UC Santa Cruz

UC Santa Cruz Electronic Theses and Dissertations

Title

Learning Hierarchical Abstractions from Human Demonstrations for Application-Scale Domains

Permalink

https://escholarship.org/uc/item/6gs3t5xn

Author

Leece, Michael Ong

Publication Date

2018

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

LEARNING HIERARCHICAL ABSTRACTIONS FROM HUMAN DEMONSTRATIONS FOR APPLICATION-SCALE DOMAINS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Michael Leece

March 2019

The Dissertation of Michael Leece
is approved:

Arnav Jhala, Chair

Michael Mateas

David Aha

Lori Kletzer
Vice Provost and Dean of Graduate Studies

Copyright © by

Michael Leece

2019

Table of Contents

List of Figures

List of Tables

Abstract

Learning Hierarchical Abstractions from Human Demonstrations for Application-Scale Domains

by

Michael Leece

As the collection of data becomes more and more commonplace, it unlocks new approaches to old problems in the field of artificial intelligence. Much of this benefit is realized by advances in the decision problems of machine learning and statistics, but value can be gleaned for more classical AI problems as well. Large databases of human demonstrations allow for bootstrapping planning models in complex domains that previously would have been computationally infeasible.

This dissertation explores algorithms and systems for learning planning abstractions from human demonstrations in RTS games, which are more similar to real-world applications than prior domains in classical planning. I believe that this is particularly challenging and valuable, due to the inconsistency of human planning and variations in style and skill level between human players, in addition to the complexity of the domain. Any algorithm that intends to learn from human data must overcome these hurdles. My approach draws inspiration from a number of machine learning algorithms and paradigms, which have been developed explicitly in the context of large-scale, noisy data.

The primary contributions of this thesis are two algorithms for learning hierarchical planning abstractions from a database of human replays, a system for evaluating a planning model's ability to explain demonstrations, and an initial approach to using this system to learn a hierarchical planning model from scratch with human demonstrations in a popular RTS game.

For Maddy, who believed in me

Acknowledgments

I would like to thank my advisor, Arnav Jhala, for his oversight and advice throughout my entire graduate school career. I am particularly grateful for his persistence in the final two years, when affairs became decidedly non-standard.

I would also like to thank all of my committee members, both past and present. While the composition has shifted around some with changing areas of focus, every single one has provided valuable feedback that has helped this dissertation become what it is today. Thank you very much, Michael Mateas, David Aha, Lise Getoor, and Matt Taylor.

Without my peers, this would have been a much more challenging and less fulfilling experience. Trevor Santarra and Morteza Behrooz made the lab an enjoyable place to work, and together with my roommates Nick Cramer and Mike Sevilla were valuable sounding boards for any problems that I ran into, technical or motivational.

Finally, I would like to thank my emotional support group, my wife Madeleine and my family. Without their encouragement, the entire process that has culminated in this document would have been immeasurably more difficult.

Chapter 1

Introduction

One of the fundamental goals of artificial intelligence research is the creation of human-level intelligence. Over the years since the field was initially coined at the Dartmouth workshop in 1956, researchers have developed a number of common high-level paradigms for creating intelligent agents, with overlap and shared concepts between all of them. One example is heuristic search, in which agents attempt to efficiently search their action space by pruning out actions that are unlikely to achieve their end goals, giving them the computational power to look sufficiently far into the future to choose useful actions. Another example, and one that is growing in popularity, is reinforcement learning, in which agents repetitively experiment in a simulated environment, learning functions that let them estimate the value of taking a particular action in any given state that they find themselves. If this function is learned correctly, it allows for a greedy algorithm when choosing actions, since the learned value already takes into account the possible futures that follow that action. A final example of paradigms for creating intelligent agents, and the one which I will be focusing on, is planning, in which agents use domain knowledge and abstractions to schedule action sequences that will put their environment into the desired state.

Historically, one of the earliest implementations of planning was the STRIPS (STanford Research Institute Problem Solver) framework [35]. STRIPS treated its environment as a set of logical predicates that could be true or false, and defined operators that affected specific predicates in the environment. For example, if $On(A, B)$ indicated that item A was stacked on item B, the action $PickUp(A)$ might change that predicate to false, while changing another predicate such as $Holding(A)$ to true. Furthermore, one could define preconditions for operators, such as requiring that the agent not be holding anything for the $PickUp$ action to be legal. With these definitions, an agent can use any number of theorem-proving or search algorithms to find a sequence of actions that will lead to some desired end state, while satisfying all preconditions along the way, given the assumption that the environment has no exogenous changes. Some examples of these can be found in [100] and [12]. However, simple planners, while sufficient for domains that could be formally expressed in a limited number of predicates and actions, had a number of difficulties in real-world applications, from scaling to exogenous environment changes, and further research was necessary.

Since that stage, planning has branched into a number of separate but interconnected fields. Some examples of these include scheduling, which is the study of ordering actions in a plan that has partial ordering constraints, but may be optimized by correctly ordering subgoals and actions. Another example is the field of meta-reasoning for planners, which investigates how goals are formulated and added to a planner, or how re-planning and plan adaptation may be required in dynamic environments. One specific example of this would be the Goal Reasoning paradigm [132, 1]. A final area, and the one of most interest to this work, is that of hierarchical planning [147, 112]. These systems use the classical planning framework of preconditions and effects, but add the ability to encode abstract

higher-level tasks, which can be broken down into lower-level abstract tasks or concrete actions. These abstractions allow for feasible search for plans in domains in which the search space is computationally intractable, even with heuristics to narrow the search.

Hierarchical planners have been used to create full flight combat simulators [61], in which the cognitive architecture Soar was used in US Air Force training simulators in live training. They have been used to create artificial agents in award-winning interactive story-based games [96, 97, 99], something that major studios still shy away from in favor of scripted interactions due to the challenge. They have also been used in medical diagnostic software to assist doctors in performing diagnoses [111, 110] and to organize manufacturing processes [34].

However, the universal drawback that occurs in each of the systems listed above is the requirement for hand-coding expert knowledge into the hierarchical planning framework. This creates a number of serious issues for any large-scale hierarchical planning system. The first is the sheer amount of expert time required. For large-scale agents, the complexity of breaking down tasks and sub-tasks typically grows non-linearly with the complexity of the domain, and so it can take an enormous amount of expert time to translate all of their knowledge into the abstractions needed for the planner to operate intelligently.

The second concern with hand-coded agents is the problem of translation. While a human may be an expert in a domain, this does not guarantee that they are able to encode that expertise into whatever abstraction format is being used. In fact, there is evidence that human experts' descriptions of how to perform tasks differs from how they themselves perform the task. While there can be good reasons for this, in terms of increasing clarity or simplicity of explanation, it introduces a potential for error when translating between the intuitive knowledge

in an expert's head and the abstraction knowledge they wish to encode for an agent. Finally, one must consider the translation difficulty introduced when the project is complex enough to require multiple human experts, each of which may have slightly different mental abstractions, which must be reconciled together in the agent's abstraction encodings.

While there has been research in learning from observation and demonstration, as I will discuss in the Related Work sections of this manuscript, it has not reached the point where human demonstrations in application-level domains can be used to train a hierarchical planning model. My goal is to push towards that point, and in pursuit of that, I will be working on learning from demonstration in the real-time strategy genre of video games.

Historically, games have provided invaluable testbeds for the development of various techniques in artificial intelligence. In the 90s, the competition between chess champion Gary Kasparov and IBM's DeepBlue [17] brought AI to the forefront of the public imagination. More importantly, it, and the ensuing decades of research in chess agents, drove heuristic search techniques forward faster than they otherwise would have progressed. Similarly, in the 2000s, the work on Go agents drove forward what is now one of the most prominent search strategies, Monte Carlo Tree Search [38, 32, 21].

With the 'conquering' of Go by Google's AlphaZero [139], video games are being viewed as the next challenging domain for AI agents. AlphaZero's intellectual predecessor [104] was designed to learn how to play simple Atari console games using deep reinforcement learning, a task at which it has surpassed human ability. Following this, the next target is real-time strategy (RTS) games. The motivation for this is manyfold: as the name suggests, they are played at real-time, an important facet as we move towards agents that are expected to plan and act in

the real world alongside humans. They also require reasoning at many different levels, from strategic high-level tradeoffs to low-level spatial positioning. They are frequently partially observable, another way in which they mimic the real-world where prior research games have not.

Traditional RTS games follow a similar pattern. The player begins with a small number of units, capable of repeatedly collecting resources from the map. Using these collected resources, they can choose to improve either their economic or military capabilities (or some combination of the two). The ultimate goal is to have sufficient military strength to eliminate an opposing player's units, which have been built up similarly from a small starting point.

While seemingly simple when viewed from a high level, in reality there are countless decisions that a player must make to successfully play an RTS game. At the highest level, there is the tradeoff of whether to invest limited resources into military or economy. But beneath that, there are sub-choices at every level. Should the player build cheap military units now, or spend time and resources attempting to get stronger units later? Should they expand their economic growth in the area already under control, or should they try to claim new territory for more resources? Should they send units out to attempt to do some economic damage to the enemy, or keep them nearby to protect their own economy?

Furthermore, below that level, there are mid- and low-level spatial decisions that can be intricately linked to the higher strategic decisions. If one is training a defensive army, where is the best location on the map to set up a defense? When a battle is about to happen, what is the optimal position for all of the player's units to be in that maximizes their effectiveness?

See Chapter 2 for a more in-depth explanation of the specific RTS game most frequently used in this work, and the trade-offs and strategies that a player (and,

therefore, an agent) must balance for effective play.

1.1 Research Objective

In light of this, the primary research objective addressed by this dissertation can be phrased as follows:

> How can effective planning abstractions be learned from human demonstrations in RTS games?

There are a number of components to this problem statement worth discussing. The first is the goal of learning from human demonstrations. Humans, particularly in complex domains, are typically not perfect planners. As a result, some amount of their actions will not directly be in pursuit of their higher-level goals, and can even be counterproductive. This is exacerbated when the human being observed is not an expert in the domain, and is still exploring and growing their own knowledge base. While this second concern is less applicable for the domain used in this work, the existence of imprecise actions will remain. I will sometimes refer to this dilemma as the signal versus the noise problem, as it parallels that similar problem from statistics. This is the primary reason that many existing abstraction learning approaches are not applicable for learning from human demonstrations, as they rely heavily on first-order logic and would need to be modified to handle the imperfection of human planners.

Second is the choice of RTS games as the research domain. While this decision is more thoroughly discussed in Chapter 2 and above, the summary is that they provide a challenging domain that is intractable for pure heuristic search, thereby creating the need for abstractions. Additionally, there is a wealth of expert human demonstrations available to create libraries for learning from demonstration, and an existing research community for benchmarking and testing.

Finally, RTS games are a step closer than previous AI challenge domains such as Chess and Go to real world problems. In the end, the learning algorithms developed here will ideally be applicable to practical problems where expert human demonstrations are available, but encoding the sum of the expert domain knowledge is infeasible. This is more likely to be possible the more closely that the domain characteristics of the problems in which the algorithms are tested align to the real world.

1.2 Contributions

The primary contributions of this work are the following:

- Two new approaches to learning structures for hierarchical planning abstractions from human observation with no annotations. Prior work has focused either on the non-structural parameters of hierarchical planning models or on learning from well-annotated and/or noise-free examples, but I will present two algorithms for learning structure from raw human traces.

- A new technique for evaluating hierarchical models' fitness in explaining a database of examples, using a modification of the ABL programming language. By adjusting the underlying model in ABL from a deterministic to a stochastic one, it is possible to calculate how likely a given model is to produce an example trace, and thus evaluate how well it explains human behavior across a set of examples.

Some secondary contributions include hierarchical planning and reinforcement learning agents for the simplified RTS game Planet Wars, and an analysis of their effectiveness in that domain. Additionally, I will present some approaches for

using the model fitness evaluation function listed above to learn full hierarchical planning models from scratch.

1.3 Organization

Chapter 2 describes the primary RTS game that I use for this work, StarCraft: Brood War (SC:BW), both to set the stage for examples used in later chapters, as well as to motivate its choice as a domain.

Chapter 3 will discuss the reactive planning and reinforcement learning experiments I ran in the game Planet Wars, as a proxy for spatial map control in SC:BW.

Chapter 4 introduces a pattern mining approach to learning low-level hierarchical planning abstractions in SC:BW, using replays of professional players. While not able to capture all components of human play, I will demonstrate that they successfully learn the more formulaic aspects of gameplay that arise in the early game, and components of late-game play as well.

Chapter 5 presents a probabilistic approach to learning hierarchical planning abstractions in a synthetic domain. To do this, I model HTN tasks as probabilistic processes, in order to use an expectation maximization algorithm to learn the most likely structures for said tasks, again using professional replays as demonstrations.

Chapter 6 showcases a modification to the ABL programming language for the purposes of learning from observation. I show how to use this change in assumptions to learn parameters for an existing ABL model, then apply it to the hierarchical structures learned in the previous two chapters to create a SC:BW agent whose high-level strategy decisions have been learned entirely from scratch.

Chapter 7 summarizes the research presented, and discusses areas for further exploration based off of this work.

Chapter 2

StarCraft: Brood War

This chapter provides a background for the primary domain of the work to follow. Portions of the work will make use of smaller and simpler domains, which will be described in their respective chapters, but the motivating domain for the overarching work merits its own description.

StarCraft: Brood War (SC:BW) is a real-time strategy (RTS) game published by Blizzard Entertainment in 1998. It is a science fiction game set in the far future, with humanity having expanded to the stars and come into conflict with two alien races. The player(s) play as economic and military commanders, giving orders to a wide range of units with the ultimate goal of destroying the opponents' units and structures.

SC:BW is widely considered to have launched the professional gaming scene that has since grown to encompass many more games. Centered in South Korea, professional players and leagues competed for large prizes in front of millions of fans. The popularity of professional competition has many benefits for researchers using SC:BW as a domain, which I will discuss later.

At a high level, SC:BW is an exemplary domain for artificial intelligence research for a number of reasons, which this chapter will describe in sequence. The

first is the game itself, which requires reasoning at multiple levels of abstraction, with decisions made at each level tying into the reasoning at other levels. In addition, its real time nature and state/action-space complexity preclude many traditionally successful approaches like heuristic search and reinforcement learning from being successful without additional knowledge engineering. The second is the gaming community that surrounds it, from professional gamers providing high-level replays of their own play to crowdsourced knowledge repositories maintained by fans. The third is the existing development of artificial agents and AI competitions that have arisen in the past decade, which provide a benchmark against which to measure results.

2.1 Gameplay

2.1.1 Overview

At a mechanics level, gameplay consists of issuing orders to player-owned *units* and *structures* (sometimes *buildings*). Units can be ordered to move, attack, patrol, gather resources or construct buildings, while buildings can be ordered to train units and/or research *upgrades*, depending on the specific type. Upgrades will either improve the efficiency of a player's army or unlock more powerful units. The two main delineations for units are units that are able to collect resources and construct buildings, also called *workers*, and units that are primarily used for fighting the opponent, typically referred to as *military* or *army* units.

A typical game begins with each player owning one structure for creating workers and collecting resources and a small number of workers themselves. They begin by building up an economy, training new workers to improve their resource collection speed, and potentially additional bases at new resource locations. While

Figure 2.1: Screenshot of a game of StarCraft:War

this is occurring, they are also beginning to build up an army, constructing new buildings that train military units, and researching upgrades. This introduces the concept of the *tech tree*, which is a branching set of upgrades and buildings that allow different units to be trained based on what has been researched or constructed, respectively. Later in the game, each player will have researched and constructed the full tree, making every unit available, but early in the game with limited resources, different players may choose different branches to follow.

Once players have military units, there are a number of options available to them. The first is to simply keep them near their bases in a defensive posture, in order to protect their resource collecting workers. This will ensure that they are able to build up a larger and larger army, but may leave their opponent free to do so even faster. They may also use them to attack the opponent, attempting to disrupt their economy in a bid to gain a larger army in that manner. In

reality, players must typically balance between both of these, defending from their opponent's attacks while simultaneously threatening the opponent's bases and resources. Ultimately, players win when they are able to destroy every structure that the opponent controls. I will be focusing on 1v1 games between two players, but team and free-for-all modes are also popular variants.

Figure 2.1 shows a sample screenshot from a game of SC:BW. In the lower left, the player's workers collect resources from mineral patches. In the lower right are structures that the player is using to create their military units, which appear near the center of the screen. Meanwhile, the opponent's military units in purple are attacking in the upper right of the screenshot, forcing the player to defend their base with the military units they have created.

While seemingly straightforward, these mechanics create an enormous and non-trivial game-space for players to explore. In particular, the imperfect information of the game, which I will discuss in Section 2.1.3, and the time-delayed nature of actions such as training, constructing, or researching creates a tension where if one player is able to deduce the high-level goals of the other quickly enough, there are sufficient options available to choose a counter-strategy to punish whatever strategy the original player is targeting.

As an example, a common high-level classification of early game strategies can be as a *rush*, a turtle, or a *boom*. In a rush, a player builds very little economy, instead producing military units as quickly as possible and sending them to the opponent's base immediately, hoping to find little to no military to oppose them. In a turtle, a player builds economy while taking advantage of military units and/or buildings that are stronger at defending than attacking to protect their growing resources. In a boom, a player forgoes all military in order to rush to a strong economy, at which point they transition into strong military production.

Domain	Blocks World	Chess/Go	Bridge	SC:BW	Real-world navigation
Fully observable/ Partially	Fully	Fully	Partially	Partially	Partially
Single Agent/ Multiagent	Single	Multi	Multi	Multi	Multi
Deterministic/ Stochastic	Determ.	Determ.	Determ.	Determ.*	Stochastic
Episodic/ Sequential	Seq.	Seq.	Seq.	Seq.	Seq.
Static/ Dynamic	Static	Static	Static	Dynamic	Dynamic
Discrete/ Continuous	Disc.	Disc.	Disc.	Cont.	Cont.
Known/ Unknown	Known	Known	Known	Known	Known

Table 2.1: Domain characteristics for a number of frequently discussed AI domains. *-There is a very small amount of randomness involved in a specific aspect of SC:BW, but it is generally irrelevant to actors in the domain.

At a high level, if a player identifies that their opponent is being greedy and booming, they may want to rush out a military to punish the lack of military. If they identify their opponent as rushing, they will choose to turtle, giving them a stronger economy while still protecting against the early military. And finally, if they identify that their opponent is turtling, they can take advantage of the lack of offensive capability to boom economically, giving them a better setup for the later game. In reality, this example is a simplification of true gameplay, but it illustrates a single example of the kind of tradeoffs and opponent modeling that players need to balance during a game of SC:BW.

2.1.2 Complexity

Domain Properties

Table 2.1 shows the breakdown of a number of AI domains, including SC:BW, using the domain characteristics discussed in [135]. The goal of AI research in testbeds such as games is often to develop approaches that are practical in the real world. As one can see from the table, SC:BW is further along the spectrum in aligning with common unconstrained real-world problems, making it a valuable environment for research. While most of these properties will be discussed and made more clear while explaining the competencies required for an intelligent agent below, I would like to make a few notes about them here.

First, the domain is not strictly continuous, as there is a time step involved in the frames per second that the game runs at, typically 24 frames per second. However, it is considered a close enough approximation that for now, with current computational power, it falls in the continuous family.

Second, one aspect of the domain that does not particularly align with most real-world problems is the nature of the multi-agent relations, namely that it is adversarial. While some practical applications certainly include adversaries, the majority include at worst participants who simply have differing objectives, but not diametrically opposed ones. This is something to keep in mind when developing algorithms in SC:BW. If they are to be widely applicable, they cannot rely heavily on the adversarial nature of the domain.

Finally, when military units fire from low ground to high ground, there is a $\sim$47% chance that their attack will miss. This is the only randomness present in SC:BW, and is generally not considered explicitly in agents, beyond the advantage that high ground confers. As a side note, this is one of the reasons that keeping one's military units active is important, so that it is possible to engage in fights

where your units have a high-ground advantage.

Game-space Complexity

The complexity of SC:BW, both in terms of possible states and possible actions in each of those states, dwarfs that of prior AI challenge domains. Chess and Go have state-spaces of roughly 10^{40} and 10^{170}, respectively, with action-spaces that roughly average 30 and 100 respectively per move. These spaces already prove far too large for efficient search, unless an agent is able to intelligently narrow down the list of candidate actions effectively. In Chess, researchers initially surpassed human level play with fine-tuned hand-crafted heuristics [17], while in Go, search was directed by a deep neural network position evaluation function learned from millions of games, both from humans and from self-play [138, 139].

However, the state- and action-spaces of SC:BW are many orders of magnitudes larger than these examples. Weber [155] estimated the state space of a SC:BW game using the following analysis:

$$O((TXY)^U)$$

U - number of units in play

T - number of unit types in StarCraft

X - number of horizontal map tiles

Y - number of vertical map tiles

A typical map in SC:BW is 256x256 tiles, a game can support up to 1700 units at once, and there are roughly 100 different unit types available in a standard SC:BW game. This results in a state-space estimate of roughly $10^{11,500}$. While this number is already massive, in reality the state-space is much larger, as there are a large number of features not included in this estimate, such as unit health,

cooldown timers, and projectiles, all of which can factor strongly into decisions.

As for the action-space, or decision complexity, Aha et al. [2] provided an initial analysis for RTS games in general:

$$O(2^W(A * P) + 2^T(D + S) + B(R + C))$$

W - number of workers

A - number of the type of worker assignments

P - average number of workplaces

T - number of troops

D - number of movement directions

S - number of troop stances (Attack, Move, Hold)

B - number of buildings

R - average number of research options at buildings

C - average number of unit types for training at buildings

While some of these are overestimates (e.g. movement directions and troop stances containing some overlap), it nonetheless demonstrates the extreme size of the decision space of SC:BW. Weber [155] has simplified the analysis by assuming that actions for units can be chosen independently, which reduces the complexity down to

$$O((W * A * P) + (T * D * S) + (B * (R + C)))$$

Even this, for a game state with only workers on a standard map, will result in over a million possible actions. It is clear that some amount of abstraction is necessary, since narrowing this action space down to the point that it can be searched effectively is impractical, especially when considering the time constraints of a real-time environment.

2.1.3 Competencies

Competencies are abstractions representing a high-level ability to handle a specific aspect of a challenging domain. In reality, the only truly requisite ability for an agent in SC:BW is the ability to choose and issue the right commands at the right time. However, in the same way that it is impractical to attempt to plan using purely concrete actions, these competencies represent the broad challenges for expert play that have been grouped together in humans' mental models for both their own play and artificial agents.

Imperfect Information

The first challenge that agents must deal with is the fact that SC:BW is an environment with imperfect information. The maps themselves are fixed, with no randomly generated elements, meaning that a prepared player will have full knowledge of the terrain and resource areas. However, they are not granted information on what actions their opponent is taking.

The mechanics for information gathering work as follows: each unit and building that a player controls has an individual field of vision that consists of circles with varying radii centered on said units and buildings. The union of these fields provides the vision range for the player, and they may observe any units, buildings or terrain within that range. The remainder of the map is greyed out. This obscuring is typically referred to as the "fog of war", referencing Sun Tzu's Art of War. Figure 2.2 shows this in action, with the player's units and structure making the left half of the screen visible, while the right half remains shrouded. Any enemy units in that area would be invisible to the player until entering the sight radius of one of their units.

An expert player must manage this information precisely, both gathering it

Figure 2.2: Screenshot demonstrating 'fog of war'. The darker gray area is out of sight of the player, and might contain enemy units.

and denying it to the opponent, depending on the higher-level strategy they are pursuing. Players will frequently send scouting units to ascertain what their opponent is currently doing, sacrificing resources in an attempt to increase the fidelity of their mental model of the current state of the game. Any artificial agent must be able to do the same, weighing the balance of how accurate their mental model is and how much it is likely to be improved with the potential loss of resources in sending units to unseen areas of the map. It is possible, and even common, to code agents that ignore this aspect of the game, simply following one path without considering their opponent's actions, but such agents will never reach human-level gameplay, due to their predictability and inflexibility.

Long-term Planning

In addition to needing to deal with imperfect information, and made more difficult by it, players must be able to make and execute long-term plans under changing conditions. If a player wants to have a strong economic base from which to train their military 10 minutes in the future, they must begin creating that now, by training extra workers and expanding to more resources. Similarly, if one wants access to some specific type of unit, a player must chart their way through the tech tree, in a way that doesn't sacrifice too much in any other area such as defense. And yet, it can be extremely difficult to extend long-term planning out to the far future in more than an abstract, non-concrete way, due to the next required competency: the ability to handle an adversarial agent working against the player.

Adversarial Reasoning

SC:BW is an adversarial domain by definition, with each player attempting to eliminate the other. As a result, any effective player must take into account their opponent's plans while choosing actions, including the recursive mental modeling that comes from the fact that their opponent is doing precisely the same thing.

The most classical adversarial reasoning is alpha-beta search, which attempts to optimize one's reward assuming that the opponent will play optimally. While this is difficult in an imperfect information game, there have been versions developed [142], as well as versions for time-durative actions rather than turn-based [27], and hierarchical planning systems as well, and even applied to a miniature RTS game [120]. Even so, these techniques have not yet percolated into modern agents fully. At a strategic level, agents still have a tendency to focus on achieving their own goals, rather than adapting or reacting to their opponents' plans.

There are exceptions to this, and many agents will place an opponent's strategy into one of some number of buckets, in order to use the correct response, but the adaptivity is not as strong with this kind of system.

Spatial Reasoning

Another required competency is the ability to reason about the 2-dimensional space created by the map and terrain. Conceptually, this capability can be broken down into roughly three types of thinking:

Tactical – Since combat in SC:BW is not instantaneous, but rather a complex interplay of heterogeneous units acting according to their abilities and strengths, there are ways to optimize one's army in the midst of combat with an opponent. These are unit commands at the tactical level, optimizing behavior in the midst of combat.

As a brief example of this, a few of the considerations that must be managed are:

- As many of the player's units should be dealing damage as possible, meaning melee units should be in front of the ranged units when the battle starts.
- A player's high-damage-dealing units need to be kept protected throughout the battle, to maximize the amount of damage that they are able to inflict on the opponent's units.
- Conversely, the opponent's strongest units should be targeted in an attempt to eliminate them early if possible.
- Weakened units should be pulled back out of range of their attackers to save them, while not allowing the same behavior from the opponent.

- Units should focus their fire on individual enemies rather than spreading out damage, to reduce the incoming damage more quickly.

These are only some of the low-level positioning considerations that professional human players are balancing in the middle of a combat that will only last from seconds to tens of seconds.

This tactical level of spatial reasoning is one of the areas that is most well-suited to artificial agents, and their competency has matched, and in some scenarios passed, human expertise [137] [127] [28] [146]. However, this alone is not sufficient to reach human-level performance, even when only considering the spatial aspect of SC:BW, due to the gaps most agents have in the next area, strategic level spatial reasoning.

Strategic – Beyond the ability to control units in direct combat with the opponent, an expert player must consider the placement of their armies at a higher level. An army may be held back to defend a player's economy, sent to a new resource area to secure it for expansion, or placed threateningly near an opponent's bases. Each has its own strengths and weaknesses, and is appropriate at different points throughout a single game. Moreover, if a player splits their military units to achieve multiple objectives at once, they run the risk of being defeated piecemeal by a weaker force.

A concept typically used to discuss this high-level spatial reasoning is the idea of *map control*. Map control refers to the areas of the map that each player 'controls', either directly with military units, or indirectly through controlling routes to and from those regions. Some strategies will attempt to aggressively grab map control, limiting the opponent's vision of the map and opportunities to send out scouts or expand to new resources, while others will intentionally cede control, with the intention of retaking it in the future. An expert agent must

understand the tradeoffs being made with various army positionings, something that few currently do.

I will discuss this in more detail in Chapter 3, where I experiment in the simplified RTS game Planet Wars, which focuses primarily on strategic-level spatial reasoning.

Static – One final type of spatial reasoning that has grown as a sort of emergent phenomenon through years of professional play is that of building placements, which I refer to as static spatial reasoning. Intricate professional strategies have been developed that rely on very precise arrangements of buildings, creating choke points and defenses that allow smaller armies in a defensive posture to defeat larger ones. For each new map, players will analyze the proper building layout for each strategy, an offline process to reduce the consideration required when in a game.

While I will not explicitly discuss this aspect of SC:BW gameplay in this research, there are a number of papers related to it in the literature [18, 130]. While they currently exist as standalone projects, I believe it is important to integrate them into existing frameworks, as decisions made in building placement are typically influenced by the higher-level strategy choices of the player or agent.

2.2 Community

A second critical component of SC:BW as a domain is the existing community of players, from casual to professional. Though the game was released over 20 years ago, a significant number of players continued playing through that time on third-party servers like ICCup [1], Fish and Brain, though the latter two have since been shut down. In 2017, Blizzard released a remastered version of the game,

[1] www.iccup.com

reigniting interest even further, and there are currently roughly 100,000 [2] active users on Blizzard's Battle.net platform.

2.2.1 Professional

As mentioned above, SC:BW launched the professional gaming scene, with hundreds of players competing as a full-time profession [8]. Teams are sponsored by major companies, competing in year-round leagues at both an individual and team level for prizes worth tens to hundreds of thousands of dollars. Players on professional teams approach the game as seriously as any professional athlete, complete with coaches, dietitians and crowds of fans.

This is critical when analyzing SC:BW as an AI domain, as it means there are examples of high-level play where the signal-to-noise ratio of human actions is maximized. With two novice players, the players themselves will be exploring the plan space themselves and refining their own understanding of the game. On the other hand, experts will have a full understanding of the goals they are attempting to achieve at any given moment, and in what ways they are going about achieving these goals. When attempting to learn abstractions, this guarantee of intent behind actions, as compared to random exploration, is critical. A parallel field of research could involve learning alongside human novices, but this work leaves that option to future researchers.

However, this expertise is useless if there is no way to access it. Fortunately, SC:BW provides the option to save one's own replays for future analysis. Players can go back and watch through their games to find errors in their tactical or strategic gameplay. Additionally, one can study the games of other players to pick up new strategies or tactics. Some professional players take advantage of this to

[2] from www.starlog.gg

release 'replay packs' to their fans, collections of their games from practice or competitions for the fans to watch for enjoyment and/or training. In addition, some tournaments release the replays of all games played to the public, uploading them to fan websites [149, 16]. In conjunction, these two things mean that researchers have available a sizable number of replays of high-level professional play, perfect for learning from observation.

2.2.2 Amateur

In addition to, and perhaps partially due to the professional scene, there is a large scene of amateur SC:BW players who take the game seriously. These players gather on online forums [149, 11] to watch and study professional games, as well as give and receive advice on improving their own gameplay. This provides a number of useful information streams for an AI researcher, such as:

Game analysis – Game analysis is done in both text and video formats. Players will evaluate professional games, giving a textual breakdown of the major events that decided the game in one player or the other's favor. Others [125] have made a profession of creating video content that analyzes professional games in an in-depth manner. This provides a potential learning stream for researchers interested in incorporating natural language processing.

Guides – In addition, players create detailed guides explaining various concepts of gameplay. From high-level abstract strategies to step-by-step walkthroughs of an early game, these guides are a slightly more formulaic and general form of information than individual game analyses. This gives researchers a useful comparison by which to evaluate whether the abstractions or plans that they are learning align with 'common-sense' knowledge of the game, or if they are discovering something either new or incorrect.

Crowdsourced knowledge databases – Finally, all of these sources have been collected and organized in volunteer-run wikis [148], which makes them accessible and searchable without needing an in-depth, expert-level knowledge of the game itself. An example of how these information streams can potentially be useful to researchers can be found in Chapter 4, where I compare learned early build strategies to the community's knowledge base for validation.

2.3 AI Competitions

As has been amply demonstrated by the enormous success of the ImageNet challenge in the image classification and segmentation domains [30, 74], competitions can be a very useful means for driving forward research. They provide a clear metric and means to measure against other approaches, and, when properly structured, incentivize the sharing of progress at frequent intervals, which allows the research community to build on one another's ideas and breakthroughs.

SC:BW has a number of open-source competitions for artificial agents, which can provide a large suite of agents both present and historical to test against. The oldest-running competition is the AIIDE StarCraft AI Competition [3], which has run every year since 2010. It was begun by Ben Weber at UC Santa Cruz, and has been carried forward by Dave Churchill [24]. It has steadily grown, and the 2017 competition included 28 distinct agents. This collection, which contains a large range of ability, provides a fantastic benchmark for the development of any agents.

The second competition of note is the Student StarCraft AI Tournament & Ladder (SSCAIT) [19], which was started by Dave Churchill as a more open competition, less centered around research papers and more open to any AI enthusiast

[3] www.cs.mun.ca/ dchurchill/starcraftaicomp/

who wished to create a SC:BW agent. As a result, it has over 100 agents currently running, though the quality range is greater.

2.3.1 EISBot

A specific artificial agent is of notable importance to this work, named EISBot. Developed by Ben Weber as part of his dissertation at UC Santa Cruz [155], EISBot is written in the reactive planning language ABL. While there are a number of interesting features that EISBot incorporates, from goal-driven discrepancy analysis to particle-based state estimation, it is particularly useful primarily due to its use of hierarchical planning abstractions when making high-level strategic decisions, precisely the kind of abstractions that this research hopes to learn from demonstrations. In Chapter 6, I will discuss how EISBot was integrated with this work as both a launching point and a benchmark.

As a side note, an additional bot that makes use of hierarchical planning is Soar-SC, an agent that makes use of the Soar cognitive architecture [76, 150]. However, this agent is much less well-developed and complete than EISBot, and as such I will not be using it for this research.

2.4 StarCraft 2

Finally, I would like to note that Blizzard has recently published an API for the sequel to SC:BW, StarCraft 2. As a domain, many of the same advantages apply, and it is possible that StarCraft 2 will be an even better research domain than SC:BW. There are a couple of factors that make this likely. Some quirks of programming due to the age of SC:BW makes unit control fairly unpredictable, inserting a large amount of what is essentially randomness into that problem area,

an issue that is much more smoothly addressed in StarCraft 2. Additionally, the replay upload process has been improved and integrated into the game even more, leading to larger databases of human demonstrations to work from. While for now the API is intended most directly for deep neural network and reinforcement learning research, in the future it is worth considering porting this research to StarCraft 2, if that is the direction that the research community moves in.

Chapter 3

Spatial Reasoning

One of the core competencies for an agent in RTS games is spatial reasoning. RTS games are typically played out on a 2-dimensional map, where interactions between units are moderated and/or limited by their geographical distance. In addition, in many games terrain features will segment the map, creating narrow passages that can be choke points or branching paths that can be difficult to defend or attack accurately. Furthermore, reasoning in this area particularly suffers from the traditional generalization problem, since there are potentially infinite maps and starting locations. When learning from observation, some intelligent abstractions are required in order to apply the same knowledge across multiple examples.

In a game like SC:BW, there are at least 2 levels of spatial reasoning, which I will differentiate as the tactical and strategic levels. At the tactical level, units must be tightly controlled in the midst of battles. As discussed in Chapter 2, this involves balancing a large number of tradeoffs while issuing orders to individual units in order to maximize damage output while keeping as many units alive as possible. At the strategic level, a player's armies must be used to control various regions of the map. This can serve multiple purposes, from protecting newly

acquired resource areas, to forcing an opponent to split defenses by controlling a region that can threaten multiple areas, to keeping an eye on a potential back-door into the player's base. Managing map control, as it is frequently termed, is one of the hallmarks of a great professional player, and as such is an important component for any artificial agent.

In this chapter, I will discuss work in a simplified RTS game named Planet Wars. By reducing the complexity of nearly all aspects of a traditional RTS game *except* strategic level unit positioning, it creates a domain where that is the defining competency for any player, human or artificial. As such, it is an ideal testbed for working on spatial reasoning and abstraction. I will present a reinforcement learning system in addition to fixed reactive planners, and show how learning is possible, though difficult, in this domain. The results in this chapter were published in [86].

Within the larger framework of this dissertation, this chapter serves as an introduction to the types of domain that we will be working with throughout the document. In addition to this, while part of the evaluation will focus on reinforcement learning, which we will pivot away from afterward, it also will demonstrate how to apply hierarchical abstractions to an RTS game. Lastly, while this remains future work, it will be important to integrate spatial reasoning components into full-fledged hierarchical planning learning systems for them to achieve full human competency.

3.1 Related Work

Reinforcement Learning in games

Reinforcement learning has seen implementations for strategy games, but the general bent of the work has focused on other aspects of these games, tending to avoid their spatial nature.

Jaidee and Muñoz-Avila have implemented a Q-learning agent for the real-time strategy game Wargus that has shown promise in defeating fixed-strategy opponents [55]. However, their state and action space is devoid of positional information, instead using resource and troop levels to define state, and using simpler position-ignorant actions that are variations on the basic attack and defend actions. As a result, no strategic or tactical spatial abstractions are considered at all, simply a binary decision. Rörmark has shown a similar result, with an agent that primarily enforces resource management strategies [134]. Again, the action space is mostly to determine the order in which to construct buildings with limited resources, with simple attack and defend orders that ignore location on the map. In contrast, the Planet Wars environment eliminates the majority of the resource investment tradeoffs, and focuses exclusively on troop locations.

A number of papers have investigated reinforcement learning for low-level tactical unit control in games. Jaidee et al. have implemented a Wargus (an open-source clone of WarCraft, a predecessor to SC:BW) unit control agent for multiple units in a static environment, where the initial troops are the only actors throughout a test, although it ignores location to focus more on target selection [56]. Wender and Watson demonstrated an agent that was able to learn simple unit control tactics, and Ponsen et al. successfully navigated space while avoiding a pursuer [163, 128]. However, both of these only demonstrated a single agent

directing a single unit. Shantia et al. use a connectionist reinforcement learning approach for tactical level combat, using a neural network to approximate a Q-value in a form similar to the future Deep Q-networks [137]. Molineaux et. al. have proposed an integration of reinforcement learning and case-based reasoning for continuous action spaces [107]. This last set is more closely related to our work, as they take unit location as input for their decision-making process. However, the environments in which they operate have fixed numbers of units, which removes the resource control aspect of the problem.

A large number of non-reinforcement learning approaches have also been applied to the tactical spatial reasoning aspect of RTS games, arguably making it the most well-studied area of the genre. Some other approaches have included evolutionary algorithms, neural networks, game-tree search, case-based reasoning, potential/vector fields, and Bayesian models [127] [28] [146] [121]. Some of these approaches have had limited success, but others have worked quite well, approaching and even surpassing human ability in some circumstances.

Amato and Shani have implemented a number of reinforcement learning algorithms in Civilization IV, a turn-based strategy game, also with some success [3]. However, they were focused on choosing high-level strategy, and their agent chooses between fixed strategies based on current game state. This is similar to the Adaptive ABL work by [140], which modifies the ABL programming language to include a reinforcement learning component for learning specificities, effectively learning to choose between hard-coded strategies.

Outside of RTS games, reinforcement learning has been making a number of breakthroughs in significant challenge areas, many of which do include spatial reasoning as a critical component. In the MOBA (Multiplayer Online Battle Arena) genre, which is a descendant of the RTS genre, reinforcement learning is very

quickly approaching expert human-level play. The OpenAI Five is an artificial agent that plays DotA 2, one of the most popular games in the world, and recently defeated a team of 5 ex-professional players, though in a restricted game setting that banned certain limited aspects of gameplay. It used Proximal Policy Optimization in place of traditional Q-learning, and long short-term memory (LSTM) neural networks as policy functions for each of its agents [122].

Another such advancement was the work out of DeepMind using deep neural networks to learn to play a suite of Atari games purely through raw pixels and reinforcement learning, coined as Deep Q-Networks (DQN) [104]. This was a key landmark in adapting reinforcement learning to work with the high-dimensional input present in video games. Since then, this work has been improved to better handle knowledge transfer and retention [68] and exploration in spaces with sparse reward signals [7], to the point where it is able to quickly learn to beat every arcade-style game they have thrown at it.

Since then, DQNs have most notably been used to surpass human-level play in the long-standing challenge domain of Go. While the initial system was pre-trained on human demonstrations and fine-tuned using self-play [138], it has since been surpassed by a pure reinforcement learning agent that has built its knowledge base purely by playing countless Go games against itself, and learning from those [139].

Many of these reinforcement learning breakthroughs have developed after the publishing of the work presented here. As a result, it is certainly worth considering how they can be applied to this research to improve and extend it in the future.

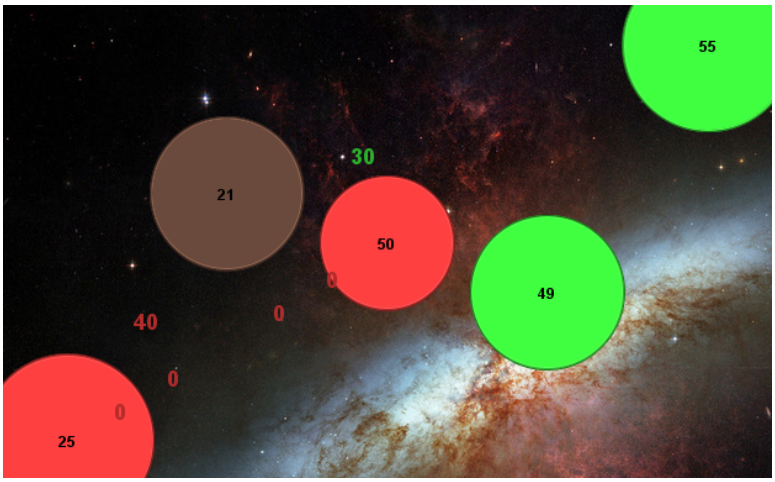

Figure 3.1: A game of Planet Wars. Red and green planets are player-owned, brown is neutral. Numbers in between planets represent fleets in transit.

3.2 Background - Planet Wars

Planet Wars is a real-time strategy game focused on spatial control. It was used in the 2nd Google AI Challenge, and is a simplified version of the commercial game Galcon [1]. Players take on the role of a galactic commander directing fleets of spaceships to occupy planets and eliminate the opposing player's fleets entirely. Victory is achieved when no enemy ships remain on the map, or in the case of a stalemate situation, by controlling the most ships when a predetermined time limit is reached. It is an adversarial, perfect-information, deterministic game. With a slightly longer tick length than SC:BW, it is on the border between turn-based and real-time: sufficiently quick that any human player would consider it real-time, but slow enough that the time between ticks is very noticeable to a human, rather than flowing smoothly.

Figure 3.1 shows a screenshot of a Planet Wars game. Planets are represented by circles, with ships represented by the numbers, either on a planet or traveling between two planets. When a planet is owned by a player, it will produce ships at

[1] https://www.galcon.com/, accessed October 2018.

a rate proportional to the size of the planet. This means that larger planets are more valuable to capture, as they will grow a player's fleet the fastest. Each tick, a player can order any number of ships from each of his planets to travel to any other planet on the board. Travel time is based on the Euclidean distance between the source and the destination planet, and once dispatched, ships cannot be recalled. On arrival at a planet, if it is owned by an enemy player or is neutral (the brown planet in Figure 3.1), ships will engage in combat. If the planet is owned by the player, they will add themselves to the total number of ships already present.

Combat is instantaneous, and ships always fight at a simple 1-to-1 trade. That is, the larger fleet will always win, and the total number of ships remaining will be the size of the larger fleet subtracted by the size of the smaller fleet. Ships will only fight on planets; they will pass without fighting if they cross paths while traveling between planets.

I will present two of the tradeoffs that players must balance for an efficient strategy, one at the tactical level and one at the higher strategic level. While this is only a subset of the factors for decision-making, it should help demonstrate the complexity of the system.

One of the most fundamental tactics is 'stealing' newly captured planets. In order to take a neutral planet and start using it to produce ships, a player must commit (and lose!) a number of ships equal to the defending neutral fleet. This incentivizes the opponent to launch a fleet to arrive at said planet one tick after the player's, after the combat losses have been sustained but before ship production has recouped the losses. To counteract this, a player either needs to take planets closer to his fleets than the opponent's, in order to react quickly enough to any steal attempts, or be able to move with overwhelming force, to provide a defense for the newly conquered planet immediately.

At a more strategic level, the positioning and repositioning of ships is likely the most critical decision-making process for a successful agent. Ships must be balanced between taking new neutral planets to increase production, and being kept near the front lines in order to respond to any moves by the opponent. Not only that, but if reinforcements to the front lines are sent on a single long flight, there is no opportunity to react mid-flight. As a result, it may be better to take extra time hopping between planets, which may take longer, but will give greater flexibility in responding to an opponent's actions.

This is a portion of what must be considered, but shows that high-level gameplay is clearly a non-trivial task. Beyond the complexity of the domain, it is also interesting as an approximation of spatial control in StarCraft. In SC:BW, a player must allocate their units to take and control various areas of the map in order to get access to greater resources, or disrupt the resource-gathering of an opponent. While there are a number of differences (units do not necessarily trade 1-for-1, movement-restricting terrain, no neutral enemies, etc.), it is sufficiently similar at an abstract level to hope that abstractions and knowledge gained in Planet Wars could be transferred to SC:BW with some retraining.

3.3 Agents

Within this domain, I implemented a number of agents for experimentation, in addition to using the competition-provided opponents. The implemented agents fall into two main paradigms: reactive planning agents and reinforcement learning agents. I wished to get an evaluation on the strengths and weaknesses of reactive planning in complex game environments as preparation for developing learning algorithms for hierarchical planning approaches, and also to evaluate how a more machine learning-focused approach would fare. From there, one can determine

how best to transfer the knowledge gained in this domain to the more complex domain of SC:BW.

The baseline agents included in the competition, with a brief description of their high-level strategy, are:

- **Random**: Executes a random legal action each turn.
- **Prospector**: Expands slowly, only attacks opponent once all neutral planets are claimed.
- **Rage**: Only attacks opponent's planets, never neutrals.
- **Bully**: Primarily attacks opponent's planets, occasionally neutrals.
- **Dual**: Attempts to get ahead of opponent in planets, then switches to attacking.

3.3.1 Reactive Planning

The reactive planning agents were programmed by hand using ABL (A Behavior Language), a language for encoding hierarchical planning abstractions. Structurally, these agents balance their resources between two primary top-level goals: defending their own planets and taking unowned planets, whether neutral or enemy. Depending on the level of optimization (mentioned below), the agents will be more or less proactive in responding to actions that the opponent has taken in response to their own goals, pulling ships from nearby planets to protect one that is under attack, and adding follow-up ships to an attack that is being defended against in a similar way. Unfortunately, they do not currently take advantage of the modifications to ABL for learning from demonstrations that I will present later in Chapter 6, and as a result are static agents. As I will discuss later, this is

one of the areas that I would like to extend this work. For now, the programmed agents have been labeled as follows:

- **WithDefense**: Expands as agressively as possible while defending its own planets. Only sends one fleet against a planet at a time.
- **GeneralMedOpt**: Weighs expanding aggressively and attacking the opponent, while still defending its own planets. Medium optimization.
- **GeneralHiOpt**: Highly optimized, projects planet strength via fleet growth and reinforcements into future turns and sends follow-up attacks if necessary.

3.3.2 QPlayer

The remaining set of agents uses reinforcement learning to iteratively improve their play over the course of many games. Collectively named QPlayer, it uses the classical Q-learning algorithm, as described in [135]. For those unfamiliar with Q-learning, the fundamental goal of the algorithm is to fill in a table (referred to as the Q-table) that accurately reflects the future value of taking any given action in any given state. If this table can be learned perfectly, an agent's policy can reduce to taking the action in whatever state it is in that has the highest projected future reward. However, since in most situations it is nearly impossible or computationally infeasible to fully learn the table, the Q-learning algorithm specifies both how to balance exploring new cells of the state-action table with exploiting existing knowledge, as well as assigning credit from the reward function backwards to decisions made along the way. For more information regarding the specifics, one may read [135].

The Q-learning algorithm relies on three components: a definition of a game state, to list the rows in the table; an action function mapping from states to legal

actions in that state, to list the columns in the table; and a reward function for the cells of the table to estimate their future value.

A full specification of a Planet Wars state involves a description of which player owns each planet, the number of troops on each planet, and all fleets in flight between planets. Fleets can only travel between two planets, and because they travel a fixed distance each tick, there is a fixed number of locations that ships can be present in on the map. At a rough estimation, this gives us a state space of

$$3^P * T^P * T^{F_l}$$

where

- P is the number of planets on the map
- T is the maximum number of troops possible on the map
- F_l is the number of possible fleet locations between planets on the map.

On a relatively small map with only 5 planets, this comes out to 10^{170}, which leads us to a Q-table far too large to explore in a reasonable number of games. Even replacing the maximum number of troops (typically in the thousands) with an average maximum fleet size in a typical game (~200), to move from an extreme upper bound to a more accurate estimation, results in a state space of roughly 10^{130}, still too large for Q-learning.

To address this, QPlayer has a number of granularity parameters, controlling how fine-grained of a state the agent considers when constructing its table. The fleet granularity parameters create buckets for fleet sizes. With a granularity of 50, for example, a fleet of size 60 and one of size 75 would be seen as identical to the agent, while one of size 25 would be in a separate bucket from one of size 75. Clearly, the higher the parameter, the more inefficient the agent's responses,

as it can overestimate the number of troops required to respond to a threat in an amount up to the granularity, but the smaller the state space it needs to explore. There is a separate fleet granularity parameter for fleets in transit between planets and those at rest on a planet. Additionally, there is a fleet distance parameter that merges fleets in transit between two planets in state space. For example, with a fleet distance parameter of 1, all fleets traveling between two planets that are more than halfway there will be merged into a single fleet, and all less than halfway there will be considered another fleet. With a higher parameter, fleets will separate out until at the highest value, every individually sent fleet is seen on its own in state space.

The action function is simply the set of all possible fleet dispatches from owned planets to all other planets, regardless of ownership. Since taking actions at a finer granularity than the agent could see the world would be meaningless based on the agent's understanding, actions are discretized in the same manner as the state space, using the same fleet size granularity parameters.

The reward function for QPlayer is the difference between its total fleet size and the opponent's fleet size. This has the advantage of providing intermediate signals throughout the game, rather than relying on the final win/loss signal to propagate backward. In a game where early actions can have a large impact but the final result can be many ticks from then, Q-learning can take a long time to accurately update those cells of its table, with its decaying blame updates. A reward function that more quickly returns the impact of decisions can alleviate this problem, though the agent must still learn that small penalties in the reward function (spending ships to take a neutral planet) can lead to greater returns (acquiring the production resources of said planet). Finally, even if the agent hyperoptimizes for this reward function at the cost of finishing the game, by

leaving an opponent alive with a single planet while owning the rest of the map, it will still achieve victory by time limit, so the incentive still aligns with victory.

Planets as Actors

The final variant of QPlayer shifts the viewpoint from which decisions are made. In the previously described agent, the game state that the agent uses to make decisions is the full description of the game. Notice that if the fleet granularity parameters are set to 1, and the distance division parameter is set to the maximum distance between planets, QPlayer will use a perfect description of the game state.

However, as mentioned earlier, this full representation has very serious scalability issues, with the Q-table quickly becoming impossible to even hold in memory. In light of this, I developed a variant of QPlayer that attempts to learn a planet-level policy, such that each planet decides independently where to send its fleets. This has the downside of making large-scale strategic planning more difficult, but the upside of parallelizing learning. Similar to the advantage that convolutional neural networks gain from reusing small kernels of weights, sharing the Q-table between all owned planets throughout a simulation allows for more updates and more states to be explored. This approach is similar to the reinforcement learning agent for the RTS game Wargus implemented in [55].

3.4 Experiments and Results

The primary goal of my experiments was to determine how well reinforcement learning was suited to learning in the Planet Wars environment, and how much of an effect the granularity abstractions had on performance. As a secondary objective, I wanted to see how abstraction choices would affect the scalability of

	Rand	Pros	Rage	Bully	Dual	Defense	MedOpt	HiOpt
Rand	-	0.0%	10.1%	0.0%	0.0%	0.4%	0.0%	0.0%
Pros	100.0%	-	60.4%	94.3%	5.7%	1.9%	3.8%	0.0%
Rage	89.9%	39.6%	-	54.7%	37.7%	45.3%	17.0%	32.1%
Bully	100.0%	5.7%	45.3%	-	0.0%	0.0%	0.0%	0.0%
Dual	100.0%	94.3%	62.3%	100.0%	-	9.4%	32.1%	3.8%
Defense	99.6%	98.1%	54.7%	100.0%	90.6%	-	66.0%	0.0%
MedOpt	100.0%	96.2%	83.0%	100.0%	67.9%	37.0%	-	7.5%
HiOpt	100.0%	100.0%	67.9%	100.0%	96.2%	100.0%	92.5%	-

Table 3.1: Win rates for fixed-strategy opponents (row wins against column represented)

the agent, and whether it was possible to scale up intelligently.

3.4.1 Static Opponents

As mentioned above, eight fixed-strategy opponents were used in testing, a mix of heuristic agents and reactive planning agents. To determine a rough ranking of these opponents' difficulty, they were played against each other on 51 maps, with the results of the tests shown in Table 3.1. In general, the reactive planning agents outperform the heuristic agents, with the higher level of optimization corresponding to higher win rates. This is encouraging in demonstrating that reactive planning abstractions can be useful in the spatial control domain. But without adaptivity and learning, it is limited to demonstrating that it is possible to encode expert knowledge for the domain using reactive planning, but not that it is particularly suited for learning said abstractions. This remains a task for future work.

3.4.2 QPlayer – Learning Rate

The first set of tests used a smaller sized map with 5 planets, symmetrically distributed. The goal for this test set was to show that QPlayer could learn to

defeat fixed opponents, and also to provide a baseline learning level for comparison with learning after training. Testing involved setting a freshly initialized QPlayer against one of the fixed strategy opponents, and running either until QPlayer converged to repeating a winning strategy or we reached 10,000 games, whichever came first. The final loss of QPlayer was recorded as the learning time for that test. Using the granularity variables in QPlayer, 18 different configurations were tested, each of which had a different fineness of state-action options.

In addition, we performed a similar suite of tests using the multi-agent variant with planets as actors on 3 full-sized maps (20-30 planets). The intention of these tests was to determine whether the modifications to the abstractions used would be able to handle the increased complexity and state-space size.

3.4.3 QPlayer – Prior Training

Another important factor to evaluate was whether the value abstractions that QPlayer learned were generalizeable, or whether they were specific to each opponent. If it was overfitting to each opponent, there is little hope that it would generalize even farther to a separate game. In light of this, I used leave-one-out training for 10,000 games with each of the 8 opponents being held out respectively, evaluating the convergence time afterwards on the remaining opponent. The same planet configuration was used as in the initial tests, and the same granularity parameters as well.

For both tests, the Random opponent was removed. While QPlayer very quickly learns to defeat the Random player an overwhelming percentage of the time, it takes an enormous number of games to reach a 100% win rate. This is a result of the opponent's random actions, which explore the state-space completely randomly. Even with QPlayer making intelligent moves, this will frequently drive

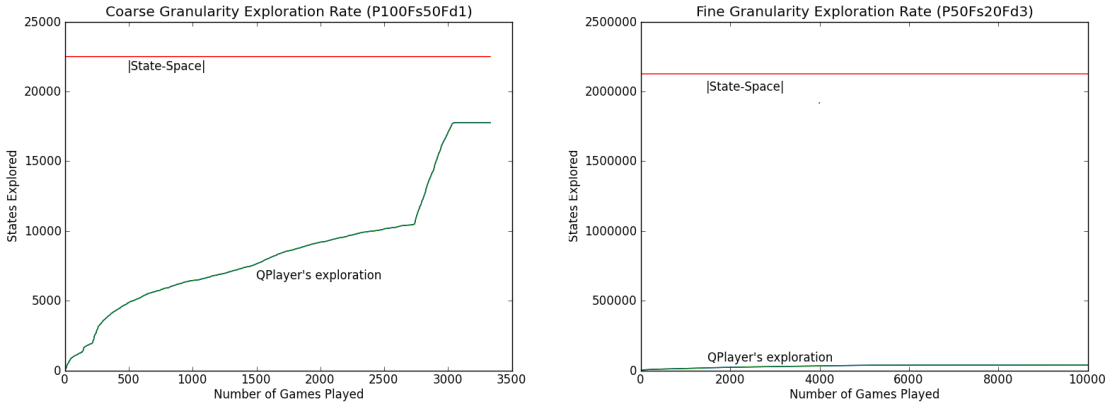

Figure 3.2: State-space exploration for QPlayer in a coarse and fine configuration, respectively. Note the change in the state space size on the y-axis as well.

the state into unexplored areas, at which point QPlayer also will play randomly in order to explore the new area of the table it is working within. In the future, a more generalizable evaluation function, for example what would be used for DQN, would be better able to generalize existing knowledge to unseen states, addressing this issue, but for now the Random opponent is simply removed.

3.4.4 Evaluation

The first observation to be made is regarding the difference that the granularity makes in the success of the agent. If we consider Table 3.2, it is clearly visible that, as a general rule, the more finely grained the state, the longer it takes to reach convergence. There are two factors at play here to be balanced as a trade-off. First, the more coarse the states and actions, the more difficult it is for the agent to effectively play the game. It may use more troops than necessary for some task, leaving it short in other areas, purely because it only can send troops in large increments. On the other hand, the state-space of a coarser configuration is much smaller, and so the agent can explore it more fully, getting more accurate value assignments for each state-action pair. Figure 3.2 shows the state-space

	Pros	Rage	Bully	Dual	Defense	MedOpt	HiOpt
Ps100Fs50Fd0	116	10001	89	100	417	1618	2777
Ps100Fs50Fd1	95	176	58	102	299	4521	4936
Ps100Fs50Fd3	513	269	228	83	718	4942	4235
Ps100Fs20Fd0	416	8906	136	327	189	1064	6050
Ps100Fs20Fd1	250	326	1294	743	201	2412	6788
Ps100Fs20Fd3	324	1043	1909	1872	5235	5320	8948
Ps100Fs10Fd0	220	10001	381	671	413	1463	3746
Ps100Fs10Fd1	773	6850	1658	4669	531	4563	9003
Ps100Fs10Fd3	1398	9442	1518	2448	2541	5752	8241
Ps50Fs50Fd0	128	10001	207	636	120	2340	1909
Ps50Fs50Fd1	63	2002	1271	2808	49	1186	3129
Ps50Fs50Fd3	316	3190	80	2820	179	3399	6137
Ps50Fs20Fd0	331	10001	259	408	408	2393	5465
Ps50Fs20Fd1	1541	3629	2867	1683	1062	2423	7754
Ps50Fs20Fd3	359	9938	3047	348	2061	6168	10001
Ps50Fs10Fd0	1531	9277	1537	1664	310	412	3650
Ps50Fs10Fd1	1566	10001	5873	3354	1239	1579	8617
Ps50Fs10Fd3	2532	10001	2395	2896	3379	5067	10001

Table 3.2: Average number of games before convergence for QPlayer for various granularity configurations (Ps = Planet size granularity, Fs = Fleet size, Fd = Fleet distance divisions)

exploration for a coarse- and fine-grained test, respectively. The discontinuities in the number of states explored arises from the agent opening up a promising new area of the state space through random exploration and needing to learn evaluations for these states. It is clear that the coarser agent was able to explore a majority of the state, while the finer agent explored only a small fraction. These results show that, in this domain, the tradeoff between state-space coverage and representation precision is more heavily weighted toward the former. It is possible that this is due to the adversarial nature of the environment, where accurate values for actions can be difficult to determine, as they depend heavily on opponents' responses. More experimentation would be necessary to prove this postulation, however.

	Pros	Rage	Bully	Dual	Defense	MedOpt	HiOpt
Ps100Fs50Fd0	183	10001	0	2508	720	1200	2607
Ps100Fs50Fd1	0	176	**0**	32	1567	158	992
Ps100Fs50Fd3	5	**164**	83	83	75	3396	205
Ps100Fs20Fd0	0	3414	25	53	37	766	5581
Ps100Fs20Fd1	33	143	111	15	2307	7122	6719
Ps100Fs20Fd3	0	390	43	45	262	10001	6717
Ps100Fs10Fd0	313	10001	19	**0**	128	2456	6678
Ps100Fs10Fd1	209	6667	0	66	775	8240	4435
Ps100Fs10Fd3	5067	10001	95	5000	151	5361	5156
Ps50Fs50Fd0	91	10001	0	90	2163	5279	3460
Ps50Fs50Fd1	**0**	453	0	277	**1**	569	7434
Ps50Fs50Fd3	0	7358	0	1728	210	7286	3404
Ps50Fs20Fd0	110	10001	11	2357	3335	3487	205
Ps50Fs20Fd1	1255	1914	0	101	3353	3415	5404
Ps50Fs20Fd3	0	**9292**	0	199	149	3720	**3398**
Ps50Fs10Fd0	3981	5836	942	1822	1246	41	1960
Ps50Fs10Fd1	112	**6667**	3606	3130	75	3970	10001
Ps50Fs10Fd3	908	10001	0	6698	**233**	189	10001

Table 3.3: Average number of games to convergence for QPlayer following 10,000 games of leave-one-out training. Bolded values are statistically significant ($p < 0.05$)

While this granularity conclusion is nontrivial, it is still secondary to the main question of this work, whether Q-learning is able to learn strategies and abstractions in a spatial management game. To answer this question, consider Table 3.3, where entries of 10,000 indicate a failure to learn a winning strategy, and anything less indicates the mean number of games required to learn a winning strategy. One can see clearly that, while the convergence time depends on the granularity and the optimization level of the opponent, QPlayer does in fact converge to a winning strategy in the large majority of cases. In particular, with correct state granularities, it will converge to defeat every fixed-strategy opponent in the suite.

While they are slightly difficult to read, by comparing Tables 3.2 and 3.3, one can see that when QPlayer has the opportunity to train against the remaining

opponents prior to testing, its learning time is reduced. This is evidence that QPlayer is truly learning strategies for spatial control and resource management, rather than simply randomly exploring until it hits on a winning strategy and then exploiting it, since knowledge gained against one opponent is being generalized to others. Representationally, these strategies may be difficult to extract, as they are stored in a Q-value table. Even so, this is a key result, as it indicates that Q-learning is a legitimate approach for reasoning about spatial control in fully integrated agents. At the same time, the improvement is not universal, and there is not an immediately clear pattern as to when it does not occur. This gives some cause for concern, and is why further development would be necessary before fully committing to this approach.

Unfortunately, the attempts to scale Q-learning up with planets as actors to larger maps were unsuccessful. While the agents were able to compete on the large maps without running out of memory, and even defeat the opponent occasionally, they never were able to win consistently. There are two potential reasons for this. The first is that it has simply not had enough training games, and is still attempting to learn accurate values for each state-action pair. Continued testing, with greater than 10,000 games, can determine if this is the case. However, there is a point at which the amount of time required to train an agent becomes infeasible. The second potential flaw is that, in removing state information from each planet's individual view, I have eliminated information that is necessary to correctly assess the value of a state.

Fixing this issue within the classic Q-learning paradigm would require more fine-tuning of the expert knowledge by testing many different state features, which is of less interest for this work than the agent learning on its own. However, another possibility would be to investigate DQNs as an alternative. This approach,

which merges Q-learning with deep neural networks as an evaluation function, has been successful in games with larger state spaces than Planet Wars. One thing that would need to be addressed is the fact that many of the examples in which they have succeeded have smaller action spaces, but even that is not universal, with their recent success in the RTS game Dota 2 [122].

3.5 Conclusion and Future Work

I have presented the simplified RTS game Planet Wars, which focuses on high-level spatial control by removing many other classical RTS elements. In addition, I have analyzed how both expert-encoded hierarchical abstractions and reinforcement learning agents perform in the domain, and how capable of learning generalizable knowledge the latter are in particular.

Moving forward with this work, one of the key things to work on is attempting to learn hierarchical abstractions, rather than relying on knowledge encoded by a human whose expertise in the domain is questionable (namely, myself). The existing reactive planning agents are of a moderate complexity, with the obvious first-level reasoning decisions encoded, like defending against attacks and supporting one's own attacks, which results in agents with roughly 10-30 behaviors, depending on their optimization level. However, this work was completed prior to the later work on learning hierarchical structures that will be described in the upcoming chapters, and revisiting it with that in mind should be able to yield results. The main challenge in this application will be a lack of human demonstrations, as there is not a human professional scene for Planet Wars. Two potential solutions for that are: using high-level AIs to generate demonstrations, or developing a method for learning useful hierarchical abstractions from self-play, which would also be a valuable result.

Second, while I discuss Planet Wars as an approximation of strategic level spatial reasoning in SC:BW, the knowledge and abstractions learned here have yet to be integrated with an agent in the more complex game. As I will discuss in Chapter 7, I think that this integration has the potential to improve the functionality of multiple components at once, by removing the noise introduced by ignoring some aspect or another of a domain. This is a medium to high priority for the greater work presented in this dissertation moving forward.

Chapter 4

Pattern Mining

The previous chapter deals with learning in a spatial RTS game where the main focus is on unit/fleet positioning and area control, with a brief treatment of how this and other planning abstractions may be applicable to the components of SC:BW that mirror those, namely army positioning and region control. However, the primary focus of this work is on learning hierarchical planning abstractions. As discussed in the introduction, hierarchical planning is a very powerful model for creating intelligent agents that has been applied in many different settings. In this chapter, I will discuss my initial work on learning hierarchical planning abstractions, which will set the stage for the following chapters.

When considering learning from demonstration at a high level, I have found it useful to conceptualize an 'explanatory' dimension along which to measure various approaches. What this dimension represents is the amount to which a given learning method attempts to learn the reasoning behind given decisions, as compared to simply duplicating the behaviors observed. For example, on one extreme would be pure memorization learning, where an agent simply keeps a lookup table of every state observed in demonstrations, then acts by finding the closest example state to its current state and mirroring the action taken by the demonstrator.

Moving away from that extreme, memorization is relaxed into more intelligent applications of expert databases, such as case-based reasoning [118, 157, 58]. In these approaches, while the demonstration library is still retained as a database for choosing actions, transformations are applied based on the current state that attempt to account for whatever difference there may be between the agent's state and the example state from the demonstration. On the other extreme, an agent may attempt to create a full explanatory model that is able to reason out every action taken in its examples based on its domain and planning abstractions. In this case, an agent would, at the end of the learning process, discard the demonstrations entirely and rely purely on the abstractions that it has learned to operate in the domain. In some ways, this concept can be seen as a mirror to the discriminative vs. generative model distinction in machine learning, in that the first simply attempts to learn decision boundaries for classification problems, while the second attempts to learn the underlying model from which the examples are being generated.

In general, my goal is to create a system capable of learning to explain human demonstrations in a more generative fashion. That is, to learn an explanatory model that is able to forget its training examples entirely and still function intelligently. This can also be viewed as an eager learning approach, as compared to the lazy learning of case-based approaches. However, it can be difficult to build a model capable of explaining human demonstrations without some starting point from which to improve. In part, this is due to the 'noise' present. In a domain like SC:BW where many simultaneous plans and goals are being followed at once, professional players frequently describe their reasoning processes as either re-planning constantly, or leaving plans open, or even simply having slightly different orderings of actions while following identical plans based on personal

preference. Many of the logic-based deterministic learning approaches developed previously for hierarchical planners [49, 78] degrade to case memorization when presented with human demonstrations, due to the slight differences in ordering and actions chosen by players. As a result, I decided to investigate the applicability of pattern matching for finding low-level goals, primarily for its flexibility in the face of extraneous elements.

The premise for this work is the following: frequently repeated sets of actions imply some higher-level goal being achieved. Intuitively, this assumption makes sense, even at the atomic level. In an environment where every action has some cost, even if it is only the time spent giving the command, if a specific sequence of actions is performed, it implies that there is some goal being achieved by those actions. As a concrete example in SC:BW, if players using a particular strategy always attack after creating their third military unit, they are likely pursuing some goal to disrupt the opponent's early strategy. The purpose of the work presented here is simultaneously to test that premise in a wider manner to verify that it is true, and to see if it can allow us to start building hierarchical planning abstractions.

In this chapter I will discuss work dedicated to learning the first level of abstractions, purely groups of concrete actions in the domain. In Chapter 5 I will discuss scaling this work up to learn full planning structures. The work presented here was published in [84].

4.1 Related Work

The prior work most similar to this chapter algorithmically was presented in [13], which also used sequential data mining to analyze RTS games, in this case StarCraft 2, the sequel to SC:BW. However, their work focused on the extraction

itself, with some additional analysis of high-level strategy success/failure rates, with an eye towards game balance. I believe that the approach has much more potential than this, in particular for learning strategy abstractions themselves.

One of the inspirations for this work is HTN-MAKER [49]. This system learns hierarchical planning methods from observation of expert demonstrations, which is the stated goal of this dissertation. However, it has a tendency to create large method databases even in simple domains, something that will explode when transposed to the complexity of SC:BW. Additionally, it uses a more logical approach, which is less appropriate when working with human demonstrations that are likely to contain errors and/or extraneous actions. Some extensions to HTN-MAKER, HTN-MAKERND and Q-MAKER [46, 47] propose approaches to dealing with imperfect signals, both from the domain and from the demonstrations, but both still struggle with scaling up. Yang et al. address the issue of noise using an EM clustering of primitive actions to abstract tasks to incrementally build up a hierarchy of methods [167], but must use a total-order assumption in their assignment of actions to tasks that does not hold in human SC:BW gameplay. I will discuss this more in Chapter 5, where I remove that assumption to extend that work.

More generally related to the motivation for this work, there has been some amount of work on both hierarchical planning in real-time games and also learning from unlabeled demonstrations. Hoang et. al used hierarchical task network (HTN) representations in the first person shooter game Unreal Tournament to good success, merging event-driven agents with the higher-level planning to achieve both reactiveness and strategy [45]. Another great success of HTNs in games was Bridge Baron 8, which won the 1997 computer bridge championship [142]. While not real-time, its management of the imperfect information aspect

of the game is highly relevant to the RTS genre.

The end goal of this work is to learn hierarchical abstractions based entirely off of expert play, while prior work on learning from demonstration in the RTS domain has mostly focused on working from case libraries. When viewing this as a distinction between eager and lazy learning approaches, most fall into the lazy learning category. Weber et al. implemented a goal-driven autonomy system and extended it to use a case library from expert replays for detecting discrepancies and creating goals in [161]. Additionally, Ontañón et al. used case-based planning to implement a successful Wargus agent based on demonstrations of a human player executing various strategies [118]. While more supervised in that the demonstrations provided had partial labeling by the players themselves, it demonstrated that it was possible to learn abstractions if an expert provides a reduced but sufficient set of information. This system has been extended in a number of ways, including towards automated plan extraction from human demonstrations in [119], in which the authors use plan dependency graphs to map actions to goals, though it still requires some amount of goal encoding from a human moderator.

4.2 Generalized Sequential Patterns

Generalized Sequential Patterns (GSP) is a sequential pattern mining algorithm developed by Srikant et al. in [143]. One of its key strengths is in the flexibility that it affords the searcher in placing restrictions on the types of patterns to be searched for. In particular, it introduced the notion of a maximum or minimum gap between elements in the pattern, which places a hard limit on how separated consecutive elements in a pattern are allowed to (or must) be. This was useful for the purposes of this work, as I intended to search for short-term patterns to identify actions that are linked together in expert play. Without this

gap limitation, the search might identify "Build Barracks, Train Worker, Train Tank" as a common pattern, since it would appear in nearly every Terran game (with other actions in between), while the actions themselves are not necessarily directly linked in the player's mind. Another capability offered by GSP is user-defined taxonomies, with support for patterns that include items from different levels of the tree. While I have not yet included this aspect, it may be valuable in the future.

GSP works by performing a series of scans over the data-sequences, each time searching for frequent patterns one element longer than the prior scan. Given a set of frequent n-length patterns, it constructs a *candidate* set of $n + 1$-length patterns by searching for overlapping patterns within the frequent set (that is, a pair of patterns where the last $n - 1$ elements in one matches the first $n - 1$ elements in the other). It stitches these together to create a $n + 1$-length pattern for the candidate set. It then searches for each candidate in each sequence, to determine the amount of support and whether to add it as a frequent pattern. This approach is guaranteed to generate all frequent patterns (due to the fact that frequent patterns must be made up of frequent patterns), and in practice greatly reduces extraneous searching.

A replay of SC:BW can be seen as two sequences of actions, one performed by each player. However, if one looks at the full actions, one will find no overlapping patterns between games, due to the ever-present RTS problem of action-space size. Two players may move two units to minutely different locations, and these actions will not match up in a pure pattern match. As a result, we must blur our vision to some degree to find meaningful patterns. For this work, I zoomed far out, removing location information entirely from actions. Some example commands from the resultant sequences would be 'Train(Worker)', 'Build(Barracks)',

or ‘Move(Tank)’. The last is the main weakness of our abstraction, and our highest priority moving forward with this work is to reintroduce locality information via high-level regions. Even so, the patterns that we extract are meaningful starting points for learning goals and tasks.

To demonstrate both the GSP algorithm and our processing of replays, consider Fig. 4.1. Imagine that the maximum acceptable gap between pattern elements has been set at 4 seconds, and I require support from every trace to consider a pattern frequent. The initial pass will mark “Move(Probe)", “Train(Probe)", and “AttackMove(Zealot)" as frequent 1-element patterns, as they all appear in each trace. Then, every combination of these patterns will be generated as a candidate 2-element pattern, of which only “Move(Probe), Move(Probe)", “Move(Probe), Train(Probe)", and “Train(Probe), AttackMove(Zealot)" will be supported by all 3 traces. The only 3-element candidates generated are then “Move(Probe), Move(Probe), Train(Probe)" and “Move(Probe), Train(Probe), AttackMove(Zealot)", as any other 3-element pattern would have a non-frequent sub-pattern, and thus can be guaranteed to be non-frequent itself.

Of the candidates, “Move(Probe), Train(Probe), AttackMove(Zealot)" does not find support, as it cannot be satisfied in Trace 3 without using a pattern with elements more than 4 seconds apart. Therefore, it adds “Move(Probe), Move(Probe), Train(Probe)" to the frequent list and terminates, as it cannot generate any 4-element candidates. Based on the hypothesis stated above, these common patterns, which have shown up in close temporal proximity, would be promising candidates for low-level goals in a hierarchical planning structure.

Second	Action	
161	Move(Probe)	**Trace 1**
162	Move(Probe)	
164	Move(Probe)	
166	Train(Probe)	
167	AttackMove(Zealot)	
168	AttackMove(Dragoon)	
388	Move(Probe)	**Trace 2**
389	Build(Gateway)	
391	Train(Probe)	
394	AttackMove(Zealot)	
402	Move(Probe)	
403	Move(Probe)	
407	Train(Probe)	
222	Move(Probe)	**Trace 3**
223	Move(Probe)	
224	Move(Probe)	
225	Move(Probe)	
226	Train(Probe)	
239	Train(Probe)	
240	AttackMove(Zealot)	
243	AttackMove(Dragoon)	
244	AttackMove(Zealot)	

Figure 4.1: Snippets from three replay traces that have been preprocessed into our system's format. A Probe is a worker unit, Zealots and Dragoons are military units, and a Gateway is a training structure.

4.2.1 Implementation

I extracted the action sequences using the fan-developed program BWChart[1]. Once the sequences were extracted from replays and preprocessed to the level of abstraction described above, I then ran the actual GSP algorithm on them.

For this system, I used the open-source data mining library SPMF[2], which includes an implementation of GSP. However, the existing implementation is designed for larger databases of smaller traces, on the order of 10-20 elements. The traces in the SC:BW library, once cleaned of extraneous clicks, are in the thousands to tens of thousands range. As a result, the recursive implementation in SPMF ran out of memory. I have re-implemented an iterative version that does not suffer from the same problems, though it relies on a slightly tighter set of assumptions regarding the input data. This is functional in all but the more extreme edge cases, which are guaranteed not to arise in this domain.

4.3 Experiments and Results

For my experiments, I used 500 professional replays downloaded from the website TeamLiquid[3]. I focused on the Terran vs. Protoss matchup for the analysis, though it can be extended to the other 5 matchups as well. After the initial round of testing, I ended up with two main lines of inquiry: micro- and macro-level patterns. In the former, I ran the system as described above, with maximum gaps of 1-4 seconds, to search for actions that human players tend to chain together one immediately after the other. In the latter, I attempted to look for higher level goals and plans by removing the unit training and movement actions, leaving only

[1]available at http://bwchart.teamliquid.net/
[2]http://www.philippe-fournier-viger.com/spmf/
[3]http://www.teamliquid.net

the higher level strategy-directing actions: constructing buildings and researching upgrades.

One thing to note is that it would be preferable to use a larger number of replays to attain even more confidence in the mined patterns, but I was restricted by system limitations. Because the GSP algorithm needs to loop through every sequence for each pattern to see if support exists, it ends up storing all sequences in memory. For StarCraft:Brood War traces, with thousands of actions, this fills up memory rather quickly. The most prevalent sequence mining application is purchase histories, which are much shorter, and therefore the algorithm implementations are generally more geared towards that problem type, as mentioned in the implementation discussion above. That being said, a possible extension to this work would be to use a batch approach, where candidate patterns are generated per batch, then tested over the whole suite to determine if they are truly supported or not.

4.3.1 Micro-level Patterns

One type of pattern that I investigated was sequences of actions separated by small amounts of time, which I term 'micro-level patterns'. These are actions that occur frequently and immediately in sequence, thereby indicating that they are linked to each other and in pursuit of the same goal.

In order to find these patterns, I ran the GSP algorithm allowing gaps between actions of 1, 2, and 4 seconds. A standard professional player will have an APM (actions per minute) in the 200-300 range, which averages to 4-5 actions per second. In the end, there was not a qualitative difference between the results for any of these gaps, so all results shown here are using a 1 second maximum gap.

Upon examination, the mined micro-level patterns fell into three main classes:

action spamming, army movement, and production cycles, examples of which are shown in Figure 4.2.

Action Spamming

Action spamming is the habit of performing unnecessary and null operator actions purely for the sake of performing them. It is a technique often used by professional players at the beginning of a game when there are not enough units to tax their abilities, in order to warm up for the later stages of the game when they will need to be acting quickly. For the most part, these commands consist of issuing move orders to worker units that simply reinforce their current order. Since the habit is so prevalent, it is unsurprising that we find these patterns, although they are not particularly useful. If in the future their existence becomes problematic, we should be able to address the problem by eliminating null-operation actions.

Army Movement

Another category of extended pattern that is frequent in the data set is that of army movement. In general, there are two main types of army movements. Due to limitations imposed by the game mechanics, players can only issue order to a maximum of 12 units at once. As a result, when moving around large numbers of units, the players must select a portion, issue an order, select another group, and repeat until all units have been ordered to move to the new location. The second is mid-battle army control, when a player is positioning and re-positioning their army in order to maximize damage output and protect damaged units.

This type of pattern is more in line with what this work hopes to find, as the movement of one military unit followed by another is very likely to be two primitive

actions in pursuit of the same goal. Ideally, one could even discern between general army movement and combat movement, as there are different abstractions to learn for each. Unfortunately, actually identifying the goals pursued would require more processing of the data, due to the loss of location information from our representation abstraction. This is the kind of pattern that suffers most from the location abstractions, and it will require reintroducing the location data in the future, perhaps at different levels of abstraction, to learn more usable patterns for army movement.

Production Cycles

The final micro-level pattern that shows up in the data is what I term 'production cycles'. Professional players tend to sync up their production buildings in order to reissue training commands at the same time. For example, if a Protoss player has 4 Gateways, he will likely time their training to finish at roughly the same time, so that he can queue up 4 more units at once, requiring less time for mentally switching between his base and his army. This is reflected in the patterns GSP finds, as these Train commands tend to follow immediately after one another. This is another example of a promising grouping of primitive actions that could be translated into a low-level abstraction in a hierarchical planning model, after preconditions and postconditions had been learned.

4.3.2 Macro-level Patterns

In the opening stages of SC:BW, there is limited interaction and information flow between players. As a result, a relatively small number of fixed strategies have been settled upon as accepted for the first few minutes of play. These are commonly referred to as 'build orders', and they are generally an ordained order

Action Spamming

1: Move(Probe)
2: Move(Probe)
3: Move(Probe)
4: Move(Probe)
5: Train(Probe)
6: Move(Probe)
7: Move(Probe)

Army Movement

1: AttackMove(Zealot)
2: AttackMove(Zealot)
3: AttackMove(Zealot)
4: AttackMove(Dragoon)
5: AttackMove(Dragoon)
6: AttackMove(Dragoon)
7: AttackMove(Dragoon)

Production Cycle

1: Train(Dragoon)
2: Train(Dragoon)
3: Train(Dragoon)
4: Train(Dragoon)

Figure 4.2: A sample of frequent patterns generated by the system. The maximum gap between subsequent actions is 1 in-game second.

of constructing tech buildings and researches. How long players remain in these build orders, similar to chess, is dependent upon the choice of each, and whether either player manages to disrupt the other's build with military aggression.

In order to search for high-level goals, of which build orders are the most stable example, I removed unit training and movement actions from our traces and expanded the amount of time allowed between actions to 60 seconds. With these modifications, the GSP algorithm ends up with two main types of patterns.

The first are simple chains of production structures and supply structures. Players in SC:BW must construct supply structures in order to support new units. As a result, once the economy of a player is up and running, construction comes down to increasing training capacity and building supply structures to support additional military units. These patterns will translate well to long-term high-level goals of building up infrastructure.

The second type of pattern aligns with the more strategic-level abstractions discussed above, build order patterns. These are long chains of specific training, tech, and supply structures in a particular order. Figure 4.3 shows two examples of these that have been identified by the GSP system. In order to verify these results, I compared them with the fan-moderated wiki at TeamLiquid, and found that each of the early game patterns generated by the system was posted as a well-known and feasible build order. Note that this is only an evaluation of the precision of the discoveries. It is difficult to evaluate recall, since the community has recorded a very large number of openings, even some that aren't used in professional play, with slight changes sometimes being labeled as a different opening and sometimes simply as variations. However, as an estimate, the pattern matching algorithm finds 3 Protoss openings out of roughly 10 used in professional play, and 4 Terran openings, also out of approximately 10. As one would expect, GSP finds the

Build Orders

1: Build(SupplyDepot)
2: Build(Barracks)
3: Build(Refinery)
4: Build(SupplyDepot)
5: Build(Factory)
6: AddOn(MachineShop)

1: Build(Pylon)
2: Build(Gateway)
3: Build(Assimilator)
4: Build(CyberneticsCore)
5: Build(Pylon)
6: Upgrade(DragoonRange)
7: Build(Pylon)

Figure 4.3: Two build orders generated by our system. According to TeamLiquid, the first is a Siege Expand, "one of the oldest and most reliable openings for Terran against Protoss", while the second is a One Gate Cybernetics Core, which "can be used to transition into any kind of mid game style".

most common openings, but does not identify those that are less frequently used. Even so, these patterns are the strongest of the ones found, and the most easily translated into high-level goals.

4.4 Conclusion and Future Work

The initial question that I wished to investigate with this work was whether pattern mining algorithms can be used to find promising candidates for hierarchical goal structures. In particular, I wanted a starting point to improve on, other than random actions.

While this claim is not definitively proven until a full model can be learned, this work indicates that, at least at the initial level of abstraction, GSP is identifying useful structures. The patterns extracted align with abstractions that human experts use when describing how to play SC:BW,

In Chapter 5, I will discuss the ways in which this work has been extended. This research is an initial step in determining the viability of the pattern mining approach, while there I will show its ability to be extended to learning a full structural framework for a hierarchical planning model.

Even so, there are other direct extensions of this work that remain open to future research. First, I would like to reduce the location abstraction that is occurring, perhaps in conjunction with the work discussed previously in Chapter 3. Planet Wars can be viewed as a rough abstraction for strategic-level spatial reasoning in SC:BW, and some knowledge transfer may be possible between hierarchical planning agents. A build order can be drastically different based on where on the map it is occurring, an example being a military training building constructed near the opponent's base early in the game to rush an attack, as compared to one built at home to provide units for defense. A second extension would be to take advantage of the taxonomy system included in the original GSP algorithm. This would allow for multiple levels of abstraction in the various units and structures, possibly identifying more general patterns. An example of this could be identifying the pattern of constructing a military building, then training basic units out of it. Currently this must be learned for each separate building, but with taxonomies generalization could occur.

Additionally, there is existing research that indicates that observational learning can perhaps be improved by mixing both successful actions with mistakes [131]. In these circumstances, an agent uses the negative information from mistakes to contrast with the positive information found in successful examples. For this domain, that may mean comparing the prevalence of patterns from winning examples to losing examples. While I am using games that include both wins and losses, I am not explicitly treating them differently. This is certainly an area where

additional value could be gleaned in the future, although it will be important to not be caught up in the correlation vs. causation blunder, and identify behaviors that arise *because* a player is winning as things that *cause* a player to win.

Chapter 5

Probabilistic Learning

In the previous chapter, I discuss using pattern mining to identify the lowest level of abstractions for a hierarchical planning system, groups of concrete actions. However, in order for a hierarchical planning system to live up to its name, there must actually be a hierarchical structure underlying it. That is, it must be able to decompose high-level goals into sub-goals that can be further decomposed, until eventually reaching concrete actions in the domain. Learning this decompositional structure has been one of the primary challenges for learning hierarchical planning abstractions, as compared to learning applicability conditions [54, 52].

Prior approaches to learning hierarchical planning abstractions, particularly for hierarchical task networks (HTNs), have mainly focused on learning from labeled or annotated demonstrations when attempting to learn decomposition structures. This is a strong approach, since it reduces the amount of effort required from a human expert, allowing them to provide high-level commentary (in a programmatic format) on what goals they were achieving with each of their actions, rather than needing to fully encode their reasoning into abstractions themselves. However, it still has the drawback of being difficult to scale. Annotating thousands or tens of thousands of demonstrations is very time-consuming for a human, and

if one attempts to alleviate the issue by bringing in multiple experts, it introduces the risk of non-identical mental models adding noise to the annotations.

However, I believe that there is inspiration that can be drawn from this work when considering the task of learning from raw non-annotated demonstrations. In particular, the concept of learning abstractions *given* a set of annotations gives rise to the question: is it possible to generate that set of annotations? And is it possible to use the learned abstractions to then return and generate an improved set of annotations? This loop of generating predictions using some model, then using the predictions themselves to improve the model, and iterating, is the expectation-maximization paradigm, and is a common technique for machine learning with unlabeled data.

In this chapter, I will present work on developing an expectation-maximization algorithm for hypothesizing hierarchical planning structures, as well as extending the pattern mining approach discussed previously into a full structure learning algorithm. For the development process, I used a synthetic domain, and will present results showing the effectiveness of both of these algorithms in learning to accurately model an artificial oracle that is generating traces for them. This chapter is based on work that I published in [85] In Chapter 6, I will discuss how these approaches have been applied to learning abstractions in SC:BW, where they will need to learn from noisy human demonstrations instead.

5.1 Related Work

Yang et al. present an expectation-maximization approach to method learning that introduces the HTN Task Cluster Model (HTCM), modeling hierarchical task network tasks as Markov chains [167]. This allows them to calculate probabilities of chains belonging to a given task, with the additional assumption that the input

includes which tasks are achieved by which traces. This work was the primary algorithmic inspiration for this research, and its assumptions and how those limit the applicability will be discussed below.

Li et al. uses a greedy approach to structure hypothesizing while attempting to learn probabilistic HTNs to simulate user preferences [88]. This took the most common pair of actions across all traces and replaced it with an abstract task, and repeated, with a special case check for recursive structures. This generates a grammar in Chomsky Normal Form, allowing them to use pCFG learning techniques for learning method expansion probabilities.

HTN-MAKER is another example of a system for learning HTNs from demonstration [48, 49]. While it does not assume the hierarchical structure of traces as given, it does take the abstract tasks of the domain, in the form of pre- and post-conditions, from which it learns method decompositions to achieve these tasks by searching for sequences of actions in demonstrations that achieve these tasks. [115] approaches a similar problem of learning HTNs from demonstration without hierarchical structure, but assume that some information about the final goal is provided, from which they construct their learned methods incrementally.

Within the greater cognitive architecture research field, the major architectures– which are not explicitly utilizing HTNs, but are using other hierarchical planning processes–have also addressed the issue of learning hierarchical structures for whichever model they are utilizing. [73] and [152] both explore learning from experts in the context of the Soar architecture. However, it is a more interactive process than the problem statement that I use here, with the expert annotating traces generated by the planner, or analyzing and annotating the agent's self-annotations. Within the context of the ICARUS architecture [77] [23], researchers have confronted a very similar problem to this in [115] and [89]. They address

the model explosion problem by differentiating between *skills* and *concepts*, making the distinction between actions that are performed for their immediate effect on the next action and actions that are part of some long-term goal, but they still rely rather heavily on the appropriate choice of concrete actions, using backchaining from the end state to build structure. As a result, these approaches are less applicable to human demonstrations.

Finally, there are a number of other systems that perform HTN learning, but most presume structure as a given, rather than including it in the learning problem. Garland et al. in [37] prove soundness and completeness for learning bindings, parameters, and constraints, given that the hierarchical decomposition for all traces is provided as input. In [168], Zhuo et al. approach the problem of dealing with partial state observations when learning preconditions and an action model, but also assume that decomposition tree information is given.

5.2 Hierarchical Task Networks

While I have been referring to hierarchical planning in general thus far, this work uses a specific formulation of hierarchical planning named Hierarchical Task Networks (HTNs). I will provide a brief overview of HTNs in order to acquaint the reader with them and make clear the language that I will be using in this chapter, but for a more thorough explanation the reader is encouraged to refer to their treatment in any number of respected textbooks, e.g. [40].

Terminology: An HTN *domain* $\mathcal{H}$ consists of a 3-tuple: $\mathcal{H} = \langle \mathcal{A}, \mathcal{T}, \mathcal{M} \rangle$. $\mathcal{A}$ is the set of *primitive actions* in the domain, $\mathcal{T}$ is the set of abstract *tasks*, and $\mathcal{M}$ is the set of *methods* that achieve the elements of $\mathcal{T}$. A method has four components: a unique identifier, which abstract task it achieves, a decomposition into subtasks, and a set of constraints, which can consist of preconditions, postconditions, and

conditions that must hold throughout the execution of the method. An HTN *planning problem*, then, consists of an initial state, a set of tasks to be achieved, and an HTN domain. It is then the goal of the HTN planner to find a valid plan that achieves all of the tasks using the decompositions provided in the domain methods, while maintaining all constraints imposed by those methods. Learning these decompositions is the target of this work.

5.3 Algorithm Descriptions

Baseline

The baseline algorithm that I used for this project is the Structure Hypothesizer algorithm from [88], which I will summarize briefly for the reader.

Given a library of plans for achieving some high-level task, first search for evidence of recursive expansion rules. This evidence is of the form of long chains of a repeated action, followed or preceded by some other action. For example, if the pattern $aa \ldots ab$ occurs frequently in the library, it would imply the recursive rule $B \rightarrow aB; B \rightarrow b$. If the frequency and average length of a recursive pattern are both above some user-defined threshold, add the recursive expansion to the HTN method library and replace each instance of it with a new abstract symbol.

If no recursive pattern exceeds the thresholds, find the most frequent pair of adjacent actions in the plan library. Add a task to the HTN that has a single expansion corresponding to this pair of actions, and replace all instances of the pair in the plan library with the newly added task.

This process is repeated until all traces in the plan library have been compressed to a single task, the original high-level task. Since each step through the algorithm will compress at least one demonstration in the library, and both the

library and the demonstrations contained within are finite size, the algorithm is guaranteed to terminate.

Pattern Mining Algorithm

The second approach I will use for hypothesizing structure is a generalization of the greedy approaches used in Chapter 4. The underlying assumption is that subsequences of actions that are both common and closely linked in the time domain are the best candidates for HTN methods. When taken to the extreme, this results in taking the most frequently occurring pair of actions as a task method, replacing them with an abstract task, and repeating until all traces have condensed to a single task, similar to the baseline algorithm. However, this approach is not robust when presented with traces from partially-ordered plans. In particular, it may require an exponential number of pairs to explain plans in which tasks are overlapped. For example, consider what happens if we have two methods consisting of the actions ab and cd, and the following three demonstrations exist: $\{a, b, c, d\}$, $\{a, c, b, d\}$, and $\{a, c, d, b\}$. cd will be identified as a common pattern, but ab will not, and we will have to construct a far more complicated structure without it.

My solution to this problem is to use the Generalized Sequential Pattern (GSP) algorithm developed by [143], in replacement of simple pair-matching. GSP is a pattern mining algorithm that takes a database of traces and searches for common subsequences. These common subsequences act as the HTN method candidates, and I can replace them with an abstract task in L and iterate to find higher-level tasks (lines 7-10 of Algorithm 1). In addition, it includes two parameters that will be useful to us:

- Minimum Support - The minimum amount of traces a subsequence must

appear in to be considered a frequent pattern.

- Maximum Gap - The maximum gap allowed between elements of a subsequence. If 0, the subsequence must be contiguous. If the length of the trace, any subsequence is allowed.

By relaxing these two parameters (line 13), we can identify additional subsequences, and consequently more HTN method candidates. The evaluation in Section 5.4 tests two different approaches to this relaxation strategy. This iterative relaxation is the extension of the work presented in Chapter 4 that allows the algorithm to move from simply identifying the lowest-level patterns present in the data to constructing a full hierarchical structure.

Algorithm 1: GSP Structure Hypothesizer algorithm

Input : L : Library of action traces for the same top-level task
S : Minimum Support (initial)
G : Maximum Gap (initial)
Output: T : A set of HTN tasks
M : A library of method expansions for T

```
1  T ← ∅
2  M ← ∅
3  while |L| > 0 do
4  |  while GSP(L, S, G) ≠ ∅ do
5  |  |  patterns ← GSP(L, S, G)
6  |  |  foreach p ∈ patterns do
7  |  |  |  t ← nextAbstractSymbol()
8  |  |  |  T ← T + {t}
9  |  |  |  M ← M + {(t, p)}
10 |  |  |  L ← replaceTask(p, t)
11 |  |  end
12 |  end
13 |  Relax S or G
14 end
```

Algorithm 2: EM Structure Hypothesizer algorithm

Input : L : library of action traces for the same top-level task
Output: T : A set of HTN tasks
M : A library of method expansions for T

$T \leftarrow \emptyset$
$M \leftarrow \emptyset$
while $|L| > 0$ **do**
  Initialize latent tasks Y randomly
  while Y *not converged* **do**
    $taskSequences \leftarrow viterbiBacktrack(L, Y)$
    $Y \leftarrow updateTasks(taskSequences)$
  end
  $t_r \leftarrow lowestEntropy(Y)$
  $T \leftarrow T + \{t_r\}$
  $M \leftarrow M + generateMethods(t_r)$
  $L \leftarrow replaceTask(t_r)$
end

EM Algorithm

For my second algorithm, I begin by making the weak assumption that the decompositions for a given task in an HTN may be modeled as a Markov chain. This is valid for simple hierarchies—for example, where each task has only one method decomposition with non-overlapping subtasks—and it even allows for some complexities in expansions, allowing a probabilistic choice between multiple method expansions, so long as they do not overlap. On the other hand, while it may be possible to set up a Markov chain to model an HTN, this particular view of task decomposition is clearly not able to model the full complexity of HTN tasks. A simple demonstration of this would be two methods for a single task which each utilize the same subtask at some point in their expansion. It would be possible for the Markov chain to begin with an expansion in line with the first method, reach the shared subtask, and switch tracks from the first method to the second, even if this led to an invalid plan.

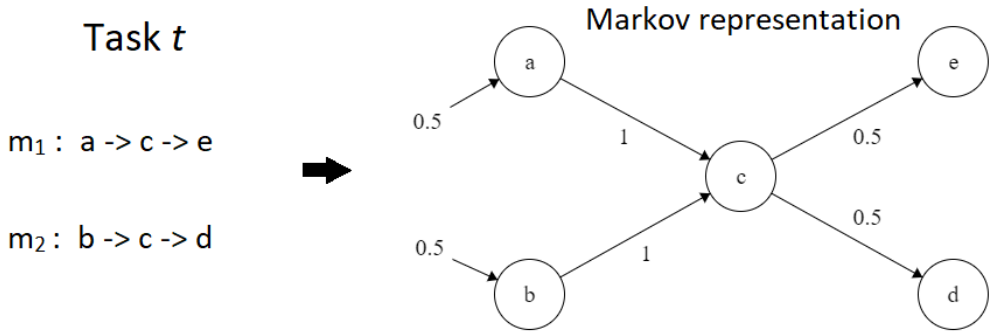

Figure 5.1: An example of the Markov representation of an HTN task. Note that this also demonstrates the representational risk of this approach, as discussed.

As an example, consider a task t that is achieved by two different methods. If the first method is comprised of the concrete actions $a \rightarrow c \rightarrow e$ and the second is $b \rightarrow c \rightarrow d$, the Markov chain formulation will have 5 states, (a, b, c, d, e). There is a 50/50 chance of entering at a or b, each of which will have a 100% chance of transitioning to c, which will have a 50/50 chance of transitioning to d or e. Note that in this formulation, it is possible to follow the $a \rightarrow c \rightarrow d$ or $b \rightarrow c \rightarrow e$, which were not valid methods for the original task. This is where the Markov assumption creates a gap in the representational ability of the new setup. Figure 5.1 shows this example graphically.

However, this assumption *does* make it possible to calculate the probability of decomposing a task T in a way that would generate a sequence of n actions l, given the prior and transition distributions Θ_T for that task:

$$P(l|\Theta_T) = P(l_0|\Theta_T) * \prod_{i=1}^{n} P(l_i|l_{i-1}, \Theta_T) \tag{5.1}$$

The original EM method-learning algorithm presented in [167] presumes that

all abstract tasks formulated in a given trace would be included, in order, with the input. Thus, the primary goal for the algorithm was determining break points in the chain of primitive actions. With these, the provided tasks could be overlaid on the appropriate subsequences and learned from the collection of appearances of that task in the trace library.

My relaxation of those input assumptions, in which the algorithm receives only the top-level task and the trace of primitive actions, eliminates the validity of this approach, due to the introduction of multiple overlapping goals being pursued simultaneously. In this problem setup, a pure partitioning of the primitive actions would be unable to group the required actions together, as others may be interleaved between them. Instead, I introduce a set of latent abstract tasks Y (line 4 of Algorithm 2), representing the HTN tasks that the top-level task decomposes into, but which are not explicitly provided for us. As mentioned, these are modeled as Markov chains, and so are represented with a prior and transition distribution ($P(a_0|\Theta_T)$ and $P(a_t|a_{t-1}, \Theta_T)$, respectively).

Lines 6 and 7 represent the Expectation-Maximization loop of the algorithm. In the terminology of EM, it first calculates the most likely explanation for which tasks generated which action sequences using the current model parameters (stored in $taskSequences$), then updates the model parameters based on the actions that have been assigned to each respective task.

To be more precise, I use a Viterbi-like dynamic programming algorithm to calculate the most probable assignment of primitive actions to tasks for each trace in the library. In the classic Viterbi algorithm, one calculates the probability of the most likely trace ending in a particular state and time step by calculating the probability of the most likely traces to each state in the time step prior, and taking the best of each of those when multiplied with the associated transition

probability.

A similar approach for this, if the goal is to assign actions in some trace l to tasks, rather than states, results in the following recursive definition:

$$OPT(i,t) = \max(OPT(i-1,t) * P(l_i|l_{i-1}, Y_t), \max_{u \in Y - \{t\}}(OPT(i-1,u) * P(l_i|Y_u))) \tag{5.2}$$

where $OPT(i,t)$ is the probability of the most likely assignment that assigns the i-th action to task t.

However, this definition will fail when given ground out traces of partially-ordered plans. Namely, it assumes that an action either belongs to the same task as the previous action, or is the start of a new task. This corresponds to the first-order Markov assumption in the Viterbi algorithm.

My solution is to introduce a parameter k that determines the maximum number of intermediate actions allowed before two actions may no longer be considered as part of the same task. The higher this parameter, the more capable the algorithm is of dealing with mixed action groundings. However, there is a run-time tradeoff as well, limiting k in practice.

This changes the formulation of the recursive definition to the following:

$$OPT(i,t,s) = \max_{u \in T}(OPT(i-1, s[-1], [u]+s[:-1]) * nextTaskProb(l_i, t, [u]+s[:-1]) \tag{5.3}$$

where

- s is a suffix of length k representing the task assignments of the previous k actions.
- $nextTaskProb$ is a function that takes an action a, a task t, and a suffix s, and calculates the probability of that action being assigned to that task

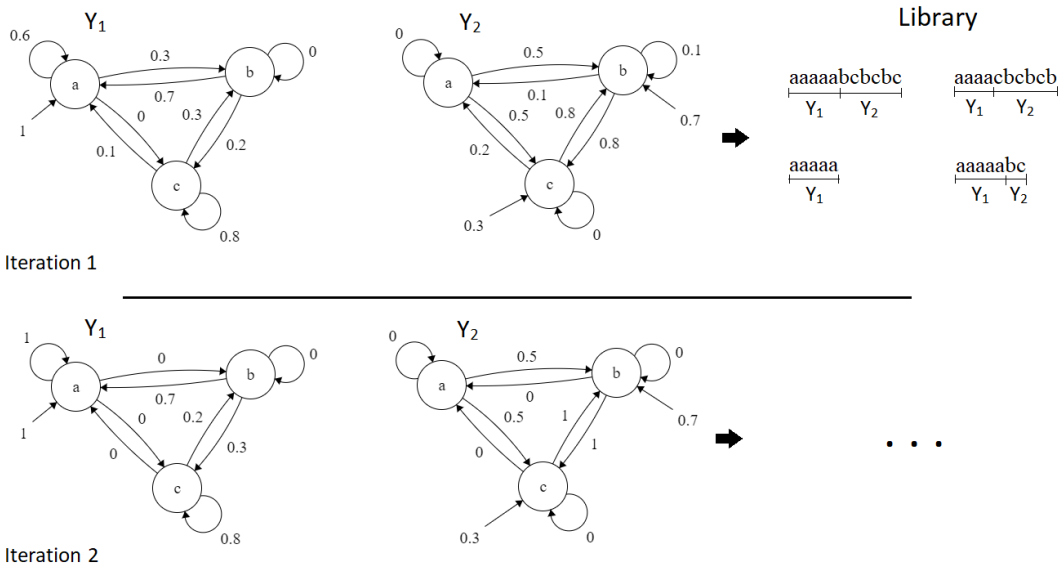

Figure 5.2: An example of the EM iteration process. In the first step, the latent tasks are initialized with random probabilities, which are then used to calculate the most probable assignment of concrete actions to tasks in the library. These are used to update the latent tasks for the next iteration, and so on.

given that it follows the suffix. This is $P(a|t)$ if none of the actions in the suffix are assigned to t, and otherwise is the transition probability from the latest t-assigned action to a.

Using standard dynamic programming backtracking techniques, I can store the assignments chosen up to each point while making these calculations. Using that stored information at the conclusion of the algorithm I can calculate the most probable assignment of actions to tasks for every trace. With these, I update the parameters of the model Y to maximize the probability of each task generating the action subsequences that it has been assigned according to Equation 1 (line 7 of Algorithm 2).

Figure 5.2 shows a small example of this EM loop in action. In the first iteration, the latent tasks Y are initialized randomly. Then, using the modified

Viterbi algorithm, the algorithm calculates an assignment of concrete actions to tasks for each trace in the library. In this case, Y_1 learns to represent the repetitive a pattern, while Y_2 learns to represent the alternating $bcbc$ pattern. In this simple example, the loop stabilizes after a single iteration, but in practice it frequently takes much longer, with less clear roles being taken on by each of the latent tasks.

Once the EM loop has converged to the point where assignments are unchanged from one step to the next (not guaranteed to occur, but has never failed to in practice), and the parameters of the model Y are no longer changing (or changes are below some threshold), I take the task model in Y with the lowest weighted entropy, representing the simplest learned model. In the extreme case of zero entropy, this corresponds to a sequence of actions that always occurs in a set order, which corresponds to a single-method task. I add this task to the task library, and generate a set of methods to achieve it, using one of two approaches. The first is to simply add every sequence assigned to this latent task in the final expectation step; that is, every instance of this task in the most probable assignment of actions to tasks (hereafter referred to as the Most Probable Explanation (MPE) approach). The second is finding all walks in the Markov chain with probability greater than some user-defined threshold Θ_m. The first approach has the advantage of only including sequences that actually appear in the database in at least one replay, but it can suffer from any extraneous sequences that this task has been forced to absorb due to limited latent tasks. The threshold approach is less likely to suffer from that noise, as it will likely fall below the threshold, but it runs the risk of composing portions of two common sequences into a new method, even if they don't appear together anywhere in the demonstrations. These method selection processes are how the EM algorithm is able to introduce method branching, as each task has the potential to end up with multiple methods, depending on the

final Markov chain. Finally, I replace the subsequences in L that were assigned to this task with an abstract symbol for the task, clean up L (adding any single-task traces as expansions for the top-level task), and reiterate.

As a note, I would like to state that this algorithm is a hard EM algorithm, using the most probable assignment as definitive labels when doing the maximization step. This is useful in reducing the run-time complexity of the calculations needed for the expectation step, at the cost of some accuracy in the maximization step. A soft EM approach may be feasible as well, calculating a distribution of assignments for each concrete action to each task, using an algorithm similar to the forward-backward algorithm. However, the complexity would increase, as it would need a posterior probability distribution for transitions, rather than simply assignments, and this is complicated by both the interleaving of assignments and the fact that multiple iterations of the same task can appear in each trace, requiring start and end estimations as well. For these reasons, I believe a hard EM algorithm is the correct approach.

5.4 Experiments and Results

5.4.1 Synthetic Domains

For this evaluation, I used randomly generated synthetic planning domains with a single high-level task, for two primary reasons. First, it provides an oracle planner that can be exhaustively searched to generate all valid plans. Second, it allows for running a suite of tests and therefore achieves stronger confidence in the results than if confined to a single domain. In Chapter 6, I will discuss how to apply both of these algorithms to SC:BW, in pursuit of the primary objective of this work.

Some domain characteristics for the randomly generated domains used in these evaluations are as follows:

- Depth of plan decomposition tree: Between 3 and 8. Median depth of 5.
- Method expansions per task: Randomly chosen between 1, 2 and 3
- Subtasks per method: Randomly chosen between 2 and 3

In truth, this only explores a small fraction of the potential space of HTN planning domains, and it is possible that different results would be obtained in domains with different characteristics. However, the synthetic domains are sufficiently complex to present a challenge to learning systems, and to inform the research moving forward.

Using these domains, I generated primitive action traces by sampling the plan space for partially-ordered plans and their groundings, using a simple probabilistic check ($p = 0.1$ in this evaluation) when expanding a task to decide whether or not to interleave its subtasks' actions when grounding. These formed a demonstration library of primitive action traces that could be fed to the algorithms for hypothesizing method structure.

The following two metrics are what I will use to evaluate the effectiveness of the hypothesized method structures:

- *Coverage* — A measurement of how well the hypothesized structures cover the valid plans of the oracle. The fraction of valid plans generated by the oracle that exist as a grounding of a partially-ordered plan in the learned structure.
- *Signal* — A measurement of how much noise is in the plans of the hypothesized structure. The fraction of partially-ordered plans generated by the learned structure that can be ground to a valid oracle plan.

Algorithm			Coverage	Signal
Baseline			0.491 ± 0.101	1.00 ± 0.00
EM MPE	α			
	1		0.662 ± 0.045	0.043 ± 0.03
	0.1		0.695 ± 0.055	0.027 ± 0.012
	0		0.662 ± 0.048	0.072 ± 0.017
EM Threshold	α	Θ_m		
	1	0.05	0.420 ± 0.082	0.662 ± 0.070
	1	0.01	0.390 ± 0.147	0.138 ± 0.049
	0	0.1	0.525 ± 0.052	0.450 ± 0.086
	0	0.05	0.620 ± 0.094	0.052 ± 0.024
	0	0.01	0.827 ± 0.083	0.024 ± 0.019
GSP			0.532 ± 0.090	0.983 ± 0.02

Table 5.1: Coverage and Signal for a range of algorithm configurations over 12 random domains, with a demonstration library of 200 plans. α is the additive smoothing parameter for Laplace estimation in Maximization-step updates, and Θ_m is the probability threshold for paths in the Markov chain to be added as methods.

Note that these correspond loosely to the precision and recall measures from classification, adjusted to fit this problem.

As the envisioned first step in a learning pipeline, it is clear that the first target is high coverage, with a secondary goal of high signal. This is because future learning steps will be pruning the method structure via learning applicability conditions, and in doing so therefore remove large numbers of potential plans that do not correspond to valid oracle plans. On the other hand, one can see that a simple greedy approach such as our baseline will maximize signal, as it only builds structure that can expand into exact traces that it has observed. However, this hurts its generalization ability in being able to generate unseen plans.

5.4.2 Method Selection for EM

Table 5.1 shows the average coverage and signal of a number of parameter configurations for the EM algorithm over 12 different randomly generated domains. The two different configurations refer to the approaches used to generate methods for the abstract task once EM has been run and the lowest-entropy task selected. MPE uses all action subsequences assigned to the task in the final Expectation step as methods, while Threshold uses all walks through the final Markov chain that are above a given probability Θ_m.

MPE has the advantage of having one less parameter to tune (α, which I will discuss in the following section), and appears to be consistently stronger than the baseline. The inherent risk of this approach is of a task's Markov chain acquiring responsibility for explaining some extraneous action sequences that match no task well during the EM process, and these being added as a method expansion while not reflecting the main task the representation has learned. However, this risk is mitigated, though not entirely removed, by choosing the lowest entropy task.

Thresholding removes this risk, as it will only take the major action paths that have been learned (though it retains the risk of a task learning to represent two non-colliding oracle tasks and thus merging them incorrectly). However, it performed much less consistently, responding drastically to changes in the threshold parameter. In a domain with no oracle to test against, such as SC:BW, it may be difficult to determine whether one is using the correct parameter values.

Note that if no walk exceeded the threshold probability, the highest probability walk is used as the only method expansion for that task. In the $\Theta_m = 0.1$ case, this behavior leads the algorithm to a structure much closer to the baseline's single method per task expansion, resulting in a higher signal, at the cost of a mediocre coverage.

A third possibility for generating task methods from a Markov chain is using Monte Carlo rollouts. In testing, this approach performed much worse than either of the prior two. Due to the nature of the EM algorithm, each latent task absorbs some amount of 'noise', whether it be true noise or the result of higher-level structures that can't currently be modeled. Choosing the lowest-entropy task means that if the rollout *does* happen to leave the common paths, it tends to meander among actions randomly until reaching a common chain again. This generates a very long and low-quality method.

5.4.3 Smoothing

During the Maximization step of the EM algorithm, it performs an update to each task's Markov chain representation based on the action sequences assigned to it by the previous Expectation step. I tested the performance of the algorithm with and without additive smoothing during this update. The parameter for this smoothing, α, determines how many 'pseudo-observations' of each action transition are added. When $\alpha = 0$, the update is a maximum likelihood update. If an action transition, say action a followed by action b, is not present in the subsequences assigned to a latent task for a given loop, that transition's probability will drop to 0 in the Markov chain. This means that this transition will never again be assigned to this task, no matter how many loops we iterate through. As a result, the latent tasks have a tendency to converge more rapidly and rigidly. When $\alpha = 1$, a full pseudo-observation of each action transition is included, a slightly more forgiving approach.

Table 5.1 shows that a small amount of smoothing was marginally (though not statistically significantly) beneficial to the MPE version of the algorithm, but it was severely detrimental to the coverage of the Thresholding configuration. While

a deeper investigation is required before drawing definitive conclusions, I believe the reason for this is that HTN tasks, once they have been expanded into methods, are in fact fairly rigid processes. As a result, leaving them flexible is an inaccurate representation of the underlying mechanism. One avenue for further exploration is to reduce α over the course of each meta-step of the EM algorithm, to allow flexibility in the beginning when latent tasks are finding their identity, and rigidity toward the end, when they are solidifying which action sequences they account for.

5.4.4 GSP

Two different procedures for relaxing the minimum support and maximum gap parameters of the GSP algorithm were tested in this synthetic domain. Using an approach similar to grid search, one parameter was slowly relaxed until a defined limit was reached, then tightened again and the other parameter relaxed by a single increment. Changing which parameter belonged to which 'dimension' of the search resulted in two different learned method structures.

Unfortunately, due to a lack of branching factor introduction to the hypothesized methods, each structure was only able to reproduce the traces provided to it, and so their metrics were identical (though the expansions to reproduce those traces were different). As a result, the GSP algorithms were only able to slightly outperform the baseline approach, likely due to their ability to deal with interleaved actions due to searching for subsequences rather than contiguous pairs. I will revisit the GSP structure hypothesizer algorithm in Chapter 6 when applying it to SC:BW.

5.4.5 Tradeoff: Coverage vs. Signal

A final observation that is made clear by Table 5.1 is the tradeoff between coverage and signal that each of these algorithms is making. Fundamentally, this makes sense. On the one hand, a basic algorithm could achieve perfect signal by simply memorizing and repeating each example it is shown, though this would have no generalization. Similarly, an algorithm could accept all possible action sequences as plans, which would provide full coverage with almost no signal. The goal is to use an algorithm that is able to generalize sufficiently from the examples that it is shown to achieve high coverage without sacrificing too much signal. The other factor to remember, as mentioned earlier, is that this is simply a first step in the abstraction learning process, and further steps will prune possible plans, but not add more. This is why the EM approach, which sacrifices more signal for coverage, is particularly interesting.

However, it is likely that the decision of which type of algorithm to use is very heavily domain- and demonstration-dependent. While I have not done empirical testing of a range of domain characteristics, based on the development of the algorithms, I would posit the following: In domains where tasks are 'wider', that is, where they can be achieved in multiple ways, I believe the EM style approaches are more appropriate, as they have a better affordance for modeling the multiple methods per task required. Meanwhile, in domains that are 'deeper', with longer tasks but which are more formulaic in their execution, GSP will likely perform better, as it doesn't have the issue of taking a probabilistic misstep on a long path, like the EM algorithm does.

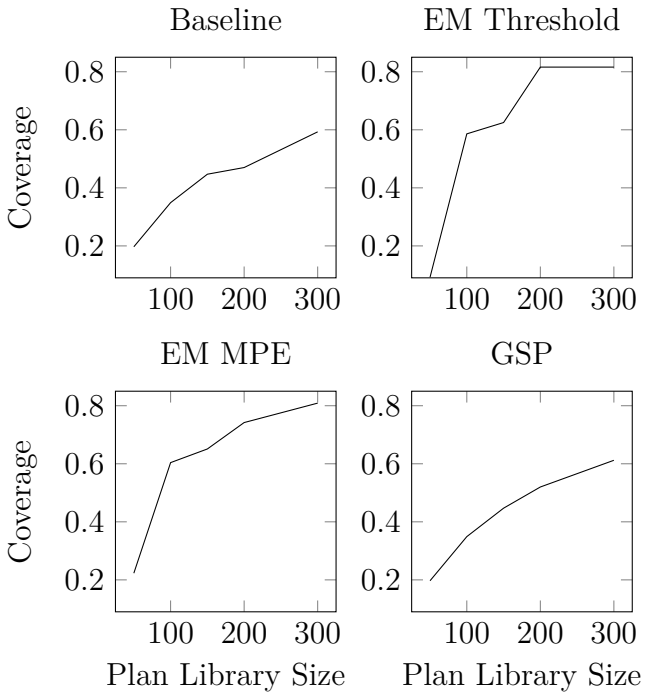

Figure 5.3: Change in coverage as the size of the provided plan library increases. The total number of valid oracle plans was 400.

5.4.6 Plan Library Size

Finally, Figure 5.3 shows the progression of coverage as the algorithms receive greater numbers of demonstration plans from the oracle. As expected, both the Baseline and GSP are relatively linear, with low generalization ability. On the other hand, the EM algorithms both improve very quickly, implying that they are learning substructures across demonstrations that can be used to generate the valid plans that have not yet been observed.

5.5 Conclusion and Future Work

Hypothesizing method structures for hierarchical abstractions is one of the primary challenges that a learning algorithm faces. In this chapter I have pre-

sented two approaches for learning these decompositional structures particularly for HTNs, though they can be adapted to other hierarchical planning systems. The first of these, a repeated pattern matching algorithm, slightly outperforms the greedy baseline, primarily as a result of its ability to handle parallel subgoals with interleaved actions. The second, an expectation-maximization algorithm that utilizes a weak Markov assumption of HTN tasks to iteratively improve its task definitions, shows an improvement in learning unseen plan decompositions via substructures.

There are a number of ways in which this work can be extended in the future. The first is to develop a more principled approach to the GSP parameter relaxation, one that is dependent on the example data available and the model being learned, rather than the basic grid search that I have described here. While it will not address the other weakness of the GSP algorithm, namely high signal and low coverage, it should reduce the amount of tuning required when moving from one domain to the next. The second, in a similar vein, is to remove the parameterization component of the EM algorithm that requires choosing an appropriate number of latent tasks to model. Not only is the correct number of tasks dependent on the domain, it is also dependent on how deep into the plan library construction the algorithm is. As a result, it would be useful to have a self-moderating system that can choose the appropriate number of latent tasks in an intelligent manner.

However, the final and most important note is that these algorithms do not in and of themselves learn a full hierarchical planning system, even if they are successful at learning the decompositional structure. In Chapter 6, I will discuss how to integrate them into a more fully developed learning pipeline to learn the strategic competencies of a SC:BW agent from raw replays, using a probabilistic

modification of an existing hierarchical planning system, the ABL programming language.

Chapter 6

pABL and Integration

In the previous chapter, I present two algorithms for hypothesizing method structures for a particular form of hierarchical planning abstractions, HTNs, from observation. However, there were a number of unresolved issues that needed to be addressed.

The first missing component was an evaluation on actual human data. While they were designed with the noise and simultaneous pursuits of multiple goals present in human demonstrations in mind, they were tested on an artificial domain that simply attempted to simulate these properties.

The second was the lack of integration in a full model-learning pipeline. In particular, the algorithms that I presented lacked any form of applicability condition learning (i.e. preconditions, postconditions, etc.). As a result, they ended up with plan libraries that generalized much too widely, assuming that any task or method could be applied at any time. On a related note, these algorithms both make only a single pass at the data, building up their plan library in a greedy fashion. While they each take their next choice with different computations, once they have committed to adding a task or method into their library, it is defined permanently. It would be desirable to have some way to analyze the quality of a

finished structure abstraction, in order to programmatically find ways to improve it.

In a traditional hierarchical planning system, both of these issues are difficult to address for primarily the same reason: the deterministic, first-order logic based operation of the planners. In particular, given a traditional HTN model, for example, and a human demonstration, the only evaluation possible is whether or not it was possible for the HTN planner to have generated the same plan as the human. This binary signal is both low in information, giving us only one 'bit' of feedback per demonstration, and also very sparse, since due to the noise and extraneous actions generally present in human examples, the planner will nearly always report that it would not be able to generate a human trace (with probability approaching 1 as the domain grows as complex as SC:BW). However, if one softens the constraints of the planning language, allowing for probabilistic choices rather than deterministic ones, it is possible to alleviate these issues.

In this chapter, I will present a probabilistic modification of the reactive planning language ABL that enables algorithms for measuring the explanatory quality of a given planning model when compared against a database of demonstrations. This will provide both a metric to use when learning models from observation and simultaneously a method for estimating assignments of concrete actions to goals and sub-goals, both of which will be used for learning model parameters from human demonstrations. Finally, I will evaluate the explanatory power of the hierarchical structure learning algorithms presented previously in Chapter 5 for SC:BW, before discussing the possible future directions for this work.

6.1 Related Work

Learning Hierarchical Abstractions

As mentioned in Chapter 5, there has been a good deal of work on learning the structure of hierarchical planning abstractions from demonstrations, though it typically either assumes some amount of annotation from a human expert or a lack of noise in the provided examples, or both. The HTN planning framework HTN-MAKER [49] and its extensions [46, 47] learn from scratch with human demonstrations, but struggle with model explosion in complex domains, while Garland et al. have developed provably sound and complete algorithms for learning HTN models, but assume the full decompositional tree structure is provided, learning instead which methods belong to matching tasks and applicability conditions [37, 36]. Unfortunately, this is an impractical restriction for a real-world domain such as SC:BW.

In the cognitive architecture field, where large developed systems typically use hierarchical planning as a key part of their reasoning process, learning from demonstrations has also been studied. Soar [76] has explored learning from demonstrations in [152] and [105], but use an interactive process involving a human expert annotating data for the agent. ICARUS [77] has a few different learning mechanisms, primarily using its own problem solving [23, 78], but it is heavily dependent on first-order logic and back-chaining and therefore will likely struggle with noisy human demonstrations.

Learning Non-structural Parameters

In addition to learning the structural and decompositional components of a hierarchical planning model, other parameters must be learned as well. The most

notable of these are the applicability conditions for methods, i.e. preconditions and postconditions, but others can include priorities for selecting between goals or even determining the effects of concrete actions themselves in an unspecified domain.

This is an intuitive place to start for learning, as the learner can consider all applications of a particular method and examine similarities and differences between the states where this method was applicable. One system developed for this task is CaMeL [54]. CaMeL, short for Candidate elimination Method Learner, learns preconditions via a candidate elimination algorithm. To explain how this works, the authors introduce the idea of a minimal vs. a maximal policy. In a minimal policy, a domain is accepted only if the evidence proves it. In a maximal policy, a domain is accepted so long as it is not contradicted by the evidence. CaMeL's candidate elimination algorithm keeps two version spaces, one the minimal set supported by the evidence seen so far, and another the maximal set. Given enough examples, these two spaces condense to a single point, which is the true precondition for the method. In an extension to this work, CaMeL++ [52] explored the possibility of planning before the version spaces had converged to a definite precondition. In order to deal with the uncertainty, a voting algorithm was used, and a threshold of votes required to consider a precondition as satisfied was set by hand. While these were more theoretical results, this work was implemented in a more practical manner by Kuter et al. in [75], in which they use these ideas to learn safe constraints in the airspace control domain, based on human demonstrations.

An Adaptive Behavior Language (AABL) is an extension of ABL that adds reinforcement learning into the behavior selection process [140]. This is similar to the approach of this work, but the intention was to facilitate partial programming,

allowing authors to ignore certain aspects of their ABL program, leaving them to be learned through simulations. In particular, it automatically learned a value function for each behavior from a pre-defined set, so an author could provide all means of decomposing a subgoal and allow the system to learn the order they should be chosen in after filtering for preconditions. I instead would like to learn these components through demonstrations, working towards human-readable interpretability for agents while still being able to learn the entirety of an agent's structure and parameters, and this change in fundamental goals leads to a novel set of modifications to the ABL language.

Other Related Work

Additionally, there is a plethora of work in the more general field of modifying traditionally deterministic systems to be probabilistic. Two of note are the Probabilistic Soft Logic [66] framework that uses hinge-loss Markov random fields to solve inference problems as soft first-order logic formulations, and is able to use this formulation to learn parameters of the model in much more scaled domains. FACTORIE [101] combines the statistical semantics of factor graphs with imperative definitions, giving the user freedom to mix declarative and procedural domain knowledge. Finally, Hybrid Markov Logic Networks have been successfully used in learning probability distributions over logical rule states of a Markov Logic Network (MLN). Learning MLN structure is a rich and ongoing area of research, including approaches using inductive logic programming [70], hypergraph lifting [71], utilizing structural motifs [72].

6.2 ABL - A Behavior Language

ABL (A Behavior Language) is a reactive programming language designed for creating believable agents, typically for interactive domains. Developed by Mateas and Stern [95], it is an extension of the Oz Project language Hap [91]. While its original design principles were centered around creating believable agents for game environments, it has also been applied to more complex planning problems as well, namely the real-time strategy game StarCraft:Brood War (SC:BW) [155]. Strong theoretical foundation in programming language design, working applications in a variety of domains, and availability of full source code were the features of the language that led to this project's choice to select this language. This work focuses on explicit abstraction as part of the design process, thus making a higher level language desirable. These abstractions can be applied to more efficient implementations for particular domains but the learning approaches and action relationships should be applicable across implementations and to a certain degree even domains.

I will present here a high-level overview of the language; for more details see the original language references [95, 98]. ABL uses hierarchical definitions of goals and actions, tied in to a 'working memory' system and sensorimotor connections, to define how an agent acts and reacts to its environment. Approaches to achieving goals are encoded in *behaviors*, which can be either sequential or parallel, and are composed of *actions* and *subgoals*. The former is a primitive action in the domain for the agent, and the latter inserts a new goal into the *Active Behavior Tree* (ABT) (see Figure 6.1). Finally, behaviors can execute arbitrary code (known as a *mental act*), allowing an agent to make meta-reasoning adjustments to its own goals and knowledge base mid-operation.

```
sequential behavior AnswerTheDoor() {
 act sigh();
 subgoal OpenDoor();
 subgoal GreetGuest();
}
```

In this example behavior (taken from [95]), the agent breaks down the task of answering the door into a concrete action of sighing, followed by two subgoals of opening the door and greeting whoever is there. The sigh, as a concrete action, is perhaps an animation or sound effect, while the OpenDoor and GreetGuest subgoals are placed in order into the ABT. When they are chosen to expand, they may have multiple behaviors available depending on the situation. For example, the agent may need to put down whatever it is holding before opening the door. Alternatively, there may be different greetings depending on who it is at the door.

The ABT is where an ABL agent keeps track of its current goals. At each step, it is examined for open goals, and a meta-reasoning step selects the goal with the highest *priority* for expansion, at which point the agent selects the behavior for that goal with the highest *specificity*, filtering for those whose preconditions are satisfied. Priorities and specificities are encoded by the author, and ties in priority and specificity are broken arbitrarily.

The ABT can be altered in a number of ways. One of the most common is a behavior failing, due to some exogenous change in the environment or internal conflict. This failure propagates upward in the tree, where each successive parent handles the failure of its subgoal, either by ignoring it and continuing on, or failing itself and passing the failure upward. Additionally, behaviors can directly modify the ABT themselves with mental acts, as mentioned earlier.

```
sequential behavior chase() {
  precondition {
    (PlayerWME locationX::chaserX)
    (TargetWME locationX::targetX)
    (chaserX < targetX)
  }
  specificity 2;
  act moveRight();
  subgoal chase()
}
sequential behavior chase() {
  precondition {
    (PlayerWME locationX::chaserY)
    (TargetWME locationX::targetY)
    (chaserY < targetY)
  }
  specificity 1;
  act moveUp();
  subgoal chase()
}
```

The above behavior is an example for a simple pursuit behavior. In this situation, there is a precondition asserting that the agent is to the left of the target, after binding the agent and target's locations from the knowledge base to variables. If so, it takes a primitive moveRight action, and then adds the chase goal to the ABT. Similarly, the agent has another behavior for the chase goal that instructs it to move upward if the target is above it. Note that the specificity definitions for these two actions mean that in the situation where the target is

Figure 6.1: A segment of the Active Behavior Tree (ABT) for a StarCraft agent.

above and to the right of the agent, it will first move right until it is even with the target, then move upwards until reaching the target.

As expected, in a more complex domain, the ABT can become much larger and more complex. Figure 6.1 shows a small portion of the ABT for EISBot, an ABL agent for SC:BW. Each goal shows the subgoals that it has spawned underneath it. This portion shows the goals related to high-level strategy decisions and repetitive supply management, though finer details such as priorities are not displayed.

There is significantly more explanation needed for a full understanding of ABL [95], such as the implementation of context conditions, or the infrastructure of how an agent senses its environment and translates this into a knowledge base to use in planning and reacting, but this project leave these parts of ABL unchanged, and so I will pass over them in the interest of brevity.

6.3 Probabilistic ABL

As has been stated throughout this dissertation, the primary goal is to develop a system for learning hierarchical abstractions (in this instance, ABL model definitions) from human demonstrations. However, it is difficult to progress in this without a metric for evaluating how well an ABL model explains a given trace of human actions. The only non-heuristic measurement available is whether or not it was possible for the model to have generated that sequence of actions given the environment, a binary metric with very sparse signal, which makes learning incredibly difficult.

While many deterministic frameworks [89, 73] have made impressive strides with heuristics and backtracking, I chose instead to alter the framework, for two main reasons. First, I felt that it was justified given the principles of ABL. An agent with no stochasticity can have trouble being believable. While a physicist or neurologist might argue that human behavior *may* be deterministic if you are able to examine neurons at a sufficiently precise level, when working at the level of agent programming that frequently expresses itself as stochasticity to an external viewer. And so one likely needs some amount of randomness if one is to be creating believable agents. Second, from a technical standpoint, it gives the agent more flexibility and enables the calculation of a more nuanced metric for model quality given a set of human actions, which can allow agents to more easily learn from

demonstration.

The modifications are centered around the goal and behavior selection processes. Recall that during each step, ABL selects the goal from the ABT with the highest priority for expansion, and then chooses the behavior to achieve that goal with the highest specificity, under the constraint that its preconditions are satisfied. I replace this deterministic selection behavior with a probabilistic one. Rather than take the goal with the highest priority, the agent draws from a generalized Bernoulli distribution where the probability of selecting each goal is linearly proportional to its priority. I use this as opposed to the more common softmax distribution primarily for simplicity of authorship. If a goal has twice the priority of another, it will have twice the probability of being selected. On the other hand, this scale invariance is not true for the softmax, and it would introduce an additional temperature parameter that would require tuning. Similarly, for behavior selection, once the agent has filtered the behaviors for satisfied preconditions, it draws from a generalized Bernoulli distribution where the probability of selecting each behavior is linearly proportional to its specificity.

As a final note, the original authors of ABL were well aware of the value of nondeterminism in action selection, and ABL actually does have it built in. When two or more goals or methods have matching priorities or specificities, one is chosen at random from the uniform distribution. In essence, this is simply generalizing that decision to the entire goal/method selection process, for a number of benefits.

6.3.1 Trace Evaluation

Most importantly, this change allows us to calculate the probability of a given ABL model producing a certain trace of actions, given the environment state throughout the trace. Namely, it is the sum of the probabilities for all possible

expansions of the ABT that would generate that specific sequence of actions, since it is possible for different interleaved behaviors to generate the same sequence of actions. Mathematically, with

- $A = \{a_1, \ldots, a_n\}$ as the trace of concrete actions, with A_t denoting the actions from time t onwards
- θ as the pABL model
- $S = \{s_1, \ldots, s_n\}$ as the environment states at each action taken, with S_t denoting the states from time t onwards
- G as the ABT of open goals
- B_g is set of behaviors that achieve goal g
- $e(b)$ is the subgoals and/or primitive actions that result in expanding behavior b

we can say that

$$P(A_t|\theta, S_t, G) = \sum_{g \in G} \sum_{b \in B_g} \begin{cases} P(g|G) * P(b|g, s_t) * & \\ \quad P(A_{t+1}|\theta, S_{t+1}, \{e(b), G - \{g\}\}) & e(b)[0] == a_t \\ P(g|G) * P(b|g, s_t) * & \\ \quad P(A_t|\theta, S_t, \{e(b), G - \{g\}\}) & e(b)[0]\ is\ subgoal \\ 0 & else \end{cases}$$

$$P(A_{n+1}|\theta, S_{n+1}, G) = 1$$

Intuitively, we can understand this equation starting with the base case: once we've reached the end of the action sequence, the probability of a given model generating an explanation for an empty sequence of actions is 1. For the recursive definition, we sum across all possible selection of goals from the ABT and all possible behaviors for each of these goals. If the first action in the expansion of this behavior matches with the next action in the sequence (the first case), we add the probability mass of selecting this particular behavior and recurse to the next action with an updated ABT. If the first action is a subgoal (the second case), then we don't know yet if this will lead to a legal action, so we recurse at the same point in the trace, but with an updated ABT. Finally, if we expand to a concrete action that doesn't match the next action in the demonstration (the final case), then this explanation branch is not a valid way to generate the demonstration, and we discard the probability mass associated with it.

In practice, computing this precise value is frequently computationally intractable for a complex domain. It requires modeling all possible expansions of the ABT over the course of the trace, and while it is possible to prune expansions that lead to primitive actions that do not match the trace, it still results in an exponential blowup. In such a situation, I perform Dijkstra's search over a directed acyclic graph (DAG) with nodes representing states of the ABT and edges representing goal/behavior expansions, using the negative logarithm of $P(g|G) * P(b|g, s_t)$ as the weight. This guarantees that the learning system will find the most probable explanation for the trace under a given ABL model, which can be used as a proxy for the full calculation in tasks such as updating preconditions or relative evaluation of model quality. In addition, if it tracks the probability mass that has been pruned away due to mis-matching primitive actions, when it reaches the most probable explanation it can also calculate an upper bound for the true probability,

by summing the remaining probability mass that has not been discarded.

Note the difference between this evaluation metric and the coverage and signal that was used in Chapter 5. In that work, we had access to an oracle that could tell us precisely which of all correct plans the structure hypothesizers had found, and which extraneous plans they were generating. In this situation, we lack that ability, and instead rely on measuring how well the model being learned matches with the demonstrations available.

6.4 Experiments and Results

To evaluate pABL, I conducted two separate experiments in two different domains in which ABL has been historically used. The first demonstrates an improvement in solution quality in a simple domain. It is primarily a proof of concept for the pABL modifications, and exhibits no abstraction learning. The second is an initial examination of how pABL can be used to learn abstractions from human demonstrations in a complex domain.

6.4.1 Pursuit

The introductory domain for ABL programming is a basic pursuit domain, with a simple ABL agent pursuing a target, controlled by the user in an interactive mode in an open field, to demonstrate how ABL works. It has the benefit of being a component of behavior that is desirable in more complex domains, while at the same time being simple enough to easily grasp how the agent's strategy is encoded.

The target is a simple randomly moving agent, starting in the upper right quadrant. The field itself is a 128 by 64 grid, allowing movement in the cardinal directions. The pursuing agent was initialized in the far corner of the opposite

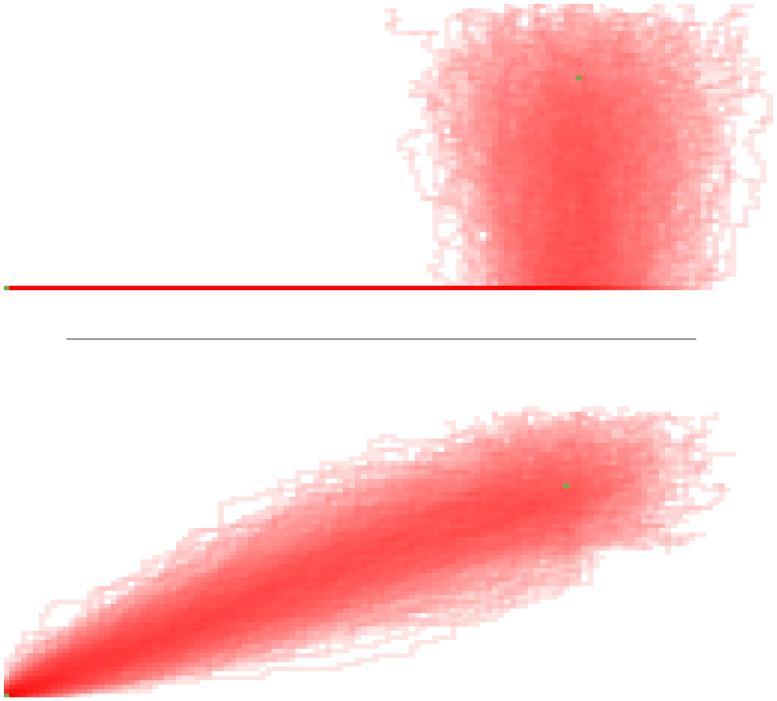

Figure 6.2: Heatmaps for the paths taken by the deterministic ABL agent (top) and the pABL agent (bottom). The chaser agent begins in the bottom left, and the target begins in the center of the top right quadrant.

quadrant, and we measured how many steps were taken before it overtakes the target. The two agents move at the same speed, but since the target's movement is random, the pursuer will eventually make the catch provided it continues to move toward the target.

We tested two agents for this experiment. The first is the deterministic ABL agent that is traditionally used for this introductory domain. An excerpt of the code was shown above; the agent essentially has one goal with four behaviors, with preconditions and specificities such that it will always move towards the target, but will prefer horizontal moves over vertical moves. The second agent is identical to the first, but run using pABL rather than ABL. This means that it will be drawing from a distribution that allows it to take either horizontal or vertical moves at any given step, but is twice as likely to make a horizontal move.

Agent	Mean Steps
Deterministic ABL	192.7 ± 21.4
pABL	151.5 ± 13.0

Table 6.1: Number of steps required for each agent to reach the moving target over 10k simulations.

As we can see in Table 6.1, the pABL agent with a weighted preference for moving horizontally reaches the target more quickly. This is an expected result, as the other agent will prematurely reach the horizontal coordinate of the target before the vertical, and will then need to spend steps tracking future changes in the horizontal dimension while attempting to close the gap on the vertical dimension. Therefore, while not surprising, it does demonstrate the potential advantage of probabilistic behavior selection.

Also worth noting, as a side benefit, from a qualitative standpoint the behavior of the agent itself is much more believable. At its core, the problem with the deterministic agent is that it attempts to get a single location dimension exactly correct, and only then attempts to match the other dimension. See Figure 6.2 for a visualization of this problem. Meanwhile, a human will typically attempt to get generally 'close enough' in all dimensions, before attempting to make final refinements, which is the behavior exhibited by the pABL agent, as it makes improvements in each dimension according to its probabilistic biases. As such, this is a useful case study for how stochasticity can produce more believable agents.

There are more interesting variations that could be explored in this domain, but in truth it is mostly a proof-of-concept. It demonstrates the potential pitfalls of simple deterministic ABL agents, and how the addition of stochasticity in goal/behavior selection can help to alleviate those issues.

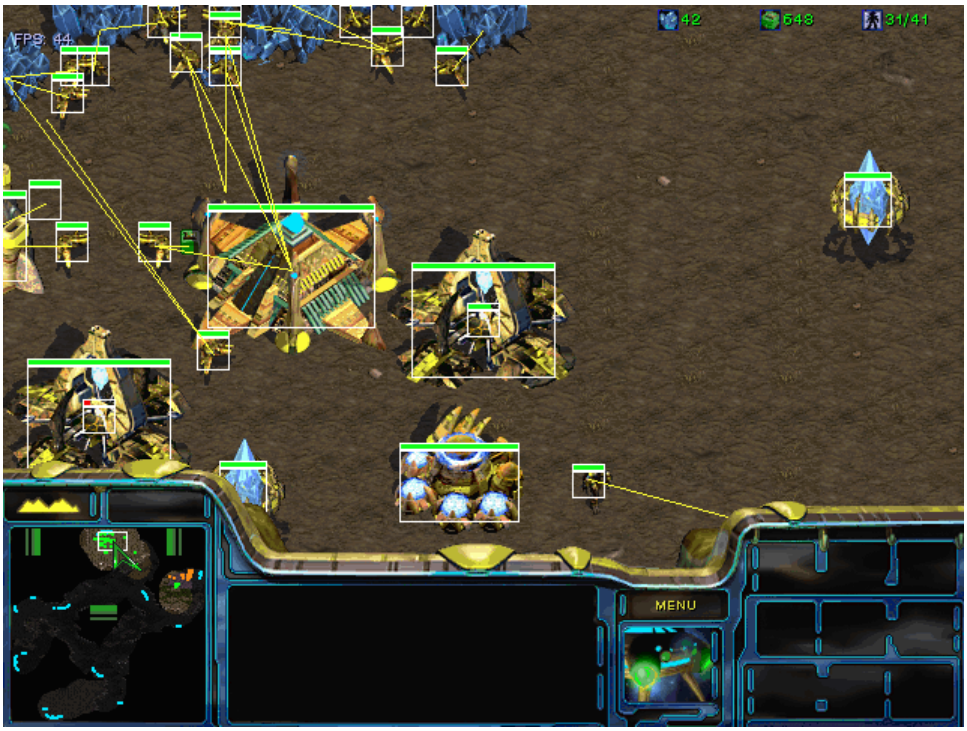

Figure 6.3: Example screenshot of EISBot playing the real-time strategy game StarCraft:Brood War. For the additional drawn information, boxes represent EISBot's knowledge of specific units, while lines delineate the orders that the unit is following currently.

6.4.2 StarCraft: Brood War

As discussed in Chapter 2, SC:BW is a real-time strategy game of deep complexity. While there are many features that make it a compelling research domain, I would like to highlight two in particular as most relevant to this work.

The first is the professional community that has developed around the game. This community is a group of highly-skilled players who can be considered 'experts' of the field, due to years of practice in a high-stakes competitive environment. What is more, professional replays are made available online, as gifts to the wider community for studying and enjoyment, and which researchers can use in the development of agents capable of learning from observation.

The second feature of particular interest to this work is the SC:BW agent

EISBot [155], which is written in ABL. EISBot has a number of fixed strategies encoded in ABL's reactive planning paradigm, as well as a goal reasoning component that performs meta-reasoning over the agent's goals themselves, looking for discrepancies between the expected state and actual state when a goal is completed. We disable the goal reasoning component of EISBot, which requires additional resources outside of ABL, namely, a knowledgebase to query for goal introspection. Instead, we use the pure ABL formulation of EISBot, which is a competent SC:BW agent in its own right, and more tractable for the purposes of this work.

The initial tests involve two agents: Naive and EISBot. Naive is a simple ABL agent with a looping goal for each primitive action in the domain. That is, it is made up of a number of goals that are of the form 'Build(x), repeat this goal', and it will attempt to learn when and at what priority to execute these. EISBot is a translated version of the aforementioned SC:BW agent into pABL, initialized with the priority/specificity and precondition parameters from the original model. As mentioned earlier, when evaluating how well a model θ explains a particular trace, I use the negative log likelihood of generating the trace given θ and the environmental state at each decision step. However, since the full probability is computationally intractable, I approximate it with the negative log likelihood of the most probable explanation. While not a precise *absolute* approximator, this provides a good *relative* approximator for comparing the quality of different models.

Behavior Parameter Learning

I began by using these two goal/behavior structures (Naive and EISBot) as frameworks within which I wished to learn parameters for the behaviors.

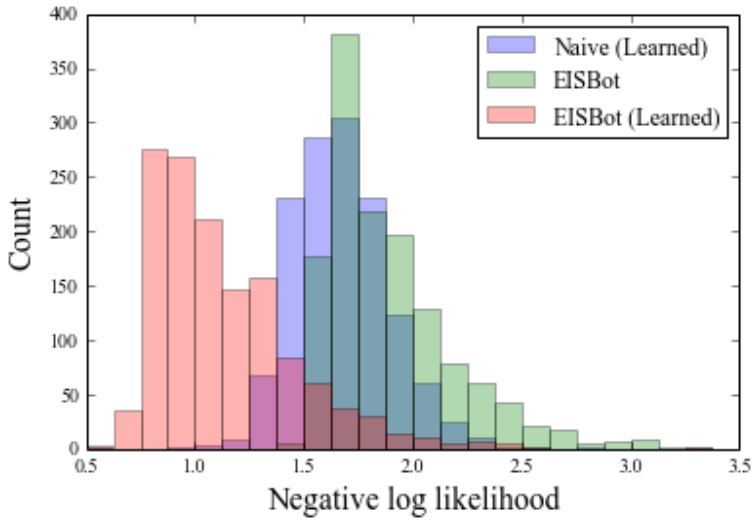

Figure 6.4: A histogram of negative log-likelihoods for the most probable trace explanation for each replay in the database. Learned denotes a system with both precondition and priority/specificity learning. Lower is better.

- **Priority/Specifity** For priority and specificity learning, I take advantage of the simplicity of the generalized Bernoulli distribution. Once the algorithm has computed the most probable explanations for all traces in the training set, it sets each goal's priority and behavior's specificity to the number of times it appears in the set of explanations. In order to smooth out the distribution and prevent goals and behaviors from being completely zeroed out after a single assignment, I use Laplace smoothing with $\alpha = 1$, essentially adding a pseudo-observation of each goal and behavior.

- **Preconditions** For learning behavior preconditions, I take the most discriminative state space possible that still satisfies each instance of the behavior's application. That is, given again the most probable explanations for all traces, the system examines the set of states S in which a given behavior was applied. For each state variable, it considers the maximum and minimum value it takes in S as potential upper or lower bounds for the ap-

Agent	Median $-ln(L)$
Naive (no learning)	3.95
Naive (precondition)	3.35
Naive (priority/spec)	1.82
Naive (both)	1.73
EISBot (no learning)	1.82
EISBot (precondition)	1.55
EISBot (priority/spec)	1.12
EISBot (both)	1.06

Table 6.2: Median negative log likelihoods across the test set, for different initial models and degrees of learning. Lower is better.

plication of this behavior. It then computes the fraction of states satisfying this bound over the total number of states in the dataset. A small ratio indicates a very discriminative condition.

For example, suppose that there exists a behavior whose applicable states have a lower bound of owning a single worker. This is not a useful precondition, as it is trivially true in nearly all states. However, if a behavior is also only ever applied when the agent has a very large army (a more rare event), this is a much more informative precondition.

For now preconditions are restricted to conjunctions of single variables in the state being above or below some constant, but I hope to expand this in the future to multi-variable expressions.

Results for these learning methods can be found in Table 6.2 and Figure 6.4. As one can see, EISBot is much more capable than the Naive agent of generating probable explanations for traces, and only improves with precondition and goal reasoning learning. Moreover, it appears that both agents benefit more from learning priorities and specificities than from preconditions. This indicates this is a domain where many different goals/behaviors may be applicable at any given

```
sequential behavior researchLegSpeeed(){
  precondition {
    (ReconWME zealotCount>=10)
    (CitadelWME ID::citadelID)
    (PlayerWME minerals>=150 gas>=150)
  }
  act upgrade(citadelID,LegEnhancements);
}
sequential behavior researchLegSpeeed(){
  precondition {
    (CitadelWME ID::citadelID)
    (PlayerWME minerals>=150 gas>=172)
    (ReconWME gatewayCount>=3)
    (ReconWME probeCount>=16)
  }
  act upgrade(citadelID,LegEnhancements);
}
```

Figure 6.5: Preconditions for the researchLegSpeed behavior before and after learning from human demonstrations. The expert-encoded preconditions are above, and the learned are below.

step, and the more important capability is the ability to choose *which* to apply.

Note the sharp edge of the distribution of trace explanation quality for EISBot in Figure 6.4. EISBot has the most common human strategies encoded in its plan structure, and when human players hew closely to those common strategies, EISBot has a good probability of generating the given trace. However, the more the demonstration deviates from common gameplay, the more EISBot struggles to generate a good explanation. Without structure learning, it is impossible for EISBot to learn new hierarchical structures to explain these rarer games well, and so even improved preconditions and specificities/priorities are insufficient.

Figure 6.5 shows the preconditions for the researchLegSpeed (an upgrade to a basic military unit) behavior of EISBot before and after precondition learning. For this experiment, the algorithm learned the preconditions from scratch, using

EISBot's goal structure but not its encoded preconditions. Some terminology: Zealots are the unit that benefits from the upgrade, Gateways are the buildings from which they are produced, and probes are the resource collecting unit. Both identify the requirements of resources and research location that are hard preconditions imposed by the game itself, but the original used army size to determine when to make the upgrade, while the learned preconditions indicate that human players make their decision relative to the amount of economic infrastructure they have set up instead.

Gameplay Evaluation

In addition, I took these parameters (priorities, specificities, and preconditions) and re-implemented them in the original EISBot for actual gameplay evaluation. I tested both the original EISBot and our modified version (hereafter referred to as pEISBot) against a number of agents, in order to see how the learned parameters affected gameplay. Table 6.3 shows the results of these tests. The Naive agent is the built-in AI in SC:BW, Nova [124] and BTHAI (2012) [43] are two contemporaries of EISBot, and UAlbertaBot [27] is a more finely tuned modern agent. UAlbertaBot is able to play multiple races, so I evaluated against its main race (P) and one of its off-races (T). Each matchup was played 100 times, spread across 5 different maps.

As the table shows, the learned agent performed similarly to the hand-crafted agent, though slightly worse. In conjunction with the prior evaluation, this suggests that while pEISBot may be better at modeling how humans play the game, it is slightly inferior when it comes to actual gameplay. However, this is actually encouraging to us, as EISBot was developed over multiple years, with its parameters being tuned through iteration and evaluation, while our parameters were

Opponent	EISBot	pEISBot
Naive	100%	98%
Nova (2012)	100%	100%
BTHAI (2012)	91%	92%
UAlbertaBot (T)	64%	54%
UAlbertaBot (P)	0%	0%
EISBot	-	38%

Table 6.3: Winrates for EISBot and pEISBot against a range of opponents. UAlbertaBot was primarily developed to play the Protoss race (P), hence the much stronger capabilities.

learned automatically, and perform nearly as well.

Learned Structures

In addition to evaluating the learning capabilities of pABL when applied to a human-encoded structure, I wished to investigate how it performed when applied to goal/behavior structures learned from the traces themselves. For this, I use the structure learning approaches presented in Chapter 5. While originally developed for Hierarchical Task Networks, the underlying structures of HTN and ABL models are sufficiently similar that they can be adapted to our use case, with the main differences between the two planning frameworks arising in the planning algorithms themselves.

The following is a brief review of each approach:

- **Pattern Mining** – This approach uses the Generalized Sequential Pattern (GSP) algorithm [143] to iteratively extract structures from a database of traces. GSP is parameterized by a maximum time gap between actions allowed within a pattern, which is gradually increased over the course of the search. When a pattern is found that surpasses a set threshold of frequency, that pattern is encoded as a behavior, assigned an abstract symbol representing its goal, and replaces all instances of itself within the trace library

with said abstract symbol. In this way, the first behavior is guaranteed to only contain concrete actions, but any subsequent learned behavior can be any mix of concrete actions and previously encoded goals. This is shown empirically to be capable of learning common SC:BW build orders, though it results in a relatively inflexible structure.

- **Expectation Maximization** – The second approach uses an expectation maximization (EM) algorithm to hypothesize structure. Similarly to the first, it iteratively builds up a model by creating a goal/behavior(s), adding it to the model, then updating the trace library to replace instances of that behavior with the appropriate abstract goal. However, instead of using pattern matching to find the next candidate behavior, it instead models the behaviors as a set of latent Markov processes. Using EM with a modification of the Viterbi algorithm used to solve for hidden Markov models, it can find the most likely parameters for these approximations of behaviors, which can then be converted back into behaviors, keeping only behaviors whose likelihood is above some threshold. This results in more flexible structures, as goals can end up with multiple different behaviors achieving them, if the encoded process had a number of probable paths through it.

These two approaches both result in hypothesized goal/behavior structures, and do not explicitly learn preconditions (and/or priorities and specificities, though that is expected due to their origin in HTNs). In this evaluation, I wish to show that, used in conjunction with our pABL learning formulation, they can learn a working model from scratch. To do this, I encode the learned HTN structures as pABL goals and behaviors, initializing behaviors with no preconditions, and uniform priorities and specificities for goals and behaviors respectively.

Table 6.4 shows the effects that the different parameter-learning approaches

Agent	Median $-ln(L)$
EM (no learning)	1.60
EM (precondition)	1.28
EM (priority/spec)	1.22
EM (both)	1.16
GSP (no learning)	1.47
GSP (precondition)	1.29
GSP (priority/spec)	1.05
GSP (both)	0.99
EISBot (both)	1.06

Table 6.4: Median negative log likelihoods for the hypothesized structures under various amounts of parameters learning (preconditions, priorities/specificities, or both). EISBot included for reference.

have on the generated structures' explanatory capabilities. As before with the Naive and EISBot agents, there is a comparatively greater improvement from shifting priorities and specificities from a uniform distribution to a more accurate one, but learning accurate preconditions is also shown to improve the likelihood of the most probable explanation.

Figure 6.6 shows the distribution of explanation likelihoods across the trace library, before and after parameter learning. Viewed in contrast to Figure 6.4, one can see that both are generally outperforming the Naive model from earlier. However, what is more interesting is that they are also nearing or even providing better explanations than the EISBot goal/behavior structure with learned parameters. While this is an encouraging result, there are some caveats to note, which I will consider when discussing future work.

6.5 Conclusion and Future Work

In this chapter, I have presented a probabilistic modification of the ABL programming language and justified its use with a simple domain example. Following

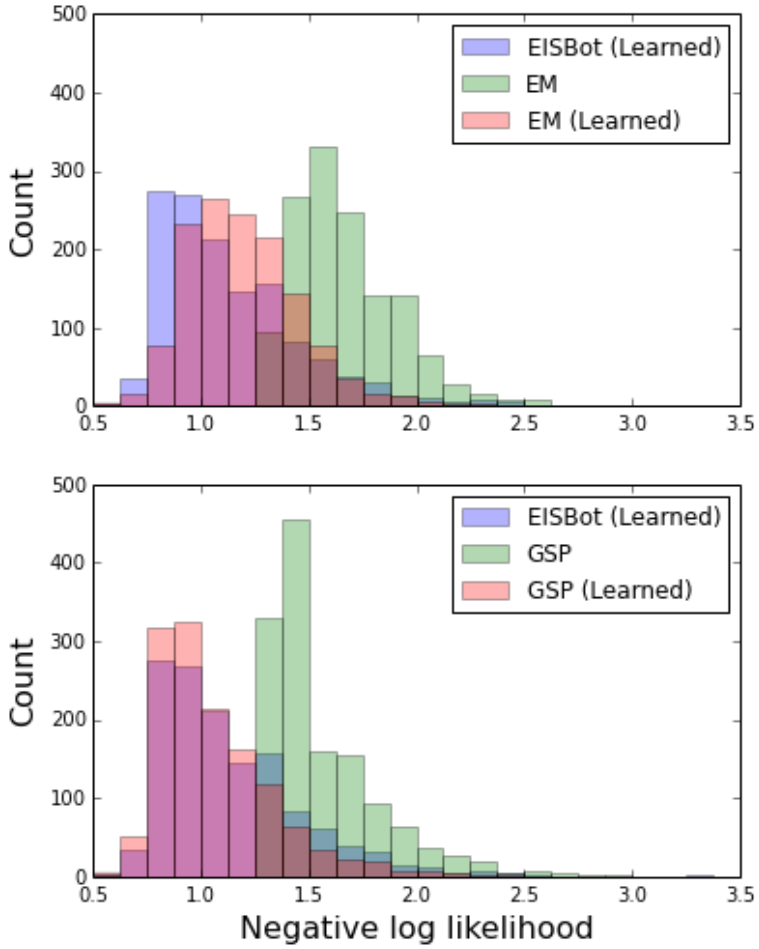

Figure 6.6: Histogram of likelihoods for the goal/behavior model generated via expectation-maximization and GSP pattern mining, both before and after parameter learning. EISBot included for reference.

that, I have shown that pABL is more suited to the task of learning from human demonstrations, with all of the noise and inconsistent actions present therein. The ability to calculate a metric for how well a model 'explains' a trace allows us to compare model quality, and the ability to calculate a most likely explanation allows for the development of algorithms to iteratively improve a model's parameters.

The results with learned goal/behavior structures imply that, if one is solely interested in the task of predicting the next action a human will take, these learned

models would match the expert-encoded model of EISBot, which is very impressive at first glance. There are two caveats to this claim, however. The first is that EISBot was not developed to predict moves of other players, but rather to simply play as an intelligent agent itself. The second note mirrors the first, which is that simply predicting a next move is not guaranteed to translate directly into expert play. For reference, consider the recent success of AlphaGo [138]. Merely achieving a high accuracy in predicting expert moves was not sufficient to break the expert barrier, the agent also needed to learn a policy network that pushed towards winning the game. I anticipate a similar additional learning step will be required to tune the learned agents.

As such, the most pressing future work for this project is to implement the learned agents as fully-functional SC:BW-playing bots. This is critical to understanding how the existing learning process can be improved, and what additional learning steps might be necessary for further improvements. Furthermore, this should be fairly straightforward, as the mechanisms for connecting an ABL model have already been created for EISBot itself.

While the current approaches have the capability to iteratively improve their structure based on the feedback available from a metric for explanatory power, they do not respond to that information. Such an optimization step is certainly not trivial, of course. Gradient-based methods are not applicable due to the non-differentiability of the space of hierarchical structure abstractions, but other approaches like evolutionary algorithms may be possible. This is the other most promising immediate direction for this work to be extended.

As a final note on pABL, it is certainly possible to construct situations in which a pure probabilistic ABL will have difficulties. In such situations when authoring agents by hand, one may wish for deterministic behavior. In the end,

it may be the case that the best authoring language is a hybrid that offers both deterministic and stochastic behavior, depending on what is appropriate. This is a kernel of a concept regarding future work that I will discuss in Chapter 7. Regardless, I believe this work demonstrates that the inclusion of stochasticity is a useful addition in terms of agent flexibility and learning utility.

Chapter 7

Conclusion

In this dissertation I have presented work on learning abstractions for artificial agents from human observation, with a particular focus on hierarchical planning abstractions. As AI domains begin to more and more mirror the real world, in its continuous space and infinite set of possible actions, it is important that artificial agents are able to reason and plan at a higher level. Due to the nature of these complex domains, it is also becoming impractical to fully author the requisite knowledge and planning abstractions. Simultaneously, we are entering the age of big data, with huge amounts of data on humans, human behaviors, and human demonstrations being stored. The conjunction of these two trends leads to a strong pull towards developing abstraction learning algorithms that can handle human demonstrations.

As mentioned in Chapter 1, the primary research question that this work attempted to answer was the following:

> How can effective planning abstractions be learned from human demonstrations in RTS games?

The contributions of this work are a reinforcement learning system for the simplified RTS game Planet Wars, two algorithms for learning hierarchical planning

structures from a database of human demonstrations, and a system for learning parameters for a reactive hierarchical planning model (in this case, using the ABL programming language) using only raw human demonstrations.

One of the primary concerns entering this work was the challenge of handling the presence of extraneous actions in human demonstrations, which will throw off many first-order logic approaches. In response to this, I have tended towards algorithms that are inspired by different machine learning paradigms, allowing for the filtering out of low-signal actions as irrelevant.

A second major challenge was the goal of learning from completely unlabeled and non-annotated demonstrations. Most prior work focused on simply reducing the amount of expert labeling required, rather than eliminating it entirely. To address this, I presented two algorithms that attempt to learn decompositional structure from the ground up, learning low-level goals and methods first, then composing those into higher level abstractions. In addition, within the presented model evaluation framework of pABL, I have demonstrated that one can use an existing hypothesized model to pseudo-label demonstrations, which can then be used to learn an improved model, with the whole process repeatable.

I do not claim that these are the only approaches to learning hierarchical abstractions from human demonstrations. By definition, when attempting to learn a model that is based off of examples generated by some different planning process, in this case the human mind, there will inevitably be misalignments, or 'noise'. We can attempt to ameliorate these with simplifying assumptions and sufficient data, but in the end one is still only learning an approximation of a human's planning system. As a result, different researchers with different assumptions and algorithms will arrive at different learning approaches, and we as a research community must endeavor to push forward the most promising. As a specific

example, recent work from Malỳ [92] formulates the learning from demonstration problem as a multi-agent system and uses inverse reinforcement learning, another possible area for future work.

7.1 Future Work

There are a number of areas for future research to branch off of and extend this dissertation. I believe that the following are the most immediate research areas to follow up on this work:

- **Stronger gameplay evaluation** – In Chapter 6 I showed that the final learned structures for SC:BW have demonstrated their capacity to explain human replays well. However, this is not necessarily synonymous with strong gameplay ability, which is something that remains untested. I believe that it would be beneficial to run a comprehensive gameplay study with these agents, and that beyond simply measuring progress, it would help direct future research in improving the learning components. In addition, I would like for the evaluation to include measurement against human opponents rather than solely artificial agents, similar to Weber's ablative studies in [155].

- **Iteratively improving structures** – Using the probabilistic modifications to ABL in Chapter 6, I was able to learn model parameters for hierarchical planning models based on a database of demonstrations. However, I have not yet fully taken advantage of the capabilities that this offers us. In particular, it is now possible to iteratively improve the model's hierarchical planning structure itself, adjusting goal and sub-goal decompositions to better explain the input data. While the lack of a clear gradient makes developing an

algorithm for this task non-trivial, it is the most needed improvement in a full end-to-end learning system. My system as currently set up is sufficient to learn a model, but can be improved with this additional step on the end.

- **Integrating spatial and hierarchical planning abstractions** – Currently, the spatial reasoning abstractions and the hierarchical planning abstractions are two separate research foci, learning independently. This hampers each, by forcing them to ignore certain aspects of gameplay when analyzing traces, removing signal and adding noise. A principled approach to integrating the two should improve the performance of each, in addition to learning a more accurate model for human gameplay. I believe that this will be a critical step for this work in the process of becoming widely and easily usable in practical domains.

7.2 Discussion

There are a number of similarities between this work and the recent growth in reinforcement learning for games. Each work to remove the agent authoring bottleneck, with the goal of surpassing authored agents, and as mentioned in Chapter 6, the difference between prediction and policy arises for both. However, the assumed mental model differences between the two lead to differing strengths and weaknesses, in areas such as interpretability, number of demonstrations required, and flexibility.

These similarities, in conjunction with the challenges that I have found in working with classical hierarchical planning systems, leads me to believe that an important next step in the larger research field is an integration between said planning systems and the machine learning approaches currently being developed at a

breakneck pace. In some ways, I have attempted to make this integration in this work. While I am attempting to learn more classical hierarchical planning models, the learning itself takes inspiration and ideas from a number of more statistical machine learning algorithms and paradigms, like Markov decision processes and the expectation-maximization process. However, this needs to be driven forward and made more interwoven into the models themselves.

I believe that it is critical to develop hybrid approaches between classical AI and more modern machine learning approaches. The power of logic and search was proven effective when computational capacity was limited and domains were well-understood, but it has struggled more in the practical world. Meanwhile, reinforcement learning and neural networks have been proven to be highly effective in a large number of real-world tasks, but they are still poorly understood and explained, leading us to not definitively know where the boundaries of their capabilities are or aren't, and also to struggle with interpreting *why* some system may be working.

It is a guarantee that statistical approaches will need to be integrated into classical planning systems, if only for the reason that many real-world inputs in the future will be based on them. For example, any planner that uses vision as an input will almost certainly be taking input from a deep neural net, and needs to be able to interpret that data. Rather than treating this interface as a one-off, I believe that it should be deeply researched. What is more, I believe that it will have large implications for learning from demonstration in these systems, as they will be more intricately linked to machine learning approaches and algorithms.

Bibliography

[1] David W Aha, Tory S Anderson, Benjamin Bengfort, Mark Burstein, Dan Cerys, Alexandra Coman, Michael T Cox, Dustin Dannenhauer, Michael W Floyd, Kellen Gillespie, et al. Goal Reasoning: Papers from the ACS workshop. Technical report, Georgia Institute of Technology, 2015.

[2] David W Aha, Matthew Molineaux, and Marc Ponsen. Learning to win: Case-based plan selection in a real-time strategy game. In *International Conference on Case-Based Reasoning*, pages 5–20. Springer, 2005.

[3] Christopher Amato and Guy Shani. High-level reinforcement learning in strategy games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: volume 1-Volume 1*, pages 75–82. International Foundation for Autonomous Agents and Multiagent Systems, 2010.

[4] David Andre and Stuart J Russell. Programmable reinforcement learning agents. In *Advances in neural information processing systems*, pages 1019–1025, 2001.

[5] Carol Applegate, Christopher Elsaesser, and James Sanborn. An architecture for adversarial planning. *Systems, Man and Cybernetics, IEEE Transactions on*, 20(1):186–194, 1990.

[6] Marcelo G Armentano and Analía A Amandi. Modeling sequences of user actions for statistical goal recognition. *User Modeling and User-Adapted Interaction*, 22(3):281–311, 2012.

[7] Marc Bellemare, Sriram Srinivasan, Georg Ostrovski, Tom Schaul, David Saxton, and Remi Munos. Unifying count-based exploration and intrinsic motivation. In *Advances in Neural Information Processing Systems*, pages 1471–1479, 2016.

[8] Alex Bellos. Rise of the e-sports superstars, June 2007. [Online; posted 29-June-2007].

[9] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[10] Nate Blaylock and James Allen. Fast hierarchical goal schema recognition. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 796. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[11] Blizzard. Starcraft forums. `https://us.battle.net/forums/en/starcraft/`. Accessed: 2018-08-01.

[12] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

[13] Guillaume Bosc, Mehdi Kaytoue, Chedy Raïssi, and Jean-François Boulicaut. Strategic Patterns Discovery in RTS-games for E-Sport with Sequential Pattern Mining. In *MLSA@ PKDD/ECML*, pages 11–20, 2013.

[14] Bruno Bouzy and Tristan Cazenave. Computer Go: an AI oriented survey. *Artificial Intelligence*, 132(1):39–103, 2001.

[15] Michael Buro and David Churchill. Real-time strategy game competitions. *AI Magazine*, 33(3):106, 2012.

[16] BWReplays. Starcraft:broodwar replays. `http://www.bwreplays.com/`. Accessed: 2018-08-01.

[17] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1):57–83, 2002.

[18] Michal Certicky. Implementing a wall-in building placement in starcraft with declarative programming. *arXiv preprint arXiv:1306.4460*, 2013.

[19] Michal Certicky and David Churchill. The Current State of StarCraft AI Competitions and Bots. 2017.

[20] Eugene Charniak and Robert P Goldman. A Bayesian model of plan recognition. *Artificial Intelligence*, 64(1):53–79, 1993.

[21] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *AIIDE*, 2008.

[22] Ho-Chul Cho, Kyung-Joong Kim, and Sung-Bae Cho. Replay-based strategy prediction and build order adaptation for StarCraft AI bots. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–7. IEEE, 2013.

[23] Dongkyu Choi and Pat Langley. Learning teleoreactive logic programs from problem solving. In *Inductive Logic Programming*, pages 51–68. Springer, 2005.

[24] Dave Churchill. 2013 AIIDE StarCraft AI competition report. http://webdocs.cs.ualberta.ca/∼cdavid/starcraftaicomp/report2013.shtml. Accessed: 2014-07-20.

[25] David Churchill and Michael Buro. Build Order Optimization in StarCraft. In *AIIDE*, 2011.

[26] David Churchill and Michael Buro. Incorporating search algorithms into rts game agents. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[27] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[28] David Churchill, Abdallah Saffidine, and Michael Buro. Fast Heuristic Search for RTS Game Combat Scenarios. In *AIIDE*, 2012.

[29] CrosSpade. Twinbot. http://crosspade.com/twinbot/. Accessed: 2014-07-22.

[30] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. Ieee, 2009.

[31] Ethan W Dereszynski, Jesse Hostetler, Alan Fern, Thomas G Dietterich, Thao-Trang Hoang, and Mark Udarbe. Learning Probabilistic Behavior Models in Real-Time Strategy Games. In *AIIDE*, 2011.

[32] Markus Enzenberger, Martin Muller, Broderick Arneson, and Richard Segal. FuegoâĂŤan open-source framework for board games and Go engine based on Monte Carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[33] Stefano Ermon, Carla P Gomes, Ashish Sabharwal, and Bart Selman. Taming the curse of dimensionality: Discrete integration by hashing and optimization. *Journal of Machine Learning Research*, 28:334–342, 2013.

[34] Susana Fernández, Ricardo Aler, and Daniel Borrajo. Machine learning in hybrid hierarchical and partial-order planners for manufacturing domains. *Applied Artificial Intelligence*, 19(8):783–809, 2005.

[35] Richard E Fikes and Nils J Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, 2(3-4):189–208, 1971.

[36] Andrew Garland and Neal Lesh. Learning hierarchical task models by demonstration. *Mitsubishi Electric Research Laboratory (MERL), USA–(January 2002)*, 2003.

[37] Andrew Garland, Kathy Ryall, and Charles Rich. Learning hierarchical task models by defining and refining examples. In *Proceedings of the 1st international conference on Knowledge capture*, pages 44–51. ACM, 2001.

[38] Sylvain Gelly and David Silver. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, 175(11):1856–1875, 2011.

[39] Melinda T Gervasio and Janet L Murdock. What were you thinking?: filling in missing dataflow through inference in learning from demonstration. In *Proceedings of the 14th international conference on Intelligent user interfaces*, pages 157–166. ACM, 2009.

[40] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[41] Robert P Goldman. A Semantics for HTN Methods. In *ICAPS*, 2009.

[42] Wen-Xiang Gu, Lei Wang, and Yong-li Li. Research for Adversarial Planning Recognition and Reply in the Complex Domains and the more Agents Conditions. In *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*, volume 1, pages 225–230. IEEE, 2005.

[43] Johan Hagelbäck. Potential-field based navigation in starcraft. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 388–393. IEEE, 2012.

[44] A Heinermann. BWAPI: Brood War API. http://code.google.com/p/bwapi/. Accessed: 2014.

[45] Hai Hoang, Stephen Lee-Urban, and Hector Muñoz-Avila. Hierarchical Plan Representations for Encoding Strategic Game AI. In *AIIDE*, pages 63–68, 2005.

[46] Chad Hogg, Ugur Kuter, and Hector Muñoz-Avila. Learning hierarchical task networks for nondeterministic planning domains. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[47] Chad Hogg, Ugur Kuter, and Hector Munoz-Avila. Learning Methods to Generate Good Plans: Integrating HTN Learning and Reinforcement Learning. In *AAAI*, 2010.

[48] Chad Hogg and Hector Munoz-Avila. Learning hierarchical task networks from plan traces. In *Proceedings of the ICAPS-07 Workshop on AI Planning and Learning*, 2007.

[49] Chad Hogg, Hector Munoz-Avila, and Ugur Kuter. HTN-MAKER: Learning HTNs with Minimal Additional Knowledge Engineering Required. In *AAAI*, pages 950–956, 2008.

[50] Jesse Hostetler, Ethan Dereszynski, Thomas Dietterich, and Alan Fern. Inferring Strategies from Limited Reconnaissance in Real-time Strategy Games. In *Proceedings of the Twenty-Eighth Conference Annual Conference on Uncertainty in Artificial Intelligence (UAI-12)*, pages 367–376, Corvallis, Oregon, 2012. AUAI Press.

[51] Jesse Hostetler, Ethan W Dereszynski, Thomas G Dietterich, and Alan Fern. Inferring strategies from limited reconnaissance in real-time strategy games. *arXiv preprint arXiv:1210.4880*, 2012.

[52] Okhtay Ilghami, Hector Munoz-Avila, Dana S Nau, and David W Aha. Learning approximate preconditions for methods in hierarchical plans. In *Proceedings of the 22nd international conference on Machine learning*, pages 337–344. ACM, 2005.

[53] Okhtay Ilghami, Dana S Nau, and Hector Munoz-Avila. Learning to Do HTN Planning. In *ICAPS*, pages 390–393, 2006.

[54] Okhtay Ilghami, Dana S Nau, Hector Munoz-Avila, and David W Aha. CaMeL: Learning method preconditions for HTN planning. In *AIPS*, pages 131–142, 2002.

[55] Ulit Jaidee and Hector Muñoz-Avila. Classq-l: A q-learning algorithm for adversarial real-time strategy games. In *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.

[56] Ulit Jaidee, Hector Muñoz-Avila, and David W Aha. Case-based learning in goal-driven autonomy agents for real-time strategy combat tasks. In *Proceedings of the ICCBR Workshop on Computer Games*, pages 43–52, 2011.

[57] Ulit Jaidee, Hector Muñoz-Avila, and David W Aha. Integrated learning for goal-driven autonomy. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Three*, pages 2450–2455. AAAI Press, 2011.

[58] Ulit Jaidee, Hector Muñoz-Avila, and David W Aha. Case-based goal-driven coordination of multiple learning agents. In *Case-Based Reasoning Research and Development*, pages 164–178. Springer, 2013.

[59] Hua Jiang, Yi-Qun Dong, and Qiu-Shi Zhao. A Counterplanning Approach Based on Goal Metric in the Adversarial Planning. In *Machine Learning and Cybernetics, 2006 International Conference on*, pages 78–84. IEEE, 2006.

[60] Paul E Johnson. What kind of expert should a system be? *Journal of Medicine and Philosophy*, 8(1):77–97, 1983.

[61] Randolph M Jones, John E Laird, Paul E Nielsen, Karen J Coulter, Patrick Kenny, and Frank V Koss. Automated intelligent pilots for combat flight simulation. *AI magazine*, 20(1):27, 1999.

[62] Hyuckchul Jung, James Allen, Lucian Galescu, Nathanael Chambers, Mary Swift, and William Taysom. Utilizing natural language for one-shot task learning. *Journal of Logic and Computation*, 18(3):475–493, 2008.

[63] Svenja Kahn, Tobias Klug, and Felix Flentge. Modeling temporal dependencies between observed activities. In *Proceedings of the 2007 workshop on Tagging, mining and retrieval of human related activity information*, pages 27–34. ACM, 2007.

[64] Henry A Kautz and James F Allen. Generalized Plan Recognition. In *AAAI*, volume 86, pages 32–37, 1986.

[65] Roni Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1):125–148, 1999.

[66] Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS Workshop on Probabilistic Programming: Foundations and Applications*, pages 1–4, 2012.

[67] Angelika Kimmig, Stephen H. Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *NIPS Workshop on Probabilistic Programming: Foundations and Applications*, 2012.

[68] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, page 201611835, 2017.

[69] Matthew Klenk, Matt Molineaux, and David W Aha. Goal-Driven Autonomy For Responding To Unexpected Events In Strategy Simulations. *Computational Intelligence*, 29(2):187–206, 2013.

[70] Stanley Kok and Pedro Domingos. Learning the structure of Markov logic networks. In *Proceedings of the 22nd international conference on Machine learning*, pages 441–448. ACM, 2005.

[71] Stanley Kok and Pedro Domingos. Learning Markov logic network structure via hypergraph lifting. In *Proceedings of the 26th annual international conference on machine learning*, pages 505–512. ACM, 2009.

[72] Stanley Kok and Pedro Domingos. Learning Markov logic networks using structural motifs. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 551–558, 2010.

[73] Tolga Könik and John E Laird. Learning goal hierarchies from structured observations and expert annotations. *Machine Learning*, 64(1):263–287, 2006.

[74] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[75] Ugur Kuter, Geoffrey Levine, Derek Green, Anton Rebguns, Diana Spears, and Gerald DeJong. Learning Constraints via Demonstration for Safe Planning. In *Proc. of the AAAI'07 Workshop on Acquiring Planning Knowledge via Demonstration*, 2007.

[76] John E Laird. *The Soar cognitive architecture*. MIT press, 2012.

[77] Pat Langley and Dongkyu Choi. A unified cognitive architecture for physical agents. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 1469. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[78] Pat Langley, Dongkyu Choi, and Seth Rogers. Acquisition of hierarchical reactive skills in a unified cognitive architecture. *Cognitive Systems Research*, 10(4):316–332, 2009.

[79] Karim Lari and Steve J Young. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer speech & language*, 4(1):35–56, 1990.

[80] Tessa Lau, Lawrence Bergman, Vittorio Castelli, and Daniel Oblinger. Sheepdog: learning procedures for technical support. In *Proceedings of the 9th international conference on Intelligent user interfaces*, pages 109–116. ACM, 2004.

[81] Chang-Shing Lee, Mei-Hui Wang, Guillaume Chaslot, J-B Hoock, Arpad Rimmel, F Teytaud, Shang-Rong Tsai, Shun-Chin Hsu, and Tzung-Pei Hong. The computational intelligence of MoGo revealed in Taiwan's computer Go tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(1):73–89, 2009.

[82] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 609–616. ACM, 2009.

[83] Michael Leece and Arnav Jhala. Opponent state modeling in RTS games with limited information using Markov random fields. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–7. IEEE, 2014.

[84] Michael Leece and Arnav Jhala. Sequential Pattern Mining in StarCraft: Brood War for Short and Long-Term Goals. page 8, 2014.

[85] Michael Leece and Arnav Jhala. Hypothesizing Method Structure for HTN Learning from Demonstration. In *Proceedings of the Fifth Annual Conference on Advances in Cognitive Systems (ACS)*, 2017.

[86] Michael A Leece and Arnav Jhala. Reinforcement Learning for Spatial Reasoning in Strategy Games. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[87] James C Lester and Brian A Stone. Increasing believability in animated pedagogical agents. In *Proceedings of the first international conference on Autonomous agents*, pages 16–21. ACM, 1997.

[88] Nan Li, Subbarao Kambhampati, and Sung Wook Yoon. Learning Probabilistic Hierarchical Task Networks to Capture User Preferences. In *IJCAI*, pages 1754–1759, 2009.

[89] Nan Li, David J Stracuzzi, Pat Langley, and Negin Nejati. Learning hierarchical skills from problem solutions using means-ends analysis. In *Proceedings of the 31st Annual Meeting of the Cognitive Science Society. Amsterdam, Netherlands: Cognitive Science Society, Inc*, 2009.

[90] Viliam Lisỳ, Branislav Bosanskỳ, Michal Jakob, and Michal Pechoucek. Adversarial search with procedural knowledge heuristic. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 899–906. International Foundation for Autonomous Agents and Multiagent Systems, 2009.

[91] Aaron B Loyall. Believable Agents: Building Interactive Personalities. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1997.

[92] Jan Malỳ. Learning to play real-time strategy games from demonstration using decentralized MAS. 2017.

[93] Mario Martín and Hector Geffner. Learning generalized policies in planning using concept languages. 2000.

[94] Mario MartWXn and H Wector Geffner. Learning generalized policies in planning using concept languages. 2000.

[95] Michael Mateas and Andrew Stern. A Behavior Language for Story-based Believable Agents. *IEEE Intelligent Systems*, 17(4):39–47, 2002.

[96] Michael Mateas and Andrew Stern. Façade: An experiment in building a fully-realized interactive drama. In *Game developers conference*, volume 2, pages 4–8, 2003.

[97] Michael Mateas and Andrew Stern. Integrating plot, character and natural language processing in the interactive drama Façade. In *Proceedings of the 1st International Conference on Technologies for Interactive Digital Storytelling and Entertainment (TIDSE-03)*, volume 2, 2003.

[98] Michael Mateas and Andrew Stern. A Behavior Language: Joint action and behavioral idioms. In *Life-Like Characters*, pages 135–161. Springer, 2004.

[99] Michael Mateas and Andrew Stern. Procedural authorship: A case-study of the interactive drama Façade. *Digital Arts and Culture (DAC)*, page 27, 2005.

[100] David McAllester and David Rosenblatt. Systematic nonlinear planning. 1991.

[101] Andrew McCallum, Karl Schultz, and Sameer Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, pages 1249–1257, 2009.

[102] Thomas Leo McCluskey, SN Cresswell, N Elisabeth Richardson, and Margaret M West. Automated acquisition of action knowledge. 2009.

[103] AB Meijer and H Koppelaar. Pursuing abstract goals in the game of Go. *BNAIC'01 Sponsors*, 2001.

[104] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[105] Shiwali Mohan and John E Laird. Learning Goal-Oriented Hierarchical Tasks from Situated Interactive Instruction. In *AAAI*, pages 387–394, 2014.

[106] Matthew Molineaux and David W Aha. Learning Unknown Event Models. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[107] Matthew Molineaux, David W Aha, and Philip Moore. Learning Continuous Action Models in a Real-Time Strategy Environment. In *FLAIRS Conference*, volume 8, pages 257–262, 2008.

[108] Joris M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, August 2010.

[109] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[110] Mark A Musen. Automated support for building and extending expert models. *Machine Learning*, 4(3-4):347–375, 1989.

[111] Mark A Musen, Lawrence M Fagan, David M Combs, and Edward H Shortliffe. Use of a domain model to drive an interactive knowledge-editing tool. *International Journal of Man-Machine Studies*, 26(1):105–121, 1987.

[112] Dana Nau, Yue Cao, Amnon Lotem, and Hector Muñoz-Avila. SHOP: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2*, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[113] Dana Nau, Hector Munoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-order planning with partially ordered subtasks. In *IJCAI*, volume 1, pages 425–430, 2001.

[114] Dana S Nau, Stephen JJ Smith, Kutluhan Erol, et al. Control strategies in HTN planning: Theory versus practice. In *AAAI/IAAI*, pages 1127–1133, 1998.

[115] Negin Nejati, Pat Langley, and Tolga Konik. Learning hierarchical task networks by observation. In *Proceedings of the 23rd international conference on Machine learning*, pages 665–672. ACM, 2006.

[116] Nils Nilsson. Teleo-reactive programs for agent control. *Journal of artificial intelligence research*, 1:139–158, 1993.

[117] Richard E Nisbett and Timothy D Wilson. Telling more than we can know: verbal reports on mental processes. *Psychological review*, 84(3):231, 1977.

[118] Santi Ontañón, Kinshuk Mishra, Neha Sugandh, and Ashwin Ram. ON-LINE CASE-BASED PLANNING. *Computational Intelligence*, 26(1):84–119, 2010.

[119] Santiago Ontañón, Kane Bonnette, Prafulla Mahindrakar, Marco A Gómez-Martín, Katie Long, Jainarayan Radhakrishnan, Rushabh Shah, and Ashwin Ram. Learning from human demonstrations for real-time case-based planning. 2009.

[120] Santiago Ontañón and Michael Buro. Adversarial hierarchical-task network planning for complex real-time games. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015.

[121] Santiago Ontañón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. 2013.

[122] OpenAI. Openai Five. `https://blog.openai.com/openai-five/`. Accessed: 2018-08-07.

[123] Ricardo Palma, Antonio A Sánchez-Ruiz, Marco Antonio Gómez-Martín, Pedro Pablo Gómez-Martín, and Pedro Antonio González-Calero. Combining expert knowledge and learning from demonstration in real-time strategy games. In *Case-Based Reasoning Research and Development*, pages 181–195. Springer, 2011.

[124] Alberto Uriarte Pérez and S Villar. Multi-Reactive Planning for Real-Time Strategy Games. *M. Sc. Report. Universit Autònoma de Barcelona, Spain*, 2011.

[125] Sean Plott. Day9 tv - be a better gamer. `https://www.day9.tv/`. Accessed: 2018-08-01.

[126] Marc Ponsen, Hector Munoz-Avila, Pieter Spronck, and David W Aha. Automatically generating game tactics through evolutionary learning. *AI Magazine*, 27(3):75, 2006.

[127] Marc Ponsen, Pieter Spronck, Hector Munoz-Avila, and David W Aha. Knowledge acquisition for adaptive game AI. *Science of Computer Programming*, 67(1):59–75, 2007.

[128] Marc Ponsen, Pieter Spronck, and Karl Tuyls. Hierarchical reinforcement learning in computer games. *Adaptive Learning Agents and Multi-Agent Systems*, pages 49–60, 2006.

[129] Chandra Reddy and Prasad Tadepalli. Learning goal-decomposition rules using exercises. In *AAAI/IAAI*, page 843, 1997.

[130] Florian Richoux, Jean-François Baffier, and Alberto Uriarte. Ghost: A combinatorial optimization solver for rts-related problems. 2015.

[131] AJ Riopelle. Observational learning of a position habit by monkeys. *Journal of comparative and physiological psychology*, 53(5):426, 1960.

[132] Mark Roberts, Swaroop Vattam, Ronald Alford, Bryan Auslander, Justin Karneeb, Matthew Molineaux, Tom Apker, Mark Wilson, James McMahon, and David W Aha. Iterative goal refinement for robotics. Technical report, KNEXUS RESEARCH CORP SPRINGFIELD VA, 2014.

[133] Glen Robertson and Ian Watson. A Review of Real-Time Strategy Game AI. 2014.

[134] Richard Rörmark. *Thanatos-a learning RTS Game AI*. PhD thesis, Master Thesis, University of Oslo, 2009.

[135] Stuart Russell and Peter Norvig. AI a modern approach. *Learning*, 2(3):4, 2005.

[136] Frederik Schadd, Sander Bakkes, and Pieter Spronck. Opponent Modeling in Real-Time Strategy Games. In *GAMEON*, pages 61–70, 2007.

[137] Amirhosein Shantia, Eric Begue, and Marco Wiering. Connectionist reinforcement learning for intelligent unit micro management in starcraft. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1794–1801. IEEE, 2011.

[138] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[139] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354, 2017.

[140] Christopher Simpkins, Sooraj Bhat, Charles Isbell Jr, and Michael Mateas. Towards adaptive programming: integrating reinforcement learning into a programming language. *ACM Sigplan Notices*, 43(10):603–614, 2008.

[141] Dustin Smith and Kenneth C Arnold. Learning hierarchical plans by reading simple english narratives. In *Proceedings of the Commonsense Workshop at IUI-09*. Citeseer, 2009.

[142] Stephen J Smith, Dana Nau, and Tom Throop. Computer bridge: A big win for AI planning. *Ai magazine*, 19(2):93, 1998.

[143] Ramakrishnan Srikant and Rakesh Agrawal. *Mining sequential patterns: Generalizations and performance improvements*. Springer, 1996.

[144] Marius Stanescu, Nicolas A Barriga, and Michael Buro. Hierarchical adversarial search applied to real-time strategy games. In *Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2014.

[145] John Douglas Sweeney. A teleological approach to robot programming by demonstration. 2011.

[146] Gabriel Synnaeve, Pierre Bessiere, et al. A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. In *AIIDE*, 2011.

[147] Austin Tate. *Project planning using a hierarchic non-linear planner*. Department of Artificial Intelligence, University of Edinburgh, 1976.

[148] TeamLiquid. Liquipedia. `https://www.liquipedia.net/`. Accessed: 2018-08-01.

[149] TeamLiquid. Starcraft esports news and community. `http://www.teamliquid.net/`. Accessed: 2018-08-01.

[150] Alex Turner. Soar-sc. `https://github.com/bluechill/Soar-SC`, 2013.

[151] Gerrit C Van Der Veer, Bert F Lenting, and Bas AJ Bergevoet. GTA: Groupware task analysisâĂŤModeling complexity. *Acta Psychologica*, 91(3):297–322, 1996.

[152] Michael van Lent and John E Laird. Learning procedural knowledge through observation. In *Proceedings of the 1st international conference on Knowledge capture*, pages 179–186. ACM, 2001.

[153] Swaroop S Vattam, David W Aha, and Michael Floyd. Case-based plan recognition using action sequence graphs. In *Case-Based Reasoning Research and Development*, pages 495–510. Springer, 2014.

[154] Manuela Veloso, Jaime Carbonell, Alicia Perez, Daniel Borrajo, Eugene Fink, and Jim Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):81–120, 1995.

[155] Ben Weber. *Integrating learning in a multi-scale agent*. PhD thesis, UC Santa Cruz, 2012.

[156] Ben G Weber, Michael Mateas, and Arnav Jhala. Case-based goal formulation. In *Proceedings of the AAAI Workshop on Goal-Driven Autonomy*, 2010.

[157] Ben G Weber and Santiago Ontañón. Using automated replay annotation for case-based planning in games. In *International Conference on Case-based Reasoning Workshop on CBR for Computer Games*, pages 15–24. Citeseer, 2010.

[158] Ben George Weber and Michael Mateas. A data mining approach to strategy prediction. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 140–147. IEEE, 2009.

[159] Ben George Weber, Michael Mateas, and Arnav Jhala. Applying Goal-Driven Autonomy to StarCraft. In *AIIDE*, 2010.

[160] Ben George Weber, Michael Mateas, and Arnav Jhala. A Particle Model for State Estimation in Real-Time Strategy Games. In *AIIDE*, 2011.

[161] Ben George Weber, Michael Mateas, and Arnav Jhala. Learning from Demonstration for Goal-Driven Autonomy. In *AAAI*, 2012.

[162] Ben George Weber, Peter Mawhorter, Michael Mateas, and Arnav Jhala. Reactive planning idioms for multi-scale game AI. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 115–122. IEEE, 2010.

[163] Stefan Wender and Ian Watson. Applying reinforcement learning to small scale combat in the real-time strategy game StarCraft: Broodwar. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 402–408. IEEE, 2012.

[164] David E Wilkins, Karen L Myers, John D Lowrance, and Leonard P Wesley. Planning and reacting in uncertain and dynamic environments. *Journal of Experimental & Theoretical Artificial Intelligence*, 7(1):121–152, 1995.

[165] Steven Willmott, Alan Bundy, John Levine, and Julian Richardson. Adversarial planning in complex domains. *DAI RESEARCH PAPER*, 1998.

[166] Steven Willmott, Julian Richardson, Alan Bundy, and John Levine. An adversarial planning approach to Go. In *Computers and Games*, pages 93–112. Springer, 1999.

[167] Qiang Yang, Rong Pan, and Sinno Jialin Pan. Learning recursive HTN-method structures for planning. In *Proceedings of the ICAPS-07 Workshop on AI Planning and Learning*, 2007.

[168] Hankz Hankui Zhuo, Derek Hao Hu, Chad Hogg, Qiang Yang, and Hector Munoz-Avila. Learning HTN Method Preconditions and Action Models from Partial Observations. In *IJCAI*, pages 1804–1810, 2009.

[169] Hankz Hankui Zhuo, Derek Hao Hu, Qiang Yang, Hector Munoz-Avila, and Chad Hogg. Learning applicability conditions in AI planning from partial observations. In *Workshop on Learning Structural Knowledge From Observations at IJCAI*, volume 9, 2009.