Monte Carlo Tree Search with Options for General Video Game Playing

de Waard, M.; Roijers, D.M.; Bakkes, S.C.J.

DOI
10.1109/CIG.2016.7860383

Publication date
2016

Document Version
Final published version

Published in
2016 IEEE Conference on Computational Intelligence and Games (CIG 2016)

Link to publication

Citation for published version (APA):
de Waard, M., Roijers, D. M., & Bakkes, S. C. J. (2016). Monte Carlo Tree Search with Options for General Video Game Playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG 2016): Santorini, Greece, 20-23 September 2016* (pp. 47-54). IEEE. https://doi.org/10.1109/CIG.2016.7860383

Monte Carlo Tree Search with Options for General Video Game Playing

Maarten de Waard
University of Amsterdam
Science Park 904
Amsterdam, Netherlands
Email: mrtndwrd@gmail.com

Diederik M. Roijers
University of Oxford
Wolfson Building, Parks Road
Oxford, United Kingdom
Email: Diederik.Roijers@cs.ox.ac.uk

Sander C.J. Bakkes
Tilburg University
Warandelaan 2, Dante Building
Tilburg, Netherlands
Email: S.C.J.Bakkes@uvt.nl

***Abstract*—General video game playing is a challenging research area in which the goal is to find one algorithm that can play many games successfully. "Monte Carlo Tree Search" (MCTS) is a popular algorithm that has often been used for this purpose. It incrementally builds a search tree based on observed states after applying actions. However, the MCTS algorithm always plans over actions and does not incorporate any higher 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 paper, 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. Our new algorithms are compared to MCTS 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 outperforms MCTS on most of the games, 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.**

I. Introduction

Recent game programming research focusses on algorithms capable of solving several games with different types of objectives. 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)*.

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 [1]. 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.

A limitation in tree search algorithms is that since many games are too complex to plan far ahead in a limited time frame, many of these 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 make assumptions about its mechanics, e.g., pressing the left button often results in the player's avatar moving to the left on the screen. Furthermore, we expect human players to define specific subgoals for themselves, e.g., when there is a portal on screen, a player is likely to try to find out what the portal does by walking towards it. The player will remember the effect of this and use that information for the rest of the game.

In certain situations it is clear how such a subgoal can be achieved and a *policy*, which defines which actions to take in which state, can be defined to achieve it. A policy to achieve a specific subgoal is called an *option* [2]. Thus, an option selects an action, given a game state, that aims at satisfying its subgoal. In this paper, options are game-independent. The options are expected to guide the exploration of a game's search space to feasible areas.

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 more efficiently, at a higher level of abstraction. Furthermore, we introduce *option learning MCTS (OL-MCTS)*, an extension of O-MCTS that approximates which of the available options work well 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 to subsequent levels in order to increase performance.

We compare our algorithms to MCTS on games from the GVGAI competition. Our results indicate that the O-MCTS and OL-MCTS algorithms outperform 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 well as MCTS.

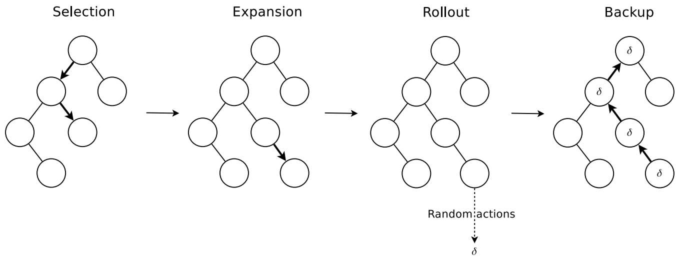


Fig. 1. One MCTS iteration. This process is repeated in order to improve the estimates of action values. Circles represent states, edges represent actions.

II. Background

We first explain the most important concepts needed to understand the algorithms that are proposed in this paper. We first describe *Markov decision processes (MDPs)*, then MCTS, then options and finally the *video game description language (VGDL)*.

A. Markov Decision Processes

We treat games as MDPs, which provide a mathematical framework for use in decision making problems. An MDP is 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 $R : S \times A \times S \to \mathbb{R}$. When the game score changes, the difference is viewed as the reward. Algorithms maximize the cumulative reward. In the scope of this paper algorithms only observe state transitions and rewards when they happen and do not have access to T and R.

B. Monte Carlo Tree Search

The success of MCTS started in 2006, when the tree search method and UCT formula were introduced, yielding good results in Computer Go [3]. Since 2006, the algorithm has been extended with many variations. It is still being used for other computer games [4], including the GVGAI competition [5]. In this paper, we use MCTS as the basis for the new algorithms.

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. Each node in the tree represents a state and each edge represents an action taken in that state. MCTS consists of four phases that are constantly repeated, as depicted in Figure 1. The root node of the tree represents the current game state, Then, the first action is chosen by an *expansion* strategy and subsequently simulated. This results in a new game state, for which a node is created. After expansion, a *rollout* is done from the new node, which means that a simulation is run from that node, applying random actions until a predefined stop criterion is met. Finally, the score difference resulting from the rollout is *backed up* to the root node, which means that the reward is saved to all visited nodes, after which 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 selection strategy is UCT, which employs an exploration bonus to balance 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 [6]. A child node j is selected to maximize

$$UCT = v_{s'} + C_p \sqrt{\frac{2 \ln n_s}{n_{s'}}} \tag{1}$$

Where $v_{s'}$ is the value of child s' as calculated by the backup function, n_s and $n_{s'}$ are the number of times nodes s and child s' have been visited and $C_p > 0$ is a constant that shifts priority from exploration to exploitation.

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. 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 [7], which is an expansion strategy that was originally designed for Go. We will use an adaptation of this strategy in the algorithm proposed in Section V. When using crazy stone, an action i is selected with a probability proportional to u_i

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

Each action has an estimated value μ_i ordered in such a way that $\mu_0 > \mu_1 > \cdots > \mu_N$, and a variance σ_i^2. K is a constant that influences the exploration — exploitation trade off. ε_i prevents the probability of selecting a move to reach zero. Its value is proportional to the ordering of the expected values of the possible actions: $\varepsilon_i = \frac{0.1+2^{-i}+a_i}{N}$. Here, 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.

C. Options

In order to mimic human game playing strategies, such as defining subgoals and subtasks, we use options. Options, or macro-actions, have been proposed by Sutton et al. [2] as a method to incorporate temporal abstraction in reinforcement learning. The majority of the research seems to focus on learning algorithms, little work has been done on combining options with tree search methods [8], although most learning algorithms are time and memory heavy and tree search methods have shown more promising results on complex games.

An option 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, $\pi : S \times A \rightarrow [0, 1]$ is a policy and $\beta : S^{+} \rightarrow [0, 1]$ is a termination condition.

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 π 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 β. This 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.

A popular algorithm that uses options instead of actions is SMDP Q-learning [2]. In general, it estimates the expected rewards for using an option in a certain state, in order to find an optimal policy over the option set.

D. General Video Game Playing

We use the general video game playing problem as a benchmark for our algorithms. Recent developments in this area include VGDL [9], a framework in which a large number of games can be defined and accessed in a similar manner. Using VGDL, algorithms can access all the games similarly, resulting in a method to compare their performances on several games.

The GVGAI competition provides games written in VGDL. The games function as a black box from which algorithms can only observe the game state. In each game tick, algorithms have limited time to plan their action, during which they can access a *forward model*, which simulates new states and rewards for actions. The actions that are used on the forward model do not influence the real game score. Algorithms should return actions before their simulation time runs out.

The algorithms proposed in this paper will be benchmarked on the GVGAI game sets, using the rules of that 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.

III. Related Work

This section covers some popular alternative methods for general video game playing and prior work on tree search with options.

Deep Q networks (DQN) [10] 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. In this case 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.

Another alternative is the algorithm *Planning under uncertainty with Macro-Actions (PUMA)*, which applies forward search to options and works on *Partially Observable MDPs (POMDPs)* [11]. PUMA automatically generates goal-oriented MDPs for specific subgoals, the advantage of which is that effective options can be created without requiring any prior knowledge of the POMDP. The disadvantage is that this takes a lot of computation time and thus would not work in the GVGAI framework, where only 40 milliseconds of computation time is allowed between actions. Furthermore PUMA has to find out the optimal length per macro-action, our algorithm can use options of variable length with starting an stopping conditions.

Another algorithm that uses MCTS with macro-actions is called *Purofvio*. Purofvio plans over simple macro-actions which are defined as repeating one action several times [12]. No more complex options are defined. All the options are of exactly the same size. Purofvio is created solely for the physical travelling salesperson problem. Although Purofvio could also work on other games, we decided to create a different algorithm, that is capable of using more complex options.

IV. O-MCTS

We propose O-MCTS, a novel algorithm that simulates the use of subgoals by planning over options using MCTS, enabling the otherwise infeasible use of options in complex MDPs. The resulting algorithm achieves higher scores than MCTS on complex games that have several subgoals. It works as follows: like in MCTS, a tree of states is built by simulating game plays. The algorithm chooses options instead of actions. When an option is chosen, each next node in the search tree represents an action chosen by that option. The search tree can only branch if an option is finished, i.e., its subgoal is reached. 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 section describes how the process works in more detail.

The tree representation of an O-MCTS tree is the same as in MCTS: a node represents a state, a connection represents an action. An option spans several actions and therefore several nodes in the search tree, as shown in Figure 2. 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 β), a new option can be chosen by the expansion or selection strategy. The search tree can only branch when an option is finished.

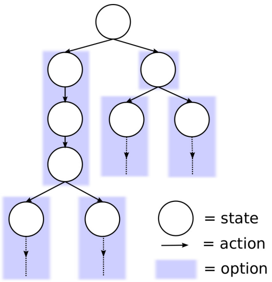


Fig. 2. The search tree constructed by O-MCTS. In each blue box, one option is followed. The arrows represent actions chosen by the option. An arrow leading to a blue box is an action chosen by the option represented by that box.

Algorithm 1 $\mathsf{O-MCTS}(O, r, t, d)$

```
 1: $C_{s \in S} \leftarrow \emptyset$                      ▷ $\mathbf{c}_s$ is the set of children nodes of $s$
 2: $\mathbf{o} \leftarrow \emptyset$                       ▷ $o_s$ will hold the option followed in $s$
 3: while $time\_taken < t$ do
 4:     $s \leftarrow r$                                    ▷ start from root node
 5:     while $\neg\mathsf{stop}(s, d)$ do
 6:         if $s \in \beta(o_s)$ then                      ▷ if option stops in state $s$
 7:             $\mathbf{p}_s \leftarrow \cup_o(s \in I_{o \in O})$   ▷ $\mathbf{p}_s$ = available options
 8:         else
 9:             $\mathbf{p}_s \leftarrow \{o_s\}$           ▷ continue with current option
10:         end if
11:         $\mathbf{m} \leftarrow \cup_o(o_{s \in \mathbf{c}_s})$   ▷ set $\mathbf{m}$ to expanded options
12:         if $\mathbf{p}_s = \mathbf{m}$ then             ▷ if all options are expanded
13:             $s' \leftarrow \max_{c \in \mathbf{c}_s} \mathsf{uct}(s, c)$   ▷ Eq. 1
14:             $s \leftarrow s'$                           ▷ continue loop with child
15:         else
16:             $\omega \leftarrow \mathsf{random\_element}(\mathbf{p}_s - \mathbf{m})$
17:             $a \leftarrow \mathsf{get\_action}(\omega, s)$
18:             $s' \leftarrow \mathsf{expand}(s, a)$       ▷ create child $s'$ using $a$
19:             $\mathbf{c}_s \leftarrow \mathbf{c}_s \cup \{s'\}$   ▷ add to set of children
20:             $o_{s'} \leftarrow \omega$
21:             break
22:         end if
23:     end while
24:     $\delta \leftarrow \mathsf{rollout}(s')$               ▷ simulate until stop
25:     $\mathsf{back\_up}(s', \delta)$    ▷ save reward to parent nodes (Eq. 3)
26: end while
27: return $\mathsf{get\_action}(\max_{o \in \mathbf{c}_r} \mathsf{value}(o), r)$
```

We describe O-MCTS in Algorithm 1. 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. The set of options used for our experiments is described in Section VI. 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* and the current state s shows no NPCs on screen. If o is an option for avoiding NPCs, 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.

O-MCTS consists of the same four phases as MCTS. 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 δ. 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.

A number of functions is used by Algorithm 1. 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 ω 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:

$$v_s \leftarrow v_s + \delta\gamma^{d_{s'} - d_s}, \tag{3}$$

where δ 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.

When the time limit is reached, the algorithm chooses an option from the children of the root node, $\mathbf{c}_r$, 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.

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. This is illustrated in Figure 2, where the tree can not branch inside blue boxes. Furthermore, we expect that the algorithm will be able to identify and meet a game's subgoals by using options. In the experiments section we show results that support our expectations.

V. OL-MCTS: Learning Option Values

Although we expect O-MCTS to be 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 section, we will define *option values*: the expected mean and variance of an option.We adjust O-MCTS to learn the option values and focus more on the options with higher option values. Especially when a level is played several times, we expect this to be advantageous. 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, μ and σ. The expected mean return of an option o is denoted by μ_o. This state-independent number represents the returns that were achieved in the past by an option for a game. Similarly, the variance of all the returns of an option o is saved to σ_o.

For the purpose of generalisation, 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 μ_o and σ_o are updated by respectively taking the mean and variance of all the returns of this subtype. This enables the algorithm to generalize over subtypes.

Using option values, we can incorporate the progressive widening algorithm from Equation 2, crazy stone, 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 (Algorithm 2) 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 μ_o and variance σ_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. If the node is visited more than v times, it 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 ω by using these weights. If ω 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 ω 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.

Algorithm 2 $\mathsf{OL-MCTS}(O, r, t, d, v, \mu, \sigma)$

```
 1: $C_{s\in S} \leftarrow \emptyset$
 2: $\mathbf{o} \leftarrow \emptyset$
 3: while $time\_taken < t$ do
 4:     $s \leftarrow r$
 5:     while $\neg\mathsf{stop}(s, d)$ do
 6:         if $s \in \beta(o_s)$ then
 7:             $\mathsf{update\_values}(s, o_s, \mu, \sigma)$    ▷ update $\mu$ and $\sigma$
 8:             $\mathbf{p}_s \leftarrow \cup_o(s \in I_{o\in O})$
 9:         else
10:             $\mathbf{p}_s \leftarrow \{o_s\}$
11:         end if
12:         $\mathbf{m} \leftarrow \cup_o(o_{s\in\mathbf{c}_s})$
13:         if $n_s < v$ then        ▷ if state is visited $< v$ times
14:             $\mathbf{u}_s \leftarrow \mathsf{crazy\_stone}(\mu, \sigma, \mathbf{p}_s)$        ▷ Eq. 2
15:             $\omega \leftarrow \mathsf{weighted\_random}(\mathbf{u}_s, \mathbf{p}_s)$
16:             if $\omega \notin \mathbf{m}$ then        ▷ option $\omega$ not expanded
17:                 $a \leftarrow \mathsf{get\_action}(\omega, s)$
18:                 $s' \leftarrow \mathsf{expand}(s, a)$
19:                 $\mathbf{c}_s \leftarrow \mathbf{c}_s \cup \{s'\}$
20:                 $o_{s'} \leftarrow \omega$
21:                 break
22:             else                ▷ option $\omega$ already expanded
23:                 $s' \leftarrow s \in \mathbf{c}_s : o_s = \omega$    ▷ child that uses $\omega$
24:             end if
25:         else                                    ▷ apply UCT
26:             $s' \leftarrow \mathsf{uct}(s)$
27:         end if
28:         $s \leftarrow s'$
29:     end while
30:     $\delta \leftarrow \mathsf{rollout}(s')$
31:     $\mathsf{back\_up}(s', \delta)$
32: end while
33: return $\mathsf{get\_action}(\max_{o\in\mathbf{c}_r} \mathsf{value}(o), r)$
```

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.

VI. Experiments

In this section we describe our experiments on O-MCTS and OL-MCTS. The algorithms are compared to the MCTS algorithm, as described in Section II-B. 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). Each game has five levels.

Firstly, we compare O-MCTS to MCTS by showing the win ratio and mean score of both algorithms on all the games. Secondly we show the improvement that OL-MCTS makes compared to O-MCTS when it is allowed 4 games of learning time. Lastly we compare the three algorithms by summing up all the victories of all the levels of each game.

For these experiments we construct an option set which is aimed at providing action sequences for any type of game, since the aim here is general video game playing. Note that since the option set is a variable of the algorithm, either a more specific or automatically generated set of options can be used by the algorithm as well.

The set of option types consists of one option that executes a specific action once, an option that avoids the nearest NPC by moving away from it, an option that moves to a movable sprite until it is close to it (but not on it). There are options that go to a movable sprite and options to go to a certain position in the game. Lastly we create an option that waits until an NPC is at a certain distance and then fires the weapon. The specifics about the option set can be found in the original thesis [13].

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* 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. *go to movable* and *go near movable* options are created for each of the three monsters and for the key. A *go to position* option is created for the portal. One *go to nearest sprite of type* option is created per sprite type. One *wait and shoot* option is created for the monsters and one *avoid nearest NPC* option is created. This set of options is O, as defined in Section II-C. In a state where, for example, all the monsters are dead, the possible option set $\mathbf{p}_s$ does not contain the *avoid nearest NPC*, *go to movable* and *go near movable* options for the monsters.

The *go to …* and *go near …* options utilize an adaptation of the A^* algorithm to plan their routes [14]. 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^* 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. Our A^* keeps a *wall score* for each sprite type. When a sprite blocks the avatar, its wall score is increased by one. Additionally, to prevent the avatar from walking into deadly sprites, when a sprite kills the avatar, its wall score is increased by 100. Traditionally the A^*'s heuristic uses the distance between two points. Our A^* adaptation adds the wall score of the goal location to this heuristic, encouraging the algorithm to take paths with a low wall score. This method enables A^* 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^* implementation will still be able to plan a path to the door once the key is picked up, to win the game. Note that because the games can be stochastic, A^* has to be recalculated for each simulation.

We empirically optimize the parameters of the algorithms for the experiments. We use discount factor $\gamma = 0.9$, maximum action time $t = 40$ milliseconds. 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. This is possible because the search tree has a relatively low branching factor. The number of node visits after which uct is used, v, is set to 40. Crazy stone parameter K is set to 0.5. For comparison, we use the MCTS algorithm provided with the Java implementation of VGDL which employs a maximum tree depth of 10, because the branching factor is higher. Both algorithms have uct constant $C_p = \sqrt{2}$. Unfortunately, comparing to Q-learning with options was impossible, because the state space of these games is too big for the algorithm to learn any reasonable behavior. All the experiments are run on an Intel 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.

First, we will describe the results of the O-MCTS algorithm in comparison with MCTS. This demonstrates the improvement that can be achieved by using our algorithm. The games in this and the following experiments are ordered from left to right by the performance of an algorithm that always chooses a random action, indicating the complexity of the games. Figure 3 shows 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 more than half.

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

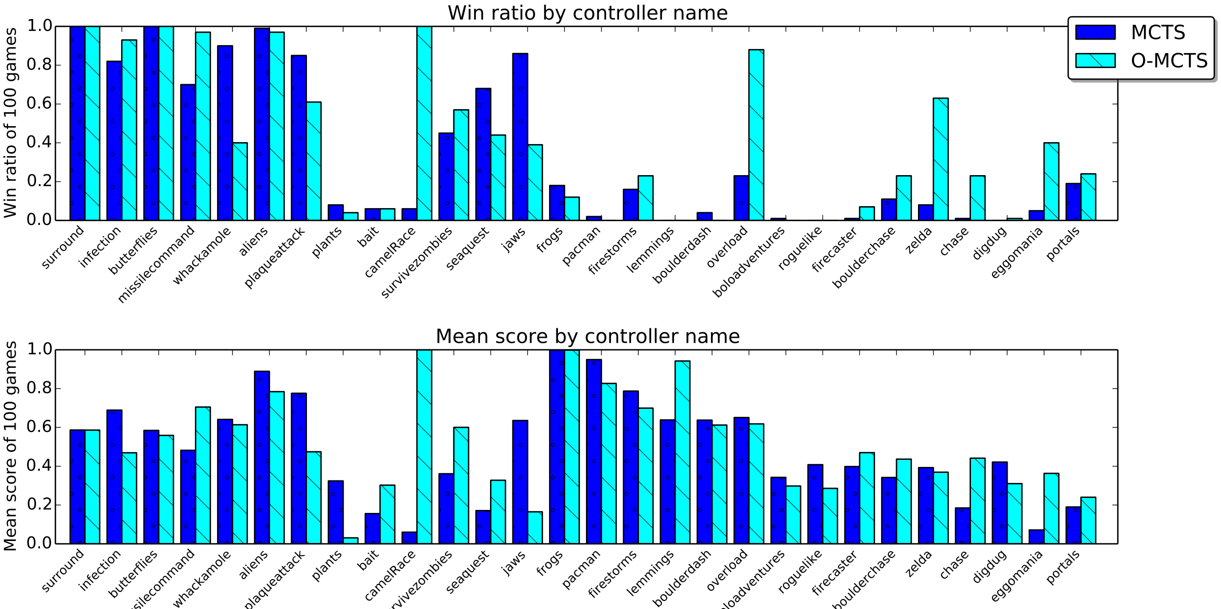


Fig. 3. Win ratio and mean normalized score of the algorithms per game. O-MCTS outperforms MCTS.

search than MCTS. *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 *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 big number of sprites, for each of which several options 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 ran the same test with an increased computation time of 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*.

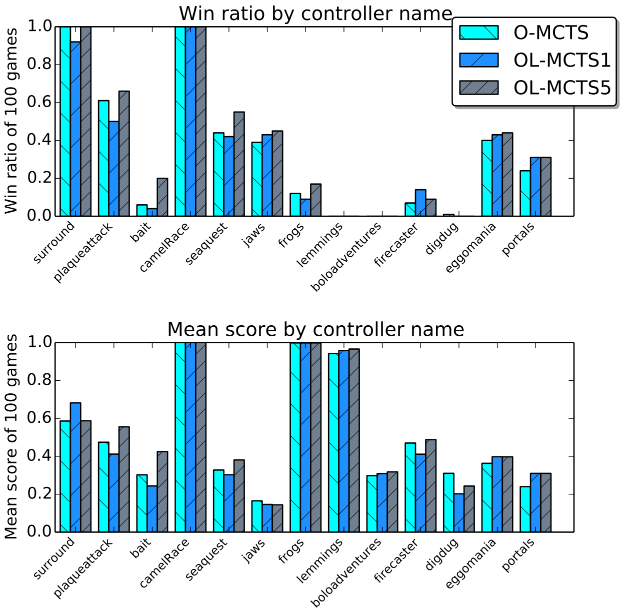
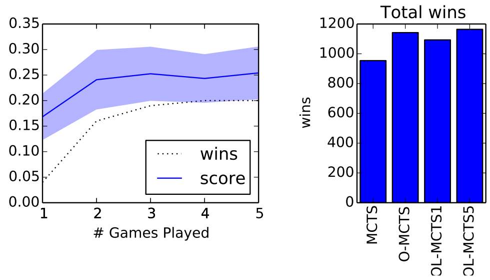


Fig. 4. Normalized win ratio and score comparison of OL-MCTS and O-MCTS. OL-MCTS outperforms O-MCTS by a small margin in some games. In the games that are not shown both algorithms perform equally.

Secondly, we compare OL-MCTS to O-MCTS by running it on the same set of games. The option learning algorithm is allowed four learning games, after which the fifth is used for the comparisons. Figure 4 shows the performance difference between O-MCTS and OL-MCTS on some games. For the other games, the performance was approximately the same. Here OL-MCTS1 shows the performance of OL-MCTS on the first game. OL-MCTS5 shows the performance of the



(a) Learning *bait* (b) Totals

Fig. 5. Learning improvement on game *bait*, it shows win ratio and normalized score. Total number of wins of the algorithms on all games.

algorithm after learning for four games.

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 option values has to take place in each tree node. The learning algorithm significantly improves score and win ratio for the game *bait*. Figure 5(a) 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 the algorithm learns to avoid 2. The algorithm learns that it should pick up the key.

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

A. Results

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 performance can be found.

An overview of the results is depicted in Figure 5(b), which 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 between performance of OL-MCTS over O-MCTS, although our results suggest that it does improve for a subset of the games.

VII. Conclusions and Future Work

From the experimental results we may conclude that the O-MCTS algorithm almost always performs at least as well as MCTS. It 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. An inherent advantage of having deep search trees is that the probability of an promising option not finishing reduces. We confirm our hypothesis that by 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, the algorithm can be further improved.

Furthermore, more research should be done in the influence of the option set. The A^* algorithm could be replaced by a simpler algorithm, such as Enforced Hill Climbing [15]. The learning algorithm could be improved by calculating the option values differently. An alternative method can use discounting in order to prioritize more recent observations.

References

[1] 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."

[2] R. S. Sutton, D. Precup, and S. Singh, "Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning," *Artificial intelligence*, vol. 112, no. 1, pp. 181–211, 1999.

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

[4] 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*, vol. 4, no. 1, pp. 1–43, 2012.

[5] 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*. IEEE, 2014, pp. 1–8.

[6] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Machine Learning: ECML 2006*. Springer, 2006, pp. 282–293.

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

[8] A. G. Barto and S. Mahadevan, "Recent advances in hierarchical reinforcement learning," *Discrete Event Dynamic Systems*, vol. 13, no. 1-2, pp. 41–77, 2003.

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

[10] 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.

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

[12] 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*. IEEE, 2012, pp. 234–241.

[13] M. de Waard, "Monte carlo tree search with options for general video game playing," Ph.D. dissertation, University of Amsterdam, February 2016.

[14] 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*, vol. 4, no. 2, pp. 100–107, 1968.

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