Using Hierarchical Reinforcement Learning to Balance Conflicting Sub-Problems

Stephen Robertson
Department of Computer Science
Rhodes University
Grahamstown, 6140
g02r3566@campus.ru.ac.za
Supervised by: Mr Philip Sterne

November 8, 2005

Abstract

In this paper two approaches to hierarchical reinforcement learning are applied to a complex gridworld navigation problem. The first method is an adaption of Feudal Reinforcement Learning by Dayan and Hinton, and the other is a novel method called the State Variable Combination approach (SVC), designed for a problem consisting of multiple conflicting sub-problems. Feudal Reinforcement Learning was not easily adaptable to the gridworld navigation problem, and proved inefficient. SVC proved successful for most cases, but was erratic in its performance.

Contents

Chapter 1

Introduction

Reinforcement learning is an attractive form of machine learning that has been successfully applied to problems in the past, such as a backgammon playing program, called TD-Gammon [Tesauro, GJ (1995)]. However, as the complexity of a given problem increases, efficiency decreases exponentially. This is the main disadvantage of the method, and research on reinforcement learning tends to be aimed at methods of getting around this.

Any given problem will have associated with it a number of different states which it can be in. From any one of these states there are a set of actions which can be performed. Reinforcement learning maps predicted rewards to all state-action pairs, based on previous rewards received. The set of all possible states is known as the state space. Increasing the complexity of a problem can cause an explosion of the state space. This causes learning time to decrease, as there are a vast number of states that need to be experienced in order for learning to occur. This is known as the curse of dimensionality. This also forces sub-problems to be dependent on each other, even when they aren't inherently dependent. Learning to solve sub-problem A, while sub-problem B is in one state, does not teach the agent anything of how to solve sub-problem A, while sub-problem B is in another state, even if the solution is the same, which is usually the case.

If a complex problem consisting of multiple conflicting sub-problems could be broken up into its separate sub-problems, and each sub-problem tackled individually, with a "Divide and Conquer" approach, this would lead to a much more efficient solution to the problem. This approach is termed hierarchical reinforcement learning, because of the manner in which the overall problem is divided up into a hierarchical structure of sub-problems. Many different approaches to hierarchical reinforcement learning exist, each with different approaches to breaking up the overall problem into sub-problems, and each with different levels of designer intervention. Some methods aim at breaking up the actual state space into a hierarchical structure, while others deal

more with directly breaking up the problem into sub-problems. Some methods try to automatically identify sub-problems within a given problem, while others rely on the designer to explicitly define sub-problems. These methods will be discussed in the next chapter.

In this project a complex gridworld problem was designed in which a primitive creature had to navigate the gridworld, keeping health high and thirst and hunger low. These are all conflicting sub-problems as they are all solved by a completely different series of actions. They all need to be balanced, as just keeping one of the needs satisfied is not sufficient. All three sub-problems need to be solved individually, balancing them so that each of them is in an optimally solved state simultaneously. A hierarchical reinforcement learning approach is therefore well suited to solve this problem. Two different hierarchical reinforcement learning approaches were implemented to solve the complex gridworld navigation problem and compared to that of a flat reinforcement learning approach.

In chapter 2 of this thesis various previous designs and implementations of algorithms of Hierarchical Reinforcement Learning are discussed, and how applicable they are to the complex gridworld navigation problem. In chapter 3 the complex gridworld navigation problem is fully described. Chapter 4 discusses a flat reinforcement learning approach that is used as a benchmark against further solutions to the problem. In chapter 5 the adaption and implementation of Feudal Reinforcement Learning by [Dayan and Hinton, 1993] to the complex gridworld problem is discussed and in chapter 6 an original algorithm called the State Variable Combination approach (SVC) is described and implemented. Chapter 7 concludes the thesis with a discussion of the various methods attempted.

Chapter 2

Hierarchical Reinforcement Learning Survey

Reinforcement learning is a form of machine learning which learns which action to take for a given situation in order to maximise a numerical reward. The learner is not told what to do, as with the majority of other machine learning algorithms, but rather is given a measure of how good an action is which it performed by means of a reward. A chosen action not only affects the given reward, but also the resulting state the learner will find itself in after taking the action, and hence all further rewards. A learner will have to do a fair amount of exploration in order to learn the solution to a problem. Learning by trial and error, and delayed rewards are the two main factors which distinguish reinforcement learning from other forms of machine learning [Sutton and Barto, (1998)].

Reinforcement learning keeps track of predicted rewards for predicted actions for a given state by means of a table called a Q-table. After every step, the predictions are updated according to the real reward received by the learner. At any time the learner will have a highest predicted reward for every action from every state. This set of most optimal actions for every state is known as the learner's policy.

In this section I will give a brief overview of some common algorithms of hierarchical reinforcement learning, including those in which the hierarchical structure is supplied by the designer, and also those in which the agent attempts to uncover the hierarchical structure for itself.

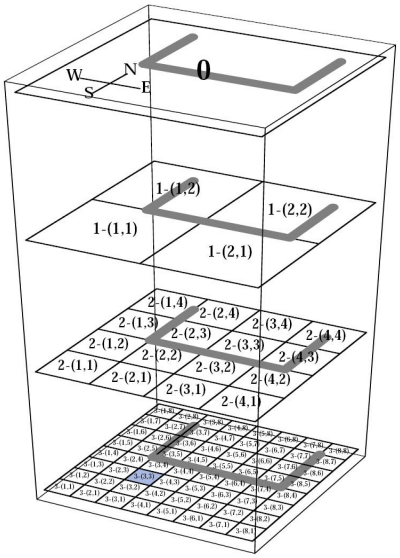

Figure 2.1: Hierarchical breakdown of a maze into sub-mazes from [Dayan and Hinton, 1993]

2.1 Feudal Reinforcement Learning

[Dayan and Hinton, 1993] give an approach to hierarchical reinforcement learning, in which the hierarchical structure is given by the designer. It is called Feudal reinforcement learning and in it, they apply their algorithm to a task in which an agent has to navigate a gridworld maze which contains in it a barrier. The hierarchical structure is designed by the designer to consist of multiple levels. At each level of the hierarchy the gridworld blocks are grouped into increasingly large blocks, as in figure 2.1.

Managers are assigned at each level of the hierarchy. Managers are in charge of choosing which sub-manager to assign next to complete a desired goal. Therefore each manager will have a super-manager, except for the highest level manager. In the same way, every manager will have a number of different sub-managers which it will have control over, and only the lowest level managers are allowed to perform actions inside the gridworld. There are certain rules that need to be applied when applying this method. Managers must reward sub-managers for achieving the manager's desired task, even if this task is not desired by the super-manager. Similarly, sub-managers do not get rewarded even if they perform the desired task of the super-manager, unless they satisfy what the manager has told them to do. Therefore rewards are strictly only given one level down the hierarchy. Managers also only need to know the state of the system at the

granularity of their own choice of tasks.

In testing the performance of this system, the designers tested it on a given gridworld and compared the results to that of a flat reinforcement learning agent. For the first few iterations, the flat reinforcement learning agent outperformed the Feudal reinforcement learning agent, but the Feudal reinforcement learning agent quickly overtook the performance of the flat reinforcement learning agent and after about 500 iterations was by far outperforming it.

A method of dealing with a problem with conflicting sub-problems is that of hierarchical reinforcement learning. Hierarchical reinforcement learning is an approach to reinforcement learning which splits a given problem into a number of smaller sub-problems, and then attempts to tackle each sub-problem separately. By doing this the overall state space is decreased and therefore the efficiency increased. In [Kaelbling, Littman and Moore, 1996], hierarchical reinforcement learning is mentioned as a good way of increasing efficiency in reinforcement learning.

In hierarchical reinforcement learning, conventional methods of reinforcement learning are used for learning at each level of the hierarchy, and therefore hierarchical reinforcement learning does not lose the desirable qualities of reinforcement learning. Reinforcement learning is very good at dealing with stochastic errors which might creep into the description of the state. A hierarchical reinforcement learning approach to navigating a gridworld was used in [Bakker and Schmidhuber, 2004], and to test its ability to handle stochastic errors, the latter were introduced into the description of the state. The algorithm dealt with them very well, and was almost as efficient as the errorless example.

For reinforcement learning, the current choices of actions by an agent for its different states is known as its policy. In hierarchical reinforcement learning, with the breaking up of an overall goal into subgoals, comes the introduction of subpolicies. These subpolicies are a common aspect in literature on hierarchical reinforcement learning and are given different names by various researchers, such as actions, options, skills, behaviours and modes [Barto and Mahadevan, 2003].

2.2 A Method of Reducing the State Space in Hierarchical Reinforcement Learning

[Asadi and Huber, 2004] give a method for dividing up the global state space of a problem into smaller state spaces. Their method is an extension of a method called ϵ-learning. It describes how a problem's state space can be divided up into a series of intervals if the states in the same interval block have the same properties in terms of transitions and rewards. This type of division

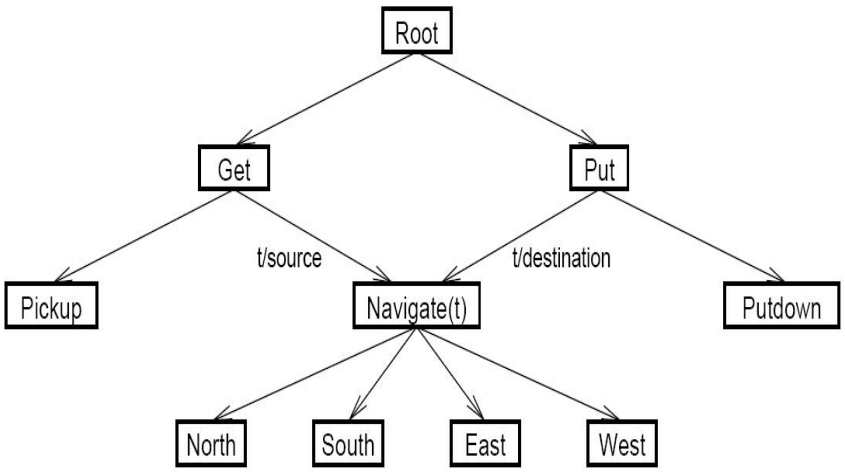

Figure 2.2: Hierarchy of sub-problems for the taxi problem from [Dieterich, 1999]

can be used to reduce the overall state space of the given problem. This method is not directly related to this thesis. However, the different divisions of the state space could be thought of as subgoals, and hence this method fits in with the idea of automatically discovering subgoals.

Asadi and Huber describe how the method was tested on a gridworld problem to see its ability to reduce state space. The algorithm worked well, and reduced the state space to the full extent which the restrictions of the method allowed.

2.3 MAXQ Value Function Decomposition

Another method of hierarchical reinforcement learning, called MAXQ value function decomposition, is proposed by [Dieterich, 1999]. It is an algorithm which expects the hierarchical structure to be supplied by the designer. It suggests breaking the main problem's value function up into an additive combination of smaller value functions, each associated with a smaller problem. As an example the author gives a problem where a taxi agent needs to move around a gridworld picking up and dropping off passengers at there desired locations. The author breaks up the problem into a hierarchy of problems, represented by figure 2.2. The MAXQ algorithm is tested against flat Q-Learning and significantly outperforms it.

2.4 The HASSLE Algorithm

In [Bakker and Schmidhuber, 2004] the HASSLE algorithm is introduced. It is described in more rigorous mathematical detail than some of the other algorithms, such as that on Feudal reinforcement learning [Dayan and Hinton, 1993]. It follows the same general idea as the most of the other algorithms, but is more advanced and has the ability to automatically discover subgoals and extract high level observations from the given information, without the intervention of a designer. For simplification, in the description of the algorithm, only two levels of the control hierarchy are considered, but in general, the control hierarchy will be the same at all levels.

At the higher level there is a high level policy π^H and at the lower level there are a number of low level policies π^L. The action of a high level policy is to request a low level policy to perform a specific subgoal. The subgoals are a set of high level observations o^H which is completely different from and smaller than the low level observations o^L. π^H takes as its input o_s^H, the current observation, and its action is another high level observation o_g^H which is the next desired observation. There are a limited set of low level policies π_i^L, none of them are initially associated with a specific subgoal o_g^H or any of the possible o_s^H, the associations are learned over time. Every π_i^L contains a table of so-called C-values of (o_s^H,o_g^H) pairs each which represents the capability of π_i^L to reach subgoal o_s^H from the high level observation o_s^H, the input observation. For the current (o_s^H, o_g^H) pair, a low level policy is selected based on its capability, C(o_s^H,o_g^H), to get from the current observation to the desired observation. If a low level policy gets to the desired observation o_g^H, its capability, C(o_s^H,o_g^H), is increased, otherwise its capability is decreased. The low-level policy also receives a positive reward if it reaches the subgoal, and zero reward if it doesn't, so that it becomes better at reaching it. Hence different low-level policies will have different capabilities of reaching different observations o_g^H, and therefore different low-level policies learn to specialise at the tasks that they are the most capable of. To arrive at high level observations, the HASSLE algorithm is not constrained to any particular method. In the example given, a method called ARAVQ (Adaptive Resource Allocation Vector Quantization) was used.

Bakker and Schmidhuber tested the HASSLE algorithm against various flat reinforcement learning agents, some using linear function approximation and others using multilayer feed-forward neural networks for value function approximation. HASSLE outperformed all the flat reinforcement learning algorithms.

2.5 Automatic Discovery of Subgoals

As [Bakker and Schmidhuber, 2004] highlights, most studies on hierarchical reinforcement learning assume that the actual hierarchical structure is given by the designer, and the agent is allowed to learn within this concrete structure. For most problems, where the hierarchical structure can easily be extracted, this is sufficient. However, in some cases, it might be desirable for the agent itself to be able to identify subgoals, as this minimises the designers role in the system. It also makes the learning process more flexible, as an agent can automatically learn a new hierarchical structure, if the hierarchical structure is suddenly changed.

In this thesis, both algorithms of hierarchical reinforcement learning used, the modified Feudal reinforcement learning algorithm and SVC, have built in methods of discovering subgoals, and hence techniques of automatic discovery of subgoals are not necessary. However, algorithms which do not have the ability to automatically discover subgoals, could make good use of the following methods. The methods are briefly described to give the reader a general idea of alternative methods available to a designer of a hierarchical reinforcement learning algorithm, and are not discussed further, as they are not relevant to methods applied in this thesis.

2.5.1 Automatic Discovery of Subgoals in Hierarchical Reinforcement Learning Using Diverse Density

[McGovern and Barto, 2001] propose another algorithm for the automatic discovery of subgoals. Their theory is that subgoals often occur at bottlenecks in the state space. As an example they describe a gridworld in which there are different rooms and connecting the rooms are doorways. If an agent starts inside one room, and the goal is in another room, getting through the door to the other room could be thought of as a subgoal, because it needs to be accomplished before the final goal is reached. The doorway can also be thought of as a bottleneck in the state space, and hence the correlation between bottlenecks and subgoals. Bottlenecks are not confined to navigation tasks though, and the concept of bottlenecks can be applied to a range of different state spaces. As a general definition of bottlenecks the authors talk about an agents path through the state space as its trajectory, and say that a bottleneck is a region which the agent experiences somewhere in the state space on every successful trajectory, but not at all on an unsuccessful trajectory. A method of finding bottlenecks for general state spaces, and hence for finding subgoals is then given, which uses the concept of diverse density.

These types of bottleneck subgoals show that the reason hierarchical reinforcement learning would be more efficient than flat reinforcement learning isn't only because breaking goals up

into subgoals significantly decreases the state space, but also because it enables more efficient exploration. In flat reinforcement learning, if an agent starts in a room, but has to get through a door to get to the goal, it might spend far too much time exploring the first room, because the chances of it finding its way through the door while still exploring are quite slim.

The algorithm was tested in a gridworld navigation task, and the agent correctly identified a doorway as a subgoal, which was the desired behaviour.

2.5.2 Automatic Discovery of Subgoals Using Learned Policies

[Goel and Huber, 2003] offer another method of automatic subgoal discovery. The authors agree with [McGovern and Barto, 2001] that a good example of a subgoal is that of a doorway in a gridworld navigation task. They describe a subgoal as a state with the following structural property: the state space trajectories originating from a significantly larger than expected number of states lead to the subgoal state while its successor state does not have this property. This property is due to the fact that states that are not a subgoal have a much higher connectivity than states that are a subgoal. This can be understood quite easily by considering the gridworld navigation task. The doorways have much less connectivity with other states than the open spaces in the rooms, and therefore would be identified as subgoals by this algorithm, which is what is desired.

This algorithm was also tested on a gridworld navigation task, where there were multiple doorways, and the algorithm recognised all doorways as subgoals, except one, which it failed to recognise as a subgoal. The authors explained that the reason for this one failure was that the size of the room with the undiscovered doorway subgoal was significantly smaller than the other rooms. Although this did not seem like a significant drawback, it suggests that this algorithm may have limitations.

2.6 Hierarchical Reinforcement Learning Implementation to The Settlers of Catan

Hierarchical reinforcement learning has been applied to some complex problems with great success. In [Pfeiffer, 2004] a hierarchical reinforcement learning approach was used to create a program which plays a board game called The Settlers of Catan, which is a popular modern board game in the German-speaking area. It is a very complex board game and therefore a flat reinforcement learning approach would have been inefficient. In their approach they use hierarchical

reinforcement learning and model trees for value function approximation. Both Q-learning and SARSA are used as the conventional reinforcement learning algorithms at different stages of the learning process. Self-play is used for the actual training process. Self-play is a method where an agent is played against a copy of itself. Self-play is used for learning strong policies in adversarial domains. The major drawback of self-play though is that without sufficient exploration, agents only learn to play against a very small set of policies.

2.7 Navigating Continuous Spaces using Hierarchical Reinforcement Learning

[Borga, 1993] describes his algorithm for hierarchical reinforcement learning in which two different hierarchical levels are given by the designer. The lower level is made up of certain actions, and the higher level is a set of strategies, each strategy consisting of a series of actions. The algorithm he describes is specifically for navigating continuous environments and hence the possible actions are a set of vectors, specifying in which direction to move. He gives an example problem, to which his algorithm could be applied, of trying to walk around an obstacle. In this case, an action would be a step in a certain direction and two different strategies would be to either choose a path to the left of the obstacle, or a path to the right of the obstacle.

2.8 Conclusion

Many different approaches to hierarchical reinforcement learning exist. For this thesis Feudal reinforcement learning was chosen as an appropriate algorithm to modify and implement, because it was also implemented on a gridworld navigation problem. MAXQ value function decomposition could also have been implemented, but one drawback of this algorithm is that sub-goals must be explicitly defined by the designer, and rewards for completing different sub-problems must be clearly distinguishable from each other, as sub-agents must only be rewarded for completing their designated sub-goals. This is an undesirable attribute if designer intervention is to be minimised, and algorithm extensibility is important, and therefore is not discussed further in this thesis. MAXQ value function decomposition in conjunction with some form of automatic sub-goal discovery such as sub-goal discovery using diverse density or learned policies, would be an extensible method with minimal designer intervention, and implementing this method could be a possible extension to this project, but is beyond the scope of this thesis. The HASSLE al-

gorithm and Asadi and Huber's reduction of state space methods also seem feasible for solving a gridworld navigation problem, and in designing SVC, ideas such as capabilities of sub-agents and division of state space were taken from these algorithms. In the next chapter the task to be solved is clearly defined.

Chapter 3

The Complex Gridworld Navigation Problem

In this chapter a complex gridworld navigation problem is presented. The rules of navigating the world are discussed, and sub-problems identified. A suitable reward function is decided upon to enable learning within a reinforcement learning solution.

3.1 Rules of the Gridworld

The complex gridworld navigation problem is made up of a 6x6 gridworld in which a creature can move around. It has 5 possible primitive actions; move left, move right, move up, move down and rest. There are 5 designated blocks on the maze with special properties. These are food, drink, shelter, a hazard, and wood and are represented by appropriate pictures as in figure 3.1. The creature is also represented by an appropriate picture.

The creature's overall goal is to keep health high, and to keep its thirst and hunger low. To keep all of these attributes standard, from now on these will be referred to as nourishment and hydration instead of hunger and thirst, and the overall goal is to keep all of nourishment, hydration and health high. This overall goal, can be divided into a set of sub-goals or sub-problems. These are keep nourishment high, keep hydration high, repair the shelter, rest in a repaired shelter, and finally avoid the hazard. Each sub-problem is solved in the following way:

- Keep nourishment high: The creature needs to navigate onto the food block as often as possible to keep nourishment as high as possible. The relevant state variables for this sub-problem are x, y and nourishment.

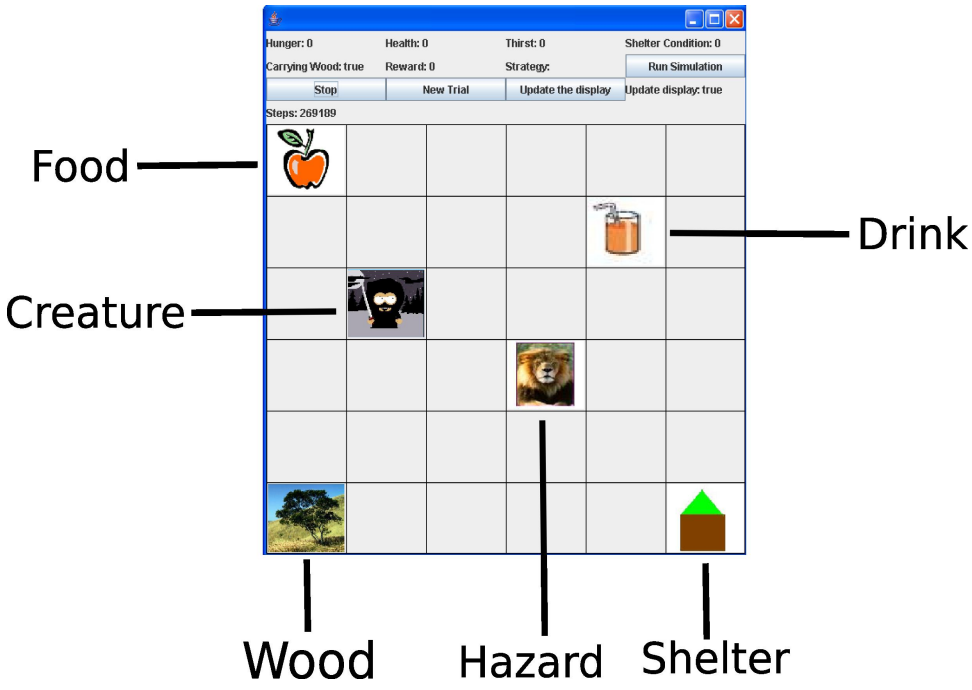

Figure 3.1: Complex Gridworld Navigation Problem Screenshot

- Keep hydration high: The creature needs to navigate onto the drink block as often as possible to keep hydration as high as possible. The relevant state variables for this sub-problem are x, y and hydration.

- Repair the shelter: The creature needs to first navigate onto the wood block to collect wood, and then navigate onto the shelter. After wood has been collected, there is no time limit for repairing the shelter, and the creature can carry wood for as long as it likes before repairing the shelter. There is no special action needed to repair the shelter, the creature just has to navigate onto it while carrying wood. The relevant state variables for this sub-problem are x, y, carrying wood and shelter condition.

- Rest in a repaired shelter: The creature needs to navigate onto the shelter block, when shelter is in as good a condition as possible, and then perform the rest action. Because this sub-problem requires the shelter to be in a good condition, this sub-problem requires the previous sub-problem to first be completed, before it can be completed. The relevant state variables for this sub-problem are x,y, shelter condition and health.

- Avoiding the hazard: Wherever the creature moves, it must avoid the hazard by not landing

on the hazard block. The relevant state variables for this sub-problem are just x and y.

At any point in time the creature's overall state can be represented by 7 different attributes. These are x position, y position, health, nourishment, hydration, shelter condition and whether it is carrying wood. X and y positions can range from 0 to 5, as it is a 6x6 gridworld. Nourishment, hydration, health and shelter condition each range from 0 to 20. Carrying wood is either true or false at any time.

After every 4 moves nourishment, hydration and shelter condition are decreased by 1. After every 10 moves health is decreased by 1. A move is either a step in a direction or a rest.

If the creature navigates onto a designated food block or drink block, its respective food or drink level is increased to its maximum of 20. If it navigates onto the designated hazardous block, its health is decreased to 0. It also receives an immediate punishment as described in the next section. If it navigates onto the wood block, it will be carrying wood from then on. If it navigates onto the shelter location while carrying wood, its shelter will be fully repaired i.e. shelter condition increased to its maximum of 20 and it will no longer be carrying wood. And finally, if it navigates onto the shelter, and then *also* rests on that block, health will be replenished by the level of shelter condition (if shelter condition is zero, health will not be replenished). Hence the better the shelter condition, the more health will be replenished per rest.

The problem isn't terminal, and there is no 'perfect' goal state to be in. The creature just carries on wandering the gridworld keeping nourishment, hydration and health as high as possible, and avoiding the hazard.

The sub-problems are conflicting, since the agent can't try to solve one sub-problem without risking that it fails to solve another and each sub-problem is solved with a completely different series of actions. The sub-problems cannot be solved simultaneously, but rather be solved separately, in a balanced fashion, so that the overall task is optimally satisfied.

With 7 different state variables, the state space is large. There are 6 possible values for x and y, 20 possible values for health, nourishment, hydration and shelter condition and 2 different values for carrying wood, and 5 possible actions making the overall q-table size $6 \times 6 \times 20 \times 20 \times 20 \times 20 \times 2 \times 5 = 57\,600\,000$. This is a vast table size, and with a flat reinforcement learning approach each of these state-action pairs would have to be experienced at least a few times to get a good idea of the associated reward for taking any action from any state. Hence flat reinforcement learning would prove inefficient.

3.2 The Reward Function

A reward function was chosen so as to mimic how a real creature living in a primitive world would receive rewards and punishments, such as contentment and pain.

The basic reward function was calculated as $(N + W + H \times 2)^2$, where N is nourishment, W is hydration, and H is health. Health was multiplied by 2 because keeping health high is a more complex task than keeping hydration and nourishment high, and it also takes more steps to complete. The whole function was squared in order to make it much more favourable to have all of nourishment, hydration and health high rather than just having one of them high. This reward is only issued at certain points in the problem, when certain events occur, as described in the next paragraph.

For food and drink, a reward is only given immediately after the creature lands on a food or drink block, and are not given if $N > 15$ or if $W > 15$. This prevents the creature from staying on a food and drink block, continuously receiving a reward for it. This also closely mimics reality, as a creature in real life would not want to eat or drink immediately after it has just eaten or drunk, but rather only when it is sufficiently thirsty or hungry.

For health, a reward is also only given immediately after the creature rests. To solve the problem of continuously resting on the block and collecting rewards, a similar approach to that of food or drink could not be taken, since health is not always increased to maximum after a rest has just been made, but rather only increased proportionally to how good the shelter condition is. Instead, the creature cannot receive a reward for resting for ten steps after it has just rested. This again closely mimics reality, since a creature would not want to rest if it had just rested, and was not tired.

When the creature lands on the hazardous location, the reward function is ignored, and the creature receives a punishment of 1000, in other words a reward of -1000.

If the creature makes any other arbitrary move which does not result in one of the above situations occurring, a reward of 0 is given. The maximum of N, W and H are all 20, and therefore the maximum reward is given by $(20 + 20 + 20 \times 2)^2 = 6400$ (although not actually possible, because of spatial constraints), and the minimum reward is -1000. The creature can therefore get a reward from $-1000 \rightarrow 6400$ for any action.

In order to be able to make sensible comparisons between the different methods presented in this thesis, the various items in the navigation problem, i.e. food, drink, etc., are given set positions within the gridworld in remote locations from each other. Figure 3.1 shows these set positions.

3.3 Conclusion

In this chapter the complex gridworld navigation was fully described, sub-problems were identified and a suitable reward function was established. With a suitable gridworld navigation problem established, it can now be solved by various algorithms of reinforcement learning. In the chapters that follow, the attempts will be made at solving the algorithm with a flat reinforcement learning approach and with 2 different hierarchical reinforcement learning approaches, one an adaption of Feudal reinforcement learning by [Dayan and Hinton, 1993] and the other a novel method called SVC which intelligently breaks up the state space into a set of smaller state spaces, each with a different combination of state variables.

Chapter 4

Flat Reinforcement Learning Implementation

In order to have a benchmark with which to compare different methods of solving the complex gridworld navigation problem, a flat reinforcement learning solution was implemented. To present fair results, the flat reinforcement learning algorithm was extensively optimised.

4.1 The Flat Reinforcement Learning Algorithm

Sarsa was chosen as the flat reinforcement learning approach, and used eligibility traces in order to speed up learning. This method of reinforcement learning is given the abbreviation $Sarsa(\lambda)$ from [Sutton and Barto, 1998], and will be referred to as such from now on. The $Sarsa(\lambda)$ update rule is given as follows:

Initialise $Q(s, a)$ arbitrarily and $e(s, a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ϵ-greedy)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow 1$
    For all s, a:
      $Q(s, a) \leftarrow Q(s, a) + \alpha\delta e(s, a)$
      $e(s, a) \leftarrow \gamma\lambda e(s, a)$
    $s \leftarrow s'; a \leftarrow a'$
  until s is terminal

The state and action are given by s and a respectively. The next state and next action are given by s' and a' respectively. Q is the q-table which holds the reward predictions for the state-action pairs (s,a), and e is the eligibility trace table, which holds eligibility of each state-action pair (s,a) for the given reward, represented by r.

A possible alternative to $Sarsa(\lambda)$ would be Q-learning, but $Sarsa(\lambda)$ proved sufficient for the problem at hand, and known drawbacks of $Sarsa(\lambda)$ such as irrational fear of punishments, did not have any unwanted effects.

The parameters λ, γ and α each have a mathematical function within the Sarsa(λ) algorithm, but also have an intuitive meaning associated with each of them. λ is the decay constant of the eligibility trace, and therefore represents in what proportion action leading up to a favourable state get rewarded. γ is the degree to which the predicted reward for the next state and next action affect the updating of the current prediction of the state and action. It is a "Look ahead" constant which determines how important future rewards are in the problem. α is the degree to which the previous predicted reward is updated with the new received reward. It is the extent to which we trust current reward signals over the prediction obtained from all past rewards. The parameters were given values which seemed to be intuitive for the given problem, and were then adjusted until they worked well for the gridworld navigation problem. The values given are $\lambda = 0.9$, $\gamma = 0.9$ and $\alpha = 0.2$.

Since the task is not terminal, the eligibility trace is never cleared. However, as the traces gradually decay, states and actions far enough into the past will have little or no effect on the updates to the Q-table.

4.2 Reducing the State Space

To make it feasible to solve the problem with a flat reinforcement learning algorithm, the state space had to be reduced. Each of nourishment, hydration, health and shelter condition were minimised to have just 5 levels each. The following table shows the manner in which state levels were divided.

State Variable Value	Minimised State Variable Value
0	0
$1 \rightarrow 5$	1
$6 \rightarrow 10$	2
$11 \rightarrow 15$	3
$16 \rightarrow 20$	4

This was achieved through

$$Minimised\ State\ Variable\ Value = \lfloor (State\ Variable\ Value + 4)/5 \rfloor$$

With x position and y position each having 6 possible states, nourishment, hydration, health and shelter condition, 5 possible states, and carrying wood, 2 possible states, and with 5 actions, gives the overall q-table size as $6 \times 6 \times 5 \times 5 \times 5 \times 5 \times 2 \times 5 = 225\ 000$. This is still quite large, but far more manageable than the alternative of $6 \times 6 \times 20 \times 20 \times 20 \times 20 \times 2 \times 5 = 57\ 600\ 000$ state-action pairs.

4.3 Efficient Exploration

The task at hand inherently needs a significant amount of exploration in order to find the best solution. Therefore just implementing a simple exploration algorithm such as an ϵ - Greedy proved inefficient. Instead, the agent was set to initially totally explore, i.e. taking random moves every time, gradually decreasing the amount of exploration done, until finally the agent was exploiting fully, i.e. by following the policy of the Q-table every move. The exploration was linearly decayed from total exploration in the beginning to no exploration after a specified number of steps. This specified number of steps is called Ω, and will be referred to as such from now on in this thesis.

In order to encourage efficient exploration, optimistic initialisation of the Q-Tables was used, by initially setting all values in the table to 6400, the maximum possible reward. This forces the

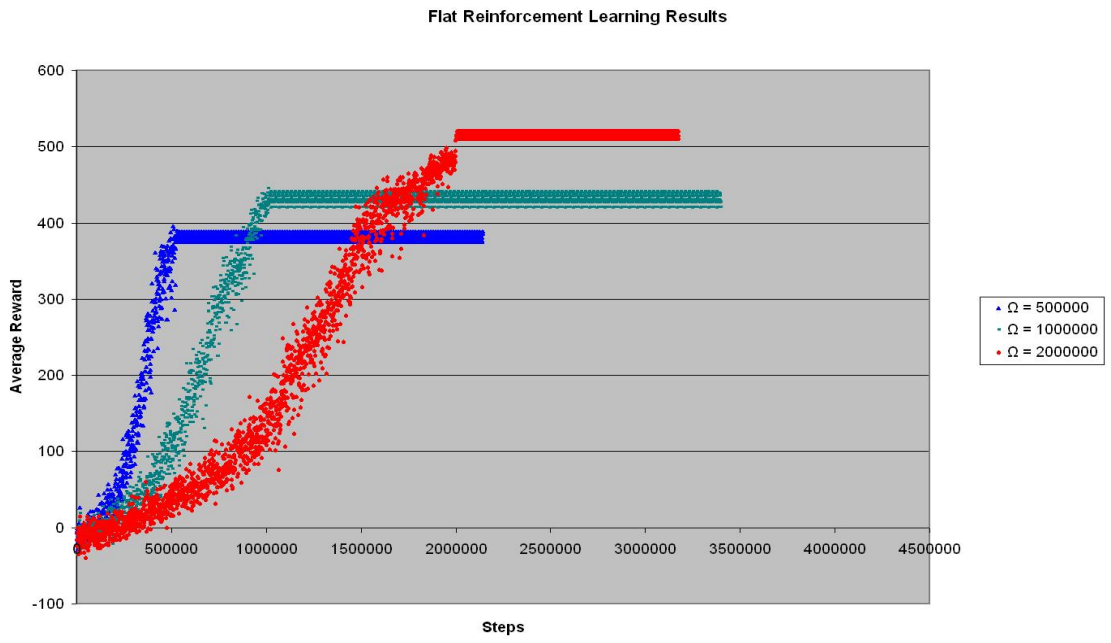

Figure 4.1: Flat reinforcement learning results for 3 trials

agent to try out all possible actions, as it will cause actions that have not been tried out before to have higher predicted rewards, and therefore choose them whenever the policy is followed. As actions which do not yield high rewards are attempted, the agent will learn that the high initial predictions were false, and the predictions will slowly decrease to their true values.

4.4 Results

Results were obtained by averaging the received reward over every 1000 steps. Because the number of steps taken to learn are large, learning times were long, often > 12 hours.

Figure 4.1 shows the flat reinforcement learning agent run for three different trials with Ω values of $500\ 000$, $1\ 000\ 000$ and $2\ 000\ 000$. For each trial the agent started off totally exploring, gradually starting to exploit, until finally it was totally exploiting, as explained previously. As can be seen, the more exploration the agent is allowed to do, the better the final solution it reaches. This emphasizes the need for a large amount of exploration in the gridworld navigation task.

The graphs may appear misleading, as it seems that the agents that explore more, learn slower, because the graphs increase less steeply. This is only because with a higher Ω value, the agent is

forced to explore more, which will often lead to the agent getting punished or missing out on a reward.

Although there is a clear difference in the final average reward reached at the end of each trial, just by visually watching the creature perform, it can be seen that for all three a very sensible policy has been adopted. The creature tends to continually move between food and drink, satisfying both thirst and hunger, until health reaches a low enough value. The creature will then collect wood, repair its shelter, and immediately rest. It will then return to alternating between eating and drinking, then just before the shelter has completely deteriorated, quickly return for one more rest, and then head back to its food and drink, and starts the loop once again. Its policy is therefore, eat, drink and be merry, until it's just too tired, and needs rest.

4.5 Conclusion

A flat reinforcement learning solution which shows successful learning has now been implemented to solve the complex gridworld navigation problem. The hierarchical reinforcement learning implementations in the chapters that follow are compared against this learning.

Chapter 5

Feudal Reinforcement Learning Approach

The first hierarchical reinforcement learning approach that was attempted to solve the gridworld navigation problem, was that of Feudal reinforcement learning, as described in [Dayan and Hinton, 1993]. This method seemed well suited, since it was also applied to a gridworld navigation problem, although a much simpler one.

Since the complex gridworld navigation problem is continual, as opposed to periodic like the simple maze problem in [Dayan and Hinton, 1993], and because the state variables can change simultaneously, Feudal reinforcement learning needed to be adapted.

In this chapter all adaptions to the algorithm are discussed, and the amount of performance impact associated with each adaption. Implementation of the modified algorithm is then discussed. The results are then presented, and the poor performance of the algorithm is then discussed.

5.1 Division of the State Space

Dayan and Hinton applied their Feudal reinforcement learning a simple maze problem, where a creature had to navigate the gridworld in order to find a designated goal block. The task is episodic, and once the goal block is found, an episode is over, and the creature is restarted in a new position. At each level of the hierarchy, each state variable, x and y in this case, were divided into 2 equal groups, and sub-agents or sub-managers as they are referred to, are each assigned a different permutation of the divisions. Each sub-manager is then divided in the same way. Hence, at each level of the hierarchy, a manager is divided into $2^2 = 4$ sub-managers.

To adapt this method of dividing the state space to a problem with n state variables, we need to divide each of the n state variables into equal groups, and put a sub-manager in charge of

each of permutations of divisions. In [Dayan and Hinton, 1993], the state variables are each divided into 2 equal groups at each level of the hierarchy, and this was left unchanged in the adapted algorithm. This results in 2^n divisions of managers at each level of the hierarchy. For the complex gridworld navigation problem, which consists of 7 state variables, this gives the number of divisions at each level as $2^7 = 128$ divisions. If this division is to occur at each level of the hierarchy, the number of sub-agents at the bottom of the hierarchy is going to be very large ($128^{(L-1)}$, where L is the number of levels of the hierarchy), and ultimately unmanageable. The number of levels was therefore limited to 2 for this implementation, with one super-manager at the top level and 128 sub-managers at the bottom level. This causes the granularity of sub-states controlled by sub-managers to be very coarse, having just two divisions per state variable, which could very easily cause major inefficiencies, but this was unavoidable due to the way in which the algorithm is designed.

5.2 Determining Abstract High Level Actions

The division of the overall state space into sub-state spaces, introduces the concept of high level states and high level actions. A high level state is made up of all states that are incorporated within any sub-manager. A high level action is a move between high level states, i.e. a switch of control amongst sub-managers.

In the simple maze problem, because the state space is defined by just two variables x and y, which by definition of the primitive actions cannot increment or decrement simultaneously, high level actions can be, and are defined as being exactly the same as the four primitive actions i.e. move left, right, up or down. A high level move would just be a directional transition from one high level grid of blocks to another.

In the complex problem, state variables can change simultaneously, i.e. shelter can deteriorate to a different level and hunger can be satisfied all in one move. For this reason, high level actions are harder to define. As a result, high level actions are defined as being the desired high level state that the agent is trying to get into at any time. In this way all possible high level actions are included, but because not all high level states are reachable from a given high level state, this causes there to be a number of meaningless high level actions for every high level state, which is undesirable, but unavoidable.

With this definition of high level actions, there are as many actions as states for the super-agent. Therefore the table size of the super-agent is $128 \times 128 = 16384$. This table only gets updated when a change of high-level state occurs, or the time-out threshold is reached, so it can

take up to the time-out number steps to occur before one update is made to the table.

There is one sub-agent in charge of every high level state-action pair, therefore 16384 sub-agents in this case. Each sub-agent has a slightly different number of states associated with it, because some state variables contain an odd number of states, and therefore a division into 2 equal groups sometimes results in a difference of 1 in the number of states of the 2 resulting state groups. However, on average, the size of a sub-agent table is, $\frac{State-Action\ Pairs}{High\ Level\ States}$ states associated with them, i.e. $\frac{225\ 000}{128} = 1757.8$, for the complex gridworld navigation problem. This would be cause a large increase in efficiency if sub-agents learnt in parallel, but in this method sub-agents can only learn sequentially, with no two sub-agents able to learn at the same time. This causes there to be 1757.8 state-action pairs to learn for 16384 sub-agents, sequentially. This is a total of $1757.8 \times 16384 = 28\ 799\ 795$ state-action pairs that need to be learnt sequentially. This is a vast number of states, and could cause there to be no increase in efficiency over flat reinforcement learning. However, this may seem misleading, as some of these state-action pairs are incorporated within sub-agents that describe an impossible transition between 2 high level states, and would therefore never have to learn if these high level state-action pairs are never chosen by the super-agent, as explained in section 5.4. Nevertheless, a large number of the sub-agents will need to learn and the vast number of state-action pairs still may result in inoptimal performance.

5.3 Terminal and Non-Terminal Managers and Timeouts

In the simple maze problem, the task was terminal, and when the agent found the goal block, the task was completed. The complex gridworld problem however, is not terminal, which again causes problems with extending Dayan and Hinton's method.

Although the complex gridworld problem is not terminal, sub-managers are, because of the fact that they are called on to perform an action until completion, and then a different sub-manager is called upon. If the high level manager is already in a very desirable state, it may wish to stay in that state, and hence a high level action of the current state being the desired state was included.

Timeouts were implemented in order to give the sub-agent the appropriate reward when it has been trying to stay in the same state and hence no change of high level state has occurred. If the agent is in the same state as before after the timeout is finished, the sub-manager is rewarded if staying in the same state was the desired action, and if not gets a reward of 0. If the high level state changes before the timeout is reached, the timeout is reset, and the sub-manager is rewarded

if being in the resulting high level state was the desired action, otherwise it is given a reward of 0.

A timeout of 5 steps was initially implemented. This seemed like a reasonable value, considering that the low level state variables, x and y, describe 4 3x3 grids within the gridworld when subdivided, and 5 steps seems like a sufficient number of steps for the creature to be expected to make a transition between any of these 3x3 grids. Any of the other state variables, which change more slowly, could also change at any time, possibly causing a sooner high level state transition, which again makes a timeout of 5 seem reasonable. A timeout of 50 was also implemented in a trial to see if it made significant difference to the results, as described in section 5.6.

5.4 Real and Pseudo Rewards

In [Dayan and Hinton, 1993] it is described how the highest level manager is rewarded according to the "real" low level rewards, while the low level agents are given pseudo rewards according to whether or not they satisfy what the high level managers tell them to do. The pseudo rewards were quite easily implemented as a reward of 1 if the sub-manager performs the task required by the super-manager, and a reward of 0 if it doesn't.

Rewarding the super-manager according to the "real" rewards in the non terminal complex gridworld problem proved to be much less trivial than what it was for the simple terminal maze problem. For one, because the problem is not terminal, and because high level states often don't change for multiple steps, it is not clear when to issue these rewards. The moment a transition between high level states occurs, the predicted reward for being in the resulting high level state can not be determined. The predicted reward for being in this high level state can only be determined after the creature has spent time in the high level state collecting low level rewards. To solve this problem, an accumulated reward per step ($\frac{Accumulated\ Reward}{Steps\ Taken}$) is kept track of until a high level state transition occurs, or the time-out occurs. A value table is then updated to hold the predicted reward for being in any of the high level states. The update occurs as follows:

$$V(s) = (1 - \alpha)V(s) + \alpha(Accumulated\ Reward\ per\ Step)$$

An alternative method would be to keep track of not only the previous state and action, but also the state and action before them, and feed the accumulated reward per step for being in a state back to the previous state and action pair, and use this reward value to update the q-tables. This would be an equally valid method.

A boolean table is kept of all possible high level transitions as they occur in order to later determine which high level transitions are possible. When issuing a reward to the super-manager for choosing an action from a given state, the predicted accumulated reward as held by the value table for the chosen action (or desired state, as actions defined) is issued to the super-manager, but only if this transition was previously possible. We therefore don't reward the super-manager for choosing an impossible action. This prevents a sub-agent which has been associated with an impossible transition between high-level states from being called, and hence prevents useless learning. However, this may cause inefficiency, as some possible transitions may only rarely occur, and a reward can only ever be given for a choosing a desired transition once it has previously occurred.

This method of reinforcement learning at the super-manager level is different from any tried and trusted method of reinforcement learning, and could fail horribly, but this was unavoidable, as a simple tried and trusted method could not be implemented.

5.5 The Modified Feudal Reinforcement Learning Algorithm

When implementing the feudal reinforcement learning algorithm, because of the adjustments made to high level actions, and because of the complexity of the problem, memory started becoming a problem. Therefore the algorithm was limited to just two levels i.e. a super-manager with multiple sub-managers. The levels of hunger, thirst, health, and shelter condition were also divided into fewer discrete levels, in order to decrease the total number of possible states.

The following is a summary of the steps of the adapted algorithm:

- nextState = The current state from Maze object, for both levels
- nextAction = The action for nextState, for both levels
- Loop:
    1. state = nextState, for both levels
    2. action = nextAction, for both levels
    3. get low level action from super-manager
    4. perform low level action
    5. give super-manager low level reward

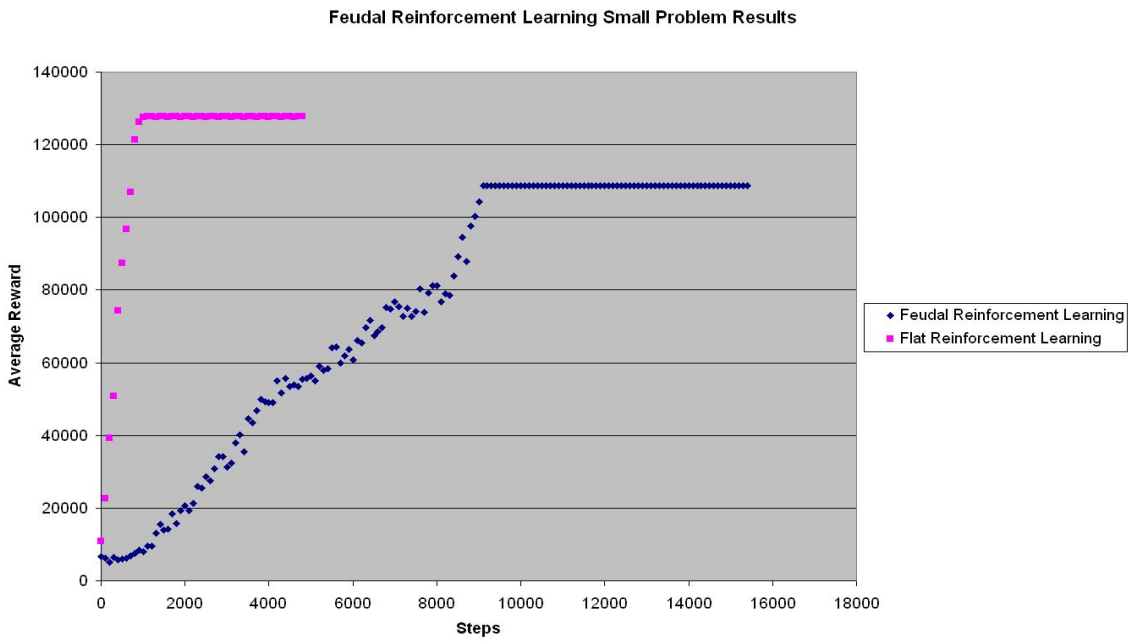

Figure 5.1: Feudal reinforcement learning results for the simplified problem

6. call learn for super-manager, which will appropriately reward and call learn for sub-managers

- Goto top of loop

$Sarsa(\lambda)$ was used as the reinforcement learning algorithm at the sub-manager level.

5.6 Results of the Feudal Implementation

In order to make sure the Feudal reinforcement learning implementation was working, it was first applied to a very simplified problem, in which very high rewards were given for keeping both nourishment and hydration high. A flat reinforcement solution, with the same method as in the previous chapter was applied to the same problem. Rewards were averaged over every 100 steps. Figure 5.1 shows the performance of the Feudal reinforcement learning implementation in comparison to the flat reinforcement learning implementation. Even with this simplified problem, it can be seen that the Feudal reinforcement learning implementation could not perform nearly as well as flat reinforcement learning.

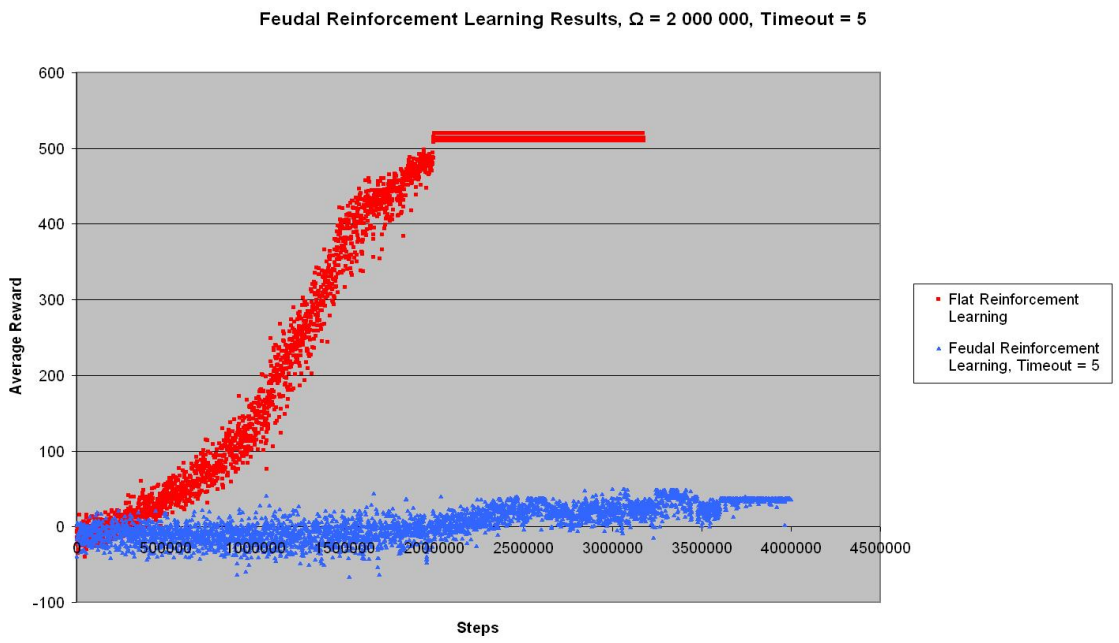

Figure 5.2: Feudal reinforcement learning results for the complex gridworld navigation task

The Feudal reinforcement learning implementation was then applied to the full complex gridworld navigation problem, with a Ω value of $2\ 000\ 000$, for 2 different trials with the timeout threshold equal to 5 steps for one, and to 50 steps for the other. These are compared to the flat reinforcement learning implementation with Ω of $2\ 000\ 000$, as in figure 5.2 and figure 5.3. For both trials the algorithm performed even worse than for the simple problem, and never reached an average reward higher than 50. The trial with a timeout of 5 showed slightly better results than the trial with a timeout of 50, as expected, as 5 seems like a high enough value, although there was no significant difference, and both trials showed poor results.

5.7 Failure of Feudal Reinforcement Learning

The main reason for the inefficient performance is mainly due to the fact that high level actions are not easy to define, resulting in there being far too many high level actions. With high level actions being defined as the high level state which it is desirable to get into, results in there being 128 high level actions, for each of the 128 high level states, each of which initially having an equal probability of being chosen, even though some of them are not even possible. This results

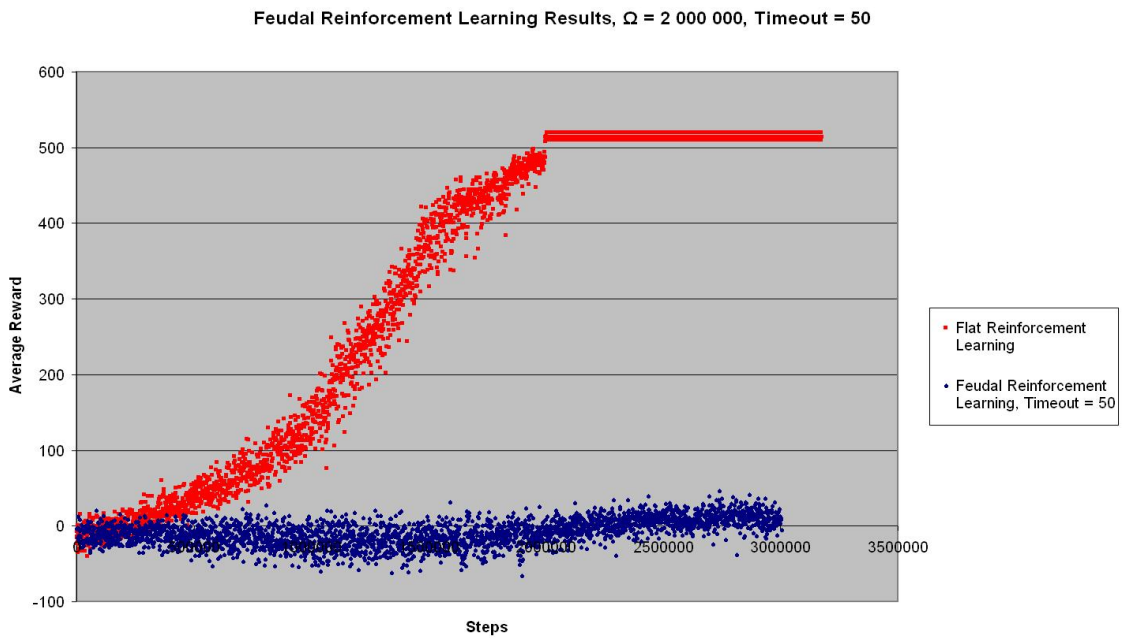

Figure 5.3: Feudal reinforcement learning results for the complex gridworld navigation task

in a lot of time being wasted in trying to do what is not even possible, and only very slowly learning which actions are possible. A direct application of Feudal reinforcement learning is only possible in a problem where high level actions are very clearly defined.

5.8 Conclusion

Dayan and Hinton's Feudal reinforcement learning was not easily adaptable to a the complex gridworld navigation problem, which although more complex, is not significantly different from the simple maze problem the algorithm was originally applied to, and is therefore not a very robust method. High level actions were difficult to define, which ultimately led to major inefficiencies in the algorithm. In the next chapter an alternative method is presented, called SVC, in which the state space is divided in a more intelligent manner, enabling sub-agents to learn in parallel, rather than sequentially.

Chapter 6

The State Variable Combination Approach

The Feudal reinforcement learning approach to solving the problem performed poorly, with no obvious direction for improvement. An altogether new approach was therefore designed and implemented. This time, instead of taking a very literal interpretation of a method which was designed to solve a very specific problem, an hierarchical approach inspired by all the methods on hierarchical reinforcement learning, but not direct clones of them, was attempted. This method was named the State Variable Combination approach (SVC), and consists of a super-agent which sets up sub-agents, each in charge of a possible combination of a given number of state variables. The super-agent is responsible for extracting useful information out of each sub-agent to determine which action is the best to perform. This chapter provides a description of this novel method and discusses its effectiveness.

6.1 Description of the Method

In a problem that consists of multiple conflicting sub-problems, the state of each sub-problem can usually be described by a limited set of all the given state variables. In this method, the overall state space is broken up into a set of smaller state spaces, each with a limited set of state variables associated with them. By including all combinations of state variables, certain sub-agents will automatically be equipped to tell the best action to perform for the given state of a certain sub-problem. Also, by including all combinations of state variables, the amount of intervention needed by the designer is minimised, which is a favourable attribute of any machine learning algorithm. If the sub-problems should suddenly be changed, and suddenly incorporate a different set of state variables, sub-agents need not be changed, as all possible combinations of state variables are already included.

Learning of sub-agents occurs in parallel, each one getting its tables updated after every action taken. There is therefore no penalty for having many sub-agents, and the smaller tables within each sub-agent should enable faster learning.

Since sub-problems need to be balanced, and solved at the correct time and in the correct order within the full problem, sub-agents need to be intelligently chosen between at each point in time within the full problem. Also, sufficient state variables need to be assigned per sub-agent in order to fully describe any of the sub-problems. Therefore, if there is a sub-problem that takes, say, 5 state variables to fully describe its state, no combination of only 4 state variables would ever be able to fully describe its state.

Given a problem with n state variables, with a minimum of k state variables required to describe any given sub-problem, there are $\begin{pmatrix} n \\ k \end{pmatrix}$ different combinations of state variables that can be assigned to any sub-agent, and are therefore $\begin{pmatrix} n \\ k \end{pmatrix}$ sub-agents necessary.

For example, for a problem with 8 state variables, with sub-problems needing a maximum of 5 state-variables to describe them, each state variable consisting of 10 different states, and 5 actions, we would have $\begin{pmatrix} 8 \\ 5 \end{pmatrix} = 56$ sub-agents, each having a q-table of $10^5 \times 5 = 50000$ state-action pairs each. The 56 sub-agents learn in parallel, each with a table of 50000 state-action pairs, so there are effectively just 50000 different state-action pairs which can be experienced. A Flat Reinforcement Learning approach to this problem would consist of $10^8 \times 5 = 500\ 000\ 000$ state-action pairs which can be experienced. This is a significant size difference factor of $10^3 = 1000$.

6.2 Choosing Between Sub-agents

Each sub-agent will contain a table consisting of all the combinations of values for the k state variables that describe the sub-agent's state, and predicted rewards for the different actions. If the problem has progressed to a point in which one of the sub-problems is the most advantageous to solve at that point in time, the sub-agent which has been assigned the state variables pertinent to that sub-problem should predict the highest reward for one of its actions. Therefore if the super-agent searches through all of the sub-agent tables at the current state of the state variables assigned to them, searching for the one which predicts the highest reward for one of its actions, it should find the sub-agent which is the best to obey, and the action with the highest predicted reward should be the best action to take.

One problem with this method of choosing the best action to take, is that some sub-agents which do not describe any given sub-problem, might falsely predict a high reward for a given action, and would incorrectly be chosen by the above method. Therefore the reliability of a sub-agents prediction also needs to be taken into account when choosing between sub-agents. An agent with a higher reliability could be thought of as having a higher *capability*, as described in the HASSLE algorithm [Bakker and Schmidhuber, 2004], to successfully complete the current necessary sub-task. Whether or not a sub-agent predicts a high reward, if the sub-agent is reliable in its predictions, the prediction will remain relatively constant, i.e. it will have a low variance. Therefore by keeping track of the variance of every prediction of every sub-agent, it is possible to tell whether or not the prediction is reliable or not, using the variance. In other fields, variance and standard deviation are used in a similar way to determine reliability. In Econometrics the standard deviation of a market performance is used to determine the risk of investing in that market. This method of determining reliability therefore seems feasible.

The above method seems to be reasonable for most cases, but there may be some cases where a sub-agent with a really high variance is still the best sub-agent to obey for a given situation. This may come about in a situation where, for a given sub-problem, rewards are very erratic, but nevertheless very high. In a case like this the above method would not work well since, although a sub-agent describing this sub-problem will predict a very high reward, it will not be trusted, as it has a high variance and is labelled as being "unreliable" by the main agent. Therefore, some care needs to be taken in choosing the correct sub-agent to obey in different situations. A method of choosing between sub-agents may need to be tailor-made for a given problem, and there may be trade-offs involved in choosing between high predicted rewards and reliability of sub-agents. This lead to a investigation into a slightly different approach at choosing between sub-agents, which involved using the weighted average of predicted rewards, according to their reliability, as discussed in section 6.5.2.

6.3 Applying SVC to the Complex Gridworld Navigation Problem

For the complex gridworld navigation problem, the sub-problems can be identified as eating, drinking, repairing shelter, resting and avoiding the hazard. Each of these sub-problems can be described by a limited set of state variables. Eating can be fully described by x, y and nourishment. Drinking can be fully described by x, y and hydration. Repairing shelter can by fully described by x, y, carrying wood, and shelter condition. Resting can be fully described by x, y,

shelter condition and health. Avoiding the hazard can be described by just x and y. Hence, the maximum number of states required to define any sub-problem is 4.

Therefore, for the complex gridworld navigation problem, k = 4 and n = 7. The number of sub-agents is given by $\begin{pmatrix} n \\ k \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \end{pmatrix} = 35$.

All sub-agents are in charge of a combination of 4 different state variables. Some sub-agents will have a completely meaningless combination of states associated with them, for example an agent that is in charge of x, carrying wood, nourishment and hydration is not equipped to solve any of the 4 identified sub-problems, and would therefore never be of much use, but this is not a problem if sub-agents are intelligently chosen.

Every sub-agent is in charge of a predicted reward table, or Q-table, for every permutation of the 4 state variables, for each of the 5 low level actions. Each sub-agent is also in charge of an eligibility trace, and $Sarsa(\lambda)$ is used as the reinforcement learning algorithm. Initial reward predicted are again set optimistically to 6400, the maximum possible reward.

The variance of each reward prediction for taking any action in any state, was kept track of for every sub-agent. An extra table, the same size as the Q-table, was kept for every sub-agent which held the corresponding variance for every action for every state. Variance was also initialised optimistically, that is, all variances were initialised to zero, assuming that all initial predictions are reliable. The variance at each step was updated as follows:

$$\sigma(s,a) \leftarrow (1-\alpha)\sigma(s,a) + \alpha(Q(s,a) - (r + \gamma Q(s',a')))^2$$

Q, s, a, s', a', r, α, γ have the same meaning as detailed in section 4.1.

To choose between sub-agents, for the current state, for each possible action in turn, the variance of each of the 35 sub-agents is checked, and the predicted reward is recorded for the sub-agent with the lowest variance. Out of each of the 5 recorded predicted rewards, the highest is chosen, and this is taken as the chosen action.

In the same way as for the flat reinforcement learning implementation, to ensure sufficient and efficient exploration, the agent initially takes only random actions, and gradually increases the amount of exploitation by starting to take the chosen actions.

Values in the tables are also initialised optimistically, to 6400, the highest possible reward, to help with efficient exploration. Although this does improve exploration efficiency, it does not to the same level as it does with flat reinforcement learning. In flat reinforcement learning, with optimistic initialisation of tables, forces all possible states to be experienced at least once. In SVC, because state variables are broken up into different sets, all possible states are not forced,

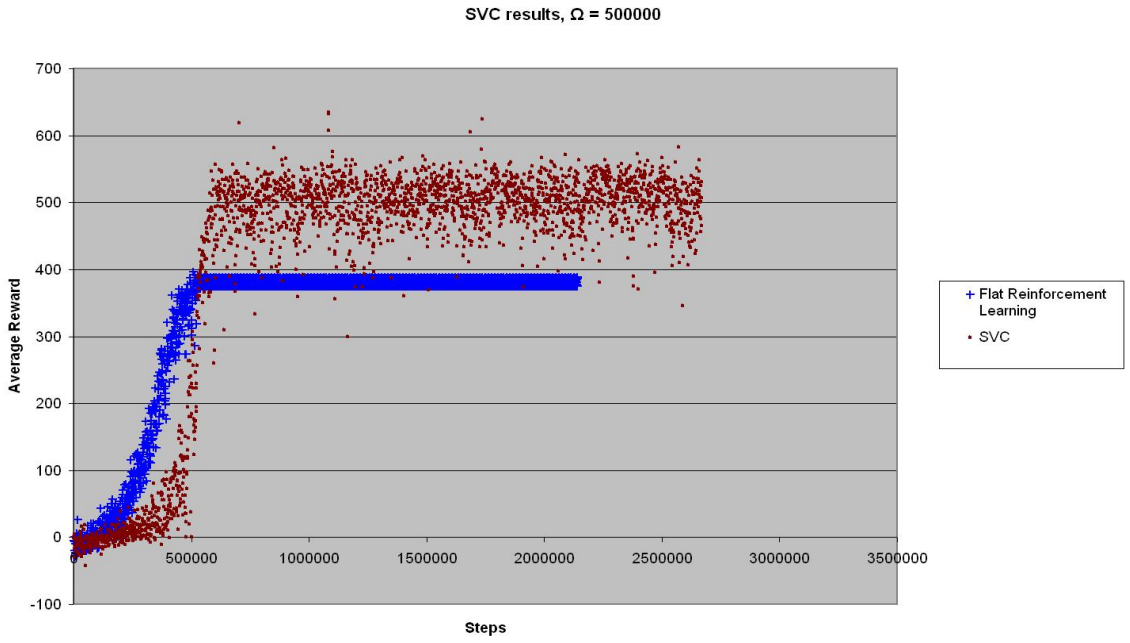

Figure 6.1: State Variable Combination Approach, $\Omega = 500\ 000$

but rather just all combinations of each group of 4 state variables forced. Optimistic initialisation of tables is therefore not as advantageous in this situation as for flat reinforcement learning, but nevertheless still implemented.

6.4 Initial Results

Three different trials were run to test SVC, with Ω values of $500\ 000$, $1\ 000\ 000$ and $2\ 000\ 000$. Rewards are averaged over every 1000 steps and plotted against their flat reinforcement learning counterparts.

Figure 6.1 shows the trial with $\Omega = 500\ 000$ and figure 6.2 shows the trial with $\Omega = 2\ 000\ 000$. Both trials show that SVC learns slower at first, but as it reaches the Ω threshold, learning happens very rapidly, when the curve makes an almost vertical climb. This is due to the fact that random actions, although necessary for exploration, might confuse the agent as to which sub-problem is the best to solve at the current point in time. As the Ω threshold is reached, and random actions stopped, the agent is able to put together all previous collected knowledge learned together to solve the problem as a whole.

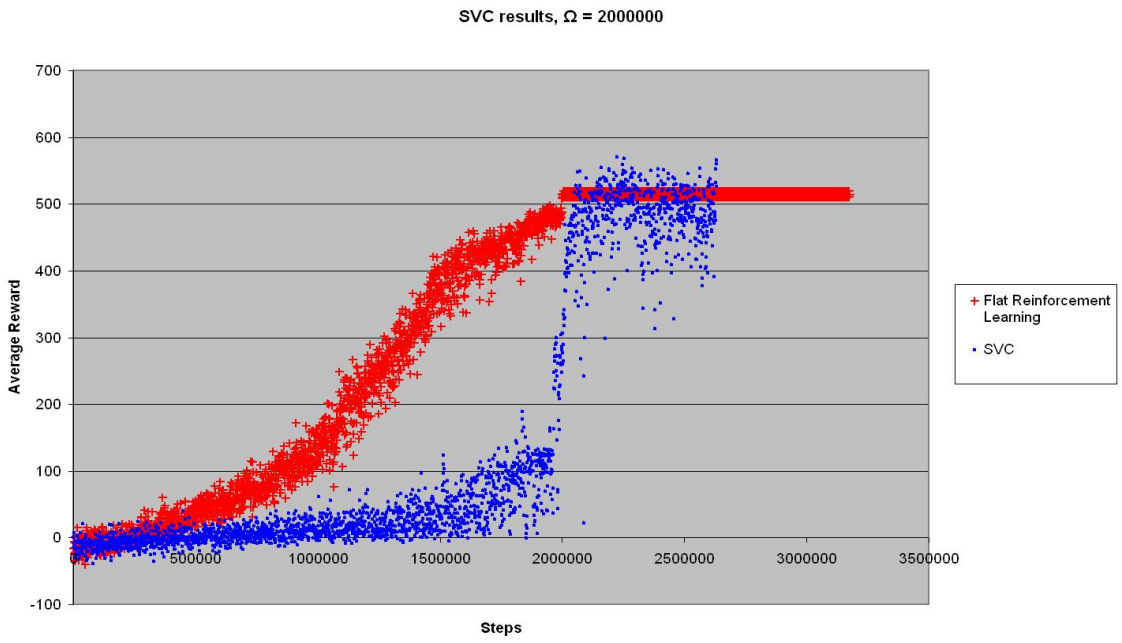

Figure 6.2: State Variable Combination Approach, $\Omega = 2\ 000\ 000$

For both trials the curves for SVC seem to level off, after the Ω value, to a much more erratic state than for the flat reinforcement learning. This is an interesting characteristic of this method, and will be dealt with later in this chapter.

For the trial with $\Omega = 2\ 000\ 000$ in figure 6.2, the final solution reached by the agent seems to be as good as that of the flat reinforcement learning agent, although it is difficult to tell because of the erratic nature of the final solution. For the trial with $\Omega = 500\ 000$ in figure 6.1, the final solution of the SVC method, although erratic, is clearly better than that obtained by the flat reinforcement learning agent. This shows that SVC is capable of finding an optimal solution to the gridworld navigation problem with less exploration necessary than that of flat reinforcement learning, and therefore has the ability to learn more quickly. SVC therefore outperforms flat reinforcement learning.

However, the final trial run on of SVC, with $\Omega = 1\ 000\ 000$, figure 6.3, displayed some rather unexpected results. The curve followed a similar trend to its predecessors up until the point where it reached the Ω value, and levelled off at an average reward of 0. The trial was run again, with little change in the results. This behaviour suggests that SVC is not guaranteed to converge to an optimal solution with insufficient exploration.

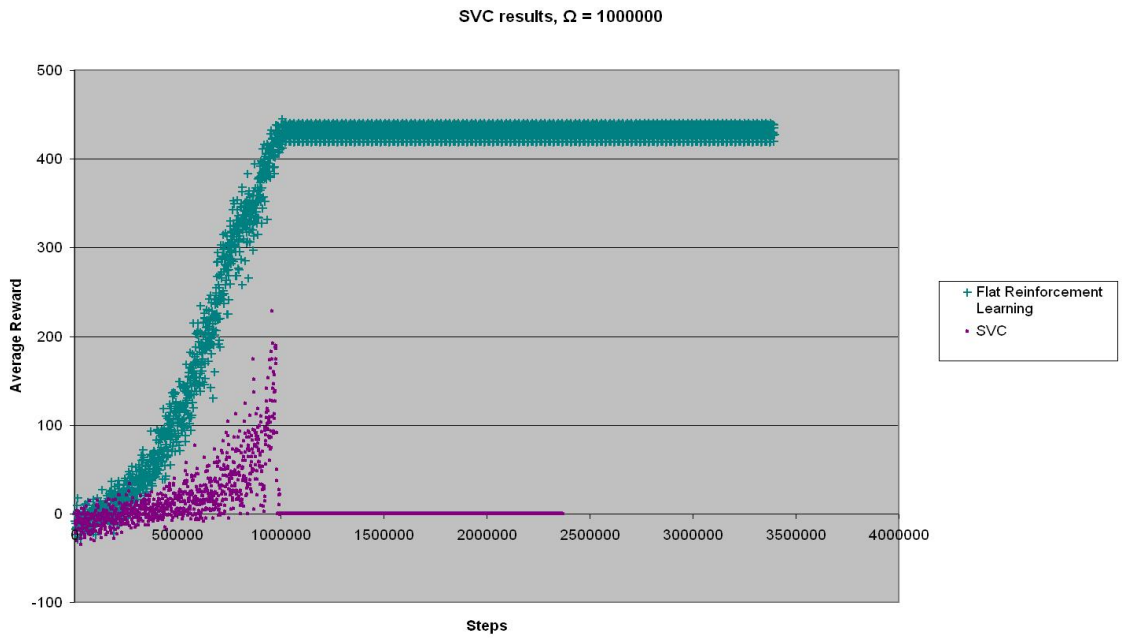

Figure 6.3: State Variable Combination Approach, $\Omega = 1\ 000\ 000$

6.5 Attempts at Improving SVC

Further investigation into the reason for the sudden drop of the average reward to 0 revealed that the agent was getting stuck in a loop of uncertainty of which sub-problem to try solve for a given state. For example, it would decide that for a certain state, the best sub-problem to solve was that of getting food. Then in moving towards food, it would find itself in a state where getting drink would seem like the best sub-problem to attempt, and so the loop continued. Sub-problems were conflicting to such an extent, that they were causing the learner to get totally caught up between which sub-problem to solve, and causing it to fail altogether.

Some possible solutions to this problem are suggested as follows:

1. Never exploit fully, rather always keep a small percentage exploration to stop the agent from getting stuck in a loop

2. Have an alternative, more efficient method of choosing between sub-agents

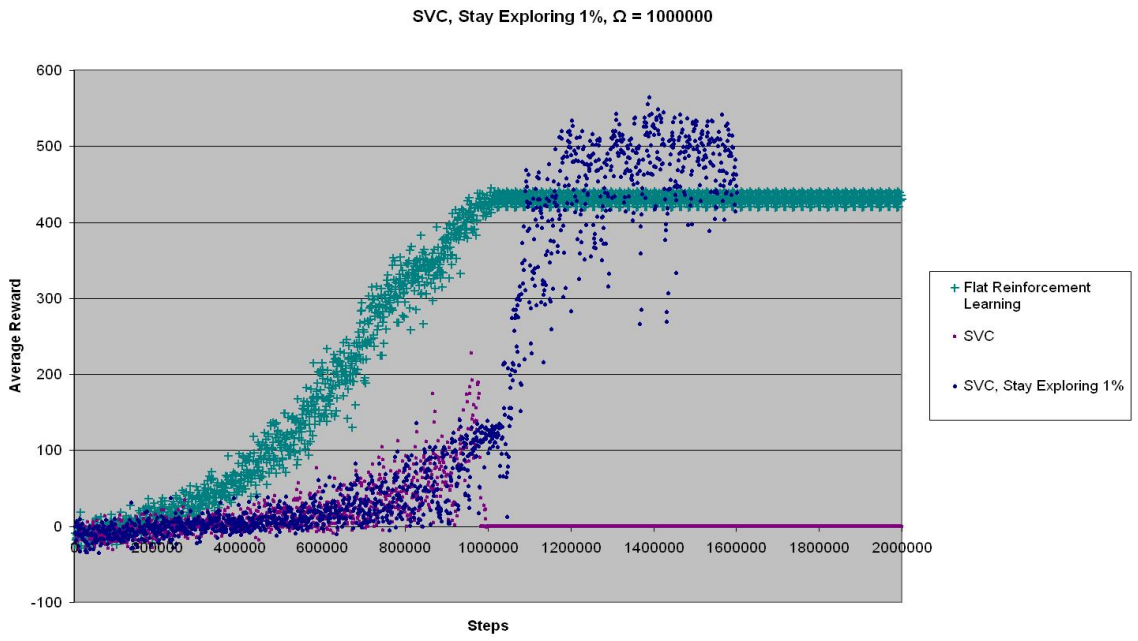

Figure 6.4: State Variable Combination Approach, Stay Exploring 1%, $\Omega = 1\ 000\ 000$

6.5.1 Keeping a Small Amount of Exploration

The trial with $\Omega = 1\ 000\ 000$ was rerun, this time, exploration was again linearly decreased from full exploration until it reached the Ω threshold, but never decreased it below 1%, keeping on average 1 random action in every 100 performed. It can be seen (figure 6.4) that this time, the agent did not lose track of the optimal solution, and performed well. The forced extra exploration after the Ω threshold prevented the agent from getting stuck in a loop of confusion between sub-problems. It can be seen that there are sudden drops in the curve, which may indicate that at times, the agent almost got stuck in a loop, but was then kept on track by an exploratory action. There is also an upward trend after the Ω value, which shows that the agent continued learning during this period because of the 1% exploration.

6.5.2 An Alternative Method For Choosing Between Sub-Agents

The previous method suggested for choosing between sub-agents involved the following steps: for each action choose the predicted reward with the lowest variance out of all sub-agents, and then out of all the lowest variance predictions, choose the action with the highest predicted

reward. This method is reasonable for most cases but has a few downfalls, which is a factor that may well be contributing to the erratic nature of the previous results, and perhaps also to the methods failure for the trial with Ω = $1\ 000\ 000$. The main reason for the downfall of the this method is due to the following: if a sub-agent constantly predicts a low reward, and hence has a low variance i.e. high reliability, it will be chosen instead of sub-agents with a higher variance i.e. lower reliability, even those with a much higher predicted reward. This is because the method takes preference to higher reliability, over higher predicted reward.

An alternative method is therefore suggested for choosing between sub-agents, which instead of taking preference to either reliability or to predicted rewards, attempts to balance the two. It takes a normalised average of each of the predicted rewards for the different sub-agents, weighted in proportion to the reliability of the predictions, i.e. the variance. Mathematically, for each action, j:

$$Weighted\ Prediction_j = \left[\frac{\frac{q_{1j}}{\sigma_{1j}} + \frac{q_{2j}}{\sigma_{2j}} + \ldots + \frac{q_{nj}}{\sigma_{nj}}}{\frac{1}{\sigma_{1j}} + \frac{1}{\sigma_{2j}} + \ldots + \frac{1}{\sigma_{nj}}} \right]$$

Where q is the predicted reward for each sub-agent, σ is the variance and n is the number of sub-agents.

This alternative method of choosing between sub-agents was implemented and yielded the results in figure 6.5. The curve shows that with this method of choosing between sub-agents, learning is much less erratic than with the previous method, however learning occurs a lot slower than with the previous method. Unlike with the previous method, learning starts off quickly, even quicker than for flat reinforcement learning. The curve then has a very rounded tip, where it peaks at a far from optimal solution. This is a rather puzzling result, since the algorithm that gave preference to reliability over predicted reward learnt a lot quicker.

This is due to the fact that for the above method, the moment the agent learns a good action for a given situation, it's prediction for taking this action will increase, but the variance will also increase, hence decreasing the overall predicted reward for taking this action. Therefore we effectively throw away anything new that we have just learnt. In time, after a sub-agent starts correctly predicting a high reward, the variance will eventually decrease, allowing the overall predicted reward to increase, but unfortunately this is a slow process, and this slows down learning.

To solve this problem, an attempt was made to favour predicted rewards over predicted variance, by taking $q \times |q|$, (effectively q^2, but keeping the + or - sign), instead of q. The weighted prediction formula then comes out as follows:

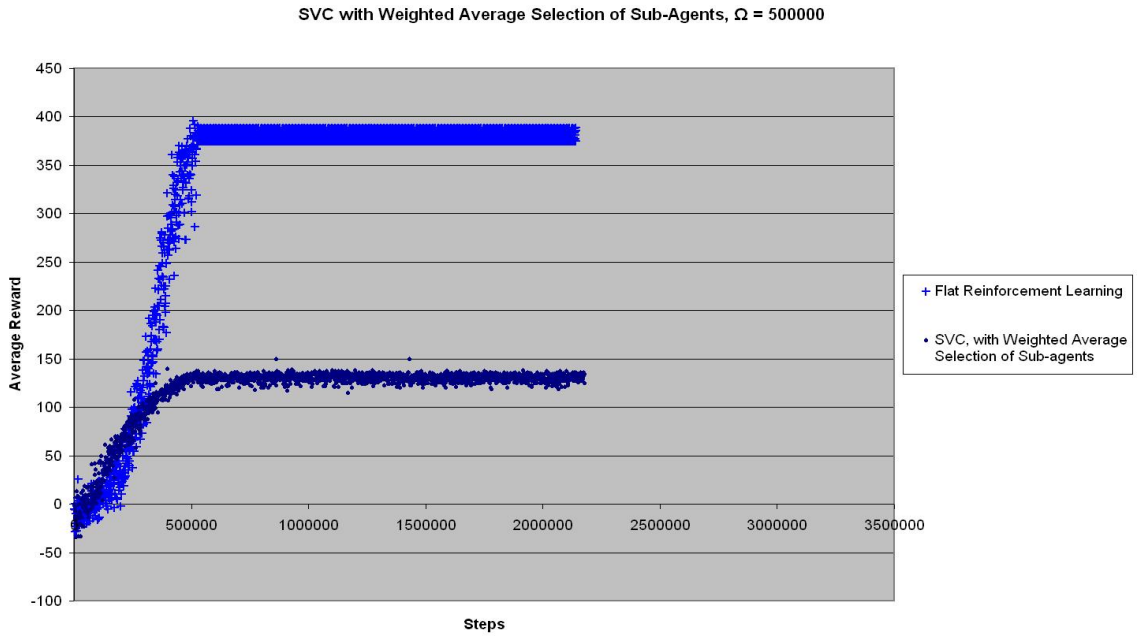

Figure 6.5: State Variable Combination Approach with Weighted Average Selection of Sub-Agents, $\Omega = 500\ 000$

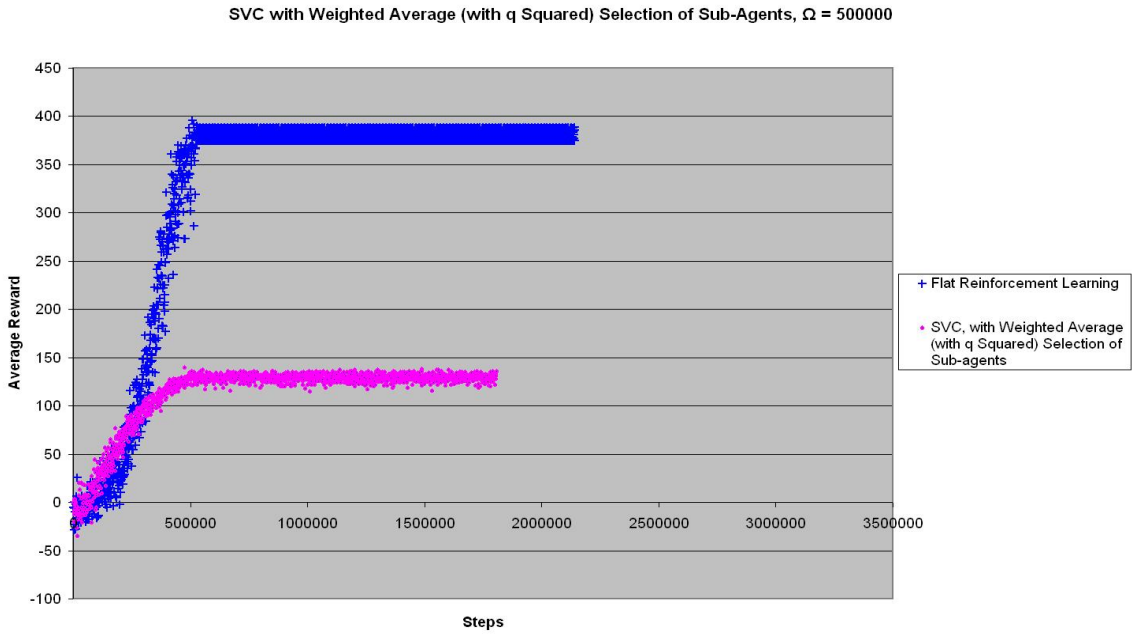

Figure 6.6: State Variable Combination Approach with Weighted Average (with Q squared) Selection of Sub-Agents, $\Omega = 500\ 000$

$$Weighted\ Prediction_j = \left[\frac{\frac{q_{1j} \times |q_{1j}|}{\sigma_{1j}} + \frac{q_{2j} \times |q_{2j}|}{\sigma_{2j}} + \ldots + \frac{q_{nj} \times |q_{nj}|}{\sigma_{nj}}}{\frac{1}{\sigma_{1j}} + \frac{1}{\sigma_{2j}} + \ldots + \frac{1}{\sigma_{nj}}} \right]$$

This method was implemented, and yielded the results as seen in figure 6.6. The results followed a very similar trend to the previous implementation, and the attempt to favour rewards over variance did not solve the problem of slow learning.

6.6 Discussion of SVC

SVC outperformed flat reinforcement learning in most trials. This is due to its efficient approach to intelligently distributing the state space between sub-agents. Specific sub-agents automatically emerge as being more equipped to solve individual sub-problems, which incorporates the favourable attribute of automatic discovery of sub-goals into the approach. This minimises designer intervention and enables easy adaptability of the method to different problems.

SVC however did not settle to a stable solution, and for some cases got stuck in a loop when confused between which sub-problem to tackle. This instability is due to the super-agent choosing an action from the incorrect sub-agent at certain times. Since sub-agents are chosen based on the how low the variance is in their predictions, the variance predictions may not always correlate with reliability. A sub-agent may have a sudden correct increase in the prediction of a reward, which will cause the variance to increase and label the sub-agent as unreliable, even if it isn't. Also, some correct predictions of sub-agents, although reliable, may have fluctuations in values. This will also label them as unreliable, even when they aren't. Possible solutions to this might be to take into account more long term fluctuations in variance.

Another possible reason for the instability in the method may be because, as the variance predictions are updated, they are overshooting the correct predictions. This may be preventing variance from converging to a relatively constant value, constantly changing the predicted reliability of sub-agents. A possible solution to this would be to decrease the α value, especially after Ω is reached, to allow more gradual updates of the variance tables, as discussed in section 6.7.

Overall, slight changes in SVC resulted in quite drastic changes in the behaviour of the learner. An alternative method of choosing between sub-agents increased the stability of the algorithm, but slowed down learning time.

6.7 Possible Extensions to SVC

The main challenge of SVC seemed to be how to decide on which sub-agent to obey at any time. Extracting useful information from the sub-agents was hard at times. Some possible extensions to the algorithm could be to research some alternative methods of choosing between sub-agents.

SVC also did not settle down to a stable solution, and this may have been due to variance values not converging to relatively constant values, and also may be due to variance not correlating with the reliability of sub-agents. Decreasing the α value, especially after the Ω threshold would be an interesting investigation into the effect of rate of variance update on the stability of SVC. An alternative method for calculating variance which takes into account a more long term variance would be interesting. The effect of adjusting other parameters such as γ and λ, and also initialising q-tables and variance tables differently would also be interesting.

SVC has a built in method of identifying sub-goals automatically, and should be easily transferred to an altogether different problem, given that the problem consists of multiple conflicting sub-problems and a set of state variables. It would therefore be a logical extension to this project to apply the method to another problem.

It would be interesting to explicitly identify sub-goals for SVC, and explicitly assign sub-agents capable of solving the given sub-problems, and see how well this approach performs. This will identify the overhead of incorporating automatic discovery of sub-goals into SVC. If performance is very negatively affected by automatic discovery of sub-goals, perhaps the effort of designer intervention is worth the increase in overall performance. Otherwise, if there is little change in performance, then the incorporation of automatic discovery of sub-goals is a very useful aspect of SVC.

It would also be interesting to extend the gridworld navigation to a larger gridworld, with more levels for each of the state variables. This will have a very negative impact on the performance on flat reinforcement learning implementation, because of the curse of dimensionality, but should not have as much of a negative impact on the performance of SVC, due to its approach to dealing with the curse of dimensionality.

Another possible extension to this project would be to apply a different algorithm to the complex gridworld navigation problem, such as MAXQ value function approximation from [Dietterich, 1999] in conjunction with some form of automatic discovery of sub-goals, such as methods like using diverse density from [McGovern and Barto, 2001] or learned policies from [Goel and Huber, 2003] to identify sub-goals.

6.8 Conclusion

SVC performed well as a hierarchical reinforcement learning algorithm, dealing with the curse of dimensionality effectively. However, the algorithm did not settle on a stable solution and in some cases failed to find an optimal solution. By forcing a small amount of exploration after the Ω value was reached, the problem of not finding an optimal solution was solved, however the final solution was still erratic. This is a result of reliable sub-agents being hard to identify. If sub-agents could correctly be identified as reliable, SVC would be a promising method.

Chapter 7

Conclusion

Reinforcement learning suffers from the curse of dimensionality, and increasing the complexity of a problem can cause learning times to slow significantly, because the number of state-action pairs to be learnt gets too large. Hierarchical reinforcement learning deals with the curse of dimensionality by breaking the main problem up into a hierarchical structure of sub-problems. In SVC, for example, as described in this thesis, each sub-problem has a limited state space associated with it, so if sub-agents in charge of these smaller state spaces can learn how to solve the sub-problems, and a super-agent can decide which sub-problem should be solved at any time, learning times will decrease.

In this thesis, a complex gridworld navigation problem was constructed, and sub-problems were identified within the main problem. A flat reinforcement learning approach was implemented to solve the problem, and with sufficient exploration, a satisfactory solution was found. Two methods of hierarchical reinforcement learning were implemented to solve the complex gridworld navigation problem.

One method is an adaption of Feudal reinforcement learning from [Dayan and Hinton, 1993]. Dayan and Hinton's method was not easily adaptable to the complex gridworld navigation problem, as high level actions were hard to define.

The next method of hierarchical reinforcement learning is SVC, a novel method which breaks up the state space according to different combinations of state variables. SVC worked well, and outperformed flat reinforcement learning in most trials. It also had a built in ability to automatically discover sub-goals, which is a favourable ability for a hierarchical reinforcement learning algorithm. Choosing which sub-agent to obey for a given situation proved quite difficult, and the resulting learner did not settle to a stable solution, sometimes not finding a satisfactory solution at all. Keeping a small amount of exploration in the algorithm solved the problem of

not finding a satisfactory solution, but the final solution was still erratic. The reason for the erratic behaviour is due to the incorrect sub-agent being chosen as the most reliable by the super-agent. Further attempts to increase stability, by choosing actions based on a combined prediction obtained from the predicted variance and from the predicted reward, resulted in slower learning. There therefore seems to be a trade off between stability and performance of the algorithm and small changes in the initial conditions result in drastic changes in the final performance of the agent.

Applying a method of hierarchical reinforcement learning proved to be a challenging task, and at times seemed to be more of an art than a science. Methods of hierarchical reinforcement learning which successfully solve a problem may not always be adaptable to problems of a different nature, and methods may need to be tailor-made, which is a time consuming task.

Possible extensions to this project would be to investigate different methods of choosing between sub-agents within SVC, and also experimenting with the initial conditions of tables and parameters. Increasing the state space of the complex gridworld navigation problem by increasing the size of the gridworld and making the granularity of state variables finer, should not have such negative effects on SVC, as it should not significantly suffer from the curse of dimensionality, and investigating this would be another possible extension. Explicitly assigning sub-agents to combinations of state variables which are known to describe given sub-problems would be an interesting investigation into the overhead of incorporating automatic discovery of sub-goals into SVC. Also, it would be interesting to test the portability of the SVC approach by testing its performance on a completely different problem. Another possible extension would be to solve the complex gridworld navigation problem with a different approach such as that of MAXQ value function decomposition from [Dietterich, 1999], in conjunction with techniques of automatically discovering sub-goals, by using diverse density as described in [McGovern and Barto, 2001], or by using learned policies as described in [Goal and Huber, 2003].

Hierarchical reinforcement learning has shown potential to overcome the curse of dimensionality if applied correctly. This has improved the outlook of reinforcement learning as a machine learning algorithm for the future. If the curse of dimensionality within reinforcement learning could be completely overcome, reinforcement learning could be implemented to teach machines how to solve complex real world problems.

References

Asadi, M and Huber, M (2004). *State Space Reduction for Hierarchical Reinforcement Learning*. In Proceedings of the 17th International FLAIRS Conference, pp. 509 - 514, Miami Beach, FL. 2004 AAAI

Bakker, B and Schmidhuber, J (2004). *Hierarchical Reinforcement Learning Based on Subgoal Discovery and Subpolicy Specialization.* In F. Groen, N. Amato, A. Bonarini, E. Yoshida, and B. Krse (Eds.), Proceedings of the 8-th Conference on Intelligent Autonomous Systems, IAS-8, Amsterdam, The Netherlands, p. 438-445.

Barto, A and Mahadevan, S (2003). *Recent advances in hierarchical reinforcement learning*. DiscreteEvent Systems journal, 13:41-77, 2003.

Borga, M (1993). *Hierarchical Reinforcement Learning*. S. Gielen and B. Kappen, ICANN'93, Springer-Verlag, Amsterdam, p. 513-525

P. Dayan and G. E. Hinton (1993). *Feudal reinforcement learning.* In Advances in Neural Information Processing Systems, volume 5, pages 271–278, San Mateo, CA, 1993. Morgan Kaufmann. T.G.

Dietterich, TG (1999). *Hierarchical Reinforcement Learning with the MAXQ Value Function Decomposition.* In Fifteenth International Conference on Machine Learning, 13:227-303, 1999.

Goel, S and Huber, M (2003). *Subgoal Discovery for Hierarchical Reinforcement Learning Using Learned Policies*, In Proceedings of the 16th International FLAIRS Conference, pp. 346-350, St. Augustine, FL. 2003 AAAI

Kaelbling LP, Littman ML and Moore AW, (1996). *Reinforcement Learning: A Survey*. Journal of Artificial Intelligence Research, 4:237-285, 1996.

McGovern, A and Barto, A (2001). *Automatic discovery of subgoals in reinforcement learning using diverse density*. Proc. 18th International Conf. on Machine Learning, 361-368, 2001.

Pfeiffer, M (2004). *Reinforcement Learning of Strategies for Settlers of Catan*. Proceedings of the International Conference on Computer Games: Artificial Intelligence, Design and Education, Reading, UK., p. 384-388, November 2004.

Sutton, RS and Barto, AG (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

Tesauro, GJ (1995). *Temporal Difference Learning and TD-Gammon*. Communications of the ACM, 38(3) p. 58-68, 1995.