REINFORCEMENT LEARNING USING THE GAME OF SOCCER

By

Roger David Ford

B. Sc. (Computer Science) University of Lethbridge, 1992

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October, 1994

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

(Signature)

Department of Computer Science

The University of British Columbia
Vancouver, Canada

Date Oct. 14, 1994

Abstract

Trial and error learning methods are often ineffective when applied to robots. This is due to certain characteristics found in robotic domains such as large continuous state spaces, noisy sensors and faulty actuators. Learning algorithms work best with small discrete state spaces, discrete deterministic actions, and accurate identification of state. Since trial and error learning requires that an agent learn by trying actions under all possible situations, the large continuous state space is the most problematic of the above characteristics, causing the learning algorithm to become inefficient. There is rarely enough time to explicitly visit every state or enough memory to store the best action for every state.

This thesis explores methods for achieving reinforcement learning on large continuous state spaces, where actions are not discrete. This is done by creating abstract states, allowing one state to represent numerous similar states. This saves time since not every state in the abstract state needs to be visited and saves space since only one state needs to be stored.

The algorithm tested in this thesis learns which volumes of the state space are similar by recursively subdividing each volume with a KD-tree. Identifying if an abstract state should be split, which dimension should be split, and where that dimension should be split is done by collecting statistics on the previous effects of actions. Continuous actions are dealt with by giving actions inertia, so they can persist past state boundaries if it necessary.

Table of Contents

List of Tables

List of Figures

Acknowledgement

I would like to address my sincere appreciation to my supervisor Craig Boutilier, for putting up with me and showing me how to do research. Even if I didn't always follow his teaching.

I would also like to thank Michael Sahota for creating the reactive-deliberation controller on which much of the results of the thesis depend, and for aiding me in finding numerous problems in my code and how it interfaced with his controller.

Finally, thanks to the various people, and agencies that funded my research. The majority of the money provided came from a NSERC graduate scholarship.

Chapter 1

Introduction

1.1 Motivation

One of the goals of the Artificial Intelligence field is the creation of *intelligent agents.* An agent is an autonomous system that is able to sense its environment, act on that information and influence the environment. These agents may be physical like robots and factory controllers, or like schedulers they may be entirely software. An intelligent agent must do more than simply interact with its environment. It must act to cause the environment to reach desirable configurations or goals. The agent often has several goals that may be conflicting. Intelligent behavior often involves achieving these goals where the order and manner in which they are achieved is important.

Simple agents like a chess player can be programmed directly because the environment is well understood, unchanging with deterministic actions. The agent may not be able to predict exactly how the opponent will behave, but it expects actions to be completely observable and deterministic. If the agent moves its knight to square C3 then the knight will be on square C3 not some other square nearby. It is possible to give a complete and correct description of the environment and the task, from which the agent can plan its best actions. One reason why there are not more successful intelligent agents, like the chess playing ones, is that as the task and environment become more complex specifying complete and correct models becomes increasingly difficult. There are three situations where *learning* can be used to attack this complexity.

Most agents rigidly follow a set of preprogrammed responses. The programmer must program a response for all possible situations that the agent is likely to encounter. This is not possible for most environments. Usually the environment is so complex that it is *hard to program* for all possible situations, especially in nondeterministic environments where the number of possible situations could be very large. So the environment is constrained, as in the case of assembly line robots, to allow only a few possible situations. With the reduced complexity the programmer is able to explicitly tell the robot what to do. Telling a housekeeping robot how to wash dishes would be an nearly impossible task, if we had to specify torques and joint angles, and model every possible dish it might encounter. It would be nice if the robot could learn without having to be explicitly told what to do.

Even with the reduced environment the human programmer may not understand the environment well enough, or the *information may be unknown.* A programmer typically runs through a test-calibrate cycle several times before the agent has a reasonable performance. Here the programmer has learned about the interaction between the robot and the environment, and adapted its behaviour. Having a human do the learning works well because humans are very good at learning. However, its not always practical or possible to have a human tune the robot to its environment. Having to send out a specialist with every housekeeping robot, just to tune it to every different house, would not be profitable.

The third situation is when there is a *Dynamic environment.* Even if the agent had a good model of the environment initially it would soon become outdated. An agent in one of these environments would have to constantly update its model of the environment, for example, I might have to reprogram my robot to vacuum the rug, just because I moved the furniture. This could be better solved by having the robot "learn" the location of the furniture.

Robotic agents make good cases for studying learning algorithms because they often fall prey to all three of the above difficulties. In addition, in order for any learning system

work effectively in a robotic domain it must have several properties [19]:

- Fast convergence: Many learning systems require extended training data. Performing millions of training examples on a real robot is impractical, and it could be expensive, especially if the robot is continually breaking from driving into walls or off the edge of a cliff, from following a plan made before it has learned the appropriate behaviour

- Noise immunity: Robotic sensors are typically very noisy. They often give only approximations to the state information. Any learning system that requires *correct* training data would not be very effective.

- Incremental: It may be the case that the agent needs to actually perform its actions while learning. As in the case of a dynamic environment, it may not be possible to do a separate off-line learning phase.

- Tractable: Robots must keep pace with the environment, there is little time to sit and think before choosing the right action. If the agent sits idle for too long the environment will have changed and the plan is worthless. Any learning system must be able to perform the required computations in a reasonable amount of time.

- Grounded: The system should only depend upon information available from the sensors on the robot. While tabula rasa learning is impossible, the robot might not be able to depend upon a human to give it the necessary information. The human might not known the information, or the information may be incorrect.

Robotic systems make excellent test cases for evaluating learning systems. One criticism of learning systems is that they are only tested on toy examples. Learning on a robot forces the system to deal with very large sized domains, that are often not well-behaved

or well-understood. In short if it performs well on a robotic system then it should be able learn in a lot of other environments that are better behaved and better understood. Whether the robot benefits from learning depends upon if the cost of doing the learning out-weighs the cost of programming the robot directly.

1.2 Goals and Objectives

The main goal of this thesis is the development of a reinforcement learning algorithm [29, 32, 17] that is capable of learning on very large continuous state-spaces, with the characteristics needed by a robot, that is, fast convergence, noise immunity, incrementality, and real-time operation. Current reinforcement algorithms only work on discrete states with discrete actions. They also suffer the "curse of dimensionality": while learning can be polynomial in the number of states, the number of states tends to grow exponentially [31]. Overcoming the problem of too many inputs is necessary before learning can be applied to more than just simple problems.

The solution to this problem outlined in this thesis is to automatically divide up the state space into abstract states. By testing the ability of different learning paradigms to learn to play soccer, an objective measure of the effectiveness of each paradigm in a complex domain can be achieved. This thesis develops experiments for testing different learning strategies. In comparing the performance of each learning strategy against an independently developed, non-adaptive agent an objective empirical measure of the strength and weakness of different algorithms can be determined. Creating a winning soccer player is not an essential goal.

1.3 Soccer

The game of soccer was chosen as an environment for testing learning agents because, as an abstraction of the world, it simplifies the problem but keeps the essential essence of the problem. In particular it preserves the aspects of the world corresponding to the three situations listed above. Soccer contains knowledge that is hard to program, knowledge that is incomplete, a dynamic changing environment. In addition it contains hostile agents and provides an *objective measure of performance* [26].

The world is not completely predictable, and much of the information needed to do classical planning is not known. This state of affairs should be preserved in any abstraction used to test AI theories. Soccer preserves unpredictability and unknowability. It is not possible in practice to predict where the ball will go, where the opponent is going, or where team mates will be. The information needed to create the optimal soccer playing agent is not known. True, experts exist, but they are unable, or maybe just unwilling, to divulge this information. It is also not clear that their experience is transferable to a robot, whose sensors, actuators, and dynamics are different.

Even though an agent cannot be compared to an optimal agent, the standard way of judging correctness, they can be judged as being relatively more correct than another agent. An agent that is able to score goals, prevent goals from being scored, and have an overall higher score, can be said to be more correct than another agent without such abilities. Such an objective measure is essential for testing learning agents.

1.4 Outline

Chapter 2 describes the reinforcement learning problem, and some of the other work done in an attempt to solve the input generalization problem.

Chapter 3 describes the *Dynamite testbed* developed at the University of British

Columbia. An outline of the *Reactive Deliberation* controller, developed by Michael Sahota[25, 26], is given, as well as a description of how it is adapted to perform learning.

Chapter 4 details the different learning algorithms tested, the results and their significance. The basic learning algorithms tested are all based on Q-learning [32]. The only differences between them is how they divide up the state-space.

Chapter 2

Background

2.1 Markov Decision Processes

Classical planning [2] works in a well structured deterministic domain. When actions begin to have non-deterministic effects, sensors only give a probability of being in a state, or when the environment behaves non-deterministically, classical planners run into major problems. The planner must plan for all the effects of the actions, not just the expected or desired ones. This causes the search tree to expand rapidly and a planner quickly runs into computational difficulty. Usually the plan is generated assuming a deterministic world, and if an action "fails" a new plan is generated. The non-deterministic world can be more adequately modeled by a *Markov Decision Process* [5, 13]. A Markov Decision Process (MDP) consists of four parts:

- A set of states S. The state space of the world. Usually this is thought of as a finite discrete set, but the model allows continuous state spaces as well.

- A set of actions $A(s_i)$. This is a function that gives all the possible actions for each state s_i.

- A transition function $P(s_i, a_i, s_j)$. Actions in a MDP are not assumed to be deterministic. The function $P(s_i, a_i, s_j)$ gives the probability that performing action a_i while in state s_i will change the world into state s_j. As expected, the sum of all the transitions from a single state should be equal to one. That is for any state

$s_i \in S$,and each action $a \in A(s_i)$,

$$\sum_{s_j \in S} P(s_i, a, s_j) = 1$$

.

- A reward function $R(s_i)$ This describes the immediate value of the agent being in that state. The simplest function is to set to $R = 1$ at the goal states and 0 everywhere else.

The one essential property of MDPs is that deciding the best action to perform depends only on the current state. The current state of the process contains all the information necessary to make a prediction about the usefulness of executing a given action. Knowing the past history, states and actions, does not give any more useful information. This means that regardless of how the state was reached the actions at the current state still have the same transition probabilities. That is the state space captures all the essential properties for predicting rewards and transitions. This is known as the *Markov property.* Any system can be modeled as a Markov process if enough details are provided.

2.2 Dynamic Programming

Deciding on the best action to do next with a MDP is just a matter of maximizing rewards. The agent must not only pick the action that maximizes the expected immediate reward, but also maximize the potential for rewards in the future. Picking an action that gives a high immediate reward may not be the best thing to do, if in the next state the agent can only pick actions that give negative rewards. A *policy* π is a function, or universal plan, $\pi(s_i)$, that specifies the action to perform for each state s_i.

The expected value of the state s, given a fixed policy π, is denoted $V_\pi(s)$, and depends not only on the reward at that state but the expected rewards received by following the policy from that state. Future rewards are discounted by a factor, γ, a constant between 0 and 1. This forces the agent to pick the actions that reach the goals as quickly as is possible, since, rewards received further in the future are worth less to the decision maker at the immediate state. The value function then is written as the immediate reward added to the discounted expected value of following the policy. Equation 2.1 gives a system of $|S|$ linear equations that can be solved to give the value of any policy at any state.

$$V_\pi(s_i) = R(s_i) + \gamma \sum_{s \in S} P(s_i, \pi(s_i), s) V_\pi(s) \tag{2.1}$$

The decision of what to do at each state is equivalent to finding the *Optimal policy*, $\pi^\star$ and following it. An optimal policy is a policy that if followed will produce a higher expected value than any other policy for any state. Let s_0 be the initial state of the agent. Then the value of the optimal policy $V_{\pi^\star}(s_0)$ must be greater then or equal to the expected value of all possible policies, starting from s_0. This will hold regardless of which $s_0 \in S$ is chosen as the initial state. As long as the agents start in the same state, the one following policy $\pi^\star$ is expected to do as least as well as the agent following any other policy. There is always an optimal policy for a MDP.

Given the transition function and the reward function it is possible to find the optimal policy for the system. Finding the optimal policy by trying all policies is very inefficient. Finding the decisions that could improve the policy is not simple. For example, in chess the actions that give a high immediate reward are not necessarily the best. Similarly, actions that may get low immediate rewards can lead to good positions. It is possible also to get into a doomed state, where even the best decision leads to failure. Blaming the

actions just before failure would be wrong when the decision that needs to be changed is somewhere further back in the chain. The process of finding which actions to improve when the actions interact is called *credit assignment.* Dynamic Programming finds the decisions that need to be changed by an incremental procedure, building successively closer approximations to the optimal policy. These methods work well but they need to be given a model, the reward function and the transition function. These methods become too time consuming for large state spaces. Recent work has addressed the idea of how to shrinking the state space until the computation becomes reasonable [12, 7].

2.2.1 Value iteration

The value iteration algorithm proposed by Bellman[5] (see table 2.1), does not actually solve the equations for the value of a policy, thus avoids the time consuming step. The idea is intuitively simple, making successive approximations to the optimal policy, and the value function, until the difference between one approximation and the next is close to zero. Initially, the value of each state, call it V^0 is set to 0. In practice the initial values are usually set to the immediate reward. This is the value of the state when following the optimal policy for 0 steps. Then a closer approximation of the Value function is created from the last approximation. For each state the action that maximizes the expected immediate reward plus the old estimate of the expected value of the state reached is found. Since, V^0 was set to the immediate reward, V^1 becomes the value of performing only one action in that state, and then stopping. This is done repeatedly,(step 3(b)), the V^n value of the state being calculated from the V^{n-1}. In general V^n is the value of the state for following the optimal policy if the agent was only going to perform n actions. So as $n \rightarrow \infty$, $|V^n - V_{\pi^\star}| \rightarrow 0$. Value iteration suffers from the problem, of not knowing when it has reached the optimal policy. Clearly as $n \rightarrow \infty$ the policy approaches optimal. The algorithm itself, however does not continue on indefinitely, but terminates

1. Initialize V^0 to the immediate reward values.
2. i:=0
3. repeat
   (a) i:=i+1
   (b) $\forall s \in S$ do

$$V^i(s) := \max_{a \in A(x)} \left\{ R(s,a) + \gamma \sum_{s' \in S} P(s,a,s') V^{i-1}(s') \right\}$$

   (c) $\pi(s)$:= the action a from previous step.

   until $(v^i - v^{i-1}) = \epsilon$

Table 2.1: The Value Iteration Algorithm

when the improvement between successive iterations is small. It is not known in advance how many iterations this will take; for this reason policy iteration, which is known to converge in a fixed number of steps, is usually the preferred approach.

2.2.2 Policy Iteration

The policy Iteration Algorithm [13], does solve the system of $|S|$ equations generating the value function for the sub-optimal policy. Unlike value iteration, which successively estimates the value of the optimal policy, policy iteration works on improving a sub-optimal policy. This is done by identifying states that can be improved, and quits when the policy can no longer be improved. A search is done through the states, finding states s where there is an action a which if taken for just this one state, and then following π results in a higher expected return than just following $\pi(s)$. After finding such a state and action the policy is changed so that $\pi(s)$ now selects the action a. This results in a new policy that must have a higher value than the old policy. When no more such states

1. Let π' be any policy on S
2. while $\pi \neq \pi'$ do
   (a) $\pi := \pi'$
   (b) Calculate $V_\pi(s)\ \forall s \in S$
   (c) $\forall s \in S$ do
   if $\exists a \in A$ s.t.
   $$\left(R(s) + \gamma \sum_{s' \in S} P(s, a, s') V_\pi(s')\right) > V_\pi(s)$$
   then $\pi'(s) := a$ else $\pi'(s) := \pi(s)$
3. return π

Table 2.2: The Policy Iteration Algorithm

and actions can be found, then the optimal policy has been found. This algorithm, unlike value iteration, is guaranteed to converge in a finite number of iterations, in practice it is often polynomial in $|S|$.

2.3 Reinforcement learning

Both value iteration and policy iteration require that the transition function, $P(s_1, a, s_2)$, and the reward function $R(s)$, be known. For many real problems these functions may be unknown. The only alternative then is to try actions and observe the results. One can then either try to learn the transition function and the reward function and solve for the policy using one of the above methods, or one can attempt to learn the policy directly from the observations. Learning the transition probabilities would require a table of about size $|S|^2|A|$, while learning the policy directly would only require order $|S||A|$ space. Learning the transitions also has the problem that the agent is not able to incrementally use the information it has learned. The dynamic programming methods

are too computationally intensive to perform after each observation. This means that the agent is required to follow the old policy until such time as the policy can be updated to incorporate the new observations, usually after a goal state is reached.

Reinforcement learning is learning from rewards and penalties. This is learning from trial and error, as opposed to supervised learning. In supervised learning the agent is given pairs of items, an input vector and an output vector. After training the agent is supposed to produce the correct output vector, when given an input vector. The obvious problem with this learning by example is that the agent inherits the bias of the teacher, and this may be bad if the teacher's bias is incorrect. The main advantage of learning by example is its rapid convergence, they give good approximations with a lot less training than self-guided learners. Since, reinforcement learning is self-guiding and able to learn almost tabula rasa it can be considered a more general form of learning. It still requires a teacher, or at least a means of identifying which situations are "good" and which situations are "bad". This is a much weaker requirement on the teacher than the one required by example learning systems. Its conceptually easier to tell when something is right or wrong then it is to show how to do the task.

Learning methods where nothing is predefined are called *strong learning* methods. These methods may be either taught by a teacher or self-guided. Like most methods that are capable of learning everything, strong learning methods are very slow to learn anything. *Weak learning* methods require some prior knowledge and are able to learn faster because of this background information. Weak learning methods suffer from the fact that they must have background information, and they must accept the background information as absolutely correct. The majority of weak learning methods are unable to determine if the domain knowledge they started with is reliable or not. Clearly what is needed is a general strong learning algorithm that is able to accept existing domain knowledge, and override such knowledge if it sees fit.

2.3.1 Temporal Credit Assignment

The field of reinforcement learning is mostly concerned with learning controllers for Markov decision processes when a model is not available. This involves two separate *credit assignment problems.* The first is the *temporal credit assignment*, and the second is the *structural credit assignment. Temporal Credit assignment* is the process of assigning responsibility for a good or a bad outcome to the sequence of decisions observed, that is, identifying which decisions, in the sequence that lead to a reward, were primarily responsible for that reward [29, 32, 19]. *Structural credit assignment* assigns values to decisions at a state, a state that may not have been observed, such that similar states have would have similar values for a given decision. Early work done in this area includes Samuel's checker player [27] and the BOXES algorithm [21]. The dynamic programming methods above solve both credit assignment problems, since DP has a model, it can adjust the values of every state and decision with out having to observe it. Although DP suffers from the curse of dimensionality, the problem is not as bad as that experienced by learning systems, since they have to actually visit each state.

One of the earliest learning systems to solve the temporal credit assignment problem was Samuel's checker player. There are two basic methods for solving the temporal credit assignment problem. The first, and most obvious, is to save every decision and state until a reward is received. Then the values of all the decisions are adjusted according to some variant of Equation 2.1. The second, and the one used by Samuel, is the method of Temporal Differences (TD) [29]. Samuel's checker player was to learn an evaluation function for board positions in a checker game. Although a straightforward simple model for checkers exists, the size of the state space makes solving for the optimal policy difficult. The evaluation function gives a heuristic that tells how good the position is and can be used to constrain a search. Samuel's learning method was to apply the current evaluation

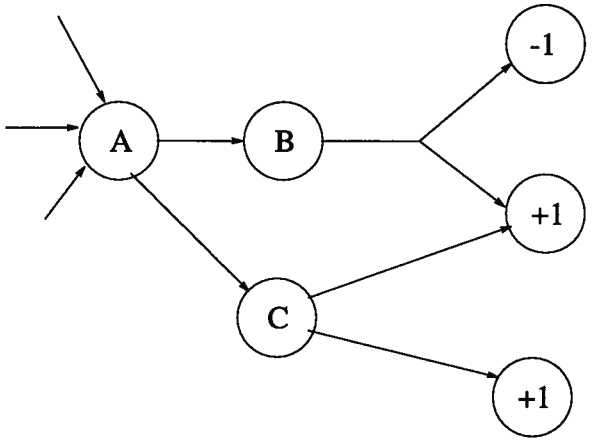


Figure 2.1: A MDP that cause problems for a 1-shot update

function to the current state, perform a move, apply the evaluation function to the new state, then use the difference between the two values to adjust the evaluation of the first state. When a sequence of moves lead to a reward, the state just before will have its value increased. Each successive time through the sequence the increase will back up to the earlier states (see figure 2.1).

The method of Temporal differences, although used frequently earlier, was formalized by Sutton in 1987 [29]. Most importantly he developed a convergence theorem showing that TD learning methods asymptotically converge to the correct values. Also it was shown that TD methods are able to converge to optimal values with finite training. Here optimal, refers to the best values that can be determined from the training, not optimal in the sense of a dynamic programming solution. This idea of optimal is the *maximum-likelihood estimate.* Take, for example, tossing a coin. If the coin was tossed ten times and heads turned up seven, then the prediction of getting a head on the next toss based on a model of a fair coin would be 0.5, but asked to estimate based solely on the observations it would be $\frac{7}{10}$. From the observations the maximum-likelihood estimate is in a sense the optimal prediction. Sutton proved that the TD methods, with judiciously chosen parameters, will converge to the same predictions as a method that remembers all state

1. Initialize Q-values.
2. While true do
   (a) Find current state of world, s
   (b) find $a = max_a Q(s, a)$, the current "policy" action
   (c) Decide whether to follow policy or to explore. Perform either a or a random action. Call the action actually performed a'.
   (d) Get the new state, s_1, and the reward $R(s_1)$
   (e) Update Q-values with the formula

$$Q(s, a') = Q(s, a') + \beta \left(R(s_1) + \gamma max_a Q(s_1, a) - Q(s, a') \right)$$

Table 2.3: The Q-learning Algorithm

transitions that occurred and their outcomes which means that TD learning converges to the maximum-likehood estimate.

Watkins [32] applies Suttons TD learning methods to perform incremental dynamic programming. Watkins's algorithm, called *Q-learning* because of the notation used in his thesis, learns the optimal policy and is incrementally updated, generating the improved policy after every step. The temporal difference methods, as used by Samuel and Sutton above, learned an evaluation function. It was still necessary to store the actual policy as an additional data structure which needs to be recomputed whenever the value function changes. Another method, the one used by Samuel's checker player, uses the value function to control a search. The agent searches through all possible actions at a state and picks the action that leads to the highest valued state. Where the value of the states is determined using the value function that was learned. Q-learning learns the policy and the evaluation function at the same time using a single data structure. A look-up table is created with one entry in the table for each state-action pair. The table entry $Q(s, a)$ is the current estimate of the value of performing action a while in state s. The value of

the state is the value of the best action in that state. So the table of Q-values is able to store the policy, the value functions, and the value of doing non-policy actions. Deciding what is the best action to perform is done just by finding the action with the maximum value for that state.

Q-learning (table 2.3) is a form of value-iteration describe in section 2.2.1. The important fact to notice is that value-iteration does not require that each V^i be updated in a systematic way, i.e.. $V^1, V^2, \cdots$. The method still converges, although perhaps not as quickly, if the value of every state is updated in a arbitrary order, so long as each state is updated often [32]. The value of the entry $Q(s,a)$ is defined as the value of the immediate reward for performing action a plus the (discounted) expected value of the state reached. where the expected value of any state is just the value of that states best action. Letting s_1 be the state reached by performing action a in state s, the expected value of performing action a then is:

$$Q(s,a) = r(s_1) + \gamma \max_{a_i \in A} Q(s_1, a_i)$$

This of course requires that the values of s and a be remembered. So that after performing the action, the new state s_1 can be observed and the table values updated. Following the method of temporal differences, the values for the Q-table are updated by the rule:

$$Q(s,a) \leftarrow Q(s,a) + \beta((r(s_1) + \gamma \max_{a_i \in A} Q(s_1, a_i)) - Q(s,a))$$

The new Q-value is the error between the earlier estimate and the observation, multiplied by a learning rate parameter β, used to control the convergence rate. β is set in the range $(0, 1]$; by setting β to 1, the slope given by the error term is descended to the minimum, and the new Q-value is updated to treat the new observation as correct. This one-shot updating of the Q-values causes problems when performing an action sometimes results in a positive reward and sometimes results in a negative reward. For example,

take the situation shown in figure 2.1. State B allows only one action, this action however, leads to either a negative reward or a positive reward with equal probability. If the first time the action is performed a positive reward is received, the value of that action is given a value of about (1), making the expected value of B appear to any state that can reach it, like A, appear to be higher then is should. If the next time the action is tried the negative reward is received, then the value of the action is moved all the way to around (-1). With β set as one, the value of such a non-deterministic action, will never have the more appropriate value of $\frac{1}{2}$, that using transition probabilities would have given. Setting β to a value less then one causes the Q-values change slower, and preserves more of the past observations.

Q-learning has several beneficial properties. It does not require an initial model or *a priori* domain knowledge, which as explained in chapter 1, is not always available. All Q-learning requires is a reinforcement function. Q-learning, also, does not require that entire sequence of states be saved; like all TD methods only one observation at a time needs to be stored. Finally, Q-learning is incremental, there is no need for a separate learning phase. This requires some method of determining if the current, possibly sub-optimal, policy should be followed, or some other action that might improve the policy should be performed (Exploitation or Exploration). This is usually solved by some stochastic method.

The disadvantage of Q-learning is that it is slow to converge. Q-learning solves the structural credit assignment the same as dynamic programming does, by actually visiting each state. Since the model of state transitions is not known, it is not possible to know which actions will lead to unexplored states. Q-learning relies on the fact, that by occasionally choosing random actions, eventually the entire state space will be explored. This is even more of a problem with robotic systems, since, they tend to have extremely large, usually continuous, state spaces. Also, it is not possible to do the thousands of

trials needed for Q-learning to converge on most robotic systems.

2.3.2 Structural Credit Assignment

The structural credit assignment problem is the problem of assigning a value to states and actions that haven't been visited, based upon some measure of similarity between states. This is also called the input generalization problem. Dynamic programming methods, as well as standard Q-learning, don't solve the structural credit assignment problem: they must either solve for, or visit, every state. The problem is less acute for dynamic programming since solving a system of equations takes less time, then random exploration. However, in robotic domains the state spaces are extremely large, often more than 2^{100} states [10]. With state spaces this large even a polynomial time algorithm would take too long. The major difficulty in assigning structural credit is knowing which unvisited states are similar to states that have been experienced and for which an informed decision can be made.

The BOXES algorithm[21] used the most obvious method of assigning structural credit. Boxes's purpose was to learn the classic problem of balancing a pole. The problem is to balance a pole on top of a car. A negative reward is given if the pole falls over. In the simplest case the car has two possible actions: move forward or move back. The state space is defined by four continuous variables: the position of the car x, the angle θ of the pole from the vertical, and the rate of change of these two variables over time, $dx, d\theta$. The boxes algorithm divides up each of the dimensions into fixed sized intervals, arbitrarily chosen by the programmer. With four dimensions this discretises the state space into boxes (thus the name). Every value that falls within the boundaries of the box is treated as a member of one state. The obvious problem is knowing beforehand how many boxes are needed, and where the boundaries should be.

Dividing up the space uniformly along each dimension results in the hypercubes,

or the boxes used above (see figure 2.2). When the actual Q-values are not uniformly distributed through the space, then many of the boxes will be nearly empty; not only is this a waste of space but the number of states can grow exponentially. Certain areas of state space need to be divided up finely to correctly partition the spaces where Q-values are tightly grouped, but areas where Q-values are sparse should not be divided up as fine. The adaptive grid (figure 2.2) is more flexible then the regular grid; rather than uniformly partitioning each dimension, the location of the boundaries of boxes can vary along each dimension. Adaptive grids still suffer from the "curse of dimensionality". Each partition is global, in effect partitioning every state along its path. Recursive partitioning schemes, (Figure 2.2 recursive grids, KD-trees, Arbitrary Hyperplanes), don't have the global partitioning problem, further partitions only effect the subspace in which they are applied.

The other common way to assign values to unexplored states is to use Neural networks. The backpropagation algorithm[24] is an extension of the percepetron[22], that allows multilevel networks with hidden nodes to be trained. Backpropagation uses gradient descent to adjust the weights of connections in the network. A supervised learning method, the network is presented with a training (input,output) pair. The current net is applied to the input vector, and the result vector is compared to the correct output. The error between the output the net generated and the actual output is used to adjust the weights. The adjustments are propagated backwards through the net from output nodes to input nodes. Backpropagation is primarily orientated to solving the structural credit problem. Each node can be considered as performing a test if the weighted sum of its inputs is greater than a threshold. Essentially this is just a partition of the input space by a hyperplane, with multilevel networks allowing recursive partitioning. Backpropagation suffers from the lack of a good theory explaining when it should work, how many nodes are needed, and how they should be arranged. Lin[17] and others have had success

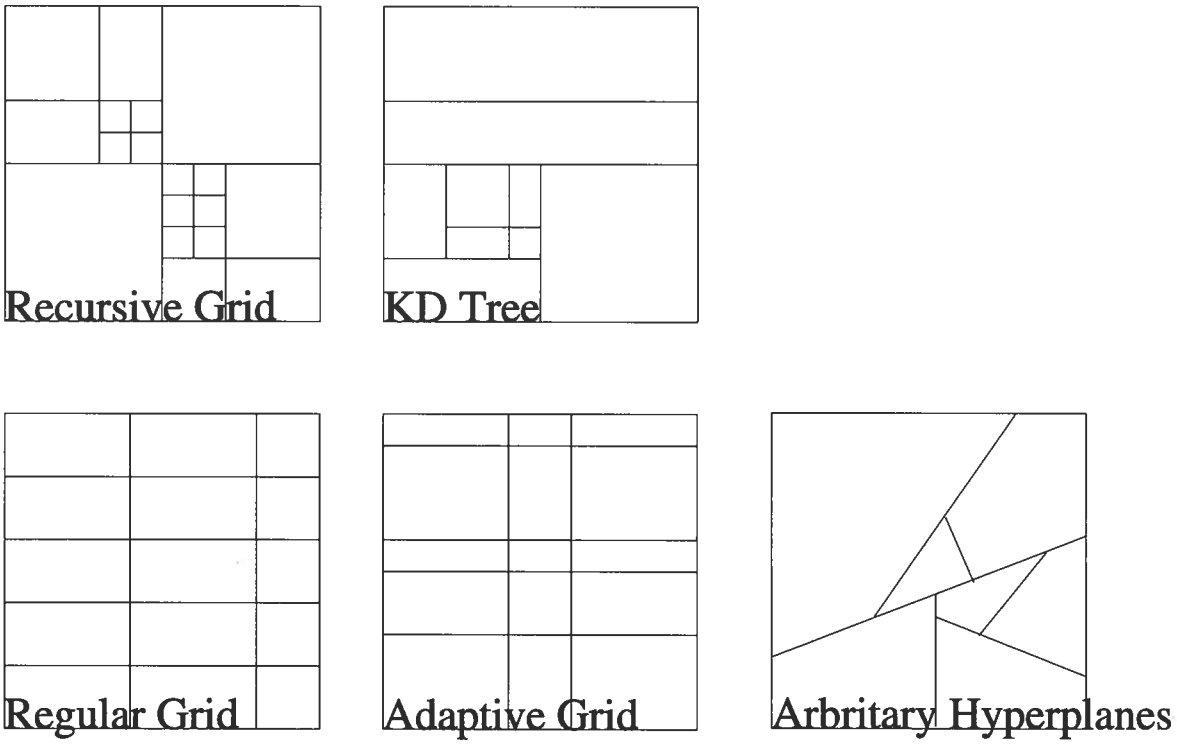


Figure 2.2: Different ways to divide up a 2D space

coupling NN and Q-learning, still others like Chapman and Kaelbling [10] have had poor results.

Chapman and Kaelbling [10] developed another type of generalization algorithm, called the G algorithm. The G algorithm grows a tree, the internal nodes which test selected input bits and follow either the left or right branches of the tree according to the results. The leaves of the tree each contain a single Q-value. This Q-value represents the Q-values for all the states that have the same value for the bits tested in the internal nodes. By testing only the most important bits the leaves of the tree can represent a large number of states, states that have similar Q-values, with a single Q-value. This saves both space and time (see figure 2.3).

Initially the algorithm considers all the bits to be irrelevant, clustering the entire state space into one state. Statistical evidence is collected for the relevance of bits. When a bit is discovered, by the t-test, to be significant the node is split into two leaves, one for when the bit is off and one for when the bit is on. Then more statistics are gathered within the leaves, which themselves can be split if necessary. The G algorithm has one major drawback, that is the testing is done at the bit level. If the sensor information is not presented such that it makes sense to test individual bits, then the method will not be efficient. Consider again the pole balancing problem, here the sensors give four real values. Testing real numbers a single bit at a time would give a rather large tree, larger then might really be necessary. Also it needs the bits to be individually relevant.

The other alternative, to clustering based upon dividing up the state space into subspaces, is to cluster based upon similarity in the states response vectors. This enables states, that might not be nearby in the state space, to be clustered together simply because they produce "similar" responses. One such method used by Mahadevan and Connell [19] clusters states together solely on the basis of a similarity metric. Each observation is compared to all the clusters and if it matches a cluster it is added to it. If

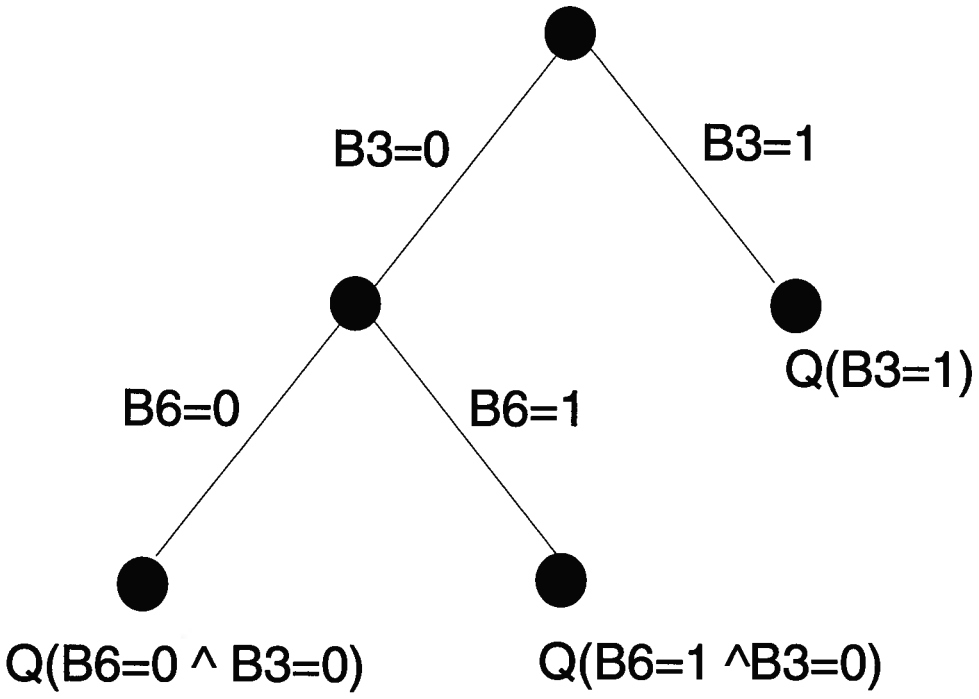


Figure 2.3: A G tree that has split the space into 3 clusters.

it does not match any clusters a new cluster is formed with only one member. A cluster description is a vector of probabilities, $< p_1, p_2, ...p_n >$ where each p_i is the probability that the i_th bit is on given that a state matches the cluster. From this, the probability that a state matches a cluster can be found, if this probability is greater then a threshold, ϵ then the state is said to match the cluster. Clusters can be merged together if the "distance" between clusters is less then the threshold ρ and the Q-values of the cluster agree in sign and the difference in magnitude is less then the threshold δ. The best action is selected by maximizing the combination of the probability of matching the cluster, and that clusters Q-value.

All methods that solve structural credit assignment work by defining a similarity metric across the state space defined by the sensors. This similarity metric is used to interpolate between known states and states that have not been visited yet. The hope is that given a good metric the entire space will not have to be explored, and the learning method will converge to an adequate policy in a reasonable amount of time. Methods are usually based on some variation on dividing up the state space or maximizing some measure of similarity of points in a cluster.

Chapter 3

The Dynamite Testbed

The Dynamite testbed is a platform for testing theories about controlling mobile robots in dynamic domains, in particular mobile robots in the soccer domain. This provides a common task for comparison of different control strategies. It consists of several radio controlled cars that share a common vision system, and play a version of soccer on a dining table sized field.

3.1 System Overview

The Dynamite testbed, see figure 3.4, consists of a small playing field, about 224 cm long and 122 cm wide.Unlike a real soccer field it has boards around the outside to prevent the ball from going out of bounds. The field provides a playing surface for multiple soccer playing agents. The agents are made from off the shelf remote controlled cars retrofitted to work with the vision system. Each car is fitted with two circular colour markers that allows the vision system to determine position and orientation of the car. Each car also has small bumpers enabling them to push the ball, since cars cannot kick.

The vision system consists of a single overhead camera that transmits full colour video to special purpose dataflow computer, named the *DataCube*. The Datacube is able to perform rapid filtering of colours, classifying pixels into colour groups, or blobs. The colour information is then given to some transputer nodes for additional processing. The transputer nodes are able to output the world coordinates for the center of each colour blob. With the two circles on each car, the vision system is able to produce real-valued

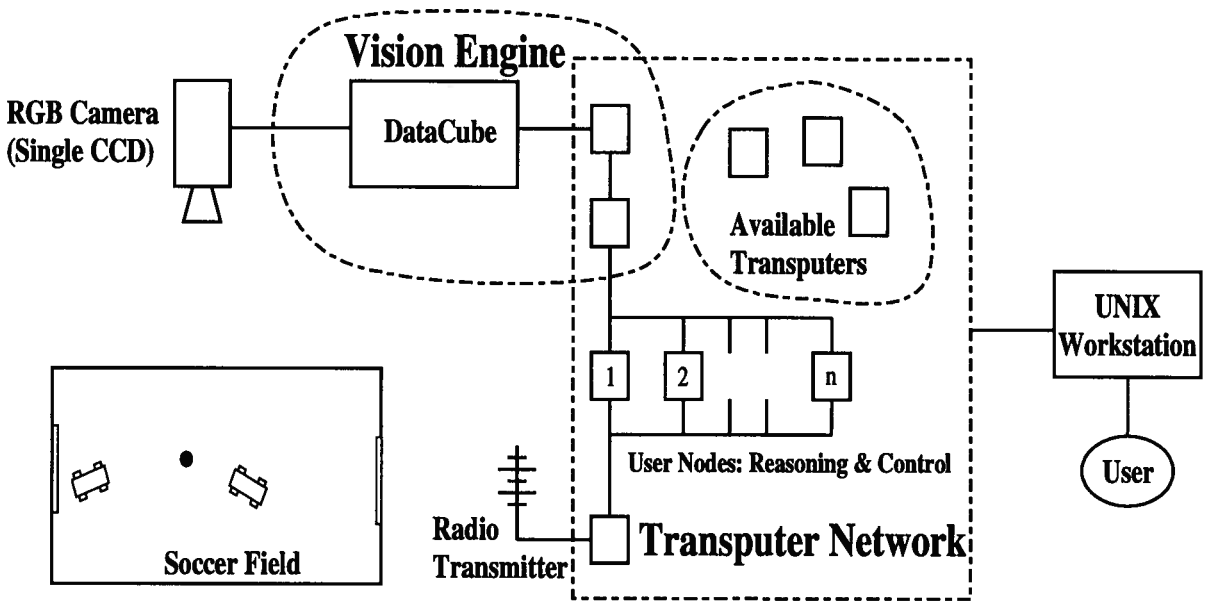


Figure 3.4: The dynamite soccer playing testbed

coordinates and orientation for each car, and the ball, sixty times a second. The accuracy of the values are within millimeters of the real position. The agents operate on the remote brain concept: all sensing, reasoning and control is done on a remote computer. The cars can receive commands from a transputer controlled radio transmitter, fifty times a second. The entire team, for both sides, can be controlled on the same set of transputers, since each of the n transputers nodes in figure 3.4 can control an agent independent of the rest. Complete details of the system are given in [25, 26, 3, 4].

3.2 The Reactive Deliberation Controller

The Reactive Deliberation Controller is a robot control architecture proposed by Michael Sahota [25]. It consists of two basic parts, the *Executor* and the *Deliberator* (see figure 3.5). The main idea is that the two halves run asynchronously – in this case they each get one transputer node. This allows the executor to keep up with the environment, always

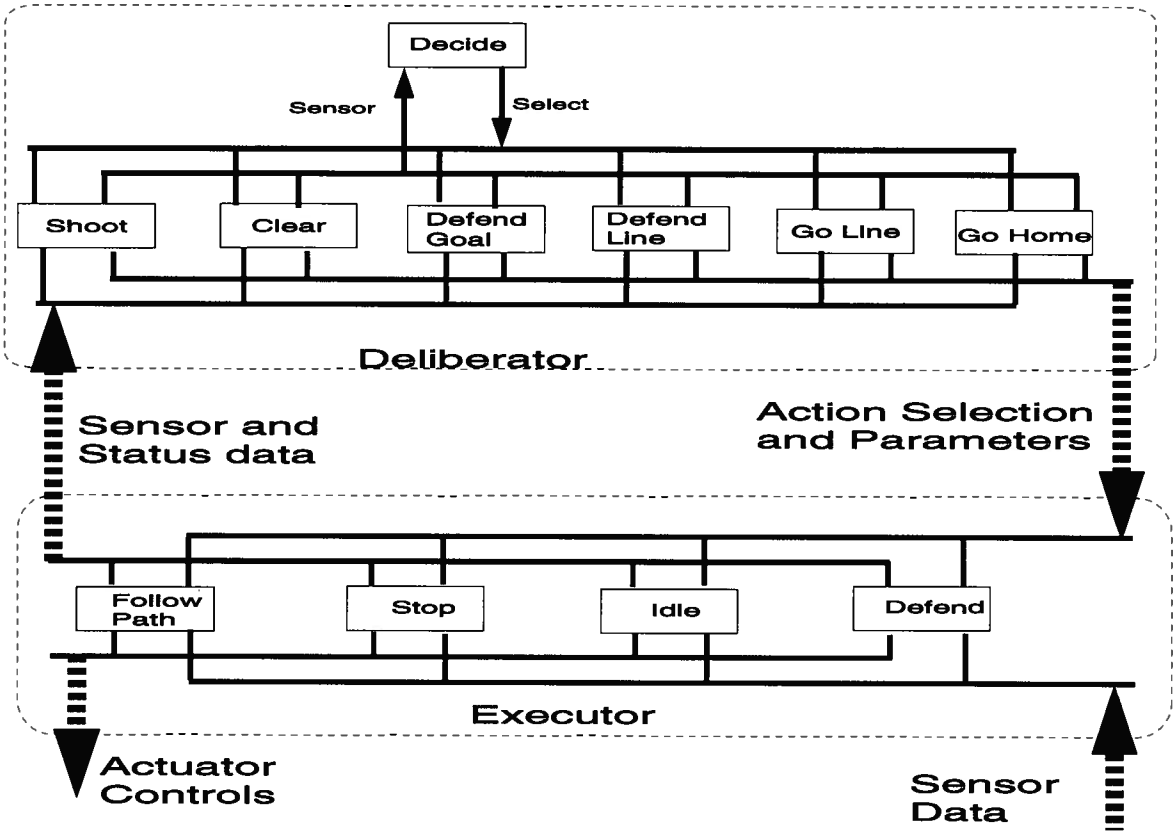


Figure 3.5: The controller

monitoring the environment and sending control signals. The deliberator however, is able to become more involved in deciding what to do.

3.2.1 Executor

The lowest level of the controller's hierarchy is the Executor. It consist of behaviors that can be executed rapidly, with the sensor to actuator loop being as short as possible. Lengthy computations cannot be performed at this level. These behaviors, which Sahota calls action schemas, perform the low-level control of the robot, receiving raw sensor data, and action selection with parameters then sending out low-level controls to the actuators. It also sends filtered sensor and status data to the next higher level. For example, one of the behaviours from the soccer agent is "follow path." *Follow path* is activated and given a path to follow by the deliberator. The behaviour then compares the world coordinates from the sensors to the path data and sends directional controls to the robot. The higher levels of the controller can actually have crashed and the robot won't notice until the behaviour is finished.

3.2.2 Deliberator

The Deliberator is similar in structure to the executor. The main difference is that it is allowed to spend more time between actions, thus the name, and it cannot sense or control the environment directly. The Deliberator is forced to receive all its sensor information from the executor. This could be simply the raw sensor data passed through, or some more complex filtering could be done. For the soccer playing robot, the sensor data is filtered to provide accelerations and status of the executor. Each behaviour of the deliberator uses the sensor information given by the executor to compute which of the executor actions should be selected. It then computes the run time parameters for that action, and produces a bid based upon the results of the planning and how appropriate

that action is thought to be. The behaviour with the highest bid then gets to select the action that gains control of the executor. Unlike the executor, each behaviour is running in parallel with the others. However, only the behaviour that produces the highest bid get control of the executor. If one of the other behaviours, one not currently in control, produces a bid greater then the current bid, then the active behaviour is interrupted and the new behaviour is put in control.

3.2.3 Adapting Reactive Deliberation to do learning

Adaption of the Reactive Deliberation Controller such that it is able to learn requires that the voting system of the Deliberator be removed and arbitration be performed by a higher level. Originally the arbitration is very simple, simply a comparison loop to find the highest bid then select that behaviour. This simple selection will be changed. No longer will bids be sent to the next level, rather the full sensor data will be sent. This sensor data defines the state space, the arbitrator then performs Q-learning. The set of actions for the Q-learning algorithms is the set of behaviours of the deliberator. Q-learning requires that the states and actions be discrete. Both of these conditions are violated with this architecture, for the state space is continuous and the actions persist through state boundaries. The continuous state space can be divided into discrete spaces, but that cannot be done with the actions. Actions are given an *inertia*, so that when an action crosses a state boundary, the executor is allowed the chance to continue the action it had started earlier.

In the spirit of behavioral control only one behavior per level is allowed to send actuator control commands at one time. Unlike subsumption[9], the decision of which behaviour is in control is not fixed. Instead, the decision is made at the next higher level of the controller. This scheme allows greater flexibility, since unlike subsumption, the priority between behaviours is not hardwired at compile time.

Chapter 4

Experiments: Learning to play Soccer

The game of soccer that the agent is required to learn is a simplified version of the real game. The major simplification is in the number of players: in this version there are only two players. The state of each object in the game is given by the vector $< x, y, \theta, v >$. This vector is made up out of the position of the object on two dimensional field x, y, the orientation of that object θ, and the object's velocity v. The ball's orientation is found by tracking it over time, unlike the cars the ball's orientation corresponds to the direction its moving, not which way it's pointing. In the one-on-one version of soccer played here, there are only three objects on the field at one time, this gives a total of twelve dimensions for the agent to partition and explore. It would be easy to create more input dimensions (like acceleration), but the twelve given here would seem to be the reasonable dimensions for the task.

Each agent was trained on a simulator, where a large number of games can be played easier then they can on the real robots. The agent receives a +1 reward for scoring a goal, and a -1 reward when the opponent scores a goal. After training the agents were tested in pure exploitation mode with no learning. In other words they are tested following the policy that was learned, both in simulation and on the real robots.

The visual system is easily able to resolve the position of each object to the nearest half-centimeter and smaller [25]. As a result the discrete state space, defined by dividing up each input dimension at the resolution of the sensors, is extremely large. The playing surface being 244cm long and 122cm wide can be resolved into nearly 120,000 states,

velocity and angular orientation contribute about 100 states each. In total each object in the game can be accurately positioned into about 1.2 billion different states. Since the state of any object in the game is potentially important, following all three objects in the game would require a state space of about $1.7 * 10^{27}$ or 2^{91} states. It would be impractical to create a look-up table of this size, let alone explore the states adequately and find a good policy. Furthermore, if the game is scaled up, the state space is going to be multiplied about 1.2 billion times for every new player added. Even a linear time algorithm would have problems exploring that many states in a reasonable amount of time. Reinforcement algorithms are at best polynomial time algorithms [31, 16] and would be completely unreasonable. The first job of any learning agent in this domain is to define a similarity metric over the states, grouping similar states together until the number of states is small enough to explore in a reasonable amount of time.

4.1 Different Algorithms

Each of the learning agents have a similar structure. The only major difference is the strategy they use to divided up the state space. They all perform Q-learning with exactly the same parameters. Any differences between the different learning algorithms should be entirely due to the way the state space was subdivided.

The first step for every learning agent tested is to determine which state it is currently occupying. This is done by taking the input vector $< x_0, y_0, \theta_0, v_0, \ldots, x_3, y_3, \theta_3, v_3 >$ and determining into which bucket this places the agent. When one of the 12 input dimensions has changed enough so that a bucket boundary is crossed, or when 10 milliseconds has passed, the agent "changes state" and a new action will be selected.

At each state an agent is to select one of the following nine actions for the deliberator to perform:

1. *Wait:* This is the simplest behaviour. In short it waits for the situation to improve.

2. *Shoot:* This is the behaviour responsible for planning and following a path that will cause the ball to go into the opponent's goal.

3. *Clear:* This behaviour finds and follows a path that will cause the ball away from the home goal. This is a weaker requirement then shoot which requires a path that causes the ball to reach the opponent's goal.

4. *Go Home:* A simple behaviour that follows a path back to the agents own net.

5. *Defend Home:* Take up goalie position and keep the car situated between the ball and the net.

6. *Go Red:* Drives the car to the centre of the playing surface.

7. *Defend Red:* Similar to Defend Home except it occurs at the centre line.

8. *Servo:* Equivalent to "kicking the ball". This drives the car at the ball.

9. *Unwedge:* Tries to drive the car out of a stuck position.

After selecting an action a_{new} the agent must also check which action is currently active at the deliberator level a_{old}. If a_{new} is different from a_{old} then the deliberator is interrupted and started on the new task. Since actions are not discrete in this model, when a_{new} is the same as a_{old} the agent must decide if this means that the deliberator is to continue performing its current action, or if the deliberator is to replan and start the same action again. The method used here is to give the actions "inertia". The idea of inertia is that an action automatically is assumed to continue into the next state (unless the action happened to have already stopped). Another way of solving this would be to add one bit per action to the input vector so that the agent learns at each state if is should

continue or re-select the action. This would probably be more robust but would add extra dimensions to an already large state space.

Q-learning has three tunable parameters ($\beta = 0.9$, $\gamma = 0.5$, $PBEST = 0.9$) which control the learning rate, the discount for future rewards and the exploration strategy respectively. Since all the agents have exactly the same values for these parameters any detrimental effects of using suboptimal values should be similar for each agent.

The choice for γ is some what arbitrary. Setting $\gamma 0.9$ would be more realistic. However, because rewards are only a 1 or a 0 and goal states are self-absorbing, this means that if one Q-value is higher then means that the action with the higher Q-value leads to a goal state more quickly. The choice of γ only effects the magnitude of the Q-values, not their relative size

Semi-uniform exploration was chosen for the exploration strategy. This is the simplest strategy for deciding when to follow the current policy and when to try new actions to gain better information. The method is very simple, at point 2(c) of table 2.3 there is a fixed probability $PBEST$ of choosing the current policy action and $1 - PBEST$ of choosing a random action. A more intuitive approach is for $PBEST$ to start at a small value to encourage more exploration and slowly increase. Other strategies involve adjusting this probability according to how often the state has been visited, or according to the magnitude of change between Q-value estimates [31]. Adjusting $PBEST$ only effects the convergence rate.

Each different agent was tested first on the simulator and then the best of those were tested on the real robots. On the simulator the robot play until a goal is scored, than the robots and the ball are automatically positioned at the centre for a new trial. A game for a fixed interval of time, about 5 minutes. The only change required for games run on the physical robots is that the robots and the ball must be repositioned by hand after every goal.

4.1.1 The Hand Coded controller

The hand-crafted, non-adaptive controller created by Sahota [25] provides the standard agent that all the other agents use as the standard opponent for training and evaluation. Since, the optimal policy is not known, this hand coded controller plays the part of the optimal policy for evaluation purposes, therefore to converge to the "optimal" performance is to perform as well as the hand-coded controller (i.e. to tie or perhaps beat this controller).

The Hand-coded agent uses more information than the learning agents. The learning agents are limited to the four variables $< x, y, \theta, v >$ for each object. In the Hand-coded agent, however, the controller at the top, does not need to know positional information. Each of the lower behaviours decides for itself how important it is. They do this using a series of heuristic rules of the form "If I am already selected then add 2 to my value" and "If I can't find a path then make my value 0". The arbitrator receives a value for how important each behaviour thinks it is, and selects the behaviour with the highest value. Since, each behaviour has access to its internal state, it uses more information in generating its bid than just the four the learning agents use.

4.1.2 Regular Grid

This is the first of the learning agents. This agent uses the BOXES [21] approach to learning. First state space is divided up *a priori* into a fixed grid and the Q-learning performed on this discrete state space. The purpose of including this controller is to compare learning with a fixed similarity metric versus a adaptive similarity metric.

The inherent problem with dividing up each dimension globally versus recursively is that when fine resolution is needed the entire state space must be divided up finely. This gives a huge number of states, with twelve dimensions dividing up each dimension into

four boxes would give 4^{12}, about 17 million, total states. Each state requires one real number for each action. Assuming 32bit floating point numbers, 2^{24} states require about 2^{26} bytes of memory, an excessive amount for a mobile robot. Four boxes per dimension is probably not fine enough resolution, BOXES used about 256, but it only had two dimensions and two actions.

The boxes for this soccer player were created by dividing up the four dimensions of each object in the same way. The x coordinate value was divided into four equally sized boxes, same with the y coordinate. The orientation of each object was given two possible boxes, either towards the goal or away from it. The velocity of the object was given only one possible value. This gives a total of 2^{15} states occupying about 128Kbytes. It would be better if velocity and orientation also were divided into four boxes, but the size of the Q-table would be enormous.

Deciding how many buckets each dimension should be divided into and where the boundaries of the buckets should be are problems that are difficult to solve without extensive experience with the problem. The usual approach is to divide up the state space test to see how well it performs. Then the state space is divided up again using the information learned from the first test. This generate, test and generate again cycle is repeated until either satisfactory performance is reached or the programmer gives up in despair!

4.1.3 Random Agent

This agent is included as a means of comparing the how the agents improve over time. It performs no learning merely chooses a random action at each state. Using the same state space partition as the regular grid agent. The performance of the random agent is due entirely to the inherent information of the deliberator's behaviours.

4.1.4 Adaptive KD-tree

This algorithm adaptively divides up the state space, rather then learning on a fixed partition. It is based upon Chapman and Kaelbling's G-algorithm. However, rather then testing individual bits of the perceptual inputs and growing a binary tree that tests the relevant bits, a KD-tree is grown that tests multidimensional real valued inputs.

A KD-tree [28, 6] is a binary tree that branches on all dimensions rather then the branching on just one dimension like a normal binary tree. A normal binary tree organizes records based upon the "index" dimension, as a result queries based on other dimensions are very inefficient. A KD-tree allows multivariate queries, like those needed in nearest neighbour searches, to be performed in logarithmic time [6]. Bentley's algorithm for creating a KD-tree (table 4.4), is to first identify the best dimension to split. Next a partition plane is chosen, perpendicular to the dimension, such that the records are split into roughly equal subsets, each record being either above or below the partition plane. This procedure is continued recursively until the leaves of the tree contain only one record. This process has been extended to arbitrary planes that are not perpendicular to a dimension [28].

Repeat until all each leaf contains one record:
For each leaf do:

1. Compute the variance of the records for each dimension. The partition dimension is the one with the greatest variance.
2. Erect a partition plane perpendicular to the partition dimension, that passes through the point d_m, the median value of the records.
3. Partition the records into the high child if they are above the plane, or into the low child if they are below the plane.

Table 4.4: KD-Tree Algorithm

The KD-tree algorithm needs to be changed slightly to solve the structural credit

1. Initialize Tree
2. While true do
   (a) Use sensor input s to find current leaf of the tree $L(s)$
   (b) Find the best action: $a = max_{a_1} Q(L(s), a_1)$ Decide either to follow the current policy or to try a random action. Let a be the action performed.
   (c) Find the new state reached s_1 and immediate reward $R(s_1)$
   (d) calculate the New Q-value for the leaf,$L(s)$ using:

$$Q(L(s), a) = Q(L(s), a) + \beta(R(s_1) + \gamma \max_{a_1} Q(L(s_1), a_1) - Q(L(s), a))$$

   (e) Push the pair $< s, Q(L(s)) >$ on the history stack for $L(s)$
   (f) Decide if the tree should updated. For each leaf with a history do:
      i. Compute the variance of all the records in the history stack. Let D be the input dimension with the greatest variance.
      ii. Sort the records into two subsets based on which side of the plane, perpendicular to D passing through $Med(D)$, they are on.
      iii. Perform a t-test to decide if it is a good split.

Table 4.5: Adaptive KD-tree algorithm

assignment problem. First, Bentley's algorithm assumes that all the records exist, so the KD-tree is just a method of organizing them. For learning applications this is not true, the records (i.e. the Q-values) do not exist. The reason for clustering the states is that there is not enough time to collect all the q-values. The algorithm needs to be adapted to work without having all the records available for classification. Second, the termination condition needs to be changed. Rather then stopping when each record is uniquely classified, the algorithm should terminate when all the Q-values in a leaf are sufficiently similar, or like the ID3 algorithm, when information gained by splitting is small [23].

The algorithm tested here (table 4.5) solves the first problem, the problem of not

having the Q-values to build the tree with, by keeping a history queue. Whenever the Q-value of a node is updated, a tuple is created consisting of the Q-value and the input vector that caused the update, and is pushed on the queue. When the queue fills up the oldest observation is discarded and replaced with the new one. This allows statistics on how the Q-values within the cluster vary. From these statistics the best dimension and location for a partition plane can be calculated.

Initially the entire state space is considered one cluster. Q-values are collected and stored in the history queue. The variance of each dimension is calculated in two steps:

1. Calculation of the mean value for a dimension k:

$$m_k = \frac{\sum_i^n Q_i * s_{ik}}{\sum Q_i}$$

   where (s_i, Q_i) is the i^{th} pair in a history queue of length n. s_{ik} is the value of the k^{th} dimension of observation s_i.

2. Calculation of variance for a dimension k:

$$sd_k^2 = \frac{\sum_i^n Q_i(s_{ik} - m_k)^2}{\sum_i^n Q_i}$$

The dimension with the greatest variance is identified and a partition plane is created and the q-values in the queue are sorted onto one side of the plane or the other. A t-test is performed to see if continued splitting is required. The sample means x_1, x_2, the sample variances S_1^2, S_1^2, and the sample sizes n_1, n_2 are calculated for each of the two subsets. A t-test is computed using the equation:

$$t = \frac{x_1 - x_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

If the t-value is above a threshold the original leaf is replaced with the two new leaves, otherwise the original leaf is kept. This threshold corresponds to a fixed probability that

the two subsets are actually distinct and not just two groups of samples on the same set. The exact value of this threshold depends upon the data set and the number of clusters desired, and should be tuned for maximum performance.

An example of a KD-tree generated using a fixed depth update strategy, updating every 7.5 seconds to a fixed depth of 5 nodes, is shown in appendix D. It shows the dimensions that are split, where they are split, and the best action for each cluster.

4.2 Evaluation

4.2.1 Measures of Performance

The performance of each agent is judged on the final score of each game played. The score is transformed into a reward measure by taking the goals the agent scores and subtracting the number of goals the opponent scored. This gives a good estimate of how well the agent does in each game with a single integer. A tie game would be zero, a win would be a positive integer and a loss a negative integer. The performance of the agent over time is calculated by taking the moving average over the last 100 games played.

The criteria for comparing agents is done on three measures [15]:

Correctness is conceptually the simplest measure of performance. A agent is correct if for a given input it selects the same outputs as the agent following the optimal policy would select. The problem with this measure is it can be hard to define, as it is in this case, since the optimal policy may not be known. For this domain the hand coded agent plays the part of the optimal agent, therefore the the agent can be said to be correct if it receives continued positive rewards, i.e. it always wins. To measure the correctness of the agent all learning parameters are turned off, and the agent follows the policy it learned in pure exploitation mode.

Convergence is another important measure of performance. The fact that a learning

system converges to the optimal policy is not always sufficient, especially if the agent has poor performance over most of its lifetime, only improving after hundreds of thousands of trials. An agent that improves rapidly is usually better then one that does it slowly, although the slower converging agent may eventually be more correct. This distinction is very important for agents that operate in the real world, where finding an approximate solution quickly can have better results (win more games) over time.

Complexity is not directly related to the reward the agent receives. The time taken for computations and the space required should be bounded. Agents that take unbounded time to compute, might not keep up changes in the environment, and as a result will tend to achieve fewer rewards. Similarly, if the memory requirements can increase without bound then it might outgrow the memory available. The faster an agent can compute and determine the best action the better. Again, there might be a tradeoff between a slower more complex agent that is more correct and a simpler faster algorithm that is less correct.

4.2.2 Simulation results

Q-learning has several parameters that can be tuned to maximize performance. However, for these tests the only parameters of interest are not those of the Q-learning algorithm but the two parameters of the KD-tree algorithm, namely the threshold for the t-test, and the frequency with which tree is updated.

Figure 4.6 shows the best results for convergence for all the algorithms tested, after all the parameters were tuned. The sine like shape of the fixed grid agent is probably due to the fact that it does not have enough information. The limitations of memory only allowed a very coarse grid to be created. It is also important to note the performance of the random agent. This agent only loses by an average of 3 to 4 goals in 100 games. This quality of performance is due entirely to the "domain information" implicitly contained

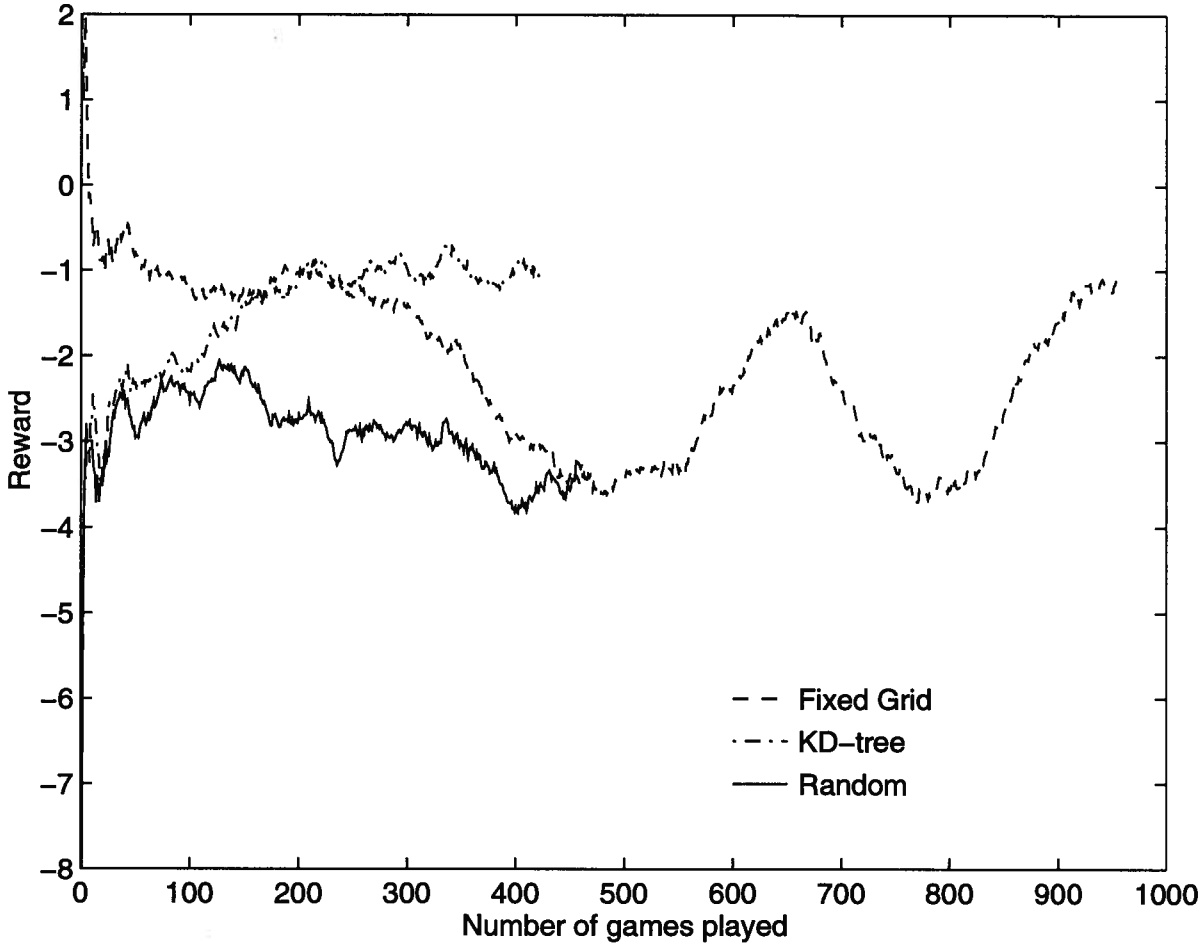


Figure 4.6: Convergence on the simulator

in the behaviours.

Figure 4.7 shows the effect of adjusting the threshold for the t-test. A low threshold causes the algorithm to more susceptible to noise and it divides up the state space poorly. A high threshold causes the state space not to be divided up at all. Choosing the threshold at about 0.7 or 0.75 seems to give the best performance.

After fixing the threshold for splitting states at 0.7, the update schedule is adjusted. This the parameter that selects when a trial split on the tree should be performed. Since it is a rather lengthy calculation it cannot be performed at every update. Three methods

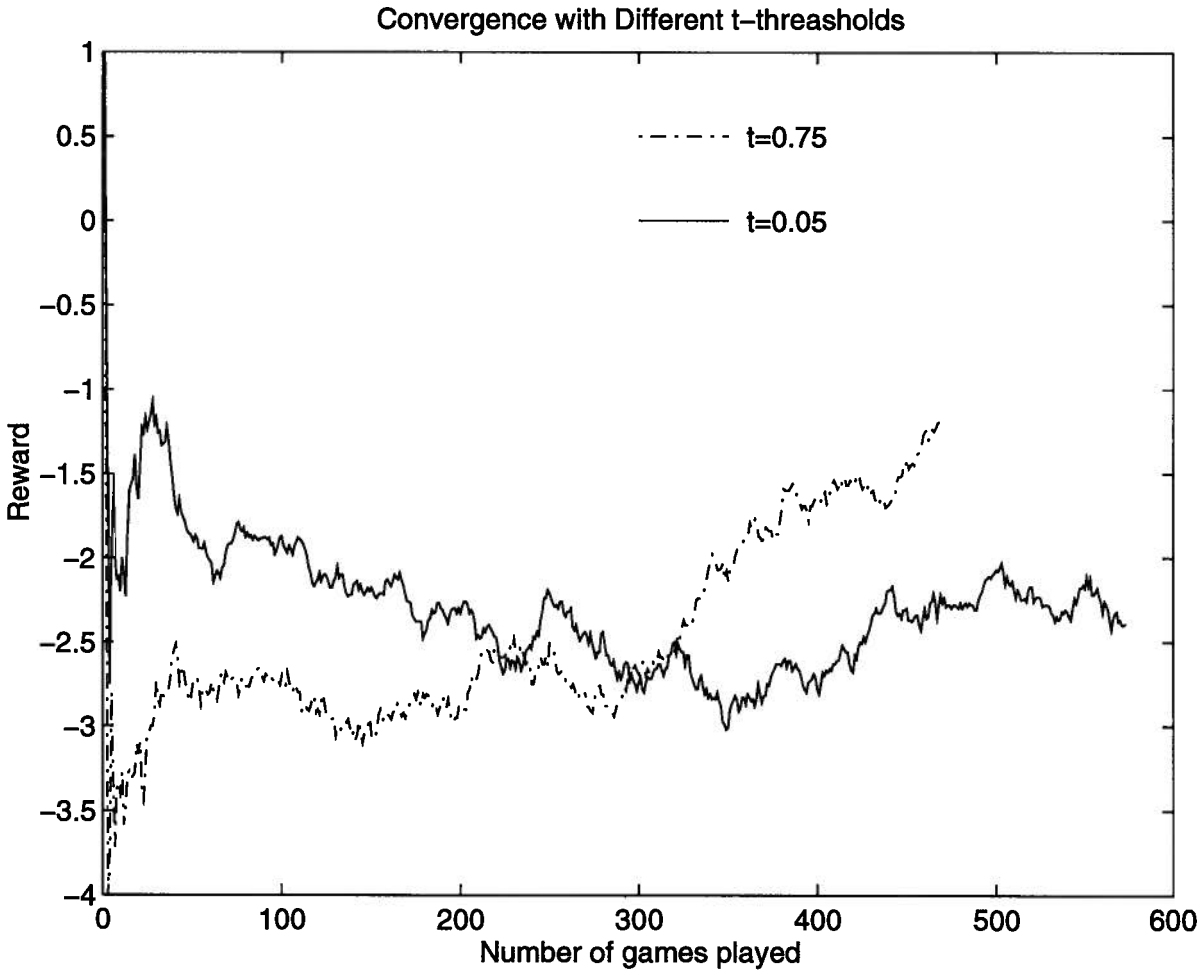


Figure 4.7: Adaptive KD-tree with different t thresholds

were tested, each with the same t-threshold (figure 4.8). A fixed update interval, where the the tree was tested at every n second interval, was tested for 5, 7.5 and 10 second intervals. This method worked great initially, the best results were achieved with a interval of 7.5 seconds. The 5 second interval was too short, the agent quickly started spending more time growing the tree then selecting actions. The 10 second interval was too long allowing the relevant history to be pushed out of the buffer. As the tree grows larger, however, the amount of computation that must be performed at each interval grows. The agent begins to spends more and more time trying to split the tree and ignoring what is happening on the soccer field and performance declines.

The second method tried was to increase the interval as the tree increased in size. Initially the tree was checked every 7.5 seconds, this interval was increased by 7.5 seconds everytime a new state was added. This method had the best performance but the state space was not divided up as well as it could have been. This is because as the delay between testing the tree increases the history buffer tends to contain more similar values and the t-values decrease.

The third method is to update on a fixed interval but only over a selected area of the tree. The agent tested, checks the tree every 7.5 seconds, but only to a fixed depth of 5 levels. This corresponds to checking 32 leaf nodes to see if they can be split which takes about the 10 milliseconds before a new action is selected. In this way the Q-values of the history buffer are not lost, and the agent does not spend an increasing amount of time growing the tree. However it tends to build smaller but more balanced trees. This would explain why it does not perform as well.

Updating the tree means the agent stops paying attention to the task at hand. This is why the convergence curves of figure 4.8 have a sinusoidal shape, particularly the increasing interval method. Here the tree may not be updated for scores of games and then it may have to update the whole tree at once. This allows the opponent to win

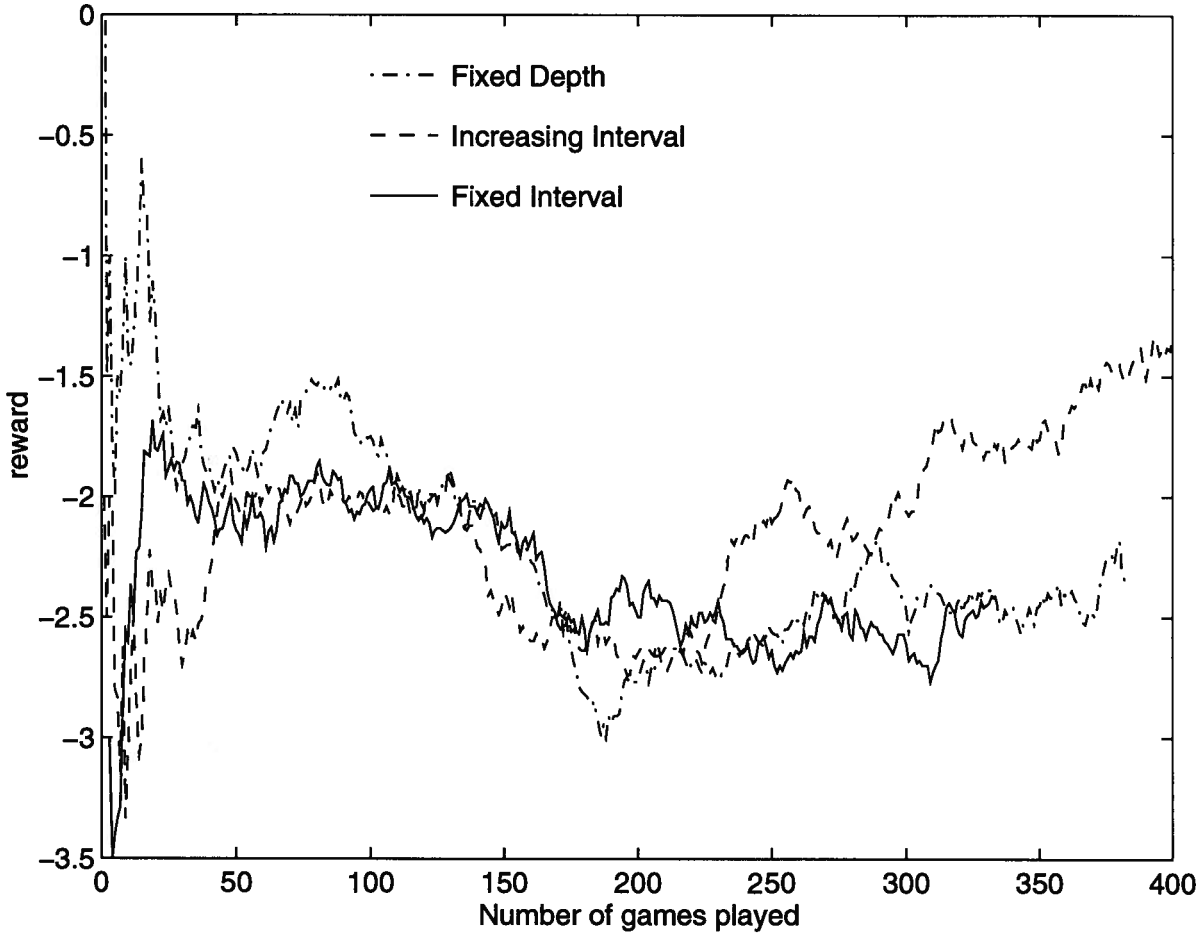


Figure 4.8: Adaptive KD tree with different update schedules

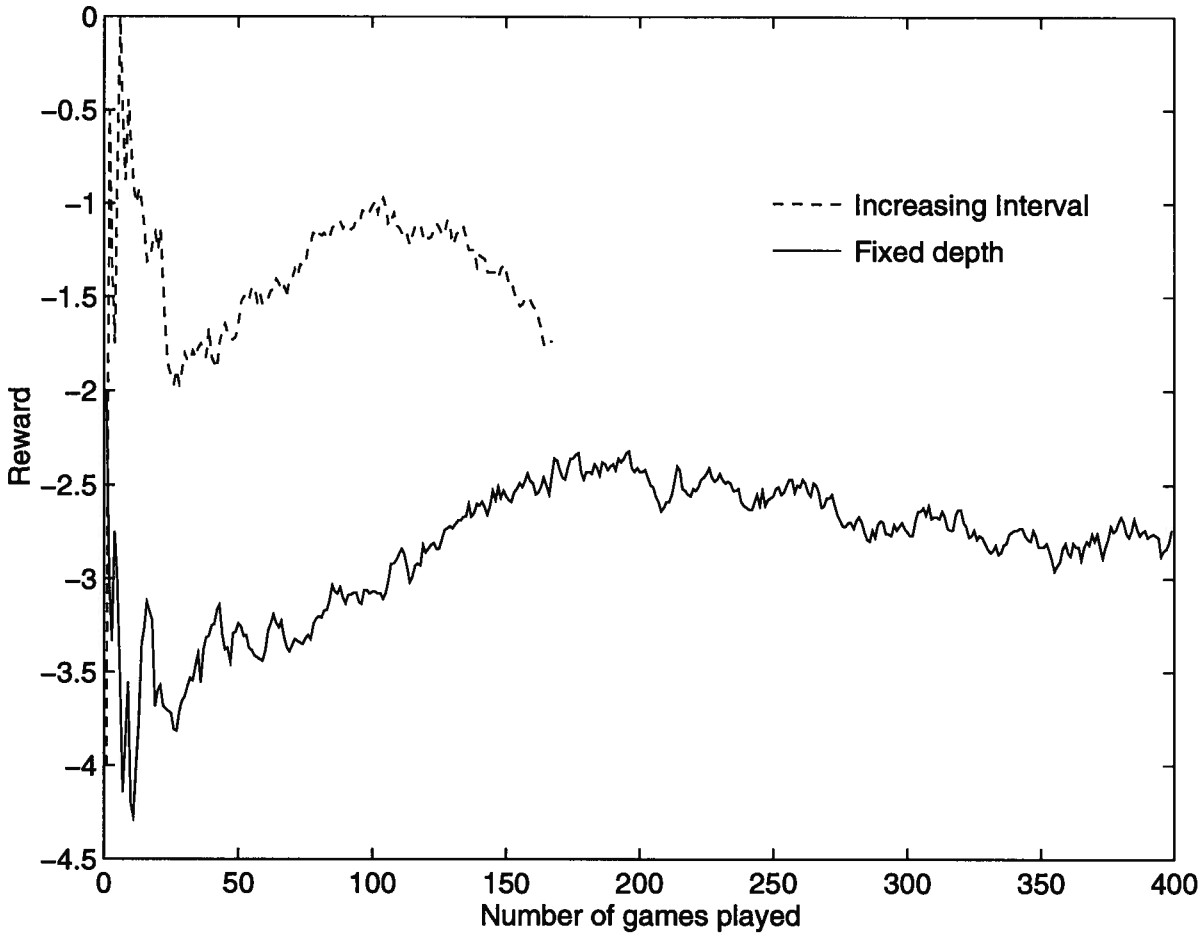


Figure 4.9: Convergence using a fixed tree

several games and brings down the average score.

To test the quality of the trees created by the different parameters. The KD-tree algorithm was turned off, and Q-learning was performed on the fixed tree that had been generated. The results are shown in figure 4.9. This only makes it more obvious that the increasing interval update method has built a better tree.

Testing the correctness of the agents required that all learning parameters be turned off and the agents act in pure exploitation mode. The results are shown in Figure 4.10. Both KD-tree algorithms perform about the same, averaging about -0.5 reward overall.

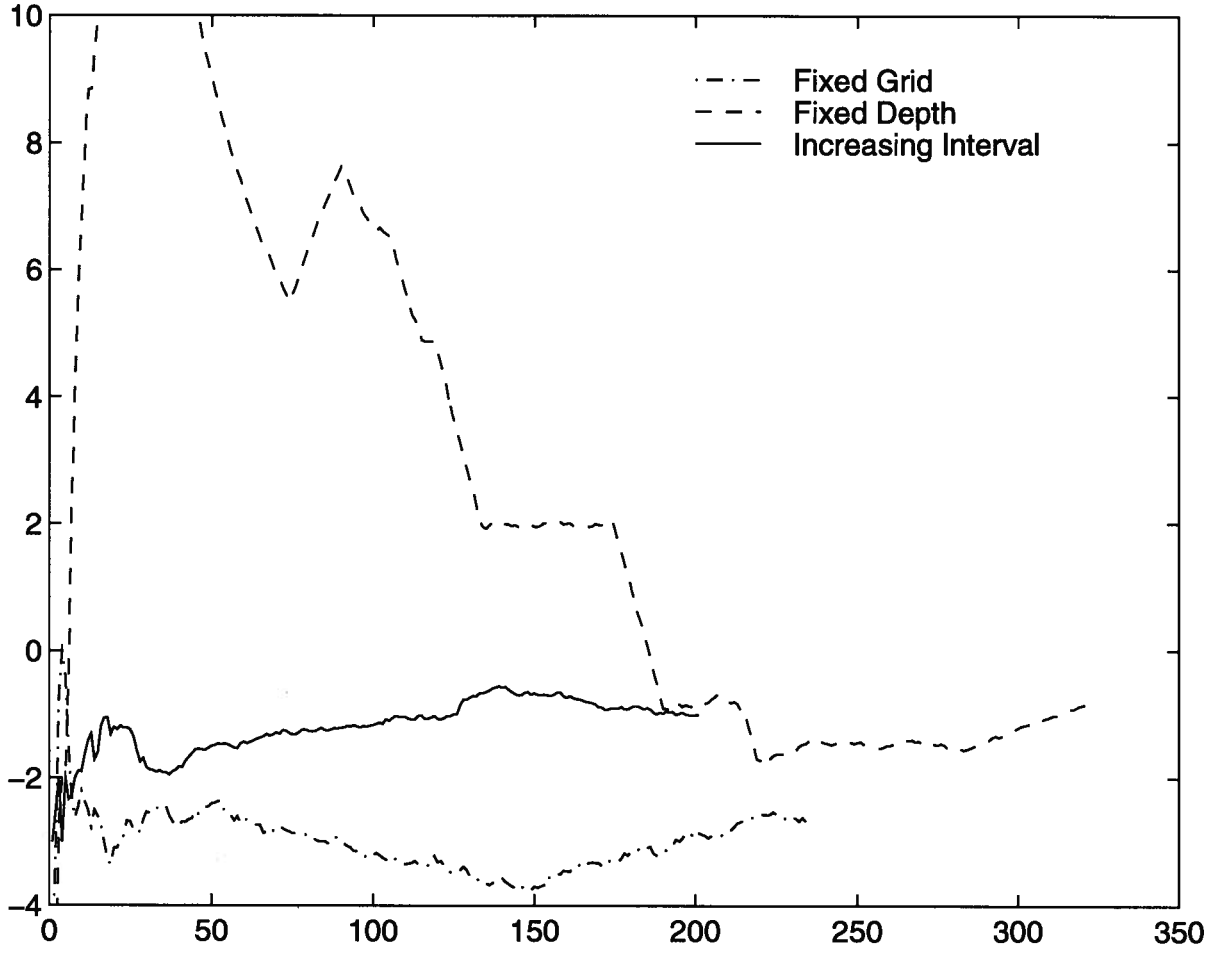


Figure 4.10: Performance using pure exploitation (simulator)

The regular grid learner performs about the same level as the random agent around an average reward of -3. Notice that the KD-tree created using the fixed depth update initially does extremely well, a reversal from its performance in the training phase.

4.2.3 Real Robot results

Doing all the training and testing on a simulator has some serious drawbacks. The biggest being that the simulator is never the same as the physical system. The agents should always be tested on the physical robots. Ideally they should be trained on the physical system as well but because of logistics this would have been impractical. Instead only the pure exploitation performance was tested on the robots and only for the two KD-tree agents. The results for the increasing interval KD-tree algorithm, were similar to those achieved in simulation. After playing for 5 minutes the score was 6 to 5 for the hand-coded agent. The fixed depth KD-tree agent performed very poorly on the real robots, loosing spectacularly 5 to nothing. This only shows that the results achieved in simulation are not directly transposable to real robots.

Chapter 5

Conclusion

5.1 Summary

In many systems where it would be beneficial to have a learning agent, the agent is ineffective because of the curse of too many inputs. In order for an agent to learn in a reasonable amount of time some method of identifying the important characteristic of the inputs must be used. In this thesis a methods of automatically identifying the important areas of the inputs was explored. Previous methods surveyed were able to cluster large state spaces but depended upon having continuous state spaces discretized, and having discrete actions.

The *Adaptive KD-tree Algorithm* was designed to work with continuous real valued inputs of high dimension and persistent actions. The algorithm does this by:

- Collecting statistics on the previous effects of actions. Then using those statistics to identify if and where a cluster should be further divided.
- Giving an "inertia" to actions, so that they can exist through different states; If an action is selected in two states in sequence. Then the action knows that this second selection is a continuation of the action started in the first state (unless the action happened to already have finished its task).

In general the Adaptive KD-tree algorithm works better then learning with a naive hand divided state space, although not as well as the non-adaptive agent that has been tuned by hand. To be fair the hand-coded controller does use more information than any

of the learning agents. It also remains to be seen how well other clustering algorithms, like neural nets, work on this domain.

The one drawback of the adaptive KD-tree algorithm is its *complexity.* Having a history buffer is very memory intensive, and imposes a limits on the type of trees that can be built. This can be partly rectified by choosing a good strategy for deciding when to test clusters. Testing at a fixed interval just does not work because the agent starts to neglect selecting actions and spends all of its time testing the tree. Increasing the interval as the tree grows works better, but it means the relevant history may be missed. A better approach might be to keep a measure of the relative error at each cluster and update at fixed intervals only the nodes with large errors.

Further testing is also needed to determine the effect of the various Q-learning parameters and to see how varying them effects convergence rates and optimality. The two main tests that need to be done in this area are to vary the learning rate β and the exploration parameter $PBEST$. Using the Boltzmann model [31] for adjusting $PBEST$ so that more exploration occurs earlier in trials but once information is learned the best action is more often chosen. Changing these parameters shouldn't effect the relative performance between the various algorithms only cause them to all improve.

5.2 Future Work

One extension to the adaptive KD-tree algorithm is to replace the history buffer with a spatial buffer. Keeping a buffer that only stores the last n decisions at the cluster makes the assumption that those n decisions will be randomly distributed through the entire volume of the cluster. This assumption is often incorrect. A buffer that stores decisions based upon where in the cluster they occurred would probably work better. This requires that some method of ensuring that all the samples in the buffer are of a similar age if

one area is much older then another, it may appear as if the state should be split, when in fact it would not be split if the samples were all the same age.

The next step would be to get rid of the buffer altogether. The buffer is uses a lot of space and imposes limits on the size of the tree that can be built. To remove the buffer would require that the best dimension to split and the location of the split be found without using statistics.

5.2.1 Adding existing knowledge

One observation about learning agents is that the more knowledge they have built in the faster they are to learn a task. The disadvantage is that they usually have to assume that the knowledge is correct. The multi-level learning system used in this thesis allows knowledge to be included into the system in such a way that the agent is able to override the knowledge if it seems irrelevant.

Knowledge can be included in the system in the form of behaviours. Each behaviour represents a skill or concept the programmer has explicitly written into the system. If the information is incorrect or irrelevant the agent will determine that through the course of normal reinforcement learning.

A similar method of adding information is through the creation of sensor behaviours. The standard *reactive-deliberation* controller as it is tested in this thesis uses only control behaviours. A controller behaviour is primarily concerned with the question: "This is the current inputs; what should the output at the lower level be?" A sensor behaviour answers the question: "This the raw sensor information (from the lower level), what does this mean?" For example, in soccer a typical sensor behaviour that might be useful is the *Closer* behaviour. This behaviour simply looks at the x, y coordinates of each object and sends a 0 or a 1 to the higher level, a 1 if the agent is closer to the ball, or a 0 if the opponent is closer. This simple function provides a new measure on the state space, a

measure that would have been hard for the agent to develop with only a lookup table.

Adding in sensor behavours is more costly then control behaviour. Each new control behaviour linearly increase the size of the lookup table. A sensor behaviour adds an new dimension to the state space, potentially an exponential growth in the size of the table. Of course if the agent uses a good clustering algorithm this is not a major problem. However, learning could be done by converting all raw sensor data into sensor behaviours and learning only over the space of sensor behaviours, discarding raw data in the learning process. Through judicious choice of sensor behaviours, a programmer can drastically reduce the size of the learning space.

Another way of adding information to the learner is through more detailed reward functions. In the experiments performed here, the reward function is very sparse, widely spread in time. This make learning difficult, a real soccer player don't learn this way; rather they receive intermediate rewards by coaches. When an soccer player performs a good action, like clearing the ball out of his own end, they are given an immediate evaluation of their action by the coach. This leads to (subjectively) good behaviour which can speedup the learning process by leading to "ultimate rewards" or goals more quickly.

Bibliography

[1] E. W. ABOAF, *Task-level robot learning*, Tech. Report AI-TR 1079, MIT Artificial Intelligence Laboratory, 1988.

[2] J. ALLEN, J. HENDLER, AND A. TATE, eds., *Readings In Planning*, Morgan Kaufmann Publishers Inc., 1990.

[3] R. BARMAN, S. KINGDON, A. MACKWORTH, D. PAI, M. SAHOTA, H. WILKINSON, AND Y. ZHANG, *Dynamo: real-time experiments with multiple mobile robots*, in Proceedings of Intelligent Vehicles Symposium, 1993.

[4] ——, *Dynamite: A testbed for multiple mobile robots.*, in Proceedings IJCAI Workshop on Dynamically Interacting Robots, 1993.

[5] R. BELLMAN, *Dynamic Programming*, Princeton University Press, 1957.

[6] J. L. BENTLEY, *Multidimensional binary search trees used for associative searching.*, Communications of ACM, (1975).

[7] C. BOUTILIER AND R. DEARDEN, *Using abstraction for decision-theoretic planning with time constraints*, in Proceedings of the 12th National Conference on Artificial Intellegence, AAAI press/The MIT press., 1994.

[8] R. A. BROOKS, *A robust layered control system for a mobile robot*, IEEE Journal on Robotics and Automation, (1986).

[9] ——, *A robot that walks: Emergent behavior from a carefully evolved network*, Neural Computation, (1989).

[10] D. CHAPMAN AND L. P. KAELBLING, *Learning from delayed reinforcement*, Tech. Report TR-90-11, MIT, 1990.

[11] P. CHEESEMAN, J. KELLY, M. SELF, J. STUTZ, W. TAYLOR, AND D. FREEMAN, *Autoclass: A bayesian classification system*, in Readings in Machine Learning, J. W. Shavlik and T. G. Dietterich, eds., Morgan Kaufmann Publishers, INC., 1990.

[12] T. DEAN, L. P. KAELBLING, J. KIRMAN, AND A. NICHOLSON, *Planning with deadlines in stochastic domains*, in Proceedings of th 11th National Conference on AI., AAAI press/The MIT press, 1993, pp. 574–579.

[13] R. A. HOWARD, *Dynamic Programming and Markov Processes*, MIT press, Cambridge, Massachusetts, 1960.

[14] R. A. JACOBS AND M. I. JORDON, *Learning piecewise control stategies in a modular neural network architecture*, IEEE transactions on Systems, Man and Cybernetics, (1993).

[15] L. P. KAELBLING, *Learning in embedded systems*, The MIT Press, Cambridge, Massachusetts, London, UK, 1993. Book Version of Phd. Thesis.

[16] S. KOENING AND R. G. SIMMONS, *Complexity analysis of real-time reinforcement learning applied to finding shortest paths in deterministic domains*, Tech. Report CMU-CS-93-106, Caregie Mellon University, 1992.

[17] L.-J. LIN, *Self-improving reactive agents: Case studies of reinforcement learning frameworks*, Tech. Report CMU-CS-90-109, Carnegie Mellon University, 1990.

[18] P. MAES AND R. A. BROOKS, *Learning to coordinate behaviors*, AAAI, (1993).

[19] S. MAHADEVAN AND J. CONNELL, *Automatic programming of behavior-based robots using reinforcement learning*, tech. report, IBM T.J. Watson Research Center, 1990.

[20] M. J. MATARIC, *A distributed model for mobile robot environment-learning and navigation*, Tech. Report AI-TR-1223, MIT Artificial Intelligence Laboratory, 1990.

[21] D. MICHIE AND R. CHAMBERS, *Boxes: An experiment in adaptive control.*, in Machine Intelligence 2, E. Dale and D. Michie, eds., Oliiver and Boyd, 1968.

[22] M. L. MINSKY AND S. A. PAPERT, *Learning*, in Readings in Machine Learning, J. W. Shavlik and T. G. Dietterich, eds., Morgan Kaufmann Publishers, INC., 1990. Originally published in Perceptrons: An Introduction to Computational Geometry, 1969. MIT Press Cambridge, MA.

[23] J. R. QUINLAN, *Induction of decision trees*, in Readings in Machine Learning, J. W. Shavlik and T. G. Dietterich, eds., Morgan Kaufmann Publishers, INC., 1990.

[24] D. RUMELHART, G. E. HINTON, AND R. J. WILLIAMS, *Learning internal representations by error propagation*, in Readings in Machine Learning, J. W. Shavlik and T. G. Dietterich, eds., Morgan Kaufmann Publishers, INC., 1990.

[25] M. K. SAHOTA, *Real-time intelligent behaviour in dynamic environments: Soccer-playing robots*, master's thesis, The University of British Columbia, 1993.

[26] M. K. SAHOTA AND A. K. MACWORTH, *Can situated robots play soccer?*, Canadian AI-91, 1994. Submitted.

[27] A. SAMUEL, *Some studies in machine learning using the game of checkers.*, IBM Journal on Research and Development, (1959), pp. 210–229.

[28] R. F. SPROULL, *Refinements to nearest-neighbor searching in k-dimensional trees*, Algorithmica, 6 (1991), pp. 579–589.

[29] R. S. SUTTON, *Learning to predict by the methods of temporal differences*, Tech. Report TR87-509.1, GTE Laboratories Incorportated, 1987.

[30] ——, *Integrated architectures for learning, planning, and reacting based on approximating dynamic programming*, AAAI, (1990).

[31] S. B. THRUN, *Efficient exploration in reinforcement learning*, tech. report, Carnegie Mellon University, 1992.

[32] C. WATKINS, *Learning from delayed Rewards*, PhD thesis, King's College, 1989.

[33] J. Y. YANGSHENG AND X. C. CHEN, *Hidden markov model approach to skill learning and its application to telerobotics*, Tech. Report CMU-RI-TR-93-01, Carnegie Mellon University The Robotics Institute, 1993.

Appendix A

Main Reinforcement algorithm

```
/*****************************************************************************
    DECIDE()
    This is the arbritrator between the behaviors
*****************************************************************************/
void decide()
{
    double pri;

    static int old_behaviour=0;
    static int old_time;
    static int R_time;
    static Object old_obj[MAX_OBJECTS];
    int i;
    Ktree *state,*old_state;
    new_exec = FALSE;
    if (program == PROGRAM_NORMAL_PLAY) {
      command_flag=TRUE;
      evaluate_flag=TRUE;
      state=get_state(obj);
      old_state=get_state(old_obj);
      i=reward(obj);
      if((state!=old_state)||(current_time*TYME_TO_MS>(old_time+10)))
{
/* Dont select a new action */
/* until state changes or times out*/
  if(state->high!=NULL) printf("Error \n");
  behaviour=select_action(state);
  old_time=current_time*TYME_TO_MS;
  update_Qvalues(old_state,state,i,old_behaviour,old_obj);
}
      if((current_time*TYME_TO_MS)>(R_time+7500)) /*milliseconds*/
```

```
   {
/* decides how often to rebuild the tree ms*/
/* 5.0,7.5,10.0 seconds*/
  K=rebuild_tree(K);
  R_time=current_time*TYME_TO_MS;
}

      if((behaviour==old_behaviour)&&(car.exec != EXEC_IDLE))
     /* INERTIA for actions: only replan if new action or executor is idle*/
  return_of_control=TRUE;
      else
  return_of_control=FALSE;
      pri=fn[behaviour](); /* Do the behaviour*/
      for(i=0;i<MAX_OBJECTS;i++)
{
  old_obj[i].x=obj[i].x;
  old_obj[i].y=obj[i].y;
  old_obj[i].head=obj[i].head;
  old_obj[i].speed=obj[i].speed;
}
      old_behaviour=behaviour;
    }


/*****************************************************************************/

Ktree *get_state(Object *o)
  return(search_tree(K,o));

/*****************************************************************************/

int select_action(Ktree *s)
{
  /*select the maximum behaviour*/

  int i,j,k;
  float PBEST=0.9; /* the prob. that the best action is chosen */
  int action=0; /* the best action */
  int best[MAX_BEH]; /* All the best actions */
  int nbest=0;
```

```
/* find the Best Action(s) */


  if((((float) (rand()%100))/100.0)< PBEST)
    {
      /* randomly select one of the best */
      action=response(s);
    }
  else
    {
      /* select any randomly */
      action=(rand()%MAX_BEH);
    }

  return(action);

}
```

Appendix B

Update Q-values algorithm

```
/*****************************************************************************/
int update_Qvalues(Ktree *s,Ktree *s1,int r,int b,Object *o)
{
  /* Qvalues q, state s, reward r, action b */
  int i,j;
  float E,E1,error,resp[MAX_BEH],BETA=0.9,GAMMA=0.5;
  E1=0;/* find the expected utility E */
  if(s1!=NULL)
    {
      i=response(s1); /* i is the best action */
      E1=0;
      for(j=0;j<(s1->d);j++)
E1=E1+(s1->bucket[j].response[i]);
      if((s1->d)!=0)
E1=E1/(float)(s1->d);
           /* avg reward for best action action  */
      else
E1=0;
    }

  E=0;
  for(i=0;i<MAX_BEH;i++) resp[i]=0;
  if(s!=NULL)
/* the top of the stack is the current Qvalue for this state*/
      for(i=0;i<MAX_BEH;i++)
resp[i]=(s->bucket[(s->d)-1].response[i]);
  E=resp[b];
  error=(r+GAMMA*E1-E);
  resp[b]=E+BETA*error;
  update(s,o,resp);
}
```

Appendix C

KD-tree algorithms

```
#include "ktree.h"
/*****************************************************************************/

Ktree *search_tree(Ktree *kt, Object *o)
{
  float m[DIMEN];
  if(o==NULL) return(NULL);
  else
    {
      /*12 dimensions */
      /*  BALL*/
      m[0]=o[0].x;
      m[1]=o[0].y;
      m[2]=o[0].head=2;
      m[3]=o[0].speed=3;
      /* CAR 1 */
      m[4]=o[1].x;
      m[5]=o[1].y;
      m[6]=o[1].head;
      m[7]=o[1].speed;
      /* CAR 2 */
      m[8]=o[2].x;
      m[9]=o[2].y;
      m[10]=o[2].head;
      m[11]=o[2].speed;
    }
  if(kt==NULL) return(NULL);
  if(kt->bucket==NULL)
      /*check where to split. kt->d is the dimension and kt->k is the value*/
      if(m[(kt->d)]>(kt->k))
  if(kt->high==NULL) printf("Error in  tree (search)\n");
```

```
  else return(search_tree(kt->high, o));
      else
  if(kt->low==NULL) printf("Error in tree(search)\n");
  else return(search_tree(kt->low, o));
  else
      /* return the correct model */
      if((kt->bucket!=NULL)&&((kt->high!=NULL)||(kt->low!=NULL)))
printf("Error in get state\n");
      return(kt);
}

/*****************************************************************************
  procedure: split
  description: finds the best dimension and location to split. Does a trial
split and performs a t-test.
*****************************************************************************/

Ktree *split(Ktree *kt)
{

  Ktree *n1,*n2,*n3;
  float t,k,m[DIMEN];
  float N1,N2,m1,m2,df,s,s1,s2;
  int d,i;
  if((n1=(Ktree *) malloc(sizeof(Ktree)))==NULL)
    {kt->c=0; printf("Err: Malloc\n"); return(kt);}
  if((n2=(Ktree *) malloc(sizeof(Ktree)))==NULL){kt->c=0; return(kt);}
  if((n3=(Ktree *) malloc(sizeof(Ktree)))==NULL){kt->c=0; return(kt);}
  n1->high=n2;
  n1->low=n3;
  n1->bucket=NULL;
  n2->d=0;
  n2->c=1;
  n2->high=NULL;
  n2->low=NULL;
  if((n2->bucket=(sample *)malloc(SAMPLES*(sizeof(sample))))==NULL)
    {
      kt->c=0;
      return(kt);
```

```
    }
  n3->d=0;
  n3->c=1;
  n3->high=NULL;
  n3->low=NULL;
  if((n3->bucket=(sample *)malloc(SAMPLES*(sizeof(sample))))==NULL)
    {
      kt->c=0;
      return(kt);
    }

/* find the dimension with the*/
/* maximum variance */
  var(kt->d,kt->bucket,m);
  k=0;d=0;
  for(i=0;i<DIMEN;i++)
    if(m[i]>k){k=m[i];d=i;}
  n1->d=d;
/* Find the location for the partion plane */
  k=median(d,kt->d,kt->bucket);
  n1->k=k;
/* split the samples high or low */
  for(i=0;i<(kt->d);i++)
    if((kt->bucket[i].inputs[d])>k)
      {
n2->bucket[n2->d]=kt->bucket[i];
(n2->d)++;
      }
    else
      {
n3->bucket[n3->d]=kt->bucket[i];
(n3->d)++;
      }
  s1=0;s2=0;t=0;
  if(((n2->d)>5)&&((n3->d)>5)) /* need at least 5 samples each */
    {
/* Do a t-test to see if the split is valid */
/* Null hypothesis means are equal */
      mean(n2->d,n2->bucket,m);
```

```
      m1=m[d];
      mean(n3->d,n3->bucket,m);
      m2=m[d];
      var(n2->d,n2->bucket,m);
      s1=m[d];
      var(n3->d,n3->bucket,m);
      s2=m[d];
      N1=0;
      for(i=0;i<(n2->d);i++)
N1=N1+max(n2->bucket[i].response);
      N2=0;
      for(i=0;i<(n3->d);i++)
N2=N2+max(n3->bucket[i].response);
      t=((s1*s1)/N1)+((s2*s2)/N2);
      if (t<0) t=-1*t;
      t=sqrt(t);
      t=(m1-m2)/t;
    }
  if((t<-0.70)||(t>0.70)) /* threshold???*/
    {
      /* reject Null hypothesis */
      /* Split the node */
      free(kt);
      n1->c=0;
      return(n1);
    }
  else
    {
      /* accept Null hypothesis */
      free(n1);
      free(n2);
      free(n3);
      kt->c=0;
      if(kt->high!=0) printf("Error in split\n");
      return(kt);
    }
}
```

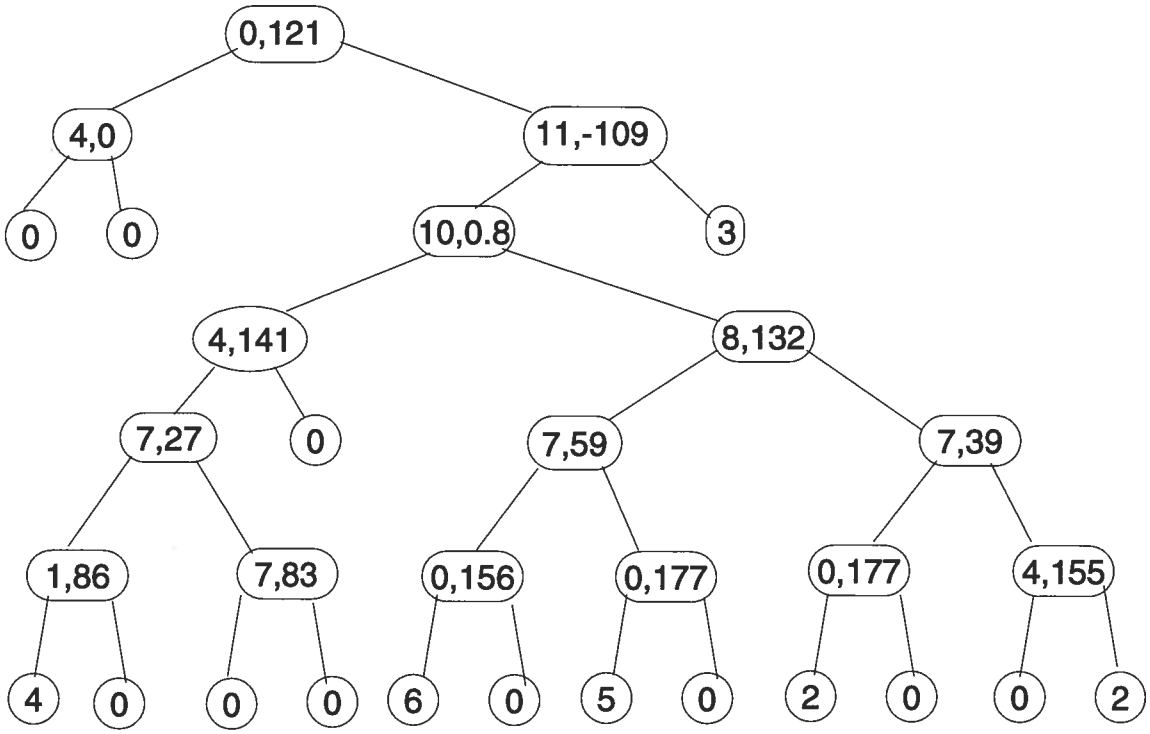
```
/*****************************************************************************
*****************************************************************************/

int response(Ktree *kt)
{
  int N,i,j;
  sample *B;
  float r[MAX_BEH];
  if(kt==NULL){ printf("Error in tree(response)\n");}
  else
    {
      N=kt->d;
      B=kt->bucket;
      if((N>SAMPLES)||(B==NULL)) printf("Error in Bucket\n");
      for(j=0;j<MAX_BEH;j++) r[j]=0;
/* use the top element */
      i=0;
      for(j=0;j<MAX_BEH;j++)
{
  r[j]=B[N-1].response[j];
  if(r[j]>r[i])i=j;
}
      return(r[i]);
    }
}


/*****************************************************************************
*****************************************************************************/

void update(Ktree *kt,Object *o,float *r)
{
  int i,j;
  kt->c=1;
  if((kt->bucket!=NULL)&&((kt->high!=NULL)||(kt->low!=NULL)))
    printf("Error in Update\n");
    if((kt->d)<SAMPLES)
      {
for(i=0;i<MAX_BEH;i++)
```

```
  kt->bucket[kt->d].response[i]=r[i];
kt->bucket[kt->d].inputs[0]=o[0].x;
kt->bucket[kt->d].inputs[1]=o[0].y;
kt->bucket[kt->d].inputs[2]=o[0].head;
kt->bucket[kt->d].inputs[3]=o[0].speed;
kt->bucket[kt->d].inputs[4]=o[1].x;
kt->bucket[kt->d].inputs[5]=o[1].y;
kt->bucket[kt->d].inputs[6]=o[1].head;
kt->bucket[kt->d].inputs[7]=o[1].speed;
kt->bucket[kt->d].inputs[8]=o[2].x;
kt->bucket[kt->d].inputs[9]=o[2].y;
kt->bucket[kt->d].inputs[10]=o[2].head;
kt->bucket[kt->d].inputs[11]=o[2].speed;
kt->d++;
      }
    else
      {
/*push down like a queue*/
for(i=0;i<(SAMPLES-2);i++)
  {
    for(j=0;j<MAX_BEH;j++)
      kt->bucket[i].response[j]=kt->bucket[i+1].response[j];
    for(j=0;j<DIMEN;j++)
      kt->bucket[i].inputs[j]=kt->bucket[i+1].inputs[j];
  }
j=(SAMPLES-1);
for(i=0;i<MAX_BEH;i++)
  kt->bucket[j].response[i]=r[i];
kt->bucket[j].inputs[0]=o[0].x;
kt->bucket[j].inputs[1]=o[0].y;
kt->bucket[j].inputs[2]=o[0].head;
kt->bucket[j].inputs[3]=o[0].speed;
kt->bucket[j].inputs[4]=o[1].x;
kt->bucket[j].inputs[5]=o[1].y;
kt->bucket[j].inputs[6]=o[1].head;
kt->bucket[j].inputs[7]=o[1].speed;
kt->bucket[j].inputs[8]=o[2].x;
kt->bucket[j].inputs[9]=o[2].y;
kt->bucket[j].inputs[10]=o[2].head;
```

```
kt->bucket[j].inputs[11]=o[2].speed;
      }
  if((kt->bucket!=NULL)&&((kt->high!=NULL)||(kt->low!=NULL)))
    printf("Error in Update\n");
}



/*****************************************************************************
*****************************************************************************/

Ktree *rebuild_tree(Ktree *kt)
{
int MAX_DEPTH=6;
static int depth=0;
/* oct2. modified to only split to a fixed depth.*/
if(depth<MAX_DEPTH)
  if((kt->bucket)!=NULL)
    {
      printf("depth=%d\n",depth);
     if((kt->c)==1) kt=split(kt);
    }
  else
    {
      depth=depth+1;
      if((kt->high)!=NULL) (kt->high)=rebuild_tree(kt->high);
      if((kt->low)!=NULL)(kt->low)=rebuild_tree(kt->low);
      depth=depth-1;
    }
if((kt->bucket!=NULL)&&(kt->low!=NULL)) printf("Error in Rebuild tree\n");
return(kt);
}
```

Appendix D

A KD-tree



A KD-tree generated after 360 games. The KD-tree is grown using a fixed depth update schedule, testing all the nodes less then a depth of 5 every 7.5 seconds.

Each internal node of the tree contains the number of the dimension that is split and the location of the partition plane. For example the root node contains the pair $(0, 121)$ which means that the balls x-dimension was split at the value 121. If the balls x-value is greater then 121 the right path is followed, otherwise the left branch is followed. The numbers of each dimension are:

0 The balls x-dimension.

1 The balls y-dimension.

2 The balls orientation θ.

3 The balls speed.

4 The opponents x-dimension.

5 The opponents y-dimension.

6 The opponents orientation.

7 The opponents speed.

8 The agents x-dimension.

9 The agents y-dimension.

10 The agents orientation.

11 The agents speed.

The leaf nodes contain the number of the action that was learned for that cluster. The list of possible actions and their corresponding numbers are:

0 wait

1 Shoot

2 Clear

3 Go Home

4 Defend Home

5 Go Red

6 Defend Red

7 Servo

8 Unwedge