ON-LINE Q-LEARNER USING MOVING PROTOTYPES

by

Miguel Ángel Soto Santibáñez

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

The UNIVERSITY OF ARIZONA

2004

ACKNOWLEDGMENTS

I would like to thank Dr. Michael Marefat and everybody in the ISL lab for all your help and feedback. Also, thanks to the Framsticks team, especially to Maciej Komosinski, for all your help. You have developed a wonderful simulator.

Dr. Cellier, thank you very much for all your help. In spite of your busy schedule I always felt welcomed. Thank you very much.

TABLE OF CONTENTS

TABLE OF CONTENTS - *Continued*

LIST OF FIGURES

LIST OF FIGURES – *Continued*

LIST OF TABLES

ABSTRACT

One of the most important breakthroughs in reinforcement learning has been the development of an off-policy control algorithm known as **Q-learning**. Unfortunately, in spite of its advantages, this method is only practical on a small number of problems. One reason is that a large number of training iterations is required to find a semi-optimal policy in even modest-sized problems. The other reason is that, often, the memory resources required by this method become too large.

At the heart of the Q-learning method is a function, which is called the Q-function. Modeling this function is what takes most of the memory resources used by this method. Several methods have been devised to tackle the Q-learning's shortcomings, with relatively good success. However, even the most promising methods do a poor job at distributing the memory resources available to model the Q-function, which, in turn, limits the number of problems that can be solved by Q-learning. A new method called Moving Prototypes is proposed to alleviate this problem.

Chapter 1

MOTIVATION

Reinforcement learning, also known as RL, is a method that tackles the problem of learning control strategies for autonomous agents. This method assumes that training information is available in the form of a real-valued reward signal given for each state-action transition. The goal of the agent is to learn an action policy that maximizes the total reward it will receive from any starting state [18].

Very often, reinforcement learning fits a problem setting known as a **Markov decision process** [6]. In this type of problem setting**,** the outcome of applying any action to any state depends only on this action and state and not on preceding actions or states. Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems.

Dynamic programming [5] is another method that has, often, been used to solve Markov decision processes (also known as MDPs). The main difference is that, historically, the dynamic programming approaches assume that the **reward** and **state transition functions** are known, while RL does not.

Not requiring a state transition function is a very useful characteristic. For instance, in many practical problems, such as robot control, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state. For instance, Tom Mitchell [18] illustrates this problem by asking us to imagine the difficulty in describing the state transition function for a robot arm shoveling dirt when the resulting state includes the positions of the dirt particles.

One of the most important breakthroughs in reinforcement learning has been the development of an off-policy control algorithm known as **Q-learning** [24]. By off-policy we mean that, in this method, the policy being learned is independent from the policy used to explore the state space. This characteristic is very important because it has greatly simplified the analysis of the algorithm and has allowed proving that, under certain conditions, the Q-learning algorithm converges to an optimal policy in both deterministic and nondeterministic MDPs.

Unfortunately, Q-learning is only practical on a small number of problems. There are two main reasons for this:

a) Q-learning can require many thousands of training iterations to converge in even modest-sized problems [18]. In other words, this method can be a "slow learner".

b) Very often, the memory resources required by this method become too large [3].

In most problems, it looks like the only way to learn anything at all is to generalize from previously experienced states to ones that have never been seen (i.e. to use **generalization**).

Tile coding and **radial basis functions** are examples of methods that have been used for generalization [1]. However, Richard Sutton and Andrew Barto [3] argue that, even after making use of these methods, the model becomes impractical for solving problems with more than a few dimensions. The problem with these methods is that their memory requirements increase exponentially as we add more dimensions to the state space even if the complexity of the problem stays the same. This problem is often called **curse of dimensionality** [5]. Furthermore, Richard Sutton and Andrew Barto mention that there are a number of tricks that can reduce this growth, such as **hashing**, but even these become impractical for solving problems with more than a few tens of dimensions.

Summing up, if we want to be able to use Q-learning to solve a larger number of problems, we require a method that is unaffected by dimensionality. We need a method that is limited only by, and scales well with, the complexity of what it approximates.

Methods such as **Kanerva Coding** [13] and others, using **regression trees** ([20], [23], [11], and [9]) tackle this problem. However, in spite of their important contributions, these methods do a poor job on distributing the available memory resources, limiting the number of problems for which Q-learning can be used (this will be described in more detail in the next chapter).

LSPI [16] is other method that has been used to alleviate Q-learning's memory resources problem. This method, typically, needs very little memory. Unfortunately, in order to make use of the LSPI method, the user needs to specify a set of functions, called **basis functions**. The problem is that these functions are, at best, difficult to find. Furthermore, the usability of these basis functions seems to be very sensitive to the section of the state space of the problem that is being analyzed by this method. In other words, a given set of basis functions may allow LSPI to solve a certain problem, but not a very similar problem.

Another method used with Q-learning is **Artificial Neural Networks** (ANNs). This method does not, normally, have very high memory requirements. However, it has been known to require a large number of learning experiences in order to generate reasonably good policies ([20], [11], and [12]). This slow convergence also limits the number of problems for which Q-learning can be used because, in a sense, ANNs make the agent a "slow learner".

In a nutshell, these are some of the most important shortcomings found on current Q-learning methods:

- Methods, such as Tiling, require memory resources that grow exponentially with the number of dimensions of the problem.

- Methods, such as Kanerva coding, are not affected by the curse of dimensionality but do a poor job at distributing the memory resources, limiting the number of problems for which Q-learning can be used.

- Methods, such as LSPI, require very little memory resources. However they also require information which, in many cases, is difficult or impossible to find.

- Methods, such as ANN, need to process a very large number of learning experiences in order to find a good, if not optimal, policy.

We believe that addressing these shortcomings will allow Q-learning to solve a much larger number of problems.

We propose a new method called **moving prototypes** which we believe tackles these shortcomings. This method is described in detail in Chapter 3. Chapter 4 describes the use of this new method to solve three different problems. Furthermore, Chapter 4 describes how other methods solve the same three problems and compares the results obtained by all the different methods.

The next chapter (i.e. Chapter 2), provides a more in depth description of the type of problems normally solved by Q-learning. Furthermore, it describes some of the methods that have been proposed to try to solve the most critical shortcomings associated with Q-learning.

Chapter 2

PREVIOUS WORK

As mentioned before, very often, reinforcement learning fits a problem setting known as a Markov decision process (MDP). An MDP is a decision process in which the outcome of applying any action to any state depends only on this action and state and not on preceding actions or states. Markov decision processes cover a wide range of problems including many robot control, factory automation, and scheduling problems [18].

Many methods have been developed to solve MDPs, such as **Value Iteration**, **Policy Iteration**, **Modified Policy Iteration** and **Linear Programming** (see the Vocabulary section for more details on these methods). The problem with these methods is that, typically, they assume that we have a model of the environment. However, uncertainty is common place in the real world. Sensors, themselves, are limited and/or inaccurate, measurements cannot be taken at all times and many systems are unpredictable. In other words, the transition function is not available [15].

Fortunately, there is a method called Reinforcement Learning (RL) which does not require this model of the environment in order to solve MDPs. All that this model requires is to be able to observe the current state and sample the unknown transition and reward functions through interaction with the environment.

More information about this important learning method is provided on the following section.

2.1 Reinforcement Learning

Reinforcement Learning (RL) has, recently, received significant attention [3].

Some people feel that it is better to think of RL as a class of problems, rather than a class of methods [15]. They mention that RL is the class of problem where a complete, interactive, goal-seeking agent learns to act in its environment by trial-and-error.

There are some issues associated with RL problems such as **Delay Rewards**, **Credit assignment problem** and **Exploration and Exploitation**.

Delay Rewards:

In some problems, reward is not received immediately but after much iteration with the environment has taken place. A typical example of this is the problem of learning to play chess. In this problem, the agent is only rewarded at the very end of the game.

Credit Assignment Problem:

Let us assume that at some iteration with the environment, the agent receives a big amount of reward (or punishment). The problem is: How do we find out which iteration or what part of the system was responsible for this **good finding** (or **bad finding**). Without knowing this, it may not be possible to assign credit for this finding appropriately and, therefore, it may not be possible to find the optimal policy (which is the RL agent's ultimate goal).

Exploration and Exploitation:

Typically, an RL agent is not given any application specific knowledge in advance. Therefore, it has to learn by exploration. After some exploration, the agent gains some knowledge, which can be exploited to gain more and more reward. For instance, the agent may have found out that a certain zone of the state space provides larger than normal rewards. However, this does not mean that this zone is the best one in the state space. There may be another zone in the state space, which may have not yet been explored and which provides even larger rewards. In other words, **exploration** and **exploitation** "pull" toward different directions. Therefore, the issue is how to balance the trade-off between exploration and exploitation.

An RL agent can solve a problem using a **Model-Based Approach** or a **Model-Free approach**.

Model-Based Approach:

In this case, the agent learns a model of the environment and then uses this model to derive an optimal policy [15]. This approach is also called Indirect RL [3]. Natasha Balac, Daniel M. Gaines and Doug Fisher provide and example of this approach [2]. In this paper, the agent, first, tries to learn models to predict the effects of its navigation actions under various terrain conditions. For instance, it tries to find out the effect of turning right on a wet surface, etc. Then, these models are fed to a planner which is able to derive an optimal policy.

Model-Free approach:

In this case, the agent learns a policy without learning a model. The agent is able to do this by creating a model of the expected reward that will be obtained from any given position in the state space of the problem. This estimate is commonly known as the optimal **value function** of the MDP. Once this optimal value function is obtained, it is possible to figure out the optimal policy without a model of the environment [15]. This approach is also called Direct RL [3].

Both, Model-Based and Model-Free approaches have advantages and disadvantages. Model-Based methods often make fuller use of a limited amount of experience and thus achieve a better policy with fewer environmental interactions. On the other hand, Model-Free methods are much simpler and are not affected by biases in the design of the model [3].

Since we want to be able to solve problems for which we have little or no domain specific knowledge, we decided to focus on the methods based on the Model-Free approach.

There are two main classes of Model-Free approaches: **Monte-Carlo** and **Temporal Difference**.

Monte-Carlo (MC) Methods:

This type of method estimates the value function of the MDP by sampling total rewards over long runs starting from particular state-action pairs. With enough data, this method leads to a very good approximation of the optimal value function. By changing the initial state-action pair or by using a stochastic policy, the agent is able to keep exploring during learning. This method does not learn after each **step** (i.e. each interaction with the environment). This method learns after each **episode** (i.e. a group of steps).

Temporal Difference (TD) methods:

This method estimates the value function by using sample rewards and previous estimates. The TD methods, as opposed to the MC methods, are fully **incremental** and learn on a step-by-step basis. Furthermore, there is a type of algorithms called **TD(λ)**, which are able to add some MC flavor to the TD algorithm. This property is very important because it allows the propagation of recent information through the most recent path in the state space using the so called eligibility traces. The use of Eligibility traces often implies faster learning.

Both MC and TD methods are able to learn the value function of one policy, while following another. For example, it is possible to learn the optimal value function while following an exploratory policy. This is what is called **Off-policy learning**. On the other hand, when the two policies match, we talk about **On-policy learning**.

2.2 Q-learning

One of the most important breakthroughs in reinforcement learning has been the development of an off-policy TD control algorithm known as **Q-learning** [24].

The fact that this is an off-policy method has greatly simplified the analysis of the algorithm and has allowed proving that, under certain conditions, the Q-learning algorithm converges to an optimal policy in both deterministic and nondeterministic MDPs [18].

In a nutshell, Q-learning is a method for learning to estimate the long-term expected reward for any given state-action pair [20]. The set of all these long-term expected rewards is what is called the **Q-function**.

The ultimate goal of a Q-learning agent is to come up with a **control policy**. This policy consists of a group of state-action pairs. There is a pair for each possible state. The action associated with each state represents the “best” action that can be taken from that state. By “best” action we mean the action that is expected to provide the largest reward in the long run.

Once, Q-learning generates a Q-function, it is very simple to come up with its associated control policy. All that needs to be done is to analyze each state in the Q-function and select the action that maximizes the expected reward.

Since Q-learning does not require a model of the environment, it can be used for on-line learning. **On-line learning** occurs when the agent learns by moving about the real environment and observing the results.

The Q-learning algorithm is shown in procedural form in the table below:

```
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode)
    Initialize $s$
    Repeat (for each step of the episode)
        Choose $a$ from $s$ using an exploratory policy
        Take action $a$, observe $r, s'$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$
```

TABLE 2.1. The Q-learning algorithm in procedural form.

To illustrate this algorithm let us use it to solve a simple problem. Assume we have an agent, whose goal it is to drive a vehicle from point A to point B in the least amount of time. The area where the agent is allowed to drive is surrounded by water, except at point B, which is a bridge. The agent is allowed to interact with the environment by moving North, South, East or West at any given position. An *episode* (i.e., a group of iterations with the environment) will end, if the agent drives the vehicle into water or arrives at point B. At this time the vehicle will be moved back to its initial position (i.e., point A) and a new episode will begin. The agent will receive a reward of -8 if the vehicle is driven into the water and a reward of 8 if the vehicle is driven to point B. Figure 2.1 shows the environment, the agent and the rewards received at each position.

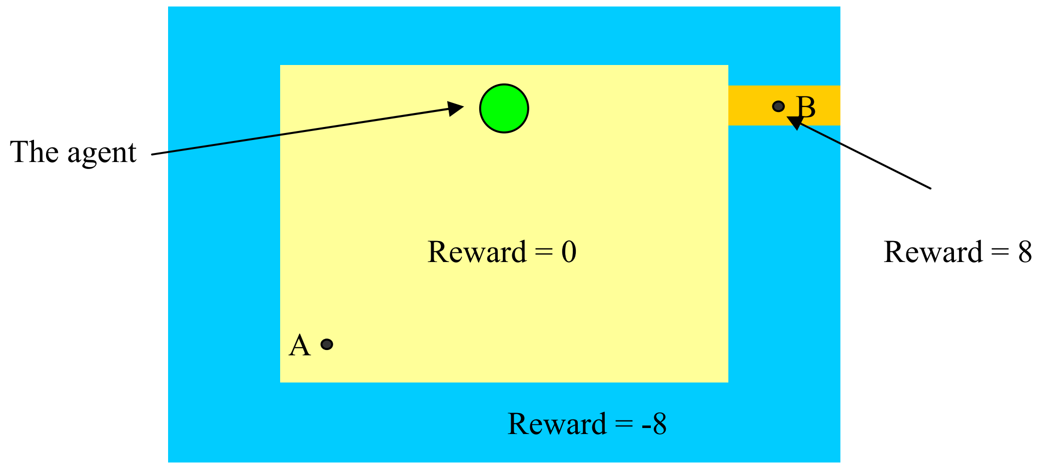


FIGURE 2.1. Sample Q-learning problem.

Let us assume that the agent selects a new action, *a*, (i.e., going North, South, East or West) once every second. This action, *a*, is capable of moving the vehicle one third of the horizontal distance between point A and point B. In other words, at the beginning of any iteration with the environment, the vehicle can be in any one of a finite number of states. Each possible state, *s*, is illustrated as a cell in a grid in Figure 2.2.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20

FIGURE 2.2. Each cell as a possible state.

Each one of these states has a value associated with it. This value is the immediate reward, *r*, the agent will receive as soon as it arrives at this state, *s*.

-8	-8	-8	-8	-8
-8	0	0	0	8
-8	0	0	0	-8
-8	-8	-8	-8	-8

FIGURE 2.3. The immediate reward associated with arriving at each state.

In this problem, we have 20 different states (i.e., 1, 2, 3... 20) and 4 different actions (North, South, West and East). If we use a table to represent the set of *Q(s, a)* we will end up with something like this:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
N																				
S																				
W																				
E																				

FIGURE 2.4. Representation of *Q(s, a)*.

The first step in the Q-learning algorithm is to initialize *Q(s, a)* arbitrarily. Let us initialize each possible *Q(s, a)* value to zero.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
N	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 2.5. Initialization of *Q(s, a)* table.

After initialization of *Q(s, a)*, the Q-learning algorithm initializes the state, *s*. In this problem, the initial state, *s*, is always set to the state 12, which corresponds to the position of point A.

After this the Q-learning algorithm selects an action, *a*. There are many methods that can be used to select this new action. A simple method, and the one we use in this example, is to select the new action, *a*, at random. This action will take the agent to a new state, *s*', and it will receive a reward, *r*. This new information is used to update *Q(s, a)* using the formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

The first thing this formula does is to estimate the new *Q(s, a)* value. The new estimated *Q(s, a)* value is given by:

$$r + \gamma \max_{a'} Q(s', a')$$

The parameter γ can assume any value between 0 and 1, including 0 and 1, and it is usually called the **discount rate**. This parameter determines the present value of future rewards. If γ is equal to 0, the agent becomes myopic and it is only concerned with maximizing immediate reward. On the other hand, as γ approaches 1, the agent takes future rewards more strongly into account, and the agent becomes more farsighted.

We may be tempted to update the *Q(s, a)* value by just using the formula:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$

However, there may arise a situation, in which it may be desirable to dampen the effect of a single learning experience on the *Q(s, a)* value. For instance, some learning experiences may have some noise in them, and it may preferable to slowly converge to the newly estimated *Q(s, a)* value, rather than just adopt this new value.

In order to accomplish this, Q-learning makes use of a parameter α that can have any value between 0 and 1, including 1, and it is sometimes called the **step-size**. This parameter determines how fast *Q(s, a)* will converge to the newly found *Q(s, a)* value. If α is set to 1 the update formula will simplify to:

$$Q(s,a) \leftarrow r + \gamma \max_{a'} Q(s',a')$$

On the other hand, if α is set to a value very close to 0, the *Q(s, a)* will be updated, but it will only change slightly in the direction of the newly estimated *Q(s, a)* value.

Once the *Q(s, a)* value is updated, the next and last step is to update the current state *s*. We do this by simply assigning to it the state encountered after performing action *a*. In other words:

$$s \leftarrow s'$$

This process is repeated until the episode ends, at which time the current state *s* is reset to the initial state (i.e., state 12).

Let us go through this algorithm using the problem we just set up. For simplicity, we shall set γ to 0.5 and α to 1. We assume that the agent begins interacting with the environment and comes up with an episode that contains the following state-action pairs:

{(12, E), (13, N), (8, W), (7, N)}

Figure 2.6 shows graphically the movement of the agent during this episode:

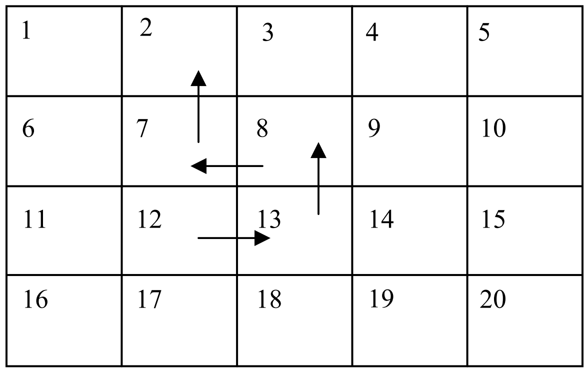


FIGURE 2.6. An episode.

Notice that the agent will receive a reward of zero at every new state except for the last one, in which it will receive a reward of -8.

The first step in the episode will update *Q(s₁₂, E)* in the following way:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

$$Q(s_{12},E) \leftarrow 0 + 1*[0 + 0.5*(0) - 0]$$

$$Q(s_{12},E) \leftarrow 0$$

Notice that this update did not really change the value of Q(s_{12}, *E*). This is because all the rewards found so far have been equal to zero, as all the *Q(s, a)* had been initialized to zero. The same with happen with the update of Q(s_{13}, *N*) and with Q(s_8, *W*).

However, the updating of Q(s_7, *N*) will look like this:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

$$Q(s_7, N) \leftarrow 0 + 1*[-8 + 0.5*(0) - 0]$$

$$Q(s_7, N) \leftarrow -8$$

Therefore, the *Q(s, a)* table will now look as follows:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
N	0	0	0	0	0	0	**-8**	0	0	0	0	0	**0**	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
W	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	0	0	0	0	0	0	0	0	0	**0**	0	0	0	0	0	0	0	0

FIGURE 2.7. *Q(s, a)* after one episode.

If we continue using this algorithm, we may end up with a *Q(s, a)* table that looks like this:

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
N	0	0	0	0	0	0	**-8**	**-8**	**-8**	0	0	**1**	**2**	**4**	0	0	0	0	0	0
S	0	0	0	0	0	0	**0.5**	**1**	**2**	0	0	**-8**	**-8**	**-8**	0	0	0	0	0	0
W	0	0	0	0	0	0	**-8**	**1**	**2**	0	0	**-8**	**0.5**	**1**	0	0	0	0	0	0
E	0	0	0	0	0	0	**2**	**4**	**8**	0	0	**1**	**2**	**-8**	0	0	0	0	0	0

FIGURE 2.8. *Q(s, a)* after many episodes.

Getting a policy out of the *Q(s, a)* table is very simple. All that needs to be done is to select the action, *a*, for every state, *s*, that has the largest *Q(s, a)* value. Figure 2.8 shows the largest *Q(s, a)* value for every state, *s*, using circles.

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
N	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	**-8**	**-8**	**-8**	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{1}$	$\textcircled{2}$	$\textcircled{4}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$	$\textcircled{0}$
S	0	0	0	0	0	0	**0.5**	**1**	**2**	0	0	**-8**	**-8**	**-8**	0	0	0	0	0	0
W	0	0	0	0	0	0	**-8**	**1**	**2**	0	0	**-8**	**0.5**	**1**	0	0	0	0	0	0
E	0	0	0	0	0	0	$\textcircled{2}$	$\textcircled{4}$	$\textcircled{8}$	0	0	**1**	**2**	**-8**	0	0	0	0	0	0

FIGURE 2.9. Highest *Q(s, a)* value for each state.

This yields the following policy:

$s_1 \rightarrow N$
$s_2 \rightarrow N$
$s_3 \rightarrow N$
$s_4 \rightarrow N$
$s_5 \rightarrow N$
$s_6 \rightarrow N$
$s_7 \rightarrow E$
$s_8 \rightarrow E$
$s_9 \rightarrow E$
$s_{10} \rightarrow N$
$s_{11} \rightarrow N$
$s_{12} \rightarrow N$
$s_{13} \rightarrow N$
$s_{14} \rightarrow N$
$s_{15} \rightarrow N$
$s_{16} \rightarrow N$
$s_{17} \rightarrow N$
$s_{18} \rightarrow N$
$s_{19} \rightarrow N$
$s_{20} \rightarrow N$

The figure below shows this policy graphically. Each arrow represents the action associated with each state:

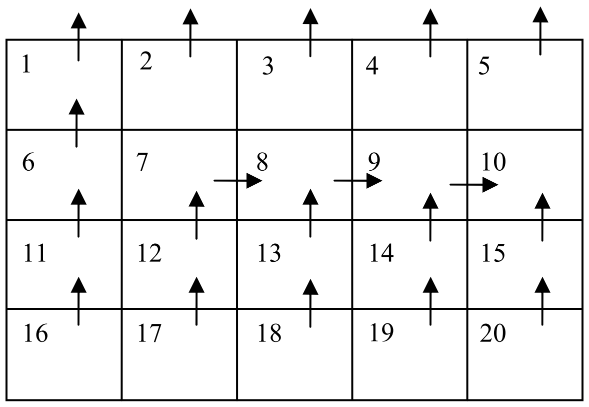


FIGURE 2.10. A Policy.

Notice that this is not unique. The following would also be appropriate:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
N	0	0	[✓ 0]	0	0	0	-8	-8	-8	[✓ 0]	[✓ 0]	1	2	[✓ 4]	0	[✓ 0]	[✓ 0]	[✓ 0]	[✓ 0]	[✓ 0]
S	[✓ 0]	0	0	0	[✓ 0]	0	0.5	1	2	0	0	-8	-8	-8	[✓ 0]	0	0	0	0	0
W	0	[✓ 0]	0	0	0	[✓ 0]	-8	1	2	0	0	-8	0.5	1	0	0	0	0	0	0
E	0	0	0	[✓ 0]	0	0	[✓ 2]	[✓ 4]	[✓ 8]	0	0	[✓ 1]	[✓ 2]	-8	0	0	0	0	0	0

FIGURE 2.11. Another appropriate highest *Q(s, a)* value for each state.

This would yield the following policy:

$s_1 \rightarrow S$
$s_2 \rightarrow W$
$s_3 \rightarrow N$
$s_4 \rightarrow E$
$s_5 \rightarrow S$
$s_6 \rightarrow W$
$s_7 \rightarrow E$
$s_8 \rightarrow E$
$s_9 \rightarrow E$
$s_{10} \rightarrow N$
$s_{11} \rightarrow N$
$s_{12} \rightarrow E$
$s_{13} \rightarrow E$
$s_{14} \rightarrow N$
$s_{15} \rightarrow S$
$s_{16} \rightarrow N$
$s_{17} \rightarrow N$
$s_{18} \rightarrow N$
$s_{19} \rightarrow N$
$s_{20} \rightarrow N$

The figure below shows this policy graphically:

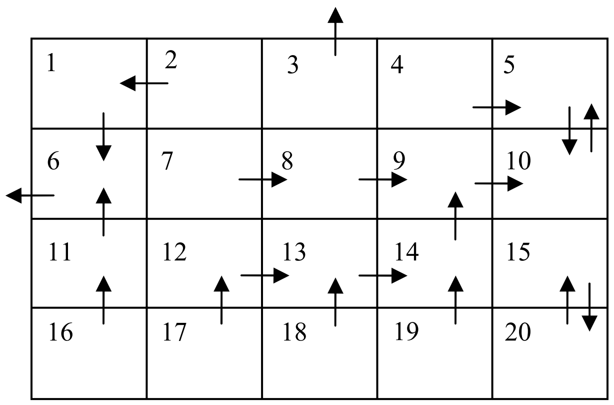


FIGURE 2.12. Another Policy.

Notice that, anyone of these two policies will accomplish the task at hand (i.e., to move the vehicle from point A to point B as fast as possible).

Unfortunately, the original Q-learning method (which is also called **tabular Q-learning** because it uses an explicit table to represent its Q-function), can only solve a small number of problems. The reason for this has to do, in part, with the fact that computers have limited memory resources, and in many problems these tables just become too large. Another reason for the limited use of the pure Q-learning method is that many problems have continuous variables and complex sensations, such as visual images. The only way to learn anything at all on these tasks is to **generalize** from previous experienced states to others that have never been seen [3].

The following section elaborates on the need of generalization and the use of function approximation.

2.3 Generalization and Function Approximation

There are many ways to represent the Q-function. The success or failure of a Q-learning agent can depend on what representation is used.

The naïve way is to use a table to store all the possible Q values (i.e. the *Q(s, a)* values for every (*s*, *a*) combination). The advantage on doing this is that this table can represent any function with discrete values (no matter how complex it is) and that convergence is guaranteed.

However, using a table has several disadvantages. The first disadvantage is that the agent will learn very slowly because it will need to visit each state many times in order for the Q-function to converge. Furthermore, using a table means that we will only be able to model functions with discrete values and not functions with continuous values. Finally, using a table to represent the Q-function may require huge amounts of memory resources.

For instance, think of a simple problem in which the state is represented by three signals, s_1, s_2 and s_3. Each signal has a discrete value in the range [0,100). In each state we are allowed to choose an action a, which can have any of 100 different values. This means that, in this problem, the Q-function will have four dimensions (i.e., s_1, s_2, s_3 and a), where each dimension can have any of 100 different values. Therefore, we will need to have a table that has 100,000,000 cells. This is a relatively small problem but we would already have to deal with a huge table.

Another way to implement the Q-function is to use a **function approximator**. The intuition is that "similar" entries (s, a) should have "similar" values $Q(s, a)$. Therefore, depending on the definition on similarity and the generalization capabilities of the function approximator, we may end up with a representation that models the function quite well while using very little memory resources.

It is very common to implement Q-function approximators by keeping track of a relatively small number of (s, a, v) tuples, also called **prototypes**. If the Q-function approximator is asked to return the value associated with a given (s, a) pair, it first looks for it in the list of prototypes. If the list does not contain a prototype associated with this specific (s, a) pair, it then uses similar prototypes and generalization to come up with this new value.

The advantage of using function approximators is rapid adaptation and handling of continuous spaces. All this, while possibly using very little memory resources. However, the disadvantage is that convergence of the Q-function is no longer guaranteed. A discussion on the advantages and disadvantages of using function approximators is provided by Anton Schwartz and Sebastian Thrun [19].

There are a number of methods that use generalization to model the Q-function, such as **Tiling**, **Kanerva coding**, **ANN**, etc. The following sections mention some of the most important methods, and point out some of their advantages and disadvantages.

2.4 Tile coding and Radial Basis Functions

Tile coding is a way to represent the Q-function that is especially well suited for use on sequential digital computers and for efficient on-line learning.

In this method, the state space is partitioned. Each partition is called a **tiling**, and each element of the partition is called a **tile**. For example:

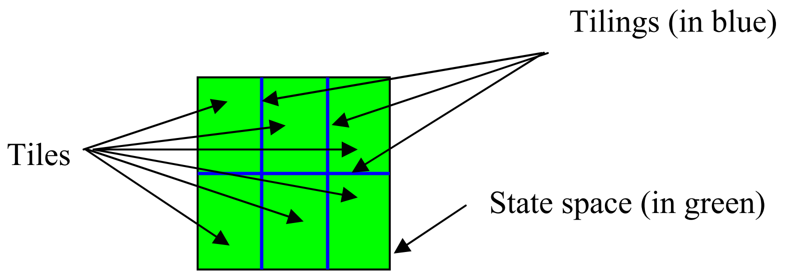


FIGURE 2.13. Tile coding example.

In Tile coding, each tile is now a single possible state. This has the advantage that the overall number of states that are present at one time can be controlled and that this number is independent of the state space of the problem.

If a grid like tiling is used, figuring out the current state becomes particularly easy. However, width and shape of the tiles should be chosen to match the generalization that one expects to be appropriate. The number of tilings should be chosen to influence the density of the tiles. However, one needs to be aware that the denser the tiling, the finer and more accurately the desired function can be approximated, but the greater the computational costs.

Tilings can be arbitrary and need not be uniform grids. For instance:

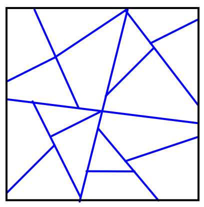


FIGURE 2.14. Arbitrary tiling example.

Not only can the tiles be strangely shaped, but they can be shaped and distributed to give particular kinds of generalization.

For instance, the tiling below will promote generalization along the vertical dimension and discrimination along the horizontal dimension, particularly on the left.

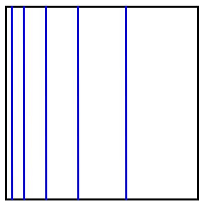


FIGURE 2.15. Tiling promoting generalization along a specific dimension.

The diagonal stripe tiling below will promote generalization along one diagonal:

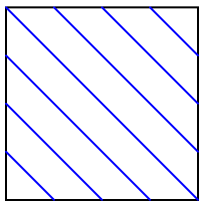


FIGURE 2.16. Tiling promoting generalization along one diagonal.

The method Radial basis functions (RBFs) is very similar to Tile coding. The main difference is that in RBFs, a point in the state space does not necessarily map to a single state. Remember that in the Tile coding case a point in the state space either maps to a state, 1, or it does not map to that state, 0. On the other hand, in the RBF case, a position in the state space can, partially, map to several states (i.e. anything in the range [0, 1]). Typically, this value in the range [0, 1] is given by a Gaussian function which is dependent on the distance between the position in the state space and a given state. The value associated with this position is generalized from all the states this position maps to, weighted by their associated value in the range [0, 1]. The weights are normalized before their use so that their sum adds up to 1.

Tile coding and RBFs are very practical if you have an idea of what the Q-function will look like and therefore know what is a good way of distributing discrete states in the state space.

However, in many problems, the Q-function is completely unknown. In these cases, it is only possible to make assumptions. This can be very costly memory-wise, because we may end up using lots of memory resources modeling very simple areas of the state space. For instance, the Q-function may be completely independent of a certain dimension in the state space and we may end up tiling along this dimension anyway. Furthermore, a lack of *a priori* knowledge on the Q-function may mean that some zones in the state space may end up with less number of states that necessary to properly model the function.

This lack of *a priori* knowledge about the Q-function not only affects the usability of function approximators such as Tile coding and Radial Basis Functions (RBF) but, as we will see later, it also affects the method LSPI.

2.5 Fuzzy function approximation

Julie Dickerson and Bart Kosko define a fuzzy system as a set of fuzzy rules that map inputs to outputs. In other words, a fuzzy system defines a function f: $X \rightarrow Y$. The rules are if-then rules of the form "if X is A, then Y is B." A and B are multi-valued or fuzzy sets that contain members to some degree. A is a subset of input space X. B is a subset of output space Y [10].

Fuzzy rules have a simple geometry in the input-output state $X \times Y$. They define fuzzy patches or subsets of the state space. Less certain rules are large patches. More precise rules are small patches. The rule "if X is A, then Y is B" defines the fuzzy patch or Cartesian product $A \times B$. In the precise limit the rule patch $A \times B$ is a point if A and B are binary spikes [10].

A given input may map to more than one patch. In these cases some type of generalization, such as getting an average, is done on the overlapping patches.

Unfortunately, all fuzzy systems $F : R^n \rightarrow R^p$ suffer from rule explosion in high dimensions. In other words, a fuzzy system F needs on the order of K^{n+p-1} rules (where each rule defines a patch in the state space) to represent the input to outputs function.

According to Sanya Mitaim and Bart Kosko rule explosion remains the chief limit to the fuzzy systems approach [17].

2.6 Hashing

Hashing is a method that can be used to reduce memory requirements. In a nutshell, hashing does a consistent pseudo-random collapsing of a large tiling into a much smaller set of tiles.

Hashing produces tiles, consisting of noncontiguous, disjoint regions randomly spread through the state-space, which still form an exhaustive tiling [3].

This method often reduces memory requirements by large factors with little loss of performance. However, even hashing becomes impractical after a few tens of dimensions [3].

Andrew Barto and Richard Sutton argue that if we want to be able to use Q-learning to solve a larger number of problems, we require a method to represent the Q-function that is unaffected by dimensionality. We need a method that is limited only by, and scales well with, the complexity of what it approximates.

The following section describes a method called Kanerva coding which tackles this problem.

2.7 Kanerva coding

As we mentioned before, it is very common for function approximators to store a relatively small set of (*s*, *a*, *v*) tuples, also called **prototypes**, in order to model the Q-function.

When the function approximator is asked to return the value associated with a given position, (*s*, *a*), in the state space, it first looks in the list of prototypes for one that has associated this (*s*, *a*) position. If it finds it, it just returns the value, *v*, associated with this prototype. If it does not find it, then the approximator uses generalization and a group of "nearby" prototypes to generate this value.

In Kanerva coding, "the storage locations and their addresses are given from the start, and only the contents of the locations are modifiable" [13]. In other words, the (*s*, *a*) part of the prototypes are chosen *a priori* and they never change. In other words, these are **static prototypes**. Only the value, *v*, associated with each prototype may be modified during learning.

Kanerva coding uses the concept of "Hamming distance" to determine if a prototype can be considered "nearby" or not. Under this concept, states (s, a) are considered similar if they agree on enough dimensions, even if they are totally different on others.

Kanerva coding has the advantage that the complexity of the functions that can be learned depends entirely on the number of prototypes, which bears no direct relationship to the dimensionality of the task. The number of prototypes can be larger or smaller than the number of dimensions. Only in the worst case must the number of prototypes be exponential in the number of dimension. In other words, dimensionality itself is no longer a problem. Complex functions are still a problem as they have to be. To handle more complex Q-functions, Kanerva coding simply needs more prototypes [3].

Unfortunately, as we shall see later in this chapter, the fact that Kanerva coding uses static prototypes causes poor allocations of memory resources. This shortcoming seriously limits the number of problems that can be solved using Q-learning.

2.8 LSPI

Michail G. Lagoudakis and Ronald Parr have proposed using a method called **LSPI** (Least-Squares Policy Iteration) to approximate the Q-function in cases where the memory requirement for the explicit table implementation becomes too large [16].

LSPI is an off-line, model-free method that combines least squares function approximation with policy iteration. In LSPI, learning experiences are gathered and then processed to come up with a policy (i.e., an agent using LSPI is a **batch learner**).

In LSPI, the Q-function is represented implicitly by a finite set of n parameters (w_1, w_2 … w_n). The Q-function can be determined on demand for any given state by using the formula:

$$Q(s,a) = \sum_{i=1}^{n} \varphi_i(s,a) w_i$$

Note: The set of φ_i ( ) functions are called **basis functions**

At each iteration, the LSPI method takes in a set of learning experiences and tries to fit this data to the *Q(s, a)* parametric equation (i.e., LSPI finds a set of values for the *w* parameters that minimize the square of the difference between the parametric equation and each one of these experiences).

The advantage of using LSPI is that it typically requires very little memory to represent the Q-function (i.e., we are assuming LSPI will use a small number of *w* parameters). However, since this method processes learning experiences in batch, it may need significant resources to store all these experiences while they are being processed.

Furthermore, this method requires the user to provide a set of “good” **basis functions**. The problem is that, often, it is necessary to have an idea of what the Q-function looks like, in order to come up with these “good” **basis functions**. Since, in many problems we have no *a priori* knowledge of what the Q-function will look like, this becomes a vicious circle. Furthermore, the “goodness” of these basis functions seems to be very sensitive to the section of the state space of the problem that is being analyzed by this method. In other words, a small change on the range of one of the dimensions of the state space can convert a very “good” set of **basis functions** into one that makes the method fail on solving the problem at hand. This was often observed in the experiments described in section 4.2.

2.9 Artificial Neural Networks

The method **Artificial Neural Networks** (also called ANNs) allows learning real-valued, discrete valued and vector valued functions.

Algorithms, such as BACKPROPAGATION, can be used to find a set of network parameters that allow the internal model to fit a given set of training examples (i.e. a set of input-output pairs). ANNs have the advantage that they are very robust to errors in the training data.

ANNs have been used as a function approximator in several studies. For instance, a program called TD-GAMMON used neural networks to play backgammon very successfully [21]. Furthermore, Zhan and Dietterich used ANNs to solve job-shop scheduling tasks [8]. Crites and Barto used an ANN to solve an elevator scheduling task [4]. Finally, Sebastian Thrun used a neural network based approach to Q-learning to learn basic control procedures for a mobile robot with sonar and camera sensors [22].

Q-learning agents using neural networks have found reasonably good policies while requiring relatively small memory resources. However, they often require excessively long training times ([20], [11], [12], and [9]). This is a major drawback in some problems, such as economic market places, in which agents must adapt quickly to an ever changing market place environment. In this type of problems learning speed is very important.

The goal of improving on the training time of neural networks while generating policies of at least equal quality is what has motivated use of regression trees to approximate the Q-function. This scheme is described in the following section.

2.10 Regression trees

Some research has been done on the use of regression trees as a function approximator of the Q-function ([20], [23], [11], and [9]).

By itself, the word “regression” means approximating a real-valued target function [18]. Therefore, it comes as no surprise that a regression tree is a data structure that is used to approximate a real-valued target function.

In general, a regression tree divides the state space of a function into areas that can themselves be represented by relatively “simple” functions. The target function may be itself very complex, but the functions at the leaves of this tree are relatively simple.

Another way to see this is that a regression tree models a target function as a group of smaller and possibly simpler functions (something like a piece-wise function).

These smaller functions at the leaf nodes of the regression tree are often modeled by using prototypes. As we mentioned before, a prototype is just an example of what the function is worth at certain position in the state space (i.e., a (*s*, *a*, *v*) tuple in the Q-function case). The methods using regression trees can use any type of generalization, such as interpolation, to come up with the value of the function for any position in the state space. All that it needs to do is to find the leaf node associated with the given position in the state space, and then use the prototypes associated with this node to generate the requested value.

For instance, assume we want to represent the following function using a regression tree:

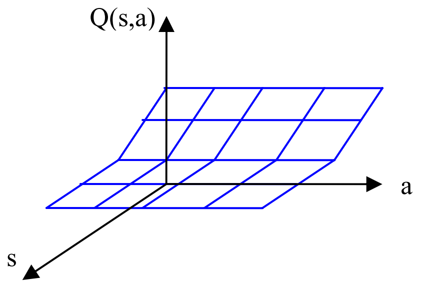


FIGURE 2.17. A function to be represented using a regression tree.

Notice that in this case, the state space would be something like this:

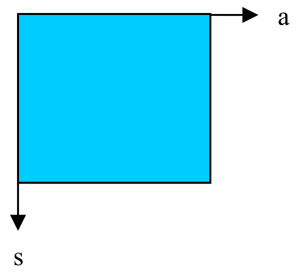


FIGURE 2.18. Visualization of the state space of a function.

Let’s suppose that all we know about this function is its value at the following 10 positions in the state space (i.e. we only have the following 10 prototypes):

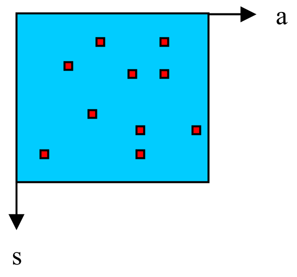


FIGURE 2.19. Ten prototypes in the state space of a function.

A regression tree breaks the state space in a way as to minimize the variance of the prototypes. Therefore, in this problem, the state space will be broken in two parts in the following way:

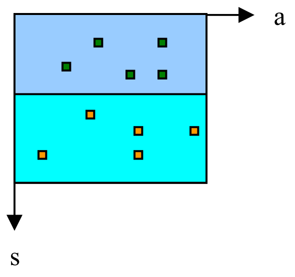


FIGURE 2.20. Best way to break the state space into two parts.

The regression tree will look like this:

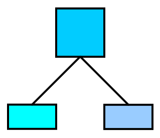


FIGURE 2.21. Regression tree showing the new two parts.

Each one of the two leaf nodes will have five prototypes associated with it.

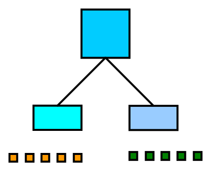


FIGURE 2.22. Visualization of the regression tree and its prototypes.

Notice that we could use interpolation and extrapolation to find out the value of the function at any position in the state space for which we do not have any prototype.

For instance, let's suppose that we want to know the value of the function at the following position in the state space:

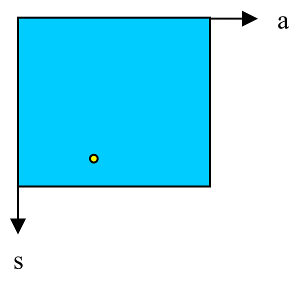


FIGURE 2.23. Position in the state space.

The first step would be to select the appropriate leaf node. In this case, this would be the leaf node to the left of the root node.

The next and final step would be to use the prototypes associated with this node to generate the requested value. To do this we may do something as simple as getting the average value associated with these prototypes or we could go as fancy as trying to fit these prototypes into a parametric equation and then using this function to find out the requested value.

In summary, a regression tree is used to represent a function as a set of smaller and simpler functions. Each leaf node in the regression tree represents one of these smaller and simpler functions. Each leaf has, associated with it, a set of prototypes (i.e., a set of (s, a, value) tuples). The prototypes associated with this leaf node model this smaller and simpler function by using some form of generalization (e.g. interpolation, extrapolation, etc).

Previous work:

As we mentioned before, some research has already been done on the use of regression trees as a function approximator of the Q-function ([20], [23], [11], and [9]).

The methods proposed by Manu Sridharan and Gerald Tesauro [20], William Uther [23], and Thomas Dietterich and Xin Wang [9] all process the learning experiences in batch. What this means is that the Q-function is updated using a set of learning experiences, as opposed to only using a single learning experience (see Appendix section A.4 for more details)

Unfortunately, any type of batch updating of the Q-function produces a couple of shortcomings:

a) Slow propagation of good findings:

The fact that this method processes learning examples in batches means that the Q-function will only be updated once after processing a whole set of learning experiences. On the other hand, other methods, such as the tabular Q-learning, will update it several times. To better visualize this, assume that the agent is learning to play soccer. In this scenario, the agent will only receive a reward after scoring a goal. In the case of the "batch" method, this reward will only propagate to the previous state after processing all the learning experiences. On the other hand, a method such as the tabular Q-learning may easily propagate this reward to several previous states because the Q-function is updated after processing each learning experience. This "slow propagation of good findings" can make the "batch" method a much slower learner because it may need to process many more learning experiences in order to learn a given task.

b) Large requirement of memory resources:

Since this method processes learning experiences in batches, it will need to store them in memory. This can be especially expensive, memory-wise, in problems in which the state space is very large.

The method proposed by Adele Howe and Larry Pyeatt [11] has the advantage that it updates the Q-function as soon as a new learning experience is available (i.e., it is an on-line learner, as opposed to a batch learner).

Unfortunately, the Howe-Pyeatt method does not attempt to do any pruning on the tree used to model the Q-function. This is a problem because this method has no way of reallocating memory resources around the state space in order to do a better job at modeling the function.

It is reasonable to expect that the Q-function will go through several transformations during the learning process. This implies that some zones of the state space, which at some point could be modeled by a few prototypes, may, at some other point in time, require a much larger number of prototypes. The opposite is also plausible. Therefore, is seems logical that in order to make good use of memory resources, the tree used to model the Q-function will constantly have to go through transformations. In some cases new nodes and new prototypes will need to be added to the tree. In some other cases, a section of the Q-function may become much more regular and therefore some prototypes may become "redundant". However, without doing some type of pruning on the tree and freeing some of these "redundant" prototypes, we may end up running out of memory resources. Without pruning we may get to a point in which we no longer have memory resources available to add new nodes and prototypes in zones of the state space where the Q-function has become more complex.

2.11 Summary

This chapter provided some background on the type of problems commonly tackled by the method **Q-learning**.

Furthermore, this chapter described and analyzed many of the methods used by Q-learning to model its **Q-function**. We discovered that, although these methods have some very advantageous properties, they also have some shortcomings.

For instance, we discovered that **tabular Q-learning** is not a very practical method because the table used to model its Q-function becomes too large even for relatively small problems.

Methods, such as **Tile Coding**, can help shrinking the memory required to store the Q-function. However, even this method becomes impractical when applied to problems with many dimensions in their state space. This problem is often called **curse of dimensionality**.

We also found out that methods, such as **Kanerva coding**, can be used to tackle the curse of dimensionality problem. However, even these methods often make bad use of the memory resources, restricting the number of problems for which they can provide a practical solution.

Furthermore, we discussed methods, such as **LSPI**, which require very little memory resources to model the Q-function. Unfortunately, we also discovered that this method requires the user to provide a set of “good” basis functions. Normally, in order to come up with this set of “good” basis functions, we need to have an idea of what the Q-function looks like. Unfortunately in many problems, the user does not have this *a priori* knowledge, which creates a vicious circle (i.e., we need the “good” basis functions to find the Q-function, and we need the Q-function to generate these “good” basis functions).

We also discussed the use of **Artificial Neural Networks** (ANNs) as a way to model the Q-function. We discovered that, although this method is very robust to errors in the sample data, this method converges very slowly, which in turn makes the agent a relatively slow learner.

Finally, we found out that this shortcoming on the method Artificial Neural Networks, served as inspiration for the development of new methods based on **regression trees**. These new methods, as Kanerva coding did, allowed creating a model of the Q-function that was independent of the dimensionality of the problem.

Unfortunately, most of the methods using a regression tree are batch learners. This property tends to make them “slow” learners. Furthermore, these methods become impractical in many problems in which the batch size has to be particularly large in order for the agent to learn.

We were able to find a method that used a type of regression tree to model the Q-function and was an on-line learner. However, as in the case of Kanerva coding, this method also made bad use of the available memory resources.

This “bad use of memory resources” found in Kanerva coding and in the regression tree approaches is mostly caused by their inability to properly allocate the memory resources used to model the Q-function. These methods may run into any of these two types of problems:

1) Certain zones of the Q-function may end up modeled by more prototypes than needed (for instance, in zones of the state space where the function is very simple). In other words, some of the defined prototypes may end up being “redundant”, which is a waste of memory resources.

2) Certain zones of the Q-function may end up being modeled by fewer prototypes than needed (for instance, in zones of the state space where the function is very complex).

It would be great if we could move these prototypes around and use the “redundant” prototypes in zones where the Q-function is more complex.

This idea is what inspired a new method that we call **Moving prototypes**. This method is described in detail in the following chapter.

Chapter 3

Method Description

3.1 Goal

We need to come up with a method to model Q-functions, which has the following properties:

1) Memory requirements should not explode exponentially with the dimensionality of the problem.

2) It should tackle the pitfalls caused by the usage of "static prototypes" (i.e. having redundant prototypes in some zones of the Q-function and not enough prototypes in others).

3) It should try to reduce the number of learning experiences required to generate an acceptable policy.

This method should do this without requiring *a priori* knowledge of the Q-function.

3.2 Dimensionality Problem

To tackle the dimensionality problem (i.e. memory requirements growing exponentially as we add more and more dimensions to the problem), I suggest the use of a fixed number of prototypes, just as Kanerva Coding does (for more details on Kanerva coding see section 2.6).

Let us denote this fixed number of prototypes with the letter *N*:

$$N \geq MAX\{ \mid Q(t_1) \mid, \mid Q(t_2) \mid, \mid Q(t_3) \mid, \ldots, \mid Q(t_n) \mid \}$$

Where $|Q(t_i)|$ is the number of prototypes used to describe the Q-function at time t_i and n is the total number of possible values for time t_i.

In other words, at any given time, this method will use *N* or fewer prototypes to model the Q-function.

The value *N* (i.e. the maximum number of prototypes) will depend on the memory resources available to the agent. This predetermined number of prototypes will be all the agent will have to describe the Q-function. Therefore a good use of these prototypes will be critical to the agent's chances to successfully finding an optimal or semi-optimal policy to the problem at hand.

3.3 Static Prototypes Problem

As we mentioned before, methods that use “static prototypes”, such as Kanerva Coding, tend to do a poor job on placing the available “prototypes” throughout the state-space of the problem. This causes two problems:

1) The Q-function is not described as accurately as it could be with the available prototypes.

2) Some of the prototypes often become “redundant”, therefore wasting memory resources.

If we were able to move these prototypes around on the state-space of the problem and place them where they appear to be most needed, we probably would do a better job, both at describing the Q-function accurately, and at avoiding wasting memory resources.

Therefore, I propose the use of a method in which the prototypes are no longer “static” but “dynamic”. I call this method **Moving Prototypes**. The details on this implementation can be found in the following sections.

3.4 Proposed Method Overview

For starters, this method makes use of a data structure called a regression tree to represent mutually exclusive zones in the state-space of the problem (more details on regression trees can be found in section 2.9).

For instance, the picture below shows a state-space containing 4 zones and its corresponding regression tree:

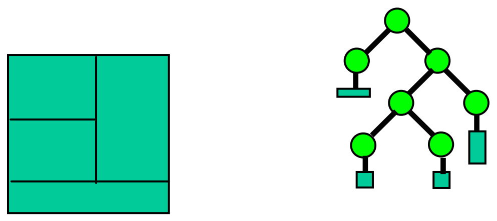


FIGURE 3.1. A partitioned state space and its representation as a regression tree.

Each one of these zones has a set of prototypes associated with it (see section 2.9 for more details on prototypes). Prototypes are represented by red squares in the picture below:

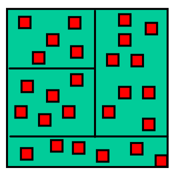


FIGURE 3.2. A set of prototypes in a partitioned state space.

These prototypes are used to describe the Q-function at any point inside the zone. The Q-function is just a concatenation of these functions (a kind of a piece-wise function).

In order to reduce the use of “redundant” prototypes (prototypes without which the Q-function could be described equally well), we decided to only have prototypes at each one of the zone’s corners.

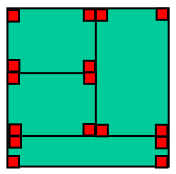


FIGURE 3.3. A prototype at each corner of each zone.

Furthermore, in the spirit of saving memory, we decided to share prototypes between contiguous zones, whenever possible:

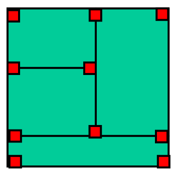


FIGURE 3.4. Sharing the same prototype if at the same corner.

The proposed method has mechanisms that allow for the rupture and merging of zones. The following picture shows the rupture of a zone:

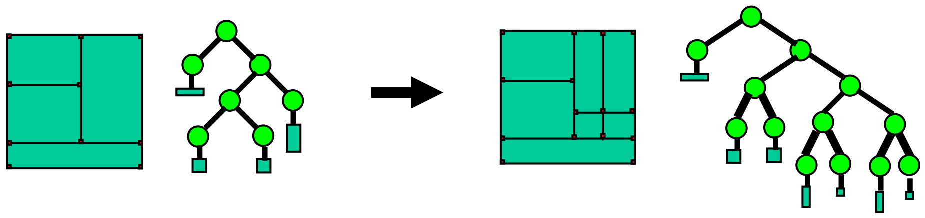


FIGURE 3.5. Zones and regression trees before and after a rupture.

Notice that the resulting regression tree is not unique. There may be more than one regression tree that would properly represent the zones on the state space, after a rupture. For instance, the following is also a valid regression tree:

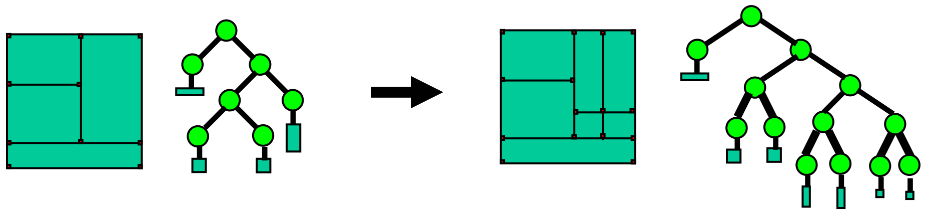


FIGURE 3.6. An alternative resulting regression tree.

Notice that the difference is the order in which rupture occurs at each dimension of the state space. In this method the dimensions in the state space are indexed. This index is what determines the order in which each dimension will be ruptured. For instance in the first case, the horizontal dimension was given the index 1 and the vertical dimension was given the index 2. In the second case, the indexes were reversed.

The following picture shows the merging of two zones:

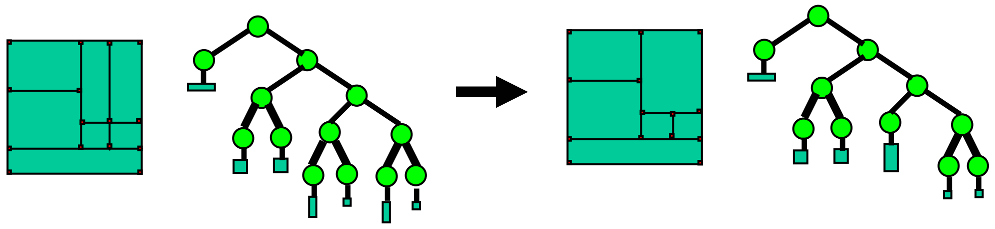


FIGURE 3.7. Zones and regression trees before and after a merging.

Notice that, after this merging, a prototype was liberated (note: other merging operations may liberate more prototypes).

This release of prototypes permits a better use of the available memory resources by “migrating” prototypes from zones where the function is very simple to zones where the function is more complex (and, therefore, more prototypes are required).

For instance, Figure 3.8 shows a 2-D Q-function in which the prototypes are concentrated in a zone of the state-space, in which the function is more complex. The upper part of the picture shows the prototypes, as small circles, and the “relief” of the Q-function using different colors (for instance the zone painted in pink defines the zone in which the Q-function has values in the range 30 to 39, and the zone painted in red has values in the range 40 to 49). The lower part of the picture shows the 9 leaf nodes of the regression tree:

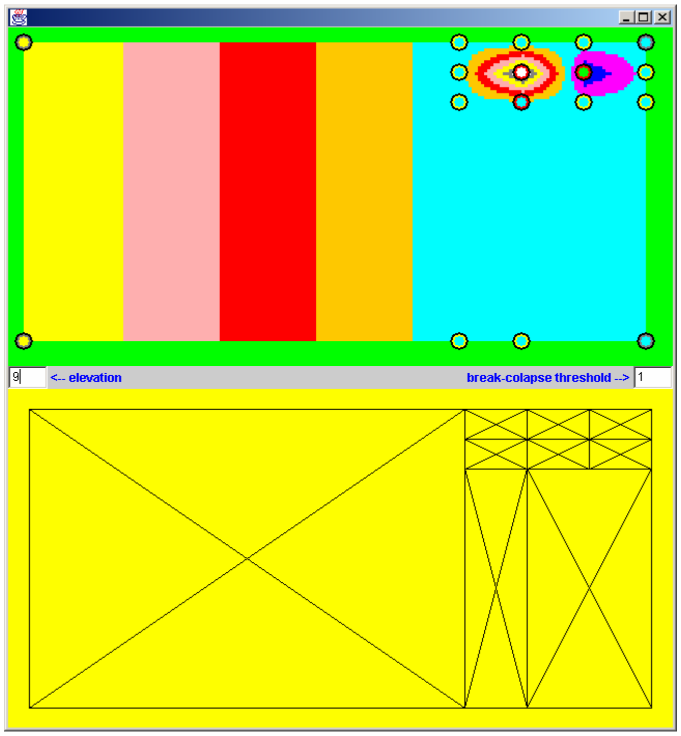


FIGURE 3.8. Q-function with simple and complex zones.

3.5 Merging and collapsing on the proposed regression tree

The following pictures illustrate the rupture and merging processes. In this example, we decided to initialize the Q-function to zero. Figure 3.9 shows this initial Q-function. The white color represents this value. At this time, since the Q-function has the value 0 throughout the whole state space, the regression tree modeling this function only has a single node. This leaf node has 4 prototypes, one at each corner of the node (denoted by the small circles). The *v* value associated with each one of these 4 prototypes has been set to 0.

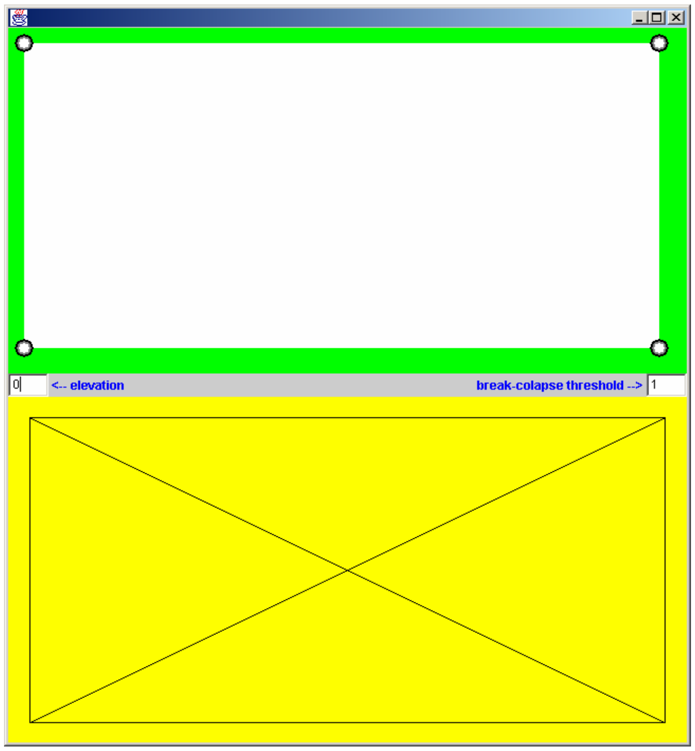
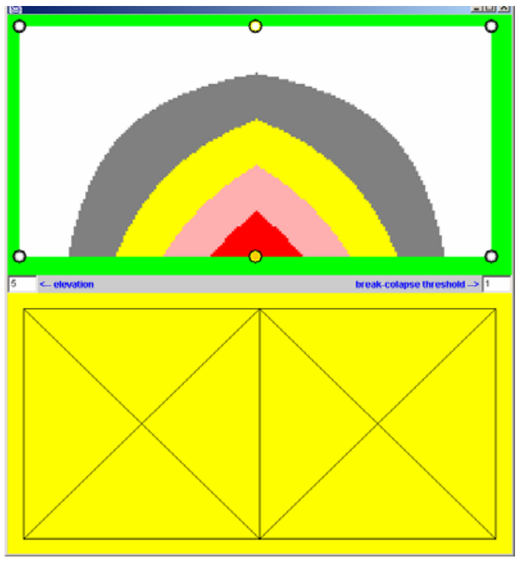


FIGURE 3.9. The initial Q-function.

After this, we decide to update the Q-function with the new value 4.9. The state for this new value is located at the lower-middle position of this 2D function. In Figure 3.10, the range 4 to 4.99 is represented by the red color.

Notice that, this time, the regression tree has two leaf nodes (represented by the two yellow squares at the bottom of the picture). Furthermore, we now have 6 prototypes (two of these prototypes are now shared by the two leaf nodes).

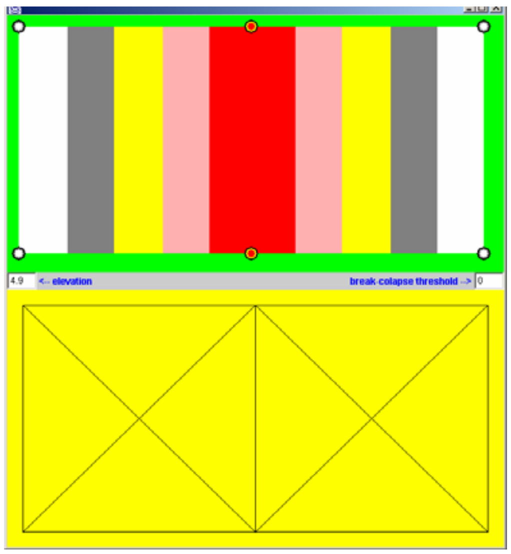


5 <-- elevation

break-colapse threshold --> 1

FIGURE 3.10. First update of the Q-function.

Now, we decide to update the Q-function again, this time, by setting the upper-middle position to the value 4.9.



4.9 <-- elevation

break-colapse threshold --> 0

FIGURE 3.11. Second update of the Q-function.

Notice that in this case, no node has to be ruptured. It is enough to update the *v* value of the prototype at the uppermost-middle position.

After this, we decide to update the state that corresponds to the upper-right position of the state space. This state is given the new value 9.8. Note that the range 9.0 to 9.99 is represented by the green color.

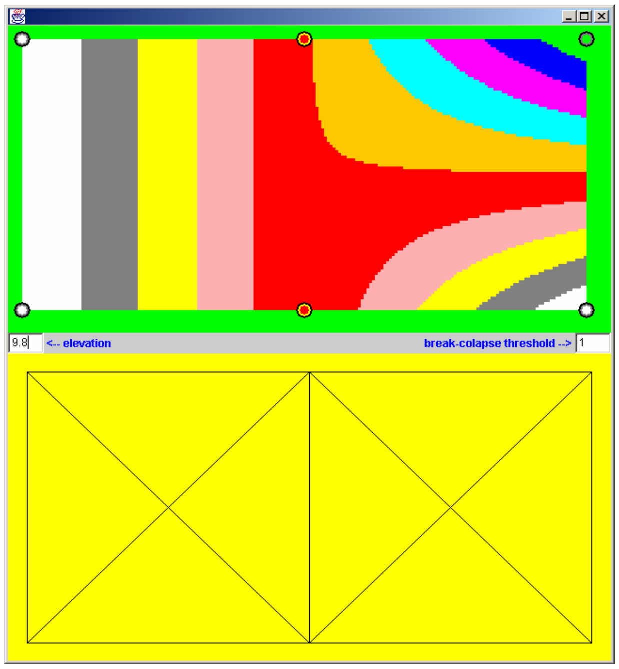
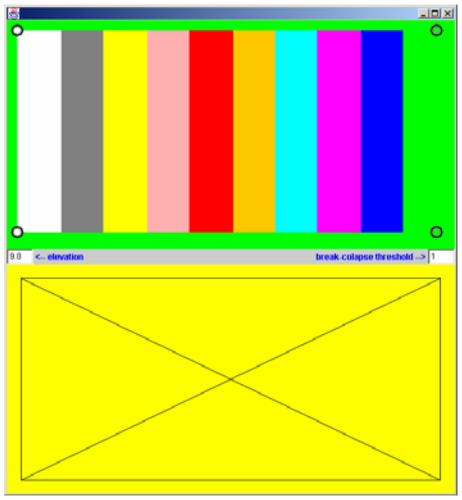


FIGURE 3.12. Third update of the Q-function.

Again, it was not necessary to rupture any of the zones in the regression tree. All we needed to do was to update the v value associated with the prototype.

Finally, we decide to update the state that corresponds to the lower-right position in the state space. The new value is 9.8.



9.8 <-- elevation

break.colapse threshold --> 1

FIGURE 3.13. Fourth update of the Q-function.

Notice that this last update produced a merging. The two prototypes with value 4.9 are no longer needed because they can be generated using generalization and the other 4 prototypes. Therefore, the two child nodes of the tree are collapsed into a single node and we release the two “redundant” prototypes.

These two freed prototypes can now be used in zones of the state-space where they are needed more.

3.6 Proposed Method Details

In order to have a sound regression tree, the following rules must be satisfied at any moment:

a) All the "areas" represented by the nodes in the regression tree are mutually exclusive.

b) The concatenation of all the "areas" represented by the leaf nodes produces an "area" equivalent to the state space.

c) The merging of any two sibling "areas" is equal to their parent's "area".

The following picture illustrates these 3 rules:

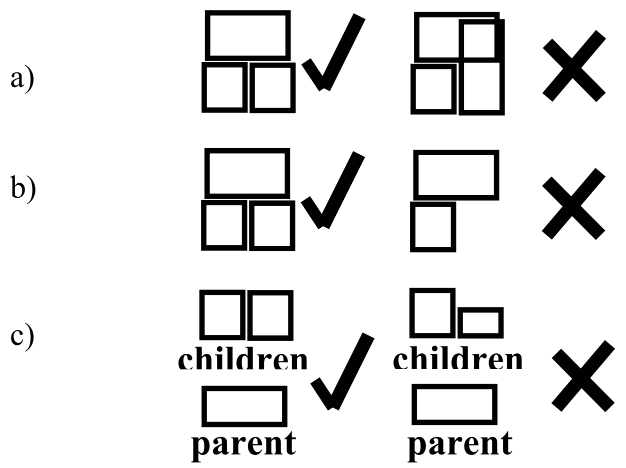


FIGURE 3.14. Rules for a sound regression tree.

This takes us to the sub-problem:

> "What policy should be applied at the time of merging two "areas", so that we end up with a sound tree?"

Merging is simple if the nodes that represent the two "areas" are siblings in the regression tree. For instance:

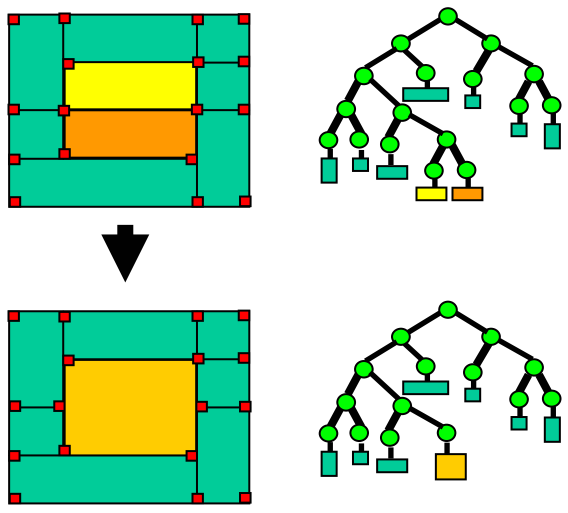


FIGURE 3.15. Easy type of merging.

However, things get much trickier when the nodes representing the two "areas" are not siblings in the binary tree. This is because, in some situations, the two "areas" cannot be merged. See Figure 3.16 for an example.

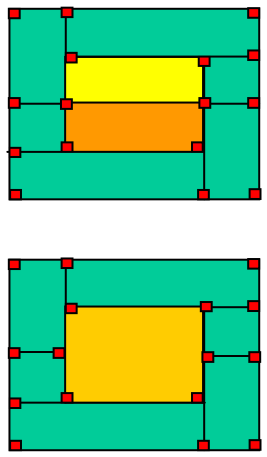


FIGURE 3.16. Impossible merging and the resulting zones.

Notice that this merge creates a regression tree that is not sound. In other words, once the two “areas” are merged, it will be impossible to cut the state space into two “areas” (any cut will violate the 3rd rule for a sound tree).

Therefore, we need a method that is able to merge two “areas”, if this is possible, and identify the situation, when it is not.

This is the method we propose:

a) Look for the “smallest” node (i.e. one with the smallest “area”) that is a predecessor of the two nodes, whose areas we want to merge. We call this node the **smallest predecessor**. In the example below, the smallest predecessor is the root node but it does not need to be.

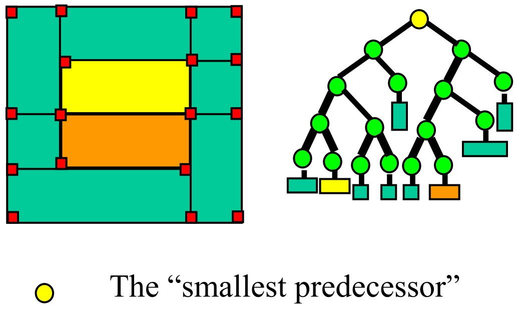


FIGURE 3.17. A merging example and its smallest predecessor.

b) Create a list that contains all the leaf nodes that are rooted at the smallest predecessor node (but instead of adding the two merging nodes to the list, insert a node that represents the already merged “area”). In this case, the list contains all the nodes in the tree but this is not always the case.

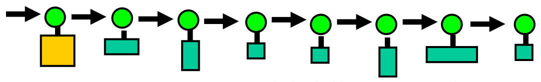


FIGURE 3.18. Initial list of nodes.

c) Create a temporary tree, with a root node equivalent to the smallest predecessor node.

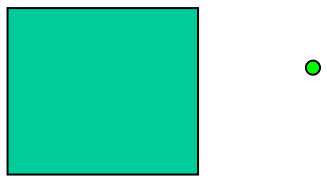


FIGURE 3.19. The initial temporary tree and the zone it represents.

d) Look for a way to break this node. The possible breakages are along any of the boundaries of the nodes in the list. However, no breakage can take place if it crosses any of the “areas” of the nodes in the list. The picture below illustrates a possible breakage and three not possible breakages

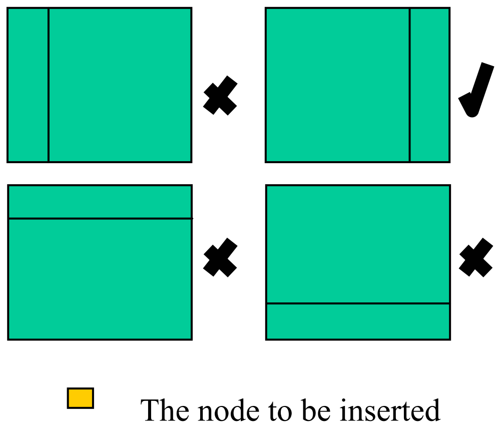


FIGURE 3.20. One possible breakage and three not possible ones.

e) Use this breakage to generate two children nodes for this root node. After this, break the current list of nodes into two lists, one associated with each one of the new children nodes. In other words, each list will contain the nodes that fit inside its associated child node.

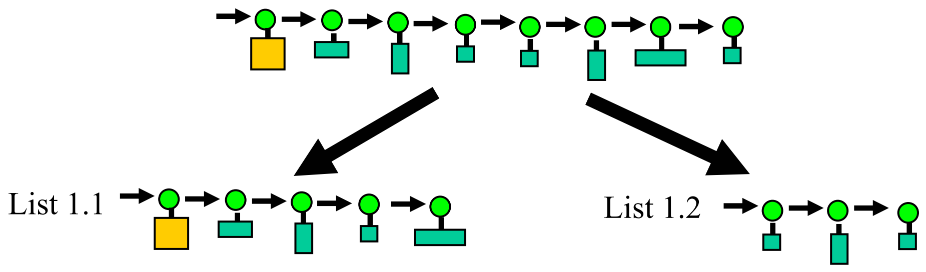


FIGURE 3.21. Two new lists after breakage.

f) This continues recursively until each list contains exactly one node. At this time, the node is inserted in the regression tree. The picture below illustrates the breakages selected until one of the nodes is inserted.

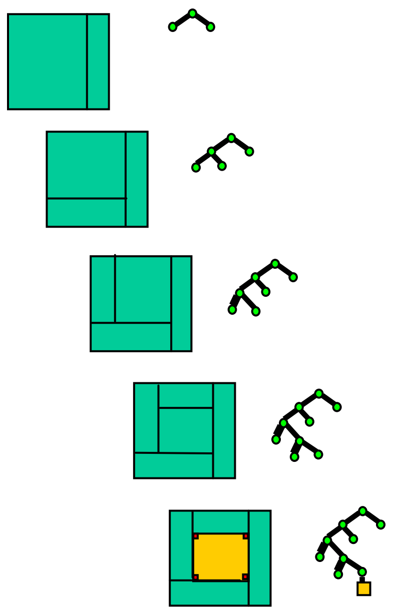


FIGURE 3.22. Breakages needed until the first node is inserted.

g) If any of the nodes in the original list fails to be inserted, the binary tree is left untouched and the two "areas" to be merged are left separated (some other merging process may be able to merge them in the future).

h) If all the nodes are inserted successfully, the temporal tree replaces the sub-tree rooted at the "smallest successor" node.

The picture below illustrates the steps needed to finish inserting the rest of the nodes:

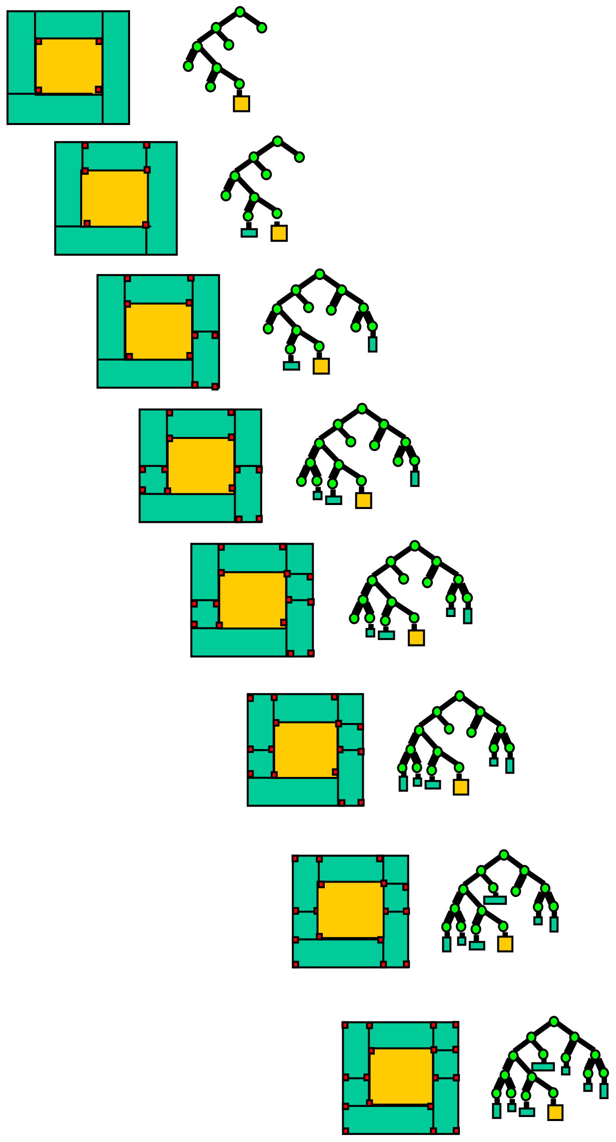


FIGURE 3.23. Steps needed until the rest of the nodes are inserted.

3.7 Summary

In the previous sections, we discussed several properties of the proposed method. For instance, we mentioned that a regression tree and prototypes are used to model the Q-function. Furthermore, we mentioned that these prototypes are no longer static, as they were in previous methods. These prototypes can and do move around in the state space during the learning process.

However, what we have not mentioned yet is how this model of the Q-function is updated, and how it is used by the agent. Let us start from the top:

The agent is an entity that can interact with a given environment through the use of **detectors** and **actuators**.

The detectors, such as cameras, Global Positioning Systems (also called GPS), inclinometers, etc allow the agent to perceive the current state of the environment. The information from each one of these detectors comes as a single analog signal with a certain range. For instance, the signal coming from an inclinometer could be in the range (-180, 180] and would represent the inclination in degrees.

On the other hand, the actuators are things like wheels, legs, etc that can change the environment's state. The agent controls these actuators by sending a signal to them. The values in this type of signal are also limited by a given range. For instance the signal going to some motor, which moves a couple of wheels could have the range [0,10) and could represent the voltage applied to this motor.

Furthermore, the agent has one more signal coming in, which provides a **reward** or a **punishment** after each interaction between the agent and the environment. This signal has also a certain range. The picture below illustrates the agent and the signals coming in and out of it:

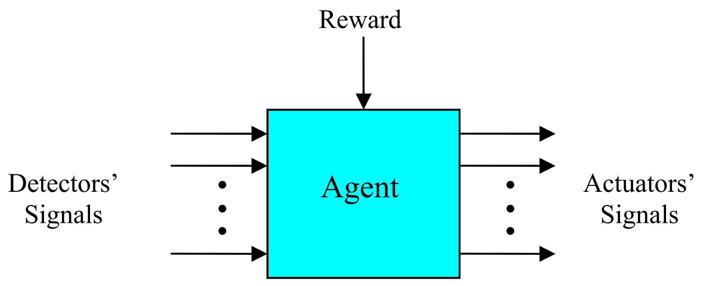


FIGURE 3.24. Signals coming in and out of the proposed agent.

When we talk about the number of dimensions in a problem, we refer to the total number of state and action signals coming in and out of the agent. For instance, in the case in which the agent only has one detector and one actuator, we will say that the problem at hand is a 2D problem. Notice that the dimensionality of the problem and the dimensionality of the state space for that problem are equivalent.

The way the proposed agent works is different to the way other agents using regression trees operate. Instead of interacting with the environment and gathering a group of learning experiences (i.e. a group of (*s*, *a*, *r*, *s*’) tuples) and then processing them in batch, the proposed agent processes the learning experiences as they come in, one by one. The way to interpret a learning experience, (*s*, *a*, *r*, *s*’), is the following:

> The agent applied an action *a* while in state *s* and received an immediate reward of *r* and it took the agent to the new state *s*’.

The fact that the new proposed method processes the learning experiences one by one has two important advantages:

a) First, it does not need to allocate memory resources to store the batch of learning experiences. This can be a very important feature, especially in problems with very large state spaces, where the size of each batch of learning experiences can be very large.

b) Second, in the proposed method, the Q-function is updated once for each learning experience as opposed to once for every batch of learning experiences, as the other methods using regression trees do. Trying to update the Q-function once for every learning experience has the potential of making the agent a much faster learner than the batch agent.

The way the proposed method updates the Q-function is the following:

1) The agent uses the new learning experience tuple to calculate how much the Q-function should change at this position in the state space (i.e. at the (*s*, *a*) position). The following formula is used to do that:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

Note: for more details on this formula refer to section 2.2.

2) Once this value is obtained, it is compared with certain threshold, b_{th}, called the **breaking threshold**. If the *Δvalue* is larger than b_{th}, the Q-function will be updated. Otherwise, the Q-function will be left untouched.

3) If the Q-function needs to be updated, we will look for a prototype in the regression tree that has associated the position (*s*, *a*) (remember that a prototype is defined as a (*s*, *a*, *v*) tuple).

   a) If such prototype exists, we will just change its value, *v*, which is calculated using the following formula:

$$v_{new} = v + \Delta value$$

b) If no such prototype exists, the regression tree will look for the leaf node whose area covers the position (s, a) and will partition this node as to include a new prototype with this new value. For instance, in the picture below, the Q-function needs to be updated at a position in the state space marked by the yellow circle.

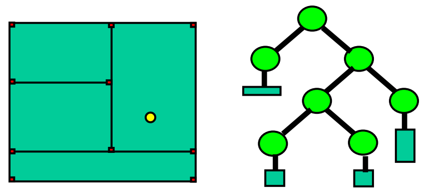


FIGURE 3.25. Q-function needs to be updated at indicated position.

The updated regression tree may look as follows:

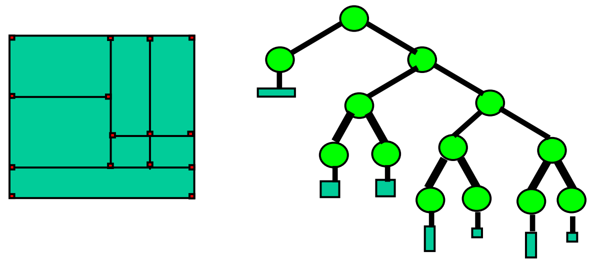


FIGURE 3.26. Q-function after the update.

After this, the merging process will kick in and will check whether it is possible to merge nearby nodes. After the merging process is completed, the Q-function may now look as follows:

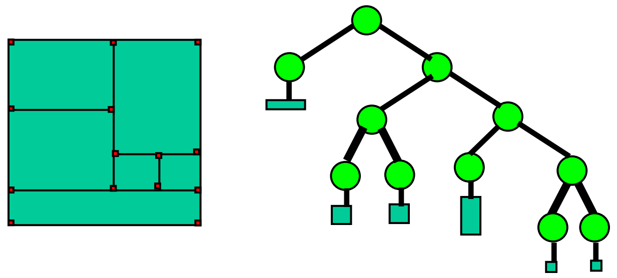


FIGURE 3.27. Q-function after the merging process.

Notice that, in this case, two nodes could be merged into one because they represented very similar functions. A threshold, m_{th}, called the **merging threshold**, is used to determine if two nodes are similar enough to be merged.

Thanks to the merging process we may be able to free prototypes that may be used in other zones of the state space in which the function may be more complex.

Remember that the new method uses two new parameters: the breaking threshold, b_{th}, and the merging threshold, m_{th}. As we described before, the breaking threshold is used to determine if a change to the Q-function is large enough to be worth modifying the regression tree. In a similar way, the merging threshold is used to determine if two nodes are similar enough to be merged. A nice feature of the proposed method is that the value of these two parameters is dynamic and depends on the number of free prototypes that are available at any given moment. The larger the number of free prototypes, the smaller the value assigned to these parameters. The smaller the number of free prototypes, the larger the value assigned to these parameters. In other words, if the algorithm is running out of free prototypes, this may mean that the Q-function is trying to approximate the Q-function too closely, for the given memory resources. At this time, it may be a good idea to stop updating the Q-function as frequently and try to perform more mergings (i.e., start approximating the Q-function less closely). This policy should free more prototypes and should bring about a kind of balance between wanting to follow the Q-function very closely and realizing that the memory resources are limited.

Finally, this method is aware that, in certain situations, the merging process may take a long time to finish. For instance, the merging process may find out that after merging two nodes, it may be possible to merge the new node with another one and so on to the point that it may be possible to merge the whole regression tree into a single node. This may cause some problem in situations, in which the agent is trying to learn at regular intervals of time or when the agent is both, learning and using the current policy. For instance, suppose that the agent is learning to drive a vehicle while driving a real vehicle. It is, at least theoretically, possible that the agent may find one of these long merging processes, and may “wake up” at the bottom of a cliff because no new action was taken in a long time.

However, this problem has a simple solution. After every interaction with the environment, we can put a limit on the amount of time the merging process can run. This will avoid this problem and will not really cause any serious drawback because further merging can take place in future interactions (freeing prototypes does not have to happen immediately).

Chapter 4

IMPLEMENTATIONS

We selected three problems in order to characterize quantitatively the properties of the proposed method against other existing methods.

Roughly, these problems are:

a) Come up with a policy that allows an agent to optimize the profit received by a store that sells a given product. This agent needs to optimize profit by setting the price for this product. We call this problem, the **Seller** problem. This problem was inspired by the work of Kephart and Tesauro [14].

b) Come up with a policy that allows an agent to balance a pendulum placed on a cart. This agent needs to balance the pendulum by moving the cart, either left, right or by not moving at all. We call this problem, the **Pendulum** problem.

c) Come up with a policy that allows an agent to control a Worm that needs to swim in water. This agent needs to make the worm move as fast as possible in a certain direction by controlling the worm's muscles. We call this problem, the **Worm** problem.

Each one of these problems provided a framework that was used to evaluate the proposed method and to compare it against other methods.

The following sections describe all this in detail.

4.1 Seller

4.1.1 Problem definition

In this problem we have two stores. Both stores sell the exact same product. Both stores get this product from the same wholesaler for the same price, C.

These two stores sell this product to 100 buyers. 90 of these buyers shop around before any purchase. 10 of these buyers buy the product from any vendor at random without shopping around (i.e. statistically, 5 buyers will buy from the first store while 5 buyers will buy from the second store)

The price set by the first store is controlled by a myopic agent. A myopic agent has a fixed price policy. This policy considers all the possible prices and selects the one that maximizes profit during that iteration. The price set by the second store is controlled by a Q-learning agent.

In this problem, the two stores take turns on setting the price of their product. They can set any price they want in the range [C + 0, *C* + 100], with granularity of 1, where *C* is the price the stores pay to the wholesaler for each product. For simplicity, we assume that *C* is equal to 0.

After one of the stores sets its price, each buyer buys exactly one piece of this product from one of the two stores. 90 of these buyers will purchase the product from the store that offers the lowest price and, statistically, 5 of these buyers will purchase the product from the first store and 5 will buy from the second store.

This problem can be stated as follows:

Provide a Q-learning implementation that is able to generate a price policy that maximizes the accumulated profit (i.e. profit in the long term). This method should try to come up with this optimized policy while trying to minimize memory requirements, the number of required learning experiences and the computational complexity.

4.1.2 Q-learning implementations

The Q-learning agent is implemented in three ways:

1) Using an explicit table (which we call the **explicit table implementation**).

2) Using regression trees and a batch method (which we call the **batch implementation**).

3) Using a regression tree and an on-line method (which we call the **proposed implementation**)

4.1.2.1 Explicit table implementation

In this implementation we used an explicit table to represent the Q-function. Since each agent is allowed to set the price to anything in the range [0,100] with a granularity of 1, we used a 101x101 table.

The horizontal dimension of this table represents the price set by the store controlled by the myopic agent (i.e. the "state" from the point of view of the Q-learning agent). The vertical dimension of this table represents the price set by the store controlled by the Q-learner (i.e. the "action applied" from the point of view of the Q-learning agent).

4.1.2.2 Batch implementation

This implementation was based on the one suggested by Dr. Manu Sridharan and Dr. Gerald Tesauro [20]. This method uses two regression trees, *T1* and *T2*, and processes the learning experiences in batch mode (i.e. a batch of learning experiences at a time).

This method begins by creating a root node for the two regression trees *T1* and *T2*.

A root node in *T1* is initialized to a value such as zero (i.e. the Q-function has a value of 0, everywhere).

The second regression tree, *T2*, is created using a given set of learning experiences. This regression tree has axis-parallel splits, select splits that minimize variance, and approximate the function by constant values in the leaf nodes.

Splits are selected as to minimize the formula:

$$|N1|\, v1 + |N2|\, v2$$

Note:	*N1*:	Number of learning experiences in the first group.
	v1:	Variance of the values associated with the learning experiences in the first group.
	N2:	Number of learning experiences in the second group.
	v2:	Variance of the values associated with the learning

Splits stop when either the variance of a group is small enough (I made this threshold equal to 0.1, the value range of the original batch of learning experiences) or until the group has less than n learning experiences in it (I made n = 2 in some experiments and n = 6 in others). At this point the regression tree node associated with this group of learning experiences is given a value (i.e. the average value of the learning experiences in this group).

Once the regression tree *T2* is created, it becomes regression tree *T1*, and a new *T2* is recalculated. During the construction of *T2*, the regression tree *T1* is used to obtain the parameter “max-Q” that is needed to evaluate the Q-learning update formula:

$$Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

More details can be found in [7] and [20].

4.1.2.3 Proposed implementation

We used the regression tree described in chapter 3 to implement the Q-function. This function was initialized to a value of zero.

The learning experiences were used, one at a time, to update the Q-function (i.e., on-line updating) [3]. This is very different from the previous implementation, in which the Q-function was updated after receiving a group of learning experiences (off-line updating) [3].

4.1.3 Experiments

We ran each of these implementations against a myopic agent for 10800 iterations (i.e. 10800 learning experiences). Each implementation came up with a different policy. Each policy was evaluated by letting it run against the myopic agent for 300 steps.

The policy obtained by using an explicit table obtained a total profit of $2,423,355 dollars.

The policy obtained by using the batch method obtained a total profit of $2,297,100 dollars.

The policy obtained by using the proposed method (Moving Prototypes) obtained a total profit of $2,423,355 dollars.

After normalizing these rewards, we obtain the following results:

Explicit table method	1.000
Batch method	0.948
Proposed method	1.000

TABLE 4.1, Normalized rewards for the three methods.

As expected, the method using an explicit table had the largest profit. The reason for this is that the pure Q-learning doesn't do any generalization on the found Q-function (as the batch and the proposed methods do). However, the Q-function does not need to be exact in order to imply an optimal policy (e.g. it does not matter if the Q-function has a value of 900.0 or 887.5 as long as this value is the largest for the given state). This seems to have happened with the proposed implementation and it managed to come up with a policy that was identical to the one obtained by the explicit table implementation. The batch method was not so "lucky" due to some of its properties, which will be discussed later in this section.

4.1.4 Explicit table vs. proposed

Performance:

Performance-wise, the proposed implementation did as well as the explicit table implementation. As a matter of fact, both implementations came up with the exact same policy (even though their Q-functions were not identical).

Computational complexity:

In terms of computational complexity, the explicit table implementation does better (O(*n*) vs. the proposed implementation's O(*n* log(*n*)), where *n* is the number of learning experiences used). To see the computational analysis for all 3 implementations refer to Appendix A.3.

Memory:

Memory-wise, the proposed implementation did much better than the explicit table implementation. The proposed method only needed to keep track of a maximum of 413 state-action-value tuples at any given time (i.e., one learning experience plus 412 prototypes in the regression tree). In contrast, the explicit table implementation needed to keep track of 10,202 state-action-value tuples at any given time (one learning experience plus 10201 prototypes in the explicit table).

Note: we call a prototype a piece of information used to model the Q-function.

4.1.5 Batch vs. proposed

Performance:

Performance-wise, the proposed implementation did much better than the batch implementation (the batch method performed 5.2% worse than the proposed implementation).

We let the batch method process more than 6 times the number of learning experiences used by the proposed method. At this time, the batch method had generated a policy that was 0.9% worse than the proposed implementation. This policy was better, but it was still not as good as the one obtained by the proposed method. This simulation was not run any longer (it had already taken more than 7 hours), but it is expected that eventually it would have generated a policy as good as the one generated by the proposed method.

Note: Manu Sridharan and Gerald Tesauro [20] talk about running this implementation for several dozens of iterations in order to achieve convergence (we ran it for less than a dozen of iterations).

We considered the possibility that the batch method might work better by shrinking the number of learning experiences in each batch and doing more iterations. We used batches of size 2601 (instead of size 10201) and then we run the batch algorithm for 10 extra iterations. The resulting policy performed 3.0% worse than the proposed implementation. This was a little better than using a big batch and less number of iterations. However, shrinking the number of learning experiences in each batch even more, again decreased the resulting policy performance (e.g. we tried batches of size 441 and 40 more iterations, and obtained a policy that was 13.2% worse than the proposed implementation). All these simulations not only performed worse than the proposed implementation, but also required about twice as many learning experiences (i.e. about 25,000 learning experiences vs. about 10,000 learning experiences used by the proposed implementation)

Finally, we tried changing the splitting policy. Instead of stopping splitting if we only had one prototype associated with the node, we decided to stop it as soon as we had 5 or fewer prototypes (this was the policy used by Manu Sridharan and Gerald Tesauro [20]). Unfortunately, this decreased the performance even further:

Size of batch	Number of extra iterations	Performance w.r.t. explicit table
441	46	32.1% worse
2601	10	16.4% worse
10201	10	7.7% worse

TABLE 4.2, Results using Sridharan-Tesauro Stopping convention.

Computational complexity:

In terms of computational complexity, the proposed implementation does much better than the batch implementation. In the best case scenario the proposed implementation has a computational complexity of O(n log(n)) vs. the batch implementation's O(n^2). In the worst case scenario, the proposed implementation has a computational complexity of O(n^2), whereas the batch implementation has a computational complexity of O(n^3). To see the computational analysis for all 3 implementations refer to Appendix A.3.

Memory:

Memory-wise, the proposed implementation performed much better than the batch implementation. The proposed method only needed to keep track of a maximum of 413 state-action-value tuples at any given time (i.e., one learning experience plus 412 prototypes in the regression tree). In contrast, the batch implementation needed to keep track of a maximum of 11975 state-action-value tuples at any given time (10201 learning experience plus 209 prototypes in the regression tree *T1* and 1565 prototypes in the regression tree *T2*).

4.1.6 Conclusions

As expected, the explicit table implementation had a better or equal performance than the other two implementations (i.e., the explicit table implementation doesn't lose information by using an approximate model of the Q-function). However, the proposed method did just as well as the explicit table implementation (it came up with a different Q-function but the exact same policy)

Also, the explicit table had a better computational complexity that the other two implementations. However, the proposed implementation was not much worse (i.e. O(n log(n)) vs. O(n)). On the other hand, the computational complexity of the batch implementation was significantly worse than the one for the explicit table implementation (O(n^2) vs. O(n)).

In addition, the proposed model did much better memory-wise than the explicit table method. The proposed method required about one order of magnitude less memory resources than the explicit table method. The proposed method has the capability of requiring even more orders of magnitude less memory if the granularity of the problem would be taken from $1 to something like 10 cents. Also, the proposed model has the capability of doing very well memory-wise as the number of dimensions increases (we know that the number of prototypes required for the explicit table method grows exponentially as we add more dimensions to the problem; but the proposed method does not share this problem, the proposed method only requires more prototypes as the complexity of the Q-function increases, which is completely independent from the number of dimensions of the Q-function).

Furthermore, the batch implementation did not do a very good job in this specific application. Performance-wise, it required the processing of many more learning experiences in order to generate a policy comparable to the one created by the explicit table and proposed implementations. Furthermore, it failed memory-wise because it required having a large number of state-action-value tuples all at once (this was mostly due to the fact that this method processes learning experiences in batches). We tried using smaller batches of learning experiences, but this prevented the method from generating a policy as good as the ones generated by the other two implementations. I believe two factors affect the batch implementation:

1) The fact that the nodes of the regression tree approximate the Q-function by using a constant (in the proposed implementation the value at any given position is doubly interpolated using the node's four prototype, which creates a hyper-plane offering a much smoother model of the Q-function).

Note: We also tried replacing the normal regression tree with the one used in the proposed method. This hybrid implementation resulted to be better than the pure batch method, but not as good as the proposed method.

2) The batch updating used by this implementation propagates rewards very slowly because it only updates the Q-function once after processing a whole batch of learning experiences. On the other hand, other methods such as the one we propose have the capability of updating the Q-function once for every processed learning experience.

Finally, this implementation proved to have a very bad computational complexity (O(n^2) in the best case scenario and O(n^3) in the worst case scenario). For illustration purposes, the proposed implementation ran for 10 minutes and was able to come up with a better policy than the one generated by the batch implementation after running for more than 7 hours (both were implemented using the same programming language and run on the same computer).

4.2 Pendulum

4.2.1 Problem definition

In this problem we have an inverted pendulum attached to the middle of a cart. The cart is allowed to move either forward or backward or to stay in the same place.

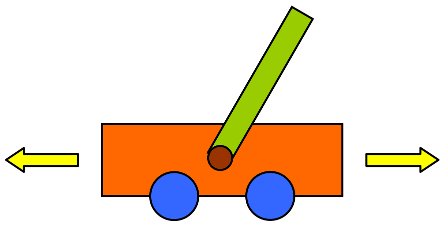


FIGURE 4.1. An inverted pendulum attached to the middle of a cart.

This problem can be stated as follows:

> *Provide a method that is able to generate a control policy that allows the cart to keep the pendulum in an upright position. This method should try to come up with this optimized policy while trying to minimize the memory requirements, the number of required learning experiences, and the computational complexity.*

4.2.2 Implementations

We attempted to solve this problem by using two implementations:

1) One based on the LSPI method
2) One based on the proposed method.

Michail G. Lagoudakis and Ronald Parr [16] proposed using a method called LSPI (Least-Squares Policy Iteration) in cases where the pure Q-learning memory requirement is too large. Theirs is an off-line method, in which learning experiences are gathered and then processed to come up with a policy.

We were able to obtain an implementation of this method through Dr. Parr's web page. Michail G. Lagoudakis and Ronald Parr used Matlab to implement their method and attempted to solve the Pendulum problem.

LSPI is a model-free method, and its Q-function is represented implicitly by a finite set of *n* parameters (w_1, w_2 ... w_n). The Q-function can be determined on demand for any given state by using the formula:

$$Q(s,a) = \sum_{i=1}^{n} \varphi i(s,a) w_i$$

where the set of *φi()* functions are called basis functions.

For a more detailed description of LSPI the reader is referred to [15] and [16].

4.2.3 Experiments

We used the LSPI and the proposed method to run the following three experiments:

Experiment 1 (big state space):

In this experiment we used the following state space:

Angle:	[-1.2, 1.2]
Angular velocity:	[-5, 5]

Also, we used the following reward function:

Reward = if (|angle| > π/3) ==> -1 (note: π /3 = ~1.04)

else ==> 0

LSPI:

After allowing LSPI to use 1902621 learning experiences, the policy generated was only able to keep the pendulum upright for an average of 26 time units.

Proposed:

After allowing the proposed method to use 216 learning experiences, the policy generated was able to keep the pendulum upright indefinitely.

Experiment 2 (medium size state space):

In this experiment we used the following state space:

Angle:	[-1.0, 1.0]
Angular velocity:	[-3, 3]

Also, we used the following reward function:

$$\text{Reward} = \begin{cases} \text{if } (|\text{angle}| > \pi/3.4) \implies -1 & (\text{note: } \pi/3.4 = \sim 0.92) \\ \text{else} \implies 0 & \end{cases}$$

LSPI:

After allowing LSPI to use 183618 learning experiences, the policy generated was only able to keep the pendulum upright for an average of 170 time units.

Proposed:

After allowing the proposed method to use 324 learning experiences, the policy generated was able to keep the pendulum upright indefinitely.

Experiment 3 (small state space):

In this experiment we used the following state space:

Angle:	[-0.6, 0.6]
Angular velocity:	[-3, 3]

Also, we used the following reward function:

Reward = if ($|angle| > \pi/6$) ==> -1 (note: $\pi/6 = \sim 0.52$)

else ==> 0

LSPI:

After allowing LSPI to use 648 learning experiences, the policy generated was able to keep the pendulum upright indefinitely.

Proposed:

After allowing the proposed method to use 216 learning experiences, the policy generated was able to keep the pendulum upright indefinitely.

4.2.4 Realizations

From these results we came up with the following realizations:

4.2.4.1 Performance

The proposed model was always able to come up with a policy that solved the pendulum problem. Furthermore, the proposed method was able to do this using very few learning experiences (i.e. about 300 learning experiences).

In contrast, the LSPI was not always able to solve the problem (i.e. it failed in two out of three experiments). However, we have to concede that the LSPI was able to solve the problem in one of the experiments and it did this using a relative small amount of learning experiences (i.e. 648 learning experiences).

If you are wondering why the second experiment used 324 learning experiences while the first and third only required 216, I'm afraid I may not be able to provide a definite explanation. However, I may be able to offer some insight as to why this can happen. The reason why I cannot provide a definite explanation is because of the way Q-learning works. Q-learning does not work by manipulating logical statements, which I may be able to review and find exactly what happened. Instead, Q-learning solves a problem indirectly by making a very large number of small updates to its Q-function. This large number of operations plus the fact that the problem is solved indirectly makes any direct tracing almost impossible. However, all this said, it is possible to provide some insight as to what may have produced this. First, the way these experiments were carried out was by going through episodes. Each episode contained exactly 108 learning experiences. In the case of the first and third experiment, the agent was able to come up with a good enough policy after processing two episodes. On the other hand, the second experiment required a third episode. Perhaps processing a whole new episode may have been an overkill and just using a few more learning experiences might have done the job, but at the time we were very happy with the results and we did not think it was important to shrink this number any more (i.e., 324 was very good when compared to the 183618 learning experiences LSPI had used without any success). The second and last insight is that the learning experiences used at each one of the experiments were different which makes each experiment unique. In other words, it is not possible to compare different experiments quantitatively with each other, and it is entirely possible that each experiment requires a different number of learning experiences in order to come up with a good enough policy. It just so happened that the number of learning experiences required by the first and third experiments fell within the second episode, while the second experiment required a few more learning experiences than the ones contained in two episodes.

4.2.4.2 Memory

The current implementation of LSPI, apart from storing the samples, requires only $O(k^2)$ space (independently of the size of the state and the action space) [16]. Although this is not the way it is currently implemented, it would be easy to change the program such that the samples are fed one by one. That way this method would not require memory to store a batch of samples (some problems may require these batches of samples to be very large and changing this may be very advantageous).

In the case of the proposed method, the memory requirements depend completely on the complexity of the Q-function. In this case, the proposed method only required memory to store less than 200 state-action-value tuples (one incoming sample and about 170 prototypes to define the Q-function)

4.2.4.3 Computational complexity

For each sample, LSPI incurs a cost of O(k^2) to update the matrices A and b and a one time cost of O(k^3) to solve the system and finding the weights [16].

Therefore, the computational complexity of LSPI is:

$$n * O(k^2) + O(k^3)$$

where:

n = the number of learning experiences (or samples)
k = the number of w parameters.

In other words, the computational complexity of LSPI is:

$$O(n*(k^2)) + O(k^3)$$

This method may prove to be a good choice in cases where k is a very small number. In these cases the computational complexity of this method becomes:

O(n)

This is better than the proposed method computational complexity. Remember that the proposed method has a computational complexity that goes from O(n log(n)), in the best case scenario, to O(n^2), in the worst case scenario.

However, as we will see in the following section, LSPI has some disadvantages that may preclude this method from doing a very good job in most problems.

4.2.4.4 Notes on the LSPI method

In some cases, the LSPI method may be able to learn very quickly and using very small memory resources. However this method appears to have several disadvantages:

1) In order for LSPI to work well, the user needs to provide this method with good basis functions. Finding good basis functions may turn out to be extremely difficult.

Explanation:

The LSPI method uses a linear function (a parametric function) to model the Q-function. Obviously, most Q-functions are not linear. It is my belief that the LSPI method uses basis functions to linearize the Q-function.

The problem is that normally, the Q-function is not known *a priori*. Therefore, looking for good basis functions is almost a blind search (all you can try to use is your knowledge for that specific application, which in many cases may be very limited).

2) The effectiveness of a set of basis functions seems to be very sensitive to the range of the samples used to learn the task at hand:

As an example, look at the 3 experiments. All of these experiments used the same basis functions, but the first and second experiments failed to solve the problem (i.e., to keep the pendulum in an upright position).

What seems to have happened here is that the used basis functions were able to "linearize" the Q-function over a certain area of the state space, but not over other areas.

3) This method appears to build the value function off-line, using all the available learning experiences at once. This has two disadvantages:

   a) This method, as is, cannot be used for some of the larger problems:

      It is plain to see that if we build the Q-function using all the available learning experiences, for some problems with very large state space and complex "topology", we will need many of them, to the point that it is not feasible to store them all in memory.

      Note: It may be feasible to change the LSPI method so that it builds the Q-function using only a few learning examples at a time. However, since LSPI solves a set of linear equations every time it updates the Q-function (with a computational complexity of O(k^3)) this may turn out to be too expensive, CPU-wise.

b) This method may not converge very fast (in comparison with the one I propose):

    The method LSPI only updates the Q-function from time to time (after gathering and processing a batch of learning experiences). On the other hand, the proposed method updates its Q-function after each learning experience. This “slow” update of the Q-function slows down the learning process.

4.2.5 Summary

Moving Prototypes and LSPI were used to try to solve the so called **pole-balancing task** [3]. Performance-wise, Moving Prototypes outperformed LSPI by solving the balancing problem in all three experiments (LSPI was only able to solve the problem in one experiment). Memory-wise, LSPI outperformed Moving Prototypes. Furthermore, LSPI had a better computational complexity than Moving prototypes (i.e., O(n) versus O(n log(n)), in the best case scenario). However, for LSPI to work well, the user needs to provide this method with good basis functions. Finding good basis functions may turn out to be extremely difficult and in some cases impossible without *a priori* knowledge of the Q-function, which in most problems is not available. Furthermore, LSPI builds the Q-function off-line. This means that the method may become impractical for problems with very large state spaces and/or complex “topology,” because they may require storing simultaneously large numbers of learning experiences.

4.3 Worm

4.3.1 Problem definition

This problem involves the use of a black box (called Framsticks) that is able to simulate agents, an environment and their iteration. This black box is configurable and it is possible to define things like the environment's relief, its water level, etc. It is also possible to configure the number of agents interacting in this environment and their characteristics, such as the number of articulations, where their muscles are located, their strength, etc. All these agents are composed of sticks, hence the name for this simulator (i.e. Framsticks). In this problem we decided to use a single agent, which we call WORM, and an environment, which we call SEA.

The environment, SEA, has the following characteristics:

1) The relief is flat.

FIGURE 4.2. Relief in the SEA environment.

2) The water level is 20 units.

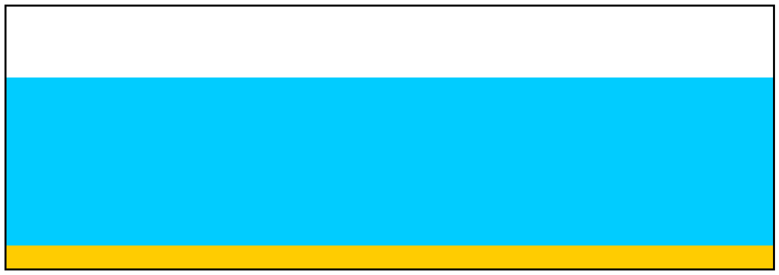


FIGURE 4.3. Water level in the SEA environment.

3) The environment is large enough that the agent will never reach the end.

The agent, WORM, has the following characteristics:

1) It is conformed of 4 sticks (each 1 unit long)

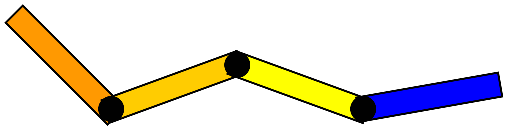


FIGURE 4.4. The frame in the WORM agent.

2) The worm has 3 muscles (each moving two contiguous sticks)

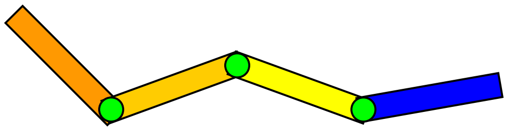


FIGURE 4.5. The muscles and the frame in the WORM agent.

3) Each of the muscles has different capabilities. The first muscle is able to rotate its two sticks with a range of (-120,120) degrees. The range for the other two muscles are (-90, 90), and (-40, 40) respectively.

4) When placed in the environment, the agent WORM stays submerged under water (but without touching the floor):

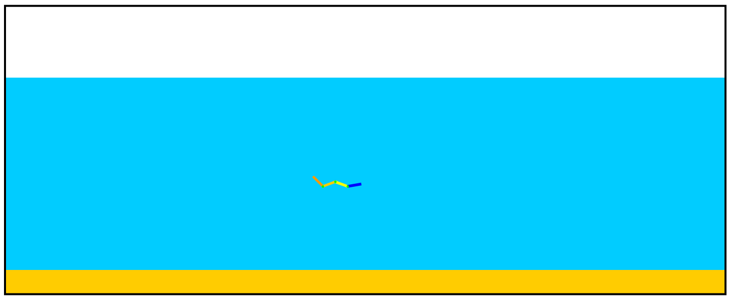


FIGURE 4.6. The WORM agent's elevation in the SEA environment.

This problem can be stated as follows:

Provide a method that is able to generate a control policy that allows the agent WORM to move as fast as possible in the positive x direction without moving significantly in any other direction. This method should be able to come up with this optimized policy without requiring a model of the environment, the agent or their interaction.

The proposed learning method was used to come up with this policy and the following reward function was used:

$$Reward(t) = \Delta x(t) - 0.33|\Delta y(t)|$$

Where:

t = current time index

$\Delta x(t) = x(t) - x(t\text{-}1)$

$\Delta y(t) = y(t) - y(t\text{-}1)$

Note: the $\Delta x(t)$ part of the reward function, indirectly, tells our method that we want to move as fast as possible in the x direction. The $0.33|\Delta y(t)|$ part of the reward function, indirectly, tells the proposed method that we want to move in the x-direction as straight as possible.

The state space in this problem is any integer in the range [0, 9]. These values represent the general heading of the agent WORM. More specifically, these values represent the following angular ranges:

0 ➔	(-180,-144]	5 ➔	(0, 36]
1 ➔	(-144,-108]	6 ➔	(36, 72]
2 ➔	(-108,-72]	7 ➔	(72, 108]
3 ➔	(-72,-36]	8 ➔	(108,144]
4 ➔	(-36, 0]	9 ➔	(144, 180]

The agent is allowed to use any action in the range of integers [0, 74]. These values are codes that represent the movement of the 3 muscles. There are three types of movements encoded in these values:

1) The **D** movement (with possible values: 0, 1 and 2)
2) The **Phase-1** movement (with possible values: 0, 1, 2, 3 and 4)
3) The **Phase-2** movement (with possible values: 0, 1, 2, 3 and 4)

All the combinations of these tree values are encoded into a unique value in the range [0, 74], by using the following formula:

ActionCode = (**D**)*25 + (**Phase-1**)*5 + (**Phase-2**)

The 3 muscles move as to complete a cycle of their allowed angular values (for instance, the first muscle moves as to achieve the angles: 110, 109, 108, 107…2, 1, 0, -1, -2, -3, -4 . . .-119, -120, -119, -118…-2, -1, 0, 1, 2… 108, 109, 110, 111,…119, 120, 119, 118, … 113, 112, 111 and 110).

Each muscle completes its cycle with a certain phase (i.e. phase-0 for the first muscle, phase-1 for the second muscle and phase-2 for the third muscle). The phase “phase-0” is always equal to 0. The value of the phases “phase-1” and “phase-2” is given by the value of the variable **ActionCode**.

Finally, the value of the parameter **D** affects the range of values for the first muscle in the following way:

D = 0 ➔ range for the first muscle [-30, 120]

D = 1 ➔ range for the first muscle [-120, 120]

D = 2 ➔ range for the first muscle [-120, 30]

4.3.2 Experiments

We ran the proposed implementation for 25000 steps. Then, ordered the agent to use the learned policy to control the worm to move as fast as possible in the positive x-direction.

The agent WORM did this as told. These are some frames showing the agent's performance (note: a blue dot indicates what we call the "head" of the worm):

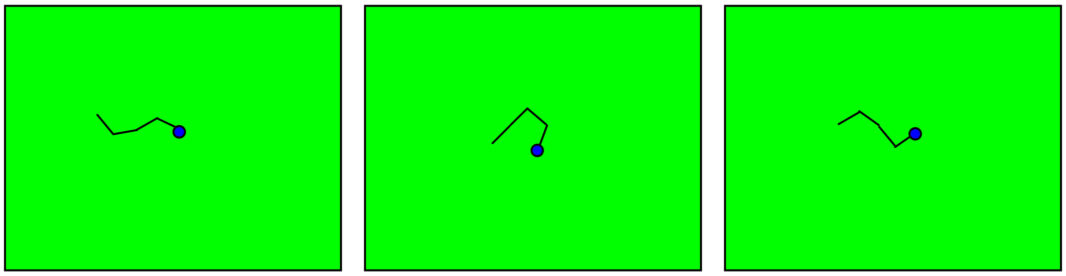


FIGURE 4.7. First three frames of the WORM's performance.

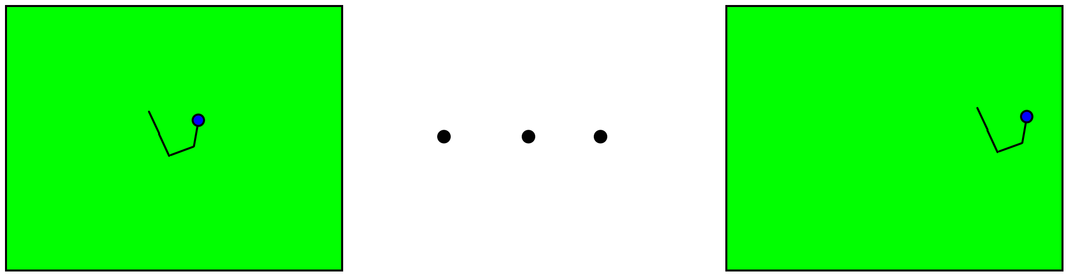


FIGURE 4.8. frames 4th and 12th of the WORM’s performance.

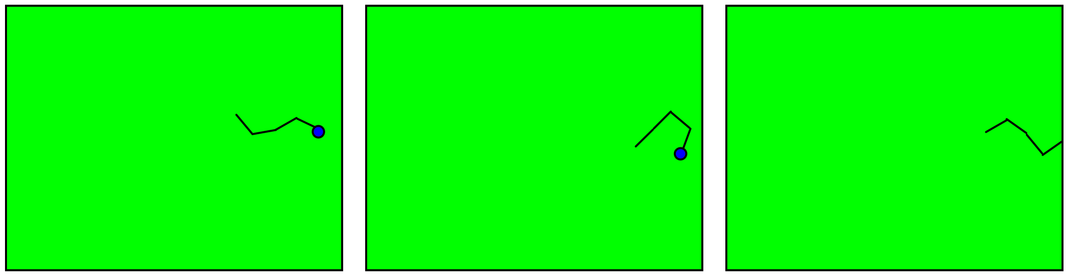


FIGURE 4.9. frames 13th through 15th of the WORM’s performance.

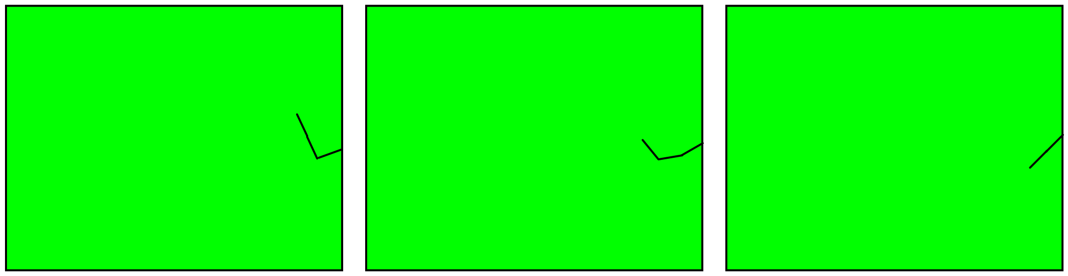


FIGURE 4.10. frames 16th through 18th of the WORM’s performance.

After asking the agent to move in the positive x direction, we asked it to go in the negative x direction. To do that we just introduced an offset of 180 in the input signal.

We began this experiment by placing the agent so that its “head” pointed in the positive x direction.

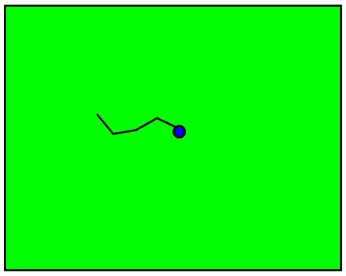


FIGURE 4.11. Initial position of the WORM agent.

It was very interesting to observe that the agent found that the most efficient way to move in the negative x direction was to, first, quickly turn around and then move in the negative x directions.

The following frames show how the agent WORM moves in a way that turns its body around.

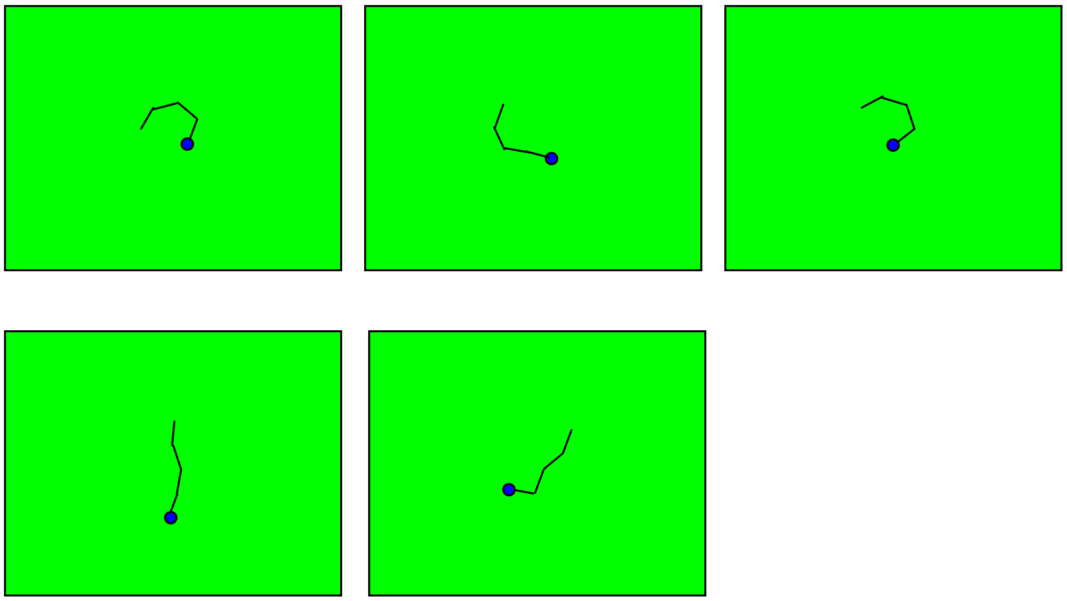


FIGURE 4.12. First five frames showing the WORM turning around.

Once the body has been turned around the agent begins to move very efficiently in the negative x-direction.

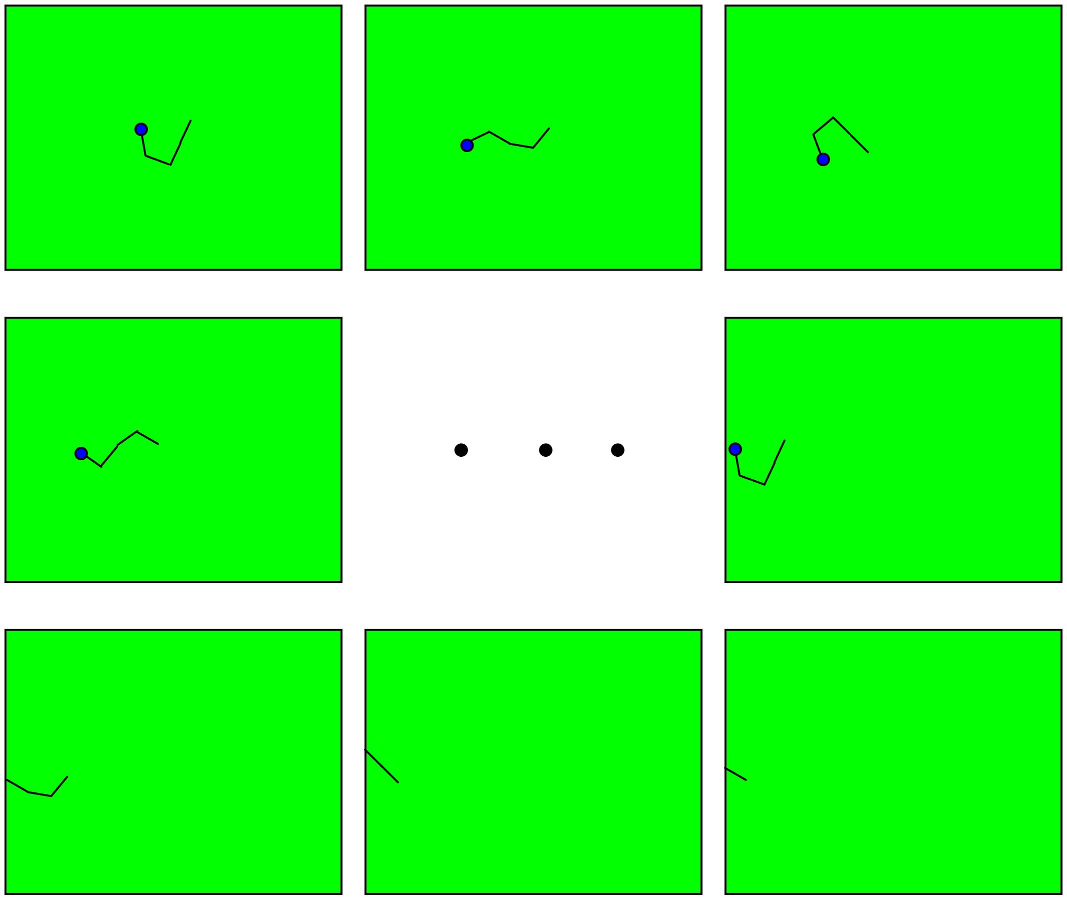


FIGURE 4.13. The other frames before the WORM disappears of our view.

As a last experiment we asked the agent to move in a circle (by slowly increasing the offset on the input signal). The agent WORM moved in a practically perfect circle.

Subsection 4.3.3: Summary

This problem was selected not as much as a mean to compare Moving Prototypes with other methods but rather as a way to evaluate the proposed method in a scenario we think is the ideal for it.

In other words, we wanted to try to solve a problem for which:

1) The model of the environment is unknown.

2) A human being is unable to provide the Learning Agent with application specific knowledge.

Framsticks provided a framework in which these two situations are true:

1) The model used by Framsticks to simulate the environment is a black box for the user.

2) Framsticks allows the construction of complex entities, such as a water worm. A human may be hard pressed if asked to come up with good tips on how this water worm should move as to attain maximum velocity in a given direction from any initial configuration (e.g. the head of the worm pointing 125 degrees from the desired direction).

The proposed method was able to solve this problem quickly (i.e., in less than 2 minutes and using less that 25,000 learning experiences) and required very little memory (i.e., less than 900 state-action-value tuples)

As a side note, we would like to mention that **genetic learning** has also been used to solve this problem. We were not able to get details on the used implementation but we do know that this method was also successful on solving this problem. However, genetic learning required many more learning experiences (i.e. thousands of generations, each one using thousands of learning experiences [7]).

Chapter 5

SUMMARY

After analyzing several methods used to model the Q-function we identified the following shortcomings:

- Some methods, such as Tile coding, require memory resources that grow exponentially with the number of dimensions of the problem.

- Some methods, such as Kanerva coding, do a poor job at modeling the Q-function because they distribute memory resources evenly throughout the state space (instead of moving memory resources from places where the function is "simpler" to places where the function is more "complex").

- Some methods, such as LSPI, require the user to provide information that is difficult or impossible to find. For instance, in the case of LSPI, the user of this method needs to provide good examples of what are called "basis functions". However, in order to provide a good set, the user often needs to have an idea of what the Q-function looks like. This is a vicious circle, since the ultimate goal is to find the Q-function.

- Some methods, such as ANN, require the processing of a very large number of learning experiences in order to find a good, if not optimal, policy.

In order to address the shortcomings found in these methods we proposed a new approach to modeling a Q-function, which we called **moving prototypes**. This method was based on the use of a regression tree, just as some other methods do such as the one proposed by Sridharan and Tesauro. Furthermore, this method adopts the idea of using prototypes to model the Q-function, as Kanerva coding does. However, in this new method the prototypes are no longer "static". In other words, we may initially distribute the prototypes evenly throughout the state space but their position may change during the learning process. This movement of prototypes will not only try to avoid having "redundant" prototypes but will also position new prototypes in zones where the Q-function turns out to be more complex.

This new approach was implemented and tested in three different scenarios: a) a market place, b) a balanced pendulum and c) an autonomous swimmer.

In the first scenario, Moving Prototypes was compared to a tabular Q-learning method and to a batch method using regression trees, an approach proposed by Sridharan and Tesauro. Performance-wise, Q-learning using Moving Prototypes and the tabular Q-learning outperformed the batch method. Although the tabular Q-learning method offered a better computational complexity than Moving prototypes (i.e. O(n) versus O(n log(n)), in the best case scenario, and O(n^2), in the worst case scenario), Moving Prototypes performed much better memory-wise (by two orders of magnitude). Moving prototypes will do even better memory-wise when confronted with problems of larger dimensionality and/or finer granularity. The batch implementation performed very poorly in this specific application. First, its computational complexity was found to be O(n^3). Second, it required the processing of a very large number of learning experiences in order to generate a policy comparable to the one created by the other two implementations. Furthermore, it failed memory-wise, because it required a large number of learning experiences to be stored all at once. We tried using smaller batches of learning experiences, but this prevented the method from generating a policy as good as those generated by the other two implementations.

In the second scenario, Moving Prototypes was compared to LSPI. Performance-wise, Moving Prototypes outperformed LSPI by solving the balancing problem in all three experiments (LSPI was only able to solve the problem in one experiment). Memory-wise, LSPI outperformed Moving Prototypes. Furthermore, LSPI had a better computational complexity than Moving prototypes (i.e., O(n) versus O(n log(n)), in the best case scenario). However, for LSPI to work well, the user needs to provide this method with good basis functions. Finding good basis functions may turn out to be extremely difficult and in some cases impossible without *a priori* knowledge of the Q-function, which in most problems is not available. Furthermore, LSPI builds the Q-function off-line. This means that the method may become impractical for problems with very large state spaces and/or complex “topology,” because they may require storing simultaneously large numbers of learning experiences.

In the third scenario, Moving Prototypes was used to train an agent, whose task it was to swim as fast as possible through water in a given direction. The reason for this experiment was to evaluate the performance of the proposed method in a scenario that we consider ideal for this method, namely one, for which there is no application specific knowledge available. In this scenario, the agent does not know anything about the model controlling the environment. From the point of view of the agent, the environment is a black box. Moving prototypes was able to solve this problem quickly. It took this method less than 2 minutes and used less that 25,000 learning experiences. Furthermore, this method required very little memory (less than 900 state-action-value tuples). Genetic learning had been proposed in the past for solving this problem. However, it required many more learning experiences. That method required thousands of generations, each one using thousands of learning experiences.

In continuation of this research effort, it may be interesting to explore the use of different types of splits. In the research effort presented in this thesis, we only applied axis-parallel splitting, but it may turn out that allowing other types of splits, such as diagonal splitting, may offer advantages.

Furthermore, we should continue to characterize the method Moving Prototypes, so that we can better pinpoint the types of problems that this method is most suitable for. Right now we suggest using this method on problems, for which little or no application specific knowledge is available, and for which an optimal distribution of prototypes in the Q-function cannot be provided *a priori*.

In the future, we may want to investigate the use of moving prototypes as the first stage of learning. Once, we have a rough idea of what the Q-function looks like, we may decide to use another method, such as LSPI, to further refine the Q-function.

One venue we have high hopes for is the use of *eligibility traces* in conjunction with moving prototypes. Eligibility traces is a mechanism that allows an agent to learn more efficiently by keeping track of a set of recent learning experiences [3]. The use of Eligibility traces allows the agent to propagate rewards beyond the previous state, as the basic Q-learning method does. This allows multiple updates of the Q-function per learning experience, which in turn accelerates learning. The fact that updating the Q-function is so cheap in moving prototypes, as compared to batch methods such as the one proposed by Sridharan and Tesauro, makes us think that the use of eligibility traces may produce a very powerful method.

Appendix A

Notes

A.1 Leaf nodes to total number of nodes ratio in a regression tree

Initially, a regression tree has one leaf node and the total number of nodes is one:

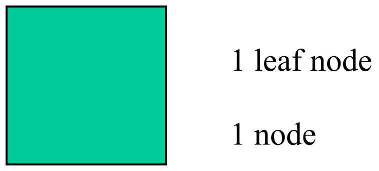


FIGURE A.1, Number of leaf nodes and total nodes in the initial regression tree.

Every time we split the tree we will end up with 1 more leaf node and 2 more nodes:

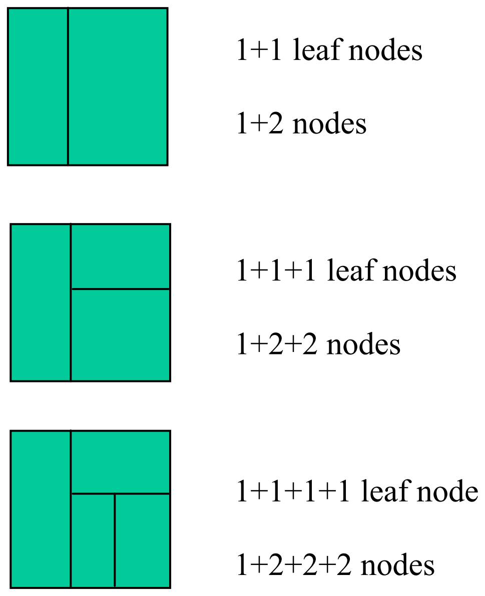


FIGURE A.2, Number of leaf nodes and total nodes as we keep splitting.

Therefore:

$$numberOfNodes = (numberOfLeafNodes - 1)*2 + 1$$

$$numberOfNodes = numberOfLeafNodes*2 - 2 + 1$$

$$numberOfNodes = numberOfLeafNodes*2 - 1$$

A.2 Relationship between nodes and prototypes in the proposed method

Best case scenario:

Let us assume we have a 2D regression tree. In this case, the proposed method will initialize this tree by putting 4 prototypes (one in each corner):

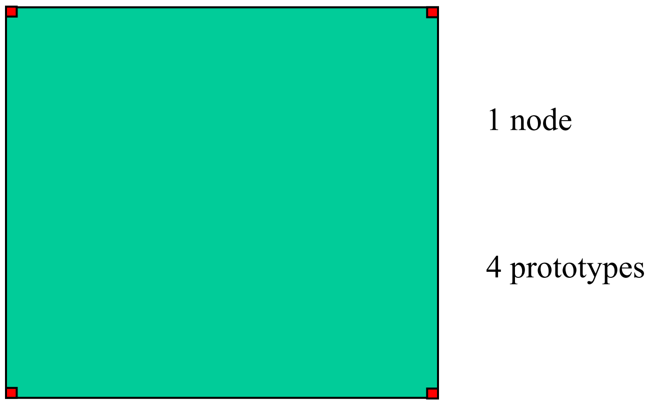


FIGURE A.3, Initial regression tree and its four prototypes.

In the best case scenario, every time we split a node in the regression tree we will end up with 2 more prototypes and 2 more nodes.

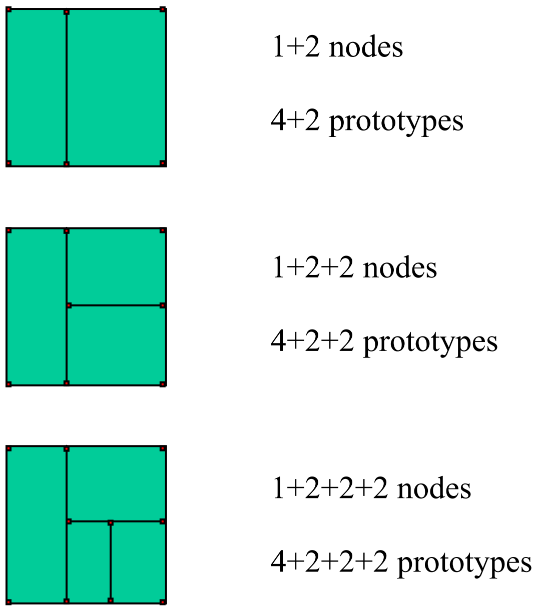


FIGURE A.4, Best case scenario splitting (more prototypes per node).

Therefore, in the best case scenario:

$$numberOfNodes \geq numberOfPrototypes - 3$$

$$numberOfPrototypes \leq numberOfNodes + 3 \qquad \text{(in the 2D case)}$$

Worst case scenario:

In the worst case scenario, each prototype will be shared by 4 leaf nodes (and therefore will be counted 4 times):

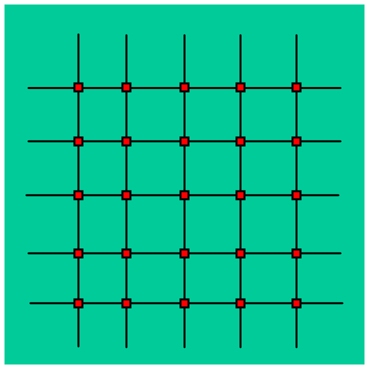


FIGURE A.5, Worst case scenario splitting (fewer prototypes per node).

Therefore:

$$numberOfPrototypes \geq \frac{numberOfLeafNodes * numberOfPrototypesPerNode}{numberOfTimesWeCountEachPrototype}$$

$$numberOfPrototypes \geq \frac{numberOfLeafNodes * 4}{4}$$

$$numberOfPrototypes \geq numberOfLeafNodes$$

Since:

$$numberOfNodes = 2*(numberOfLeafNodes) - 1$$

$$numberOfLeafNodes = (numberOfNodes + 1)/2$$

Then:

$$numberOfPrototypes \geq (numberOfNodes + 1)/2 \quad \text{(in the 2D case)}$$

Summary:

In summary, the relationship between the number of prototypes and the number of nodes in the proposed regression tree is:

$$(numberOfNodes + 1)/2 \leq numberOfPrototypes \leq (numberOfNode + 3)$$

In other words, if we know that we have P prototypes in the tree, we should expect to have somewhere between P and 2P nodes in the proposed regression tree (in the 2D case).

A.3 Computational complexity analysis:

In this analysis we shall make a few simplifying assumptions based on the seller implementation.

In this implementation we had a 2D Q-function state space that was completely square:

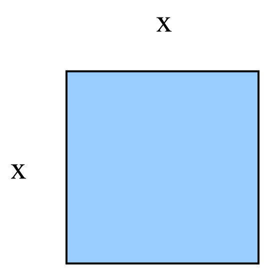


FIGURE A.6, The two dimensions of the state space have the same range.

Let us call x the number of possible values in each dimension.

Let us call n the number of samples fed to each of the three implementations.

In the seller problem, the number of samples used and the number of possible values in each dimension have the following relationship:

$$n = x*x$$

In other words, we select a sample for every possible position in the state space.

A.3.1 Explicit table implementation:

For this application, the computational complexity of updating the Q-function using a single sample is:

$$O(1).$$

Therefore, the computational complexity of updating the Q-function using n samples is:

$$O(n)$$

A.3.2 Batch implementation:

Best case scenario:

The best case scenario for this implementation is one in which each split divides samples in groups of equal cardinality.

Let us call NoC the number of possible splits for a given node, and let us call NoS the number of samples analyzed at each possible split.

Therefore, if we start splitting in this fashion we will obtain something like this:

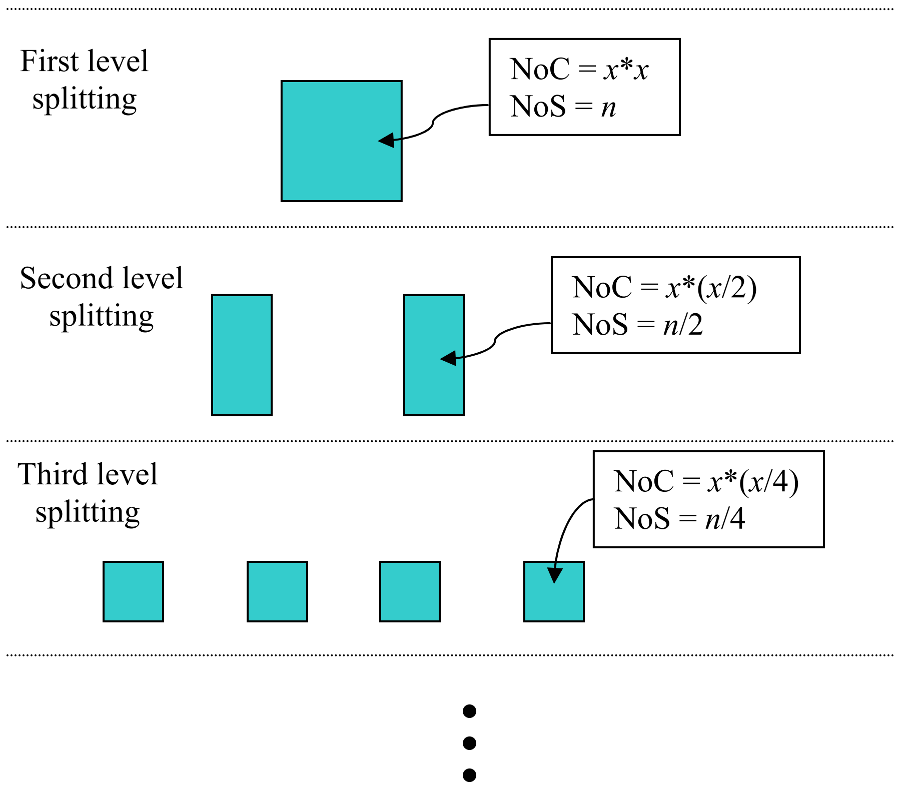


FIGURE A.7, Best case splitting (spitting of equal cardinality).

Since the work, W, done at each split is equal to NoC times NoS, then the work to create the regression tree given a set of n samples will be:

$$W = 1[x*x*n] + 2[x*(x/2)*(n/2)] + 4[x*(x/4)*(n/4)] + \ldots.n[x*(x/n)*(n/n)]$$

$$W = (x^2)*n + (x^2)*(n/2) + (x^2)*(n/4) + \ldots(x^2)*(n/n)$$

$$W = (x^2)*n*[\ 1 + \tfrac{1}{2} + \tfrac{1}{4} + \ldots\ 1/n]$$

Since we know that the following infinite series converges to 2:

$$1 + \frac{1}{2} + \frac{1}{4} + \ldots\ldots.$$

then:

$$W \leq (x^2)*n*2$$

Therefore, the computational complexity in this best case is:

$$O(x^2)*n$$

And, since for the seller problem, n is typically equal to x^2, then the computational complexity becomes:

$O(n^2)$ (best case scenario for the seller problem)

Worst case scenario:

In the worst case scenario, each split divides the samples into one group with only one sample and another group with the other samples.

If we start splitting in this fashion, we will obtain something like this:

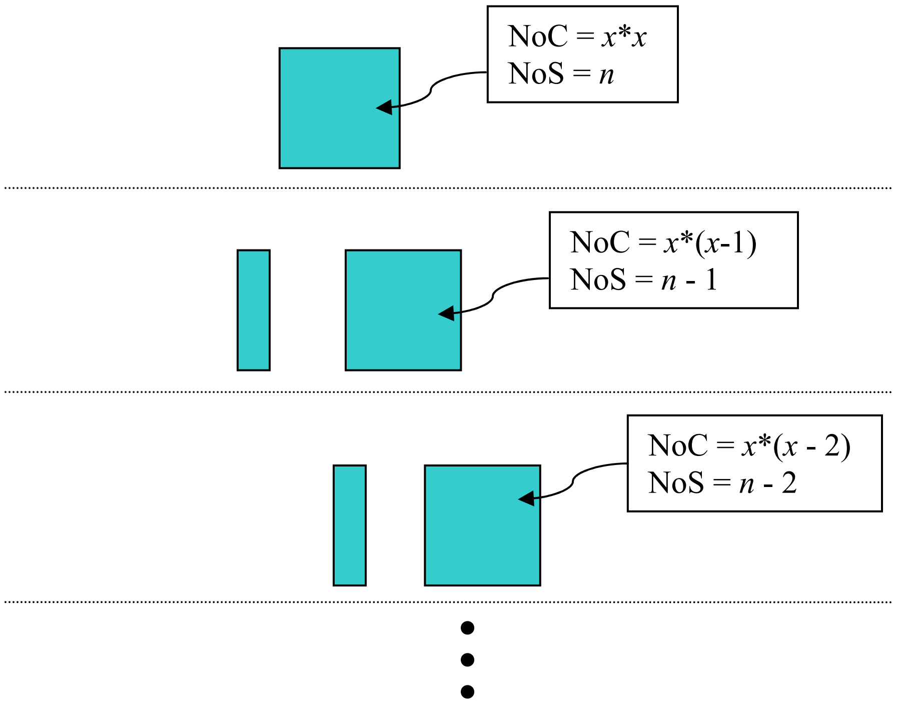


FIGURE A.8, Worst case splitting (always one sample on one side).

Therefore:

$$W = x*x*n + [x*(x-1)*(n-1)] + [x*(x-2)*(n-2)] +$$

$$[x*(x-3)*(n-3)] + \ldots. + [x*(x-n)*(n-n)]$$

$$\mathrm{W} = (x^2)*n + [(x^2 - x)*(n-1)] + [(x^2 - 2x)*(n-3)] +$$

$$[(x^2 - 3x)*(n-3)] + \ldots + [(x^2 - nx)*(n-n)]$$

Since we know that:

$$O(x^2 - ax) = O(x^2) \qquad \text{(where a is a is some constant)}$$

then:

$$W = O(\ (x^2)*n + [(x^2)*(n\text{-}1)] + [(x^2\ )*(n\text{-}3)] +$$
$$[(x^2)*(n\text{-}3)] + \ldots + [(x^2)*(n\text{-}n)]\ )$$

$$W = O(\ (x^2)*[n + (n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots + (n\text{-}n)]\ )$$

We know that:

$$n + (n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots + (n\text{-}n) =$$

$$(n+(n\text{-}n)) + ((n\text{-}1) + (n\text{-}(n\text{-}1))) + \ldots =$$

$$n + n + n + n + \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots =$$

$$\mathit{numberOfElements}*\mathrm{n} =$$

$$(n/2)*n$$

Therefore:

$$W = O(\ (x^2)*[(n/2)*n])$$

$$W = O(\ (x^2)*(n^2)/2)$$

$$W = O(\ (x^2)*(n^2))$$

And, since for the seller problem, n is typically equal to x^2, then the computational complexity becomes:

$O(n^3)$ (worst case scenario for the seller problem)

A.3.3 Proposed implementation:

Best case scenario:

The best case scenario for this implementation is one in which the tree is perfectly balanced.

In this case it will take $O(\log(n))$ to traverse the tree to get to a leaf node.

Therefore:

$$W = O(\log(n)) + O(K) + O(L)$$

Where $O(K)$ is the computational complexity of updating the leaf node with the value associated with a new sample. Since this update requires in the worst case the creation of 6 new nodes we can say that:

$$O(K) = O(1)$$

$O(L)$ is the computational complexity of collapsing nodes after updating the regression tree. In the worst case scenario, this collapsing could be very costly computationally (we may need to collapse the whole tree). However this method does not require collapsing all the way. This method provides a constant amount of time after each update to try to collapse as many nodes as possible. This amount is completely configurable and can even be set to 0 if desired. Therefore:

$$O(L) = O(1)$$

Therefore:

$$W = O(\log(n))$$

Hence, the computational complexity for the proposed method (in the best case scenario) is:

$$O(n \log(n))$$

Worst case scenario:

The worst case scenario for this implementation is one in which the tree is completely unbalanced:

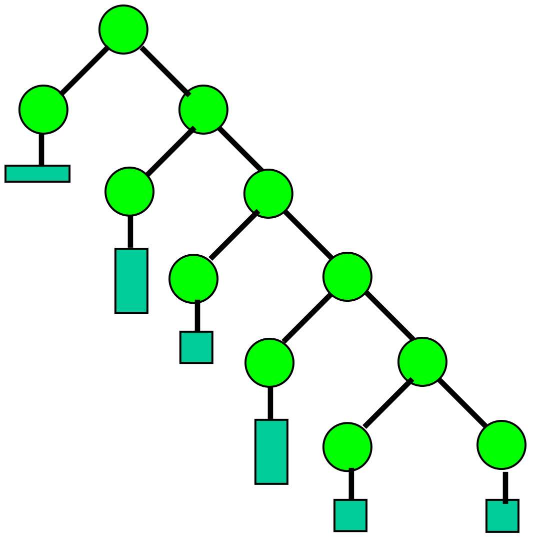


FIGURE A.9, Worst case scenario (completely unbalanced tree).

In this case it will take O(n) to traverse the tree to get to a leaf node.

Therefore:

$$W = O(n) + O(K) + O(L)$$

As we mentioned in the worse case scenario:

$$O(K) = O(1)$$

and

$$O(L) = O(1)$$

Therefore:

$$W = O(n)$$

Hence, the computational complexity for the proposed method (in the worst case scenario) is:

$$O(n^2)$$

A.4 Batch update of a regression tree

For instance, in the Sridharan-Tesauro method, the first step is to gather a set of learning experiences. This set of learning experiences is just a group of tuples of the form (*s*, *a*, *r*, *s'*), where *s'* is the new state encountered after performing action *a* while in state *s*. The value *r* is the immediate reward obtained after performing action *a* while in state *s*.

The next step is to generate a prototype (i.e., a (*s*, *a*, *v*) tuple) out of each learning experience (i.e., a set (*s*, *a*, *r*, *s*')). The *s* and *a* values in a prototype are taken directly from the *s* and *a* values of its associated learning experience. The *v* value of each prototype is initialized to the *r* value of its associated learning experience.

Given this set *N* of prototypes, a regression tree is constructed recursively as follows:

- First, this method calculates the average and the variance of the function values of the cases (i.e., the *v* part of the prototypes). If $|N|$ is less than a given threshold or the variance is sufficiently small, the algorithm terminates, returning a leaf node that approximates the function by a constant equal to the average value. Otherwise, the best axis-parallel split of the cases is found by examining all possible splits on each dimension of the state space.

- The best split is defined as follows: consider a split that divides *N* into sets *N1* and *N2*, with respective variances *v1* and *v2*. The split that minimizes the quantity |*N1*|*v1* + |*N2*|*v2* is the best split.

- A new internal node defined by this split is then created. The training set is separated into two subsets, and two sub-trees for the new node are created recursively.

- Finally, the algorithm returns the node and terminates.

The training cases in this method are taken from random positions in the state space, corresponding to a uniform random exploration of the state-action space.

After an initial regression tree is built, repeated sweeps are performed through the set of prototypes, in which the *v* value part of each prototype is adjusted according to:

$$v = v + \alpha[r + \gamma \max_{a'} Q(s', a') - v]$$

As described in section 2.2, α is the **discount rate** parameter and γ is the **step-size** parameter. This formula is practically the same to the one described in section 2.2. The only difference is that in this version Q(*s*, *a*) has been replaced by *v*.

Notice that the max-Q value is found using the regression tree generated during the last iteration. A new tree is built after each sweep through the set of prototypes. The algorithm terminates after a fixed number of sweeps.

REFERENCES

[1] Anderson, C. W. and Kretchmar, R. M. (1997). *Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning*. In Proceedings of the International Conference on Neural Networks, ICNN'97, Houston, TX.

[2] Balac, N., Gaines D. and Fisher D. (2000). *Using Regression Trees to Learn Action Models*. Proceedings of the IEEE Systems, Man, and Cybernetics Conference, Nashville.

[3] Barto, A. G. and Sutton, R. S. (1998). *Reinforcement learning: An introduction*. Cambridge, MA: MIT press.

[4] Barto, A. and Crites, R. (1996). *Improving Elevator Performance Using Reinforcement Learning*. In Advances in Neural Information Processing Systems 8 (NIPS8). D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo (Eds.), Cambridge, MA: MIT Press.

[5] Bellman, R. E. (1957a). *Dynamic Programming*. Princeton University Press, Princeton, NJ.

[6] Bellman, R. E. (1957b). *A Markov decision process*. Journal of Mathematical Mechanics.

[7] Breiman, L. et al. (1984). *Classification and Regression Trees*. Monterey CA: Wadsworth International Group.

[8] Dietterich, T. G. and Zhang, W. (1996). *High-performance job-shop scheduling with a time-delay TD(λ) network*. In D. S. Touretzky, M. C. Mozer, & M. E. Hasselmo (Eds.), Advances in neural information processing systems.

[9] Dietterich, T. G. and Wang, X. (1999). *Efficient Value Function Approximation Using Regression Trees*. Proceedings of: IJCAI-99 Workshop on Statistical Machine Learning for Large-Scale Optimization, Stockholm, Sweden.

[10] Dickerson, J. A. and Kosko, B. (1996). *Fuzzy function Approximation with Ellipsoidal Rules*. IEEE Transactions on Systems, Man, and Cybernetics, volume 26.

[11] Howe, A. E. and Pyeatt, L. D. (1998). *Decision Tree Function Approximation in Reinforcement Learning*. Technical Report CS-98-112, Colorado State University.

[12] Kaelbling, L.P., Littman, L.M. and Moore (1996), A.W., "*Reinforcement learning: a survey*," Journal of Artificial Intelligence Research, volume 4.

[13] Kanerva, P. (1988). *Sparse Distributed Memory*. Cambridge, MA: MIT press.

[14] Kephart, J. and Tesauro, G. J. (2002). *Pricing in agent economies using multi-agent Q-learning*. Autonomous Agents and Multi-Agent Systems, Volume 5, Number 3. Kluwer Academic Publishers.

[15] Lagoudakis, M. G. (1999). *Balancing and Control of a Freely-Swinging Pendulum Using a Model-Free Reinforcement Learning Algorithm*. Duke University.

[16] Lagoudakis, M. G. and Parr, R. (2001). *Model-Free Least-Squares Policy Iteration*. Proceedings of NIPS 2001: Neural Information Processing Systems: Natural and Synthetic. Vancouver, BC.

[17] Mitaim, S. and Kosko, B. (1998). *Neural Fuzzy Agents for Profile Learning and Adaptive Object Matching*. Presence, volume 7.

[18] Mitchell, T. M. (1997). *Machine Learning*. Boston, MA: WCB/McGraw-Hill.

[19] Schwartz, A. and Thrun, S. (1993). *Issues in Using Function Approximation for Reinforcement Learning*. Proceedings of the Fourth Connectionist Models Summer School Laurance Erlbaum Publisher, Hillsdale, NJ.

[20] Sridharan, M. and Tesauro, G. (2000). *Multi-agent Q-learning and Regression Trees for Automated Pricing Decisions*. ICML 2000: 927-934. Stanford University.

[21] Tesauro, G. (1995). *Temporal Difference Learning and TD_GAMMON*, Communications of the ACM, Volume 38, Number 3.

[22] Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Boston, MA: Kluwer Academic Publishers.

[23] Uther, W. T. B. (2002). *Tree Based Hierarchical Reinforcement Learning*. Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA.

[24] Watkins, C. (1989). *Learning from delayed rewards*. Ph.D. Thesis, Cambridge University, Cambridge, UK.