Universiteit Leiden
The Netherlands

Opleiding Informatica

The Effect of Monte Carlo Tree Search on Modern Board Game Elements

David Nieuwenhuizen

Supervisors:
Walter Kosters & Hendrik Jan Hoogeboom

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

22/01/2021

Abstract

Small World is a territory building game designed by Philippe Keyaerts and published by Days of Wonder in 2009. Players choose their civilization from a stack of randomly determined combinations of possible races and special powers. In this thesis we implement and experiment with a Monte Carlo Tree Search (MCTS) agent using the UCT tree policy for this game. The game has a rather substantial branching factor which we attempt to overcome by splitting the turn into separate sequential actions. We discovered the agent to perform better when set to explore the game tree more regularly. During analysis of the actions of the agent we also determined that even though MCTS performs better than random play it still has quite some room for improvement due to its lacking ability to plan ahead.

Contents

1 Introduction **1**

2 Small World **2**
2.1 Overview . . . 2
2.2 Rules of the Game . . . 3

3 Related Work **6**

4 Monte Carlo Tree Search **6**
4.1 UCT Tree Policy . . . 7

5 Agents **9**
5.1 Random Agent . . . 9
5.2 Pure Monte Carlo Agent . . . 9
5.3 MCTS Agent . . . 9

6 Experiments **10**
6.1 Algorithm Tuning . . . 10
6.2 Computation Time and Depth . . . 12
6.3 MCTS Experiments . . . 13
6.4 Game Analysis . . . 13

7 Conclusions and Further Research **19**

References **20**

1 Introduction

Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results [BPW+12]. Over the years it has gained considerable traction due to its success at playing many classic games at a high level and beating professional human players, among them being Go [HPA+18].

The ability of MCTS to play classic games very effectively is no secret, yet the game industry has not been sitting idly by in the meanwhile. As opposed to the classics where just a few rules and short turns are enough, modern games have become increasingly more complex. One page of rules is no longer enough to get board game enthusiasts interested in purchasing your product. Many modern games flaunt in their marketing that "No single game ever plays the same". This tends to mean that a single turn of a player could consist of multiple different actions and that stochastic elements are almost ubiquitous.

So for this bachelor's thesis the aim is to create an implementation of an MCTS agent for one such modern board game and analyse its behaviour. The chosen game is titled Small World (See Figure 1), a territory building game designed by Philippe Keyaerts and published by Days of Wonder in 2009 [Day]. This game was primarily chosen because it contains a relatively small amount of stochastic elements, has a fixed game length and was expected to have a tremendous search space due to its many different actions and additional rules.

In order to effectively make use of MCTS the size of the search space has to be overcome. The game Aarima [SS03] is notoriously difficult to solve due to its high branching factor of 16,000, which does not come close to the potential branching factor of over 600,000 that we calculated for Small World. It has been attempted to tackle Aarima's large branching factor via MCTS, but the random playouts that MCTS uses do not appear to effectively deal with the many useless moves that a player can take [Koz09]. This is why for Small World we decided to split a turn into more manageable chunks. A node is no longer defined as a single turn of a player but by the individual actions a player takes in a turn. A turn tends to consist of slightly more than 10 of these actions. The idea is to attain a deeper search tree, between 200 and 300 nodes against 21 nodes, rather than an impossibly wide one.

Figure 1: A two player setup of the game Small World. Source: https://www.daysofwonder.com.

The structure of this thesis is the following: this chapter contains the introduction; Section 2 includes details on the game; Section 3 discusses related work; Section 4 describes some background knowledge about Monte Carlo Tree Search.
A digital version of Small World had to be implemented for this thesis. In Section 5 we give details about the implemented agents used for the experiments. These agents were written in order of increasing complexity during the process of implementing the game itself. The random agent was primarily used as a tool to search for bugs and as a base for the Pure Monte Carlo and the MCTS agent.
Section 6 describes the experiments and their outcome. Relatively broad experiments are performed against the random agent in order determine settings for the MCTS agent that would provide the most effective play during later experiments. After that we take a look at the computation speed of the MCTS agent. The smartest version of the MCTS agent is then made to play against players that can actually put up a fight against it: the Pure Monte Carlo agent, weaker MCTS agent and itself. Finally we have a look at the specific actions the agent tends to take and determine what type of actions are more effective than others.
Section 7 then gives a concluding statement.
This bachelor's thesis at LIACS, the Leiden Institute of Advanced Computer Science at Leiden University, is supervised by Walter Kosters and Hendrik Jan Hoogeboom.

2 Small World

In this chapter we will provide a short overview of the game Small World followed by a detailed explanation of the rules of the game.

2.1 Overview

Small World is a turn-based territory building game designed by Philippe Keyaerts and published by Days of Wonder in 2009 [Day]. In the game players control civilizations of fantasy creatures wishing to dominate the world that they live in. Players choose their civilization from a stack of randomly determined combinations of possible races and special powers. These two components that form such a civilization will be referred to as *combos* in this thesis.
The board on which the game is played is divided into regions, with each region having a specific traits. These traits usually do not impact the value of a region, unless the combo of a player says so. A player wins the game by having the most victory coins at the end of the ten turn game. A player starts the game with five victory coins and gains more based on the amount of regions on the board that they control at the end of their turn, plus any that they may earn from their combo. The only way to lose victory coins is by spending them when a player chooses a new combo. A player takes over regions entirely deterministically by spending an amount of tokens that represent their troops based on the defences present in the region they wish to conquer. However, for some conquests, usually the last one in the turn, a player is allowed to roll a so-called reinforcement die that can give them zero to three bonus troops. The combo stack and this reinforcement die are the only stochastic elements in the game.
The game can be played by two to five players, and we will limit it to only two players in this thesis. Although the rules state that the amount of victory coins a player owns is to be kept hidden

it is very easy for any player to keep track of how many victory coins another player has earned and spent through the game, we will therefore treat this game as a game with perfect information.

2.2 Rules of the Game

In this section we will give an outline of the rules of the game Small World.

Structure of a Turn

Every turn follows the same order of actions that a player will take. These actions will be elaborated upon later. Here is an overview of them:

- If a player does not currently own a combo they must choose a new one from the stack.
- Or if a player did already own a combo they may put that combo into decline and end their turn.
- A player takes all but one of their active combo's tokens from any regions this combo controls. They may also choose to take the last token from any of these regions, therefore abandoning this region.
- A player conquers as many regions as they can or wish by spending their tokens.
- For their last conquest a player may roll the reinforcement die to gain additional troops.
- A player may redistribute all of their active tokens across the regions this combo controls.

At the end of their turn a player scores an amount of victory coins equal to the amount of regions they control plus any that they may have earned from their combo.

Choosing a Combo

At the start of the game the fourteen race banners and the twenty special power badges are shuffled separately and placed on a stack. Five of each are removed from the stack and placed in a column above it to form five different combos. The top combo of the stack remains visible, revealing a final sixth combo. This column can be seen in Figure 1 to the left of the map. Each race and special power has a special rule associated with it that will influence how the player will play the game. A player chooses a combo from the six currently visible to them. To do so they must place a victory coin on all of the combos that are above the combo that they wish to choose, so if a player wants to choose the combo that is currently on top of the stack they must place a coin on all the other combos. They then remove the chosen combo from the column and move the combos below it upwards in the column, revealing a new combo on top of the stack. If the chosen combo already has some victory coins on it the player receives these extra coins. Each combo has two numbers written on it. The sum of these numbers shows how many tokens the player receives when they choose this combo. Some combos may receive more tokens throughout the game, but they are still limited by the number of tokens physically available in the game. An overview of the special rules associated with all of the powers and races available in the game can be found in Figure 2. For this

thesis we will not be using every power and race available. The Berserk power gives the player an overly random rule to use for our implementation. The Ghouls race essentially gives the player two turns in a row which would inflate the search space far too much. Without these two options we end up with $13 \cdot 19 = 247$ possible combos to exist.

Figure 2: A summary of all of the races and powers. Source: https://www.daysofwonder.com.

Going into Decline

Instead of regular turn a player may instead choose for their current combo to go into decline. This is done by flipping the race banner of their combo and discarding their power badge, unless specified otherwise by the power. They then remove all but one token from each owned region and flip the remaining token. A player may only have one combo in decline at one time. If they put a second one in decline they must remove all of the tokens of the previous combo still on the map and discard the race banner. They may then proceed with putting the new combo into decline. Normally a discarded race banner is put at the bottom of the stack, but in this thesis it is shuffled back into it instead. At the end of a player's turn they will now also receive victory coins from any of their tokens that are in decline. The turn after going into decline is played following the same rules as their first turn by picking a new combo from the stack.

Abandoning a Region

When readying tokens to conquer new ones a player takes all but one of their active combo's tokens from any regions this combo controls into their hand. If they wish to use extra tokens the player may at this stage decide to abandon any of the regions this combo owns, grabbing the last token.

This means that this region no longer belongs to them and must be re-conquered. A player may choose to abandon all of their owned regions as well in order to start conquering as if it was their first turn.

Conquering Regions

In order to conquer a region a player must spend an amount of tokens from their hand and place it on the region they wish to conquer. The amount required equals two tokens plus one additional token for each token already on the region plus one additional token for any defence the region may have. A region could have a mountain in it, which counts as such a defence.
If there were any tokens present in the conquered region one of these tokens is immediately discarded. If there are any tokens remaining they are given to the player that owns them. At the end of the current turn this player may then redeploy these tokens to any regions they own. If they no longer own any regions they must keep these tokens in hand until their next turn.
A player may not conquer just any region. If it is their first conquest, meaning that they currently do not own any regions with this combo, they must conquer a region that is on the outer edge of the map or adjacent to a sea region. A sea region may normally not be conquered. After this first conquest they may conquer any region that is adjacent to a region that their combo owns.
For a player's final conquest they may wish to conquer a region that they currently do not have enough tokens for. If they still have at least one token available they may then roll the reinforcement die for this final conquest. The reinforcement die has three blank sides, one side for one additional token, one side for two and one side for three tokens. The player must first declare which region they wish to attempt this final conquest on. If they roll enough extra tokens they successfully conquer the region and place any remaining tokens they had on this region. If they fail they then must place any remaining tokens they had on a region they already own. If they do not own any regions they simply keep these tokens in hand for their next turn.

End of Turn

At the end of a player's turn they may redeploy their troops. This is done by redistributing their tokens among all of their owned regions as they wish, but always leaving at least one token per region. They then score one victory coin for each region that they currently own, plus any that they may receive from their combo's special rules.

Important Highlights

This thesis will only be looking at games between two players on the two player map provided in the game Small World.
Whenever a race banner is discarded completely it is normally placed at the bottom of the stack. In the digital version of the game used for this thesis it is instead shuffled back into the stack. This makes it more likely that one race could be played twice in one game.
We will be playing without the Ghouls race and the Berserk power.
There are two major random components to the game. The first being the shuffling of the combo stack, meaning that the starting state of the game is usually different. The second is the reinforcement die that a player may occasionally roll on their turn that could give them extra temporary troops to conquer one more region.

3 Related Work

In this section we present some related work with the Monte Carlo Tree Search (MCTS) algorithm in similar games. No previous work in specific regarding Small World was found.

MCTS has been used in many different strategic games. It has already seen success in highly non-deterministic games such as Settlers of Catan [SCS10]. This paper shows that in games such as these adding minor domain knowledge to the MCTS algorithm can greatly increase the strength of the agent.
An implementation of MCTS has recently been used to create a relatively powerful agent for the game of Diplomacy [TC20]. In this complex game players take their turns simultaneously which makes the success of this agent all the more impressive. Notable as well is the fact that the reward value used for the algorithm is a reasonably greedy interpretation of a successful turn.
MCTS has a powerful general effect. It manages to combine the accuracy of complete search trees with the speed and efficiency of random sampling. Using this framework in conjunction with deep learning and reinforcement learning, MCTS has been used to create the *AlphaGo* algorithm [SSS+17].

Aarima is a game invented in 2002 by Omar Syed. It was specifically designed to be difficult to solve by computer, having a branching factor of roughly 16,000. Aarima thus provides a similar problem as Small World. Several times David Wu has successfully tackled this problem without the use of MCTS [Wu11, Wu15]. Wu accomplishes effective play in real time by employing machine learning and classic algorithms like Alpha-Beta search. A significant portion of the substantial search space is negated by quickly pruning the tree and only test moves that are known to be powerful. This method involves predicting the most powerful move, ordering them as such and accurately quantifying the strength of any particular game state. It is important to note the strength of these methods in the complex game of Aarima since MCTS has also been tried to solve it [Koz09]. The agents that use MCTS, however, have been found to be relatively weak. The large search space of Aarima includes many moves that are actively bad for the player and MCTS does not effectively sift through these actions with its random playout.

4 Monte Carlo Tree Search

In this section we will explain the basic Monte Carlo Tree Search (MCTS) algorithm and the tree policy we have selected for our implementation.

The basic algorithm involves iteratively building a search tree until some predefined computational budget — typically a time, memory or iteration constraint — is reached, at which point the search is halted and the best performing root action returned. Each node in the search tree represents a state of the domain, and directed links to child nodes represent actions leading to subsequent states [BPW+12].
One iteration of the algorithm consists of four steps [CBSS08]. A diagram of a single iteration can also be found in Figure 3.

1. **Selection** Starting at the root node the next expandable node is selected via the recursive application of some tree policy. A node is considered expandable if it does not represent a

terminal state and has unvisited children.

2. **Expansion** A child node is added to the tree, following the available actions.
3. **Simulation** The remaining game from the child is played out according to some default policy, usually via random actions.
4. **Backpropagation** The result of the simulation is passed back up through the tree and updates the statistics of all parent nodes.

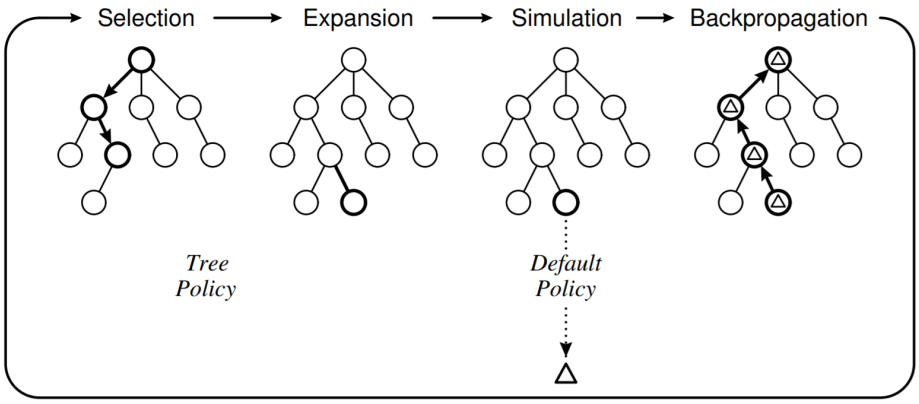

Figure 3: One iteration of the general MCTS approach [BPW+12].

The pseudocode for this general MCTS approach can be found in Algorithm 1 [BPW+12]. Here v denotes a node with state s and action a. Δ is the reward gained from running the default policy from a state s.

Algorithm 1 General MCTS approach

```
function MctsSearch(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_ℓ ← TreePolicy(v_0)
        Δ ← DefaultPolicy(s(v_ℓ))
        Backup(v_ℓ, Δ)
    return a(BestChild(v_0))
```

4.1 UCT Tree Policy

The order in which nodes are expanded is critical for MCTS. This order is determined by the tree policy used. The Uniform Confidence bound for Trees (UCT) tree policy determines the next node j to be expanded by trying to maximize:

$$UCT = X_j + 2 \cdot C_p \cdot \sqrt{\frac{2 \cdot \ln(n)}{Visits_j}}$$

Here X_j is the normalized average reward of child node j, $C_p > 0$ is a constant, n is the amount of times the parent node has been visited and $Visits_j$ is the amount of time the child node has been visited. The pseudocode for the complete UCT Monte Carlo Tree Search can be found in Algorithm 2 [BPW+12]. Here $f(s, a)$ gives the resulting state after applying action a to state s. $A(s)$ is the set of all actions possible from state s. $N(v)$ is the visit count of node v and $Q(v)$ is the total simulation reward of node v. $\Delta(v, p)$ is the reward associated with node v and player p.

Algorithm 2 The UCT algorithm

function UctSearch(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TreePolicy(v_0)
    $\Delta \leftarrow$ DefaultPolicy($s(v_l)$)
    Backup(v_l, Δ)
  return a(BestChild(v_0))

function TreePolicy(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return Expand(v)
    else
      $v \leftarrow$ BestChild(v, Cp)
  return v

function Expand(v)
  choose $s \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BestChild(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\operatorname{argmax}} \quad \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$$

function DefaultPolicy(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function Backup(s, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Source: [BPW+12]

5 Agents

In this section we give some details on how all of the agents used in the experiments operate.

5.1 Random Agent

The basic random player plays the game in a step by step fashion just like a human player would. At each step the decision is then made based on an extremely simple random number generator. This agent is notably bad at using any combo that gives it extra options during the conquering phase, since using that power is seen as a 50/50 choice during each conquest. Each legal conquest has an equal chance of being picked, the agent does not look whether this conquest would be their last or not.
When redistributing tokens at the end of the turn the agent first picks up all but one of their tokens from owned regions as if readying troops. They then randomly place them down token per token.

5.2 Pure Monte Carlo Agent

The Pure Monte Carlo agent was made as a proof of concept and will also function as a more intelligent player to test the MCTS agent against. For both this agent and the MCTS agent only the major actions are properly calculated. Most actions relating to special rules from a combo are still executed randomly or done whenever it is possible. Under major actions we include:

- Choosing a combo
- Declining or not
- Abandoning regions
- Conquering regions
- Redeploying tokens

When the agent has to take one of these actions it will attempt every available option at that time and play 100 randomly simulated games to completion for each option. The option with the most simulated wins will then be chosen. In case of a tie the first action tried will be chosen.

5.3 MCTS Agent

For the MCTS agent the definition of a node in the search tree is important. If we were to treat a single turn as one node the resulting tree would have an exorbitantly large branching factor. Assume that every combo simply gives ten tokens to play with and no special rules. Then the first turn would already have $6 \cdot (30 \cdot 36 + 245 \cdot 84 + 623 \cdot 126) = 600,948$ distinct end states. Instead we define a node as a game state where the agent will have to perform a major action as defined in Section 5.2. This definition does not actually reduce the size of the search, yet it does reduce the branching factor of the tree to between 2 (when choosing to decline or not) and 20 (when conquering for the first time with the Flying power). It also allows the agent to potentially "give up" on a series of actions early on if it finds the first action already does not look promising. Besides

the game state a node also contains the action that led to the current state, which type of action is expected from the agent, the already calculated reward value and the amount of times the node has been visited. The reward value for our implementation is simply the amount of wins that have been simulated for the current node and its children.
The MCTS agent closely follows the algorithm as defined in Algorithm 2. When it has to perform a major action it will do the following:

1. Choose a node to expand via the UCT tree policy
2. Randomly choose an available action
3. Create a new child node with the chosen action
4. Randomly play out a complete game from the result of the action
5. Propagate the reward (win/loss) from the simulation back through the tree
6. Repeat until a set amount of nodes have been expanded

When the agent has to expand a node which involves one of the two stochastic elements it will simply do so and ignore any other potential outcomes that would have been possible. The reward from this action is still reduced based on the chance of that outcome occurring. Disabling the stochastic elements of the game did not appear to have a significant impact on the playing strength or behaviour of the agent.
The code of our implementation was written in C++.

6 Experiments

In this section we will look at the results from experimenting with the agents and interpret the data. First the Monte Carlo Tree Search (MCTS) agent will play several games against the random agent using varying settings for C_p and the cut-off point. From these same games we also look at the total computation time and the average depth of the game trees created. The most promising version of MCTS then plays against other agents to gauge its playing strength. Lastly we have a close look at the individual actions that the agent takes in order to determine if the agent managed to properly understand Small World. All experiment data is taken from 100 games played unless stated otherwise. Experiments were played on a laptop with an Intel i7-8550-U 8 core 1.8 GHz CPU.

6.1 Algorithm Tuning

In order to find the settings where our MCTS agent would have the most effective play several simple experiments were run first. All of these experiments were played against the random agent with the MCTS agent as the first player. Two results were looked at in order to determine playing strength: amount of games won and average victory coins (VC) at the end of the game. Small World does have a tie-breaker rule in place. Since this rule could still end up in a tie game we chose to ignore this rule and simply count an equal amount of victory coins as a tie.
A proper computation cut-off point was found first. For these experiments the exploration parameter C_p was chosen to be 0.5. From these experiments it was found that the MCTS agent benefits

from the extra granted computation time. However when given too much time playing strength drastically decreases. This is presumably due to the greedy low exploration parameter. The total wins for every cut-off point can be seen in Figure 4. The average victory coins per game for every cut-off point can be seen in Figure 5.

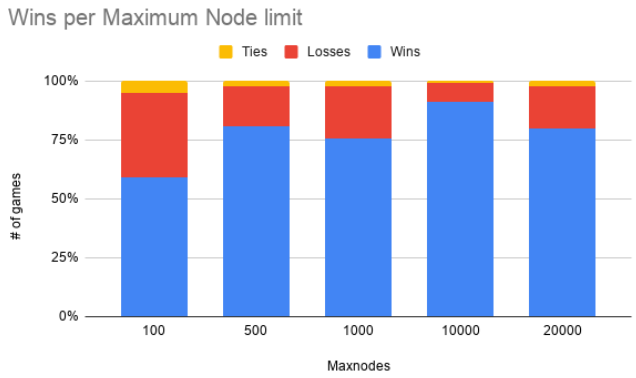

Figure 4: Winrate of MCTS vs. Random

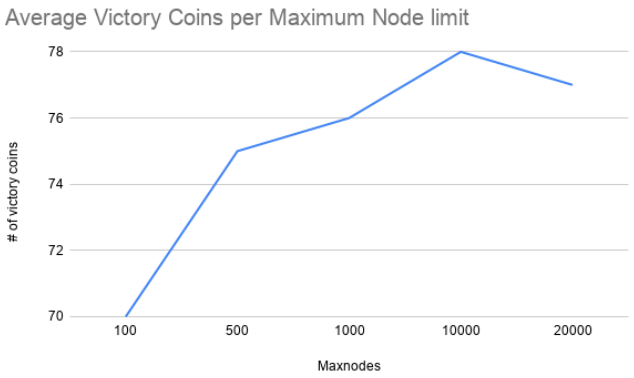

Figure 5: Victory Coins of MCTS vs. Random

Two separate experiments were run in order to tune the exploration parameter of UCT: One with a low cut-off point of 500 nodes and another with the most promising cut-off point of 10,000 nodes per search tree. The wins for the 500 nodes version can be seen in Figure 6 and the average victory coins per game in Figure 7. The wins for the 10,000 nodes version can be seen in Figure 8 and the average victory coins per game in Figure 9. The chosen tested values of C_p are the ones that have been found to be effective in other complex games [Roe12]. The agent with 10,000 nodes has more effective play across the board, yet both seem to improve more with $C_p > 1$. A remarkable result is the run with $C_p = 4$ and 10,000 nodes. It manages to win all of the games played in that experiment whilst attaining less overall victory coins than other values for C_p. Reviewing these results reveals that the random agent achieved less overall victory coins than it does otherwise. So this outlying result could be the result of the random agent playing exceptionally bad against this particular agent.

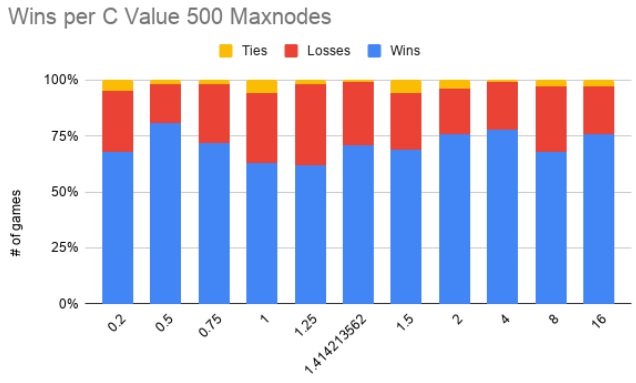

Figure 6: Winrate of MCTS vs. Random

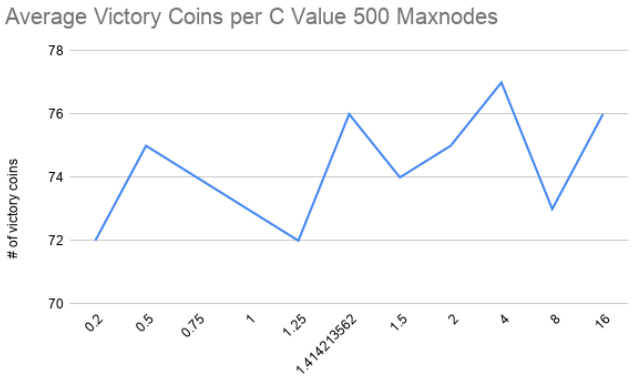

Figure 7: Victory Coins of MCTS vs. Random

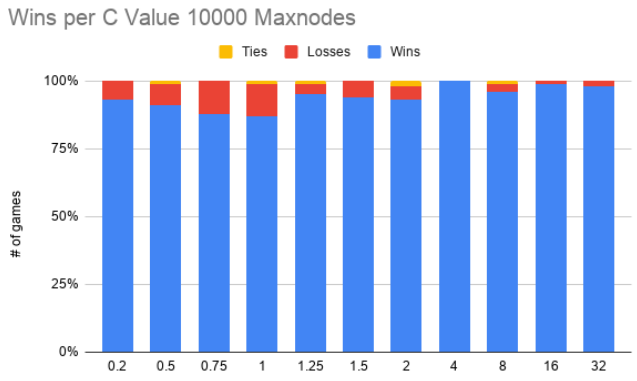

Figure 8: Winrate of MCTS vs. Random

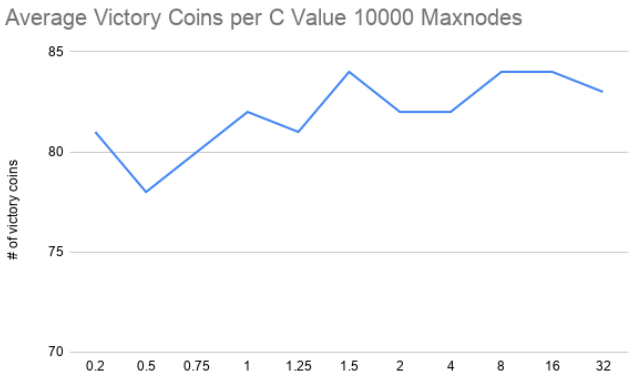

Figure 9: Victory Coins of MCTS vs. Random

From these results it seems that an MCTS agent with cut-off point of 10,000 nodes and $C_p = 4$ or $C_p = 16$ offers the most effective playing strength.

6.2 Computation Time and Depth

The following results are taken from games played between the random agent and the MCTS agent with $C_p = 0.5$. In Figure 10 we look at the average depth of a game tree for various cut-off points. The total execution time of 100 games between MCTS with an increasing cut-off point against a random agent can be found in Figure 11.

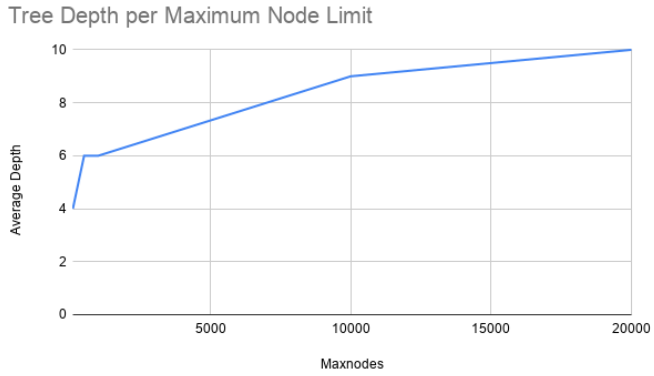

Figure 10: Depth of MCTS vs. Random

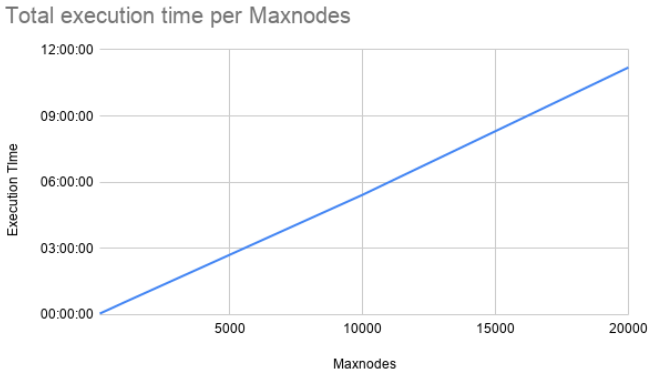

Figure 11: Time of MCTS vs. Random

The amount of decisions a player makes in a single turn ranges between one, when declining, to nineteen in the most extreme case. This means that even at 20,000 nodes the MCTS agent does not always reach the end of its own turn when constructing the search tree. Looking at the total execution there is not much to mention save for the fact that it is quite noticeable that no effort was made towards improving the calculation speed during implementation.
Throughout the course of a game the agent picks up in speed as the search space of the game naturally decreases and it becomes more obvious who is more likely to win. With a cut-off point of 10,000 nodes the first turn can take up to 5101 ms to calculate whilst the last turn would take 796

ms. In total the calculation of 10,000 nodes averages out at about 2691 ms throughout the games played.

6.3 MCTS Experiments

The following results are taken from games played with an MCTS agent with a cut-off point of 10,000 and $C_p = 4$ against several other agents. PureMC is the Pure Monte Carlo agent. MCTS[500/0.5] is an MCTS agent with a cut-off point of 500 and $C_p = 0.5$. MCTS[10,000/16] is an MCTS agent with a cut-off point of 10,000 and $C_p = 16$. The wins and average victory coins (VC) can be found in Table 1.

Opponent	Wins as Player 1	VC as Player 1	Wins as Player 2	VC as Player 2
PureMC	68	70	60	68
MCTS[500/0.5]	81	82	75	80
MCTS[10,000/16]	43	81	45	80

Table 1: MCTS against other agents

The MCTS agent outperforms both of the strictly simpler agents. The PureMC agent simulates more games per move on average than MCTS[500/0.5], thus it was expected to see the result of PureMC proving more difficult to beat than MCTS[500/0.5]. Against both of these agents the MCTS agent gets significantly more wins and slightly more victory coins when it is the first player as opposed to second. This suggests that the game holds a slight advantage for the first player. From these results it also appears that MCTS[10,000/16] is in fact a superior player. Because of this MCTS[10,000/16] was put against itself in another round of experiments. The results of these 100 extra games can be found in Table 2.

Wins for Player 1	VC for Player 1	Wins for Player 2	VC for Player 2
52	83	39	80

Table 2: MCTS[10,000/16] against itself

The difference between the first and second player is now more apparent. We can now determine that when both players have an equal playing strength there is reasonably high advantage for the first player.

6.4 Game Analysis

The following results are taken from 500 games played. We look at what type of actions our MCTS agent considers to be strong. For this agent the cut-off point is set at 10,000 nodes and $C_p = 16$. The agent is always first and is against a far weaker MCTS agent with the cut-off point set at 500 nodes and $C_p = 0.5$. Out of the 500 games played the agent won 436 games, lost 56 and tied 8.

Region Priority

We look at whether the MCTS agent determines if certain regions have a higher priority than others. We calculate a ratio of conquering and redeployment for each region. The conquering ratio is the

average amount of times a region is chosen to be conquered by the agent per game. Throughout the game regions can be conquered several times. These ratios for each region can be found in Figure 12. We expect this ratio to be slightly inflated for regions along the edge since they can be conquered on a player's first turn whilst the inner regions have to be reached first. The redeployment ratio is the average amount of tokens dedicated to defending a region if it is owned by the agent per game. These ratios for each region can be found in Figure 13. Both Figure 12 and Figure 13 show a graph representation of the map.

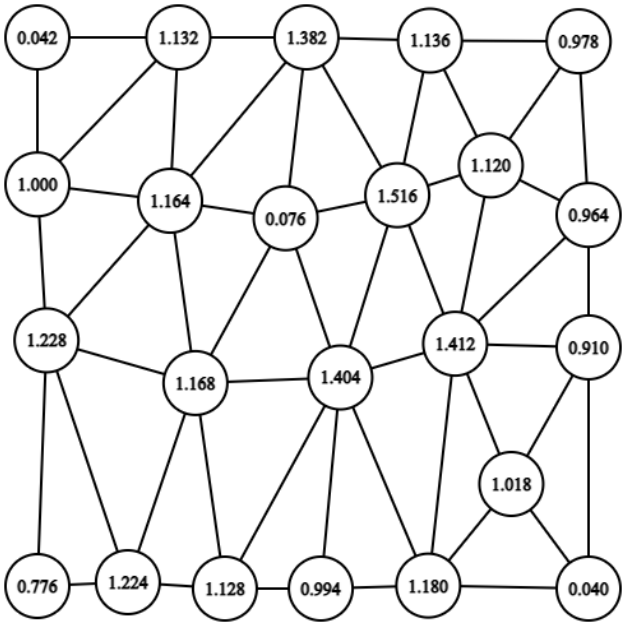

Figure 12: Conquering Ratio

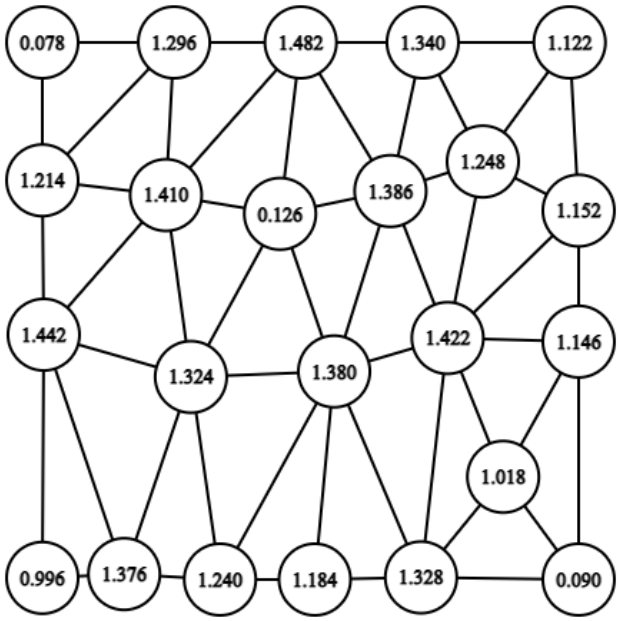

Figure 13: Redeployment Ratio

First let us explain the anomaly of the three regions with a conquering ratio < 0.100. These regions have the property of being either a lake or ocean, which may only be conquered with the Seafaring power. This power gets chosen in 15.4% of the games played, see Table 5. When adding the conquering ratios of the three regions together we get 0.158. This means that when the agent picks the Seafaring power it will conquer one of these special regions at least once, heavily prioritizing the lake over the two ocean regions. The redeployment ratio of these regions is so low since if a region is never owned by the agent the redeployment score for that game is set to 0.
Some regions cost more tokens to conquer than others from the start of the game. This is either due to natural defence which is always present in that region or because of a neutral token that is placed when setting up the game. The extra cost of attacking the neutral token appears to affect the priority of the agent. Regions with this token have a lower conquering ratio when comparing them to regions that do not and share a similar strategic position on the map. The natural defences have an even higher effect on the agent. Four regions have this property and three of them have a conquering ratio ≤ 1.000. The exception is the region with a superior strategic position in the center of the map which has a ratio of 1.516, which is higher than any other region. These regions do prove to be easier to defend, which can be seen by the fact that the agent dedicates fewer tokens to defending than it would for other regions.

The single biggest factor in the agent's priority appears to be the location of the region and its access to other regions. There is a positive correlation between the amount of neighbours a region has and both ratios of that region. The two most conquered regions both have seven neighbours and happen to border each other, making them both appealing for our agent. Considering that the only prerequisite for conquering a region is that it must border an owned region and that the player has enough total tokens to spend this tactic does make sense. Being surrounded by an opponent is strangely beneficial. It provides more options to conquer on the next turn and it does not leave the player more open to attacks. It is likely that these regions are also conquered more often simply because the agent has more chances to conquer due to the amount of regions that border them.

Miscellaneous Behaviour

We now look at some smaller details of the behaviour of our agent. The aggression of a player is defined by the average amount of times in one game that the player directly attacks one of the other players by successfully conquering one of their regions. Abandoning is then the average amount of times in a game the player abandons one of their owned regions when preparing for their next conquering phase. Declining is the average amount of times in a game the player declines their current combo and chooses a new one the next turn. The results for the random agent are also included for the sake of comparison and can all be seen in Table 3.

Agent	Aggression	Abandoning	Declining
MCTS[10,000/16]	3.082	3.466	2.476
Random	2.236	2.846	3.204

Table 3: Miscellaneous behavior data

Even though the MCTS agent does seek out the opponent more than the random agent an aggression ratio of 3.082 conquers is not that high. Having a region border one of the opponent's becomes quite common after turn 2, which means that our agent is specifically choosing to avoid the opponent if possible. This behaviour lines up with the results found in Figure 12. The agent heavily prefers to conquer as many regions as possible with as little cost as it can manage.
Contrary to the aggression ratio the abandoning and declining ratio are both blown out of proportion somehow. Abandoning is an action that is rarely useful. If that extra token only gives the player one additional region then there is no benefit unless that new region is of higher priority than the abandoned one. So it is remarkable that the MCTS agent abandons more often than random play. The declining ratio is sitting higher than two declines in a game. Considering the low amount of aggression in both players this means that the agent willingly sacrifices potential victory coins by repeatedly going into decline. Skipping more than two turns in a ten turn game is quite a loss. We decided to look at exactly which turns the agent uses to go into decline to see if we could find the source of this behaviour. These results can found in Table 4.

Turn	# of Declines
1	21
2	85
3	190
4	139
5	92
6	104
7	162
8	108
9	204
10	133

Table 4: Declining Timing

Do note that going into decline on turn 1 is only possible with the Stout power. Around turn 3 and 7 are both reasonable timings to go into decline, so it is no surprise to see those turns with a higher frequency. More notable is the high occurrence of both turn 9 and 10, close to the end of the game. After closer inspection we found that this behaviour occurs when the MCTS agent knows it has already won no matter what the other player might do. In those cases it will usually decline since that is faster. The same is true for the frequency of abandoning in later turns. If all actions have a perfect win-rate the agent will pick the first action that it tried. In the case of the abandoning action it is more likely for this to be abandoning at least one region than not abandoning at all.
The agent was quickly put against itself for a few games to see if any of these behaviours would change. Overall fewer regions were conquered, meaning that the conquering ratio of all regions was lower. The relative priority of these regions remained essentially unchanged. When facing a stronger opponent the agent got an aggression ratio of 3.88. The abandoning ratio and the declining ratio dropped to 2.33 and 2.10 respectively. The agent has to fight more for territory and does so accordingly. Fewer games have an obvious winner before the final turn, meaning that in the final turns the agent does not simply decline or abandon because it can. These data points appear more in line with games seen between human players.

Combo Choices

One of the most important aspects of Small World is choosing a combo that the player believes to be the most powerful with the current board state and adapting a strategy around that. To determine our agent's ability to do this we determined which combos it would pick over others. First we looked at all the powers and races individually, see Table 5 and Table 6.

Power	Times Chosen
Alchemist	86
Bivouacking	67
Commando	149
Diplomat	80
Dragon Master	72
Flying	59
Forest	55
Fortified	32
Heroic	70
Hill	69
Merchant	80
Mounted	96
Pillaging	110
Seafaring	77
Spirit	96
Stout	90
Swamp	75
Underworld	95
Wealthy	147

Table 5: Power Popularity

Race	Times chosen
Amazons	185
Dwarves	71
Elves	126
Giants	129
Halflings	106
Humans	83
Orcs	136
Ratmen	180
Skeletons	135
Sorcerers	79
Tritons	189
Trolls	85
Wizards	101

Table 6: Race Popularity

This was the first time we noticed the severe flaws in our MCTS agent. There is a significant difference between the most and least popular choices. We looked at all of these extreme outliers to determine the cause of this spread. A brief explanation of all of the powers and races can be found in Figure 2 on page 4. The outliers discussed in this section will be briefly explained here as well. We start with the three most popular powers: Commando, Wealthy and Pillaging. Commando reduces the cost of conquering by one token. Wealthy gives a one-time bonus of seven victory coins. Pillaging gives one additional victory coins per region conquered that already had a token in it. Commando falls in line with the behaviour seen in Figure 12. The agent likes to conquer as many regions as possible, disregarding the fact that this can leave it very open to attacks from the opponent. Wealthy is the most greedy power possible and it is no surprise the agent chooses it often. It does not have the time to look more than its own turn into the future. This means that it has a hard time estimating if another power could give it more than seven victory coins. Given the low aggression of the agent Pillaging is a strange anomaly to see. Further review of this power reveals it tends to be picked only if it is available early in the game. At this time the neutral tokens are still in play as opposed to only empty regions.
The three least popular powers are: Fortified, Forest and Flying. Fortified gives one owned region per turn an extra defence which also gives one victory coin per turn. Forest gives one extra victory coin per forest region owned. Flying allows the player to conquer any region regardless if it borders one of theirs or not. A fortified combo starts with fewer tokens than most other combos. Combining this with the MCTS agent's complete lack of future planning this combo is not appealing to it. The Forest power requires the agent to conquer specific regions in order to use this bonus. This does not appear very difficult to accomplish, yet the agent still does not like the Forest power and powers

like it. The Flying power is an interesting one since this power increases the potential search space of the game. This means even more bad choices for the random playouts to try, for which the agent does not have time.
The three most popular races are: Tritons, Amazons and Ratmen. Tritons reduce the cost of conquering a region that borders a water region by one token, which holds true for ten regions in total. Amazons gain four extra tokens to conquer with, these extra tokens can not be used to redeploy and are put aside at the end of conquering. Ratmen do not have a special rule, they simply have more tokens from the start. All of these races follow the same simple principle: conquering more regions is always better.
The three least popular races are: Dwarves, Sorcerers and Humans. Dwarves gain one extra victory coin per mine that they own, which is one of the few special rules that still works when in decline. Sorcerers have a special way of conquering a region with one token in it once per turn. Humans gain one extra victory per farmland region that they own. Both Dwarves and Sorcerers require a significant amount of planning to use effectively. Dwarves also start with the least amount of tokens of any race due to their strong bonus. Humans appear to have the same problem for the MCTS agent as the Forest power. It does not appear to be able to prioritize regions based on their combo and is therefore less likely to pick that combo to begin with.

From the results we also selected the ten most popular combos to see if the agent is able to spot any potential synergy between a power and race choice, see Table 7.

Combo	Times chosen
Commando Tritons	21
Commando Amazons	21
Underworld Ratmen	17
Diplomat Ratmen	16
Wealthy Tritons	16
Spirit Amazons	16
Underworld Tritons	15
Wealthy Halflings	15
Wealthy Elves	15
Commando Halflings	14

Table 7: Most popular combos

All of these combos contain at least one of the most popular individually good powers or races. The agent does not appear to recognize anything special between combos, simply having a good power or race is what it looks for.
Overall we can determine the following from the behaviour of our agent. It has a very simple game plan: conquer as many regions as possible regardless of the consequences. Due to the random playouts of MCTS and the vast search space this is the only plan that it can come up with that will offer any reasonable playing strength. Any combos that are more suited to simply owning more regions is picked far more regularly regardless of the state of the game. Combos that are harder to play or require planning more than the current turn are essentially ignored. If a combo gives even one more starting token it is already more likely to be chosen. Even though we do not believe

that the MCTS agent knows exactly what a good move looks like it can still play Small World effectively. It attempts to play the game greedily through a less than ideal strategy. If at any point the combos it can play well with do not show up, the agent almost always loses. This signifies a failure to adapt to the situation as is required in many modern board games.

7 Conclusions and Further Research

In this thesis we implemented a Monte Carlo Tree Search (MCTS) agent for the board game Small World. In an attempt to counteract the high branching factor of this game we chose to split a turn into smaller actions and consider them individually in sequence. We evaluated the agent against other opponents and had a close look at its behaviour and playing strength. Even though MCTS offers a drastic improvement over random play we still deem this implementation only a partial success. The basic concept of conquering regions appears to be well suited to the MCTS approach. Adapting to which combos are available and how to play them effectively is handled rather poorly. Splitting up the turns did manage to deal with the branching factor of the game, yet the overall size of the search space still proved too much. Less favourable actions are quickly discarded without wasting too much time. We found a relatively high C_p value to offer more effective play when using this method. The agent does not have the ability to plan any long term sequences of play. With such a vast search tree both in depth and width it almost exclusively looks at just its own turn.
The primary issue with using MCTS in a games such as Small World is how an action is evaluated and rewarded through random play. More complex actions are completely overlooked favouring choices that are generally good to take. Instead of combos that would benefit the agent's current situation it will attempt to find one that is simpler and therefore more effective during the random playouts. In a way we have implemented a method of play that is very well suited to beginning players.
For future research we intend to find an effective method of evaluating a game state in Small World. Many of the complexities and long term goals are lost in random play. If such an evaluation can be found it would be interesting to test if splitting the turns into separate components is even required. Another path to follow would be to use a different tree policy than the UCT method. Random play could also be replaced as the default policy to test its effects.

References

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo Tree Search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[CBSS08] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-Carlo Tree Search: A new framework for game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217, 2008.

[Day] Days of Wonder. Publisher's website with details about the game. https://www.daysofwonder.com/. Last visited: 16-12-2020.

[HPA+18] S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang. Overview on Deepmind and its Alphago zero AI. In *Proceedings of the 2018 international conference on big data and education*, pages 67–71, 2018.

[Koz09] T. Kozelek. Methods of MCTS and the game Arimaa. 2009.

[Roe12] G. Roelofs. *Monte Carlo Tree Search in a modern board game framework*. BSc thesis, Maastricht University, 2012.

[SCS10] I. Szita, G. Chaslot, and P. Spronck. Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games*, pages 21–32. Springer, 2010.

[SS03] O. Syed and A. Syed. Arimaa-a new game designed to be difficult for computers. *ICGA Journal*, 26(2):138–139, 2003.

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[TC20] A. Theodoridis and G. Chalkiadakis. Monte Carlo Tree Search for the game of Diplomacy. In *11th Hellenic Conference on Artificial Intelligence*, SETN 2020, page 16–25. ACM, 2020.

[Wu11] D. J. Wu. Move Ranking and Evaluation in the game of Arimaa. PhD thesis, Harvard University, 2011.

[Wu15] D. J. Wu. Designing a winning Arimaa program. *ICGA Journal*, 38(1):19–40, 2015.