# Monte Carlo Tree Search with Options for General Video Game Playing

**MSc Thesis** *(Afstudeerscriptie)*

written by

**Maarten de Waard**
(born November 14th, 1989 in Alkmaar, the Netherlands)

under the supervision of **dr. ing. Sander Bakkes** and **Diederik Roijers, MSc.**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

**MSc in Artificial Intelligence**

at the *Universiteit van Amsterdam.*

| **Date of the public defense:** | **Members of the Thesis Committee:** |
|---|---|
| *February 22nd, 2016* | dr. Maarten van Someren |
| | dr. Piet Rodenburg |

UNIVERSITEIT VAN AMSTERDAM

**Abstract**

General video game playing is a popular research area in artificial intelligence, since AI algorithms that successfully play many different games are often capable of solving other complex problems as well. "Monte Carlo Tree Search" (MCTS) is a popular AI algorithm that has been used for general video game playing, as well as many other types of decision problems. The MCTS algorithm always plans over actions and does not incorporate any high level planning, as one would expect from a human player. Furthermore, although many games have similar game dynamics, often no prior knowledge is available to general video game playing algorithms. In this thesis, we introduce a new algorithm called "Option Monte Carlo Tree Search" (O-MCTS). It offers general video game knowledge and high level planning in the form of options, which are action sequences aimed at achieving a specific subgoal. Additionally, we introduce "Option Learning MCTS" (OL-MCTS), which applies a progressive widening technique to the expected returns of options in order to focus exploration on fruitful parts of the search tree. Furthermore, we offer an implementation of "SMDP Q-learning" for general video game playing, which is the algorithm that is traditionally used in combination with options. Our new algorithms are compared to MCTS and SMDP Q-learning on a diverse set of twenty-eight games from the general video game AI competition. Our results indicate that by using MCTS's efficient tree searching technique on options, O-MCTS performs better than SMDP Q-learning on all games. It outperforms MCTS on most of them, especially those in which a certain subgoal has to be reached before the game can be won. Lastly, we show that OL-MCTS improves its performance on specific games by learning expected values for options and moving a bias to higher valued options.

# Contents

# Chapter 1

# Introduction

An increasingly important goal in decision theory is creating AI algorithms that are capable of solving more than one problem. A general AI, instead of a specific alternative, is considered a step towards creating a strong AI. An especially challenging domain of AI is general video game playing. Since many real-world problems can be modeled as a game, algorithms that can play complex games successfully are often highly effective problem solvers in other areas. Although decision theory and general video game playing are different research areas, both can benefit from each other. An increase in performance in general video game playing can be found by using algorithms designed for complex decision-theoretic problems, whereas the algorithms that are created for general video game playing can be applied to other decision-theoretic problems. Furthermore, applying decision-theoretic algorithms to a new class of problems, like general video game playing, can lead to a better understanding of the algorithms, and new insights in their strengths and weaknesses.

In the history of decision theory, an increase in the complexity of the games can be observed. Early AI algorithms focused on simple games like tic-tac-toe [14]. Focus later shifted to chess and even later to Go [27, 4]. Nowadays, many algorithms are designed for winning computer games. A lot of strategy games, for example, offer the player computer-controlled contestants. Recent research focuses on the earlier introduced general video game playing. A common approach is to use a tree search in order to select the best action for any given game state. In every new game state, the tree search is restarted until the game ends. A popular example is *Monte Carlo tree search (MCTS)*, which owes its fame to playing games [9], but is used for other decision-theoretic problems as well, e.g., scheduling problems or combinatorial optimization problems (see [3], section 7.8 for an extensive list).

A method to test the performance of a general video game playing algorithm is by using the framework of the *general video game AI (GVGAI)* competition [18]. In this competition, algorithm designers can test their algorithms on a set of diverse games. When submitted to the competition, the algorithms are applied to an unknown set of games in the same framework to test their general

applicability. Many of the algorithms submitted to this contest rely on a tree search method [19, 22, 25].

A limitation in tree search algorithms is that since many games are too complex to plan far ahead in a limited time frame, many tree algorithms incorporate a maximum search depth. As a result, tree search based methods often only consider short-term score differences and do not incorporate long-term plans. Moreover, many algorithms lack common video game knowledge and do not use any of the knowledge gained from the previous games.

In contrast, when humans play a game we expect them to do assumptions about its mechanics, e.g., pressing the left button often results in the player's avatar moving to the left on the screen. Players can use these assumptions to learn how to play a game more quickly. Furthermore, human players are able to have an abstraction layer over their action choices; instead of choosing one action at a time they define a specific subgoal for themselves: when there is a portal on screen, a human player is likely to try to find out what the portal does by walking towards its sprite (the image of the portal on the screen). The player will remember the effect of the portal sprite, and use that information for the rest of the game. In this case, walking towards the portal can be seen as a subgoal of playing the game.

In certain situations, it is clear how such a subgoal can be achieved and a sequence of actions, or *policy*, can be defined to achieve it. A policy to achieve a specific subgoal is called an option [28]. Thus, an option selects an action, given a game state, that aims at satisfying its subgoal. Options, in this context, are game-independent. For example, an option that has reaching a specific location in the game (for example a portal) as its objective selects actions using a path planning heuristic that will reach the goal location. The probability of this kind of subgoal being achieved by an algorithm that does not use options is smaller, especially when the road to the subgoal does not indicate any advantage for the player. For instance, in the first few iterations of MCTS, the algorithm will be equally motivated to move 10 steps into the direction of a certain game sprite as it will be to do any other combination of 10 actions.

An existing option planning and learning approach is *SMDP Q-learning* [8]. It was originally proposed for solving *semi-Markov decision processes (SMDPs)* which are problems with a continuous action time. It was later used in combination with options to navigate a grid world environment [28, 26]. The traditional Q-learning is adapted to apply the update rules for SMDPs to problems with a given set of options, in order to be able to find the optimal option for each game state.

However, SMDP Q-learning does not have the same favorable properties as Monte Carlo tree search. For instance, although they are both anytime algorithms (they can both return an action that maximizes their hypothesis at any time), SMDP Q-learning usually has to play a game several times before it can return good actions. In contrast, MCTS can return reasonable actions with less simulations. We expect that combining MCTS with the option framework yields better results.

Therefore, we propose a new algorithm called *option Monte Carlo tree search*

*(O-MCTS)* that extends MCTS to use options. Because O-MCTS chooses between options rather than actions when playing a game, we expect it to be able to plan at a higher level of abstraction. Furthermore, we introduce *option learning MCTS (OL-MCTS)*, an extension of O-MCTS that approximates which of the options in the option set is more feasible for the game it is playing. This can be used to shift the focus of the tree search exploration to more promising options. This information can be transferred in order to increase performance on the next level.

Our hypothesis is that because O-MCTS uses options, exploration can be guided to find out the function of specific game objects, enabling the algorithm to win more games than MCTS. In this thesis, we aim to incorporate the use of options in general video game playing, hypothesising that the use of options speeds up planning in the limited time algorithms often have before taking a decision. Furthermore, this thesis will explain how SMDP Q-learning can be implemented for general video game solving in the GVGAI competition.

The new algorithms are benchmarked on games from the General Video Game AI competition, against SMDP Q-learning and the Monte Carlo tree search algorithm that is provided by that competition. For these experiments, a specific set of options has been constructed which aims to provide basic strategies for game playing, such as walking towards or avoiding game sprites and using a ranged weapon in a specific manner. Our results indicate that the O-MCTS and OL-MCTS algorithms outperform traditional MCTS in games that require a high level of action planning, e.g., games in which something has to be picked up before a door can be opened. In most other games, O-MCTS and OL-MCTS perform at least as good as MCTS.

# Chapter 2

# Background

This chapter explains the most important concepts needed to understand the algorithms that are proposed in this thesis. The first section describes *Markov decision processes (MDPs)*, the problem formalization we will use for games. The next section describes MCTS, a tree search algorithm that is commonly used on games and other MDPs. Subsequently, options will be formalized, these simulate the idea of defining subgoals and how to reach them. Then, the basics of Q-learning are described, after which SMDP Q-learning is explained. Finally the *video game description language (VGDL)* is explained. This is the protocol that is used by the GVGAI competition to implement the many different games the competition offers, each of which uses the same interaction framework with the game playing algorithms.

In this thesis games are considered to be an unknown environment and an agent has to learn by interacting with it. After interaction, agents receive rewards that can be either positive or negative. This type of problem is called reinforcement learning [31]. The concepts introduced in this chapter all relate to reinforcement learning.

## 2.1 Markov Decision Processes

In this thesis games will be treated as MDPs, which provide a mathematical framework for use in decision making problems. An MDP is formally defined as a tuple $\langle S, A, T, R \rangle$, where $S$ denotes the set of states, $A$ is the set of possible actions, $T$ is the transition function and $R$ is the reward function. Since an MDP is fully observable, a state in $S$ contains all the information of the game's current condition: locations of sprites like monsters and portals; the location, direction and speed of the avatar; which resources the avatar has picked up; etcetera. $A$ is a finite set of actions, the input an agent can deliver to the game. $T$ is a transition function defined as $T : S \times A \times S \to [0, 1]$. It specifies the probabilities over the possible next states, when taking an action in a state. $R$ is a reward function defined as $R : S \times A \times S \to \mathbb{R}$. In this case, when the

game score changes, the difference is viewed as the reward. Algorithms typically maximize the cumulative reward, which is analogous to the score. An MDP by definition has the *Markov property*, which means that the conditional probability distribution of future states depends only upon the present state. No information from previous states is needed. In the scope of this thesis algorithms do not have access to $T$ and $R$.

For example, for the game *zelda*, a state $s$ consists of the location, rotation and speed of the avatar and the location of the avatar's sword, the monsters, the walls and the key and portal that need to be found. $S$ is the set of all possible states, so all possible combinations of these variables. The action set $A$ consists of the movement actions UP, DOWN, LEFT and RIGHT, and USE, which in this case spawns the sword in front of the avatar for a couple of time steps. The transition function $T$ defines the transition from a state, given an action. This means that the transition defines the change in location of the monsters and the avatar and if any of the sprites disappear, e.g., when the avatar picks up the key. Note that, since the transition function is not by definition deterministic, the resulting state from an action $a$ in state $s$ is not necessarily the same state (a *non player character (NPC)* that moves about randomly can move in any direction between states independent of the agent's action). The reward function describes the change in game score, given a state, action and resulting next state. For example, when the avatar kills a monster with the action USE, its score will increase with 1.

An important trade-off in decision theory is the choice between exploration and exploitation. Exploration means to use the actions that have been used little and find out whether they lead to a reward. On the other hand, exploitation means to use the actions that you already know will lead to a reward. When an algorithm prioritizes exploitation over exploration too much, it has the risk of never finding unknown states which have higher rewards. It will keep exploiting the states with a lower reward that it has already found. In contrast, too much exploration might lead to lower rewards, because the algorithm takes the action that maximizes reward less often.

## 2.2 Monte Carlo Tree Search

Monte Carlo methods have their roots in statistical physics, where they have been used to approximate intractable integrals. Abrahamson [1] demonstrated theoretically that this sampling method might be useful for action selection in games as well. In 2001, Monte Carlo methods were effectively used for bridge [10]. The real success of MCTS started in 2006, when the tree search method and *upper confidence tree (UCT)* formula were introduced, yielding very good results in Computer Go [9]. Since 2006, the algorithm has been extended with many variations and is still being used for other (computer) games [3], including the GVGAI competition [17].

This section explains how MCTS approximates action values for states. A tree is built incrementally from the states and actions that are visited in a game.

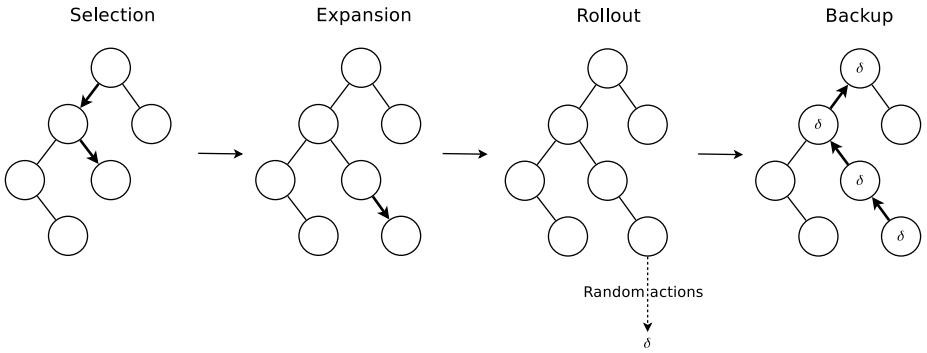



Figure 2.1: One Monte Carlo tree search iteration

Each node in the tree represents a state and each connection in the tree represents an action taken in that state leading to a new state, which is represented by the next tree node. The process, as explained in Figure 2.1, consists of four phases that are constantly repeated. It is started with the current game state, which is represented by the root node of the tree. The first action is chosen by an *expansion* strategy and subsequently simulated. This results in a new game state, for which a new node is created. After expansion, a *rollout* is done from the new node, which means that a simulation is run from the new node applying random actions until a predefined stop criterion is met or the game ends. Finally, the score difference resulting from the rollout is *backed up* to the root node, which means that the reward is saved to the visited nodes. Then a new iteration starts. When all actions are expanded in a node, that node is deemed *fully expanded*. This means that MCTS will use its *selection* strategy to select child nodes until a node is selected that is not fully expanded. Then, the expansion strategy is used to create a new node, after which a rollout takes place and the results are backed up.

The selection strategy selects optimal actions in internal tree nodes by analyzing the values of their child nodes. An effective and popular selection strategy is the UCT formula [13]. This balances the choice between poorly explored actions with a high uncertainty about their value and actions that have been explored extensively, but have a higher value. A child node $j$ is selected to maximize

$$UCT = v_{s'} C_p \sqrt{\frac{2 \ln n_s}{n_{s'}}} \tag{2.1}$$

Where $v_{s'}$ is the value of child $s'$ as calculated by the backup function, $n_s$ is the number of times the current node $s$ has been visited, $n_{s'}$ is the number of times child $s'$ has been visited and $C_p > 0$ is a constant that shifts priority from exploration to exploitation. By increasing $C_p$, the priority is shifted to exploration: states that have been visited less, will be visited with a higher priority than states that have been visited more. A decrease shifts priority to

exploitation: states that have a high value are visited more, in order to maximize reward.

The traditional expansion strategy is to explore each action at least once in each node. After all actions have been expanded, the node applies the selection strategy for further exploration. Some variants of MCTS reduce the branching factor of the tree by only expanding the nodes selected by a special expansion strategy. A specific example is the *crazy stone* algorithm [6], which is an expansion strategy that was designed specifically for Go. We will use an adaptation of this strategy in the algorithm proposed in Chapter 6. When using crazy stone, an action $i$ is selected with a probability proportional to $u_i$

$$u_i = \exp\left(K \frac{\mu_0 - \mu_i}{\sqrt{2\left(\sigma_0^2 + \sigma_i^2\right)}}\right) + \varepsilon_i \tag{2.2}$$

Each action has an estimated value $\mu_i$ ordered in such a way that $\mu_0 > \mu_1 > \ldots > \mu_N$, and a variance $\sigma_i^2$. K is a constant that influences the exploration – exploitation trade off. $\varepsilon_i$ prevents the probability of selecting a move to reach zero and its value is proportional to the ordering of the expected values of the possible actions.

$$\varepsilon_i = \frac{0.1 + 2^{-i} + a_i}{N} \tag{2.3}$$

Where $a_i$ is 1 when an action is *an atari move*, a go-specific move that can otherwise easily be underestimated by MCTS, and otherwise 0.

After a rollout, the reward is backed up, which means that the estimated value for every node that has been visited in this iteration is updated with the reward of this simulation. Usually the estimated value of a node is the average of all rewards backed up to that node.

## 2.3 Options

In order to mimic human game playing strategies, such as defining subgoals and subtasks, we use options. Options have been proposed by Sutton et al. as a method to incorporate temporal abstraction in the field of reinforcement learning [28]. The majority of the research seems to focus on learning algorithms and little work has been done on combining options with tree search methods, which offer a model free planning framework [2].

An option, sometimes referred to as a macro-action, is a predefined method of reaching a specific subgoal. Formally, it is a triple $\langle I, \pi, \beta \rangle$ in which $I \subseteq S$ is an initiation set, $\pi : S \times A \to [0, 1]$ is a policy and $\beta : S^+ \to [0, 1]$ is a termination condition. The initiation set $I$ is a set of states in which the option can be started. This set can be different for each option. The option's policy $\pi$ defines the actions that should be taken for each state. The termination condition $\beta$ is used to decide if an option is finished, given that the agent arrived at a certain state.

When an agent starts in state $s$, it can choose from all of the options $o \in O$ that have $s$ in its initiation set $I_o$. Then the option's policy $\pi$ is followed, possibly for several time steps. The agent stops following the policy as soon as it reaches a state that satisfies a termination condition in $\beta$. This often means that the option has reached its subgoal, or a criterion is met that renders the option obsolete (e.g., its goal does not exist anymore). Afterwards, the agent chooses a new option that will be followed.

Using options in an MDP removes the Markov property for that process: the state information alone is no longer enough to predict an agent's actions, since the actions are now not only state-dependant, but dependant on what option the agent has chosen in the past as well. According to [28], we can view the process as a *semi-Markov decision process (SMDP)* [8], in which options are actions of variable length. In this thesis, we will call the original action set of the MDP $A$, and the set of options $O$.

Normally, an agent can never choose a new option in a state that is not in the termination set of any of the options. Some algorithms use *interruption*, which means they are designed not to follow an option until its stop criterion is met, but choose a new option every time step [28, 21]. Using this method, an agent can choose a new option in any state that is present in the MDP, which can lead to better performance than without interruption.

Most options span over several actions. Their rewards are discounted over time. This means that rewards that lay further in the future are valued less than those that lay nearer. A reward $r_o$ for option $o$ is calculated for using an option from timestep $t$ to timestep $t + n$ with

$$r_o = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots + \gamma^n r_{t+n}, \tag{2.4}$$

where $\gamma$ is the discount factor, which indicates the importance of future states.

Normal actions can be treated as options as well. An option for action $a \in A$ has a initiation set $I = S$, the policy $\pi$ is taking action $a$ in all the states. The termination condition $\beta$ is that action $a$ should be performed once.

## 2.4 Q-learning

Q-learning is a relatively simple method for reinforcement learning, that was proposed in 1992 [30]. The *Q-value* of an action for a state, $Q(s, a)$, is the discounted reward that can be achieved by applying action $a$ to state $s$ and following the optimal policy afterwards. By learning Q-values for every action in every state, it can estimate an optimal policy.

The general idea of Q-learning is to incrementally save the combination of a reward and the current estimate of the Q-value of an action and game state. An agent starts in state $s$, takes action $a$ and arrives in state $s'$ with reward $r$. The update function uses the reward and the maximum of the Q-values of the next state $s'$. By always using the maximum value of the next state, the Q-value function will eventually converge the maximum discounted future reward for a

state-action pair. The update function for a state-action pair is denoted by

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a) \right), \tag{2.5}$$

where $r$ is the reward that is achieved by using action $a$ in state $s$, leading to state $s'$. The algorithm has two parameters: $\gamma$ is the discount factor and $\alpha$ is the learning rate, which determines the magnitude of the Q-value updates. Q-learning is shown to converge if $\alpha$ decreases over time. However, in practice it is often set to a small constant. The Q-table can be used to find the optimal policy by, for each state $s \in S$, selecting the action $a \in A$ that maximizes $Q(s,a)$. Because after each action only one state-action pair is being updated, Q-learning can take a long time to converge. It is, however, guaranteed to converge to the optimal policy, given that during exploration each state-action pair is visited an infinite number of times.

## 2.5 SMDP Q-learning

When options were introduced, they were used in combination with Q-learning. In order to be able to compare our contribution to the Q-learning approach, we implement *SMDP Q-learning* [28] for VGDL, based on the description made by Sutton et al. in section 3.2 of their paper. In general, SMDP Q-learning estimates the expected rewards for using an option in a certain state, in order to find an optimal policy over the option set $O$.

Like traditional Q-learning, SMDP Q-learning estimates a value function. The option-value function contains the expected return $Q(s,o)$ of using an option $o \in O$ in state $s \in S$. Updates of the $Q$ function are done after each option termination by an adaptation of the original Q-learning update function from Equation 2.5:

$$Q(s,o) \leftarrow Q(s,o) + \alpha \left( r + \gamma^k \max_{o' \in O_{s'}} Q(s',o') - Q(s,o) \right), \tag{2.6}$$

where $\gamma$ is the discount factor, $k$ denotes the number of time steps between the start state $s$ and stop state $s'$ for option $o$, $r$ denotes the cumulative and discounted option reward from Equation 2.4, and step size parameter $\alpha$ is similar to the learning rate parameter from traditional Q-learning. The difference with Equation 2.5 is that here we take the $k^{\text{th}}$ power of $\gamma$, which penalizes options that take more actions. Hereby, the desired effect is reached that an option that takes half the time, but has more than double the reward of another option gets preferred over the other.

Using the Q-table, an option policy can be constructed. A *greedy policy* selects the option that maximizes the expected return $Q(s,o)$ for any state $s \in S$ and option $o \in O$ with $s$ in its initiation set $I_o$. When a Q-table is fully converged, the greedy policy is the optimal policy[1] [28]. Exploration is often

[1] Note that it is optimal for the maximum achievable by using only the options in the option set $O$.

done by using an *ε-greedy policy*. This is a policy that chooses a random option with probability $\varepsilon$ and the greedy option otherwise. The parameter $\varepsilon$ is set to a value between 0 and 1. A higher value for $\varepsilon$ leads more exploration of unknown states, whereas a lower value shifts the algorithm's focus to exploiting the currently known feasible states.

## 2.6 Generalized Video Game Playing

In this thesis, algorithms will be benchmarked on the general video game playing problem. Recent developments in this area include VGDL [24], a framework in which a large number of games can be defined and accessed in a similar manner. VGDL aims to be a clear, human readable and unambiguous description language for games. Games are easy to parse and it is possible to automatically generate them. Using a VGDL framework, algorithms can access all the games in a similar manner, resulting in a method to compare their performances on several games.

To define a game in VGDL, two files are required. Firstly, the game description should be made, which defines for each type of object what character in the level description it corresponds to, what it looks like in a game visualization, how it interacts with the rest of the game and when it disappears from the game. Secondly, a level description file is needed, in which each character maps to an object in the game. The location in the file corresponds to its grid location in the game. By defining these two files, a wide spectrum of games can be created. A more extensive explanation and an example of a game in VGDL can be found in Section 7.1.

The General Video Game AI competition provides games written in VGDL. Furthermore, it provides a Java framework in which algorithms can be benchmarked using a static set of rules. Algorithms are only allowed to observe the game information: score, game tick (timestep), the set of possible actions and information about if the game is over and if the player won; avatar information: its position, orientation, speed and resources; and screen information: Which sprites are on the screen, including their location and the size (in pixels) of the level grid's blocks.

Algorithms have a limited amount of time to plan their actions, during which they can access a simulator called the *forward model*. The forward model acts as a black box that returns a new state when an algorithm applies an action. The actions that are used on the forward model do not influence the real game score. Before the simulation time runs out, the algorithm should return an action to apply to the real game.

The algorithms proposed in this thesis will be benchmarked on the GVGAI game sets, using the rules of the competition. This means that the algorithms do not have any access to the game and level descriptions. When an algorithms starts playing a game, it typically knows nothing of the game, except for the observations described above.

# Chapter 3

# Related Work

This chapter covers some popular alternative methods for general video game playing and prior work on tree search with options. *Deep Q networks (DQN)* is a popular algorithm that trains a convolutional neural network for a game [15], *Planning under Uncertainty with Macro-Actions (PUMA)* is a forward search algorithm that uses extended actions in *partially observable MDPs (POMDPs)* [12]. *Purofvio* is an algorithm that combines MCTS with macro-actions that consist of repeating one action several times [20].

DQN is a general video game playing algorithm that trains a convolutional neural network that has the last four pixel frames of a game as input and tries to predict the return of each action. A good policy can then be created by selecting the action with the highest return. This is one of the first algorithms that successfully combines neural network learning with reinforcement learning. In this case, however, it was not desirable to implement DQN because of the limitations proposed by our testing framework. The GVGAI competition framework currently works best for planning algorithms, that use the forward model to quickly find good policies. Learning over the course of several games is difficult. In contrast, DQN typically trains on one game for several days before a good policy is found and does not utilize the forward model, but always applies actions directly to the game in order to learn.

There are, however, examples of other learning algorithms that have been successfully implemented in the GVGAI framework. These algorithms can improve their scores after playing a game for several times, using a simple state representation [23]. Their features consist of:

- the game score,
- the game tick,
- the winner ($-1$ if game is still ongoing, 0 if the player lost and 1 if the player won),
- game over (0 if the game is ongoing, 1 if the game is over),

- a list of resources,
- a list of Euclidean distances to the nearest sprite of each type,
- the speed of the avatar.

The results of the paper show that the algorithms are capable of learning in the course of 1000 game plays of the first level of each game. It has to be noted that no results of how many times the algorithms *win* the game are reported and that it seems (looking at the score that is achieved) that many of the games are actually lost most of the times. The learning algorithms proposed in this thesis will focus more on early results than on long term learning.

An alternative tree search algorithm is PUMA, which applies forward search to options (referred to as macro-actions) and works on POMDPs, which means it does not restrict an MDP to be fully observable. PUMA automatically generates goal-oriented MDPs for specific subgoals. The advantage of this is that effective options can be generated without requiring any prior knowledge of the (PO)MDP. The disadvantage is that this takes a lot of computation time and thus would not work in the GVGAI framework, in which only a limited amount of computation time is allowed between actions. Options generated for one game, would not necessarily be transferable to other games, meaning that option generation would have to be done prior to every game that the algorithm plays. Furthermore, PUMA has to find out the optimal length per macro-action, whereas the algorithms proposed in this thesis can use options of variable length.

Another algorithm that uses MCTS with macro actions is called Purofvio. Purofvio plans over macro-actions which, in this case, are defined as repeating one action for a fixed number of times. No more complex options are defined. The algorithm is constructed for the physical traveling salesperson problem, which offers the same type of framework as the GVGAI competition: a simulator is available during limited action time, after which an action has to be returned. An important feature of this algorithm is that it can use the time budget of several actions to compute which macro-action to choose next. This is possible because a macro-action is not reconsidered after it has been chosen. The paper notes that their options must always be of the same length, because they found that otherwise MCTS seems to favor options with a longer time span over shorter options. It is suggested that Purofvio could work on other games as well, but this has not been shown.

# Chapter 4

# Options for General Video Game Playing

In the past, options have mainly been used in specific cases and for relatively simple problems. To specify the problem domain, this chapter will cover how a game is defined in VGDL and how it is observed by the game playing algorithm. Furthermore, we will explain what options mean in the context of general video game playing and what we have done to create a set of options that can be used in any game. Lastly, this chapter explains what is needed to use SMDP Q-learning on the domain of general video game playing.

## 4.1 Toy Problem

As described in the background section, the foundation of a game lies in two specifications: the game's dynamics and its levels. A game has several levels, each of which is different. Typically the last level of a game is harder than the first. The game dynamics are the same for each level of a game.

In VGDL the game description defines the game dynamics. Each game has one game description file, which describes the game sprites and their interaction with each other. The levels are defined in separate level files and define the layout of the screen.

In this section we introduce our test game *prey*. We created this game for testing the learning capacities of the algorithm. Furthermore, this section contains an in-depth explanation of the VGDL. The game aims to be simple to understand and easy to win. It should be possible to see any improvement an algorithm could achieve as well. The game is based on the *predator & prey* game, in which the player is a predator, that should catch its prey (an NPC) by walking into it. We decided to have three types of prey, one that never moves, one that moves once in 10 turns and one that moves once in 2 turns. This section describes how the game is made in VGDL and how a GVGAI agent can interact with it.

Listing 4.1: prey.txt

```
1  BasicGame
2    SpriteSet
3      movable >
4        avatar     > MovingAvatar img=avatar
5        prey       > img=monster
6          inactivePrey     > RandomNPC cooldown=3000
7          slowPrey         > RandomNPC cooldown=10
8          fastPrey         > RandomNPC cooldown=2
9
10   LevelMapping
11     A   > avatar
12     I   > inactivePrey
13     S   > slowPrey
14     F   > fastPrey
15
16   InteractionSet
17     prey avatar      > killSprite scoreChange=1
18     movable wall     > stepBack
19
20   TerminationSet
21     SpriteCounter stype=prey limit=0 win=True
22     Timeout limit=100 win=False
```

Listing 4.2: Prey level 1

```
1  wwwwwww
2  wA    w
3  w     w
4  w     w
5  w    Iw
6  wwwwwww
```

Listing 4.3: Prey level 2

```
1  wwwwwwwwwwwww
2  wA      w     w
3  w       w     w
4  w       w     w
5  w wwwwww      w
6  w             w
7  w             w
8  w             w
9  w             w
10 w             w
11 w             w
12 w         wwwww
13 w            Iw
14 wwwwwwwwwwwww
```

The code in Listing 4.1 contains the game description. Lines 2 to 8 describe the available sprites. There are two sprites in the game, which are both `movable`: the avatar (predator) and the monster (prey). The avatar is of type `MovingAvatar`, which means that the player has four possible actions (up, right, down, left). The prey has three instantiations, all of the type `RandomNPC`, which is an NPC that moves about in random directions: the `inactivePrey`, which only moves every 3000 steps (which is more than the timeout explained shortly, so it never moves); the `slowPrey` which moves once every 10 steps and the `fastPrey` which moves once every 2 steps. By default, the `MovingAvatar` can move once in each

time step.

Lines 10 to 14 describe the level mapping. These characters can be used in the level description files, to show where the sprites spawn.

In the interaction set, Line 17 means that if the prey walks into the avatar (or vice versa) this will kill the prey and the player will get a score increase of one point. Line 18 dictates that no `movable` sprite can walk through walls.

Lastly, in the termination set, line 21 shows that the player wins when there are no more sprites of the type `prey` and line 22 shows that the player loses after 100 time steps.

Listing 4.2 shows a simple level description. This level is surrounded with walls and contains one avatar and one inactive prey. The game can be used to test the functionality of an algorithm. The first level is very simple and is only lost when the agent is not able to find the prey within the time limit of 100 time steps, which is unlikely. A learning algorithm should, however be able to improve the number of time steps it needs to find the prey. The minimum number of time steps to win the game in the first level, taking the optimal route, is six. Listing 4.3 defines the second level, which is more complex. The agent has to plan a route around the walls and the prey is further away. We chose to still use the inactive prey in this case, because then we know that the minimum number of time steps needed to win is always twenty.

An agent that starts playing *prey* in the GVGAI framework, observes the world as defined by the level description. It knows where all the sprites are, but it does not know in advance how these sprites interact. The other relevant observations for this game are the avatar's direction and speed and the game tick. When an algorithm plays the game, it has a limited amount of time to choose an action, based on the observation. It can access the *forward model* that simulates the next state for applying an action. The forward model can be polled indefinitely, but an action has to be returned by the algorithm within the action time.

## 4.2 Option Set

Human game players consider subgoals when they are playing a game. Some of these subgoals are game-independent. For example, in each game where a player has a movable avatar the player can decide it wants to go somewhere or avoid something. These decisions manifest themselves as subgoals, which are formalized in this section in a set of options.

In this section, we describe our set of options which can be used in any game with a movable avatar and provides the means to achieve the subgoals mentioned above. Note that a more specific set of options can be created when the algorithm should be tailored to only one type of games and similarly: options can be added and removed from the set easily. The following options are designed and will be used in the experiments in Chapter 7:

- `ActionOption` executes a specific action once and then stops.

  - Invocation: the option is invoked with an action.
  - Subtypes: one subtype is created for each action in action set $A$.
  - Initiation set: any state $s_t$
  - Termination set: any state $s_{t+1}$
  - Policy: apply the action corresponding to the subtype.

- `AvoidNearestNpcOption` makes the agent avoid the nearest NPC
  - Invocation: this option has no invocation arguments
  - Subtypes: this option has no subtypes
  - Initiation set: any state $s_t$ that has an NPC on the observation grid
  - Termination set: any state $s_{t+1}$
  - Policy: apply the action that moves away from the NPC. This option ignores walls.

- `GoNearMovableOption` makes the agent walk towards a movable game sprite (defined as movable by the VGDL) and stops when it is within a certain range of the movable
  - Invocation: this option is invoked on a movable sprite in the observation grid.
  - Subtypes: the subtype corresponds to the type of the sprite this option follows.
  - Initiation set: any state $s_t$ with the goal sprite in the observation grid
  - Termination set: any state $s_{t+n}$ in which the path from the avatar to the goal sprite is smaller than 3 actions and all the states $s_{t+n}$ that do not contain the goal sprite.
  - Policy: apply the action that leads to the goal sprite.

- `GoToMovableOption` makes the agent walk towards a movable until its location is the same as that of the movable
  - Invocation: this option is invoked on a movable sprite in the observation grid.
  - Subtypes: the subtype corresponds to the type of the sprite this option follows.
  - Initiation set: any state $s_t$ with the goal sprite in the observation grid
  - Termination set: any state $s_{t+n}$ in which the goal sprite location is the same as the avatar location and all the states $s_{t+n}$ that do not contain the goal sprite.
  - Policy: apply the action that leads to the goal sprite.

- `GoToNearestSpriteOfType` makes the agent walk to the nearest sprite of a specified type

  - Invocation: the option is invoked with a sprite type.
  - Subtypes: the subtype corresponds to the invocation sprite type.
  - Initiation set: any state $s_t$ that has the sprite type corresponding to the subtype in its observation grid
  - Termination set: any state $s_{t+1}$ where the location of the avatar is the same as a sprite with a type corresponding to the subtype, or any state that has no sprites of this option's subtype.
  - Policy: apply the action that leads to the nearest sprite of this option's subtype.

- `GoToPositionOption` makes the agent walk to a specific position.
  - Invocation: the option is either invoked on a specific goal position in the grid or with a static sprite.
  - Subtypes: if the option was invoked on a specific sprite, the subtype is that sprite's type.
  - Initiation set: any state $s_t$. If the goal is a sprite, this sprite has to be in the observation grid.
  - Termination set: any state $s_{t+n}$ in which the avatar location is the same as the goal location.
  - Policy: apply the action that leads to the goal location.

- `WaitAndShootOption` waits until an NPC is in a specific location and then uses its weapon.
  - Invocation: the option is invoked with a distance from which the avatar will use his weapon.
  - Subtypes: a subtype is created for each invocation distance
  - Initiation set: any state $s_t$
  - Termination set: any state $s_{t+n}$ in which the agent has used his weapon
  - Policy: do nothing until an NPC moves into a given distance, then use the weapon (action USE)

For each option type, a subtype per visible sprite type is created during the game. For each sprite, an option instance of its corresponding subtype is created. For example, the game *zelda*, as seen in Figure 4.1, contains three different sprite types (excluding the avatar and walls); monsters, a key and a portal. The first level contains three monsters, one key and one portal. The aim of the game is to collect the key and walk towards the portal without being killed by the monsters. The score is increased by 1 if a monster is killed ,i.e., its sprite is on the same location as the sword sprite, if the key is picked up, or when the game is won. `GoToMovableOption` and `GoNearMovableOptions` are created for each of the three monsters and for the key. A `GoToPositionOption` is created

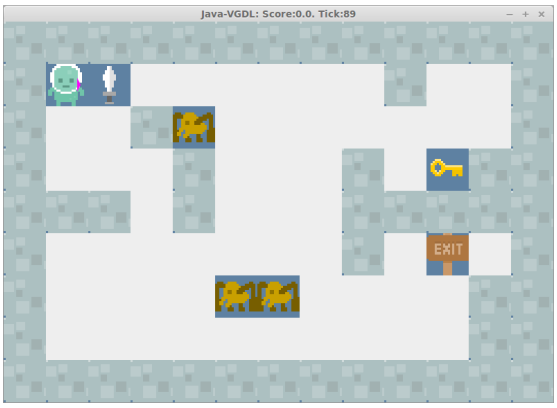



Figure 4.1: Visual representation of the game *zelda*.

for the portal. One `GoToNearestSpriteOfType` is created per sprite type. One `WaitAndShootOption` is created for the monsters and one `AvoidNearestNpc-Option` is created. This set of options is $O$, as defined in Section 2.3. In a state where, for example, all the monsters are dead, the possible option set $\mathbf{p}_s$ does not contain the `AvoidNearestNpcOption` and `GoToMovableOption`s and `GoNearMovableOption`s for the monsters.

The role of the `GoTo...` options is to enable the avatar to reach the key and the portal. The `GoNear...` options can be used to motivate the avatar to go near a monster, because if the avatar uses its sword (or the `WaitAndShootOptions` on a monster, the game can be won with a higher score. The `AvoidNearestNpcOption` functions to save the avatar from monsters that come too close. If the algorithms encounters a game in which these options can not lead to winning the game, it can use the `ActionOptions`, that function the same as normal actions.

The `GoTo...` and `GoNear...` options utilize an adaptation of the A Star algorithm to plan their routes [11]. An adaptation is needed, because at the beginning of the game there is no knowledge of which sprites are traversable by the avatar and which are not. Therefore, during every move that is simulated by the agent, the A Star module has to update its beliefs about the location of walls and other blocking objects. This is accomplished by comparing the movement the avatar wanted to make to the movement that was actually made in game. If the avatar did not move, it is assumed that all the sprites on the location the avatar should have arrived in are blocking sprites. A Star keeps a *wall score* for each sprite type. When a sprite blocks the avatar, its wall score is increased by one. Additionally, when a sprite kills the avatar, its wall score is increased by 100, in order to prevent the avatar from walking into killing sprites. Traditionally the A Star's heuristic uses the distance between two points. Our A Star adaptation adds the wall score of the goal location to this heuristic, encouraging the algorithm to take paths with a lower wall score. This method enables A Star to try to traverse paths that were unavailable earlier, while

preferring safe and easily traversable paths. For example in *zelda*, a door is closed until a key is picked up. Our A Star version will still be able to plan a path to the door once the key is picked up, winning the game.

## 4.3 SMDP Q-learning for General Video Game Playing

This option set is used by the SMDP Q-learning algorithm. The algorithm was adjusted to work as an anytime algorithm in the GVGAI competition framework. Our implementation uses interruption when applying actions to the game, but not during planning. This way, the algorithm can identify options that are good during planning, but will not follow them in dangerous situations that have not been encountered during planning. Because of the time limitation we added a maximum search depth $d$ to the algorithm that limits exploration to the direct surroundings of a state.

**Algorithm 1** SMDP Q − learning$(Q, O, s, t, d, o)$

```
1:  i ← ∅                          ▷ i_o counts how many steps option o has been used
2:  while time_taken < t do
3:      s' ← s                                              ▷ copy the initial state
4:      if s' ∈ β(o) then                                   ▷ if option stops in state s'
5:          o ← epsilon_greedy_option(s', Q, O)             ▷ get epsilon greedy option
6:          i_o ← 0                                         ▷ reset step counter i_o
7:      end if
8:      for depth in {0 ... d} do
9:          a ← get_action(o, s')                           ▷ get action from o
10:         (s', r) ← simulate_action(s', a)                ▷ set s' to new state, get reward r
11:         update_option(Q, o, s', r, i)                   ▷ Algorithm 2
12:         if s' ∈ β(o) then                               ▷ same as lines 4 to 7
13:             o ← epsilon_greedy_option(s', Q, O)
14:             i_o ← 0
15:         end if
16:     end for
17: end while
18: return get_action(greedy_option(s, Q, O), s)
```

**Algorithm 2** update_option$(Q, o, s, r, \mathbf{i})$

```
1: i_o ← i_o + 1                          ▷ increase this option's step count by 1
2: o_r ← o_r + γ^{i_o} r;                 ▷ change this option's discounted reward
3: if s ∈ β(o) then                       ▷ if option stops in state s
4:     update_Q(o, s, o_r, i_o)           ▷ Equation 2.6
5: end if
```

Our implementation of the algorithm is formalized in Algorithm 1. We initialize Q-learning with the table $Q$ (0 for all states in the first game), a predefined option set $O$, the current state $s$, a time limit $t$, the maximum search depth $d$ and the currently followed option $o$ (for the first iteration of a game, initialize $o$ to any finished option). The algorithm starts with a loop that keeps running as long as the maximum time $t$ is not surpassed, from line 2 to 17. In line 3, the initial state $s$ is copied to $s'$ for mutation in the inner loop. Then, in lines 4 to 7, the algorithm checks if a new option is needed. If the currently used option $o$ is finished, meaning that state $s'$ is in its termination set $\beta(o)$, a new option is chosen with an epsilon greedy policy.

Then, from lines 8 to 16 the option provides an action that is applied to the simulator in line 10 by using the function simulate_action. Afterwards, the function update_option, displayed in Algorithm 2, updates the option's return. If the option is finished it updates the Q-table as well, after which a new option is selected by the epsilon greedy policy in line 13 of Algorithm 1. The inner loop keeps restarting until a maximum depth is reached, after which the outer loop restarts from the initial state.

Algorithm 2 describes how our update_option function works: first the step counter $i_o$ is increased. Secondly, the cumulative discounted reward is altered. If the option is finished, $Q$ is updated using Equation 2.6 (note that in the equation the cumulative reward $o_r$ is $r$ and step counter $i_o$ is $k$). By maintaining the discounted option value, we do not have to save a reward history for the options.

By repeatedly applying an option's actions to the game, the Q-values of these options converge, indicating which option is more viable for a state. When the time runs out, the algorithm chooses the best option for the input state $s$ according to the Q-table and returns its action. After the action has been applied to the game, the algorithm is restarted with the new state. Section 7.2 describes the experiments we have done with this implementation of SMDP Q-learning.

SMDP Q-learning is a robust algorithm that is theoretically able to learn the best option for each of the states in a game. The problem, however, is that the algorithm has a slow learning process, which means that it will take several game plays for SMDP Q-learning to find a good approximation of the Q-table. Furthermore, Q-learning requires that the algorithm saves all the state-option pairs that were visited, which means that the Q-table can quickly grow to a size that is infeasible. Therefore, we propose a new algorithm that combines the advantages of using options with Monte Carlo tree search.

# Chapter 5

# O-MCTS

In order to simulate the use of subgoals we will introduce planning over options in MCTS, instead of SMDP Q-learning. In this chapter we introduce *option Monte Carlo tree search (O-MCTS)*, a novel algorithm that plans over options using MCTS, enabling the use of options in complex MDPs. The resulting algorithm achieves higher scores than MCTS and SMDP Q-learning on complex games that have several subgoals.

Generally, O-MCTS works as follows: like in MCTS, a tree of states is built by simulating game plays. Instead of actions the algorithm chooses options. When an option is chosen, the actions returned by its policy are used to build the tree. When an option is finished a new option has to be chosen, which enables the tree to branch on that point. Since traditional MCTS branches on each action, whereas O-MCTS only branches when an option is finished, deeper search trees can be built in the same amount of time. This chapter describes how the process works in more detail.

In normal MCTS, an action is represented by a connection from a state to the next, so when a node is at depth $n$, we know that $n$ actions have been chosen to arrive at that node and the time in that node is $t+n$. If nodes in O-MCTS would represent options and connections would represent option selection, this property would be lost, because options span over several actions. This complicates the comparison of different nodes at the same level in the tree. Therefore, we chose to keep the tree representation the same: a node represents a state, a connection represents an action. An option spans several actions and therefore several nodes in the search tree, as denoted in Figure 5.1. We introduce a change in the expansion and selection strategies, which select options rather than actions. When a node has an unfinished option, the next node will be created using an action selected by that option. When a node contains a finished option (the current state satisfies its termination condition $\beta$), a new option can be chosen by the expansion or selection strategy.

Methods exist for automatically generating options [5], but these have only been used on the room navigation problem and generating the options would take learning time which the GVGAI framework does not provide. Therefore, in

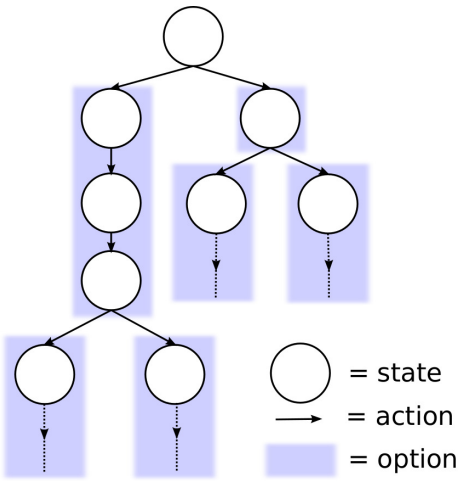



Figure 5.1: The search tree constructed by O-MCTS. In each blue box, one option is followed. The arrows represent actions chosen by the option. An arrow leading to a blue box is an action chosen by the option represented by that box.

this thesis O-MCTS uses the option set that was proposed in Section 4.2.

We describe O-MCTS in Algorithm 3. It is invoked with a set of options $O$, a root node $r$, a maximum runtime $t$ in milliseconds and a maximum search depth $d$. Two variables are instantiated. $C_s$ is a set of sets, containing the set of child nodes for each node. The set $\mathbf{o}$ contains which option is followed for each node. The main loop starts at line 3, which keeps the algorithm running until time runs out. The inner loop runs until a node $s$ is reached that meets a stop criterion defined by the function stop, or a node is expanded into a new node. In lines 6 until 10, $\mathbf{p}_s$ is set to all options that are available in $s$. If an option has not finished, $\mathbf{p}_s$ contains only the current option. Otherwise, it contains all the options $o$ that have state $s$ in their initiation set $I_o$. For example, the agent is playing *zelda* (see Figure 4.1) and the current state $s$ shows no NPCs on screen. If $o$ is the *AvoidNearestNpcOption*, $I_o$ will not contain state $s$, because there are no NPCs on screen, rendering $o$ useless in state $s$. $\mathbf{p}_s$ will thus not contain option $o$.

The four phases of Figure 2.1 are implemented as follows. In line 11, $\mathbf{m}$ is set to the set of options chosen in the children of state $s$. If $\mathbf{p}_s$ is the same set as $\mathbf{m}$, i.e., all possible options have been explored at least once in node $s$, a new node $s'$ is *selected* by uct. In line 14, $s$ is instantiated with the new node $s'$, continuing the inner loop using this node. Else, some options are apparently unexplored in node $s$. It is *expanded* with a random, currently unexplored option by lines 15 to 22. After expansion or when the stop criterion is met, the inner loop is stopped and a *rollout* is done, resulting in score difference $\delta$. This score difference is *backed up* to the parent nodes of $s$ using the backup function, after which the tree traversal restarts with the root node $r$.

**Algorithm 3** $\mathsf{O-MCTS}(O, r, t, d)$

```
1: C_{s∈S} ← ∅                                   ▷ c_s is the set of children nodes of s
2: o ← ∅                                         ▷ o_s will hold the option followed in s
3: while time_taken < t do
4:     s ← r                                     ▷ start from root node
5:     while ¬stop(s, d) do
6:         if s ∈ β(o_s) then                    ▷ if option stops in state s
7:             p_s ← ∪_o(s ∈ I_{o∈O})            ▷ p_s = available options
8:         else
9:             p_s ← {o_s}                       ▷ no new option can be selected
10:        end if
11:        m ← ∪_o(o_{s∈c_s})                    ▷ set m to expanded options
12:        if p_s = m then                       ▷ if all options are expanded
13:            s' ← max_{c∈c_s} uct(s, c)        ▷ select child node (Eq. 2.1)
14:            s ← s'                            ▷ continue loop with new node s'
15:        else
16:            ω ← random_element(p_s − m)
17:            a ← get_action(ω, s)
18:            s' ← expand(s, a)                 ▷ create child s' using a
19:            c_s ← c_s ∪ {s'}                  ▷ add s' to c_s
20:            o_{s'} ← ω
21:            break
22:        end if
23:    end while
24:    δ ← rollout(s')                           ▷ simulate until stop
25:    back_up(s', δ)                            ▷ save reward to parent nodes (Eq. 5.1)
26: end while
27: return get_action(max_{o∈c_r} value(o), r)
```

A number of functions is used by Algorithm 3. The function stop returns true when either the game ends in state $s$ or the maximum depth is reached in $s$. The function get_action lets option $\omega$ choose the best action for the state in node $s$, The function expand creates a new child node $s'$ for node $s$. $s'$ contains the state that is reached when action $a$ is applied to the state in node $s$. Typically, the rollout function chooses random actions until stop returns true, after which the difference in score achieved by the rollout is returned. In O-MCTS however, rollout always applies actions chosen by option $o$ first and applies random actions after $o$ is finished. The back_up function traverses the tree through all parents of $s$, updating their expected value. In contrast to traditional MCTS, which backs up the mean value of the reward to all parent nodes, a discounted value is backed up. The backup function for updating the value of ancestor node $s$ when a reward is reached in node $s'$ looks like this:

$$v_s \leftarrow v_s + \delta\gamma^{d_{s'}-d_s}, \tag{5.1}$$

where $\delta$ is the reward that is being backed up, $v_s$ is the value of node $s$. $d_s$ and $d_{s'}$ are the node depths of tree nodes $s$ and $s'$. Thus, a node that is a further ancestor of node $s'$ will be updated with a smaller value. This discounting method is similar to that of SMDP Q-learning done in Equation 2.6, where $\gamma^k$ is used to discount for the length of an option.

When the time limit is reached, the algorithm chooses an option from the children of the root node, $c_r$, corresponding to the child node with the highest expected value. Subsequently, the algorithm returns the action that is selected by this option for the state in the root node. This action is applied to the game.

In the next state, the algorithm restarts by creating a new root node from this state. Note that since O-MCTS always returns the action chosen by the best option at that moment, the algorithm uses interruption.

We expect that since this implementation of MCTS with options reduces the branching factor of the tree, the algorithm can do a deeper tree search. Furthermore, we expect that the algorithm will be able to identify and meet a game's subgoals by using options. In the experiments chapter we show results that support our expectations.

# Chapter 6

# OL-MCTS: Learning Option Values

Although we expect O-MCTS is an improvement over MCTS, we also expect the branching factor of O-MCTS's search tree to increase as the number of options increases. When many options are defined, exploring all the options becomes infeasible. In this chapter, we will define *option values*: the expected mean and variance of an option. These can be used to estimate which options need to be explored, and which do not. We adjust O-MCTS to learn the option values and focus more on the options that seem more feasible. We call the new algorithm *Option Learning MCTS (OL-MCTS)*. We expect that OL-MCTS can create deeper search trees than O-MCTS in the same amount of time, which results in more accurate node values and an increased performance. Furthermore, we expect that this effect is the greatest in games where the set of possible options is large, or where only a small subset of the option set is needed in order to win.

In general, OL-MCTS saves the return of each option after it is finished, which is then used to calculate global option values. During the expansion phase of OL-MCTS, options that have a higher mean or variance in return are prioritized. Contrary to O-MCTS not all options are expanded, but only those with a high variance or mean return. The information learned in a game can be transferred if the same game is played again by supplying OL-MCTS with the option values of the previous game.

The algorithm learns the option values, $\mu$ and $\sigma$. The expected mean return of an option $o$ is denoted by $\mu_o$. This number represents the returns that were achieved in the past by an option for a game. It is state-independent. Similarly, the variance of all the returns of an option $o$ is saved to $\sigma_o$.

For the purpose of generalizing, we divide the set of options into *types* and *subtypes*. The option for going to a movable sprite has type `GoToMovableOption`. An instance of this option exists for each movable sprite in the game. A subtype is made for each sprite type (i.e., each different looking sprite). The option values are saved and calculated per subtype. Each time an option $o$ is finished,

its subtype's values $\mu_o$ and $\sigma_o$ are updated by respectively taking the mean and variance of all the returns of this subtype. The algorithm can generalize over sprites of the same type by saving values per subtype.

Using the option values, we can incorporate the progressive widening algorithm, crazy stone, from Equation 2.2 to shift the focus of exploration to promising regions of the tree. The crazy stone algorithm is applied in the expansion phase of OL-MCTS. As a result, not all children of a node will be expanded, but only the ones selected based on crazy stone. When using crazy stone, we can select the same option several times, this enables deeper exploration of promising subtrees, even during the expansion phase. After a predefined number of visits $v$ to a node, the selection strategy uct is followed in that node to tweak the option selection. When it starts using uct, no new expansions will be done in this node.

The new algorithm can be seen in Algorithm 4 and has two major modifications. The updates of the option values are done in line 7. The function update_values takes the return of the option $o$ and updates its mean $\mu_o$ and variance $\sigma_o$ by calculating the new mean and variance of all returns of that option subtype. The second modification starts on line 13, where the algorithm applies crazy stone if the current node has been visited less than $v$ times, or alternatively applies UCT similarly to O-MCTS. The crazy_stone function returns a set of weights over the set of possible options $\mathbf{p}_s$. A weighted random then chooses a new option $\omega$ by using these weights. If $\omega$ has not been explored yet, i.e., there is no child node of $s$ in $c_s$ that uses this option, the algorithm chooses and applies an action and breaks to rollout in lines 17 to 27. This is similar to the expansion steps in O-MCTS. If $\omega$ has been explored in this node before the corresponding child node $s'$ is selected from $c_s$ and the loop continues like when uct selects a child.

We expect that by learning option values and applying crazy stone, the algorithm can create deeper search trees than O-MCTS. These trees are focused more on promising areas of the search space, resulting in improved performance. Furthermore, we expect that by transferring option values to the next game, the algorithm can improve after replaying games.

**Algorithm 4** $\mathsf{OL-MCTS}(O, r, t, d, v, \mu, \sigma)$

$C_{s \in S} \leftarrow \emptyset$
$\mathbf{o} \leftarrow \emptyset$
**while** $time\_taken < t$ **do**
    $s \leftarrow r$
    **while** $\neg\mathsf{stop}(s, d)$ **do**
        **if** $s \in \beta(o_s)$ **then**
            $\mathsf{update\_values}(s, o_s, \mu, \sigma)$ ▷ update $\mu$ and $\sigma$
            $\mathbf{p}_s \leftarrow \cup_o(s \in I_{o \in O})$
        **else**
            $\mathbf{p}_s \leftarrow \{o_s\}$
        **end if**
        $\mathbf{m} \leftarrow \cup_o(o_{s \in \mathbf{c}_s})$
        **if** $n_s < v$ **then** ▷ if state is visited less than $v$ times
            $\mathbf{u}_s \leftarrow \mathsf{crazy\_stone}(\mu, \sigma, \mathbf{p}_s)$ ▷ apply crazy stone, Eq. 2.2
            $\omega \leftarrow \mathsf{weighted\_random}(\mathbf{u}_s, \mathbf{p}_s)$
            **if** $\omega \notin \mathbf{m}$ **then** ▷ option $\omega$ not expanded
                $a \leftarrow \mathsf{get\_action}(\omega, s)$
                $s' \leftarrow \mathsf{expand}(s, a)$
                $\mathbf{c}_s \leftarrow \mathbf{c}_s \cup \{s'\}$
                $o_{s'} \leftarrow \omega$
                **break**
            **else** ▷ option $\omega$ already expanded
                $s' \leftarrow s \in \mathbf{c}_s : o_s = \omega$ ▷ select child node that uses $\omega$
            **end if**
        **else** ▷ apply $\mathsf{uct}$
            $s' \leftarrow \mathsf{uct}(s)$
        **end if**
        $s \leftarrow s'$
    **end while**
    $\delta \leftarrow \mathsf{rollout}(s')$
    $\mathsf{back\_up}(s', \delta)$
**end while**

# Chapter 7

# Experiments

This chapter describes the experiments that have been done on SMDP Q-learning, O-MCTS and OL-MCTS. The algorithms are compared to the Monte Carlo tree search algorithm, as described in Section 2.2. The learning capacities of SMDP Q-learning and OL-MCTS are demonstrated by experimenting with the toy problem, *prey*. Furthermore, all algorithms are run on a set of twenty-eight different games in the VGDL framework. The set consists of all the games from the first four training sets of the GVGAI competition, excluding puzzle games that can be solved by an exhaustive search and have no random component (e.g. NPCs). The games have 5 levels each, the first of which traditionally is the easiest and the last of which is the hardest.

Firstly, this chapter demonstrates the performance of our implementation of SMDP Q-learning. Secondly, we compare O-MCTS to MCTS, by showing the win ratio and mean score of both algorithms on all the games. Then, we show the improvement that OL-MCTS makes compared to O-MCTS if it is allowed 4 games of learning time. We demonstrate the progress it achieves by showing the first and last of the games it plays. Lastly we compare the three algorithms by summing over all the victories of all the levels of each game. Following the GVGAI competition's scoring method, algorithms are primarily judged on their ability to win the games. The scores they achieve are treated as a secondary objective.

## 7.1 Game Test Set

The algorithms we propose are tested on a subset of the first four training sets of the GVGAI competition. We exclude puzzle games that can be solved by an exhaustive search, because the algorithms will not be able to benefit from the option set that was constructed for these experiments, since the games have no random components. This leaves us with the twenty-eight games that are described in this section. All games the game have a time limit of 2000 time steps, after which the game is lost unless specified otherwise.

The games are described in decreasing order of the performance of a random algorithm, an algorithm that always chooses a random action, as an indication of the complexity of the games. The graphs in the experiments have the same ordering.

1. Surround: The aim of the game is to walk as far as possible. You get a point for every move you are able to make and afterwards, a non-traversable sprite is added in your previous location. The game is ended, and won, by using the `use` action. An NPC is doing the same as you and kills you on contact.

2. Infection: The aim of the game is to infect as many healthy (green) animals (NPCs) as possible, by colliding with a bug or infected animal first, and then with a healthy animal. When animals collide with other infected animals, they get infected as well. Blue sprites are medics that cure infected animals and the avatar, but can be killed with the avatar's sword. The player wins when every animal is infected.

3. Butterflies: The avatar hunts butterflies, the avatar wins if he has caught (walked into) all of them. Butterflies spawn at certain locations, so sometimes waiting longer to end the game can increase the eventual score.

4. Missile Command: Missiles move towards some city-sprites that need to be defended by the avatar. The avatar can destroy missiles before they reach the city by standing next to a missile and using his weapon. When all missiles are gone and some of the city sprites are still left, the game is won. If all the city sprites are gone, the game is lost.

5. Whackamole: The avatar must collect moles that appear at holes. The enemy player is doing the same. The player wins after 500 time steps, but loses if it collides with the enemy.

6. Aliens: Based on the commonly known game *space invaders*. Aliens appear at the top of the screen. The avatar can move left or right and shoot missiles at them. The avatar loses when the aliens reach the bottom and wins when all the aliens are dead. The avatar should evade the aliens' missiles as well.

7. Plaque attack: Hamburgers and hot dogs attack teeth that are spread over the screen. The avatar must shoot them with a projectile weapon in order to save at least one tooth. The avatar can repair damaged teeth by walking into them. The game is won if all the food is gone and lost if the teeth are gone.

8. Plants: Emulating the *Plants vs. Zombies* game, the avatar should plant plants that shoot peas at incoming zombies. The zombies can kill the plants by shooting back. The avatar wins when the time runs out, but loses when a zombie reaches the avatar's defensive field.

9. Bait: The avatar should collect a key and walk towards a goal. Holes in the ground kill the avatar when he moves into them, but can be closed by pushing boxes into them (after which both the hole and the box disappear). By collecting mushrooms, the player can get more points.

10. Camel Race: The avatar must get to the finish line before any other camel does.

11. Survive Zombies: The avatar should evade the zombies that walk towards him. When a zombie touches a bee, it drops honey. The avatar can pick up honey in order to survive one zombie attack. When a zombie touches honey, it dies. If the avatar survives for 1000 time steps, it wins the game.

12. Seaquest: The avatar can be killed by animals, or kill them by shooting them. The aim is to rescue divers by taking them to the surface. The avatar can run out of oxygen, so it must return to the surface every now and then. The game is won after 1000 time steps, or lost if the avatar dies.

13. Jaws: The avatar must shoot dangerous fish that appear from portals to collect the resources they drop. A shark appears at a random point in time, that can not be killed by shooting, but can be killed by touch, given that the avatar has enough resources. If he has too little resources, the game is lost. Otherwise, after 1000 time steps, the game is won.

14. Firestorms: The avatar must avoid flames while traversing towards the exit. He can collect water in order to survive hits by flames, but the game is lost if a flame hits the avatar when he has no water.

15. Lemmings: Lemmings are spawned from a door and try to get to the exit. The avatar must destroy their obstacles. There are traps that have to be evaded by the avatar and the lemmings. The game is won when all the lemmings are gone, or lost when the avatar dies.

16. Firecaster: The avatar must burn wooden boxes that obstruct his path to the exit. The avatar needs to collect ammunition in order to be able to shoot. Flames spread, being able to destroy more than one box, but the avatar should evade them as well. When the player's health reaches 0 he loses, but when he reaches the exit he wins.

17. Pacman: The avatar must clear the maze by eating all the dots, fruit pieces and power pills. When the player collides with a ghost he is killed, unless he has eaten a power pill recently.

18. Overload: The avatar must collect a determined number of coins before he is allowed to enter the exit. If the avatar collects more coins than that, he is trapped, but can kill marsh sprites with his sword for points.

19. Boulderdash: The player collects diamonds while digging through a cave. He should avoid boulders that fall from above and avoid or kill monsters. If the avatar has enough diamonds, he may enter the exit and wins.

20. Zelda: The avatar should collect a key and walk towards the door in order to win. He can kill monsters with his sword for additional points, but the monsters kill the avatar on touch.

21. Chase: The avatar chases and kills goats that are scared. However, when a scared ghost walks into the carcass of one of the others, he becomes angry and capable of killing the avatar. When there are no scared goats left, the avatar wins.

22. Digdug: The aim of the game is to collect all the gems and gold coins by digging a way through a cave. Enemies kill the player on collision. When the avatar presses USE in two consecutive time steps, it shoots a boulder to kill the enemy.

23. Bolo Adventures: The avatar should reach the goal. He can push around boxes 1 cell at a time, or boulders that roll until they hit another obstacle. Boulders can fill holes that obstruct the player's movement, and both boulders and boxes can block laser beams that kill the avatar.

24. Roguelike: The avatar should escape a maze through the door. There are monsters that can be killed with a sword, after it has been picked up. Doors can be opened with keys. Extra points are awarded for picking up gold and gems. After being hit by an enemy, it is possible to exchange gold for health.

25. Boulderchase: Similar to *boulderdash*, but enemies drop diamonds when they are killed and they can dig paths through the cave, like the player.

26. Eggomania: A chicken at the top of the screen throws eggs. The aim of the game is to move from left to right in order to catch the eggs. If the avatar fails to catch even one, the game is lost. When the avatar has collected enough eggs, it is possible to shoot the chicken to win the game.

27. Portals: The avatar needs to find the exit of the maze. He can use the available portals to teleport from one place to another. The aim of the game is to find the correct sequence of portals that lead to the exit. There are moving objects that kill the player on touch.

28. Frogs: Commonly known as *frogger*, the avatar is a frog, that needs to cross a busy road and a river. On the road, the frog should evade the trucks that move by. On the river, the frog should leap from tree trunk to tree trunk in order to reach the exit.

We apply Monte Carlo tree search to this set of games as our baseline algorithm. Furthermore, we run SMDP Q-learning, O-MCTS and OL-MCTS. The GVGAI competition comes with some predefined competition parameters: every algorithm has a maximum of one second to initialize, before starting the game. Then, the algorithm has a maximum of forty milliseconds per time step to return an action, after which a new game state is given. If the agent

exceeds the time limit, the action NULL will be applied and the avatar will not do anything[1]. To enable inter-game learning, the framework is edited to be able to pass information from one game to the next. After a game is finished, the learning algorithms OL-MCTS and SMDP Q-learning get the possibility to save their values to a file. The file can be read during the initialization of the next game. Since the initialization has a time limit, the algorithms are encouraged to minimize their file size.

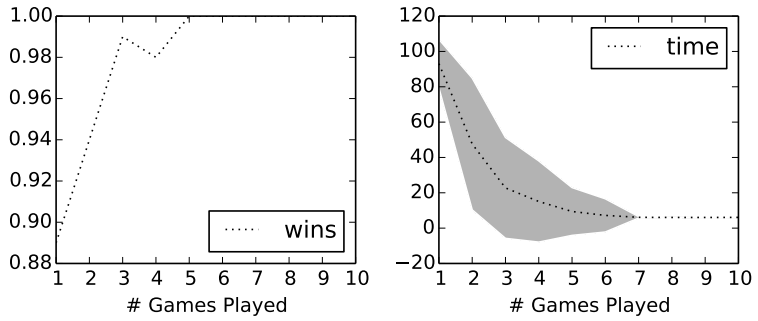
We empirically optimize the parameters of the O(L)-MCTS algorithm for these experiments. We use discount factor $\gamma = 0.9$. The maximum search depth $d$ is set to 70, which is higher than most alternative tree search algorithms, for example in the GVGAI competition, use. The number of node visits after which `uct` is used, $v$, is set to 40. Crazy stone parameter $K$ is set to 2.5. The maximum search depth for MCTS is set to 10, which is value the GVGAI implementation uses by default. MCTS and O-MCTS have `uct` constant $C_p = \sqrt{2}$. SMDP Q-learning has a maximum exploration depth $d$ of 20, an exploration parameter $\varepsilon$ of 0.2 and $\alpha$ is set to 0.1. All the experiments are run on an Intel Core i7-2600, 3.40GHz quad core processor with 6 GB of DDR3, 1333 MHz RAM memory. In all the following experiments on this game set, each algorithm plays each of the 5 levels of every game 20 times.

## 7.2 SMDP Q-learning

In this section, the SMDP Q-learning implementation that was proposed in chapter 4.3 is tested. Q-learning's first implementation requirement is a serialization of the state space. Furthermore, to prevent the Q-table to become too big we apply a state space reduction similar to that of other GVGAI learning algorithms [23]. The following values are saved:

- the avatar orientation;
- the avatar's resources (one or more quantities, e.g., `"diamonds:3"`);
- the direction and Euclidean distance to the nearest:
  - NPC,
  - other movable sprite,
  - resource,
  - portal,
  - other immovable sprite.

[1]There seems to be a bug in the system, that causes actions to sometimes take hundreds of milliseconds to simulate. This bug has been reported to the competition's discussion platform by several users. Because this bug was not solved at the time of writing, we chose to not honor the GVGAI competition's original rule to disqualify a controller that does not return an action within fifty milliseconds.




(a) Win ratio on *prey*

(b) Time in game ticks for *prey*. The grey area shows standard deviation.

Figure 7.1: Mean performance over 100 times 10 games of Q-learning the *prey* toy problem.

Although with this representation information about the map and the grid location of the avatar is lost, it enables the algorithm to keep a smaller and more generalized Q-table. As a result, it can work on more games without running out of memory.

The first experiment is a proof of concept of our implementation of SMDP Q-learning with our option set and state space reduction. We apply the algorithm to the first level of the toy problem, *prey*. The results can be seen in figure 7.1. We can see that the algorithm wins consistently starting from the fifth game The time it takes to win converges to 6, which is the least number of steps possible to reach the prey. We can conclude that it wins consistently by applying the correct options. This indicates that the values for the Q-table converge to their optima. Unfortunately, when applied to the second level of *prey*, which requires a more complex and longer action sequence, SMDP Q-learning is not able to find a winning policy in the first 500 games, after which its Q-table becomes too large to load in the initialization time and the algorithm is disqualified.

For the next experiment, SMDP Q-learning and Monte Carlo tree search are applied to all the games described in Chapter 7.1. The results are shown in Figures 7.2 and 7.3. The scores in Figure 7.3 are normalized using the data of all the algorithms, including O-MCTS and OL-MCTS and the random algorithm. The best score achieved by any of the algorithms in a level of a game is 1, the worst is normalized to 0. Like the game descriptions in Section 7.1, the bars are ordered by the performance of the random algorithm. From left to right its win ratio and score decreases, from which we can assume that the leftmost game is the easiest and the rightmost is the hardest.

The bars labeled Q-LEARNING1 show the performance of SMDP Q-learning in the first game it plays. It is then allowed three subsequent games to learn. The score of the fourth game is shown by the bars labeled Q-LEARNING4. Our further experiments have shown that a longer learning sequence decreases the total score for Q-learning, because the size of the Q-table causes the algorithm

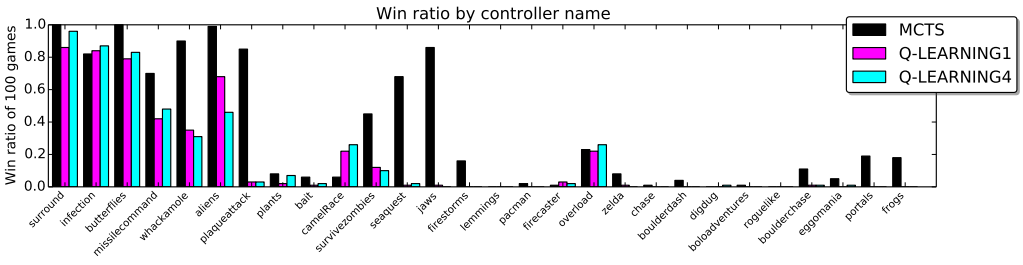



Figure 7.2: Win ratio of SMDP Q-learning per game on all levels, compared to Monte Carlo Tree Search.

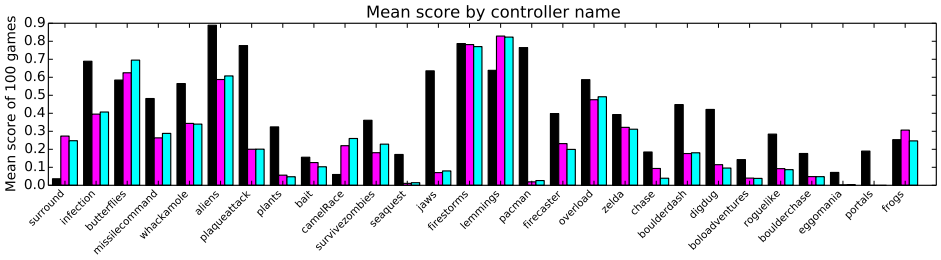



Figure 7.3: Mean normalized score of the algorithms per game 1 means the highest score achieved by all the algorithms (including all of the following tests), 0 the lowest.

to be disqualified during initialization.

The figures show that MCTS outperforms SMDP Q-learning in almost all the games. Our theory is that if the algorithm is allowed to learn for more games and is given more time in the initialization phase to load the Q-table, its performance will increase. However, the framework does not allow this, and the size of the Q-table might still become infeasible, since the number of states in these games is many times larger than the type of problem that SMDP Q-learning with options was demonstrated with [28]. For this reason, in the next section the results of SMDP Q-learning will be omitted and we will compare O-MCTS to MCTS.

## 7.3 O-MCTS

This section describes the results of the option Monte Carlo tree search algorithm in comparison with the standard Monte Carlo tree search algorithm. This demonstrates the improvement that can be achieved by using our method to enhance MCTS with our option set.

Figures 7.4 and 7.5 respectively show the win ratio and normalized score of the algorithms for each game. In short, the O-MCTS algorithms performs at least as good as MCTS in almost all games, and better in most.

O-MCTS outperforms MCTS in the games *missile command*, *bait*, *camel race*, *survive zombies*, *firestorms*, *lemmings*, *firecaster*, *overload*, *zelda*, *chase*,

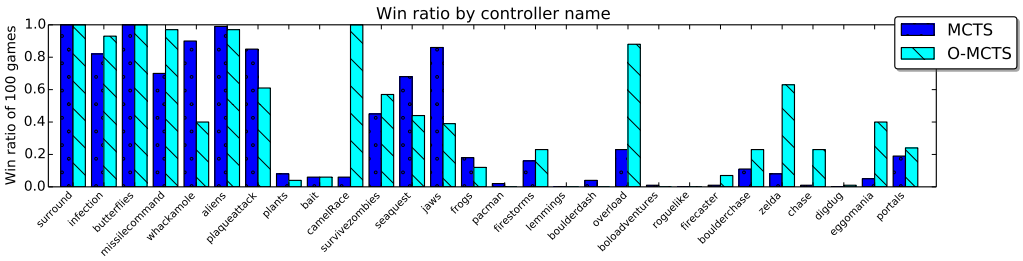



Figure 7.4: Win ratio of the algorithms per game on all levels.

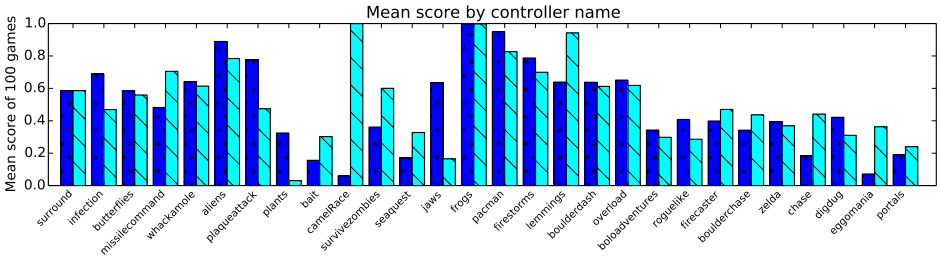
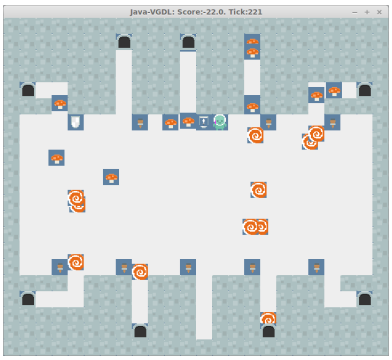


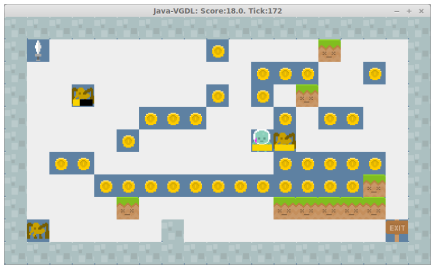

Figure 7.5: Mean normalized score of the algorithms per game 1 means the highest score achieved by all the algoriths, 0 the lowest.

*boulderchase* and *eggomania* winning more games or achieving a higher mean score. By looking at the algorithm's actions for these games, we can see that O-MCTS succeeds in efficiently planning paths in a dangerous environment, enabling it to do a further forward search than the ordinary Monte Carlo tree search. *Camel race* requires the player to move to the right for 80 consecutive turns to reach the finish line. No intermediate rewards are given to indicate that the agent is walking in the right direction. This is hard for MCTS, since it only looks 10 turns ahead. O-MCTS always wins this game, since it can plan forward a lot further. Furthermore, the rollouts of the option for walking towards the finish line have a bigger chance of reaching the finish line than the random rollouts executed by MCTS. In *Overload*, a sword has to be picked up before the avatar can finish the game, which seems to be too hard for MCTS, but poses less of a problem for O-MCTS. Furthermore, in *zelda* we can see that the MCTS algorithm achieves roughly the same score as O-MCTS, but does not win the game, since picking up the key and walking towards the door is a difficult action sequence. We assume that the mean score achieved by MCTS is because it succeeds in killing the monsters, whereas O-MCTS achieves its score by picking up the key and walking to the door. These results indicate that O-MCTS performs better than MCTS in games where a sequence subgoals have to be reached.

The MCTS algorithm performs better than O-MCTS in *pacman*, *whackamole*, *jaws*, *seaquest* and *plaque attack* (note that for *seaquest*, O-MCTS has a higher mean score, but wins less than MCTS). A parallel between these games is that they have a very big number of different sprites, for each of which several options



(a) *Plaque attack*, level 4



(b) *Overload*, level 4

Figure 7.6: The different in size of *plaque attack* ($24 \times 21$) and *overload* ($19 \times 11$)

have to be created by O-MCTS. When the number of options becomes too big, constructing the set of possible options $\mathbf{p}_s$ for every state $s$ becomes so time-consuming that the algorithm has too little time to build a tree and find the best possible action. To test this hypothesis we increased the computation time for O-MCTS to 120ms and found that the win ratio of O-MCTS increases to around 0.8 for *seaquest* and *plaque attack*, whereas the win ratio for MCTS increased to 0.9 and 0.7 respectively. This means that with more action time, the difference between O-MCTS and MCTS is reduced for *seaquest* and O-MCTS outperforms MCTS on *plaque attack*.

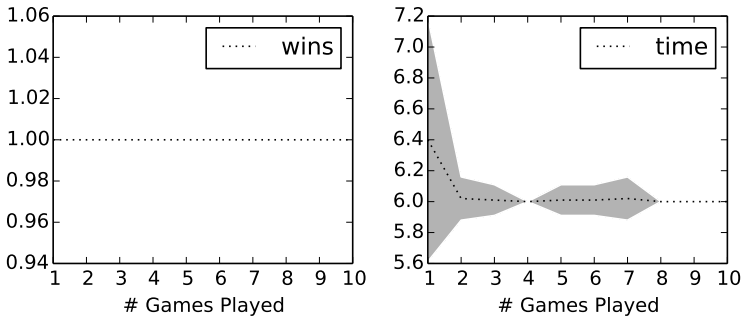
Another difficulty for O-MCTS in these games is illustrated in figure 7.6: the games *seaquest*, *plaque attack* and *pacman* have very large maps, in which using A Star to calculate routes becomes increasingly time consuming. For example, *plaque attack*, seen in Figure 7.6(a), has a grid of 24 by 21 blocks, which is more than twice the size of *overload*, seen in Figure 7.6(b) with a grid of 19 by 11 blocks.

Lastly, the game *jaws* starts with an almost empty screen. The algorithm is then inclined to walk towards the portals that are on the screen, because there are more options that lead towards portals than options that do not. These portals spawn enemies, which upon spawning immediately kill the avatar.

In summation, O-MCTS performs good on most games. When it does not, it mostly struggles with the overhead that is created by maintaining the option set: planning routes with A Star, checking if options are finished and constructing the possible option set, $\mathbf{p}_s$. By learning option values, OL-MCTS should be able to improve performance on these games.

## 7.4 OL-MCTS

This section describes the improvements that can be found on the O-MCTS algorithm, when a bias can be learned towards certain more feasible options. The




(a) Win ratio on *prey* (b) Time in game ticks for *prey*

Figure 7.7: Mean performance over 100 times 10 games of OL-MCTS on the *prey* toy problem.

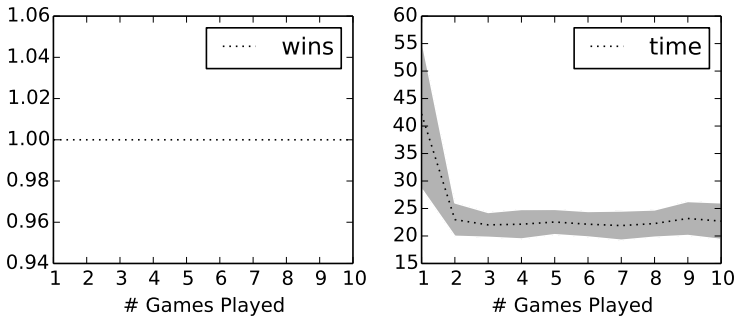
improvement that was proposed in Chapter 6 is compared to the performance of O-MCTS in this section.

Firstly, by running Option Learning MCTS on the toy problem *prey*, we demonstrate its capacity to learn, similar to what was done in section 7.2. The results are displayed in Figure 7.7. OL-MCTS never loses the toy problem and after the first game almost always takes 6 time steps to catch the prey.

As more challenging test, the algorithm is run on the second level and we set the maximum rollout distance $d$ to 10, so the prey is outside the reach of the rollouts. Figure 7.8 demonstrates that OL-MCTS successfully learns to prefer the options that lead to the avatar over the other options. This results in the algorithm taking a path that is close to the smallest number of time steps needed starting from game 2, even though in the first ten time steps, the algorithm has no rollouts that lead to the winning condition. From this we can conclude that the algorithm is capable to learn from which options it benefits and bias its search towards those. Furthermore, we observe that the performance of OL-MCTS is better than that of SMDP Q-learning on the same problem.

We compare OL-MCTS to O-MCTS, by running it on the game test set. The option learning algorithm is allowed four learning games, after which the fifth is used for the comparisons. Figures 7.9 and 7.10 show the results of the algorithm on these games. OL-MCTS1 denotes the performance of OL-MCTS on the first game, OL-MCTS5 shows the performance of the algorithm after learning.

We can see that, although the first iteration of OL-MCTS sometimes performs a bit worse than O-MCTS, the fifth iteration often scores at least as high, or higher than O-MCTS. We expect that the loss of performance in OL-MCTS1 is a result of the extra overhead that is added by the crazy stone algorithm: A sorting of all the options has to take place in each tree node. The learning algorithm significantly improves score and win ratio for the game *bait*, which is a game in which the objective is to reach a goal portal after collecting a key. The player can push boxes around to open paths. There are holes in the ground




(a) Win ratio on *prey* level 2 (b) Time for *prey* level 2

Figure 7.8: Mean performance over 100 times 10 games of OL-MCTS on the second level of the *prey* toy problem.

that kill the player unless they are filled with boxes, which make both the hole and the box disappear. Figure 7.11 shows the improvement in score and win ratio for this game. There are two likely explanations for this improvement: 1.) There are sprites that kill the player, which are evaded by the algorithm when it has learned to do so. 2.) The algorithm learns that it should pick up the key.

Furthermore, we can see small improvements on the games *seaquest* and *jaws*, on which O-MCTS performs worse than MCTS. Although OL-MCTS does not exceed the score of the original Monte Carlo tree search algorithm, this improvement suggests that OL-MCTS is on the right path of improving O-MCTS.

## 7.5 Totals

Figure 7.12 shows the sum of wins over all games, all levels. It shows a significant ($p < 0.05$) improvement of O-MCTS and OL-MCTS over MCTS. There is no significant difference in performance of OL-MCTS over O-MCTS, although our results suggest that it does improve for a subset of the games. SMDP Q-learning was not proven to be useful for general video game playing, and scores marginally better than the random algorithm. This might be a result of the rules of the GVGAI competition, which indirectly prevent the algorithm to learn for more than four games, which is a relatively low amount for a learning algorithm.

Summarizing, our tests indicate that on complex games O-MCTS outperforms MCTS. For other games it performs at least as well, as long as the number of game sprites is not too high. The OL-MCTS algorithm can increase performance for some of the games, such as *bait* and *plaque attack*. On other games, little to no increased perfromance can be found.

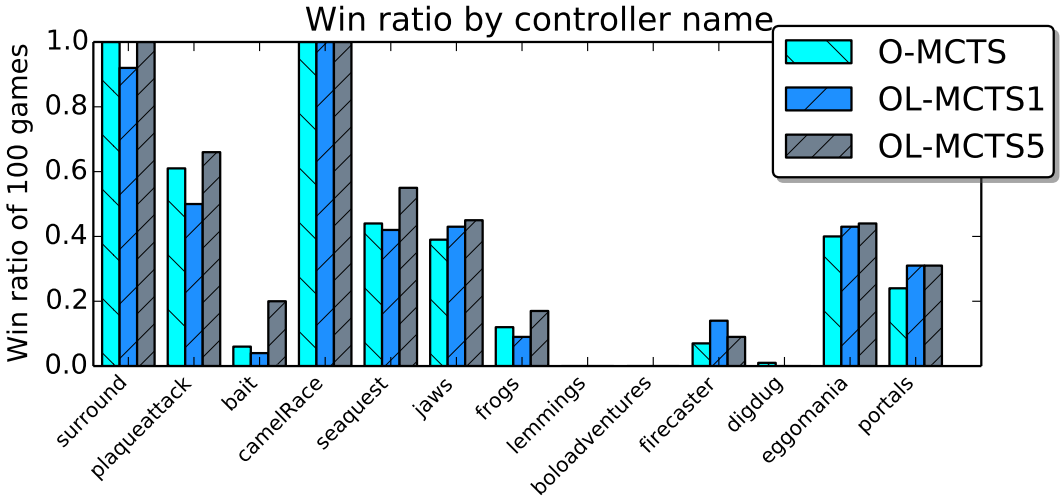



Figure 7.9: Win ratio of OL-MCTS compared to O-MCTS in its first and fifth game.

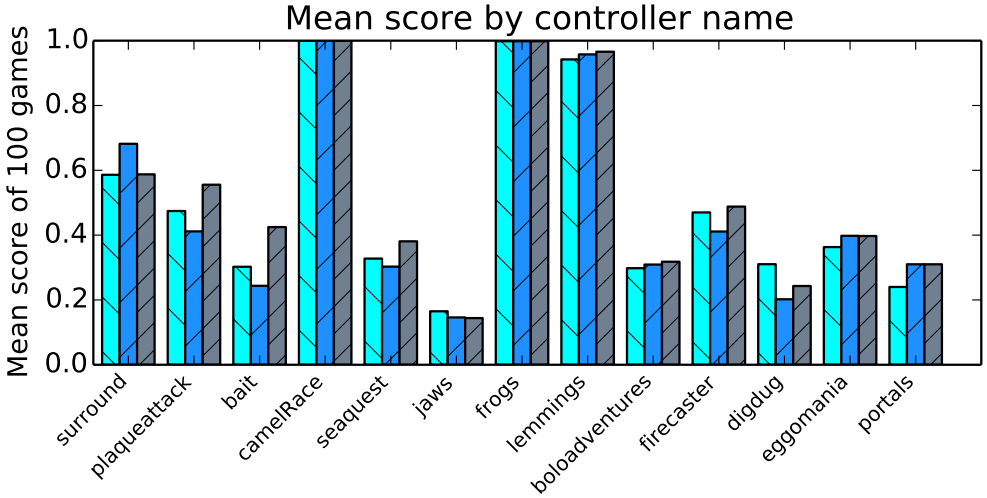



Figure 7.10: Mean normalized score comparison of OL-MCTS and O-MCTS.

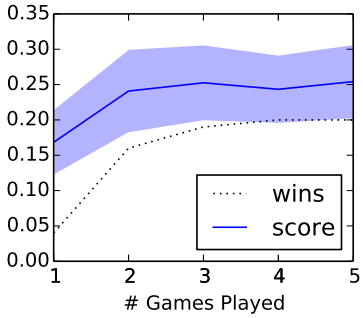



Figure 7.11: Learning improvement on *bait*

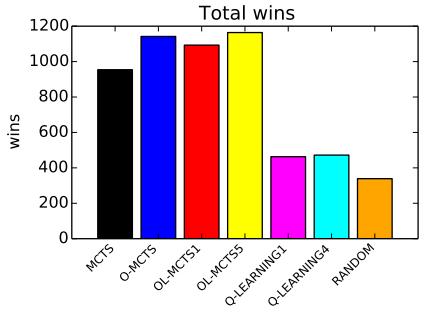



Figure 7.12: Sum of all wins of each algorithm.

# Chapter 8

# Discussion

## 8.1 Conclusion

We can conclude that the Option MCTS algorithm almost always performs at least as good as MCTS. The O-MCTS algorithm excels in games with both a small level grid or a small amount of sprites and high complexity, such as *zelda*, *overload* and *eggomania*. Furthermore, O-MCTS can look further ahead than most tree searching alternatives, resulting in a high performance on games like *camel race*, in which reinforcement is sparse. This confirms our hypothesis that using options, O-MCTS can win more games than MCTS. The algorithm performs worse than expected in games with a high amount of sprites, since the size of the option set becomes so large that maintaining it takes a lot of time, leaving too little time for tree building. Over all twenty-eight games, O-MCTS wins more games than MCTS.

The results of OL-MCTS indicate that it is possible to learn about which options work better, meaning that in the future it should be possible to completely remove infeasible options that have low expected rewards from the option set. We expect that this could reduce the computation time O-MCTS needs to construct and check all the options. However, more work needs to be done in this area to enable improvement.

## 8.2 Discussion

This section will first discuss the trade-offs between scoring, winning and playing time are discussed. Then, the evaluation of the SMDP Q-learning algorithm is discussed and lastly this section discusses why OL-MCTS does not introduce a significant improvement upon O-MCTS.

A difficulty in game solving research is the trade-off between scoring, playing time and winning. Winning earlier is often better, although for some games winning later increases the eventual score. For example, in the game *butterflies*, butterflies keep spawning indefinitely. Each butterfly that is caught, increases

the score. When all the butterflies are gone, the game is won. An algorithm that catches all the butterflies as quickly as possible will have a lower score than an algorithm that catches them less efficiently. The question to ask then is: which of these is better? Another trade-off between winning and scoring can, for example, be found in games such as *zelda*. The goal is to pick up the key and walk to the portal. If the monsters are killed, this increases the eventual score. But trying to kill a monster comes at a risk: the player has to come close to the monster in order to use his sword on it. So increasing the score, comes with the risk of dying, decreasing the win ratio. It is often unclear how to prioritize one of these objectives over the other.

A solution can be found in the area of multi-objective optimization. Multi-objective algorithms do not optimize one objective, but search for optimal solutions that maximize a combination of several objectives at once instead. An example of a multi-objective algorithm that has been used for general video game playing, is *multi-objective Monte Carlo tree search (MO-MCTS)* [29, 16].

For the comparisson in this thesis, both SMDP Q-learning and OL-MCTS had to adhere the same rules of the GVGAI competition. As a result, SMDP Q-learning was allowed one second of initialization time. However, SMDP Q-learning with options was originally proposed without any constraints on the initialization time or Q-table size [28]. Experiments with a longer initialization time could bear interesting resulds, because Q-learning could learn from more games without getting disqualified and its performance might improve.

The experiments that have been done in Chapter 7, all use the same option set. However, the addition of a bad option in the option set might add overhead without improving the mean score of the algorithm on any of the games. Bad options should be taken out of the option set, improving the performance of O-MCTS by removing overhead.More experiments need to be done to find out the influence of each of the options.

The OL-MCTS algorithm was constructed to reduce the overhead introduced by the O-MCTS algorithm, but offers little improvement on most games. The current hypothesis is that because the algorithm does not completely exclude the poorly performing options from the option set, little of the overhead is reduced. If a threshold is set to the option values below which the options are not maintained, the overhead of the algorithm could be reduced. This could, however, introduce overfitting: options might be excluded prematurely, e.g., the `GoToPosition` option for the portal in *zelda*, which is closed initially, should not be discarded because after picking up the key it becomes useful.

A similar limitation of the OL-MCTS algorithm is that it does not learn the conditions in which options are good: it only learns the mean and variance of an option's return. Although the crazy stone expansion algorithm is inclined to try options with a high variance, an algorithm that can learn the conditions in which these options result in a score improvement would yield better results.

## 8.3 Future Work

This section discusses future research that can be conducted to improve O-MCTS and OL-MCTS. We propose methods for getting a better understanding of the algorithms.

Firstly, more research should be done in the individual options. Currently the effectiveness of most of the options relies on the A Star implementation. This implementation is too time consuming, leaving little computation time for tree building. In future work, trials should be done with simpler and computationally cheaper alternatives, such as Enforced Hill Climbing as proposed in [22]. Although that algorithm has the problem that the agent can get stuck, its reduced computation time might make it better suited for our goal than the A Star algorithm. Alternatively, the human-defined option set can be replaced by creating goal-oriented MDPs similar to PUMA's [12] or by using MAXQ [7]. Automatically generated options have the advantage that they are generated for one game specifically, whereas our option set was created to work on as many games as possible. Because of the limitations posed by the GVGAI competition, more investigation should be done into how goal-oriented MDPs can be created with limited action time.

Another method to decrease the time it would take to maintain the option set is by removing options from the option set after several games have been won. By requiring the algorithm to first win some of the games, it is possible to know which options never contribute to a positive score change. These options can be removed from the option set, reducing the time it takes to maintain it.

Early in the development process, many of the options resulted in the agent getting killed because it was following a specific option for too long. For this reason interruption was introduced, as described in Section 2.3. Since then, the options have been improved to less often lead to the avatar's death, but because interruption was embedded in the algorithms it was never disabled. Although it is thought that this contributes to the performance of the algorithm, research needs to be done in the impact it has on the algorithms.

If interruption is disabled, some actions do not require any extra computation time, since no new option has to be selected. The computation time of these actions can be used to contribute to building the tree for the next option. This means that whereas normally a tree has to be built in the time for *one* action, with interruption disabled it can be built in the combined time of all the actions of an option. Since this multiplies the available computation time, it is expected to have a positive effect on the performance of O(L)-MCTS.

One of the algorithms described in Chapter 2 is Purofvio [20]. This algorithm is comparable to O-MCTS, but was only tested on the physical traveling salesperson problem. In order to compare the performance of O-MCTS to shis algorithm it should be run on the PTSP problem as well.

Powley et al. suggest that Purofvio always prefers longer options over shorter ones. We hypothesize that O-MCTS does not have this problem, due to the discounting that is done by the backup function. To support this claim, a test should be done by creating an option set with two options. One that achieves

a subgoal and one that achieves the same goal with more actions. If O-MCTS prefers the shorter option, we can conclude that our backup method indeed solves Purofvio's problem.

The complicated trade-off between winning, scoring and playing time has to be balanced by O-MCTS. This problem can be solved by using a multi-objective algorithm. By basing another version of O-MCTS on MO-MCTS [29] it might be possible to balance these trade-offs.

Lastly, in order to improve the learning algorithms, some other improvements can be investigated. Firstly, the backup method can be tweaked. In [6], instead of the mean value other values like the maximum return are used in the backup phase. This has a positive effect on the action (or in our case option) choices, resulting in a better performance. Furthermore, the mean and standard deviation of option returns are now calculated over all the games, without regarding how long ago this game was played. This might lead to underrated options, for example with doors that unlock under specific conditions (for example when the key is picked up in *zelda*). Using a maximum return or discounting the option values might have a different effect.

# Bibliography


[1] B. Abramson. Expected-outcome: A general model of static evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(2):182–193, 1990.

[2] A. G. Barto and S. Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[3] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, et al. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[4] B. Brügmann. Monte carlo go. Technical report, Citeseer, 1993.

[5] P. S. Castro and D. Precup. Automatic construction of temporally extended actions for mdps using bisimulation metrics. In *Recent Advances in Reinforcement Learning*, pages 140–152. Springer, 2012.

[6] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.

[7] T. G. Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *J. Artif. Intell. Res.(JAIR)*, 13:227–303, 2000.

[8] S. J. Duff and O. Bradtke Michael. Reinforcement learning methods for continuous-time markov decision problems. *Advances in Neural Information Processing Systems 7*, 7:393, 1995.

[9] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of uct with patterns in monte-carlo go. 2006.

[10] M. L. Ginsberg. Gib: Imperfect information in a computationally challenging game. *Journal of Artificial Intelligence Research*, pages 303–358, 2001.

[11] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[12] R. He, E. Brunskill, and N. Roy. Puma: Planning under uncertainty with macro-actions. In *AAAI*, 2010.

[13] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *Machine Learning: ECML 2006*, pages 282–293. Springer, 2006.

[14] D. Michie and R. A. Chambers. Boxes: An experiment in adaptive control. *Machine intelligence*, 2(2):137–152, 1968.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[16] D. Perez, S. Mostaghim, S. Samothrakis, and S. Lucas. Multi-objective monte carlo tree search for real-time games. 2015.

[17] D. Perez, S. Samothrakis, and S. Lucas. Knowledge-based fast evolutionary mcts for general video game playing. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.

[18] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. Lucas, A. Couëtoux, J. Lee, C.-U. Lim, and T. Thompson. The 2014 general video game playing competition.

[19] D. Perez Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. Lucas. Open loop search for general video game playing. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, pages 337–344. ACM, 2015.

[20] E. J. Powley, D. Whitehouse, P. Cowling, et al. Monte carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 234–241. IEEE, 2012.

[21] D. Precup. *Temporal abstraction in reinforcement learning*. PhD thesis, University of Massachusetts, 2000.

[22] B. Ross. General video game playing with goal orientation, September 2014.

[23] S. Samothrakis, D. Perez-Liebana, S. M. Lucas, and M. Fasli. Neuroevolution for general video game playing. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, pages 200–207. IEEE, 2015.

[24] T. Schaul. A video game description language for model-based or interactive learning. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8. IEEE, 2013.

[25] T. Schuster. Mcts based agent for general video games, July 2015.

[26] M. Stolle and D. Precup. Learning options in reinforcement learning. In *SARA*, pages 212–223. Springer, 2002.

[27] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 1998.

[28] R. S. Sutton, D. Precup, and S. Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1):181–211, 1999.

[29] W. Wang and M. Sebag. Multi-objective monte-carlo tree search. In *Asian conference on machine learning*, volume 25, pages 507–522, 2012.

[30] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[31] M. Wiering and M. van Otterlo. *Reinforcement Learning: State-of-the-art*, volume 12. Springer Science & Business Media, 2012.