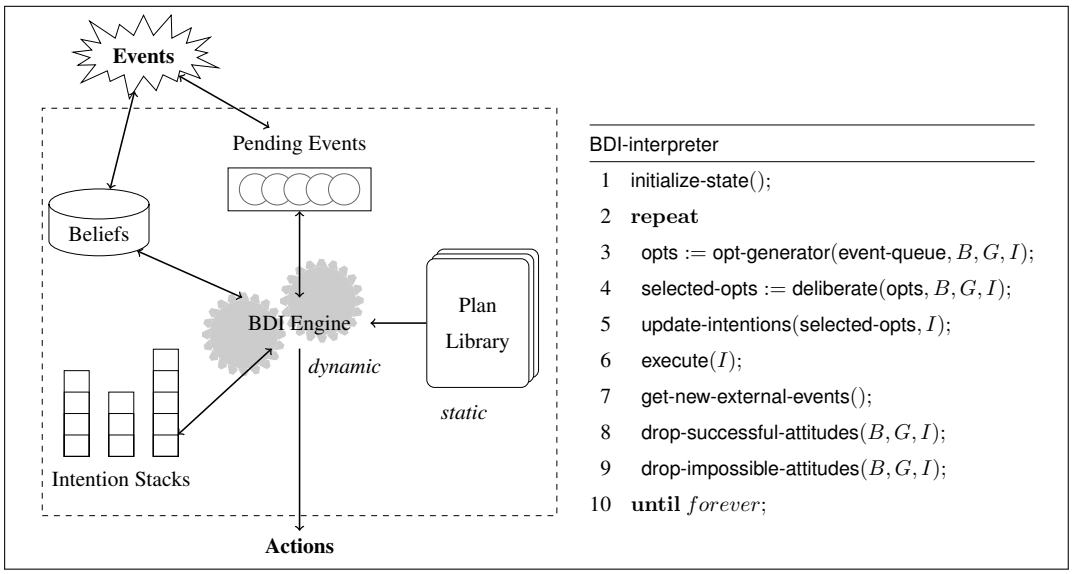


Figure 2.2: A typical BDI architecture.

Options opts constitute procedures that specify task and action sequences and are typically represented in most implementations as *plans*. Plans in themselves are suitable procedures for resolving instances of a particular event type. A plan is considered *relevant* if it was written to resolve the given event type and it is also *applicable* if its invocation condition, or *context condition*, holds in the given situation. For instance, an unmanned aerial vehicle (UAV) controller may contain two plans for landing the plane — one in fine weather and another in a storm when visibility is reduced — and which one of these applies will depend on the current weather conditions.

Next, the interpreter adds the selected options to the intention database I (line 5). The agent then executes a step (line 6) within any intention in I, which may involve executing a primitive action in the world or performing a sub-task by posting other events that in turn get resolved resulting in additional sub-intentions being added to I. Finally, the cycle ends by incorporating any new external events into the event queue (line 7), and removing all satisfied (line 8) and unachievable (line 9) goals and intentions from the respective databases.

This integration of deliberative (triggered by the agents goals and beliefs) and reactive (triggered by the external environment) reasoning in the BDI architecture delivers a responsiveness that is well suited to many complex domains [Burmeister et al., 2008; Karim and Heinze, 2005].

2.1.2 BDI Programming Languages

There are several programming languages that are based in the BDI tradition that implement, in some way or another, Rao and Georgeff's abstract interpreter (Figure 2.2) and the IRMA architecture. These include: AgentSpeak(L) [Rao, 1996] and related languages (including JACK [Busetta et al., 1999], Jason [Bordini et al., 2007], CANPlan [Sardina and Padgham, 2010; Winikoff et al., 2002], JADEX [Pokahr et al., 2003] and GORITE [Rönnquist, 2008]), the GOAL [de Boer et al., 2007; Hindriks et al., 2001] agent programming language, and 3APL/2APL [Dastani, 2008; Hindriks et al., 1999]. JACK is of particular interest to this work since all experimentation and application development was done using this language.

AgentSpeak(L) and Jason

AgentSpeak(L) [Rao, 1996] was developed to provide operational and proof-theoretic semantics to existing practical BDI systems at the time. While on the one hand, it provided semantics that allowed agent programs to be written and interpreted in a manner similar to logic programs, on the other, it borrowed heavily from practical implementations such as PRS [Georgeff and Ingrand, 1989] and its descendant dMARS [d'Inverno et al., 1998]. It was a significant development in BDI programming since it allowed derivations to be performed in the specified logic that could be used to prove various properties of BDI agents implemented in this language.

AgentSpeak(L) is based on a restricted first-order language with events and actions. It does not explicitly model the agent's beliefs, desires, and intentions, but instead the onus is on the programmer to realise these attitudes in the language itself. The language simply allows for facts (or *base beliefs* that are ground atoms in the logic programming sense) and plans (context-sensitive hierarchical event-driven recipes as we have introduced before) to be specified. The state of the agent then represents its beliefs, the states that it wishes to bring about its goals, and the adopted plans to bring about these states its intentions. The operational semantics of the language constitutes sets of beliefs, plans, intentions, events, actions, and selection functions, and a proof theory that describes the transition of the agent from one configuration to another.

More specifically, an AgentSpeak(L) plan P takes the form $+!g : \psi \leftarrow P_1; \ldots; P_n$, where: $+!g$ is the *triggering event* and indicates that event-goal $!g$ is handled by this

plan; ψ is the *context condition* that specifies the runtime conditions under which the plan applies; and each P_i in the plan *body* is either *(i)* an operation $+(-)b$ for adding (deleting) a belief atom b to (from) the agent's set of base beliefs; *(ii)* a primitive action *act*; *(iii)* a subgoal $!g'$, i.e., a state where the formula g' is a true belief; or *(iv)* a test goal $?g'$ to determine if the formula g' is a true belief or not.

Since the initial specification of AgentSpeak(L) in [Rao, 1996], several improvements have been proposed. From an implementation viewpoint, d'Inverno and Luck [1998] give a complete syntax and semantics for AgentSpeak(L) using the Z specification language. Their work provides an explicit representation of (including operations on) states that must be accommodated by any implementation, identifies data structures for operation, and corrects certain errors in the initial specification. Moreira and Bordini [2002] further extend this work by providing complete operational semantics for AgentSpeak(L) including certain features that were omitted in the initial specification such as how to execute the belief add/delete operations. Bordini et al. [2003] subsequently propose a finite version of the language called AgentSpeak(F) that allows guarantees to be obtained for the agent's behaviour with respect to specifications expressed as logical formulae, using model checking [Clarke, 1997]. Finally, Hübner et al. [2006] show how the use of plan patterns in AgentSpeak(L) programs can realise *declarative goals*, i.e., goals that explicitly represent the state of affairs to be achieved, without having to extend the language with new constructs.

Jason [Bordini et al., 2007; Bordini and Moreira, 2004; Moreira et al., 2004; Moreira and Bordini, 2002] is an open-source interpreter that implements the operational semantics of AgentSpeak(L). Jason has a simple language for defining a multi-agent system, where each agent runs its own AgentSpeak(L) interpreter, and where customisations are provided by Java classes. Having formal semantics also allows Jason to precisely specify practical notions of beliefs, desires, and intentions in AgentSpeak(L) agents, enabling use of formal verification to prove properties of the implemented BDI agents.

JACK

JACK [Busetta et al., 1999] is an industrial agent development environment in the BDI tradition that is extensively used in research and industry. It extends the Java language itself, by providing keywords and statements that allow agents and plans to be specified as first class components of the language. The JACK compiler generates Java source

and bytecode, while the runtime environment provides an execution model similar to the abstract BDI interpreter of Rao and Georgeff [1995]. Since its initial specification, the concept of *capabilities* has been added to the language, that allows reasoning components (plans) of the agent to be clustered into separate groups that capture related behaviours. In this thesis, we use JACK for all experimentation and application programming, and we will discuss it again in some detail later in Section 2.1.3.

JADEX

JADEX [Pokahr et al., 2003, 2005] is another Java-based agent development environment similar to JACK. It is built as a layer on top of the JADE (Java Agent Development Framework) platform. JADEX maps BDI concepts to object-oriented concepts and explicitly represents goals in order to allow reasoning over them. Moreover, goals are more closely aligned with Bratmans's theory of desires and intentions [Bratman, 1990] since JADEX allows for conflicting goals (desires) as long as the goals pursued (intentions) are non-conflicting. Goals in JADEX can be of four types: *perform* goals that are concerned with the execution of actions, *achieve* goals that aim to realise a desired external world state, *query* goals that are similar but concerned with realising internal belief states, and *maintain* goals that aim to maintain a desired state. The representation of plans and capabilities is similar to that in JACK.

CAN/CANPlan

CAN (Conceptual Agent Notation) [Winikoff et al., 2002] is an agent development framework that makes the notion of goals explicit in order to allow reasoning over them: such as dropping goals when they have been achieved or become unachievable [Rao and Georgeff, 1992]. It combines the *procedural* view of goals (such as in AgentSpeak(L) where goals are represented as instantiated plans) with the *declarative* view that treats goals as first class citizens together with beliefs and plans. The operational semantics of CAN describe the construct Goal(ψ_s, P, ψ_f) which reads as "achieve ψ_s using plan P; failing if ψ_f becomes true." Here ψ_s and ψ_f are (mutually exclusive) logical formulae over the agent's beliefs that capture the declarative aspects of the goal, and P is the procedural aspect that consists of a set of context specific plans. The key idea is to disassociate the success and failure of the goal from the success and failure of the plan itself. This means that a plan to achieve a goal may complete its execution but

not necessarily achieve ψ_s, therefore requiring further plan choice; whereas the goal may be dropped altogether, including any active plan to achieve it, if the goal becomes unachievable, i.e., ψ_f becomes true. Moreover, unlike AgentSpeak(L) where failure handling is explicitly specified by the programmer using the $-!g$ event, CAN has an in-built mechanism that allows alternative plans to be tried on failure, consistent with practical BDI systems like PRS, dMARS, and JACK.

CANPlan [Sardina et al., 2006] extends CAN with an on-demand hierarchical task networks (HTN) style planning mechanism, using a new language construct Plan(P) which means "plan for P offline, searching for a complete hierarchical decomposition." Finally, Sardina and Padgham [2010] extend CAN with details for dropping goals, pro-actively instantiating goals, and handling variables rather than being restricted to a propositional language.

GORITE

GORITE (Goal Oriented Teams) [Rönnquist, 2008] combines the goal-driven execution model of BDI systems with the team-driven view of cooperating agents, into a Java-based framework that is oriented towards large-scale software development. The design process consists of creating goal-plan hierarchies that describe a breakdown of tasks (goals) into sub-tasks (subgoals) and the procedures (plans) to accomplish them. The team view is represented in the notion of "roles" allowing behaviours to be described in terms of the team organisation and irrespective of any individuals. This view results in a de-coupling between the skill-sets of agents and the skill-set requirements of tasks.

GOAL

A comparable attempt to AgentSpeak(L) that was also aimed at consolidating BDI theory and practice, was initiated by Hindriks et al. with the GOAL agent programming language [de Boer et al., 2007; Hindriks et al., 2001]. GOAL takes the *declarative* concept of goals seriously and allows for goals to be programmed in the same way as beliefs in a propositional logic language. The authors provide a complete programming theory over the GOAL programming language including its formal operational semantics and a proof theory, based on temporal logic, that enables reasoning about the beliefs and goals of the agent in any state during its execution. The semantics of the logic is provided by

the GOAL agent semantics which guarantees that properties proven in the logic are also properties of the GOAL agent.

Actions of the agent are of the form $\phi \rightarrow do(a)$, where a is an action derived from the agent's *capabilities*, i.e., basic operations that update the agent's belief (but not goal) base, and ϕ is a *mental state* condition that specifies when the action applies (assuming that it is enabled in ϕ as specified by the partial function $\mathcal{M}(a, \phi)$). This bears semblance to the way that a context condition in AgentSpeak(L) specifies when a plan applies. A key difference between GOAL and AgentSpeak(L) apart from the use of declarative goals, is that in GOAL the agent's behaviour is specified by actions whereas in AgentSpeak(L) it is specified by procedures (plans).

In recent years, several improvements have been proposed for the GOAL agent programming framework. In [Hindriks, 2008] the notion of modules was introduced, similar to the notion of capabilities in JACK, that provides a means for the agent to focus attention on only those aspects of its behaviour that are relevant to the situation. This in turn helps to minimise the inherent non-determinism that is typical of agent programs. In [Hindriks et al., 2009b], the GOAL programming language was augmented with temporally extended goals that allow the agent to reason about a desired *sequence of states* rather than simply the desired set of *final states*. In [Hindriks et al., 2009a], new programming primitives were introduced that allow for the specification of utility-based heuristics for action selection. The process involves associating a quantitative number, or *utility*, with the execution of an action that represents how much value is to be gained from executing that action. This utility may also be viewed as the sum of the *cost* associated with taking an action in the starting state and a *reward* associated with getting to the resulting state. The overall idea is to improve action selection by prioritising based on programmer specified utilities. Finally, in [Broekens et al., 2010] an alternative approach for prioritising action selection was proposed: through the use of reinforcement learning to learn rule selection in different situations in a domain-independent manner.

3APL and 2APL

3APL [Hindriks et al., 1999] is a popular agent programming language that originated around the same time as (see Figure 2.1), and offered an alternative to, AgentSpeak(L). The authors have shown in [Hindriks et al., 1998] that the concepts of event and intention in AgentSpeak(L) are formally mappable to 3APL goals, and that it is possible

to specify the former in latter terms since 3APL can simulate AgentSpeak(L). They conclude that 3APL strictly has more expressive power than AgentSpeak(L) since it provides a mechanism for goal revision, that is not mappable to the latter. The main difference between the control structures of two languages is that the 3APL-equivalent interpreter for the abstract BDI-Interpreter of Figure 2.2 effectively has an additional filtering step: 3APL allows for failure handling rules that may be used as an additional check to prevent as much failure as possible. This notion of preventative failure rules does not exist in AgentSpeak(L). Over the years, 3APL has been extended to include declarative goals [Dastani et al., 2004; van Riemsdijk et al., 2003].

More recently, the 2APL [Dastani, 2008] agent programming language has been proposed, that extends 3APL with programming constructs for multi-agent systems. A second difference is that in 2APL the semantics of the failure handling rules are more in line with AgentSpeak(L) than 3APL since they apply only when the execution of the initial plan fails.

Related Languages and Frameworks

"Golog-like" languages [De Giacomo et al., 2000, 2009; Levesque et al., 1997] are based in the *situation calculus* [McCarthy and Hayes, 1969; Reiter, 2001]; a logical formalism for representing and reasoning about dynamic environments, where a dynamic world is modeled as a progression through a series of *situations* that result from *actions* being performed. Where Golog [Levesque et al., 1997] (the first situation calculus-based agent programming language) and its successor ConGolog [De Giacomo et al., 2000] (that added support for concurrency) were designed to be executed *offline* (i.e., the complete execution of the program is determined upfront prior to taking the first action), the latest addition IndiGolog [De Giacomo et al., 2009] allows programs to be executed *online* by interleaving acting with on-demand local planning (offline) as required. The work of Sardina and Lespérance [2010] shows that given any execution of a BDI agent, there exists an equivalent execution of a corresponding IndiGolog agent, and vice-versa.

Prometheus [Padgham and Winikoff, 2002], while not strictly a language, is a methodology for developing agent systems that is actively used in the design and implementation of BDI systems. It follows a three step process: a specification phase to capture the system functionality, an architectural design phase that identifies the agents in the

system and their interactions, and a detailed design phase that looks at the functionality of each identified agent. While a majority of Prometheus-based implementations have been in JACK languages like Jason are also perfectly suitable for that purpose. In [Bordini et al., 2005], the authors suggest that since AgentSpeak(L) code is considerably more readable than other languages such as JACK and JADEX, that Jason arguably provides a more intuitive way of implementing Prometheus designs. They further suggest that the Jason solution is more elegant than JACK or JADEX as it provides a clean interface for integrating agent actions as Java functionality within the formal framework. Another platform for agent-based development is the Agent Factory Framework [Muldoon et al., 2009; O'Hare, 1996]. Apart from providing a fully integrated development environment, Agent Factory also allows for different agent programming languages to be integrated using a common language framework. An example is AF-AgentSpeak, that is an implementation of the AgentSpeak(L) language for Agent Factory.

Other works include the actor model [Hewitt et al., 1973] and process algebras [Agha et al., 1997; Gaspari and Zavattaro, 1999; Hoare, 1978; Milner, 1982, 1999] that are formal approaches to modelling concurrent systems. Here, an *actor* is a universal primitive for concurrent decision making. An actor receives messages asynchronously, and responds to messages by making local decisions that can create new actors, send messages to other actors, and determine how to respond to subsequent messages. Systems in this tradition include extensions to the Smalltalk object-oriented programming language such as Actalk [Briot, 1989] and Concurrent Smalltalk [Yokote and Tokoro, 1987], and concurrent functional programming languages such as Erlang [Armstrong et al., 1993] and Scala [Odersky and al., 2004], among others [Karmani et al., 2009; Varela and Agha, 2001]. More closely related to our work in this domain are the MobileSpaces [Satoh, 2000] and CLAIM [Fallah-Seghrouchni and Suna, 2004; Fallah Seghrouchni and Suna, 2005] agent programming platforms that are inspired by the ambient calculus which incorporates concepts for mobile actors. In both systems, actors are organised in hierarchies that loosely resemble the knowledge structure of the BDI plan library, and the techniques that we develop in this thesis can also be applied in this setting.

Finally, in automated planning (i.e., "look-ahead" reasoning) literature, the work in hierarchical task network (or HTN) planning [Erol et al., 1994; Nau et al., 2005] is known to be closely related to BDI programming [de Silva and Padgham, 2005; de Silva, 2009; Sardina et al., 2006]. We discuss this relationship and related works in

HTN learning separately in Chapter 7 Section 7.1.

2.1.3 JACK Intelligent Agents

JACK [Busetta et al., 1999] is a commercial agent programming language that integrates BDI concepts on top of the Java programming language. JACK is closely related to BDI implementations like PRS [Georgeff and Ingrand, 1989] and dMARS [d'Inverno et al., 1998], but unlike Jason [Bordini et al., 2007], does not have formal semantics. This means that it is not possible to reason about the beliefs and goals of JACK agents in a formally verifiable way. Instead, the onus is on the programmer to ensure that the consistency of an agent's state is maintained when performing state updates. For instance, when an agent updates some belief based on an event, then the programmer must ensure that this update does not make its world-view inconsistent.

JACK was designed with large-scale software applications in mind where agents must co-exist and communicate with other legacy components that are not inherently agent-based. It offers a complete programming environment including *(i)* programming constructs that extend the Java programming language to include BDI concepts like agent, plan, event, etc.; *(ii)* a compiler to parse the JACK language into pure Java language source code; and *(iii)* a set of classes that provide run-time support such as the management of concurrent intentions, default behaviours, and an infrastructure for multi-agent communication.

A JACK agent is described by its beliefs, the events (internal and external) that it processes, its library of plans to handle such events, and any set of utility Java classes to interact with the external environment. Plans are similar to AgentSpeak(L) plans in that they have a triggering *context condition*, and if selected, execute a series of procedural steps such as primitive actions, subgoals, and belief tests. In addition, since JACK is a practical BDI system, it provides a variety of additional features such as *meta-level reasoning* that allows plan selection rules to be specified by the programmer, and *maintenance* conditions that provide a mechanism for triggering a response when some monitored conditions are no longer true.

Figure 2.3 shows an example JACK agent for a travelling domain. The agent maintains certain beliefs about the weather outlook and the amount of money at hand, and has

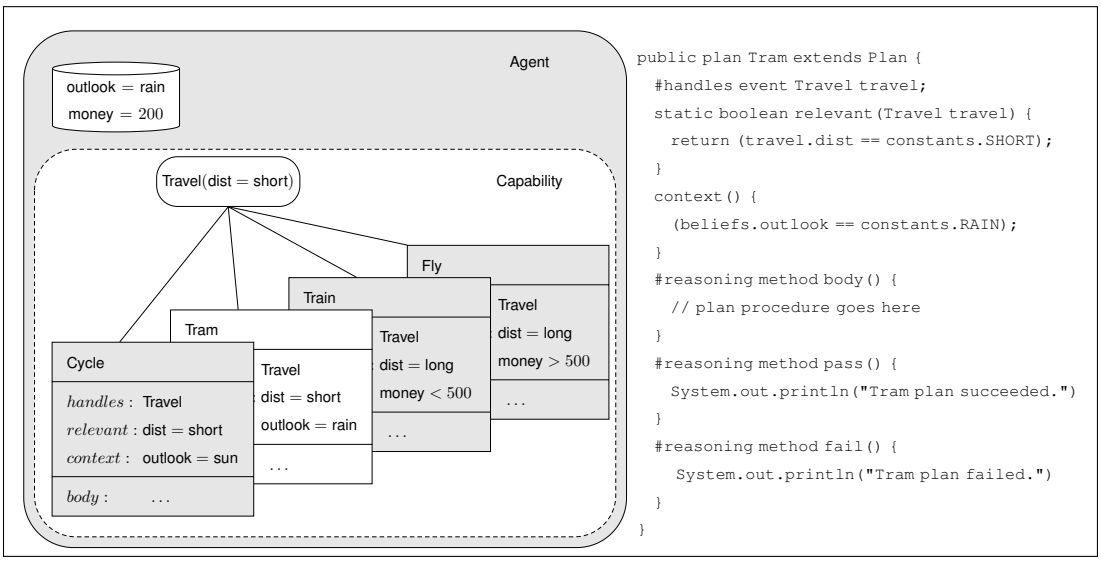


Figure 2.3: An example JACK program showing a listing of the Tram plan.

a related travelling capability that it employs to select between several short and long-distance travel plans based on these beliefs.

The agent has four plans (i.e., Cycle, Tram, Train, and Fly) enclosed within a travelling capability (a means of grouping similar behaviours together). Each plan was written to resolve events of type Travel as highlighted by the $handles$ JACK keyword in the plan. This means that whenever an instance of the Travel event occurs, the agent will consider these plans as candidate responses.

In deciding which plan to choose from the list of candidate plans, the agent executes each plan's $relevant()$ method to determine whether the given plan is also relevant to the event instance. A plan is considered relevant if, and only if, it handles the given event and the event's parameters match the pattern specified in the plan's $relevant()$ method. This provides a way to refine the candidates list based on the event instance. In this example, only the Cycle and Tram plans are relevant for short distance travel (i.e., for Travel event parameter dist $=$ short), while the Train and Fly plans are relevant for long distance travel only (i.e., dist $=$ long).

Each plan's $context()$ method provides a second filter to determine if a candidate plan should be executed in the current context. The context specifies a logical condition that must be satisfied if the plan is to be considered applicable for handling the event instance

in the current situation. Normally, the agent will query its beliefset that contains its beliefs about the world in order to determine this. In this example, the Cycle and Tram plans apply under different weather conditions (i.e., sun and rain respectively), while the Train and Fly plans apply for different values of money at hand (i.e., money < 500 and money > 500 respectively).

The figure highlights the only plan whose $relevant()$ and $context()$ conditions are satisfied (i.e., Tram) and therefore will be chosen, given the agent's current beliefs about the world (i.e., outlook $=$ rain and money $= 200$). Note that it the general case the final applicable set may contain several plans and not just one, and in such cases the agent must decide which plan to choose. JACK provides default selection schemes for this purpose: pick the first entry in the list or pick an entry at random. When using the former, the ordering in the list is decided using prominence (i.e., according to the order in which plans are declared) or precedence (i.e., based on a calculated rank). However, if required, JACK's meta-level reasoning functionality allows programmer specified schemes to be used. For instance, in our framework, a customised scheme is provided that selects plans *probabilistically* based on the learnt likelihood of success of the candidates.

Finally, a plan's $body()$ method describes the procedure that the agent will follow whenever it executes an instance of that plan. In our example, this may be assumed to consist of a sequence of steps (i.e., primitive actions, subgoals, and so forth) that the agent will follow to make the actual trip based on the mode of transportation.

For the purpose of this thesis, any reference to a plan's context condition should be interpreted as a reference to the combined $relevant()$ and $context()$ filter in JACK.

2.2 Decision Tree Learning

Learning, in the general sense of the word, like intelligence and knowledge, may be difficult to define precisely. For the purpose of this thesis, therefore, we will narrow down the scope to something more manageable. Learning, in our context, refers to the discipline of *machine learning* that is concerned with developing computational models of learning in machines [Mitchell, 1997]. Put simply, machine learning is the study of computer algorithms that improve automatically through experience. Formally, when

we refer to learning, we imply the following definition [Mitchell, 1997]:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."*

For example, a program that learns to play chess might improve its performance *as measured by its ability to win* at the class of tasks involving *playing chess games*, through experience *obtained by playing games* against a human player.

In particular, we are interested in a branch of machine learning called *decision tree learning* [Mitchell, 1997; Quinlan, 1986, 1993] that involves the use of decision trees to make conclusions based on a history of observations. Our choice of decision trees for learning is motivated by several factors. Firstly, decision trees support hypotheses that are a disjunction of conjunctive terms and this representation is compatible with how context formulas are generally written. Secondly, decision trees are robust against training data that may contain errors. This is specially relevant in stochastic domains where applicable plans may nevertheless fail due to unforeseen circumstances. Finally, decision tree learning is a well-developed technology: it has several competitive implementations and a mature theory behind it.

A decision tree may be viewed as an tree-like flowchart, where each node represents a decision point, i.e., a test for some attribute, and each outgoing branch represents a possible value of that attribute. Each path from the root node to a leaf node constitutes a decision path, and terminates in a categorisation, or *classification*.

Figure 2.4 shows an example decision tree for the travelling domain for deciding whether to travel by tram or not. Here, the decision paths terminating in the $\surd$ (or $\times$) classification indicate the final decision based on the chosen attribute values. In this example, travelling by tram is a good idea for short distances in wet weather, i.e., $(\mathsf{dist} = \mathsf{short} \wedge \mathsf{outlook} = \mathsf{rain})$ but not otherwise, i.e., $(\mathsf{dist} = \mathsf{long}) \vee (\mathsf{dist} = \mathsf{short} \wedge \mathsf{outlook} = \mathsf{sun})$.

The typical use of decision trees is for generalising from past experiences to categorise unseen situations. For instance, one may recall several ways in which to entertain chil-

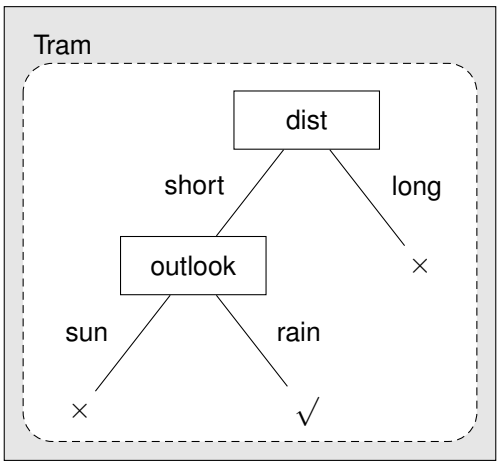


Figure 2.4: A decision tree for the travelling domain to decide if one should travel by tram.

dren, such as by taking them to the park, reading books, and watching trains go by. If one were to use these past experiences to guide their decision making when interacting with a new child, then this would constitute an inductive decision making process. A key concern here is determining *how* to construct the decision tree for deciding what activity to perform with the child.

The problem of learning decision trees may be described as follows: *given a set of examples, each described by a set of attributes and a known categorisation, the task is to learn the structure of a decision tree that correctly classifies the examples and may be used to decide the category of an unseen example.*

Putting together a decision tree is a matter of choosing attributes to test at each node in the tree. The key is to decide which attributes to test first since the order in which various attributes are tested will invariably impact the size of the final tree. Intuitively, we would like to test the most important attributes first. However, since examples are generally not annotated with information about the importance of the attributes, we require a more generic method for determining this. One way to do this is by looking at how different combinations of attribute values impact the categorisation.

As an example, consider again the travelling problem where we would like to decide the best mode of transportation for any given situation. Say we decided to cycle to work, but it rained on the way and so we concluded that that was an unsatisfactory outcome. Now,

we may record this categorisation, i.e., unsatisfactory, against the situation in which we made the decision, i.e., the values of such attributes as money $= 200$, outlook $=$ rain, day $=$ Monday, and so on. However, simply by considering this one experience we cannot determine which attribute(s) actually contributed to the unsatisfactory outcome. If, on the other hand, we had several experiences of cycling to work under different situations, then by analysing them collectively we may justifiably conclude that the weather outlook was indeed most influential to the outcome.

For the purpose of this thesis, we use the algorithm J48, a version of Quinlan's C4.5 [Quinlan, 1993] algorithm for inducing decision trees, from the weka learning package [Witten and Frank, 1999]. The basic algorithm conceptually performs a similar analysis to our example above by calculating the *information gain* of an attribute with respect to the set of examples. It then *(i)* places the attribute with the highest information gain at the root of the decision tree; *(ii)* creates a branch for each observed value of that attribute; *(iii)* assigns the relevant examples to each branch; and *(iv)* repeats the process for each subset thus created. The end result is that the attributes that contribute the most to the outcome are placed earlier in the decision path, and are considered first when evaluating a new situation.

Assuming consistent data, i.e., where no two examples have the same values for the attributes but are categorised differently, it is always possible to construct a decision tree that correctly classifies the training cases with complete accuracy. However, full accuracy in itself may not be a valid measure for the usefulness of the decision tree if the data is incomplete, and may indicate *overfitting*, i.e., where the decision tree performs well on the training data but does not generalise well to unseen data.

Approaches to address overfitting in decision trees broadly aim to do one of two things. They *(i)* either stop growing the tree earlier i.e before it perfectly classifies all training samples; or *(ii)* allow the tree to grow fully but then *prune* it afterwards: this latter being generally considered to be more effective [Mitchell, 1997]. Overall though, the induction process will trade-off some accuracy in classification for compactness of representation.

2.3 Related Work in BDI Learning

The issue of combining *online* learning with deliberation in BDI agent systems has not been widely addressed in the literature. In terms of *offline* approaches, Guerra-Hernández et al. [2005] reported preliminary results on learning the context condition for a single plan using a decision tree in a simple paint-world example, although they do not consider issues of learning in plan hierarchies, non-deterministic domains, and nuances such as the presence of noisy training data, all of which we address in this thesis. The work in [Lokuge and Alahakoon, 2007] gives a detailed account using a real-world ship berthing logistics application. The authors take operational shipping data to train a neural network offline that is then integrated into the BDI deliberation cycle to improve plan selection. They show that the trained system is able to outperform the human operators in terms of scheduling the docking of ships to loading berths. Similar approaches integrating previously (offline) learnt knowledge with BDI deliberation have also been used in robotic soccer [Brusey, 2002; Riedmiller et al., 2001], although no new learning is done in the deployed system. In [Nguyen and Wobcke, 2006] learnt user preferences are incorporated during BDI plan selection in a dialogue manager application using a decision tree learner. In contrast, [Karim et al., 2006] take the approach of refining existing BDI plans or learning new plans as a sequence of recorded actions based on prescriptions provided by the domain expert.

A closely related area to BDI is that of hierarchical task network (HTN) planning where task decompositions used are similar to BDI goal-plan hierarchies [Erol et al., 1994]. Particularly, we are interested in the fact that BDI and HTN systems map quite well to each other, and that plans' context conditions in BDI systems are synonymous with methods' preconditions in the HTN case. We explore several related works in this area in some detail later in Chapter 7. The key difference between learning in HTN systems and our BDI approach, however, is that in our case learning is performed online in a trial-and-error manner since we do not have a model of the environment, whereas in HTN planning systems it is predominantly done offline and a model of the environment is assumed. As such, the issue of determining confidence in the ongoing learning (Chapter 4) that may not be reliable due to insufficient data, is generally not a concern in HTN systems.

The work of Simari and Parsons [2006] has highlighted the relationship between BDI and Markov Decision Processes on which the reinforcement learning literature is founded.

Recently, Broekens et al. [2010] reported progress on integrating reinforcement learning to improve plan selection in GOAL, a declarative agent programming language in the BDI flavour. They use an abstract state representation using only the count of action rules and a sum cost heuristic that captures the number of pending goals. The intent is to keep the representation domain independent, with the focus on improving the plan selection functionality in the framework itself. In that way, their approach complements ours, and may be integrated as "meta-level" learning to influence the plan selection. We note that such work is still preliminary and it is difficult to ascertain the generality of their approach in other domains. Nevertheless, their early results are encouraging in that the agent always achieves the goal state in less number of tries with learning enabled than without. Our work also relates to the existing work in hierarchical reinforcement learning [Barto and Mahadevan, 2003], where task hierarchies similar to those of BDI programs are used. We discuss this related area further in Chapter 7. Of particular interest is the early work by Dietterich [2000] that supports learning at all levels in the task hierarchy (as we do in our learning framework described in Chapter 3) in contrast to waiting for learning to converge at the bottom levels first.

To our knowledge, the first attempt at a principled integration of online learning in BDI systems was started within our own research group by Airiau et al. [2009], where the use of decision trees for learning plan selection in BDI systems was initially introduced. Their work explored the nuances of learning within the hierarchical structure of a BDI program, and showed that it can be problematic to assume a mistake at a higher level in the hierarchy, when a poor outcome may have been the result of a wrong decision at lower levels. That research formed the starting point for this thesis, and the learning framework described here builds upon this earlier work.

Chapter 3

A BDI Learning Framework[†]

In this chapter we discuss the elements that constitute our BDI learning framework. Our learning task is one of plan selection, in that we would like our BDI agent to improve its plan selection in any situation based on ongoing experience. Our approach to this is to learn to refine the applicability or *context* conditions of plans over time.

To this end, we provide a new account of plans' context conditions to include decision trees. The idea is that as more experiences are collected regarding outcomes under different situations in which a plan was selected, the induced decision tree from those samples will provide a meaningful generalisation of the real applicability conditions of that plan. We present the key mechanisms that are required for this scheme to function: first, an approach to determining the input for the decision trees and recording the input experience samples from the hierarchy of decisions in the BDI plan library; and second, a new selection scheme that probabilistically selects from the candidate plans based on each plan's believed likelihood of success in the situation as given by its decision tree. Next, we discuss learning in the context of the BDI goal failure recovery mechanism, and in recursive goal-plan structures.

We conclude with a discussion of an important challenge in this setup: that of the reliability of ongoing learning. Since the decision trees we use for plan selection are built from ongoing experiences, then initially the decision trees will not be so reliable. Our solution for this issue of confidence in ongoing learning is given separately in

[†] Parts of the work presented in this chapter have appeared or will appear in [Airiau et al., 2009; Singh et al., 2010a,b, 2011].

Chapter 4.

What Causes Plan Failure?

In saying that we wish to improve plan selection in any situation we imply that we would like to avoid, as much as possible, plan selections that lead to failures. If we are to learn in a meaningful way from failures, it becomes important to also understand the reasons for such failures.

As described previously in Chapter 2, the context condition of a plan encodes the programmed applicability conditions in which the plan is considered to be a reasonable strategy to address a given event-goal. The agent's plan library captures the "know-how" information about the domain that the agent operates in and is specified by the domain expert. So, given that a plan's selection in a given situation implies applicability in that situation, why should the chosen plan fail? It may be for one of the following reasons:

1. The plan was a bad choice in the situation. This may happen if there is a mismatch between the programmed context condition and the real applicability conditions of the plan. In other words, the context condition does not fully capture the state of affairs of the world.

2. The plan was the correct choice in the situation but the environment changed during plan execution. In other words, the reasons for executing the plan changed, while the plan was executing. *This is perhaps the most common reason for failure in a dynamic environment, and is also the motivation behind the BDI failure recovery mechanism.*

3. The plan was the correct choice in the situation but nevertheless failed due to unknown reasons. It may be that the world is only partially observable in which case the reasons for failure are non-deterministic.

4. The plan was the correct choice in the situation but a poor plan choice was made further below in the goal-plan hierarchy. Since plans often post subgoals that are then addressed by further plan choices, it may be the case that the failure occurred at the sub-task level.

5. The plan was a correct choice for addressing the event-goal and all choices in the hierarchy below were also correct, but the way in which prior event-goals were resolved meant that there was no way for the plan to succeed. This may occur, for example, when two subgoals interact over some common resource, such that the resolution of the first subgoal depletes the resource in a way that makes the second subgoal impossible to achieve.

For instance, consider the example of an agent controller for an unmanned aerial vehicle (UAV) that may contain several plans in its library to address the event-goal of landing the airplane. While some plans may apply in normal weather conditions, others may apply only in what are classified as emergency situations.

It may be that a plan to land the UAV in a field in case of an emergency fails because, despite the programmer's best attempts, it was not possible to craft its context condition to capture every situation that constitutes an emergency (reason 1).

Even if the plan was activated correctly in an emergency, it may be aborted during execution if the agent no longer believes that landing in the field is an option, perhaps based on new sensor data confirming risk to farm animals in the field below (reason 2). In this case, if an alternative exists, for landing on a nearby airstrip for example, then the agent could recover from the initial failure by trying this alternative. Otherwise, if no alternatives remain then it might have to abort the goal to land safely.

That is not to say that landing on a nearby airstrip could not fail for all sorts of unknown factors beyond its control (reason 3). Or, it could potentially fail causing the plane to overshoot the airstrip, because the subgoal to determine final approach speed was incorrectly resolved for the current weather conditions (reason 4).

Finally, consider the case where a landing event-goal is the final subgoal in a higher level plan to survey the landscape and where the prior subgoals are used for navigating a set of waypoints in the flight path. It is foreseeable that the UAV successfully navigates all waypoints, but in the process consumes too much fuel, making returning to base and landing the plane unachievable (reason 5).

3.1 Augmenting Context Conditions with Decision Trees

A plan's context condition is a logical formula that encodes its applicability conditions as specified by the domain expert at design time. When the context condition holds in a given situation, that plan is considered applicable for that situation. From the perspective of improving plan selection, we treat the encoded conditions as *fixed and necessary but possibly insufficient* applicability conditions for the plan. The aim is to use learning to *refine* and not replace the original conditions. For instance a domain expert may design the context condition of a plan for landing the UAV based on some standard operating knowledge. After deployment, however, the UAV may be able to adapt its procedure (e.g., approach speed and angle) for landing on a particular airstrip.

To achieve this we augment a plan's context condition with a decision tree that we will learn over time. The idea is that the decision tree induced from the set of current experiences will represent our best estimate of the real conditions under which the plan applies. In some sense then, this new account of a plan's applicability may be viewed as a two step filter: an initial (static) programmed filter that restricts the set of worlds where the plan may apply, and a second (dynamic) learnt filter that possibly further restricts this set based on ongoing experience.

The decision tree inductive bias gives preference to smaller trees. In other words, the induction process will trade-off some accuracy in classification for compactness of representation. What this means is that of the full training set used to induce the tree, some samples may be incorrectly classified in the wrong "bucket" (where the bucket name is *success* or *failure* in our case) such that the actual outcome class of those training samples is different. On dissecting an induced tree in our setting for instance, we may find that of the several samples, say m in total, that got classified as *success*, a portion, say n in number, should have actually been classified as *failure*. This ratio $1 - (n/m)$ then, gives the likelihood that a sample (i.e., situation) classified as *success* (i.e., will succeed) is classified correctly, and is the value we use when we talk about the *expected likelihood of success of the plan as given by its decision tree*. [1]

[1] In our study we use algorithm J48, a version of C4.5 [Mitchell, 1997], from the weka learning package [Witten and Frank, 1999] that automatically provides this ratio.

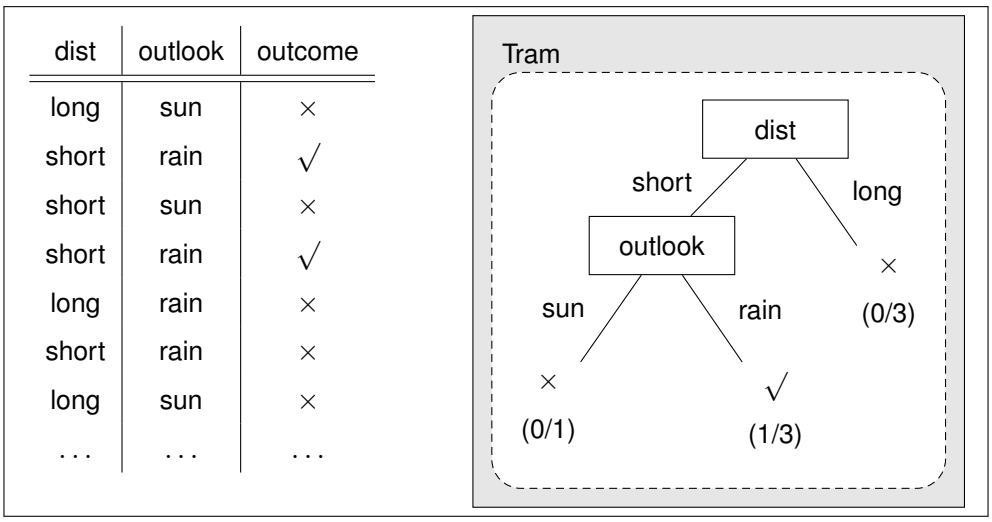


Figure 3.1: An example decision tree to decide if one should travel by tram, based on observed outcomes over time.

Figure 3.1 shows an induced decision tree for a plan to travel by tram, based on observations over a seven day period. The result (i.e outcome $\surd$ (yes) or $\times$ (no)) indicates if we succeeded in reaching our destination on time, and depends on the distance to travel (i.e., short or long) and the weather outlook (i.e., sun or rain). The numbers below the decision nodes in the tree represent the ratio of incorrectly classified (i.e n) to total samples (i.e., m) in that branch.

For instance, the decision tree predicts that the plan for travelling by tram should succeed when the distance to travel is short and the outlook is rainy, as given by the branch $(\mathsf{dist} = \mathsf{short}) \wedge (\mathsf{outlook} = \mathsf{rain})$. The number (1/3) below the branch indicates that a total of three samples from the training set (also shown) were classified this way, out of which one sample was misclassified. If we look at the training samples, we can see that this is because our data is inconsistent: indeed travelling a short distance by tram on a rainy day does not always get us to our destination on time. The likelihood of the plan succeeding for a short trip on a rainy day is therefore $1 - (1/3)$, i.e., 66%.

State Representation for Decision Trees

For each plan, the training set for its decision tree contains samples of the form $[s_w, r]$, where s_w is the representation of the world state w in which the plan was executed, and r was the outcome (*success* or *failure*). The representation s_w itself is a set of discrete attributes that together represent the state of affairs in w. As the agent tries the plan in different situations and records each result, the hope is that over time the decision tree induced from these recordings will contain only those attributes of w that are relevant to that plan's real context condition. Overall, the attributes in s_w belong to the following three sets:

Environment Features These are variables that describe the state of affairs in the external world, and represent features of the world that are independent of any agent. For instance, the fluent outlook=rain might describe the situation that the current weather outlook is rainy. This set of features of the world is *common to all plans*.

Event-Goal Parameters Parameterised event-goals, such as $G(\vec{x})$ where $\vec{x}$ are the parameters of an event-goal type G, are used extensively in practical BDI systems for passing data and control information. A general treatment of event-goals should consider the event-goal type and allow solutions to be learnt over different *instances* of it. For instance, an event-goal Travel(dist) might specify a goal to travel a given distance. Here, we may be interested in learning how to handle different instances of this event-goal, such as Travel(dist=short) and Travel(dist=long). We include such an account by augmenting the training samples for the decision tree with the event-goal parameters. The set of variables $\vec{x}$ will generally *differ between plans for handling different event-goals*.

Plan Variables Often the programmed context condition of the plan will include variable bindings and tests on those bindings to determine its applicability. The applicable set in a situation then may contain *multiple instances of the same plan* with different bound values that satisfy the condition. If we wish to learn the impact of the different bindings on the actual success of the plan, then such variables must be included in the plan's decision tree. For instance, to handle the Travel(dist=short) event-goal we may have a Tram plan that dictates taking a tram going west, bound to variable t in the context condition say. To learn how suc-

cessful the strategy is in getting us to our destination in time, we may be interested in learning over different bindings of t as some trams travel express while others stop frequently. A set of such variables is particular to a given plan, and may *differ between plans to handle the same event-goal*.

Observe how the sets are increasingly specialised: the environment features are common across all plans, the event-goal parameters are particular to plans that handle the same event-goal type, and plan variables are particular to a given plan type.

As an example, consider the UAV controller that may have the following two plan rules as part of the landing procedure:

$\mathsf{AttemptLanding} : \mathsf{GetNearestAirstrip}(a) \wedge (\mathsf{Dist}(a) < \mathsf{FuelRange}()) \leftarrow \mathsf{LandOnAirstrip}(a)$

$\mathsf{FinalApproach}(a, m, s) : \mathsf{Dist}(a) < 1000 \leftarrow \mathsf{FixSlopeAndSpeed}(m, s)$

The first plan resolves the landing request (event-goal AttemptLanding) by determining the closest airstrip a (bound using $\mathsf{GetNearestAirstrip}(a)$), then checking if it has enough fuel to travel the distance (condition $(\mathsf{Dist}(a) < \mathsf{FuelRange}())$). If it does, it proceeds by posting the subgoal $\mathsf{LandOnAirstrip}(a)$ to attempt landing on airstrip a. Now, it may well be that some airstrips are harder to land on than others (as measured by various sensor readings on board the UAV). If we wish to learn the likelihood of landing successfully over time, then the bindings for plan variable a over different executions of the plan must be included in the training samples for the plan's decision tree.

The second plan handles the final approach goal $\mathsf{FinalApproach}(a, m, s)$ for landing the plane on airstrip a, when the distance to the airstrip becomes less than 1000 metres. When executed, it posts a subgoal $\mathsf{FixSlopeAndSpeed}(m, s)$ for fixing the final approach angle m and approach speed s for landing. Clearly, the parameters m and s will impact the landing outcome on airstrip a. If we wish to learn this information then the event-goal parameters m and s should also be included in the plan's decision tree.

It is also reasonable to assume that the current weather conditions may impact the success of the plan. Those features of the world that capture this information, such as outlook=rain, should therefore be included in the state representation. Observe that outlook is not part of either plan rule. In this case, the decision to represent it as part of the world state in this application is based on the domain knowledge of the programmer.

Overall, the number of attributes initially included in the state representation s_w and their range has a bearing on the size of the training set required to correctly learn the context condition. Normally, a plan will not be tried in all states because earlier plans in a sequence of executions will mean that it is reached only in a subset of possible states. For example, a plan for fixing the final approach of a UAV will only be considered in situations when the plane is executing a landing procedure, and not in situations where it is taking off or standing still on the ground. This means that a plan will generally run, and learning occur, in only a meaningful subset of the full state space described by s_w.

3.2 Recording Plan Outcomes for Learning

To allow us to easily discuss some of the details of the framework, we will first introduce two notions.

BDI Goal-Plan Hierarchy The first is the idea that the BDI plan library can be viewed as a tree-like hierarchy. Consider the example goal-plan structure of Figure 3.2. Here all plans (for instance, P_f, P_g and P_h) that are *relevant* for achieving a given event-goal (e.g., G_3) are grouped together as children nodes of that event-goal. Similarly, a plan node (e.g., P) has children nodes representing the subgoals (e.g., G_1 and G_2) of that plan that are to be resolved in sequence from left to right. [2] The goal-plan structure may be seen as an AND/OR tree: for a plan to succeed all of the subgoals and actions must succeed in sequence (AND relationship), while for a subgoal to succeed any one of the plans to achieve it must succeed (OR relationship). Finally, leaf plans are those that interact directly with the environment by performing primitive actions only, so in a given world state they may either succeed or fail, as shown in Figure 3.2 by the $\surd$ and $\times$ symbols respectively.

Active Execution Traces The second notion is of an *active execution trace* to describe the decisions in the BDI goal-plan hierarchy, that is, the sequence of decisions starting from a top-level plan choice and terminating in a leaf plan selection. Consider again the goal-plan structure of Figure 3.2 that shows the possible outcomes when plan P is selected in a given world w to resolve top-level event-goal G. In order for the first subgoal G_1 to succeed, plan P_a must be selected followed by P_h that suc-

[2] A plan may include primitive actions along with subgoals. For simplicity in this example only subgoals are used.

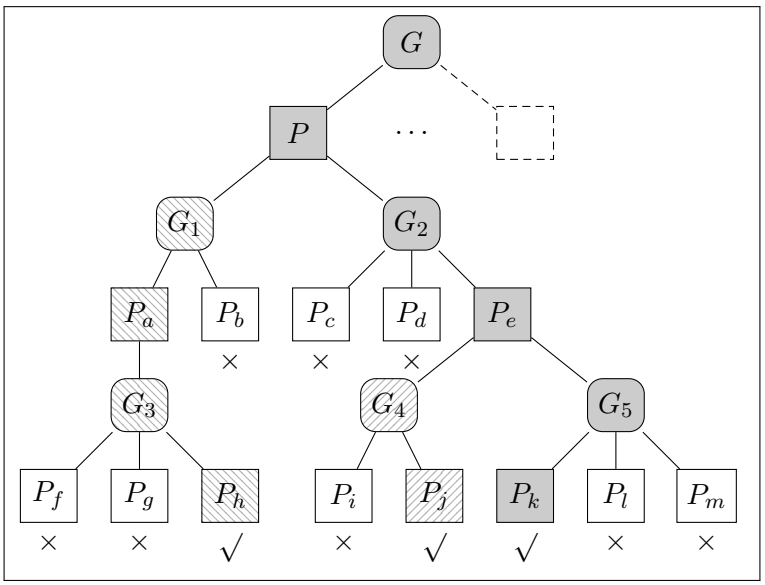


Figure 3.2: An example BDI goal-plan hierarchy.

ceeds as indicated by the $\surd$ symbol. We describe the active execution trace for the top-level event-goal G as $\lambda_1 = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_h : w]$ (highlighted as the line-shaded path starting at G and terminating in P_h) where the notation $G[P : w]$ indicates that event-goal G was resolved by the selection of a plan P in world state w. Subsequently subgoal G_2 is posted whose successful resolution is described by the intermediate trace $\lambda_2 = G[P : w] \cdot G_2[P_e : w'] \cdot G_4[P_j : w']$ followed by the final trace $\lambda_3 = G[P : w] \cdot G_2[P_e : w'] \cdot G_5[P_k : w'']$.[3] The world w' in λ_2 is the resulting world state from the successful execution of leaf plan P_h in the preceding trace λ_1. Similarly, w'' is the resulting world state from the execution of P_j in λ_2. There is only one way for plan P to succeed in the initial world w, as described by the traces $\lambda_1 \ldots \lambda_3$. All other execution traces lead to failures as depicted by the $\times$ symbol.

We can now begin discussing how plan outcomes will be recorded for learning, i.e., how successes and failures will be recorded for hierarchical plan choices in the BDI goal-plan structure using active execution traces.

[3] Trace λ_2 (also λ_1) is intermediate because not every plan (for example P) in that trace has finished executing. In contrast, trace λ_3 is final because all plans have completed (succeeded).

Recording Results

A plan is *relevant* for achieving a given event-goal if it was written to address that event-goal. A plan is also *applicable* for addressing the event-goal when its context condition holds at the time of deliberation. Figure 3.2 shows that while plan P_f is relevant for addressing event-goal G_3, it is not in reality applicable in the given situation (world state w), as selecting it in that situation would lead to failure as indicated by the $\times$ symbol. So revisiting our earlier discussed reasons for failures, plan P_f is a bad choice in world state w for addressing event-goal G_3, and from a learning perspective, we would like to avoid choosing P_f in this situation in the future.

The correct choice for addressing event-goal G_3 in world w is plan P_h and we are equally justified in expecting to learn this relationship. Suppose however, that the domain was non-deterministic so that P_h on occasion fails for reasons not directly ascertainable. What may we infer in this case? Here, instead of learning that the plan either succeeds or fails, it may indeed be more pragmatic to associate a *likelihood of success* of the plan to the situation. Relating this back to our UAV example, we may like to determine the likelihood of success of the emergency landing procedure (plan) using outcomes under different emergency situations.

Overall, once a leaf plan completes, we *record* its outcome, be it success or failure, in its set of ongoing experiences from which its decision tree is induced. In fact, we record the outcome not only for the leaf plan, but for all other higher-level plans in the execution trace that have also completed as a result of that plan's completion. For instance, when plan P_f fails in world w, then all parent plans in the implied active execution trace $\lambda = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_f : w]$ will also fail and we record the outcome not only for plan P_f, but also for plans P_a and P.[4] On the other hand, when plan P_h succeeds in world w, i.e., the trace is $\lambda_1 = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_h : w]$, then we record the success for plans P_h and P_a, both of which have completed their execution (and succeeded), but not for plan P since it is yet to resolve its second subgoal G_2.

Algorithm 1 describes how experiences are (recursively) recorded in our framework given a completed active execution trace λ. Here the *RecordResult*(P_1, w_1, r) step records the outcome $r \in [\textit{success}, \textit{failure}]$ for a given plan P_1 when executed in world

[4] For the purpose of this discussion, assume that failure recovery is not enabled, i.e., the agent does not try alternatives when the initial choice fails. Failure recovery is indeed important in practical BDI systems and we address the nuances of learning with failure recovery separately in Section 3.4.

Algorithm 1: *Record*(λ, r)

Data: $\lambda = G_1[P_1 : w_1] \cdot \ldots \cdot G_n[P_n : w_n]$, with $n \geq 1$; $r \in [\textit{success}, \textit{failure}]$

Result: Records the outcome r for plans in λ.

```
1 if (n > 1) then
2     λ' = G_2[P_2 : w_2] · ... · G_n[P_n : w_n];
3     RecordResult(P_1, w_1, r);
4     Record(λ', r);
5 else
6     RecordResult(P_1, w_1, r);
```

w_1: the experience appended to a growing list of past experiences for plan P_1. When it comes to inducing the decision trees (normally after every few samples as specified by a user defined parameter), we extract the set of past experiences thus recorded for each plan and use these as training samples for building the classifier for that plan.

Specific Issues

We now describe three specific issues related to recording outcomes for learning. First, that higher-level plans will inevitably record failures due to wrong lower-level plan choices until a solution is eventually found; second, that it is not possible to learn correct behaviour when there are dependencies between subgoals of a plan; and third, that we have an infinitely growing training set.

Consider once more the selection decision for event-goal G_3 and suppose that we select plan P_f that fails. This implies the active execution trace $\lambda = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_f : w]$ and the subsequent failure of all the active plans (i.e., P_a and P) in the trace.[5] Since our approach is to record outcomes for all completed plans, then the failure will be recorded against all the plans in the trace, including the higher-level plans P_a and P. The issue here is that in fact a solution does exist for world w in plans P_a and P. It is just that we have not found it yet. This situation is akin to the UAV example of the landing procedure failing due to an incorrect approach speed calculation by a sub-plan, and not because the choice of the landing plan itself was incorrect. This means that plans P_a and P, and all higher-level plans in general, may record several failures until

[5] Again, assume that failure recovery is not enabled.

eventually a success is found. In developing our framework, we indeed experimented with the option of recording failures conditionally only when we were convinced that we were not missing good choices below [Airiau et al., 2009; Singh et al., 2010b] (we discuss these later in Section 3.6). However as we experimented further, we found that with an appropriate approach to confidence in our plan selection (Chapter 4), this was not necessary. The reasoning is that any false-negative samples (i.e., where the a good plan failed due to a bad sub-choice) would eventually be eliminated as "noisy" data by the decision tree induction algorithm itself.

Next, in the example from page 38 suppose that all correct choices were made (as given by $\lambda_1 \ldots \lambda_3$) leading up to the final selection of plan P_k in world w''. In our example P_k would succeed, however it is possible that the way in which prior subgoals were resolved (i.e., plans P_h and P_j) may impact the success of P_k. For instance, plan P_j may have been originally written by a different domain expert who at the time had no knowledge of how it may be used in a higher context (i.e., P_e). It is foreseeable that the way P_j depletes a shared resource for instance, may impact the availability of that resource for plan P_k. If the interaction is such that P_k may never succeed if preceded by P_j then we would like to learn to avoid this selection sequence. However, at the subgoal level there is no information about the higher-level "agenda" in which the plan is being used and this information cannot be represented as part of its context condition. As such there is no way for such dependencies to be learnt. This is a current limitation of our learning framework that is discussed in more detail in Chapter 8.

Lastly, an important implication of our approach is that in effect we keep a growing set of all experiences of the agent. The benefit is that we are able to build the ideal classifier given an agent's experience so far. However, simply storing this data may become impractical after the agent has been operating for a very long time. Moreover, the larger the training set, the more effort is required to induce the corresponding decision tree. Using *incremental* approaches for inducing decision trees [Swere et al., 2006; Utgoff, 1989; Utgoff et al., 1997] will certainly address both problems, but may also impact classification accuracy. We discuss this limitation further with our conclusions in Chapter 8.

3.3 A Probabilistic Plan Selection Scheme

We have so far described the integration of decision trees with plans' context conditions and how they may be induced using the ongoing experiences of the agent. Here we focus on how this new account of a plan's applicability may be used to improve plan selection. Typical BDI platforms offer several mechanisms for plan selection from a set of applicable plans, such as plan precedence and random selection. However, since these are pre-programmed and do not take into account the experience of the agent, we provide a new *probabilistic plan selection* scheme for this purpose.

For each plan P, given its expectation of success $\mathcal{P}(P, w)$ in world w as determined by its decision tree, we calculate a final *selection weight* that will determine the likelihood of the plan being selected for execution. Equation 3.3.1 shows how this plan selection weight $\Omega(P, w, n)$ is constructed:

$$\Omega(P, w, n) = 0.5 + [\mathcal{C}(P, w, n) \times (\mathcal{P}(P, w) - 0.5)]\,. \tag{3.3.1}$$

The component $\mathcal{C}(P, w, n) \in [0.0 : 1.0]$ is a *dynamic confidence measure* that reflects our perceived confidence in the current learning (i.e., the decision tree prediction $\mathcal{P}(P, w)$), as well as how well we know the worlds we are witnessing, calculated over the last n executions of P. We will discuss how this measure is constructed in detail later in Chapter 4.

The idea is to combine the likelihood of success of the plan $\mathcal{P}(P, w)$ with the confidence bias $\mathcal{C}(P, w, n)$ to determine a final plan selection weight $\Omega(P, w, n)$. When the perceived confidence is maximum, i.e., $\mathcal{C}(P, w, n) = 1.0$, then $\Omega(P, w, n) = \mathcal{P}(P, w)$, and the final weight equals the likelihood of success given by the plan's decision tree; when the confidence is zero i.e $\mathcal{C}(P, w, n) = 0.0$, then $\Omega = 0.5$, and the decision tree has no bearing on the final weight (a default weight of 0.5 is used instead).

Given the selection weight $\Omega(P_i, w, n)$ for each plan P_i with $i \in [1 : k]$ in a set of k applicable plans, we then *choose a plan with a probability directly proportional to its selection weight*, i.e., the probability of selecting a plan P_i is $\Omega(P_i, w, n) / \sum_{j=1}^{k} \Omega(P_j, w, n)$.

This probabilistic selection scheme ensures a balance between the *exploitation* of current (learnt) knowledge, and the *exploration* of new choices (to increase knowledge) that is necessary for any online learning system. The key component in this balance

is our perceived confidence (i.e., $\mathcal{C}(P, w, n)$) in what we know: the more we trust our knowledge, the more we use it to make plan choices; the less we trust it, the less we rely on it, and the more we explore to improve understanding and build confidence.

Plan Applicability

Our new account of plans' context conditions has implications for the meaning of applicability in a given situation. In a typical BDI system, a plan's applicability is fully defined by its boolean logical context formula, such that a plan is considered applicable only when the context condition holds. In our new setting a plan's applicability is *additionally* defined by its selection weight $\Omega(P, w, n)$ that gives the probability of the plan being selected in the given situation (i.e world w). In other words, a plan's applicability is decided by two filters: a first *programming filter* that constitutes the encoded context condition of the plan, and a second *learning filter* given by its selection weight $\Omega(P, w, n)$.

The fact however, that the second filter $\Omega(P, w, n)$ is no longer boolean and that a low value (i.e., nearing 0.0) implies a low chance of success, poses the question of deciding what the minimum acceptable $\Omega(P, w, n)$ should be for a plan to still be considered applicable. It may be reasonable to assume that answers to questions like "Should a plan whose likelihood of success in a situation is 5% be considered applicable in that situation?" are domain-dependent. In a domain where success is rare, a different *threshold* may apply for such decisions, as compared to a domain where success is easily achieved.

As such, where appropriate the domain expert should provide an *applicability threshold* to be used during plan selection. The specified threshold has the effect that every plan P in the applicable set for world w (i.e., the set obtained after considering the initial context conditions) whose selection weight $\Omega(P, w, n)$ falls below this value, is discarded from consideration.

3.4 Learning with BDI Failure Recovery

A core aspect of BDI agent systems is the *failure recovery* mechanism which allows the agent to be responsive and robust in the face of environmental changes while it is pursuing some course of action. A common cause of failures in dynamic environments

is when the conditions for executing a plan change while the plan is executing. When this happens, the failure recovery mechanism allows the agent to reassess the situation and try alternative plans to achieve the event-goal. The idea is to enable errors to be resolved on the go, which may be a more effective strategy in many domains over completely abandoning the event-goal after the first failure. BDI failure recovery also provides robustness to non-deterministic failures, as well as when the environment is only partially observable. In the UAV example, if a plan to land on an airstrip was aborted during execution, due to unexpected severe winds for example, the agent could use failure recovery to re-evaluate the situation and consider alternative plans, such as climbing to higher altitude and trying again later, or seeking landing permission at a nearby airport, instead of aborting the goal to land altogether.

The use of BDI failure recovery has implications for our learning framework that we discuss here. Consider again the example goal-plan hierarchy of Figure 3.2 and suppose an initial decision trace $\hat{\lambda} = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_f : w]$ that ends in the failure of leaf plan P_f. As determined earlier, we will record the failure of P_f in w for learning purposes.

If failure recovery is now enabled, it means that the execution of non-leaf plan P_a is not complete yet, until all other possible options for resolving its subgoal G_3 are first considered. At this point, the applicable set for G_3 is recalculated to take into account the possible change in the state of affairs from the failure of plan P_f in world w. Let's call this new world state w' and suppose that the applicable set for G_3 in w' is now $\{P_h\}$ (option P_f has already been tried and P_g is no longer considered applicable in w', i.e., its context condition does not hold). Plan P_h being the only applicable plan for G_w in w' is therefore selected, and say that it succeeds so that we get $\hat{\lambda}' = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_h : w']$. As before, we may record the success of P_h in w' for learning. This time around, plan P_a has also completed execution since it has succeeded, however we will *not* record this success in P_a against the initial world w in which it was invoked (even though indeed in Figure 3.2 a solution exists for P_a in w). This is because *the success of P_h was preceded by the failure of P_f, and it may have been precisely this failure that impacted the world in a way that caused P_h to succeed.* In other words, it may well be that the only way for P_a to succeed in w is to first select P_f and fail, and then select P_h. Of course, were this in fact the case, then we would like to avoid learning this behaviour.

To address this, *in our learning framework we do not propagate results to the parent nodes of the event-goal where failure recovery was performed.* [6] In the above example with $\hat{\lambda}'$ then, the success would not get propagated to the parent plans of subgoal G_3, i.e., plans P_a and P. This does not preclude us from recording the results for goal-event G_2 that gets posted after the success of P_h as those results will be relative to whatever the world is: if it happens to have been changed by the earlier failed plan P_f then that is at this stage irrelevant.

Finally, note the subtle disconnect between the intended use of failure recovery in BDI systems and its potential use while learning. Failure recovery by definition implies a well founded goal-plan hierarchy: where things *generally go to plan*; and where recovery is tried when something unexpected occurs. While learning from scratch, on the other hand, failure (instead of success) is generally the norm, as the agent gradually begins to make sense of its environment. Arguably then, the use of failure recovery in the initial stages of learning should be discouraged. Indeed, it is possible that failure recovery may force the selection of every possible decision path until all options are exhausted. In domains where failures cause irreversible changes, such perseverance may well be futile. A possibility here may be to gradually enable failure recovery and limit the extent to which recovery is tried in the early stages of learning.[7] Nevertheless, since the utility of failure recovery partly depends on the domain in question and how good the initially programmed context conditions are, then for this work we treat the enabling of failure recovery as a user specified option.

3.5 Learning in Recursive Hierarchies

Recursion in our context refers to the case where the resolution of an event-goal instance $G(\vec{x_1})$, where $\vec{x_1}$ are the parameters of an event-goal type G, involves first the resolution of goal-event instance $G(\vec{x_2})$ of the same type. The result is a growing stack of pending $G(\vec{x_i})$ event-goals that eventually terminate in $G(\vec{x_n})$ whose parameters satisfy the termination conditions, i.e., where a non-recursive plan choice is made. Such recursive use of event-goals is typical of many practical BDI systems: it provides a mechanism for "looping" in event-driven architectures; and for solving problems by decomposing

[6] An exception to this rule is when the failure does not change the world state in any noticeable way. Here, it may be reasonable to ignore the preceding failure and record as normal.

[7] We have not implemented this option and such analysis is left for future work.

them into smaller (similar) sub-problems.

For example, the plan library of an elevator controller may contain the following recursive plan rule:

$\mathsf{Goto}(\mathsf{floor}) : \mathsf{At}(x) \wedge x < \mathsf{floor} \leftarrow \mathsf{GoUp}; !\mathsf{Goto}(\mathsf{floor})$

That is, to resolve a request to go to a particular floor (i.e., goal $\mathsf{Goto}(\mathsf{floor})$) that is above the current location of the elevator (i.e., context condition $\mathsf{At}(x) \wedge x < \mathsf{floor}$), it needs to go up one floor (i.e., execute primitive action GoUp) and then post again the (sub)goal of reaching the floor in question (i.e., $!\mathsf{Goto}(\mathsf{floor})$).

In order to understand the impact of recursion on context learning, we extend our active execution trace notation to the form $G(\vec{x})[P : w]$ to also include the event-goal parameters $\vec{x}$. Consider, for example, the BDI goal-plan hierarchy of Figure 3.3 that shows a high level plan P for resolving event-goal G. Plan P in turn posts subgoal $G_1(\vec{x_1})$ that is handled by plans P_1, P_2 and P_3, and $G_2(\vec{y_1})$ that is handled by plans P_4, and P_5. Plans P_1, P_3 and P_4 are leaf plans that directly interact with the environment, while plans P_2 and P_5 post instances of the same event-goal that they handle, leading to recursion. Figure 3.3 highlights a decision sequence that ends in the failure of plan P_4 which was selected to address event-goal instance $G_2(\vec{y_1})$. The relevant execution traces here are $\lambda_a = G[P : w] \cdot G_1(\vec{x_1})[P_2 : w] \cdot G_1(\vec{x_2})[P_3 : w]$ and $\lambda_b = G[P : w] \cdot G_2(\vec{y_1})[P_4 : w']$

The first trace λ_a describes the selection of plan P to handle top-level event-goal G in world w. Plan P posts subgoal $G_1(\vec{x_1})$ in world w that is handled by plan P_2, that in turn posts $G_1(\vec{x_2})$ that is successfully handled by the non-recursive plan P_3. Plan P then posts its second subgoal $G_2(\vec{y_1})$ in the resulting world state w', which then is handled by the leaf plan P_4 that fails as given by λ_b. If plan P_5 had instead been selected to handle $G_2(\vec{y_1})$ then a deeper recursive call would have ensued. Similarly if earlier in the execution trace plan P_2 was selected to handle event-goal $G_1(\vec{x_2})$ then a different recursive sub-tree would have unfolded. These possibilities are highlighted as dashed nodes in Figure 3.3.

The immediate implication of a recursive goal-plan structure is that the size of the hierarchy is no longer static but instead unfolds in a dynamic manner. The risk then is that since the conditions that terminate recursion are not guaranteed at the start (we are indeed trying to learn them), then the agent may get trapped in an infinite recursive loop

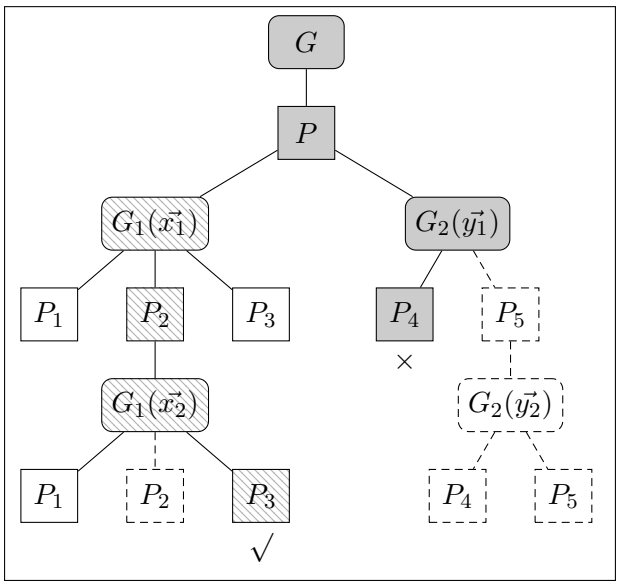


Figure 3.3: Goal-plan hierarchy containing two parameterised goals G_1 and G_2. Plans P_2 and P_5 also post the event-goals that they handle, resulting in recursion. Two levels of recursive unfolding are shown. Dashed nodes indicate unexplored recursive sub-trees.

during exploration. To resolve this issue in our framework, we use a bounded recursion approach whereby we limit such recursive unfolding to a maximum allowed depth. It follows then that wherever a recursive structure applies, a *maximum recursion value* must also be supplied by the domain expert. This may not be an unrealistic requirement given that the domain expert will usually have some understanding of how much recursion is sufficient for a given parameterised event-goal.

3.6 Summary and Discussion

In this chapter we presented our framework for learning plan selection in BDI agent systems. We extended the account of a plan's context condition to include a decision tree and described how plan selection may be improved at runtime by recording ongoing experiences of the agent, inducing decision trees for all plans from such experiences, and using the decision trees to probabilistically select plans. We described the use of the framework with BDI failure recovery enabled, and in goal-plan hierarchies that employ event-goal recursion.

Implementation

We have implemented our framework in the JACK [Busetta et al., 1999] BDI agent programming language. JACK is implemented as a programming layer on top of the Java [Gosling et al., 2005] programming language and allows for standard Java code to be written and easily integrated. As such, significant portions of our framework including the recording infrastructure have been coded in Java. A JACK plan provides convenience functions called *pass* and *fail* that are appropriately called after the plan body has finished execution, that we use to record outcomes for learning purposes. Our probabilistic plan selection heuristic is implemented within a meta-level plan that is invoked by JACK whenever an applicable set is to be evaluated. The initialisation of our framework is done inside the BDI agent initialisation routines. Finally, for the decision trees, we use an off-the-shelf Java implementation of the algorithm J48, a version of c4.5 [Mitchell, 1997], from the weka learning package [Witten and Frank, 1999].

On the Reliability of Learning

In our preliminary investigations [Airiau et al., 2009; Singh et al., 2010b], we have considered two options for dealing with failures when learning in plan hierarchies. The first is that of careful consideration where we use a failure for learning only if we believe that the decisions that led to the failure were reasonably well-informed, or "stable." In [Singh et al., 2010b], we defined such stability in terms of how the rate of success (i.e., the ratio of successful to total executions) of a plan is *changing* in a given world: the plan is considered stable in the world if this rate is changing below a specified threshold, and as long as all plans below it in the execution trace are also stable. In our example trace $\lambda = G[P : w] \cdot G_1[P_a : w] \cdot G_3[P_f : w]$ from page 40, we would record the failure in P_a only if P_f and P_a were deemed stable; while for P we would record if P_f, P_a and P were all deemed stable. So whereas successes (if any) were always recorded, we started recording failures only when the plan's outcome is considered stable in the world. While this stability filter resolves our initial concern with recording failures incorrectly, it is also too restrictive. For example, in Figure 3.2, it may take many executions of plan P before it becomes stable (since all possible options below P must become stable or succeed first), and since no recording happens in P until that happens, then potentially useful information is being discarded.

The alternative approach, and the one we use in this thesis, is to instead record the failure at all levels in the hierarchy for every failed trace, in the hope that the generalisation learnt will eventually eliminate any "noisy" data. This approach is much simpler and works well for the most part (in fact in [Singh et al., 2010b] we show that both approaches have their advantages in different types of goal-plan structures). However it can sometimes lead to a complete inability to learn. This happens when a lot of failures are recorded before a success is found (for instance, in Figure 3.2 there are significantly more failure paths for plan P than success paths) and the likelihood of success given by a plan's decision tree becomes so low (before success is achieved) that it is never considered applicable enough (based on some applicability threshold).

The issue, of course, is that the decision tree prediction is poor when there is insufficient training data. In order to decide how much trust to put in its predictions then, some measure of the ongoing reliability in the decision tree would be useful.

The notion of stability introduced in [Airiau et al., 2009] was one such measure of reliability. It was used to estimate our *confidence* in the decisions below a plan in the goal-plan hierarchy. Plan outcomes then, were recorded for learning purposes only if the underlying decisions were considered stable. The aim was to filter out as much as possible "noisy" training samples in order to build a more reliable learner. A different approach was taken in [Singh et al., 2010b] where a confidence measure was instead used to adjust plan selection probabilities. Unlike the previous stability-based confidence measure that was used to filter the training set, this new coverage-based confidence measure was used to directly adjust the exploration strategy. The idea was that the confidence in a plan's decision tree was related to how many of the possible choices below the plan had been explored or "covered": the more that such decision paths had been explored, the greater the confidence in the resulting learner. This approach complemented our earlier approach [Airiau et al., 2009] as it allowed any recording scheme to be used. However, as we later showed in [Singh et al., 2010a], the coverage-based measure has several limitations that make it impractical for use beyond synthetic structures.

This issue of confidence is central to our learning discussion, and one that we discuss in depth in Chapter 4 where we describe our final dynamic confidence measure that is suitable for use in practical BDI systems.

Chapter 4

Determining Confidence in Ongoing Learning †

A BDI agent tasked with improving ongoing plan selection (using the learning framework described in Chapter 3), does so in an *online* manner, where learning and acting are interleaved and understanding of the domain comes from "trial and error" in the environment. The typical use of decision trees, however, lies in their offline induction from a complete training set. In that sense, the use of decision trees in our framework is unorthodox since the training set is built incrementally using accumulated samples from each new plan execution. This results in incomplete information in the early stages of learning, leading to high levels of *misclassification*.

Consider the case of a controller agent for an unmanned aerial vehicle (UAV) that is trying to optimise its landing procedure (plan) by fine-tuning the approach speed and angle of the airplane, perhaps in a virtual simulated environment. Suppose that it tries to land the plane for some bound values of these parameters, and fails. What may the agent learn from this experience alone? Possibly that the given combination of approach parameter values does not work. If it were also to extrapolate from this experience to every new situation (i.e., combination of approach speed and angle), then it would invariably, and rather inappropriately, conclude that all attempts at landing must fail. The problem is simply that it does not have enough information to make well-informed judgements, or *predictions*, about the real likelihood of landing the plane. In other words, a pre-

† Parts of the work presented in this chapter will appear in [Singh et al., 2011].

diction of success or failure in itself does not say anything about how *informed* that prediction is. So how should one decide how much trust to put in the current learned information?

The situation is not dissimilar to the typical machine learning setting of an agent trying to maximise some reward, where the dilemma is whether to exploit the current learning to obtain the maximum benefit known so far, or to explore further options in the hope of finding solutions that yield even higher benefits. This issue is normally addressed in such cases using a pre-specified strategy for deciding the level of exploration during learning. For instance, in ϵ-greedy exploration [Sutton and Barto, 1998] commonly seen in reinforcement learning, the agent chooses the action that it believes has the best long-term benefit with probability $1 - \epsilon$, and chooses randomly otherwise. Here the parameter ϵ is generally either fixed, or gradually reduced over time to reflect increasing confidence in the ongoing learning.

A pre-determined exploration strategy such as ϵ-greedy however is completely decoupled from the learning itself. It does not take into account how the learning is actually progressing. As such, careful parameter (i.e., ϵ) selection is required to ensure that the learning and our associated confidence (reflected in the exploration strategy) are aligned: a process that in itself involves trial-and-error on the part of the programmer. This is an important issue in online learning where the programmer does not have the option to adjust exploration parameters once the agent has been deployed. What is required, instead, is an adaptive confidence measure that accounts for the ongoing learning and adjusts accordingly. This view has recently been supported by Tokic [Tokic, 2010], who questions the value of such ad-hoc approaches as ϵ-greedy, and proposes a more general strategy that ties the exploration to the learning performance itself.

We investigated the issue of confidence in the context of ongoing decision-making in BDI goal-plan hierarchies in our original work on this topic [Singh et al., 2010a,b]. These "coverage" based confidence measures (page 49), however, lack in two important ways. Firstly, they do not cater for a changing dynamics of the environment that often results in prior learning becoming less effective. For instance, consider the case where the UAV controller has learnt the optimal approach speed and angle for landing the airplane on a remote airstrip. If however, the surface of the unattended airstrip were to deteriorate over time, then the previously learnt landing parameters may no longer work, and the controller would have to *dynamically adapt* its learning to this ongoing change

in the environment dynamics. Clearly, a static exploration strategy that monotonically converges (such as ϵ-greedy or one constructed using the coverage-based approaches) will not suffice for this requirement. Secondly, the previous coverage-based measures do not scale well for complex goal-plan structures, since they rely on an estimate of the number of choices in the goal-plan hierarchy that is not always easy to calculate (e.g., in recursive structures). In this chapter, we describe a confidence measure that overcomes both these limitations.

4.1 A Dynamic Confidence Measure

In Equation 3.3.1 of Chapter 3 we introduced a confidence measure $\mathcal{C}(P, w, n)$ for the last n executions of plan P in world w. This measure represents our perceived confidence in the plan's decision tree and is used to adjust the final plan selection probabilities. We are now ready to define this in some detail.

We start by first constructing a component metric that captures our confidence in the ongoing hierarchical decision-making. We do this by extending the previously introduced idea of stability (page 48) to what we will term as the *degree of stability*. In effect, we translate the boolean notion to a numerical one, that conceptually relates to the extent to which a decision sequence is considered informed (by considering the stability of each decision in the sequence). Since stability is calculated on observed outcomes, then this measure reflects the ongoing performance of the agent and dynamically adapts when changes in performance occur. When used in the exploration heuristic, this allows the agent to increase exploration if a previously learnt behaviour were to start to fail.

A second component of confidence (apart from how many options we have tried in a given world) is how many different world states we have experienced. For our UAV controller example, this relates to the issue of over-generalising from a single failed landing experience as discussed in the beginning of this Chapter. To account for this, we build a second metric that captures the familiarity with the environment by measuring *the rate at which the plan is being tried in known worlds*. To do this, we compare a sample of the most recent worlds where the plan was executed to what has been witnessed before. Conceptually this relates to how well we believe we know the domain: the more we are seeing what we have seen before, the greater our confidence with respect to our understanding of the domain.

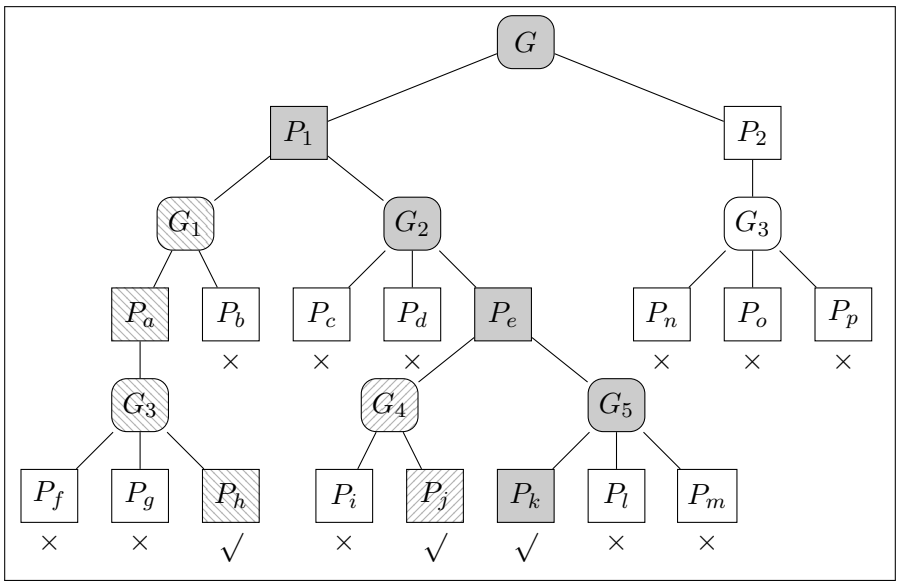


Figure 4.1: An example BDI goal-plan hierarchy.

Our final confidence measure $\mathcal{C}(P, w, n)$ is then constructed from these two component metrics.

4.1.1 Stability-Based Component Metric

Our initial definition of stability from [Singh et al., 2010b] is as follows:

> *"A failed plan P is considered to be stable for a particular world state w if the rate of success of P in w is changing below a certain threshold. [...] a failed goal is considered* stable *for world state w if all its relevant plans are stable for w."*

To see how stability is calculated, consider the example goal-plan structure of Figure 4.1 that shows the possible resolution of event-goal G in a given world state w. Observe that plan P_2 always fails in world w (as indicated by the $\times$ symbol against its children), whereas plan P_1 succeeds: the solution requiring three leaf plans P_h, P_j and P_k to be expanded (marked with $\surd$).

Let us take the simple case of plan P_n and suppose that the stability threshold is set to 0.1. Say that P_n was executed in world w and failed, as expected from Figure 4.1. Its rate of success in w (i.e., the ratio of total successes to total most recent few executions) therefore would be $\frac{0}{1} = 0.0$. Now, to determine if the rate of success is *changing*, we must compare two rates for which we require P_n to execute in world w a second time. Suppose that this happens and P_n fails in w yet again. The new rate of success is therefore $\frac{0}{2} = 0.0$. At this point we can calculate how the rate of success is changing by taking the difference $|\frac{0}{1} - \frac{0}{2}| = 0.0$. Since the stability threshold is 0.1 and the change in success rate (i.e., 0.0) is less than that, then this would imply that plan P_n becomes stable after these two executions in w. The result is the same for plan P_h that always succeeds in w, i.e., $|\frac{1}{1} - \frac{2}{2}| = 0.0$.

Stability is similarly calculated for non-leaf plans *per execution trace*. Recall that stability is meant to be a measure of how well informed our decisions were which led to a particular outcome. So it only makes sense to ask that question for a particular selection sequence, i.e., active execution trace. For instance, for a failed trace $\lambda = G[P_1 : w] \cdot G_1[P_a : w] \cdot G_3[P_g : w]$, plan P_1 will be considered stable only when the above stability calculations hold for it *and* all the plans below it in the trace (i.e., P_g and P_a) are also stable. Observe that this definition of stability does not include all *possible* plans below P_1 in the hierarchy of Figure 4.1, but only those that were actually selected. This also means that not every option below the plan must be tried for it to become stable. For instance, consider for a moment that plan P_h were to also fail in world w. That would mean that subgoal G_1 would never succeed and therefore subgoal G_2 and its children will never be tried. In this case, plan P_1 will still become stable when all the options below that are actually tried eventually become stable.

The real benefit of the stability measure becomes evident when considering stochastic domains. For instance, suppose that the agent was operating in such an environment and plan P_h now sometimes fails (say 25% of the time) due to non-deterministic reasons. Say that the first four executions of P_h in w result in $\{$success, failure, success, success$\}$. The progressive change in the success rate therefore would be $|\frac{1}{1} - \frac{1}{2}| = 0.50$, $|\frac{1}{2} - \frac{2}{3}| = 0.17$, and $|\frac{2}{3} - \frac{3}{4}| = 0.08$. Given that the stability threshold is set to 0.1, this would mean that plan P_h would become stable after the fourth execution in w when the change in success rate (i.e., 0.08) drops below the threshold.

Since stability calculation depends on the threshold parameter among other things, it is possible that we sometimes prematurely believe that a plan's outcomes have stabilised. This will normally correct itself when the plan is chosen again (often) due to probabilistic selection. The user can also fix this by lowering the stability threshold parameter during early experimentation.

With this understanding in place, we now extend our stability idea developed in [Airiau et al., 2009; Singh et al., 2010b] to the execution trace itself. The aim is to ascertain the extent or *degree* to which the decisions in the trace as a whole may be considered stable or well-informed. This is particularly meaningful for failed execution traces where low stability suggests that we were not well-informed and more exploration is needed before assuming that no solution exists for the top event-goal in the trace.

To capture this, we define the *degree of stability* of a failed execution trace λ, denoted $\zeta(\lambda)$, as the ratio of stable plans to total applicable plans in the active execution trace below the top-level event-goal in λ. Formally, when $\lambda = G_1[P_1 : w_1] \cdots G_n[P_n : w_n]$ we define

$$\zeta(\lambda) = \frac{|\bigcup_{i=1}^{n} \{P \mid P \in \Delta_{app}(G_i, w_i),\ \mathit{stable}(P, w_i)\}|}{|\bigcup_{i=1}^{n} \Delta_{app}(G_i, w_i)|}, \tag{4.1.1}$$

where $\Delta_{app}(G_i, w_i)$ denotes the set of all applicable plans (i.e., relevant plans whose boolean context conditions hold true) in world state w_i for event-goal G_i, and $\mathit{stable}(P, w_i)$ holds true if plan P is deemed stable in w_i.

To understand what this means, let us take a failed execution trace $\lambda = G[P_1 : w_1] \cdot G_2[P_e : w_2] \cdot G_5[P_m : w_3]$ and suppose that the applicable plans are $\Delta_{app}(G, w_1) = \{P_1, P_2\}$, $\Delta_{app}(G_2, w_2) = \{P_c, P_e\}$, and $\Delta_{app}(G_5, w_3) = \{P_k, P_l, P_m\}$. The trace λ is shown in Figure 4.2: the plans in dotted outline are all the *relevant* plans for all goals in the hierarchy, while the applicable plans for all the goals in λ are shown in normal outline. Here, the given active trace λ implies that P_h and P_j were successfully executed: w_2 and w_3 being the world states that resulted from the successful execution of those plans. Let's also say that P_c and P_l are the only plans deemed stable (in worlds w_2 and w_3 respectively).

Then the degree of stability for the whole trace is $\zeta(\lambda) = \frac{2}{7}$ since the union set of all applicable plans is $\{P_1, P_c, P_e, P_k, P_l, P_m, P_2\}$ and two plans in the set are stable. Similarly, for the sub-trace $\lambda' = G_2[P_e : w_2] \cdot G_5[P_m : w_3]$ the union set is

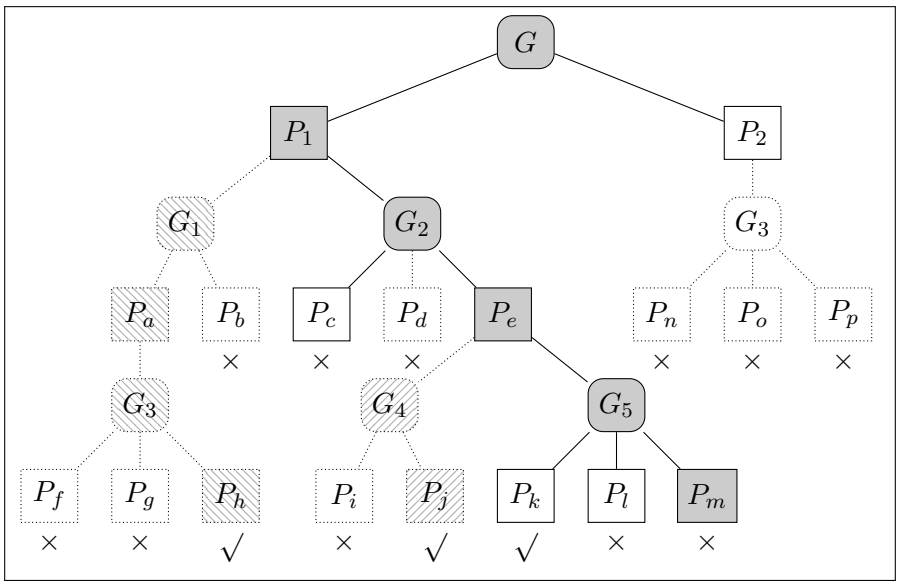


Figure 4.2: The example BDI goal-plan hierarchy of Figure 4.1 showing the failed trace λ that ends in plan P_m along with the applicable plans (solid outline boxes) for each goal in the trace.

$\{P_c, P_e, P_k, P_l, P_m\}$ and $\zeta(\lambda') = \frac{2}{5}$, while for the sub-trace $\lambda'' = G_5[P_m : w_3]$ we get $\zeta(\lambda'') = \frac{1}{3}$.

The idea is that every time the agent reaches a failed execution trace, the degree of stability of each completed sub-trace[1] is stored in the plan that produced that sub-trace. For instance in our example above, for plan P_1 we record degree $\zeta(\lambda') = \frac{2}{5}$ whereas for plan P_e we record degree $\zeta(\lambda'') = \frac{1}{3}$. Leaf plan nodes, like P_m, make no choices so their degree of stability is assigned 1. Intuitively, by doing this, we record against each plan in the failed trace, an estimate of how informed the current choices made for the plan were. Algorithm 2 describes how this hierarchical recording is done for each plan in a given active execution trace λ. Here, *RecordDegreeStability*(P, w, d) records (i.e., saves to memory) the degree of stability d for plan P in world state w.

As a plan execution produces new failed experiences, the calculated degree of stability is appended against it each time. When a plan finally succeeds, we take an optimistic

[1] By completed we mean that each plan in the sub-trace has completed every subgoal (and primitive action).

Algorithm 2: *RecordDegreeStabilityInTrace*(λ)

Data: $\lambda = G_1[P_1 : w_1] \cdot \ldots \cdot G_n[P_n : w_n]$, with $n \geq 1$.

Result: Records degree of stability for plans in λ.

```
if (n > 1) then
    λ' = G_2[P_2 : w_2] · ... · G_n[P_n : w_n];
    d = ζ(λ');
    RecordDegreeStability(P_1, w_1, d);
    RecordDegreeStabilityInTrace(λ');
else
    RecordDegreeStability(P_1, w_1, 1);
```

view and record 1 (i.e., full stability) against it. This, together with the fact that all plans do eventually become stable, means that the degree of stability $\zeta(\lambda)$ tends to become more stable over time.[2]

Given this sequence of $\zeta(\lambda)$ recordings for a plan P in world w, we may now construct our measure of confidence in these decisions. We do this by aggregating over the most recent $n \geq 1$ executions of plan P in w, denoted by $\mathcal{C}_s(P, w, n)$ and as shown in Equation 4.1.2. [3] Intuitively, this *average degree of stability* is a numeric value that relates to how well-informed we perceive our decisions to be in the n most recent invocations of P in w.

$$\mathcal{C}_s(P, w, n) = \sum_{i=1}^{n} \zeta(\lambda_i). \tag{4.1.2}$$

The parameter n decides the averaging window and determines the sensitivity of the measure to changes in the degree of stability. The reason why we take an average rather than the most recent value is because each degree of stability relates to a specific path (trace) in the goal-plan hierarchy, but we are interested in obtaining an overall approximation of the degree of stability that is irrespective of any trace.

[2] Assuming that the dynamics of the environment and the non-determinisism in the environment are not changing.

[3] The notation $\mathcal{C}_s$ means *stability*-based component metric, or simply, stability-based confidence.

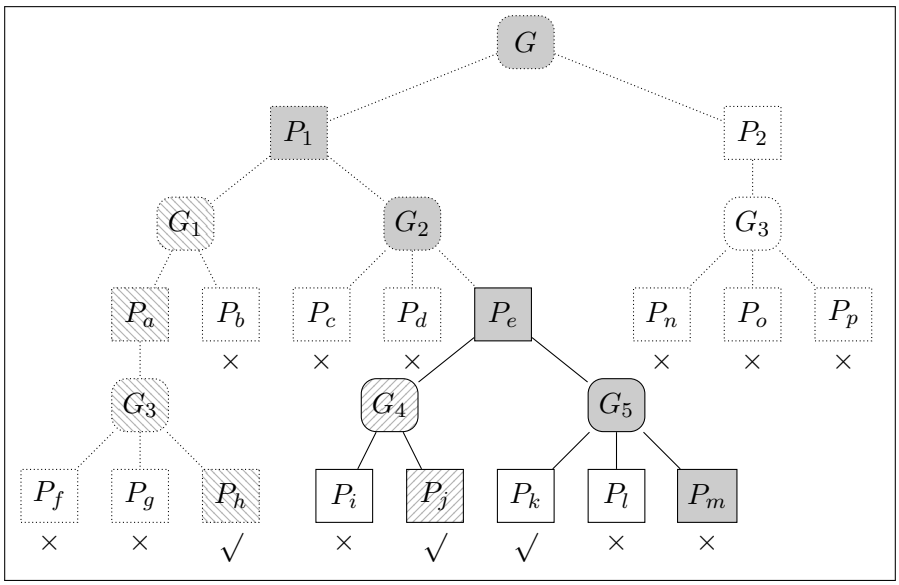


Figure 4.3: The example BDI goal-plan hierarchy of Figure 4.1 focussing on plan P_e.

To demonstrate how the stability-based confidence is computed, we will use the example of plan P_e (from Figure 4.1) in world w_1 with $n = 5$, i.e., $\mathcal{C}_s(P_e, w_1, 5)$. Suppose that the applicable plans are $\Delta_{app}(G_4, w_1) = \{P_i, P_j\}$ and $\Delta_{app}(G_5, w_2) = \{P_k, P_l, P_m\}$. The situation is shown in Figure 4.3. Further suppose that the rate of success of a plan stabilises (i.e., the plan becomes stable) after two executions in a given world. Table 4.1 shows the first few example executions of P_e in world w_1. In reality the executions of P_e would have been interleaved with other worlds than w_1, however these are not shown in Table 4.1 as they do not contribute to the calculation of $\mathcal{C}_s(P_e, w_1, 5)$.

In the beginning, P_e has never been tried in w_1, and all plans have the same likelihood of being selected. So when G_4 is first posted in w_1, the agent has an equal likelihood of selecting either of the two applicable plans $\{P_i, P_j\}$. Let's say it makes the correct choice, i.e., it chooses plan P_j that succeeds. This results in subgoal G_4 succeeding, and plan P_e posting its second subgoal G_5. Let's say that this time plan P_l is selected that fails (as expected from Figure 4.1). The final active execution trace therefore is $\lambda_1 = G_5[P_l : w_2]$. As P_l in this trace is not stable yet, then $\zeta(\lambda_1) = \frac{0}{3}$ (since $|\Delta_{app}(G_5, w_2)| = 3$) and therefore $\mathcal{C}_s(P_e, w_1, 5) = 0.0$ as shown in the first row of Table 4.1.

The next time P_e is invoked in w_1, suppose that the final trace is $\lambda_2 = G_5[P_m : w_2]$ meaning that the execution terminated in the failure of P_m (again as shown in Figure 4.1), and implying once more that G_4 succeeded via plan P_j. This situation is depicted in the second row in Table 4.1. However, once more $\mathcal{C}_s(P_e, w_1, 5) = 0.0$ since no plan in λ_2 is stable yet.

The third execution (row three) terminates in the failure of P_i that is not stable yet, however $\zeta(\lambda_3) = \frac{1}{2}$ since $|\Delta_{app}(G_4, w_1)| = 2$ and the second applicable plan P_j is considered stable as it succeeded in w_1 previously. The average degree of stability $\mathcal{C}_s(P_e, w_1, 5)$ over the last five executions (there have only been three so far) therefore increases to 0.10 as shown.

The fourth, fifth, and sixth executions (rows four, five, and six respectively) see the stability-based confidence increase gradually as first P_m becomes stable (giving $\mathcal{C}_s(P_e, w_1, 5) = 0.17$), then the agent finds success in P_k that is immediately considered stable (giving $\mathcal{C}_s(P_e, w_1, 5) = 0.37$), and finally a second failure of P_i makes it stable (giving $\mathcal{C}_s(P_e, w_1, 5) = 0.57$).

The final three executions all succeed in P_k and the average degree of stability $\mathcal{C}_s(P_e, w_1, 5)$ gradually increases: to 0.77, then 0.87, and finally 1.00. Note that convergence (to 1.00) does not automatically imply that all applicable plans are stable. In this example plan P_l is not yet stable as it has only been tried once.

Observe that since stability is a measure of the rate of change of success of a plan, then $\mathcal{C}_s(P, w, n)$ (that amounts to the average degree of stability) invariably captures the stability of the agent's performance over time. Given an environment with fixed dynamics then, this measure generally increases from 0 as plans below P start to become stable (or succeed), and reaches 1 when all plans below P in the last n execution traces are considered stable (or successful). This is what one might expect in the typical learning setting. Importantly, if the dynamics of the environment changes and previously learnt solutions start to fail because they no longer work, then the measure adjusts confidence accordingly to reflect the new instability in performance. Any such drops in confidence consequently impact the plan selection weight (Equation 3.3.1) and promote new exploration.

λ	$\zeta(\lambda)$	$\mathcal{C}_s(P_e, w_1, 5)$	Explanation
$G_5[P_l : w_2]$	$\frac{0}{3}$	$\left[\frac{0}{3}\right] \times \frac{1}{5} = 0.00$	$G_4[P_j : w_1]$ succeeded first. P_l is not stable yet.
$G_5[P_m : w_2]$	$\frac{0}{3}$	$\left[\frac{0}{3} + \frac{0}{3}\right] \times \frac{1}{5} = 0.00$	$G_4[P_j : w_1]$ succeeded first. P_m is not stable yet.
$G_4[P_i : w_1]$	$\frac{1}{2}$	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2}\right] \times \frac{1}{5} = 0.10$	P_i is not stable yet; P_j is considered stable as it succeeded in w_1.
$G_5[P_m : w_2]$	$\frac{1}{3}$	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2} + \frac{1}{3}\right] \times \frac{1}{5} = 0.17$	P_m becomes stable.
$G_5[P_k : w_2]$	1	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2} + \frac{1}{3} + 1\right] \times \frac{1}{5} = 0.37$	P_k succeeds and is considered stable.
$G_4[P_i : w_1]$	$\frac{2}{2}$	$\left[\frac{0}{3} + \frac{1}{2} + \frac{1}{3} + 1 + \frac{2}{2}\right] \times \frac{1}{5} = 0.57$	P_i now becomes stable.
$G_5[P_k : w_2]$	1	$\left[\frac{1}{2} + \frac{1}{3} + 1 + \frac{2}{2} + 1\right] \times \frac{1}{5} = 0.77$	
$G_5[P_k : w_2]$	1	$\left[\frac{1}{3} + 1 + \frac{2}{2} + 1 + 1\right] \times \frac{1}{5} = 0.87$	
$G_5[P_k : w_2]$	1	$\left[1 + \frac{2}{2} + 1 + 1 + 1\right] \times \frac{1}{5} = 1.00$	

Table 4.1: Example executions of plan P_e (see Figure 4.3) in world w and the related stability-based confidence $\mathcal{C}_s$ calculation for $n = 5$.

4.1.2 World-Based Component Metric

The stability-based confidence measure $\mathcal{C}_s(P, w, n)$ defined above would make a useful heuristic for exploration (i.e., plan selection) in its own right: when the confidence is at its lowest the agent does maximum exploration, and when it is at its highest, the agent fully utilises the decision trees. Normally, however, generalising using decision trees is justified, and is useful, if one has collected "enough" data. For a plan, this equates to not only trying it in meaningful ways in a given world (as captured by the stability-based metric $C_s(P, w, n)$), but also *trying it in all the different worlds where it applies*. We will now define a second metric that quantifies this latter aspect.

Since we do not know upfront the full set of worlds where a plan may be considered, then one way to approximate this is by monitoring *the rate at which new worlds are being witnessed by a plan* P. During early exploration, it is expected that the majority of worlds that a plan is selected for will be unique, thus yielding a high rate (corre-

sponding to low confidence). Over time, as exploration continues, the plan would get selected in all worlds in which it is reachable and the rate of new worlds would approach zero (corresponding to full confidence). Given this, we define our second world-based confidence metric as

$$\mathcal{C}_d(P, n) = \frac{|\mathit{OldStates}(P, n)|}{n}, \tag{4.1.3}$$

where $\mathit{OldStates}(P, n)$ is the set of world states in the last n executions of P that have also been witnessed before. Clearly, $\mathcal{C}_d$ will converge to 1.0 after all worlds where the plan might be considered are eventually witnessed. However, it does not behave monotonically since it is quite possible that $\mathcal{C}_d$ increases to 1.0 before the full set of worlds is witnessed, meaning it would decrease when the remaining worlds are witnessed, before eventually converging to 1.0 again.

Referring back to our example of Figure 4.3, consider once again plan P_e, and assume that the (initially unknown) set of worlds where it may be considered is $\{w_1, w_2, w_3, w_4\}$. Table 4.2 shows a sequence of example executions of P_e over time in these world states and the related world-based confidence calculation for $n = 5$.

In the beginning there is no history to compare to. The first four executions of P_e (rows 1–4 of Table 4.2) are all in previously unseen worlds so $\mathcal{C}_d$ is zero. The fifth execution is in world w_2 which was seen before (just prior) so $\mathcal{C}_d(P_e, 5) = 1/5 = 0.2$.

The sixth execution of P_e is in world w_3 which also was witnessed before. So in the more recent n executions, i.e., $\{\mathbf{w_3}, \mathbf{w_2}, w_2, w_4, w_3\}$, there are now two worlds, i.e. w_3 and w_2 (highlighted in bold font), that have been seen before. So $\mathcal{C}_d(P_e, 5) = 2/5 = 0.4$.

After that, $\mathcal{C}_d$ gradually increases to its maximum as follows: for execution seven the old worlds in the last n executions are $\{\mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, w_2, w_4\}$ giving $\mathcal{C}_d(P_e, 5) = 3/5 = 0.6$; for execution eight the old worlds are $\{\mathbf{w_4}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, w_2\}$ giving $\mathcal{C}_d(P_e, 5) = 4/5 = 0.8$; while for the final execution the old worlds are $\{\mathbf{w_4}, \mathbf{w_4}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}\}$ and $\mathcal{C}_d(P_e, 5) = 5/5 = 1.0$.

Suppose, however, that the first six executions of P_e happened to be in world w_1 followed by w_2, w_3, and w_4. In this case, $\mathcal{C}_d$ would have increased directly from 0.0 to 1.0 by execution six, then decreased when w_2, w_3, and w_4 were witnessed, before eventually increasing to 1.0 on subsequent executions.

State	$\mathit{OldStates}(P_e, 5)$	$\mathcal{C}_d(P_e, 5)$	Explanation
w_1	0	0.0	State w_1 is new.
w_3	0	0.0	All states in w_3, w_1 are new.
w_4	0	0.0	All states in w_4, w_3, w_1 are new.
w_2	0	0.0	All states in w_2, w_4, w_3, w_1 are new.
w_2	1	$\frac{1}{5} = 0.2$	Bold states in $\mathbf{w_2}, w_2, w_4, w_3, w_1$ are old.
w_3	2	$\frac{2}{5} = 0.4$	Bold states in $\mathbf{w_3}, \mathbf{w_2}, w_2, w_4, w_3$ are old.
w_1	3	$\frac{3}{5} = 0.6$	Bold states in $\mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, w_2, w_4$ are old.
w_4	4	$\frac{4}{5} = 0.8$	Bold states in $\mathbf{w_4}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, w_2$ are old.
w_4	5	$\frac{5}{5} = 1.0$	Bold states in $\mathbf{w_4}, \mathbf{w_4}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}$ are old.

Table 4.2: Example executions of plan P_e (see Figure 4.3) over time, and the related world-based confidence $\mathcal{C}_d$ calculation for $n = 5$.

4.1.3 Dynamic Confidence Measure

In summary, we have defined two confidence metrics over two orthogonal dimensions. Stability-based confidence $\mathcal{C}_s(P, w, n)$ is meant to capture how well-informed the last n executions of plan P in world w were, whereas world-based confidence $\mathcal{C}_d(P, n)$ is meant to capture how well-known were the worlds in the last n executions of plan P.

With this, we now have all the necessary components to define our final confidence measure $\mathcal{C}(P, w, n)$ that we introduced earlier in Chapter 3 Equation 3.3.1. Specifically, this overall confidence in the decision tree of plan P in world w relative to the last n experiences is defined as follows:

$$\mathcal{C}(P, w, n) = \alpha\mathcal{C}_s(P, w, n) + (1 - \alpha)\mathcal{C}_d(P, n), \tag{4.1.4}$$

where α is a weighting factor used to set a preference bias between the two component metrics.

To illustrate the basic behaviour of the dynamic confidence $\mathcal{C}(P, w, n)$, let us take the example of plan P_e from Figure 4.3 once again. As before, suppose that the applicable plans are $\Delta_{app}(G_4, w_1) = \{P_i, P_j\}$ and $\Delta_{app}(G_5, w_2) = \{P_k, P_l, P_m\}$, that a plan

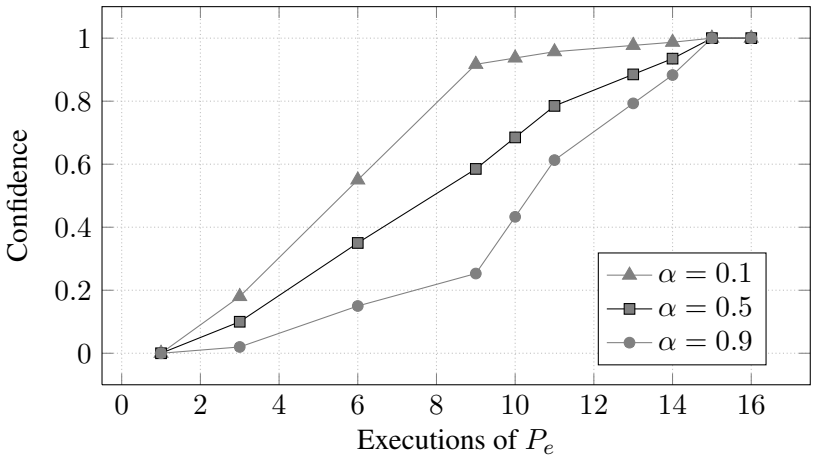


Figure 4.4: Dynamic confidence $\mathcal{C}(P_e, w_1, 5)$ over successive executions of plan P_e in world w_1 using $\alpha = 0.5$ (as calculated in rows 1-16 of Table 4.3). The impact of varying the preference bias α is also shown.

becomes stable after two executions in a given world, and that the set of worlds where P_e may be considered is $\{w_1, w_2, w_3, w_4\}$. Rows 1-16 of Table 4.3 describe an example run of P_e over time, and the related dynamic confidence $\mathcal{C}(P_e, w_1, 5)$ calculation for world w_1 using $n = 5$ and $\alpha = 0.5$. The same information is plotted in Figure 4.4.

The impact of changing the preference bias α is also shown in Figure 4.4 (plots $\alpha = 0.1$ and $\alpha = 0.9$). As can be seen, the choice of α marks a $\mathcal{C}$ trajectory that lies in between that of C_s (i.e., $\alpha = 1.0$) and $\mathcal{C}_d$ (i.e., $\alpha = 0.0$).

In order to see how the confidence measure $\mathcal{C}(P, w, n)$ behaves against changes in the environment, consider the remaining rows 17-24 of Table 4.3. After execution 16, a change in the environment causes the previous solution (shown in Figure 4.3) for world w_1 to no longer work. The new solution requires P_m to be selected instead of P_k for resolving the subgoal G_5 (there is no change to its applicable set).

The impact of the change is immediate when the agent tries the previous solution and fails (row 17). The failure of leaf plan P_k makes it "unstable" causing a change in the stability-based metric $\mathcal{C}_s$ (only plan P_m in the applicable set $\{P_k, P_l, P_m\}$ is now stable so $\zeta(\lambda) = \frac{1}{3}$) and consequently confidence $\mathcal{C}(P_e, w_1, 5)$ drops to 0.93. This

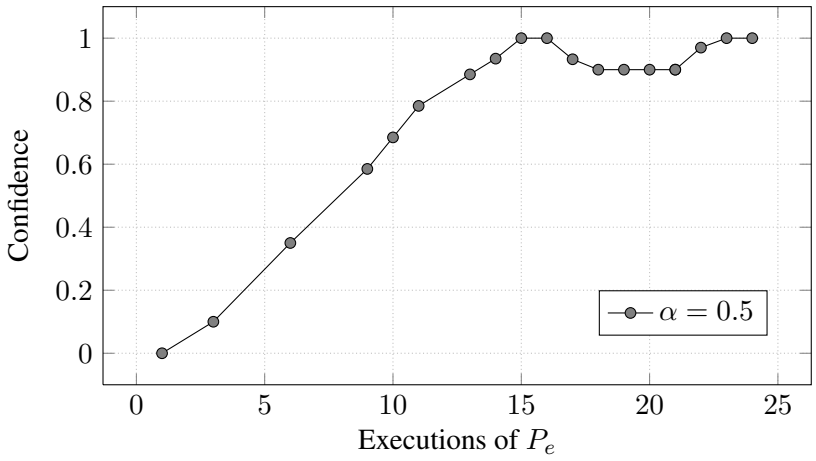


Figure 4.5: Dynamic confidence $\mathcal{C}(P_e, w_1, 5)$ over successive executions of plan P_e in world w_1 using $\alpha = 0.5$ (as calculated in rows 1-24 of Table 4.3). A change in the environment after execution (row) 16 causes the previous solution to no longer work. The figure shows how the confidence $\mathcal{C}(P_e, w_1, 5)$ dynamically adjusts to this change until the new solution is found (rows 17-24).

results in new exploration of the applicable plans for subgoal G_5 until the new solution is discovered (rows 18-20). Finally, as the new solution is repeatedly successful, the confidence $\mathcal{C}(P_e, w_1, 5)$ once again climbs to 1.0 (rows 21-24) as shown in Figure 4.5.

Table 4.3: Example executions of plan P_e (see Figure 4.3) and the final dynamic confidence $\mathcal{C}(P_e, w_1, 5)$ calculation in world w_1 for $n = 5$ and $\alpha = 0.5$. For legibility, the dynamic confidence calculations for other worlds in which P_e is executed are omitted (indicated by "...").

#	λ	$\mathcal{C}_s(P_e, w_1, 5)$	$\mathcal{C}_d(P_e, 5)$	$\mathcal{C}(P_e, w_1, 5)$
1	$G_4[P_j : w_1]$ & $G_5[P_l : w_2]$	$\left[\frac{0}{3}\right] \times \frac{1}{5} = 0.00.$ $\{P_k, P_l, P_m\}$ has no stable plans.	0.00. States $\{w_1, -, -, -, -\}$.	0.00
2	$G_4[P_j : w_3]$	...	0.00. States $\{w_3, w_1, -, -, -\}$	...
3	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[\frac{0}{3} + \frac{0}{3}\right] \times \frac{1}{5} = 0.00.$ $\{P_k, P_l, P_m\}$ has no stable plans.	$\frac{1}{5} = 0.2.$ States $\{\mathbf{w_1}, w_3, w_1, -, -\}$.	0.10

Continued on next page

Continued from previous page

#	λ	$\mathcal{C}_s(P_e, w_1, 5)$	$\mathcal{C}_d(P_e, 5)$	$\mathcal{C}(P_e, w_1, 5)$
4	$G_4[P_i : w_4]$	...	$\frac{1}{5} = 0.2$. States $\{w_4, \mathbf{w_1}, w_3, w_1, -\}$.	...
5	$G_4[P_i : w_3]$	...	$\frac{2}{5} = 0.4$. States $\{\mathbf{w_3}, w_4, \mathbf{w_1}, w_3, w_1\}$.	...
6	$G_4[P_i : w_1]$	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2}\right] \times \frac{1}{5} = 0.10$. $\{P_i, P_j\}$ has stable P_j (succeeded in w_1).	$\frac{3}{5} = 0.6$. States $\{\mathbf{w_1}, \mathbf{w_3}, w_4, \mathbf{w_1}, w_3\}$.	0.35
7	$G_4[P_j : w_2]$	...	$\frac{4}{5} = 0.8$. States $\{\mathbf{w_2}, \mathbf{w_1}, \mathbf{w_3}, w_4, \mathbf{w_1}\}$.	...
8	$G_4[P_i : w_3]$	...	$\frac{4}{5} = 0.8$. States $\{\mathbf{w_3}, \mathbf{w_2}, \mathbf{w_1}, \mathbf{w_3}, w_4\}$.	...
9	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2} + \frac{1}{3}\right] \times \frac{1}{5} = 0.17$. P_m now stable in $\{P_k, P_l, P_m\}$.	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, \mathbf{w_1}, \mathbf{w_3}\}$.	0.59
10	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[\frac{0}{3} + \frac{0}{3} + \frac{1}{2} + \frac{1}{3} + 1\right] \times \frac{1}{5} = 0.37$. P_k succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}, \mathbf{w_1}\}$.	0.69
11	$G_4[P_i : w_1]$	$\left[\frac{0}{3} + \frac{1}{2} + \frac{1}{3} + 1 + \frac{2}{2}\right] \times \frac{1}{5} = 0.57$. Both plans in $\{P_i, P_j\}$ now stable.	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_3}, \mathbf{w_2}\}$.	0.79
12	$G_4[P_i : w_2]$	...	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_2}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_3}\}$.	...
13	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[\frac{1}{2} + \frac{1}{3} + 1 + \frac{2}{2} + 1\right] \times \frac{1}{5} = 0.77$. P_k succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_1}, \mathbf{w_2}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.89
14	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[\frac{1}{3} + 1 + \frac{2}{2} + 1 + 1\right] \times \frac{1}{5} = 0.87$. P_k succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.0$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.94
15	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[1 + \frac{2}{2} + 1 + 1 + 1\right] \times \frac{1}{5} = 1.00$. P_k succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_2}, \mathbf{w_1}\}$.	1.00
16	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[\frac{2}{2} + 1 + 1 + 1 + 1\right] \times \frac{1}{5} = 1.00$. P_k succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_2}\}$.	1.00

Continued on next page

Continued from previous page

#	λ	$\mathcal{C}_s(P_e, w_1, 5)$	$\mathcal{C}_d(P_e, 5)$	$\mathcal{C}(P_e, w_1, 5)$
At this point, a change in the environment causes the previous solution for world w_1 (shown in Figure 4.3) to no longer work. Instead, the new solution requires that P_m be selected to resolve subgoal G_5 instead of P_k.				
17	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[1+1+1+1+\frac{1}{3}\right] \times \frac{1}{5} = 0.87$. P_k becomes unstable. Only P_m in $\{P_k, P_l, P_m\}$ is stable.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.93
18	$G_4[P_j : w_1]$ & $G_5[P_l : w_2]$	$\left[1+1+1+\frac{1}{3}+\frac{2}{3}\right] \times \frac{1}{5} = 0.80$. $\{P_l, P_m\}$ in $\{P_k, P_l, P_m\}$ are stable.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.90
19	$G_4[P_j : w_1]$ & $G_5[P_k : w_2]$	$\left[1+1+\frac{1}{3}+\frac{2}{3}+\frac{3}{3}\right] \times \frac{1}{5} = 0.80$. All plans stable in $\{P_k, P_l, P_m\}$.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.90
20	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[1+\frac{1}{3}+\frac{2}{3}+\frac{3}{3}+1\right] \times \frac{1}{5} = 0.80$. P_m succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.90
21	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[\frac{1}{3}+\frac{2}{3}+\frac{3}{3}+1+1\right] \times \frac{1}{5} = 0.80$. P_m succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.90
22	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[\frac{2}{3}+\frac{3}{3}+1+1+1\right] \times \frac{1}{5} = 0.93$. P_m succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	0.97
23	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$\left[\frac{3}{3}+1+1+1+1\right] \times \frac{1}{5} = 1.00$. P_m succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	1.00
24	$G_4[P_j : w_1]$ & $G_5[P_m : w_2]$	$[1+1+1+1+1] \times \frac{1}{5} = 1.00$. P_m succeeds so $\zeta(\lambda)$ assigned 1.	$\frac{5}{5} = 1.00$. States $\{\mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}, \mathbf{w_1}\}$.	1.00

4.2 Summary and Discussion

For a BDI agent learning to improve plan selection based on experience, an important consideration is how much to trust (and therefore exploit) what has been learnt so far

versus how much to explore to further improve the learning. In this chapter, we have described a dynamic confidence measure that combines ideas of plan stability [Airiau et al., 2009] and plan coverage-based confidence [Singh et al., 2010a,b] from earlier versions of this work, with a sense of the rate at which new worlds are being witnessed. This new confidence measure provides a simple way for the agent to judge how much it should trust its current learning, and adjust its exploration strategy accordingly. The measure dynamically adapts based on agent performance, allowing in principle, infinitely many learning phases. This means that our confidence in a plan's decision tree may not necessarily increase monotonically: it will drop whenever the learned behavior becomes less successful in the environment, thus allowing for new plan exploration to recover goal achievability. Finally, the new mechanism does not require any account of the number of possible choices below a plan in the hierarchy, as is the case with the earlier coverage-based approaches we had explored [Singh et al., 2010a,b], and hence scales up for any general goal-plan structure irrespective of its complexity.

Chapter 5

Experimental Evaluation[†]

We are now ready to describe experiments for testing the learning framework of Chapters 3 and 4. In this Chapter we present experiments with *synthetic* goal-plan hierarchies, i.e., testbed programs composed of several goals and plans combined in a hierarchical manner, and yielding goal-plan tree structures of different shapes.[1] We also show experiments to validate the use our learning framework with BDI failure recovery enabled, i.e., where alternative plans are tried when an initially selected plan fails. The environment for all experiments is modelled as *non-determinisitic* such that correct plans for a given world may nevertheless fail sometimes due to unknown reasons. The overall idea here is to empirically test performance in a controlled environment where various parameters may be systematically adjusted in order to better understand their impact on learning. Later, in Chapter 6, we will look at some actual BDI programs that also make use of parameterised goals and recursion.

5.1 Experimental Setup

We crafted goal-plan tree structures representing different cases of BDI programs with one main top-level goal to be resolved. For each structure there is always some way of

[†] Parts of the work presented in this chapter have been previously published in [Singh et al., 2010b].

[1] We have implemented the learning agent system in the JACK BDI platform [Busetta et al., 1999]. The fact that JACK is Java based and provides powerful meta-level reasoning capabilities allows us to integrate weka and probabilistic plan-selection mechanisms with little effort. Nonetheless, all the results are independent of this choice and could be reproduced in other BDI implementations.

addressing the main goal, i.e., there is at least one successful execution of the top-level goal provided the right plan choices are made. Of course, the successful sequence of plan choices will be different for different world states.

The world states in these experiments are described by a set of logical (binary) propositions, representing the *fluents* or features of the environment that are observable to the agent.[2] For instance, the fluent OutlookSunny states whether the outlook is believed to be sunny or not. We use boolean values in these experiments for simplicity. Of course, features of the world could also be represented as multi-valued variables, such as, outlook $=$ sun, outlook $=$ rain, and so on.

Each experiment consisted of posting the top-level goal repetitively under random world states, and recording (against every chosen plan) whether the execution terminated successfully or not. We calculate the average rate of success of the goal by first averaging the results at each time step over *five* runs of the same experiment, and then smoothing using a moving average of the previous 100 time steps to get the trends reported in the figures.

For stability calculation we used a threshold of 0.3, i.e., a plan is considered to be stable when the difference between its two consecutive rates of success (where the rate of success is the ratio of successful to total executions) is under 0.3.

Finally, we assume the agent is acting in a non-deterministic environment in which actions that are expected to succeed may still fail with some probability. In our experiments we assign a 0.1 probability of unexpected failure to all actions.

5.2 Performance Under Various Goal-Plan Hierarchies

From our set of experiments, we have selected three hierarchical structures, namely $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, that best illustrate the results that we have obtained. We will now describe them one by one and show how learning progresses under each of them.

Structure $\mathcal{T}_1$ (Figure 5.1) In this hierarchy, the agent has 20 options of comparable complexity to resolve the top-level goal G. For each world state, the top-level

[2] To handle continuous attributes (e.g., *temperature*) our approach requires that either these attributes are discretised (e.g., *cold*, *warm*, and *hot*), or additional discrete attributes be used to test the continuous ones (e.g., *temperature* < 25.2).

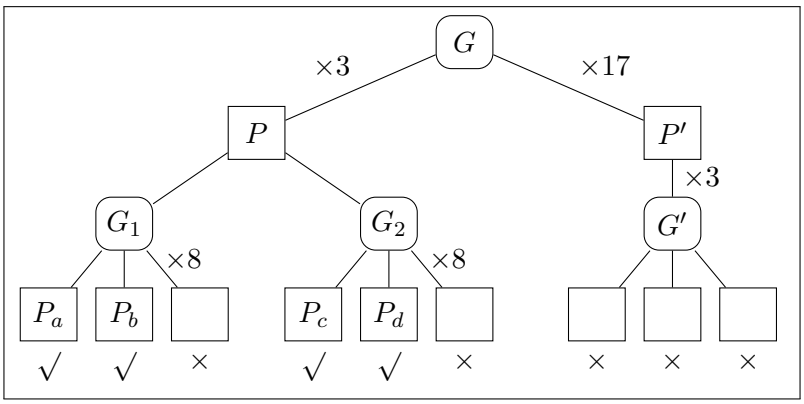


Figure 5.1: Goal-plan structure $\mathcal{T}_1$.

goal G has a few plans that can succeed (three plans of a similar structure to the plan P shown), but many other options of comparable complexity [3] that are bound to fail (17 plans identical to plan P' that fail in every world). This is an extreme case for illustrative purposes: of course plan P' should, in a real program, contain a solution in some world states or it would not make sense to include them. To succeed, the agent must make three correct choices. For example, in Figure 5.1, which shows the solution for some given world, the agent must choose plan P for goal G, plan P_a or P_b for goal G_1, and plan P_c or P_d for goal G_2. There are a total of 2^3 world states and the solution for each lies in the same sub-tree P, although the complete sequence of plan choices is different for different world states.

The key feature of this setup is that at any given point, the agent has many options to choose from. Recall that the choice of plan depends (probabilistically) on its selection weight (Equation 3.3.1 in Chapter 3), that in turn depends on the stability of ongoing choices. Therefore, one may expect that the more choices the agent has, the longer it will take to accurately determine the correct plan in each world state.

In structure $\mathcal{T}_1$, the initial selection weight for each top-level plan is 0.5 (according to Equation 3.3.1 in Chapter 3). As each plan is tried and the respective decisions begin to stabilise, these selection weights will slowly converge to the plans' decision

[3] Here, plan complexity refers to the size of the fully expanded plan, as represented by the number of levels of abstraction and the number of goals at each level.

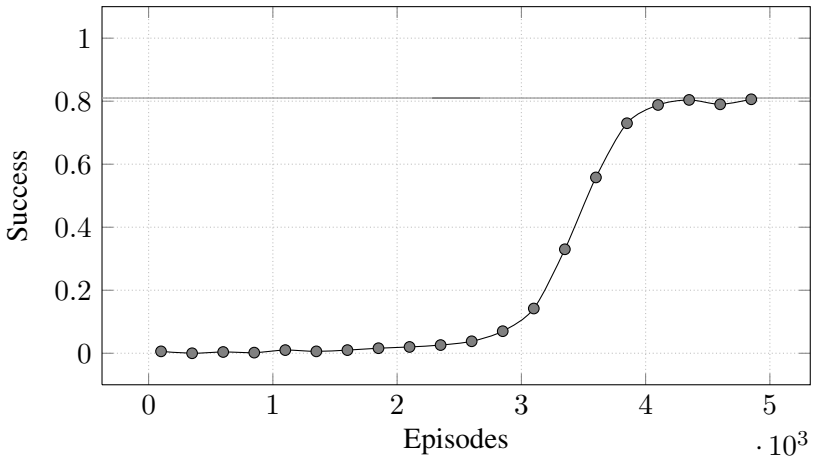


Figure 5.2: Agent performance under structure $\mathcal{T}_1$. Optimal performance is 81% (solid line) since the solution requires two correct leaf plans to be selected, however each has a 10% (non-deterministic) likelihood of failure.

tree predictions. From Figure 5.1 we can see that this will happen first for the plans labelled P' since none of its three subgoals ever succeed: in fact only the first is ever tried as it always fails.[4] As such, the selection weights for these choices will rapidly converge to zero. On the other hand, the P plans take much longer to converge to the true decision tree probabilities due to the sheer number of choices (10 each for the subgoals G_1 and G_2). In relative terms then, the selection weights for the P plans will become higher than the P' plans as learning progresses. This also means that the selection probabilities for the three P plans will gradually increase from the initial $3/20$ to the final $20/20$ while that of the P plans will decrease from the initial $17/20$ to $0/20$ (albeit at a different rate). Figure 5.2 shows this sigmoidal transition from zero success to optimal (the two leaf actions each have a 10% chance of non-deterministic failure so optimal is 0.9^2 or 81%). The reason why it takes almost 3000 episodes before performance starts to improve, even though we have only 2^3 world states, is because there are significantly more "bad" options for every correct choice (for instance, at the top level, for any given world, only one plan in 20 possibilities has the solution), and it takes time for the agent to become confident in this knowledge.

Structure $\mathcal{T}_2$ (Figure 5.3) In this structure, all successful executions (for 2^3 worlds) are

[4] Plan P' will become stable even though the last two of its three subgoals are never tried. For a detailed example of how stability is calculated, see page 53.

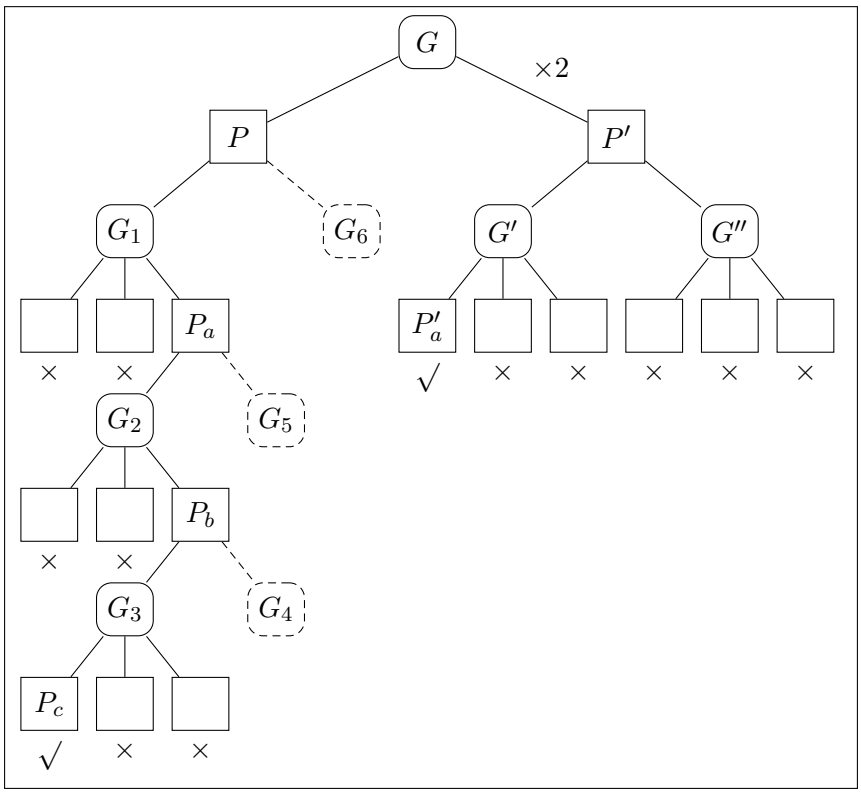


Figure 5.3: Goal-plan structure $\mathcal{T}_2$.

encoded in a complex plan P. The other two options (similar to plan P' shown) are of less complexity, but do not contain solutions for any world (only subgoal G' has a solution in plan P'_a but G'' always fails). Since the plan containing the solution, namely P, is fairly complex, there are many ways the agent may fail when exploring the decomposition of P. The agent needs to make several correct choices to obtain a successful execution in resolving the subgoals $G_1 \ldots G_6$. Here, subgoal G_4 has a similar structure to G_3, G_5 is similar to G_2, and G_6 to G_1. Overall, the successful resolution of G requires 15 correct plan choices including eight leaf-plan choices (similar to P_c), and the optimal performance in our non-deterministic world is 0.9^8 or 43%.

The important difference from the previous structure $\mathcal{T}_1$ is that the plan containing the solution is of substantially higher complexity than before. Where the former structure $\mathcal{T}_1$ shows the impact of the breadth of the hierarchy, this structure shows the impact of the depth of the hierarchy. It is easy to see in Figure 5.3 that the P' plans, that are much

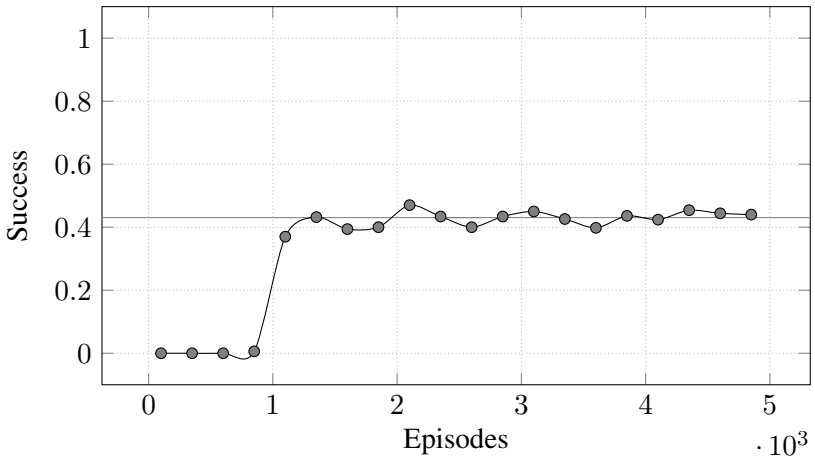


Figure 5.4: Agent performance under structure $\mathcal{T}_2$. Optimal performance amounts to 43% since the solution requires the selection of eight non-deterministic leaf plans. (Outcomes are always 0 or 1 so more than expected consecutive successes may seem like "above" optimal performance when averaged.)

shallower and have far less potential choices than P plans, will stabilise quicker, resulting in their selection weights converging to their decision tree probabilities quicker: in this case zero, since P' plans never succeed. Once this has occurred, the agent will explore the P plans almost exclusively until the solutions are found. Figure 5.4 shows the result. Observe that even though the sequence of plan choices is far greater than $\mathcal{T}_1$ (eight leaf choices compared to two), and the real likelihood of success much lower (43% compared to 81%), the agent converges to the solutions much quicker (≈ 1000 episodes compared to ≈ 4000). This result may seem non-intuitive, however it clearly shows that the length and quality of the solution is not always the governing factor in performance, and in fact the structure of the hierarchy (i.e., the domain know-how) plays a very important role.

Structure $\mathcal{T}_3$ (Figure 5.5) This hierarchy represents a more "balanced" structure than the previous ones. Furthermore, whereas previously the "bad" plans were relatively simple, here they are significantly more complex. This is because the solutions for the world states (2^4 in all) are evenly distributed among the four potential choices $\{P_1, P_2, P_3, P_4\}$, that are all of a similar (high) complexity. For any given world state, only one particular path leads to a successful execution and this is different for different world states. For instance, Figure 5.5 shows the

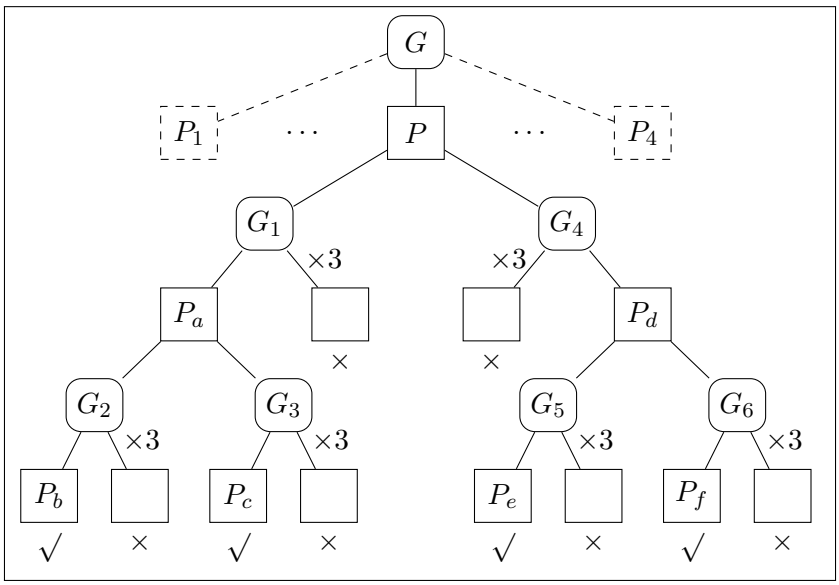


Figure 5.5: Goal-plan structure $\mathcal{T}_3$.

solution (for some world) that requires selecting the top-level plan P, and eventually the leaf plans P_b, P_c, P_e, and P_f, (a total of seven choices). Among other things, this means that the top-level plan selection is very important as an incorrect choice is bound to lead to failure. We argue that this is a common feature found in many BDI agent applications, in that even though the agent has been programmed with several strategies for resolving a goal, each one is crafted to cover uniquely a particular subset of states.

One may expect the agent performance under structure $\mathcal{T}_3$ to be somewhere in between the former two as it captures important aspects of both: it has complex solution structures as well as sufficiently rich alternatives that fail; also the solution requires the choice of four appropriate leaf plans, compared to two and eight before. Figure 5.6 shows the results. As expected, the performance in this case is indeed midway, and the convergence to optimal occurs around 2500 episodes.

In summary, the structures $\mathcal{T}_1$, $\mathcal{T}_2$, and $\mathcal{T}_3$, show what impact the BDI goal-plan hierarchy can have on the learning performance. Overall learning in general is impacted by several factors including the number of choices at each decision point and their complexity, the number of actions (leaf node plans) in the set of choices, and the non-determinism in the environment. In our suite of experiments we have tested for a range

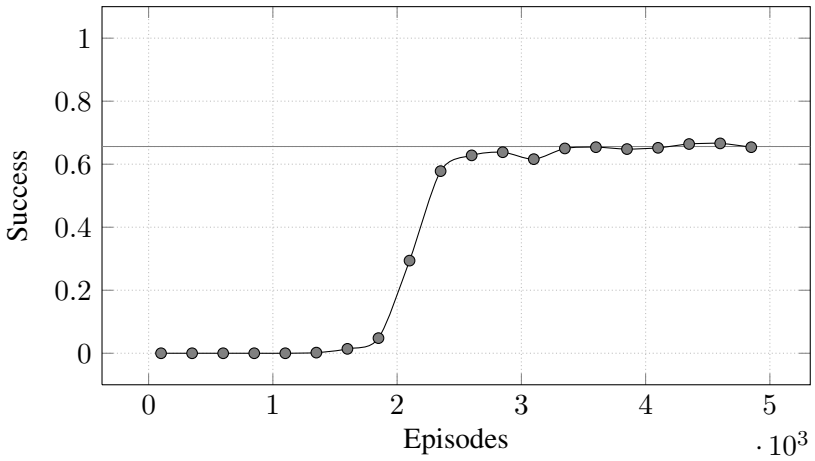


Figure 5.6: Agent performance under structure $\mathcal{T}_3$. Optimal performance in this case is 66%, resulting from four non-deterministic leaf plan choices.

of these factors, and the hierarchies and experiments described here summarise our key results.

5.3 Impact of Failure Recovery

The BDI failure recovery mechanism (see Chapter 3 Section 3.4) allows the agent to reconsider its options if the plan initially selected to resolve a goal were to fail. To evaluate the impact of failure recovery on learning, we ran experiments with two different goal-plan hierarchies, first with, and then without, failure recovery enabled.

Structure $\mathcal{T}_4$ (Figure 5.7) This structure has four relevant plans for resolving goal G depending on the world state (described by the binary fluents a, b, c, and z). The solutions are evenly distributed among these options: plan P_a succeeds in all states where $\bar{a}\bar{b}\bar{z}$ holds (i.e., a total of two states since we don't care about the value of c), plan P_b succeeds for $\bar{a}b\bar{z}$ (i.e., two states), P_z succeeds for z (i.e., eight states) and P_c succeeds for the remainder (i.e., four states). Plans marked with a $\times$ symbol always fail in every world they are invoked. All plans that fail, including correct plans that fail non-deterministically, have the side-effect of toggling z.

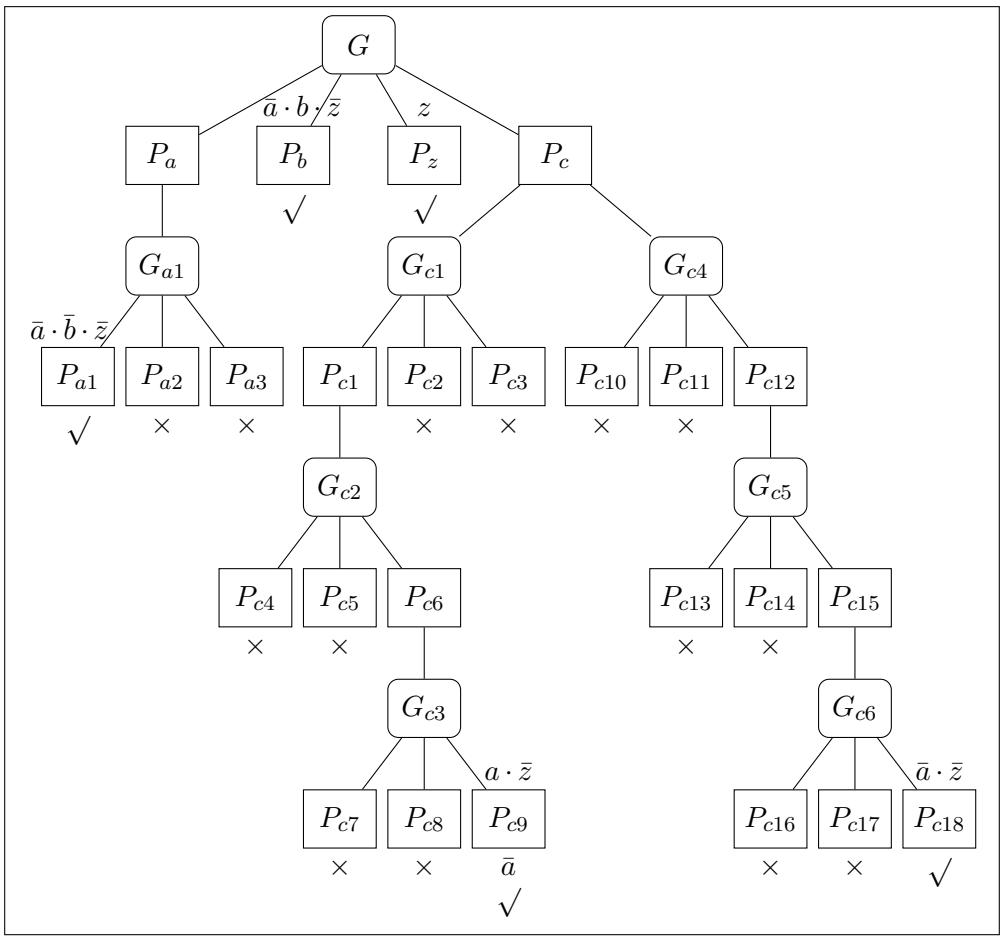


Figure 5.7: Goal-plan hierarchy $\mathcal{T}_4$.

The side-effect of failures in structure $\mathcal{T}_4$ has implications for failure recovery since an incorrect choice may impact the world state adversely. Consider the case where the initial world state is $\bar{a}\bar{b}cz$. Here clearly plan P_z is the correct choice. However, say that the agent instead selects plan P_a, then P_{a3} that fails and toggles z, so that the new world state is $\bar{a}\bar{b}c\bar{z}$. Since recovery is enabled, the failure of plan P_{a3} does not immediately imply the failure of goal G_{a1}. Instead, goal G_{a1} is reposted and other available options considered.

Observe that the initially (i.e., in state $\bar{a}\bar{b}cz$) applicable plans for goal G were $\{P_a, P_z, P_c\}$ and for G_{a1} were $\{P_{a2}, P_{a3}\}$. After the failure of P_{a3}, the applicable set for G_{a1} in the resulting state $\bar{a}\bar{b}c\bar{z}$ is $\{P_{a1}, P_{a2}\}$ (plan P_{a3} would also normally apply here but it has already been tried and so is not included again). Say the agent were to select P_{a1} this time which succeeds (as shown in Figure 5.7) meaning the top-level

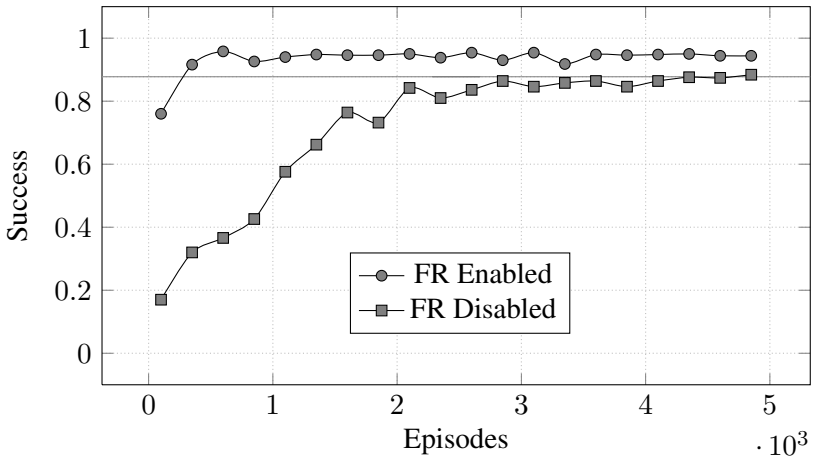


Figure 5.8: Agent performance under structure $\mathcal{T}_4$.

plan P_a also succeeds. Now clearly, we do not want to learn that P_a is the correct choice for the initial world $\bar{a}\bar{b}cz$. Indeed it is not, and the only way for it to succeed in this world is to first fail using P_{a3} (or P_{a2} for that matter) and then select P_{a1}. In fact, this is precisely the kind of learning we wish to avoid, as described earlier in Chapter 3 Section 3.4.

Figure 5.8 shows the results for structure $\mathcal{T}_4$ with and without failure recovery enabled. The solid line shows the optimal performance of 87.75% (the combined non-deterministic failure of leaf action sequences). The performance of the system without failure recovery enabled is as expected and gradually converges to this optimal.

The performance with failure recovery enabled requires some explanation. Observe that in Figure 5.8, the performance with failure recovery enabled is higher than the average expected success of the top-level goal G (solid line). The reason is that for every time that a correct plan choice fails (due to the inherent non-determinism), the system performs failure recovery and often finds a solution in a different plan (as illustrated by the success of P_a in the example earlier). Although these successes are not used for learning purposes, they still constitute successes of the top-level goal G, hence the anomaly.

One must be careful in taking the results of Figure 5.8 at face value, since they only compare performance in terms of the success of the top-level goal G. *An equally valid, and in some cases more appropriate, performance measure might be the number of actions taken to achieve the top level goal, regardless of how many times the top goal*

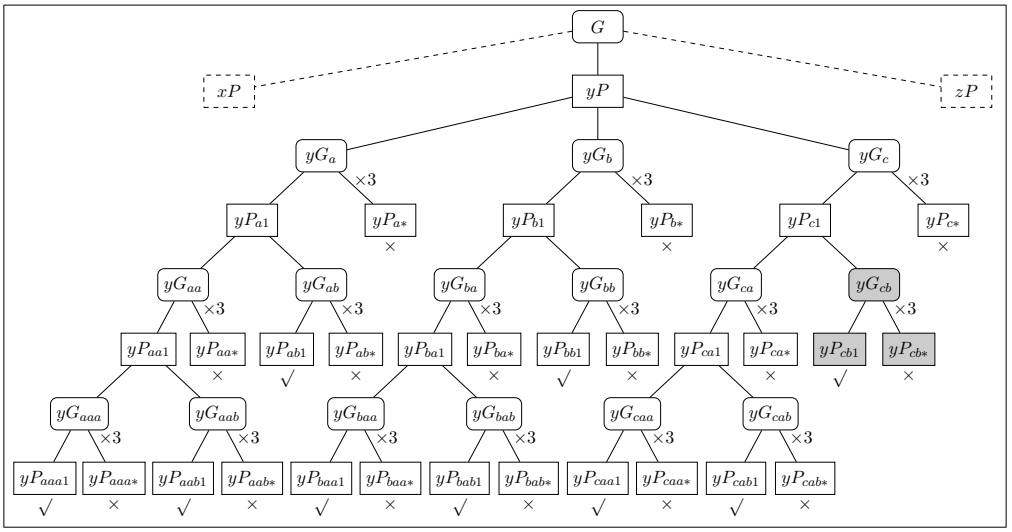


Figure 5.9: Goal-plan hierarchy T_5 for a world with five fluents $\{a, b, c, d, e\}$. All solutions exist in plan yP. Leaf plans marked $\times$ always fail and have the side-effect of toggling *some* randomly selected state variable.

was posted. Clearly, it is possible that were performance measured in such terms, that failure recovery may well fare poorly. To see if this were the case, we calculated the average number of actions taken in each case to achieve the first 100 successes. Indeed, we found that in the case without failure recovery the average number of actions per success was 3.97, whereas with failure recovery this number was more than double at 8.86.

Intuitively, one might expect to do better with failure recovery, in terms of the number of actions, in structures where the solutions are harder to find. Consider, for instance, a the more complex structure $\mathcal{T}_5$ of Figure 5.9.

Structure $\mathcal{T}_5$ (Figure 5.9) The hierarchy has three plans $\{xP, yP, zP\}$ to handle the top-level goal G: each structurally the same as the other. The only difference is that plan yP contains the solutions while the other two always lead to failures: subgoal xG_{cb} in plan xP and subgoal zG_{cb} in zP (the related subgoal yG_{cb} in plan yP is shaded in Figure 5.9) have no solutions and always fail. Importantly, to get to the final subgoal in each hierarchy (i.e., subgoal xG_{cb}, yG_{cb}, and zG_{cb}; let's collectively call these $*G_{cb}$) the agent must make a total of 15 correct plan choices before it can fully determine if a solution exists or not. All failures have

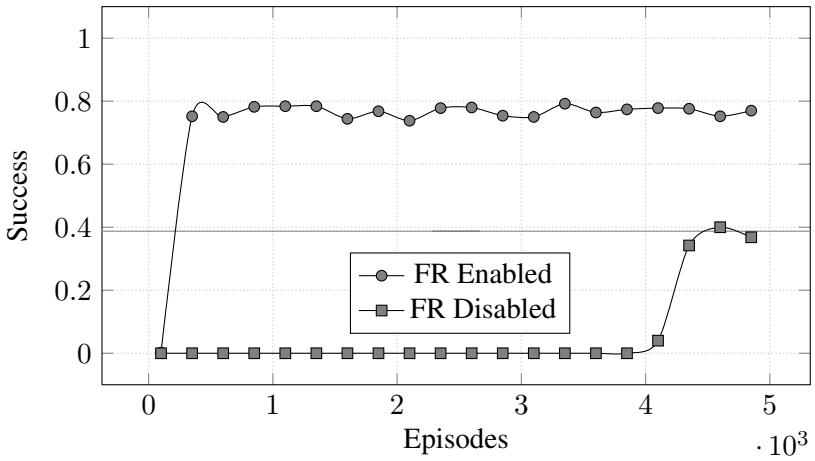


Figure 5.10: Agent performance under structure $\mathcal{T}_5$.

the side-effect of toggling *some* randomly selected fluent. There are a total of 2^5 world states.

Let's assume that the agent has in fact done the hard work and managed to get to the point where it must choose between one of the four leaf choices, i.e., $\{*P_{cb1}, \ldots, *P_{cb4}\}$ to resolve subgoal $*G_{cb}$. It is easy to see that if the agent now made a bad choice that failed it would have to repeat all the hard work to get here again (i.e., make the preceding 15 choices from scratch again), before it can try one of the remaining options. On the other hand, were failure recovery enabled, the agent would try all options for $*G_{cb}$ now (albeit in altered world states), until one succeeded or all failed. In this case, one may expect that it will take less actions to find the solution with failure recovery than without.

Figure 5.10 shows the results. As was the case earlier, the number of top-level goals resolved is much higher with failure recovery enabled than without (optimal performance being 38.74%). When we also measured the average number of actions for the first success in both cases, we found that, contrary to the previous experiment, indeed the agent took less actions with failure recovery enabled (5573) than without (7128). This despite the fact that failures have side-effects: they toggle the value of some randomly selected fluent.

A final observation for both structures $\mathcal{T}_4$ and $\mathcal{T}_5$ is that the number of training samples collected and the size of the associated decision tree is significantly larger with fail-

ure recovery enabled than without. This is because more plans are tried when failure recovery is enabled: often in states that would not eventuate otherwise under normal operation (and when there are no external changes to the environment).

In summary, our experiments with structures $\mathcal{T}_4$ and $\mathcal{T}_5$ aim to illustrate the usability of our framework when failure recovery is enabled. In general, we envisage that failure recovery will always be enabled in the kind of BDI applications where our learning framework is used. Our overall aim is to benefit as much as possible from the robustness of BDI systems, but at the same time make them more *adaptable* by adding a learning capability. The idea is to use learning to dynamically adjust existing behaviour (where indeed failure recovery makes good sense), when (certain) changes in the environment cause the initially programmed (or learnt) behaviour (context conditions) to no longer work effectively. In Chapter 6 we describe one such domain, and discuss the development and implementation of a complete BDI controller agent for a modular battery storage installation.

5.4 Understanding Plan Applicability

So far, we have assumed that the agent considers all plans that are relevant for a goal to also be *applicable*, even though some may have a very low chance of success. This means that, in contrast with standard BDI systems, our extended learning BDI framework will *always* select a plan from the relevant options. Since executing a plan is often not cost-free in real systems, it is desirable that an adequate plan selection mechanism in fact *not* execute plans with too low a probability of success. This in turn implies that the system may decide to fail a goal without even trying it, if it believes that the high likelihood of failure does not justify the cost of attempting any of the candidate plans. This is precisely what typical BDI systems do: when no applicable plan is found for a certain event-goal, that event-goal is failed immediately.

To understand the impact of plan applicability in our framework, we modified the probabilistic plan selection mechanism so that the agent does *not* consider plans whose likelihood of success is below a given threshold. For our next test, we set this threshold to 20%, and re-ran the previous experiment with structure $\mathcal{T}_2$. The result was the same as that reported in Figure 5.4, *the only difference being that the number of leaf plans actually tried in this case were significantly less that before.*

The threshold value for plan applicability is something that must be chosen with some consideration on the part of the user. For instance, a threshold of 20% for structure $\mathcal{T}_2$ seems *reasonable* since the real likelihood of success of plan P (Figure 5.3) is 43%. In general, by setting the threshold too low the agent may often try actions that are not very meaningful in the given situation, whereas by setting it too high it may risk not finding the solution at all. The difference between the default plan selection weight (i.e., 0.5 in Equation 3.3.1 of Chapter 3) and the threshold value (i.e., 0.2 in this case) decides how much "give" we have in the exploration. The closer the threshold is to the default weight the greater the chance that the plan will be aborted before a solution is found. An option here is to use a dynamic threshold value that starts off low when our confidence (Chapter 4) is also low, and gradually increases as our understanding of the domain improves: however we have not implemented this yet. Nevertheless, our aim here is to describe the impact of plan applicability on learning. In the next Chapter we will show how plan applicability may be used in a meaningful way when the cost of executing actions is significant, in a battery storage application.

Chapter 6

Developing BDI Systems that Learn[†]

In this Chapter we describe two complete BDI systems that we have implemented in the JACK agent-programming language and that utilise our learning framework described in Chapters 3 and 4:

1. **Towers of Hanoi** The Towers of Hanoi [Petković, 2009] game consists of three pins and (in our case) five discs of different sizes that may be placed onto the pins. The aim of the game is to build an ordered single tower with the biggest disc at the bottom and the smallest disc on top (Figure 6.1). The rules allow only one free disc to be removed at a time, to be placed on top of another larger disc or an empty pin. Similar to the well-known Blocks World domain [Fahlman, 1974; Nilsson, 1982; Slaney and Thiébaux, 2001; Winograd, 1971], an important feature of this domain is that a solution is always possible from any intermediate game state.

   We chose this application as an initial test of our learning framework as we had access to a benchmark implementation of the game that came packaged with the JACK [Busetta et al., 1999] agent system distribution. We essentially converted this into a learning system by replacing the context conditions of the original plans with decision trees. This also gave us a clear evaluation criteria: *is our learning framework able to achieve the performance of the existing system?*

2. **Modular Battery System Controller** In this application the task is to build a

[†] Parts of the work presented in this chapter have appeared or will appear in [Singh et al., 2010a, 2011].

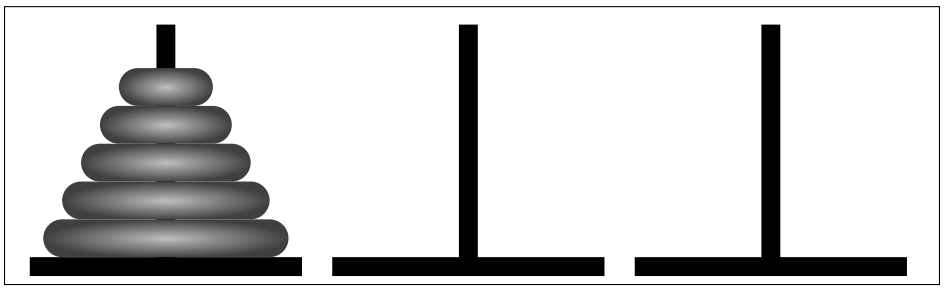


Figure 6.1: The Towers of Hanoi game.

controller for a large battery installation that consists of five individual modules each with its own operational constraints (due to different chemical properties, for instance). The aim is to learn to configure the battery to deliver a desired charge/discharge rate on every cycle by configuring the individual modules appropriately. An important consideration, however, is that the environment dynamics is not fixed and may change frequently and without any explicit notification. This means that learnt behaviours may often become sub-optimal, requiring unlearning and relearning on a continuous basis.

The implementation we describe here is a simplified version of such a battery controller. Even so, it is useful for showing how a real system may be developed using our learning framework.

6.1 Towers of Hanoi

The original Towers of Hanoi application included in the JACK distribution contains a Player agent that solves the game for any given legal initial configuration. The agent's high level strategy for solving the game is to build a tower of discs on pin number 2, and is encoded in the top-level plan BuildTower. To achieve this, it uses the plan DiscStacker that stack the discs one by one on pin 2, starting from the largest disc (disc n) and ending with the smallest disc (disc 1). Each disc stacking is realised by the (successful) achievement of a subgoal event Solve(?d,?p): move disc ?d to pin ?p. Event Solve(?d,?p) is indeed the most interesting and complex one from a learning point of view, since it is posted recursively with different event parameter values. To resolve this event, the agent has four plans at its disposal:

SolveTopMove This is the plan that performs the physical move of the disc. It applies when disc ?d is not on the destination pin ?p and it is movable (i.e., it is on top of some other pin), and a move to the destination pin ?p is valid (i.e., the top disc on the destination pin ?p is *larger* than disc ?d). In that case, the disc is just moved to the destination pin by performing the single primitive action move(?p2,?p), where ?p2 is pin where disc d is located.

SolveTop This is a recursive plan. It solves for the case where disc ?d is not on the destination pin ?p and it is movable, however the move to the destination pin ?p is invalid (i.e., the top disc on the destination pin ?p is *smaller* than disc ?d). In this case, the plan first recursively solves moving all the discs in the destination pin ?p that are smaller than disc ?d to the third (auxiliary) pin, and then re-posting the original subgoal to move disc ?d on to pin ?p, i.e., event Solve(?d,?p).

SolveMiddle This is a recursive plan. It solves moving a disc from the *middle* of a stack. The plan first cleans up all the discs above disc ?d so that ?d becomes free to move. This is done by posting the subgoal Solve(?d2,?p2) where disc ?d2 is the disc currently on top of the disc to be moved and ?p2 is the (auxiliary) third pin. Once disc ?d is at the top of the pin, the plan re-posts the original subgoal of moving it to pin ?p, i.e., event Solve(?d,?p).

SolveRight This plan solves moving a disc to the pin it is already on, i.e., disc ?d is on pin ?p. Since the goal is already true, the plan does *nothing* and is simply used to terminate the recursion in the program.

Figure 6.2 shows the goal-plan hierarchy of the system. For brevity, only Solve(i,p) (i.e., the case when plan DiscStacker is moving the i-th disc to the destination pin ?p) is expanded; all the other instances of the goal posted by DiscStacker have the same form. The plan-library relies heavily on parametric events and recursion to resolve Solve(?d,?p) since the SolveMiddle and SolveTop plans both utilise the same event type to achieve subgoals: their strategy being to first clear an obstruction and then repost the original goal. The first subgoal for these plans uses some disc ?d2 and pin ?p3 that are computed by the plan body. For example, in plan SolveMiddle, disc ?d2 is the disc that is currently on top of disc i and ?p3 is the third pin different from the destination pin ?p and the pin where i is located.

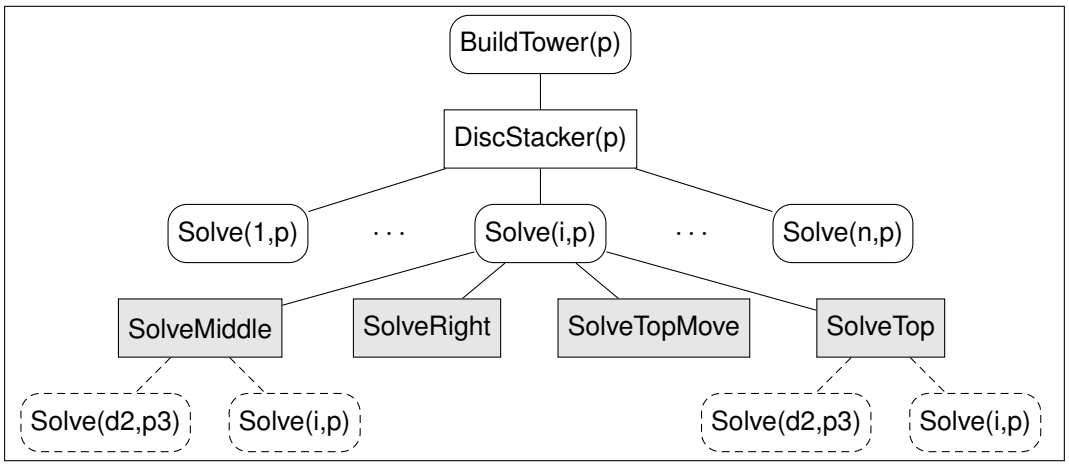


Figure 6.2: Goal-plan hierarchy for the Towers of Hanoi game.

The existing system of Figure 6.2 is a fully functional solution in that *it always solves the problem for any given initial configuration*. In other words, all plans in the existing application were programmed with *correct* context conditions. The context condition of plan SolveRight, for instance, checks to see if the current location of the disc ?d that is to to be moved does in fact match the destination pin ?p. Similarly, the context condition of plan SolveTop checks that there is a disc that is smaller than disc ?d on top of the destination pin ?p.

Since our aim is to see if we could learn to resolve the Solve(?d,?p) event, the first step in our experiment involved *deleting* all preconditions from the plans that handle that event, i.e., SolveTopMove, SolveTop, SolveMiddle, and SolveRight. This meant that initially each plan was, in principle, always feasible; however, after experimenting enough in the domain, the agent would eventually (hopefully) *learn* the preconditions of each plan.

Two problems arise when plans are stripped of their original context conditions. First, some plans may no longer be "self-sufficient," in that their logic relied on variables obtained in the context condition. For instance, consider the context condition $number(x) \wedge (x > 10)$ that binds the logical variable x and performs a test on that value. The variable x may then be used in the plan procedure. If we were to simply remove the context condition of the plan, then variable x will be unbound prior to its use in the plan's body, leading to an error. We solve this by requiring that the plan body indeed be self-sufficient: it must be executable just by itself. So where a plan procedure de-

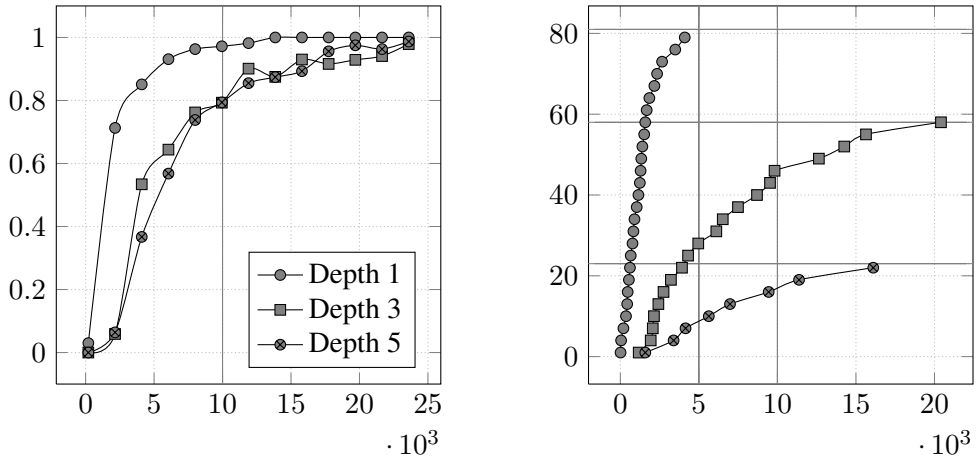
pends on variables bound in the context condition (e.g., variable x), these bindings (e.g., $number(x)$) must be transferred to the plan body.

The second problem is that plans may succeed in their execution *without* actually realising the goal. For instance, the body of plan SolveRight simply states that the disc on top of the pin where ?d is located be moved to the destination pin ?p, if the move is legal. Of course, the original context condition checks to ensure that in fact ?d is located on top of its pin. However, when we remove the context condition, then the plan will apply also when ?d is not on top of its pin. In this case, the plan may still succeed by moving whichever pin is on top to the destination pin ?p. This, however, does not achieve the goal since disc ?d has not moved at all, let alone to the correct pin. To overcome this problem, we require that every plan include, as its final step, a test condition to check for the goal it is meant to achieve. In this example then, we require that all four plans for event Solve(?d,?p) test that indeed disc ?d is on pin ?p.

6.1.1 Experimental Setup

Our experimentation focusses on learning to resolve the recursive event Solve(?d,?p) only and not on learning the strategy that solves the full Towers of Hanoi problem, i.e., plan DiscStacker(?p). We proceed by running the *existing* JACK program for a number of randomly generated Solve(?d,?p) events in randomly generated disc configurations (but valid according to the game rules). For each run we record the Solve(?d,?p) event, the initial pins configuration, and the maximum recursion encountered for the solution. This gives us sets of initial configurations that have solutions at different recursion levels. Next, using an appropriate set, we run several experiments whose solutions all lie at some known recursive depth, in order to understand how learning progresses for increasingly more difficult problems.

When performing multiple runs of the same experiment, we use a fixed random generation seed so that the same sequence of Solve(?d,?p) events is generated each time from the same initial configuration. This allows us to estimate the average rate of success for a sequence of Solve(?d,?p) goals across several runs. The result for each goal can still differ across runs because the agent selects plans probabilistically. The trends reported in the figures are obtained by first averaging the results at each time step over *five* runs of the experiment, and then smoothing using a moving average of the previous 100 time steps.



(a) Average Success

(b) Solutions Found

Figure 6.3: Performance of the system in terms of the average success and number of solutions found (y axis) against the number of episodes (x axis), for solutions at recursion depths one, three, and five.

The following results are for a Towers of Hanoi problem with *five* discs. We use five discs in order to keep the state space rich enough yet sufficiently small to allow learning runs to be completed and evaluated in reasonable time. In all experiments, the recursion is bound to a maximum of eight levels that is sufficient to solve all configurations for a five-disc Hanoi problem.

6.1.2 Results

Our initial experiments were designed to help us understand how the system performs for solutions of varying difficulty. For this we conducted a set of tests that consisted of learning to resolve a given set of Solve(?d,?p) events, saved earlier as explained in Section 6.1.1, and whose solutions all required the same recursive depth.

Figure 6.3 shows the results for solutions at recursive depths one (Depth 1), three (Depth 3) and five (Depth 5) respectively. The subfigures illustrate different aspects of the same experiment. First, Figure 6.3(a) shows the performance of the system in terms of the average success of the Solve(?d,?p) event (plotted on the y axis) against the number of episodes (on the x axis). As can be seen, the system learns the simpler solutions (Depth 1) much earlier than the deeper solutions (Depth 3 and Depth 5). The performance for

the deeper solutions (Depth 3 and Depth 5), however, is relatively similar.

To analyse the result further, for each recursive depth, we also plotted the total number of unique solutions discovered during the course of the same experiment. Figure 6.3(b) shows this data for Depth 1, Depth 3 and Depth 5, along with the maximum number of solutions at each level (solid lines at 23, 58 , and 81 respectively). The difference between the performance at each level (in terms of finding new solutions) is much more obvious here. For instance, by 5000 episodes (vertical line in Figure 6.3(b)), the agent had discovered all 81 solutions that are one level deep (Depth 1), compared to 28 solutions three levels deep (Depth 3) and only 8 solutions five levels deep (Depth 5).

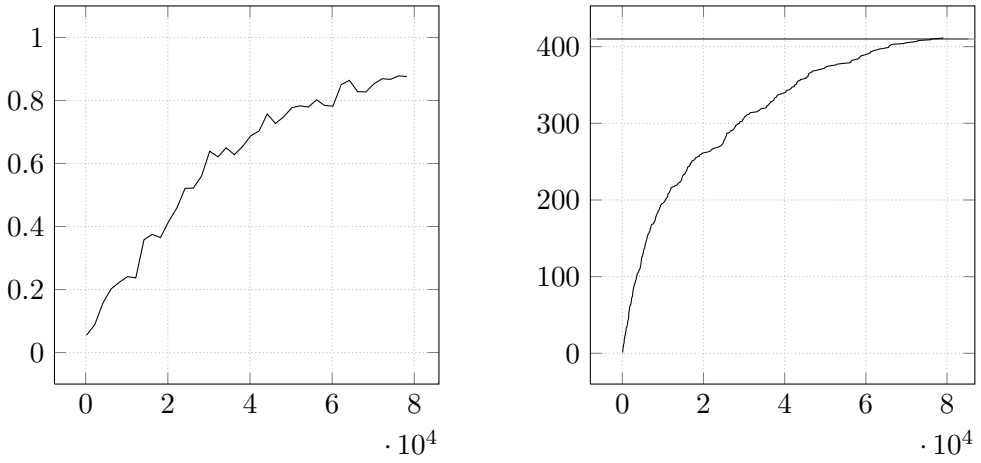
It is clear from Figure 6.3(b) that deeper solutions take longer to discover. However, as observed earlier, this fact does not seem to reflect in Figure 6.3(a) where the plots for Depth 3 and Depth 5 are quite similar. The reason is that Figure 6.3(a) results are indicative of the relative ratio of goals solved, not absolute. For instance, observe that the average success at $10,000$ episodes for Depth 3 and Depth 5 in Figure 6.3(a) is around 79%. In Figure 6.3(b) we can find the same information in absolute terms: the agent had discovered $46/58$ (i.e., about 79%) solutions at Depth 3 and $18/23$ (i.e., about 79%) solutions at Depth 5.

Overall, the results of Figure 6.3 show that *the system takes incrementally longer to find deeper solutions*. This is expected since deeper solutions require longer sequences of (correct) choices to be made, and learning these choices invariably requires more samples.

Our next experiment consisted of resolving the full set of saved Solve(?d,?p) events for solutions at all depths from one to five. As before, we plotted the average success of the top level goal (Figure 6.4(a)) and the number of unique solutions discovered (Figure 6.4(b)), against the number of episodes. The system discovers all 411 solutions in around $75,000$ episodes.

Observe that the system does not reach the performance of the hand-crafted JACK program and converges to about 90% success (Figure 6.4(a)) even though it successfully discovers all solutions (Figure 6.4(b)). This is because the decision tree representation does not guarantee that the training data will always be correctly classified.[1] This is specially true when the training data is "noisy" as is the case here: two executions of a

[1] We discussed this accuracy versus compactness trade-off on Page 2.2 and Page 3.1.



(a) Average Success

(b) Solutions Found

Figure 6.4: Performance of the system in terms of the average success and number of solutions found (y axis) against the number of episodes (x axis), for all solutions at depths one to five.

plan in the same game configuration may result in different outcomes due to different plan choices below it (in the recursive hierarchy).

Our experiments in the Tower of Hanoi domain were designed to highlight the use of our learning framework with event parameters and recursion: a common feature of many practical BDI implementations. Therefore, our focus has not been so much on improving the efficiency of the learning algorithm but instead on showing that learning is indeed possible in recursive programs. In this domain, learning times may be improved, for instance, using a relational state representation where variables capture a class of situations, rather than an absolute one that represents each situation uniquely using disc names. For instance, recall that plan SolveTopMove applies whenever the top disc on the source pin is smaller than the top disc on the destination pin. For our set of five discs this represents 15 unique configurations (states) where the top disc can be moved to the destination pin. Using two variables src and $dest$ to represent the source and destination discs, these 15 states could be combined into a single state that describes the case when src is smaller than $dest$.

In our implementation that uses disc names, the number of states with five discs is over 1 million. For our experiments we completely removed the existing context conditions and had to learn these from scratch. Normally one would expect to start with some

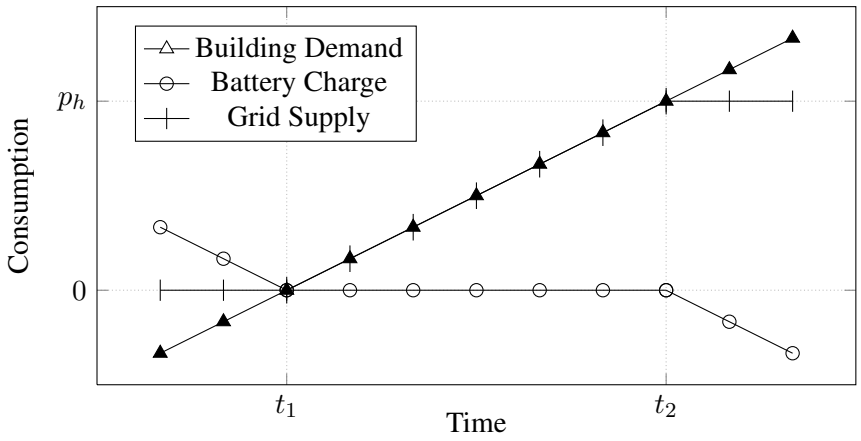


Figure 6.5: Use case scenario for a modular battery system.

initial necessary (but possibly insufficient) context conditions that would help reduce the state space for learning further. As we will show next, this is precisely what we do to make learning feasible in a domain with over 10 million states.

6.2 Modular Battery System Controller

Energy storage enables increasing levels of renewable energy in our electricity system, and the rapidly maturing supply chains for several battery technologies encourages electricity utilities, generators, and customers to consider using large battery systems.

Consider a controller for managing the overall energy demand of a smart office building comprising of a set of loads (e.g., appliances in the building), some renewable sources (e.g., solar panels on the roof and a local wind turbine), and a large modular battery system. The building is connected to the main grid, and economics govern that the grid power consumption of the building be maintained within the range $[0 : p_h]$ (Figure 6.5). However, in any given day, since there is little control over the demand in the building and certainly no control over the renewable generation, it is possible that the power consumption of the building will fall outside the desired range. For instance, if the renewable generation is high relative to the building loads, then net consumption may fall below 0 (e.g., period prior to t_1). Similarly, if demand is higher than generation then the net building consumption may rise above p_h (e.g., period after t_2). While the net building demand and generation is fixed for all practical purposes, we do have control over the use of the battery system (Battery Charge). Hence, by suitably ordering the

battery system to charge (i.e., act as a load) or discharge (i.e., act as a generator) at determined rates through this period we may influence the net demand in the building and consequently the energy drawn from the electricity grid (Grid Supply).

Large battery systems usually comprise of multiple modules and in many installations these may be controlled independently. Modules may be operated in synchrony but often there are strategic reasons to keep some modules in a different state to others. For example, if it is undesirable to change the direction of power flow between charging and discharging too frequently, a subset of modules may be used for each direction until it is necessary to change their roles. Also, some technologies have specific requirements, such as the zinc-bromine flow battery for which a complete discharge at regular intervals is desirable to "strip" the zinc plating and ensure irregularities never have an opportunity to accumulate. Where they exist, these requirements place further constraints on module control.

So, given a large battery installation, we are interested in a control mechanism to achieve a desired rate of charging or discharging, by suitably setting each module in the battery, such that the output rate is the sum over the modules' rates. While programmed control of such response is certainly possible, it is not ideal since battery performance is susceptible to change over time and may diverge from normal. For example, batteries tend to lose capacity over time, and this change will depend on their chemical properties and use. What is required is a means of adaptable control that accounts for such drift, and as such, a machine learning approach may be appropriate.

6.2.1 System Design

We design our adaptive BDI controller for the battery system in two stages as follows:

1. Build a system that caters to the initial requirements of the battery controller: This stage corresponds to the normal task of building any BDI program and includes designing the overall goal-plan hierarchy and any initial context conditions for the plans. The output of this step is the functional battery controller BDI program, albeit void of any learning capability. We call this the *basic* system.

2. Integrate the learning framework: In this stage we integrate our learning framework into the basic system. The result is that we have a two-step filter to decide a plan's applicability. The plan's initial context condition that we specified in

the basic design constitutes the first filter, and the plan's decision tree that captures ongoing outcomes makes up the second filter. We will call this the *adaptive* system.

Basic Design

The idea is that at the beginning of every period of deliberation the controller receives a request from the environment, and responds by operating the battery system for that period in a suitable operational state that resolves the goal. The accuracy of the system (i.e., how well the battery response matches the desired rate) depends on the *frequency* of the requests as well as the *resolution* of the battery system. For real-time matching of demand with supply, the frequency of system requests may be as high as one request every second. The resolution of the battery system decides how closely it can match the desired response and is affected by the number of modules m in the installation. For simplicity, we will assume homogeneous capacity c of the modules (but with possibly different chemical properties and constraints), such that the overall system capacity is $c \times m$. Each module in turn may be configured in one of three ways — charging (i.e $+c$), discharging (i.e., $-c$), or not in use (i.e., 0) — and the sum of the configured values over all modules gives the net response of the system. By appropriately setting each module's operational state then, the response of the battery system may be adjusted in steps of $\pm c$.

Figure 6.6 shows our basic design for the BDI controller for a battery system with m modules. Here SetRate(r, k, s) is the periodic request from the environment. The parameter $r \in [-1.0 : +1.0]$ is the desired charge or discharge rate (normalised) where -1.0 indicates a maximum discharge rate (where all modules are discharging) and $+1.0$ indicates a maximum charge rate (where all modules are charging). The parameter s represents the current state of the battery system derived from sensor readings, and k is initially set to the number of modules m in the system. Conceptually, the controller works by recursively configuring each module (i.e., $k > 0$) for the period in question using the plans SetCharge (charging at rate +c), SetDischarge (discharging at rate -c), and SetNotUsed (disconnected), and finally, after all modules have been configured (i.e., $k = 0$), physically operating the battery (i.e., all modules simultaneously) for one period using the Execute plan.

Of course, we would like to avoid running the battery in configurations that we know

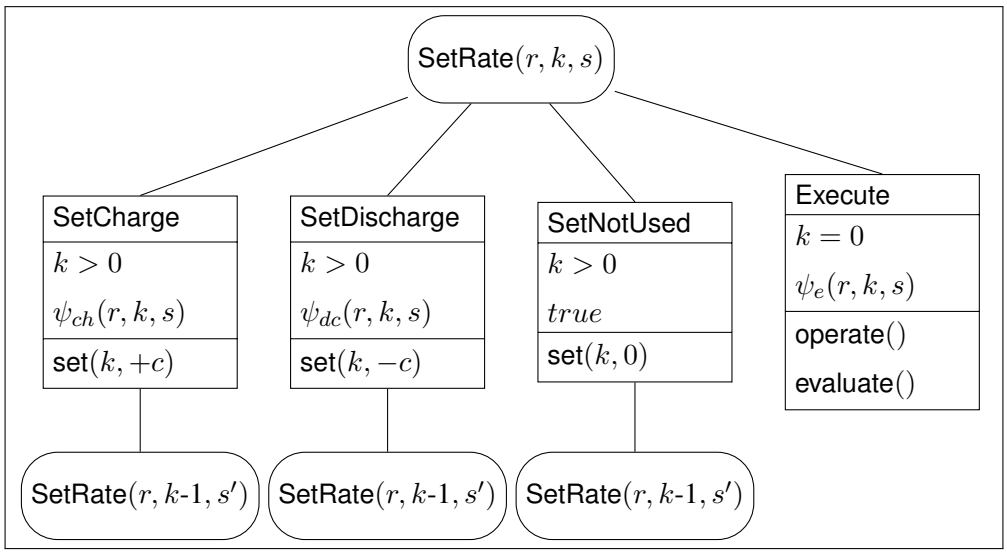


Figure 6.6: Goal-plan hierarchy for the battery system.

will not achieve the requested rate. We can do this by specifying initial context conditions $\psi_X(r, k, s)$ for the plans. The following rules for the plans of Figure 6.6 show the implemented context conditions:

SetRate(r, k, s) : CheckChargeConstraints(r, k, s) $\wedge$ (GetCharge(k) $<$ GetCapacity(k)) $\leftarrow$ SetCharge(r, k, s)

SetRate(r, k, s) : CheckDischargeConstraints$(r, k, s) \wedge$ (GetCharge$(k) > 0$) $\leftarrow$ SetDischarge(r, k, s)

SetRate(r, k, s) : $true \leftarrow$ SetNotUsed(r, k, s)

SetRate(r, k, s) : GetConfiguredRate$(s) == r \leftarrow$ Execute(r, k, s)

For instance, plan SetCharge may not be considered in a given instance if the module is only allowed to change charge directions once every four periods and charging in this period would violate this constraint (i.e., the $condition$CheckChargeConstraints(r, k, s) fails). Plan SetDischarge may be ruled out if the module is already discharged (i.e., the condition GetCharge$(k) > 0$ fails). Or, plan Execute may not execute because the chosen configurations imply that the response is bound to fall short of the request (i.e., condition GetConfiguredRate$(s) == r$ fails).

We use BDI failure recovery during this process of finding a configuration that fulfils known constraints. The idea is to try out other configuration possibilities if the process of recursively configuring the modules leads to a "dead end", i.e., where the Execute

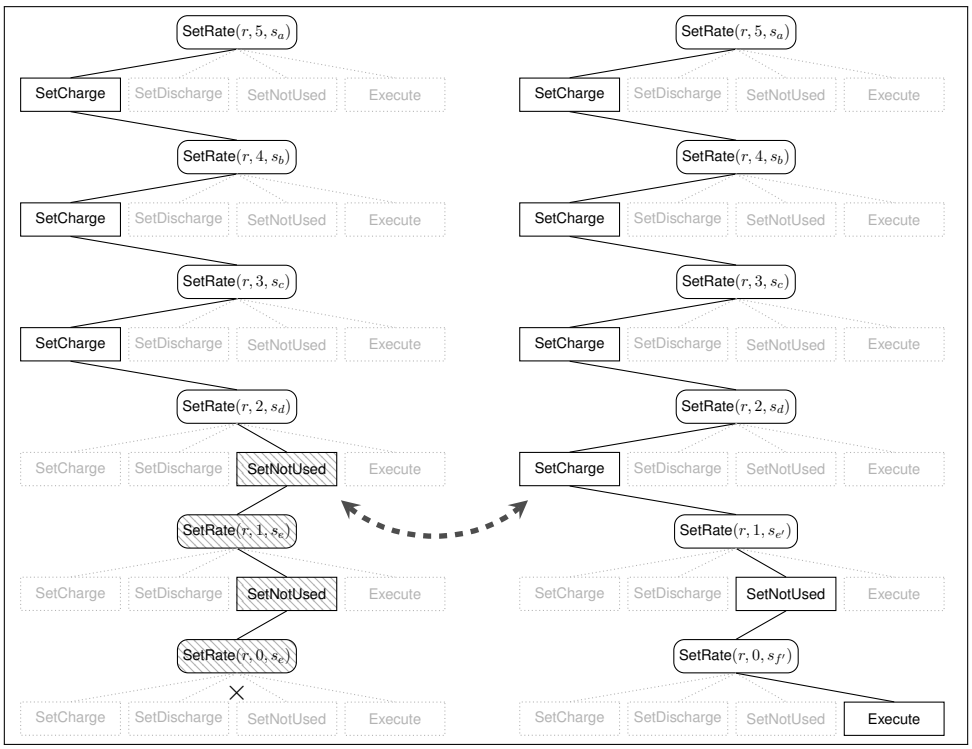


Figure 6.7: An example showing use of failure recovery in the battery controller.

plan does not apply. When this happens, we can backtrack up and select a different configuration path until all constraints are satisfied or all options exhausted.

For example, consider the case when a given request requires that four of the five modules be set to charge, and suppose that module configuration proceeds as follows: SetCharge for module 5, SetCharge for module 4, SetCharge for module 3, and SetNotUsed for module 2. At this point, the only way to satisfy the initial request is to set the final module 1 to charge, but suppose that plan SetCharge is not applicable because module 1 is already at maximum charge. Say plan SetDischarge is also not applicable because discharging the module will violate some internal constraint. Plan SetNotUsed will therefore be selected, giving the following selection trace:

$$\begin{aligned}\lambda_1 = \quad & \mathsf{SetRate}(r, 5, s_a)[\mathsf{SetCharge}] \cdot \quad \mathsf{SetRate}(r, 4, s_b)[\mathsf{SetCharge}] \cdot \\ & \mathsf{SetRate}(r, 3, s_c)[\mathsf{SetCharge}] \cdot \quad \mathsf{SetRate}(r, 2, s_d)[\mathsf{SetNotUsed}] \cdot \\ & \mathsf{SetRate}(r, 1, s_e)[\mathsf{SetNotUsed}].\end{aligned}$$

Subgoal $\mathsf{SetRate}(r, 0, s_f)$ will now be posted. However, the context condition of the

only applicable Execute plan will fail because when $\psi_e(r, 0, s_f)$ is evaluated it will return false as the final configuration in s_f (total charge rate) does not match the request. When it does, the failure recovery mechanism will look for alternatives. Since no alternatives exist for SetRate$(r, 0, s_f)$, the subgoal will fail. This failed trace λ is depicted in the left half of Figure 6.7 terminating in the $\times$ symbol. The search for alternatives will then proceed at the parent level, i.e., for subgoal SetRate$(r, 1, s_e)$, where again no other options apply. The process will continue until a satisfactory alternative configuration is eventually found, such as SetCharge for module 2 followed by SetNotUsed for module 1, and finally Execute whose context, this time around, is satisfied:

$$\begin{aligned}\lambda_1 = \quad & \mathsf{SetRate}(r, 5, s_a)[\mathsf{SetCharge}] \cdot \quad \mathsf{SetRate}(r, 4, s_b)[\mathsf{SetCharge}] \cdot \\ & \mathsf{SetRate}(r, 3, s_c)[\mathsf{SetCharge}] \cdot \quad \mathsf{SetRate}(r, 2, s_d)[\mathsf{SetCharge}] \cdot \\ & \mathsf{SetRate}(r, 1, s_{e'})[\mathsf{SetNotUsed}] \cdot \quad \mathsf{SetRate}(r, 0, s_{f'})[\mathsf{Execute}].\end{aligned}$$

In Figure 6.7, the right half shows the trace λ_1 found using failure recovery. The shaded boxes in the left trace show the extent of backtracking before an alternative path is found as highlighted by the arrow.

Programming for Adaptability

The basic controller described so far will work correctly for the initial specification of the system if deployed. However, if the physical battery properties were to change over time, then the system will inevitably perform sub-optimally. As an example, consider a time in the future where the capacity of a module has deteriorated, so in effect it holds less charge than it did initially. Here, it is easy to see that some solutions that worked initially will no longer work. This is because the controller will not know what the new capacity is, and will try charging the module in some situations only to find that the net battery response no longer matches the expected result. What we would like is to program the controller with adaptability in mind in order to rectify for such foreseeable changes.

Our strategy for encoding such adaptability into the basic design is using the BDI learning framework described in Chapters 3 and 4. The idea is to evaluate how well the configurations selected according to the programmed context conditions actually work

when the battery is operated. If the environment is behaving as expected by the programmed basic system then the configurations should work correctly, however if the environment dynamics has changed, then some configurations may not work optimally anymore and we would like to learn to isolate these over time.

The process involves adding a decision tree to each plan for capturing the changing applicability conditions, and providing the probabilistic confidence-based plan selection mechanism. Next, we add a feedback step to the initial design (step evaluate in the Execute plan) that evaluates the actual battery response against the original request. The battery response is deemed successful only if it was within tolerance of the desired response, otherwise it is deemed failed. This way, every time the Execute plan finishes, the *evaluated* pass/fail result is recorded against all active plans that led to that invocation. By training over the outcomes of plan selections in each situation, the system learns, over time, correct plan selection for all possible system requests. This is our *adaptive* system.

The net result is that we have a two-step filter to decide a plan's applicability. The programmed context conditions of the basic system make up the first filter, and the learning that captures ongoing performance makes up the second filter.

Useful learning takes place in the adaptive system even while the system is performing correctly to initial specification, i.e., never experiences a failure. This is because "internal" failures during deliberation, when bad configuration paths are abandoned for alternatives using failure recovery (such as in the example on page 94 requiring four modules to be charged), provide the necessary negative samples to build a useful classifier. The benefit is that the agent is continually collecting samples and building an incrementally better classifier for the state space experienced so far. Then, when the environment does change, it does not have to start learning from scratch as it already has a substantial amount of data to assist it in recovering from the change.

Design Trade-Offs

Overall, the system learns a response to the immediate request. It does not learn any temporal relationship in the sequence of system requests. For instance, the request sequence may have some diurnal or seasonal pattern, however the proposed system does not attempt to learn this. This is acceptable since the time-scale for decision making

(normally in the order of seconds) is very short compared to the frequency of any potential pattern.

One important implication of this setup is that there are limitations to what can be learnt. If the initial context conditions eliminate some world states that are potential solution states in the changed environment, then there is no way to "include" these again since the programmed filter code cannot be modified at runtime. As a result, the programmer has some responsibility to ensure that the initial context conditions cater for all initial and foreseeable future solution states. Overall, this is a trade-off between specificity and adaptability: the more specific the initial conditions, the better they cater to initial specifications, but the more limited the adaptability for future changes.

We should point out here that the basic design of Figure 6.6 is only one way of specifying the controller. Another perhaps more intuitive design, may consist of a high level plan with six sequential subgoals: the first five for configuring each of the five modules, and the sixth subgoal to operate the battery. Such a design, however, would not work well for learning purposes. The reason is that there would be no way for the SetCharge, SetDischarge, and SetNotUsed plans to know how their "local" actions impact the "global" result, i.e., whether plan Execute was subsequently executed. For them success would simply mean satisfying the local constraints of the module. If that did not lead to a battery configuration that is usable (according to the context conditions of the Execute plan), then there would be no way to pass this information back to them as they would have already succeeded (i.e., finished executing). In contrast, in our design the global result can be passed back to the contributing plans as they are all active in the recursive chain. This issue relates to the limitation (discussed in Chapter 8) that our learning framework cannot account for inter-dependence between subgoals of a higher-level plan, and highlights the importance of understanding how the learning works when developing a suitable solution often involving recursion.

6.2.2 Experimental Setup

We conduct experiments for a battery system with *five* modules, i.e., $k = 5$. In this system, the charge state of each module is described by a discrete value in the range $[0 : 3]$ where zero indicates a fully discharged state and three indicates a fully charged state. As well as this, each module has an assigned configuration for the period from the set $\{+c, 0, -c\}$ where $c = 1/k$. The operational model is simple: charging is meant

to add $+c$ while discharging is meant to add $-c$ to a module's charge state, otherwise the state is unchanged for the period (i.e., there is no charge loss).

The desired response is in the range $[-1.0 : +1.0]$ in discrete intervals of $\pm c$ giving a total of 11 possible requests. The complete state space for the problem is described by the number of modules (5), the possible requests (11), the charge state of the system (4^5), and the assigned configuration of the system (3^5), that is, $5 \times 11 \times 4^5 \times 3^5 \approx 13.7$ million states. Though significant, the agent does not have to learn over this entire space, because the filtering of nonsensical configurations by the plans' context conditions $\psi_X(r, k, s)$ will reduce it substantially (to ≈ 1.5 million).

At the beginning of a learning episode the configuration of each module is reset to 0, i.e., not in use. The charge state of each module, however, is left untouched and carries over from the previous episode as would be the case in the deployed system. This has implications for learning, particularly that the state space is *not* sampled uniformly. Each episode corresponds to one SetRate$(r, 5, s)$ request from the environment. For simplicity of analysis, the environment only generates satisfiable requests given the state of the battery, such that a solution always exists for the generated request.[2] The outcome of each episode is a single invocation of the Execute plan that operates the battery and evaluates the response. The tolerance level is set to 0.0 so that the battery response is deemed successful only when the sum of the module configurations matches the request exactly.

In normal BDI operation, only plans that are applicable, as determined by their context condition, are considered for execution. For our learning framework, where applicability is additionally defined by a plan's decision tree, this means that only plans with a reasonable likelihood of success should be allowed. To represent this, we use an *applicability threshold* for plan selection (of 40%), meaning that plans with a likelihood of success below this value are removed from consideration.[3] While this feature does not alter the overall learning performance of the battery controller (we ran our experiments with and without the applicability threshold and found no significant difference

[2] If unsatisfiable requests are also generated then calculation of the optimal performance becomes harder. The implication of this choice is that the agent does not spent time in learning which requests never succeed. Therefore the learning is likely quicker. Our focus however, is not on learning efficiency but on whether the controller program learns correctly.

[3] The threshold value used in our experiments was selected after trials with different values. Generally, this is a domain dependent setting. We discussed these considerations in Chapter 3 Section 3.3.

in performance), it does preclude the battery system from being operated (i.e., plan Execute being called) with module configurations that are likely to be unsuccessful. Conceptually, the cost of operating the battery when the chances of success are poor, is considered to be higher than not operating the battery at all. In fact, we found that the threshold used reduces the number of battery operations by 12%, which is substantial when considering battery life.

For all of our experiments, the threshold parameter for stability calculation (Chapter 3) is set to 0.5.[4] We use an averaging window of $n = 5$ for both the stability-based metric $\mathcal{C}_s(\cdot, \cdot, n)$ and the world-based metric $\mathcal{C}_d(\cdot, n)$, and a (balanced) weighting of $\alpha = 0.5$ for the final confidence measure $\mathcal{C}(\cdot, \cdot, \cdot)$.[5] Finally, each experiment is run five times and the reported results are averages from these runs.

6.2.3 Results

We now describe three types of experiments to show the adaptability of the implemented system to various environmental changes. In the first we describe the case where the system starts to fail due to changes in module capacities, and show how it adapts and recovers performance following this change. In the second experiment, we show how the system adapts to a series of partial failures in the system where (different) individual modules fail and are thereafter restored. Lastly, we show how the system responds to a complete failure of all modules, and how it recovers when they are subsequently restored.

Experiment: Capacity Deterioration

In this experiment we model the situation where module capacities deteriorate over time. In a real system this will happen gradually over several years of typical use. However, to show the response to substantial change, we force this deterioration to occur instantaneously in this experiment. Figure 6.8 shows the results for this case. In the beginning of the experiment, the system performs correctly as programmed, and

[4] Higher values mean that plans become stable quicker. A value of 0.5 generally works well for non-deterministic domains.

[5] The impact of these parameters is discussed in Chapter 4. Performance is not very sensitive to α and a value of 0.5 works well in most situations. The parameter n decides the sensitivity of the confidence measure: lower values make the confidence measure more sensitive to performance changes. Values of n in the range $[5 \ldots 10]$ work well in most situations.

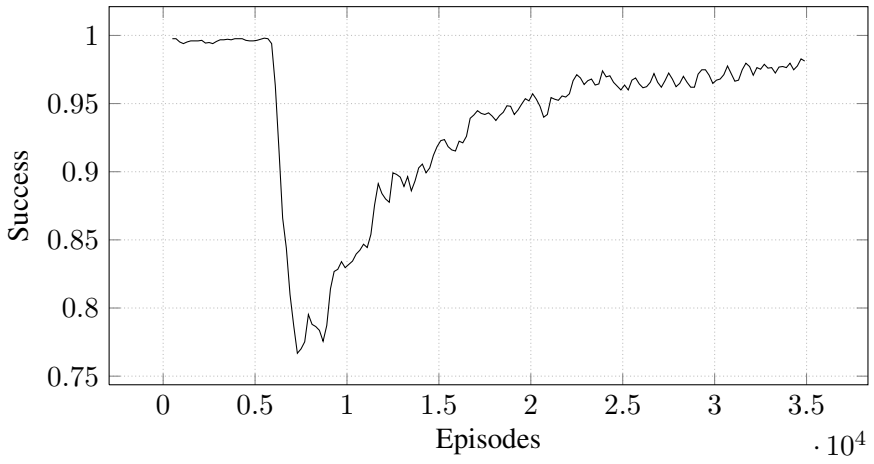


Figure 6.8: Controller performance around battery capacity deterioration.

goes about recording its experiences although there is no evident use of the resulting learning yet. After some time (about 5k episodes), the capacity of all five modules drops instantaneously, from the initial range $[0 : 3]$ to range $[0 : 2]$. These capacity changes result in a rapid drop in performance corresponding to the set of programmed/learnt solutions that no longer work. The basic programmed system would, at this point, converge to around 76% performance, i.e., on average the basic controller would satisfy 76% of all requests. The adaptive controller, however, rectifies the situation by learning to avoid the module configurations that no longer work, and preferring alternatives that do. The reason why the system is able to recover is because more often than not there are several possible ways of configuring the modules for the effective output rate: the key is to learn which of these possibilities actually work.

In Figure 6.8, the system performance tapers at around 95% and never quite reaches optimal (we confirmed this by running the experiment to 50k episodes). The reason is that the change in the environment dynamics causes a significant increase in "noisy" training data, as plans start failing in situations where they would have previously succeeded, or start succeeding in situations where they would have failed earlier. This conflicting data impacts the classification accuracy of the resulting decision trees and leads to incorrect plan selection in around 5% of the cases.

The underlying issue is that the training data is partially "outdated." To confirm if this is the case, we implemented a simple filter on the training set such that only the most recent 5k worlds in which a plan was invoked were used to build the decision tree. We

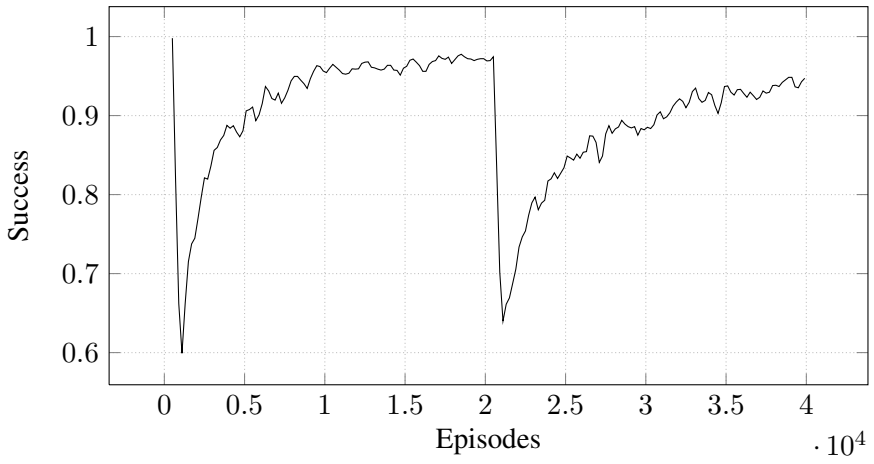


Figure 6.9: Controller performance with different module failures over time.

found that in this case the recovery of the system was indeed significantly better and the system converged to around 98% performance in around 25k episodes. We note, however, that such a filter is at best arbitrary and cannot be generally applied in any domain. This is indeed an open issue that requires further investigation, and one we hope to address in future work.

Experiment: Partial System Failure with Restoration

In our next scenario, we model a series of module malfunctions and their subsequent restoration, to see how ongoing changes impact controller performance. During all such changes, the battery system always remains capable of successfully responding to the request using alternative configurations. In the experiment, the first battery module fails for the duration $[0 : 20\text{k}]$ after which it is reinstated, the second module fails for the period $[20\text{k} : 40\text{k}]$, and so on.

Figure 6.9 shows the system performance for this setting. At the beginning of each change, namely, at 0k and at 20k, the performance drops dramatically, as the expected solutions that utilise the failed module no longer work. Following each module failure, the system learns to operate the battery without it, by always configuring the failed module to not in use (i.e., state 0). By the time each failed module is restored (e.g., episode 20k for the first module), the system has already learnt to operate without it, and hence, will not try to re-use it unless really required.

The difference in the recovery to around 97% at 20k episodes and around 94% at 40%

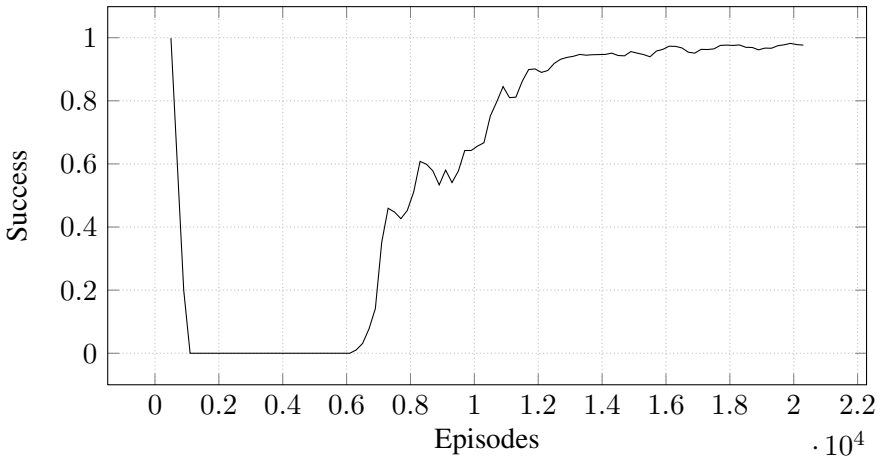


Figure 6.10: Controller performance when initial learning is superseded.

episodes is indicative of the issue of accumulating conflicting data with each change. We were able to resolve this issue by filtering out old samples as we did in the first experiment, however again this is an arbitrary solution to the problem.

The apparent difference between the performance drop at 0k to around 60% and at 20k to around 65% is not meaningful in any way. It just happens that more "bad" cases occurred in the first failure. The theoretical drop in performance is to 65%.

Experiment: Complete System Failure with Restoration

In this experiment, we model the extreme scenario of complete failure of the system for some time followed by full restoration. Technically, *all* module configurations would fail for the period $[0 : 5\text{k}]$, after which they are restored to normal operation. The results are shown in Figure 6.10. At the beginning of the experiment, overall performance drops to zero rapidly, as none of the programmed configurations are successful in responding to the request. After a while (at around 2k episodes), the estimated likelihood of success of all plans drops below the applicability threshold of 40%. At this point, the battery operation comes to a complete halt: no plans are ever applicable and plan Execute is never invoked. Then, at episode 5k, the failed modules are repaired so that the battery is restored to normal operation. However, observe that since no plans are ever tried due to the applicability threshold, then new learning may not occur. To address this issue, we use a "soft" applicability threshold mechanism: the 40% applicability threshold mechanism applies 90% of the time. This allows the battery to operate with some likelihood and the agent system to eventually start recovering at around 6k

episodes.[6]

We note that, in this case, we have assumed that an explicit signal indicating that some important changes have occurred (e.g., some batteries have been replaced/repaired) is *not* available. Were such an explicit notification of changes available, it could be used as input for the controller, such as to re-start battery operation once repairs are completed. Such a signal would also be useful for constructing a more general-purpose mechanism for filtering outdated training data, however this investigation is beyond the scope of this thesis.

It is worth highlighting that following a drop in performance from changes in the environment dynamics, the controller agent always recovers to above 90% accuracy within 10k iterations of the change in all experiments. For a battery being configured once every second, this equates to just under three hours, which may be a reasonable timeframe for recovery in a practical controller. Moreover, while performance does deteriorate following a change in the environment, it does not always drop off completely, as illustrated in the first two experiments. Overall, the agent is able to recover without having to re-learn from scratch which is likely to be more time consuming.

In summary, the three types of scenarios described here for the energy storage domain empirically demonstrate the ability of a learning BDI agent to adapt to changes in the dynamics of the environment using the framework of Chapter 3 and the dynamic confidence measure developed in Chapter 4.

[6] The battery will be actually operated after five module configuration plans $\mathsf{Set}^{\star}$ have been selected and carried out (one per existing module). In the best case, only one of these plans has failed the threshold and hence there is a 10% chance that the battery will operate. On the other hand, if all plans are failing the applicability threshold, then there is only a 0.1^5 (i.e., 0.001%) chance that the battery will operate.

Chapter 7

Related Areas

In Chapter 2 we described the BDI model of agency and discussed related works where learning has been integrated with deliberation in BDI systems. In this chapter we compare our approach to work in two separate areas of research: hierarchical task network (HTN) planning and hierarchical reinforcement learning. In Section 7.1 we present an overview of HTN planning and how it relates to BDI programming. Particularly we are interested in the fact that the two concepts map quite well to each other, and that plans' context conditions in BDI systems are synonymous with methods' preconditions in the HTN case. We summarise efforts in HTN planning research that are concerned with learning preconditions and compare them to our own approach for learning context conditions. Section 7.2 introduces the concepts in hierarchical reinforcement learning and discusses the key related works in this area.

7.1 Learning in Hierarchical Task Networks (HTN)

Background

Planning constitutes "look-ahead" or hypothetical reasoning to decide on a course of action to achieve some desired outcomes [Russell and Norvig, 2009]. The process involves choosing and organising an agent's actions by anticipating their expected outcomes. For instance, an agent planning a holiday may evaluate several different destinations and modes of transportation to construct a travel itinerary that best satisfies its vacation preferences and budget constraints. In general, a *planner* takes as input a

planning problem (i.e., a *state-transition system* that is a formal "model" of the real environment for which a plan is to be constructed), an initial situation (i.e., a description of initial states in the system), and some objective (i.e., goal states to achieve or avoid, a measure or "utility" to optimise, or a set of tasks to perform), and outputs a plan (i.e., sequence of actions) that solves the problem.

Planning can broadly be categorised into two areas. *Domain independent* planning refers to the case where only basic (primitive) actions are given and the planner must decide how to put these together. Here, the problem of planning is defined as one of finding a sequence of actions corresponding to a sequence of state transitions that when applied to the initial state will result in a goal state. Classical planning is the most popular form of domain independent planning, and dates back to the work of Fikes and Nilsson [1971] on the STRIPS automated planner. The majority of research in planning falls under the banner of classical planning. Even under the assumption that the system is deterministic, static, finite, and fully observable, classical planning is a hard problem and has PSPACE-complete complexity [Bylander, 1994; Erol et al., 1995]. *Domain dependent planning* uses a similar notion of a planner but with an additional input — that of domain know-how, that is, knowledge of how primitive actions may be put together. One popular way of providing know-how is by specifying what actions apply in which situations and how the available actions may be ordered in various stages of the planning process. The result is that the search is pruned and hence plans may be found faster than with classical planning. The technique has been successfully applied to numerous practical applications, predominantly in the area of hierarchical task network (or HTN) planning [Erol et al., 1994; Nau et al., 2005].

The idea behind HTN planning is to use an expert-provided *task hierarchy* that captures domain know-how to guide the planning process. HTN planning works by recursively decomposing high level tasks into networks of lower level tasks that eventually reduce to primitive actions that interact with the environment. This provides an intuitive way of breaking down a planning problem by abstracting out the details to different levels in the hierarchy. There many be many ways of achieving a task, and HTN *methods* capture the "standard operating procedures" that specify different ways of decomposing it. HTN planners in general are much faster solvers than classical planners because the task hierarchy serves to significantly reduce the number of options that need to be considered. They are heavily used in industrial applications where domain knowledge about the structure of and relationship between tasks is available [Nau et al., 2005].

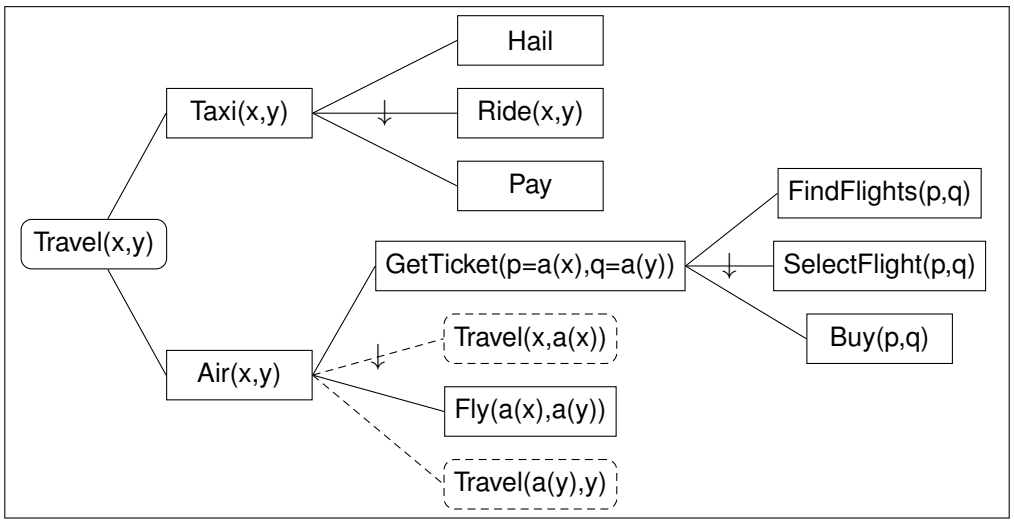


Figure 7.1: HTN methods for an example travel-planning domain [Nau, 2007].

Consider a travel-planning problem [Nau, 2007] (Figure 7.1) where the task is to plan a journey from location x to destination y (i.e., Travel(x,y)) for which two methods are provided: a taxi commute (i.e., Taxi(x,y)) for short distances, and air travel (i.e., Air(x,y)) for longer distances. Taxi travel involves the sub-tasks of hailing the taxi, riding to the destination, and paying the fare; while air travel involves purchasing a plane ticket, travelling to the local airport, flying to the airport closest to the destination, and then travelling to the destination (↓ means that tasks are performed from top to bottom.)

Say we wish to plan a trip from London to Greenwich. Since only the Taxi method applies for this short distance, it is decomposed further resulting in a final plan that entails hailing a taxi, riding it to Greenwich, and paying the fare. In contrast, for a longer trip from London to New York, the Air method would apply. Planning would proceed by first expanding the GetTicket method for finding an appropriate flight from LHR Heathrow International Airport to JFK International Airport and buying the ticket. Next, the planner would solve for travelling from the location in London to LHR (invariably via taxi), flying from LHR to JFK, and finally travelling from JFK to the destination in New York (again via taxi). The end result is a plan with the following ordering of primitive tasks: FindFlights(LHR,JFK) SelectFlight(LHR,JFK) Buy(LHR,JFK) Hail Ride(London,LHR) Pay Fly(LHR,JFK) Hail Ride(JFK,New York) Pay.

HTN planners share several similarities with BDI systems. First, the HTN task hierarchy and the BDI goal-plan hierarchy serve a similar purpose: they capture the "procedural" domain knowledge by specifying different ways in which tasks may be achieved. In that sense, *tasks* in the HTN terminology are synonymous with event-goals in the BDI framework. Second, a HTN *method* specifies a way of achieving a task by decomposing it into, and then solving, a task network. This is similar to the way a BDI plan procedure specifies how to resolve a given event-goal. Moreover, the treatment of HTN methods and BDI plans is similar: the applicability of methods (or plans) is determined at runtime by evaluating if their *preconditions* (context conditions) hold in the given situation. Finally, both schemes provide a "backtracking" mechanism to recover from unsuccessful decision paths such as when no suitable option can be found. The key difference, however, is that HTN planners look for complete solutions for achieving a task before committing to any actions, while BDI systems "act as they go" and execute actions (plans) at each step. A detailed comparison between HTN planning and BDI systems is provided by [de Silva and Padgham, 2005; de Silva et al., 2009; Sardina et al., 2006].

7.1.1 CaMeL

There are several examples of learning HTN method preconditions in the literature. These are directly related to our work due to the many similarities that HTN planners share with BDI systems. One such system for learning HTN preconditions is CaMeL [Ilghami et al., 2002] that uses the notion of *candidate elimination* in a version space. The idea behind the approach is to start with a maximum possible set of explanations of the concept being learnt (i.e., method preconditions) and incrementally eliminate the candidates (i.e., possible preconditions for the method) as more information (i.e., training samples) is collected. Given sufficient consistent samples over time, the version space converges to a single answer. Training samples in CaMeL consist of plan traces that are similar in principle to the execution traces we use in our framework (see Chapter 3). Note, however, that in our case the negative training samples come from failed executions, which is not an option in HTN planning where a complete decomposition is performed prior to any action being taken. To overcome this, CaMeL uses a deductive method to construct negative samples. Since plan traces list all applicable methods that decompose a task in a given world state, then if other methods are known to also decompose this task (for some different world states), it may be inferred that those other

methods were not applicable in the first instance and hence may be used to construct negative samples. CaMeL is a sound and complete learner and under certain assumptions will also converge to a single explanation of the concept in a finite number of training traces.

An important difference between our framework and CaMeL is that in our case the learning is performed online in a trial-and-error manner, whereas in CaMeL the training samples are generated, and learning performed, offline. Moreover, CaMeL assumes a deterministic domain and requires training samples to be free of "noise", while our framework enforces neither of those constraints. Another difference is that CaMeL assumes some knowledge of the form of the preconditions (such as whether only conjunctions are allowed, or if disjunctions are permitted too) in order to guarantee that *accurate* preconditions can be learnt. In our framework, no knowledge of the structure of the context condition is assumed and the learnt conditions are not precise formulae but rather decision trees.

CaMeL++ [Ilghami et al., 2005] is an extension of CaMeL to allow planning to proceed while the preconditions are still being learnt, i.e., before the version space has fully converged to a single precondition. This allows reasonable plans to be derived with significantly less training samples, but also raises the issue of determining confidence in the given learning to decide when a method may be considered applicable enough: issues that we have discussed in detail in Chapters 3 and 4. In order to gauge confidence in the learning, CaMeL++ uses a voting scheme: each member of the version space is allowed to accept or reject the world state in question, and if the sum of acceptances is more than the *acceptance threshold*—a similar notion to our applicability threshold from Chapter 3—then the method is considered applicable in a given situation.

7.1.2 SiN

SiN [Muñoz-Avila et al., 2001] is another system concerned with improving method selection in HTN planning. It uses case-based reasoning where a case is similar to a HTN method instance but augmented with a set of preferences represented as question-answer pairs. The idea is to use direct feedback obtained from the domain expert (using conversational questions) to select between applicable methods where this information cannot be automatically extracted. Planning proceeds using automated decomposition where possible but switches to case-based (conversational) retrieval when this is no

longer possible. A list of all cases that apply in a given situation (ranked using existing conversations) is then provided to the user who must select one case in order for planning to proceed again. SiN was designed as an *interactive system* with military operations in mind, where complete domain knowledge is generally not available for automated planning and therefore expert opinion is valuable for guiding the planning process. SiN aims to improve method decomposition over time and in this sense shares our goal, even though it does not use feedback to directly learn method preconditions like we do (it uses the input only to produce a ranking to aid user selection). The approach could nevertheless be adapted also for improving plan selection in human-in-the-loop BDI systems where user feedback is invaluable.

7.1.3 DInCAD

DInCAD [Xu and Muñoz-Avila, 2005] extends the SiN idea to remove the dependence on a domain theory, i.e., the HTN methods for generating the plans: instead only cases consistent with the methods (similar to SiN) are assumed to be available, along with a type ontology that expresses relationships among variables and types. Under these assumptions, and given sufficient cases, the case-bases may be used as a direct substitute for methods during planning. The motivation is that often such hierarchical cases can be automatically extracted for the domain, such as from work-breakdown structures used in project planning. Cases are stored in a generalised form using variables along with any binding preferences based on the actual case instances. Where several cases apply in a given situation, a ranking is produced based on the number of preferences that are satisfied for each candidate, and the highest ranking case is selected for use in task decomposition. DInCAD effectively learns method preconditions given the hierarchical relationship between tasks and the action models. As with the previous systems, DInCAD assumes a deterministic domain and performs learning offline, and differs from our system in that regard.

7.1.4 Icarus and HTN-MAKER

Other systems learn method preconditions as part of the task of learning the hierarchical task network itself. For instance, Icarus [Nejati et al., 2006] uses a form of explanation based learning to find the task hierarchy in *teleo-reactive logic programs* [Langley and Choi, 2006]: a special class of HTNs in which non-primitive tasks always map onto

declarative goals and in which top-level goals and the preconditions of primitive methods are always single literals. Teleo-reactive programs comprise of two databases: a *hierarchical concept database* that describes the state of the world at different levels of abstraction, and a *skills database* that contains the primitive and hierarchical skills that are available to the agent. Given a goal instance, the initial state, a trace of primitive skills that achieve the goal, and the action model (i.e the preconditions and effects of primitive skills), Icarus aims to learn high-level skills by repeatedly reasoning backwards from the final primitive skill in the trace to explain the achieved goal. If the primitive skill contains the goal as one of its effects, the algorithm explains the goal using *skill chaining* by tagging the precondition of this task as its new goal and reasoning about the previous solution steps with respect to it; otherwise it aims to explain the result using higher-level concepts (*concept chaining*) by reasoning over their subconcepts and effects. A key difference is that Icarus is free to construct the abstract tasks as it sees fit, i.e., it learns *any* hierarchy that satisfies the observed trace, whereas in our case the abstract task hierarchy is given by the BDI programmer.

Where Icarus learns full HTN methods for achieving a classical goal, HTN-MAKER [Hogg et al., 2008] is a sound and complete planner for the class of *classically-partitionable* planning problems. Formally, this means that for a planning problem (s_0, g, O), where s_0 is the initial state, g is the goals, and O is the set of planning operators, there exists a partition $(g_0, g_1, \ldots, g_k)$ of the conditions in g such that each planning problem in the sequence $(s_0, g_0, O), (s_1, g_1, O), \ldots, (s_k, g_k, O)$ is a classical planning problem and the condition g_i holds in the state s_{i+1} for $i = 0, \ldots, k-1$. As with Icarus, HTN-MAKER takes as input a set of operators, the initial states, and the action sequence that achieves the goals for the classical planning problem. However unlike Icarus, it does not require the classical goal g as input. Instead it uses an *equivalent annotated task* $t = (n, 0, g)$ for some HTN task n with no preconditions and the goal g as its effect. The benefit is that HTN-MAKER is then able to solve the classically-partitionable problem without having to solve the individual partitions separately like Icarus. HTN-MAKER$^{\text{ND}}$ [Hogg et al., 2009] extends the initial algorithm to include non-deterministic domains, by accounting for all possible outcomes of a primitive action in the trace. The extended algorithm is also sound and complete, and learns HTNs in low-order polynomial times with respect to the number of input plan traces and the maximum length of those traces.

Systems like Icarus, HTN-MAKER, and HTN-MAKER$^{\text{ND}}$ are fundamentally different

from our approach even though they effectively learn method preconditions as part of the task of learning the HTN methods. Firstly, they assume that solutions to the problem exist for use in learning. In our case, the plan traces are generated during exploration that is integrated with the learning process. As such, not only do our traces capture limited information, but they are also rarely positive examples of success at the start. Second, we do not have flexibility in choosing the structure so our hypothesised context conditions must fit the given hierarchy; whereas in the former the task is to find a hierarchy and related preconditions that together are consistent. Finally, our learning framework does not require an action model, handles noisy and incomplete training data, and works under conditions of partial observability of the world, unlike the mentioned approaches (bar HTN-MAKERND that handles non-deterministic domains).

7.1.5 HTN-learner

In the HTN domain, as far as we are aware, the only system that also supports learning with partial observability and does not assume an action model is HTN-learner [Zhuo et al., 2009], which learns method preconditions along with the action model. HTN-learner takes as input observed task decomposition trees whose leaves are all primitive actions and uses this to build different kinds of constraints: for instance, a predicate that frequently appears after an action is executed is likely to be an effect of the action; a sub-task often provides some post-conditions that make the next sub-task applicable; and actions cannot add existing facts or delete ones that do not exist. The algorithm then uses a weighted MAX-SAT solver to solve these constraints and converts the result back to methods preconditions and action models. This approach to finding preconditions by solving constraints of course is significantly different to ours, and assumes availability of suitable plan traces upfront.

In summary, the underlying difference between HTN planning and learning in BDI systems relates to the availability of training data. Whereas in the HTN case the training data is assumed to be available *offline*, in our case it is not, and learning and acting are tightly interleaved in an *online* manner. In fact, one of our main concerns is to define a fitting exploration policy that improves the quality of traces obtained, whereas in the former the domain expert must provide a suitable subset of decomposition trees to make learning effective. In the next section we explore the related area of hierarchical reinforcement learning that also uses trial-and-error learning, and while not as closely

related to the BDI architecture as HTN systems, nonetheless shares similar concerns in hierarchical learning as we do.

7.2 Hierarchical Reinforcement Learning

A natural way to simplify a challenging task is to break it into smaller sub-tasks at different levels of abstraction and to solve these independently. The benefit is that at the level of each sub-task one does not have to consider those details of the larger problem that are irrelevant to that sub-task. This is the intuition behind *hierarchical reinforcement learning* [Barto and Mahadevan, 2003]. While this only makes sense for problems that lend themselves to such decomposition, nevertheless, like HTN planning does for classical planning, hierarchical reinforcement learning often provides significant performance improvement over "flat" reinforcement learning at the expense of slightly sub-optimal performance. In this section we outline three important approaches to hierarchical reinforcement learning and how they relate to our work in this thesis: the options framework [Sutton et al., 1999], hierarchical abstract machines (HAMs) [Parr, 1998], and value function decomposition with MAXQ [Dietterich, 2000]. First though, we cover some background knowledge of markov decision processes, dynamic programming, and reinforcement learning, that forms the foundation for this work.

Background

Markov Decision Processes (MDPs) provide a formal framework for modelling environments where outcomes are only partly attributed to decision making by the agent, and are partly stochastic. Most research in dynamic programming [Bellman and Dreyfus, 1962; Howard, 1960], and lately reinforcement learning [Sutton and Barto, 1998], is concerned with solving optimisation problems using MDPs. Precisely, an MDP is a *sequential discrete time stochastic control* process. At any given time step, the process is known to be in a given state s that represents the state of affairs in the environment. At such a time step, the agent chooses an action a from a set of admissible actions in state s, and the process moves to a new state s' and returns a reward $R(s, a)$ to the agent. The probability that the process advances to s' depends in part on the action a that was taken in state s, and is given by the state transition function $P(s'|s, a)$. Importantly, the transition to s' depends only on s and a, and is independent of all previous states and

actions – this is known as the *Markov Property*.

Dynamic programming [Bellman, 1957] is a theory for optimally solving multistage decision problems given a perfect model of the environment as an MDP. The idea is to describe the *value* of a decision problem at a given time step in terms of the payoffs received from choices made so far, and the value of the remaining problem that results from those initial choices. The best achievable value at any step depends on the current state s and is given by the *value function*. Equation 7.2.1 shows the Bellman equation for an optimal value function. Bellman's important contribution was to show that the optimisation problem could be written by relating the value function $V^*(s)$ in one time step to the value function in the next time step $V^*(s')$. Here, $\gamma \in [0, 1)$ is the discount factor that captures the increasing uncertainty about future rewards, thus helping to limit the infinite horizon sum.

$$V^*(s) = R(s, a) + \max_{a \in A_s} \gamma \sum_{s'} P(s'|s, a) V^*(s'). \tag{7.2.1}$$

Then if a sequence of decisions is a *policy*, and an *optimal policy* is one that is most beneficial given some criterion, the theory of dynamic programming may be described as prescribing optimal policies for appropriating decisions at each time step in terms of the current state of the system.

The reinforcement learning [Sutton and Barto, 1998] setting is similar to that of dynamic programming except that the state transition function $P(s'|s, a)$ and the reward function $R(s, a)$ in Equation 7.2.1 are unknown. Here the agent has no choice but to physically act in the environment to observe the reward, and use the samples over time to build estimates of the expected return in each state. The goal then is to obtain an approximation of the optimal policy, and the key concern is the number of actions required to converge to this policy. This is characteristic of the *online* learning problem where acting and learning are interleaved and the agent must strike a balance between the *exploration* of new choices in the hope of finding better solutions, and the *exploitation* of current knowledge to its maximum advantage.

Formally, the reinforcement learning scenario relates to MDPs as follows: At each time step t in a multistep problem, the agent perceives the current state $s \in S$ of the environment and has at its disposal a set of possible actions A_s. It then chooses an action $a \in A_s$ that causes the environment to transition to state s' at time step $t + 1$ and

return a reward with the expected value $R(s, a)$. The agent's behaviour is described by a policy π that determines how the agent chooses an action in each state. The policy may be *deterministic* so that it specifies exactly what action to take in which state, i.e., $\pi : S \rightarrow A_s$, or it may be *stochastic* so that it gives the probability of taking an action in a given state, i.e., $\pi : S \times A_s \rightarrow [0, 1]$. The reward captures the immediate impact of taking the action a in state s, however says nothing about the long-term impact of that action on the achievement of the goal. Since the reward function $R(s, a)$ is unknown, the agent instead tries to maximise some cumulative function of the immediate rewards, typically the *expected discounted return* $R^{\pi}(s)$ at each time step t, as described by:

$$R^{\pi}(s) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots\}. \tag{7.2.2}$$

The quantity $R^{\pi}(s)$ captures the infinite-horizon discounted (by γ) sum of the rewards that the agent may expect to receive starting in state s and following the policy π. The goal for the agent then is to maximise this long-term return while only receiving feedback about its immediate single step performance.

The utility of reinforcement learning techniques came to prominence with the success of TD-Gammon [Tesauro, 1995]: a program that competed in several tournaments and achieved a level of play almost at par with the world's best backgammon players. Work by [Gosavi, 2009; Kaelbling et al., 1996] categorises reinforcement learning under model-free approaches based on value-function estimation (such as temporal difference [Sutton, 1988] and Q-learning [Watkins, 1989]), and model-based approaches that first estimate a model and then use it for finding the policy (such as prioritised sweeping [Moore and Atkeson, 1993] and E^3 [Kearns and Singh, 2002]). While the practical success of reinforcement learning added to its popularity, it also highlighted areas for improvement. The version of TD-Gammon that played impressively against champion player Neil Kazaross, for instance, required 1.5 million training games to learn such skill. Such lengthy convergence times led researchers to explore ways in which reinforcement learning performance could be improved, such as through the use of hierarchical abstraction.

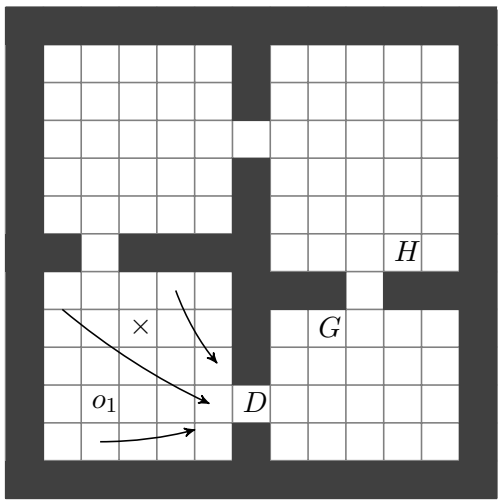


Figure 7.2: A room navigation problem [Sutton et al., 1999].

7.2.1 Options

The overall idea of an option [Sutton et al., 1999] is simple: to extend the choices available to the agent to also include *temporally extended actions* that encode "extra" domain knowledge. In its elementary form, *an option denotes a fixed policy that the agent follows for some period.* The option's policy may be thought of as providing a local, domain dependent strategy, in a wider context. In that sense, options may be used to specify subgoals that are to be achieved during learning.

In the options framework each state $s \in S$ is associated with a set of options O, synonymous with the action set A_s in the standard reinforcement learning setting. The set O may also supply the primitive actions A_s specified as single-step options. Each option $o \in O$ is further defined by an input set $I \subseteq S$, and the option is considered to apply in state s only if $s \in I$. If option o is selected is state s, then actions are executed according to the programmed policy $\pi : S \times A_s \rightarrow [0, 1]$ until the option terminates stochastically according to a termination condition $\beta : S \rightarrow [0, 1]$. Using an equivalent time-extended reward function $R(s, o)$ and state transition function $P(s'|s, o)$, Sutton et al. [1999] define a generalised form of the Bellman optimality equation $V^*_O(s)$ over a given option set O that reduces to the standard form (see Equation 7.2.1) if all options are single-step options. As such, the options framework provides a generalisation for the standard definition of actions in reinforcement learning.

Sutton et al. [1999] use a grid world example to illustrate the options idea. Consider the case where an agent is situated in a building with interconnected rooms with a single door connecting adjoining room as shown in Figure 7.2. Primitive actions allow the agent to move to an adjacent cell in either direction in a stochastic manner, i.e., with some probability the move action fails and the agent actually moves in a different direction.

The task is to learn to navigate from anywhere within one room to anywhere within another room, such as from the cell marked $\times$ to the cell marked G. Clearly, the best way to achieve this is to take the door marked D. One way to impart this information to the agent is through an option (o_1) that applies in all states that correspond to the agent being in the south-west room, and if selected, applies a policy that directs the agent towards the state that corresponds to being at cell D. In this way, while in the south-west room and given a goal G, the agent follows the local policy specified by o_1. On reaching cell D, the option o_1 terminates and is no longer applicable, at which point the agent can choose from the remaining single-step options (that correspond to the primitive actions) in order to reach G. Were the goal to navigate to cell H instead, one could specify a second option o_2 that applies in the south-east room and directs the agent to the interconnecting door to the north, and so no.

The motivation behind the options framework is to augment rather than reduce the set of available actions, such as in the grid world example where the sub-task of navigating between rooms is "reusable" across all individual tasks that require navigation from a cell in the first room to another in the second. While this means that the option set O is larger than the action set A of the initial MDP, it should be noted that the added options capture a sequence of primitive actions that make sense for the domain as specified by the expert. So, executing a relevant option is generally more advantageous compared to trying other (possible meaningless) combinations of primitive actions for the same period. Further, *if the option set is specified such that it does not contain every possible single-step option, then the search space for the problem is significantly reduced.* Such a specification by the expert that reduces the state space is also the motivation behind Hierarchical Abstract Machines (HAMs) that we discuss next.

The options framework allows an expert to supply procedural domain knowledge to the learner in the form of policies for sub-tasks. In this way, the options framework is

closely related to our learning framework, and an option is comparable to the procedural know-how encoded in a BDI agent's plan library. Further, the initial set I for a given option is comparable to the context condition of a given plan: both specify the conditions under which the procedure applies. In our case, however, this set is not known upfront (indeed we are trying to learn it), whereas knowledge of the set I is necessary in the options framework.

Another important difference between the two approaches is in their treatment of subgoals. In the BDI learning framework, procedural knowledge is encoded as a rich hierarchy of (sub)goals and plans to handle them. In the options framework, subgoals are not represented explicitly but may be viewed as *states* that are desirable during the execution of an option: the value of such states being encoded in subgoal specific reward functions. The idea is that the agent may wish to bring about certain desired states (subgoals) during the execution of an option.

This separation of concerns using a hierarchy of (sub)goals in the BDI system allows us to learn to resolve such subgoals independent of any higher level plan in which they may be used. In the options framework however, one cannot accommodate a rich hierarchy of subgoals (by specifying options in terms of options for instance), and learning over subgoals (i.e., learning inside options) is significantly more restricted. Overall, options are intended to resemble actions as much as possible (hence the name *temporally extended actions*), in order to inherit and benefit from the rich formal theory of reinforcement learning.

7.2.2 Hierarchical Abstract Machines

Hierarchical abstract machines [Parr, 1998; Parr and Russell, 1998], or HAMs, provide a mechanism for specifying domain knowledge for constraining the search space of a learning problem. The idea is to capture this knowledge as a hierarchy of partially specified machines. So while options [Sutton et al., 1999] (Section 7.2.1) capture sub-tasks as fixed policies, HAMs specify them using non-deterministic finite state machines. HAMs may be conceptualised as a tiered system of connected finite state machines whose transitions may invoke lower-level machines, and where the layers represent different levels of abstraction or detail in the system. Further, HAMs cater for non-deterministic decision making in a Markovian process by providing what are termed as *choice states* where the optimal selection is to be decided by the learning process. This

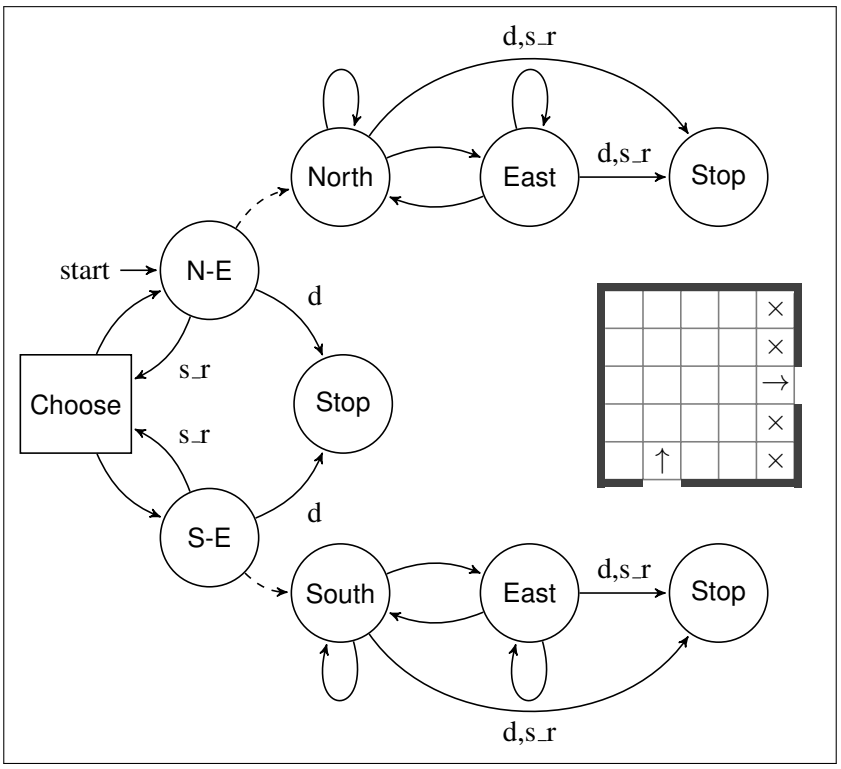


Figure 7.3: A machine for the room navigation problem [Parr, 1998].

framework for constricting the set of possible policies to be considered combined with the ability to specify such constraints at different levels of abstraction, allows HAMs to be applied to problems with much larger state spaces than possible in traditional reinforcement learning.

Specifically, a HAM is a program that when executed by an agent in a given state determines the set of actions that are allowed in that state. It is described by a set of states, a transition function that stochastically determines the next state, and a start function that specifies the initial state of the machine. States in themselves may be of four types: action states that directly interact with the environment, call states that invoke other HAMs as a subroutine, choice states that non-deterministically select the next state, and finally stop states that terminate execution of the machine (and optionally return control to a preceding call state).

Consider a grid world problem (Figure 7.3), taken from [Parr, 1998], where a robot is navigating a set of interconnected rooms in order to exit a building. The robot is

equipped with sonar sensors that detect when it has reached an obstacle in either direction. Suppose that the robot enters a given room via a southern entrance (marked by $\uparrow$) and the only exit is to the east (marked by $\rightarrow$). Given that the exit is always to the right of the robot entering this room, the domain expert may encode this knowledge in a HAM-constricted policy that effectively directs the robot towards the right.

An example of such a machine is also shown in Figure 7.3. The idea is to try and locate the exit by moving in an easterly direction. The HAM specifies two strategies for this: sub-machine N-E, i.e., a "move north or east" strategy, and S-E, i.e., a "move south or east" strategy. When invoked, they choose between moving east or north (south) with equal probability, and terminate and return control back to the parent machine when the door or right wall is reached (i.e., d,s_r). The robot begins by adopting the N-E strategy for finding the door. If that does not work and it reaches the eastern wall instead (the cells marked $\times$ and as indicated by the right sonar reading s_r), then it must adjust its strategy. The choice of which strategy to select next, however, is not exactly specified by the machine and is left up to the robot to decide (denoted by state Choose).

As described in [Parr and Russell, 1998], HAMs offer two important properties: first, given an MDP and an expert-provided HAM, there exists a new MDP in which the optimal policy is also optimal in the original MDP (in the set of policies that satisfy the constraints specified by the HAM), and an algorithm exists to determine this optimal policy; and second, a reinforcement learning algorithm may be constructed to find an optimal policy that satisfies the constraints of that HAM, without needing to construct a new MDP from it first (this is important since the environment model is not generally known *a priori*). The benefit is that *HAM-constrained exploration during reinforcement learning allows the agent to focus on a significantly reduced state space while still ensuring that the optimal solution is found*. Evidently, this comes at the cost of offloading some of the onus of decision making to the designer in the construction of the machines.

HAMs allow procedural knowledge to be encoded in a hierarchical manner similar to the way a BDI plan library does. Learning constitutes optimising decisions at each choice point that may lie at different levels in the hierarchy, and is conceptually similar to learning plan selection at different levels in a BDI goal-plan hierarchy.

Of course, HAMs are tied to the theory of MDPs, and BDI systems to logics and programming, and so they use very different languages. Overall, HAM-constrained rein-

forcement learning is focussed on finding *optimal* solutions using the MDPs (similar to the options framework [Sutton et al., 1999]), given a (hierarchical) model of the agent's behaviour. In contrast, the focus of our work in BDI learning is on maintaining the existing structure and benefits of BDI programs but seamlessly integrating (existing) machine learning techniques. Our aim is to address the nuances of learning in BDI hierarchies for use in practical applications. To that end, our contribution is to agent programming rather than to machine learning research.

7.2.3 Value Function Decomposition with MAXQ

Similar to the previous approaches, value function decomposition with MAXQ [Dietterich, 2000] also uses expert-provided procedural knowledge to constrict the set of available actions in a state. Here domain know-how takes the form of a hierarchy that describes a task graph, where each node represents a (macro) task that may further be decomposed into hierarchies of sub-tasks that finally terminate in primitive actions. However, in contrast to the previous approaches where a *single* value-function is learnt, MAXQ uses the task hierarchy to decompose the problem into *several* smaller MDPs whose solutions may be learnt independently and simultaneously. A critical point of difference then is that MAXQ produces *recursively optimal policies*, i.e., where the policy of each sub-task is optimal with respect to the policies of its children, whereas HAM-constrained learning for instance produces *hierarchically optimal policies*, i.e., the best policies given the constraints of the hierarchy even though the policy of a sub-task may not be optimal with respect to its children. That said, MAXQ learning may easily be adapted to produce hierarchically optimal policies by injecting "global" knowledge about the higher agenda in which a sub-task is being used.

Figure 7.4 illustrates a grid world taxi domain problem from [Dietterich, 2000], where the task is for the taxi (marked by $\times$) to locate and pickup passengers from a given address, drive them to the destination, and drop them off. The pickup and destination addresses are always one of $\{R, G, B, Y\}$ and are randomly selected for each learning episode along with the initial location of the taxi. The primitive actions available to the taxi agent include four actions to move one cell in either direction, and actions to pickup and drop off passengers. Dietterich's solution to the taxi problem is also shown in Figure 7.4. The graph shows the hierarchical decomposition of the task into multiple

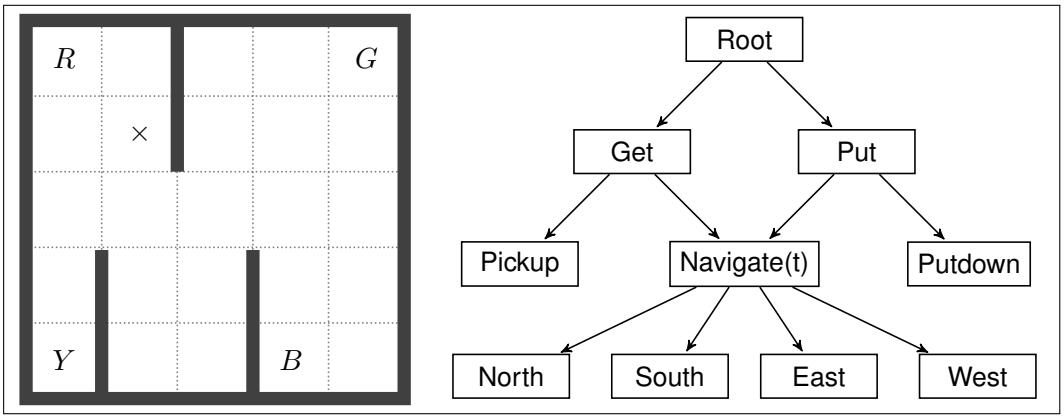


Figure 7.4: An example task graph for the taxi domain [Dietterich, 2000].

sub-tasks, and the idea is to use Q-learning to simultaneously learn the localised policy for each sub-task.

There are several attributes of this toy domain that justify a hierarchical approach to learning. Firstly, it highlights the benefits for temporal abstraction. While different instances of the task will require different number of actions to achieve (depending the pickup and drop off locations and where the taxi is initially located), all such tasks may be generalised as temporally extended "macro" actions for picking up and dropping off passengers, thus simplify the learning problem. Secondly, it shows that state abstraction could be used. For instance, the sub-task of picking up passengers is independent of the destination and therefore may be learnt locally. Finally, it suggests that the reuse of learning may be beneficial. For instance, if the agent has learnt to navigate between locations effectively, then it should be able to reuse this learning for both the pickup and drop off tasks rather than duplicating the learning effort.

The MAXQ approach to hierarchical reinforcement learning has the most semblance to our approach to learning plans' context conditions in a BDI goal-plan hierarchy. In particular we highlight the following points:

- In both approaches the intent of the structure is the same: the hierarchies capture the procedural know-how of the domain and are used to constrict the available actions in any situation. Further, learning conceptually achieves the same purpose in both: that of determining the applicability of available choices at each level

of the hierarchy in a given situation. Moreover, in both cases learning occurs simultaneously at all levels in the hierarchy. Of course, the learning technique in use is completely different: MAXQ uses Q-learning with a formal model of the environment given as an MDP. whereas our framework uses decision trees and does not use such a world model.

- The concepts of temporal abstraction (i.e., abstracting the time-extended sequence of primitive actions to a higher level task) and state abstraction (i.e., considering only states that are relevant to the local task) that apply in MAXQ learning also map directly to our framework. In fact, one of the main reasons that we are able to apply our framework to problems with large state spaces (see applications in Chapter 6) is due to these properties of BDI hierarchies that dramatically reduce the effective number of states to consider.

- The localised nature of the learning in our framework allows learnt solutions to be re-used in different higher level tasks. However, the same flexibility also introduces the issue of inter-dependence between tasks put together in this way. For instance, if the success of a higher level task depends on the successful resolution of two sub-tasks, then the way in which the first sub-task succeeds may impact the success of the second (such as when a shared resource is consumed). However, since the two sub-tasks do not "see" each other due to state abstraction, then there is no direct way of resolving this conflict (see our discussion of this limitation in Chapter 8). The solution is to expose the "extra" knowledge about the higher level agenda, but that directly impacts reusability as it somewhat binds the learning at the sub-task level to the larger context in which it is being used. As such, there is a balance that must be struck in terms of flexibility and learning ability [Dietterich, 2000]. As we discuss in Chapter 8, this trade-off is also of direct concern in our framework. The implication is that learning at the plan level cannot account for inter-dependence between subgoals of a higher-level plan, i.e., the higher context in which the plan is being used.

- Our proposed dynamic confidence measure (in Chapter 4) for accessing the ongoing reliability of the learnt solution is generic in nature and conceivably may be employed to guide exploration also in hierarchical reinforcement learning. This is valuable since determining the learning parameters for a given hierarchy normally requires trial and error on the part of the designer. However, the dynamic nature

of the measure will likely have implications for the convergence guarantees in reinforcement learning. One area of work that may be beneficial in the future would be to examine how such dynamic exploration strategies may be incorporated into hierarchical reinforcement learning in order to extend their application into environments with changing dynamics while still upholding convergence properties.

Overall, research in hierarchical reinforcement learning shares two key concerns with our work in BDI learning. First, that acting and learning are interleaved in an online manner and the agent must somehow balance between using current knowledge and discovering new knowledge when deciding what to do next; and second, that learning takes place in a hierarchy of decisions. Indeed, from a learning point of view, in some domains one solution could replace the other. The key point, however, is that our aim is to improve BDI programming. In that sense, it is more appropriate to think about techniques like MAXQ as candidate technologies to replace our decision tree based approach. In other words, hierarchical reinforcement learning technologies may well be a good match for learning in BDI goal-plan hierarchies.

Chapter 8

Discussion and Conclusion

In this thesis we have discussed the question of how BDI agent programs can be made more robust by incorporating a learning capability. Particularly, we have shown how BDI agents can improve plan selection in complex domains by integrating knowledge acquired from ongoing experience.

Summary of contributions

To this end, in Chapter 3 we have proposed a learning framework that augments plans' applicability, or context, conditions with decision trees. The idea is that a plan's applicability is determined by a two-step filter: first the programmer-specified context conditions, and second its associated decision tree that over time provides a meaningful generalisation of the likelihood of success of the plan in different situations. To select plans using this modified applicability criteria, we provided a probabilistic mechanism that selects from the set of candidate plans (whose programmed context conditions are satisfied) based on their predicted likelihood of success as well as the perceived confidence in current knowledge. This probabilistic selection ensures a balance between the exploitation of current understanding to make plan choices, and the exploration of available choices to further improve understanding. The dynamic confidence measure that we developed in Chapter 4 is built using a quantitative understanding of how well the agent's recent decisions have fared, combined with a sense of how well it knows the worlds it is witnessing. This is important as the dynamics of the environment may change over time in a way that makes prior learning less effective. Our learning frame-

work can be used in BDI programs of significant complexity including those that use recursion and failure recovery. We have demonstrated this using not only synthetic BDI programs in Chapter 5, but also two complete applications: the Towers of Hanoi puzzle and a modular battery system controller that we described in Chapter 6.

Design Considerations

An important limitation of our framework is that it does not consider interactions between a plan's subgoals. The implication is that learning cannot account for inter-dependence between subgoals of a higher-level plan, i.e., the higher context in which a sub-plan is being used. For instance, consider a travel-agent system that has two subgoals to book a flight and hotel accommodation on a fixed budget. Indeed, the way a flight is booked will impact the funds remaining for the next hotel booking goal, and some flight options may leave the agent unable to book any hotel at all. Since our agents have no information of the higher "agenda" at the subgoal level, there is no way for such dependencies to be learnt. We discussed this concern also in the design of our modular battery controller in Chapter 6, and indeed this limitation had a bearing on the final design of the system. One way of addressing this may be to consider extended notions of execution traces in Chapter 3 that take into account *all* subgoals that led to the final outcome, and not only the final chain of *active* subgoals. In general, however, exposing any "extra" knowledge about the higher level agenda will directly impact reusability of a subgoal as it somewhat binds the learning at the sub-task level to the larger context in which it is being used (a concern also identified by Dietterich [2000] in his work on hierarchical reinforcement learning). As such, there is a balance that must be struck in terms of design flexibility and learning ability.

We note the subtle disparity between the intended use of failure recovery in BDI systems and its potential use while learning. Failure recovery generally only makes sense for well founded goal-plan hierarchies, as it provides a fallback mechanism for unexpected failures. On the other hand, failures are commonplace when the agent begins learning. Arguably then, the use of failure recovery in the initial stages of learning should be discouraged. Indeed, it is possible that failure recovery may force the selection of every possible decision path until all options are exhausted. In domains where failures cause irreversible changes, such perseverance may well be counterproductive. An avenue for future work then may be to gradually enable failure recovery as learning progresses.

Learning Considerations

One issue, of course, has to do with maintaining the training set of past execution experiences per plan, indexed by world states. Simply storing such data may become unfeasible after the agent has been operating for a long period of time. Importantly, the larger the training set, the more effort is required to induce the corresponding decision tree. For the latter problem, one option is to filter the training data at hand based on some heuristic, and only use a subset of the complete experience set to induce the decision tree. For instance, we experimented with filtering the training data based on the recency of the world states experienced. In our battery controller application (Chapter 6), we were able to reduce the size of the data set used in training by almost 75% by removing "old" experiences with no significant change in performance. The generality of such data-filtering heuristics, however, is unclear and requires further investigation to make any claims. Using incremental approaches for inducing decision trees [Swere et al., 2006; Utgoff et al., 1997] will certainly address both problems, but may impact classification accuracy.

In the current framework, another consideration is the choice of propositions to include in the state representation for learning. In Chapter 3 we have discussed in detail how this set may be constructed, such as by considering the variables in the plan's context condition and the parameters of the event-goal it handles. One possibility for reducing this work for the programmer in the future is to extract the potential set of relevant propositions *automatically* by analysing the variables used in the goal-plan sub-tree below it, together with the preconditions and effects (if available) of actions that might be executed when handling the goal and subgoals.

For all experiments and applications described in this thesis, the plan applicability threshold is a user parameter that must be selected with some care. In general, by setting the threshold too low the agent may often try actions that are not very meaningful in the given situation. By setting it too high it may risk not learning the solution at all. An option here is to use a dynamic threshold value that starts off low when our confidence (Chapter 4) is also low, and gradually increases as our understanding of the domain improves.

Our work in this thesis is a step towards the future of agent programming languages. While we hope we have contributed to this vision in some small way, we know that

much still remains to be done.

Bibliography

Agha, G., Mason, I., Smith, S., and Talcott, C. (1997). A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72. 21

Airiau, S., Padgham, L., Sardina, S., and Sen, S. (2009). Enhancing the adaptation of BDI agents using learning techniques. *International Journal of Agent Technologies and Systems (IJATS)*, 1(2):1–18. 4, 29, 30, 41, 48, 49, 55, 67

Aite Group (2006). Algorithmic trading 2006: More bells and whistles. Accessed 10 March 2011, http://www.aitegroup.com/Reports/Default.aspx. 1

Armstrong, J., Virding, R., and Williams, M. (1993). *Concurrent programming in ERLANG*. Prentice Hall. 21

Barto, A. and Mahadevan, S. (2003). Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379. 4, 29, 112

Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press. 113

Bellman, R. E. and Dreyfus, S. E. (1962). *Applied Dynamic Programming*. Princeton University Press. 112

Benfield, S. S., Hendrickson, J., and Galanti, D. (2006). Making a strong business case for multiagent technology. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 10–15. ACM Press. 1

Bordini, R., Fisher, M., Pardavila, C., and Wooldridge, M. (2003). Model checking AgentSpeak. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 409–416. ACM Press. 16

Bordini, R., Hanebeck, Uwebner, J., and Wooldridge, M. (2007). *Programming multi-agent systems in AgentSpeak using Jason*. Wiley-Interscience. 3, 9, 13, 15, 16, 22

Bordini, R., Hübner, J., and Vieira, R. (2005). Jason and the golden fleece of agent-oriented programming. *Multi-Agent Programming*, pages 3–37. 21

Bordini, R. and Moreira, A. (2004). Proving BDI properties of agent-oriented programming languages: The asymmetry thesis principles in AgentSpeak (L). *Annals of Mathematics and Artificial Intelligence*, 42(1):197–226. 16

Bratman, M. (1987). *Intention, Plans, and Practical Reason*. Harvard University Press. 2, 10, 11, 12

Bratman, M. (1990). What is intention? In *Intentions in Communication*, pages 15–31. MIT Press. 10, 11, 17

Bratman, M., Israel, D., and Pollack, M. (1988). Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4(3):349–355. 2, 10, 12

Briot, J. (1989). Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 109–128. Cambridge University Press. 21

Broekens, J., Hindriks, K., and Wiggers, P. (2010). Reinforcement Learning as Heuristic for Action-Rule Preferences. In *Programming Multi-Agent Systems (ProMAS)*. 19, 29

Brusey, J. (2002). *Learning Behaviours for Robot Soccer*. PhD thesis, RMIT University. 3, 28

Burmeister, B., Arnold, M., Copaciu, F., and Rimassa, G. (2008). BDI-Agents for Agile Goal-Oriented Business Processes. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 37–44. International Foundation for Autonomous Agents and Multiagent Systems. 3, 10, 14

Busetta, P., Rönnquist, R., Hodgson, A., and Lucas, A. (1999). JACK intelligent agents: Components for intelligent agents in Java. *AgentLink Newsletter*, 2:2–5. Agent Oriented Software Pty. Ltd. 3, 9, 13, 15, 16, 22, 48, 68, 82

Bylander, T. (1994). The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204. 105

Clarke, E. (1997). Model checking. In *Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer. 16

Cohen, P. and Levesque, H. (1990). Intention is choice with commitment. *Artificial intelligence*, 42(2-3):213–261. 2, 10, 12

Dastani, M. (2008). 2APL: A practical agent programming language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248. 9, 15, 20

Dastani, M., van Riemsdijk, M., Dignum, F., and Meyer, J. (2004). A programming language for cognitive agents: goal directed 3APL. *Programming Multi-Agent Systems (ProMAS)*, pages 111–130. 20

de Boer, F., Hindriks, K., van der Hoek, W., and Meyer, J. (2007). A verification framework for agent programming with declarative goals. *Journal of Applied Logic*, 5(2):277–302. 3, 9, 15, 18

De Giacomo, G., Lespérance, Y., and Levesque, H. (2000). ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169. 9, 20

De Giacomo, G., Lespérance, Y., Levesque, H. J., and Sardina, S. (2009). IndiGolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming: Languages, Platforms and Applications*, chapter 2, pages 31–72. Springer. 9, 20

de Silva, L. and Padgham, L. (2005). A comparison of BDI based real-time reasoning and HTN based planning. *AI 2004: Advances in Artificial Intelligence*, pages 271–299. 21, 107

de Silva, L. P. (2009). *Planning in BDI Agent Systems*. PhD thesis, RMIT University. 21

de Silva, L. P., Sardina, S., and Padgham, L. (2009). First principles planning in BDI systems. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, volume 2, pages 1001–1008. International Foundation for Autonomous Agents and Multiagent Systems. 107

Dennett, D. C. (1987). *The Intentional Stance*. The MIT Press. 2, 10

Dietterich, T. G. (2000). Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition. *Journal of Artificial Intelligence Research*, 13(1):227–303. xi, 29, 112, 120, 121, 122, 125

d'Inverno, M., Kinny, D., Luck, M., and Wooldridge, M. (1998). A formal specification of dMARS. *Intelligent Agents IV Agent Theories, Architectures, and Languages*, pages 155–176. 13, 15, 22

d'Inverno, M. and Luck, M. (1998). Engineering AgentSpeak (L): A formal computational model. *Journal of Logic and Computation*, 8(3):233. 16

Erol, K., Hendler, J., and Nau, D. S. (1994). HTN Planning: Complexity and Expressivity. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, volume 2, pages 1123–1128. AAAI Press. 3, 21, 28, 105

Erol, K., Nau, D., and Subrahmanian, V. (1995). Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88. 105

Fahlman, E. (1974). A planning system for robot construction tasks. *Artificial Intelligence*, 5(1):1–49. 82

Fallah-Seghrouchni, A. and Suna, A. (2004). CLAIM: A computational language for autonomous, intelligent and mobile agents. *Programming Multi-Agent Systems (ProMAS)*, pages 90–110. 21

Fallah Seghrouchni, A. and Suna, A. (2005). Claim and sympa: a programming environment for intelligent and mobile agents. *Multi-Agent Programming*, pages 95–122. 21

Fikes, R. and Nilsson, N. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208. 105

Gaspari, M. and Zavattaro, G. (1999). An algebra of actors. In *Proceedings of Formal Methods for Open Object-based Distributed Systems (FMOODS)*, pages 3–18. Citeseer. 21

Georgeff, M. P. and Ingrand, F. F. (1989). Decision making in an embedded reasoning system. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 972–978. 9, 15, 22

Gosavi, A. (2009). Reinforcement learning: A tutorial survey and recent advances. *INFORMS Journal on Computing*, 21(2):178–192. 114

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java (TM) Language Specification*. Addison-Wesley Professional. 9, 48

Guerra-Hernández, A., Fallah-Seghrouchni, A. E., and Soldano, H. (2005). Learning in BDI multi-agent systems. In *Computational Logic in Multi-Agent Systems*, volume 3259 of *Lecture Notes in Computer Science*, pages 39–44. Springer. 3, 28

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 235–245. Morgan Kaufmann. 21

Hindriks, K. (2008). Modules as policy-based intentions: Modular agent programming in GOAL. *Programming Multi-Agent Systems*, pages 156–171. 19

Hindriks, K., Boer, F. D., Hoek, W. V. D., and Meyer, J. (1999). Agent programming in 3APL. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401. 3, 9, 15, 19

Hindriks, K., de Boer, F., Van Der Hoek, W., and Meyer, J. (1998). A formal embedding of AgentSpeak (L) in 3APL. *Advanced Topics in Artificial Intelligence*, pages 155–166. 19

Hindriks, K., de Boer, F., van der Hoek, W., and Meyer, J.-J. (2001). Agent programming with declarative goals. In *Proceedings of the International Workshop on Agent Theories, Architectures, and Languages (ATAL)*, volume 1986 of *Lecture Notes in Computer Science*, pages 248–257. Springer. 9, 15, 18

Hindriks, K., Jonker, C., and Pasman, W. (2009a). Exploring heuristic action selection in agent programming. *Programming Multi-Agent Systems (ProMAS)*, pages 24–39. 19

Hindriks, K., van der Hoek, W., and van Riemsdijk, M. (2009b). Agent programming with temporally extended goals. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 137–144. International Foundation for Autonomous Agents and Multiagent Systems. 19

Hoare, C. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8):666–677. 21

Hogg, C., Kuter, U., and Munoz-Avila, H. (2009). Learning hierarchical task networks for nondeterministic planning domains. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1708–1714. Morgan Kaufmann Publishers Inc. 110

Hogg, C., Munoz-Avila, H., and Kuter, U. (2008). HTN-MAKER: Learning HTNs with minimal additional knowledge engineering required. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, volume 8, pages 950–956. 110

Howard, R. A. (1960). *Dynamic Programming and Markov Process*. The MIT Press. 112

Hübner, J., Bordini, R., and Wooldridge, M. (2006). Programming declarative goals using plan patterns. In *Declarative Agent Languages and Technologies (DALT)*, pages 123–140. Springer. 16

Ilghami, O., Munoz-Avila, H., Nau, D., and Aha, D. (2005). Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the International Conference on Machine Learning*, pages 337–344. ACM Press. 108

Ilghami, O., Nau, D., and Aha, D. (2002). CaMeL: Learning method preconditions for HTN planning. In *International Conference on AI Planning and Scheduling*, pages 131–141. AAAI Press. 107

Ingrand, F., Georgeff, M., and Rao, A. (1992). An architecture for real-time reasoning and system control. *IEEE Intelligent Systems*, pages 34–44. 13

Kaelbling, L. P., Littman, M. L., and Moore, A. W. (1996). Reinforcement learning: A Survey. *Journal of artificial intelligence research*, 4(237-285):102–138. 114

Karim, S. and Heinze, C. (2005). Experiences with the Design and Implementation of an Agent-based Autonomous UAV Controller. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 19–26. ACM Press. 3, 10, 14

Karim, S., Subagdja, B., and Sonenberg, L. (2006). Plans as Products of Learning. In *Proceedings of the International Conference on Intelligent Agent Technology (IAT)*, pages 139–145. IEEE Computer Society. 28

Karmani, R., Shali, A., and Agha, G. (2009). Actor frameworks for the JVM platform: A comparative analysis. In *Proceedings of the International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM. 21

Kearns, M. and Singh, S. P. (2002). Near-optimal reinforcement learning in polynomial time. *Machine Learning*, 49(2):209–232. 114

Kernighan, B., Ritchie, D., and Ejeklint, P. (1978). *The C Programming Language*. Prentice Hall. 8

Langley, P. and Choi, D. (2006). Learning recursive control programs from problem solving. *The Journal of Machine Learning Research*, 7:493–518. 109

Lati, R. (2009). The real story of trading software espionage. Accessed 10 March 2011, http://advancedtrading.com/algorithms/showArticle.jhtml?articleID=218401501. 1

Levesque, H., Reiter, R., Lespérance, Y., Lin, F., and Scherl, R. (1997). GOLOG: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1-3):59–83. 9, 20

Lokuge, P. and Alahakoon, D. (2007). Improving the Adaptability in Automated Vessel Scheduling in Container Ports Using Intelligent Software Agents. *European Journal of Operational Research*, 177(3):1985–2015. 3, 28

Massive Software (2010). Avatar, Weta Digital & Massive plants. Accessed 10 March 2011, http://www.massivesoftware.com/avatar-weta-digital-massive-plants/. 2

McCarthy, J. and Hayes, P. J. (1969). Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence 4*, pages 463–502. Edinburgh University Press. reprinted in McC90. 20

Milner, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 21

Milner, R. (1999). *Communicating and mobile systems: the pi-calculus*, volume 13. Cambridge University Press. 21

Mitchell, T. (1997). *Machine Learning*. McGraw Hill. 4, 5, 24, 25, 27, 33, 48

Mitchell, T. (2006). The discipline of machine learning. http://www.cs.cmu.edu/ tom/pubs/MachineLearning.pdf. 5

Moore, A. W. and Atkeson, C. G. (1993). Prioritized Sweeping: Reinforcement Learning With Less Data and Less Time. *Machine Learning*, 13(1):103–130. 114

Moreira, Á., Vieira, R., and Bordini, R. (2004). Extending the operational semantics of a BDI agent-oriented programming language for introducing speech-act based communication. *Declarative Agent Languages and Technologies (DALT)*, pages 1270–1270. 16

Moreira, Á. F. and Bordini, R. H. (2002). An operational semantics for a BDI agent-oriented programming language. In *Proceedings of the Workshop on Logics for Agent-Based Systems (LABS)*. 16

Muldoon, C., O'Hare, G., Collier, R., and O'Grady, M. (2009). Towards pervasive intelligence: Reflections on the evolution of the Agent Factory Framework. *Multi-Agent Programming: Languages, Tools and Applications*, pages 187–212. 21

Muñoz-Avila, H., Aha, D., Nau, D., Weber, R., Breslow, L., and Yamal, F. (2001). SiN: Integrating case-based reasoning with task decomposition. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, volume 17, pages 999–1004. 108

Nau, D. (2007). Current trends in automated planning. *AI magazine*, 28(4):43. xi, 106

Nau, D., Au, T., Ilghami, O., Kuter, U., Wu, D., Yaman, F., Muñoz-Avila, H., and Murdock, J. (2005). Applications of SHOP and SHOP2. *Intelligent Systems, IEEE*, 20(2):34–41. 21, 105

Nejati, N., Langley, P., and Konik, T. (2006). Learning hierarchical task networks by observation. In *Proceedings of the International Conference on Machine Learning*, pages 665–672. ACM Press. 109

Nguyen, A. and Wobcke, W. (2006). An Adaptive Plan-Based Dialogue Agent: Integrating Learning into a BDI Architecture. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 786–788. ACM Press. 3, 28

Nilsson, N. (1982). *Principles of artificial intelligence*. Springer Verlag. 82

Odersky, M. and al. (2004). An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland. 21

O'Hare, G. (1996). Agent factory: an environment for the fabrication of multiagent systems. In *Foundations of distributed artificial intelligence*, pages 449–484. John Wiley & Sons. 21

Padgham, L. and Winikoff, M. (2002). Prometheus: A methodology for developing intelligent agents. In *Proceedings of the 3rd international conference on Agent-oriented software engineering III*, pages 174–185. Springer-Verlag. 20

Parr, R. E. (1998). *Hierarchical control and learning for Markov decision processes*. PhD thesis, University of California at Berkeley. xi, 112, 117, 118

Parr, R. E. and Russell, S. (1998). Reinforcement learning with hierarchies of machines. In *Proceedings of the Conference on Advances in Neural Information Processing Systems*, pages 1043–1049. The MIT Press. 117, 119

Petković, M. (2009). *Famous puzzles of great mathematicians*. American Mathematical Society. 82

Pokahr, A., Braubach, L., and Lamersdorf, W. (2003). JADEX: Implementing a BDI-infrastructure for JADE agents. *EXP - in search of innovation (Special Issue on JADE)*, 3(3):76–85. 3, 9, 15, 17

Pokahr, A., Braubach, L., and Lamersdorf, W. (2005). Jadex: A BDI reasoning engine. In *Multi-Agent Programming*, volume 15 of *Multiagent Systems, Artificial Societies, And Simulated Organizations*, pages 149–174. Springer. 17

Quinlan, J. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106. 4, 25

Quinlan, J. (1993). *C4. 5: programs for machine learning*. Morgan Kaufmann. 25, 27

Rao, A. (1996). Agentspeak(l): Bdi agents speak out in a logical computable language. In *Agents Breaking Away*, volume 1038 of *Lecture Notes in Computer Science*, pages 42–55. Springer. 3, 9, 15, 16

Rao, A. and Georgeff, M. (1991). Modeling rational agents within a BDI-architecture. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 473–484. Morgan Kaufmann. 2, 10, 13

Rao, A. and Georgeff, M. (1992). An abstract architecture for rational agents. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 439–442. 2, 10, 13, 17

Rao, A. and Georgeff, M. (1995). BDI agents: From theory to practice. In *Proceedings of the first international conference on multi-agent systems (ICMAS)*, pages 312–319. San Francisco. 2, 3, 10, 13, 17

Reiter, R. (2001). *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. The MIT Press. 20

Riedmiller, M., Merke, A., Meier, D., Hoffman, A., Sinner, A., Thate, O., and Ehrmann, R. (2001). Karlsruhe Brainstormers - A Reinforcement Learning Approach to Robotic Soccer. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 367–372. Springer-Verlag. 3, 28

Rönnquist, R. (2008). The goal oriented teams (GORITE) framework. In *Programming Multi-Agent Systems*, volume 4908 of *Lecture Notes in Computer Science*, pages 27–41. Springer Berlin / Heidelberg. 9, 15, 18

Russell, S. and Norvig, P. (2009). *Artificial intelligence: a modern approach*. Prentice Hall. 9, 104

Sardina, S., de Silva, L. P., and Padgham, L. (2006). Hierarchical planning in BDI agent programming languages: A formal approach. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1001–1008. ACM Press. 9, 18, 21, 107

Sardina, S. and Lespérance, Y. (2010). GologSpeak: Golog speaks the BDI language. In *Proceedings of the Programming Multiagent Systems Languages, Frameworks, Techniques and Tools workshop (PROMAS)*, volume 5919 of *Lecture Notes in Computer Science (LNCS)*, pages 82–89. Springer. 20

Sardina, S. and Padgham, L. (2010). A BDI agent progarmming language with failure recovery, declarative goals, and planning. *Autonomous Agents and Multi-Agent Systems*, 23(1):18–70. 15, 18

Satoh, I. (2000). MobileSpaces: A framework for building adaptive distributed applications using a hierarchical mobile agent system. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 161–168. IEEE. 21

Shoham, Y. (1993). Agent-oriented programming. *Artificial intelligence*, 60(1):51–92. 2, 9

Simari, G. I. and Parsons, S. (2006). On the Relationship between MDPs and the BDI architecture. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 1041–1048. ACM Press. 28

Singh, D., Sardina, S., and Padgham, L. (2010a). Extending BDI plan selection to incorporate learning from experience. *Robotics and Autonomous Systems (RAS)*, 58:1067–1075. 4, 30, 49, 51, 67, 82

Singh, D., Sardina, S., Padgham, L., and Airiau, S. (2010b). Learning context conditions for BDI plan selection. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 325–332. 4, 30, 41, 48, 49, 51, 53, 55, 67, 68

Singh, D., Sardina, S., Padgham, L., and James, G. (2011). Integrating learning into a BDI agent for environments with changing dynamics. In Toby Walsh, C. K. and Sierra, C., editors, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2525–2530, Barcelona, Spain. AAAI Press. 30, 50, 82

Slaney, J. and Thiébaux, S. (2001). Blocks world revisited. *Artificial Intelligence*, 125(1-2):119–153. 82

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley. 9

Sutton, R., Precup, D., and Singh, S. (1999). Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211. xi, 112, 115, 116, 117, 120

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, 3(1):9–44. 114

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. The MIT Press. 51, 112, 113

Swere, E., Mulvaney, D., and Sillitoe, I. (2006). A fast memory-efficient incremental decision tree algorithm and its application to mobile robot navigation. In *Proceedings of the International Conference on Intelligent Robots and Systems*. 41, 126

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology RTI Project*. 1

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68. 114

Tokic, M. (2010). Adaptive ϵ-greedy exploration in reinforcement learning based on value differences. In *KI 2010: Advances in Artificial Intelligence*, volume 6359 of *Lecture Notes in Computer Science*, pages 203–210. Springer Berlin / Heidelberg. 51

Utgoff, P. E. (1989). Incremental induction of decision trees. *Machine Learning*, 4(2):161–186. 41

Utgoff, P. E., Berkman, N. C., and Clouse, J. A. (1997). Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44. 41, 126

van Riemsdijk, B., van der Hoek, W., and Meyer, J. (2003). Agent programming in dribble: from beliefs to goals using plans. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 393–400. ACM Press. 20

Varela, C. and Agha, G. (2001). Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34. 21

Verrier, R. (2006). A mind of their own. Accessed 10 March 2011, http://articles.latimes.com/2006/jul/28/business/fi-animation28. 2

von Neumann, J. (1945). First draft of a report on the edvac. Technical report, University of Pennsylvania. 7

Watkins, C. J. (1989). *Learning from delayed rewards*. PhD thesis, King's College London. 114

Winikoff, M., Padgham, L., Harland, J., and Thangarajah, J. (2002). Declarative & procedural goals in intelligent agent systems. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 470–481. Morgan Kaufmann. 9, 15, 17

Winograd, T. (1971). *Procedures as a representation for data in a computer program for understanding natural language*. PhD thesis, Massachusetts Institute of Technology. 82

Witten, I. and Frank, E. (1999). *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann. 27, 33, 48

Xu, K. and Muñoz-Avila, H. (2005). A domain-independent system for case-based task decomposition without domain theories. In *National Conference on Artificial Intelligence*, volume 20, page 234. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 109

Yokote, Y. and Tokoro, M. (1987). Concurrent programming in concurrent smalltalk. In *Object-oriented concurrent programming*, pages 129–158. MIT Press. 21

Zhuo, H. H., Hu, D. H., Hogg, C., Yang, Q., and Munoz-Avila, H. (2009). Learning HTN Method Preconditions and Action Models from Partial Observations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1804–1809. 111