INTEGRATING DETERMINISTIC PLANNING AND REINFORCEMENT LEARNING FOR COMPLEX SEQUENTIAL DECISION MAKING

by

Tim Ernsberger

Submitted in partial fulfillment of the requirements
For the degree of Master of Science

Thesis Advisor: Dr. Soumya Ray

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

January 2013

CASE WESTERN RESERVE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

We hereby approve the thesis of

Tim Ernsberger

candidate for the Master of Science degree*.

(signed) Soumya Ray

(chair of the committee)

H. Andy Podgurski

Michael Lewicki

(date) November 30, 2012

*We also certify that written approval has been obtained for any proprietary material contained therein.

TABLE OF CONTENTS

List of Algorithms

LIST OF TABLES

LIST OF FIGURES

Integrating Deterministic Planning and Reinforcement Learning for Complex Sequential Decision Making

Abstract

by

TIM ERNSBERGER

This thesis presents a novel approach to solving decision-making problems in discrete, stochastic domains. The method for solving these problems is often dictated by the availability of information about how the environment responds to actions taken by the agent. When the agent is given a model of the environment, it can plan out its actions beforehand, whereas an agent without a model must learn to differentiate good and bad decisions through direct experience. Until now, little attention has been paid to situations in which a model is only available for a part of the environment. We propose an algorithm which combines automated planning with hierarchical reinforcement learning in order to take advantage of the model when it is available and sample from the environment when it is not. We prove that the same guarantees of optimality that apply to hierarchical reinforcement learning also apply to to this approach. Using experiments performed in two different domains, we demonstrate that hierarchically integrated planning and reinforcement learning outperforms pure RL and pure planning hierarchies and that this approach can scale to larger problems than are reasonably computable by other approaches.

Chapter 1

Introduction

Artificial intelligence (A.I.) is becoming increasingly prevalent in our society. While we still do not have artificial sentience, advances in A.I. algorithms and computer hardware have made possible projects like IBM's Watson [1] and Apple's Siri [2], which can understand human language and respond with relevant information. Such applications help improve awareness and acceptance in society, in addition to assisting people in tasks ranging from keeping track of their meeting schedule to performing medical research.

However, understanding natural language and answering queries are only part of the overall purview of artificial intelligence. Another important sub-field of A.I. is sequential decision-making. Life does not just consist of "one and done" problems where the solution consists of a single answer or action. Humans perform sequential decision-making constantly, whether they are going to work, preparing a meal, or playing a game. Imagine how impractical it would be to figure out how to get to work by looking at every possible route that goes from your house to your office and picking one based off the myriad attributes of each possibility. The natural approach is to figure out where you want to go, determine which roads go in that direction using a map, and put together a sequence of turns that take you to your destination. Along the way, you may discover that everyone else has decided on the same sequence or that construction has blocked your way. Through the experience of driving to work, you learn all the little nuances and day-to-day variations that no map would be able to tell you, and as a result find a better way to commute. These are two ways of making decisions which we study in this thesis.

[1] http://watson.ibm.com
[2] http://www.apple.com/ios/siri

The approaches mentioned above correspond to two different categories of decision-making algorithms: model-based and model-free. In the first case, the decision-maker (the intelligent ”agent”) has an explanation of how the actions it takes affect its situation. In the example of commuting to work, the map serves as a model. It shows you exactly how all the roads connect, so you can trace out a route and know exactly what actions to take before you hit the road.

Unfortunately, not every problem is easy to model or even possible to model before directly trying to solve it. This occurs commonly in competitive or adversarial situations. In the example of commuting to work, there are many other people on the road, each making their own decisions. While you can be reasonably certain they are also solving the same general problem as you, their goals most likely involve going to different places. Additionally, you may get held up by a nice person letting cars merge in ahead of them or an inconsiderate person blocking an intersection so they don't have to wait for the next light. Unless you live in a very small town, odds are you do not know this person and cannot model their thought processes. In this situation, you must learn how to anticipate seemingly erratic driving, common points of congestion, and myriad other obstacles to getting to work on time.

Both of these types of solutions have their strengths and weaknesses. Model-based decision-making is similar to solving a problem in one's head, meaning that the agent does not need hands-on experience before it can adequately perform a task. The downside is that this only works in situations where an accurate model is available. Model-free decision-making can solve problems regardless of the availability of a model, but it can take a lot more time, scrapes, and bruises to learn the right way to do things. Naturally, it would be advantageous to combine the two. Research to this end for A.I. agents have primarily focused on using a model to guide learning [18][20] or learning to build a model [4]. The thesis takes a different approach. Instead of providing information to each other, we propose that these two methodologies can benefit from solving specific parts of problems and trusting the other approach to solve other parts for which it would be more suitable.

Some problems are big enough to merit being broken down into separate tasks. The earlier example of driving to work is fairly straightforward to solve as a single task, but what about the rest of the day? Doing work is dependent on getting to work, but the individual decisions made during each part of the day aren't always relevant in considering other actions. You don't need to figure out when to put dinner in the oven in order to determine whether to turn on first avenue or second avenue, and vice versa. Humans intuitively separate sequences of actions into tasks, which roll up into other tasks, and so on until it all converges in the task of living. We use this notion of a *hierarchy* of tasks to combine model-based and model-free decision-making: solve tasks with a model when we can, and learn how to do them when we can't.

The main contribution of this thesis is the introduction of a novel approach to combining model-based and model-free algorithms in a hierarchical framework. We demonstrate both through theoretical proof and multiple experiments that this new approach performs at least as well as existing approaches, while being able to solve larger problems with less resources. Additionally, as the new approach incorporates multiple methods of solving decision-making problems, we offer improvements on several algorithms that came as a result of designing an agent to test our approach.

The remainder of this thesis is organized as follows. Chapter two introduces a framework for decision-making problems known as Markov decision processes and common algorithms that use this framework. Chapter three presents previous work on combining model-based and model-free decision-making and our new approach. The fourth chapter contains various improvements we have made to some of the algorithms mentioned in chapter two. Chapter five contains the experiments we performed to test our new approach in practice. Finally, the last chapter summarizes the contributions and results in this work and outlines directions for future research.

Chapter 2

Background

In order to solve a problem, we must first have a way of representing it in a way that an agent can understand. To this end, we represent problems in this domain as Markov decision processes (MDPs). Based on the Markov property, this approach simplifies the considerations the agent has to make about the actions it takes, leading to a number of powerful methods for solving problems expressed in this manner. This chapter gives a brief description of Markov decision processes and various algorithms which use them. For a more in-depth discussion of MDPs, we recommend the textbook written by Sutton and Barto [21] or Richard Bellman's paper on dynamic programming solutions for MDPs [2].

2.1 Markov Decision Process

The Markov decision process formalism is a simple yet flexible way of expressing sequential decision making problems that is often used in artificial intelligence research. This formulation is an adaptation of a basic Markov process, which consists of discrete states and the probabilities of transitioning from one to another. What makes the process Markovian is that the transition probabilities are purely dependent on a finite history, usually taken to be just the current state. In this way, Markov processes are essentially non-deterministic finite state machines with specific information about the transitions. The decision component of the Markov decision process adds actions. Instead of being a passive observer, the agent is an active participant that determines which probability distribution determines the next transition.

Definition 2.1 A Markov Decision process is represented as a tuple $\langle S, A, T, R\rangle$, consisting of:

- S, a set of states,
- A, a set of actions,
- $T(s' \mid s, a)$, a function that determines the probability of ending up in state s' after taking action a in state s, and
- $R(s, a)$, a real-valued function that provides feedback on an action in the form of a reward.

Definition 2.2 A *policy* $\pi(s) : S \mapsto A$ is a function that an intelligent agent uses to determine what action to take in a given state.

To measure the *utility* of a state with respect to a specific policy, i.e. how useful it is to be in the state with regard to obtaining future rewards, we use the equation:

$$V^\pi(s_0) = E(r_0 + \gamma r_1 + \gamma^2 r_2 + ...|s_0, \pi) = E(\sum_{i=0}^{\infty} \gamma^i R(s_i, \pi(s_i))) \tag{2.1}$$

Because there is no guarantee that the series of rewards received will be finite, problems usually have a discount factor $\gamma \in [0, 1]$. If $\gamma < 1$ and there exists some upper bound on the reward function such that $\max_{s,a}(R(s, a)) < \infty$, then the sum of the series is finite. The discount factor is similar to the notion of immediate gratification; receiving a reward now is considered to be better than receiving an equivalent reward in the future. An important thing to note is that because this is an infinite sequence and distant rewards have negligible impact on the value of the current state, the sequence effectively contains the same sequence that would be used to evaluate s_1, s_2, etc. Therefore, we can rewrite the value function of a given state in terms of the value function of its neighbors:

$$V^\pi(s) = R(s, \pi(s)) + \gamma \sum_{s' \in S} V^\pi(s')T(s'|s, \pi(s)) \tag{2.2}$$

This equation is known as the Bellman equation, named after Richard Bellman, who first proposed this means of representing state value in his paper on dynamic programming for decision problems [2]. The theoretical maximum value that can be obtained for a given state is given by:

$$V^*(s) = \max_{a \in A}(R(s,a) + \gamma \sum_{s' \in S} V^*(s')T(s'|s,a)). \tag{2.3}$$

The optimal policy is one which produces the best action for each state according to the optimal value function:

$$\pi^*(s) = \arg\max_{a \in A}(R(s,a) + \gamma \sum_{s' \in S} V^*(s')T(s'|s,a)). \tag{2.4}$$

Although Markov decision processes are restrictive in their mechanics, the overall representation is very flexible. For example, processes for which the transition mechanics do rely on a finite history of previous states can still be represented as MDPs by re-defining the domain of S to be the cross product of some number of primitive states. Additionally, while the transition function assumes that transitions happen if and only if an action is taken, the addition of a ”no-op” action allows for modeling dynamic environments (such as ones with multiple independent agents). MDPs can also be adapted to handle durative actions. While it may seem unwieldy to have to redefine a problem in these ways, it also extends certain benefits due to the well-understood nature of Markov processes and strong guarantees on the convergence of algorithms which use the Bellman equation.

2.2 Reinforcement Learning

Reinforcement learning (RL) is an approach to solving decision-making problems that does not require a model, i.e. the transition and reward functions do not need to be specified in advance. RL stems from two model-based control algorithms, value iteration and policy iteration, which are dynamic programming approaches to solving Markov Decision processes [21, pp. 89,97-102]. These approaches worked by keeping estimates of the value function and performing updates using the Bellman equation until either the values or policy converge to V^* or π^*, respectively. In the mid 1980s, a new approach to solving decision processes was introduced called temporal difference (TD) learning [19]. While TD relies on the same value function as the previously mentioned dynamic programming algorithms, it does not require a model of the environment. Instead, it uses Monte Carlo sampling to gradually reduce the error between what it believes to be the value of each state and what the value actually is.

Two implementations of temporal difference learning for optimal control, Q-learning [23] and SARSA [17], have become popular model-free RL algorithms since their introductions in 1989 and 1994, respectively. This thesis makes use of these two algorithms and their derivatives. As opposed to the original TD algorithm which calculates values for states, these *control* algorithms calculate the quality of state-action pairs:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} V^{\pi}(s')T(s'|s,a'). \tag{2.5}$$

Note the similarity between the control function in equation 2.5 and the value function in equation 2.2. The following equivalence relates the two:

$$V^{*}(s) = \max_{a \in A}(Q^{*}(s,a)). \tag{2.6}$$

The goal of reinforcement learning can be restated as the search for the policy $\pi^{*}(s)$ such that:

$$\forall s(Q(s,\pi^{*}(s)) = \max_{a \in A} Q(s,a)). \tag{2.7}$$

Q-learning and SARSA use Monte Carlo sampling and weighted updates to sample state-action pairs and estimate their value. Before discussing the method by which they determine the optimal policy, it is important to explain how they gather samples. Q-learning and SARSA are *online* algorithms, meaning that they learn from real experience, as opposed to being given training samples or a model. As such, it is important to balance sample gathering with performance. The implementations of both algorithms used in this thesis tend to prefer exploiting current estimates of the optimal policy, while with a certain probability they will explore a random action. The probability function varies with the problem, but in our experiments these functions are fixed, uniform probability distributions or ones that decrease linearly with respect to the amount of experience acquired. This approach is known as ϵ-greedy exploration. Other action selection strategies include the softmax function and the Boltzmann distribution, which weight options based on the current value estimate.

2.2.1 Q-learning and SARSA

Q-learning, as the name suggests, finds the optimal policy by maintaining and revising an estimate of $Q(s, a)$ for all $\langle s, a\rangle \in S \times A$. Each time a Q-learning agent observes a state transition $\langle s, a, r, s'\rangle$, it performs the following update:

$$Q(s,a)_{t+1} = (1-\alpha) * Q(s,a)_t + \alpha * (R(s,a) + \gamma \max_{a' \in A} Q_t(s', a')), \tag{2.8}$$

where $\alpha \in [0, 1]$ is the learning rate, which balances the value of new information with accumulated experience. In essence, the update function reduces the error in the agent's estimate of the state-action pair's worth by incorporating the reward it observes and the value of the next state it observes. SARSA differs from Q-learning in that it is *on-policy* instead of *off-policy*. The distinction lies in how they perform updates to the Q function: on-policy means that a' is the action chosen by the exploratory policy, whereas off-policy means that a' is the hypothetical best action for the next state. The update equation for SARSA uses a tuple $\langle s, a, r, s', a'\rangle$ in which a' is the action it took, rather than the argmax of the Q function for s':

$$Q(s,a)_{t+1} = (1-\alpha) * Q(s,a)_t + \alpha * (R(s,a) + \gamma Q_t(s', a')). \tag{2.9}$$

Although these algorithms can be used interchangeably, the Q-learning update calculates $\max_{a' \in A} Q(s, a)$ at every step, which can be costly in large action spaces. When paired with an exploration policy such as ϵ-greedy that selects exploratory actions purely at random, SARSA saves agents from calculating the maximum Q value in approximately ϵ percent of all steps, resulting in an appreciable speedup.

2.2.2 Scalability Issues

While temporal difference learning algorithms such as Q-learning and SARSA are only guaranteed to converge to the optimal policy when given infinitely many samples, a high degree of confidence can be achieved with a limited number of samples, given that all state-action pairs

Algorithm 1 Q-Learning and SARSA

```
Require: $Q(s,a)$ is initialized to random values, $\pi(s)$ is a policy that incorporates exploration
  (e.g. $\epsilon$-greedy)
  function Q-LEARNING
      for each episode do
          $s \leftarrow getState()$
          $a \leftarrow \pi(s)$
          while $s$ is not terminal do
              $(s', r) \leftarrow observeTransition(s,a)$
              $\delta \leftarrow \alpha(r + \max_{a' \in A} Q(s',a') - Q(s,a))$
              $Q(s,a) \leftarrow Q(s,a) + \delta$
              $s \leftarrow s', a \leftarrow \pi(s)$
          end while
      end for
  end function
  function SARSA
      for each episode do
          $s \leftarrow getState()$
          $a \leftarrow \pi(s)$
          while $s$ is not terminal do
              $(s', r) \leftarrow observeTransition(s,a)$
              $a' \leftarrow \pi(s)$
              $\delta \leftarrow \alpha(r + Q(s',a') - Q(s,a))$
              $Q(s,a) \leftarrow Q(s,a) + \delta$
              $s \leftarrow s', a \leftarrow a'$
          end while
      end for
  end function
```

are explored. In other words, the reliability of a TD agent is proportional to the size of $S \times A$. This results in what is commonly referred to as the ”curse of dimensionality.” In most problems, states and actions can be represented as the combination of multiple features that each describe an element of the environment. Without any prior knowledge about relations between these features, representations must assume that all combinations of all features are significant, or else risk over-generalizing and arriving at a sub-optimal result. As such, the size of the problem representation grows exponentially with respect to the number of features. The time cost also scales exponentially since each unique feature combination must be sampled a certain number of times. The common solutions to this problem involve either partitioning the problem into smaller sub-problems (discussed later) or by representing the Q function as a weighted sum of multiple functions that deal with smaller portions of the state-action space.

Function approximation is one way to combat the curse of dimensionality. The exponential growth of the problem representation is based on the assumption that the reward and all features of s' are affected by all features of s and a. However, this is not always the case. In many instances, at least some features are independent or weakly dependent. As such, we can use alternative representations such as linear function approximation:

$$Q(s, a) = w \cdot \phi(s, a), \tag{2.10}$$

where w is a set of weights and ϕ is a vector of functions based on the features present in s and a. The update equation for the Q function becomes:

$$w_{t+1} = w_t + \alpha(r + \gamma w_t \cdot \phi(s', a') - w_t \cdot \phi(s, a))\phi(s, a) \tag{2.11}$$

The basic, tabular way of representing $Q(s, a)$ as a mapping $S, A \mapsto \mathbb{R}$ is equivalent to a linear function where each element of $\phi(s, a)$ is an indicator variable $\in \{0, 1\}$ that is 1 for exactly 1 state and 0 for all others. In this case, w would be identical to the Q function where the vector index is equivalent to some arbitrary ordering on state-action pairs. The function $\phi(s, a)$ can be any mapping of the state-action space to a real-valued vector, leading to arbitrarily compact representations. Unfortunately, the guarantee of convergence to the globally optimal policy may be lost for any representation that does not assign unique weights to each unique state-action pair.

2.2.3 Hierarchical Reinforcement learning

Hierarchical reinforcement learning (HRL) methods were devloped starting in the late 1990s. Some notable methodologies for HRL include options [22] and hierarchical abstract machines (HAMs) [15], but the work in this thesis is based on the MAXQ hierarchy proposed by Dietterich shortly after [6]. The core idea of HRL is that some problems can be treated as multiple related problems of different levels of detail. The bottom level consists of problems that involve primitive actions, while the upper levels deal with abstract actions that correspond to selecting which problem on the level below to solve. If designed correctly, each problem only has to deal with parts of the state space or a subset of the state variables.

Dietterich provides a good example of the advantages of hierarchical decomposition in his taxi domain, which appears in figure 2.1. The state space consists of the location of the taxi, the location of the passenger, and whether the passenger is in the taxi. The primitive action space consists of moving in cardinal directions and picking up or dropping off the passenger. An agent that considers all actions in all states can solve this problem, but many of these considerations are unnecessary, such as trying to pick up the passenger when it is already in the taxi, or moving to random corners of the map. Dietterich separates the problem into multiple tasks which use combinations of primitive actions and/or other tasks to accomplish specific parts of the overall objective. A diagram of the task hierarchy appears in figure 2.2. The top task, root, uses calls to get and put, which in turn can call pickup and putdown, respectively, or use navigate to control movement. This decomposition is both straightforward and effective in reducing the number of unique combinations of state and action that the agent must consider. In an $m \times n$ word with l possible passenger locations, the full state-action space is $m \times n \times l^2 \times 2 \times 6$, while the previously mentioned hierarchy consists of separate problems of $m \times n \times 6$, $l \times (l+1) \times 2$, $l \times (l+1) \times 2$, and 2×2 [6, pp. 15-17].

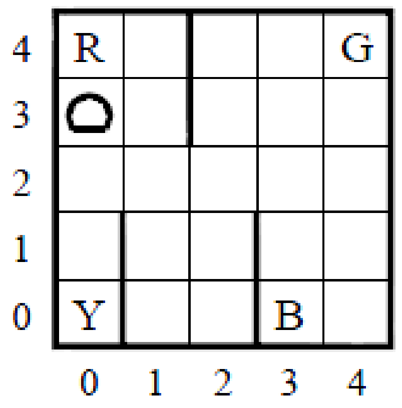


Figure 2.1: The taxi domain used by Dietterich as a motivating example for hierarchical reinforcement learning.

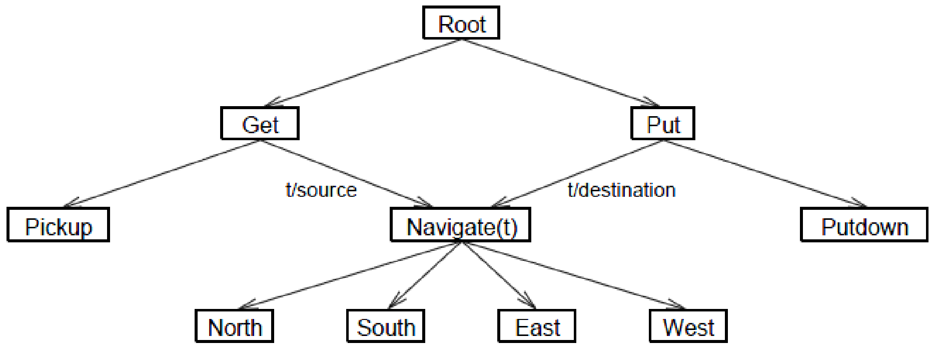


Figure 2.2: The task hierarchy used to solve the taxi domain. The root task is responsible for calling the get and put tasks as appropriate, which in turn can call primitive actions or the parametrized task navigate.

The MAXQ algorithm works similar to most modern programming languages in that it treats each separate node in the hierarchy like a function that may be called by its parents. Parent nodes choose which child node and which parameters to provide it and places that information on an execution stack. A node has a set of terminal states (similar to return statements); it will continue executing primitive actions or making calls to sub-tasks until it reaches one. Because it is possible (and expected) that a child task will perform multiple actions for each call from a parent, our prior assumption that state transitions only depend on the current state and action no longer hold from the perspective of nodes with non-primitive children. To a parent node, calling a child node appears as if it is performing one atomic action that results in multiple transitions between states. As a result, it becomes necessary to model the sub-problems given to tasks as *semi*-Markov decision processes (SMDPs), in which each action can last for a variable number of time steps. The definition of a semi-Markov decision process is:

- S: a set of states
- A: a set of actions
- $T(s', N|s, a)$: a function that determines the probability of ending up in state s' after N steps by taking action a in state s
- $R(s, N, a)$: a real-valued function that provides feedback on actions

The Bellman equation for an SMDP then becomes:

$$V^{\pi}(s) = R(s, \pi(s)) + \sum_{s' \in S, N} (\gamma^N V^{\pi}(s') T(s', N|s, a)) \tag{2.12}$$

The equations for calculating the Q-value of a state-action pair work similarly [6, p. 7].

Algorithm 2 The MaxQ-Q Algorithm

```
Require: $n$ is a MAXQ Node, $s$ is a State
  function MAXQ-Q($n, s$)
      $seq \leftarrow new\ List\ of\ State$
      if n is primitive then
          $(s', r) \leftarrow execute(n)$
          $V(n, s) \leftarrow (1 - \alpha_t(n)) * V(n, s) + \alpha_t(n) * r$
          push($seq$,s)
      else
          while $s \notin terminalStates(n)$ do
              $a \leftarrow \pi(n, s)$                      ▷ use policy to determine child node to call
              $childSeq \leftarrow MAXQ\text{-}Q(a, s)$        ▷ recurse, executing child node
              $s' \leftarrow lastElementOf(childSeq)$
              $a^* \leftarrow \arg\max_{a'}(\tilde{C}(n, s', a') + V(a', s))$     ▷ determine next optimal action
              $N \leftarrow length(childSeq)$
              for $State\ s_c\ in\ childSeq$ do
                  $\tilde{C}(n, s_c, a) \leftarrow (1 - \alpha(n)) * \tilde{C}(n, s_c, a) + \alpha(n) * \gamma^N * (\tilde{R}(s') + \tilde{C}(n, s', a') + V(a^*, s_c))$     ▷ update pseudo-completion function with pseudo-reward and Q(s',a')
                  $C(n, s_c, a) \leftarrow (1 - \alpha(n)) * C(n, s_c, a) + \alpha(n) * \gamma^N * (C(n, s', a') + V(a^*, s_c))$     ▷ update completion function with Q(s',a')
                  $N \leftarrow N - 1$
              end for
              $append(seq, childSeq)$
              $s \leftarrow s'$
          end while
      end if
      return $seq$
  end function
```

Algorithm 3 Value computations used by MAXQ-Q

```
function V($n, s$)
    if n is primitive then
        return $V(n, s)$                    ▷ stored value, not recursive call
    else
        return $\max_a Q(n, s, a)$
    end if
end function
function Q($n, s, a$)
    return $V(a, s) + C(n, s, a)$
end function
```

Algorithm 2 contains pseudocode for the MAXQ algorithm. The main function, MAXQ-Q, works differently depending on whether the node represents a task or a primitive action. In the latter case, the node simply executes the one action it has and updates its estimate of $V^{\pi}(s)$, where $\pi(s)$ is to always take the one action assigned to the node. Here Dietterich assumes that the reward function is stochastic, and that the actual value of the state-action pair only becomes apparent after taking multiple samples. If the reward function is known to be deterministic, the primitive node need only execute its action and return the resulting state, while its value function is fixed as being equal to the first reward it observes. The other case, where the node represents an abstract task, will make recursive calls to the MAXQ-Q function to execute child nodes until it reaches one of the states in its set of terminal states. After executing a child node, the function iterates over each state that was encountered by the child node, making updates to the value functions $C(parent, state, child)$ and $\tilde{C}(parent, state, child)$ according to the semi-Markovian variation on the Bellman equation. After reaching a terminal state, the function returns a sequence of all states encountered while executing the task so that the value function can be updated for the parent node that initiated the task.

One thing to note is that the nodes of a MAXQ hierarchy do not store Q-values directly. Instead, they use a combination of a *completion function* $C(n, s, a)$ and a recursive function

$V(n,s)$. The update for the completion function is similar to updating the Q function, with the key differences being that it does not store the part calculated by the child action and that the child's value function also acts as a substitute for the reward function:

$$\begin{aligned}
C(n,s,a)_{t+1} &= (1-\alpha_t)C(n,s,a)_t + \alpha\gamma(C(n,s',a')_t + V(a',s)_t) \\
&= (1-\alpha_t)C(n,s,a)_t + \alpha\gamma Q(n,s',a')_t \\
&= (1-\alpha_t)(C(n,s,a)_t + V(a,s)_t) + \alpha(V(a,s)_t + \gamma Q(n,s',a')_t) - V(a,s)_t \\
&= (1-\alpha_t)Q(n,s,a)_t + \alpha(V(a,s)_t + \gamma Q(n,s',a')_t) - V(a,s)_t \\
C(n,s,a)_{t+1} &+ V(a,s)_t = (1-\alpha_t)Q(n,s,a)_t + \alpha(V(a,s)_t + \gamma Q(n,s',a')_t) \\
vs. & \\
Q(n,s,a)_{t+1} &= (1-\alpha_t)Q(n,s,a)_t + \alpha(R(s,a) + \gamma Q(n,',a')_t) \qquad (2.13)
\end{aligned}$$

The optimality of the MAXQ algorithm relies on the assumption that primitive nodes are optimal and that the completion and pseudo-completion function updates converge iff the value functions of the node's children also converge. Based on the fact that the value of primitive nodes is just the reward function, it is easy to see from the comparison of the completion and Q functions in equation 2.13 that the completion function for a node will converge in the same manner as the Q function would as long as the completion function of each non-primitive node lower in the hierarchy is also optimal.

In addition to the completion function, there is a pseudo-completion function $\tilde{C}(n,s,a)$, which relies on an additional pseudo-reward $\tilde{R}(s)$ that augments the child's value function. MAXQ uses the pseudo-completion function for greedy policy decisions, which allows each node to focus on solving localized problems. This feature is important for state abstraction, which is a key benefit of hierarchical decomposition [6, pp. 18, 29-31]. The trade off made by using the pseudo-completion function for policy-making is that the solution produced by MAXQ is only guaranteed to be recursively optimal, rather than hierarchically optimal.

Definition 2.3 A policy is *recursively optimal* if it achieves the highest cumulative sum of the actual and pseudo-reward functions for each node in the task hierarchy.

This is a slightly weaker guarantee than that of a hierarchically optimal policy:

Definition 2.4 A policy is *hierarchically optimal* if it achieves the highest cumulative reward of all policies that can be represented by the given hierarchy.

Pseudo-rewards are meant to direct the policy of a node toward actions which solve the given task, but risk the possibility of biasing the agent toward local optima [6, p. 18].

2.3 Automated Planning

Automated planning is another common method of solving decision-making problems. Like reinforcement learning, it has a long history that extends back to the beginning of modern computing. In this thesis we focus primarily on classical planning and the extensions used in PDDL 2.1 and above [7]. We also consider alternatives for stochastic domains such as sample-based planning. Although classical planning and discrete control evolved separately until the 1990s, they share many features in common as a result of modeling problems as Markov decision processes.

2.3.1 Classical Planning

A classical planning problem is defined as the tuple $\langle S, A, T, s_0, g\rangle$, where

- S is the set of states,
- A is the set of actions (also called operators),
- T is a function $S \times A \rightarrow S$ that defines deterministic transitions,
- $s_0 \in S$ is the initial state, and
- $g \subseteq S$ is the set of goal states.

There are three key differences between this representation and the one used for reinforcement learning. First, the transition function is a mapping instead of a probability distribution. It is assumed that all transitions are deterministic in classical problems. Second, classical planning uses a set of states as goals instead of a reward function. In essence, a set of goal states is like a sparse reward function with a non-zero discount factor such that $R(s,a)$ is some arbitrary positive value for all $\langle s,a\rangle$ such that $T(s,a) \in g$, and 0 otherwise. Finally, the classical representation assumes a single starting state for a given problem. Whereas a reinforcement learning problem asks, "How can I maximize the benefits available in this environment from any state?", a classical planning problem asks, "How can I (most efficiently) get from point A to region B?" The measure of success for a classical planner is whether it can reach a goal state from the start state and in the case of multiple paths pick the shortest one.

There are several extensions to the classical model which we take advantage of in this thesis. First is the state-variable representation. Instead of treating states as atomic objects, this representation treats a state as the assignment of values to a set of functions. The main advantage of this representation is that the transition function and goal can take much more compact forms. Because the state space is represented as the cross product of a set of functions $S = v_1 \times v_2 \times \ldots \times v_n$, the transitions can be expressed as operations on subsets of variables, greatly increasing the number of transitions covered by a single rule. For example, a transition rule that is defined over a set of variables $\{v_1, v_2, \ldots, v_m\}$ could replace $\prod_{i=m+1}^{n} |v_i|$ mappings in the original definition of T. The definition of the goal can be decomposed similarly, resulting in the size of the representation being proportional to the specificity of the goal, rather than the number of relevant states. In the worst case (full interdependence of all state variables for transitions and goal regions that do not fully span any dimensions), this representation is no less compact than the traditional one [8, pp. 41-47].

Another very useful extension to classical planning is the notion of a plan metric. Plan metrics help bridge the gap in expressiveness between rewards and goals. While plain classical planning favors shorter plans, planning with a metric can favor certain goal states over others.

Definition 2.5 A plan metric is an arbitrary function that can use any variable in the final state as well as the makespan (total length) of the plan:

$$metric(\langle a_1, a_2, \ldots, a_n\rangle) = f[s_n, n] \to \mathbb{R}, \tag{2.14}$$

where s_n is the final state after starting in s_0 and applying the sequence of actions.

Compare the definition of a plan metric to the value of following a policy in equation 2.1. With$R(s, a) = 0 \; \forall s \notin g$ and a discount factor of 1 only when the metric does not include plan length, the two functions become equivalent. In this way, a plan metric can be used to express a limited form of the reward function. The cost of this extra expressiveness is that the problem becomes more computationally expensive to solve. In fact, if the metric function is not bounded, the problem becomes undecidable. Without providing additional information to a planner, it must do an exhaustive search over the entire goal region (or entire space of feasible plans if makespan is a component of the metric) to determine the optimal plan [7, pp. 6-7]. As a compromise, a class of planners called "satisficing" planners has arisen. These planners are typically bounded by time, search depth, or number of attempts and assume that any plan that satisfies the goal is sufficient, but that they should keep searching for plans of greater value as long as they can.

One approach to solving classical planning problems is to rephrase them in the form of other well-understood problems. This was the motivation for SATPlan, a method which converts the definition of a planning problem into propositional logic statements representing actions and the initial state. The problem of finding a plan then becomes one of determining which series of statements can be joined with the initial state such that the final result entails the goal while still being self-consistent. One of the major advantages of this approach is that satisfiability is an extensively studied problem for which a number of powerful solvers exist [12].

Another way to solve planning problems is to use a more compact representation of the problem. While it is relatively simple to search through a graph of states connected by actions, the number of possible states is often exponential with respect to the number of state variables, making the graph too large for an exhaustive or uninformed search to be practical. The Graphplan algorithm proposed by Blum and Furst in 1995 [3] seeks to mitigate this issue by making a more compact graph over which to search. This approach this algorithm uses consists of two steps, performed alternately until a plan is found: build a directed, acyclic graph called a *plan* *graph*, then search over it. A plan graph differs from a state space graph in that it consists of alternating layers of features and actions. The edges of the graph only link between layers, and represent preconditions and effects when linking features to actions and actions to features, respectively. Each feature layer i represents a superposition of all the features and negations of features that could be present after taking up to i actions, while each action layer represents the set of all actions for which the preconditions are satisfied in any past state. The layers also include information about which features and actions are mutually exclusive based on whether previous layers could include their achievers without interference. Graphplan builds the graph by adding one layer at a time until the last one contains the goal or it reaches a steady state in which no new features or actions appear in successive layers. Graphplan then searches backwards over the graph, finding actions that achieve each goal condition until it obtains a plan. Since the mutual exclusion relations contained in the plan graph consist only of pairs of features or actions, they cannot necessarily capture all of the conditions under which one action would prevent the execution of another. As a result, the search may fail to find a valid plan. In this case, Graphplan iteratively expands the plan graph with additional layers and repeats the search. While this approach is not theoretically guaranteed to be faster than state-space search, the fact that plan graph construction takes polynomial time and space and repeated searches are only necessary in situations with complex mutual exclusion rules leads to significant practical gains in the cost of finding a valid plan [3].

A third approach to classical planning is simply to do a forward search through the state space, using actions as edges between the states. This approach is improved significantly by the introduction of heuristics to guide the planner toward the goal while examining less states. One notable instance of heuristic state-space planners is Fast-Forward (FF) [1], which caused a large upset at the Artificial Intelligence Planning and Scheduling conference in 2000 by outperforming all other planners, including those based on Graphplan and SATplan. The distinguishing feature of Fast-Forward is that it uses a relaxed form of Graphplan as a heuristic function. The relaxation FF uses is to remove all delete effects from the action descriptions used by the

Graphplan component. Without delete effects, no actions or features can be mutually exclusive, causing both graph generation and backward search performed over the generated plan graph to have a polynomial time cost. The plan generated by this approach is not guaranteed to be valid, but the length of the plan is a sound, consistent heuristic that can be computed in polynomial time [11]. Further improvements on the FF algorithm include handling concurrent actions and metric functions [1].

2.3.2 Planning with Resources

Another pair of extensions to classical planning involve the concept of actions that take time and resources. Traditional planning problem definitions assume that all actions are instantaneous or take unit time and can only be executed one at a time. In practice, many more problems become expressible when we assume that actions take different amounts of time and we have multiple executors or one multi-tasking executor. We can express these types of problems using a classical representation that includes numerical variables (for time) and inequalities in preconditions (for resources). Alternatively, we can treat the problem as one of scheduling where each action no longer has preconditions and effects but instead is a combination of the following four resource constraints:

- Require: the action is only applicable in states where there is at least x amount of resource y
- Consume: the action reduces the quantity of resource y by amount x when it begins (implicitly includes require)
- Produce: the action increases the quantity of resource y by amount x when it ends
- Borrow: equivalent to consuming then producing x amount of resource y

Scheduling using this action framework allows us to plan about parallel, durative actions without having to include time as an explicit state variable. Additionally, it allows a number of other metrics such as total or weighted latency for actions [8, pp. 351-357]. Normally, classical planning with state-variable representation and negative effects is NEXPTIME-complete [8, p. 59], but scheduling is only NP-hard [8, pp. 359-360]. Therefore, whenever possible, using a scheduler following the above semantics can be significantly than a general-purpose PDDL-compliant planner.

In 2008, Chan et al. developed an algorithm that plans with resources as described above. Theirs is a heuristic scheduler which takes advantage of the concept of renewable resources - resources which do not get consumed, but are borrowed in order to produce other resources. While the problem remains NP-hard, the structure of the planner helps achieve good plans more quickly, making it a strong satisficing planner. Incorporating renewable resources allows the planner to separate the problem into two levels - one in which it finds an optimal plan to achieve the required resources and one in which it determines whether prioritizing the production of additional resources can improve the makespan of the plan. The inner planner uses a simple means-ends backwards state-space search to produce feasible plans while the outer planner effectively does plan-space planning guided by problem-specific heuristics. While there are a number of alternative planners out there based on more sophisticated algorithms such those mentioned in section 2.3.1, we chose this approach to planning because it greatly simplifies the planning process and provides a good reference point for the performance of other systems as in its simplest form it is just backwards search [5].

2.3.3 Monte Carlo Planning

It is important to note that while the planning techniques we have discussed here deal specifically with deterministic settings, there is a separate class of model-based algorithms that work in stochastic domains. Like reinforcement learning, these algorithms rely on value functions to determine an optimal policy; a plan is no longer sufficient for representing actions because the algorithm must accommodate deviations from the ideal state sequence. However, because they have a model, they do not need to collect examples from the real environment. Some algorithms in this vein such as value and policy iteration compute policies by performing iterations of the Bellman update, reducing the error in the value function until either the values or the policy cease to change [8, pp.380-386]. In this thesis, we focus on a different approach to planning in stochastic domains known as Monte Carlo planning (MCP). This technique uses the model as a simulator, gathering samples by observing repeated transitions between states. In this sense, MCP is in the middle ground between planning and RL in the sense that it uses a model offline (i.e. not interacting with the real environment) to gather evidence that provides incremental refinements to its estimate of state value. Online RL and the iterative methods, the goal of MCP is not to create a policy for every state. Instead, its objective is to determine the best policy for the starting state. By performing repeated executions of sampling in successive states, a Monte Carlo planner can produce a tree of policies for single states. The major advantage that comes of this approach is that the number of samples is no longer proportional to the size of the state space, and as such this algorithm avoids the curse of dimensionality.

One instance of Monte Carlo planning is the UCT algorithm, proposed by Cocis and Szevespari in 2006. The basic idea of UCT is to treat policy generation as a multi-armed bandit problem. Each action has a different payoff in terms of expected future rewards. The algorithm can determine which one pays out the most reward by sampling outcomes until it reaches a sufficient confidence level. The pseudocode for UCT appears in algorithm 4. The representation of the model that UCT uses is a *generator*, which is a function $S \times A \mapsto \langle S, \mathbb{R} \rangle$ that for a given state-action pair $\langle s, a \rangle$ will produce states s' in proportion to the transition probability $T(s'|s, a)$ and reward $r = R(s, a)$. The execution of the algorithm involves recursive calls to the $search$ function. The base cases are terminal states, where the can be no future rewards and therefore have a value of 0, and states at the maximum depth of the search tree, in which case a heuristic evaluation of the state can be employed to estimate rewards beyond the scope of the tree. The non-terminal executions of $search$ involve sampling actions $numSamples$ times, where $numSamples$ is set based on the desired accuracy of the result. What sets UCT apart from other MCP algorithms is the way in which it chooses which actions to sample. The action selection scheme of UCT is based on that of the UCB algorithm (UCT is UCB with trees):

$$a^* = \arg\max_a)Q(s,a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}}) \tag{2.15}$$

, where $N(s)$ is how many samples have been taken in state s using any action, $N(s,a)$ is the number of samples taken in s where a was the selected action, and c is a tunable parameter that balances exploration and exploitation. The UCT algorithm is particularly good at solving finite-horizon non-discounted problems, but can still find the optimal policy in polynomial time for the alternative [14, pp. 6-7].

Algorithm 4 The UCT algorithm for bandit-based planning.

```
Require: $G(s, a) : aGeneratorthatsimulatesstatetransitions$
  function MONTECARLOPLANNING($s : State$)
      while has more time do
          $search(G, s, 0)$
      end while
      return $\max_a Q(s, a)$
  end function
  function SEARCH($s : State, depth : Int$)
      if $s \in terminalStates$ then
          return 0
      else if $depth = maxDepth$ then
          return $evaluate(s)$
      else
          for $i \leftarrow 0; i \leftarrow i + 1\ until\ i\ =\ numSamples$ do
              $a \leftarrow \arg\max_a(Q(s, a) + c\sqrt{\frac{\ln N(s)}{N(s,a)}})$
              $(s', r) \leftarrow G.simulate(s, a)$
              $Q(s, a) \leftarrow Q(s, a) + r + \gamma search(s', depth + 1)$
              $N(s) \leftarrow N(s) + 1$
              $N(s, a) \leftarrow N(s, a) + 1$
          end for
          return $\max_a Q(s, a)$
      end if
  end function
```

Chapter 3

Combining Model-Based and Model-Free Methods

This chapter investigates combining model-based and model-free methods of solving Markov decision processes. The first section illustrates a sample domain for which a mix of the two techniques is a natural approach. The second section reviews prior work on this topic. Finally, the third section presents a novel algorithm for combining automated planning and reinforcement learning on a task-specific basis.

3.1 Motivating Example

Consider the task of baking. To prepare baked goods, a baker would start by picking a recipe for the food they want to make. After gathering all the necessary ingredients and accoutrements, the baker follows the steps listed in the recipe to produce the end product. The baker has a sequential plan which they know how to follow, such as ”mix butter and sugar” or ”sift flour with baking powder.” A well-versed baker will also have a model (like the one in figure 3.1) of how the ingredients react to his or her actions: egg yolks break apart when stirred, semi-sweet chocolate melts at a certain temperature, and so on. If the baker follows the steps correctly, the results are predictable. Given a description of the final product, a set of actions to take such as mixing ingredients or turning the oven on for a specific temperature and time, and a list of ingredients and equipment, a baker can use their knowledge of how ingredients behave to plan how to produce baked goods. The process of filling in the steps of a recipe is a good example of a problem that can be solved easily by automated planning.

```
Ingredients:
2.5 cups flour, 0.5 cups sugar, 2 eggs, 1 cup applesauce,
1 tablespoon baking powder, 1 teaspoon vanilla, 1 teaspoon salt,
4 teaspoons cinnamon, 1 apple

Equipment:
1 oven, 2 mixing bowls, 1 muffin tin, 1 mixer

Actions:
BreakEgg(bowl)[10 seconds]:
  preconditions: eggs > 0
  effects: eggs -= 1, not empty(bowl), yolks(bowl) += 1,
           whites(bowl) += 1
SeparateEgg(bowl1, bowl2)[20 seconds]:
  preconditions: eggs > 0
  effects: eggs -= 1, not empty(bowl1), not empty(bowl2),
           yolks(bowl1) += 1, whites(bowl2) += 1
WhipEggWhites(bowl, mixer)[2 minutes]:
  preconditions: yolks(bowl) = 0, clean(mixer)
  effects: whippedWhites(bowl) = whites(bowl), whites(bowl) = 0,
           not clean(mixer)
Bake(oven, muffin tin)[25 minutes]:
  preconditions: dough(muffin tin) > 0, muffins(muffin tin) = 0,
                 empty(oven), preheated(oven)
  effect: muffins(muffin tin) = dough(muffin tin),
          dough(muffin tin) = 0
```

Figure 3.1: A partial model definition for planning a way to bake apple muffins.

The problem of making food that tastes good, on the other hand, is not as straightforward. Humans judge food based on a variety of measures such as smell, taste, and texture. Even putting aside variations between individuals, the rules behind what makes some food appealing and other food unappetizing are complex and difficult to model. Sugar and fat may be important for making food taste good, but a stick of butter is sickening on its own. A minor detail such as a decigram of salt can make the difference between a good batch of cookie and an excellent cookie. Adjusting a recipe to maximize the experience of eating the results is a control problem for which there is not a good model, making reinforcement learning an ideal means of solving it. In this way, baking is a planning task nested in the learning task of adjusting the recipe. The baker can always determine a plan to produce the baked goods from information given in the recipe, but exactly how much of each ingredient, how long to leave the oven on, etc. are determined through learning.

Now consider the task of running a kitchen in a bakery. Instead of just one baker, there are several, each able to do work. Being the seasoned workers that they are, they can give an accurate estimate of the time and resources it would take to produce a cake, a batch of cookies, or loaf of bread. In addition to their individual problems of how to make the food, the presence of multiple bakers in one kitchen brings up a new problem: resource allocation. If the kitchen has less ovens than bakers, what the makers make when must be carefully coordinated. Ovens and counter tops can only hold so many items at once, different recipes require different oven temperatures. This problem is complex but well understood. With the use of a model like the one in figure 3.2, a resource-based planning algorithm would have no problem coordinating the activities of a busy bakery. On top of the planning problem of making baked goods and the learning problem of improving recipes, we now have another problem of planning who gets to make what when.

```
Resources: chefs, assistants, ovens, stoves, sinks,
           clean pots, dirty pots,
           clean bowls, dirty bowls,
           empty cake pans, full cake pans,
           empty cookie sheets, full cookie sheets,
           cakes, cookies


Actions:
MakeCakeBatter[15 minutes]:
  borrows:  (chefs, 1),
  consumes: (clean bowls, 3), (empty cake pans, 1)
  produces: (dirty bowls, 3), (full cake pans, 1)
BakeCake[45 minutes]:
  requires: (chefs, 1),
  borrows:  (ovens, 1), (full cake pans, 1)
  produces: (cakes, 1)
StoreCake[1 minute]:
  borrows:  (assistants, 1),
  consumes: (full cake pans, 1)
  produces: (empty cake pans, 1)
WashBowl[30 seconds]:
  borrows:  (assistants, 1), (sinks, 1)
  consumes: (dirty bowls, 1),
  produces: (clean bowls, 1)
```

Figure 3.2: A partial definition of bakery management as a resource scheduling problem

The world is full of problems like this. Problems which nest planning tasks in learning tasks in more planning tasks, etc. Improving the scalability of the algorithms that solve these problems is important for scaling problems vertically. As such, the rest of this chapter is dedicated to investigating ways in which automated planning and reinforcement learning can be used together to solve large problems with both model-based and model-less elements.

3.2 Previous Work

The concept of combining reasoning with models and learning from experience goes back more than 20 years. The first, most basic example of combining the two approaches to solving Markov Decision Processes was the Dyna algorithm proposed by Richard Sutton. Motivated by the idea that humans keep internal models of the world in their heads, the Dyna family of algorithms augments temporal difference-based learning with additional cycles of learning using an internal model. While this does not involve planning in the strictest sense, it is the earliest precedent of incorporating a model into an RL-based agent[21, pp. 230-233]. While there are a number of examples such as this of combining model-based and model-free learning, there are decidedly fewer examples of combining learning with a non-learning model-based approach, such as planning. Even fewer works have focused on combining the two in a hierarchical manner.

Perhaps the most significant contribution to the topic of hierarchical combinations of planning and reinforcement learning is Malcolm Ryan's Ph.D. thesis. In his work, Ryan also proposes two algorithms that integrate model-based reasoning and model-free learning in a hierarchical manner. The first algorithm he presents is Planned Hierarchical Semi-Markov Q-Learning, or Planned HSMQ-learning. Similar to MAXQ, Planned HSMQ-Learning uses a network of behaviours (i.e. tasks) to perform abstract reasoning. What sets this new algorithm apart is that it uses a means-ends planner to derive a plan for completing each behaviour. This plan consists of multiple layers of sub-goals which the agent must achieve in succession to arrive at the final goal of the behaviour. The agent executes such a plan by performing actions that will achieve the sub-goals in the first layer, followed by actions for the second layer, and so on. The actions which are relevant to the current layer are considered to be "active." Each behaviour in the hierarchy has a corresponding reinforcement learning component. The definition of which child behaviours are "active" helps to reduce the number of state-action pairs the RL component will try when executing its exploratory policy. In addition to using a modified policy, the termination conditions of a behaviour in a Planned HSMQ-learning hierarchy are different. Each behaviour defines a set of preconditions and effects, which both allow its parent behaviours to reason about it while planning, and determine when its task has terminated. Yet another difference between MAXQ and Planned HSMQ-Learning is that while the former uses a completion function to recursively compute Q values, the latter maintains a Q function for each behaviour. Other contributions of Ryan's thesis include the incorporation of teleo-reactive semantics for behaviours and maintenance of the agent's model through inductive logic programming, both of which are beyond the scope of this paper [18].

One further improvement was published by Matthew Grounds and Daniel Kudenko in 2008. As opposed to Ryan's approach, which is to use planning to improve RL action selection, Grounds and Kudenko use planning and reinforcement learning in separate levels of a hierarchical agent. They define a two level hierarchy consisting of a STRIPS planner that decides what sequence of abstract actions to take and reinforcement learners that learn how to perform the primitive actions that compose each abstract action. Similar to MAXQ, this approach known as PLANQ-learning uses a pseudo-reward function to teach the reinforcement learner how to achieve the desired outcome by providing a positive reward for satisfying the effects of the abstract action and a negative reward for violating preconditions. While they only specify how to separate planning and reinforcement learning into two explicit levels, their paper was the first published work that fully separates the two in a hierarchy, rather than using one to inform the other [9].

3.3 Task-Specific Planning and Reinforcement Learning

Building on the work done by Ryan and Grounds, we propose a framework for creating fully hierarchical agents that use both reinforcement learning and planning. Instead of incorporating both at each level, we acknowledge that some state-action spaces have attributes that make them amenable to one or the other at different levels of abstraction. Consider an environment in which the primitive actions are stochastic, but the transition probability matrix is sparse and most state-action pairs tend towards a specific related set of states. Examples of this type of environment are commonly used for demonstrating improvements in reinforcement learning, such as stochastic gridworlds in which the agent has a high but non-guaranteed probability of moving in the desired direction (e.g. the domain used to evaluate the PLANQ algorithm by Grounds [9, pp. 5-6]). In this case, a reinforcement learning agent with a proper pseudo-reward function can effectively mitigate the non-determinism when traveling over long distances. A classical planner would be suitable for making higher level decisions regarding long-term goals or selection of landmarks for which to aim. Conversely, a problem may contain long chains of near-deterministic transitions, while the higher level goals rely on stochastic variables. Environments where there are few, but long and ordered sequences of actions that lead to high utility states are hard to learn for RL. This situation could come up in any complex control problem which involves a long pipeline of inputs leading to a stochastic process. In this second case, it makes more sense to entrust the navigation of primitive, deterministic actions to a planner, while a reinforcement learning component can make higher level decisions by setting the goal of the planner. The experiments chapter later in this thesis contains an example of this situation. Our approach, which we call task-specific algorithm selection (TSAS), takes advantage of these special structures in decision making problems.

Definition 3.1 A TSAS hierarchy as a directed, acyclic graph G containing a set of nodes N that represent individual tasks. The set of nodes N consists of three distinct subsets:

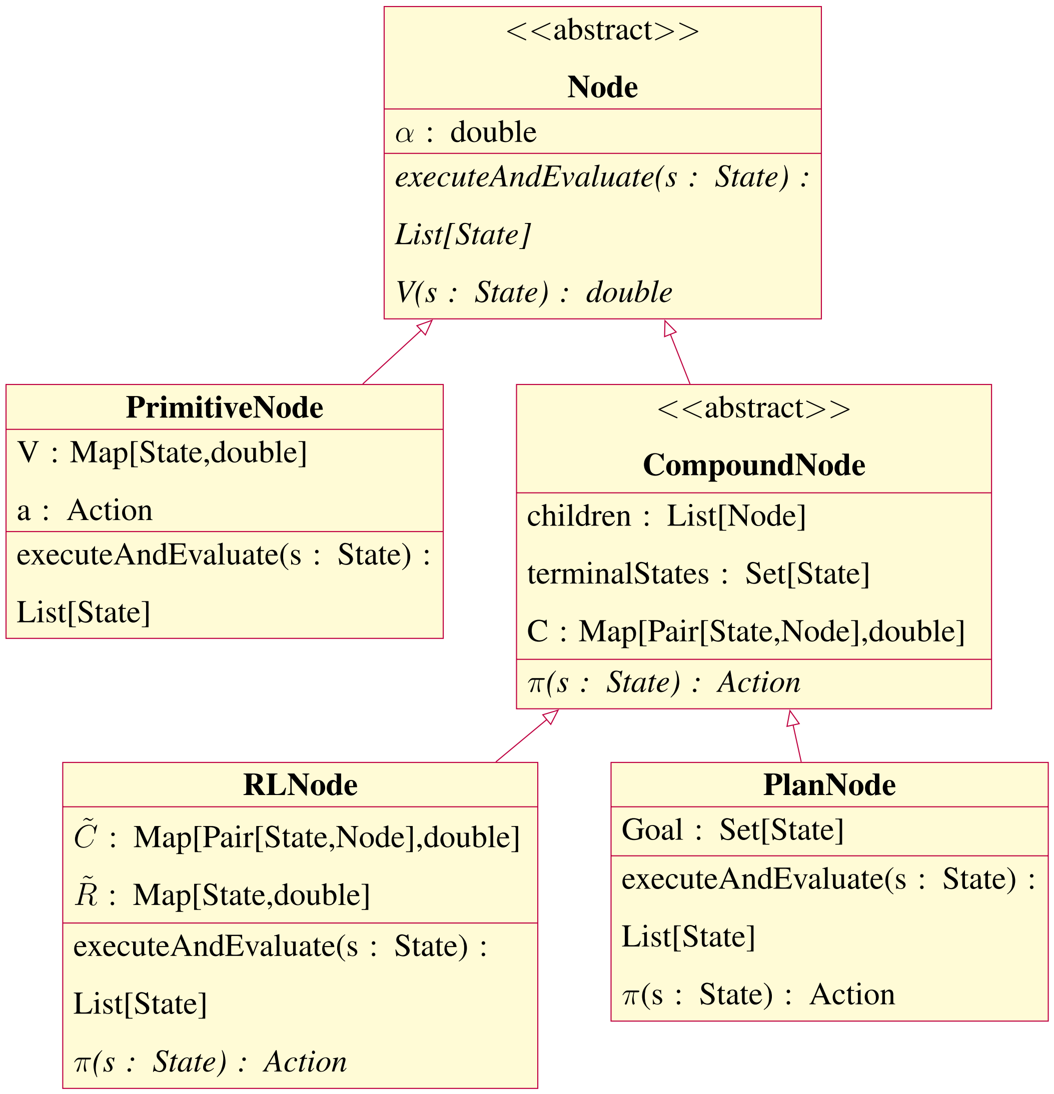
- N_A, the set of primitive nodes, which execute single actions,

- N_R, the set of RL nodes, which use the policies generated by reinforcement learning algorithms such as Q-learning and SARSA to execute tasks, and
- N_P, Plan nodes, which use plans to execute tasks.

Each node $n \in N$ has two functions $n.V(s)$ and $n.executeAndEvaluate(s)$ which are responsible for the node's estimate of the value of state s and the node's behavior in s. Additionally, each non-primitive node $n \in N_R \cup N_P$ has functions $C(s, a)$ and $\pi(s)$, the completion function and policy, and member variables $terminalStates$ and $childNodes$, which are sets of states and nodes, respectively. The variable $childNodes$ is equivalent to the edges of the graph: $e = \langle n_1, n_2 \rangle \in G.edges$ if and only if $n_2 \in n_1.childNodes$. A class diagram demonstrating the common and different properties of the three types of nodes appears in figure 3.3.

This definition is an extension of the definition of a MAXQ hierarchy. The functions $V(s)$ and $C(s, a)$, as well as the variables $terminalStates$ and $childNodes$ are equivalent to the ones in MAXQ (see section 2.2.3 for a more detailed description).

The algorithm for solving decision-making problems with a TSAS hierarchy begins in algorithm 5. This algorithm starts by finding the root of the graph containing the TSAS nodes and calls its $executeAndEvaluate()$ function. Although the implementation of this function depends on the type of node that implements it, the purpose is to execute the node's policy, evaluate the rewards it receives, and return a sequence of states encountered during execution. For primitive nodes, the task consists of only one action. Their implementation is simply to execute the action, update their evaluation of the reward, and return a sequence containing the state they observed (see algorithm 9).



<<abstract>>
Node
α : double
executeAndEvaluate(s : State) : List[State]
V(s : State) : double
PrimitiveNode
V : Map[State,double]
a : Action
executeAndEvaluate(s : State) : List[State]
<<abstract>>
CompoundNode
children : List[Node]
terminalStates : Set[State]
C : Map[Pair[State,Node],double]
π(s : State) : Action
RLNode
$\tilde{C}$: Map[Pair[State,Node],double]
$\tilde{R}$: Map[State,double]
executeAndEvaluate(s : State) : List[State]
π(s : State) : Action
PlanNode
Goal : Set[State]
executeAndEvaluate(s : State) : List[State]
π(s : State) : Action

Figure 3.3: The structure of the three types of TSAS nodes. All three have functions for executing actions (MAXQ) and evaluating states (V), but only non-primitive nodes have completion functions (C) and policies (π). RL nodes and plan nodes differ in the structures they maintain for the purpose of computing the policy.

The implementation of $executeAndEvaluate()$ for non-primitive nodes is nearly identical to the MAXQ-Q algorithm. An RL or plan node treats its task as a semi-Markov decision process where the state space is the same as that of the full problem and the action space is defined by the node's children. Execution of the task consists of alternately selecting a child node a based on the node's policy $\pi(s)$ and calling $a.executeAndEvaluate(s)$. The node continues this process until it reaches a state in $terminalStates$, after which it returns the sequence of all states encountered throughout all the sub-task executions it performed. Just as the node returns the states it encountered during its execution, each call to a child node results in a sequence of states encountered during execution of that node's task. The calling node performs the evaluation part of $executeAndEvaluate()$ by updating its completion function $C(s, a)$ for each state returned by the child task. The update is the same as the one used by MAXQ-Q (see algorithm 2).

RL nodes and plan nodes differ in the way that they select child tasks. RL nodes maintain some policy that balances exploration of new state-action pairs with exploitation of the current best solution. The way that the RL node defines which solution is "best" is based off of the pseudo-completion function $\tilde{C}(s, a)$. This function gets updated in parallel to the actual completion function and incorporates a pseudo-reward which helps orient the policy toward desirable terminal states. Algorithm 6 contains the pseudo code for $RLNode.executeAndEvaluate()$. Plan nodes, on the other hand, don't need to do any exploration to determine the true optimal solution. Their implementation, shown in algorithm 7, replaces the exploratory policy with calls to the node's planner. The sub-tasks executed by the plan node are just the first action in the plan to get from the current state to the node's goal. As a result, they do not need to maintain a pseudo-completion function. Additionally, instead of having to maximize over all actions to determine a^* for the completion function update, they simply call the planner again.

Algorithm 5 The TSAS algorithm

```
function TSAS($G : Graph$)
    for each episode do
        $s \leftarrow observeState()$
        $root \leftarrow G.N.rootNode$
        $root.executeAndEvaluate(s)$
    end for
end function
```

Algorithm 6 Methods for RLNode

```
function RLNODE.EXECUTEANDEVALUATE(s : State)
    $seq \leftarrow new\ List[State]()$
    while $s \notin terminalStates$ do
        $a \leftarrow \pi(s)$
        $childSeq \leftarrow a.executeAndEvaluate(s)$    ▷ get the list of states visited by descendants
        $s' \leftarrow observeState()$
        $a^* \leftarrow \arg\max_{a'}(\tilde{C}(n, s', a') + V(a', s))$    ▷ for Q-learning style update
        $n \leftarrow childSeq.length$
        for $State\ s_c\ in\ childSeq$ do
            ▷ update the pseudo-completion and completion functions with observed results
            $\tilde{C}(s_c, a) \leftarrow (1-\alpha) * \tilde{C}(s_c, a) + \alpha * \gamma^n * (\tilde{R}(s') + \tilde{C}(s', a') + a^*.V(s_c))$
            $C(s_c, a) \leftarrow (1-\alpha) * C(s_c, a) + \alpha * \gamma^n * (C(s', a') + a^*.V(s_c))$
            $n \leftarrow n - 1$
        end for
        $append(seq, childSeq)$    ▷ keep track of all states visited while executing this node
        $s \leftarrow s'$
    end while
    return $seq$
end function

function $\pi(s)$    ▷ $\pi(s)$ can be any ordered, GLIE policy such as $\epsilon$-greedy exploration
end function
```

Algorithm 7 Methods for PlanNode

```
function PlanNode.executeAndEvaluate(s : State)
    $seq \leftarrow new\ List[State]()$
    while $s \notin terminalStates$ do
        $a \leftarrow \pi(s)$
        $childSeq \leftarrow a.executeAndEvaluate(s)$    ▷ get the list of states visited by descendant nodes
        $s' \leftarrow observeState()$
        $a^* \leftarrow \pi(s')$    ▷ plan nodes have no notion of maximization - only the optimal plan
        $n \leftarrow childSeq.length$
        for $State\ s_c\ in\ childSeq$ do    ▷ this step is only necessary if this node has an RL ancestor, in which case an up-to-date estimate of the completion function is necessary for the recursive calculation of V(s)
            $C(s_c, a) \leftarrow (1 - \alpha) * C(s_c, a) + \alpha * \gamma^n * (C(s', a') + a^*.V(s_c))$
            $n \leftarrow n - 1$
        end for
        $append(seq, childSeq)$    ▷ keep track of all states visited while executing this node
        $s \leftarrow s'$
    end while
    return $seq$
end function

function $\pi(s)$
    return plan(s)    ▷ the goal is fixed for a given plan node, so the action to take in $s$ is the first one in the optimal plan to get to the goal
end function
```

Algorithm 8 Methods for CompoundNode (common to both RLNode and PlanNode)

```
function V(s : State)
    return $\max_{a \in children} a.V(s) + a.C(s, a)$
end function
```

Algorithm 9 Methods for PrimitiveNode

```
function PRIMITIVENODE.EXECUTEANDEVALUATE(s : State)
    $(s', r) \leftarrow execute(a)$    ▷ execute the node's primitive action
    $V(s) \leftarrow (1 - \alpha_t) * V(s) + \alpha_t * r$ ▷ update the estimated value, which is just the reward
    return new List[State](s)    ▷ give the state history back to the parent
end function
```

The optimality of a normal MAXQ hierarchy is detailed in theorem in three in Dietterich's paper [6, pp. 22-25], which we include here:

Theorem 3.2 Let $M = \{S, A, P, R, P_0\}$ be either an episodic MDP for which all deterministic policies are proper or a discounted infinite horizon MDP with discount factor γ. Let H be a MAXQ graph defined over sub-tasks $\{M_0, \ldots, M_k\}$ such that the pseudo-reward function $\tilde{R}_i(s'|s, a)$ is zero for all i, s, a, and s'. Let $\alpha_t(i) > 0$ be a sequence of constants for each Max node i such that

$$\lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t(i) = \infty and \lim_{T\to\infty} \sum_{t=1}^{T} \alpha_t^2(i) < \infty. \tag{3.1}$$

Let $\pi_x(i, s)$ be an ordered, GLIE policy at each node i and state s and assume that $\mid V_t(i, s) \mid$ and $\mid C_t(i, s, a) \mid$ are bounded for all t, i, s, and a. Then with probability 1, algorithm MAXQ-0 converges to π_r^*, the unique recursively optimal policy for M consistent with H and π_x.

An *ordered*, GLIE policy is one which defines a consistent ordering of all actions $a_1 \prec a_2 \prec ...$ such that:

$$\forall a_\alpha, a_\beta, \pi(s) = a_\alpha \; iff \; Q(s, a_\alpha) > Q(s, a_\beta) \; or \; Q(s, a_\alpha) = Q(s, a_\beta) \; and \; a_\alpha \prec a_\beta. \tag{3.2}$$

The original proof also includes a corollary which states that pseudo-reward functions do not affect the guarantee recursive optimality. We assert that the same theorem applies to a TSAS hierarchy with a few additional assumptions about the parameters of the planners involved.

Corollary 3.3 Let H^+ be a directed, acyclic graph of nodes $\{N_A, N_R, N_P\}$ in which all leaf nodes in N_A represent primitive actions, all RL nodes in N_R uphold the properties in theorem 3.2, and all plan nodes in N_P contain planners for which the metric used by its planning algorithm is equal to the total (discounted) reward for the full execution of the node's plans. Given that

- for any planning node p, its set of terminal states $p.T$ includes any state s from which the goal region is unreachable and
- planning nodes with empty plans are never ordered before ones with non-empty plans of equal value by their parents' policies,

the TSAS algorithm will converge to the unique recursively optimal policy consistent with H^+.

Proof The optimal policy for reinforcement learning is one that maximizes the expected discounted rewards either within the horizon or for the remainder of the episode. If a planner uses the sum of the expected discounted rewards as a metric, and it produces a plan which maximizes that metric, a policy that follows the plan is guaranteed to be optimal. Additionally, the execution of the plan will be greedy in the limit of infinite exploration (GLIE) because it contains no exploratory component; always following the plan is greedy from the first execution. By converting plans into totally ordered plans (which is trivial to do in a consistent manner) and establishing an arbitrary but consistent method of choosing one plan for a state in the case of a tie between equally good plans, we achieve an ordered GLIE policy for all plan nodes.

While an optimal, ordered, GLIE policy is enough to ensure hierarchical optimality when the plan nodes have no RL parents, we must also demonstrate that they can provide correct estimates of state values since the optimality of RL nodes requires correct estimates of state values from their children. The function $Q(s, a)$ for node n is defined as

$$n.Q(s, a) = n.C(s, a) + a.V(s). \tag{3.3}$$

The value function in turn is defined as $\max_{a' \in a.children}(a.Q(s, a'))$. After unrolling an entire hierarchy of nodes $a_1, a_2, \ldots, a_m$, the function is:

$$n.Q(s,a) = \sum_{i=1}^{m}(n.C(s,a_i)) + a_m.V(s), \tag{3.4}$$

where $a_n.V(s)$ is simply the value of executing the primitive action a_n in s.

The MAXQ algorithm makes weighted updates to $i.C(s,a)$ as follows:

$$n.C_{t+1}(s,a) = (1 - n.\alpha_t) * n.C(s,a) + n.\alpha_t * \gamma^N * \max_{a'}(a'.V(s') + n.C(s',a')). \tag{3.5}$$

Note that this function requires the child node's estimate of the next state's value. The inductive proof Dietterich used to demonstrate optimality relies on a' providing $V^*(s)$ for any s that the parent node encounters. Because the policy of a plan node is to execute a plan which optimizes a metric equivalent to the reward function, the planner will be able to determine a^* for any s for which it can produce a plan. Since the plan node maintains its own $C(s,a)$ and it is assumed that this too will converge over time (part of the proof by induction), the plan node can return $C^*(s,a^*)$, which is equivalent to $V^*(s)$.

Assumption one (unsolvable start states are terminal) comes into play for any state for which it cannot produce a plan. Since these states are assumed to be in the set of terminal states, the planner will execute no actions. Because it executes no actions, the parent will always see a transition from s to s with an N of 0. This causes the update to become:

$$n.C_{t+1}(s,a) = n.C(s,a) + n.\alpha_t * \max_{a'}(a'.V(s)) \tag{3.6}$$

which over time overwrites the estimated value with $a^*.V(s)$. Because the update to $C(s,a)$ takes several steps to converge to $V(s)$ when $\alpha < 1$, any times when $n.C(s,a) > a^*.V(s)$ (such as when a negative example causes the value of a^* to decrease) will lead to the parent n taking action a for all greedy action choices, quickly reducing the value of the completion function until it matches again. Once they are equal, assumption two (empty plans are never preferred in ties) guarantees that the parent will not waste time trying to execute the empty plan other than by random choice while exploring.

We have just demonstrated that given our assumptions, a plan node will both make optimal decisions and provide correct value estimates to parent nodes, allowing them to make optimal decisions. The first assumption is reasonable because the definition of a terminal state for an abstract node is a property of the hierarchy definition and not the environment, meaning any agent that uses plan nodes may impose this restriction without violating any other tenet of the MAXQ hierarchy. Not favoring empty plans in case of a tie is also reasonable for MAXQ nodes because it is compatible with (and helps to form) the ordered part of the requisite ordered, GLIE policy. The assumption that a planner will execute a plan that maximizes the metric it used to make the plan is natural to an optimizing planner. In the case of a satisficing planner, it is enough to allow it to re-plan periodically and retain the highest valued plan. As long as the planner continues to investigate novel plans, the probability that it will search the entire space of plans (and by extension discover the optimal plan) approaches 1 in the limit of infinite execution. The only assumption that restricts the choice of planning algorithm for an optimal hierarchy is that the plan metric must be equivalent to the underlying reward function. The use of a classical planner implies that we already have a model, which means that knowing the reward function and deriving a metric from it is not a problem. However, the planner using the metric must be capable of understanding path-dependent metrics, or else the problem must be expressed in such a way that the cumulative reward can be calculated from the terminal state alone. If either of these is true, then the introduction of a plan node into a MAXQ hierarchy does not violate any assumptions made in Dietterich's proof. Therefore, a TSAS hierarchy consisting of any ordering of plan and MAXQ nodes is hierarchically optimal.

Chapter 4

Extensions to Reinforcement Learning and Planning

As part of developing a hierarchical agent that uses planning and reinforcement learning on a task-specific basis, we also had to write implementations of RL and planning algorithms which the hierarchical agent would use. The criteria for hierarchical optimality are not particularly restrictive; there are a number of algorithms for both types of agents that are suitable components of the hierarchy. For example, a straightforward implementation of Q-learning with a tabular representation of Q values and a forward state-space search with no heuristics would be acceptable for the RL and planning algorithms, respectively. However, this runs contrary to the goal of implementing a mixed hierarchy, which is to scale to larger problems than any one component could handle. As such, we use a more sophisticated approach to reinforcement learning that reduces the number of distinct Q values that must be determined without sacrificing the guarantee of convergence to an optimal policy. In comparing a mixed hierarchy to a hierarchy of pure RL agents that also use this approach, it becomes evident that incorporating planning provides significant improvements beyond what is possible with RL. We use a satisficing planner that uses two levels of search to find valid plans and improve their quality.

4.1 Reinforcement Learning with Factored State and Action Spaces

As mentioned in section 2.2.2, reinforcement learning has issues with scaling to large problems. Because algorithms like Q-learning and SARSA do not rely on a model, there is nothing inherent in the information that they are given which suggests a way to decompose states and actions. The naive approach to Q value representation is that each state-action pair is a unique combination and that no information about one pair is relevant to another. However, this is not always the way that real situations work. In chapter two we discussed two alternative approaches to representing the Q function: function approximation and hierarchical reinforcement learning. These two approaches allow the designer of an RL agent to give implicit information to the agent about how state-action pairs relate to each other and what transitions and rewards are representative of outcomes for multiple actions and in multiple states. The downside to these approaches is that they make compromises on the optimality of the policies they produce. For HRL, the policy is recursively optimal, which is often sufficient [1], but function approximation makes no guarantees at all about optimality. We assert that it is possible to decompose a flat Q function in such a way that preserves the optimality of the full representation of the Q function.

Suppose we know two things. First, the reward function is really the sum of multiple functions. For example, let us consider a reward function which is the sum of two other functions that are dependent on strict subsets of $\{a_1, a_2, \ldots, a_m\}$, which we shall call a_α and a_β. We can define the feature sets α and β as $a_\alpha \setminus a_\beta$ and $a_\beta \setminus a_\alpha$, respectively. Second, let us suppose that in the transition function α and β are conditionally independent given the features in $a_\alpha \cap a_\beta$. We can then write the Q function as the sum of two other functions:

$$\begin{aligned}
Q(s,a) &= Q(s,a_\alpha) + Q(s,a_\beta) \\
&= R(s,a_\alpha) + \gamma \sum_{s' \in S} V(s')T(s' \mid s,a_\alpha) + R(s,a_\beta) + \gamma \sum_{s' \in S} V(s')T(s' \mid s,a_\beta) \\
&= R(s,a_\alpha) + R(s,a_\beta) + \gamma \sum_{s' \in S} V(s')(T(s' \mid s,a_\alpha) + T(s' \mid s,a_\beta))
\end{aligned} \tag{4.1}$$

Because of our assumption about the composition of the reward function, we can combine the two functions into the full reward function. Additionally, because a_α and a_β are conditionally independent, adding one to the given variables in T provides no additional info if the other is

[1]see section 5.1 of Dietterich's paper on MAXQ for situations in which this is true [6, pp. 28-36]

already present. Therefore, equation 4.1 is equivalent to the normal Q function:

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s' \in S} V(s')(T(s' \mid s, a_\alpha) + T(s' \mid s, a_\beta)) \\ &= R(s,a) + \gamma \sum_{s' \in S} V(s')(T(s' \mid s, a_\alpha \cup a_\beta) + T(s' \mid s, a_\beta \cup a_\alpha)) \\ &= R(s,a) + \gamma / Z \sum_{s' \in S} V(s')(T(s' \mid s, a)) \end{aligned} \tag{4.2}$$

where Z is a normalization constant equal to the number of sets of conditionally independent variables. As a result, whenever the two assumptions can be made about a set of features, multiple lower-dimensional Q functions can be used without loss of optimality. In this way, the addition of certain features can be reduced from an exponential increase to a linear increase in representation size. This approach was first proposed as factored planning by Guestrin et. al. [10] and further refined by Raghavan et. al. [16], but to our knowledge this is the first application of factored state and action representations to online, discrete control methods.

4.2 Resource-Based Planning

The implementation we used in experiments for this thesis includes several improvements over the one developed by Chan et al. (see section 2.3.2). The first improvement is a refinement in the means-ends approach to finding feasible plans. When attempting to solve a sub-goal, the original planner takes the first relevant action and tries applying it as many times as necessary to achieve the desired quantity of a resource. Our implementation considers each relevant action once, keeping the resource requirements that are not fully satisfied by the action in the goal list. This improvement is particularly important in situations where resources have parametrized producers whose costs vary by the rate at which it produces the desired resource. Secondly, we improved the metric used by the heuristic scheduler so that it uses the completion time of the last action (maximum latency) after parallelizing the plan instead of the makespan of the sequential plan. This was a non-trivial flaw in the original scheduler that caused it to evaluate more complex plans unfavorably even if they had lower latency. Finally, we added a caching scheme to the planning agent. The original planner was meant to solve problems in isolation; ours was made with the understanding that as a part of a hierarchy of decision-making agents, the planner may be called repeatedly, from different start states, and with different goals. Caching plans turns future instances of problems into constant or linear time access as opposed to NP-hard scheduling.

Algorithm 10 Our resource-based planning algorithm. The algorithm consists of a cache of plans, a re-planning component that grafts plans together, a plan-space heuristic search, and a state-space search.

```
Require: $cache$ : a $Map\ \langle s : State, g : Goal\rangle \mapsto Plan$ that stores previously computed plans
  function GETPLAN($s : State, g : Goal$)
      $plan \leftarrow cache(s, g)$
      if $plan$ is null then
          $plan \leftarrow planAndSimulate(s, g)$
          $cache.put(s, g, plan)$    ▷ accumulate a cache of plans to avoid repeated calculations
      end if
      return $plan$
  end function
```

```
function PLANANDSIMULATE($s : State, g : Goal$)
    $realPlan \leftarrow \langle\rangle$
    $plan \leftarrow search(s, g)$
    $t \leftarrow 0$
    while $t < maxTimestep$ and $\neg plan.empty$ and $s \not\models g$ do
        if $newDecisionEpoch(t)$ then
            $plan \leftarrow search(s, g)$                          ▷ re-plan periodically
        end if
        for $a \in \{a \mid a \in plan \wedge a.startTime = t\}$ do       ▷ simulate each planned action
            if $s \models a.preconditions$ then
                $s \leftarrow applyAction(s, a)$
                $plan.remove(a), realPlan.append(a)$
            else
                $a.startTime \leftarrow a.startTime + 1$
            end if
        end for
        $t \leftarrow t + 1$
    end while
    return $realPlan$
end function
```

```
function SEARCH($s : State, g : Goal$)
    $renewables \leftarrow resources\ that\ represent\ executors\ and\ can\ be\ produced$
    $initialPlan \leftarrow plan(s, g)$
    $max \leftarrow metric(initialPlan), argmax \leftarrow initialPlan$
    $openList \leftarrow \langle\langle g, plan\rangle\rangle$    ▷ a priority queue based on a heuristic measure of the quality of the plan
    $closedSet \leftarrow \langle\rangle$
    while $executionTime < timeLimit$ do
        $objective \leftarrow openList.removeFirst()$
        $closedSet.add(objective)$
        for $resource \in renewables$ do
            $newObjective \leftarrow \langle resource, 1\rangle \cup objective$
            $plan \leftarrow plan(s, newObjective, \langle\rangle)$
            if $metric(plan) > max$ then
                $max \leftarrow metric(plan)$
                $argmax \leftarrow plan$
            end if
            $openList.enqueue(\langle newObjective, plan\rangle)$
        end for
    end while
    return $argmax$
end function
```

```
**function** PLAN($s : State, g : Goal, previouslySelected : Map[Resource, Int]$)
    $plan \leftarrow \langle\rangle$
    **while** $s \not\models g$ **do**
        $unfulfilled \leftarrow random\ resource\ in\ g\ not\ satisified\ by\ s$
        $max \leftarrow -\infty, argmax \leftarrow null$
        **for** $a \in \{a \mid unfulfilled \in a.produces\}$ **do**
            $g' \leftarrow g \cup a.requires \cup a.consumes \cup a.borrows \cup a.consumes$
            $newQuantity \leftarrow g.unfulfilled - (s.unfulfilled + a.produces(unfilfilled))$
            **if** $newQuantity < previouslySelected.get(unfulfilled)$ **then**
                $selected \leftarrow previouslySelected \cup \langle unfulfilled \mapsto unulfilled\rangle$
                $(s', seq) \leftarrow search(s, g')$
                **if** $s'\ satisfies\ unfilfilled \wedge metric(seq) > max$ **then**
                    $max \leftarrow metric(seq), argmax \leftarrow seq$
                **end if**
            **end if**
        **end for**
        **if** $argmax = null$ **then**
            return failure
        **else**
            $s \leftarrow apply(s, seq)$
            $plan.prepend(seq)$
        **end if**
    **end while**
    return plan
**end function**
```

Algorithm 10 contains our implementation of a resource-based planner. The planner starts by checking its cache for an existing plan. If it already has one, the planner returns it. Otherwise, it calls the $planAndExecute$ function, which gets a plan through the $search$ function and simulates its execution by starting with the initial state and applying the resource effects of each action in the plan sequentially. The value that the simulator adds is that it will re-plan periodically. Because the search operations are limited in how much time they can take, re-planning relaxes some of the requirements on search depth in order to produce an optimal plan.

The $search$ function implements a plan-space heuristic search. This step relies on the notion of *renewable executors*. Executors are resources which enable the production of other resources, i.e. they appear in the $borrows$ precondition of actions which produces other resources. Because the $borrows$ precondition requires a certain number of the executor to be present in the starting state of the action, the the number of times the action can be executed concurrently is limited by how many executors are available in the start state. For example, if a software process requires 1GB of memory and the computer running the program has 4GB of RAM, then at most four instances of the program can be run simultaneously. The plan-space search consists of calling the state-space planner for different sets of goals, each of which extend the initial goal with sub-goals requiring additional executors. The plan-space search uses the plan metric to evaluate whether the plans that produce more executors are better than ones that add less or no additional executors.

The $plan$ function is a straightforward state-space search. Execution consists of choosing a resource for which the quantity in the goal is greater than the quantity in the state and trying to produce it. The search iterates over each action that produces the unfulfilled resource, adding the action's preconditions and effects to the goal. It then makes a recursive call to $plan$ to try to satisfy the preconditions of the action and the remaining goals. Note that since resources are numerical variables, a single application of a producer may not fully fulfill the need for that resource. Instead of relaxing the problem and applying the same action enough times to achieve the resource requirement, we leave the partially fulfilled resource in the goal and let the recursive call determine which action is optimal for fulfilling the reduced quantity. The $plan$ function, in addition to requiring a state and a goal, also takes a third argument, $previouslySelected$. This is a mapping of resources that the planner has already tried to satisfy to the amount that was required. The purpose of this variable is to prevent infinite recursion between sets of actions where the resources that one action produces are consumed by another action which produces the resources which the first action consumes. By ensuring that the resource requirements are monotonically decreasing, the planner prevents situations in which its working goals would spiral out endlessly.

Chapter 5

Experiments

In order to test our algorithm in practice, we preformed a number of experiments in two domains. The first domain is a synthetic one created specifically for testing hierarchical agents. We refer to this domain as the business environment because it involves managing the operations of a large, vertically integrated company at various levels of detail. Using the business environment, we compare a number of different configurations of hierarchies and monolithic agents and demonstrate not just the correctness but the superiority of heterogeneous hierarchical agents. Our second domain is a real-time strategy (RTS) game called SEPIA (Strategy Engine for Programming Intelligent Agents), which was developed in conjunction with two other students specifically for testing game-playing agents. We use this domain to demonstrate the superior versatility of hierarchically integrated planning and reinforcement learning by solving a problem that is prohibitively expensive for a monolithic agent.

The business environment is a synthetic domain created specifically to test hierarchical agents. The premise of this environment is that the agent is in charge of running a vertically integrated enterprise. It is responsible for producing its own goods in factories, moving them to its stores, selling the items in the stores to consumers in the surrounding area, and assessing the profitability of new and existing markets. The agent performs these tasks by controlling production Just as with a real business, this environment lends itself to a distribution of responsibility. The definition of the Markov decision process is as follows:

- S consists of the following features:
  - $inventory_{f_1}, inventory_{f_2}, \ldots$ the inventory of each factory,

  - $inventory_{s_1}, inventory_{s_2}, \ldots$ the inventory of each store,
  - $isOpen_{s_1}, isOpen_{s_2}, \ldots$ whether each store is open for business,
  - $quality_{s_1}, quality_{s_2}, \ldots$ the level of free services and aesthetic improvements for each store,
  - $demand_{c_1}, demand_{c_2}, \ldots$ the demand in each city, and
  - $money$.

- A consists of the following actions:
  - $produce(factory, quantity)$ - produces quantity number of goods at factory (takes 1 day),
  - $transport(factory, store, quantity, [TRUCK|TRAIN])$ - moves quantity number of goods from factory to store by truck or train (takes 1 day),
  - $setPrice(store, level)$ - sets the price at which the store will sell its goods for the next week (lasts 1 week),
  - $open(store)$ - opens the closed store (takes one quarter)
  - $close(store)$ - closes the open store, liquidating its inventory (takes one quarter)
  - $upgrade/downgrade(store)$ - increases/decreases the quality of the store (takes one quarter)

- $T(s, a)$ - in addition to the effects described above, sales occur once per week for open stores after prices are set. This decreases store inventory with respect to the demand of the city in which the store resides along with the price, quality, and current inventory level. Money increases by the price times the amount sold, while it decreases whenever an item is produced or transported. Opening a store incurs a start-up cost, as does increasing a store's quality. Finally, a maintenance cost proportional to quality is applied once per quarter for each store if the agent leaves it open.

- $R(s, a)$ - the net profit, i.e. the increase in money from s to s'. While s' is not an argument to the reward function, all actions affect money deterministically, so $s'.money$ can be computed using s and a.

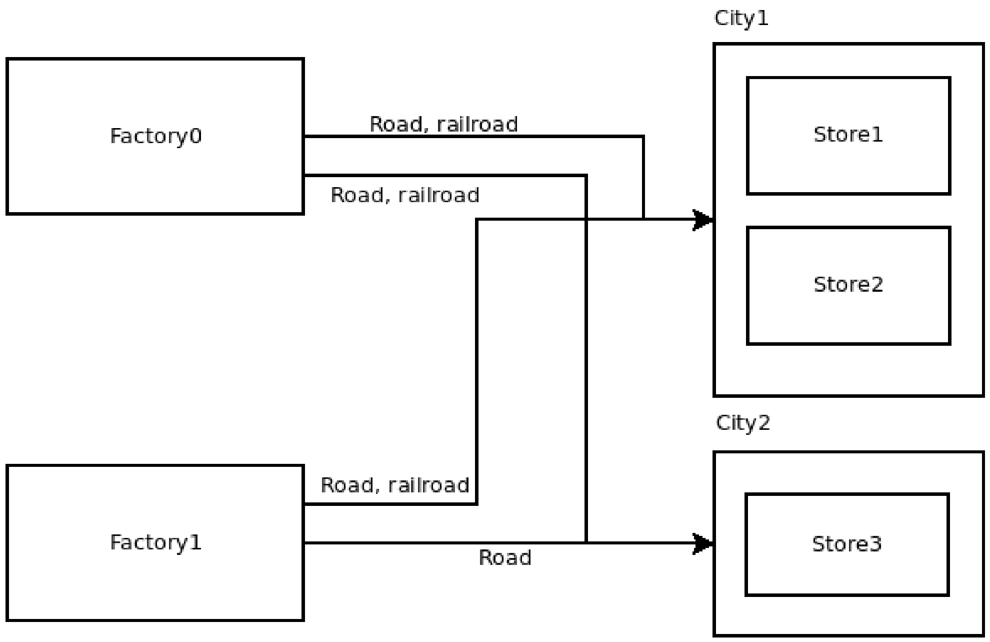


Figure 5.1: The business environment. The agent is responsible for producing goods at the factories, shipping them to the stores by truck or by train, then selling the goods to customers from within the city. Each city represents a self-contained market, where the demand is affected by all the stores located therein.

5.1 Business Environment

While one monolithic agent can solve the entire problem jointly, the state and action features are designed such that the problem can be divided into three more manageable problems: logistics, retail, and franchise. The logistics component deals with producing and transporting goods, the retail component manages prices, and the franchise component controls the opening, closing, upgrading, and downgrading of stores. This decomposition is advantageous in that it both separates out independent state features and actions and makes it so that the actions in each sub-problem all have the same duration. The task hierarchy for these three components is shown in figure 5.2, while the state spaces, action spaces, and transition mechanics are defined separately for each one below.

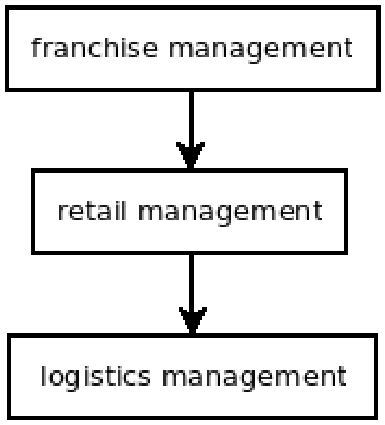


Figure 5.2: The task hierarchy for the business environment. The top level manages the quality of stores and which are open, the middle level maximizes the profit of each store, and the bottom level supplies the stores with goods.

The logistics management sub-problem consists of producing goods at factories and delivering them to stores at which they will be sold. The agent can take one action per day and has one week in which to deliver a specified quantity of items to each store. The basic definition of the Markov decision process for the logistics management problem is:

- $S = inventory_{f_1} \times inventory_{f_2} \times \ldots \times inventory_{s_1} \times inventory_{s_2} \times \ldots \times money$. Each state is a unique combination of the inventory of each factory, the inventory of each store, and the current amount of money.

- $A = production_{f_1} \times production_{f_2} \times \ldots \times transport_{f_1} \times transport_{f_2} \times \ldots$. Each action is a unique combination of production and transport features for each factory. Production can be 0 for any factory. Each factory can only transport one batch of goods per turn, but it has a choice of which store, the quantity, and which mode of transportation. Factories also have the option of not shipping anything.

- $T(s,a)$:

$$\begin{aligned}
inventory_{f_i,t+1} &= inventory_{f_i,t} + production_{f_i,t} - transport_{f_i,t}.qty \\
inventory_{s_i,t+1} &= inventory_{s_i,t} + \\
&\sum_{f_i} transport_{f_i,t}.qty \mid transport_{f_i,t}.dest = s_i \\
money_{t+1} &= money_t - cost_{production} * \sum_{f_i} production_{f_i,t} - \\
&\sum_{f_i} (cost_{truck,f_i,transport_{f_i,t}.dest} * \\
&transport_{f_i,t}.qty \mid mode = TRUCK) - \\
&\sum_{f_i} cost_{truck,f_i,transport_{f_i,t}.dest} \mid mode = TRAIN
\end{aligned}$$

The termination conditions include running out of money ($money < 0$) or running out of time (the maximum number of steps for the episode is exceeded).

As mentioned previously, the overall reward for the business environment is the net increase in money each turn. However, the only way to make money is to sell items at stores, so the logistics component must have some internal mechanism for balancing the cost of producing and moving goods with the payoff it will receive for selling them. For planning, the goal region is defined by the inventories of the stores. Both the minimum required inventories and the initial state vary from instance to instance of the problem. The metric for the planner is to minimize the amount of money used. For reinforcement learning the pseudo-reward function gives a positive value to any action that increases store inventory towards the goal, and a negative value if the inventory exceeds the goal to penalize waste.

While this sub-problem is solvable with reinforcement learning, several elements give a significant advantage to a planner. First, the action space scales up very quickly as the number of factories and stores increases. Search-based planning algorithms typically take advantage of the information about preconditions and effects of actions given to them to determine whether an action is relevant to satisfying a goal or applicable in a given state. Reinforcement learning has no model from which to get this information, so it must try all actions in order to guarantee optimality. Second, the state-action space does not lend itself to decomposition. Since producing more goods than necessary costs more and is therefore suboptimal, the agent must take into account all inventory features when deciding how much to produce. Additionally, the transport features affect multiple state features, which in turn can be affected by multiple transport features. As a result, the agent must consider all transport features jointly in order to avoid having multiple factories try to fulfill an order for the same store all at once.

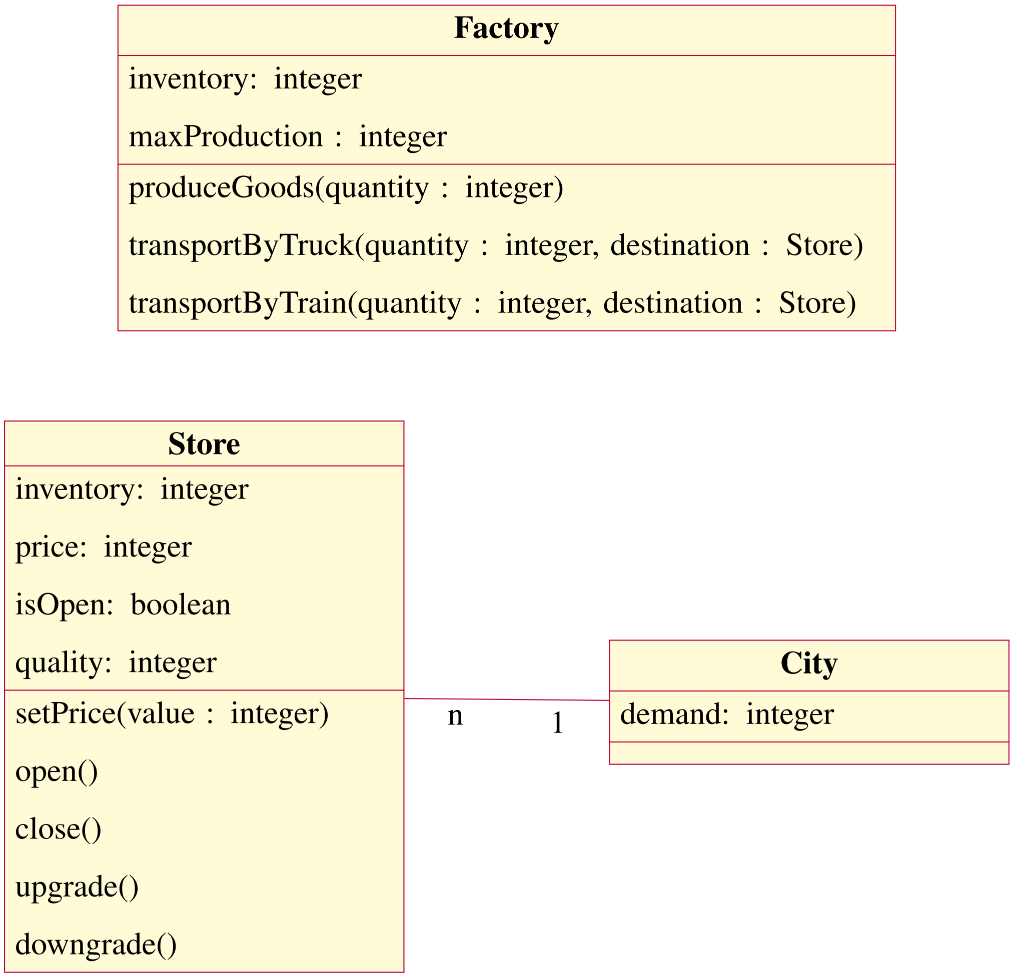

Factory
inventory: integer
maxProduction : integer
produceGoods(quantity : integer)
transportByTruck(quantity : integer, destination : Store)
transportByTrain(quantity : integer, destination : Store)
Store
inventory: integer
price: integer
isOpen: boolean
quality: integer
setPrice(value : integer)
open()
close()
upgrade()
downgrade()
n
1
City
demand: integer

Figure 5.3: The state and action space of the business environment, divided by object.

On top of logistics management is the problem of retail management. This problem consists of selling goods at stores. Once per week, the agent determines the price at which it will sell its inventory for the following week. After setting the price, consumers from the surrounding city purchase a portion of the inventory. How much they purchase is dependent on how expensive the product is and how high the demand is for the product. Additionally, this task involves replenishing inventory by calling the logistics management task parametrized with a request for more items. This request then becomes the goal/reward for the logistics agent for the remainder of the week. The Markov decision process definition for this portion of the business environment is:

- $S = inventory_{s_1} \times inventory_{s_2} \times \ldots \times demand_{c_1} \times demand_{c_2} \times \ldots \times money$
- $A = price_{s_1} \times price_{s_2} \times \ldots \times order_{s_1} \times order_{s_2}$
- $T(s, a):$

$$inventory_{s_i,t+1} = (\frac{demand_{s_i.city}}{DEMAND_{MAX}} - \frac{price_{s_i}}{2{*}PRICE_{MAX}})) * inv_{s_i,t}$$
$$money_{t+1} = money_t + \sum_{s_i} price_{s_i,t} * (inventory_{s_i,t} - inventory_{s_i,t+1})$$
$$demand_{c_i,t+1} = demand_{c_i,t} + \sum_{s_j}(\delta(s_j) \mid s_j.city = c_i) + \epsilon$$
$$\delta(s_j) = (1 - (\frac{inventory_{s_j,t+1} - inventory_{s_j,t}}{inventory_{max}} + \frac{demand_{s_j.city,t}}{demand_{max}})/2) * demand_{max}$$
$$\epsilon \in [-.15 * demand_{max}, .15 * demand_{max}]$$

- $R(s, a) = s'.money - s.money$. While s' is not an argument to the reward function, the change in money is deterministic and therefore can be derived from s and a.

While not included in the equations above, the inventory also increases between time steps based on the actions taken by the logistics task. If the agent is successful in completing the logistics task, the inventory for store s_i increases by $order_{s_i}$. Note that the introduction of random noise to the demand via ϵ makes this a stochastic problem. The objective then is to control the inputs of $price$ and $order$ such as to maximize the output, $\Delta money$.

The third component to the hierarchical representation of the business environment is franchise management. The objective of this part is to further increase the profit of the company by opening new stores, closing unprofitable ones, and provide auxiliary services to customers. Each store has a “quality” level to it, which determines the level of services available to customers. For example, if the business is a grocery store, a high quality store would have more staff to bag and carry groceries, more customer services associates, free samples and demonstrations of products, and a cafe or bank branch. Each of these improvements costs more money to maintain, but they also attract customers who are willing to pay more per item for a better experience. The effectiveness of improvements to a store is determined by the wealth of the customers, which is a property of the city in which the store is located. Customers from cities of low wealth will not use additional services, while customers in cities of higher wealth will eschew stores which do not provide the high-quality services they expect. The Markov decision process definition for franchise management is:

- $S = \mathit{isOpen}_{s_1} \times \mathit{isOpen}_{s_2} \times \ldots \times \mathit{quality}_{s_1} \times \mathit{quality}_{s_2} \times \ldots \times \mathit{money}$
- $A = \mathit{changeStatus}_{s_1} \times \mathit{changeStatus}_{s_2} \times \ldots \times \mathit{changeQuality}_{s_1} \times \mathit{changeQuality}_{s_2} \times \ldots$. The agent can choose to keep a store open or closed, or change the status between open and closed. It can also upgrade, downgrade, or maintain the quality of an open store.
- $T(s, a)$:

$$\begin{aligned} \mathit{isOpen}_{s_i,t+1} &= \mathit{isOpen}_{s_i,t} \oplus \mathit{changeStatus}_{s_i,t} \\ \mathit{quality}_{s_i,t+1} &= \mathit{quality}_{s_i,t} + \mathit{changeQuality}_{s_i,t} \in \{-1, 0, 1\} \\ \mathit{money}_{t+1} &= \mathit{money}_t - \sum_{s_i} (\mathit{maintenance}_{\mathit{quality}_{s_i,t}} \mid \mathit{isOpen}_{s_i,t}) \\ &\quad - \sum_{s_i} \mathit{upgrade\_cost}(\mathit{quality}_{s_i,t}, \mathit{quality}_{s_i,t+1}) \end{aligned}$$

- $R(s, a) = s'.\mathit{money} - s.\mathit{money}$. While s' is not an argument to the reward function, the change in money is deterministic and therefore can be derived from s and a.

The franchise action also includes a call to the retail task, which can increase the money based on sales. Additionally, money increases based on the services provided by each store. The amount of money generated by services is proportional to the amount of money produced by the store during that quarter multiplied by a number determined by the quality and wealth of the neighborhood. The franchise sub-problem is thus an optimization problem where the agent must align the quality of a store with the wealth of its surroundings and close stores that lose money. Because the decision epoch for this task is equal to a number of steps of steps of its child task (three months vs. one week), the stochasticity of this problem is determined by how effectively the child task can minimize the variance of the demand within each time period.

5.2 SEPIA

SEPIA, the Strategy Engine for Programming Intelligent Agents, is an environment specifically tailored to testing game-playing agents, with a primary focus on agents for real-time strategy (RTS) games. In this type of game, each player controls a set of units in a finite 2-D or 3-D region. Different types of units vary by the genre of the game, but typically include civilians, soldiers, vehicles, and buildings. The units are the player's means of interacting with the environment - they can move, attack each other, build more units, and so on. Each player also has a set of resources, such as money or building materials. Specific types of units collect these from resource nodes throughout the map, while others consume the resources in order to produce more units. SEPIA relies on scripted definitions of units, meaning that it can emulate any RTS with the same game mechanics. For our experiments, we modeled the unit definitions after those in Warcraft II. Definitions of two sample units are listed in table 5.1. While we could have interacted with an off-the-shelf game such as Warcraft II or Starcraft, SEPIA is much more amenable to experimenting with learning because it does not consume time and resources on multimedia such as graphics or sound. Running in its default headless mode, SEPIA can iterate through hundreds of games per second.

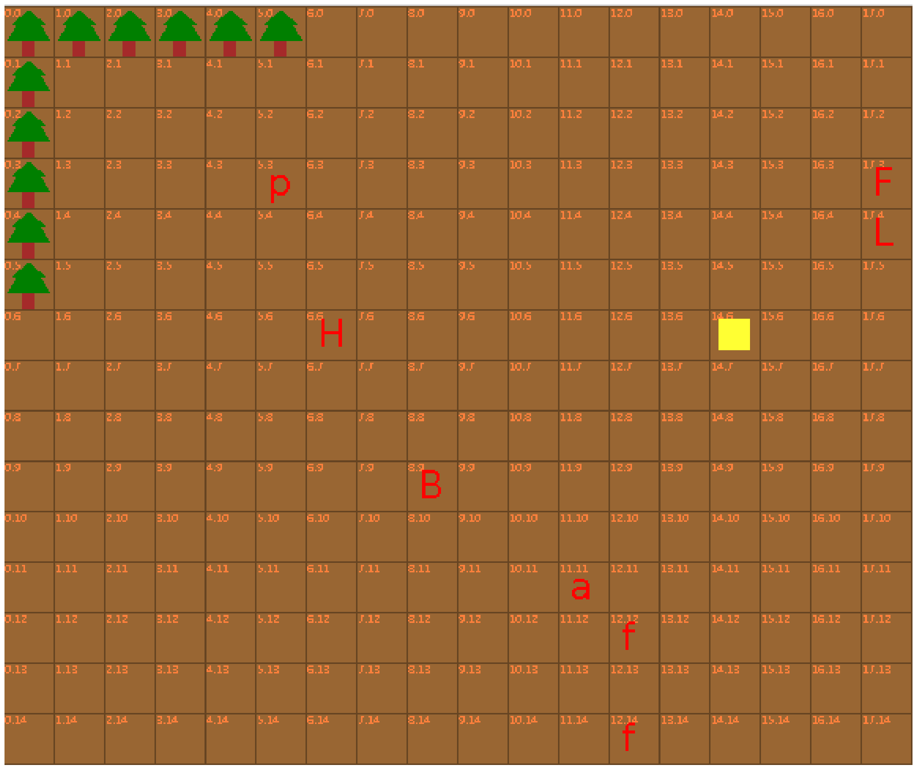


Figure 5.4: A screenshot from SEPIA. The letters represent units, such footmen, barracks, and peasants, while the trees and the gold-colored square represent resource deposits. By default, SEPIA runs without displaying any graphics.

The victory conditions for games in SEPIA are also specifically tailored to testing intelligent agents. RTS games as a whole are very large domains with numerous interesting problems. SEPIA defines three criteria for success which can be used separately or in conjunction to give agents different challenges. The first condition, conquest, is common to most RTS games. The goal in this type of scenario is to destroy all units belonging to other players by attacking them with the agent's own units and still have at least one unit left. This is an interesting problem because damage calculation contains an element of chance to it, making the outcome of multiple engagements between the same units slightly different each time. Different units also have different strengths such as moving quickly, attacking from a distance, or taking less damage per hit. This makes the composition of the agent's forces as important a factor as how well it manages the units. The second condition, Midas, requires the agent to gather a certain amount of resources. When combined with a time limit, this problem becomes one of investigating the tradeoff between conserving resources and consuming some to produce more gatherers. The third problem is manifest destiny, where the objective is to produce a certain amount of various types of units. This adds more complexity to the Midas scenario in that the agent must also manage a tech tree. For example, to build a knight, the agent must first have a stables, which in turn requires a keep, and it will also need a barracks which is the only unit that can produce knights.

	Footman	Peasant
Cost	600 gold, 0 wood, 1 food	400 gold, 0 wood, 1 food
HitPoints	60	30
Attack	6	3
Armor	2	0
Range	1	1
Gatherer	false	true
GoldPerTrip	N/A	100
WoodPerTrip	N/A	100
Builder	false	true
Produces	N/A	[TownHall,Barracks,LumberMill,Farm]

Table 5.1: Sample definitions of units in the SEPIA adaptation of Warcraft II. The attributes listed are used to determine the mechanics of how each unit type moves, how it attacks, how to calculate damage done to or by it, and what special actions such as gathering and production it can perform.

The specific problem we consider in our experiments is how to defend a cluster of buildings (a base) from waves of attackers.

5.3 Experimental Procedure

We used the business environment to test TSAS in a variety of ways. Our first experiments compare a two level hierarchy of RL for retail management and planning for logistics management to a pure RL hierarchy. The second set of experiments compare the same hierarchy of RL and planning to a two-level hierarchy in which both tasks use Monte Carlo planning. The third set tests our agent on a problem in which the state and action spaces are too large to perform either of the other approaches in working memory. Finally, our last set of tests involve a three-level hierarchy in which we introduce reinforcement learning over the franchise management task. All experiments were performed in the Java SE 6 runtime environment on a 3.2 GHz AMD FX processor with 32 GB of RAM.

Our first set of experiments involve two agents which use the exact same approach to retail management but different approaches to solving the logistics management task. The Q-function formulation for retail management is based on the decompositional approach outlined in section 4.1. Since the demands of each city are affected only be the actions of stores within that particular city, the Q-function is made out of functions that consider the demand of one city and the inventories, prices, and orders of the stores therein. The Q-function formulation for the logistics task in the pure RL hierarchy uses the full state-action space without any function approximation. Several attempts were made to reduce the size of the representation, such as considering production actions in isolation from each other and representing inventory as a sum of the individual variables, but each approach invariably led to suboptimal results in which the agent produced more goods than necessary. The planning algorithm which we use to test the TSAS approach is the resource-based planner described in section 4.2. The model on which the planner operates appears in figure 5.5. Instead of representing the action space as all the possible combinations of everything the agent can do in one time step, this representation breaks actions apart into the individual components of production and the two types of transport, parametrized by the objects involved and the quantity of goods. In addition to the state variables, the list of resources also includes "executor" resources which are used to enforce timing constraints on the actions. The planner always starts with one of each type of executor per factory and each action borrows one of these, ensuring that the planner does not try to perform the same action twice at the same time for the same objects.

Resources:

- $factory_1\_production, factory_2\_production, \ldots$
- $factory_1\_truck, factory_2\_truck, \ldots, factory_1\_train, factory_2\_train, \ldots$
- $inventory_{factory_1}, inventory_{factory_2}, \ldots, inventory_{store_1}, inventory_{store_2}, \ldots$
- $money$

Operators:

- $transportByTruck(f, s, q)[duration = 1]$:
  - $borrow(factory_f\_truck, 1)$
  - $consume(money, transport\_cost(factory_f, store_s, TRUCK) * q)$
  - $consume(inventory_{factory_f}, q)$
  - $produce(inventory_{store_s}, q)$
- $transportByTrain(f, s, q)[duration = 1]$:
  - $borrow(factory_f\_train, 1)$
  - $consume(money, transport\_cost(factory_f, store_s, TRAIN))$
  - $consume(inventory_{factory_f}, q)$
  - $produce(inventory_{store_s}, q)$
- $produce(f, q)[duration = 1]$:
  - $borrow(factory_f\_production, 1)$
  - $consume(money, production\_cost(factory_f) * q)$
  - $produce(inventory_{factory_f}, q)$

Figure 5.5: The resources and action definitions used by the planner for the logistics task. Each factory has three executors - production, truck, and train. The actions all take unit time and prevent any other actions from using the same executor at the same time.

5.3.1 Experiment 1: planning + RL vs. HRL

property	values for scenario 1	values for scenario 2
cities	2 cities	2 cities
stores	4 stores, 2 per city	4 stores, 2 per city
factories	2 factories	2 factories
max demand	6	6
max price	6	6
max inventory	5	5
max order	5	5
max production	1	2
truck cost	2 for both factories	3 for factory 1, 4 for factory 2
train cost	5 for both factories	8 for factory 1, 7 for factory 2
logistics time cost	1/2 day per action	1 day per action
starting money	150	100

Table 5.2: The two scenarios used for testing two-level hierarchies in the business environment. Both scenarios have the same state space, but the second makes the logistics task more difficult. The maximum values define the span of state and action variables. The size of the state-action space is exponential in the number of locations (cities, stores, and factories).

We begin by comparing the two agents in using the two scenarios in table 5.2. Both have large state spaces consisting of four stores in two cities and two factories. The first scenario is the easier of the two. The logistics action space is simplified by allowing the agent to reason about two production actions per day with half the possible permutations of quantity as opposed to one action per day. Second, the transportation costs are easier to reason about in the first scenario. Both factories have a simple tradeoff where using a train is cheaper for transporting three or more items at a time. In the second scenario, the agent must learn to differentiate between the two factories based on how many items it must transport. Third, the first scenario gives the agent more starting money, which gives the agent more time to manipulate demand before it must start making a profit.

The results of the first and second scenarios appear in figures 5.6 and 5.7, respectively. The first graph clearly demonstrates that using planning gives the TSAS agent a significant head start. Because the policy of the pure RL agent for the logistics task starts out using a randomized Q function for greedy actions, the parent task is unable to effectively reason about the outcomes of the task. The RL agent is able to match the returns of the TSAS agent for a while once the logistics component gets its bearings, but has subsequent periods of suboptimal performance. The points where the performance decreases correspond to points where policy changes in the retail task force the logistics task to solve new problems. This is another key advantage of planning when a model is available; the parent task does not experience any setbacks when shifting policy because the child task works as intended the first time it is executed in any state.

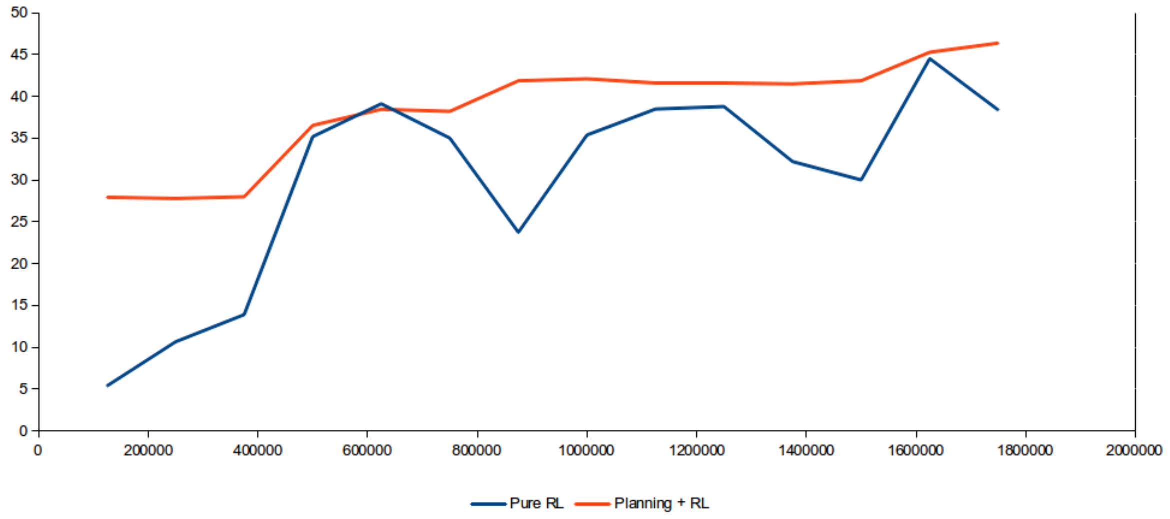


Figure 5.6: Results of running a two-level TSAS agent and a two-level MAXQ agent in a scenario with a large state-action space with easy to obtain rewards

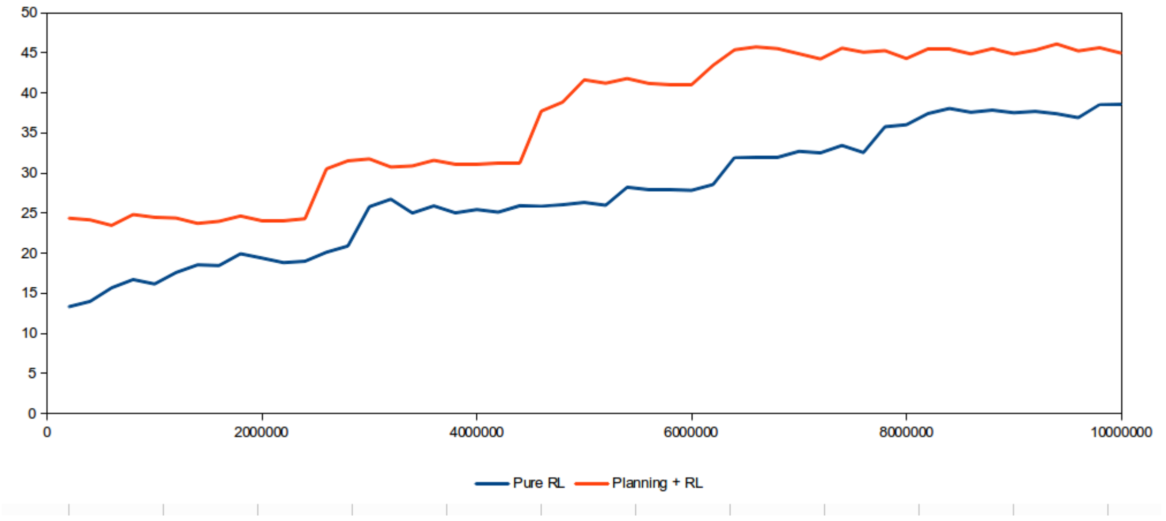


Figure 5.7: Results of running a two-level TSAS agent and a two-level MAXQ agent in a scenario with a large state-action space with more difficult to obtain rewards.

The second graph illustrates the superior efficiency of a TSAS hierarchy. As in the first graph, planning provides superior policies right from the start, resulting in a higher reward during the first episodes. In this scenario, however, the RL agent receives consistently lower rewards than the TSAS agent. In theory the RL agent will eventually catch up when both agents converge to the recursively optimal policy, but in several times the number of episodes it takes for the TSAS agent. This is due in part to the larger action space. If it is not careful about pruning the action space, the planner must take additional time as well, but it will not affect the performance of the agent in online situations. The other advantage the planner has in this scenario is that it knows to avoid using more expensive means of transport. The RL logistics component will learn to prefer using the factory with the cheaper mode of transport for its needs through experience, but any satisficing planner with an appropriate heuristic will avoid exploring these options until it runs out of possible plans involving cheaper options.

5.3.2 Experiment 2: planning + RL vs. Hierarchical MCP

Our second test compares the same RL and planning combination to a hierarchy of Monte Carlo planners using the extensions to the UCT algorithm described in section 4.3. The Q-functions for both logistics and retail are the same as those used by the reinforcement learning algorithms of the first experiments. In this test, we compare the results obtained in the first set to those of running the MCP agent on the second scenario. The results of the first scenario are omitted because the number of samples and search depth required for the logistics component to produce suitable plans reduces the number of episodes it can experience to less than a thousand per day. This is a result of the logistics task spanning 14 steps in the first scenario compared to 7 in the second. The agent can accomplish less with each step, which results in an unfavorable tradeoff between breadth and depth. Additionally, when the agent reaches a step in which the end of the task is closer than its maximum search depth, it does not need to and in fact cannot search to the full depth. Longer tasks result in more steps where the planner must build a full tree.

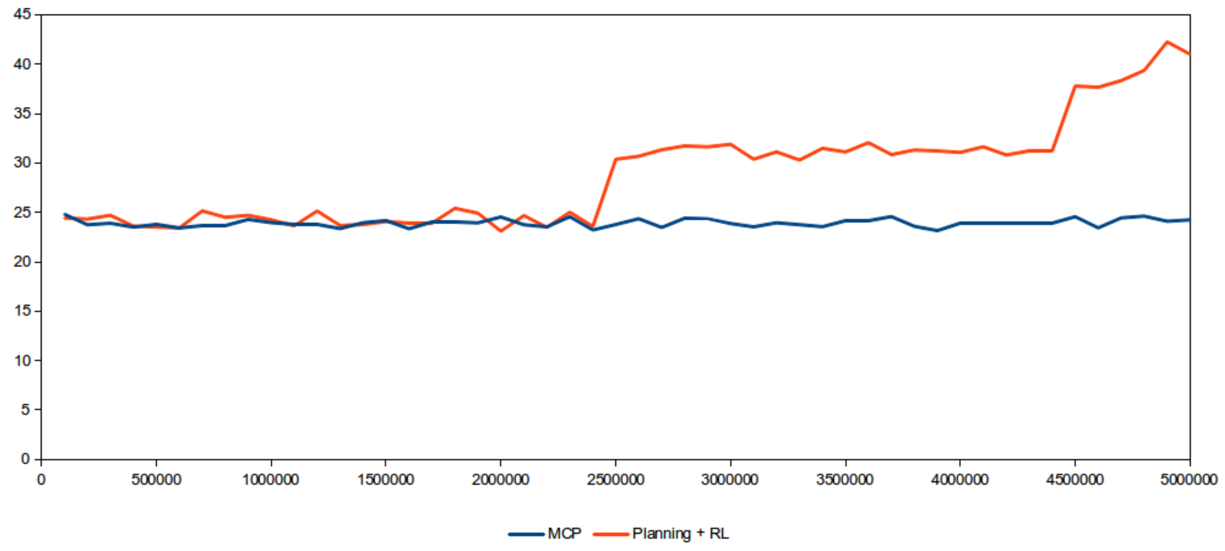


Figure 5.8: Results of running a two-level TSAS agent and a two-level hierarchy of Monte Carlo planners in a scenario with a large state-action space with difficult to obtain rewards.

As figure 5.8 demonstrates, Monte Carlo planning represents a good alternative to the specialized hierarchy at first. However, the MCP agent continues to execute almost the same policy with only slight variances based on which samples it gets, while the reinforcement learning component of the TSAS agent obtains repeated improvements in its policy. Because Monte Carlo planning algorithms do not incorporate learning, the only option for improving the performance of an algorithm like UCT is to reduce the error in its estimate of the true Q-function by obtaining more samples. Unfortunately, the number of samples needed grows exponentially with respect to the expected error [13, p. 10], leading to policies that are better but take much more time to compute, eliminating the advantage of planning over RL.

5.3.3 Experiment 3: planning + RL vs. a Very Large State-Action Space

property	values for scenario 3
cities	3 cities
stores	6 stores, 2 per city
factories	3 factories
max demand	8
max price	8
max inventory	6
max order	6
max production	2
truck cost	3 for factory 1, 4 for factory 2
train cost	8 for factory 1, 7 for factory 2
logistics time cost	1 day per action
starting money	150

Table 5.3: The third scenario used for testing two-level hierarchies in the business environment. This scenario is significantly larger than the first two, consisting of tens of millions of states and hundreds of millions of actions for the logistics task alone.

Our third experiment applies specifically to the two-level combination of planning and reinforcement learning. For this experiment, we extend scenario 2 used in the previous experiments to have six stores in three cities and three factories. We also increase the span of some of the state features. A description of this third scenario appears in table 5.3.

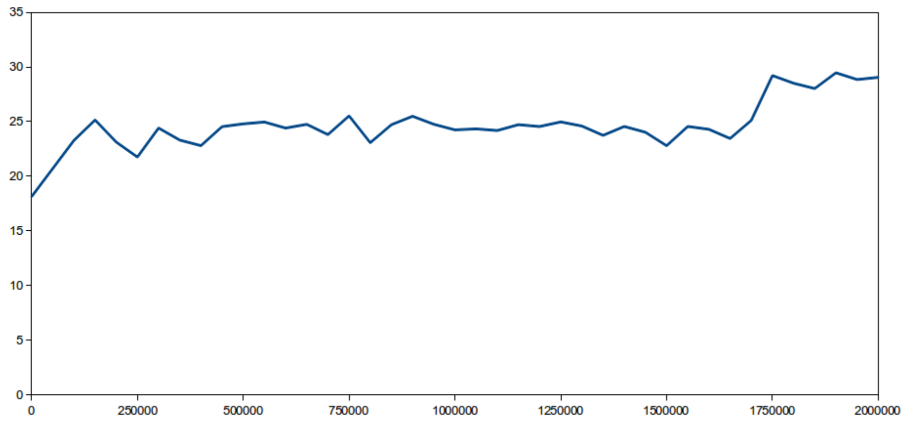


Figure 5.9: Results of running a two-level hierarchy of planning and RL on a very large scenario. The logistics task alone has approximately 10^{14} state-action pairs.

The graph in figure 5.9 demonstrates that the hybrid agent is still capable of solving extremely large problems. Because of the number of possible state-action pairs and the structure of the problem, we were unable to run our pure RL agent on this problem. Similarly, the number of samples needed for UCT to produce reasonable results in this domain were prohibitive. The fact that the hybrid agent was able to solve this problem and demonstrate improvement over time is a testament to the ability of this type of approach to solve larger problems with less time and resources.

5.3.4 Experiment 4: planning + RL + RL

property	values for scenario 4
cities	1 city
stores	2 stores
factories	1 factory
max demand	8
max price	8
max inventory	8
max order	8
max production	1
max quality	5
truck cost	3
train cost	7
logistics time cost	1 day per action
franchise decision epoch	14 weeks
starting money	150

Table 5.4: The fourth scenario, used for testing three-level hierarchies in the business environment. This scenario involves fewer locations (less stores, cities, and factories), but more state and action variables (quality and open/closed status), the management of which can be separated into a separate task.

Our fourth experiment (in table 5.4) extends the problem further by adding franchise management on top of retail and logistics management. The logistics and retail parameters are the same as scenario two from the first set of experiments. Additionally, each store can have one of five different levels of quality. Finally, the agent is given an extra challenge by starting with one store closed. The agent we use in this test performs planning for logistics tasks, RL for retail tasks, and RL for the franchise task. The Q-function formulation for the franchise task separates the state and action features by city. The franchise management task makes calls to a different retail task for each different combination of open and closed stores.

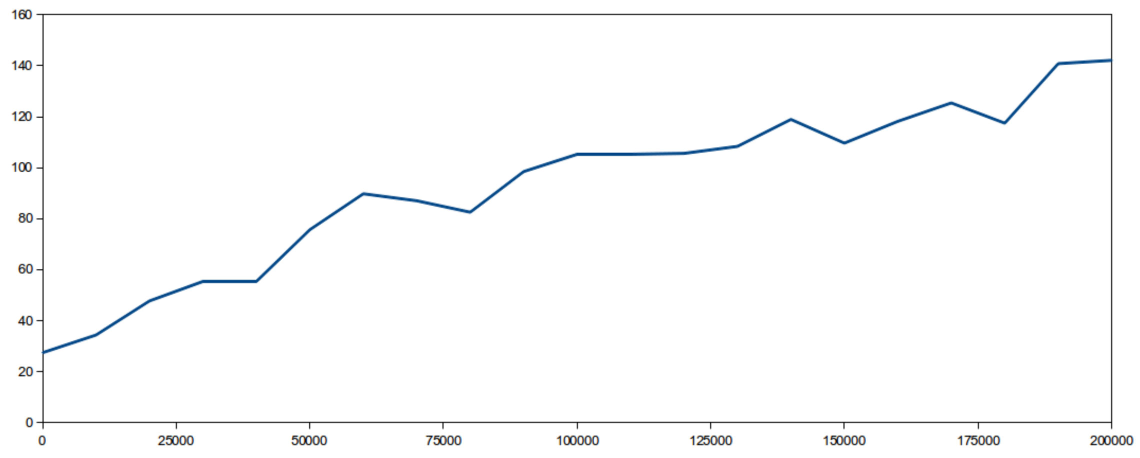


Figure 5.10: Results of running a three-level hierarchy of planning and RL on a simple scenario. The values shown in this graph represent the average values received from the real reward function, rather than a pseudo-reward function.

The graph in figure 5.10 represents the average reward received by the agent. The reward function in this scenario is the same as the one in previous ones involving a two-level hierarchy, but in this situation the additional top level of the hierarchy introduces more dependent variables. These results indicate that the agent was able to continually improve its policy over time.

task	both stores open	store 1 only	store 2 only
times tried	270k	10k	1.23M

Table 5.5: The number of times each of the retail management tasks was executed in the first experiment involving a three-level hierarchy.

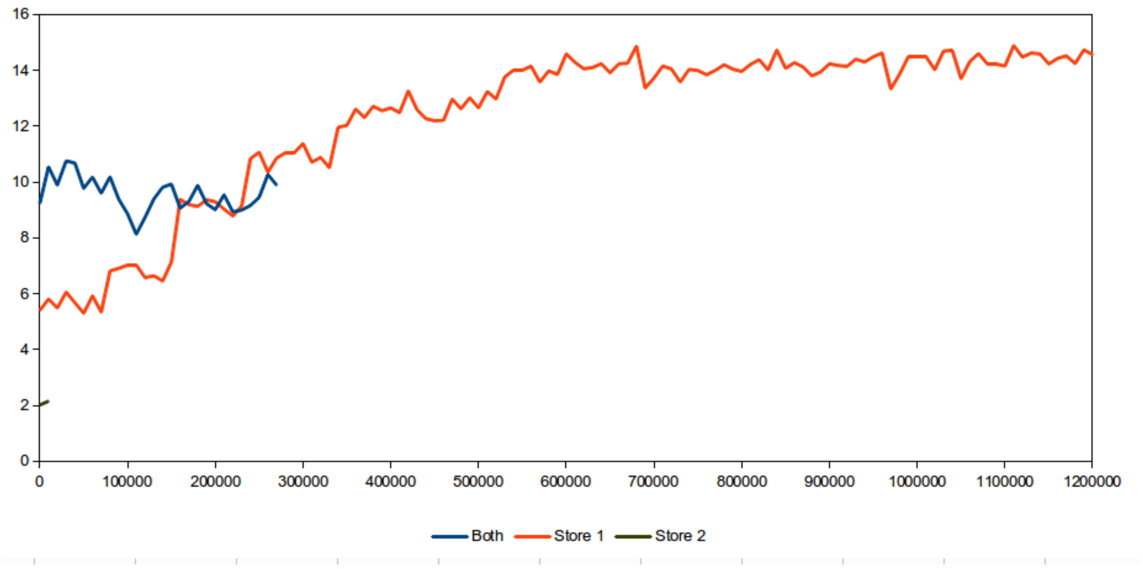


Figure 5.11: Results of running a three-level hierarchy of planning and RL on a simple scenario. The three lines represent the average reward for executing each of the three retail management tasks. The X-axis represents the number of times the task was executed.

Because the franchise management task has the ability to open and close stores, it must call on different sub-tasks depending on the current status of the stores. Table 5.5 shows the number of times each retail task was attempted and figure 5.5 shows the average reward per step for each task. One interesting property of this scenario is that it is set up so that the optimal policy is to close the first store. The initial policy that the agent used was to keep both stores open, but this changed once it explored the option of closing store 1 enough times. Although the other options are suboptimal and the agent learned to close store 1 relatively quickly, even the clearly suboptimal policy of closing store 2 and keeping store 1 open was sampled about once for every 17 task executions ($1/17 \approx .059$, slightly above the agent's exploration rate of .05).

5.3.5 Experiment 5: SEPIA

Wave	time	composition
1	250	2 archers
2	750	3 archers 2 knights
3	1250	5 footmen
4	1600	4 footmen 4 knights
5	2000	8 footmen 4 knights

Table 5.6: The unit compositions of each wave of enemies. The agent must learn to pivot its strategy from defending against ranged units to defending against melee units.

Our final experiment involves testing a different TSAS agent on SEPIA. The scenario for this experiment is one which we call "defend the base", because the agent is given a base of operations near some resources and must successfully defend it from attackers. The enemy which tries to destroy the base consists of small squads of varying types of units that spawn elsewhere in the map at fixed intervals. Our agent does not start out with any combat units. Instead, it is given a unit that gathers resources and builds buildings. By collecting resources, the agent can use its build to construct buildings which in turn produce units capable of fending off enemy units. A picture of the scenario appears in figure 5.4. The unit compositions of each wave of enemies in the scenario appear in table 5.6.

Instead of using a hierarchy with planning at the lowest level and reinforcement learning at higher levels, our SEPIA agent uses reinforcement learning at the lowest level, planning for an intermediate level, and more reinforcement learning at the top level. A diagram of this hierarchy appears in figure 5.12. The most basic task that the agent solves is combat with a single squad of units. This scenario includes three types of combat units: footmen, archers, and knights, each of which has different attack strength, defense, and range. The attacker gets the same three types of units. Given the fixed combination of units, the agent defeat the enemy while maximizing the remaining health of its units. Because the damage calculation is stochastic (actual damage is anywhere between 50 and 100 percent of the difference between attack and defense stats) and the agent does not get a model of how the enemy will behave, reinforcement learning is the most suitable approach to this problem. Because of the size of the state space, the agent uses function approximation to determine which of its units should attack which enemy unit at each timestep. Some of the features include expected damage per turn, distance, attack range, current health, and how many other units are attacking the same unit.

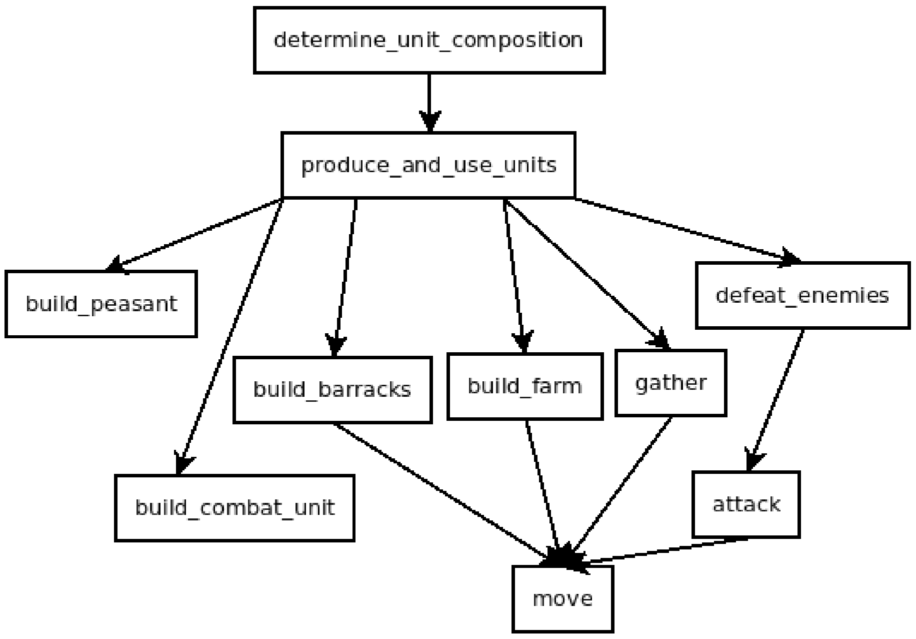


Figure 5.12: The task hierarchy of the SEPIA agent. Note that while the ”attack” and ”move” tasks technically make this hierarchy deeper than three levels, the TSAS part of the hierarchy is only three levels because we use A* search instead of planning or RL to solve these two tasks.

The upper tiers of the task hierarchy involve unit production. As previously mentioned, the agent does not start out with any combat units. If the combat task is to have a fighting chance, it must be given sufficient units in a timely manner, or else it will get overrun by the enemy. The middle tier of tasks involve producing a specific amount of units. Unit production involves a very long sequence of deterministic tasks, so the middle level uses resource-based planning to find the optimal production sequence. The top level of the hierarchy determines what to produce. As with the combat task, the army composition task uses RL with function approximation. The state features it considers include:

- the current amounts of each unit type it has,
- average health by unit type,
- expected amounts of each type of enemy unit to appear in the next wave, and
- how long it has until the next wave appears.

This task has only one child task, unit production, so its action choices consist of the different combinations of parameters, i.e. how much of each unit to produce. The top level task learns to obtain better rewards by selecting the right match-ups between different types of units. It must also learn to request the right amount of units; too few units and it will lose, too many and the child task will fail because it won't produce the units in time.

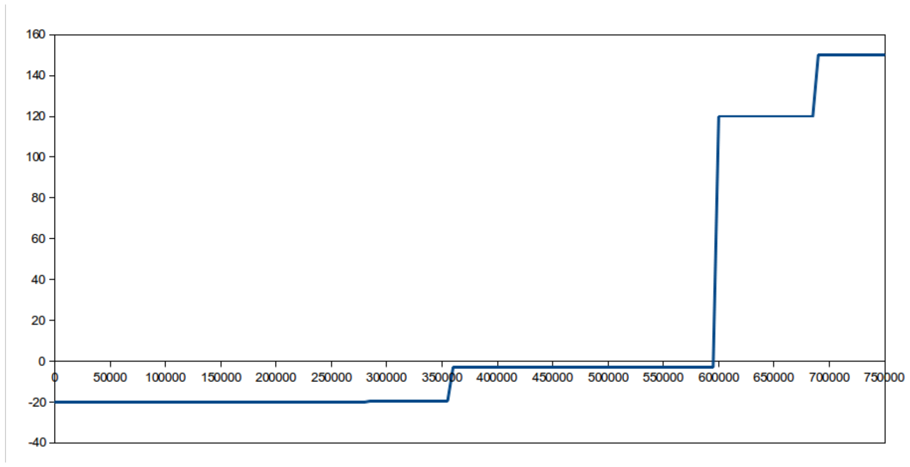


Figure 5.13: Results of running the hierarchical agent on the ”defend the base” scenario in SEPIA. Each increase in reward coincides with the agent learning the right combination of units to survive another wave.

The graph in figure 5.13 shows the performance of our hierarchical agent in the base defense scenario. The reward function for this scenario is sparse - the agent only receives a reward when it defeats a wave of enemies. As a result of sparse rewards an the learning rate we selected, the agent adjusted its policy to consistently survive another wave very rapidly. Each step in the average reward corresponds to the agent learning to survive an additional round, with the last level corresponding to successfully surviving all waves. Based on the agent's performance in this scenario, we expect that using the TSAS approach to hierarchical agent design will lead to better agents for even larger problems, such as ones where the enemy has a base of its own which must be destroyed.

Chapter 6

Conclusion

This thesis introduced Task-Specific Algorithm Selection, a new approach to solving decision-making problems. TSAS is a framework for hierarchical agents that uses planning or reinforcement learning for individual tasks. This approach is an improvement over pure planning or pure reinforcement learning because it combines the strengths of both in a synergistic way. A TSAS hierarchy that makes use of planning for some tasks is guaranteed to produce results as good as those produced by a hierarchy composed purely of RL algorithms. Additionally, the fact that planning does not need experience to make optimal decisions can significantly reduce the amount of experience needed to converge to a fully optimal hierarchical policy.

We demonstrated the effectiveness of TSAS in several scenarios and multiple domains. These experiments demonstrated that:

- a hierarchy in which a lower level tasks use planning performs better in earlier episodes than a hierarchy that only uses RL,
- the policy of following a plan produces more consistent behavior than an exploratory policy, which speeds up the convergence of parent tasks,
- planning is uses computational resources more efficiently, letting it scale to larger problems,
- TSAS works in multiple different task hierarchies, and
- TSAS works in multiple domains.

TSAS's flexibility makes a number of extensions possible. Because it works with any metric-based planner, other planning algorithms besides the one in our implementation would function just as well. While resource-based planning worked particularly well for the domains we used in our experiments, other planning algorithms such as SATplan and Graphplan may be more efficient at solving tasks in other domains. The incorporation of different planners is one way in which future work could further extend the capabilities of TSAS.

Another direction for future research lies in combining TSAS with other algorithms that mix planning and RL. Because it makes no additional stipulations beyond those of MAXQ with respect to what reinforcement learning algorithms will be hierarchically optimal, it is possible to use more specialized approaches than Q-learning and SARSA. Because the Planned HSMQ-Learning and Teleo-Reactive Q-Learning algorithms proposed by Malcolm Ryan work on similar guarantees, it seems reasonable that these two algorithms would be compatible with TSAS. Attempts to combine the two could result in a hierarchical agent which can handle a whole continuum of problems of from those with almost no model to others with nearly complete models.

In conclusion, we have presented a novel way to combine to existing approaches to decision-making processes. One of the biggest obstacles to reinforcement has been the fact that it does not scale well to large problems. A major limiting factor for planning is that it requires a model of the problem, and its solutions are only as accurate as the model. This work was one step towards tearing down those barriers and fostering more widespread adoption of these two approaches.

LIST OF REFERENCES

[1] Fahiem Bacchus and Michael Ady. Planning with resources and concurrency: A forward chaining approach. In *International Joint Conference on Artificial Intelligence*, volume 17, pages 417–424. LAWRENCE ERLBAUM ASSOCIATES LTD, 2001.

[2] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences of the United States of America*, 38(8):716, 1952.

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

[4] Ronen I. Brafman and Moshe Tennenholtz. R-max-a general polynomial time algorithm for near-optimal reinforcement learning. *The Journal of Machine Learning Research*, 3:213–231, 2003.

[5] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Online planning for resource production in real-time strategy games. In *Proceedings of the International Conference on Automated Planning and Scheduling, Providence, Rhode Island*, 2007.

[6] Thomas G. Dietterich. The MAXQ method for hierarchical reinforcement learning. In *Proceedings of the fifteenth International Conference on Machine Learning*, volume 8. Citeseer, 1998.

[7] M. Fox and D. Long. PDDL 2.1: An extension to PDDL for expressing temporal planning domains. Technical report, Department of Computer Science, Durham University, UK, 2002.

[8] M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[9] Michael Grounds and Daniel Kudenko. Combining reinforcement learning with symbolic planning. *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning*, pages 75–86, 2008.

[10] C. Guestrin, D. Koller, and R. Parr. Computing factored value functions for policies in structured MDPs. In *Proceedings of the Sixteenth International Joint Conference on Articial Intelligence (IJCAI- 99)*, 1999.

[11] J. Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[12] Henry Kautz, Bart Selman, and Yueyen Jiang. A general stochastic approach to solving problems with hard and soft constraints. *The Satisfiability Problem: Theory and Applications*, 17:573–586, 1997.

[13] Michael Kearns, Yishay Mansour, and Andrew Ng. A sparse sampling algorithm for near-optimal planning in large Markov decision processes. In *International Joint Conference on Artificial Intelligence*, volume 16, pages 1324–1331. LAWRENCE ERLBAUM ASSOCIATES LTD, 1999.

[14] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

[15] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in Neural Information Processing Systems*, pages 1043–1049, 1998.

[16] Aswin Raghavan, Saket Joshi, Alan Fern, Prasad Tadepalli, and Roni Khardon. Planning in factored action spaces with symbolic dynamic programming. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[17] Gavin A. Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*. University of Cambridge, Department of Engineering, 1994.

[18] Malcolm R. K. Ryan. *Hierarchical Reinforcement Learning: A Hybrid Approach*. PhD thesis, University of New South Wales, School of Computer Science and Engineering, 2004.

[19] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3(1):9–44, 1988.

[20] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, volume 216, page 224, 1990.

[21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[22] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: Learning, planning, and representing knowledge at multiple temporal scales. In *Journal of Artificial Intelligence Research*, 1998.

[23] Christopher J.C.H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.