Self-Adaptive Multi-Objective Option Monte-Carlo Tree Search for General Video Game Play

Stefan Bussemaker (s2004674)

May 13, 2019

Supervisors: Dr. Marco Wiering (Artificial Intelligence, University of Groningen)
and Prof. Dr. Rineke Verbrugge (Artificial Intelligence, University of Groningen)

Artificial Intelligence
University of Groningen, The Netherlands

Abstract

General video game playing (GVGP) challenges agents to play a suite of games in a real-time fashion. Monte Carlo Tree Search (MCTS) has shown particular success and many enhancements have been proposed and tested with the goal of improving its performance.

This research looks at multiple MCTS variations and tests to see whether game playing performance can be increased by combining them. The three variations tested are self-adaptive MCTS, multi-objective MCTS and MCTS with macro-actions. Eight agents are constructed from every possible combination of these variations. Each agent is tested on a game set of 20 games from the General Video Game AI (GVGAI) framework.

The results show that combining multi-objective MCTS with macro-actions yielded the biggest improvement in games won, increasing the performance of the vanilla MCTS by 13.7%. Overall, each variation outperformed vanilla MCTS. However, win rate improvement was not consistent across all games and differed between different algorithm combinations. This means that different combinations excel at different games.

Acknowledgements

First of all, I am grateful for the support and guidance of Marco Wiering and Rineke Verbrugge during this project. I would also like to thank Rebecca Adrian for her endless faith in me and the process. Furthermore, I would like to thank Patrick van der Zwan for spending many hours studying together. Finally, I would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

Contents

List of Figures

List of Tables

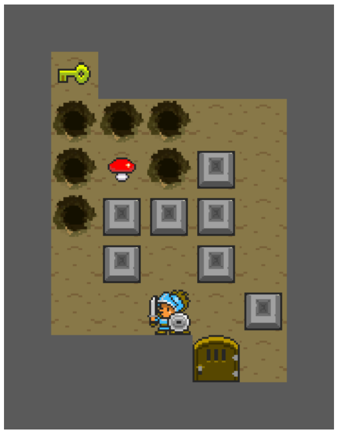
List of Algorithms

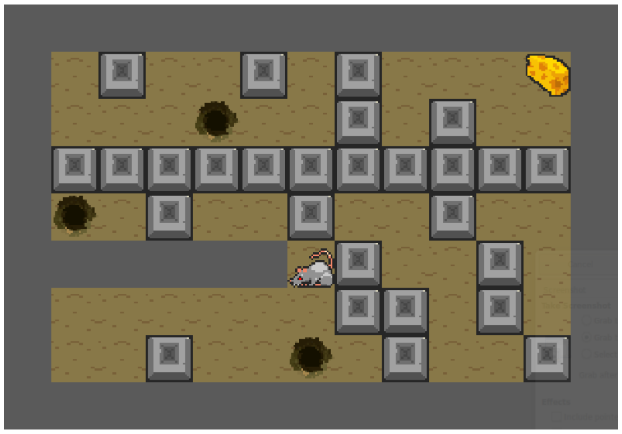
Chapter 1: Introduction

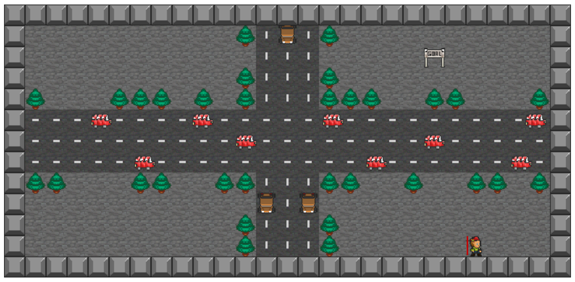
1.1 Background

Game playing is a popular topic in AI research. Games offer a simplified world and can therefore easily be used as a testing environment for new techniques, which, in turn, might transfer to more difficult real-world problems [1]. Early game play research focused on deterministic games like chess, leading to a computer winning against the best human player in 1997 [2]. This was a major feat, as it was previously thought impossible for a computer to outperform the best human chess player. Research on game play progressed steadily, and in 2016, Deepmind managed to achieve victory in the game of Go, using the AlphaGo program [3]. While traditional chess playing algorithms relied on searching through the game tree and exploring as many moves as possible, this approach is not feasible for Go. A turn in Go consists of over 100 possible moves, resulting in a far greater search space than there is for chess, which has only 35 possible moves on average per player. Also, unlike in chess, there is no easy heuristic to evaluate game states in Go. This made outperforming high-ranking Go players an important achievement [4]. Since then, the focus has shifted towards other domains such as real-time game play, in which a computer agent has to choose which action it is going to play within a couple of milliseconds. This strict time budget poses the extra challenge of having to plan very efficiently. A popular example of the types of games that are being researched is StarCraft [5].

As research on game playing continues, researchers have been able to design agents that are becoming increasingly capable of playing games successfully. These agents, however, have a narrow domain, as they can only perform well on the specific game that they were designed for. Although an agent might be a competent player in one game, it is not necessarily competent in playing other games. One of the challenges of AI is to invent general intelligence, a form of intelligence that not only spans a single task, but can cope with, and adapt to, multiple and possibly previously unseen domains [6]. For this reason, it would be of interest to design a flexible algorithm that would be capable of playing various games instead of a single one. This has led


(a) Bait


(b) Escape


(c) Intersection

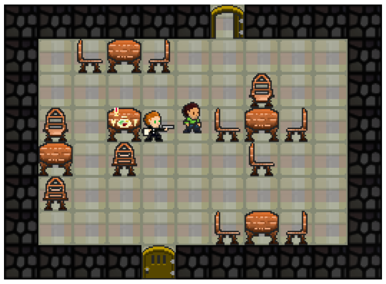

(d) Wait for Breakfast

Figure 1.1: Examples of games for the GVGAI games suite

to the emergence of the subfield General Game Play (GGP). The first attempt was the GGP framework and the accompanying competition [7], which facilitated a suite of board games including Checkers, Tic Tac Toe and Connect Four. The framework and competition could be used to test and compare performances of agents that used different algorithms. The goal is to design an agent using a subset of the available games, while said agent is also able to play new and previously unseen games. This forces the designer to implement broad and general approaches and no longer depend solely on game-specific prior knowledge.

1.1.1 General Video Game Playing

Requiring agents to be able to play multiple (possibly unseen) games in real time poses an interesting challenge. There are several competitions centered around this challenge. One of these competitions is the Arcade Learning Environment (ALE) [8]. In ALE, agents have to play multiple real-time games while only having the individual pixels input available to them. This means that the agents need to deal with a large game state space combined with strict time constraints. Another competition for general video game play, which followed GGP and ALE, is the General Video Game AI (GVGAI) competition [9]. Several of the games that are included in this competition are depicted in Figure 1.1. GVGAI differs from ALE in that agents are given the current high-level state of the game at each time step, allowing the agents to reason at a higher level. A game state consists of game state variables, such as the state of the agent, which consists of variables like its position, orientation, resources, health points and history of collisions, and additionally types and positions of the different objects in the game. This allows agent designers to focus more on the search aspect of the challenge, without having to worry about lower-level processes like image recognition needed to detect objects.

In this research, we will use the GVGAI framework[1] to study general video game play. The framework offers multiple video games implemented in the Video Game Description Language (VGDL) [10]. The VGDL implementations are not available to the game agent, with the framework only exposing the current state of the game. This ensures that agents are unable to cheat by simple exploitation of the game rules. For this reason, it is up to the agent to figure out how to play and win the game. While many techniques have been tested using the GVGAI framework, current research shows that the challenge of general video game playing is far from solved, as the current best agent only wins about 50% of the games in the test set [11]. In particular, games with a large state space or with delayed rewards are difficult for the agents. Searching the state space is extra time-consuming and the lack of intermediate rewards means that there is no indication to determine whether one action is better than another. One of the promising approaches to tackle this problem is Monte Carlo Tree Search (MCTS) [9].

1.1.2 Monte Carlo Tree Search

For an agent to play a game, it has to decide at every time step which move to play. Monte Carlo Tree Search (MCTS) is a tree search method used to find the optimal policy in such a decision process and gained popularity from its success with Go [12]. Since then, it has been researched extensively and implemented quite successfully in game play [13]. The tree search method was also used by the winner of the first edition of the GVGAI competition, who implemented an agent using open-loop MCTS. From

[1]The GVGAI framework is designed and developed by the Games Intelligence Group at the University of Essex. More information can be found at `http://www.gvgai.net/`

that moment onwards, most entries in the GVGAI competition have been agents based on MCTS that have generally outperformed other approaches [14].

With MCTS, the agent starts a turn with a game state tree consisting only of a root node containing the current state. Possible outcomes are examined by simulating how the game would play out following any of the available actions and evaluating the expected rewards. This way, actions are evaluated and the tree is built iteratively by evaluating and adding additional actions to the tree asymmetrically by exploring the most promising actions first. These are both important properties of MCTS, and part of what makes it so useful. One of the requirements of video game play is that agents operate in real time, on a strict time budget. They simply cannot determine the best action to play by exhaustively building the entire game state tree. Due to the high branching factor, there are too many possible states to consider. Therefore, with each iteration, choices have to be made about which actions to expand and evaluate. A depiction of such an iteration of MCTS is shown in Figure 1.2 and consists of four steps:

Selection A child node selection policy is applied, starting from the root node until an expandable node is found. A novel action is selected, which will be played.

Expansion The action is played by simulating the outcome. This new resulting state is added to the game tree.

Simulation Until a certain criterion is met, play continues by taking random actions, usually until some predetermined rollout depth is reached.

Backpropagation The reward obtained by taking this action and letting it play out gets propagated back up the tree to update the node statistics, such as the number of node visits and accumulative reward, and these averages are used in the following iteration to guide the next search.

There are two distinct policies that these steps use: the selection and expansion steps use a *tree policy* and the simulation uses a *default policy*. Backpropagation does not use a policy, but instead updates the statistics in the nodes. The tree policy is concerned with which action is to be considered next. A common policy to use is Upper Confidence bounds for Trees (UCT) [15], which balances between choosing an action that hasn't been sampled before (exploration) and choosing an action that is known to lead to good results (exploitation). Using UCT, each action gets assigned a value, which is computed using the formula as follows:

$$a^* = \operatorname*{arg\,max}_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{1.1}$$

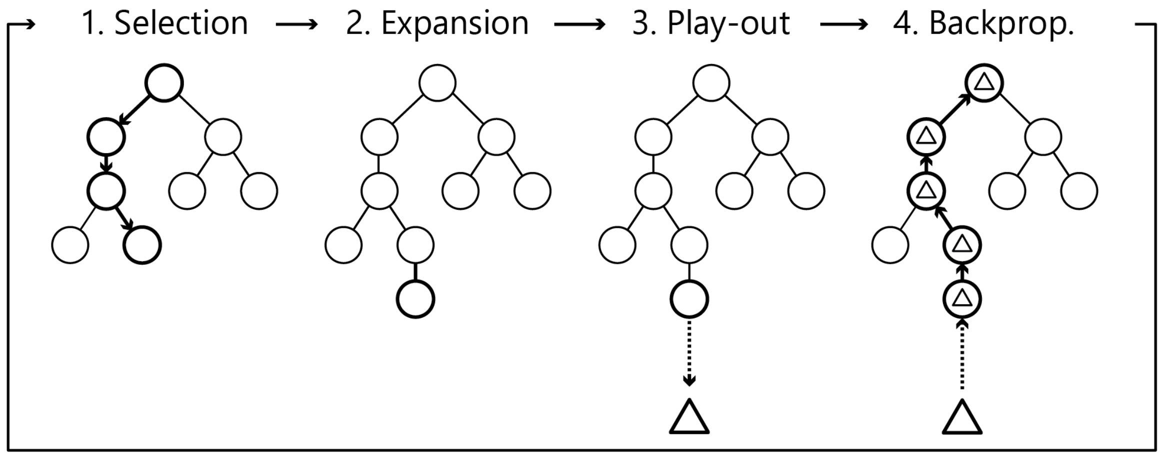


Figure 1.2: An iteration of Monte-Carlo Tree Search. The selection and expansion steps use a tree policy, while the simulation step uses a default policy. The final step is backpropagation.

For each node, the best candidate action a^* to consider from the available actions $A(s)$, can be influenced by UCT's parameter C. $Q(s,a)$ is the currently known average value of state s when playing a from the current node. Consequently, lower values of C lead to exploitation, while higher values of C put the emphasis on the second term, making the exploratory potential $\sqrt{\frac{\ln N(s)}{N(s,a)}}$ weigh heavier. This potential depends on $N(s)$, how often a certain state was visited, and $N(s,a)$, how often the currently considered action was already played from this state. When a node is found that is not yet fully expanded, a new child node is added. This way the number of children for each node equals the number of actions the agent can perform. Since building the game state tree is an iterative process, it can be cancelled at any time, making MCTS an *anytime* algorithm. This comes in handy in video game playing, as it enables the agent to stop building the search tree when the current time step comes at an end. In order to prevent the search algorithm from pursuing one branch of the tree until the entire time budget is consumed, an additional measure is usually used: an upper rollout depth limit. This builds the game state tree to a certain depth, allowing the search algorithm to explore other nodes once the current one is explored far enough into the future.

Once the search time budget is depleted, an action to play in the actual game needs to be selected, which is done using a policy called the *recommendation policy.* This policy is used to choose between the available actions. An example of what is used as recommendation policy is *greedy*, which involves simply choosing the action with the highest expected value. A drawback of this approach is that the agent can get stuck in local optima. Therefore, UCT is more commonly used to encourage a combination of both exploration and exploitation.

Many adaptations and enhancements of MCTS have been proposed and tested, all with varying results. One of these extends MCTS with online parameter tuning [16]. Since we are dealing with a suite of games instead of a single game, it might be difficult to come up with one parameter setting which will perform well on all games. By adapting the parameters during play, more appropriate settings for the current game may be used, which will potentially lead to better results. A second approach is MCTS with macro-actions [17]. Planning with macro-actions allows the agent to plan at a higher level, enabling it to search deeper into the game state tree. A third promising technique is MCTS with multiple objectives [18]. Instead of just evaluating the obtained game state after a rollout on the game score as evaluation function, it can be interesting to add more objectives between which the agent can balance. While these approaches have obtained decent results on their own, there is no research yet on how these techniques might work together. It will be interesting to take a look at whether or not the results from the GVGAI framework will be improved when mixing these different extensions.

1.2 Research Questions

In order to find out how different enhancements of MCTS perform when combined, we formulate the following research question:

1. *How does the performance of general video game play compare between agents using different combinations of self-adapting MCTS, MCTS with macro-actions and multi-objective MCTS?*

As this question is very broad and unspecific, it can be divided into multiple smaller sub-questions, targeting the specific algorithms to be examined:

2. *How does the performance of agents using self-adapting MCTS compare to agents without self-adapting MCTS on general video game play?*

3. *How does the performance of agents using MCTS with macro-actions compare to agents with MCTS without macro-actions on general video game play?*

4. *How does the performance of agents using multi-objective MCTS compare to agents without multi-objective MCTS on general video game play?*

1.3 Outline

The aim of this thesis is to give some insight into how well a mix of different MCTS approaches perform on the GVGAI framework. Chapter 2 will describe the approach of MCTS with online parameter tuning. Chapter 3 explains the use of MCTS with macro-actions. Chapter 4 explains MCTS with multi-objective optimization. Chapter 5 shows how these approaches can be mixed, which will form a new hybrid approach. The experimental work is discussed in Chapter 6 along with the obtained results. Chapter 7 draws a conclusion from the research and contains a discussion and proposed future work.

Chapter 2: Self-adapting MCTS

Like most other playing strategies, MCTS is controlled by one or more parameters. The performance of these strategies is dependant on what values these parameters are assigned. Whereas a low value of a certain value may result in the agent exhibiting a certain behavior, a high value might result in completely different expressed behavior. Therefore, game play performance is influenced by the parameters the algorithm was assigned, and higher performance can be gained by setting these parameters to good values. This tuning of the parameter values usually happens offline after the game has finished. The progression then goes as follows: a game is played, the outcome is assessed and the values are subsequently altered, after which the next game is played with the new settings. This tuning is not always an option though, as offline parameter tuning is both time-consuming and game dependant.

2.1 Online Parameter Tuning

Changing parameter values whilst playing a game has the benefit of having an agent that can adapt to the game whilst it is playing it. Self-adaptive MCTS has been used successfully in GGP, where experiments showed that agents with online tuning obtained similar results as agents that received offline tuning [19]. Online tuning allowed the agents to adapt to the current game that it was playing. Agents with online tuning have also been tested on the GVGAI framework. There, they scored similar to agents that use vanilla MCTS on the games they already performed well on, while improving the performance of games that the agents that use vanilla MCTS performed poorly on [16]. This shows that self-adaptive agents performed more robustly.

There are many ways to try and pick well-performing parameter values. One way is to view this as a multi-armed bandit (MAB) problem [20]. With this type of problem, there is a fixed set of competing possibilities with unknown reward distributions, where the agent has to choose which arm (choice) to pull (sample) in order to maximize their expected gain. A parameter corresponds to such a MAB, having a set of n possible values it can take. The properties of the possibilities are unknown, but can be estimated by trying them out a couple of times as a MAB problem has a set of n reward distributions $R = \{R_1, ...R_n\}$, corresponding to each of the arms. However, since we are dealing with multiple parameters, we will consider the variation of the Combinatorial Multi-Armed Bandit (CMAB) [21]. The CMAB problem tries to optimize the reward for problems with multiple bandits instead of a single one and consists of the following components:

- A set of n parameters $P = \{P_1, ..., P_n\}$, where parameter P_i can take m_i different values $V_i = \{v_i^1, ..., v_i^{m_i}\}$. Each of these values can be called a *local* arm. Then let $P = \{(v_1, ..., v_n) \in P_1 \times ... \times P_n\}$ be the set of possible parameter combinations. Each of these combinations can be called a *global* arm.
- An unknown reward distribution $R : V_1 \times ... \times V_n \to \mathbb{R}$ over each global arm.
- A function $L : V_1 \times ... \times V_n \to \{true, false\}$ that determines whether each global arm is valid and legal.

Multiple strategies to select a global arm in the CMAB problem can be designed. Since parameter values are usually interdependent, tuning strategies should aim to tune the parameters together. Such allocation strategies will iteratively sample the search space of possible arms and balance between exploration and exploitation in order to converge to the best possible arm. Since a MAB contains one parameter while the CMAB contains n parameters, the number of possible global arms increases exponentially with the number of parameters.

2.2 Allocation Strategy

The purpose of an allocation strategy is to find to best possible parameter value or combination of values when dealing with multiple parameters. This is done by trying to assign the highest number of samples to the best set, while minimizing the samples assigned to bad sets. The simplest way to select parameter combinations to play is by picking randomly. The agent can choose uniformly from the set of preselected parameter settings. This way, there is no need to keep statistics about previous performances, at the cost of not being able to use informed search. However, the performance is still affected by the preselection of possible values by the designer. Because of this, a random search may still yield decent results [22]. Even so, an informed search looks more promising.

2.2.1 MAB Allocation

One possibility to solve the CMAB problem is to translate to problem back to another MAB problem. Each global arm of the CMAB can be a MAB arm itself, where the new arm corresponds to a combination of parameters. We can then use selection policy π_{mab} to select which parameter combination to try. An observation one could make is that the performance of a value of a single parameter is indicative of how that value might perform when combined with other parameters. With MAB allocation, this information is ignored and therefore lost. The combinatorial structure of the parameter sets cannot be exploited. So, even though this allocation might be interesting to consider, an alternative can be found which does take the combinatorial structure in account.

2.2.2 Naïve Monte Carlo

Naïve Monte Carlo (NCM) is an allocation strategy that considers the combinatorial structure of the parameter sets. It was shown to outperform other algorithms on the GGP [23]. NMC is based on the notion that the expected reward μ of a certain configuration of parameter values can be approximated by a linear combination of expected rewards of single parameter values $\mu_1, ...\mu_n$:

$$\mu(P) \approx \sum_{i=1}^{n} \mu_i(P_i) \tag{2.1}$$

This is also known as the naïve assumption. This approaches the problem as a combination of $n+1$ MAB problems, n local MABs and one global MAB. As explained earlier, each local MAB, MAB_i, corresponds to one parameter, while the global MAB, MAB_g, corresponds to the combination of parameters. The local MABs contain all feasible values as set by the designer. Since the global MAB holds the collected reward statistics of the played combinations, it is empty at the start of the game. After subsequent iterations it will be filled with the obtained rewards, which can then be utilized in subsequent iterations.

Algorithm 1 Online parameter tuning algorithm

```
1: $MAB_g \leftarrow$ create a MAB with no arms
2: for $i \leftarrow 1 : n$ do
3:     $MAB_i \leftarrow$ create a MAB for $P_i$ with an arm for each of its possible values
4: while time not elapsed do
5:     P $\leftarrow$ ChooseParamValues($S$, $\epsilon_0$, $\pi_l$, $\pi_g$, $MAB_g$, $MAB_1$, ..., $MAB_n$)
6:     $A.set(P)$
7:     $r \leftarrow G.simulate(A)$
8:     UpdateValuesStats($P, r, MAB_g, MAB_1, ..., MAB_n$)
```

Algorithm 2 Choosing the parameter values to initialize the agent with

```
1: function ChooseParamValues($S, \epsilon_0, \pi_l, \pi_g, MAB_g, MAB_1, ..., MAB_n$)
2:     if $RAND(0,1) < \epsilon_0$ then
3:         for $i \leftarrow 1 : n$ do
4:             P[$i$] $\leftarrow \pi_l$.ChooseRandomValue($S, MAB_i$)        ▷ Exploration
5:         $MAB_g$.Add(P)
6:     else
7:         P $\leftarrow \pi_g$.ChooseCombinations($S, MAB_g$)        ▷ Exploitation
8:     return P
```

2.3 Self-adaptive MCTS

Algorithm 1 shows how NMC is combined with MCTS. Initially, the global MAB is empty, as no combinations have been tried yet, and a local MAB is created for each parameter. During the time of a game frame, the set parameter values to be played with P are chosen and the agent is initialized with it. The reward that the agent A received from playing game G with the current parameter set is collected from the MCTS iteration after the simulation step has finished. Finally, the MABs statistics are updated, after which the following iteration can take place.

Each MCTS iteration NMC has two choices on how to choose the next parameter value set, namely exploring the search space or exploiting the currently known parameter sets. The pseudocode for this process is shown in Algorithm 2. The choice is guided by search policy π_0, with a frequently used strategy being epsilon-greedy search. With epsilon-greedy search, the parameter ϵ_0 controls how often the search space is explored by thresholding a random real number between 0 and 1. If ϵ_0 is greater than the random value, as indicated in line 2, the parameter values are picked by generating

Algorithm 3 How to update the MABs with the obtained rewards

```
1: function UpdateValuesStats($P, r, MAB_g, MAB_1, ..., MAB_n$)
2:     $MAB_g$.UpdateArmStats(P, $r$)
3:     for $i \leftarrow 1 : n$ do
4:         $MAB_i$.UpdateArmStats(P[$i$], $r$)
```

new parameter values through exploration. This exploration is done by applying the policy π_l in `ChooseRandomValue`, as shown in line 4. Similarly, if ϵ_0 is lower than the random value, the parameter values are picked by evaluating the combinations which were tested so far through exploitation. The exploitation is done by applying policy π_g in `ChooseCombinations`, as shown in line 7.

Algorithm 3 shows how the statistics are updated after an iteration of MCTS. The number of pulls on the specific parameter set arm in the global MAB is incremented, as shown in Equation 2.2, and the cumulative reward summed, as shown in Equation 2.3.

$$MAB_g.visits \leftarrow MAB_g.visits + 1 \tag{2.2}$$

$$MAB_g.R \leftarrow MAB_g.R + MAB_g.r \tag{2.3}$$

Then, each parameter's corresponding local MAB updates its statistic with the same statistics as used by the global MAB.

This chapter showed how MCTS can be enhanced with online parameter tuning. In the next chapter macro-actions will be explained.

Chapter 3: Macro-actions

A problem with searching a game state tree for the best action to play is that the number of states increases drastically as the search algorithm searches, which is the case when an agent plans further ahead. Depending on the amount of eligible actions to play, this means that in certain games MCTS is able to only build the tree to depth 5, due to the strict time budget. This might be acceptable for games with many intermediate rewards, where short-time planning is sufficient. However, it will not be sufficient once a game features delayed rewards, where rewards are too far into the future to detect now. An example of such a game is Camel Race, which is depicted in Figure 3.1. In this game, the player controls a camel and races a number of other camels to the finish line. For a human, this might be one of the easiest games to beat, as you can simply keep the right arrow key pressed. For an agent building a game state tree using vanilla MCTS this is virtually impossible to achieve in real-time. One possible approach for such situations is to make use of macro-actions.

Macro-actions offer a way to combine multiple actions into one. This allows the agent to search much deeper in the game state tree, as each node might now represent several time steps ahead. Within the same amount of time, an agent now can plan further into the future, possibly detect rewards it would otherwise have missed. Macro-actions have proven particularly successful in the physical traveling salesman problem [24]. This problem is an adaptation of the regular traveling salesman problem, with the addition of physics, altering the game to let the agents move over a continuous game field instead of a discrete grid. Since the addition of physics requires agents to plan over many more but smaller grid cells, macro-actions offer the agent a way to approach the problem.

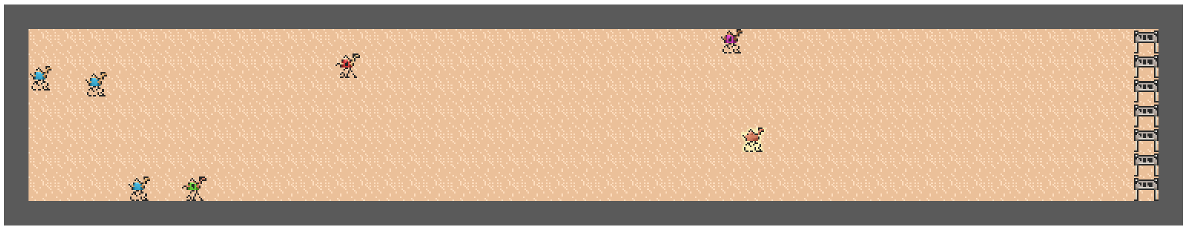


Figure 3.1: The game of Camel Race. The first agent to cross the finish line on the right end of the screen wins. The playable actions consist of movement in one of the four orthogonal directions. The amount of steps to take to the right is equal to 45, with no intermediate rewards in between. When looking at game score, the first reward is therefore detected at search depth 45, which is too deep to find using normal MCTS in real-time.

A macro-action can be a combination of any sequence of actions, for any amount of time. A common approach is to use repeating actions. In this case, a macro-action $M = \langle a_1, ..., a_t \rangle$ can be defined as the repetition of action a a fixed number of t time steps. The bigger t gets, the greater number of time steps can be represented at the same depth of the game state tree. While increasing t leads to a deeper search, it comes at the cost of precision. This trade-off can be acceptable in certain continuous domains [25].

Macro-actions have been tested on the GVGAI framework in combination with other enhancements as well. It was shown to be very successful in games from the GVGAI framework with real-world physics [26]. These games were, however, constructed specifically to incorporate physics so as to accommodate the macro-actions. Most games in the game suite lack physics and the authors were therefore also interested to see how well MCTS with macro-actions would perform on the regular non-physics games. The results showed that adding macro-actions made the agent perform worse on most games and worse overall when macro-actions were introduced [27]. The authors argued that the repeat value was a problematic factor, as the performance on each game was highly dependant on this value. Whereas a high repeat value was required for games like Camel Race to play successfully, a low repeat value is appropriate for games where precision is needed. In maze-like games, for example, it was likely to search too coarsely and move past maze junctions, meaning that the agent was not able to move around properly.

The problem with this kind of macro-action design is that it is very rigid, it enforces a granularity of looking at the game grid in a top-down manner. Therefore, it would be interesting to use a version of macro-actions which reasons using the available game information. By defining subtasks and sub-goals, human game playing can be mimicked, allowing the agent to reason in a more meaningful way. Breaking up the goals in smaller parts can be achieved using MCTS with options.

3.1 Options

Options originated as a method to incorporate temporal abstraction [28]. An option is a method of completing certain sub-goals, and is defined by the tuple $\langle I, \pi, \beta \rangle$. The initiation set $I \subseteq S$ is a set of states in which the option can be started. The policy $\pi : S \times A \rightarrow [0,1]$ defines the action that should be taken for each state. The termination condition $\beta : S^{+} \rightarrow [0,1]$ is used to decide if a state applies to let the option finish. At the start of a time step, the agent can choose from any option that has the current state in its initiation set. The policy will be applied for one or more time steps until the termination condition is met.

The first to apply options to GVGAI were De Waard, Roijers, and Bakkes where they combined options with MCTS [17]. The goal was to come up with some general tasks and sub-goals that were game independent. They showed that this approach of combining options with MCTS outperformed combining options with a Markov Decision Process. This research will use a set of options that is inspired by the set they used, which is as follows:

`DoAction` lets the agent play a certain action once.

- Initialization: This option is created for a certain action a.
- Initiation set: Any state s_t.
- Termination set: Any state s_{t+1}.
- Policy set: Play action a.

`AvoidNearestNPC` lets the agent move away from the NPC closest to it.

- Initialization: This option is initialized with the nearest NPC.
- Initiation set: Any state s_t which contains at least one NPC.
- Termination set: Any state s_{t+1}.
- Policy set: Apply the action which results in the agent moving in the direction away from the NPC.

`GoToMovableSprite` lets the agent move towards a sprite which can move.

- Initialization: This option is created for a certain movable sprite. A sprite can be any observation that moves excluding NPCs.
- Initiation set: Any state s_t with the movable sprite visible on the observation grid.
- Termination set: Any state s_t in which the agent's position equals the movable sprite's position, or the movable sprite is no longer present on the grid.
- Policy set: Apply the action which moves the agent in the direction towards the movable sprite.

`GoToNearestSpriteOfType` lets the agent move towards the nearest sprite of a certain type.

- Initialization: This option is created for the nearest sprite of each sprite type.
- Initiation set: Any state s_t where the sprite is visible on the grid.
- Termination set: Any state s_{t+k}, where the agent's position equals the sprite's position, or the sprite is no longer visible.
- Policy set: Apply the action which moves the agent in the direction towards the sprite.

`GoToPosition` lets the agent move to a certain position.

- Initialization: This option is created for interesting positions, which can be either portals or resources.
- Initiation set: Any state s_t.
- Termination set: Any state s_{t+k}, where the agent's position equals the object's position.
- Policy set: Apply the action which moves the agent in the direction towards the object.

`UseAndWait` lets the agent perform the `use` action and wait for what impact it has.

- Initialization: This option has no initialization options.
- Initiation set: Any state s_t in games in which the agent has the `use` action available.
- Termination set: Any state s_{t+k} until the maximum rollout depth is reached.

  – Policy set: Use the `use` action, then nothing.

Each option can be initialized in a few different ways. The `DoAction` option, for example, creates an option for each action that the agent can perform. The option is completed after one time step, after the agent has performed the action. This has the same effect as the regular way of approaching MCTS. The other options allow the agent to exhibit more complex behavior. `AvoidNearestNPC` lets the agent stay away from the nearest NPC if there is one, which could be useful when there are characters in the game which seek to harm the agent.

The set of `GoTo...` options lets the agent go to a specific location, the one the option is initialized with. The GVGAI framework provides high-level information about all objects in the observable grid. Objects like portals and resources are of interest and could provide the agent with some reward. For this kind of navigation, a path-finding algorithm is required. For this research we used A* [29]. At the start of the game, all routes are pre-computed and can be consulted during game play.

3.2 Option MCTS

Options could be represented as nodes of the game tree. However, this could be problematic. If a node in the search tree represents multiple actions, it would become hard to compare two nodes at depth k, as we would no longer know how many steps into the future they occur. For this reason, De Waard, Roijers, and Bakkes suggest to let the nodes of the tree keep representing actions and let options span multiple nodes. This spanning of options over one or more nodes is depicted in Figure 3.2.

The way options are combined with MCTS is detailed in Algorithm 4. An iteration is started by initializing an empty set of followed options $\mathbf{o}_s$. This set will fill as more options are explored. The inner **while** loop, from line `4` to line `17`, describes how the tree policy is applied. Traversing of the tree is continued as long as `continue` is satisfied, which is not the case if the game has ended in state s, the rollout d has been reached, or an option o was set and is now terminated by β. If the termination requirement is met, then the option set $\mathbf{p}_s$ is set to all the available options, the ones which have this state in their initiation set, else, if an option was previously set, search is continued with that one. If all the options $\mathbf{p}_s$ have been expanded ($\mathbf{p}_s = \mathbf{m}$), the tree traversal is continued using UCT.

In case that not all options have been expanded, the subset of unexplored options is made by looking at the difference between $\mathbf{p}_s$ and $\mathbf{m}$ and a random option $\mathbf{w}$ is taken from that set. This option indicates, from its policy, which action to take, given the current state s. The state s is subsequently expanded into state s' using action a. This

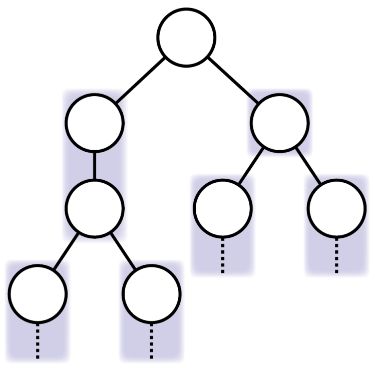


Figure 3.2: This image depicts how the game state tree is built by incorporating options into MCTS. The nodes denote states, the edges denote actions and the colored rectangles spanning one or more states denote options. To preserve the property of having a node at depth k represent a state at time step $t+k$, the options are applied as an overlay.

step corresponds to the expansion step of MCTS. The current option is extended to the new state, which can then continue the current option.

When a new state has been expanded and the new child node created, `rollout` applies the default policy and takes random actions until the maximum rollout depth is reached. The reward is propagated back up into the tree, updating the nodes' statistics, up until to root node. When the time has expired the option in the root node with the highest `value` is selected.

This chapter has detailed how macro-options can be used to enhance MCTS by allowing the agent to search more meaningful parts of the search tree. The next chapter will look at multi-objective MCTS and how it can be used to optimize for multiple goals.

Algorithm 4 One iteration of option MCTS

```
1: $\mathbf{o}_s \leftarrow \emptyset$                                  ▷ $o_s$ will hold the options followed from $s$
2: **while** time not elapsed **do**
3:     $s \leftarrow r$                                                  ▷ start from root node
4:     **while** continue_treepolicy($s, d$) **do**
5:         **if** $s \in \beta(o_s)$ **then**                            ▷ if option stops in state $s$
6:             $\mathbf{p}_s \leftarrow \cup_o(s \in I_o)$               ▷ set $\mathbf{p}_s$ to the available options
7:         **else**
8:             $\mathbf{p}_s \leftarrow \{o_s\}$                         ▷ no new option can be selected
9:         $\mathbf{m} \leftarrow \cup_o(o_{s \in \mathbf{c}_s})$        ▷ set $\mathbf{m}$ to the expanded options
10:        **if** $\mathbf{p}_s = \mathbf{m}$ **then**
11:            $s' \leftarrow \max_{c \in \mathbf{c}_s}$ uct$(s, c)$     ▷ select child node using UCT
12:        **else**
13:            $w \leftarrow$ random_element($\mathbf{p}_s$ - m)
14:            $a \leftarrow$ get_action(w, s)
15:            $s' \leftarrow$ expand(s, a)
16:            $o_{s'} \leftarrow w$
17:        $s \leftarrow s'$                                             ▷ continue loop with new node $s'$
18:    $\delta \leftarrow$ rollout($s'$)
19:    backpropagate($s'$,$\delta$)
20: **return** get_action(max$_{o \in c_r}$ value($o$), $r$)
```

Chapter 4: Using Multiple Objectives

In regular MCTS, there is one evaluation function with which states are scored. These scores are then used during the search stage to determine which parts of the game state tree to explore first and, subsequently, which action to play. In the GVGAI competition, most agents use the actual game score as provided by the framework. The agent gets a high reward for winning the game, a low reward for losing and sometimes a small reward for picking up resources or defeating enemies, although this is game dependant. There are games where the game is won some time in the future with no intermediate rewards, as was the case with Camel Race as detailed in chapter 3. Another way to deal with this type of problem is to use an algorithm which is able to pursue multiple goals, instead of optimizing for one evaluation function. This way, the agent can be encouraged to seek in different parts of the search tree to find a good action to play. It has not only produced good results on the specifically designed Deep Sea Treasure and Multi-objective Physical Travelling Salesman problems [18], but also with the more general GVGAI framework, where it has been shown to outperform single-objective algorithms [30].

4.1 Multi-objective Optimization Problems

A Multi-objective Optimization Problem (MOP) is a problem where two or more evaluation functions are to be optimized simultaneously [31]. It is defined as the maximization of a function $\vec{f}(\vec{x})$, where $\vec{x} = (x_1, ..., x_n)$ is an element from the decision space. $\vec{f}(\vec{x}) = (f_1(\vec{x}), ..., f_n(\vec{x}))$ belongs to the objective space and consists of m objective functions. Each solution $\vec{x}$ thus has m different rewards to be optimized.

A solution $\vec{x}$ is said to *dominate* another solution $\vec{y}$ iff:

1. $f_i(\vec{x})$ is not worse than $f_i(\vec{y}), \forall i = 1, ..., m$
2. For at least one objective $j : f_j(\vec{x})$ is better than its analogous counterpart in $f_j(\vec{y})$

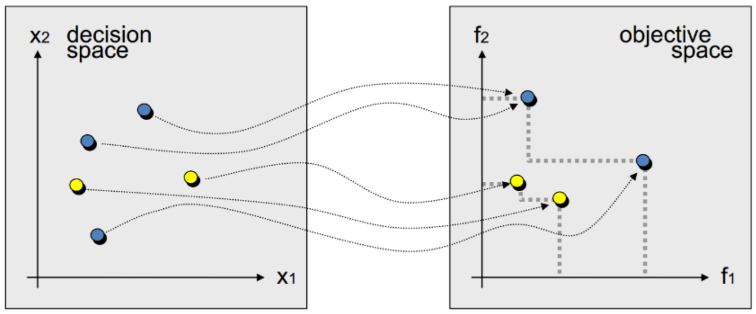


Figure 4.1: Decision and objective spaces with variables x_1 and x_2 and objectives f_1 and f_2. Blue dots form a non-dominated set, while the yellow dots are the non-optimal solutions. From: [18]

If both statements are the case, then it can be said that $\vec{x}$ dominates $\vec{y}$, which can be written as $\vec{x} \succeq \vec{y}$, and $\vec{x}$ is non-dominated by $\vec{y}$. This *dominance* condition gives a measure to compare two solutions from the objective space, to see which is better.

There are cases where neither $\vec{x} \succeq \vec{y}$ nor $\vec{y} \succeq \vec{x}$. These solutions are then non-dominant with respect to each other. The set of non-dominated solutions can be grouped in a so-called *non-dominated set*. If there is no solution that dominates any other solution in a non-dominated set P, it is said to be a *Pareto-set*. The objective vectors of the members of P form a *Pareto-front*. Figure 4.1 depicts the relation between the decision space and objective space, and domination of solution sets.

The quality of a Pareto-front can be measured by using the Hypervolume Indicator (HV) [32]. It is defined as the volume dominated by P in the objective space as measured from the origin, using the Lebesgue measure. The higher the $HV(P)$, the better the front is. An example of $HV(P)$ is depicted in Figure 4.2.

We will use the weighted-sum approach to approach the MOP in the recommendation policy. Using this approach, each objective is given its own weight by the agent designer for how they should be balanced. The algorithm can converge to different outcomes, based on how the weights of the objectives were set. These objective weights are a way to combine the multiple objective function into one, which will then be optimized. In this research, all objective have equal weights.

4.2 Multi-objective MCTS

In order to adapt MCTS to include multi-objective optimization, it is required to handle multiple rewards after a game state evaluation. Therefore, the reward vector $R =$

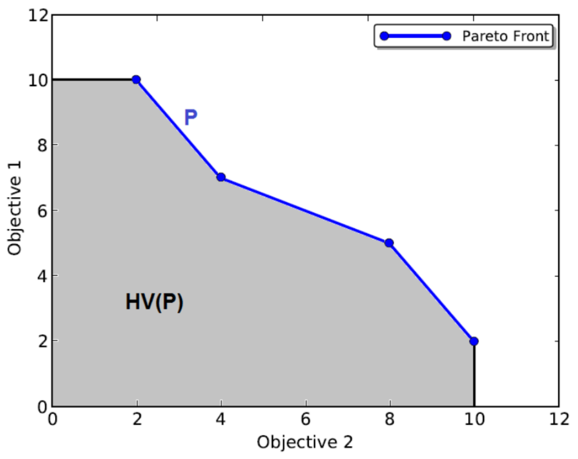


Figure 4.2: The Hypervolume Indicator of a Pareto front is calculated as the area encapsulated by the Pareto front. From: [18]

$r_1,...,r_m$ replaces the reward value r, where m is the number of objectives. After an MCTS iteration, the reward vector R is backpropagated through the nodes of the tree to update an accumulated reward vector $\overline{R}$. For the UCT formula, as explained in Equation 1.1, $Q(s,a)$ is changed to a vector which keeps the empirical average of m rewards. Since $Q(s,a)$ now is a vector instead of a single value, it now has to be used in a different way. This can be done by using the weighted-sum approach, similarly to the one used in the recommendation policy. For the tree policy, however, $Q(s,a)$ is redefined as $Q(s,a) = HV(P)/N(s)$ [33]. The updated UCT formula then becomes:

$$a^* = \operatorname*{argmax}_{a \in A(s)} \left\{ HV(P)/N(s) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \tag{4.1}$$

In MO-MCTS, the nodes will not only hold the accumulated rewards vector, but will also keep a local Pareto front P. This local Pareto front is updated with each reward vector during backpropagation. The way the node statistics are updated is depicted in Algorithm 5. If the reward vector $\overline{r}$ is not dominated by the local front P, it is added to it. In the case that $\overline{r}$ dominates some or all solutions of P, P is cleaned up by removing the now dominated solutions from the set. If P dominates $\overline{r}$, it is not added to the set and it no longer gets propagated up the tree.

Three things can be observed about using local Pareto fronts. Firstly, by keeping a local front, each node has an estimate of the quality of solutions reachable from it. Secondly, if a reward vector is dominated by the local front, it will be dominated by the

Algorithm 5 Updating the Pareto sets in nodes for MO-MCTS.

```
1: function UPDATE($node, \overline{r}, dominated \leftarrow false$)
2:     $node.Visits \leftarrow node.Visits + 1$
3:     $node.\overline{R} \leftarrow node.\overline{R} + \overline{r}$
4:     if $\neg dominated$ then
5:         if $node.P \succeq \overline{r}$ then
6:             $dominated \leftarrow true$
7:         else
8:             $node.P \leftarrow node.P \cup \overline{r}$
9:             Cleanup($node.P$)                ▷ Remove dominated solutions
10:    Update($node.parent, \overline{r}, dominated$)
```

parent nodes. Thirdly, as a consequence of the second point, the Pareto front of a node cannot be worse than the fronts of its children, meaning that the root node will end up with the best non-dominated front which was found during search. The Pareto front of the root node can then be used to look at the quality of the search by examining $HV(P)$.

4.3 Heuristic

Different heuristics can be constructed to act as objective function. The first objective O_1 that will be used in this research is the game score. This objective takes the current game score, which may be increased by shooting enemies or picking up resources, for example. On top of that, a high value is added when the game is won or lost.

The second objective O_2 that is used in this research encourages the agent to explore the environment. The idea behind this objective is that in the case that the states found during search all have the same game score, the agent can at least explore the physical space in the hope of encountering more interesting regions of the level. This technique is inspired by pheromone trails, which work similar to a potential field [34]. The agent secretes a pheromone that repels. It is secreted every time step and diffuses into nearby cells in the orthogonal directions. Each cell can have a pheromone value between 0 and 1, or: $p_{i,j} \in [0,1]$ where i and j are cell coordinates on the game grid. The pheromone level of a grid cell decays over time. The amount of pheromone level per cell at location (i,j) is given by:

$$p_{i,j} = \rho_{df} \times \rho_{\phi} + (1 - \rho_{df}) \times \rho_{dc} \times p_{i,j} \tag{4.2}$$

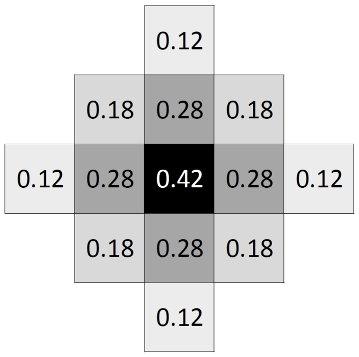


Figure 4.3: An example of pheromone diffusion. The agent is located in the center, secreting pheromones. The pheromones are diffused to the neighbouring cells. Over time the pheromones decay.

ρ_ϕ is the average pheromone level of the neighbouring cells. The pheromone diffusion rate is given by ρ_{df} and the pheromone decay rate by ρ_{dc}. Both are calculated at each time step. A visual representation of the pheromone diffusion is depicted in Figure 4.3. With the pheromone levels calculated per cell, the second objective function is now given by $O_2 = 1 - p_{i,j}$, which means that cells with a lower pheromone receive a higher reward and are to be prioritized. This way, the agent is rewarded for exploring the game level. Using equation 4.2, the pheromone levels could exceed 1, which would result in a negative objective value. To prevent this, the pheromone levels are clipped at 1.

To conclude, we defined two objective functions for the heuristic. Their combination rewards actions that lead to states with high game scores and, at the same time, allow for greater exploration of the level that is currently played. This chapter has explained how multi-objective MCTS works and how it lets the agent try to maximize multiple objectives simultaneously. The next chapter will detail how the different MCTS enhancements can be combined into one algorithm.

Chapter 5: SAMOO-MCTS

Self-adaptive MCTS, option MCTS and multi-objective MCTS are explained in chapter 2, chapter 3 and chapter 4 respectively. This chapter introduces the combination of these algorithms: Self-adaptive Multi-objective Option Monte Carlo Tree Search (SAMOO-MCTS). As the performance of alternative algorithms varies between games, it may be beneficial to make use of a combination of multiple algorithms, allowing the agent to exhibit different behaviours in response to the varying game situations [35]. The SAMOO-MCTS agent is designed in such a way that it can play using any algorithm or combination thereof. This way, the game playing performance of each added algorithm can be tested. The vanilla MCTS agent, which is provided with the GVGAI framework, serves as a basis for the comparison. An iteration of the SAMOO-MCTS algorithm is depicted in Figure 5.1 and can be divided into two following steps:

1. With each iteration the parameters are set first. In this research, the values that are varied are UCT's exploration/exploitation parameter C and the maximum rollout depth RD. Without parameter tuning these remain the same and are set to $C = \sqrt{2}$, $RD = 10$ for this research. On top of that, the agent is able to tune its UCT balancing parameter C and the rollout depth limit RD. When parameter tuning is enabled, a combination is selected using NMC. The available values that can be chosen are set to $C = (0.8, 1.0, 1.4, 2., 2.4)$ and $RD = (1, 5, 10, 20, 50, 70)$. Depending on policy ϵ_0, a new parameter set is generated for exploration, or a previous parameter set is reused to be exploited. In the SAMOO-MCTS agent, ϵ_0 is set to 0.75. In the exploration phase of the parameter selection, the new parameter values are selected with π_g and π_l using UCT as well, using $C_g = 0.7$ and $C_l = 0.7$.

2. The tree traversal in the selection step is guided by UCT or options. In case of options, when all of them are exhausted, UCT is used as a fallback. Nodes are valued by their average reward, or $HV(P)$ in case of multi-objective optimization. The exploration objective is scored by using pheromone trails, with $\rho_{df} = 0.4$, $\rho_{dc} = 0.99$ and the amount of pheromones excreted each game tick is 0.3.

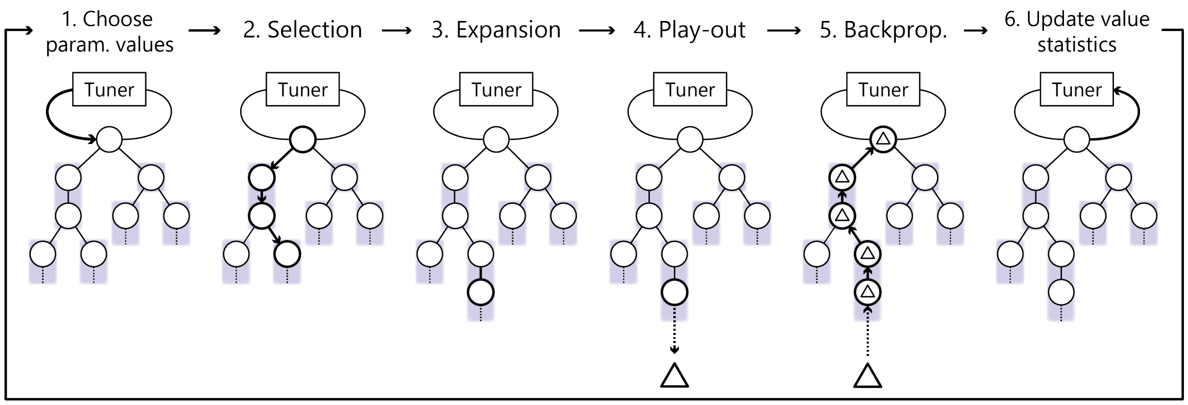


Figure 5.1: An iteration of Monte-Carlo Tree Search. The selection and expansion steps use a tree policy, while the simulation step uses a default policy. The final step is backpropagation. Additionally, the parameters for guiding the tree traversal are set by the tuner before the selection step. Similarly, after the iteration has finished, the rewards are used to update the tuner statistics, which can then be used in the next iteration.

3. Once the node to be expanded is selected in the selection step, an action is selected to be played and to expand the search tree with. Normally, this is done by an action that is selected at random from the set of actions that have not been played before. In case of options, however, the next action is selected by the policy π. The new state is determined by consulting the forward model with the current state and the action to play.

4. The game is played out in the simulation step using the default policy. This is the same in all configurations, namely taking random actions until a terminal state is reached. Since SAMOO-MCTS needs to operate in real time, the maximum rollout depth RD is used as extra measure to determine how deep the search tree can be. c

5. For single objective MCTS one evaluation function is used: the game score. For multi-objective MCTS, two evaluation functions are used: the game score and exploration value. The reward(s) gained from the evaluation function(s) are propagated back up the tree.

6. If parameter tuning was enabled, the reward from the current iteration is used to update the statistics about the local and global MABs. When there were multiple objectives, the weighted-sum approach is used to reduce the reward vector to one

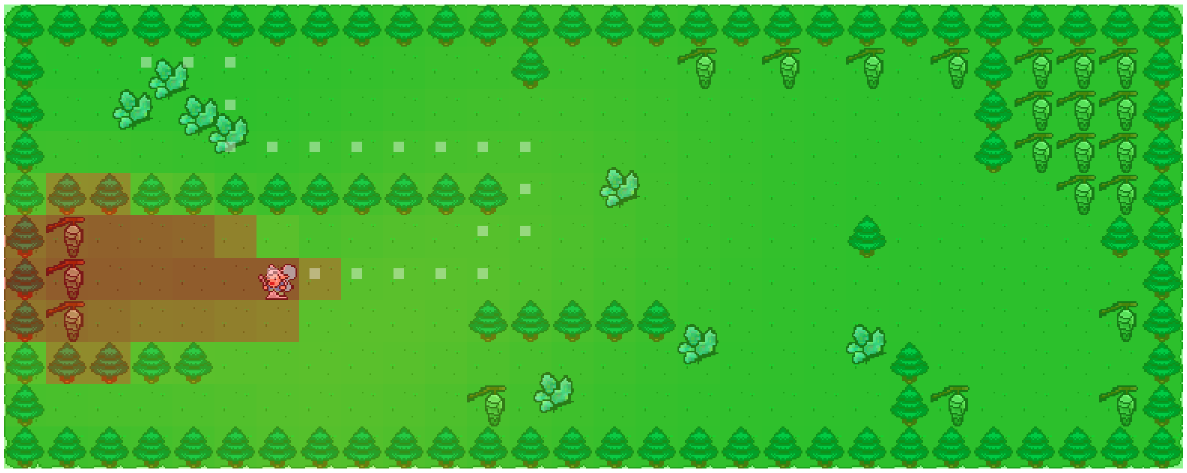


Figure 5.2: The game of Butterflies as being played by the SAMOO agent. The white dots indicate the planned route, as indicated by the `GoToNearestSpriteOfType` option, which is set to the butterfly. Green cells have no agent pheromone levels, while brown cells do have pheromones indicating that the agent was there recently.

reward. Equal weights are used for each objective.

Steps 2, 3, 4 and 5 correspond to regular MCTS, where steps 1 and 6 are added in case parameter tuning is enabled. After the iteration is finished an action has to be selected using the recommendation policy, which will be the action with the highest average reward. In the case of multi-objective MCTS, there will be a vector of rewards stored in the nodes. In that case, the weighted-sum approach is used to reduce it to a single value. In this research, each objective has the same weight and therefore contributes equally to the combined score. No information is kept between game steps, except for the parameter setting statistics, so the next iteration will begin with an empty search tree.

Since GVGP requires the agent to make decisions in real time, the agent doesn't get the time to build the complete search tree. MCTS enables it to only explore the most promising actions and return the best found action once it has reached its time limit, making it an *anytime* algorithm. The algorithm does not contain any game-specific heuristics, but only uses generic strategies, which is important as it is unknown beforehand which games it will play. An example of how the resulting SAMOO-MCTS would play a game is shown in Figure 5.2.

This chapter has described how SAMOO-MCTS combines multiple MCTS enhancements into one algorithm. The next chapter will describe the experiments and obtained results.

Chapter 6: Experiments

This chapter describes the experiments done with the SAMOO-MCTS agent. Since we are interested in the added value of each algorithm, each combination of the different algorithms has been tested. In its most basic form, without any additions, the agent will play the games with vanilla MCTS. In its most advanced form, the agent will use MCTS, extended with parameter tuning, multiple objectives and options. This means that eight different agents will be compared on their performance, namely:

A	Agent playing with MCTS without any additions.
A_s	Agent playing with MCTS with parameter tuning.
A_m	Agent playing with MCTS with multiple objectives.
A_o	Agent playing with MCTS with options.
A_{sm}	Agent playing with MCTS with both parameter tuning and multiple objectives.
A_{so}	Agent playing with MCTS with both parameter tuning and options.
A_{mo}	Agent playing with MCTS with both multiple objectives and options.
A_{smo}	Agent playing with MCTS with all three additions.

Each agent will be tested on the same set of 20 games, each consisting of 5 levels [1]. The agents play every level 100 times, totalling 500 playthroughs per game and 10,000 plays in total. During game play, the agents have 40ms to pick an action to play per game tick. The game ends if, after 2,000 game ticks, no terminal state has been reached and will be counted as a loss. These rules are the same as those used in the GVGAI competition. Firstly, this chapter will describe how the game set is constructed and a description about each game is given. Secondly, the performance of each agent is

[1] In this context, the word ‘levels’ is used to describe variations of the game, which increase in difficulty, instead of consecutive levels within the game.

assessed and compared. The performance is primarily determined by win percentages and secondarily by the obtained scores. Lastly, the visited parameter sets are examined to obtain insight in what settings are chosen.

6.1 Games Test Set

The game set that was used in the experiments consists of 20 games and was constructed by Gaina, Liu, Lucas, *et al.* [36]. The goal of the set was to create a collection of games that are diverse, resulting in a collection of 10 stochastic games and 10 deterministic games. Some of the deterministic games were later relabeled, as they were wrongly classified as deterministic [16]. The games from the test set are listed below, and are summarized alphabetically, which is the same order in which they will be presented in the results. Table 6.1 details how the games can be classified.

Aliens Similar to the existing game of *Space Invaders*. Aliens appear at the top, shooting missiles at the agent and moving towards the bottom in a left-to-right sweeping motion. The agent can move left and right and has to shoot aliens before any of them reaches the bottom. The agent must also stay alive by not getting killed by a missile.

Bait In this game, the agent has to get to the exit, which is blocked by a locked door. The agent has to collect the key first before it can get to the exit. In the maze, there are holes blocking the agent's path, which can be cleared by pushing blocks into them. Extra points can be collected by picking up mushrooms. The game is won when the agent reaches the exit.

Butterflies The agent has to catch all the butterflies in this game by walking up to them. The agent loses the game if not all butterflies have been caught within the time limit. Additional butterflies can spawn over time from cocoons. This means that the agent can further increase its score by waiting until all butterflies have appeared from their cocoons, instead of catching all currently available butterflies, which prematurely ends the game.

Camel Race In Camel Race, the agent controls a camel which has to race to the finish line against other camels. The camel to first cross the finish line wins the level. The other camels move in a straight line towards the finish line, but all in varying speeds.

Chase In *Chase* the agent has to chase and kill goats that flee from the agent. The agent kills a goat by walking into it. Whenever a goat walks into a corpse, it gets enraged. An enraged goat starts chasing the agent, killing it on collision. Once a goat gets enraged, it will not turn back to normal. The agent wins the game by killing all goats without getting killed by an enraged goat.

Chopper The agent controls a chopper in this game and flies through the air. It has to defend satellites in the sky from tanks shooting missiles at them from the ground. The agent has to collect ammunition to eliminate the tanks. The game is lost if all satellites or the agent are shot and the game is won when all tanks are eliminated.

Crossfire The agent has to reach the exit in this game. Along the maze there are cannons that fire missiles at the agent. The agent has to dodge the missiles and get to the exit alive to win the game. The game is lost if the agent gets hit by a missile.

Dig Dug In this game, the agent is located in a cave. The agent has to collect coins by digging through the cave. There are scorpions in the cave that can kill the agent. The agent can kill the scorpions by shooting rocks at them, which can be shot after applying the `use` action twice.

Escape This game is a puzzle game in which the agent controls a rat. It has to get to the cheese but cannot reach it right away. There are multiple holes in the field, which can be filled by pushing rocks into them, filling the gaps and clearing the path. The game is won when the rat grabs the cheese.

Hungry Birds In this game, the agent controls a bird. The goal is to get to the exit. The problem is that this exit is too far away to get there in one go, since the bird does not have enough food. Each game tick the food depletes, which can be restored by picking up food that is scattered across the game map. The game is lost if the bird runs out of food.

Infection The goal is to infect all animals. The agent is infected by a virus and gives it to the animals by colliding into them. Animals also transfer the virus by colliding into each other. Both the agent and the animals can be cured by medicine laying around, but this medicine can be destroyed by the agent by using its sword.

Intersection In *Intersection*, the agent is located at a crossroad on which many vehicles drive past. Collectables appear from time to time on the field, which the agent can collect in order to earn points. To get to the collectables, the agent has to cross the roads. The game is lost whenever the agent is hit by a vehicle too many times. The game is won by surviving 2,000 game ticks. Note that this is an exception on the rule that each game terminates after 2,000 in a lost state. In theory, this game can therefore be won by simply doing nothing until the time runs out. This will, however, be reflected in the final score, which will still be low.

Lemmings In *Lemmings*, the agent has to help a group of lemmings get to the exit. There are objects blocking the way that the agent can clear. There are also traps that harm the lemmings and the agent. The game is lost if this happens. Therefore, the agent needs to choose carefully which objects to remove so that the lemmings can get to the exit without walking into the traps. The game is won when all lemmings reach the exit and lost if any of the lemmings or the agent walks into a trap.

Missile Command There are a couple of cities that the agent needs to protect. Missiles are inbound for the cities and the agent has to intercept the missiles by using its weapon. The game is won if all missiles are intercepted, and lost if there is no city left.

Modality The goal of this puzzle game to push a tree into the hole. There are light and dark tiles, connected by a grey tile. The agent can walk on either the light or the dark tiles, and can switch between the sets by stepping on the grey tile. The game is won when the tree is pushed into the hole.

Plaque Attack This game is similar to *Missile Command*. However, in this scenario, food is heading towards the teeth. The agent can shoot the food, making sure it does not get to the teeth. The game is won when all food has been cleared and the game is lost if there are no teeth left. A difference with *Missile Command* is that the teeth can be repaired by walking into them.

Roguelike In this game, the agent has to escape. In the maze that the agent is located in, there are monsters that can kill the agent and doors that block the route to the exit. Monsters can be slain with a sword after the agent picks it up and additional points can be collected by picking up coins and gems. The doors can be opened by picking up the key. The game is lost if the agent's health has been depleted from monster attacks.

Sea Quest The agent can be killed by submarines that are shooting missiles at it, but it can equally kill the submarines with missiles. The goal of the game is to rescue

Deterministic games: Bait, Camel Race, Escape, Hungry Birds, Modality

Stochastic games	
Negligible randomness	Plaque Attack, Wait for Breakfast
Non-deterministic chasing/fleeing behavior	Chase, Lemmings, Missile Command, Roguelike
Random NPC(s)	Butterflies, Infection, Roguelike
Very stochastic	Aliens, Chopper, Crossfire, Dig Dug, Intersection, Sea Quest, Survive

Table 6.1: The types of the games from the test set. The games from the *Negligible randomness* and *Non-deterministic chasing/fleeing behavior* are close to deterministic, but contain small random elements, making them stochastic.

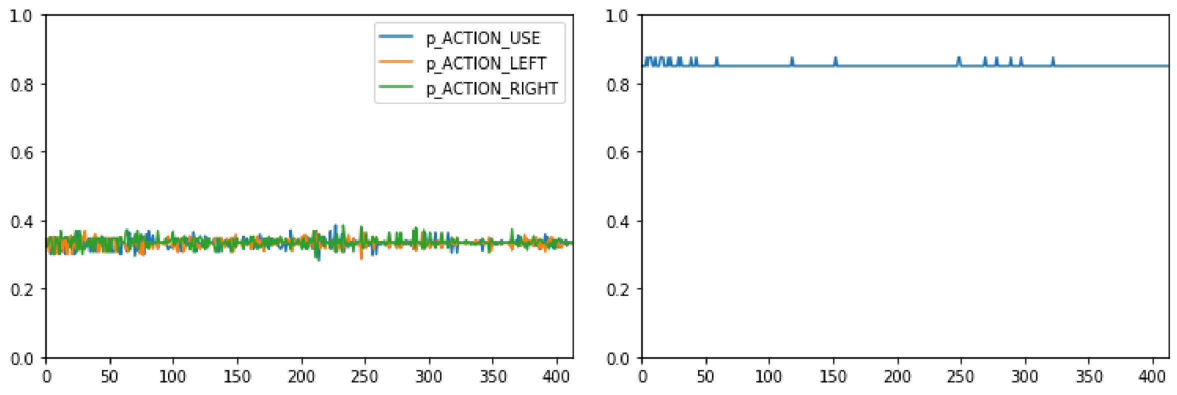
divers by bringing them to the surface of the ocean. The agent has to go up for oxygen occasionally. The game is won by not dying.

Survive Zombies In this game, there is a zombie apocalypse coming for the agent. There are also wizards that can kill the zombies. The agent can also pick up health to restore its health. The game is lost if the agent loses all health, and can be won by staying alive.

Wait for Breakfast The agent is located in a restaurant and has to wait for the waiter to serve the food. Once breakfast is served, the agent has to walk to the table on which breakfast was served to win the game. The game is lost if the agent sits at the wrong table, or the right table before the food is served. Each game tick there is a 5% chance that the server will bring the food.

6.2 Introspection

When comparing win percentages, no insight is gained from how an agent exactly did win or lose the game. For this reason, extra data can be collected during game play [37]. Each game tick the consumed time budget is noted and a probability vector $\overline{p}$, where p_i represents the probability of selecting action a_i. These metrics can also be used in the agent designing process to see if all actions are considered and the agent uses the available time. An example of the used time budget is depicted in Figure 6.1b.



(a) The probabilities of choosing each action at every game tick. The values are mostly roughly the same at 1/3.

(b) The consumed time budget. For each game tick, almost the entire budget is used.

Figure 6.1: An examination of a play-through of A_{so} on the *Aliens* game.

The data is collected during a game play of the game *Aliens* from the agent A_{so}. Over the course of the game, the agent consistently uses most of its available time, which is good, since the SAMOO agent was designed this way. It keeps building the tree until the time is almost up. An example of $\overline{p}$ is depicted in Figure 6.1a and is taken from the same play-through as before. The possible actions produce similar values each game tick, leading to similar chances of being chosen to be played.

6.3 Results

The agents' performances on the game set are discussed in subsection 6.3.1. The most visited parameter combinations are analysed in subsection 6.3.2 for the online tuning.

6.3.1 Agent Performance

The win percentages of the agents is shown in Table 6.2 and Table A.1, while the accompanying game score for each agent is shown in Table 6.3 and Table A.2. On average, every SAMOO-MCTS agent outperforms the vanilla MCTS agent with games won. For the game score, on the other hand, most agents have increased scores, but not all: A_{sm} and A_{smo} actually have lower scores. The agent with the highest win percentage is A_{mo}, winning 55.3% of the games, an increase of 13.7% with respect to the vanilla MCTS. A_{so} and A_m come in close second, scoring 54.5% and 54.2%

Game types	A	A_s	A_m	A_o	A_{sm}	A_{so}	A_{mo}	A_{smo}
Deterministic	15.2 (± 1.27)	21.9 (± 1.48)	45.8 (± 1.73)	22.0 (± 1.69)	47.4 (± 1.66)	54.8 (± 1.20)	48.0 (± 1.80)	**59.4 (± 1.27)**
Stochastic	50.5 (± 0.91)	51.7 (± 0.95)	56.4 (± 1.10)	**58.7 (± 0.95)**	39.4 (± 1.36)	54.7 (± 0.99)	58.3 (± 1.11)	41.9 (± 1.38)
Average	41.6 (± 0.98)	44.4 (± 1.11)	54.2 (± 1.22)	49.5 (± 1.13)	42.0 (± 1.42)	54.5 (± 1.08)	**55.3 (± 1.31)**	46.3 (± 1.36)

Table 6.2: The average percentage of victories achieved for each agent. The standard error is shown in parenthesis. The highest percentage for each game is shown in bold. The full results are shown in Table A.1 and omitted here for readability. The set of deterministic games consists of 5 games, while the stochastic set consists of 15 games.

Game types	A	A_s	A_m	A_o	A_{sm}	A_{so}	A_{mo}	A_{smo}
Deterministic	7.4 (± 0.47)	10.4 (± 0.51)	20.6 (± 0.19)	9.3 (± 0.47)	20.6 (± 0.13)	21.0 (± 0.11)	20.5 (± 0.26)	**21.1 (± 0.09)**
Stochastic	125.2 (± 5.27)	154.5 (± 5.80)	**196.1 (± 6.75)**	175.1 (± 6.94)	58.6 (± 5.24)	173.3 (± 6.97)	175.0 (± 5.78)	77.3 (± 5.36)
Average	95.8 (± 4.07)	118.5 (± 4.48)	**152.2 (± 5.11)**	133.7 (± 5.32)	49.1 (± 3.96)	135.2 (± 5.26)	136.4 (± 4.40)	63.3 (± 4.04)

Table 6.3: The average game scores as obtained by each agent. The standard error is shown in parenthesis. The highest score for each game is shown in bold. The full results are shown in Table A.2 and omitted here for readability.

respectively, followed by A_o, scoring 49.5%. After that follow A_{smo} (46.3%), A_s (44.4%), A_{sm} (42.0%) and A (41.6%).

Using the classifications of the game types as deterministic or stochastic, as made in section 6.1, we can examine how each agent scores on each type. This information is implicitly available in Table A.1 and Table A.2, and is summarized in Table 6.2 for win percentage and Table 6.3 for the scores. Surprisingly, there is a difference in which SAMOO-MCTS combinations perform well on deterministic games, and which perform well on stochastic games. Additionally, agents that perform well on deterministic games might perform poorly on stochastic games. For example, A_{smo} ranks first in deterministic games with 59.4% of games won, while coming in second to last in the stochastic games with 41.9% of games won.

The same holds true vice versa: A_o scores the highest of the agents on stochastic games with a win percentage of 58.7%, but wins just 22.0% of the deterministic games. The latter could be the case due to the nature of the deterministic games in the current set. They are puzzle games (*Bait*, *Escape* and *Modality*), or games with long walk sequences (*Camel Race* and *Hungry Birds*). In these cases, the options don't help solve the puzzle, or cannot find the reward since it is too far away, which lies beyond the maximum rollout depth.

Looking at the results, it can be concluded that each algorithm has its strengths. All agents outperform the vanilla MCTS on the deterministic games. On the stochastic games, however, this is not the case. Here the A_{sm} and A_{smo} agents perform worse than vanilla MCTS.

Other combinations of algorithms seem to improve win percentages. MCTS with options, for example, improved the win percentage for the game *Escape* from 0.0% to 18.4%, thanks to its sub goals. In combination with parameter tuning, it wins all games of *Camel Race*, and is the only agent that achieves this. Without the parameter tuning, the finish line is more than 10 steps away and the options are therefore not able to search far enough. A drawback of the options can be seen in the game *Chopper*, where the agent performed poorly. In this game, all the enemies are below the agent, requiring the agent to turn itself before shooting. Otherwise the agent will miss all its shots. However, there is no option that aligns the agent before applying an action. Therefore, it is very hard for the agent to eliminate the enemies.

The agent with multiple objectives performs really well on games like *Intersection*, *Wait for Breakfast* and *Crossfire*. It is able to play these games so well thanks to the exploration objection function. The goals and rewards cannot be found close to the agent at the start of the game. Instead, the level has to be explored before the goals and rewards can be found. A possible drawback of this is visible in *Survive Zombies*, in which the goal is to stay away from the zombies. Due to the inclination to explore the map, the agent endangers itself by increasing encounter changes with zombies.

6.3.2 Visited Parameter Combinations

The most visited parameter combinations for the UCT exploration/exploitation factor C and maximum rollout depth RD for each agent using parameter tuning are listed in Table 6.4. Interestingly, the parameter setting that is most often used in all agents has $RD = 1$, meaning that the agent prefers instant rewards instead of longer-term rewards. A possible explanation could be agent fright. With higher rollout depths, the chance of finding a terminal state increases. In case a winning state is found, a high reward is awarded, while a losing state results in a high penalty. In games with a low win rate or games with hostile NPCs, the self-tuning agents seem to prefer short-term rewards. Similarly, games with higher win rates correspond to higher rollout depths. For example, in the game *Hungry Birds* where there are only positive rewards in the game field, the agents A_s, A_{sm}, A_{so} and A_{smo} scored 49.6%, 99.0%, 99.6% and 100.0% respectively and all favored a maximum rollout depth of 70. Also, in *Roguelike* the

Games	A_s	A_{sm}	A_{so}	A_{smo}
Aliens	[0.8, 70]	[1.4, 1]	[0.8, 70]	[0.8, 1]
Bait	[0.8, 1]	[1.4, 1]	[2.4, 1]	[0.8, 1]
Butterflies	[1.0, 70]	[0.8, 1]	[1.4, 70]	[1.4, 1]
Camel Race	[1.4, 1]	[2.0, 1]	[1.4, 50]	[1.0, 70]
Chase	[2.4, 70]	[1.0, 50]	[2.0, 1]	[0.8, 1]
Chopper	[2.4, 70]	[2.0, 1]	[1.4, 1]	[0.8, 1]
Crossfire	[2.0, 1]	[1.0, 50]	[2.0, 1]	[2.4, 20]
Dig Dug	[2.0, 1]	[2.0, 1]	[2.0, 1]	[0.8, 1]
Escape	[1.4, 1]	[2.0, 20]	[1.4, 1]	[1.0, 10]
Hungry Birds	[1.0, 70]	[2.0, 70]	[2.4, 70]	[2.4, 70]
Infection	[2.0, 70]	[1.0, 1]	[1.0, 70]	[0.8, 1]
Intersection	[2.4, 1]	[0.8, 1]	[2.4, 70]	[1.0, 1]
Lemmings	[1.0, 1]	[2.4, 10]	[1.0, 1]	[2.4, 20]
Missile Command	[2.4, 70]	[0.8, 1]	[2.0, 70]	[0.8, 1]
Modality	[1.0, 70]	[1.4, 1]	[1.4, 1]	[2.0, 1]
Plaqueattack	[2.0, 1]	[0.8, 1]	[1.0, 1]	[2.0, 1]
Roguelike	[2.4, 1]	[0.8, 1]	[1.4, 1]	[0.8, 1]
Sea Quest	[1.0, 70]	[0.8, 1]	[1.0, 70]	[0.8, 1]
Survive Zombies	[1.0, 1]	[1.4, 1]	[2.4, 1]	[0.8, 1]
Wait for Breakfast	[2.4, 70]	[1.0, 70]	[2.0, 70]	[2.4, 50]

Table 6.4: Most visited parameter combinations per game for each of the variations of the SAMOO MCTS agent in which parameter tuning was active. Parameter combinations are denoted as $[C, RD]$, with C being the UCT exploration/exploitation factor and RD the maximum rollout depth.

agents A_s, A_{sm}, A_{so} and A_{smo} scored 0.0%, 1.0%, 1.2% and 6.2% respectively and all favored a maximum rollout depth of 1.

Chapter 7: Discussion

7.1 Conclusion

It can be concluded that mixing different MCTS enhancements improves the performance with respect to vanilla MCTS for general video game playing. Considering the algorithms on their own, MCTS with multiple objectives wins the most games, due to the emphasis on game level exploration. MCTS with macro-actions in the form of options performs better on games with sub goals. Self-adapting MCTS has the smallest effect on win percentages, but manages to improve the agent's performance when combined with options, allowing for options that stretch farther into the search tree. MCTS with parameter tuning is not successful when combined with multiple objectives, possible due to fright of the pheromone buildup. Overal, multi-objective option MCTS won the most games.

In the case of online parameter tuning, the results of the visited parameter settings suggest that the agent gets scared in games with hostile NPCs. In these games, the agent stops planning ahead, afraid of possible negative outcomes, consequently only seeking immediate rewards.

7.2 Discussion

An interesting observation made in section 6.2 is that the `use` action keeps getting considered throughout game play. In multiple games, the action the agent can perform has a cool-down time, limiting how often the agent can use the action. In *Aliens*, for example, the agent can shoot a missile once per second. After a shot is fired, it is therefore not necessary to explore the sub tree of the `use` action during the next 25 game ticks, allowing it to explore other parts of the game state tree.

A difficulty in designing an effective agent is the question of how to balance between game winning and score maximizing. There are games were the agent can keep increasing its score as long as it doesn't finish the game. An example of this is *Butterflies*, where butterflies keep spawning. It is not clear if the agent should prioritize winning the game or maximizing its score. The same holds true for defeating enemies. In some games this requires the agent to approach the enemy and strike it with a sword. However, getting close to an enemy is risky, as the enemy might hit the agent. Trying to increase the score comes at a risk here as well.

The experiments showed that online parameter tuning did not mix well with multiple objectives, possibly due to the agent being scared of its own pheromones. It might be the case that the pheromones excreted by the agent cause the tuning mechanism to reduce the maximum rollout depth in games where the agent has to move around the same spot for the duration of the game. For example, in the game *Aliens*, the agent can only move laterally. As a result, the agent often encounters its own pheromones without being able to get away from them. This results in a lower state value, and consequently a lower rollout depth (see also subsection 6.3.2). The problem of pheromone buildup is not present with *Hungry Birds*, where the agent can always move greater distances without needing to pass the same tiles in a short time span. Possible solutions to the problem would be to change the pheromone decay depending on the degrees of freedom of the agent, or normalize pheromone levels.

It would be interesting to inspect how each option contributes to the decision making. As these are hand-crafted beforehand, it could be the case that not all are used, or that new ones could be made. Another thing to note is that, on its own, the agent with options was limited by the search depth. By doing this, already available information about the game is essentially discarded. If you take a look at *Camel Race* for example, the finish line is one of the options. It is, however, excluded as it is too far away. Agent performance could be improved if options were permitted to have unrestricted depth, only limiting the default policy to a fixed number of actions into the future.

7.3 Future Work

This research opens up multiple possibilities for future research. Firstly, additional objective functions can be designed to be incorporated in the multi-objective approach. This could, for example, include defeating enemies or gathering resources. Secondly, research can focus on changes in performance when changing the weights of the different objective functions. These could also be tuned online, letting the agent adapt its evaluation function to the game being played. Thirdly, future research should take a look at hyper heuristics. The results showed that agents performing well on deterministic games did not necessarily perform well on stochastic games and vice versa. A portfolio of multiple different agents can be constructed with a hyper heuristic that chooses the most appropriate agent to play the current game. Lastly, future research could test agents on more games from the GVGAI framework. As each game is implemented in an open format, the framework can be extended with new games endlessly, continuing to pose new challenges for the game playing agents.

Bibliography

[1] T. Schaul, J. Togelius, and J. Schmidhuber, "Measuring intelligence through games", *CoRR*, 2011. arXiv: `1109.1314`.

[2] M. Campbell, A. J. Hoane Jr, and F. Hsu, "Deep Blue", *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, *et al.*, "Mastering the game of Go with deep neural networks and tree search", *Nature*, vol. 529, pp. 484–489, 2016.

[4] P. S. Churchland and T. J. Sejnowski, *The computational brain.* MIT press, 2016.

[5] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game AI research and competition in StarCraft", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, pp. 293–311, 2013.

[6] S. J. Russell, "Rationality and intelligence", *Artificial Intelligence*, vol. 94, pp. 57–77, 1997.

[7] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the AAAI competition", *AI magazine*, vol. 26, pp. 62–72, 2005.

[8] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents", *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[9] D. Perez-Liebana, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, *et al.*, "The 2014 general video game playing competition", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, pp. 229–243, 2016.

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

[11] D. Perez-Liebana, S. Samothrakis, J. Togelius, S. M. Lucas, and T. Schaul, "General video game AI: Competition, challenges and opportunities", in *Thirtieth AAAI Conference on Artificial Intelligence*, AAAI Press, 2016, pp. 4335–4337.

[12] G. Chaslot, S. C. J. Bakkes, I. Szita, P. H. M. Spronck, M. Mateas, and C. Darken, "Monte-Carlo tree search: A new framework for game AI", in *Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AAAI Press, 2008, pp. 216–217.

[13] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, *et al.*, "A survey of Monte Carlo tree search methods", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, pp. 1–43, 2012.

[14] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General video game AI: A multi-track framework for evaluating agents, games and content generation algorithms", *CoRR*, 2018. arXiv: `1802.10363`.

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

[16] C. F. Sironi, J. Liu, D. Perez-Liebana, R. D. Gaina, I. Bravi, S. M. Lucas, *et al.*, "Self-adaptive MCTS for general video game playing", in *International Conference on the Applications of Evolutionary Computation*, Springer, 2018, pp. 358–375.

[17] M. De Waard, D. M. Roijers, and S. C. J. Bakkes, "Monte Carlo tree search with options for general video game playing", in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2016, pp. 1–8.

[18] D. Perez-Liebana, S. Mostaghim, S. Samothrakis, and S. M. Lucas, "Multi-objective Monte Carlo tree search for real-time games", *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, pp. 347–360, 2015.

[19] C. F. Sironi and M. H. M. Winands, "On-line parameter tuning for Monte-Carlo tree search in general game playing", in *Workshop on Computer Games*, Springer, 2017, pp. 75–95.

[20] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multi-armed bandit problem", *Machine Learning*, vol. 47, pp. 235–256, 2002.

[21] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games", in *Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AAAI Press, 2014, pp. 58–64.

[22] C. F. Sironi and M. H. M. Winands, “Analysis of self-adaptive Monte Carlo tree search in general video game playing”, in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2018, pp. 1–4.

[23] S. Ontañón, “Combinatorial multi-armed bandits for real-time strategy games”, *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[24] D. Perez-Liebana, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, *et al.*, “Solving the physical traveling salesman problem: Tree search and macro actions”, *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, pp. 31–45, 2013.

[25] E. J. Powley, D. Whitehouse, and P. I. Cowling, “Monte Carlo tree search with macro-actions and heuristic route planning for the physical travelling salesman problem”, in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2012, pp. 234–241.

[26] D. Perez-Liebana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, “Introducing real world physics and macro-actions to general video game AI”, in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2017, pp. 248–255.

[27] F. Frydenberg, K. R. Andersen, S. Risi, and J. Togelius, “Investigating MCTS modifications in general video game playing”, in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2015, pp. 107–113.

[28] 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, pp. 181–211, 1999.

[29] P. E. Hart, N. J. Nilsson, and B. Raphael, “A formal basis for the heuristic determination of minimum cost paths”, *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, 1968.

[30] D. Perez-Liebana, S. Mostaghim, and S. M. Lucas, “Multi-objective tree search approaches for general video game playing”, in *2016 IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2016, pp. 624–631.

[31] K. Deb, *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley, 2001.

[32] E. Zitzler, *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Citeseer, 1999.

[33] D. Perez-Liebana, S. Samothrakis, and S. Lucas, “Online and offline learning in multi-objective Monte Carlo tree search”, in *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, IEEE, 2013, pp. 1–8.

[34] D. Perez-Liebana, J. Dieskau, M. Hunermund, S. Mostaghim, and S. Lucas, "Open loop search for general video game playing", in *2015 Annual Conference on Genetic and Evolutionary Computation*, ACM, 2015, pp. 337–344.

[35] C. Guerrero-Romero, A. Louis, and D. Perez-Liebana, "Beyond playing to win: Diversifying heuristics for GVGAI", in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2017, pp. 118–125.

[36] R. D. Gaina, J. Liu, S. M. Lucas, and D. Perez-Liebana, "Analysis of vanilla rolling horizon evolution parameters in general video game playing", in *European Conference on the Applications of Evolutionary Computation*, Springer, 2017, pp. 418–434.

[37] I. Bravi, D. Perez-Liebana, S. M. Lucas, and J. Liu, "Shallow decision-making analysis in general video game playing", in *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, IEEE, 2018, pp. 1–8.

Appendix A: Extended Experiment Results

Games	A	A_s	A_m	A_o	A_{sm}	A_{so}	A_{mo}	A_{smo}
Aliens	**100.0 (± 0.00)**	**100.0 (± 0.00)**	98.8 (± 0.49)	**100.0 (± 0.00)**	38.8 (± 2.18)	99.8 (± 0.20)	96.8 (± 0.79)	41.6 (± 2.21)
Bait	10.6 (± 1.38)	16.8 (± 1.67)	**19.4 (± 1.77)**	12.4 (± 1.48)	10.0 (± 1.34)	16.8 (± 1.67)	19.2 (± 1.76)	14.4 (± 1.57)
Butterflies	95.8 (± 0.90)	91.6 (± 1.24)	98.4 (± 0.56)	**99.6 (± 0.28)**	99.2 (± 0.40)	99.2 (± 0.40)	96.0 (± 0.88)	99.2 (± 0.40)
Camel Race	2.8 (± 0.74)	7.4 (± 1.17)	41.6 (± 2.21)	7.2 (± 1.16)	41.8 (± 2.21)	**100.0 (± 0.00)**	46.6 (± 2.23)	99.6 (± 0.28)
Chase	4.2 (± 0.90)	3.8 (± 0.86)	2.2 (± 0.66)	**12.6 (± 1.49)**	6.4 (± 1.10)	8.4 (± 1.24)	2.2 (± 0.66)	10.2 (± 1.35)
Chopper	**93.6 (± 1.10)**	87.4 (± 1.49)	89.0 (± 1.40)	84.6 (± 1.62)	25.6 (± 1.95)	24.6 (± 1.93)	85.8 (± 1.56)	12.8 (± 1.50)
Crossfire	3.4 (± 0.81)	5.2 (± 0.99)	26.8 (± 1.98)	4.8 (± 0.96)	19.0 (± 1.76)	5.2 (± 0.99)	37.6 (± 2.17)	**45.2 (± 2.23)**
Digdug	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)	0.0 (± 0.00)
Escape	0.0 (± 0.00)	1.6 (± 0.56)	47.2 (± 2.23)	18.4 (± 1.73)	**57.0 (± 2.22)**	30.0 (± 2.05)	52.8 (± 2.23)	52.8 (± 2.23)
Hungry Birds	38.2 (± 2.18)	49.6 (± 2.24)	98.4 (± 0.56)	43.6 (± 2.22)	99.0 (± 0.45)	99.6 (± 0.28)	96.0 (± 0.88)	**100.0 (± 0.00)**
Infection	98.0 (± 0.63)	96.8 (± 0.79)	**100.0 (± 0.00)**	96.8 (± 0.79)	**100.0 (± 0.00)**	98.8 (± 0.49)	**100.0 (± 0.00)**	**100.0 (± 0.00)**
Intersection	**100.0 (± 0.00)**	**100.0 (± 0.00)**	91.4 (± 1.26)	**100.0 (± 0.00)**	57.2 (± 2.21)	96.0 (± 0.88)	97.0 (± 0.76)	36.2 (± 2.15)
Lemmings	0.0 (± 0.00)	0.0 (± 0.00)	0.4 (± 0.28)	0.0 (± 0.00)	**1.6 (± 0.56)**	0.0 (± 0.00)	0.2 (± 0.20)	1.4 (± 0.53)
Missile Command	67.0 (± 2.10)	74.2 (± 1.96)	47.0 (± 2.23)	75.0 (± 1.94)	49.6 (± 2.24)	**78.4 (± 1.84)**	60.0 (± 2.19)	56.2 (± 2.22)
Modality	26.6 (± 1.98)	**37.0 (± 2.16)**	23.0 (± 1.88)	27.8 (± 2.01)	28.8 (± 2.03)	32.0 (± 2.09)	23.6 (± 1.90)	28.8 (± 2.03)
Plaque Attack	83.4 (± 1.67)	88.0 (± 1.45)	**100.0 (± 0.00)**	94.2 (± 1.05)	80.4 (± 1.78)	92.2 (± 1.20)	99.4 (± 0.35)	80.6 (± 1.77)
Roguelike	0.0 (± 0.00)	0.0 (± 0.00)	9.6 (± 1.32)	0.2 (± 0.20)	1.0 (± 0.45)	1.2 (± 0.49)	**11.4 (± 1.42)**	6.2 (± 1.08)
Seaquest	60.2 (± 2.19)	72.4 (± 2.00)	77.2 (± 1.88)	**89.6 (± 1.37)**	11.6 (± 1.43)	81.0 (± 1.76)	67.4 (± 2.10)	14.4 (± 1.57)
Survive Zombies	44.2 (± 2.22)	45.0 (± 2.23)	28.8 (± 2.03)	**53.8 (± 2.23)**	37.4 (± 2.17)	49.6 (± 2.24)	38.2 (± 2.18)	45.4 (± 2.23)
Wait for Breakfast	4.2 (± 0.90)	10.6 (± 1.38)	**85.2 (± 1.59)**	69.0 (± 2.07)	75.8 (± 1.92)	77.2 (± 1.88)	76.0 (± 1.91)	80.4 (± 1.78)
Average	41.6 (± 0.98)	44.4 (± 1.11)	54.2 (± 1.22)	49.5 (± 1.13)	42.0 (± 1.42)	54.5 (± 1.08)	**55.3 (± 1.31)**	46.3 (± 1.36)

Table A.1: Percentage of victories achieved for each agent. The standard error is shown in parenthesis. The highest percentage for each game is shown in bold. Additionally, the average percentage of game victories over all 10,000 playthroughs (20 games × 5 levels × 100 runs) is included.

Games	A	A_s	A_m	A_o	A_{sm}	A_{so}	A_{mo}	A_{smo}
Aliens	67.6 (± 0.65)	63.7 (± 0.58)	**69.5 (± 0.68)**	68.2 (± 0.67)	41.1 (± 0.94)	63.9 (± 0.58)	68.9 (± 0.72)	41.2 (± 0.96)
Bait	2.4 (± 0.15)	4.7 (± 0.26)	4.8 (± 0.26)	1.1 (± 0.07)	3.3 (± 0.17)	4.0 (± 0.21)	**6.0 (± 0.31)**	4.1 (± 0.21)
Butterflies	30.7 (± 0.70)	**32.0 (± 0.71)**	30.2 (± 0.62)	29.5 (± 0.64)	29.1 (± 0.60)	28.5 (± 0.59)	**32.0 (± 0.68)**	29.0 (± 0.59)
Camel Race	-0.7 (± 0.02)	-0.7 (± 0.03)	-0.2 (± 0.04)	-0.7 (± 0.02)	-0.2 (± 0.04)	**1.0 (± 0.00)**	-0.1 (± 0.04)	**1.0 (± 0.01)**
Chase	2.8 (± 0.10)	2.5 (± 0.10)	2.2 (± 0.09)	**3.5 (± 0.12)**	2.6 (± 0.10)	2.5 (± 0.10)	1.6 (± 0.07)	2.6 (± 0.10)
Chopper	**14.4 (± 0.24)**	11.2 (± 0.28)	13.4 (± 0.29)	11.0 (± 0.34)	-2.1 (± 0.29)	-2.7 (± 0.28)	12.4 (± 0.33)	-3.4 (± 0.26)
Crossfire	0.1 (± 0.04)	-0.2 (± 0.05)	0.3 (± 0.11)	0.1 (± 0.05)	-0.1 (± 0.10)	-0.1 (± 0.06)	1.1 (± 0.13)	**1.8 (± 0.13)**
Digdug	14.6 (± 0.50)	11.5 (± 0.41)	13.9 (± 0.52)	13.0 (± 0.43)	4.6 (± 0.24)	11.4 (± 0.40)	**15.2 (± 0.50)**	4.3 (± 0.22)
Escape	-0.0 (± 0.01)	-0.0 (± 0.01)	0.1 (± 0.04)	0.1 (± 0.02)	0.1 (± 0.04)	0.2 (± 0.02)	**0.5 (± 0.02)**	**0.5 (± 0.02)**
Hungry Birds	35.0 (± 2.14)	47.6 (± 2.24)	98.2 (± 0.60)	46.0 (± 2.23)	99.4 (± 0.35)	99.6 (± 0.28)	95.8 (± 0.90)	**99.8 (± 0.20)**
Infection	15.9 (± 0.40)	14.2 (± 0.37)	**22.2 (± 0.46)**	15.0 (± 0.38)	19.0 (± 0.47)	16.8 (± 0.40)	20.7 (± 0.45)	17.4 (± 0.41)
Intersection	1.0 (± 0.00)	1.1 (± 0.04)	63.3 (± 1.18)	2.6 (± 0.18)	28.4 (± 1.63)	2.4 (± 0.16)	**86.6 (± 0.93)**	24.0 (± 1.60)
Lemmings	**-0.2 (± 0.02)**	-1.8 (± 0.07)	-17.9 (± 0.51)	-1.6 (± 0.08)	-21.3 (± 0.56)	-8.1 (± 0.15)	-8.1 (± 0.35)	-24.9 (± 0.61)
Missile Command	5.1 (± 0.22)	5.2 (± 0.23)	3.9 (± 0.21)	**6.0 (± 0.23)**	3.6 (± 0.19)	5.6 (± 0.24)	4.8 (± 0.23)	3.8 (± 0.19)
Modality	0.3 (± 0.02)	**0.4 (± 0.02)**	0.2 (± 0.02)	0.3 (± 0.02)	0.3 (± 0.02)	0.3 (± 0.02)	0.2 (± 0.02)	0.3 (± 0.02)
Plaque Attack	41.7 (± 0.87)	42.7 (± 0.84)	**53.8 (± 0.69)**	44.7 (± 0.85)	24.5 (± 0.79)	42.6 (± 0.61)	52.8 (± 0.65)	22.5 (± 0.71)
Roguelike	6.0 (± 0.30)	4.7 (± 0.28)	**10.8 (± 0.48)**	3.7 (± 0.19)	4.5 (± 0.23)	4.2 (± 0.21)	6.4 (± 0.34)	5.6 (± 0.25)
Seaquest	1675.8 (± 74.83)	2128.2 (± 82.82)	**2673.4 (± 95.23)**	2427.9 (± 99.72)	740.8 (± 72.27)	2428.7 (± 100.66)	2327.4 (± 81.12)	1031.2 (± 74.19)
Survive Zombies	2.9 (± 0.16)	3.0 (± 0.16)	1.4 (± 0.13)	2.7 (± 0.14)	3.2 (± 0.16)	3.0 (± 0.15)	2.3 (± 0.14)	**3.8 (± 0.17)**
Wait for Breakfast	0.0 (± 0.01)	0.1 (± 0.01)	**0.9 (± 0.02)**	0.7 (± 0.02)	0.7 (± 0.02)	0.8 (± 0.02)	0.8 (± 0.02)	0.8 (± 0.02)
Average	95.8 (± 4.07)	118.5 (± 4.48)	**152.2 (± 5.11)**	133.7 (± 5.32)	49.1 (± 3.96)	135.2 (± 5.26)	136.4 (± 4.40)	63.3 (± 4.04)

Table A.2: The final game score which the agent achieved on each game from the test set. The standard error is shown in parenthesis. The highest score for each game is shown in bold. Additionally, the average game score over all 10,000 playthroughs (20 games × 5 levels × 100 runs) is shown.