MONTE-CARLO TREE SEARCH IN THE DOMAIN OF CARCASSONNE

Yannick S. Müller

Master Thesis DKE 14-10

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Thesis committee:

Dr. Mark Winands
Dr. Marc Lanctot

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
July 2, 2014

Preface

This Master Thesis was written at the Department of Knowledge Engineering at Maastricht University. In this thesis I investigate the application of Monte-Carlo Tree Search in the board game Carcassonne.

I would like to thank the following people for making this thesis possible. First of all, I would like to thank my supervisor Dr. Marc Lanctot. He guided me through this project and provided lots of input and ideas during the past months. Further, I would like to thank Dr. Mark Winands for his valuable advice and for examining this thesis. I would also like to thank Mareike Mutz for finding bugs in my program, Jana Bittner, who owns almost all Carcassonne expansions for lending them to me and for answering detail questions about the game, and the members of the Filmstudio association in Aachen for testing and discussing my program and for playing many games of Carcassonne with me.

Finally, I would like to thank my family and friends for supporting me while I was writing this thesis.

Yannick Müller
Aachen, June 2014

Summary

Since computers exist, researchers have used them to play games. Classic board games have received much attention in the field of Artificial Intelligence. Modern board games that often contain chance elements and can be played with more than two players have not received as much attention, but there has been some research done recently. For example, The Settlers of Catan as one of the most successful euro games has received some attention.

This thesis studies the game Carcassonne, another successful euro game, and the possibility to use Monte-Carlo Tree Search in this domain. Carcassonne is a perfect information game with chance and can be played with 2 to 5 players. This thesis focuses on the two-player case. Since there are no free and applicable computer implementations of Carcassonne available, first the game itself was implemented. The implementation is called Carcasum and is freely available online.

Using the implemented game, AI players could be developed. The problem statement is: *How can a good MCTS player for Carcassonne be developed?* To answer this question, first the applicability of the Monte Carlo method in the domain of Carcassonne was verified by writing a flat Monte Carlo player, which performs reasonably.

Carcassonne's game mechanics are different from typical classic games, in which each player does one deterministic action each turn. In Carcassonne, moves consist of different types of actions, and there is a chance event each turn. Thus it had to be investigated how MCTS can be applied to this domain. MCTS was then compared to flat Monte Carlo. As expected, MCTS works better than flat Monte Carlo.

A doubling experiment was performed, where two MCTS players with a fixed simulation count play against each other, with one having double number of simulations of the other. The law of diminishing returns can be observed clearly in this application. The law states that adding increasingly more resources of one type leads to increasingly less reward.

To improve the performance of the MCTS player, different values for the UCT parameter C_p were tried; $C_p = 0.5$ seems to be a good value. Additionally, different approaches and enhancements, such as different reward functions and playout policies, were tested. To enhance the playout policy, early terminations were tested, but without success. Also, simple heuristic based players were developed, and for comparison, the advanced heuristic player from the open source Carcassonne implementation JCloisterZone was adapted. Both were shown to work reasonably well. The best simple heuristic based player was included into MCTS' playout policy using ε-Greedy and Roulette Wheel Selection. The ε-Greedy approach shows some success using a high randomness, the tested roulette wheel selection surprisingly decreased the quality quite much.

As additional MCTS enhancements node priors, progressive widening and progressive bias were tested, but also showed no success.

Contents

Chapter 1

Introduction

This chapter gives a brief overview of AI in games in general and the game of Carcassonne in particular, and states the problem statement and research questions of this thesis.

1.1 Games and AI

Games are one of the main research domains of Artificial Intelligence (AI). There are two main approaches. The first is the minimax algorithm that leads back to von Neumann who published the minimax theorem in 1928 (Von Neumann, 1928). Since then there have been many variants and improvements that produced strong AI players. For example the backgammon programm TD-Gammon achieved a strength slightly below top human players in 1992 (Tesauro, 1995), and in 1997 the chess computer Deep Blue won a six-game match against world champion Kasparov (Campbell, Hoane Jr., and Hsu, 2002).

The second approach is the Monte Carlo method respectively its extension Monte-Carlo Tree Search. The Monte Carlo method was invented in the 1940s by Stanislaw Ulam but did not receive much attention by AI researchers (Metropolis and Ulam, 1949). In 2006, Rémi Coulom published the technique Monte-Carlo Tree Search (Coulom, 2007). It gained much attention and many improvements were researched (Browne *et al.*, 2012). Today the most powerful Go programs use MCTS but still cannot compete with the best human Go players.

1.1.1 Minimax

The minimax theorem states that if in a two-player zero-sum game both players know how the other player plays there exists a mixed strategy for both players to maximize their own outcome. Based on this theorem, the minimax algorithm can be formulated. The idea is to select the action that leads to the maximum possible reward, assuming that the other player will always do the same, thus minimizing the first players reward. This way a search tree is build, considering all possible actions, where the root node maximizes the reward and all further nodes minimize and maximize the score alternatingly (see Figure 1.1). For most games the resulting game tree is much too large, so the tree is usually not built completely, but for instance limited to a fixed depth, and a heuristic function is used to rate a game state. See Algorithm 1 for a formal description of the minimax algorithm.

1.1.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a relatively recent approach in game AI (Kocsis and Szepesvári, 2006; Coulom, 2007). MCTS has gained much interest and success in the last years. For example, the currently strongest general game AIs and computer Go programs use MCTS. It was tested on other games and applications, for instance mixed integer programming, physics simulations, printer scheduling (Browne *et al.*, 2012), but there are still interesting research challenges.

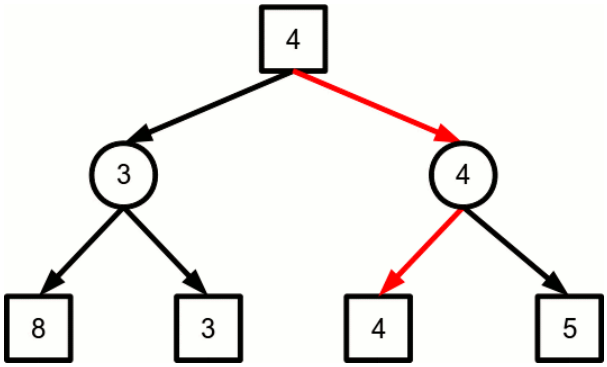

Figure 1.1: An example of a minimax tree. The value in a node denotes the reward, the square nodes are maximizing nodes, the round ones are minimizing nodes.

Algorithm 1 Minimax algorithm.

```
function MiniMax(state, depth, maximizingPlayer)
    if depth = 0 or state is terminal then
        return heuristic value of state
    if maximizingPlayer then
        score ← −∞
        for all child ∈ Successors(state) do
            value ← MiniMax(child, depth − 1, false)
            if value > score then
                score ← value
    else
        score ← +∞
        for all child ∈ Successors(state) do
            value ← MiniMax(child, depth − 1, true)
            if value < score then
                score ← value
    return score
```

1.2 Domain of Study: Carcassonne

Carcassonne is a modern tile placement board game that has become popular around the world. It is generally classified as a euro game, which describes tabletop games that typically have relatively simple rules, indirect player interaction, no player elimination and artistically designed materials (Woods, 2012). Carcassonne was translated to over 20 languages and has sold more than 6 million copies[1]. Since 2003 there are annual German championships, since 2006, there are annual international championships.

For Carcassonne there has been a Master Thesis in 2009 by Cathleen Heyden on the development of an AI player (Heyden, 2009). The work handled the two-player game only. The thesis mainly focused on classic search approaches and ended up with Star2.5, a variant of *-minimax (Ballard, 1983) to perform best. It was reported to play reasonably well and to be able to beat advanced human players sometimes, while reaching a search depth of 2. The work also had a quick look at Monte Carlo search and Monte-Carlo Tree Search, but did not have much success with it. A deeper look into the topic might give better results.

The multi-player case is probably too complex for classic game AI techniques, considering the game has a large branching factor and Heyden's work only reached a search depth of 2 with two players. MCTS might be able to handle multi-player games better than classic search approaches (Nijssen and Winands, 2013). There has been relatively little work done in multi-player MCTS and even fewer in games with chance elements. Thus an MCTS engine is worth researching, even though the multi-player case cannot be treated in this thesis.

1.3 Problem Statement and Research Questions

The goal of this thesis is to write a Carcassonne player that plays as good as possible, using MCTS. The problem statement is as follows:

- How can a good MCTS player for Carcassonne be developed?

The following research questions are addressed. MCTS is based on Monte Carlo playouts. It has been shown that Monte Carlo methods work in many different domains, however it is possible that it does not work in specific ones. Therefore we have to verify that Monte Carlo works in Carcassonne.

- Do Monte Carlo methods work in Carcassonne?

In order to get the best results, we can try to enhance MCTS. There are three main areas that can be tuned in MCTS. First, different reward functions can be tested:

- Which reward function works best for MCTS in Carcassonne?

Second, the plain Monte Carlo playout policy can be modified.

- How can the playout policy of MCTS in Carcassonne be enhanced?

Third, different modifications to the tree policy can be tested.

- How can the tree policy of MCTS in Carcassonne be enhanced?

In the past, many enhancements to the tree policy have been proven to work well in MCTS, some of them could be tested in this domain.

1.4 Thesis Outline

The outline of this thesis is as follows:

Chapter 1 gives an introduction into game AI in general and briefly introduces the main subjects of this thesis: Monte-Carlo Tree Search and Carcassonne. The chapter closes with the problem statement and the thesis outline.

[1] `http://www.hans-im-glueck.de/carcassonne/die-jubilaeumseditionen/carcassonne-das-jubilaeumsspiel/`, 2011, accessed June 6th 2014

Chapter 2 explains the game Carcassonne, discusses some of its strategic components and finally gives the state-space and game-tree complexity and compares them to other games.

Chapter 3 presents the Monte-Carlo Tree Search and introduces possible enhancements.

Chapter 4 explores how Monte-Carlo Tree Search and the introduced enhancements can be applied to Carcassonne.

Chapter 5 describes the experiments performed discusses and their results.

Chapter 6 gives the conclusion of this thesis and proposes topics for future research.

Chapter 2

Carcassonne

Carcassonne is a popular award winning board game published in 2000. It was designed by Klaus-Jürgen Wrede and was first published by Hans im Glück in Germany. Carcassonne is a perfect-information game with chance events. It has become popular around the world and spawned many expansions. This chapter explains Carcassonne's rules, some strategic elements and its complexity.

2.1 Rules

Carcassonne can be played with 2 to 5 players in the basic version. Together the players build one landscape using 72 square tiles (see appendix A.1). The tiles contain cloisters and different combinations of open field, road and city pieces. Each player has 7 figures called "meeples" in one color. A player's turn consists of two actions: First, he draws a random tile and places it on the board. Then, he can place a meeple on one of the elements of this tile.

The game starts with a predefined tile on the board (see Figure 2.1). All other tiles can only be placed with at least one edge connected to another tile on the board and all edges must match their surrounding tiles. The new tile can be rotated in all four orientations, before laying it down. Elements of the same type then become connected. For instance, two city pieces that have their open ends touching are then treated as parts of the same city.

After placing a tile, the player can then place a meeple on the tile that has just been placed. The meeple can only be placed on elements that do not already have another meeple on them. However, two different elements can be merged later by placing a connecting tile between them. If elements merge after meeples were placed they become one and keep all meeples that have been placed on them at some point. That way an element can have multiple meeples on it.

When an element is completed it is scored. The player who has the most meeples on the element gets the points, and all meeples placed on it are returned to the players. If multiple players have the highest number of meeples on an element all of those players get the points. Roads and cities are completed if they have no open ends left, cloisters are completed if they have 8 tiles surrounding them and fields cannot be completed.

The seating order, player colors and the starting player can be assigned arbitrarily. The game ends when there are no more tiles left. All remaining meeples on the board are scored. The player with the highest number of points wins the game.

Roads: Roads are worth the number of tiles that the road spans over.

Cities: Cities are worth the number of tiles that the city spans over plus the number of pennants it has. If the city is completed (i.e. mid-game) and if it contains of more than two tiles, the score is doubled.

Cloisters: Cloisters are worth the number of tiles surrounding them, including themselves, so their value is at least 2 and at most 9.

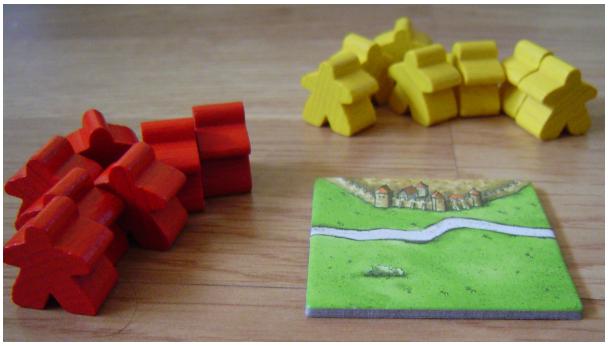

Figure 2.1: Meeples and the starting tile with one city, one road and two fields (image from Heyden, 2009).

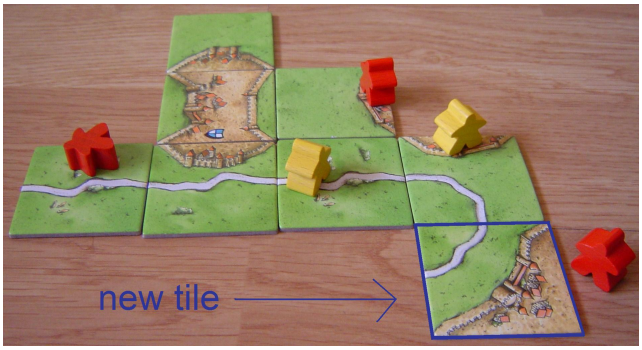

Figure 2.2: Red placed a new tile. The tile has two fields, one road and one city. The road and the bigger field are already occupied, thus Red has three options for placing a meeple: On the smaller field, on the city or not at all (image from Heyden, 2009).

Figure 2.3: The new tile connects two separate cities, so they become one, and both players have one meeple in it (image from Heyden, 2009).

Fields: A field is worth the number of completed cities it is connected to, times three.

Detailed rules can be found online[1].

2.2 Strategy

This section gives a brief overview about four possible strategic considerations during the game. The goal of the game is to score more points than the opponents. This can be reached by either scoring many points or by preventing others from doing so.

2.2.1 Gathering Points

Focusing on building up cities, streets or fields to get points is the obvious strategy that is dominant most of the time. However, building too many different elements is also not a good idea, since the number of meeples is limited. A meeple is stuck on the board until the element is completed or the game ends. Thus it is often better to extend existing elements than building new ones.

2.2.2 Stealing Elements

Another strategy often used is to try to "steal" elements. Therefore one tries to get into already existing larger elements. Large elements are rarely unoccupied, so a player has to try to get at least the same number of meeples into an element as other players. Getting the same number of meeples into it gives the same number of points for the different players, getting more meeples than all others gives points to only the one with the most meeples. Getting into occupied elements can only be done indirectly – since occupying already occupied elements is not allowed – by placing a new element of the same type close to the occupied element and connecting it later using another tile (see Figure 2.2 and Figure 2.3).

If possible it can be useful to lay the tiles in such a way that only specific tiles fit in between to force the connection of the elements. However, doing this might cause the other player not to place a fitting tile, to not give away points. Then again, not doing it will result in the other player trying to exclude the "thief". As long as the elements are not connected, the thief does not get the points from the large element.

The stealing strategy is usually only useful if the effort (i.e. the value of tiles required) of getting into the element is less than the value of the existing element, or if the thief gets more meeples in the element than other players. Otherwise the other player gains a larger profit than the stealing player. However, in multi-player games this can still be useful of course, for instance when trying to get closer to the leading player or to increase the own lead, while only giving points to the last player.

Consider Figure 2.4 and Figure 2.5 as an example. There are two players, Red and Blue; Red is the starting player. Red drew a tile with two city elements. Red now has different options. For instance, first: Connect it to its own city so that it only has one open edge and is easy to close. Closing cities is the only source of doubled points, so this is a solid option. Second: Create a new city and occupy it. But since Red already has a city, it is better to expand the existing city. This can also be combined with the first option, so Red could extend the city and in the same move occupy the second new city. Third: Red chose to do something different instead. By placing the tile next to the cloister and occupying the southern city part, it reduces the number of tiles that fit in between Blue's city and the newly placed tile. Only two fitting tiles are left (compare appendix A.1), and they both connect the two cities. If Red draws one of those tiles, it places it in the gap and connects to Blue's city. If Blue draws one of the tiles, chances are that it also places it in the gap, because otherwise its meeple gets stuck. However, in this two-player case, Blue might as well place the tile somewhere else, because Red's meeple also gets stuck, so both players have one meeple stuck, but only Blue gets more points for this city in the end.

Blue happens to draw a "tube city" tile in the next move (see Figure 2.5). Blue can now place the tile somewhere else and either give up the city, or hope the tile that fits in between the two city parts will be drawn and placed in the gap. It can also extend its city, as the first option in Figure 2.5 shows. This blocks the gap completely, as there is no fitting tile left anymore. In this case Blue and Red both have one meeple stuck, and the cities can never be completed, but Blue gets 7 points for the city in the

[1] Official English rules: `http://zmangames.com/rulebooks/Carcassonne.pdf`,
Official German rules: `http://www.hans-im-glueck.de/fileadmin/data_archive/Regeln/CarcassonneRegel.pdf`

end and Red only gets 1. Blue can also try to get a second meeple into the city, by placing the tile east of Red's tile, as shown in the right option in Figure 2.5. This has the advantage that Blue can get all the points for the city, while still being able to complete it. The disadvantage though is, that now a second gap needs to be closed, for which there are only four tiles possible left, assuming the first gap was already closed. For the first gap there are still only two tiles left, but Red will not place them, because if the city is completed, Blue obtains all the points, so Blue needs to get one of them. Also the way the tiles lie on the board, this offers some good opportunities for Red to also get a second meeple in, for instance by placing and occupying a tile of type N (see appendix A.1) at the north-east of the cloister.

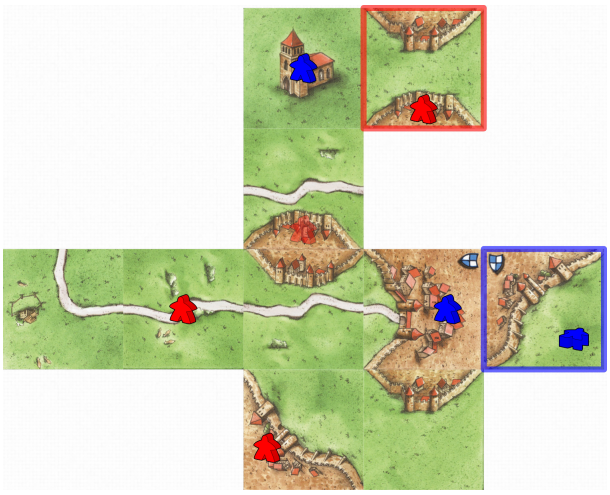

Figure 2.4: Red placed the framed tile and tries to get into Blue's city.

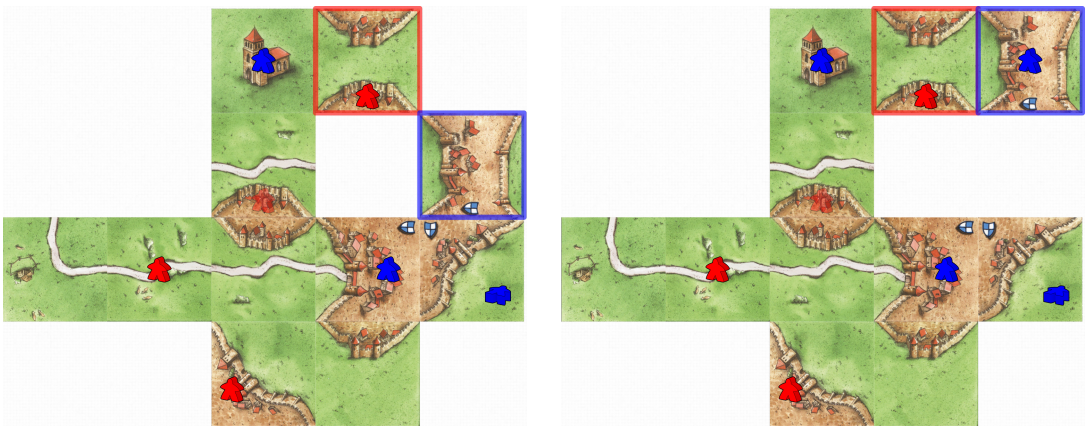

Figure 2.5: Two possible responses by Blue.

2.2.3 Blocking Opponents

Placing tiles in such a way that there are no or few remaining tiles that can fit a nearby position on the board can be useful to block other players, so that they cannot complete an element anymore, or that it is less likely. This can have two effects: First, not extending an element means fewer points, not closing a city means no score doubling. Second, the blocked player effectively looses a meeple. Not closing an element means none of the meeples on it will be returned. Having fewer free meeples means being able to occupy and score less.

2.2.4 Fields

At the end of the game fields score 3 points per completed city connected to it. Often there is only one large field that most cities are connected to, so occupying a field can be quite beneficial. However, fields only get scored at the end, so any meeple placed on a field is stuck there until the end. Thus, placing a meeple on a field means the player has one meeple less to use throughout the rest of the game for gaining points through other elements. If another player gets more meeples on the same field, it is also worthless, and effectively a wasted move and a wasted meeple. Therefore one might want to try not to occupy a field too early in the game. Additionally, occupying a field too early also makes it difficult to know which field will be large, and which will not. However, occupying fields early in the game gives an advantage over other players that then have to try to steal the field to get profit from it.

In the very end of the game it sometimes makes sense to place a tile with a field next to a completed city, so that the field does not connect to anything else, and occupy the field. This can give 3 points, which may be more than any other move.

2.3 Rule Changes

Since Carcassonne was first published in 2000, the official game rules were modified several times.

Cities. In the original rules completed cities are worth 2 points per tile and pennant, except for when the city only consists of 2 tiles, in which case the city is worth only 2 points. The exception for those tiny cities has been dropped in 2002, so currently all completed cities are worth 2 points per tile and pennant.

Fields. Originally, each city was worth 4 points for the players with the greatest number of meeples lying on any field connecting to the city. This means that every city is scored only once. As a consequence, two meeples lying on different fields can compete or collaborate for the same city, depending if they belong to the same player.

In first change the number of points per city was reduced to 3. The second change modified the way the points are calculated. Every field is now considered independently. Two meeples on two different fields connected to the same city can both get points for the city.

Rules Used. For this thesis the current version of field scoring and the tiny city exception (tiny cities score 2 points) are used.

2.4 Expansions

Carcassonne has spawned many expansions, these bring more players, new tiles, new pieces, special features and other gimmicks to the game. Currently, there are 9 expansions and more than 20 so called “mini”-expansions. Additionally there are alternative and special editions, as well as unofficial fan expansions. There is a notable official expansion, “Sheep and Hills” as it turns Carcassonne into an imperfect-information game by placing tiles face down and covered by others on the table. This thesis only covers the base game without expansions.

2.5 Complexity

This section briefly describes the complexity of two-player Carcassonne. First, the state-space complexity and the game-tree complexity are described, then a comparison to other games is given.

2.5.1 State-Space Complexity

The state-space complexity of a game is the total number of valid game states that can be reached during any game. The value is very complex to calculate, therefore usually only a lower bound is given. Heyden (2009) gives a lower bound of $5 \cdot 10^{40}$ and states it to be highly underestimated. This value is an estimation of the number of possible boards that can be created using 72 tiles, not taking into account the possibility of placing meeples on them.

2.5.2 Game-Tree Complexity

The game-tree complexity is the number of leaf nodes the game tree has from the initial game position, thus the total number of possible games that can be played. Heyden (2009) calculates this number to be $C = 55^{71} \cdot \prod_{i=2}^{24} i^3 \approx 8.8 \cdot 10^{194}$. In this formula 55 is the average branching factor, 71 is the number of plies per game, 24 is the number of different tiles and the product over i^3 the average branching factor of chance events.

2.5.3 Comparison to Other Games

Carcassonne has a state-space complexity of at least 10^{40} and a game-tree complexity of 10^{194}. For comparison, Chess has a state-space complexity of 10^{46} and a game-tree complexity of 10^{123}, Go has a state-space complexity of 10^{172} and a game tree-complexity of 10^{360}.

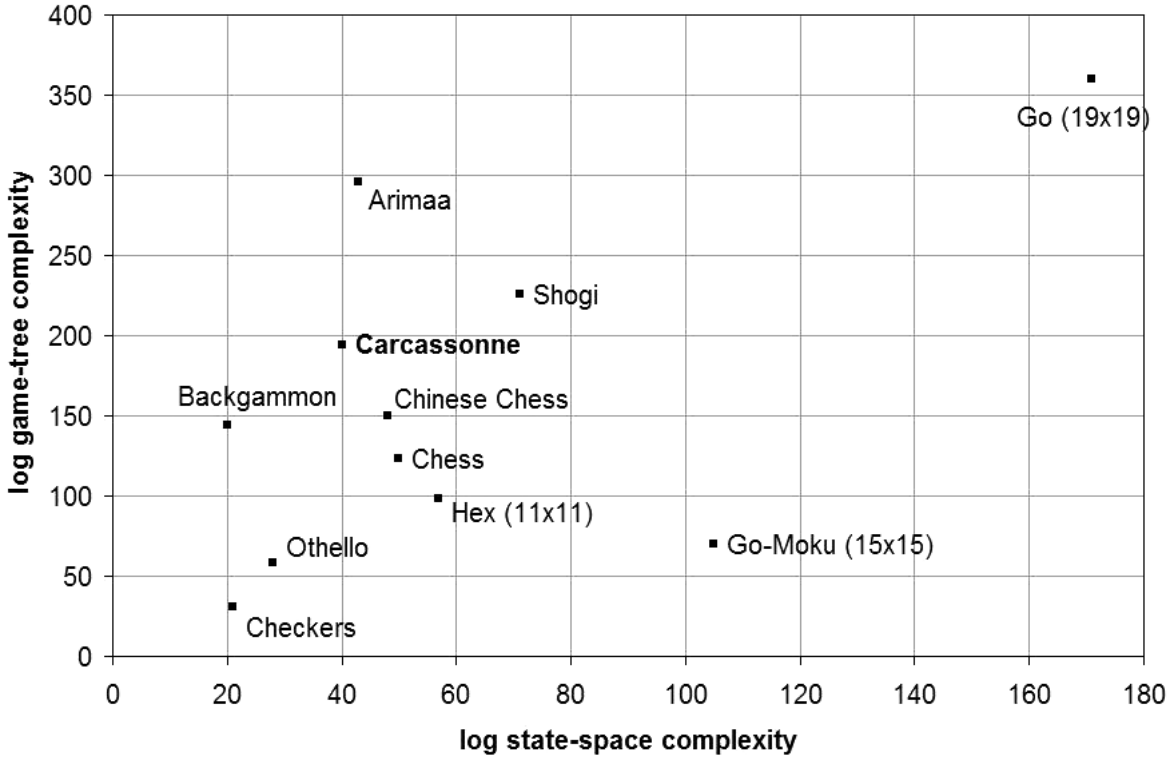

Figure 2.6: The complexities of different games (image from Heyden, 2009).

Figure 2.6 compares the complexity of Carcassonne to several other games. Carcassonne has a higher game-tree complexity than many other games that have been researched in the past, for instance Hex, Chess or Chinese Chess. In the figure, Carcassonne is plotted with a state-space complexity of 10^{40}, however the real value is probably much higher. Heyden (2009) assumes it is at least equal to Arimaa.

Chapter 3

Monte-Carlo Tree Search

This chapter explains Monte-Carlo Tree Search. First the foundation, the Monte Carlo method, is explained, then MCTS, UCT, and several enhancements are described.

3.1 Flat Monte Carlo

Monte Carlo search is a simple search algorithm that requires little to no domain knowledge. When deciding which move to play the algorithm tries every possible move and plays a completely random game until the end. This process is repeated a fixed number of times or until the computation time is over. In the end, the move with the highest winning rate is chosen.

3.2 Monte-Carlo Tree Search

The Monte-Carlo Tree Search (MCTS) extends this approach by building a tree in memory, storing statistics of wins/losses for each move that was tried in those nodes (Coulom, 2007). Contrary to classic search approaches, the tree is not built deterministically and does not use a fixed depth. Instead, in every iteration a new sequence of moves is chosen and a new node is added to the tree. The newly added node is then simulated in Monte Carlo manner and the result is stored in the tree node. When the time is over, the move with the most promising child of the root node is chosen and played.

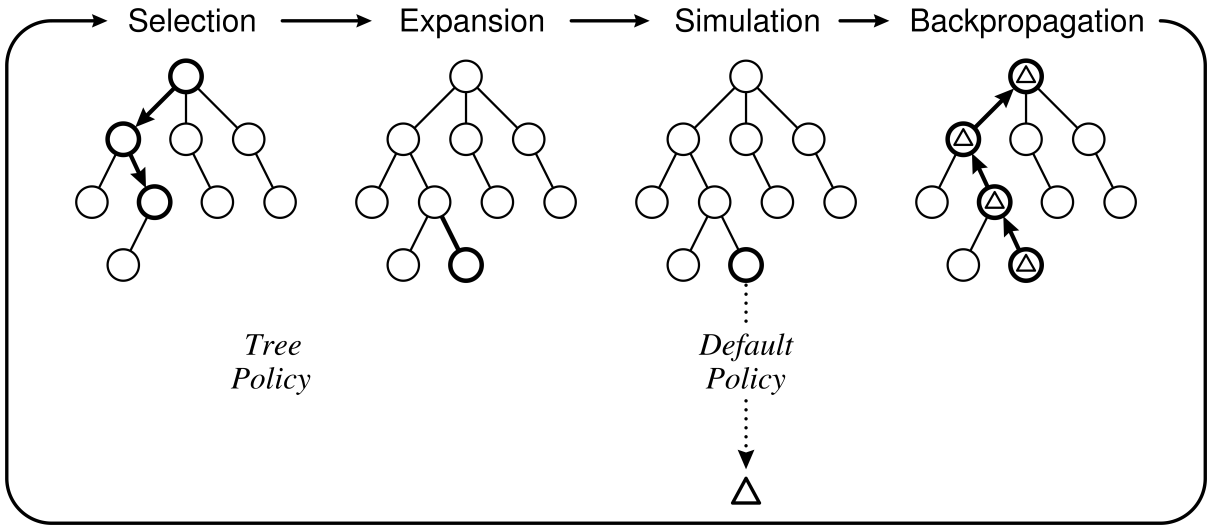

Figure 3.1: General MCTS approach (Browne *et al.*, 2012).

Usually, for a good performance using Monte Carlo methods, a large number of simulations per state is needed. MCTS tries to overcome this by storing statistics in nodes and guiding the search according to the statistics so far.

Algorithm 2 describes MCTS search formally. It can be divided into four steps that are repeatedly executed until time is up (Chaslot *et al.*, 2008):

Selection. Starting from the root node, a selection strategy (tree policy) is applied recursively to find the most urgent node.

Expansion. A new node, representing a possible action, is added to the selected node.

Simulation. From the newly created node, the game is played according to a default policy. The default policy is usually random but can also be informed.

Backpropagation. The statistics of the created node and all its parents are updated with the result of the playout.

Algorithm 2 General MCTS approach as described by (Browne *et al.*, 2012).

function MctsSearch(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))

3.3 Upper Confidence Bounds for Trees

There are different ways to implement the tree policy in the selection step in MCTS to balance exploration and exploitation. Upper Confidence Bound for Trees (UCT, Kocsis and Szepesvári, 2006) is a variant of MCTS that uses a tree policy called UCB1 (Auer, Cesa-Bianchi, and Fischer, 2002). This policy selects the child node that maximizes the following term:

$$UCB_j = \overline{X}_j + C_p\sqrt{\frac{\ln n}{n_j}}$$

where n is the number of times the node has been visited, n_j is the number of times the j^{th} child node has been visited, $\overline{X}_j$ is the average utility for the child and C_p is a constant. UCT has been proven to work well in many applications (Browne *et al.*, 2012).

Algorithm 3 describes the UCT in detail. In the notation a is an action, s a state, $A(s)$ the action state of s, and $f(s, a)$ the state transition from s using a. $Q(v)$ denotes the reward value of a node and $N(v)$ is the visit count of a node. Δ denotes a reward vector; there is also a negamax variant of UCT, where Δ is a scalar, and the Backup function is modified.

3.4 Enhancements

This section introduces five possible enhancements for MCTS: Tree Re-Usage, Node Priors, Progressive Widening, Progressive Bias and Early Terminations.

3.4.1 Tree Re-Usage

Usually, every turn a new search is started and thus a new MCTS tree is build. However, since the other player is probably also trying to make a good move and the nature of MCTS, the corresponding game tree, might have already been explored in a previous search. As a consequence, instead of starting a new search, and therefore a new tree every turn, the last tree can be reused by making the child node corresponding to the current situation the new root node (Steinhauer, 2010).

Algorithm 3 The UCT algorithm as described by Browne *et al.* (2012).

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($v_0, 0$))

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

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

function BestChild(v, c)
  return $\displaystyle\arg\max_{v' \in \text{children of } v} \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$

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(v, 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

3.4.2 Node Priors

Node priors are a way to include heuristic knowledge in MCTS (Gelly and Silver, 2007). The method uses virtual playouts that are added to MCTS nodes. Usually, a node is initialized with 0 playouts. Instead in this method the value is set to a constant and the statistics are initialized according to a heuristic function, such that $\frac{Q(v)}{N(v)} = H(s(v))$, where $H \in [0,1]$ is the heuristic function.

Algorithm 4 BestChild adapted for node priors.

function BESTCHILDNODEPRIORS(v, c)

$$\textbf{return} \quad \underset{v' \in \text{children of } v}{\arg\max} \quad \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N_{parent}(v)}{N(v')}}$$

$$\text{where } N_{parent}(v) = 1 + \textstyle\sum_{v' \in \text{children of } v} N(v')$$

There are more playouts stored in a node than actually happened, and the virtual playouts are not back propagated, so the tree is in an inconsistent state. Therefore the UCB Formula in UCT's BestChild function (see Algorithm 3) is adjusted as shown in Algorithm 4.

3.4.3 Progressive Widening

For progressive widening (Chaslot *et al.*, 2008) a move ordering is required. In a node the possible moves are ordered. Only the best moves are taken into account, the others are pruned. When more time becomes available, more moves are taken into account slowly. This way the search is guided in a certain direction and the search becomes more deep, at the cost of ignoring some moves at the beginning.

This technique can be implemented for instance by using a visit count threshold that depends on the number of explored children.

3.4.4 Progressive Bias

Progressive bias is a way to include heuristic knowledge into the UCB formula (Chaslot *et al.*, 2008):

$$UCB = \overline{X}_j + C_p\sqrt{\frac{\ln n}{n_j}} + f(n_j)$$

$$f(n_j) = \frac{H_j}{n_j + 1}$$

Where H_i is a coefficient representing heuristic knowledge for the game state in node n_j. In the beginning of a search, $f(n_j)$ has a greater influence on the term, while it becomes progressively more irrelevant when the node has been explored more.

3.4.5 Early Terminations

Sometimes it can be sufficient to not simulate the complete game, but to only simulate a certain number of turns. For that some evaluation function is needed to tell MCTS how good the state is. This has the advantage that the time for long games or games with no fixed limit can be reduced and therefore more total playouts can be simulated. This technique shows success in Amazons, Breakthrough and Mancala (Lorentz, 2008; Lorentz and Horey, 2013; Ramanujan and Selman, 2011; Lanctot *et al.*, 2014a).

As a variation, dynamic early terminations do not use a fixed-depth playout but use an evaluation function or other domain knowledge to test whether to terminate the simulation. This method has been tested successfully in Go, Lines of Actions and Breakthrough (Bouzy, 2007; Winands, Björnsson, and Saito, 2008; Lanctot *et al.*, 2014a).

Chapter 4

MCTS in Carcassonne

This chapter addresses how MCTS can be adapted to Carcassonne and discusses some changes and variations that can be used in different places.

Adapting MCTS to Carcassonne is not a trivial task. MCTS is usually applied in perfect information two-player games with simple actions and without chance. Carcassonne is a multi-player game with a chance element and its turns consist of three actions. Though the multi-player version of Carcassonne is not covered by this thesis, the MCTS adaption should be capable of handling multi-player games, or be easily extendable in the future.

4.1 Turns

Turns in Carcassonne consist of three actions:

1. Drawing a random tile.
2. Placing the tile on the board.
3. Optionally place a meeple on the newly placed tile.

It is hard to combine them into one meta action, because the possible meeple actions are only known after the tile action has been executed. Instead they are modeled as three separate actions that follow each other and are executed by the same player. Also it is generally a good idea to model chance events in separate chance nodes in an MCTS tree. Figure 4.1 visualizes the resulting tree. Note that in this tree a chance node is followed by two max nodes.

4.2 Chance

Carcassonne contains chance elements. A chance element has to be represented in its own nodes in the MCTS tree. A chance node cannot be treated the same way other nodes are treated. Chance nodes cannot try to maximize or minimize the score for a player, they always draw a tile randomly.

4.3 Result

Applying both changes to UCT results in Algorithm 5. The main differences lie in the tree policy that has an exception for chance nodes, and an extra expand function for chance nodes. Additionally, the search has to select two actions instead of one in the main UctSearch function.

Algorithm 5 UCT for Carcassonne.

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(v_l)
    Backup(v_l, Δ)
  $v_{tile} \leftarrow$ BestChild0(v_0)
  $v_{meeple} \leftarrow$ BestChild0(v_{tile})
  return $a(v_{tile}), a(v_{meeple})$

function TreePolicy(v)
  while v is nonterminal **do**
    if v is chance node **then**
      $a \leftarrow$ select action according to distribution of remaining tiles
      if child of v with action a does not exist **then**
        return ExpandChance(v, a)
      else
        $v \leftarrow$ child of v with action a
    else
      if v not fully expanded **then**
        return Expand(v)
      else
        $v \leftarrow$ BestChild(v)
  return v

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

function ExpandChance(v, a)
  add new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BestChild(v)
  return $\underset{v' \in \text{children of } v}{\arg\max} \dfrac{Q(v')}{N(v')} + C_p\sqrt{\dfrac{\ln N(v)}{N(v')}}$

function BestChild0(v)
  return $\underset{v' \in \text{children of } v}{\arg\max} \dfrac{Q(v')}{N(v')}$

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

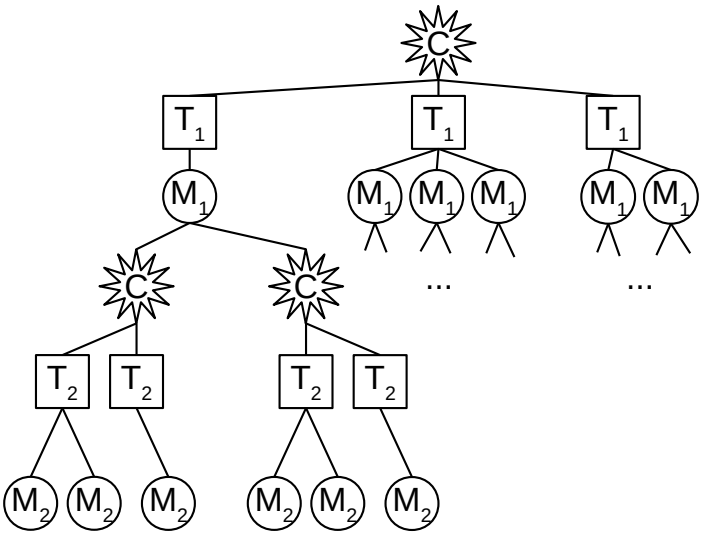

Figure 4.1: A part of the MCTS tree in Carcassonne. C is a chance node (draw a tile), T_i a tile move node for player i (place the tile on the board), and M_i a meeple move node for player i (place a meeple on the tile).

4.4 Reward Function

The default policy returns a reward vector. The vector contains one number as reward per player. There are many different possibilities to calculate the reward. This section introduces the reward functions that were tested: Simple, Score Difference, Score Portion and Heyden's Evaluation. Additionally, meta functions to modify them, such as Normalization, Bonus and the case of Early Terminations are discussed.

4.4.1 Simple

The most basic way of a reward function is to assign 1 for a win for a player, -1 for a loss and 0 for a draw. This function however contains no information about by how much a game was won or lost. An AI might try to maximize its win to make it harder for the opponents to catch up.

$$Q_{Simple} = \begin{cases} +1, & \text{win} \\ 0, & \text{draw} \\ -1, & \text{loss} \end{cases}$$

4.4.2 Score Difference

A simple way to take the win or loss margin into account is to use the difference of the scores of the players, by taking the own score and subtracting the opponents score from it.

$$Q_{ScoreDiff} = S_{my} - S_{opp}$$

S_{player} denotes the score of *player*. S_{my} is the own score, S_{opp} the opponent's score.

In two-player games, this reward function is still centered around zero, like the simple reward function: A loss is negative, a win positive and a draw is 0. Heyden (2009) reported good results for flat Monte Carlo using this reward function compared to the simple reward function. This could be confirmed, but it could not recreated with similar results.

4.4.3 Score Portion

An alternative to the simple score difference is the score portion. To calculate this reward function, the own score is divided by the sum of the scores of all players.

$$Q_{ScorePort} = \frac{S_{my}}{S_{my} + S_{opp}}$$

This score is ranged in $[0, 1]$. For two players a draw is 0.5, a loss is < 0.5 and a win is > 0.5.

4.4.4 Heyden's Evaluation

Heyden (2009) defines a heuristic evaluation function for Carcassonne and uses it in the expectimax framework. This evaluation function is called Heyden's Evaluation in this thesis. The evaluation function can also be used as a reward function in MCTS.

Heyden's Evaluation uses different features and weights them:

100 × Score Difference: The simple score difference.

10 × Meeple Difference: The difference of the number of meeples a player has left (i.e. currently not placed on the board)

10 × Incomplete Score Difference: The difference of the score of elements controlled by a player, except cities.

12.5 × Incomplete City Score Difference: The difference of the score of the cities controlled by a player, not doubled.

-40 × Fields with less than 3 cities: The number of fields controlled by the player that have less than 3 cities.

4.4.5 Normalization

For UCT reward values should be in the bounds of [-1, 1]. Some of the reward functions introduced, for example the simple score difference, do probably not work well, since they can easily have a value of over a hundred. As a consequence, they need to be scaled to work with UCT. This can be done by calculating or estimate bounds for a reward function. Therefore it can be helpful to know the maximum possible score in Carcassonne.

Maximum Possible Score

The maximum possible score is not easy to compute, but an upper bound can be given by assuming a player gets all cloisters, all city parts and completes all of them, all roads and all fields with cities.

- There are 6 tiles with cloisters on them. Each scores 9 points
- In total there are 49 city parts on all tiles. Each scores 2 points
- In total there are 10 pennants on all tiles. Each scores 2 points
- In total there are 28 tiny city parts (cities with one edge) on all tiles. Two of them can create a city that scores 3 points at a field.
- In total there are 10 triangular cities on all tiles. Four of them can create a city and score 3 points at a field
- In total there are 62 Road parts on all tiles. Each scores 1 point.

This way an upper score bound can be given: $6 \cdot 9 + 49 \cdot 2 + 10 \cdot 2 + \lfloor 28/2 \rfloor \cdot 3 + \lfloor 10/4 \rfloor \cdot 3 + 62 \cdot 1 = 282$. This upper bound can be used when computing boundaries for reward functions. However, it is impossible to get all of these possible points combined, and getting all possible elements is highly unlikely. Thus this value is very overestimated, observations confirm this.

Normalization in [0, 1]

A reward function Q_{other} can be normalized to be in the range $[0, 1]$ in the following way:

$$Q_N = \frac{Q_{other} - LowerBound(Q_{other})}{UpperBound(Q_{other}) - LowerBound(Q_{other})}$$

Normalization in [-0.5, +0.5]

A reward function Q_{other} can be normalized to be in the range $[-0.5, +0.5]$ in the following way:

$$Q_{N2} = \frac{Q_{other} - LowerBound(Q_{other})}{UpperBound(Q_{other}) - LowerBound(Q_{other})} - 0.5$$

4.4.6 Early Terminations

If early terminations are used, there should be a way for UCT to differentiate between an actual terminal state and an early termination. This can be achieved by using the value of a reward function Q_{other} in case of an early termination and its upper or lower bound to indicate wins and losses in terminal states.

$$Q_{ET} = \begin{cases} UpperBound(Q_{other}), & \text{terminal win} \\ LowerBound(Q_{other}), & \text{terminal loss} \\ Q_{other}, & \text{else} \end{cases}$$

4.4.7 Bonus

Reward functions give UCT hints which options are more promising than others. The greater the return values are the more UCT is "pushed" to one direction. The strongest hints are given by the Simple reward function, because it returns -1 and $+1$, while other normalized functions give weaker hints, but instead they offer more diversity by being able to determine between clear and narrow results. To combine the advantages of both, Bonus reward can be used. It mixes the Simple reward function with another arbitrary reward function Q_{other}.

$$Q_{Bonus} = Q_{Simple} + \omega$$

where

$$-1 \ll \omega = Q_{other}/c \ll +1$$

Q_{other} has to be centered around 0 and be normalized using a constant c such that it is never close to -1 or $+1$, thus only adding a hint to the -1 or $+1$ from Q_{simple}. A similar approach has been shown to work by Pepels *et al.* (2014).

Bonus reward can also be used with early terminations. For this, a normalized reward function Q_{other} is used which does not exceed the range $[-1, 1]$. In case of an early termination the reward function Q_{other} is returned, in case of a terminal state Q_{Bonus} is returned.

$$Q_{BonusET} = \begin{cases} Q_{Bonus}, & \text{terminal state} \\ Q_{other}, & \text{cutoff} \end{cases}$$

4.4.8 Complex Utility

Complex Utility 1 and Complex Utility 2 were originally designed for the multi-player game, and could not be evaluated in this thesis. They take the ranking of players into account and try to optimize their own ranking. Complex Utility 1 does this by multiplying the score of each player with its squared inverse rank, where "inverse rank" means player count − rank + 1, and then subtracting the results for other players from the result for itself. Complex Utility 2 works similarly, but modifies the score not by multiplying with the inverse rank, but by adding inverse rank × maximum possible score to it.

4.5 Playout Policy

When MCTS reaches a non-explored node, it simulates a game. There are different ways to do so.

4.5.1 Random

The most basic and also one of the fastest ways to simulate a game is to choose random actions in every turn until the game is finished.

4.5.2 Early Termination

As described in Subscetion 3.4.5, early terminations do a random playout but simulate only a limited number of turns. The resulting game state is then rated.

4.5.3 ε-Greedy

The ε-greedy approach (Sturtevant, 2008) requires a move ordering. When a game is simulated, instead of choosing a random move every time, a random move is chosen with a probability of ε, and the best move according to the selected move ordering is chosen with a probability of $1 - \varepsilon$. This way the quality of a playout can be improved while still taking other possibilities into account.

4.5.4 Roulette Wheel Selection

Roulette wheel selection is a mechanism for sampling from a non-uniform discrete distribution. Applied to MCTS this means heuristically better rated actions are tried more often than others.

For roulette wheel selection a move rating is needed. The roulette wheel selection creates a roulette wheel with all possible moves. A move is as wide as the move rating of the move is. Then a random move, according to the roulette wheel, is chosen and played. That way moves that are rated better are selected with a higher probability than the ones rated worse.

4.5.5 1-Ply Search

The 1-Ply Search playout policy uses a 1-Ply Search to decide which move to play. It simulates every possible move and chooses the best one, according to some evaluation, at every turn. This slows down a playout, but increases the quality of the playout. 1-Ply Search is usually combined with ε-greedy or roulette wheel selection, since otherwise there would be no randomness in the playout. 1-Ply Search playout policy is not evaluated in this thesis, because early tests indicated that the playout is slowed down drastically, and the advantage of better playouts does not come into effect and so the quality of the player is influenced quite negatively.

Chapter 5

Experiments and Results

This chapter presents a selection of experiments and their results that were performed for the development of Carcasum.

5.1 AI Players

For this thesis, a number of different players for Carcassonne were implemented. This section introduces some of them: Random Player, Monte Carlo Player, Monte Carlo Player 2, 1-Ply Search Using UCB, MCTS Player and Simple Players.

5.1.1 Random Player

Random Player implements a random Carcassonne player that chooses uniformly random moves.

5.1.2 Monte Carlo Player

Monte Carlo Player implements flat Monte Carlo. A move in Carcassonne consists of three actions: one chance action and two in which the player can decide: a tile placement move and a meeple placement move. The game implementation first asks a player for a tile move, executes it and then asks the player for a meeple move. Consequently, Monte Carlo Player executes two separate searches for both actions. When it is asked for a tile move, it tries every possible move a number of times and chooses the one with the best average reward and forgets everything about the search. When it is asked for a meeple move it, starts a new search, tries every possible move a number of times and chooses the one with the best average reward. Since a player has a fixed time to think, Monte Carlo Player divides the time between the two searches. A tile move usually has much more possibilities, so the tile move search gets 2/3 of the available time, the meeple move search gets the remaining time. However, sometimes no meeple placement is possible, since either all elements are already occupied, or because the player has no free meeples left. In that case the player is not asked for a meeple move and Monte Carlo Player does not use the remaining third of its time.

Monte Carlo Player extends the pure Monte Carlo idea by supporting the usage of different reward functions and playout policies.

5.1.3 Monte Carlo Player 2

Monte Carlo Player 2 is an alternate implementation of flat Monte Carlo. Monte Carlo Player has two problems: First, it does not always use all of the available time. Second, two separate searches do not seem to be optimal. For instance it could be possible that one tile placement in combination with a certain meeple placement is a very good move, but all other meeple placements are very bad ones. In this case the average reward for the tile placement would not be very good, and it would maybe not be placed there in the first place, so the meeple search does not get the ability to play the good meeple move.

Monte Carlo Player 2 tries to overcome both problems. It executes only one search. The search is done when it is asked for a tile move and uses the complete available time. Prior to the search it is not known which meeple placements will be possible. The moves become available by simulating a tile move. Since the simulation game is played out anyway, the possible moves become available. Once Monte Carlo Player 2 has gained all possible combinations of tile and meeple moves, it repeatedly tries out a random combination, until the time runs out. The best combination of tile and meeple move is then chosen. The meeple move is memorized and the tile move is returned. When the player is asked for a meeple move, the memorized move is returned.

Monte Carlo Player 2 also supports the usage of different reward functions and playout policies.

5.1.4 1-Ply Search Using UCB

One-Ply Search using UCB is a variant of Flat Monte Carlo that uses UCB to select which move to simulate, instead of choosing uniformly. When time runs out, still the move with the best win ratio is chosen. 1-Ply Search using UCB is implemented as a modification of Monte Carlo Player 2. Instead of selecting a random move combination every time, it first selects a tile move according to the UCB selection and then it chooses a meeple move according to the UCB selection.

5.1.5 MCTS Player

MCTS Player implements UCT according to Algorithm 5. It supports the usage of different reward functions and playout policies. It also offers options to enable different enhancements like tree re-usage, node priors, progressive widening and progressive bias.

5.1.6 Simple Players

To be able to enhance the playout policy in MCTS, different very simple players were written. The goal was to provide a playout, and thus a player, that plays better than random while being fast, so that it could be used within MCTS.

Applying and un-applying moves is time expensive, but the goal of the simple players it to be fast. Simple Player v1–3 therefore never simulate moves, but rate possible actions only by considering the current game state.

Simple Player v1

Simple Player v1 is only mentioned for the sake of completeness. The basic idea it was based on turned out to be wrong and also does not take fields into account. Thus it often generates invalid moves. As a workaround, whenever the move would be invalid, a random move is returned instead.

However, it still beats Random Player in 9999 out of 10000 games. It also beat the early versions of Simple Player v2 more often than it lost against them.

Simple Player v1 uses the same rule as Simple Player v2 does, if a cloister is drawn (see below). If the drawn tile contains no cloister, Simple Player v1 iterates over all possible tile placements and checks all already placed adjacent tiles. It then checks which roads and cities of the drawn tile and the adjacent tile would connect, and checks how many points that would give to which player and what would happened if the player occupied the element. This is the essential problem. By considering only two tiles at a time, Simple Player v1 ignores the fact that the same element could be connected to another tile at the same time. This behavior results in choosing invalid moves, because it does not see an element is already occupied sometimes. Simple Player v1 also never occupies a new element on the tile, because a new element that is not connected to another tile is therefore never taken into account. It does however occupy elements that are extended by a placement, it also extends elements it already owns instead creating or occupying new ones.

Simple Player v2

Player v2 simulates moves, but only considers the current game state to be fast. For placing a tile this can be enough, for placing a meeple it tries to guess which meeple moves will be valid after the tile would have been placed.

Simple Player v2 uses a fixed rule if the drawn tile has a cloister. In that case it iterates through all possible tile positions and counts the surrounding tiles, which score points for cloisters. If the player has a free meeple left, the tile is placed at the position with the most surrounding tiles and a meeple is placed on it. Cloisters can only be occupied when placing them, therefore the meeple placement is always possible, if the player has free meeples, and Simple Player v2 does not have to guess. If the player has no free meeples left, the tile is placed at the position with the fewest surrounding tiles to minimize the chance of helping the opponent accidentally.

If the drawn tile contains no cloister another method is used. Simple Player v2 then iterates over all possible tile placements and over all elements on the tile. For each element it checks which elements on the adjacent tiles it will be connected to, applies some rating, similar to Simple Player v1 and in the end chooses the move with the best rating. Algorithm 6 (Appendix C) describes the method in detail.

Simple Player v3

Simple Player v2 is reasonable and can be used as a playout policy and with some effort as move ordering. But it can not be used for roulette wheel selection. If there is no cloister on the tile, Simple Player v2 already uses a rating function that produces numbers, which could be used for roulette wheel selection. If there is a cloister, it does not.

Furthermore the fixed rule for cloisters can be not the best, if instead a field or road with more than 9 points can be occupied, or if the tile placement helps the opponent, for instance by connecting two fields for which the opponent then gains the majority. To overcome this, cloisters can also be integrated into the rating function. Additionally some other small improvements were included. Algorithm 7 (Appendix C) shows the resulting algorithm.

JCloisterZone

There is an open source Java implementation of Carcassonne, wich is called JCloisterZone[1], it includes a basic AI player (compare Appendix B). The AI included plays pretty well, while not taking noticeable time to think. Sebastian Sohn's SoftBoard Games list rates version 1.6.4 from June, 13th 2010 as "Advanced (better than average players)"[2] (the current release is 2.6 from January, 8th 2014).

JCloisterZone's AI is a one-ply minimax search using complex heuristic evaluation functions, that rank many features, for instance detection of trapped meeples, blocking moves, the probability to fill an open position on the board and to complete elements and the possibility to steal elements.

To be able to compete against this AI, an attempt was made to adapt its source code to Carcasum. The resulting player does not play exactly like the original JCloisterZone, since in the process a bug in JCloisterZone's undo management was discovered that could not be adapted to Carcasum. The bug has been reported and was confirmed shortly after. Leaving out the bug should, in theory, not make the algorithm work any worse, so it can be used for comparison. It could however not be used to improve MCTS' playouts, since the adapted implementation is much too slow. This is partially because the heuristic evaluation is pretty complex and partially because Carcasum's design does not fit some of JCloisterZone's operations well, which result in quite slow execution.

5.2 Experimental Setup

Performing experiments in a Game AI context can take a long time. Therefore a trade off between good statistical significance and a feasible time consumption has to be made. Most of the presented tests were run with 208 games, so each player plays 104 games as starting player and 104 games as second player. Both players had 5 seconds of thinking time per turn. The game is played from the initial position, the tiles are drawn randomly and the game is played until the end.

The default setup for MCTS Player, if not mentioned otherwise, uses Score Portion as reward function (see Section 4.4.3), Random as playout policy and a C_p value of 0.5.

A game usually consists of 71 turns, except when a tile is drawn that has no valid placements. Assuming these numbers, one experiment (one player setup vs. another) takes a time of $208 \cdot 5\text{s} \cdot 71 =$

[1] website: `http://jcloisterzone.com/en/`, source code: `https://github.com/farin/JCloisterZone`

[2] `http://www.boardgamegeek.com/geeklist/8323/item/1301418#item1301418`, June 14th 2010, accessed on July 7th 2014

73840s = 20h 30min 40s. All time dependent experiments were run on on an Intel Xeon E3-1230v2 CPU with 3.30 GHz running Arch Linux with Linux kernel 3.14.4. The software was compiled using GCC 4.9.0 and Qt 5.2.1 in 64-bit mode. Since the CPU supports hyper threading, two games were executed simultaneously on one core. This reduces the performance, and thus the number of games that can be simulated per second, but it is not cut in half, so we can profit from it.

As an indicator how good a result of a player is, the following measure, we call scoring rate is used:

$$w = \frac{\text{wins} + 0.5 \cdot \text{draws}}{\text{total number of games}}$$

Note that $w_{player1} = 1 - w_{player2}$. To calculate confidence bounds, the formula from Heinz (2000) is used:

$$b(w) = z_{\%} \cdot \sqrt{w \cdot (1-w)} / \sqrt{n}$$

such that the confidence bound is then given by

$$w \pm b(w)$$

where n is the total number of games and $z_{\%}$ denotes the upper critical value of the standard normal distribution for a %-level of statistical confidence. All numbers were calculated using $z_{90\%} = 1.645$.

In all tables in this chapter, 'win' denotes the number of wins by the respecting player, 'draw' the number of draws, and 'n' the total number of games. 'Rate' denotes the scoring rate, 'pts.' the average number of points the player scored over all games, 'ppp' denotes the average number of playouts per turn a player was able to simulate in order to choose a move, 'pps' denotes the average number of playouts per seconds the player reached while choosing the move, and 'conf.' denotes $b(w)$.

5.3 Results from Simple Players

To be able to use the simple players as heuristic improvements for Monte-Carlo Tree Search, it has to be shown that they work. Since the simple players are fast, it is possible to run more games than for other experiments.

Simple Player v1 is run vs. Random Player to show that it works. Then all possible combinations of the three Simple Player variants are run against each other, and for comparison, they were also run against JCZ Player, which is the adaption of JCloisterZone's AI that plays similar to the original (see Subsection 5.1.6). JCZ Player is slower than other simple players, therefore less games were simulated. The results are shown in Table 5.1.

Simple Player v1			player 2					
win	rate	pts.	player	win	pts.	draw	n	conf.
99995	100.00%	97.5	Random Player	5	25.3	0	100000	0.00%
41132	42.08%	89.5	Simple Player v2	56982	93.5	1886	100000	0.26%
22586	23.28%	87.9	Simple Player v3	76030	103.2	1384	100000	0.22%
1749	18.16%	79.5	JCZ Player	8118	100.5	133	10000	0.63%
Simple Player v2			player 2					
win	rate	pts.	player	win	pts.	draw	n	conf.
37940	38.80%	92.0	Simple Player v3	60348	98.2	1712	100000	0.25%
2905	29.93%	80.8	JCZ Player	6920	92.9	175	10000	0.75%
Simple Player v3			player 2					
win	rate	pts.	player	win	pts.	draw	n	conf.
4782	48.72%	90.1	JCZ Player	5038	91.5	180	10000	0.82%

Table 5.1: Results of different simple players playing against each other.

For better evaluation, Table 5.2 shows the scoring rates of the different combinations. The last column averages all other values in the row, and gives an average win scoring rate. The last row does the same for the columns and gives an average loss scoring rate, i.e. the average scoring rate of the opponents.

Figure 5.1 visualizes the average scoring rate of each player. It can be clearly seen that Simple Player v1 beats Random Player and also that every other simple player is better than Simple Player v1.

Of the three Simple Player variants, Simple Player v3 is the best. Considering that JCZ Player uses a variety of heuristics, and that it has to apply and un-apply all possible tile and meeple moves every turn, it is impressive that Simple Player v3 can still reach a scoring rate of 49% against it.

wins \losses	SimplePlayer	SimplePlayer2	SimplePlayer3	JCZPlayer	avg.
Simple Player v1	–	42.08%	23.28%	18.16%	27.84%
Simple Player v2	57.93%	–	38.80%	38.80%	45.17%
Simple Player v3	76.72%	61.20%	–	48.72%	62.22%
JCZ Player	81.85%	70.08%	51.28%	–	67.73%

Table 5.2: Scoring rates of simple players.

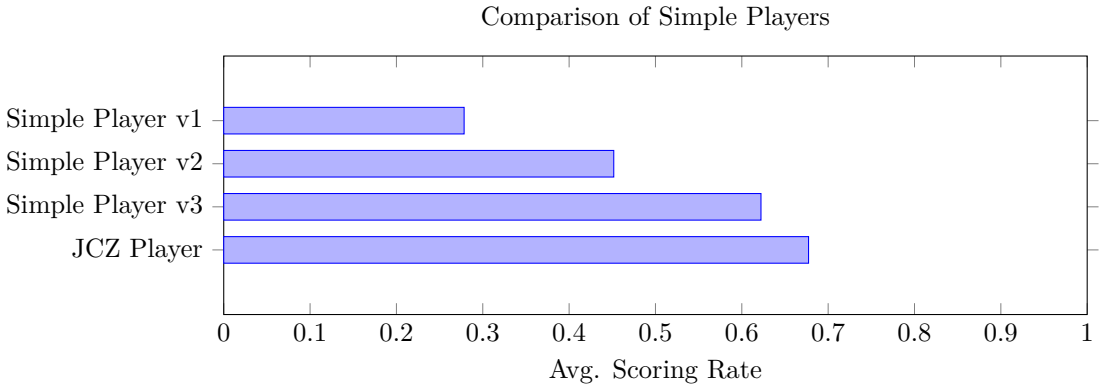

Figure 5.1: Average scoring rates of simple players.

5.4 Monte Carlo

The next step is to verify that the Monte Carlo method works in Carcassonne. The first test therefore is to run flat Monte Carlo against the Random Player (see Table 5.3 and Figure 5.2).

	player 1					player 2: Monte Carlo Player						
player	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
Random Player (MC: 50ms)	0	0.00%	13.1			208	51.7	310.0	7329.8	0	208	0.00%
Simple Player v3	50	24.52%	78.8			156	93.6	29978.6	7325.5	2	208	4.91%
JCZ Player	53	25.96%	73.2			153	83.7	29471.4	7238.3	2	208	5.00%
Monte Carlo Player 2	108	53.61%	77.0	39748.2	7942.7	93	75.3	29874.3	7403.4	7	208	5.69%
1-Ply Search using UCB	116	57.21%	79.3	38913.5	7774.6	86	76.6	29603.1	7334.3	6	208	5.64%

Table 5.3: Results of different other players vs Monte Carlo Player.

To further emphasize its capability, the thinking time of Monte Carlo Player was reduced to 50 ms per turn for the comparison with Random Player. In this time, it reached 310 playouts per ply. This means that in average, in every turn it could do 310 simulations in total for both the searches for a tile move and a meeple move combined. Monte Carlo won 100% of the games. For the other tests the thinking time was set to 5 seconds for each player. Note that Simple Player v3 and JCZ Player do not make use of the time.

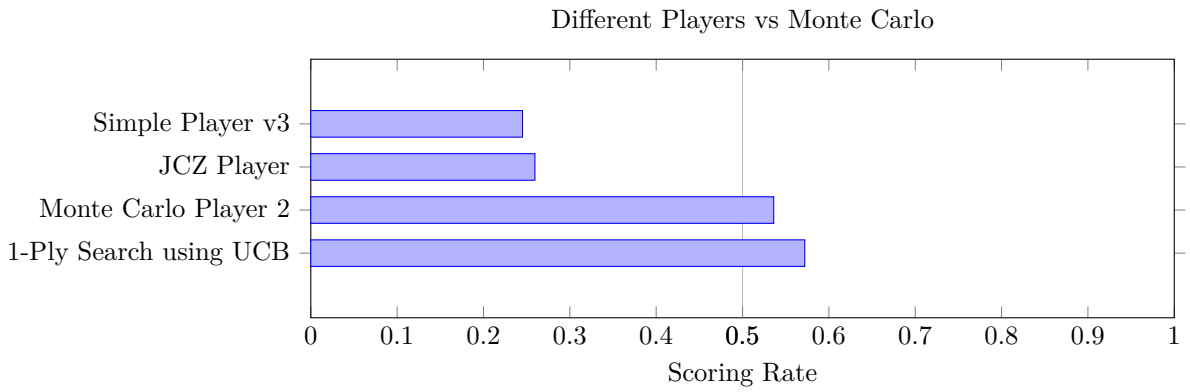

Figure 5.2: Scoring rate of different other players against Monte Carlo Player.

Next, Monte Carlo Player had to run against the simple players. It reached a scoring rate of 75% against Simple Player v3 and 74% against JCZ Player, so Monte Carlo can beat both players.

Monte Carlo Player also played against its variants, Monte Carlo Player 2, which uses one search instead of two and Monte Carlo Player UCB, which does the same, but uses UCB selection instead of random selection. Both perform slightly better than Monte Carlo Player, probably because they simply use the complete available thinking time, while Monte Carlo Player does not (see Section 5.1).

5.5 Monte-Carlo Tree Search

Since Monte Carlo Player can win against all previously tested players, Monte-Carlo Tree Search is only tested against the Monte Carlo variants, and for comparison against JCZ Player. MCTS Player uses Score Portion as reward function and Random as playout policy. As Table 5.4 and Figure 5.3 show, MCTS Player reaches a scoring rate of 84% against JCZ Player and 58% against Monte Carlo Player and can therefore considered to be better than both.

	player 1					player 2: MCTS Player						
player	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
JCZ Player	33	16.11%	69.3			174	86.0	42879.3	8547.7	1	208	4.19%
Monte Carlo Player	84	41.83%	77.7	31181.1	7701.9	118	79.2	42675.3	8506.3	6	208	5.63%
Monte Carlo Player 2	95	48.08%	77.5	41779.0	8351.6	103	79.4	42767.7	8528.7	10	208	5.70%
1-Ply Search using UCB	106	51.68%	78.3	40877.7	8170.9	99	78.1	42444.6	8463.9	3	208	5.70%

Table 5.4: JCZ Player and Monte Carlo Players against MCTS Player.

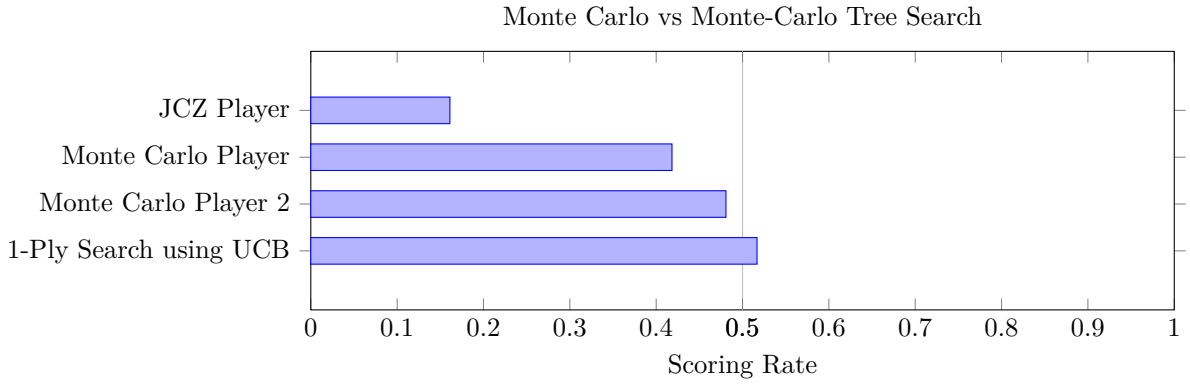

Figure 5.3: Scoring rate of different players against MCTS Player.

5.6 Doubling Experiment

In the previous section the search players used a fixed time setting of 5 seconds. In this section, we analyze the effect of computation limits. Two identical MCTS Players play against each other, as a limit they get a fixed number of playouts, instead of time. The only difference is that one player gets the double number of playouts. Both players use Score Portion as reward function and Random as playout policy. Similar experiments have been done for instance in chess using classic search approaches, where the thinking time per player was doubled (Heinz, 2000).

player 1				player 2						
playouts	win	rate	pts.	playouts	win	rate	pts.	draw	n	conf.
16	1797	90.95%	15.689	8	159	9.05%	5.594	44	2000	1.06%
32	1600	82.48%	23.024	16	301	17.53%	17.033	99	2000	1.40%
64	1380	73.20%	30.570	32	452	26.80%	27.056	168	2000	1.63%
128	1291	68.43%	39.006	64	554	31.58%	35.642	155	2000	1.71%
256	1258	66.55%	46.257	128	596	33.45%	43.770	146	2000	1.74%
512	1214	64.40%	52.389	256	638	35.60%	50.022	148	2000	1.76%
1024	1208	63.20%	57.618	512	680	36.80%	55.154	112	2000	1.77%
2048	1170	61.68%	62.438	1024	703	38.33%	60.050	127	2000	1.79%
4096	1147	60.10%	65.289	2048	743	39.90%	63.172	110	2000	1.80%
8192	1114	57.70%	67.511	4096	806	42.30%	65.618	80	2000	1.82%
16384	1077	56.33%	68.720	8192	824	43.68%	67.134	99	2000	1.82%

Table 5.5: Two MCTS Players with different playout limits play against each other.

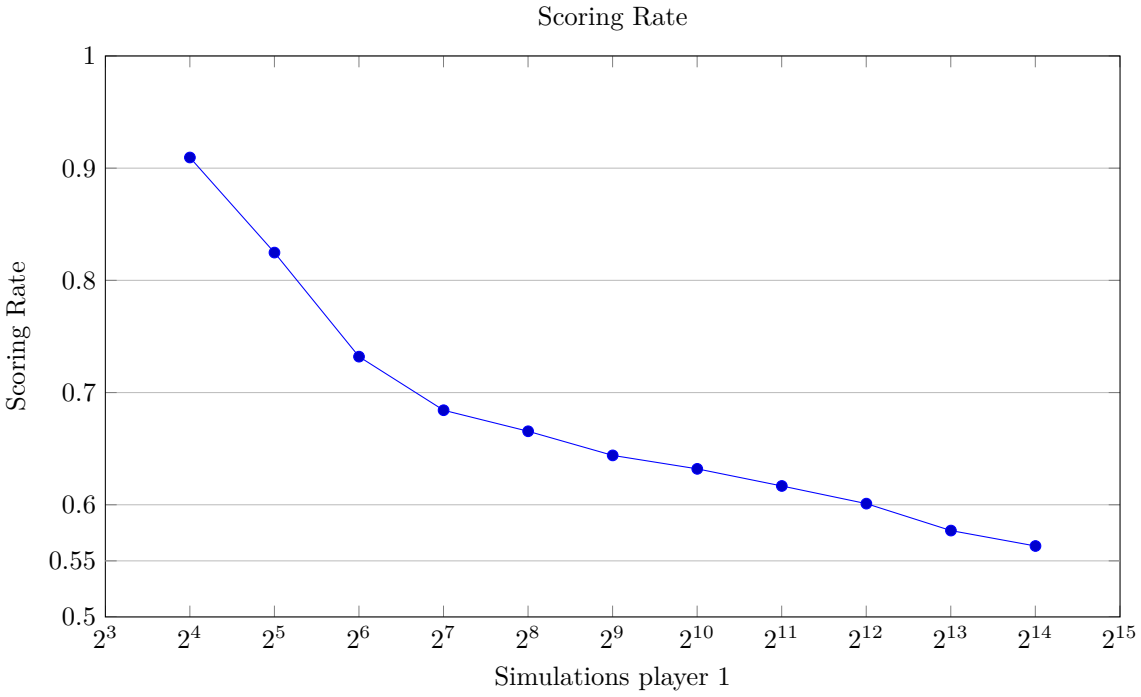

Figure 5.4: Doubling experiment. Scoring rate for player 1.

Consider Table 5.5 for detailed results and Figures 5.4 and 5.5 for a visual representation. The player with the doubled number of playouts is called player 1, the other one is called player 2.

Adding more playouts, player 2 gets increasingly closer to player 1. At some point, adding more playouts does not make much difference for their win-loss ratio. The players still become stronger, but compared to each other, their relative strengths seem to stay about equal.

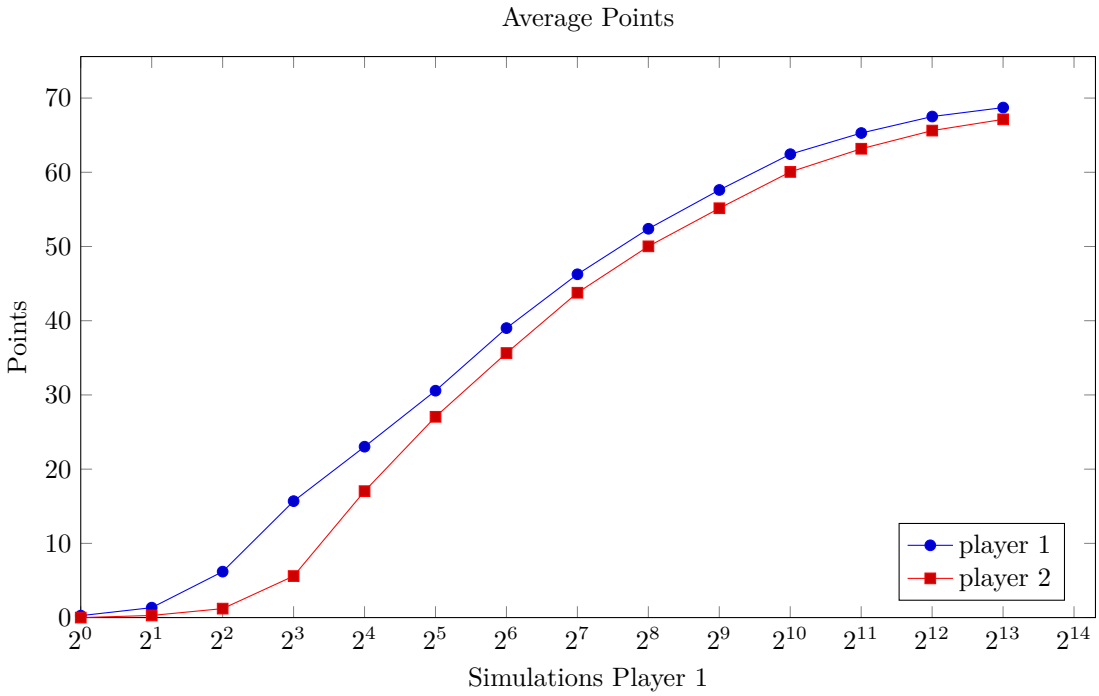

Figure 5.5: Doubling experiment. Average points the players scored in the games.

Although player 1 always has the double number of playouts, it does not win the double number of games. This is also known as the law of diminishing returns, states that the more of one resource is added to a process, while keeping all other factors constant, the less effective it is used. This has behavior has also been observed in other games, for instance in chess.

A similar behavior can be observed for the reached points. The more simulations are added, the more points they make, but also the less steeper the increase becomes. The maximum possible number of points is limited, thus at some point, the scored points cannot increase any more. Note that the number of points scored by player 2 is not the same as the number of points scored by player 1 in the previous setup, although in the previous setup player 1 had the same number of playouts. Apparently the way the players play, more simulations for one player lead to more points for both players.

5.7 UCT C_p Value

UCT uses the constant factor C_p in the UCB Formula to trade off between exploration and exploitation. To determine the best value for C_p, MCTS Player with $C_p = 0.5$ was run against other MCTS Players with different values for C_p. All players use Score Portion as reward function and Random as playout policy.

Detailed results are shown in Table 5.6. Considering Figure 5.6, it appears that there is a maximum at $C_p = 0.5$, but at this point both players are identical, and both players use $C_p = 0.5$. In theory, this point should have a scoring rate of 50%, so this data point has to be considered as statistical noise. Therefore we can only conclude that a C_p value of 0.5 is a good value, and no other value is clearly better.

5.8 MCTS Enhancements

There are multitudes of possible enhancements for Monte-Carlo Tree Search. In this section the following enhancements are tested: different reward functions, different playout policies, such as Early Terminations, ε-Greedy and Roulette Wheel selection, Tree Re-Usage, Node Priors, Progressive Widening and Progressive Bias.

	player 1					player 2: $C_p = 0.5$						
C_p	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
0	21	11.78%	62.1	44911.0	8962.2	180	81.0	42543.8	8483.6	7	208	3.68%
0.25	101	49.28%	79.6	42041.9	8382.2	104	79.1	42359.2	8444.9	3	208	5.70%
0.5	115	55.53%	79.9	42222.1	8419.4	92	77.2	42108.2	8396.6	1	208	5.67%
0.75	103	52.16%	79.4	42190.9	8411.1	94	78.2	42636.3	8499.3	11	208	5.70%
1	106	51.68%	79.5	42043.1	8382.9	99	78.0	42031.9	8380.4	3	208	5.70%
2	93	44.95%	77.4	42271.8	8428.0	114	78.9	41841.9	8342.7	1	208	5.67%
3	93	46.39%	77.0	42590.7	8490.5	108	77.7	42185.9	8410.3	7	208	5.69%

Table 5.6: MCTS Player with different values for C_p against MCTS Player with $C_p = 0.5$.

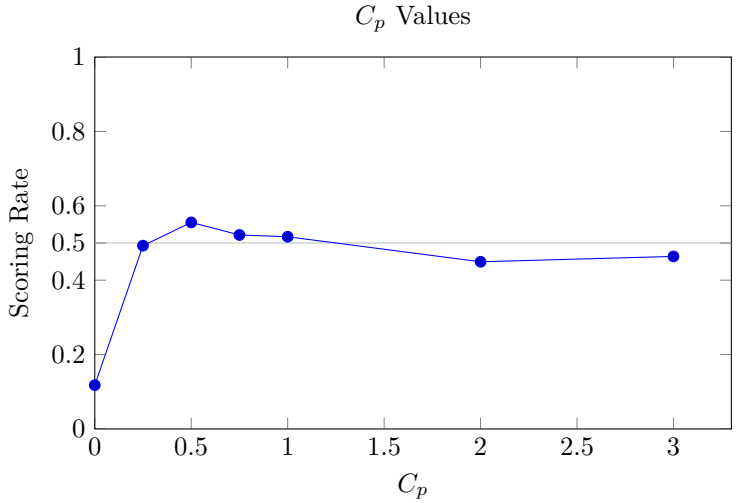

Figure 5.6: Scoring rate of MCTS Player with different values for C_p.

5.8.1 Reward Function

For this experiment, MCTS Players using different reward functions (as described in Section 4.4) play against an MCTS Player using the Simple reward function (returns +1 for a win, 0 for a draw and -1 for a loss).

	player 1					player 2: Simple Reward						
Reward Function	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
Score Difference	72	37.98%	65.7	44792.6	8938.0	122	62.0	44044.8	8784.8	14	208	5.54%
Normalized<Score Difference>	108	52.64%	80.2	42244.0	8423.0	97	65.3	42705.8	8517.2	3	208	5.70%
NormalizedNeg<Score Difference>	118	59.13%	81.4	42400.8	8454.4	80	63.6	42460.4	8468.8	10	208	5.61%
Normalized<Heyden's Evaluation>	122	59.38%	80.3	38911.5	7760.2	83	64.5	42700.0	8515.9	3	208	5.60%
Score Portion	101	49.76%	78.5	42708.1	8516.3	102	65.3	42839.0	8544.6	5	208	5.70%
Bonus<Score Portion/100>	102	50.96%	73.3	43236.7	8616.2	98	64.6	43419.2	8653.4	8	208	5.70%

Table 5.7: Comparison of different reward functions.

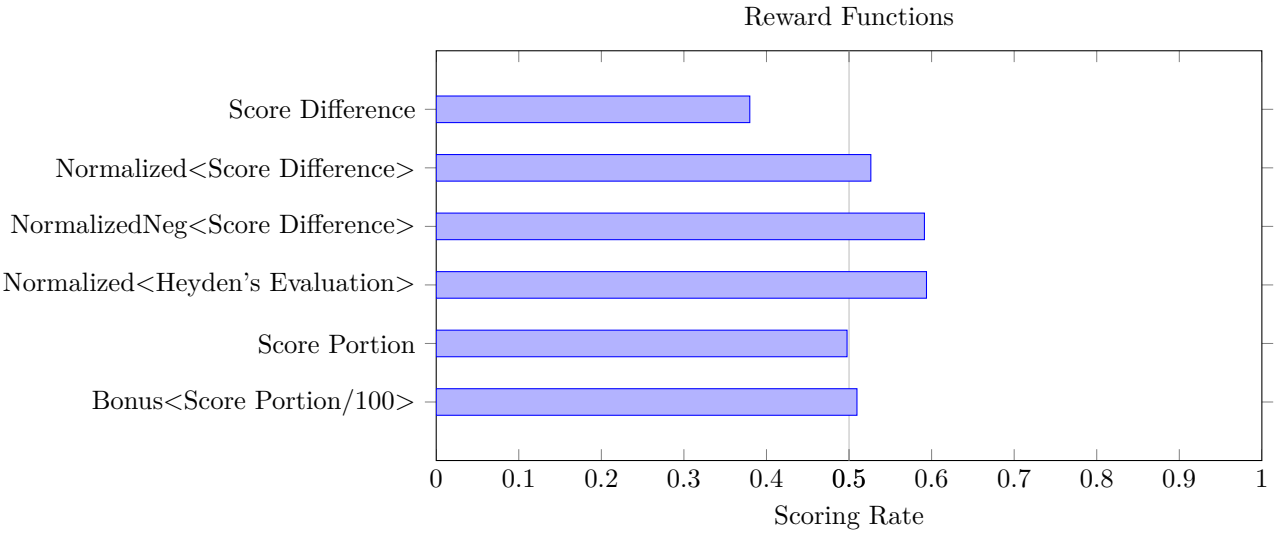

Figure 5.7: Scoring rate of different reward functions.

See Table 5.7 for results and Figure 5.7 for visualization. Normalized<> denotes the normalization described as Normalization in [0, 1] in Subsection 4.4.5, and NormalizedNeg<> denotes the normalization described as Normalization in [-0.5, +0.5] in Subsection 4.4.5. The expression in the angle brackets denotes the reward function that is normalized. Bonus<> denotes the Bonus reward function described in Subsection 4.4.7. The expression in the angle brackets denotes the reward function that is used as ω.

Score Difference is the only reward function that is not normalized and can range between the negative and positive maximum possible score (see Section 4.4.5). As expected, it does not perform very well. Most other reward functions do not seem to differ much. Comparing the results for Normalized<Score Difference> and NormalizedNeg<Score Difference> might give a hint that the latter is the better normalization approach.

Heyden's Evaluation appears to be the best reward function, if properly normalized. Considering the result of the normalized Sore Difference, Heyden's Evaluation could possibly perform even better using NormalizedNeg instead of Normalized. However, Score Difference is balanced symmetrically around 0, Heyden's Evaluation is not, so this might make a difference when using NormalizedNeg.

5.8.2 Playout Policy

In these experiments, MCTS Players using different playout policies play against an MCTS Player using the Random playout policy.

Early Terminations

For Early Terminations, MCTS Players with different cutoff limits played against MCTS Player with Random playout. Table 5.8 and Figure 5.8 show the results. For Early Terminations a game end is simulated when the cutoff is reached. This means that all remaining meeples are scored in the way they would be scored if the game ended in that turn. The resulting scores are then rated using the Score Portion reward function. If the simulation hits a terminal state before the cutoff value is reached, the upper or lower bound of the reward function is returned, i.e. in the case of Score Portion 0 for a loss and 1 for a win.

	player 1					player 2: Random Playout						
Cutoff	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
0	60	29.09%	68.5	212574.0	41542.2	147	81.3	39778.2	7931.0	1	208	5.18%
4	68	32.93%	69.4	126412.0	24975.7	139	83.1	39987.5	7974.4	1	208	5.36%
8	94	46.39%	71.7	88253.8	17517.2	109	79.5	40680.3	8111.5	5	208	5.69%
16	103	50.24%	70.7	60457.1	12036.3	102	79.9	40668.0	8109.1	3	208	5.70%

Table 5.8: MCTS Player with Early Termination and different cutoff values against MCTS Player with Random playout policy.

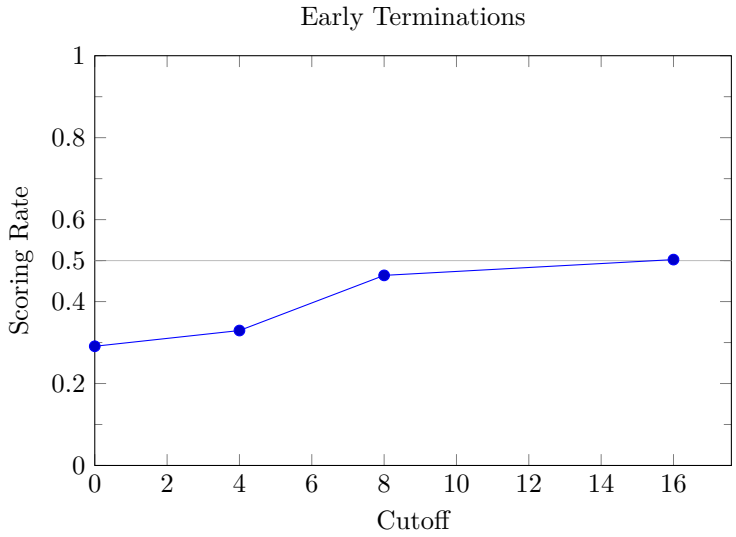

Figure 5.8: Scoring rate of MCTS Player with different values for Early Termination cutoffs.

Unfortunately, we were not able to improve performance. This is probably no problem of added overhead, in fact, this is the only improvement tested that removes overhead. It might be a problem of an unsuitable reward function. The game is treated as end game and all remaining meeples are scored, then only the resulting scores are considered. This might not be optimal, since majorities in elements can change, but this way MCTS assumes that they cannot. For instance, Heyden's Evaluation could suit this problem better.

ε-Greedy

For the ε-Greedy playout approach, with a probability of ε a random move is chosen, otherwise the move Simple Player v3 would do is chosen. Consider Table 5.9 and Figure 5.9 for results. ε-Greedy loses often with $\varepsilon = 0$, because there is very little variance in the playout, so that it always looks the same, and MCTS cannot gain information through them. For $\varepsilon = 1$ both players are equal, since ε-Greedy always plays random moves. An ε value between 0.5 and 1 appears to be the best.

	player 1					player 2: Random Playout						
ε	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
0.0	73	35.82%	64.0	22289.0	4451.5	132	72.7	43074.8	8588.1	3	208	5.47%
0.2	87	43.51%	70.2	23310.0	4655.0	114	73.3	42961.8	8565.4	7	208	5.65%
0.5	112	55.53%	73.5	27578.3	5505.8	89	70.1	43562.9	8684.1	7	208	5.67%
0.8	134	65.38%	77.2	32652.5	6516.3	70	71.3	42601.7	8493.8	4	208	5.43%
1.0	113	57.83%	79.3	42144.9	8402.0	82	78.9	41874.2	8346.9	3	198	5.77%

Table 5.9: MCTS Player with ε-Greedy playout policy and ε values against MCTS Player with Random playout policy.

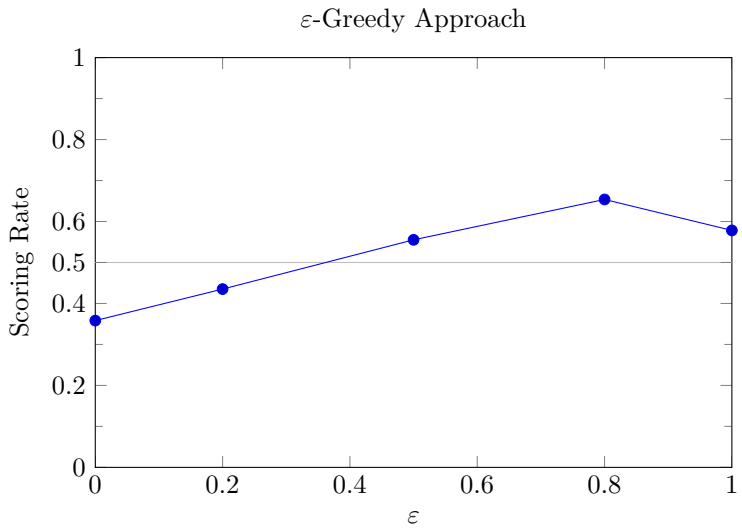

Figure 5.9: Scoring rate of MCTS Player with ε-Greedy playout and different values for ε.

Roulette Wheel Selection

Roulette Wheel Selection lets Simple Player v3 rate the possible moves and then chooses a move randomly according to the rating distribution. There are two variants of this. Roulette Wheel Selection 1 gets a list from Simple Player v3 that rates every combination of tile move and meeple move. Roulette Wheel Selection 1 chooses one combination randomly according to the ratings. Roulette Wheel Selection 2 gets a list from Simple Player v3 that contains a rating for every possible tile move without a meeple move, and a list with ratings for every possible meeple move for that tile move. Roulette Wheel Selection 2 chooses a tile move randomly according to the tile ratings, and then chooses a meeple move randomly according to the meeple ratings.

	player 1					player 2: Random Playouts						
Playout Policy	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
Roulette Wheel Selection 1	80	38.94%	68.2	22924.7	4578.4	126	73.5	43644.1	8702.0	2	208	5.56%
Roulette Wheel Selection 2	73	35.58%	65.5	22105.4	4414.6	133	74.1	43556.4	8683.5	2	208	5.46%

Table 5.10: MCTS Player with Roulette Wheel Selection against MCTS Player.

Table 5.10 shows the results for both variants. Surprisingly, Roulette Wheel Selection decreases the strength by a lot. The heuristic from Simple Player v3 plays verifiably better than a random player. The ε-Greedy approach also uses the same heuristic and reaches much better results than Roulette Wheel selection. Thus it seems likely that Roulette Wheel Selection has an implementation bug. Even after careful inspection, no bug was found. To test this, another experiment was performed that lets both variants of pure Roulette Wheel Selection play against Random Player and against Simple Player v3. It is expected that Roulette Wheel Selection beats Random Player clearly, and does not perform to well against Simple Player v3, because Simple Player v3 always takes the best move, according to its heuristic, while Roulette Wheel player can choose any move.

Roulette Wheel Player 1			player 2					
win	rate	pts.	player	win	pts.	draw	n	conf.
18868	94.76%	41.1	Random Player	965	18.1	167	20000	0.26%
94	0.49%	41.3	Simple Player v3	19897	97.4	9	20000	0.08%
Roulette Wheel Player 2			**player 2**					
win	rate	pts.	player	win	pts.	draw	n	conf.
19714	98.72%	49.5	Random Player	227	18.5	59	20000	0.13%
214	1.14%	47.9	Simple Player v3	19758	96.9	28	20000	0.12%

Table 5.11: Roulette Wheel Selection vs Random Player and Simple Player v3. The numbers are in perspective of the Roulette Wheel Players.

Table 5.11 shows this experiment. Other than in MCTS, Roulette Wheel Player 2 provides better results than Roulette Wheel Player 1. It clearly beats Random Player, which proves that it works. Roulette Wheel Player 2 could only win in 1% of the games against Simple Player v3. It was expected to lose, still it is interesting that it loses so clearly.

Another possible explanation for the bad result of Roulette Wheel Selection might be that the ranking values of the actions do not provide enough variance. If every move is rated nearly equally, all moves become nearly equally probable. As a consequence, the quality of the playout becomes similar to purely random playout, while being much slower.

In fact, MCTS with Roulette Wheel Selection 1 produces 1.90 times less playouts than with purely random playout, and Roulette Wheel Selection 2 produces 1,97 times less playouts. ε-Greedy performs better for two reasons: First, it can use the unmodified version of Simple Player v3 which only returns one move while Roulette Wheel Selection needs a modified version in order to get all values, this is more expensive because it creates and returns a list. Second, we can see that a high randomness factor performs best for ε-Greedy. A high percentage of random moves means a low percentage of heuristic function calls

which increases the speed. ε-Greedy with $\varepsilon = 20\%$ is about 1.84 times slower than random playouts, while with $\varepsilon = 80\%$, it only is about 1.30 times slower.

5.8.3 Other Enhancements

Some other common MCTS enhancements as described in Section 3.4 were tested.

	player 1					player 2: Plain MCTS Player						
Enhancement	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
Tree Re-Usage	95	46.88%	77.1	40171.3	8010.2	108	78.8	39508.0	7874.7	5	208	5.69%
Node Priors	77	37.50%	74.0	38156.1	7609.8	129	78.8	37916.1	7561.9	2	208	5.52%
Node Priors HE	109	53.61%	79.2	31775.1	6338.7	94	77.4	37880.5	7553.6	5	208	5.69%
Progressive Widening	100	50.48%	79.5	34627.0	6898.9	98	79.0	37867.9	7550.1	10	208	5.70%
Progressive Bias	98	48.32%	76.4	31650.6	6313.8	105	75.9	34744.8	6928.5	5	208	5.70%

Table 5.12: MCTS Player with different enhancements against MCTS Player.

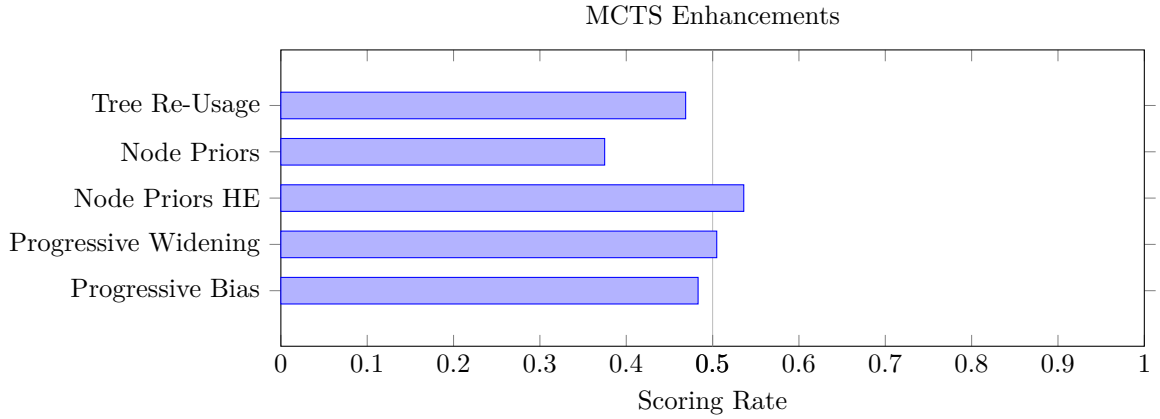

Figure 5.10: Scoring rates of different MCTS enhancements against MCTS Player.

Table 5.12 and Figure 5.10 show the results. Node Priors HE uses Heyden's Evaluation as reward and heuristic function instead of Score Portion. Using Heyden's Evaluation as a reward function has previously shown to work better than MCTS using Score Portion in Subsection 5.8.1. However, the result in this experiment using Node Priors and Heyden's Evaluation is worse than the result for only using Heyden's Evaluation.

It appears that none of these common MCTS enhancements improve the performance of MCTS in the case of Carcassonne. Tree Re-Usage might not be applicable in the case of Carcassonne, because the tree is too wide and only a few simulations can be re-used in the next turn. As the previous results already hint, for some enhancements there could be the problem of additional overhead. Many additions to MCTS makes it slower, which is a disadvantage when competing against an unmodified MCTS player, this might be stronger than the positive effect the enhancement has.

To further investigate in the problem of additional overhead, the experiment with Node Priors HE was repeated, but this time using a fixed number of playouts as a limit for both players, instead of a time. The experiment was done twice, once using the same number of playouts per ply MCTS player reached during the first experiment, and once using 40,000 playouts.

Consider Table 5.13 and Figure 5.11 for the results. Unfortunately, the result is not statistically significantly better than with using a timeout. Another possible explanation was that the used reward function, which is taken from a minimax approach, does not work as well in MCTS as in minimax.

The number of simulations that could be run was quite low. Some of the results might be explainable with statistical variance. However, using 16,384 playouts compared to 8192 only showed a scoring rate of 56% (see Section 5.6), other experiments reach about 40 thousand playouts, depending on the setup. Considering the law of diminishing returns, with these high playout numbers, the effect of any change

	player 1					player 2: Plain MCTS Player						
Playout Policy	win	rate	pts.	ppp	pps	win	pts.	ppp	pps	draw	n	conf.
Node Priors HE 38k	98	47.60%	79.3	37880	3506.7	108	78.6	37880	3694.9	2	208	5.70%
Node Priors HE 40k	110	55.05%	80.0	40000	3375.4	89	77.6	40000	3564.0	9	208	5.67%

Table 5.13: MCTS Player with Node Priors and fixed playouts against MCTS Player with the same number of playouts.

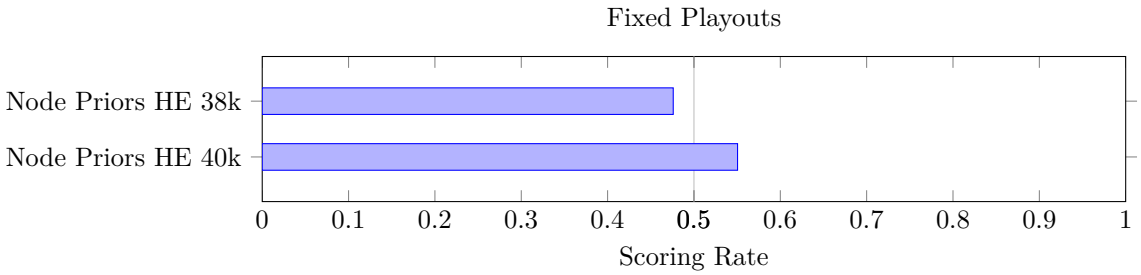

Figure 5.11: Scoring rates of MCTS with Node Priors and fixed playouts against MCTS Player.

might be pretty small. Therefore it might also be possible that the tested thinking time was too long to see any significant changes and that they could have been observed using less thinking time. Using less thinking time would also have the benefit of being able to run more experiments and getting statistically more significant results. However, if diminishing returns are the problem, the question must be asked, whether it makes any sense to reduce the thinking time just in order to see differences and to improve a player under these conditions, while 5 seconds is a fairly reasonable thinking time. Reduced thinking time could make it possible to see differences and improve a player in ways that would no longer have any effect when using 5 seconds when playing real games.

Chapter 6

Conclusions and Future Research

In this chapter, the conclusion of the thesis is given. First, the research question is answered, afterwards the problem statement is answered. Finally, ideas for future research are presented.

6.1 Answering the Research Questions

In section 1.3, research questions were defined.

- Do Monte Carlo methods work in Carcassonne?

The research has shown that flat Monte Carlo works fairly well, 1-Ply Search using UCB as a modification works about as good and Monte-Carlo Tree Search works best. Flat Monte Carlo can beat a random player even at very low time settings and it can beat the advanced heuristic AI from JCloisterZone. As a subjective impression, it seems to make reasonable moves, even though they are not always the best ones. Statistically, Monte-Carlo Tree Search has been shown to be better than Monte Carlo. The subjective impression of MCTS is a little better than the one of Monte Carlo, but the general impression stays the same.

- Which reward function works best for MCTS in Carcassonne?

Different reward functions that can be applied to Carcassonne were described in Section 4.4 and tested and compared in Subsection 5.8.1. A normalized version of Heyden's Evaluation works best.

- How can the playout policy of MCTS in Carcassonne be enhanced?

In Section 4.5 the playout policies Early Terminations, ε-Greedy, Roulette Wheel Selection and 1-Ply Search were introduced, they were tested in subsection 5.8.2. The only playout policy that worked better than random was ε-Greedy with a high randomness and therefore a lower overhead.

- How can the tree policy of MCTS in Carcassonne be enhanced?

When using UCT, the C_p value can be tuned. Different values for C_p were tested in Section 5.7. $C_p = 0.5$ turned out to be a good value. Additionally, different modifications, such as Tree Re-Usage, Node Priors, Progressive Widening and Progressive Bias were described in section 3.4. They were tested in Subsection 5.8.3 and none of then seem to improve the performance of MCTS significantly.

6.2 Answering the Problem Statement

After answering the research questions, the problem statement can be addressed.

- How can a good MCTS player for Carcassonne be developed?

A reasonable AI player for Carcassonne can be developed using Monte-Carlo Tree Search. In fact, it appears to be quite easy to create a reasonable player. Even using some simple rules can be enough for a reasonable player, as shown by Simple Player v3. Even an advanced player is possible, as demonstrated by the more complex AI from JCloisterZone. MCTS is better than both. However, improving MCTS further turned out to be difficult. Many different approaches were tested but most of them were unsuccessful.

6.3 Future Research

In the experiments section some things that could be done in the future are already noted, for instance repeating the Early Termination experiment using Heyden's Evaluation instead of Score Portion. Also, many experiments could only be run with 200 games, repeating them with a higher number of games would be good to get statistically more significant results. This could be done by reducing the thinking time or investing more time or both. The doubling experiment suggests that we run relatively quickly into diminishing returns. It could be worth finding a limit at which investing more time does not make sense any more. This way the thinking time could maybe be reduced without making MCTS noticeably worse.

Carcassonne is a multi-player game, and this topic has not been addressed yet. In fact, using MCTS for both multi-player games and games with chance have been researched very little. For the combination of both, like in Carcassonne, there has been even less research (in fact we did not find any at all). So the multi-player version of Carcassonne offers lots of room for research. The foundation for this has already been laid, as Carcasum's code is freely available and it supports multi-player games, as do all its implemented AIs; there only was not enough time to research and experiment with multi-player games properly.

The complexity of Carcassonne could be increased nearly arbitrary by using expansions. With many expansions, the number of tiles can easily be increased to 300, while also adding many additional figures and rules. The length of the game and also the game's complexity drastically increase by adding expansions.

Another topic was to research different enhancements for MCTS. First, more known enhancements could be tested, for instance implicit minimax backups (Lanctot *et al.*, 2014b). Second, this thesis hints that adding even a little computational overhead to MCTS decreases its quality. However, ε-Greedy, in which the overhead can be adjusted, performs pretty well, if it is kept low. As a consequence it could be interesting to see if there are other enhancements with such an adjustable overhead, or if new ones could be developed or derived from existing enhancements, and how well these perform.

In this thesis only one domain-specific evaluation function was tested (Heyden's Evaluation, see Subsection 4.4.4). It was taken from another thesis in which it was used in the expectimax framework. It is possible that it does not work so well in MCTS. It is also possible that better evaluation functions exist for Carcassonne, this could be researched. JCloisterZone offers lots of ideas which could be used to design such a function. It is also possible to use JCloisterZone's evaluation directly, which was not done in this thesis because the adaption was too slow. However, using only parts of it or using its rating function directly could be worth trying. Also the adapted code has never been optimized and is known to use some operations that work especially bad in Carcasum, because of incompatibilities between JCloisterZone's and Carcasum's design. However, small changes can be applied to Carcasum which should result in a great speedup of JCZ Player while slightly slowing down everything else. It might also be possible to implement some caching algorithms inside of JCZ Player which could speed up JCZ Player without slowing down other players.

References

Auer, Peter, Cesa-Bianchi, Nicolò, and Fischer, Paul (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, Vol. 47, Nos. 2–3, p. 235–256. ISSN 0885–6125. [12]

Ballard, Bruce W. (1983). The*-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, Vol. 21, No. 3, pp. 327–350. [3]

Bouzy, Bruno (2007). Old-fashioned computer Go vs Monte-Carlo Go. *IEEE Symposium on Computational Intelligence in Games (CIG).* [14]

Browne, Cameron, Powley, Edward, Whitehouse, Daniel, Lucas, Simon, Cowling, Peter I., Rohlfshagen, Philipp, Tavener, Stephen, Perez, Diego, Samothrakis, Spyridon, and Colton, Simon (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 4, No. 1, pp. 1–43. [1, 11, 12, 13]

Campbell, Murray, Hoane Jr., A. Joseph, and Hsu, Feng-hsiung (2002). Deep Blue. *Artificial Intelligence*, Vol. 134, No. 1–2, pp. 57 – 83. ISSN 0004–3702. [1]

Chaslot, Guillaume M.J-B., Winands, Mark H.M., Herik, H. Jaap van den, Uiterwijk, Jos W.H.M., and Bouzy, Bruno (2008). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 343–357. [12, 14]

Coulom, Rémi (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* (eds. H. Jaap Herik, Paolo Ciancarini, and H.H.L.M. (Jeroen) Donkers), Vol. 4630 of *Lecture Notes in Computer Science*, pp. 72–83. Springer Berlin Heidelberg. ISBN 978–3–540–75537–1. [1, 11]

Gelly, Sylvain and Silver, David (2007). Combining Online and Offline Knowledge in UCT. *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pp. 273–280, ACM, New York, NY, USA. ISBN 978–1–59593–793–3. [14]

Heinz, E.A. (2000). Self-play experiments in computer chess revisited. *Advances in Computer Games 9* (eds. H.J. van den Herik and B. Monien), pp. 73–91. [24, 27]

Heyden, Cathleen (2009). Implementing a computer player for Carcassonnne. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht. [3, 6, 9, 10, 17, 18]

Kocsis, Levente and Szepesvári, Csaba (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (eds. Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer Berlin Heidelberg. ISBN 978–3–540–45375–8. [1, 12]

Lanctot, Marc, Winands, Mark H.M., Pepels, Tom, and Sturtevant, Nathan R. (2014a). Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. *IEEE Conference on Computational Intelligence and Games.* [14]

Lanctot, Marc, Winands, Mark H.M., Pepels, Tom, and Sturtevant, Nathan R. (2014b). Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG).* [38]

Lorentz, Richard J. (2008). Amazons Discover Monte-Carlo. *Computers and Games* (eds. H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 13–24. Springer Berlin Heidelberg. ISBN 978–3–540–87607–6. [14]

Lorentz, Richard J. and Horey, Therese (2013). Programming Breakthrough. *8th International Computers and Games Conference*. [14]

Metropolis, Nicholas and Ulam, Stanislaw (1949). The monte carlo method. *Journal of the American statistical association*, Vol. 44, No. 247, pp. 335–341. [1]

Neumann, J. von (1928). Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, Vol. 100, No. 1, pp. 295–320. ISSN 0025–5831. In German. [1]

Nijssen, J.A.M. and Winands, Mark H.M. (2013). Search policies in multi-player games. *ICGA Journal*, Vol. 36, No. 1, pp. 3–21. [3]

Pepels, Tom, Tak, Mandy J.W., Lanctot, Marc, and Winands, Mark H.M. (2014). Quality-based Rewards for Monte-Carlo Tree Search Simulations. *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI)*. [19]

Ramanujan, Raghuram and Selman, Bart (2011). Trade-Offs in Sampling-Based Adversarial Planning. *ICAPS*, pp. 202–209. [14]

Steinhauer, Janik (2010). Monte-Carlo TwixT. M.Sc. thesis, Department of Knowledge Engineering, Maastricht University, Maastricht. [12]

Sturtevant, Nathan R. (2008). An analysis of UCT in multi-player games. *Computers and Games* (eds. H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 37–49. Springer. ISBN 978–3–540–87607–6. [20]

Tesauro, Gerald (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. [1]

Winands, Mark H.M., Björnsson, Yngvi, and Saito, Jahn-Takeshi (2008). Monte-carlo tree search solver. *Computers and Games* (eds. H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands), Vol. 5131 of *Lecture Notes in Computer Science*, pp. 25–36. Springer Berlin Heidelberg. ISBN 978–3–540–87607–6. [14]

Woods, Stewart (2012). *Eurogames: The Design, Culture and Play of Modern European Board Games*. McFarland. [3]

Appendix A

Tiles in Carcassonne

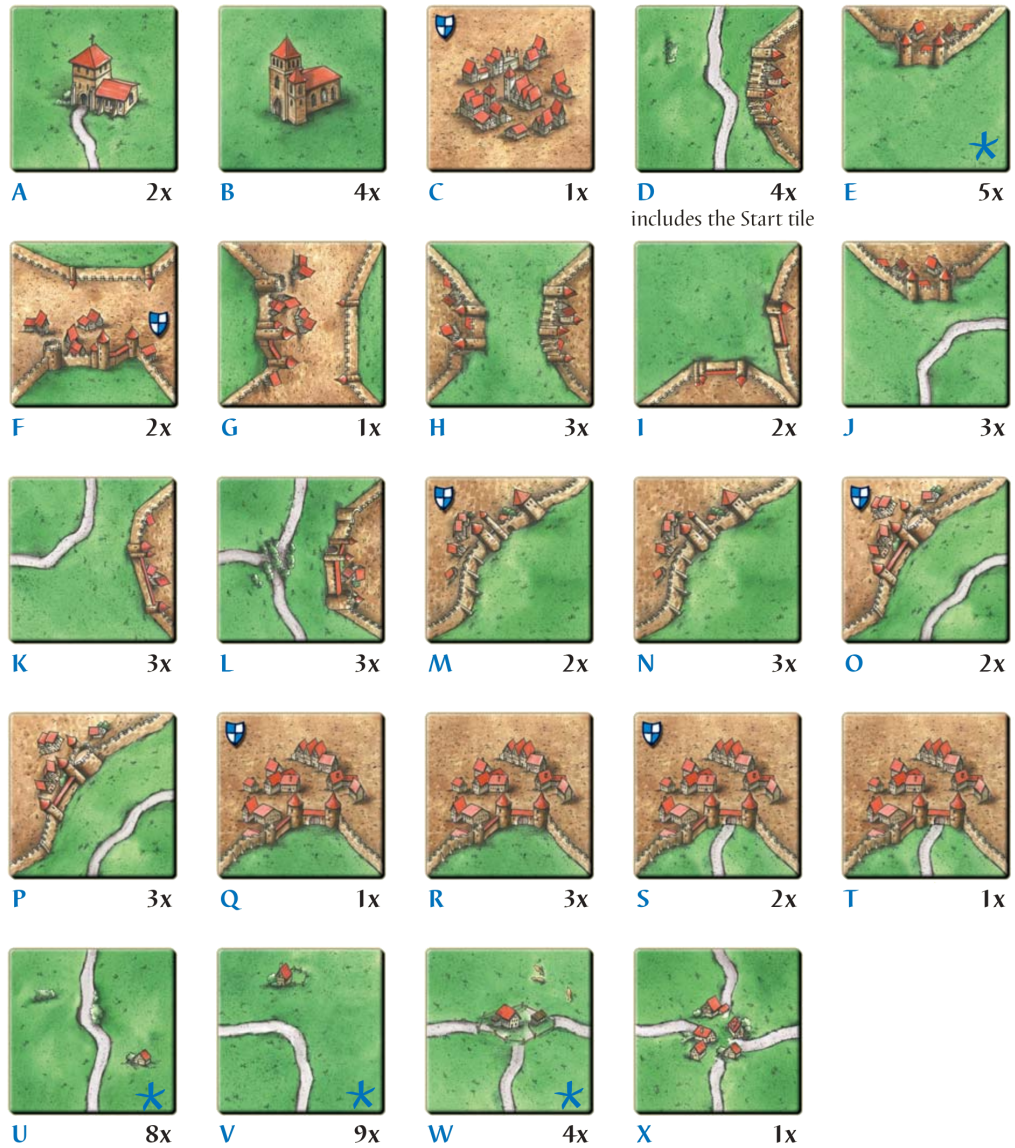

Figure A.1: Tiles in Carcassonne (© by Hans im Glück).

Appendix B

Implementation

The game was implemented from scratch. This chapter gives a brief introduction how the implementation works and how it performs.

There is only one open-source computer implementation of Carcassonne available: JCloisterZone[1]. It is written in Java and is very extensive. It has about 26 thousand lines of code[2], supports 1 to 6 players, network games, many expansions, rule variations and has a built-in AI.

For this thesis JCloisterZone is too heavy, since it contains all the mentioned features and has not been designed to work fast. Therefore the game has to be implemented from scratch.

In order to use Monte-Carlo methods in Carcassonne, the implementation has to be fast. For this reason C++11 was chosen as a programming language. The program has to be able to both run automated experiments on the command line, as well as it needs to have a GUI, to be able to check the game and it is implementation and to get a feeling for how the bots play. As a graphical toolkit Qt was used.

The implementation is called Carcasum, the source code is available online at `https://github.com/triplewhy/carcasum`. Originally it was planned to also investigate the multi-player case, but it quickly turned out it would go far beyond the scope of this thesis. Nevertheless, Carcasum is still capable of multi-player games.

B.1 Structure

The code is divided into four packages:

core core game elements

player AI code

gui GUI code

jcz some code imported from JCloisterZone, or code to interoperate with it

To give an idea of how the code works and how it becomes fast enough, some elements of the core package are explained in a little more detail.

B.1.1 Tiles

The Tile class represents a game tile. In the beginning 72 Tiles are created and placed onto the board during the game. In order to be able to represent the complex structure of Carcassonne's tiles, a tile is modeled as a graph. Each node of the graph represents one element on the board. A Tile has a list of graph nodes that lie on the tile. Additionally every tile has 3 connectors on each edge and one for a cloister. This way all tiles can be modeled.

[1] website: `http://jcloisterzone.com/en/`, source code: `https://github.com/farin/JCloisterZone`

[2] counted for the current release 2.6 using CLOC (`http://cloc.sourceforge.net/`)

When two tiles are lain next to each other, the nodes that are connected to the corresponding edges are merged. The first implementation did actually merge the two nodes into one of them and deleted the other node. However, this made undo actions impossible, so they are now merged logically.

The nodes of the tiles are modeled by the struct Node. Most of the data of a node, like the number of meeples per player, is stored in an instance of the nested struct NodeData. A Node also has pointer to a NodeData that it uses to read and modify information on. Additionally, Nodes have a stack of NodeDatas. When two nodes are merged, the data pointer of one of them is changed to point to the same data as the other node's pointer. Both nodes push the updated pointer to the stack. This way a merge can be undone by popping a pointer from the stack and setting the data pointer to its value. Afterwards, both nodes' data pointers point to the same data object they pointed to before the merge. The other data can be reconstructed by comparing the two data objects.

B.1.2 Board

The Board class stores the game board. Therefore it uses a 2-dimensional array of tile pointers. The array's size in each dimension is twice the tile count and the start tile is placed in the center of the board. Open positions are indicated by a null pointer. This way no game can exceed the limits of the array and the array does not have to be resized or shifted. To speed up iterating over all placed tiles, Board also manages a list of the tiles that have been placed. The list is mainly used to score the remaining meeples when the game ends, but it still provides a significant performance boost, up to a speedup factor of 2, depending on the measurement method.

B.1.3 Game

The Game class manages a game, including the game state. It manages all Tiles by storing the tiles left in a list and all others placed on a Board. Game executes undo steps and game steps by drawing tiles, asking players for their tile and meeple action if necessary, and informing other players about what happens.

B.1.4 Player

Player provides the interface that has to be implemented by all players.

B.2 Performance

Two experiments were done to measure the performance of the core implementation. The average number of playouts per second for a Monte-Carlo or MCTS player is much higher than these results, since the playout gets shorter during the game.

The following experiments were executed on an Intel Xeon E3-1230v2 CPU with 3.30 GHz running Arch Linux with Linux kernel 3.14.4. The software was compiled using GCC 4.9.0 and Qt 5.2.1 in 64-bit mode.

B.2.1 Setup 1

A Game object is created, two random players are added. In a loop, the Game is initialized and played out until the end. Initialization is quite an expensive step, since for example the Board allocates relatively much memory, and all tiles are newly created.

The loop was set to 100,000 repetitions and took 53,005 ms, that are 1886.61 playouts per second.

B.2.2 Setup 2

A Game object is created, two random players are added, the game is initialized. In a loop, the Game played out until the end and all moves are undone again.

The loop was set to 100,000 repetitions and took 30,719 ms, that are 3255.31 playouts per second.

Appendix C

Algorithms

Algorithm 6 Simple Player v2

```
function SimplePlayer2(tile, possibleTilePlacements)
    if tile has cloister then
        if has free meeples then
            tileMove ← arg max_{p∈possibleTilePlacements} (tiles surrounding p)
            meepleMove ← cloister;
        else
            tileMove ← arg min_{p∈possibleTilePlacements} (tiles surrounding p)
            meepleMove ← skip;
    else
        bestPoints ← −∞
        for all p ∈ possibleTilePlacements do
            points ← 0
            bestMeeplePoints ← 0
            bestMeeple ← skip
            for all e ∈ elements of tile do
                meeplePoints ← 0
                score ← Score(e)
                meeples ← [0, 0]                          ▷ # meeples occupying e per player
                if e is field then
                    for all eo ∈ fields of adjacent tiles e would be connected to do
                        score ← max(score, Score(eo))     ▷ When merged, max is an estimate.
                        meeples ← meeples + Meeples(eo)
                else                                      ▷ e is city or road
                    for all eo ∈ elements of adjacent tiles e would be connected to do
                        score ← score + Score(eo)
                        meeples ← meeples + Meeples(eo)
                if max(meeples) ≠ 0 then    ▷ e is occupied. If e is a field, this is an approximation.
                    for all p ∈ Players : meeples(p) = max(meeples) do
                        if IsMe(p) then
                            meeplePoints ← meeplePoints + score · MyBonus
                        else
                            meeplePoints ← meeplePoints − score · OppBonus
                else
                    meeplePoints ← meeplePoints − score · OpenPenalty
                    if has free meeples then
                        meeplePoints ← (score · MyBonus − MeeplePenalty(myMeeples)) · TypeBonus(e)
                        if meeplePoints > bestMeeplePoints then
                            bestMeeplePoints ← meeplePoints
                            bestMeeple ← e
                points ← points + meeplePoints · TypeBonus(e)
            points ← points + bestMeeplePoints
            if points > bestPoints then
                bestPoints ← points
                tileMove ← p
                meepleMove ← bestMeeple
    execute tileMove
    if meepleMove is valid then    ▷ Determining whether a field is already occupied can go wrong.
        execute meepleMove
    else
        execute random meeple move
```

Algorithm 7 Simple Player v3

```
function SimplePlayer3(tile, possibleTilePlacements)
    bestPoints ← −∞
    for all p ∈ possibleTilePlacements do
        points ← 0
        bestMeeplePoints ← 0
        bestMeeple ← skip
        for all e ∈ elements of tile do
            meeplePoints ← 0
            score ← Score(e)
            meeples ← [0, 0]                                  ▷ # meeples occupying e per player
            if e is cloister then
                score ← score + number of tiles surrounding p
                open ← 9 − score
                meeplePoints ← (score·MyBonus−(open·MeeplePenalty(myMeeples)/9)) · TypeBonus(cloister)
            else if e is field then
                for all eo ∈ fields of adjacent tiles e would be connected to do
                    score ← (score + Score(eo))/2             ▷ When merged, this is an estimate.
                    meeples ← meeples + Meeples(eo)
                meeplePoints ← (score · MyBonus − MeeplePenalty(myMeeples)) · TypeBonus(field)
            else                                              ▷ e is city or road
                open ← OpenEdgeCount(e)
                for all eo ∈ elements of adjacent tiles e would be connected to do
                    score ← score + Score(eo)
                    meeples ← meeples + Meeples(eo)
                    open ← open + OpenEdgeCount(eo) − 2
                meeplePoints ← (score · MyBonus − open · MeeplePenalty(myMeeples)) · TypeBonus(e)

            if e is cloister then
                if has free meeples then
                    if meeplePoints > bestMeeplePoints then
                        bestMeeplePoints ← meeplePoints
                        bestMeeple ← e
            else
                elementPoints ← 0
                if max(meeples) ≠ 0 then      ▷ e is occupied. If e is a field, this is an approximation.
                    for all p ∈ Players : meeples(p) = max(meeples) do
                        if IsMe(p) then
                            elementPoints ← elementPoints + score · MyBonus
                        else
                            elementPoints ← elementPoints − score · OppBonus
                else
                    elementPoints ← −score · OpenPenalty
                    if has free meeples then
                        if meeplePoints > bestMeeplePoints then
                            bestMeeplePoints ← meeplePoints
                            bestMeeple ← e
                points ← points + elementPoints · TypeBonus(e)
        points ← points + bestMeeplePoints
        if points > bestPoints then
            bestPoints ← points
            tileMove ← p
            meepleMove ← bestMeeple

    execute tileMove
    if meepleMove is valid then      ▷ Determining whether a field is already occupied can go wrong.
        execute meepleMove
    else
        execute random meeple move
```