

IMPLEMENTING A COMPUTER PLAYER FOR CARCASSONNE

Cathleen Heyden

Master Thesis DKE 09-15

THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE OF ARTIFICIAL INTELLIGENCE
AT THE DEPARTMENT OF KNOWLEDGE ENGINEERING
OF THE MAASTRICHT UNIVERSITY

Thesis committee:
Dr. ir. J.W.H.M. Uiterwijk
Dr. M.H.M. Winands
M.P.D. Schadd, M.Sc.
J.A.M. Nijssen, M.Sc.

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Artificial Intelligence

June 28, 2009

Preface

Thank you for reading my master thesis. The thesis was written at the Department of Knowledge Engineering at the Maastricht University. The subject is the implementation of an Artificial Intelligence for the modern board game Carcassonne.

I would like to express my gratitude to several people. First of all, I would like to thank my supervisor, Dr. ir. Jos Uiterwijk, for his support during the last months, for reading and commenting my thesis and for his suggestions. His lectures during the course *Intelligent Search Techniques* were very helpful. I also want to thank Dr. Mark Winands, Jahn-Takeshi Saito, M.Sc., and Maarten Schadd, M.Sc., for their lectures during that course. I have learnt a lot, thanks for that.

Finally, I would like to thank my family and friends who supported me during the last months. I give my special thank to my fellow student Robert Briesemeister and my boyfriend Peter Schlosshauer for playing countless games of Carcassonne during the last months.

Cathleen Heyden
Aachen, June 2009

Abstract

Classic board games are an important subject in the field of Artificial Intelligence (AI) and a lot of research is done there. Nowadays modern board games get more and more interest from researchers. Most of the games support to play with more than two players. In addition they are often non deterministic, which means that they contain an element of chance, and/or have imperfect information.

This thesis describes the analysis of the game of Carcassonne and the implementation of the game engine and an AI player using different algorithms. Carcassonne is a modern board game for 2 to 5 players with perfect information, which means, that the entire state of the game is fully observable to each of the players. It is a non-deterministic game because during the game the players draw tiles randomly. These chance events are not uniformly distributed because several tiles have different frequencies. This thesis regards only the 2-player variant of Carcassonne.

This work includes a calculation of the state-space complexity and the game-tree complexity. After that two search algorithms are investigated. The first algorithm is Monte-Carlo search and an improved version of it – Monte-Carlo Tree Search with Upper Confidence bounds applied to Trees. The second algorithm is Expectimax search, which is based on Minimax search. Additionally the improved algorithms Star1, Star2 and Star2.5 will be investigated.

The results show that Star2.5 search with a probing factor of 5 performs best. This player is able to win against advanced human players.

Contents

Preface	i
Abstract	iii
Contents	vi
List of Figures	vii
List of Algorithm Listings	ix
List of Tables	xi
1 Introduction	1
1.1 Research Domain	1
1.2 Problem Statement and Research Questions	2
1.3 Thesis Outline	3
2 Carcassonne	5
2.1 Gameplay	5
2.2 Scoring	7
2.2.1 Cloister	7
2.2.2 Road	7
2.2.3 City	7
2.2.4 Field	7
2.3 Strategies	8
3 Complexity Analysis	11
3.1 State-Space Complexity	11
3.2 Game-Tree Complexity	13
3.3 Comparison to Other Games	15
4 Monte Carlo	17
4.1 Research Done so Far	17
4.2 Implementation	17
4.3 Improvement: Monte-Carlo Tree Search	18
4.3.1 Selection	19

4.3.2 Expansion	20
4.3.3 Simulation	20
4.3.4 Backpropagation	20
5 Expectimax	21
5.1 Background	21
5.1.1 Minimax	21
5.1.2 Alpha-Beta	22
5.2 Research Done so Far	23
5.3 Implementation	23
5.4 Improvements: *-Minimax Search	25
5.4.1 Star1	25
5.4.2 Star2	27
5.4.3 Star2.5	29
5.4.4 Iterative Deepening	30
6 Experiments and Results	31
6.1 Experimental Setup	31
6.2 Monte Carlo	31
6.2.1 Basic Monte Carlo	32
6.2.2 Monte-Carlo Tree Search	34
6.3 Expectimax	36
6.3.1 Move Ordering	36
6.3.2 Evaluation Function	37
6.3.3 Expectimax vs. *-Minimax	38
6.3.4 Star2.5	39
6.3.5 Iterative Deepening	39
6.4 Monte Carlo vs. Expectimax	40
6.5 Experiments against Human Players	42
7 Conclusions and Future Research	43
7.1 Answering the Research Questions	43
7.2 Answering the Problem Statement	44
7.3 Future Research	45
Bibliography	47
Appendix A Tiles in Carcassonne	49
Appendix B Game States	51

List of Figures

2.1	Starting Position	6
2.2	Putting a Meeple on a Tile	6
2.3	Connecting Features	6
3.1	Branching Factor	14
3.2	Frequencies of Branching Factors	15
3.3	Complexities of Different Games	15
4.1	Outline of Monte-Carlo Tree Search	19
5.1	Minimax Tree	22
5.2	Alpha-Beta Tree	23
5.3	Regular Expectimax Tree	24
5.4	Example of Star1	25
5.5	Successful Star2 Pruning.	28
6.1	Monte Carlo: Average Feature Scores	33
6.2	Monte Carlo: Average Meeples	33
6.3	Monte-Carlo Tree Search: Average Feature Scores	35
6.4	Monte-Carlo Tree Search: Average Meeples	35
6.5	Comparison Expectimax, Star1, Star2: Number of Nodes	38
6.6	Comparison Expectimax, Star1, Star2: Used Time	39
6.7	MCTS vs. Star2.5: Average Feature Scores	41
6.8	MCTS vs. Star2.5: Average Meeples	41
1	Overview of the Tiles in Carcassonne	49
2	Game States	51

List of Algorithm Listings

5.1	Expectimax Algorithm	24
5.2	Negamax Algorithm	25
5.3	Star1 Algorithm	27
5.4	Star2 Algorithm	29
5.5	Probing Procedure of Sequential Star2.5	30

List of Tables

3.1	Number of Polyominoes from n=46 up to n=72	12
6.1	Monte Carlo with Different Evaluations	32
6.2	UCT with Various Values for C	34
6.3	Monte-Carlo Tree Search with Different Evaluations	35
6.4	*-Minimax with Different Move Orders	37
6.5	Evaluation Function	38
6.6	Star2.5 with Different Probing Factors	39
6.7	Iterative Deepening	40
6.8	MCTS vs. Star2.5	41

Chapter 1

Introduction

This chapter starts to give a general introduction to research in modern board games (section 1.1). Section 1.2 defines the problem statement and research questions. Finally, an outline of this thesis is described in section 1.3.

1.1 Research Domain

Modern board games are interesting to the field of artificial intelligence (AI). This kind of games forms a transition between classic board games and video games. In the past there is a lot of research done in classic board games. Most of the games are deterministic, have perfect information and are applicable for 2 players.

In games with perfect information the entire state of the game is fully observable to each of the players (e.g., Backgammon).

Deterministic games guarantee that the outcomes only depend on the player's actions. In contrast, non-deterministic games contain an element of chance through shuffling a deck of cards or rolling the dice. Deterministic games like Chess or Checkers have been studied in great depth and with great success. For that reason non-deterministic games get more and more interest from researchers. Most of the modern board games support to play with more than two players. In addition they are often non deterministic and/or have imperfect information.

This thesis researches different AI techniques for a non-deterministic modern board game with perfect information. There is some research already done on the game “Backgammon” [11], which is a non-deterministic and perfect-information game. In this game the chance events are constant. There are also games with changing chances for events during playing. So it would be interesting to research such a game. In this sense “Carcassonne” is a good choice.

Carcassonne is a popular tile-laying board game. Two to five players can participate in one game (with expansion six). It is a non-deterministic game because the players draw tiles randomly during the game. These chance events are not uniformly distributed because several tiles have different frequencies.

After drawing a tile the chance to get the same tile again decreases. In addition, the game has perfect information since the state of the game is always completely visible to each player.

1.2 Problem Statement and Research Questions

The goal of this thesis is to develop a computer player which is able to play the game Carcassonne as good as possible. This research only regards the 2-player variant of Carcassonne. The problem statement of the thesis is:

Can a computer player be built to play the game of Carcassonne as good as possible?

Therefore it is necessary to answer several research questions. The first question is:

1. *What is the complexity of Carcassonne?*

To answer this research question the state-space complexity and the game-tree complexity need to be computed.

2. *Can Monte-Carlo search be used for implementation?*

Monte Carlo is a promising technique to implement a computer player. It is easy to implement and reached good results in previous researched games.

3. *Can Expectimax be used for implementation?*

This approach already works quite well for many non-deterministic board games. So the technique will be applied also for Carcassonne.

4. *In which way can the approaches be improved?*

There are different improvements for each technique. Several of them will be investigated.

5. *Which technique reaches the best result?*

In order to compare the techniques tests have to be performed. Therefore two-player games will be simulated. The two players will use different methods. The player with more points wins the game and who wins most games can be seen as the better one.

1.3 Thesis Outline

The outline of this thesis is as follows:

- Chapter 1 contains a general introduction to research in modern board games. Additionally, this chapter describes the problem statement and the research questions.
- Chapter 2 introduces the modern board game Carcassonne. The rules of the game and some strategies are briefly described.
- Chapter 3 presents the analysis of the complexity of Carcassonne. Therefore the state-space complexity and the game-tree complexity are computed. Moreover, the complexity will be compared to other board games.
- Chapter 4 deals with the implementation of an AI using the Monte-Carlo method. Furthermore, the improved algorithm Monte-Carlo Tree Search is described.
- Chapter 5 presents another technique to implement a computer player – Expectimax. Additionally, the *-Minimax algorithms are given, they improved Expectimax.
- Both methods and there improvements are compared. Chapter 6 reports on the results of the experiments.
- The thesis finishes in chapter 7 with the conclusions. The research questions are answered and the problem statement is evaluated. Besides, this chapter describes suggestions for future research.

Chapter 2

Carcassonne

In this chapter some general information about the game, the rules and some strategies are described.

Carcassonne is a modern tile-laying board game. The game was invented by Klaus-Jürgen Wrede and published in 2000 by Hans im Glück in German and Rio Grande Games in English. It received the “Spiel des Jahres” award in 2001. The game is named after the medieval fortified town of Carcassonne in southern France, famed for its city walls.

2.1 Gameplay

Carcassonne consists of 72 tiles¹ and many figures. Each tile illustrates a section of a landscape. This landscape consists of different features – roads, fields, cities and cloisters. The players earn points by occupying these features with their figures.

The game has spawned many expansions. They contain new tiles, additional figures and features. This thesis considers only the basic game.

At the beginning each player gets 7 figures of one color, called meeples or followers. The basic game contains only “normal” meeples. Therefore the thesis always uses “meeple” instead of “follower” or “figure”. The game starts with a single tile face up (see figure 2.1). The remaining 71 tiles are covered up.

Each turn consists of two parts. First, the player draws a tile and places it to the existing terrain so that it matches neighbouring features: roads must connect to roads, fields to fields, and cities to cities.

The second step is optional. The player can put a meeple on the just-placed tile. The meeple must be placed in a feature which is not yet occupied by another meeple (see figure 2.2). Despite this it is possible to get multiple meeples in one feature: figure 2.3 shows how to connect two occupied feature parts into a single feature by adding another tile.

¹An overview of the possible tiles can be found in appendix A.



Figure 2.1: Starting tile (always the same) and meeples.



Figure 2.2: The road and the large field are already occupied so there are only two possibilities to put the meeple: the small field and the city.



Figure 2.3: Each city part is already occupied, the new tile connects them.

A completed feature scores points for the player with most meeples in it. The game is over when the last tile has been placed (of course the player can put a meeple on it). At that time also all incomplete features score points for the players with most meeples in them. The player with the highest score wins.

2.2 Scoring

If a feature is closed it will be scored. After scoring all of the meeples in this feature are returned to their owner. They can be re-used in the following turns. At the end of the game all incomplete features are also scored.

The following subsections describe the scoring for each feature.

2.2.1 Cloister

A cloister is completed when it is surrounded by eight tiles. The meeple on the cloister earns 9 points for its owning player – one point for each neighbouring tile and the cloister itself.

If a cloister is not completed at the end of the game, it awards points based on the number of existing neighboring tiles and additionally one point for the cloister itself.

2.2.2 Road

A road is completed when there is no unfinished edge from which to expand. Completed roads and incomplete roads score identically: The players with most meeples on a road get one point per tile that contains a part of this road.

2.2.3 City

A city is also completed when there is no unfinished edge from which to expand. Completed cities score two points per tile and additionally two points for each pennant.

At the end of the game incomplete cities score one point for each tile and also one point per pennant.

2.2.4 Field

Fields are only scored at the end of the game. The players with the most meeples on the field earn three points for each completed city bordering the field.

2.3 Strategies

In general there are some simple strategies to use in each turn [3]:

- try to get cheap points every turn by instantly closing a small feature (city or road which consists of 2 tiles)
- build existing cities, roads, and cloisters toward completion
- try to get the best field positions for end game scoring
- try to connect to a bigger city, road or field and take it over by having more meeples in it than the opponent
- place a tile to block a city, cloister or road of the opponent

A combination of these points forms a good strategy. An experienced player first looks at his current on-board positions and decides, whether the drawn tile may be useful to close a feature, to expand a feature, to create a new feature or to block others from getting into his features. If he cannot expand, cannot close and cannot create, then the player may use this tile to block other players.

But which feature has the biggest advantage? To answer this question each feature is determined for advantages and disadvantages.

Feature	Advantage	Disadvantage
Cloister	guarantees 5-9 points on average without doing much work	possible to get meeple stuck, if an opponent blocks one of the 8 neighboured positions
Field	gives many points (in many games there is just one central field, and whoever controls it can win the game)	meeple is placed until the end of the game, because all fields are only scored when the game is over
Road	easier to complete than cities because maximum 2 open ends	in contrast to cities lower score
City	gets additional points with pennants	sometimes difficult to close, opponent can easily connect

According to these advantages and disadvantages a ranking of the features can be defined. It is not an official ranking but a personal opinion.

Roads are the weakest; although they are easy to build they score only one point for each tile.

Cloisters are more effective. They also score only one point per tile but in most cases 4-5 tiles are already adjacent to the cloister at the moment when it is put to the landscape. The cloister terrain can be finished while working on other features. But the main disadvantage is that the meeple can get stuck.

Cities are the next ones, because they earn 2 points for each tile and for each pennant.

Fields are the most effective features. They get 3 points for each bordering city and can become very large. Because the meeple on a field stays there until the game's end it is important not to set too many meeples on fields and not too early. It is quite difficult to choose the right moment.

Chapter 3

Complexity Analysis

This chapter examines the complexity of Carcassonne. The complexity consists of two different factors: the state-space complexity and the game-tree complexity [1].

In section 3.1 the state-space complexity is computed. The game-tree complexity is determined in section 3.2. In the last section, 3.3, the computed complexities are compared to known complexities of other games.

3.1 State-Space Complexity

The state-space complexity is the total number of different board configurations which are possible in the game. To calculate an exact number is often very difficult, so a lower bound can be calculated instead. In Carcassonne the calculation is also very complicated. In principle, there are three steps to calculate the state-space complexity for Carcassonne. Firstly, the number of shapes is determined, which can be formed with up to 72 tiles. Then the placement of the tiles within the shapes will be regarded, but without meeples. Lastly, the opportunity to put a meeple on one of the features of each tile is regarded.

The first part is to determine how many shapes are possible for each number of tiles in the game. For example there is only one shape for 2 tiles (both next to each other), 2 possibilities for 3 tiles (a corner or a line) and so on. These shapes are called polyominoes or n -ominoes. Basically, there are two types of polyominoes: fixed and free polyominoes. While a mirrored or a rotated shape is one and the same free polyomino, it is multiple counted for fixed polyominoes. For Carcassonne the number of free polyominoes need to be calculated, because a mirrored or rotated map leads to the same game state. A free polyomino with no rotation or reflection corresponds to 8 distinct fixed polyominoes. Therefore, the number of fixed polyominoes is approximately 8 times the number of free polyominoes. The higher the n the lower the probability to have symmetries and the more exponentially accurate is this approximation [19].

n	free polyominoes	fixed polyominoes
46	$8.6 \cdot 10^{24}$	68557762666345165410168738
47	$3.4 \cdot 10^{25}$	272680844424943840614538634
48	$1.4 \cdot 10^{26}$	1085035285182087705685323738
49	$5.4 \cdot 10^{26}$	4319331509344565487555270660
50	$2.2 \cdot 10^{27}$	17201460881287871798942420736
51	$8.6 \cdot 10^{27}$	68530413174845561618160604928
52	$3.4 \cdot 10^{28}$	273126660016519143293320026256
53	$1.4 \cdot 10^{29}$	1088933685559350300820095990030
54	$5.4 \cdot 10^{29}$	4342997469623933155942753899000
55	$2.2 \cdot 10^{30}$	17326987021737904384935434351490
56	$8.6 \cdot 10^{30}$	69150714562532896936574425480218
57	$3.5 \cdot 10^{31}$	$2.8 \cdot 10^{32}$
58	$1.4 \cdot 10^{32}$	$1.1 \cdot 10^{33}$
59	$5.6 \cdot 10^{32}$	$4.5 \cdot 10^{33}$
60	$2.2 \cdot 10^{33}$	$1.8 \cdot 10^{34}$
61	$8.9 \cdot 10^{33}$	$7.1 \cdot 10^{34}$
62	$3.6 \cdot 10^{34}$	$2.8 \cdot 10^{35}$
63	$1.4 \cdot 10^{35}$	$1.1 \cdot 10^{36}$
64	$5.7 \cdot 10^{35}$	$4.6 \cdot 10^{36}$
65	$2.3 \cdot 10^{36}$	$1.8 \cdot 10^{37}$
66	$9.1 \cdot 10^{36}$	$7.3 \cdot 10^{37}$
67	$3.6 \cdot 10^{37}$	$2.9 \cdot 10^{38}$
68	$1.5 \cdot 10^{38}$	$1.2 \cdot 10^{39}$
69	$5.8 \cdot 10^{38}$	$4.7 \cdot 10^{39}$
70	$2.3 \cdot 10^{39}$	$1.9 \cdot 10^{40}$
71	$9.4 \cdot 10^{39}$	$7.5 \cdot 10^{40}$
72	$3.8 \cdot 10^{40}$	$3.0 \cdot 10^{41}$

Table 3.1: Number of polyominoes with 46 up to 72 squares. The bold values are accurate numbers taken from [15].

The problem is that there is no formula to enumerate polyominoes of a given size. Free polyominoes have been enumerated accurately up to $n = 45$ [23] and fixed polyominoes up to $n = 56$ [15]. If n is greater, only an approximation exists to calculate fixed polyominoes. The number of fixed polyominoes A_n can be calculated:

$$A_n \approx \frac{c \cdot \lambda_n}{n}$$

where $c = 0.3169$ and $\lambda = 4.0626$ [5, 14]. These values are only estimators. To get an estimator of free polyominoes the result need to be divided by 8. Table 3.1 shows the number of polyominoes which can be built with up to 72 squares. In other words, the table shows the number of different shapes which can be built with up to 72 tiles from Carcassonne.

The numbers of fixed polyominoes up to $n = 72$ sum up to $5 \cdot 10^{40}$. This value is a highly underestimated lower bound of the state-space complexity. Due to the facts that games with such high state-space complexity are unsolvable and the calculation of the next steps (placement of tiles and meeples) are very complicated, this result is sufficient.

3.2 Game-Tree Complexity

The game-tree complexity is the number of different games which can be played. It indicates the total number of terminal nodes in the game tree. Often it is impossible to calculate the game-tree complexity exactly, but an estimate can be made. Therefore two values have to be known: the branching factor and the game length. The branching factor indicates from how many different possible moves a player can choose on average. Moves can differ by

- (1) the position of the tile on the map
- (2) the rotation of the tile
- (3) whether a meeple is put to the tile and in which feature

The game length describes the average number of plies until the game is over. A ply consists of one move which is taken by one of the players.

The game-tree complexity can be computed by raising the average branching factor (possible moves) b to the power of the average game length d (depth of the tree) multiplied with the average branching factor of the chance nodes c to the power of the game length d :

$$C = b^d \times c^d$$

The length of the game is always the same, because there are 71 tiles¹ which have to be drawn. Each player draws only one tile per move, so the length of the game is 71.

To determine the branching factors averages are taken. There are two branching factors needed: the chance events (number of different tiles which can be drawn) and the possible moves for a drawn tile. For the branching factor of chance events an average can be taken. There are 24 different tiles and totalling 71 tiles. That means on average after 3 plies the number of different tiles decreases by 1. So the average branching factor of chance events is:

$$c^d = \prod_{i=2}^{24} i^3$$

The second branching factor, the average number of possible moves, can be determined by storing the number of possible moves for each ply while playing

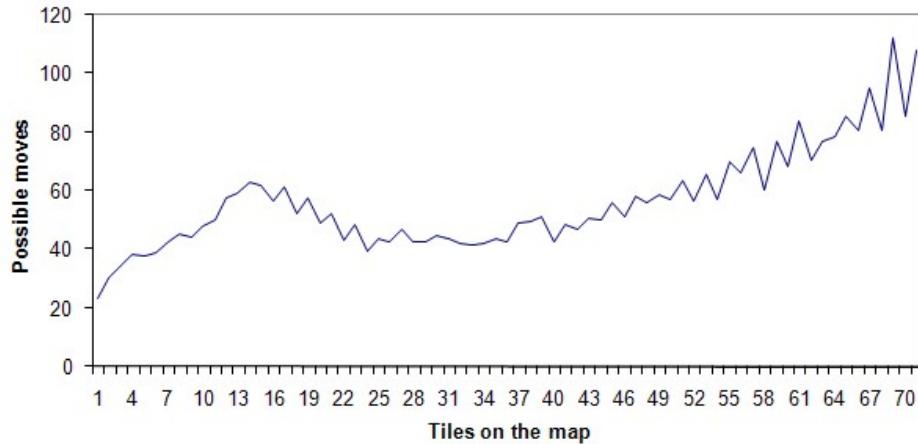


Figure 3.1: The average number of possible moves for several plies. Therefore 200 games with two *-Minimax players (depth=3) were simulated.

several games. Figure 3.1 shows the average branching factor of several plies after simulating 200 games (*-Minimax player with depth 3).

The average branching factor of the first ply is 22. Until ply 14 the average branching factor increases, because the map grows and every player is able to put a meeple on the tile. But there is a peak on ply 14 ($b = 63$). This is because every player gets 7 meeples at the beginning. If both players put a meeple in each turn and get no meeple back, there would be no more meeple in ply 15 (8th move of player 1) and ply 16 (8th move of player 2). So the player can only decide where and how to place the tile on the map, but can not put a meeple. That is why the branching factor decreases until $b = 39$. In the last phase the branching factor grows again. There are two possible reasons. In general closer to the end of the game the number of possible moves increases, if the player can put a meeple, because of the growing map. The second reason is, that more and more meeples go back to their players because features are closed. So the chance of having no meeple decreases. For that reason the average branching factors of the last plies are greater than 100.

Figure 3.2 shows the frequencies of different branching factors. There is a peak on the branching factor on 32. The minimum branching factor is 1 and the maximum was 865 during the 200 simulations. But figure 3.2 shows that branching factors which are greater than 150 occur very rarely.

Finally, the average branching factor is 55 and the game-tree complexity can be computed:

$$C = 55^{71} \times \prod_{i=2}^{24} i^3 \approx 3.7 \cdot 10^{123} \times 2.4 \cdot 10^{71} \approx 8.8 \cdot 10^{194}$$

¹Totally there are 72 tiles but one tile is the starting tile.

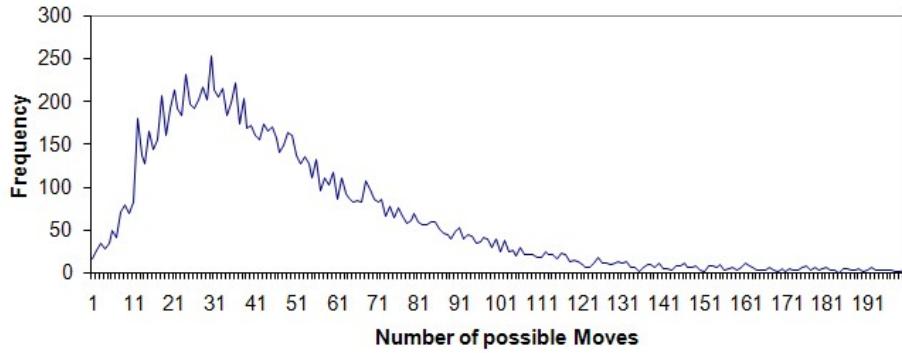


Figure 3.2: Frequencies of branching factors in 200 games played by two *-Minimax players (depth=3).

3.3 Comparison to Other Games

Finally, it is interesting to compare the calculated complexities to the complexities of other games. For example, the state-space complexity of Chess is 10^{46} and the game-tree complexity is 10^{123} . Go has a state-space complexity of 10^{172} and a game-tree complexity of 10^{360} . Figure 3.3 presents the complexities of different games.

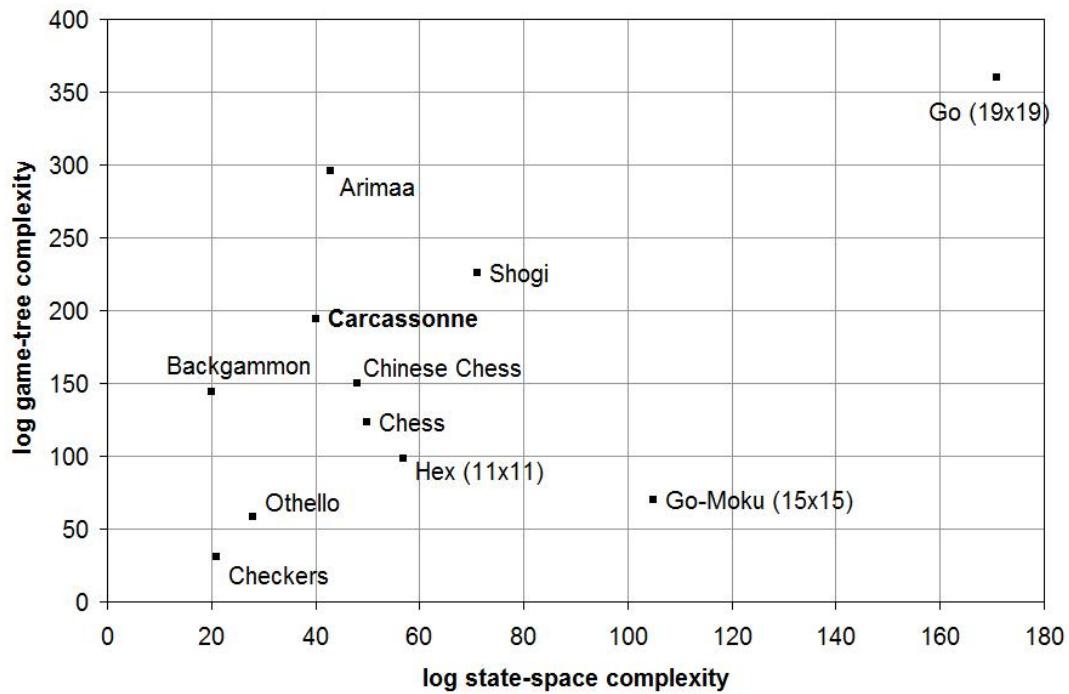


Figure 3.3: The complexities of different games.

The figure shows that the game-tree complexity of Carcassonne is smaller than the game-tree complexity of Shogi, but greater than that one of Chinese

Chess and Backgammon. Since the game-tree complexity is very high a full-depth search in Carcassonne is impossible. The state-space complexity of Carcassonne is at least 10^{40} . The real value will be much higher, because there are a lot of multiplicators missing. So the assumption is that the state-space complexity is at least equal to that of Arimaa. Due to the large state-space complexity Carcassonne is not solvable.

Chapter 4

Monte Carlo

An approach to implement a computer player for Carcassonne is Monte-Carlo search. The reason for choosing Monte Carlo is that the basic algorithm does not need any strategic knowledge. The only requirement is knowledge of the rules and the actual state of the game.

This chapter presents the use of this algorithm. First, section 4.1 describes research on the Monte-Carlo technique in other games. Section 4.2 describes the basic algorithm and its implementation for Carcassonne. Section 4.3 gives an improved algorithm.

4.1 Research Done so Far

The Monte-Carlo technique has been studied by Brügmann during the development of an AI-player for the game of Go [6]. Furthermore, it is applied in many other deterministic classic board games, such as Ataxx [10], Othello [18] or Amazons [17]. The implemented computer players play quite well on amateur or intermediate level.

However, it is quite hard to improve the performance of a Monte-Carlo player to a high level. One possible improvement is to reduce the number of possible moves by applying domain knowledge [10, 18]. The Go program MoGo uses another improvement, the Monte-Carlo Tree Search [25]. By using this Monte-Carlo technique the Go-AI reached a strong player level and became one of the strongest AIs for Go.

Carcassonne differs in two aspects from the researched games. Carcassonne is a multi-player game and it is non deterministic. However, in the following sections we limit ourselves to the 2-player variant of Carcassonne.

4.2 Implementation

The basic algorithm of Monte Carlo works quite simple. The current state is the starting position for the Monte-Carlo algorithm. At first, all possible moves at this position are determined. At least there is always one possible

move, because if a tile cannot be put to the map a new tile is drawn and the other tile returns to the pool of tiles. If there is exactly one possible move, then this one is the best one. In the case that there are more moves possible the Monte-Carlo technique decides which one is the best. For each possible move the algorithm first plays this move and finishes the game by selecting the following moves for all players randomly.

How many games for each possible move are simulated depends on the number of possible moves and the maximum time before the algorithm has to make a decision. So the maximum time is divided among the number of possible moves. This indicates how long the algorithm can simulate games for each possible move.

After running a simulation the algorithm evaluates the simulated game. There are two possibilities to score a move: first, the score is 1 if the Monte-Carlo player won the game, 0 for a draw and -1 if the Monte-Carlo player lost. This technique does not take into account whether a game was won by a clear margin or not. The second approach takes this margin into account. So the score is computed by subtracting the opponent score S_{Opp} from the score of the Monte-Carlo player S_{MC} (see section 2.2 for scoring):

$$S = S_{MC} - S_{Opp}$$

So, if S is positive the Monte-Carlo player won the game, else the opponent player did. If the Monte-Carlo player plays the game with more than one opponent, there are also two methods for the score. Either the score is determined by the difference between the best other player and the Monte-Carlo player or by the difference between the total score of all opponents and the Monte-Carlo player. The different opportunities for evaluating the moves are compared in section 6.2.1.

Finally, for each possible move the average result of all simulations for this move is computed. Monte Carlo decides to play the move with the highest average score.

4.3 Improvement: Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) extends the basic Monte-Carlo algorithm. The basic Monte-Carlo algorithm selects a move randomly to finish a simulation. The disadvantage is that unlikely chance events (in Carcassonne drawing a tile) are played with the same probability as more likely chance events. Also bad moves (putting a tile and maybe a meeple to the map) are played about as often as good moves. MCTS is a best-first search algorithm, so the algorithm is able to play better moves more often than bad ones.

In MCTS a game tree is constructed that stores statistics about map positions, which are represented by the nodes. The stored statistics can be used for improving the search, e.g., selecting the most promising move. In general a node consists of at least two variables: the number of visits and the number

of won games. MCTS starts with a tree containing only the root node. For Carcassonne there are two different nodes needed. Move nodes are used if the player must choose a move. Tile nodes correspond to situations in which a tile has to be drawn.

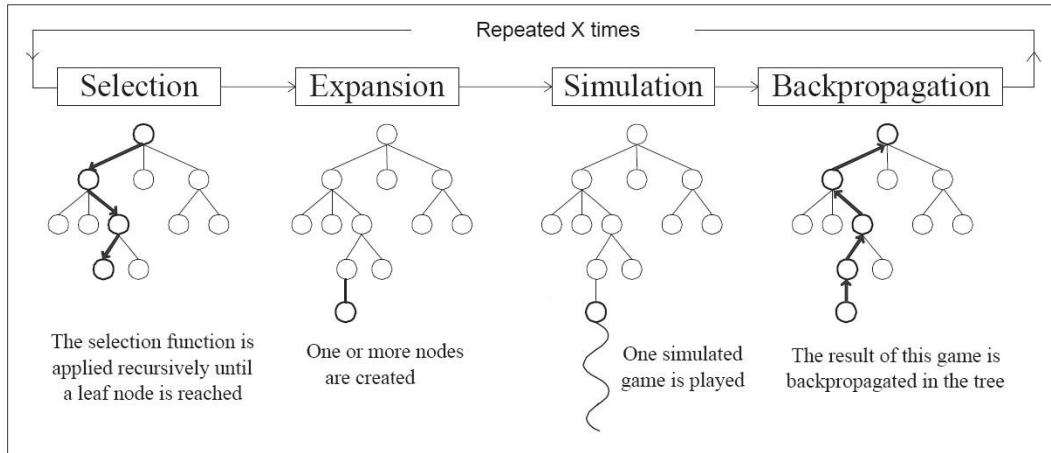


Figure 4.1: Outline of Monte-Carlo Tree Search [7].

Monte-Carlo Tree Search consists of four steps, repeated as long as there is time left. Figure 4.1 illustrates these phases. The first step selects a sequence of nodes from the root until a leaf node L of the tree is reached. The second phase expands the tree by storing one or more children of L in the tree. The third step simulates the game until it is finished. Finally, the result of this simulated game is propagated back to the root of the tree. When the time is over, the move played by the program is the child of the root with the highest win rate. The following subsections give a more detailed description of the four steps.

4.3.1 Selection

This phase selects one of the children of a given node. If the children are tile nodes, the next child is chosen randomly, but the more tiles of a tile type are in the remaining pack the greater the chance to choose the child of this tile type. If the children are move nodes the selection strategy must decide to explore more nodes or to exploit the most promising node so far. The strategy controls the balance between exploitation and exploration. On the one hand, the task often consists of selecting the move that leads to the best result so far (exploitation). On the other hand, the less promising moves must still be tried, due to the uncertainty of the evaluation (exploration). This problem of balancing of exploitation and exploration is similar to the Multi-Armed Bandit problem [8].

One possible strategy to choose the next child is Upper Confidence bounds applied to Trees (UCT) [16]. This strategy chooses the move i which maximizes

the following formula:

$$\frac{w_i}{v_i} + C \times \sqrt{\frac{\ln v_p}{v_i}}$$

where $\frac{w_i}{v_i}$ is the win rate of node i . w_i is the number of wins after visiting this node and v_i the number of visits of this node. v_p is the number of visits of the parent node. C is a constant value, which has to be tuned experimentally. If C is small the algorithm exploits the result earlier and if C is higher more children will be explored. Section 6.2.2 shows the results of different values for C and decides on the best value.

If some children of a node were not visited yet, a UCT value cannot be calculated. In that case one of these children is chosen and stored in the tree (see 4.3.2 Expansion).

4.3.2 Expansion

This step decides whether nodes will be added to the tree. The simplest rule is to add one node per simulated game [9]. The expanded node corresponds to the first position encountered that was not stored yet. This rule is used for the implementation of a MCTS player for Carcassonne.

4.3.3 Simulation

This step starts when a position is entered which is not part of the tree yet. The strategy on this step selects moves in self-play until the end of the game. The moves can be chosen plain randomly or also pseudo-randomly. The Carcassonne-MCTS player chooses a move plain randomly and a tile is chosen with regard to the number of remaining tiles of this type. So a tile which exists more often has a greater chance to be drawn.

The value of the result can be determined in the same way as for basic Monte Carlo. Either a win is scored with 1 and a loss with -1 or the difference of the score of the players can be used. Section 6.2.2 shows which method reaches the better result.

4.3.4 Backpropagation

This step propagates back the result of the simulated game to the nodes which are visited during playing. Each node stores the number of visits and the sum of the results of all played games where this node participated. An average result of a node can be calculated by dividing this sum by the number of visits of this node.

Chapter 5

Expectimax

Another algorithm to implement a computer player for Carcassonne is Expectimax search. This technique is based on the Minimax [20] search algorithm which can be used for deterministic games. Expectimax expands Minimax so that it can be applied to games with chance.

This chapter describes the usage of this algorithm. Firstly, section 5.1 explains the background mechanisms – Minimax and Alpha-Beta. Section 5.2 describes research results of using Expectimax search and its enhancements in other games. Section 5.3 presents the basic algorithm and its implementation for Carcassonne. Section 5.4 gives several enhanced algorithms.

5.1 Background

It is advisable to understand the following techniques before reading the sections about Expectimax.

5.1.1 Minimax

Minimax is used to compute a value of a game state in a two-player deterministic game. This algorithm generates all possible game states after playing a certain number of plies. This means Minimax is a full-width tree search. In most games it is not possible to search the complete tree, that is why often there is a fixed depth. The leaf nodes are scored by an evaluation function. The player to move chooses the move with the maximum value and the opponent player will perform that move which minimizes the score for the max player. That means this algorithm always plays optimally.

Figure 5.1 shows a sample minimax tree with a 3-ply search. The root node is a max node and represents the current game state. The edges leading to the successors represent the execution of the possible moves and lead to new game states. Then the opponent player, the min player, has to move. This process goes on until a leaf node is reached. After scoring the leaf nodes the max player chooses the move where the child has the highest value and the min player chooses the move with the lowest value. In that way the best node

on the root node can be determined. In the figure the first move which has the value 4 is considered to be the best one.

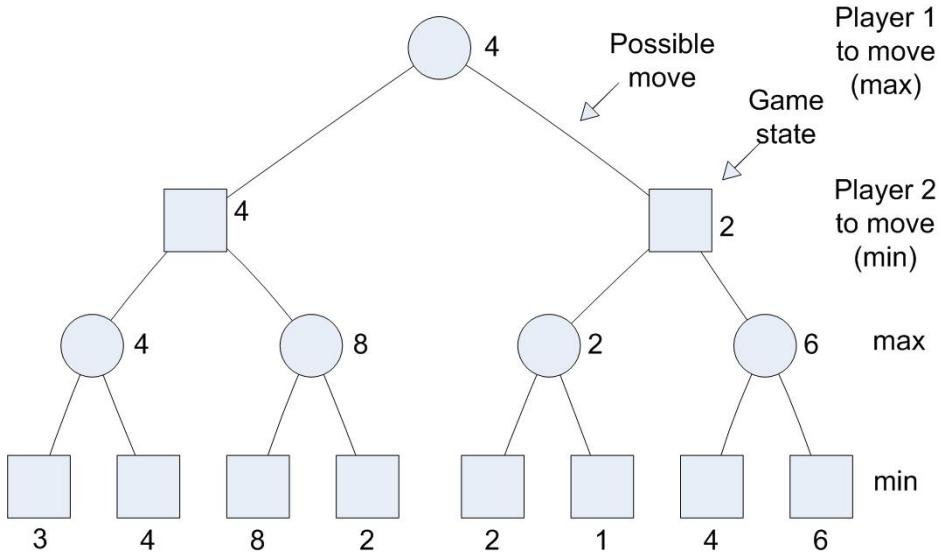


Figure 5.1: A minimax tree.

The Minimax algorithm is computationally expensive in most games because the complexity depends on the branching factor. For a fixed search depth d the complexity of the algorithm is $O(b^d)$.

5.1.2 Alpha-Beta

Searching every node is computationally very expensive and not always necessary. Alpha-beta is a pruning technique which cuts some branches off. Therefore, it defines a search window for every node. If the value of this node falls outside of this window the remaining successors need not be searched and cut-offs occur.

Figure 5.2 shows the same tree as figure 5.1, now using alpha-beta search. When the third leaf node (value = 8) is searched its parent knows it has a value of at least 8, because it is a max node. The parent of this max node which has a value of 4 until now will not choose a value which is greater than 4 because it is a min node. For that reason the other successor of the node with the value greater or equal 8 does not need to be searched. The same holds for the second child of the root node. Because it is a min node and the value of the first of its successors is 2, the value of this node is less or equal 2. The root node has already value 4 so it will not choose a value less than 4 because it is a max node. Therefore, there is a cut-off on the second branch of the node with the value less or equal 2 because its value does not matter.

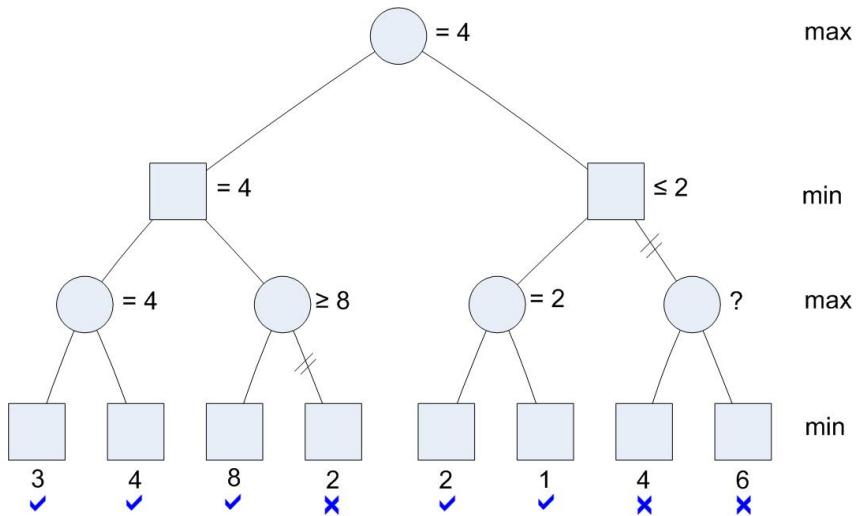


Figure 5.2: An alpha-beta tree.

5.2 Research Done so Far

A typical deterministic board game is Backgammon. There has been done much research on this game. A promising search technique is Expectimax and its enhancement *-Minimax. Hauk, Buro and Schaeffer investigated Expectimax, Star1 and Star2 [11]. They figured out that Star2 with a search depth of 5 works most efficiently. One proposal for future research is to apply this technique for Carcassonne. They mentioned that a search depth of 5 or 6 could result in expert play of that game.

Furthermore, there exists another research which refers to Carcassonne as a possible future research [24]. This thesis researches the game of Dice. In contrast to Carcassonne and Backgammon the chance events in this game are uniformly distributed.

5.3 Implementation

Expectimax [13] behaves exactly like Minimax. In this thesis only regular Expectimax trees are taken into account. That means there is a chance event in each ply. The chance event in Carcassonne is to draw a new tile which must be placed to the existing landscape. The search tree adds chance nodes between the min and max nodes. Each of these chance nodes has a probability to be chosen. In Carcassonne the chances are non-uniform distributed. The probabilities of all successors of a min or max node sum up to one. The Expectimax value of a chance node is determined by the sum of the minimax value of its successors multiplied by their probability. So for a state s the Expectimax value is calculated with the following formula:

$$\text{Expectimax}(s) = \sum_i P(\text{child}_i) \cdot U(\text{child}_i)$$

$child_i$ describes the i th successor of s , $P()$ is the probability that this successor is chosen and $U()$ is the minimax value of this successor.

Figure 5.3 shows a regular Expectimax tree. The value of the upper chance node can be calculated as follows: $0.25 \times 0.8 + 0.75 \times 2 = 1.7$.

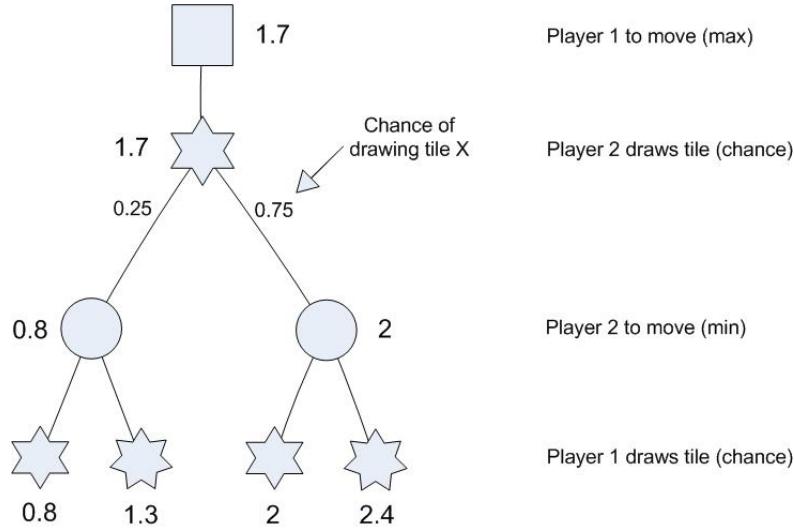


Figure 5.3: A regular Expectimax tree.

Because the complexity depends on the branching factor the computational time for Expectimax in Carcassonne is very high. Similarly to Minimax the complexity of the algorithm is: $O(b^d c^{d-1})$, where c is the branching factor of the chance nodes. So only a depth of 2 or 3 is realistic for Carcassonne because of the high branching factor. Each depth step means one ply. Searching to depth 1 only considers the moves which are possible from the root node. That means the player only sees the result after playing this move.

To avoid duplicated code, because always a maximizing and a minimizing function is needed, there is another algorithm which is based on Minimax. This is called Negamax search and it always maximizes the values. For that reason the Negamax search is used as the basic algorithm of Expectimax.

```

float expectimax (int depth){
    if (gameIsFinished() || depth==0) return getValue();
    score = 0;
    for (Tile newTile : getDifferentTiles()){
        setNextTileInGame(newTile);
        value = negamax(depth);
        undoSetNextTile();
        score += value * numberTiles(newTile)/numberRemainingTiles();
    }
    return score;
}

```

Listing 5.1: Expectimax Algorithm

Listing 5.1 shows the pseudocode of the Expectimax algorithm. Because chance nodes do not count for depth, the depth is not decreased when calling the `negamax` procedure, which can be found in listing 5.2. Negamax has only max nodes, so the values of min nodes must be negated. For that reason the

original Negamax algorithm calls itself but with a negative sign. In cooperation with Expectimax it must call the negated `expectimax` algorithm instead of itself.

```
float negamax (int depth){
    score = -Infinity;
    for (Move move : getPossibleMoves()){
        executeMove(move);
        value = -expectimax(depth-1);
        undoMove();
        score = max(value, score);
    }
    return score;
}
```

Listing 5.2: Negamax Algorithm

5.4 Improvements: *-Minimax Search

Also for non-deterministic games there are several pruning techniques. Bruce Ballard [4] introduces *-Minimax algorithms which are based on the Alpha-Beta technique. In this research Star1, Star2 and Star2.5 are investigated.

5.4.1 Star1

Star1 is a pruning technique for chance nodes by using the search windows of the min and max nodes. A cut-off occurs, if the weighted sum of all children of a chance node falls outside the $\alpha\beta$ window. If a successor has no value so far the theoretical lower value L (successor is max node) or the theoretical upper value U (min node) must be used. These values are the minimum and maximum of the evaluation function.

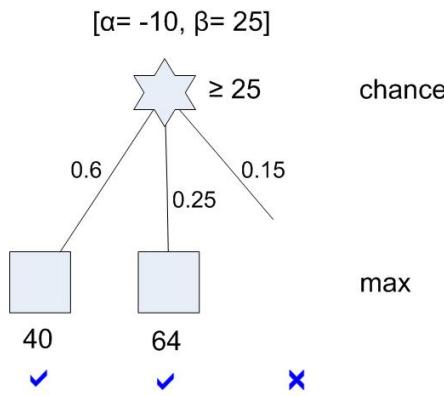


Figure 5.4: Example of Star1.

Figure 5.4 illustrates the proceeding of the Star1 algorithm at a chance node with 3 successors. The values of the evaluation function are between $L = -100$ and $U = 100$. The $\alpha\beta$ window of the upper chance node is $[-10, 25]$. That means, if the value of this node lies between these bounds an exact value must

be calculated and if the value falls outside this window an upper or lower bound is sufficient. The first two successors are already determined. The last successor gets the lower bound -100 as a value, because it is a max node. So an estimator for the root node can be calculated: $0.6 \times 40 + 0.25 \times 64 + 0.15 \times (-100) = 25$. That means the value of the chance node is at least 25. This example shows that it is important to order the chance events so that the most probable events come first.

If the i th successor of a chance node is reached, the first $i - 1$ successors are already searched and have a value. An upper bound for the chance node can be calculated by setting the theoretical upper bound U to all remaining successors. Similarly, to calculate a lower bound the remaining successors must be set to the theoretical lower bound L . A cut-off occurs, if it can be proven, that the Expectimax value falls outside the search window. If there are equal chances the following formulas are used:

$$\frac{(V_1 + \dots + V_{i-1}) + V_i + U \times (N - i)}{N} \leq \text{alpha}$$

$$\frac{(V_1 + \dots + V_{i-1}) + V_i + L \times (N - i)}{N} \geq \text{beta}$$

where N is the total number of successors and V_i is the value of the i th successor. The rearranged formulas give an alpha value A_i and a beta value B_i for the i th successor:

$$A_i = N \times \text{alpha} - (V_1 + \dots + V_{i-1}) - U \times (N - i)$$

$$B_i = N \times \text{beta} - (V_1 + \dots + V_{i-1}) - L \times (N - i)$$

For Carcassonne the formulas must be modified because the chances are not equal as long as the number of the tiles for a type is different from another tile type. Therefore the probabilities of the successors need to be taken into account:

$$(P_1 \times V_1 + \dots + P_{i-1} \times V_{i-1}) + P_i \times V_i + U \times (1 - P_1 - \dots - P_i) \leq \text{alpha}$$

$$(P_1 \times V_1 + \dots + P_{i-1} \times V_{i-1}) + P_i \times V_i + L \times (1 - P_1 - \dots - P_i) \geq \text{beta}$$

where P_i is the probability that a tile of type i is drawn.

The alpha and beta bound of the i th successor can also be calculated:

$$A_i = \frac{\text{alpha} - (P_1 \times V_1 + \dots + P_{i-1} \times V_{i-1}) - U \times (1 - P_1 - \dots - P_i)}{P_i}$$

$$B_i = \frac{\text{beta} - (P_1 \times V_1 + \dots + P_{i-1} \times V_{i-1}) - L \times (1 - P_1 - \dots - P_i)}{P_i}$$

The algorithm can update several parts of the formula separately so computational time can be saved. The value part ($P_1 \times V_1 + \dots + P_{i-1} \times V_{i-1}$) is initialized by $X_1 = 0$ and updated by $X_{i+1} = X_i + P_i \times V_i$. The probability part ($1 - P_1 - \dots - P_i$) can be initialized with $Y_0 = 1$ and updated by $Y_i = Y_{i-1} - P_i$. That is why the implemented formulas for A_i and B_i are:

$$A_i = \frac{\alpha - X_i - U \times Y_i}{P_i}$$

$$B_i = \frac{\beta - X_i - L \times Y_i}{P_i}$$

These formulas are applied in the pseudocode of the Star1 algorithm which is presented in listing 5.3. The same `negamax` from listing 5.2 is used, but instead of calling `expectimax` procedure `star1` must be called. Additionally, `negamax` prunes if a value falls outside the $\alpha\beta$ window.

```
float star1(float alpha, float beta, int depth){
    if (gameIsFinished() || depth==0)    return getValue();
    cur_x = 0;
    cur_y = 1;
    for (Tile newTile : getDifferentTiles()){
        probability = numberOfTiles(newTile)/numberOfRemainingTiles();
        cur_y -= probability;
        cur_alpha = (alpha-cur_x-U*cur_y)/probability;
        cur_beta = (beta-cur_x-L*cur_y)/probability;
        ax = max(L, cur_alpha);
        bx = max(U, cur_beta);
        setNextTileInGame(newTile);
        value = negamax(ax, bx, depth);
        undoSetNextTile();
        if (value >= cur_beta) return beta;
        if (value <= cur_alpha) return alpha;
        cur_x += probability * value;
    }
    return cur_x;
}
```

Listing 5.3: Star1 Algorithm

Star1 returns the same result as Expectimax, but uses fewer node expansions to obtain the result. In the worst case, no cut-off occurs, which means that the number of nodes which have to be searched is the same as in Expectimax. A reason of this weak pruning is that this algorithm is very pessimistic. It assumes that all unseen nodes have a worst-case value. For that reason most of the children must be searched, because if many children have a worst-case value the value is highly correlated.

5.4.2 Star2

This algorithm is an extension of Star1 and can be applied only on regular Expectimax trees. The advantage is that the algorithm knows what kind of node will follow. If the Negamax algorithm is used a chance node will always be followed by max nodes and vice versa. Star2 begins with a probing phase,

which is a speculative phase. Instead of searching each child of each successor, that phase checks only one child of each possible successor. The value of that child becomes a lower bound of this successor. In that way each successor of the node gets a lower bound. These lower bounds can be used to prove that the value of the node falls outside the $\alpha\beta$ window. Therefore all unseen children get this bound instead of L . If the value falls outside the window all these unseen successors can be skipped. If the probing phase does not exceed beta, all of the successors must be searched, but the search window can be narrowed by using the more accurate lower bounds. For the search phase the Star1 algorithm is used.

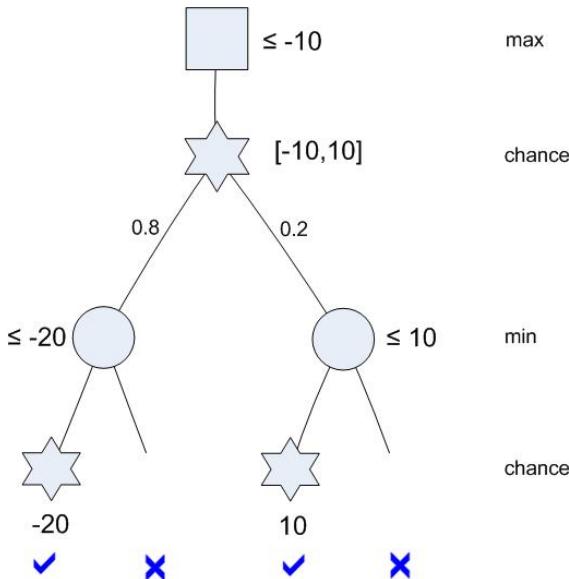


Figure 5.5: Successful Star2 Pruning.

Figure 5.5 shows an example of a successful Star2 pruning. The theoretical lower bound is -100 and the theoretical upper bound is 100 . The $\alpha\beta$ window of the upper chance node is $[-10, 10]$. In the first phase of the algorithm the first possible chance outcome is probed. The value of the first lower chance node is -20 . Thus, the value of the parent min node is at least -20 . Now the theoretical upper bound for the upper chance node can be calculated as follows: $-20 \times 0.8 + 100 \times 0.2 = 4$. A cut is not possible yet, because the value is inside the $\alpha\beta$ window $[-10, 10]$. The theoretical window of the upper chance node is now $[-100, 4]$. After searching the first child of the second successor, this window is updated to $[-100, -14]$. This falls outside the $\alpha\beta$ window. For that reason the remaining children can be pruned.

Move ordering strongly influences the benefit of Star2. The best result can be reached if the outcomes of the first children are large enough so that the Expectimax value exceeds beta. That means the most promising move should be investigated first. Also in the case that this fails, this algorithm has another advantage. The determined lower bounds can be used in the Star1 search instead of using the theoretical lower bound L , so the search window is

smaller.

The calculation of A_i in the probing phase is not necessary, because Negamax is used and it has only to be proven, that the weighted sum of the lower bound of the successors exceeds beta. So only a current beta B_i needs to be calculated. For that calculation the formula of Star1 can be used. In the search phase the calculation of A_i can also be used but for B_i the formula must be modified:

$$B_i = \frac{\text{beta} - X_i - W_i}{P_i}$$

where $W_i = W_{i+1} + \dots + W_N$ is the sum of the probing values for nodes that are not searched until now. W_i is used because it is a better estimator for the lower bound of this node than L .

Listing 5.4 illustrates the pseudocode of the Star2 probing phase. The function `nProbe` is similarly to the `negamax` function of the Star1 algorithm. But instead of searching each child, only the first child is investigated. For that reason the result is only a lower bound. The search phase is the same as in the Star1 algorithm, but for the calculation of `cur_beta` instead of `U*cur_y` the value of `cur_w` must be used.

```
float star2(float alpha, float beta, int depth){
    if (gameIsFinished() || depth==0)    return getValue();

    //probing phase
    cur_x = 0;
    cur_y = 1;
    cur_w = 0;
    cur_alpha = (alpha-U*(1-firstPossibleTile().probability))/firstPossibleTile().probability;
    ax = max(L, cur_alpha);
    for (Tile newTile : getDifferentTiles()){
        probability = number_of_tiles(newTile)/number_of_remaining_tiles();
        cur_y -= probability;
        cur_beta = (beta-L*cur_y-cur_x)/probability;
        bx = max(U, cur_beta);
        setNextTileInGame(newTile);
        value = nProbe(ax, bx, depth);
        undoSetNextTile();
        cur_w += value;
        if (value >= cur_beta) return beta;
        cur_x += probability * value;
    }

    //star1 search phase
    ...
}
```

Listing 5.4: Star2 Algorithm

5.4.3 Star2.5

Star2 searches only one child of each successor in the probing phase. Ballard also explains in his paper a method to search more than one child. The number of searched children in the probing phase is called the probing factor. The probing factor of the Star2 algorithm is 1. Star1 can be seen as having a probing factor of 0, because there is no probing phase.

However, it is also possible to set this probing factor to another value. In Star2.5 this factor is greater or equal 2. Ballard describes two methods to search more than one child [4]. The first method performs a search of one additional child in the probing phase if a cut-off has not occurred during the previous probing phase. So the first probing phase searches the first child, then a second child is considered and so on, until the number of children reaches the probing factor or a cut-off occurs. Ballard calls this form ‘cyclic Star2.5’. The second method, he mentioned, has just one probing phase for each node. That means the method gets a probing factor that indicates the number of children which have to be searched in the probing phase. This form is called ‘sequential Star2.5’. Ballard observed that sequential Star2.5 requires considerably less overhead than cyclic Star2.5, that is why it is used for Carcassonne.

The algorithm of sequential Star2.5 is the same as for Star2, but the method `nprobe` in the probing phase is different. In Star2 this method only searches one child. Now it gets a probing factor and searches for that number of children. If no cut-off occurs, it returns the highest value. Listing 5.5 shows the pseudocode of the `nprobe` procedure of the sequential Star2.5.

```
float nProbe(float alpha, float beta, int depth, int probingFactor){
    for (int i=0; i<probingFactor && i < possibleMoves.size(); i++){
        Move move = possibleMoves.get(i);
        executeMove(move);
        float value = -star2(-beta, -alpha, depth-1);
        undoMove();
        if (value >= beta) return beta;
        if (value > alpha) alpha = value;
    }
    return alpha;
}
```

Listing 5.5: Probing Procedure of Sequential Star2.5

5.4.4 Iterative Deepening

Iterative deepening is a technique which does not search the tree for a fixed depth. It starts with searching depth 1, then a search of depth 2 is performed and so on as long as time is left. The move which is considered as the best one of the previous iteration can be searched first in the current iteration, because it is reasonable that this is also a good move for that iteration. Searching a promising move at the beginning causes in more cut-offs.

This enhancement is used for the computer player. Because if a fixed depth would be used it is not clear how much time it will be used. So if an Expectimax or a *-Minimax player participates in a game the time this player has for the whole game can be predefined. To calculate the time for one move is very simple, the remaining time for the whole game must be divided by the remaining turns of this player. The remaining turns are the half of the remaining tiles, because each player draws one tile per turn.

Chapter 6

Experiments and Results

This chapter describes several tests which are performed for this research. Section 6.1 describes the experimental setup. The following sections present the results of different tests.

6.1 Experimental Setup

In each of these experiments two players compete. To determine the players and their enhancements a graphical user interface was developed. The user can choose how much time each player has for the whole game. Instead of this time limit a Monte-Carlo player can also be limited by the number of simulations and a *-Minimax player by a fixed depth. Furthermore, different factors can be determined, e.g., the probing factor for Star2.5 or the C for the calculation of the UCT-value.

The starting player changes every game. The results are written to a file. There are not only stored the scores of the players. For every player is also stored how many meeples are used in which feature and how many points each feature scores with player's participation on average. Additionally, the total time for the game and the used time for each player is saved. For Monte-Carlo players the number of simulations during the whole game is stored, too. Furthermore, for *-Minimax players the total number of visited nodes is written to the file.

A player is the winner of the game if it scores more points than its opponent. A draw is possible if both players reach the same number of points.

6.2 Monte Carlo

This section shows the results of several Monte-Carlo AIs. Section 6.2.1 compares two different evaluations for a simulated game in the basic Monte Carlo. Besides, it illustrates the average points of the several Carcassonne features earned by Monte Carlo. Section 6.2.2 describes the results of using MCTS. Firstly, different values for C of the calculation of the UCT-value are tested

and the best one is determined. Finally, the MCTS algorithm was compared to basic Monte Carlo.

6.2.1 Basic Monte Carlo

This section describes experiments and results for the basic Monte-Carlo algorithm.

Evaluation of Simulated Games

As described in section 4.2 there are two possibilities to evaluate the end position of a simulated game. Whether the difference of the scores are taken (MC) or the game is evaluated by 1 if the Monte-Carlo player won, 0 for a draw and -1 for a defeat (MCo). To decide which method is more promising 100 games were simulated with these two players. Each player has 6 minutes for the whole game. In 50 games MC began and in the other 50 games MCo was the first player. Table 6.1 shows the results.

The MC player has a win rate of 100%, so it is quite obvious that the Monte-Carlo player, which uses the difference of the players' scores, plays far better. For that reason this method is used if a Monte-Carlo player participated in a further experiment.

	MC starts		MCo starts	
	MC	MCo	MC	MCo
Minimum points	35	9	31	9
Maximum points	108	31	95	34
Average points	62.6	18.3	64.6	19.1
Wins	50	0	50	0

Table 6.1: Experimental results for basic Monte-Carlo algorithms with different evaluations.

Average Feature Scores and Meeple Allocation

The figures 6.1 and 6.2 clarify the MC's advantage over MCo but also some weak points. Figure 6.1 shows the average score for each of the four features, if it was occupied by the given player. Since a cloister scores at most 9 points, the aim is to score on average nearly 9 points. The MC player scores quite well with 7.56 points on average, but maybe it could be improved. Also the city points can be improved. In contrast, the points for a road which is occupied by the MC player are quite well. Since each road part scores only 1 point, it is not advisable to focus on roads. Finally, the fields are scored well – on average nearly 9 points, that means there are 3 closed cities bordered to this field. Of course, it could be improved, but in comparison with MCo or a random player it is already a gain. The diagram shows also that the MCo player does not

play much better than a plain random player. So this method seems to be very bad.

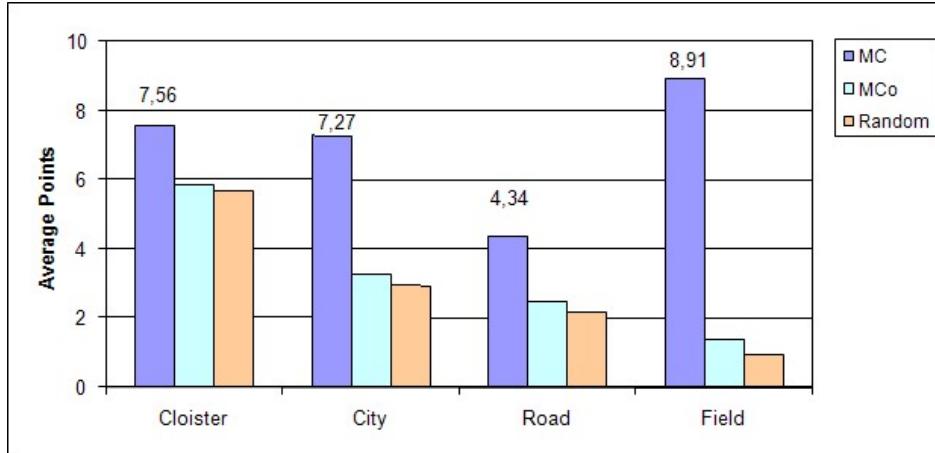


Figure 6.1: The average reached points of different features, in which the player was involved.

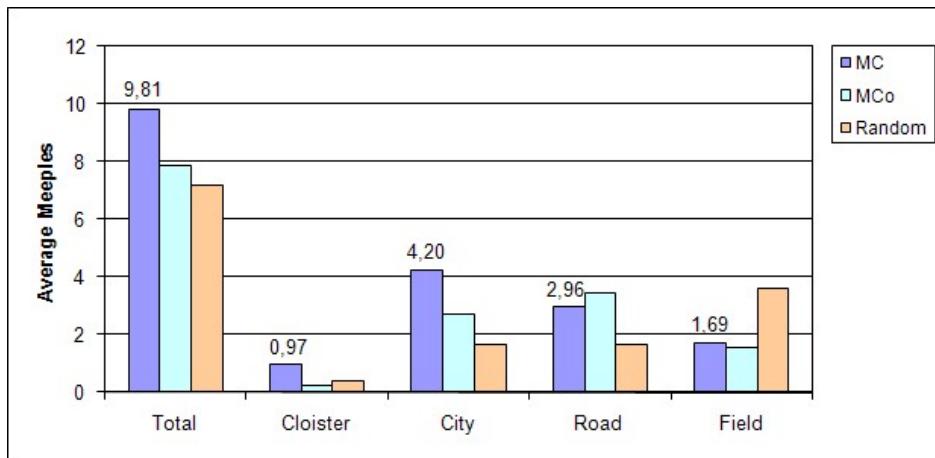


Figure 6.2: The average number of meeples in different features.

Figure 6.2 shows the average number of meeples which are put to the specific features during the game. The MC player uses on average 9.81 meeples during the game. This number is far too low. Every player starts with 7 meeples and after closing a cloister, road or city the player gets involved own meeples back and can use them again. So it should be possible to use more than 10 meeples during a whole 2-player game (at least 35 plies for the player!). Especially the number of meeples for cloisters and cities must be increased, because in this way the player earns many points additionally. The MC player uses on average nearly 3 meeples to occupy roads. This is quite well because as described in the previous paragraph it is not advisable to concentrate on occupation of roads since the poor reward. Also the fields are quite well occupied. Whilst a random player uses nearly 4 meeples for fields on average, MC only puts less than 2 meeples on average to fields, which is enough. There

are two possibilities to use more meeples during the game. First, concentrate on closing roads, cities and cloisters, where own meeples are involved. Second, decrease the chance of playing a move without putting a meeple to the tile although there are meeples in reserve.

6.2.2 Monte-Carlo Tree Search

This section describes experiments and results for the Monte-Carlo Tree Search algorithm.

Upper Confidence bounds applied to Trees (UCT)

The formula for calculating the UCT value contains a constant factor C . Table 6.2 shows the results of different values of C . For each value 50 games were simulated, where a MCTS player with UCT played against the Monte-Carlo player. Each player started in 25 games. The results show that UCT works best with $C = 3$. For that reason MCTS players in further experiments will use $C = 3$.

C	MC starts			MCTS starts			win rate
	win	draw	loss	win	draw	loss	
0.5	13	0	12	15	0	10	0.56
1	13	1	11	15	0	10	0.56
2	14	1	10	16	0	9	0.60
3	18	0	7	19	0	6	0.74
4	17	0	8	18	0	7	0.70
6	12	0	13	18	0	7	0.60
8	16	0	9	17	1	7	0.66

Table 6.2: Experimental results for MCTS with UCT with various values for C versus basic Monte Carlo.

Evaluation of Simulated Games

This experiment tests different evaluations of a simulated game. MCTSo takes only into account if it wins or loses and MCTS uses the difference between the scores of the players. Table 6.3 shows the results. These are similar to the results of basic Monte Carlo. MCTS performs better, even if it is not so clear as basic Monte Carlo. MCTS won 80% of the games. So the difference of the players' scores is also used if a Monte-Carlo Tree Search player participates in further experiments.

	MCTS starts MCTS	MCTSo starts MCTS	MCTSo starts MCTS	MCTSo starts MCTSo
Minimum points	50	16	39	17
Maximum points	111	75	122	82
Average points	77.8	48.8	79.8	54.3
Wins	41	8	39	11

Table 6.3: Experimental results for Monte-Carlo Tree Search algorithms with different evaluations.

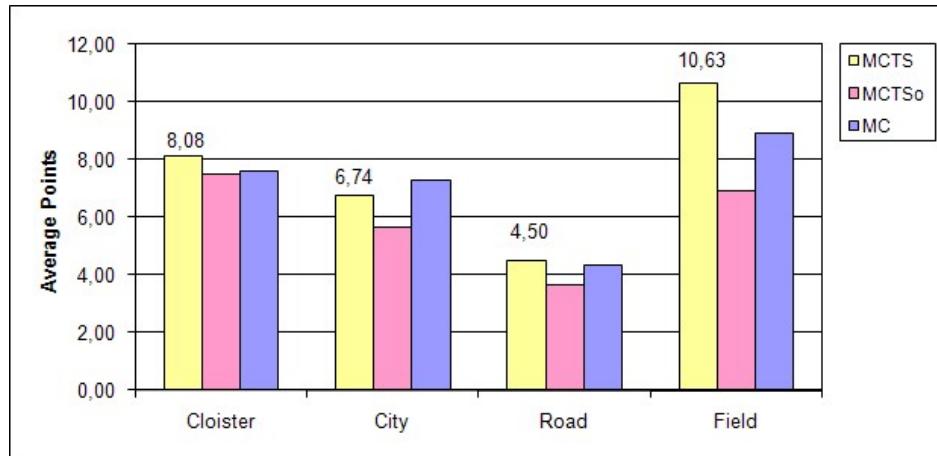


Figure 6.3: The average reached points of different features, in which the player was involved.

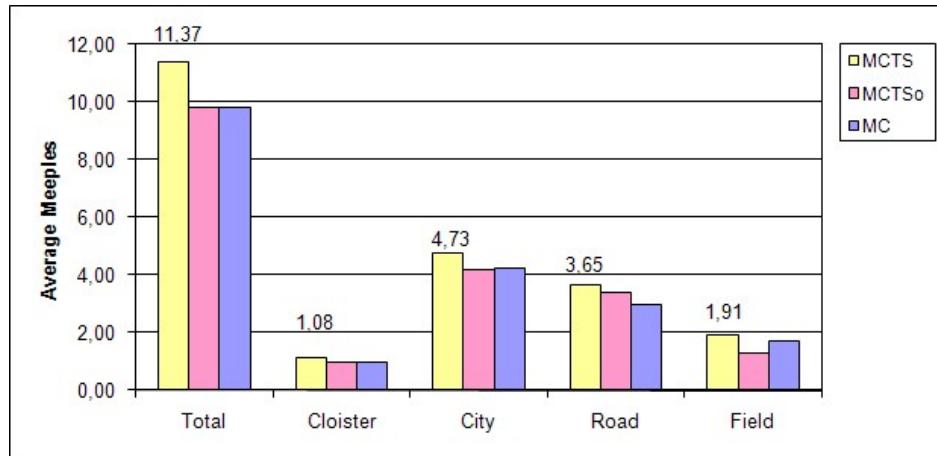


Figure 6.4: The average number of meeples in different features.

Average Feature Scores and Meeple Allocation

Figures 6.3 and 6.4 show the same interrelations as figures 6.1 and 6.2, but now the MCTS players are involved. To compare, the score of the Monte-Carlo player from the previous figures is displayed, too. The total number of used meeples is increased to 11.37. That means a MCTS player uses on average 1.5

meeples more than a MC player. The average score of a cloister is increased to 8. The number of used meeples in cloisters increases also slowly. The average score of cities decreased, but on the other hand MCTS uses nearly one more meeple for cities. The average scores and used meeples for roads and fields increase, too. It turns out that the MCTS player works more efficient than the MC player.

6.3 Expectimax

To compare the Expectimax algorithm with the *-Minimax algorithms 50 game states are defined. These game states are given in Appendix B. Each game state represents a certain game position and the next tile, which the computer player must put on the map, is predefined and so it will be always the same tile. The predefined game states are very different. So there are starting, midgame and endgame positions. Sometimes the player to move has a clear margin, sometimes this player has no chance to win.

This section presents the results of Expectimax and *-Minimax AIs. Therefore, first the best move ordering is determined (section 6.3.1), followed by tests of the evaluation function (section 6.3.2). The best setups of the algorithms where compared in section 6.3.3. Section 6.3.4 describes the results of Star2.5 algorithm. Additionally, section 6.3.5 presents experimental results of iterative deepening.

6.3.1 Move Ordering

Move ordering strongly influences the outcome of Star1 and Star2. In contrast move ordering has no influence on Expectimax, because it is a brute-force algorithm, so it searches each node anyway. Therefore in this section only Star1 and Star2 are examined.

In general there is only one possible feature to taken into account. This checks if a meeple is set on the tile or not. Checking other features, e.g., getting a meeple back or closing a feature, is not effective, because it is very time-consuming. Anyway, there are two possible orders:

- Order 1: First all moves, where a meeple is put to the tile, followed by all moves, where a tile without a meeple is put to the map.
- Order 2: Similarly to the order of the features determined in section 2.3:
 - (1) meeple occupies a city
 - (2) meeple occupies a cloister
 - (3) meeple occupies a road
 - (4) no meeple on the tile
 - (5) meeple occupies a field

Although the fields are the most efficient features, in this order they are the last ones. That is, because most field are not profitable (less or no cities). On the other hand it is not advisable to put too many meeples on fields. So the focus should be on the other features.

Table 6.4 shows the results of these move orders. The results are the averages over the 50 predefined game states. Order 2 reduced the most number of nodes, especially for Star2 there is a great benefit.

	Star1				Star2			
	nodes	gain	time (ms)	gain	nodes	gain	time (ms)	gain
No order	68,128.9	-	1,509	-	32,107.7	-	859	-
Order 1	67,698.5	0.6	1,512	-0.2	35,199.2	-9.6	963	-12.1
Order 2	67,388.9	1.1	1,487	1.4	25,465.1	20.7	717	16.5

Table 6.4: Comparison of different move orders applied to Star1 and Star2, depth 2.

6.3.2 Evaluation Function

To evaluate a certain position different features can be regarded. Obviously the number of points scored until now can be used. If only that feature is taken into account, the scores of fields or incomplete features are not included. So, this feature is not able to distinguish between a meeple occupied an incomplete feature and there is no meeple. Therefore the second feature assumes that the game is over and it is scored again. The difference of this score gets a lower weight than the true actual score, but it is an indication which features are occupied yet. Because roads are scored with 1 per tile no matter if the road is closed or not, the evaluation function cannot determine this. Often it is advisable to get a meeple back. Therefore, the difference of the number of meeples of the players is also regarded.

During testing the computer player often set a meeple on a field, although there was no completed bordered city. An explanation is that the algorithm during searching the tree completed some cities. So the computer player assumes that there will be cities in the future. That is not advisable, because the opponent is going to avoid to complete a city on an opponent's field. Therefore, it is better to put a meeple only on fields if there are at least three cities already completed. That leads to the following features:

- Feature 1: Number of points scored until now
- Feature 2: Number of points assuming the game is over (score also fields and incomplete features)
- Feature 3: Difference of meeples
- Feature 4: Field should not be occupied if it contains less than 3 cities

Table 6.5 shows the first intuitive and the final weights of the features. Different tests are performed where players with different weights compete each other. The weights of the player who won most games are seen as the better configuration. The results show that most intuitive weights are the best, but the weight of the meeple difference was modified to 10 and the weight of the fields with less cities to -40. Additionally, incomplete cities should get more than 1 point per tile, else the algorithm is not able to differentiate between incomplete roads and incomplete cities (both are scored with 1 per tile as long as they are incomplete). Tests show that 1.25 per tile is a good value for incomplete cities.

Feature	Intuitive Weight	Final Weight
Score difference	100	100
Meeple difference	1	10
Score difference of incomplete features	10	10
- Score of incomplete cities	-	1.25
Fields with less than 3 cities	-50	-40

Table 6.5: Weights of the features of the evaluation function.

6.3.3 Expectimax vs. *-Minimax

The section about the move ordering leads to the impression that Star2 outperforms Star1. Figure 6.5 and figure 6.6 verify this prediction. Star1 behaves nearly exactly like Expectimax. There are only few advantages. Star2 is a great improvement of the previous algorithms. The y-axis has a logarithm scale so the gain of Star2 on some game position is very high, e.g., the gain in searching nodes of using Star2 in the game position 2 is 97.4%.

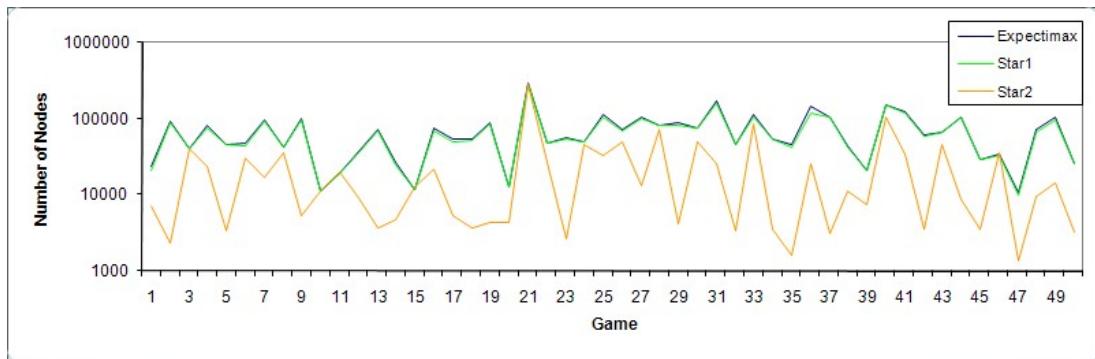


Figure 6.5: Comparison of the number of nodes of Expectimax, Star1 and Star2 (depth 2) on 50 predefined game states.

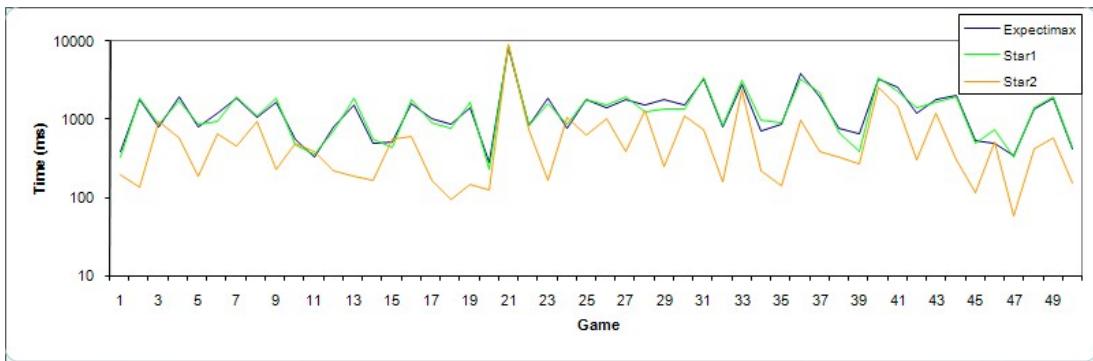


Figure 6.6: Comparison of the time of Expectimax, Star1 and Star2 (depth 2) on 50 predefined game states.

6.3.4 Star2.5

Instead of searching only one child in the probing phase of Star2, Star2.5 searches additional children. The number of searched children is called the probing factor. Table 6.6 shows the results of different probing factors. The search was performed to depth 2.

probing factor	nodes	gain	time (ms)	gain
1 (Star2)	24,463.1	-	706.9	-
2	24,701.4	-1.0	714.0	-1.0
3	24,057.6	1.7	701.3	0.8
4	23,584.4	3.6	694.4	1.8
5	23,221.6	5.1	678.8	4.0
6	23,481.7	4.0	684.2	3.2
7	23,591.8	3.6	692.8	2.0

Table 6.6: Experimental results for Star2.5 with different probing factors. The table shows the average of 50 predefined game states, searching depth 2.

The results show, that Star2.5 only leads to a small gain. The highest gain is reached with probing factor 5. For that reason this configuration is considered as the best one of the *-Minimax algorithms.

6.3.5 Iterative Deepening

Iterative deepening increases the search depth incrementally as long as time is left. In contrast to a fixed search depth, often more time is used and more nodes are searched. But because the best move of the previous iteration is investigated first in the current iteration, iterative deepening sometimes is able to outperform a search with a fixed depth. Tests showed that, especially in the first half of the game, many moves reach a depth of 3 with iterative deepening (20 seconds per move). In contrast, with a fixed depth of 3 the

average needed time for the predefined game states is 113 seconds. Totally, in 20 out of the 50 predefined game states the time of searching depth 3 can be improved by using iterative deepening, so that they can be investigated within 20 seconds. Table 6.7 shows the best effort of iterative deepening, which was reached in game state 31. This is a position of the beginning phase.

	nodes	gain	time (ms)	gain
Fixed depth of 3	3,236,169	-	76,239	-
Iterative deepening until depth 3	195,865	93.9	5,107	93.3

Table 6.7: Experimental results of game state 31, fixed depth of 3 vs. iterative deepening until depth 3.

6.4 Monte Carlo vs. Expectimax

In this section the best Monte-Carlo algorithm played against the best one of Expectimax/*-Minimax. The Monte-Carlo Tree Search algorithm with UCT and $C = 3$ is the best Monte-Carlo player. Star2.5 with probing factor 5 is the most promising of the *-Minimax algorithms. So these two players compete against each other. Each player has 6 minutes for the whole game.

First tests show that the Monte-Carlo player wins many games by a clear margin. The reason is that the upper and lower bounds in the calculation of Star2 are fixed. So the initially setup is $L = -2000$ and $U = 2000$. The basis of the evaluation function takes the difference of the players' scores times 100. So if the opponent has a 25-points lead over the Star2.5 player, the score is already -2500 . Most of the tiles are not able to reduce the lead so that the evaluation score is greater than the lower bound -2000 . For these tiles the move does not matter, the algorithm will always choose the first one, because all moves seem to be bad. It seems like the player resigns. An improvement of the Star player is to use dynamical lower and upper bounds. Therefore at the beginning of the algorithm the difference between the players' scores is determined. Normally a tile will not lead to more than 20 points, so 2000 are added to the calculated difference. This value will be the upper bound for the Star2 algorithm and the negated value the lower bound.

Table 6.8 shows the results of 100 games between the MCTS player and Star2.5 with dynamical bounds. It turns out that Star2.5 performs better than MCTS. An evidence is provided in figures 6.7 and 6.8, showing the distributions of the scores and meeples. The base of the success of Star2.5 is the used number of meeples, which increases strongly. Despite of decreasing the average points of all features, this algorithm outperforms MCTS. In general it seems, that the strategy of Star2.5 is to build small features and get the meeples back quickly. The advantage of MCTS is to put meeples on fields early. When Star2.5 decides to put a meeple on a field it is already occupied. That is why

Star2.5 only put on average 0.37 meeples on fields. On the other hand MCTS misses meeples for other features, and so the possibilities to score decrease.

	MCTS starts		Star2.5 starts	
	MCTS	Star2.5	MCTS	Star2.5
Minimum points	46	50	49	56
Maximum points	103	118	104	119
Average points	74.2	82.0	77.6	87.1
Wins	15	33	12	37

Table 6.8: Experimental results of 100 simulated games played by MCTS with $C = 3$ and Star2.5 with probing factor 5 and dynamical bounds.

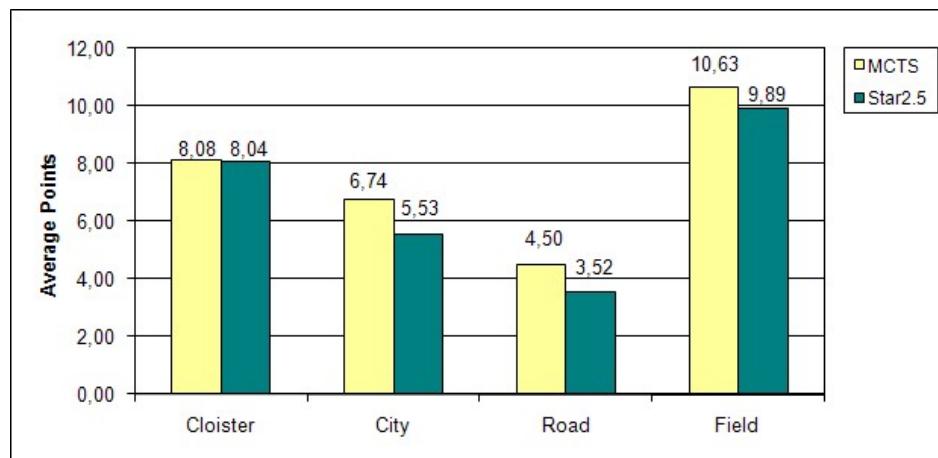


Figure 6.7: The average reached points of different features, in which the player was involved.

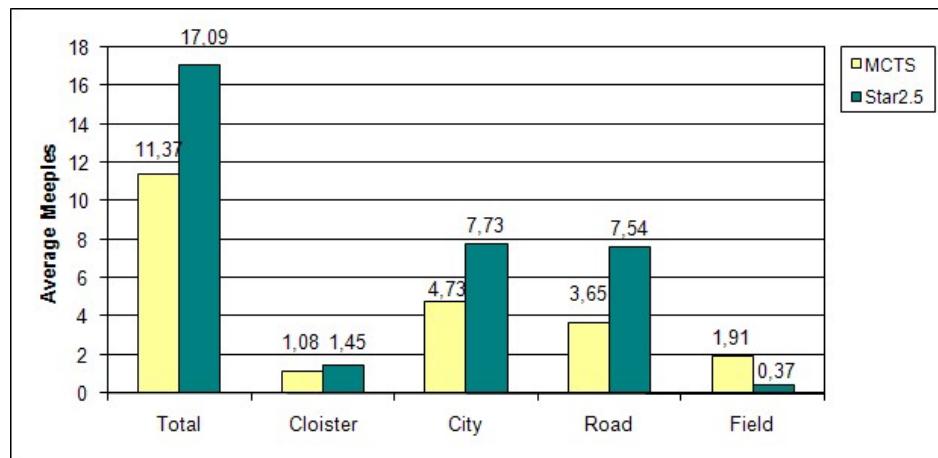


Figure 6.8: The average number of meeples in different features.

6.5 Experiments against Human Players

This section describes the results of both algorithms, the Monte-Carlo Tree Search and Star2.5, playing against human players. Firstly, the Star2.5 player played against advanced human players (Cathleen Heyden, Robert Briesemeister). Totally, there were 10 games played, whereof the Star2.5 player won 6 games. Additionally these players played 3 games against MCTS, and won all of them. Besides, the MCTS player played 5 games against an intermediate beginner (Peter Schlosshauer), whereof it won 2 games.

These results show that the Star2.5 player is able to win against advanced human players. Furthermore, the MCTS player is at least able to win against an intermediate beginner.

Chapter 7

Conclusions and Future Research

This chapter gives conclusions of the research and possibilities for future research. In section 7.1 and 7.2 the research questions and the problem statement are answered. In section 7.3 possibilities for future research are given.

7.1 Answering the Research Questions

In section 1.2 the following research questions were stated:

1. *What is the complexity of Carcassonne?*

In chapter 3 the complexities of Carcassonne are calculated. The game-tree complexity is $O(10^{194})$ and the state-space complexity is at least $O(10^{40})$. The state-space complexity is a highly underestimated lower bound, because it only regards the number of different shapes with up to 72 tiles. Also the game-tree complexity is an estimator, because the branching factor of the game-tree complexity was determined by taking the average of the branching factors of 200 simulated games.

2. *Can Monte-Carlo search be used for implementation?*

The first algorithm which was investigated during this research is Monte-Carlo search. Monte Carlo does not need any strategic knowledge. It reached good results if for the evaluation of a simulated game the difference of the scores of the players is stored instead of only 1 point for a won game.

3. *Can Expectimax be used for implementation?*

Another approach, which was investigated, is Expectimax. Expectimax is a full-width search. Because of the high branching factor of Carcassonne only

a depth of 2 was reached. The quality of the algorithm depends strongly on the evaluation function. So if it uses a good evaluation function it leads to good results.

4. In which way can the approaches be improved?

Both algorithm can be improved. Monte-Carlo Tree Search is an improved Monte-Carlo algorithm. Instead of simulating random games in the first plies the move with the highest UCT-value (Upper Confidence bounds applied to Trees) is chosen. So, often the most promising move is executed. To evaluate a simulated game the difference between the players' scores is used.

Expectimax can be improved, so that it is no longer a full-width search. The improved algorithms are Star1 and Star2. Star1 is a pruning algorithm like Alpha-beta search, but applied to trees with chance. Star2 contains an additional phase which determines a lower bound for each node by searching only the first child. To get a more accurate lower bound, more than one child can be searched. In that case the Star2 algorithm becomes to Star2.5. Additionally, the lower and upper bounds of the Star algorithms can be determined dynamically. If iterative deepening is applied a search depth of 3 can be reached.

5. Which technique reaches the best result?

The results of section 6.4 shows that Star2.5 with probing factor 5 performs best, because it won 70% of the simulated games against MCTS. The lower and upper bounds were determined dynamically. Nevertheless, Monte-Carlo Tree Search has also its advantages and therefore it should not be neglected.

7.2 Answering the Problem Statement

After answering the research questions, the problem statement can be answered:

Can a computer player be built to play the game of Carcassonne as good as possible?

Yes, a computer player for Carcassonne, which performs quite well, can be built. The best result was reached with Star2.5 search with probing factor 5 and dynamical bounds. Monte-Carlo Tree Search reached also good results, but it has disadvantages over Star2.5. Both algorithm are able to win against an intermediate beginner. The Star2.5 player is even able to win against advanced human players.

7.3 Future Research

This is the first research for the game of Carcassonne. Therefore, there are many directions for future research.

Firstly, the state-space complexity can be calculated more accurately, because the calculated value is a highly underestimated lower bound. Furthermore, the investigated algorithms can be enhanced. At the moment the Star player uses a basic evaluation function. So it can be fine-tuned more. Another idea is to investigate the effort of ChanceProbCut [21]. Also MCTS can be enhanced. Strategic knowledge is an opportunity to improve the algorithm. To avoid that the player always focuses on fields, more intelligent moves during the simulation phase of MCTS need to be chosen. It would be interesting whether this enhanced MCTS player can win against the Star2.5 player.

Additionally, one or more of the expansions can be taken into account. Nowadays, only few people play the basic Carcassonne; most players add several expansions. With each expansion new figures, tiles and rules are added. So the complexity of the game increases strongly.

Another future research direction will be to investigate algorithms for the multi-player variant, because Carcassonne can be played with up to 6 players. Therefore the implemented MCTS algorithm could be used, but the *-Minimax variants should be extended for the multi-player variant.

Bibliography

- [1] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*, Ph.D. Thesis, University of Limburg, Maastricht, 1994.
- [2] S. Appelcline: Anatomy of a Game. Carcassonne, Part One: The Original Game. Online: <http://boredgamegeeks.blogspot.com/2006/03/anatomy-of-game-carcassonne-part-one.html>
- [3] S. Appelcline. Review of Carcassonne. Online: <http://www.rpg.net/reviews/archive/9/9737.phtml>
- [4] B.W. Ballard. The *-Minimax Search Procedure for Trees Containing Chance Nodes. *Artificial Intelligence*, 21:327-350, 1983.
- [5] G. Barequet, M. Moffie, A.Ribó, and G. Rote. Counting polyominoes on twisted cylinders. In S. Felsner, editor, *European Conference on Combinatorics, Graph Theory and Applications (EuroComb '05)*, volume AE of *Discrete Mathematics and Theoretical Computer Science Proceedings*, pages 369-374, 2005.
- [6] B. Brügmann. Monte Carlo Go. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:182-193, 1993.
- [7] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In Michael Mateas and Chris Darken, editors, *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216-217. AAAI Press, Menlo Park, CA., 2008.
- [8] R. Coquelin and R. Munos. *Bandit Algorithm for Tree Search*. Technical Report 6141, INRIA, 2007.
- [9] R. Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Proceedings of the 5th International Conference on Computer and Games*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 72-83. Springer-Verlag, Heidelberg, Germany, 2007.
- [10] E. Cuppen. *Using Monte Carlo Techniques in the Game Ataxx*. B.Sc. Thesis, Maastricht University, 2007.

- [11] T. Hauk, M. Buro, and J. Schaeffer. *-Minimax Performance in Backgammon. *Computers and Games*, 2004.
- [12] T. Hauk, M. Buro, and J. Schaeffer. Rediscovering *-Minimax Search. *Computers and Games*, 2004.
- [13] T. Hauk. *Search in Trees with Chance Nodes*. M.Sc. Thesis, University of Alberta, 2004.
- [14] I. Jensen and A.J. Guttmann. Statistics of lattice animals (polyominoes) and polygons. *Journal of Physics A: Mathematical and General*, 33:L257-L263, 2000.
- [15] I. Jensen. Fixed Polyominoes. Table of n, a(n) for n = 1..56. Online: <http://www.research.att.com/~njas/sequences/b001168.txt>
- [16] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Artificial Intelligence*, pages 282-293, 2006.
- [17] R. J. Lorentz. Amazons Discover Monte-Carlo. In H. J. van den Herik, Xinhe X., Zongmin M., and M. H. M. Winands, editors, *CG 08: Proceedings of the 6th international conference on Computers and Games*, pages 1324. Springer-Verlag, Heidelberg, Germany, 2008.
- [18] J.A.M. Nijssen. *Playing Othello Using Monte Carlo*. B.Sc. Thesis, Maastricht University, 2007.
- [19] D.H. Redelmeier. Counting Polyominoes: Yet Another Attack. *Discrete Mathematics*, 36:191-203, 1981.
- [20] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Inc., Englewood Cliffs, New Jersey, 1995.
- [21] M. Schadd, M. Winands, and J. Uiterwijk. ChanceProbCut: Forward Pruning in Chance Nodes. Maastricht University, 2009.
- [22] Sensei's library: UCT. Online: <http://senseis.xmp.net/?UCT>
- [23] Toshihiro Shirakawa. Free Polyominoes. Table of n, a(n) for n=0 ..45. Online: <http://www.research.att.com/~njas/sequences/b000105.txt>
- [24] J. Veness. *Expectimax Enhancements for Stochastic Game Players*. B.Sc. Thesis, University of New South Wales, 2006.
- [25] N. Wedd. Details of program: Mogo, 2007. Online: <http://www.lri.fr/~gelly/MoGo.htm>

Appendix A

Tiles in Carcassonne

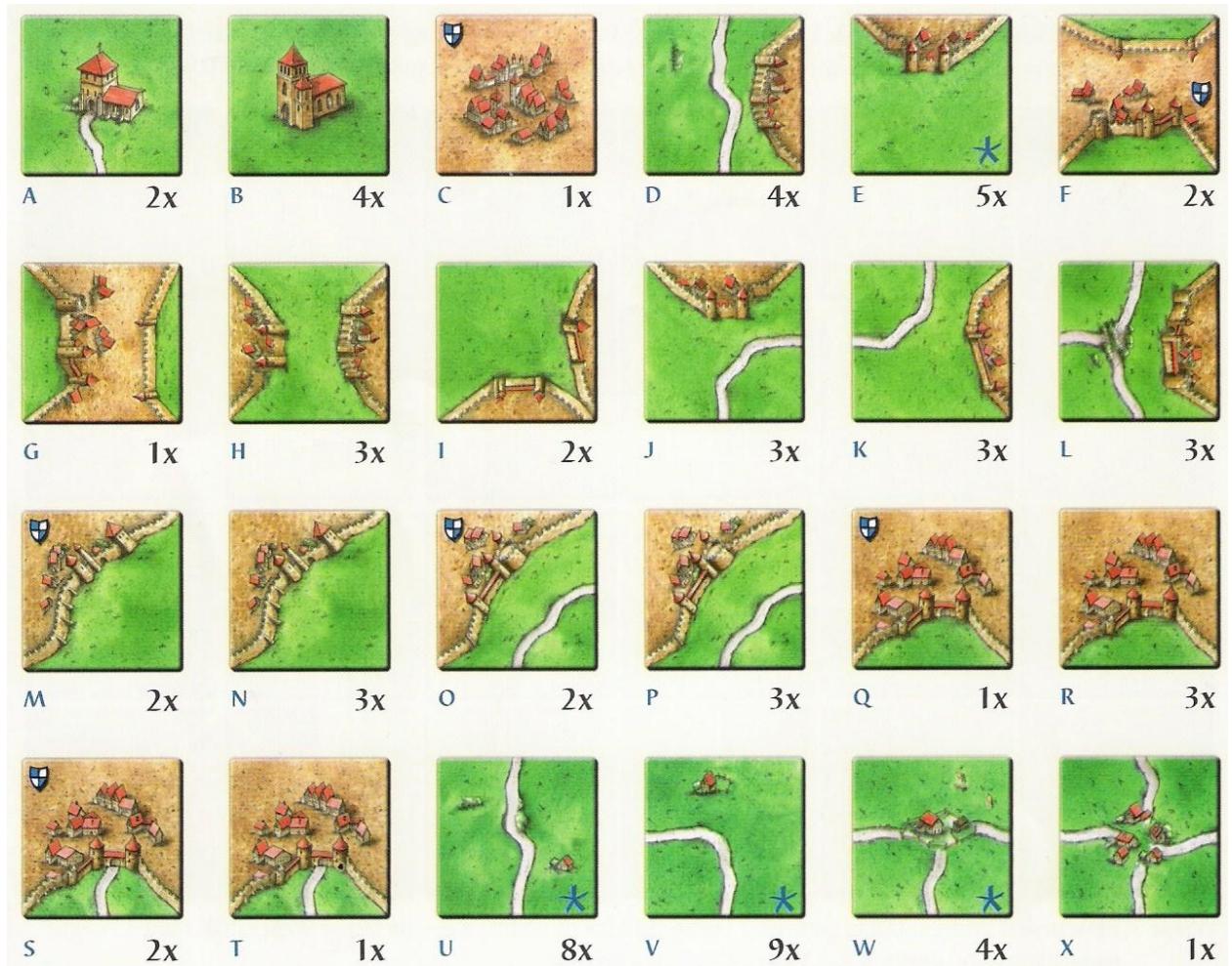
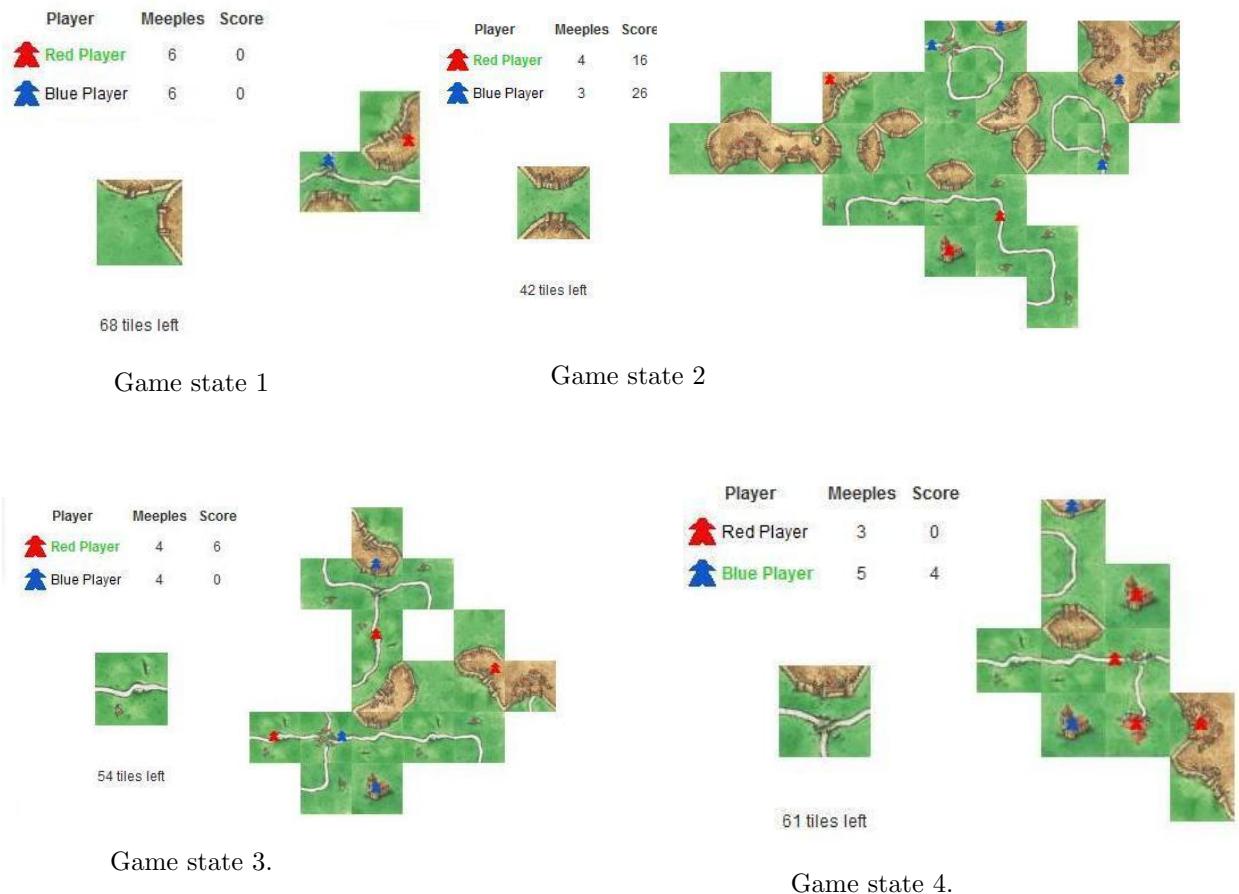


Figure 1: Overview of the tiles in Carcassonne. The numbers indicate how many tiles of this type appear during the game.

Appendix B

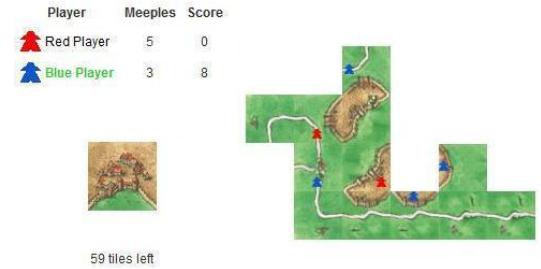
Game States

Figure 2: The subfigures below indicates the 50 predefined game states.

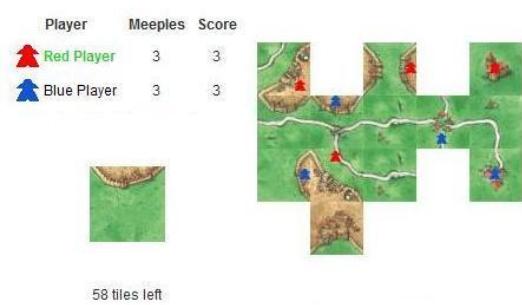




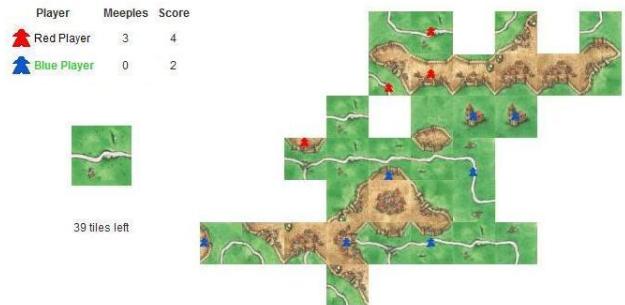
Game state 5.



Game state 6.



Game state 7.



Game state 8.



Game state 9.



Game state 10.

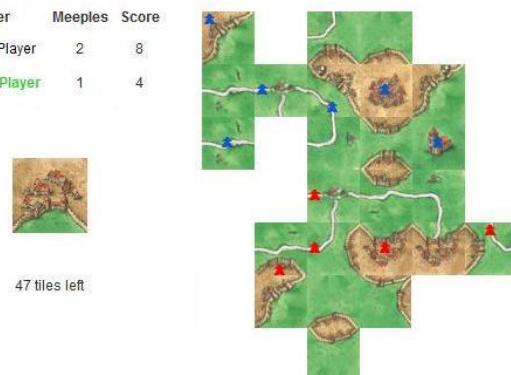


Game state 11.



Game state 12.

Player	Meeples	Score
Red Player	2	8
Blue Player	1	4



Game state 13.

Player	Meeples	Score
Red Player	7	4
Blue Player	6	0



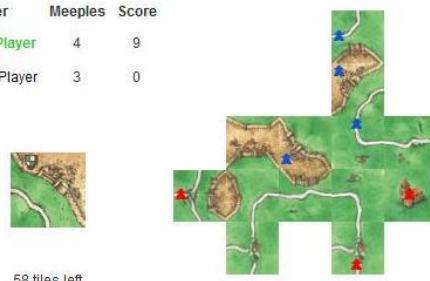
Game state 14.

Player	Meeples	Score
Red Player	0	24
Blue Player	0	27



Game state 15.

Player	Meeples	Score
Red Player	4	9
Blue Player	3	0



Game state 16.

Player	Meeples	Score
Red Player	5	4
Blue Player	6	4



Game state 17.

Player	Meeples	Score
Red Player	5	4
Blue Player	6	4



Game state 18.



Game state 19.



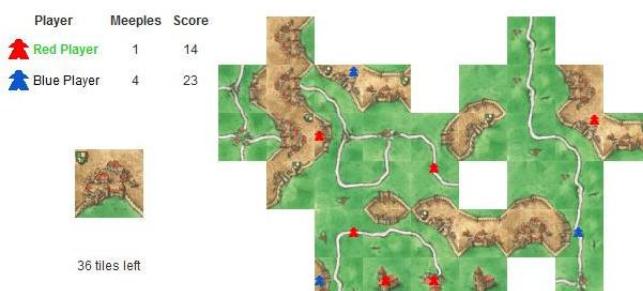
Game state 20.



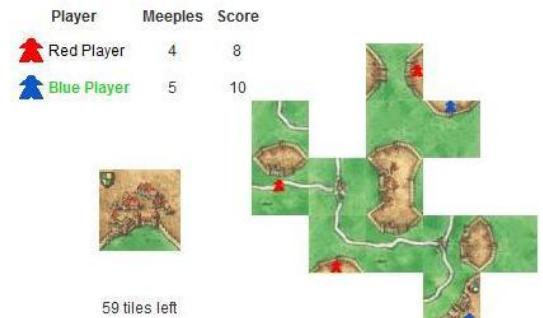
Game state 21.



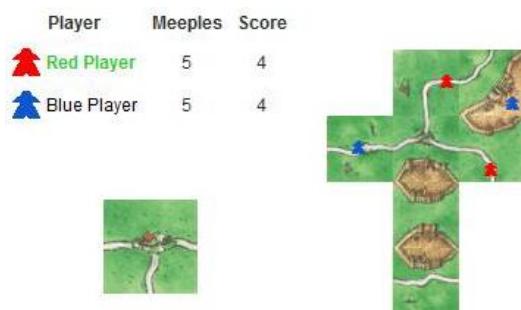
Game state 22.



Game state 23.



Game state 24.



Game state 25.



Game state 26.



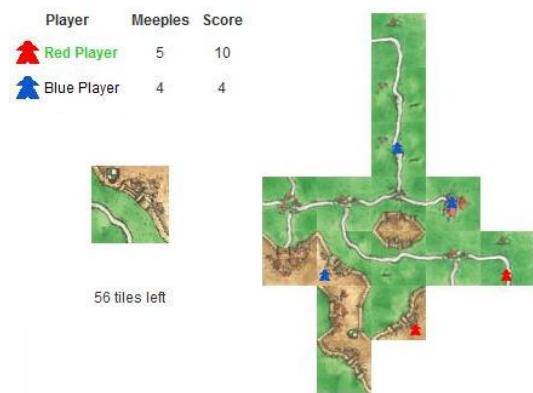
Game state 27.



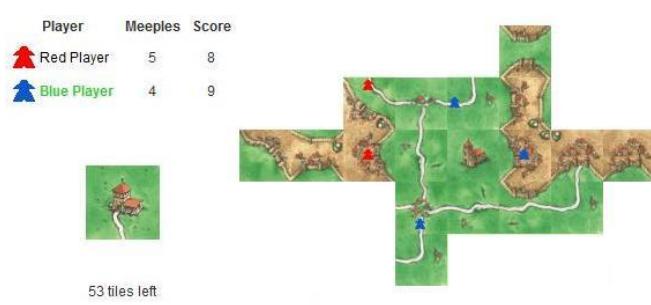
Game state 28.



Game state 29.



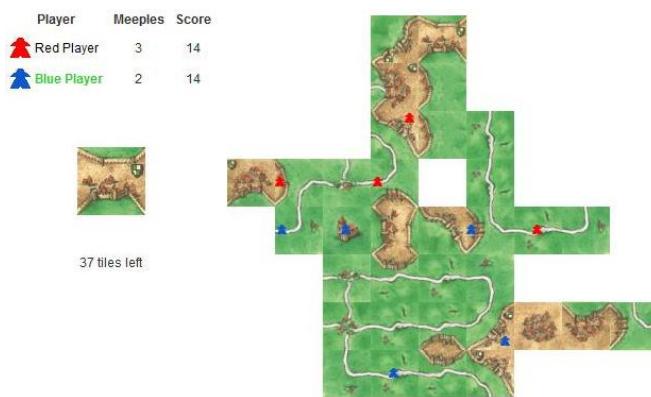
Game state 30.



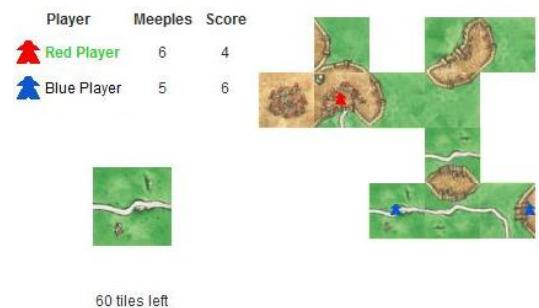
Game state 31.



Game state 32.



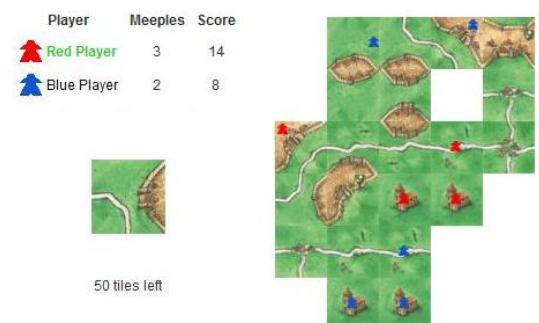
Game state 33.



Game state 34.



Game state 35.



Game state 36.



Game state 37.



Game state 38.

Player	Meeples	Score
Red Player	5	0
Blue Player	5	0



66 tiles left

Game state 39.

Player	Meeples	Score
Red Player	5	16
Blue Player	3	10

54 tiles left

Game state 40.



Player	Meeples	Score
Red Player	3	5
Blue Player	4	4



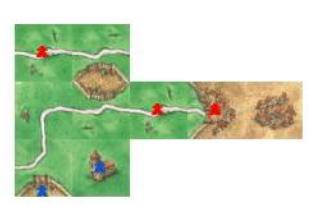
58 tiles left

Game state 41.

Player	Meeples	Score
Red Player	4	0
Blue Player	5	4

62 tiles left

Game state 42.



Player	Meeples	Score
Red Player	1	2
Blue Player	1	0



54 tiles left

Game state 43.

Player	Meeples	Score
Red Player	4	6
Blue Player	4	0

60 tiles left



Game state 44.

Player	Meeples	Score
Red Player	5	0
Blue Player	6	0




67 tiles left

Game state 45.

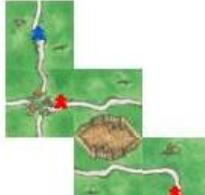
Player	Meeples	Score
Red Player	6	4
Blue Player	6	0




66 tiles left

Game state 46.

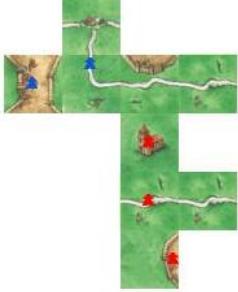
Player	Meeples	Score
Red Player	5	0
Blue Player	6	4

66 tiles left

Game state 47.

Player	Meeples	Score
Red Player	4	0
Blue Player	5	0




62 tiles left

Game state 48.

Player	Meeples	Score
Red Player	5	6
Blue Player	3	3




57 tiles left

Game state 49.

Player	Meeples	Score
Red Player	5	4
Blue Player	5	0




64 tiles left

Game state 50.