Faculty of Engineering and Science
University of Aalborg

Department of Computer Science

TITLE:

Using Reinforcement Learning in the context of computer games

PROJECT PERIOD:

DAT6,
February 2004 - June 2004

PROJECT GROUP:

d633a

GROUP MEMBERS:

Ole K. Thomsen, okt@cs.auc.dk

SUPERVISOR:

Uffe Kjærulff, uk@cs.auc.dk

NUMBER OF COPIES: 5

NUMBER OF PAGES: 55

SYNOPSIS:

In this Thesis I investigate the ability of an agent, implemented using reinforcement learning, to learn and to adapt to a changing environment. I do this using two different methods of reinforcement learning, basic Q learning and a hierarchical method, MaxQ learning.
In order to test the agent implemented using these methods, I first design Flag Hunter, a simple turnbased game that forms the basis for the testing of the agent in different situations. Flag Hunter requires the agent to go to the opponent's base, pick up its flag, and return it to the agent's homebase.
In order to test the agents ability to learn, they were trained, first by playing the game without an opponent, and second, to play the game against opponents of different levels of randomness. It was found that the agent using the MaxQ method, would reach the goal significantly faster than the basic Q learning, but converge at about the same rate. The reasons for this was found and discussed.
To examine the agents ability to adapt to a changing situation, the agents were trained without an opponent, and then set to play against the opponent. It was found that neither was very succesful at adapting.

Summary

Though the computer games industry has risen to become a multimillion dollar industry the area of game intelligence has not been much developed. Today most computer controlled agents are implemented using finite state machines, that does not allow them to learn anything about the players ability to play the game. A common learning method is reinforcement learning that does allow learning from the players behaviour. Reinforcement learning has been used in computer games in path finding, and for playing various board games.

It was decided to investigate the performance of reinforcement learning with regards to learning and adapting in a game environment. To investigate reinforcement learning it was decided to select two methods, and compare them, based on their performance in a game scenerio. To be able to test the methods in a game, Flag Hunter, a sequential, turn based game was designed. Flag Hunter contain two actors who, to win, has to capture the opponents flag and return it to its homebase to win. An opponent for this game was designed. The agent was implemented using reaction rules, and will select an action based on an optimal policy. To make the agent play less optimally, a randomness function is applied that with a given probability will make the opponent select a random action.

It was chosen to use the traditional method of Q learning as one of the methods to compare. Two hierarchical methods were discussed, and it was found that MaxQ learning, using functional decompositioning could fit the problem of solving the Flag Hunter game. In order to test the reinforcement learning methods ability to learn and adapt, agents were implemented using the methods, and made to play the game. However, it was found that the MaxQ learning algorithm required a pseudo reward in order to learn.

To investigate the methods ability to learn, both agents were first, trained in the game, by playing without an opponent, and second playing against an opponent. Playing without an opponent will test how fast the agent can learn to find the goal state, and second how many games it takes before it converges. It was also tested how dependent the different methods are on the rewards given in the game. The test showed that the Q learning was very dependent on the rewards, whereas the MaxQ, apart from the pseudo reward in order to learn, was independent on the rewards. The agents were trained against opponents, with different levels of randomness. Playing against an opponent required the agent to both navigate to avoid being shot, and to learn to shoot. In order to make the MaxQ agent shoot it should be expanded with a sub-task that would allow it to shoot. This was not implemented, and the MaxQ agent could only learn to avoid being shot. The training showed that the best solution for learning to play against an opponent, it with a primarily explorative strategy for Q learning.

To examine how the methods could adapt to a changing situation, the agents were trained without an opponent. Using that policy, the agent was set to play against an opponent. The test showed that neither method was very succesful in adapting its behaviour, though they fairly succesfully managed to avoid getting shot.

The conclusion of the report was that each method had their strengths and weaknesses, but that neither could live up to the original demand of being adaptable.

Contents

Chapter 1

Introduction

In recent years the computer game industry has blossomed worldwide and become big business. Annully, millions of dollars worth of trade is produced during the development and marketing of computer games. Producing a computer game is usually a complex and time-consuming process, requiring large teams of game developers. This is necessary in order to meet the great demand for entertaining computer games with new and revolutionary graphics as well as great sound, and challenging computer opponents of extremely high quality.

However, focus has mainly been towards the graphics and sound parts of a game, at the expense of the quality of the intelligence of the computer-based opponents. Their behaviour in a game is often easy to predict, or so random and illogical that little sense is to be found in it.

Only few mainstream games actually contain opponents that are able to change their behaviour dependent on the behaviour of the human players. Here, the most common approach for controlling a computer-based opponent is through the use of a so-called Finite State Machine. Usually, this approach provides less credible opponent behaviour, making them look less intelligent during game play.

1.1 Learning

Reinforcement learning is a commonly used learning technique with similarities to the way most children and animals learn. This is achieved through the concept of learning by mistakes: if a child gets burned by a flame, it will shun fire in the future. In the case of animals, dogs can be trained to do the right thing by rewarding them with crackers. And punishing them by pulling the leash or shouting when they are

behaving badly.

Reinforcement learning in a more computational sense can be used to find a solution to a specific problem with clearly defined goals. A commonly used example is that of a robot learning to find its way around a house. Here, the robot will at first bump into walls or run low on power before eventually learning its way round. After some time, it will know where and how far it can go, before having to turn back to its docking station[SB98].

Witin computer games, the use of reinforcement learning techniques has found a niche. Here, it can be used to teach a computer-based player how to find its way round through the use of the A*-algorithm[Rab02]. This algorithm learn by estimating a value for the next possible steps and by rewarding the player when reaching its goal, or punishing it when meeting a dead end. Another example is the TD-Gammon program, which has learned to play the game of backgammon at a world class level[SB98].

A problem with many learning techniques is the rate of learning. Some er slow and require a great many training cases in order to achieve a certain level of expertise, e.g. playing a game well against a human player. Another problem is the dependency on information available. If only little is present, learning optimal behaviour can be quite difficult.

1.2 Problem formulation

The purpose of this report is to examine and evaluate reinforcement learning techniques in the context of the game of Flag Hunter.

In order to do this, a comparison of traditional reinforcement learning methods, and reinforcement learning methods using hierarchical decomposition of the original problem, is made.

Different hierarchical approaches are discussed and compared. Based on the ability to learn to learn and adapt to new situations, the best method is selected and implemented for learning in the Flag Hunter game. The end result is then evaluated and discussed along with comments on encountered problems.

1.3 Outline of report

In Chapter 2 the design of the game of Flag Hunter along with the design of the opponent is provided. The Flag Hunter game is used for training and testing the

reinforcement learning methods.

Chapter 3 explains the theory of reinforcement learning. Included here is the basic theory as well as a discussion of two concrete methods for hierarchical reinforcement learning. Following this, one of the methods is selected for training.

The actual training of the computer-based opponent using the concept of Q learning is provided in Chapter 4.

In Chapter 5 the opponent is trained using $MaxQ$, the hierarchical method selected in Chapter 3. The performance of training using Q and $MaxQ$ is then compared based on the above-mentioned criteria.

Finally, Chapter 6 deals with the implementation or the Flag Hunter game, along with a description of the learning algorithms.

Chapter 2

The Game

In this chapter the game Flag Hunter, which will be used as the scenario in the project, is presented. This will be done by first defining the game's, goal and rules, and second, specifying the opponent in the game.

2.1 Rules of the game

In this section the specific rules of Flag Hunter are described. Listed below are the overall rules, and in the following subsections a detailed description of the rules for movement and scoring is presented, along with the map of the game.

- The game consist of two actors, playing against each other. An actor can either be an agent or an opponent. Throughout the report the term agent will be used about the actor being trained.
- The goal of the game is to capture the opponents flag, and bring it back to the homebase.
- The actors start the game at their homebases.
- The game ends when an actor reenters its own base with the opponent's flag.

Map

The map the game is played on can be seen in Figure 2.1. It is divided into 10×10 tiles. The map is also divided up into five distinct rooms, which can only be reached through a number of doors. The actor's homebases are placed at opposite ends of

the map, why an actor must navigate through other rooms to reach the opponent's homebase.

The map has 100 different positions. With two actors that cannot occupy the same tile, this gives $100 \times 99 = 9900$ different states of the board. The actors can shoot in the direction they face. This expands the state space to $9900 \times 4 \times 4 = 158,400$ different states. Adding the flags, this number is quadrubled, as both actors can have the flag, one, or the other, can have it, or neither can have it. This gives a total of $158,400 \times 4 = 633,400$ possible states.

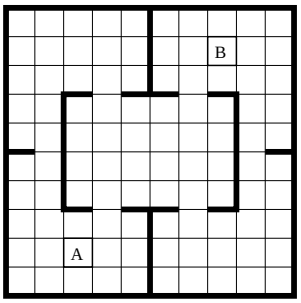

Figure 2.1: The map of the game

Actions

The game is turnbased, and a turn consists of one action. Actions in the game include both movement and specialized actions for picking up and putting down the flag, as well as shooting. The actors act sequentially. This means that the agent acts based on the board configuration, and following, the opponent acts based on the new board configuration. The actions are divided into movement actions, and 'behaviour' actions.

There are four movement actions, $north, south, east$ and $west$. A move action will move the actor in the specified direction, and afterwards the actor will be facing in that direction. Two actors cannot occupy the same tile, but can push each other. Therefore, the agent moving into a tile already occupied by the opponent results in the agent pushing the opponent out of the tile, in the direction the agent is moving, and the agent will occupy the tile. The 'behaviour' actions are pickup, putdown and shoot. These actions can be performed at any time during the game, but will only have a positive effect in certain states.

Shoot makes the agent shoot forward, in the direction it is facing. A shot will continue forward until it hits a wall, a base or the opponent. If an agent

carrying a flag is hit, the flag is returned to the base. An agent not carrying a flag is uneffected by being hit.

Pickup allows the agent to pickup the flag at the opponent's base. When the flag has been picked up, it disappears from the base, and is only returned if the agent loses it. If the agent performs the pickup action when not standing on the flag, the action has no effect.

Putdown allows the agent to put down the flag. If it is performed while not holding the flag, it will have no effect. If the agent holds the opponent's flag and puts it down while standing on its own base the game is won. If the agent puts down the flag at another position the flag is returned to the opponents base.

2.2 The opponent

In this section the opponent that is be used to test/train the agent in this project is specified. The goal of the opponent is to get the agent's flag and return it to its own base, thereby winning the game. The opponent is not meant to hunt the agent, but should still be able to shoot. Since a simple policy for a deterministic opponent in this game, given the above requirements, is optimal, the agent will not be able to beat it. Therefore, it is necessary to be able to weaken the opponent. This is by done specifying a probability p_{random} for the opponent executing a random action. This means that if $p_{random} = 0.1$, the opponent will be almost optimal, but that one of ten actions will be selected at random. An opponent with $p_{random} = 0.9$ will, on the other hand, be almost random with only one in ten actions being according to the policy.

Adding a degree of randomness in the opponents behaviour means that it is necessary for it to be able to navigate at all positions on the map, instead of just going from base to flag to base. This opponent is therefore designed using a set of reaction rules[Rab02]. This will allow the opponent to decide an optimal action based on the state it is in following a, possibly random, move.

The opponent will reach the flag, by passing a number of checkpoints on the way, one for each side of a door, and one for each base. The checkpoints are the designated tiles on the map that the opponent will have to pass to get to where it is going. Checkpoint are used because the map does not allow navigating from base A to base B by comparing the horizontal and vertical positions. However, because the checkpoints are all on the inside of room this approach is possible for navigating to them. To select an action, the opponent first checks if it can shoot the agent, second if it has the flag, third what room it is in, and fourth if it is standing on the rooms checkpoint. If the agent is standing on the checkpoint, it selects the

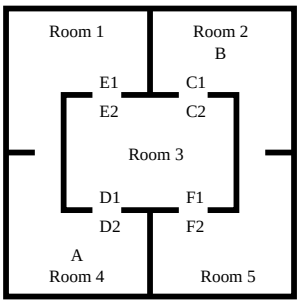

Figure 2.2: The checkpoints the opponent use to navigate

designated action, otherwise selects to go towards the checkpoint. The rules of the opponent's policy are shown in Figure 2.3, and the checkpoints positions on the map in Figure 2.2.

- if shoot agent possible then **shoot**
- else if has not flag
  - if in room 1
    - if at E1 then **south** else goto(E1)
  - if in room 2
    - if at C1 then **south** else goto(C1)
  - if in room 3
    - if at D1 then **south** else goto(D1)
  - if in room 4
    - if at A then **pickup** else goto(A)
  - if in room 5
    - if at F2 then **north** else goto(F2)
- else if has flag
  - if in room 1
    - if at E1 then **south** else goto(E1)
  - if in room 2
    - if at B then **putdown** else goto(B)
  - if in room 3
    - if at C2 then **north** else goto(C2)
  - if in room 4
    - if at D2 then **north** else goto(D2)
  - if in room 5
    - if at F2 then **north** else goto(F2)

Figure 2.3: The rules for the opponent

Chapter 3

Reinforcement Learning

In this chapter I go through the basics of reinforcement learning, Markov Decision Processes and Q-learning. Thereafter, two hierarchical methods will be compared, and the MaxQ method is described.

3.1 Markov Decision Process

A Markov Decision Process (MDP) is a 4-tuple (S, A, T, r).

- S is the set of states
- A is the set of actions
- $T : S \times A \rightarrow S$ is the transition function
- $R : S \times A \rightarrow \Re$ is the reward function

An MDP respects the Markov property that states that all transitions in any given state only depend on the current state, and that the history of states therefore is irrelevant [RN03].

An MDP works in discrete time. This means that each state can be marked by a time t, and the following state as time $t + 1$. Since the game is turnbased it will function in discrete time. A move is made a time t, and, because all agents move at the same time, the following at time $t + 1$.

Since the game is turnbased, and as such runs in discrete time, a state is the position and direction of all players, and whether the flags have been picked up or not, at the

given time. In Figure 3.1 an agent in state s with $A = \{north, south, east, west\}$ performs the action $east$ and goes to the state s'.

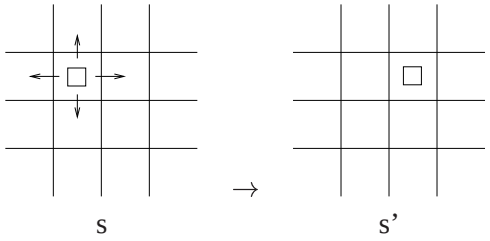

Figure 3.1: A state and an action

When dealing with a deterministic scenario T and R are deterministic. This means that an agent in state s performing action a always will go to the same state s'. The agent moving $east$ in Figure 3.1, with no opponent, can only reach the state positioning it one tile east of where it was before. However, in a non-deterministic scenario an agent performing an action a in state s only knows a probability distribution of what state s' to reach. This means that an agent in a scenario with an opponent making a simultaneus move, can reach one of four possible states, depending on the opponents action, this is shown in Figure 3.2. Therefore, it is necessary to extend the transistion functions with the probability of reaching each possible state.

There are different notations for expressing the probability of the resulting state [SB98] [Mit97], but in this report the notation used in [Die00] is chosen. Equation (3.1) gives the probability that when in the current state s_t is s, and the current action a_t is a, the next state s_{t+1} is s'.

$$P(s'|s,a) = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{3.1}$$

Though the reward function is, usually, deterministic, the expected reward is dependent on the state the agent will reach. Therefore, as the transition function in the non-deterministic scenario returns a probability distribution, can be expressed as

$$\mathcal{R}^a_{s \to s'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{3.2}$$

In Figure 3.2 agent A, can get a reward by moving $north$ and then $east$, if agent B moves $south$. However, if agent B does not move $south$, the reward will be zero. If B is an 'intelligent' agent it will most likely choose not to move south, but the agent described in Section 2.2, will not react depending on the position of the

agent. The expected reward, following a $north$ action, but before the opponent's move, is given as the sum of the products of probability of reaching that state and the reward of reaching it:

$$E(s, \pi(s)) = \sum_{s' \in S} P^{\pi}(s'|s, a) V^{\pi}(s')$$

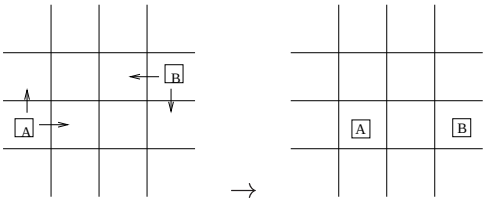

Figure 3.2: A state with two agents

A non-deterministic MDP will create a tree of states, since every action can lead to several different states. Therefore it is not possible to find a fixed path to the goal, and it is therefore necessary to have a policy that considers all possible states. The situation of the game is illustrated in the backup diagram in Figure 3.3. Because there is an opponent in the game, following each action the agent can be in four different states when making its next move.

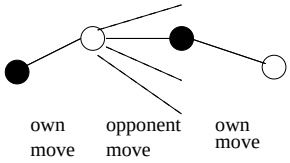

Figure 3.3: Back-up diagram for the game in this report

3.2 Reinforcement Learning

Reinforcement learning is based on the principles of *trial-and-error*. This implies that the agent will update its behaviour based on its performance. The performance is here quantified as a reward which can be either positive or negative [Mit97].

The purpose of reinforcement learning is to learn a policy, $\pi : S \rightarrow A$, which will tell the agent what to do when in a specific state. The aim of a policy is to get the highest possible reward, by reaching a goal state. In the example in Figure 3.2 a goal state for agent A could be a state where it could shoot at agent B.

To be able to deal with delayed reward it is necessary to be able to calculate the present value of a future reward. Therefore the reward discount factor, λ, is introduced. The discount factor is a value between 0 and 1, which gives the discounted reward dependent on the number of steps to the goal state. The accumulated discounted reward for following a policy with infinite horizon beginning in state s_t is given by

$$V^{\pi}(s_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} \cdots \equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{3.3}$$

γ can be used to determine whether to prioritize a fast reward or go for the future, but greater reward. A small γ value will make the agent go for the faster reward, since the discounted rewards are small due to the factor. A large γ value will make the agent more prone to go for a greater, but later, reward, as the discounted rewards are not as diminished.

Whereas (3.3) deals with an infinite horizon, scenarios exist that deals with other situations. Tic-Tac-Toe is played with a maximum of 9 turns, which fills the board [SB98]. This means that the value function should deal with a finite horizon, as shown in (3.4).

$$V^{\pi}(s_t) = \sum_{i=0}^{h} r_{t+i} \tag{3.4}$$

Solving the FlagHunter game will, in its pure form, deal with a infinite horizon, as it can, theoretically, continue forever.

The optimal policy is the policy that returns the greatest accumulated value.

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s) \tag{3.5}$$

Figure 3.4 shows an optimal strategy for a single agent in a simplified and deterministic scenario. Agent A have to get to Base B to pick up the flag, therefore the optimal policy is the policy that gets it there the fastest, i.e. in the fewest steps. The value for performing the strategy, given γ = 0.9 is

$$V^{\pi} = 0 + 0.9 \cdot 0 + 0.9^2 \cdot 100$$

A problem with the V^* value function is that it requires knowledge of the next state, s', given an action s. Since T, in the non-deterministic case, returns a probability distribution rather than the resulting state, the V^* function cannot be used. There-

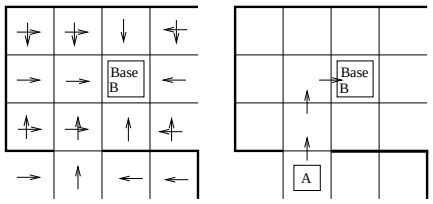

Figure 3.4: (a) An optimal policy for a single agent (b) the optimal behaviour for agent A

fore it is necessary to examine another, more appropriate reinforcement learning method. The following section will examine Q-learning.

3.2.1 Q-learning

Q-learning is a reinforcement learning method that allows an agent to learn an optimal policy without knowing the T and R functions [Mit97]. The Q function evaluates the immediate reward of an action, and the discounted reward of following the optimal strategy afterwards.

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a)) \tag{3.6}$$

This means that the optimal policy can be rewritten as:

$$\pi^*(s) = \arg\max_s Q(s,a) \tag{3.7}$$

From (3.5) and (3.7), (3.6) can be written as a recursive funtion.

$$Q(s,a) = r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a') \tag{3.8}$$

The recursive behaviour of (3.8), means that it is only necessary to maintain information about the Q values for each state. This can be done using a table with entries for each individual state. Therefore, when selecting the optimal action in figure 3.5, the Q value is given as, with a reward for pickup of 100, and a γ of 0.9:

$$Q(s,a) = 0 + \gamma 100 = 0 + 0.9 \cdot 100 = 90$$

When learning, the Q training rule (3.9) is the learners estimate of the Q value, since the actual value is not necessarily known. The values of $\hat{Q}(s, a)$ are stored, and serve as the learners hypothesis of the world.

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{s} \hat{Q}(s', a') \tag{3.9}$$

There are different ways of selecting the next action, depending on whether to explore or exploit. This is discussed in the next section. Also, alternatively, the table can be initialized with random numbers instead of zeroes. This can make the agent more prone to explore the far reaches of the game, since the Q values of the unexplored states are comparable to the known.

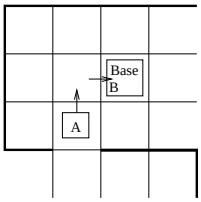

Figure 3.5: The evaluation for the optimal action

3.3 Exploitation vs. Exploration

When an agent is training to learn a policy for a given scenario, the strategy for selecting an action can determine how much of the statespace the agent actually meets. This is generally referred to as the exploitation/exploration dilemma.

Exploration is to explore the lesser known states of the scenario.

Exploitation is to exploit the known states, to maximize the immediate reward.

Selecting the action with the greatest Q value will make the agent prone to go for the immediate reward, and not explore. This can be a problem, if the table is initialized with zeros, as it can make the agent repeatedly return to the first reward/goal state it encounters. Also, even with a fully updated table, the agent can ignore a greater reward for a smaller, but closer one.

An alternative method for selecting an action is choosing a way to allow it to occasionally select an immediately less desirable action [Mit97].

$$P(a_i|s) = \frac{k^{Q(s,a_i)}}{\sum_j k^{Q(s,a_j)}} \tag{3.10}$$

This gives a probability P for choosing a_i from state s, based on the k. The k, $k > 0$, value is a constant value, determining the agent prone for choosing large or below average Q values. A large k will give a larger probability to selecting an action with a large Q, and a small k will give a larger probability for choosing an action with a smaller Q value.

The need for exploration, however, is mainly present, when the agent is still learning. If the agent already has learned an optimal policy, an exploratory strategy for action selection will only make the agent perform less than optimal. When the agent has obtained an optimal policy, it should use an exploitive strategy for action selection, to take advantage of the optimal policy.

3.4 Hierarchical Reinforcement Learning

The methods discussed above, all basically regard the state space as a simple table. Therefore, a long sequence of actions can be necessary to go from a start state to a goal state. A way to limit the number of actions necessary to learn a policy is by using hierarchical methods. In the following, two different hierarchical reinforcement learning methods will be presented and discussed for their relevance in this project.

Using macro-actions

Solving a MDP using macro-actions uses a 'flat' hierarchical structure, with two layers[HMK$^+$98]. This is done by dividing the state space up into smaller regions, and solving an abstract MDP to *traverse* the state space.

A macro-action is a local policy found for each individual region, which is executed when the agent is in the region. The borders between individual regions are defined as peripheries. The entrance peripheries are the states leading into the region, and the exit peripheries are the peripheries leading out of the region.

The abstract MDP consist of the periphery states, and uses the macro-actions as transition functions between them, leading from an exit periphery state to an entrance periphery state. A policy for the abstract MDP, then constitutes a macro policy for the original MDP.

In [HMK$^+$98], a 11×11 map consisting of 11 rooms in a maze structure, is used to test the method. A non-uniform cost is assigned to each state, except the goal state. The purpose is minimizing the expected cost of reaching the goal state. A solution using macro actions and an abstract MDP is compared to a solution using

a flat MDP. The results show that the solution using an abstract MDP converges much faster than the solution using the flat MDP. However, the abstract MDP did not find an optimal policy.

MaxQ

The MaxQ method uses functional decompositioning to reduce the original problem into a number of smaller tasks, that are hierarchically connected. This produces a tree of tasks where all leafs are the primitive actions of the original problem.

A task in the tree regards all child nodes as actions. A policy for each task is found, and executed by traversing the tree until a leaf node is met, and the action is executed.

In [Die00], the MaxQ method is tested with an example of a taxi and a passenger on a 5×5 map. The purpose of the test is for the taxi to pick up the passenger and transport it to a designated location and putting him down. The solution using MaxQ is compared to a solution using Q learning, which shows that the MaxQ converges to the optimal solution more twice as fast.

The two methods presented above, both appear to be faster than traditional reinforcement learning. The choice of what method to use, is based on the perceived gain by decompositioning the Flag Hunter game, using either method. The solution using macro-actions seems to give a huge advantage, compared to using a flat MDP, however, may not be equally advantageous for Flag Hunter. The example used in [HMK$^+$98] uses many rooms, which gives it very small sub-MDPs. The functional decomposing of MaxQ, it more comparable to Flag Hunter. The goal of Flag Hunter is practical to divide into a number of discreet subtasks, to solve individually. It is chosen to use the MaxQ method to train the agent to play Flag Hunter.

3.4.1 MaxQ

MaxQ divides the problem to be solved into a number of hierarchical subproblems. Each subproblem is then again divided into subtasks until it comprises of only the actions available in the original problem.

The MDP $M = \{S, A, T, R\}$ is divided into a finite number of sub-MDPs $\{M_0, \ldots M_n\}$ representing the subtasks. The subtaks M_0 is the root subtask, such that solving it will solve the original MDP M. Throughout this report the subtask index i, will be used to denote the subtask. The hierarchical policy for M, is a set containing a

policy for each of the subtasks in M, $\pi = \{\pi_0 \dots \pi_n\}$

The subtask is defined as a 3 tuple $T_i, A_i, \tilde{R}_i$.

- $T_i \in S_i$ is a set of termination states. The subtask M_i is only executed if in a state s in S_i. If s is in T_i the subtask is terminated. A terminatition state need not be a goal state.
- A_i is the set of actions that can be executed in M_i. The action in A_i can either be actions in A, or other subtasks.
- $\tilde{R}_i$ is a pseudo reward function. $\tilde{R}_i$ specifies a pseudo reward for an action to a terminal state $s' \in T_i$. The pseudo rewards are used to rate the desirability of terminal states, by giving a low pseudo reward to a undesirable terminal state, and a high to a goal terminal state.

Due to A_i, the set of subtasks $M_0 \dots M_n$ can be represented in a tree structure, with M_0 at the root. Representing the set of subtasks as a tree, the individual tasks are divided into two part; A maxnode that represents the actual task, and a qnode that represents the action in A_i a task can execute.

Decomposing the Flag Hunter game, the problem of getting the opponent's flag can be divided into the subtasks of *moving to the flag* and *picking up the flag*. *Picking up the flag* involves executing the action *pickup*, and *moving to the flag* involves navigating to the flag by executing a sequence of navigating actions. The tree for getting the opponent's flag is shown in Figure 3.6.

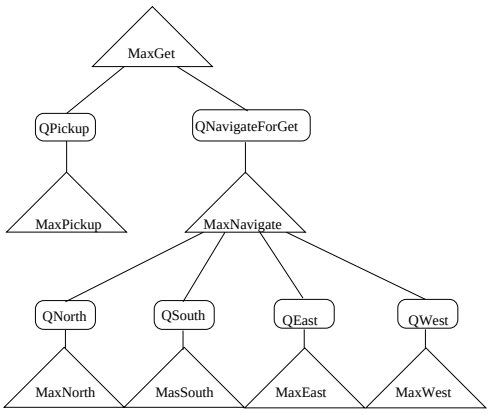

Figure 3.6: A tree decomposing the task of getting the opponent's flag. The triangles represent the maxnodes, and the squares represent the qnodes

The policy for every task, uses a Q value function to represent the expected value of every action a in A_i. The Q function for the task i, gives the discounted value

for executing an action a in state s. The MaxQ decomposes the Q function into two parts. $V(s,a)$ the immediate reward for executing an action a in s, and $C(i,s,a)$ the expected reward for finishing the task i.

$$Q(i,s,a) = V(s,a) + C(i,s,a) \tag{3.11}$$

If the action a is a primitive action, i.e. $a \in A$, $V(s,a)$ is the immediate reward received in the state s'. In *getting the opponent's flag*, selecting the pickup action will return the reward for picking up the flag, if the agent is standing on it.

$$V(s,a) = \sum P(s'|s,a)R(s'|s,a) \tag{3.12}$$

If the agent selects the *move to flag*, the $V(s,a)$ returns the value for navigating to the opponent's base. This means that it will return the maximum Q value for the subtask.

$$V(s,a) = max[V(s,a) + C(i,s,a)] \tag{3.13}$$

Therefore, the value for $Q(getflag, s, navigate)$ is written as the following, where the action a is the primitive action in $A_{navigate}$.

$$Q(getflag, s, navigate) = max[V(s,a)+C(navigate, s, a)]+C(getflag, s, navigate)$$

Learning algorithm

Learning a hierarchical policy using the MaxQ method means finding Q values for all tasks, in order for the agent to reach its goal. The learning algorithms for the MaxQ method work by applying the hierarchical structure. The task at root level selects an action a in A_0 in state s. This again selects an action and continues till a primitive action is selected. The chosen task continues until it meets a terminating state for either itself, or any ancestor task, in which case it terminates.

The MaxQ-0 algorithm is shown in Figure 5.1. This works recursively, and is called with the maxnode i, and the state s. If the maxnode is a primitive action, the immediate reward is stored. It returns the number of actions required to perform the task, which for a primitive action is one.

If the maxnode is a sub-task, it will select an action, and receive the number N of primitive actions required to execute it. The $C(i,s,a)$ is calculated as $\gamma^N V(i,s)$, which is the discounted value for reaching the terminating state.

A problem with this approach is that the information will always flow upwards in the tree. In the Flag Hunter game, the agent will have to move to the flag and then

function MaxQ-0(maxnode i, state s)

if i is a primitive Maxnode

execute i, receive r, observe state s'

$V_{t+1}(i,s) = (1-\alpha_t(i))V_t(i,s) + \alpha_t(i)r_t$

return 1

else

let $count$ =0

while $T_i(s)$ is false

choose action a

let N=MaxQ-0(s,a)

observe s'

$C_{t+1}(i,s,a) = (1-\alpha_t(i))C_t(i,s,a) + \alpha_t(i)\gamma^N V_t(i,s')$

$count = count + 1$

$s = s'$

return count

Figure 3.7: The MaxQ-0 learning algorithm [Die00]

pick it up. Using the MaxQ-0 algorithm for this, with the decomposition, shown in Figure 3.6, the agent will enter the navigate task, and remain there until it reaches a termininating task, by standing at the flag. It will then return to the getflag node, where it will choose between navigate and pickup flag. The reward for picking up the flag, will propagate up the tree, but not down to the navigate task. Therefore, it is necessary to add a reward for reaching the goal state, for the method to converge.

In [Die00], this is done by applying the pseudo reward specified for the subtask. The pseudo reward is a value given to the desirability of the terminating states. A $\tilde{C}$ is calculated, adding the $\tilde{R}$ to the value for for executing the action, thereby allowing the task i to influence the value for executing the action a.

$$\tilde{C}(i,s,a) = \gamma^N[\tilde{R}(s') + \tilde{C}(i,s',a^*) + V(a^*,s)]$$

The action selecton is then selected based on the $\tilde{C}$

$$a^* = \arg\max_{a'}[\tilde{C}(i,s',s') + V(a',s')]$$

The principal difference between adding the formal description of a pseudo reward, and adding a reward to the game is considered to be negligible. As they both require

a specification of the importance of the affected state, it is chosen to use the MaxQ-0 algorithm for testing the MaxQ method in this project. The model for testing the Flag Hunter game using the MaxQ method for training the agent, is presented in Chapter 5.

Chapter 4

Q model

In this chapter the agent is trained using the Q learning method described in Section 3.2.1. In this chapter I will first explore the influence of the individual rewards in Q learning, by training the agent playing without an opponent. Second, based on the optimal set of rewards found, the agent is trained by playing against the opponent. The algorithm used for training the agent is shown in Figure 4.1[Mit97].

For each state s, a initialize the table entry $\hat{Q}(s, a)$ to zero

Observe the current state s

do forever

- Select an action a and execute it
- Receive immediate reward r
- Observe the new state s'
- Update the table entry for $\hat{Q}(s, a)$ as: $\hat{Q}(s, a) \leftarrow r + \gamma \max_s \hat{Q}(s', a')$
- $s \leftarrow s'$

Figure 4.1: The Q-learning algorithm

The agent is trained playing on the map shown in Figure 2.1, in Section 2.1. This consists of 10×10 positions, which are considered as states. However, since the agent is also dependent on the direction it is facing, there are four states for each position on the map, as shown in Figure 4.2. This also means that a north facing state can only be reached by a *north* action.

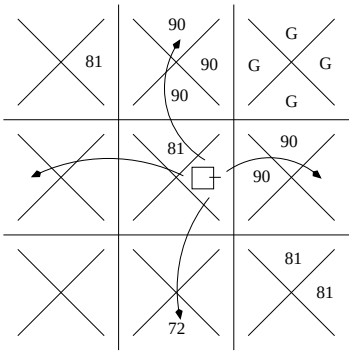

Figure 4.2: The resulting state of a movement action in the game. The number are Q values, propagating from the goal state G, with reward 100, and γ 0.9. The cross symbolises the four directions the agent can face

4.1 No opponent

The purpose of training in this section is to examine the importance of the individual rewards given throughout the game. The graphs in this chapter, if nothing else is stated, shows the accumulated number of actions of for the agent to finish a game.

When playing without an opponent, the state space for the agent is its position, its direction and whether it has the flag or not. This gives $100 \times 4 \times 2 = 800$ states. As there is no opponent, the *shoot* action is not necessary, therefore, the actions available for the agent are: *north*, *south*, *east*, *west*, *pickup* and *putdown*. The tests are made with different sets of increasing numbers of rewards. The sets are:

a:
- Reward for putting down flag at base = 100

b:
- Reward for putting down flag at base = 100
- Reward for losing flag = -10

c:
- Reward for putting down flag at base = 100
- Reward for losing flag = -10
- Reward for picking up flag = 10

d:
- Reward for putting down flag at base = 100
- Reward for losing flag = -10
- Reward for picking up flag = 10
- Reward for hitting wall = -1

Though the exact values of the individual rewards are not important, some of the relative values can have an influence on the learning. If the reward for *losing the flag* is smaller than the reward for *picking up the flag* the agent will, most likely, learn that the best policy is to stand at the opponent's base and repeatedly pickup and drop the flag, as it will give the highest Q value.

4.1.1 Exploitation

The exploitation strategy for training the agent works by always selecting the action that returns the highest Q value, and if two or more gives the same highest value, a random of these is selected. This means that the agent will always seek the known way to the goal state. The agent is trained with the sets of rewards listed above using an exploitive action selection strategy. Figure 4.3 shows the test results for all sets. Figure 4.4 shows the test results for sets **b, c** and **d**. Figure 4.5 shows a closeup of the converging of the four sets.

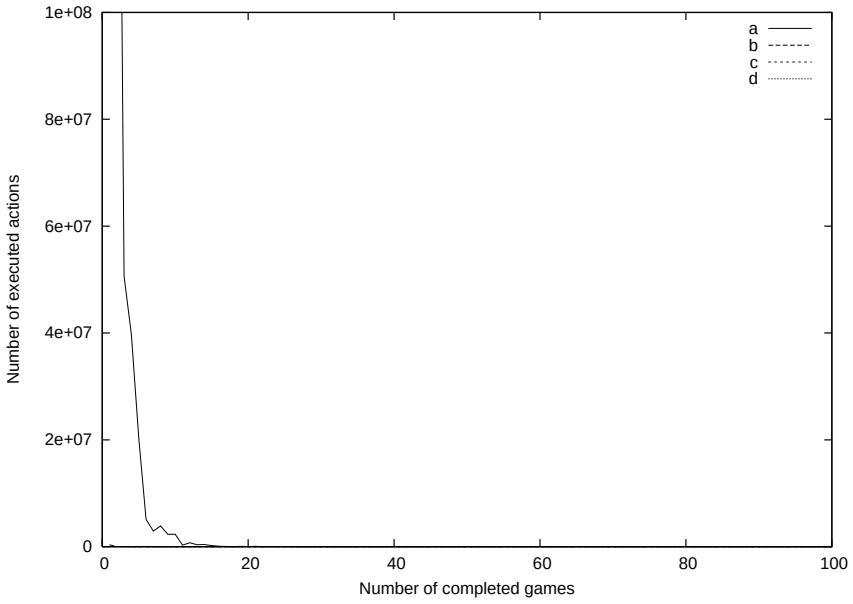

Figure 4.3: Result from training the agent without an opponent using exploitation. The results for sets **b, c** and **d** are hardly visible, because **a** take far more actions to finish. The graph for **a** is made from an average of 10 trials, and the rest are made from 50 trials

Comparing Figure 4.3 and Figure 4.4, there is a noticiable difference. The agent training with set **a** stands clearly out, by needing several million actions to finish the first few games. This is because the agent playing with set **a** will only learn when it finishes the game. Playing with set **b** finishes the first game with far less

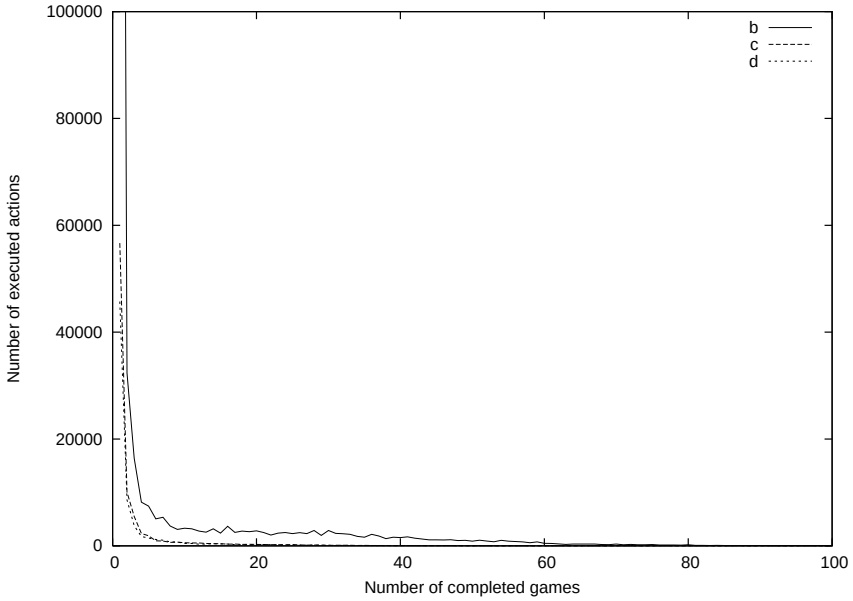

Figure 4.4: Results from training the agent without an opponent using exploitation. The sets shown are **b, c** and **d**. The graphs are made from an average of 50 trials

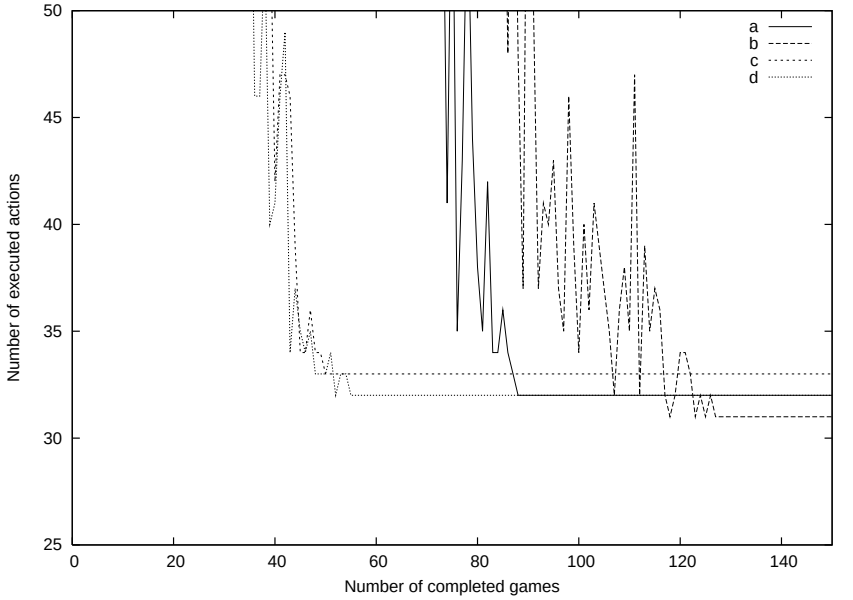

Figure 4.5: Closeup of the point of converging for all set using exploitation.

actions, but it takes more games to converge. As **b** learns to not putdown the flag at other positions than the agent's own base, it will always reach the base after fewer

actions, but it does not propagate the information faster. That **a** converges faster than **b**, may be explained as the test for **a** was made with fewer trials, and may therefore, to some extent, be noise.

However, the main difference is made by the reward for picking up the flag, **c**. Not only does it finish the first game much faster than without the reward for picking up the flag, it also converges in about half the number of games. This can be explained as the agent will potentially learn the way to the flag during the first game, as it, everytime it puts down the flag at another position than its own base, can propagate the reward for picking up. Therefore, after a few games, the agent may only have to learn the way from the flag to its base. The difference between **c** and **d** is insignificant, which indicates that the effect for the reward for hitting the wall is minimal.

4.1.2 Exploration

The explorative strategy for selecting an action allows the selection of an action that does not have the highest Q value. For training in this section the method described in section 3.3 is used.

$$P(a_i|s) = \frac{k^{\hat{Q}(s,a_i)}}{\sum_j k^{\hat{Q}(s,a_j)}}$$

Figure 4.6 shows a result of training the agent with the set **d**, using values of k of 5, 10 and 20. The graphs show the number of actions to finish the game. The graph for $k = 20$ shows that the agent, having reached the goal, does not choose to explore, why the behaviour is basically exploitive. The graph for $k = 5$ shows that the agent does require less actions to finish, but still explores extensively. During the tests, a k value is set to 5. This favours exploration, and does only exploit the known information slightly. Since the purpose of an explorative strategy for selecting an action is to explore, it may, depending on k, not converge in the sense that it will settle on a constant number of actions for reaching the goal. Therefore, the tests are made, such that the first 125 games use an explorative strategy, and thereafter an exploitive strategy.

The agent is trained with the sets of rewards listed above using an explorative action selection strategy. Given the poor performance of **a**, only **b, c** and **d** are used. Figure 4.7 shows the results for sets **b, c** and **d**. Figure 4.8 shows a closeup of the point of convergence for all three sets.

The overall results from training using an explorative action selection strategy are very similar to the results from using the exploitive strategy. Though training with neither set, converges to a fixed value, **c** and **d** appear to converge after about 40

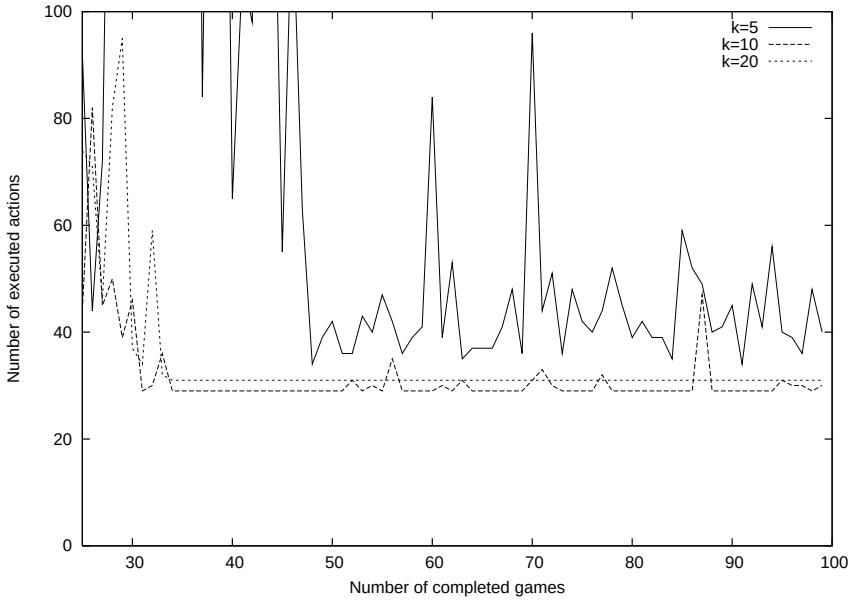

Figure 4.6: Comparing the behaviour of exploration with different values for k

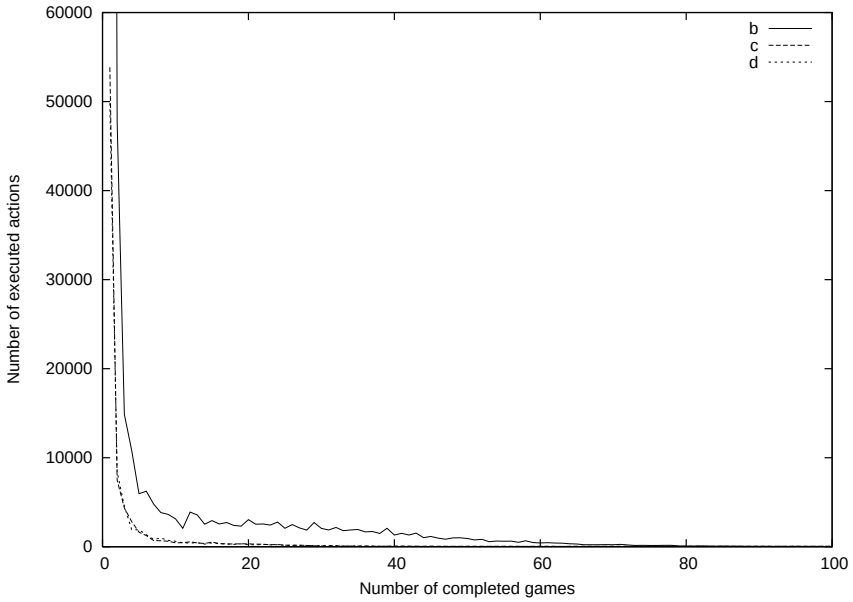

Figure 4.7: Results from training the agent without an opponent using exploration. The graphs are made from an average of 50 trials

finished games. Again **b** is slower to converge than **c** and **d**. However, Figure 4.8 shows that there is a great difference between using the explorative strategy for

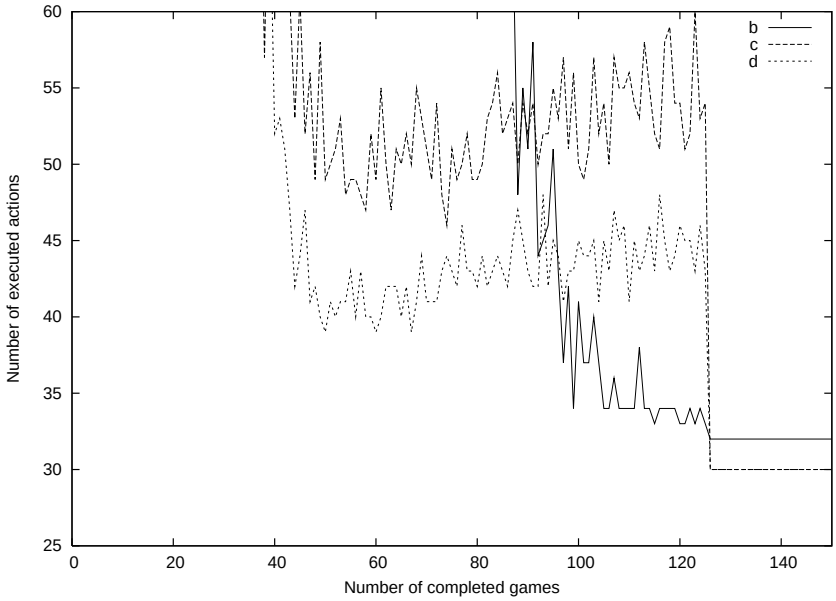

Figure 4.8: A closeup of the converging of the test using an explorative strategy

action selection (first 125 games), and the exploitive (from 125 games). However, the figure also shows that the policy found by exploring is not necessarily better that the exploitive.

Based on the results, it is clear that the *pickup* reward has a great impact on the performance, by, more or less, halfing the number of trials necessary to converge. Therefore, and though the *hit wall* reward shows little importance, it is chosen to use all available rewards in the following section. The difference between using exploitation and exploration for selecting an action, seems very small, as the policies obtained converges at about the same amount of executed actions.

4.2 Playing against the opponent

Training the agent against the opponent will show both how the agent performs in a changing environment, and also how it can perform against an opponent that can try to win, and will interfere with the agent. When playing against the opponent, the state space is expanded with the position, direction and flag of the opponent. This means that the statespace includes $800 \times 100 \times 4 \times 2 = 640000$ states. When playing against the opponent, the shooting action is also included.

The results of training the agent against an opponent, are compared to the results

of having a trained agent play against the opponent. This is done by using a policy for the agent found by training it without an opponent, and expanding it to cover all positions of the opponent for each state. The policies for used are trained with the rewards in set **d**. Playing against the opponent using a known policy, will also indicate the usefulness of the action selection policies in a non training scenario.

The $\hat{Q}$ learning rule in Figure 4.1 will always overwrite the old $Q(s, a)$, and is therefore mainly suitable when the agent plays without the opponent. Therefore, to be able to handle the uncertainty of the, non-optimal, agent's shooting, it is necessary to expand the $\hat{Q}(s, a)$ expression with the learning rate α.

$$\hat{Q}(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{s} \hat{Q}(s', a')] \tag{4.1}$$

$$\alpha = \frac{1}{1 + visit_s}$$

This will make the agent update the Q value depending on the number of times the agent has been in the state. This means that if playing against a very random opponent, i.e. an agent with high probability for selecting a random action, the agent can cross the opponent's line of fire a number of times without being shot. Without using (4.1), being shot in that situation will ignore the number of successful crossings, and make all but the most determined explorative agent avoid the situation.

The tests made training the agent by playing it against the opponent are made with the reward set **d**. Added to this are the rewards for both shooting the opponent and being shot by the opponent. This means that the set used when training against the opponent are:

e: Training by playing against an opponent, with rewards

- Rewards in set **d**
- Reward for being shot = -10
- Reward for shooting opponent = 10

f: Adopting a policy learned when not playing against an opponent, with rewards

- Rewards in set **d**
- Reward for being shot = -10
- Reward for shooting opponent = 10

The tests are again performed by using both exploitation and exploration.

Since the agent cannot compete with an opponent with an optimal strategy, it will instead play against opponents that are at different levels from begin optimal. This is done by using opponents that have 5 and 9 random actions out 10, as described in Section 2.2.

4.2.1 Exploitation

The exploitive strategy for training the agent is similar to the one described in section 4.1. The results from set **e**, training without previous knowledge is shown in Figure 4.14. The results from set **f**, playing using a known policy if shown in Figure 4.11. In both cases, the number indicates the randomness of the opponent.

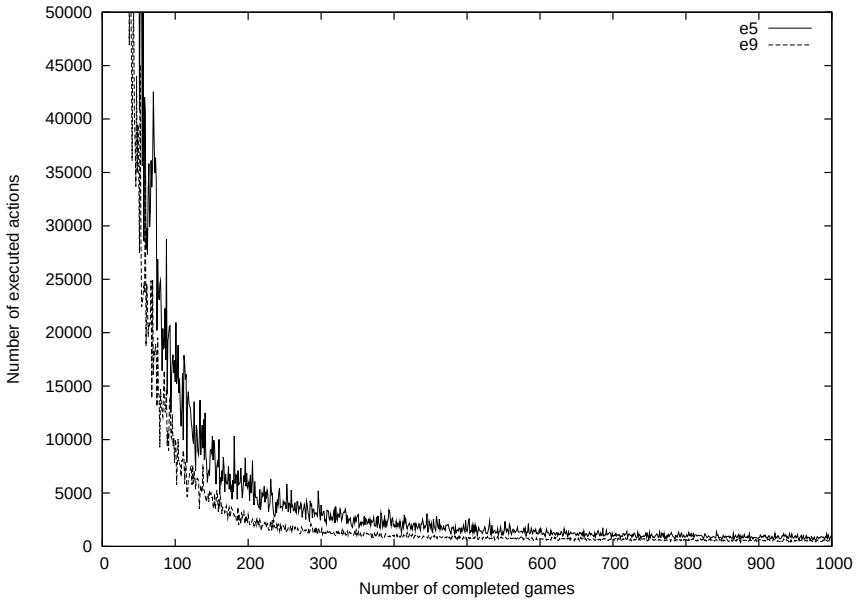

Figure 4.9: Result from training the agent against an opponent using exploitation. The graphs are made from an average of 50 trials

Training the agent, Figure 4.14, shows that the agent performs sligthly better against the most random opponent, **e5**. Though it is counter-intuitive, that the agent playing against the most random opponent should converge the fastest, an explanation can be that the agent get to perform many actions in the early part of the game, and therefore may manage to update enough states to be able to enter them again, regardless of the opponent's action. However, it may converge faster, because it plays many initial games. Another explanation can be that the less random agent simply plays better, and therefore wins the games faster, and thereby gives the agent less time to learn.

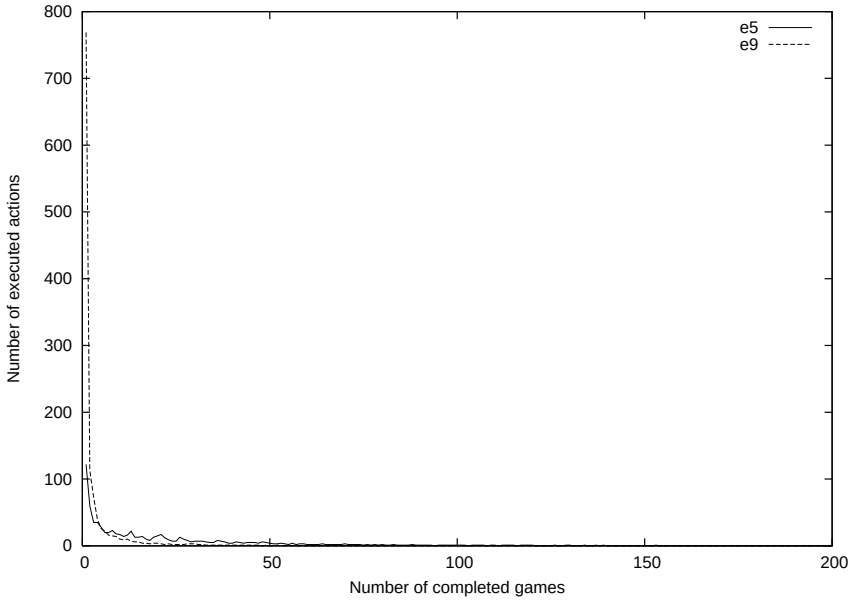

Figure 4.10: Number of games lost, during training against opponent with 1 random action. The graphs are made from an average of 50 trials

The results from playing with a policy obtained by training without an opponent, shows that the agent performs best against the near-optimal opponent. This can be explained, as the games were played using the same policy, and may not bring the agent and the opponent together. Comparing the graphs for the other opponents, shows that the agent performs slightly better against the least random. This can be explained by the least random will spend more time near the path of the agent, and therefore cause more interaction. Figure 4.12 shows that the agent was shot most by during set **e5**, which was the set that performed worst. Both Figure 4.11 and Figure 4.12, also shows that the agent has not improved its performance while playing against an opponent. That the agent does not learn to avoid being shot, can partly be explained by the use of the learning rate, α. This will make the change more gradual, and can therefore inhibit the negative reward in a state.

4.2.2 Exploration

The explorative policy used to train the agent to play against the opponent, is similar to the one described in section 4.1. The results from set **e**, training the agent against the opponent, are shown in Figure 4.13. The results from playing against the opponent using a known policy, are shown in Figure 4.15.

The results for training the agent against the opponent using exploration are su-

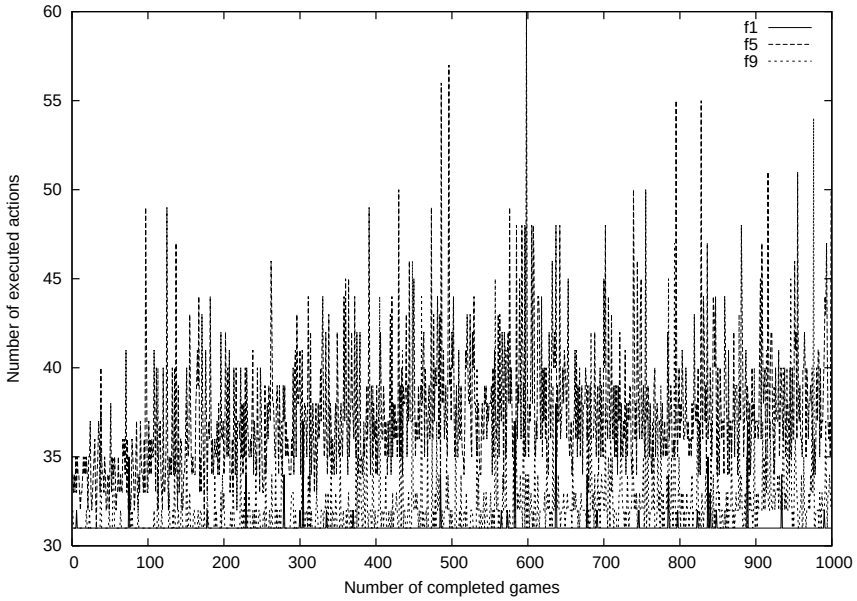

Figure 4.11: Result from playing with a policy obtained by training without an opponent, and using an exploitative action selection policy.

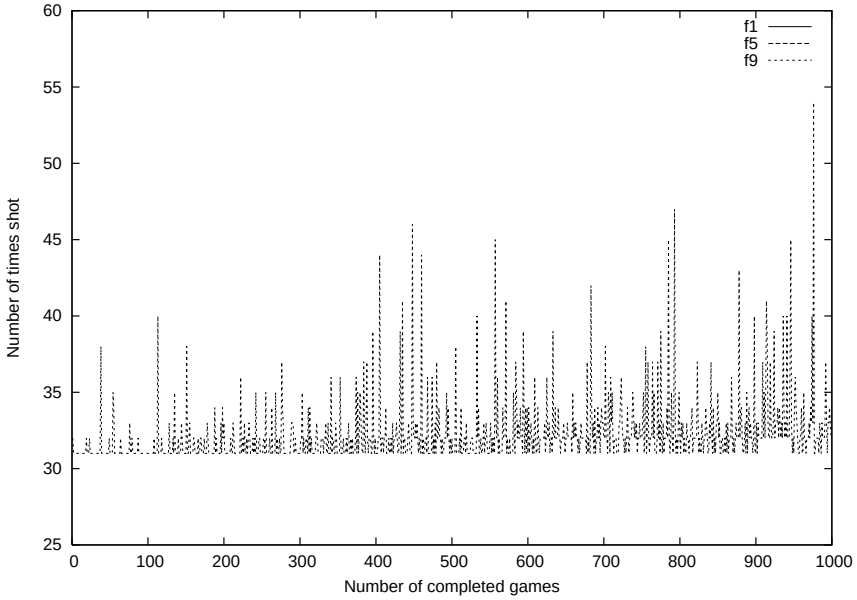

Figure 4.12: Number of times the agent has been shot, while playing with a policy obtained by training without opponent.

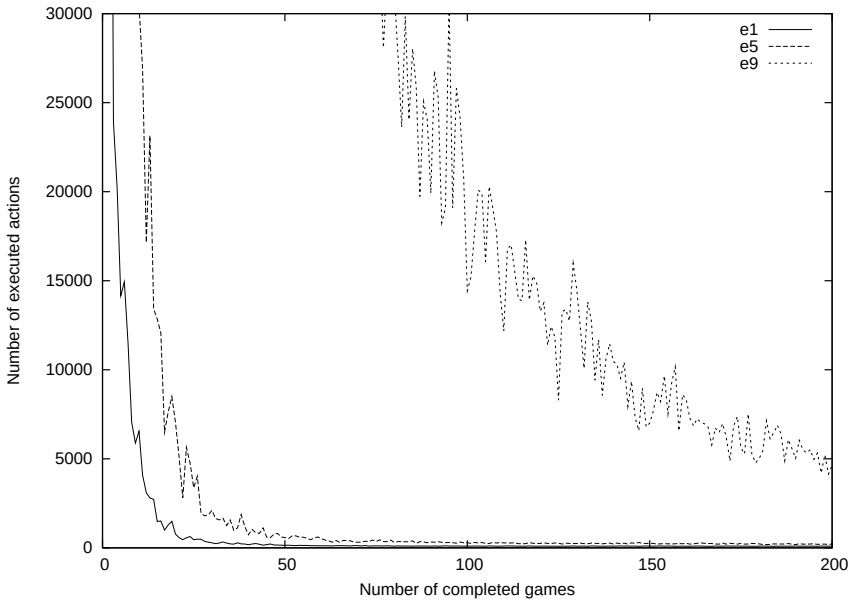

Figure 4.13: Result from training the agent against an opponent using exploration. The graph is a average of 50 trials

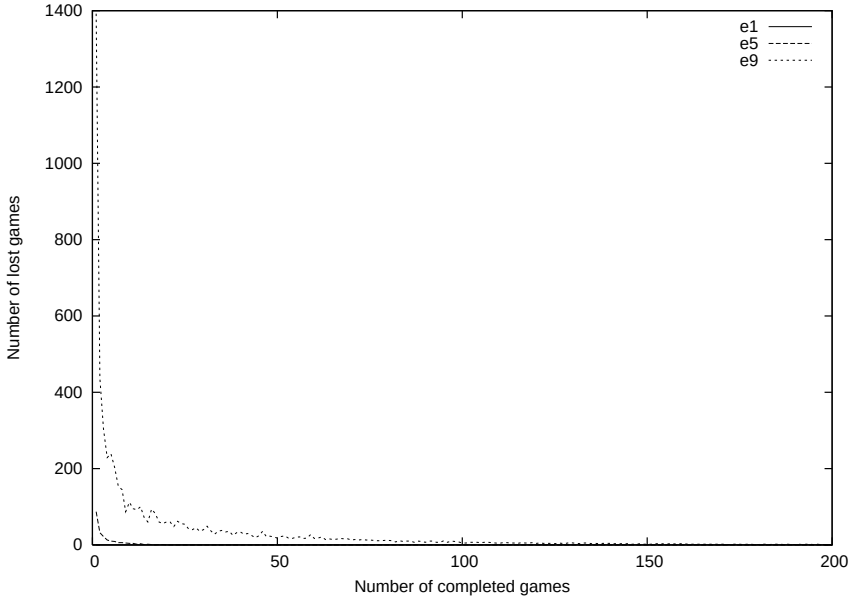

Figure 4.14: Number of games lost, during training against opponent with, using exploration

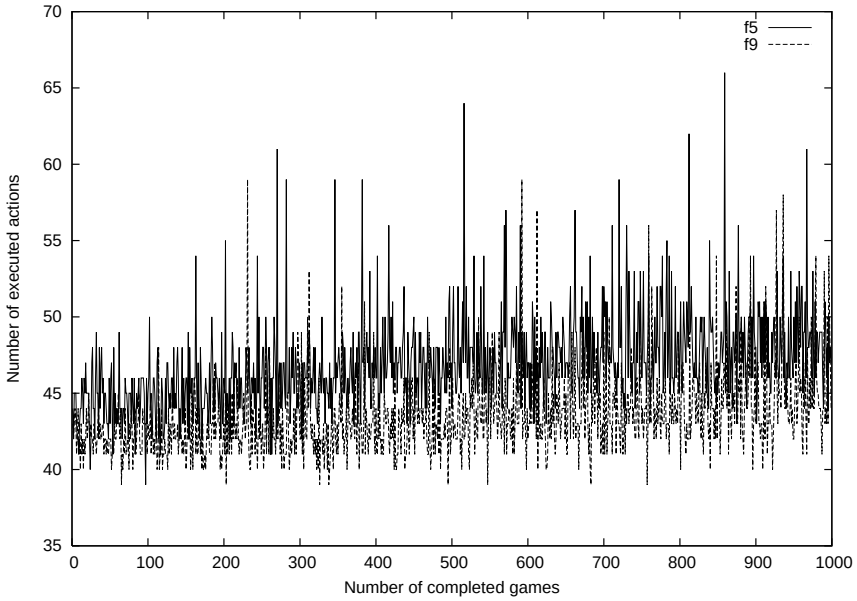

Figure 4.15: Result from playing the agent against an opponent using a policy obtained by training without an opponent.

perior to the similar results for exploitation. The best performance is against the least random opponent, which is because this allows it to more often reach a known state, and therefore propagate the information of the rewards. The sets **e1** and **e5** appear to converge after a number of games comparable to not having an opponent, whereas **e9** does not appear to have converged after 200 won games. The reason for the explorative agent performing so well, is that the exploraitive action selection allows it to explore several new states, during a single game.

Playing against the opponent using a known strategy is comparable to the performance using exploitation. Though the agent finishes all games after no more than about twice the number of actions as for playing against no opponent, it does not appear to learn adapt. Because the policy was trained against no opponent, it did not include rewards for shooting. However, the agent, even using an explorative policy, did not select to shoot the opponent.

Based on the performance of the agent in playing against an opponent, it can be concluded, both that there is a difference between using an explorative strategy for action selection, and an exploitive strategy. Another point for concluding is that it is not advantageous to train the agent without an opponent. The results in Figure 4.15 show that the agent does not manage to alter its behaviour to include the opponent. Therefore, even though the number of actions used with the known policy, the best way to train the agent, is to let it learn to play against the opponent,

and thereby learn to interact with it.

4.2.3 Summary

The Q learning method was examined regarding its ability to learn and to adapt. The ability to learn was tested by seeing how dependent it is on the rewards available, and the difference in learning between using an exploitive action selection policy and an explorative policy. It was found that the agent implemented with the Q learning method was very dependent on the rewards available, but also that applying rewards to divide the learning into smaller parts can more or less make the agent converge twice as fast, and finish the first game much faster. Comparing the importance of the action selection strategy showed that there was little difference in the obtained policy without an opponent, and that neither was optimal. However, when training against an opponent, the explorative policy proved superior, by converging about twice as fast for against a mostly random opponent, and even faster against more less random opponent. The ability to adapt was tested by training the agent without an opponent, and then playing it against an opponent. Neither the agent using an exploitive action selection policy, nor the one using an explorative policy learned to adapt to the opponent. The explorative agent did not manage to shoot.

Chapter 5

MaxQ model

In this chapter the agent is trained using the MaxQ method. This is done by first training it without an opponent, and secondly playing against the opponent introduced in Section 2.2, at different levels of randomness.

The MaxQ algorithm used is the MaxQ-0 algorithm, which is shown in Figure 5.1 [Die00]. The MaxQ-0 algorithm distinguishes between primitive and non-primitive tasks. The primitive tasks, or the actions, are executed immediately, and the immediate reward $V(a, s)$ is stored for the state. The non-primitive tasks are made up of other tasks, which are executed. This continues until a primitive task is reached and executed.

The non-primitive task updates the value till terminating. This means that information travels upwards in the tree. This also means that a task will require a reward for reaching the terminating state to be able to learn a policy. This can be seen as an unfair advantage compared to the Q learning, and all comparisons between the two should be done with the set for Q learning where there is a reward for pickup. However, it also reflects the difference between the two methods. Because the MaxQ method uses functional decomposition it is already limited to performing the given subtask. Therefore, even if it is not learning, it will always terminate the navigation task when reaching the terminal state.

The action selection is done using pure exploitation by chosing the action a that has the highest Q-value. The Q value, is the sum of V, the immediate value of executing an action, and C, the value for completing the task after a has been executed.

$$Q(i, s, a) = V(a, s) + C(i, s, a)$$

```
function MaxQ-0(maxnode i, state s)
if i is a primitive Maxnode
    execute i, receive r, observe state s'
    V(i,s) = r_t
    return 1
else
    let count =0
    while T_i(s) is false
        choose action a
        let N=MaxQ-0(s,a)
        observe s'
        C(i,s,a) = γ^N V_t(i,s')
        count = count + 1
        s = s'
        return count
```

Figure 5.1: The MaxQ-0 learning algorithm [Die00]

If a is a primitive action, $V(a,s)$ is the reward for executing it. If a is a non-primitive action, V is the highest Q value for the subtask. This means, given the tree in Figure 3.6, $V(a,s)$ for Getflag is

$$V(get,s) = \max_a [V(a,s) + C(get,s,a)]$$

If the the highest value was for navigate, $V(get,s)$ is

$$V(get,s) = \max_a [\max_{a_{nav}} [V(a_{nav},s) + C(nav,s,a_{nav})] + C(get,s,a)]$$

5.1 No opponent

The purpose of training the agent to play without an opponent is to examine the influence of the individual rewards available for the agent. This is done by training the agent with different rewards. Training the agent to play the game using the MaxQ method, requires creating a task tree. The tasks needed are listed below, along with the terminating states and actions available for each.

Capture flag:
- $A_{captureflag} = \{Getflag, returnflagtobase\}$
- $T = \{putdownflagatbase\}$

Get the flag:
- $A_{getflag} = \{Navigatetoflag, pickupflag\}$
- $T = \{pickupflag\}$

Return flag to base:
- $A_{returnflagtobase} = \{navigatetobase, putdownflag\}$
- $T = \{notholdingflag\}$

Navigate to flag:
- $A_{navigatetoflag} = \{north, south, east, west\}$
- $T = \{reaching/standingatbase\}$

Navigate to base:
- $A_{navigatetobase} = \{north, south, east, west\}$
- $T_{navigatetobase} = \{reaching/standingatbase\}$

Ordering the tasks into a tree, gives MaxQ tree shown in Figure 5.2.

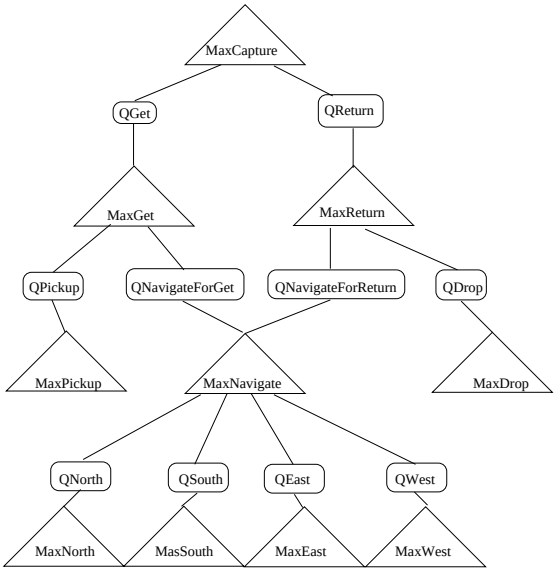

Figure 5.2: MaxQ tree for playing with no opponent

The sets of rewards used in the training are listed below. The individual rewards are the same as was used in chapter 4, except for the *reach* reward. The *reach* reward is given for reaching the terminate state for navigating. This is added to enable the agent to learn to navigate, as it would otherwise not be able to propagate information back through the navigation policy.

a:
- Win = 100

- reach = 1

b:
- Win = 100
  - reach = 1
  - pickup flag = 10

c:
- Win = 100
  - reach = 1
  - pickup flag = 10
  - lose flag = -10

d:
- Win = 100
  - reach = 1
  - pickup flag = 10
  - lose flag = -10
  - hit wall = -1

The results for training the agent with the rewards in set **a** and **b** are shown in Figure 5.3 and the results for sets **c** and **d** are shown in Figure 5.4.

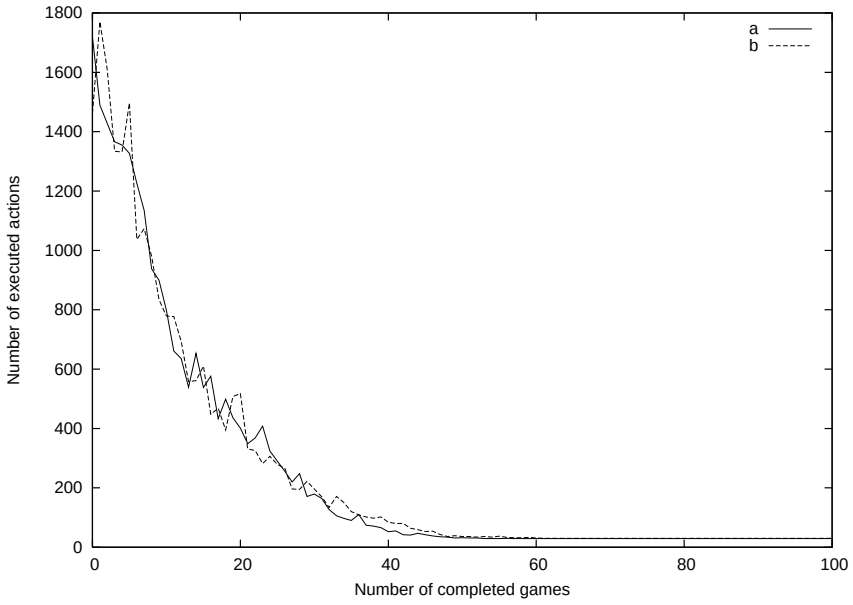

Figure 5.3: Training the agent with no opponent. The graphs were made from an average of 50 trials

The results for training the agent using the MaxQ algorithm show that the agent using all sets, **a** to **d**, converge after a number of games comparable to the better Q

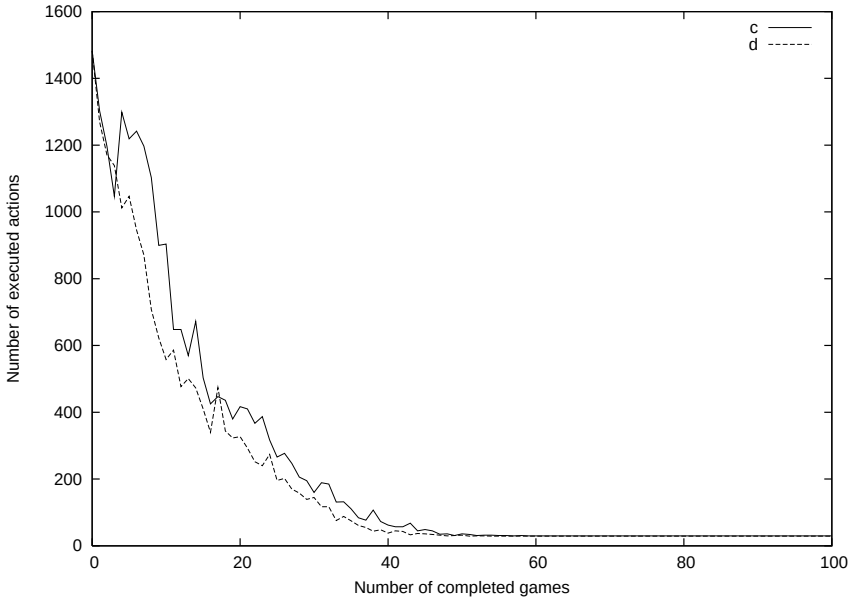

Figure 5.4: Training the agent with no opponent. The graphs were made from an average of 50 trials

learning tests. This can be explained as the MaxQ method, much like the Q learning with a reward for pickup, learns a policy for reaching a subgoal. It does not learn noticably faster than the Q learning, because the agent cannot drop the flag unless standing at either base. Therefore, the agent will only learn a new step towards the base, after putting it down at the base, and only propagate the information one step at the time.

The number of actions needed to finish the first game, using all sets of rewards, is a vast improvement compared to the Q learning. That they all finish the first game in less than 3000 executed actions, compared to 100,000 or even millions for Q learning, is because of the division into subtasks. Because the agent, after finishing reaching the flag, can only put it down while choosing between navigating to its own base, or putting down the flag, the agent can only lose the flag while standing at either base. Therefore, because an action is only performed when a primitive task is selected, the only 'vasted' action possible is the *pickup* action in the *get the flag* task, while standing at the agent's own base.

5.2 Playing against opponent

The purpose of playing against the opponent is both to examine how the agent can learn given the enlarged statespace, and also how it will perform against an adversary that will interfere with the agent. Whereas the first of these is primarily a navigation task that can be performed with the given model, the second requires a modification of the model to accomodate the interaction betwen the actors.

Specifying the model

The MaxQ tree in Figure 5.2 does not allow the agent to shoot. Therefore, it is necessary to expand that tree with another subtask, *Hunt opponent* at the same level af *Get the flag* and *Return flag to base*. *Hunt opponent* also requires a subtasks for navigating to a position that allows shooting at the opponent. This can be done by adding a QNode *QNavigate to opponent* which points to the Maxnode Navigate. *Hunt opponent* also requires the primitive action *shoot*.

The problem with adding another subtask is that the termination conditions for all the other subtasks will change to allow the new task to be chosen. In this case it is necessary to consider both the distance to the opponent, the opponent's distance to its base and the distance to the agent's own goal when giving chase. If the agent is too far away from a near optimal opponent it may not be able to reach the opponent before it has returned to its base, and the game is lost. If the agent gives chase every time the opponent holds the flag, and is closer to its homebase than the agent, the agent playing against a near optimal opponent is dependent on its ability to hunt the opponent to win. This means that if every time the opponent is close to winning, the agent stops trying to return, it will not benefit from the rare random action by the opponent, and instead attempt to learn to shoot the opponent.

The tasks for playing against and hunting an opponent are shown below. The related MaxQ tree is shown in Figure 5.5. This was not implemented.

Capture flag:
- $A_{captureflag} = \{Getflag, returnflagtobase\}$
- $T = \{putdownflagatbase\}$

Get the flag:
- $A_{getflag} = \{Navigatetoflag, pickupflag\}$
- $T = \{pickupflag\}$

Return flag to base:
- $A_{returnflagtobase} = \{navigatetobase, putdownflag\}$
- $T = \{notholdingflag\}$

Hunt opponent:
- $A_{huntopponent} = \{navigatetoopponent, shoot\}$

- $T = \{notholdingflag\}$

Navigate to flag:
- $A_{navigatetoflag} = \{north, south, east, west\}$
- $T = \{reaching/standingatbase\}$

Navigate to base:
- $A_{navigatetobase} = \{north, south, east, west\}$
- $T_{navigatetobase} = \{reaching/standingatbase\}$

Navigate to opponent:
- $A_{navigatetoopponent} = \{north, south, east, west\}$
- $T = \{inpositionallowingtoshoottheopponent\}$

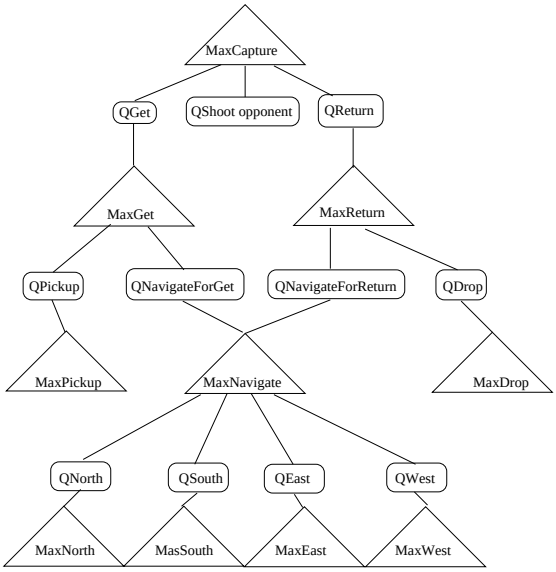

Figure 5.5: MaxQ tree for playing with an reactive agent

Testing

In this section the agent is tested, first by training it against opponents with 5 and 9 random actions out of 10. This is to examine how well the MaxQ deals with a larger statespace. Second, by applying it with a policy obtained by training without an opponent. This will show how well the MaxQ model adopts the policy to include the opponent. The tests in this section are performed using the model specified in section 5.1.

5.2.1 Without previous knowledge

As the agent cannot shoot the opponent, training it against the opponent, should make the agent try to win the game, and avoiding being hit. Training the agent to play against an agent is done with full set of rewards, and the reward for being shot.

e: Training the agent against an opponent

- reward set **d**
- being shot = -10

Figure 5.6 shows the number of actions to win a game for training the agent playing against an opponent with 5 random actions. Figure 5.7 shows the similar results for training the agent against an opponent with 9 random actions. Figure 5.8 shows the number of times the agent lost before begin able to win a game.

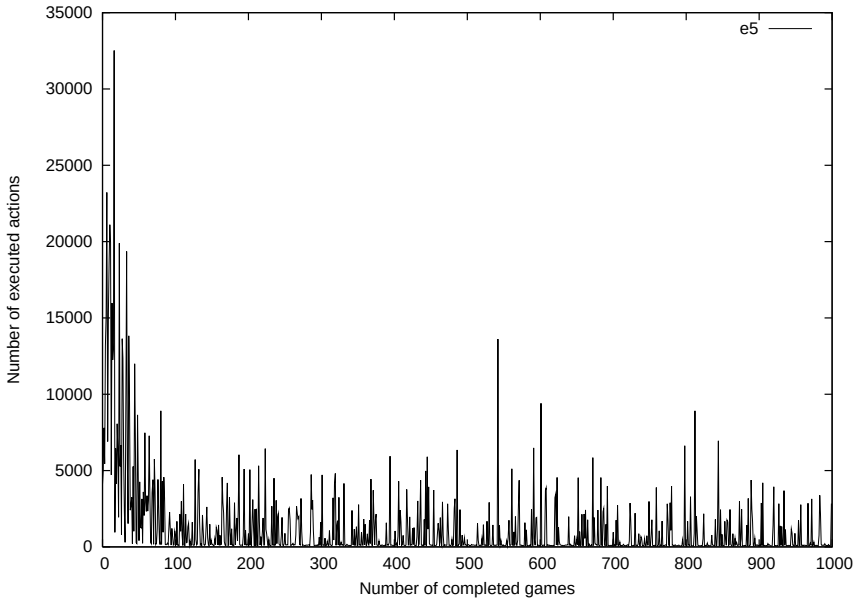

Figure 5.6: Results from training the agent against an opponent with 5 random actions. The graph was made from an average of 50 trials

Comparing Figure 5.6 and Figure 5.7 shows that the agent moves towards converging faster for the least random agent. It requires more states in the initial games, because the least random actions will finish the game faster than the most random, and therefore the game is restarted more often. However, bacause the opponent is less random, the agent will more often meet known states.

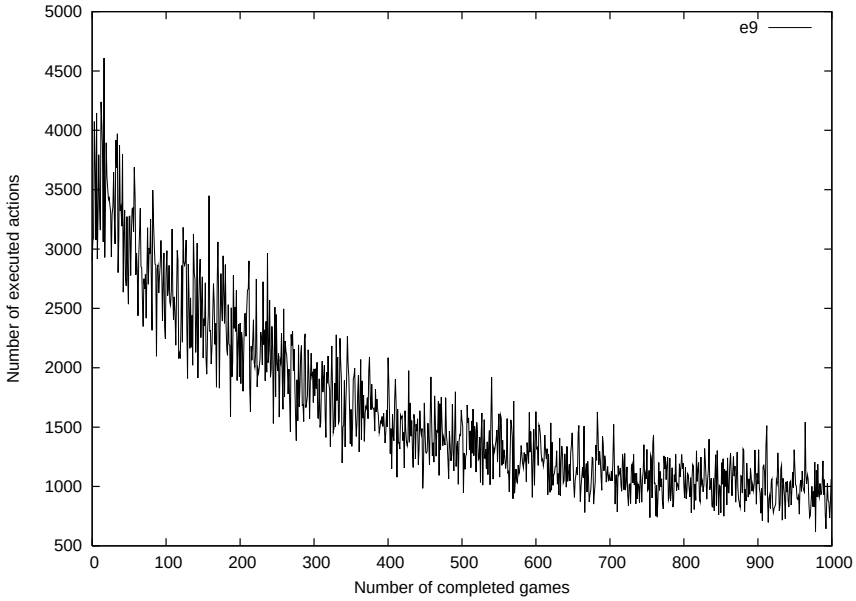

Figure 5.7: Results from training the agent against an opponent with 9 random actions

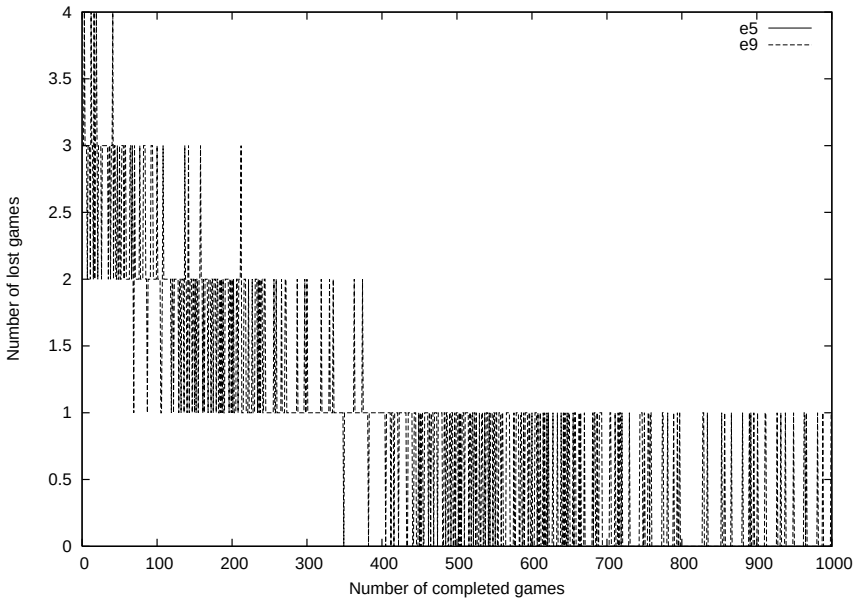

Figure 5.8: Number of lost games, while training the agent against an opponent

5.2.2 With previous knowledge

Playing with an agent that has been trained without an opponent, will show how the agent can adapt to a new situation. The policy for the agent was obtained by

training it with the full set of rewards, **d**, as it was found to be the set that gave the best performance. The set of rewards used for playing against the opponent is shown below.

f:
- win = 100
  - reach 1
  - pickup flag = 10
  - lose flag = -10
  - being shot = -10

Figure 5.9 shows the number of primitive actions executed before the agent finishes a game, when playing against the opponent with 5 random action. Figure 5.10 shows the number of primitive actions for playing against the opponent with 9 random actions. Figure 5.11 shows the number of times the agent was shot before finishing a game, playing against the opponent with 5 random actions.

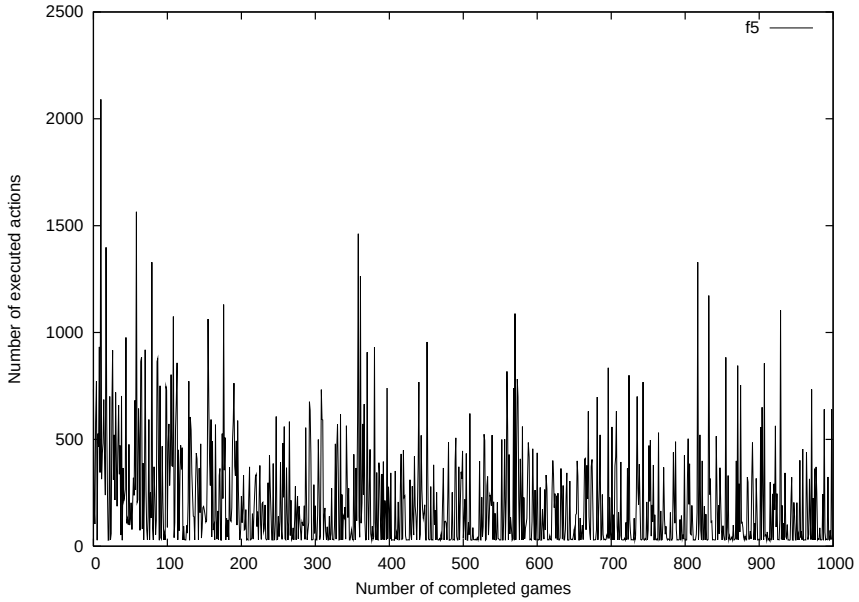

Figure 5.9: Results from playing the agent, with a policy for no opponent, againt an opponent, with 5 random actions. The graph was made from an average of 50 trials

Figure 5.9 shows that the agent does not adapt well to the addition for the opponent. Compared to Figure 5.10, it is seen that the agent performs best against the most random opponent. Figure 5.11 shows that the agent appeared to adapt to avoiding being shot. The many actions may be explained as the known policy is weakened

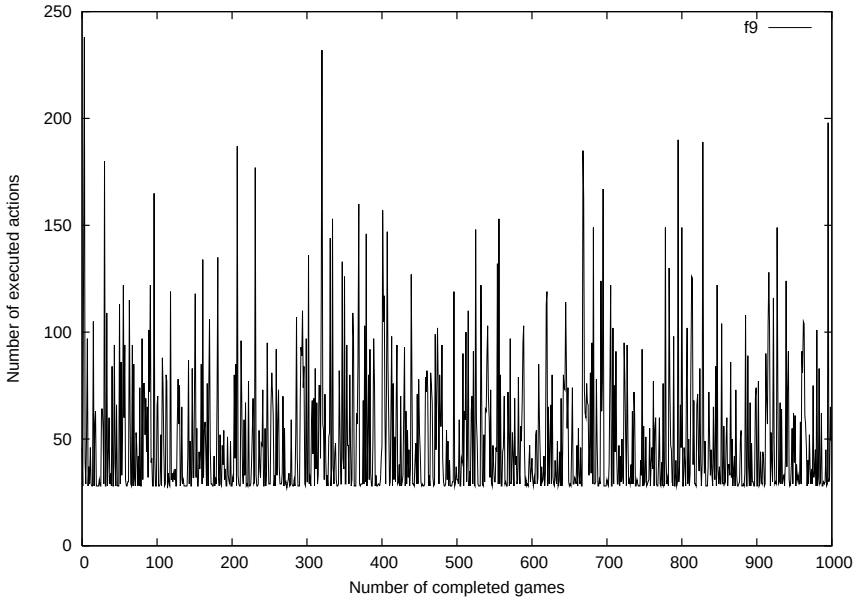

Figure 5.10: Results for playing the agent, with a policy for no opponent, against an opponent with 9 random action. The graph was made from an average of 50 trials

because the agent is shot at different positions on it, and therefore try to avoid those.

5.3 Summary

The MaxQ was found to need ar extra reward to be able to learn to navigate. This reward was given for reaching a terminal state in the navigating subtask. The MaxQ learning method was examined regarding its ability to learn, and to adapt. The ability to learn was examined be testing the it without an opponent, with different sets of rewards, and against an opponent with the full set of rewards. This showed that the MaxQ is not very dependent on the rewards, though it would not be able to converge without the reward for terminating the navigate subtask. Playing against an opponent required adding a new subtask, to allow the agent to shoot the opponent. However, this has not been implemented. Playing against the opponent, showed that the MaxQ performed comparable with the Q learning agent using an exploitive strategy for action selecition. The ability to adapt was tested by training the agent without an opponent, and using that policy to play against an opponent. This showed that the agent could adapt to avoid being shot, but did not appear to converge to a fixed level of actions.

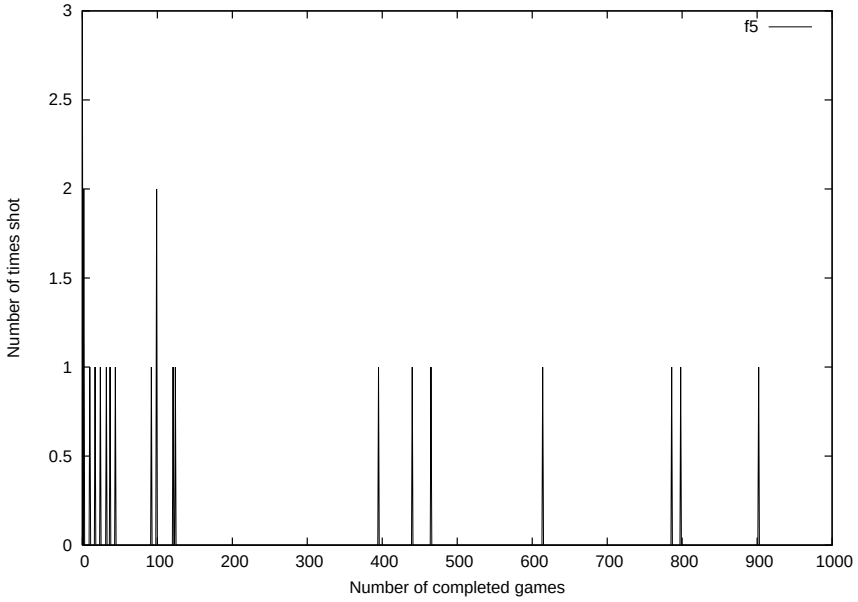

Figure 5.11: Number of times the agent, with a policy for no opponent, was shot while playing against the opponent with 5 random action. The graph was made as an average of 5 trials

Chapter 6

Implementation

In this chapter I will describe the implementation of the game, and the learning methods used. I will first go through the overall structure of the game, and second describe the Q learning, and third the MaxQ learning.

6.1 Implementation of the game

The game has been implemented to function as simple as possible, as it is not part of the problem to be solved. The game consists of the board, the opponent, and a class for handling the graphics.

The board class is the central class, which handles the actions made by the actors, and returns the rewards. To be able to handle collisions, pushing and shooting, the board stores the positions, direction and flag status of both agent and opponent.

The opponent is called with the positions of both actors. Based on this, it selects an action based on the rules in Figure 2.3 in Section 2.2. The go to checkpoint function is shown in Figure 6.1.

The game was implemented using C++. This was chosen both because it is a language I have had some experience in using, and, because it is object oriented, was suitable for handling the task.

- if south of checkpoint and north possible then **north**
- else if north of checkpoint and south possible then**south**
- else if east of checkpoint and west possible then **west**
- else if west of checkpoint and east possible then **east**

Figure 6.1: Pseudocode for the goto function used to navigate the opponent towards a checkpoint

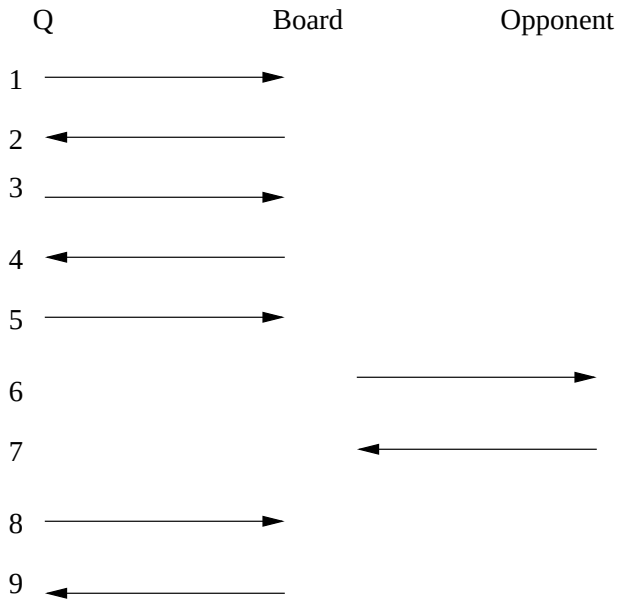

Figure 6.2: Overview of game process for Q learning

6.2 Q learning

The agent using Q learning is implemented as a single class. It maintains the statespace as an array of State structs, which again contains a the number of visits to the state, and an array for storing the Q values for each action. The statespace is ordered by the positions, direction and flag status of each actor, and before selecting an action, the agent requests these positions from the board. The flow of information between the agent and the board is shown in Figure 6.2.

The individual points in Figure 6.3 are specified below.

1. The agent method requests the state s_t
2. The board returns the state s_t
3. The agent method executes its action a
4. The board returns the reward r, and resulting state s_{t+1}
5. The agent method requests the opponent to execute an action
6. The board sends the state s_{t+1} to the opponent, and requests its action a
7. The opponent returns its action a
8. The agent requests the possible reward of the opponent's action
9. The board returns the possible reward

6.3 MaxQ learning

The MaxQ learning method is implemented in a single class. Therein, the Max and Q nodes are put together and represented as methods. To maintain the structure of the Max Q tree, the navigate Q nodes are represented as individual methods, that calls the navigate method, with the task it has to perform. The statespace in implemented as an array of structs representing the individual states. The state includes the individual C- tables, and $V(s, a)$ for the primitive actions.

Running the game in a number of subtasks, makes it important to ensure that the opponent move in only executed once per turn in the game. To ensure this, the game processes the information as shown in Figure 6.3.

The individual points in Figure 6.3 are specified below.

1. An agent task requests the state s_t
2. The board returns the state s_t
3. The agent task selects an action. This is continued until a primitive action is selected
4. A primitive action executes an action a
5. The board returns the reward r and state $st + 1$
6. The primitive actions returns the number of executed action to parent task
7. If subtask is not terminated, it requests an opponent move

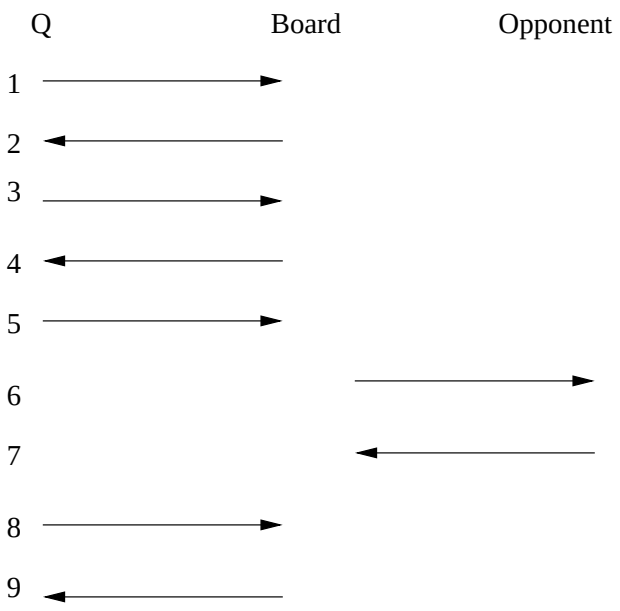

Figure 6.3: Overview of game process for MaxQ

8. The board requests an opponent move
9. The opponent returns an action

Chapter 7

Conclusion

The purpose of the project was to investigate the performance of reinforcement learning in a computer game context. This was specified to concern an agent's, implemented using reinforcement learning, ability to learn and to adapt to a new situation. It was decided to compare the performance of a traditional reinforcement learning method, Q learning, to a hierarchical method. Following a brief comparison, the MaxQ method was chosen.

A game, Flag Hunter, was designed to function as an environment for the agent to function in. Flag Hunter was made to be a sequential and turn based game, which is played against an opponent. The purpose of the game was to capture the flag from the opponent's base and return it the agent's own base. Since it would not be possible to beat an opponent using an optimal policy, it was made possible to set the degree of randomness of the opponent.

Implementing the agent with both reinforcement learning methods, the ability to learn was tested by training the agent to navigate in the map of the game using different rewards, and by training it against different opponents. For Q learning, the difference between using an exploitive and an explorative action selection strategy was also examined. Training the agent to reach the base using different rewards showed clearly that the MaxQ needed far less actions to reach a goal state. By dividing the problem into smaller subproblems it reached the first goal state more than 10 times faster than the Q learning. For the Q learning it was clear that the number of rewards played a large role for the performance. Training with only a reward in the goal state of the game, proved very hard to finish. However, both methods converges after about the same number of games.

Playing against an opponent showed that the agent using Q learning with an explorative strategy for action selection, converges significantly faster than both the exploitive Q learning agent, and the agent implemented using MaxQ, though the

MaxQ again finished the first games faster. To allow the agent using the MaxQ method to shoot, would require the addition of a new subtask to the model. This subtask was not implemented, and it was therefore only possible to test the MaxQ agent's ability to avoid being hit.

The ability to adapt was tested by letting an agent that was trained in a scenario without an opponent play against an opponent. This would require the agent to be able to change its behaviour if interacting with the opponent. Since being shot gives a negative reward, it should be able to avoid the situation that can lead to being shot. Neither method proved very successful in avoiding the opponent.

Based on the performance of the methods in the different test, in can be concluded that they have their different strenghts and weaknesses. MaxQ finishes the initial games very fast compared to Q learning, but does not appear to handle playing against an opponent as well as Q learning, with a explorative action selection strategy. However, it must also be concluded that none of the methods were able to adapt satisfiably.

Bibliography

[Die00] Thomas G. Dietterich. Hierarchical reinforcement learning with the MAXQ value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[HMK+98] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Uncertainty in Artificial Intelligence*, pages 220–229, 1998.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[Rab02] Steve Rabin. *AI Game Programming Wisdom*. Charles River Media, Inc., 2002.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 2003.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.