MULTI-PLAYER SEARCH IN THE GAME OF BILLABONG

Michael Gras

Master Thesis 12-04

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

Thesis committee:

Dr. M.H.M. Winands
Dr. ir. J.W.H.M. Uiterwijk
J.A.M. Nijssen, M.Sc.

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
July 2012

Preface

This master thesis was written at the Department of Knowledge Engineering at Maastricht University. In this thesis, I investigate the application of multi-player search algorithms, in particular Best-Reply Search and possible variations, in the game of Billabong.

First of all, I would like to thank my supervisor dr. Mark Winands. His guidance during the past months has taken this thesis to a higher level. In weekly meetings, we had fruitful discussions on ideas and problems. His input and corrections helped to constantly improve the quality of my work. Furthermore, the course of "Intelligent Search Techniques", given as part of the master programme of Artificial Intelligence by him and dr. ir. Jos Uiterwijk, inspired me for this research. I would like to thank my fellow student Markus Esser for endless discussions on how to compare multi-player Chess and Billabong and pointing out differences in game properties. Further, I would like to thank my sister Stephanie Gras for her corrections. Last but not least, I would like to thank my family and my friends for supporting me while I was writing this thesis.

Michael Gras
Maastricht, July 2012

Abstract

Humans have been playing board games to intellectually compete with each other for many centuries. Nowadays, research on Artificial Intelligence showed that computers are also able to play several games at an expert level.

In this thesis, a computer program is built that is able to play the game of Billabong. Billabong is a deterministic multi-player game with perfect information. Up to four players compete in this racing game by moving their team of "kangaroos" around a lake in middle of the board. This research focuses on evaluation-function-based search algorithms for the three- and four-player version of the game.

The thesis starts with an introduction on how to play the game of Billabong, followed by the determination of the state-space and game-tree complexities. Next, the investigated multi-player search algorithms are described. The traditional search algorithms like $\max^n$ and paranoid have conceptual weaknesses as their general assumptions on the opponents' strategy are either too optimistic or too pessimistic. Best-Reply Search (BRS), recently proposed by Schadd and Winands (2011), is not dependent on these unrealistic assumptions, but the search tree differs from the game tree as illegal game states are investigated. Two ideas are proposed to improve the strength of BRS. First, instead of ignoring all opponents except one, those players have to perform the best move according to static move ordering. This avoids to search illegal game states. Second, searching for more than just one strongest move against the root player causes BRS to be more aware of the opponents' capabilities. The resulting three algorithms $\text{BRS}_{1,C-1}$, $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ are matched against the three previously mentioned search techniques.

For the domain of Billabong, BRS turned out to be the strongest search technique. Performing a move does not change the board state too much such that the search of illegal states does not affect the search quality. $\text{BRS}_{1,C-1}$ is the most promising variation of BRS. It performs comparatively good against $\max^n$ and paranoid. Its properties are similar to BRS, but due to more overhead caused by the generation of additional moves, $\text{BRS}_{1,C-1}$ cannot search as many MAX nodes in sequence as BRS. $\text{BRS}_{1,C-1}$ requires a paranoid move ordering that prefers strong moves against the root player. In the domain of Billabong, there is only a $\max^n$ move ordering. It prefers the moves where the player increases its own progress. As such a move might also be good for the root player, it might overestimate branches in the search tree. Therefore, $\text{BRS}_{1,C-1}$ is less strong than BRS. The idea of performing multiple moves against the root player is not successful either. The variations $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ perform only a bit better than paranoid as they are less pessimistic. Both are beaten by BRS and $\text{BRS}_{1,C-1}$.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This chapter introduces the topic of the thesis. A brief overview about games and Artificial Intelligence (AI) is presented in Section 1.1. The state of the art in evaluation-function-based multi-player search is briefly discussed in Section 1.2. Next, the problem statement and research questions are introduced in Section 1.3. Finally, an outline of the thesis is given in Section 1.4.

1.1 Games & AI

Games have a long tradition in many cultures. All around the world people compete with each other in many different types of games. Especially board games are interesting because there is no physical strength but intellectual capabilities are involved.

Already with the first computers, games turned out to be an interesting field of research. People started to use board games like Chess (Shannon, 1950; Turing, 1953) to compete with human intelligence. Nowadays, there are many powerful expert AI players like TD-Gammon for Backgammon (Tesauro, 1995), Chinook for Checkers (Schaeffer *et al.*, 1996) and Deep Blue for Chess (Hsu, 2002) that are able to win against the best human players in the world. There are several reasons why games are an established domain for researching techniques in AI. In contrast to real-world problems, the rules of games are well-defined and can often be modelled in a program with little effort (Van den Herik, 1983). However, playing games is a non-trivial problem. Although the rules of Chess are explained quickly, humans have not yet figured out how to play it perfectly (Breuker, 1998). Techniques and algorithms that have been developed in the context of games are applicable in the different domains, e.g. operations research (travelling salesman problem) (Nilsson, 1971).

The game-playing programs aim to solve the problem of finding the best move in an arbitrary game situation or the related problem of finding the game-theoretic value of a position. The most applied algorithm for abstract game playing is search. The current and all future game states can be organized in a tree structure where the nodes represent the game states. Two nodes are connected if there is a move that changes the original game state in the intended way. The search space is often quite large such that a simple brute-force search cannot find the optimal solution in an acceptable time frame. Starting with minimax search (Von Neumann and Morgenstern, 1944), strong search algorithms and enhancements have been developed over the years. Most notable of them are $\alpha\beta$-Search (Knuth and Moore, 1975) and Monte-Carlo Tree Search (Kocsis and Szepesvári, 2006; Coulom, 2007). $\alpha\beta$-Search uses a heuristic evaluation function to compute which move sequence leads to the best future game state assuming that the opponents play optimally according to their strategy. Monte-Carlo Tree Search uses statistics collected while (semi-)randomly playing games to find the move sequence that most probably leads to a win. Both algorithms are preferred in different domains as both have strengths as well as weaknesses, e.g. $\alpha\beta$-Search in Chess (Hsu, 2002) and Monte-Carlo Tree Search in General Game Playing (Björnsson and Finnsson, 2009). $\alpha\beta$-Search leads to powerful play if the heuristic evaluation is strong, which is sometimes difficult to construct. Monte-Carlo Tree Search requires less domain knowledge, but performs poorly if the (semi-)randomly played games do not correlate with optimally played ones. This thesis focuses on evaluation-function-based search algorithms.

1.2 Multi-player Games

In the past, researchers mostly focused on two-player games. The complexity of playing games increases with the number of opponents. In multi-player games, players can collaborate to increase their strength or to outwit other players. If it is not commonly known which players act as a team efficient search is difficult. The two main evaluation-function-based search algorithms for multi-player games are max^n (Luckhardt and Irani, 1986) and paranoid (Sturtevant and Korf, 2000). Max^n assumes that every player tries to maximize its own score while paranoid assumes that all opponents form a coalition against the root player. Both algorithms have conceptual weaknesses and are based on either a too optimistic or a too pessimistic assumption for many games.

Schadd and Winands (2011) have proposed Best-Reply Search (BRS) as an alternative search technique. Instead of letting all opponents move, only the opponent with the strongest move against root player is allowed to move. Although this search algorithm can lead to illegal and unreachable game states as usually all players have to move according to the rules, BRS outperforms max^n and paranoid in the games Chinese Checkers and Focus (Schadd and Winands, 2011). The aim of this thesis is to investigate the performance of BRS in another test domain, the game of Billabong (Solomon, 1984), and to find out whether it is possible to improve BRS there by adapting the algorithm. Billabong is a deterministic perfect-information board game where up to four players compete in a race. Each player has control of five pieces, so called kangaroos, which have to circuit a lake in the middle of the board while blocking and exploiting the opponents' pieces.

1.3 Problem Statement & Research Questions

In computer game-playing, the goal is to create a computer program that plays a certain game as strong as possible. The problem statement for this thesis is the following:

How can one use search for the game of Billabong in order to improve playing performance?

In order to answer the problem statement, the following three related research questions are investigated.

1. *What is the complexity of Billabong?*

The complexity of a game depends on the state-space and the game-tree complexity (Allis, 1994). The state-space complexity is the total number of possible game states. A game state in Billabong is unique distribution of up to 20 kangaroos on the board. The game-tree complexity is the total number of leaf nodes in the game tree from the initial position. Both complexities have to be computed as they indicate whether the game is solvable. If the game is solvable, search techniques with a guaranteed game-theoretic value are preferred.

2. *How strong is Best-Reply Search in Billabong?*

In order to answer this question, BRS is matched against the traditional search algorithms max^n and paranoid in a three- and a four-player experimental setup.

3. *How can Best-Reply Search be improved for a given domain?*

Best-Reply Search has conceptual drawbacks. Two ideas are proposed to overcome these drawbacks. The first one allows the opponents to apply its best move according to static move ordering and the second one allows a larger subset of opponents to perform its strongest move against the root player. These concepts lead to three variations of Best-Reply Search. The performance of the proposed variations BRS$_{1,C-1}$, BRS$_{C-1,0}$ and BRS$_{C-1,1}$ are experimentally verified.

1.4 Outline of the Thesis

The outline of this thesis is as follows:

- Chapter 1 gives an introduction into multi-player games and search techniques. It closes with the research questions and the thesis outline.
- Chapter 2 explains rules and strategies for the game of Billabong.
- Chapter 3 analyses the complexity of Billabong and compares it to other games.
- Chapter 4 presents the search techniques $\alpha\beta$-search, $\max^n$, paranoid, BRS as well as possible enhancements like the transposition tables, the killer heuristic and the history heuristic. Further, a more detailed description of the variations of Best-Reply Search is given.
- Chapter 5 describes the experiments performed and their results.
- Chapter 6 gives the final conclusions and gives recommendations for future research.

Chapter 2

The Game of Billabong

In this chapter, the game of Billabong is explained in detail. After a brief introduction in Section 2.1, Section 2.2 describes the rules of the game. Subsequently, Section 2.3 discusses tips and strategies for successful game play.

2.1 Introduction

The board game *Billabong* was first described by Solomon (1984). After being licensed and marketed by the two publishers Amigo Spiel + Freizeit GmbH and Franjos-Verlag in 1993, it became a successful game in Germany and was nominated for the "Spiel des Jahres"-Award in 1994.

Billabong is a racing game for two to four players. Each player has control of a team of five kangaroos that can jump over all kangaroos on the board. The complete team has to circuit a lake in the middle of the board, the *billabong*. In this strategy game, good positioning is always the key to success as this leads to strong and long jumps. The name of the game was inspired by real billabongs that are a kind of lake in the dry Australian outback. The game has some resemblance with Chinese Checkers.

2.2 Rules

Billabong is a deterministic, turn-based and perfect-information multi-player game. The rules of the game are straight forward. As already mentioned in the previous section, Billabong can be played with two to four players. Every player has control of five pieces, so called kangaroos, on a 14×16 board. In the middle of the board there is a lake, the billabong, which is 2×4 tiles large. It is fed by a small river marking the start-finish line.

Before having a closer look at the rules, the notation for the game is introduced. Inspired by the official chess notation, all tiles on the board are labelled with a letter and a number indicating the column and row on the board, respectively (cf. Figure 2.1). In the initial phase, where all players put their kangaroos on the board, a move is only defined by the tile where the player puts the piece on. For instance, putting a kangaroo on tile **j3** is noted as **j3**. Later in the game, there are two different types of moves, step and jump moves, but both can be equally described by the starting and the landing position of the move. For instance, moving a piece from **l5** to **k4** is noted as **l5 k4**. If a move crosses the start-finish line from right to left, which is a clockwise movement around the billabong, a +-sign is added to the move description. If it crosses it from left to right, the kangaroo is moving backwards noted by −. For instance, moving from **i2** to **h2** crosses the finish-line from right to left, the corresponding notation is **i2 h2** +, while moving in the different direction crosses it from left to right and is noted by **h2 i2** −. Crossing the start-finish line twice is noted by ++ or − −.

Before playing Billabong, each player can choose the colour of its kangaroos. For simplicity reasons, a fixed order of colours is used in this thesis. Player 1 has the red pieces, Player 2 has the yellow ones, Player 3 has the orange ones and Player 4 has the white ones.

At the beginning of the game, every player can place its pieces freely on one of the initially 216 empty tiles of the board. When all pieces are placed on the board, the actual race starts clockwise around the billabong. From now on, the players can perform a step or a jump move. Step and jump moves cannot be

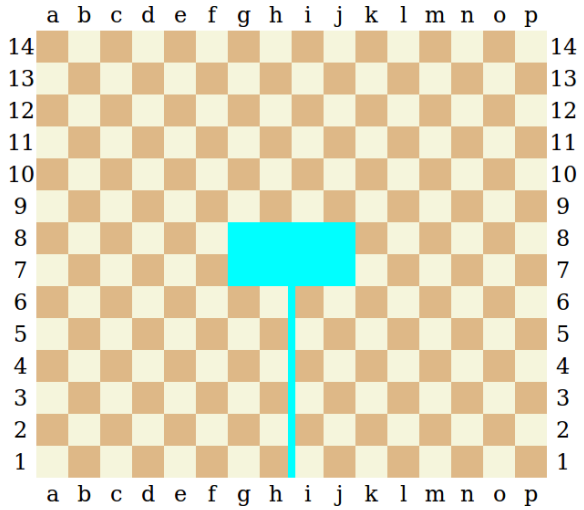


Figure 2.1: Empty billabong board

combined and passing is not allowed. In a step move a player moves one of its pieces to an adjacent and empty tile in vertical, horizontal or diagonal direction. A jump move consists of one or more leaps. The player can stop leaping even if continuing is possible. A kangaroo can leap over exactly one piece (the pivot). The pivot has to be in the same vertical, horizontal or diagonal line. The distance from the start position of the leap to the pivot position must be equal to the distance to the landing spot. While leaping, the kangaroo is not allowed to cross other pieces or the billabong. A piece cannot land on another piece, in the billabong or outside the board. A small extension to the original game rules prohibits negative numbers of start-finish line crossings. This disallows to race around the billabong anticlockwise.

An example of a jump move is given in Figure 2.2a. The initial position of this jump move is **h5** and the piece lands on **j13**. Therefore the move is noted as **h5 j13**. It consists of three leaps. The first leap from **h5** to **f5** passes the yellow piece on **g5**. The green square marks a reachable location for the current player. Its number indicates the number of leaps or steps required. From **f5** to **f9** it passes by the yellow piece on **f7** and finally leaps over the white piece on **h11**. Before starting a sequence of leaps in a turn, the player puts a referee kangaroo on the current position of the selected piece. This enables it to use the initial position as pivot as well. In Figure 2.2b, the red piece on **f2** jumps to **b8**. Therefore several leaps are necessary. The third leap from **d2** to **h2** requires a piece to be on **f2**, the initial position. As the referee, marked with **R**, has been placed on this position, the leap is legal.

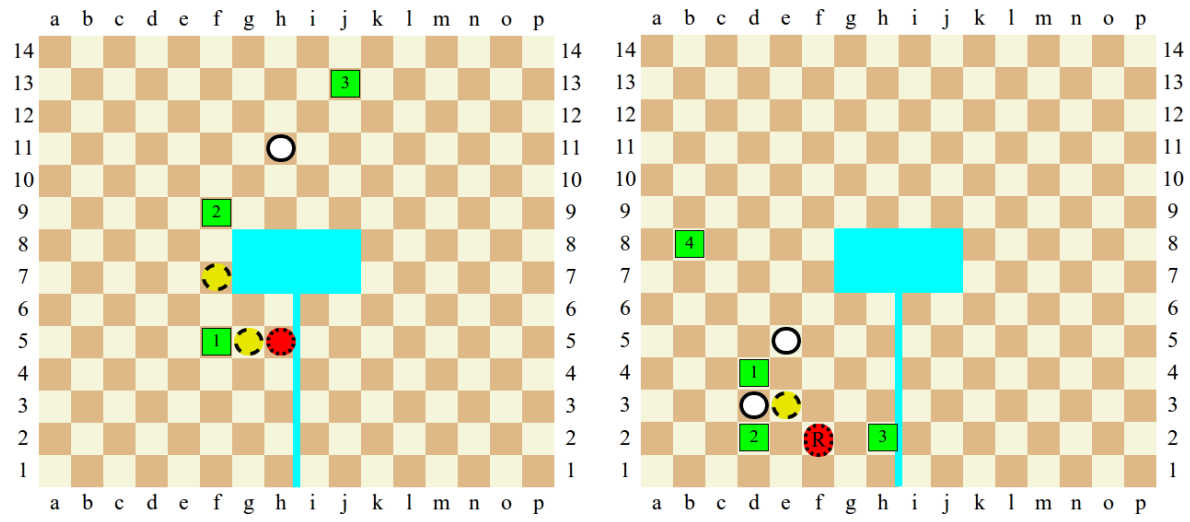
A piece that passes the start-finish line a second time is removed from the board and cannot be used as a pivot any more. The player that first realizes to cross the start-finish line twice with all of its kangaroos wins.

2.3 Strategy

It is important when playing Billabong that all of one player's pieces stay close to the centre-of-mass of all the pieces. If the opponents realize to move in such a way that at least one of the player's kangaroos cannot follow, this player usually cannot win the game any more. It cannot jump with this piece and can only step to adjacent tiles. Therefore it progresses slowly. In the meantime, all other players can progress normally.

So, it is good for a player to stay close to the peloton, where most pieces are located, and never to have control over the last piece in the game. This property should not be exaggerated, because it turns the game into a slow progression game. Figure 2.3a presents a situation where every player constantly moved all the pieces to the centre of the peloton. The green spots mark every position that is reachable for the yellow player. None of those moves causes big progress for it. If Yellow allows one of its pieces to have some distance to the peloton, it enables this piece to do a long jump (Figure 2.3b). Doing so, it is not unlikely that the player is able to move a piece around the whole board within one turn.

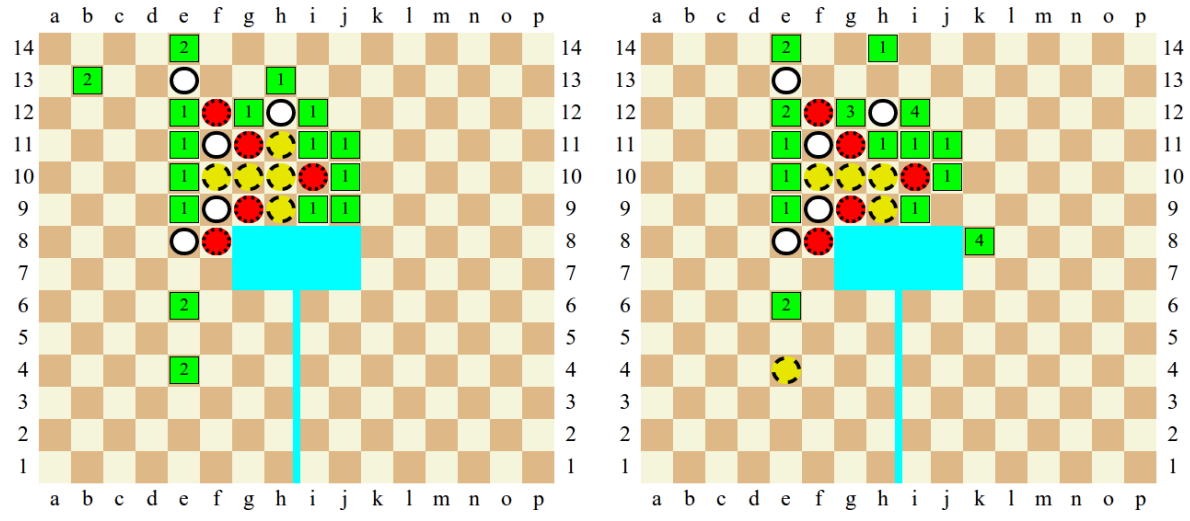
Allowing some gaps in the peloton is also the key to a good start into the race. Of course, the pieces



(a) Jump move with multiple leaps

(b) Usage of the referee kangaroo

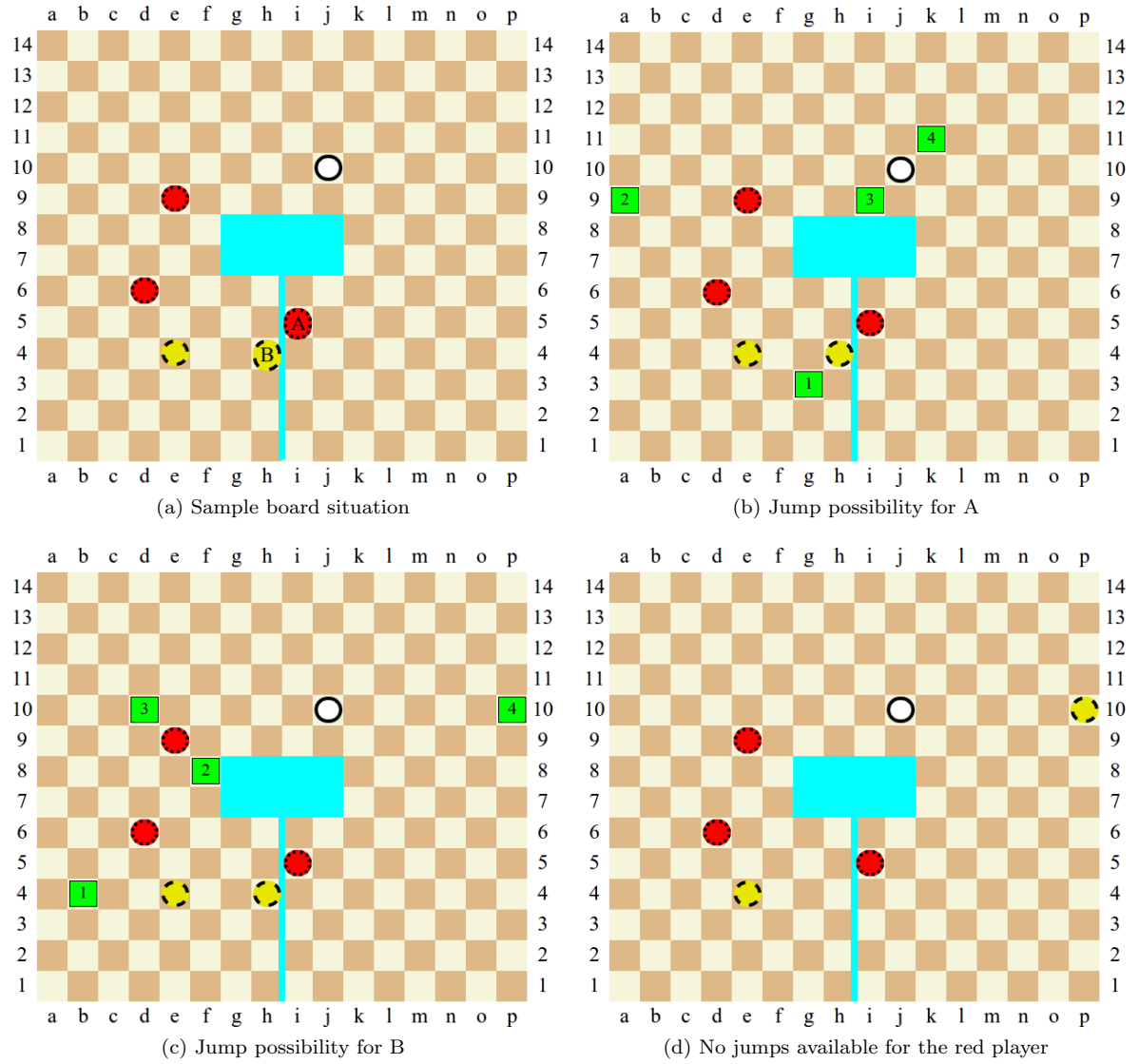
Figure 2.2: Legal jump moves



(a) No long jump moves available

(b) Strong long jump move due to large gap to the group

Figure 2.3: Jumping possibilities for Yellow



(a) Sample board situation

(b) Jump possibility for A

(c) Jump possibility for B

(d) No jumps available for the red player

Figure 2.4: Good position for Red turns into bad position

benefit from a short distance to the start-finish line in the placing phase such that placing the pieces in the area from **i1** to **p6** is preferable to placing in the area from **a1** to **h6**, but again the players should avoid immobility.

It is difficult to plan one's next moves. Usually, jumping is the most beneficial way of moving, but often a prepared jumping track can be sabotaged by moving a single piece. In the following example board situation (Figure 2.4a), both the red and yellow player have good jumps available as they can move the piece A standing on **i5** (Figure 2.4b) or B standing on **h4** (Figure 2.4c) half around the billabong. Unfortunately for the red player, it is Yellow's turn. Yellow chooses to perform the described jumping move. In the subsequent board situation (Figure 2.4d) Red has no jumping possibilities available. Furthermore, piece A cannot keep pace with all the other pieces. This requires the red player to concentrate on piece A and allows the yellow player to easily increase its lead over its opponent. Red is dependent on the pivot on **i5**. If the pivot belonged to the red player, it could be sure that it will not move away unless Red decides to do so. Therefore, it is better to prefer jumping over one's own pieces than to rely on the opponent pieces.

Another important aspect in Billabong is to find a good trade-off between cooperation with other players, blocking other players and only concentrating on its own progress. There are many situations in the game, where it is beneficial for a player to offer a good jump to an opponent instead of sabotaging. On the one hand, a sabotaging move might cause slow progress for the player himself, e.g. by preferring

a step move to a long jump move in order to hinder the opponent's progress. On the other hand, the player's next move can be dependent on the opponent's move. For instance, the opponent's move enables a long jump move in the next turn or releases a block for the current player. Furthermore, the opponent's move could cause blocking of one or more different opponents. Blocking good moves of the remaining opponents can be good for the player because it simplifies to catch up with the leading players.

2.4 Computer Billabong

The rules of Billabong were first described by Solomon (1984). In his book *Games Programming*, he explains the basics of games programming and search techniques known at that time. He discusses the minimax algorithm with $\alpha\beta$-pruning in the chapter *Abstract Games*. The general advantage of move ordering during search using static move ordering and killer moves is emphasized. As the book was already published when research on multi-player search rose, it concentrates on two-player search. Besides the rules of Billabong, Solomon proposes features of an evaluation function for the two-player game, which are discussed in Section 4.8.

In 2003, the department Mathematical Sciences of the University of Alaska Anchorage hosted a small tournament in Computer Billabong for their students as part of a semester project[1] in Artificial Intelligence. Except for the tournament results, there are no publications on the applied techniques.

[1]The specifications for the semester project and the tournament can be found the website of the university on http://www.math.uaa.alaska.edu/~afkjm/cs405/billabong/billabong.html

Chapter 3

Complexity Analysis

The development of an AI for Billabong requires having an estimate of its complexity. The state-space and the game-tree complexity of the game of Billabong are examined in Section 3.1 and Section 3.2. Finally, Section 3.3 compares the complexity of Billabong with other games.

3.1 State-Space Complexity

The state-space complexity is the total number of legal game states reachable from the initial state (Allis, 1994). For many games the exact state-space complexity is quite hard to compute such that only an upper bound is known that also contains illegal or unreachable game states.

For Billabong, the exact state-space complexity can be computed. This requires to determine the complexities during the placing and racing phase separately.

3.1.1 Complexity during Placing Phase

The initial board in Billabong is empty. During the placing phase, the player to move has to place a piece on an empty tile of the board. The players can choose every empty tile on the whole board. The state-space contains the possible board situations where not all pieces have already been put on the board. After placing the last piece of the last player, the board state already belongs to the racing phase. The state-space complexity in the placing phase is dependent on the number of players participating in the game (cf. Equation 3.1).

$$Complexity_{Placing}(n_{players}) = \begin{cases} PlacePieces_2 & \text{if } n_{players} = 2 \\ PlacePieces_3 & \text{if } n_{players} = 3 \\ PlacePieces_4 & \text{if } n_{players} = 4 \end{cases} \tag{3.1}$$

All possible placings of 0, 1, 2, 3 and 4 pieces are summed up where all players have the same number of pieces on the board. This is the point in the game where a new round begins and Red is to turn. All placings of 0, 1, 2, 3 and 4 pieces are added where Red has just placed the next piece and therefore only Red has 1, 2, 3, 4 or 5 pieces on the board. When playing with three or more players, all placings of 0, 1, 2, 3 and 4 pieces are added where Yellow also has just placed another piece and therefore Red and Yellow have 1, 2, 3, 4 or 5 pieces on the board. This is repeated for the orange player as well if four players participate (cf. Equations 3.2, 3.3, 3.4).

$$\begin{aligned} PlacePieces_2 = &\sum_{i=0}^{4} PlaceRYOW(i,i,0,0) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i,0,0) \end{aligned} \tag{3.2}$$

$$\begin{aligned} PlacePieces_3 = &\sum_{i=0}^{4} PlaceRYOW(i,i,i,0) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i,i,0) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i+1,i,0) \end{aligned} \tag{3.3}$$

$$\begin{aligned} PlacePieces_4 = &\sum_{i=0}^{4} PlaceRYOW(i,i,i,i) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i,i,i) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i+1,i,i) \\ &+ \sum_{i=0}^{4} PlaceRYOW(i+1,i+1,i+1,i) \end{aligned} \tag{3.4}$$

So far, all legal numbers of pieces on the board during placing phase are given. Equation 3.5 computes all possible placings of r red, y yellow, o orange and w white pieces on the board. Table 3.1 presents the complexity in the placing phase for 2, 3 and 4 players.

$$PlaceRYOW(r,y,o,w) = \binom{224}{r}\binom{224-r}{y}\binom{224-r-y}{o}\binom{224-r-y-o}{w} \tag{3.5}$$

Number of players	Possible game states	
2	428,810,476,298,932,169	4.29×10^{18}
3	1,567,560,064,242,835,203,596,764,465	1.57×10^{28}
4	5,088,226,013,643,554,731,036,654,528,506,554,129	5.09×10^{37}

Table 3.1: Possible game states in placing phase

3.1.2 Complexity during Racing Phase

The racing phase starts when all pieces have been placed on the board. During this phase the players can also remove pieces from the board and therefore the state-space also contains terminal positions. Again the state-space complexity depends on the number of players participating in the game (cf. Equation 3.6).

$$Complexity_{Racing}(n_{players}) = \begin{cases} RacingPieces_2 & \text{if } n_{players} = 2 \\ RacingPieces_3 & \text{if } n_{players} = 3 \\ RacingPieces_4 & \text{if } n_{players} = 4 \end{cases} \tag{3.6}$$

All possible positions of 0, 1, 2, 3, 4 and 5 red pieces are combined with 0, 1, 2, 3, 4 and 5 yellow pieces. Those are also combined with 0, 1, 2, 3, 4 and 5 orange and white pieces respectively, if the corresponding players participate. (cf. Equations 3.7, 3.8, 3.9). Note that just i, j, k or l can be 0. In this case, the game state is a terminal position.

$$RacingPieces_2 = \sum_{i=0}^{5} \sum_{\substack{j=0 \\ i+j\neq 0}}^{5} RacingRYOW(i,j,0,0) \tag{3.7}$$

$$RacingPieces_3 = \sum_{i=0}^{5} \sum_{\substack{j=0 \\ i+j\neq 0}}^{5} \sum_{\substack{k=0 \\ i+k\neq 0 \\ j+k\neq 0}}^{5} RacingRYOW(i,j,k,0) \tag{3.8}$$

$$RacingPieces_4 = \sum_{i=0}^{5} \sum_{\substack{j=0 \\ i+j\neq 0}}^{5} \sum_{\substack{k=0 \\ i+k\neq 0 \\ j+k\neq 0}}^{5} \sum_{\substack{l=0 \\ i+l\neq 0 \\ j+l\neq 0 \\ k+l\neq 0}}^{5} RacingRYOW(i,j,k,l) \tag{3.9}$$

So far, all possible numbers of pieces on the board are given. The remainder of the pieces has already been removed. Next, it is required to iterate through all possible distributions of the pieces to be in the initial or final round (cf. Equation 3.10). A piece that has not crossed the start-finish line is in the initial round. After crossing the start-finish line, it is in the final round.

$$RacingRYOW(r,y,o,w) = \sum_{i=0}^{r} \sum_{j=0}^{y} \sum_{k=0}^{o} \sum_{l=0}^{w} RacingRoundsRYOW(i, r-i, j, y-j, k, o-k, l, w-l) \tag{3.10}$$

There are r_0 red pieces in the initial round and r_1 red kangaroos in the final round. The same setup holds for y_0, y_1, o_0, o_1 ,w_0 and w_1 pieces of the yellow, orange and white player. Equation 3.11 computes all possible distributions of these eight piece types. Table 3.2 presents the complexity in the racing phase for 2, 3 and 4 players including terminal positions.

$$
\begin{aligned}
RacingRoundsRYOW(r_0, r_1, y_0, y_1, o_0, o_1, w_0, w_1) = & \binom{224}{r_0} \\
& \binom{224 - r_0}{r_1} \\
& \binom{224 - r_0 - r_1}{y_0} \\
& \binom{224 - r_0 - r_1 - y_0}{y_1} \\
& \binom{224 - r_0 - r_1 - y_0 - y_1}{o_0} \\
& \binom{224 - r_0 - r_1 - y_0 - y_1 - o_0}{o_1} \\
& \binom{224 - r_0 - r_1 - y_0 - y_1 - o_0 - o_1}{w_0} \\
& \binom{224 - r_0 - r_1 - y_0 - y_1 - o_0 - o_1 - w_1}{w_1}
\end{aligned}
\tag{3.11}
$$

Number of players	Possible game states	
2	18,881,880,068,059,986,775,168	1.89×10^{23}
3	2,183,106,355,847,003,167,660,811,261,615,232	2.18×10^{34}
4	224,150,674,783,996,785,413,663,613,941,646,052,458,531,840	2.24×10^{45}

Table 3.2: Possible game states in racing phase

As previously mentioned, if i, j, k or l is 0 in Equations 3.7, 3.8 or 3.9, the corresponding game state is a terminal position. Table 3.3 presents the number of terminal states for 2, 3 and 4 players. The total state-space complexity is the sum of the complexities in the placing and racing phase and is presented in Table 3.4.

Number of players	Terminal states	
2	290,851,515,136	2.90×10^{12}
3	56,645,640,203,307,405,780,096	5.66×10^{23}
4	8,732,425,423,161,430,109,830,015,423,340,544	8.73×10^{34}

Table 3.3: Terminal states

Number of players	Possible game states	
2	18,882,308,878,536,285,707,337	1.89×10^{22}
3	2,183,107,923,407,067,410,496,014,858,379,697	2.18×10^{33}
4	224,150,679,872,222,799,057,218,344,978,300,580,965,085,969	2.24×10^{45}

Table 3.4: Total state-space complexity

3.2 Game-Tree Complexity

The game-tree complexity of a game is the total number of leaf nodes in the game tree from the initial position (Allis, 1994). For many games, including Billabong, it is not feasible to compute the exact game-tree complexity. Instead, the size of the game tree is estimated by the average branching factor b to the power of the average game length d. The averages for the branching factor and game length are estimated by performing self-play experiments.

The experiments are performed as described in Section 5.1. In the two-player setup, there are two players using $\alpha\beta$-search. In the three player setup, there are one player using $\max^n$, one player using paranoid and one player using BRS. In the four player setup, there are the three players from the three player setup and an additional player using either $\max^n$, paranoid or BRS. After playing 432 games for each the two-player to the four-player setup, the estimates of the game-tree complexities are collected in Table 3.5.

Number of players	Branching factor b	Game length d	Game-Tree Complexity b^d
2	64.53	69.33	2.96×10^{125}
3	68.00	95.00	1.22×10^{174}
4	69.66	124.15	6.40×10^{228}

Table 3.5: Game-tree complexity

3.3 Comparison to Other Games

Figure 3.1 compares the complexities of two-player, three-player and four-player Billabong to other board games. This figure is inspired by Allis (1994) and updated on the basis of the research of Van den Herik, Uiterwijk, and Van Rijswijck (2002).

This figure shows that the two-player Billabong has a complexity similar to Abalone. Both games have a state-space complexity that is close to the one of Checkers, but their game-tree complexity is much higher than in Checkers. Instead, it is between Chess and Havannah. The game of Checkers is solved (Schaeffer *et al.*, 2007), but because of the high game-tree complexity, it is not possible to solve the two-player Billabong with current hardware. The state-space complexity for three-player Billabong is higher than for the two-player variant and comparable to Draughts. Its game-tree complexity is higher than in Havannah. Four-player Billabong has a state-space complexity comparable to Chess, while the game-tree complexity is similar to the one of Shogi. It is also not possible to solve the three-player and the four-player variant of Billabong in the near future.

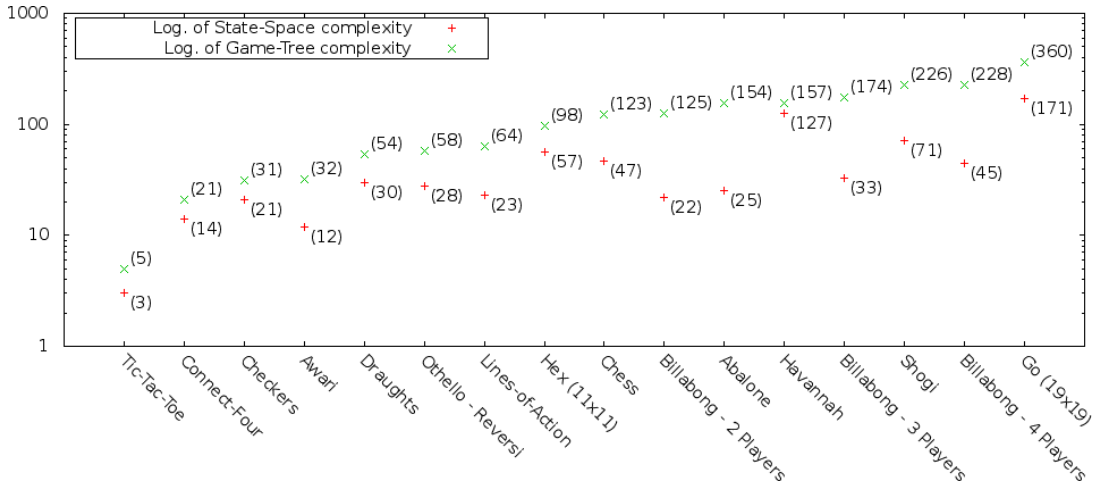


Figure 3.1: Estimated game complexities

Chapter 4

Search Techniques

This chapter describes the search techniques and enhancements used in the Billabong program. Section 4.1 shows how search can be used to play games. Next, two algorithms for two-player games are discussed in Section 4.2. Section 4.3 and Section 4.4 explain some enhancements to $\alpha\beta$-Search. The concept of iterative deepening is presented in Section 4.5. After that, the multi-player search techniques $\max^n$, paranoid and Best-Reply Search are introduced in Section 4.6. Subsequently, Section 4.7 proposes and specifies variations of BRS. Finally, the chapter closes with Section 4.8 characterising the heuristic evaluation for Billabong.

4.1 What is Search?

Imagine someone is playing the game of Billabong. The player is trying to win the game. In other words, it wants to reach a terminal game state where it is the winning player. In order to find such a game state, the player is searching the game tree. The game is usually a directed graph represented by a tree structure. The *nodes* of the tree correspond to game states and *edges* to legal moves in that game state. The *root node* of the game tree is the initial game state, *leaf nodes* correspond to terminal states that cannot be *expanded* according to the game rules.

Games are complex in the sense that it is often not possible to build the complete game tree in an acceptable time frame and given limited memory resources. Therefore, search algorithms do not investigate the entire game tree, but a *search tree*. The search tree and game tree have the same root node, but the search tree is limited to a certain depth. The leaf nodes in the search tree are not necessarily terminal states of the game. A *heuristic evaluation function* assigns a value to each leaf node. The value represents the utility of a game state and correlates with the winning possibility and the game-theoretic score of the player.

The solution of the search is a discrete sequence of actions that lead to the desired state assuming optimal play of all players. In the context of board games like Billabong the sequence of actions is a sequence of moves.

4.2 Two-player Search

Searching in two-player games has an additional challenge compared to single-player games. The player has an opponent which is searching for a terminal game state where the opponent is the winning player. In many two-player games there is at least one terminal state with exactly one winning and one loosing player. All remaining leaf nodes in the game tree are a draw for both players. As winning is the preferable outcome for the player and the opponent, the desired terminal states are not the same. From the perspective of the searching player, the player itself is the MAX player and its opponent is the MIN player. The MAX player tries to maximize the evaluation function while the MIN player tries to minimize it. The distinction between the MAX player and the MIN player is the basis of the minimax algorithm that is discussed in Subsection 4.2.1. Knuth and Moore (1975) proved that it is possible to prune the minimax tree using an $\alpha\beta$-search window without affecting the quality of the search. This technique is described in Subsection 4.2.2

4.2.1 Minimax

Minimax is a recursive search algorithm (Von Neumann and Morgenstern, 1944). For each child node, minimax is applied until a leaf node in the search tree has been reached. The evaluation function assigns a game-theoretic value to the leaf node which is backed up to the parent node. The parent node's value is the maximum (minimum) of all its children's values if it is the MAX (MIN) player's turn at the current ply.

Figure 4.1 depicts an example minimax tree. The evaluation function computes the utilities of 8, 3, 4 and 5 for the leaf nodes. Node (b) and (c) are MIN nodes as the MIN player is to move. The MIN player tries to minimize the game-theoretic value of the tree and therefore chooses 3 and 4 for Node (b) and (c), respectively. Node (a) is a MAX node as the MAX player is to move. The MAX player tries to maximize the game-theoretic value of the tree and therefore chooses 4 for Node (a).

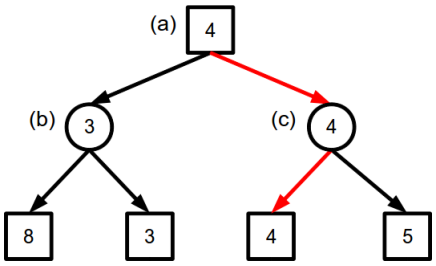


Figure 4.1: An example minimax tree

4.2.2 $\alpha\beta$-Search

It is not necessary to investigate all nodes in the search tree. During the search process, it is possible to eliminate branches of the search tree if it is already clear that an ancestor of a node is considered as non-optimal and therefore never will be chosen. This type of branch elimination is called *pruning*. Pruning allows to reduce the complexity of a search given a fixed search depth. Smaller complexity leads to faster search and may enable deeper search given the same time frame.

The most famous pruning technique is $\alpha\beta$-pruning (Knuth and Moore, 1975). $\alpha\beta$-pruning updates the upper and lower bound of each node's value. For the root node, the initial lower bound (α) is set to $-\infty$ and the upper bound (β) to ∞. The currently known $\alpha\beta$-window initialises the lower and upper bound for the search in the subtree. The MAX player updates the lower bound and the MIN player the upper bound. If $\alpha \geq \beta$, there is a *cutoff* and the corresponding subtree does not need to be investigated further.

The pseudo code for $\alpha\beta$-pruning can be found in Appendix A (cf. Algorithm A.2). The initial values for α and β are set to $-\infty$ and ∞, respectively. An example how the $\alpha\beta$-algorithm prunes is given in Figure 4.2. After the investigation of Node (c), the $\alpha\beta$-boundaries for Node (b) are set to $-\infty$ and 3. The values for α and β are propagated to Node (d). Its first child proves that the MAX player can achieve a score of at least 5 and updates α. As $\alpha \geq \beta$ holds, there is a β-cutoff at the MAX node. At Node (c), the MIN player chooses 3 over ≥ 5.

At Node (e), α and β have the values 3 and ∞, respectively. After the investigation of Node (f), it is known that the MAX player cannot achieve more than 2 in this branch. β is updated to this value. As $\alpha \geq \beta$ holds, there is an α-cutoff at the MIN node. It is not necessary to check Node (g) and its children.

4.3 Move Ordering

The strength of $\alpha\beta$-search is highly dependent on the move ordering (Marsland, 1986). Searching more plausible moves first, leads to early cutoffs. In the best case, $\alpha\beta$-pruning can reduce the search from $O(b^d)$ to $O(b^{d/2})$, where b is the average branching factor of the game and d is equal to the search depth (Knuth and Moore, 1975).

It is distinguished between *static* and *dynamic* move ordering. Static move ordering is often domain dependent. For instance, in capturing games like Chess, capturing moves are searched at first. Static

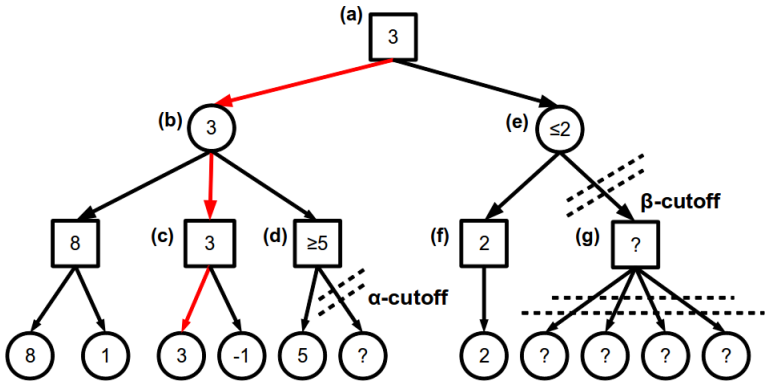


Figure 4.2: An example minimax tree with $\alpha\beta$-pruning

move ordering in Billabong is described in Section 4.3.1. Furthermore, learning techniques can be applied to improve the static move ordering, e.g. neural networks (Kocsis, Uiterwijk, and Van den Herik, 2001). Dynamic move ordering relies on knowledge obtained during the search. Two techniques are implemented for Billabong. The *Killer Heuristic* is discussed in Subsection 4.3.2. Subsequently, the *History Heuristic* is described in Subsection 4.3.3.

4.3.1 Static Move Ordering in the Game of Billabong

Domain dependent move ordering weights moves on their "natural" strength. The situation of the board is often not considered. A simple mechanism to order moves in Billabong is preferring the jump moves to the step moves.

Solomon (1984) proposed a technique to compute the angle of a piece on the board. It is used by his evaluation function (cf. Section 4.8). The angle is measured clockwise from start-finish line to the straight line from the centre of the billabong to the piece. For each crossing of the start-finish line 360° is added to the angle. For performance reasons, the angles are stored in a table. Figure 4.3 depicts the angles of some example tiles.

Inspired by this idea, the following static move ordering is proposed. During the placing phase, the moves are ordered on the angle of the placing location. During the racing phase, every move has a start and end position. The moves are arranged according to the difference between the angles of these positions. Doing so, placing moves close to the start-finish line as well as jump moves that cover a long distance in the right direction are preferred.

4.3.2 Killer Heuristic

The *killer heuristic* tries to produce an early cutoff assuming that a move that already caused a cutoff at some node is likely to cause another cutoff of a different node at the same ply (Akl and Newborn, 1977). The killer heuristic stores at least one *killer move* at each ply. Searching a node, the killer moves of the same ply are investigated first. If a killer move is legal according to the game rules and produces another cutoff it is not necessary to compute all possible moves for the corresponding game situation. A move that causes a cutoff is stored as a killer move. The number of killer moves is limited, such that the killer move that has not been used for the longest time is replaced.

4.3.3 History Heuristic

Schaeffer (1983) proposed the *History Heuristic* to rank the strength of moves over the whole game. Unlike the killer heuristic, it keeps a history for every legal move seen in the search tree. If the number of possible moves is limited, it is possible to preserve the score for each move in a table. For board games, moves are typically indexed by the coordinates on the board. For instance, the table for Chess and Checkers consists of 4,096 entries (64 from squares × 64 to squares). The history table reserves memory for illegal moves as well. This can be a problem for games with a larger dimensionality of moves (Hartmann, 1988). For Billabong, the number of start-finish line crossing has to be respected for both

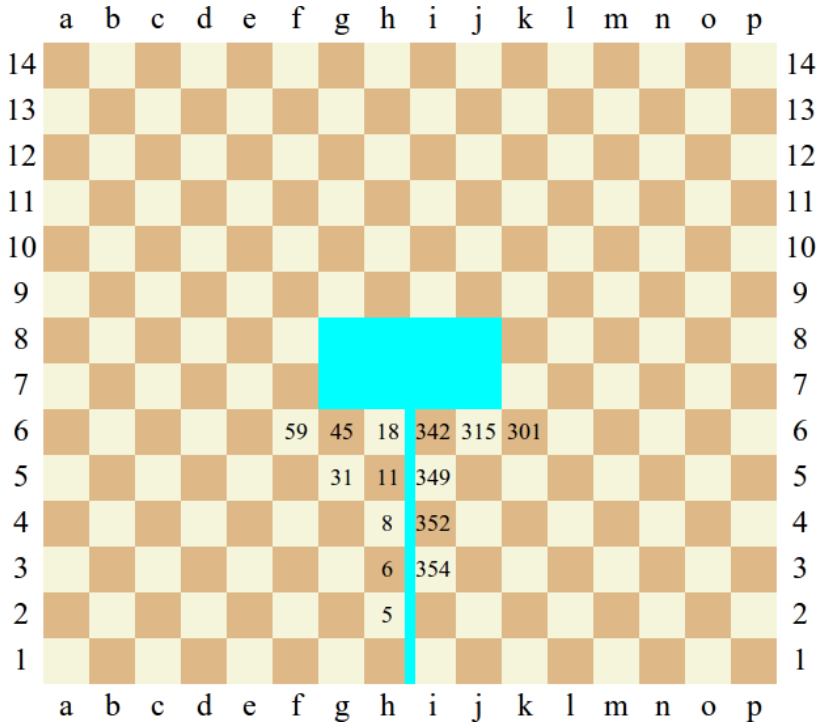


Figure 4.3: Tile angles

the from tiles and the to tiles, such that there are each 432 from and to tiles. Additionally, there are 216 possible placing moves and 432 moves that remove a piece from the board. In total, the table consists of 187,272 entries. The strength of a move might depend on the player performing it such that history tables are maintained separately for each player.

After generating all moves in an interior node during the search, the moves with the same priority according to static move ordering are sorted according to their score in the history table in descending order. This might cause earlier $\alpha\beta$-cutoffs. Then having investigated all moves in a node, the history table entry of the best move found is incremented by some value. Like originally proposed, the increment is 2^d in the Billabong program, where d is the search depth of the subtree under the node.

Hartmann (1988) proposed an alternative to the history heuristic to draw attention to the drawback that the history heuristic assumes that all moves occur equally often. The *butterfly heuristic* reorders moves based on their frequency. Instead of incrementing only the best move in the history table, all moves that are investigated during the search update their score in the *butterfly board*. This excludes non-searched moves in case of an $\alpha\beta$-cutoff. The butterfly board is defined in the same way as the history table. Its inventor assumes the heuristic to be less effective than the history heuristic. The *relative history heuristic* proposed by Winands *et al.* (2006) orders the moves according to the quotient of the score in history table divided by the move frequency in the butterfly boards. The technique improves search performance in the games of Lines of Action (LOA) and Go even more.

4.4 Transposition Tables

In Section 4.1, it is mentioned that a game is represented as a tree instead of a directed graph. A tree structure can be mapped easily into the memory and can be processed fast. In a tree, there is always a unique path to a node, but in game there might exist several sequences of moves that end in the same game state. Nodes representing the same search game state are called *transpositions*. Figure 4.4 depicts an example transposition in the game of Tic-Tac-Toe. Node (a) and (b) represent the same state although the sequences of moves are different. A search algorithm that is not able to detect transpositions considers both subtrees of Node (a) and (b).

Transposition tables are a simple and fast technique to detect transpositions (Greenblatt, Eastlake,

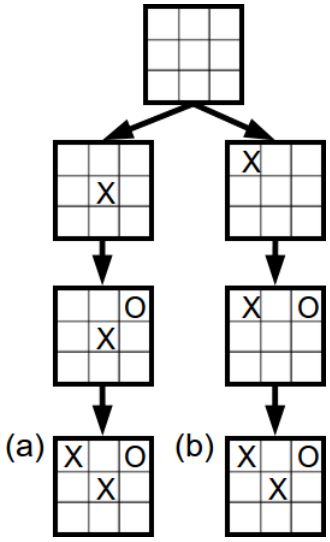


Figure 4.4: Transposition in Tic-Tac-Toe

and Crocker, 1967). Each game state is mapped to a hash value using some hashing method. The state-space of many games is too large to maintain each possible position in memory such that a finite hash table stores a subset of states (Knuth, 1973). If the size of the table holds 2^n entries, the n low-ordered bits of the hash value are used as *hash index*. The remaining bits are the *hash key* and identify two states with the same hash index. A hash table entry typically preserves the following information (Breuker, 1998). The reserved number of bits is domain dependent such that the specification of the Billabong program is given.

key The hash key distinguishes two states with the same hash index. There are 49 bits reserved for the hash key.

move The best move found during the search is used for move ordering. There are 27 bits reserved for the best move.

score The computed utility of the state is preserved. When using $\alpha\beta$-pruning, it can also be a lower or upper bound. There are 32 bits reserved for the score.

flag The flag indicates whether the *score* is a lower bound, an upper bound or an exact value. There are 2 bits reserved for this flag.

depth The depth of the investigated subtree shows how deep the position has been searched. There are 8 bits reserved for the depth.

A hash table entry in Billabong has a total size of 15 bytes. When preserving $2^{24} = 16,777,216$ positions, 240 mega bytes memory are reserved for a transposition table.

For computing the hash value of a game state in Billabong, Zobrist (1970) hashing is applied. Each piece can have 432 different positions on the board, e.g. **a1** with no start-finish line crossings or **j10** with one start-finish line crossing as well as 2 positions not on the board where it has not yet been placed or has already been removed. In the four-player version, there are 20 pieces participating such that there $434 \times 20 = 8,680$ features. As the game state should also contain some information about which player to move, there are 4 additional features indicating if Player 1, 2, 3 and 4 is to move. There are as many random numbers required as there are features. Any game state can be described by the subset of the features which hold in that state. The hash value of that state is equal to the result of the corresponding subset of random numbers after applying the XOR operation.

Before searching a node, it is looked up in the transposition table. If there is an entry that matches the current position, a transposition encounters. If the search depth in the transposition table is sufficiently high ($\geq$ remaining search depth), the preserved *score* either is returned as exact value without further investigation or is used to update the $\alpha\beta$-window depending on the stored *flag*. If no cutoff occurs or

the search depth is too small, it is checked whether the preserved *move* in the table causes a cutoff. Otherwise, the moves of the player have to be generated and the search continues normally.

After the investigation of all children, the best move in this situation and its utility are known. The obtained information might be used to update the transposition table. As the table is limited in size, not all positions with the same hash index can be stored. A replacement scheme decides whether to preserve or to overwrite the previous situation (Breuker, Uiterwijk, and Van den Herik, 1994; Breuker, Uiterwijk, and Van den Herik, 1996). The following replacement schemes are implemented.

Old Always keep the first situation.

New Overwriting is preferred to keeping a value.

Deep The situation with the most deeply searched subtree is preserved in the table.

Two-Deep Two situations with the same index are preserved. One is updated according to *Deep* and the other one according to *New*.

The relation game state to hash value is non-injective. Different positions might have the same hash value. This might cause a serious error, the *type-1 error* (Breuker, 1998). As it is hardly detectable, the search result can be affected by incorrect information. The probability of a type-1 error is given in Equation 4.1, where N is the number of distinguishable nodes and M is the number of different nodes which have to be stored in a transposition table.

$$P(\text{Type-1 error}) \approx 1 - e^{-\frac{M^2}{2N}} \tag{4.1}$$

The Billabong program searches about 500,000 positions a second. If it plays for a total of two hours think time, it investigates 3,600,000,000 nodes. As leaf nodes are not stored, it is assumed that for about 30% of the nodes, an attempt is made to store them in the transposition table (Breuker, 1998). These are 1,200,000,000 nodes in total. The hash value consists of 73 bits (49 for hash key; 24 for the hash index) such that the error probability is:

$$1 - e^{-\frac{(1.2\times 10^9)^2}{2\times 2^{73}}} \approx 7.6 \times 10^{-5} \tag{4.2}$$

It is possible to reduce probability of a type-1 error by checking if the stored move is legal in the current position. If it is not legal, a type-1 error is detected and avoided. Although the probability of a type-1 error is low, the Billabong program verifies the preserved move to be legal. As the program assumes that only legal moves are applied, illegal moves can corrupt the representation of the board in memory. A type-2 error, also called *collision*, occurs when two positions share the same hash index but have different hash values. The positions can be distinguished by the hash key such that the collision is handled without errors.

4.5 Iterative Deepening

The $\alpha\beta$-algorithm investigates a search tree up to a predefined depth. However, it is difficult to predict the runtime of the search. The required time depends on the branching factor which usually varies during the game. Finding the best move in an arbitrary situation is a real-time problem with time constraints. *Iterative deepening* is a possibility to dynamically adapt the search depth. Therefore the search tree is investigated multiple times with increasing search depth starting with a depth of 1. If a search is aborted due to a time-out, the best move of previous iteration is performed. Researching the tree with increasing search depth causes overhead compared to a fixed-depth search. When searching a tree with a branching factor of 40 up to a depth of 5, it is required to investigate $40 + 1,600 + 64,000 + 2,560,000 + 102,400,000 = 105,025,640$ nodes. A fixed-depth search considers $102,400,000$ nodes. The overhead for iterative deepening is about 2.6%. However, obtained knowledge from previous iterations can reduce the amount of nodes to be searched in next iteration as the quality of history heuristic, killer heuristic and transposition tables improves (Slate and Atkin, 1977). Furthermore, iterative deepening finds the shortest path to a win if several paths exist. After completing an iteration, it is checked whether the game-theoretic value of the current game state indicates a win for the searching player. If yes, the search is stopped. If the player searched deeper, it would be possible to find more winning positions that are

deeper in the search tree. Such a situation might lead to never ending game. For instance, the player always prefers a win in three moves to a win in the next move.

4.6 Multi-player Search

Having a strong evaluation function, choosing minimax with $\alpha\beta$-pruning as search technique is a straightforward decision for deterministic perfect-information two-player games because the strategy of the opponent is clear. It wants to optimize its situation in comparison to its opponent. For multi-player games, the opponents' strategy may vary as they are able to form *coalitions*. It is distinguished between *cooperative* and *non-cooperative* games (Osborne and Rubinstein, 1994). For cooperative games usually holds that two or more players can form a coalition and by this achieve higher scores than by playing individually. For non-cooperative games the property can hold as well, but at the end of the game all players prefer winning solely to drawing the game. Therefore, coalitions are only temporary. It is usually unknown which players are in a coalition. It is even possible that player A assumes to be in a coalition with player B, but player B prefers to play solely. Billabong is a non-cooperative game. The players forming a coalition can help its members to perform long jump moves instead of e.g. moving the pivot away.

The two main search algorithms for non-cooperative multi-player games are max^n (Luckhardt and Irani, 1986) and paranoid (Sturtevant and Korf, 2000). Max^n assumes that every player tries to maximize its own score while paranoid assumes that all opponents form a coalition against the root player. Schadd and Winands (2011) have recently proposed Best-Reply Search (BRS) as an alternative search technique. It assumes that only the opponent with the best move against the root player is allowed to move. These three search techniques are discussed in the following subsections in more detail.

4.6.1 Max^n

When performing max^n search (Luckhardt and Irani, 1986), n-tuples are generated for all leaf nodes, where every entry is set to the score that the corresponding player can achieve. At internal nodes, the player to move chooses the child node with the highest score for itself. An example max^n tree is shown in Figure 4.5.

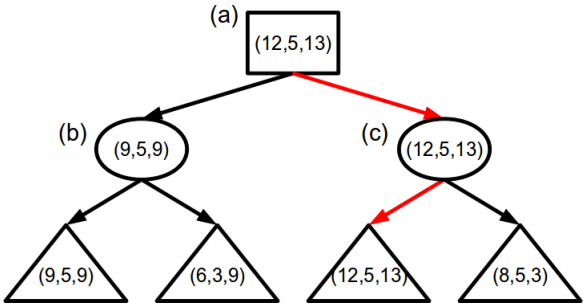


Figure 4.5: An example max^n tree for three players

At Node (b), the second player is to move and prefers $(9, 5, 9)$ over $(6, 3, 9)$ as the first child maximizes its score. At Node (c), both children are equally good and therefore none of them is preferable. Player 2 chooses the left child. At Node (a), Player 1 performs the second move as it will achieve a score of 12.

There can exist several multiple equilibrium points for multi-player games. It has been proven by Luckhardt and Irani (1986) that max^n finds one equilibrium point. Which one is dependent on the tie-breaking rule (Sturtevant, 2003a). In the example at Node (c), the left-most child node has been chosen in case of a tie. If preferring the state in which the root player scores less, like it is implemented in the Billabong program, Player 2 chooses the right child at Node (c) and therefore Player 1 prefers Node (b) over (c). The found equilibrium point changes from $(12, 5, 13)$ to $(8, 5, 3)$.

Max^n has a small lookahead as due to lack of pruning (Sturtevant, 2003b). Deep pruning can incorrectly affect the value of the search tree and is therefore not applicable. Shallow pruning is possible if there is an upper bound for the sum of all players' scores (Korf, 1991).

Depending on the game, coalition forming is a natural behaviour for a player. In this case the general assumption of max^n is unrealistic and too optimistic, resulting in weak play. Several variations of the algorithm try to overcome one or both conceptual drawbacks, e.g. Careful max^n (Lorenz and Tscheuschner, 2006) and soft-max^n (Sturtevant and Bowling, 2006). Constructing an evaluation function that respects possible coalition forming is another possibility to let max^n play more carefully.

4.6.2 Paranoid

The paranoid algorithm (Sturtevant and Korf, 2000) reduces any n-player game to a two-player game. According to the algorithm's general assumption, all opponents form a coalition against the root player. In contrast to max^n, paranoid search allows $\alpha\beta$-pruning. An example paranoid tree is depicted in Figure 4.6.

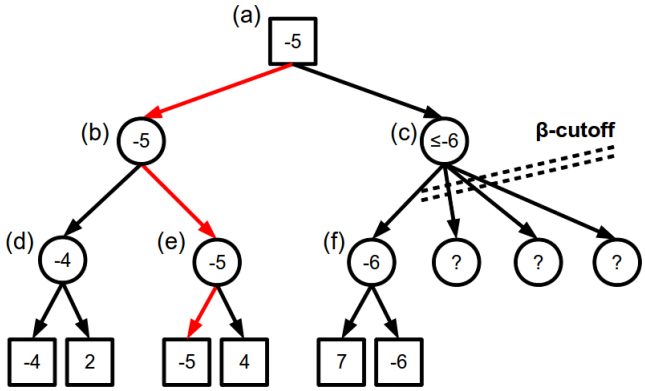


Figure 4.6: An example paranoid tree for three players

In this three-player setup both opponents behave as MIN players trying to minimize the score of the root player, the only MAX player. At Node (b), the second player has two moves leading to Node (d) and Node (e). The third player has two moves at both nodes. As both opponents are MIN players, Node (b) has a value of -5. After searching the first child of Node (c), all remaining children can be safely pruned. Therefore, Node (a) has a value of -5.

In best case, pruning reduces the search complexity to $O(b^{d\times(n-1)/n})$ (Sturtevant and Korf, 2000). This usually leads to higher search depth and therefore paranoid search might outperform max^n, e.g. in Chinese Checkers, Hearts (Sturtevant, 2003a), Focus and Rolit (Schadd and Winands, 2011). The main drawback of paranoid search is that the paranoid assumption is unrealistic and often leads to defensive play. Furthermore, given sufficient search time the algorithm might find out that the current position is a loss because it is usually not possible to win against cooperating opponents. In this case, the player starts to play weak. Designing an evaluation function that emphasizes coalition forming less is a possibility to let paranoid play less defensively.

4.6.3 Best-Reply Search

As seen in the previous subsections, both paranoid and max^n search have conceptual weaknesses. Schadd and Winands (2011) have proposed Best-Reply Search (BRS) as an alternative search technique. Instead of letting all opponents move, only the opponent with the strongest move against root player is allowed to move. Afterwards, the root player is to move again. An example Best-Reply Search tree for three players is depicted in Figure 4.7.

At Node (a), the root player has two options to move. The first option leads to Node (b), a MIN node having four children. Its edges are labelled indicating the player performing a move. Player 2 can perform two different moves. Although it is the second player's turn, the moves of the third player are considered as well. At Node (b), Player 3 has also two options to move and its second move minimizes the utility. At Node (c), the second move of the second player, which is the third child node, causes a β-cutoff. It is notable that move ordering allows to mix the opponents' moves. This is important as the branching factor of a uniform game tree is $(n-1)$-times larger at MIN nodes than at MAX nodes, where n is the number of players.

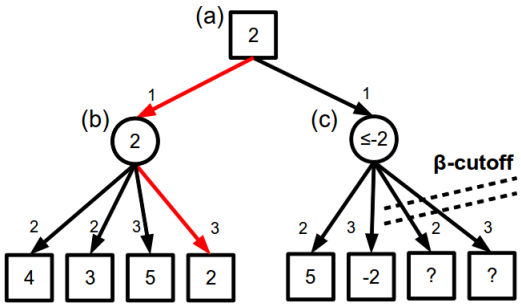


Figure 4.7: An example BRS tree for three players

Best-Reply Search has two advantages compared to max^n and paranoid. In both max^n and paranoid search, a MAX node occurs at every n^{th} ply. This allows only short-term planning when playing against many opponents as only one own turn might be considered. In Best-Reply Search every second node along the search path is a MAX node which enables more long-term planning. Best-Reply Search does not depend on unrealistic coalition assumptions like max^n and paranoid. In contrast to max^n, Best-Reply Search allows deep pruning. In the best case, $O\left((b \times (n-1))^{\lceil \frac{2 \times d}{n} \rceil / 2}\right)$ nodes are explored (Schadd and Winands, 2011). Best-Reply Search also has two drawbacks. First, illegal game states and unreachable game states are investigated during the search. For example in trick-based card games like Hearts and Spades, the investigated states can be considerably different from legal game states, such that Best-Reply Search performs weak. Second, Best-Reply Search does not consider coalitions that are beneficial for the root player.

4.7 Variations of Best-Reply Search

To overcome the conceptual drawbacks of BRS, two ideas are proposed. First, instead of just performing the best move against the root player, all remaining opponents perform the best move according to the static move ordering as well. As a consequence, only legal positions are searched. Second, instead of just applying one move against root player, all opponents except one are allowed to search for a strong move. The two concepts can be combined such that three variations of Best-Reply Search are proposed. The following subsections will discuss the variations BRS$_{1,C-1}$ (Subsection 4.7.1), BRS$_{C-1,0}$ (Subsection 4.7.2) and BRS$_{C-1,1}$ (Subsection 4.7.3). The indices indicate the number of opponents searching for the best counter move, followed by the number of opponents performing the best move according to the static move ordering. C is the number of all competitors.[1] For instance, BRS can also be noted as BRS$_{1,0}$.

4.7.1 BRS$_{1,C-1}$

The first variation of BRS is BRS$_{1,C-1}$. Instead of not letting the opponents move, they are allowed to perform the best move according to the static move ordering. First, the general concept of the algorithm is explained. Therefore a game with n players is considered. After applying one of the MAX player's moves, all moves of the first opponent are generated and ordered first. They are first performed on the board. Afterwards the best move ordering moves (BMOM) are applied for the remaining $n-2$ opponents one after another. The next node would be then a MAX node. After the moves of the first opponent have been explored, the BMOM of the first opponent is selected and applied. All moves for the second opponent are generated. Each of them is applied and afterwards the BMOMs for the remaining $n-3$ opponents are performed one after another, respectively. Again, the next node is then a MAX node. This is repeated $(n-1)$-times so that every opponent's moves have been searched once. An example tree for three players is depicted in Figure 4.8.

At Node (a), which is a MAX node, all moves of the root player are generated. Node (b) is a MIN node and the second player has two moves. The first move leads to Node (d) and second move leads to Node (e). At both nodes, the third player only performs the BMOM. The values of the leaf nodes, which in this example are MAX nodes, are back propagated to update the $\alpha\beta$-window at Node (b). After that,

[1] In order to resolve ambiguities, the term *competitor C* is used instead of *opponent O*. Both terms are equivalent.

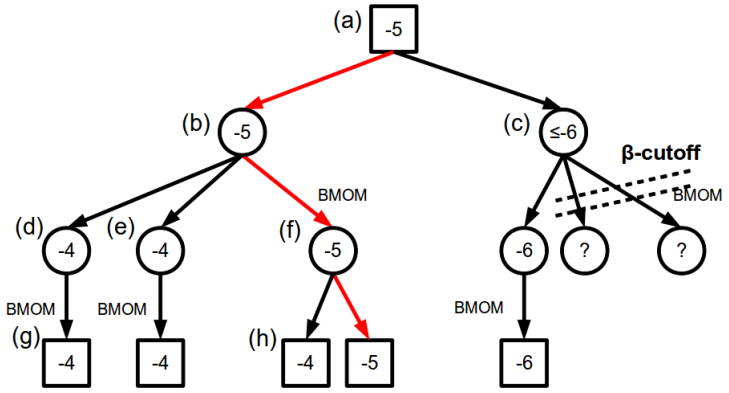


Figure 4.8: Concept of $\text{BRS}_{1,C-1}$ for three players

the second player performs its BMOM. At Node (f), the third player has to two options to move and chooses -5. At Node (c), a cutoff occurs after checking the first move of the second player and the BMOM of the third player.

If it is assumed that the first move investigated is the one preferred by the static move ordering, Node (g) and (h) are transpositions. The edges from (b) to (d) and from (b) to (f) as well as from (d) to (g) and (f) to (h) represent the same moves. In order to avoid researching transpositions, a different implementation of the algorithm is used. Again, a game with n players is considered. After applying one of the MAX players moves, all moves of the first opponent are generated and searched. If the just applied move is the BMOM, the next opponent continues searching. Otherwise, the BMOMs for the remaining opponents are selected. As all opponents have moved, the next node is a MAX node again. The adapted example of the $\text{BRS}_{1,C-1}$ tree for three players is depicted in Figure 4.9.

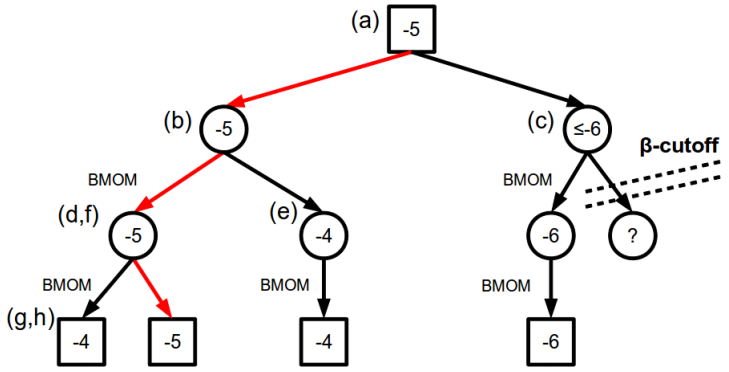


Figure 4.9: Optimized $\text{BRS}_{1,C-1}$ tree for three players

This variation of BRS does not generate illegal positions as all players move according to the rules. Combining the MIN nodes of a branch to a *merged* MIN node, the *effective* branching factor of a merged MIN node increases almost linearly with the number of opponents instead of exponentially like in $\max^n$ and paranoid search assuming a uniform game tree. At a MIN node, only one child is expanded completely and for $b-1$ children BMOMs are generated. The number of MAX nodes followed by a merged MIN node is given in Equation 4.3. As $b = 2$ and $n = 3$ holds in the example tree in Figure 4.9, there are 3 MAX nodes at the left branch.

$$
\begin{aligned}
N_{MAX}(n) &= \begin{cases} (b-1) + N_{MAX}(n-1) & \text{if } n > 2 \\ b & \text{if } n = 2 \end{cases} \\
&= \underbrace{(b-1) + (b-1) + \cdots + (b-1)}_{(n-2)\text{-times}} + b \\
&= (b-1) \times (n-2) + b \\
&= (b-1) \times (n-1) - (b-1) + b \\
&= (b-1) \times (n-1) + 1
\end{aligned}
\tag{4.3}
$$

As the effective branching factor at merged MIN nodes is smaller than the one of $\max^n$ and paranoid, it is possible to have more long-term planning like in the standard version of BRS. Furthermore, it is neither too optimistic like $\max^n$ nor too pessimistic like paranoid. It also has three drawbacks. First, $\text{BRS}_{1,C-1}$ works similarly to paranoid search, where not all opponents' moves are considered, and therefore it is still not aware of coalitions that might help the root player like in paranoid and BRS. Second, generating BMOMs often requires computing all moves which is computationally expensive and causes overhead at the non-expanded MIN nodes. Third, when applying dynamic move ordering techniques like killer moves which are designed to produce a cutoff before generating the moves, it is not known whether the stored moves are equal to the best move according to static move ordering. If stable search is preferred, it is required to generate and order the moves before searching them at MIN nodes otherwise the search becomes unstable. In a two-player game, the algorithm behaves like minimax search.

The transposition tables can only be used at MAX nodes and at MIN nodes that are expanded. Assume that Node (d,f) and (e) of the example tree (Figure 4.9) are transpositions. Node (d,f) is fully expanded, but at Node (e) only the best move ordering move is considered. Therefore, Node (d,f) has a value of -5 while Node (e) has a value of -4. As the data in transposition tables are independent from the search path, it is not known whether the stored node is completely expanded or not.

Computational Overhead of $\text{BRS}_{1,C-1}$

As previously pointed out, generating the BMOMs causes computational overhead. At each expanded MIN node, $(b-1)$ moves are not the best according to the move ordering. For each of those moves a sequence of BMOMs to the next MAX node has to be computed. The number of additionally generated BMOMs per merged MIN node in the uniform search tree is given in Equation 4.4. Thus, its complexity is $O(b \times n^2)$.

$$
\begin{aligned}
N_{Add.BMOMs}(n) &= \begin{cases} (n-2) \times (b-1) + N_{Add.BMOMs}(n-1) & \text{if } n > 2 \\ 0 & \text{if } n = 2 \end{cases} \\
&= \underbrace{(n-2) \times (b-1) + (n-3) \times (b-1) + \cdots + (n-(n-2+1))(b-1)}_{(n-2)\text{-times}} + 0 \\
&= \sum_{i=1}^{n-2} i(b-1) \\
&= (b-1) \sum_{i=1}^{n-2} i \\
&= (b-1) \frac{(n-1)(n-2)}{2}
\end{aligned}
\tag{4.4}
$$

Best-Case Analysis of $\text{BRS}_{1,C-1}$

This best-case analysis calculates the minimum number of leaf nodes to be examined within the game tree. Therefore, a strategy for the MIN and MAX player is needed. Analogous to the best case analysis for the BRS tree (Schadd and Winands, 2011), the search depth d reduces to $\lceil \frac{2 \times d}{n} \rceil$ because the layers of n successive players are reduced to 2 layers, the MAX layer and the MIN layer (between two MAX

nodes). For finding a strategy for the MAX player, 1 node has to be searched at a MAX layer and $(b-1)\times(n-1)+1$ merged MIN nodes at a MIN layer, resulting in $((b-1)\times(n-1)+1)^{\lceil\frac{2\times d}{n}\rceil/2}$ nodes. For finding a strategy for the MIN player, b nodes have to be searched at a MAX layer and 1 merged MIN node at a MIN layer, resulting in $b^{\lceil\frac{2\times d}{n}\rceil/2}$ nodes. Therefore, the total number of leaf nodes for both the MIN and MAX player is $((b-1)\times(n-1)+1)^{\lceil\frac{2\times d}{n}\rceil/2}+b^{\lceil\frac{2\times d}{n}\rceil/2}$ nodes. Thus, $\text{BRS}_{1,C-1}$ explores in the best case $O\left(((b-1)\times(n-1)+1)^{\lceil\frac{2\times d}{n}\rceil/2}\right)$ leaf nodes.

With respect to the computational overhead, the total number of MIN nodes in a merged MIN node is computed as well. The first MIN node has $b-1$ successors where $n-2$ BMOMs are applied until the next MAX node is reached and 1 successor that expands similarly to the first MIN node. Therefore, the number MIN nodes in a merged node is computed recursively (cf. Equation 4.5).

$$
\begin{aligned}
N_{MIN}(n) &= \begin{cases} 1+(b-1)\times(n-2)+N_{MIN}(n-1) & \text{if } n>2 \\ 1 & \text{if } n=2 \end{cases} \\
&= \underbrace{1+(b-1)\times(n-2)+1+(b-1)\times(n-3)+\cdots+(b-1)\times(n-(n-2+1))}_{(n-2)\text{-times}}+0 \\
&= (n-1)+(b-1)\times\sum_{i=2}^{n-1}(n-i) \\
&= (n-1)+(b-1)\times\left(\sum_{i=2}^{n-1}n-\sum_{i=2}^{n-1}i\right) \\
&= (n-1)+(b-1)\times\left((n-1-2+1)\times n-\sum_{i=1}^{n-1}i+1\right) \\
&= (n-1)+(b-1)\times\left((n-2)\times n-\frac{(n-1)(n)}{2}+1\right) \\
&= (n-1)+(b-1)\times\left(\frac{2n^2-4n-n^2+n}{2}+1\right) \\
&= (n-1)+(b-1)\times\left(\frac{n^2-3n}{2}+1\right)
\end{aligned}
\tag{4.5}
$$

The effective branching factor of $\text{BRS}_{1,C-1}$ is smaller than the one of BRS, but the number of nodes that are combined to a merged MIN node is higher than in BRS. Considering the overhead, $\text{BRS}_{1,C-1}$ has a complexity of $O\left((n-1)\times((b-1)\times(n-1)+1)^{\lceil\frac{2\times d}{n}\rceil/2}\right)$ in the best case as for every expanded MIN node at most $n-2$ moves are generated.

4.7.2 $\text{BRS}_{C-1,0}$

The second variation of Best-Reply Search is $\text{BRS}_{C-1,0}$. Instead of letting only one opponent move between two MAX nodes along the search path, all opponents move except one. For instance in a four-player game, the move sequences where only Opponent 1 and Opponent 2 move are searched first. After that, the move sequences where Opponent 1 and Opponent 3 move are investigated. Finally, the moves of Opponent 2 and Opponent 3 are considered. An example $\text{BRS}_{C-1,0}$ tree for four players is depicted in Figure 4.10.

For simplicity reasons, the MAX player has only one move at Node (a). At Node (b), the second player considers two moves leading to Node (d) and (e). At Node (d), the third player has two possible moves. Both moves are connected to a MAX node as the moves of player 4 are ignored. When skipping the moves of the third player at Node (d), there are two moves for the fourth player. Player 3 chooses left child having a value of 3. At Node (e), there are two moves when skipping the fourth player and also two moves when skipping the third player. The value 1 is back propagated to its parent node. When skipping the moves of the second player at Node (b), there are two moves for the third player, that lead to Node (f) and (g). At both nodes, the left child is chosen.

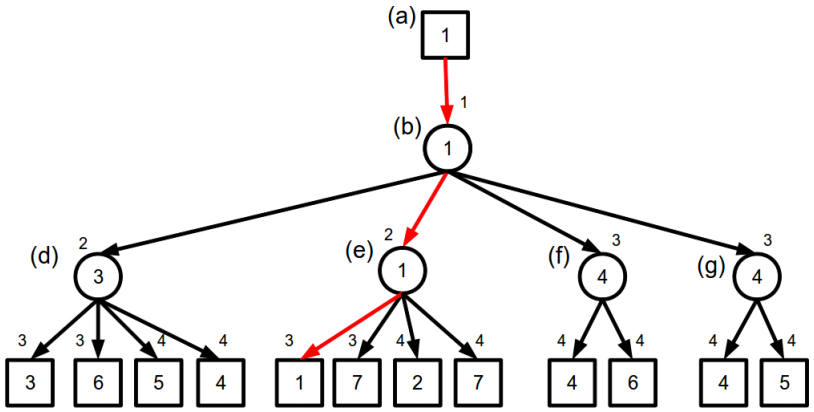


Figure 4.10: An example $\text{BRS}_{C-1,0}$ tree for four players

$\text{BRS}_{C-1,0}$ is a more paranoid variation of BRS having similar properties. On the one hand, the occurrence of MAX nodes is higher than for traditional search methods as every $(n-1)^{th}$ node along the search path is a MAX node. On other hand, the effective branching factor at a MIN node is bigger. The first MIN node has b children where the first opponent is skipped and therefore no other opponent will be ignored until the next MAX node is reached. Furthermore, it has b children which are of a similar type as first MIN node because the first opponent performs a move. Thus, the effective branching factor is $2b$ if no opponent has been skipped, otherwise it is b. The number of MAX nodes followed by a merged MIN node is given in Equation 4.6.

$$\begin{aligned} N_{MAX}(n) &= \begin{cases} b^{n-2} + b \times N_{MAX}(n-1) & \text{if } n > 3 \\ 2b & \text{if } n = 3 \end{cases} \\ &= \underbrace{b^{n-2} + b \times b^{n-3} + \cdots + b^{n-2} \times b^{n-(n-3+1)-2}}_{(n-3)\text{-times}} + b^{n-3} \times 2b \\ &= (n-3) \times b^{n-2} + b^{n-3} \times 2b = (n-3) \times b^{n-2} + 2b^{n-2} \\ &= (n-1) \times b^{n-2} \end{aligned} \tag{4.6}$$

Its general assumption is less paranoid compared to paranoid search as fewer opponents move against the root player. Furthermore, it has the same drawbacks as Best-Reply Search. It is still not possible to consider opponents' moves that are beneficial for the root player and illegal positions are searched. The usage of transposition tables is limited. At MIN nodes, the transposition tables can only be used if no player has been skipped so far. Without this limitation, a lookup can incorrectly affect the search tree as in one case the next player is allowed to perform another move and the second case it is skipped. In a three-player game, $\text{BRS}_{C-1,0}$ behaves exactly as the standard version of BRS. In a two-player game, the single opponent is not skipped and therefore $\text{BRS}_{C-1,0}$ becomes a standard minimax search.

Best-Case Analysis of $\text{BRS}_{C-1,0}$

To calculate the minimum number of leaf nodes to be examined within the game tree, a strategy for the MIN and MAX player is needed. Again, the search depth d reduced to $\lceil \frac{2 \times d}{n} \rceil$ because the layers of n successive players are reduced to a MAX layer and a MIN layer. For finding a strategy for the MAX player, 1 node has to be searched at a MAX layer and $(n-1) \times b^{n-2}$ merged nodes at a MIN layer, resulting in $((n-1) \times b^{n-2})^{\lceil \frac{2 \times d}{n} \rceil / 2}$ nodes. For finding a strategy for the MIN player, b nodes have to be searched at a MAX layer and 1 merged node at a MIN layer, resulting in $b^{\lceil \frac{2 \times d}{n} \rceil / 2}$ nodes. Therefore, the total number of leaf nodes for both the MIN and MAX player is $((n-1) \times b^{n-2})^{\lceil \frac{2 \times d}{n} \rceil / 2} + b^{\lceil \frac{2 \times d}{n} \rceil / 2}$ nodes. Thus, $\text{BRS}_{C-1,0}$ explores in the best case $O\left(((n-1) \times b^{n-2})^{\lceil \frac{2 \times d}{n} \rceil / 2}\right)$ leaf nodes.

4.7.3 BRS$_{C-1,1}$

BRS$_{C-1,1}$ combines the ideas of the variants BRS$_{1,C-1}$ and BRS$_{C-1,0}$. Similar to BRS$_{C-1,0}$, all opponents moves are searched except one. Instead of ignoring that move, the best move ordering move is performed like in BRS$_{1,C-1}$. An example tree for four players is depicted in Figure 4.11.

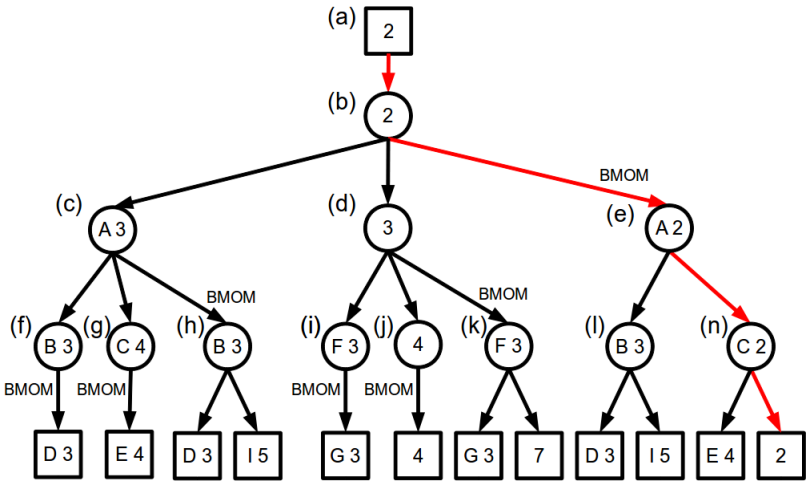


Figure 4.11: Concept of BRS$_{C-1,1}$ for four players

For simplicity reasons, the MAX player has only one move at Node (a). The game tree where Node (b) is the root node is uniform with a branching factor of 2. However the search tree looks differently. At Node (b), the second player has two moves leading to Node (c) and Node (d). At Node (c), the third player has also two moves leading to Node (f) and (g). At Node (f), the fourth player is only allowed to perform its BMOM as the MIN nodes of the second and the third player are completely expanded. The same holds for Node (g). At Node (c), it is also considered to perform the BMOM of the third player such that all moves of the fourth player are investigated at Node (h). The search at Node (d) behaves similarly. Both, Node (c) and Node (d) have a value of 3. Going back to Node (b), it is also considered that the second player is only allowed to apply its BMOM such that both the third and the fourth player are completely expanded, resulting in Node (e) having a value of 2.

In this conceptual tree, many transpositions occur while searching although the game tree has none. Next to the values inside the nodes, a capital letter labels the corresponding game state. The game states A, C, E, F, G and I occur two times and the game states B and D are visited even three times. Again, a different implementation can avoid these transpositions. A standard paranoid search is performed but the last MIN node before a MAX node is only expanded when at least one of the previous MIN players performed a best move ordering move. The adapted example of the BRS$_{C-1,1}$ tree for four players is depicted in Figure 4.12. The dashed edges show which paths are not searched in BRS$_{C-1,1}$, but in paranoid search.

BRS$_{C-1,1}$ is the most paranoid variation of BRS among our proposals. Except for ignoring the paths where not at least one player performs a BMOM, the complete game tree is searched. This reduces the number of the MIN nodes between two MAX nodes from b^{n-1} to $n \times b^{n-2}$ in the conceptual tree and even to $b^{n-1} - (b-1)^{n-1}$ in the optimized one, where b is the average branching factor and n is the number of players (cf. Section B.1). If the game tree is uniform, the BRS$_{C-1,1}$ tree is a paranoid tree reduced by second paranoid tree with $b' = b - 1$, which is the number of Non-BMOMs. In contrast to the standard version of BRS only legal states are searched. Therefore, it is required to generate BMOMs which causes more overhead. Like in BRS$_{1,C-1}$, the preference of stable search might requires to generate all moves before applying killer moves. Furthermore, transposition tables can only be used at MAX nodes and expanded MIN nodes like in BRS$_{1,C-1}$. In a three-player game, BRS$_{C-1,1}$ behaves exactly as BRS$_{1,C-1}$. In a two-player game, the algorithm behaves like minimax search.

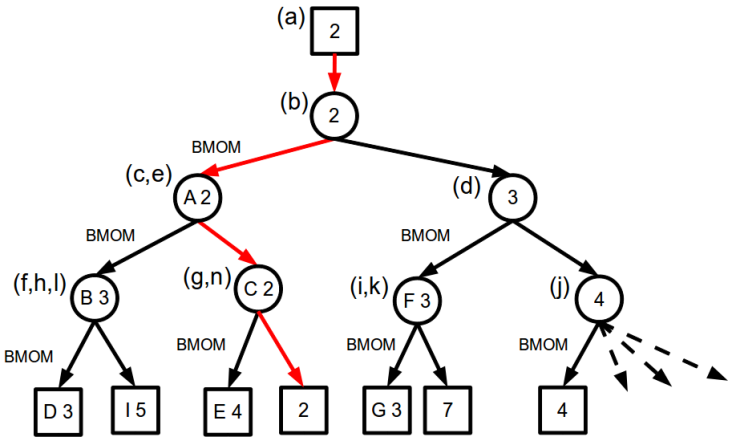


Figure 4.12: Optimized $\text{BRS}_{C-1,1}$ tree for four players

Computational Overhead of $\text{BRS}_{C-1,1}$

As previously pointed out, generating the BMOMs causes computational overhead. At each expanded MIN node, $(b-1)$ moves are not optimal according to the move ordering. There are $(b-1)^{n-2}$ last MIN nodes before a MAX node, where it is required to generate one additional BMOM. Thus, the number of additionally generated BMOMs in uniform search tree per merged MIN node is also $(b-1)^{n-2}$. The resulting complexity is $O(b \times n)$.

Best-Case Analysis of $\text{BRS}_{C-1,1}$

To calculate the minimum number of leaf nodes to be examined within the game tree, a strategy for the MIN and MAX player is needed. The search depth d reduces to $\lceil \frac{2\times d}{n} \rceil$ like in $\text{BRS}_{1,C-1}$ and $\text{BRS}_{C-1,0}$. For finding a strategy for the MAX player, 1 node has to be searched at a MAX layer and $b^{n-1} - (b-1)^{n-1}$ merged nodes at a MIN layer, resulting in $(b^{n-1} - (b-1)^{n-1})^{\lceil \frac{2\times d}{n} \rceil/2}$ nodes. For finding a strategy for the MIN player, b nodes have to be searched at a MAX layer and 1 merged node at a MIN layer, resulting in $b^{\lceil \frac{2\times d}{n} \rceil/2}$ nodes. Therefore, the total number of leaf nodes for both the MIN and MAX player is $(b^{n-1} - (b-1)^{n-1})^{\lceil \frac{2\times d}{n} \rceil/2} + b^{\lceil \frac{2\times d}{n} \rceil/2}$ nodes. Thus, $\text{BRS}_{C-1,1}$ explores in the best case $O\left((b^{n-1} - (b-1)^{n-1})^{\lceil \frac{2\times d}{n} \rceil/2}\right)$ leaf nodes.

With respect to the computational overhead, the total number of MIN nodes in a merged MIN node is computed as well. The $\text{BRS}_{C-1,1}$ tree is a reduced paranoid tree. The standard paranoid tree has b^{n-1} MAX nodes followed by a merged MIN node. As the number of internal nodes is equal to the number of external nodes, there are $2b^{n-1}$ nodes in total. The number of leaf nodes of a paranoid tree with $b' = b-1$ is subtracted. The total number of MIN nodes combined in a merged MIN node is $2b^{n-1} - (b-1)^{n-1}$. Considering the overhead, the complexity of $\text{BRS}_{C-1,1}$ is $O\left((b^{n-1} - (b-1)^{n-1})^{\lceil \frac{2\times d}{n} \rceil/2}\right)$ in the best case. For every expanded MIN node, it is not required to generate more than one BMOM such that the complexity increases by a constant.

4.8 Evaluation Function

The $\alpha\beta$-based search algorithms require an evaluation function to assign a utility to a leaf node in the search tree. The utility of a game state correlates with the winning possibility and game-theoretic value of the player. If the game ends at a leaf node the utility indicates whether the player has won the game or not. When using $\max^n$ search, every player's utility is computed, otherwise, only the position of the root player is evaluated. The evaluation function can either be dynamic or static. An dynamic evaluation function gathers statistics for the current state. For instance, Monte Carlo Evaluation simulates multiple games (semi-)randomly until a terminal node is reached (Abramson, 1990). The state utility is set to the average of the simulated scores. Just as in Monte-Carlo Tree Search (Kocsis and Szepesvári, 2006; Coulom, 2007), the strength of this technique depend on the quality of the (semi-)randomly chosen

moves. A heuristic evaluation function usually consists of several features. The importance of these features differs such that each feature is weighted. As the game of Billabong is a multi-player game, each feature belongs to a player. The sum of all weighted features belonging to a player represents the progress of this particular player ignoring the progress of the other players (cf. Equation 4.7). Finally, the utility of the game state is the progress of the evaluating player minus the sum of all opponents' progresses multiplied by a factor c (cf. Equation 4.8). The factor, so called *Player-Opponent-Balance*, influences whether the own strength is preferred to the opponents' weaknesses.

$$Progress(playerId) = \sum_{i} w_i \times f_i \tag{4.7}$$

$$Utility(playerId) = Progress(playerId) - c \times \sum_{\substack{i=1 \\ i \neq playerId}}^{numPlayers} Progress(i) \tag{4.8}$$

For the game of Billabong the following features can be evaluated for each player.

Total Angle The *total angle* shows the covered distance by all pieces and is the sum of the angles of the pieces of one player that are on the board. The angle is computed as described in Subsection 4.3.1.

Removed Pieces The number of pieces that have been removed from the board is taken into account.

Angle of the Last Piece The *angle of the last piece* adds a penalty of letting a piece behind. As described in Section 2.3, it is usually difficult to close a large gap to the peloton.

Long jump The *long jump* rates the strongest jump possibility for all pieces. It is computed by summing up the static move ordering score of the long jump move for every piece (cf. Subsection 4.3.1).

The first three features were proposed by Solomon (1984). He combined the total-angle and the removed-pieces features with an additional weighting, such that for each piece that is removed from the board the value $4 \times 360^\circ = 1440$ is added to the total angle. Therefore, he used the formula $X^2 + 2X$ to compute the round of a piece, where $X = -1$ indicates that a piece has not crossed the start-finish line yet. The evaluation function that is used in this thesis emphasizes removing a piece more if the corresponding weight is greater than 1. The computation of the long jump is computationally expensive as all moves need to be generated. This reduces the search depth. Finally, the utility of the state is randomly increased or decreased by up to 5%. Randomness leads to safer play. The player is more likely to prefer a state where it has multiple options to perform a strong move (Beal and Smith, 1994). The specific weights were not given by Solomon, such that they are determined experimentally in Section 5.2.

Chapter 5

Experiments & Results

In this chapter, the experiments performed are described and analysed. First, Section 5.1 introduces the experimental setup. Next, Section 5.2 compares the performance of different configurations of the evaluation function. The strength of the move ordering techniques is evaluated in Section 5.3. Afterwards, the average search depth that the different search algorithms are able to achieve is shown in Section 5.4. Subsequently, Section 5.5 matches the performance of BRS, $\text{BRS}_{1,C-1}$, $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ with $\max^n$ and paranoid. Finally, the four BRS versions, BRS and its variations, directly compete each other in different experiments described in Section 5.6.

5.1 Experimental Setup

The experiments that are described in the following sections are performed in self-play. Therefore, a Billabong program is developed in Java 6. Sequentially, the algorithms to be matched search for the best move in the current board situation given a thinking time of 5 seconds if not specified differently. Each algorithm uses the evaluation function and the enhancements which are considered to be optimal according to the results of Section 5.2 and 5.3. The tests are performed on a 64-bit Linux machine with 2.4GHz.

When matching m algorithms in an n-player game, it is required to test different configurations. Further, as it is unknown whether the winning possibilities depend on being the first, second, third or fourth player, these configurations are tested in all possible orders. Table 5.1 lists all tested configurations with algorithms A, B, C and D. For all experiments consisting of at least 200 games, the winning percentage and the 95%-confidence interval are given.

5.2 Evaluation Function

Section 4.8 described the different features of the evaluation function. The first experiment finds the best configuration of its weights. Therefore, the five configurations shown in Table 5.2 are tested. The five evaluation functions are tested in subsets of four. For each subset, e.g. Configuration 1, 2, 4 and 5, 432 games are simulated in total. The tests are performed for all six search algorithms such that 72 games are simulated for each search algorithm. As the influence of the search algorithm itself has to be as small as possible, the search algorithms are not mixed in the games such that e.g. $\max^n$ is just matched against $\max^n$.

Configuration 1 and 2 are the strongest configurations. Configuration 4 is comparatively good and Configuration 3 and 5 perform weak. The total angle and the angle of the last piece are important features. The long jump might improve the strength of the evaluation function but its merit does not outperform a deeper search. A search that does not analyse the long jump is about one ply deeper. When putting too much weight on this feature, the player performs weak. The player prefers positions where it is able to do a long jump move to positions where it performed them already. Further, the opponents' strength is less important than the progress of the player itself such that a high weight decreases performance. As Configuration 1 has the highest win ratio, it is used for all following experiments.

Players	Search Algorithms	Configurations
3	2	6: $(A\ A\ B)$; $(A\ B\ A)$; $(B\ A\ A)$; $(A\ B\ B)$; $(B\ A\ B)$; $(B\ B\ A)$
	3	6: $(A\ B\ C)$; $(A\ C\ B)$; $(B\ A\ C)$; $(B\ C\ A)$; $(C\ A\ B)$; $(C\ B\ A)$
4	2	14: $(A\ A\ A\ B)$; $(A\ A\ B\ A)$; $(A\ B\ A\ A)$; $(B\ A\ A\ A)$; $(A\ A\ B\ B)$; $(A\ B\ A\ B)$; $(A\ B\ B\ A)$; $(B\ A\ A\ B)$; $(B\ A\ B\ A)$; $(B\ B\ A\ A)$; $(A\ B\ B\ B)$; $(B\ A\ B\ B)$; $(B\ B\ A\ B)$ $(B\ B\ B\ A)$
	3	36: $(A\ A\ B\ C)$; $(A\ A\ C\ B)$; $(A\ B\ A\ C)$; $(A\ C\ A\ B)$; $(A\ B\ C\ A)$; $(A\ C\ B\ A)$; $(B\ A\ A\ C)$; $(C\ A\ A\ B)$; $(B\ A\ C\ A)$; $(C\ A\ B\ A)$; $(B\ C\ A\ A)$; $(C\ B\ A\ A)$; $(B\ B\ A\ C)$; $(B\ B\ C\ A)$; $(B\ A\ B\ C)$; $(B\ C\ B\ A)$; $(B\ A\ C\ B)$; $(B\ C\ A\ B)$; $(A\ B\ B\ C)$; $(C\ B\ B\ A)$; $(A\ B\ C\ B)$; $(C\ B\ A\ B)$; $(A\ C\ B\ B)$; $(C\ A\ B\ B)$; $(C\ C\ B\ A)$; $(C\ C\ A\ B)$; $(C\ B\ C\ A)$; $(C\ A\ C\ B)$; $(C\ B\ A\ C)$; $(C\ A\ B\ C)$; $(B\ C\ C\ A)$; $(A\ C\ C\ B)$; $(B\ C\ A\ C)$; $(A\ C\ B\ C)$; $(B\ A\ C\ C)$; $(A\ B\ C\ C)$
	4	24: $(A\ B\ C\ D)$; $(A\ B\ D\ C)$; $(A\ C\ B\ D)$; $(A\ C\ D\ B)$; $(A\ D\ B\ C)$; $(A\ D\ C\ B)$; $(B\ A\ C\ D)$; $(B\ B\ D\ C)$; $(B\ C\ A\ D)$; $(B\ C\ D\ A)$; $(B\ D\ A\ C)$; $(B\ D\ C\ A)$; $(C\ A\ B\ D)$; $(C\ A\ D\ B)$; $(C\ B\ A\ D)$; $(C\ B\ D\ A)$; $(C\ D\ A\ B)$; $(C\ D\ B\ A)$; $(D\ A\ B\ C)$; $(D\ A\ C\ B)$; $(D\ B\ A\ C)$; $(D\ B\ C\ A)$; $(D\ C\ A\ B)$; $(D\ C\ B\ A)$

Table 5.1: Overview of test configurations

Features	Configuration				
	1	2	3	4	5
Total Angle	1	1	1	1	1
Removed Pieces	1	1.2	1	1.2	1
Angle of the Last Piece	3	6	3	3	3
Long Jump	0	0.2	0.7	0.4	0
Player-Opponent-Balance	0.2	0.1	0.2	0.2	1.0
search depth	5.2	4.2	4.1	4.2	5.0
winning percentage	39.7 (±2.31)	39.1 (±2.30)	4.1 (±0.93)	34.4 (±2.24)	7.8 (±1.26)

Table 5.2: Winning percentage for different weighting configurations for the used evaluation function

5.3 Move Ordering

The search algorithms max^n, paranoid, BRS, BRS$_{1,C-1}$, BRS$_{C-1,0}$ and BRS$_{C-1,1}$ as well as possible enhancements like the transposition tables, the killer and the history heuristics were discussed in Chapter 4. Given a set of 125 board positions for four players that were randomly selected from self-play experiments, the number of nodes to be searched given a fixed depth are compared. This shows how effective the technique can be applied in the game of Billabong. The fewer nodes have to be searched given a fixed depth the deeper an algorithm can search given the same the computation time. All tests use iterative deepening and Configuration 1 as evaluation function (cf. Section 4.8). The randomness in the evaluation function is disabled such that the same search trees are investigated.

Table 5.3 presents the numbers of nodes for the five different algorithms paranoid, BRS, BRS$_{1,C-1}$, BRS$_{C-1,0}$ and BRS$_{C-1,1}$. For each algorithm, the full and the $\alpha\beta$-pruned trees are investigated. Except for $\alpha\beta$-pruning and static move ordering, there are no enhancements applied. The static move ordering is strong for all algorithms as least 89.9% and up to 99.9% of the nodes are pruned. As there are no pruning techniques for max^n, it investigates the same number of nodes as the non-pruning version of paranoid.

Depth	Paranoid		BRS		$BRS_{1,C-1}$		$BRS_{C-1,0}$		$BRS_{C-1,1}$	
	full	$\alpha\beta$-pruned	full	$\alpha\beta$-pruned	full	$\alpha\beta$-pruned	full	$\alpha\beta$-pruned	full	$\alpha\beta$-pruned
2	1,736	160 (90.2%)	5,120	275 (94.6%)	1,736	160 (90.8%)	3,446	292 (91.5%)	1,736	160 (90.8%)
3	89,027	5,046 (94.3%)	282,401	3,170 (98.9%)	6,748	595 (91.2%)	266,028	23,677 (91.1%)	89,027	5,046 (94.3%)
4	5,094,052	337,456 (93.4%)	47,614,714	57,894 (99.9%)	16,706	1,703 (89.8%)	16,025,835	79,353 (99.5%)	431,512	35,739 (91.7%)
5	-	1,130,163	-	382,401	298,548	6,689 (97.8%)	-	2,516,949	16,172,459	107,205 (99.3%)
6	-	21,226,842	-	10,105,757	15,723,237	56,158 (99.6%)	-	194,938,281	-	1,909,668
7	-	-	-	-	61,625,460	215,181 (99.7%)	-	-	-	83,107,057
8	-	-	-	-	-	533,565	-	-	-	-
9	-	-	-	-	-	1,471,080	-	-	-	-
10	-	-	-	-	-	15,015,015	-	-	-	-

Table 5.3: Nodes to be searched with and without $\alpha\beta$-pruning

When searching up to a depth of 4 in a four-player game, the leaf nodes in paranoid are MAX nodes. The same holds for $BRS_{1,C-1}$ and $BRS_{C-1,1}$ as both are sparse versions of paranoid search. For BRS, it is only required to search up to a depth of 2 and for $BRS_{C-1,0}$ up to a depth of 3, respectively. Considering the full trees explored up to the first MAX node after the root node, BRS investigates 5,120 nodes, $BRS_{1,C-1}$ investigates 16,706 nodes, $BRS_{C-1,0}$ investigates 266,028 nodes, $BRS_{C-1,1}$ investigates 431,512 nodes and paranoid investigates 5,094,052 nodes. Therefore, BRS has the largest lookahead.

The potential of transposition tables is shown in Tables 5.4, 5.5, 5.6, 5.7 and 5.8. The transposition tables can store $2^{24} = 16,777,216$ positions requiring 240 mega bytes memory. The current node is looked up in the transposition table before generating the moves. As stable search is preferred, the BMOM is determined using static move ordering. For all search algorithms, transposition tables perform well. As expected, the replacement scheme Old performs worst. Once the transposition table is full, it is not possible to use its data as the board states will not occur anymore. Two-Deep and Deep are similarly good. There is a tendency to consider Deep to be the best scheme when exploring up to 2,500,000 nodes which can be searched within five seconds computation time. In this setup, it is possible to reduce the number of nodes to be searched up to 49.1% for $BRS_{1,C-1}$. This algorithm benefits most from the usage of transposition tables. For paranoid, BRS and $BRS_{C-1,1}$ the amount of nodes reduces about 30%. As $BRS_{C-1,0}$ cannot apply transposition tables at every MIN node, the technique can only reduce the number of nodes by 21.5%.

Depth	Paranoid				
	Without TT	Two-Deep	Deep	New	Old
2	160	155 (3.0%)	155 (3.0%)	155 (3.0%)	155 (3.0%)
3	5,046	4,581 (9.2%)	4,581 (9.2%)	4,581 (9.2%)	4,833 (4.2%)
4	337,456	146,582 (56.6%)	146,272 (56.7%)	272,529 (19.2%)	334,679 (0.8%)
5	1,130,163	810,317 (28.3%)	760,383 (32.7%)	999,794 (11.5%)	1,121,278 (0.8%)
6	21,226,842	13,283,194 (37.4%)	14,145,574 (33.4%)	18,788,420 (11.5%)	21,297,621 (-0.3%)

Table 5.4: Nodes to be searched using transposition tables in paranoid

Depth	BRS				
	Without TT	Two-Deep	Deep	New	Old
2	275	263 (4.7%)	263 (4.7%)	263 (4.7%)	263 (4.7%)
3	3,170	2,916 (8.0%)	2,911 (8.2%)	2,911 (8.2%)	2,930 (7.6%)
4	57,894	39,185 (32.3%)	38,941 (32.7%)	39,291 (32.1%)	48,613 (16.0%)
5	382,401	252,191 (34.1%)	246,908 (35.4%)	277,123 (27.5%)	313,661 (18.0%)
6	10,105,757	5,490,533 (45.7%)	5,446,893 (46.1%)	6,825,400 (32.5%)	9,020,974 (10.7%)

Table 5.5: Nodes to be searched using transposition tables in BRS

Depth	$BRS_{1,C-1}$				
	Without TT	Two-Deep	Deep	New	Old
2	160	155 (3.0%)	155 (3.0%)	155 (3.0%)	155 (3.0%)
3	595	558 (6.2%)	558 (6.2%)	558 (6.2%)	571 (3.9%)
4	1,703	1,280 (24.9%)	1,280 (24.9%)	1,281 (24.8%)	1,647 (3.3%)
5	6,689	5,047 (24.5%)	5,045 (24.6%)	5,099 (23.8%)	6,492 (3.0%)
6	56,158	26,670 (52.5%)	26,376 (53.0%)	32,661 (41.8%)	53,283 (5.1%)
7	215,181	106,683 (50.4%)	101,158 (53.0%)	144,389 (32.9%)	201,310 (6.4%)
8	533,565	249,436 (53.3%)	235,436 (55.9%)	375,326 (29.7%)	496,729 (6.9%)
9	1,471,080	782,176 (46.8%)	748,859 (49.1%)	1,171,140 (20.4%)	1,339,545 (8.9%)
10	15,015,015	5,835,126 (61.1%)	5,646,455 (62.4%)	11,910,998 (20.7%)	13,702,448 (8.7%)

Table 5.6: Nodes to be searched using transposition tables in $BRS_{1,C-1}$

Depth	$BRS_{C-1,0}$				
	Without TT	Two-Deep	Deep	New	Old
2	292	284 (3.0%)	284 (3.0%)	284 (3.0%)	284 (3.0%)
3	23,677	18,816 (20.5%)	18,800 (20.6%)	18,800 (20.6%)	19,358 (18.2%)
4	79,353	51,968 (34.5%)	51,505 (35.1%)	67,652 (14.7%)	52,031 (34.4%)
5	2,516,949	1,983,709 (21.2%)	1,975,930 (21.5%)	2,133,236 (15.2%)	1,828,496 (27.4%)
6	194,938,281	153,397,072 (21.3%)	153,028,306 (21.5%)	162,860,932 (16.5%)	178,978,501 (8.2%)

Table 5.7: Nodes to be searched using transposition tables in $BRS_{C-1,0}$

Depth	$BRS_{C-1,1}$				
	Without TT	Two-Deep	Deep	New	Old
2	160	155 (3.0%)	155 (3.0%)	155 (3.0%)	155 (3.0%)
3	5,046	4,581 (9.2%)	4,581 (9.2%)	4,581 (9.2%)	4,833 (4.2%)
4	35,739	18,694 (47.7%)	18,645 (47.8%)	20,867 (41.6%)	34,899 (2.4%)
5	107,205	55,863 (47.9%)	55,172 (48.5%)	85,608 (20.1%)	105,054 (2.0%)
6	1,909,668	1,393,334 (27.0%)	1,391,218 (27.1%)	1,706,110 (10.7%)	1,892,278 (0.9%)
7	83,107,057	46,758,250 (43.7%)	46,678,605 (43.8%)	71,571,946 (13.9%)	81,475,247 (2.0%)

Table 5.8: Nodes to be searched using transposition tables in $BRS_{C-1,1}$

The benefit of history heuristics and killer moves combined with transposition tables using the Deep scheme is shown in Tables 5.9, 5.10, 5.11, 5.12 and 5.13. First, the current node is looked up in the transposition table. If possible, the preserved move is investigated. Second, the killer moves are applied. Finally, the remaining moves are ordered according to the static move ordering. If several moves are equally preferred, history heuristics are applied. For all search algorithms, neither history heuristics nor relative history heuristics improve playing performance a lot. In the best case, it is possible to reduce the number of nodes by 3.6%, but in the worst case the number of nodes even increases by 0.5%. Furthermore, there is no preference for using history heuristics or relative history heuristics. As the static move ordering already distinguishes many categories of moves, it is less likely that the static ordering of the moves is different from the one enhanced with the history heuristics.

The usage of killer moves has a higher impact on the search performance. For all algorithms, there is no significant difference between the application of one or two killer moves. Their potential highly depends on the reached search depth. For instance, killer moves improve a 5-ply paranoid search about 37% while a one ply deeper search gains only 16% to 20%. In BRS, killer moves reduce the number of nodes by 18% (5-ply) and 10% (6-ply). For $BRS_{1,C-1}$, killer moves reduce the number of nodes about 30% to 35% (9-ply and 10-ply). In $BRS_{C-1,0}$, killer moves increase the number of nodes about 7% to 9% in a 4-ply deep search and reduce the number of nodes by 7% to 14% in 5-ply or 6-ply deep search.

Killer moves are powerful in $BRS_{C-1,1}$ as they are able to reduce the number of nodes to be searched in a 6-ply search by 49%.

The previous tests have shown that the number of nodes is minimal for these 125 board positions when using transposition tables combined with relative history heuristics and two killer moves. The transposition tables are updated according to the Deep scheme. The dynamic move ordering techniques are combined with the proposed static move ordering.

Depth	Paranoid						
	HH: no; KM: 0	HH: rel.; KM: 0	HH: abs.; KM: 0	HH: rel.; KM: 1	HH: abs.; KM: 1	HH: rel.; KM: 2	HH: abs.; KM: 2
3	4,581	4,579 (0.1%)	4,579 (0.1%)	2,545 (44.4%)	2,545 (44.4%)	2,462 (46.3%)	2,462 (46.3%)
4	146,272	146,001 (0.2%)	146,001 (0.2%)	134,944 (7.7%)	134,944 (7.7%)	139,259 (4.8%)	139,259 (4.8%)
5	760,383	760,051 (0.0%)	760,069 (0.0%)	484,303 (36.3%)	484,308 (36.3%)	475,204 (37.5%)	475,223 (37.5%)
6	14,145,574	14,152,935 (-0.1%)	14,153,268 (-0.1%)	11,788,956 (16.7%)	11,782,162 (16.7%)	11,335,546 (19.9%)	11,319,253 (20.0%)

Table 5.9: Nodes to be searched using dynamic move ordering in paranoid

Depth	BRS						
	HH: no; KM: 0	HH: rel.; KM: 0	HH: abs.; KM: 0	HH: rel.; KM: 1	HH: abs.; KM: 1	HH: rel.; KM: 2	HH: abs.; KM: 2
3	2,911	2,908 (0.1%)	2,908 (0.1%)	2,711 (6.8%)	2,708 (7.0%)	2,728 (6.3%)	2,727 (6.3%)
4	38,941	38,793 (0.4%)	38,797 (0.4%)	30,900 (20.6%)	30,897 (20.7%)	31,524 (19.0%)	31,546 (19.0%)
5	246,908	242,502 (1.8%)	242,675 (1.7%)	200,953 (18.6%)	200,484 (18.8%)	201,271 (18.5%)	201,591 (18.4%)
6	5,446,893	5,433,474 (0.2%)	5,427,531 (0.4%)	4,891,384 (10.2%)	4,884,633 (10.3%)	4,909,677 (9.9%)	4,912,311 (9.8%)

Table 5.10: Nodes to be searched using dynamic move ordering in BRS

Depth	$BRS_{1,C-1}$						
	HH: no; KM: 0	HH: rel.; KM: 0	HH: abs.; KM: 0	HH: rel.; KM: 1	HH: abs.; KM: 1	HH: rel.; KM: 2	HH: abs.; KM: 2
3	558	557 (0.1%)	557 (0.1%)	460 (17.6%)	460 (17.6%)	439 (21.3%)	439 (21.3%)
4	1,280	1,279 (0.1%)	1,279 (0.1%)	1,073 (16.1%)	1,073 (16.1%)	1,078 (15.7%)	1,078 (15.7%)
5	5,045	5,041 (0.1%)	5,043 (0.0%)	4,581 (9.2%)	4,583 (9.2%)	4,700 (6.8%)	47,00 (6.8%)
6	26,376	26,334 (0.2%)	26,344 (0.1%)	23,691 (10.2%)	23,695 (10.2%)	23,139 (12.3%)	23,138 (12.3%)
7	101,158	100,947 (0.2%)	101,104 (0.1%)	76,827 (24.1%)	76,726 (24.2%)	75,470 (25.4%)	75,482 (25.4%)
8	235,436	234,128 (0.6%)	234,758 (0.3%)	170,552 (27.6%)	170,502 (27.6%)	169,600 (28.0%)	169,739 (27.9%)
9	748,859	751,074 (-0.3%)	752,903 (-0.5%)	519,202 (30.7%)	519,533 (30.6%)	507,387 (32.2%)	508,594 (32.1%)
10	5,646,455	5,445,954 (3.6%)	5,454,479 (3.4%)	3,766,654 (33.3%)	3,756,615 (33.5%)	3,702,217 (34.4%)	3,666,676 (35.1%)

Table 5.11: Nodes to be searched using dynamic move ordering in $BRS_{1,C-1}$

Depth	$BRS_{C-1,0}$						
	HH: no; KM: 0	HH: rel.; KM: 0	HH: abs.; KM: 0	HH: rel.; KM: 1	HH: abs.; KM: 1	HH: rel.; KM: 2	HH: abs.; KM: 2
3	18,800	18,790 (0.1%)	18,790 (0.1%)	17,007 (9.5%)	17,007 (9.5%)	17,204 (8.5%)	17,204 (8.5%)
4	51,505	51,458 (0.1%)	51,458 (0.1%)	55,495 (-7.7%)	55,495 (-7.7%)	56,420 (-9.5%)	56,411 (-9.5%)
5	1,975,930	1,975,231 (0.0%)	1,975,394 (0.0%)	1,682,937 (14.8%)	1,683,031 (14.8%)	1,677,484 (15.1%)	1,676,993 (15.1%)
6	153,028,306	153,175,499 (-0.1%)	153,187,693 (-0.1%)	142,590,553 (6.8%)	142,607,682 (6.8%)	141,404,552 (7.6%)	141,854,490 (7.3%)

Table 5.12: Nodes to be searched using dynamic move ordering in $BRS_{C-1,0}$

Depth	$BRS_{C-1,1}$						
	HH: no; KM: 0	HH: rel.; KM: 0	HH: abs.; KM: 0	HH: rel.; KM: 1	HH: abs.; KM: 1	HH: rel.; KM: 2	HH: abs.; KM: 2
3	4,581	4,579 (0.1%)	4,579 (0.1%)	2,545 (44.4%)	2,545 (44.4%)	2,462 (46.3%)	2,462 (46.3%)
4	18,645	18,611 (0.2%)	18,611 (0.2%)	14,895 (20.1%)	14,895 (20.1%)	16,648 (10.7%)	16,648 (10.7%)
5	55,172	55,112 (0.1%)	55,129 (0.1%)	47,628 (13.7%)	47,627 (13.7%)	48,928 (11.3%)	48,928 (11.3%)
6	1,391,218	1,391,649 (0.0%)	1,391,171 (0.0%)	707,988 (49.1%)	707,849 (49.1%)	713,726 (48.7%)	712,748 (48.8%)
7	46,678,605	46,691,593 (0.0%)	46,706,990 (-0.1%)	39,986,494 (14.3%)	39,953,028 (14.4%)	38,453,235 (17.6%)	38,200,710 (18.2%)

Table 5.13: Nodes to be searched using dynamic move ordering in $BRS_{C-1,1}$

5.4 Average Search Depth

Table 5.14 shows the average search depths that can be achieved in 5 seconds. These statistics are collected from all experiments discussed in the following sections. On average, BRS searches up to a depth of 5.6 for three players and up to a depth of 5.1 for four players. The MAX player can plan a sequence of three or four of its own moves. In the three-player setup, $BRS_{1,C-1}$ can plan a sequence of three moves of the MAX player as it achieves a search depth of 7.6. In the four-player setup, $BRS_{1,C-1}$ can just visit two or three MAX nodes, although the average search depth is higher. $BRS_{C-1,0}$ and $BRS_{C-1,1}$ just visit two MAX nodes achieving an average search depth of 4.7 and 6.0. For max^n, the average search depth is 3.5 for three players and 3.6 for four players. In a three-player setup, max^n can sometimes plan two owns moves in a row. In a four-player setup, it usually cannot investigate two own moves in a row. Paranoid achieves a depth of 5.4 for three players and 5.1 for four players. On average, it searches two moves of the MAX player.

BRS has the largest long-term planning. The MAX player can plan up to four moves in sequence. $BRS_{1,C-1}$ is close to that performance as it visits three MAX nodes in sequence. $BRS_{C-1,0}$, $BRS_{C-1,1}$ and paranoid search two moves of the MAX player, but $BRS_{C-1,0}$ and $BRS_{C-1,1}$ usually investigate more MIN nodes. Max^n has the smallest lookahead. These observations are consistent with the theoretic analyses in Chapter 4.

Players	BRS	$BRS_{1,C-1}$	$BRS_{C-1,0}$	$BRS_{C-1,1}$	Max^n	Paranoid
3	5.6	7.6	-	-	3.5	5.4
4	5.1	8.7	4.7	6.0	3.6	5.1

Table 5.14: Average Search Depth for BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$, $BRS_{C-1,1}$, max^n and paranoid

5.5 Best-Reply Search and Variations Compared to Traditional Methods

After selecting the best evaluation function and verifying the strength of the search enhancements, the traditional search methods, max^n and paranoid are matched against BRS and its variations first. As all search algorithms reduce to $\alpha\beta$-search in a two-player game, only the three-player and four-player setups are examined. Further, as $BRS_{C-1,0}$ and $BRS_{C-1,1}$ reduce to BRS and $BRS_{1,C-1}$ in a three-player setup, there are no experiments performed for $BRS_{C-1,0}$ and $BRS_{C-1,1}$ in this setup.

5.5.1 BRS and Variations vs. Max^n or Paranoid

The first experiment matches each BRS version against max^n or paranoid in a two-algorithm setup (cf. Table 5.1). Table 5.15 shows that all BRS versions play strong against max^n and win at least 89% of the games. BRS and $BRS_{1,C-1}$ also perform well against paranoid winning about 60% to 70% of the games. Their performance increases with the number of players as paranoid becomes more pessimistic. When playing against max^n, the performance of BRS decreases with the number of players because the illegal board positions cause more errors. The performance of $BRS_{1,C-1}$ decreases as it is more likely to find an overestimated branch. A branch might be overestimated if the performed BMOM is also improving the position of the root player in a situation where the opponents have the possibility to weaken the root player. Nevertheless, both search algorithms still perform impressively. BRS and $BRS_{1,C-1}$ are equally strong. $BRS_{C-1,0}$ and $BRS_{C-1,1}$ perform a bit better than paranoid winning at least 56% of the games because they are less pessimistic than paranoid.

5.5.2 BRS and Variations vs. Max^n and Paranoid

In this experiment, BRS or one of its variations is matched against max^n and paranoid in a three-algorithm setup (cf. Table 5.16). If all three search algorithms were equally strong, a winning percentage of 33.3% would be expected. In the three-player setup, both BRS and $BRS_{1,C-1}$ win the majority of

Players	Search Algorithm	Max^n	Paranoid
3	BRS	95.4 (±2.81)	63.2 (±4.55)
	$BRS_{1,C-1}$	97.7 (±2.01)	60.4 (±4.62)
4	BRS	89.8 (±4.04)	68.9 (±4.25)
	$BRS_{1,C-1}$	89.4 (±4.12)	69.7 (±4.22)
	$BRS_{C-1,0}$	94.0 (±3.18)	57.3 (±3.81)
	$BRS_{C-1,1}$	90.3 (±3.96)	56.2 (±3.82)

Table 5.15: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against max^n or paranoid

the games. Max^n performs weak, just winning 1% of the games. BRS wins more games against max^n and paranoid as $BRS_{1,C-1}$ does. Paranoid is better in exploiting the max^n player when playing against $BRS_{1,C-1}$ than when playing against BRS. The same holds for the four-player setup. Paranoid is more pessimistic and therefore wins fewer games than in the three-player setup. However, it is still better in exploiting the max^n player than when playing against $BRS_{1,C-1}$. Furthermore, as paranoid becomes weaker, max^n wins 2% of the games. Compared to the results of Table 5.15, $BRS_{C-1,0}$ performs better against paranoid if there is an additional max^n player while $BRS_{C-1,1}$ performs worse. This indicates that $BRS_{C-1,0}$ is stronger in exploiting weaknesses of max^n than the paranoid algorithm. $BRS_{C-1,1}$ is less effective in exploiting the max^n player.

Players	BRS	Max^n	Paranoid
3	63.9 (±3.70)	1.1 (±0.80)	35.0 (±3.68)
4	72.5 (±3.11)	1.5 (±0.85)	26.0 (±3.06)
Players	$BRS_{1,C-1}$	Max^n	Paranoid
3	58.6 (±3.79)	0.9 (±0.74)	40.4 (±3.78)
4	63.2 (±3.47)	2.7 (±1.16)	34.1 (±3.41)
Players	$BRS_{C-1,0}$	Max^n	Paranoid
4	61.1 (±5.04)	2.2 (±1.52)	36.7 (±4.98)
Players	$BRS_{C-1,1}$	Max^n	Paranoid
4	52.8 (±5.16)	3.3 (±1.86)	43.9 (±5.13)

Table 5.16: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against max^n and paranoid

5.6 Best-Reply Search and Variations Compared to Each Other

In this section, BRS and its variations are directly matched against each other. First, only two algorithms compete in a three- and four-player setup. Second, two BRS-based search algorithms are matched with a max^n and a paranoid player in a four-player setup. Finally, all BRS versions play against each other in the same experiment.

5.6.1 Two BRS-Based Algorithms

Table 5.17 matches each BRS version against all other versions in a two-algorithm setup. In the three-player setup, BRS is a bit stronger than $BRS_{1,C-1}$. In the four-player setup, BRS outperforms all variations. $BRS_{1,C-1}$ is as strong as $BRS_{C-1,0}$ and is significantly better than $BRS_{C-1,1}$. $BRS_{C-1,0}$ might be a little stronger than $BRS_{C-1,1}$.

The next experiment measures the performance of BRS against $BRS_{1,C-1}$ in the four-player setup for different time settings. Table 5.18 shows the results for 2, 5 and 10 seconds thinking time per move. As the performance does not change significantly, BRS is always stronger than $BRS_{1,C-1}$ in this setup.

Players	Search Algorithm	BRS	$BRS_{1,C-1}$	$BRS_{C-1,0}$	$BRS_{C-1,1}$
3	BRS	51.6 (±4.72)	-	-	-
	$BRS_{1,C-1}$	-	48.4 (±4.72)	-	-
4	BRS	-	61.0 (±5.05)	65.7 (±6.34)	70.4 (±6.10)
	$BRS_{1,C-1}$	39.0 (±5.05)	-	49.5 (±6.68)	62.5 (±6.34)
	$BRS_{C-1,0}$	34.3 (±6.34)	50.5 (±6.68)	-	52.8 (±6.67)
	$BRS_{C-1,1}$	29.6 (±6.10)	37.5 (±6.34)	47.2 (±6.67)	-

Table 5.17: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against each other in a two-algorithm setup

Experiment	BRS	$BRS_{1,C-1}$
BRS vs. $BRS_{1,C-1}$ (2 seconds)	61.3 (±4.60)	38.7 (±4.60)
BRS vs. $BRS_{1,C-1}$ (5 seconds)	61.0 (±5.05)	39.0 (±5.05)
BRS vs. $BRS_{1,C-1}$ (10 seconds)	63.0 (±4.56)	37.0 (±4.56)

Table 5.18: Winning percentage of BRS and $BRS_{1,C-1}$ with different time settings in a four-player setup

5.6.2 Two BRS-Based Algorithms with Max^n and Paranoid

Table 5.19 shows the results of six experiments in the four-algorithm setup. In each experiment, two BRS-based search algorithms are matched against each other when a max^n player and a paranoid player are participating additionally. The experiments confirm the results of the previous experiments. BRS wins the majority of the games against all search algorithms. $BRS_{1,C-1}$ outperforms $BRS_{C-1,0}$ and $BRS_{C-1,1}$. $BRS_{C-1,0}$ is stronger than $BRS_{C-1,1}$. Furthermore, all BRS versions outperform max^n and paranoid significantly.

Experiment	BRS	$BRS_{1,C-1}$	$BRS_{C-1,0}$	$BRS_{C-1,1}$	Max^n	Paranoid
BRS, $BRS_{1,C-1}$, Max^n, Paranoid	50.0 (±5.78)	29.5 (±5.28)	-	-	3.1 (±2.01)	17.4 (±4.38)
BRS, $BRS_{C-1,0}$, Max^n, Paranoid	52.4 (±5.78)	-	26.0 (±5.08)	-	0.7 (±0.96)	20.8 (±4.70)
BRS, $BRS_{C-1,1}$, Max^n, Paranoid	54.5 (±5.76)	-	-	28.5 (±5.22)	0.7 (±0.96)	16.3 (±4.28)
$BRS_{1,C-1}$, $BRS_{C-1,0}$, Max^n, Paranoid	-	43.7 (±5.76)	32.6 (±5.42)	-	1.4 (±1.35)	22.2 (±4.81)
$BRS_{1,C-1}$, $BRS_{C-1,1}$, Max^n, Paranoid	-	39.6 (±5.66)	-	33.0 (±5.44)	2.1 (±1.65)	25.3 (±5.03)
$BRS_{C-1,0}$, $BRS_{C-1,1}$, Max^n, Paranoid	-	-	42.0 (±5.71)	31.9 (±5.39)	1.4 (±1.35)	24.7 (±4.99)

Table 5.19: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against max^n and paranoid

5.6.3 All BRS-Based Algorithms Against Each Other

In the next experiments, the performance of all BRS version is measured when there are three or four different BRS-based algorithms competing. As $BRS_{C-1,0}$ and $BRS_{C-1,1}$ reduce to BRS and $BRS_{1,C-1}$ in a three-player game, the experiments are performed in a four-player setup. Table 5.20 shows the results for the three-algorithm setup. BRS is the strongest algorithm winning about 50% of the games in all experiments. $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ are comparatively strong when there is a player participating using BRS. $BRS_{1,C-1}$ is the strongest algorithm among the proposed variations as it wins 38.7% of the games against $BRS_{C-1,0}$ and $BRS_{C-1,1}$. In this setup, $BRS_{C-1,0}$ is stronger than $BRS_{C-1,1}$.

Experiment	BRS	$BRS_{1,C-1}$	$BRS_{C-1,0}$	$BRS_{C-1,1}$
BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$	46.5 (±4.02)	28.4 (±3.63)	25.2 (±3.50)	-
BRS, $BRS_{1,C-1}$, $BRS_{C-1,1}$	50.8 (±3.49)	25.4 (±3.04)	-	23.8 (±2.97)
BRS, $BRS_{C-1,0}$, $BRS_{C-1,1}$	51.0 (±2.89)	-	23.2 (±2.44)	25.8 (±2.53)
$BRS_{1,C-1}$, $BRS_{C-1,0}$, $BRS_{C-1,1}$	-	38.7 (±2.81)	32.6 (±2.71)	28.7 (±2.61)

Table 5.20: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against each other in a three-algorithm setup

Table 5.21 shows the result of the last experiment. In this experiment, all BRS versions are matched against each other in the four-algorithm setup. BRS wins 39% of the games. $BRS_{1,C-1}$ is the second strongest search algorithm winning about 23% of the games. $BRS_{C-1,0}$ and $BRS_{C-1,1}$ are comparatively strong as each just wins about 19%. This result is expected as it is in line the findings of the previous experiments.

Experiment	BRS	$BRS_{1,C-1}$	$BRS_{C-1,0}$	$BRS_{C-1,1}$
BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$, $BRS_{C-1,1}$	39.4 (±3.82)	23.1 (±3.29)	18.4 (±3.03)	19.1 (±3.07)

Table 5.21: Winning percentage of BRS, $BRS_{1,C-1}$, $BRS_{C-1,0}$ and $BRS_{C-1,1}$ against each other in a four-algorithm setup

Chapter 6

Conclusion & Future Research

This chapter presents the final conclusions of the research. First, Section 6.1 revisits the research questions and Section 6.2 answers the problem statement. Finally, Section 6.3 gives some recommendations for future research.

6.1 Answering the Research Questions

In Section 1.3, three research questions were derived from the problem statement. In this section, the answers given in previous chapters are revisited.

1. *What is the complexity of Billabong?*

This research question has been answered in Chapter 3. The state-space as well as the game-tree complexity for the two-, three- and four-player version of Billabong was computed. It was possible to give the exact state-space complexity while the game-tree complexity was determined by self-play experiments. For two, three and four players, Billabong has state-space complexities of 10^{22}, 10^{33} or 10^{45}, respectively. The state-space complexities are respectively comparable to Checkers, Draughts and Chess. Billabong's game-tree complexities of 10^{125}, 10^{174} and 10^{228} are considerably high as the ratios of game-tree to state-space complexity are higher than for many other games. The game-tree complexities are comparable to Abalone, Havannah and Shogi, respectively.

2. *How strong is Best-Reply Search in Billabong?*

Chapter 5 has answered this research question. BRS was matched against the traditional search algorithms $\max^n$ and paranoid in a three- and a four-player experimental setup. BRS is the strongest search technique among those three algorithms. It outperforms $\max^n$ impressively with a win ratio of 89.8% for three players and 95.4% for four players. The lookahead of $\max^n$ is too small as the reached depth is about 3.5 while BRS reaches a search depth of 5.6. Further, it is also significantly stronger than paranoid, winning about 64.8% and 69.9% of the games for three and four players, respectively. In general, paranoid is weaker in the four-player game than in the three-player game as it is too pessimistic.

3. *How can Best-Reply Search be improved for a given domain?*

As Best-Reply Search has some conceptual drawbacks, two ideas to overcome these drawbacks were proposed in Chapter 4. The three proposed algorithms $\text{BRS}_{1,C-1}$, $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ were matched in Chapter 5. $\text{BRS}_{1,C-1}$ is the strongest algorithm among of the proposed algorithms. Its performance is comparable to the one of BRS when playing against $\max^n$ and paranoid. $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ are only a bit stronger than paranoid but outperform $\max^n$. When matching BRS against $\text{BRS}_{1,C-1}$, BRS outperforms $\text{BRS}_{1,C-1}$ significantly. There may be three reasons why $\text{BRS}_{1,C-1}$ does not perform that well. (1) $\text{BRS}_{1,C-1}$ has an additional overhead to avoid illegal positions and therefore cannot search as many MAX nodes in sequence as BRS given the same time frame. It depends on the game whether the concept of avoiding illegal states by applying BMOMs is worth its overhead. (2) The game of Billabong

is resistant to the negative effect illegal positions as the board usually does not change too much after a move. (3) The static move ordering in the game of Billabong is a $\max^n$ move ordering that favours moves that advances the current player as much as possible, independently to the fact whether the move might help other players as well. Therefore, it can happen that $\text{BRS}_{1,C-1}$ overestimates certain branches in the tree. Unfortunately, there is no paranoid move ordering in the game of Billabong that prefers strong moves against the root player. For instance in capturing games like multi-player Chess (Esser, 2012), the best move according to a paranoid move ordering is usually weakening the position of the root player. This reduces the probability of overestimating a branch in the search tree. The concept of ignoring only one opponent implemented in $\text{BRS}_{C-1,0}$ and $\text{BRS}_{C-1,1}$ does not pay off well, but it is better than paranoid as it is less pessimistic than paranoid.

6.2 Answering the Problem Statement

As the research questions have been revisited, an answer to the problem statement can be formulated.

How can one use search for the game of Billabong in order to improve playing performance?

During this research, BRS turned out to be the best evaluation function-based search algorithm. In spite of its weaknesses it has been able to outperform traditional search algorithms as well as the just proposed variations. The most promising variation of BRS is $\text{BRS}_{1,C-1}$ which is a comparatively strong algorithm against $\max^n$ and paranoid. Due to computational overhead it investigates fewer nodes than BRS and therefore plays weaker in a direct comparison. Furthermore as there is no paranoid move ordering in the game of Billabong, $\text{BRS}_{1,C-1}$ might overestimate branches in the search tree. Transposition tables and the killer heuristic combined with the proposed static move ordering reduce the number of nodes significantly and enable deeper search given the same time resources.

6.3 Future Research

There are two areas for potential future research. First, it is interesting whether it is possible to further improve the playing performance of search techniques in the game of Billabong. This can be done by finding a better evaluation function for $\alpha\beta$-search respecting more features and / or using better weights for the existing ones. For instance, a mobility feature counting the number of moves could improve the quality of the evaluation function. Furthermore, given a good static move ordering, Monte-Carlo Tree Search might be a strong search technique, too. Therefore, the strength of a Monte-Carlo Tree Search has to matched against BRS. Second, it is interesting to investigate whether there are more games like multi-player Chess where the computational overhead of $\text{BRS}_{1,C-1}$ is worth the effort. The search in Billabong is relatively independent from the moves of the opponents. Schadd and Winands (2011) doubt that BRS is an appropriate method in trick-based card games, like Hearts or Spades. It would be interesting to see how $\text{BRS}_{1,C-1}$ performs in these domains. Further, it is possible to modify the selection of the BMOM. For instance, $\text{BRS}_{1,C-1}$ chooses the best move according to dynamic move ordering or selects several BMOMs instead of only one such that multiple BMOM branches are investigated at a non-expanded node. Searching multiple BMOM branches reduces the probability of overestimating branches at non-expanded nodes but requires more computation time.

References

Abramson, B. (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12, No. 2, pp. 182–193. [31]

Akl, S.G. and Newborn, M.M. (1977). The Principal Continuation and the Killer Heuristic. *Proceedings of the ACM Annual Conference*, pp. 466–473. [19]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands. ISBN 9090074880. [2, 11, 15]

Beal, D. and Smith, M.C. (1994). Random Evaluations in Chess. *ICCA Journal*, Vol. 17, No. 1, pp. 3–9. [32]

Björnsson, Y. and Finnsson, H. (2009). CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, Vol. 1, No. 1, pp. 4–15. [1]

Breuker, D.M. (1998). *Memory versus Search in Games*. Ph.D. thesis, Department of Computer Science, Maastricht University, Maastricht, The Netherlands. [1, 21, 22]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, No. 4, pp. 183–193. [22]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. van den (1996). Best Reply Search for Multiplayer Games. *Replacement Schemes and Two-Level Tables*, Vol. 19, No. 3, pp. 175–180. [22]

Coulom, R. (2007). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games (CG 2006)* (eds. H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers), Vol. 4630 of *Lecture Notes in Computer Science (LNCS)*, pp. 72–83, Springer-Verlag, Heidelberg, Germany. [1, 31]

Esser, M. (2012). Best-Reply Search in Multi-Player Chess. M.Sc. thesis, Maastricht University, Maastricht, The Netherlands. [44]

Greenblatt, R.D., Eastlake, D.E., and Crocker, S.D. (1967). The Greenblatt Chess Program. *Proceedings of the AFIPS Fall Joint Computer Conference 31*, Vol. 4250, pp. 801–810. Reprinted (1988) in *Computer Chess Compendium* (ed. D.N. L. Levy), pp. 56–66. B.T. Batsford Ltd., London, United Kingdom. [20]

Hartmann, D. (1988). Butterfly Boards. *ICCA Journal*, Vol. 11, Nos. 2–3, pp. 64–71. [19, 20]

Herik, H.J. van den (1983). *Computerschaak, Schaakwereld en Kunstmatige Intelligentie*. Ph.D. thesis, Delft University of Technology, Delft, The Netherlands. In Dutch. [1]

Herik, H.J. van den, Uiterwijk, J.W.H.M., and Rijswijck, J. van (2002). Games solved: Now and in the Future. *Artificial Intelligence*, Vol. 134, Nos. 1–2, pp. 277–311. [15]

Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press. [1]

Knuth, D.E. (1973). *The Art of Computer Programming. Volume 3: Sorting and Searching*. Addison-Wesley Publishing Company, Reading, MA, USA. [21]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [1, 17, 18]

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. *Proceedings of the 17th European Conference on Machine Learning (ECML 2006)* (eds. J. Fürnkranz, T. Scheffer, and M. Spiliopoulou), Vol. 4212 of *Lecture Notes in Computer Science (LNCS)*, pp. 282–293. Springer-Verlag. [1, 31]

Kocsis, L., Uiterwijk, J.W.H.M., and Herik, H.J. van den (2001). Move Ordering Using Neural Networks. *Engineering of Intelligent Systems* (eds. L. Montosori, J. Váncza, and M. Ali), Vol. 2070 of *Lecture Notes in Artificial Intelligence*, pp. 45–50. Springer-Verlag. [19]

Korf, R.E. (1991). Multi-Player Alpha-Beta Pruning. *Artificial Intelligence*, Vol. 48, No. 1, pp. 99–111. [23]

Lorenz, U. and Tscheuschner, T. (2006). Player Modeling, Search Algorithms and Strategies in Multi Player Games. *Advances in Computer Games (ACG 2005)* (eds. H.J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H.H.L.M. Donkers), Vol. 4250 of *Lecture Notes in Computer Science (LNCS)*, pp. 210–224. Springer-Verlag. [24]

Luckhardt, C.A. and Irani, K.B. (1986). An Algorithmic Solution for N-Person Games. *Proceedings of the National Conference of Artifical Intelligence (AAAI)*, pp. 158–162. [2, 23]

Marsland, T.A. (1986). A Review of Game Tree Pruning. *ICCA Journal*, Vol. 9, No. 1, pp. 3–19. [18]

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NY, USA. [1, 18]

Nilsson, N.J. (1971). *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill Book Company, New York, NY, USA. [1]

Osborne, M.J. and Rubinstein, A. (1994). *A Course in Game Theory*. MIT Press, Cambridge, MA, USA. [23]

Schadd, M.P.D. and Winands, M.H.M. (2011). Best Reply Search for Multiplayer Games. *Transactions on Computational Intelligence and AI in Games*, Vol. 3, No. 1, pp. 57–66. [v, 2, 23, 24, 25, 27, 44]

Schaeffer, J. (1983). The History Heuristic. *ICCA Journal*, Vol. 6, No. 3, pp. 16–19. [19]

Schaeffer, J., Lake, R., Lu, P., and Bryant, M. (1996). Chinook: The Man-Machine World Checkers Champion. *AI Magazine*, Vol. 17, No. 1, pp. 21–29. [1]

Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007). Checkers is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [15]

Shannon, C.E. (1950). Programming a Computer for Playing Chess. *Philosophical Magazine*, Vol. 41, No. 7, pp. 256–257. [1]

Slate, J.D. and Atkin, L.R. (1977). *Chess Skill in Man and Machine*, Chapter CHESS 4.5: The Northwestern University Chess Program, pp. 82–118. Springer-Verlag, New York, NY, USA. [22]

Solomon, E. (1984). *Games programming*. Cambridge University Press. [2, 5, 9, 19, 32]

Sturtevant, N.R. (2003a). A Comparison of Algorithms for Multi-player Games. *Computers and Games (CG 2002)* (eds. J. Schaeffer, M. Müller, and Y. Björnsson), Vol. 2883 of *Lecture Notes in Computer Science (LNCS)*, pp. 108–122. Springer-Verlag. [23, 24]

Sturtevant, N.R. (2003b). *Multi-player Games: Algorithms and Approaches*. Ph.D. thesis, University of California, Los Angeles, CA, USA. [23]

Sturtevant, N. and Bowling, M. (2006). Robust game play against unknown opponents. *Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 713–719, ACM. [24]

Sturtevant, N.R. and Korf, R.E. (2000). On Pruning Techniques for Multi-Player Games. *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pp. 201–208. [2, 23, 24]

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

Turing, A.M. (1953). *Faster than Thought*, Chapter Digital Computers Applied to Games. Sir Isaac Pitman & Sons, London. [1]

Winands, M.H.M., Werf, E.C.D. van der, Herik, H.J. van den, and Uiterwijk, J.W.H.M. (2006). The Relative History Heuristic. *Proceedings of the 4th international conference on Computers and Games*, CG'04, pp. 262–272, Springer-Verlag, Berlin, Germany. [20]

Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical report 88, Computer Science Department, The University of Wisconsin, Madison, WI, USA. Reprinted in (1990) *ICCA Journal*, Vol 13, No. 2, pp. 68–73. [21]

Appendix A

Pseudocode for Search Algorithms

A.1 Minimax

```
1  double minimax(depth, isMaxNode) {
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (isMaxNode) {
6          maxScore = -Infinity;
7          foreach(Move aMove in generateAllMoves())
8          {
9              board.doMove(aMove);
10             double curScore = minimax(depth-1, false);
11             board.undoMove(aMove);
12
13             if (curScore > maxScore) {
14                 bestMove = aMove;
15                 maxScore = curScore;
16             }
17         }
18         return maxScore;
19     }
20     else {
21         minScore = +Infinity;
22         foreach(Move aMove in generateAllMoves())
23         {
24             board.doMove(aMove);
25             curScore = minimax(depth-1, true);
26             board.undoMove(aMove);
27
28             if (curScore < minScore) {
29                 bestMove = aMove;
30                 minScore = curScore;
31             }
32         }
33         return minScore;
34     }
35 }
```

Algorithm A.1: Pseudocode for Minimax

A.2 $\alpha\beta$-Search

```
1  double alphabeta(depth, isMaxNode, alpha, beta) {
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (isMaxNode) {
6          maxScore = -Infinity;
7          foreach(Move aMove in generateAllMoves())
8          {
9              board.doMove(aMove);
10             curScore = alphabeta(depth-1, false, alpha, beta);
11             board.undoMove(aMove);
12
13             if (curScore > maxScore) {
14                 bestMove = aMove;
15                 maxScore = curScore;
16             }
17             if (maxScore > alpha) {
18                 alpha = maxScore;
19             }
20             if (alpha >= beta) {
21                 break; // beta cutoff
22             }
23         }
24         return maxScore;
25     }
26     else {
27         minScore = +Infinity;
28         foreach(Move aMove in generateAllMoves())
29         {
30             board.doMove(aMove);
31             curScore = alphabeta(depth-1, true, alpha, beta);
32             board.undoMove(aMove);
33
34             if (curScore < minScore) {
35                 bestMove = aMove;
36                 minScore = curScore;
37             }
38             if (minScore < beta) {
39                 beta = minScore;
40             }
41             if (beta <= alpha) {
42                 break; // alpha cutoff
43             }
44         }
45         return minScore;
46     }
47 }
```

Algorithm A.2: Pseudocode for $\alpha\beta$-Search

A.3 Max^n

```
1  double[] maxn(depth) {
2      if (depth == 0 || isTerminal() ) {
3          double scoreTuple[numPlayers];
4          for (int iPlayer=0; iPlayer!=numPlayers; ++iPlayer) {
5              scoreTuple[iPlayer] = evaluate(board, iPlayer);
6          }
7          return scoreTuple;
8      }
9      else {
10         maxScoreTuple = [Infinity, Infinity, ...];
11         maxScoreTuple[curPlayerId] = -Infinity;
12
13         foreach(Move aMove in generateAllMoves())
14         {
15             board.doMove(aMove);
16             scoreTuple = maxn(depth-1);
17             board.undoMove(aMove);
18
19             if (scoreTuple[curPlayerId] > maxScoreTuple[curPlayerId])
20             {
21                 bestMove = aMove;
22                 maxScoreTuple = scoreTuple;
23             }
24             else if (scoreTuple[curPlayerId] == maxScoreTuple[curPlayerId]
25                 && scoreTuple[rootPlayerId] < maxScoreTuple[rootPlayerId])
26             {
27                 // tie breaking
28                 bestMove = aMove;
29                 maxScoreTuple = scoreTuple;
30             }
31         }
32         return maxScoreTuple;
33     }
34 }
```

Algorithm A.3: Pseudocode for Max^n

A.4 Paranoid

```
1  double paranoid(depth, nextMaxNodeCount, alpha, beta){
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (nextMaxNodeCount == 0)
6      {
7          // max node
8          maxScore = -Infinity;
9          foreach(Move aMove in generateAllMoves())
10         {
11             board.doMove(aMove);
12             curScore = paranoid(depth-1, numPlayers-1, alpha, beta);
13             board.undoMove(aMove);
14
15             if (curScore > maxScore) {
16                 bestMove = aMove;
17                 maxScore = curScore;
18             }
19             if (maxScore > alpha) {
20                 alpha = maxScore;
21             }
22             if (alpha >= beta) {
23                 break; // beta cutoff
24             }
25         }
26         return maxScore;
27     }
28     else {
29         // min node
30         minScore = +Infinity;
31         foreach(Move aMove in generateAllMoves())
32         {
33             board.doMove(aMove);
34             curScore = paranoid(depth-1, nextMaxNodeCount-1, alpha, beta);
35             board.undoMove(aMove);
36
37             if (curScore < minScore) {
38                 bestMove = aMove;
39                 minScore = curScore;
40             }
41             if (minScore < beta) {
42                 beta = minScore;
43             }
44             if (beta <= alpha) {
45                 break; // alpha cutoff
46             }
47         }
48         return minScore;
49     }
50 }
```

Algorithm A.4: Pseudocode for Paranoid

A.5 Best-Reply Search

```
1  double BRS(depth, isMaxNode, alpha, beta){
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (isMaxNode)
6      {
7          maxScore = -Infinity;
8          foreach(Move aMove in generateAllMoves())
9          {
10             board.doMove(aMove);
11             curScore = BRS(depth-1, false, alpha, beta);
12             board.undoMove(aMove);
13
14             if (curScore > maxScore) {
15                 bestMove = aMove;
16                 maxScore = curScore;
17             }
18             if (maxScore > alpha) {
19                 alpha = maxScore;
20             }
21             if (alpha >= beta) {
22                 break; // beta cutoff
23             }
24         }
25         return maxScore;
26     }
27     else {
28         foreach(Opponent anOpponent) {
29             moveList += generateAllMoves()
30         }
31
32         minScore = +Infinity;
33         foreach(Move aMove in moveList)
34         {
35             board.doMove(aMove);
36             curScore = BRS(depth-1, true, alpha, beta);
37             board.undoMove(aMove);
38
39             if (curScore < minScore) {
40                 bestMove = aMove;
41                 minScore = curScore;
42             }
43             if (minScore < beta) {
44                 beta = minScore;
45             }
46             if (beta <= alpha) {
47                 break; // alpha cutoff
48             }
49         }
50         return minScore;
51     }
52 }
```

Algorithm A.5: Pseudocode for Best-Reply Search

A.6 $\mathbf{BRS}_{1,C-1}$

```
1  double BRS_1_C1(depth, nextMaxNodeCount, onBestMoveOrderingPath, alpha, beta) {
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (nextMaxNodeCount == 0) {
6          // max node
7          maxScore = -Infinity;
8          foreach(Move aMove in generateAllMoves())
9          {
10             board.doMove(aMove);
11             curScore = BRS_1_C1(depth-1, numPlayers-1, true, alpha, beta);
12             board.undoMove(aMove);
13
14             if (curScore > maxScore) {
15                 bestMove = aMove;
16                 maxScore = curScore;
17             }
18             if (maxScore > alpha) {
19                 alpha = maxScore;
20             }
21             if (alpha >= beta) {
22                 break; // beta cutoff
23             }
24         }
25         return maxScore;
26     }
27     else if ( onBestMoveOrderingPath ) {
28         // standard min node
29         minScore = +Infinity;
30         foreach(Move aMove in generateAllMoves())
31         {
32             board.doMove(aMove);
33             curScore = BRS_1_C1(depth-1, nextMaxNodeCount-1,
34                 aMove == bestMoveOrderingMove, alpha, beta);
35             board.undoMove(aMove);
36
37             if (curScore < minScore) {
38                 bestMove = aMove;
39                 minScore = curScore;
40             }
41             if (minScore < beta) {
42                 beta = minScore;
43             }
44             if (beta <= alpha) {
45                 break; // alpha cutoff
46             }
47         }
48         return minScore;
49     }
50     else {
51         // min node, but only best move ordering move to check
52         board.doMove(bestMoveOrderingMove);
53         curScore = BRS_1_C1(depth-1, numPlayers-1, false, alpha, beta);
54         board.undoMove(bestMoveOrderingMove);
55     }
56 }
```

Algorithm A.6: Pseudocode for $\mathrm{BRS}_{1,C-1}$

A.7 $\mathbf{BRS}_{C-1,0}$

```
1  double BRS_C1_0(depth, nextMaxNodeCount, skippedPlayer, alpha, beta) {
2      if (depth == 0 || isTerminal() ) {
3          return evaluate(board);
4      }
5      else if (nextMaxNodeCount == 0) {
6          // max node
7          maxScore = -Infinity;
8          foreach(Move aMove in generateAllMoves()) {
9              board.doMove(aMove);
10             curScore = BRS_C1_0(depth-1, numPlayers-2, true, alpha, beta);
11             board.undoMove(aMove);
12             if (curScore > maxScore) {
13                 bestMove = aMove;
14                 maxScore = curScore;
15             }
16             if (maxScore > alpha) {
17                 alpha = maxScore;
18             }
19             if (alpha >= beta) {
20                 break; // beta cutoff
21             }
22         }
23         return maxScore;
24     }
25     else {
26         // min node: initialise sub tree, where current player is skipped
27         minScore = +Infinity;
28         if (!skippedAPlayerAlready) {
29             float curScore = BRS_C1_0(depth, nextMaxNodeCount, true, alpha, beta);
30             if (curScore < minScore) {
31                 bestMove = aMove;
32                 minScore = curScore;
33             }
34             if (minScore < beta) {
35                 beta = minScore;
36             }
37             if (beta <= alpha) {
38                 break; // alpha cutoff
39             }
40         }
41         // min node: don't skip player
42         if (!cutoff) {
43             foreach(Move aMove in generateAllMoves()) {
44                 board.doMove(aMove);
45                 curScore = BRS_C1_0(depth-1, nextMaxNodeCount-1,
46                     skippedPlayer, alpha, beta);
47                 board.undoMove(aMove);
48                 if (curScore < minScore) {
49                     bestMove = aMove;
50                     minScore = curScore;
51                 }
52                 if (minScore < beta) {
53                     beta = minScore;
54                 }
55                 if (beta <= alpha) {
56                     break; // alpha cutoff
57                 }
58             }
59         }
60         return minScore;
61     }
62 }
```

Algorithm A.7: Pseudocode for $\mathrm{BRS}_{C-1,0}$

```
double BRS_C1_1(depth, nextMaxNodeCount, appliedBMOMAlready, alpha, beta) {
    if (depth == 0 || isTerminal() ) {
        return evaluate(board);
    }
    else if (nextMaxNodeCount == 0) {
        // max node
        maxScore = -Infinity;
        foreach(Move aMove in generateAllMoves())
        {
            board.doMove(aMove);
            curScore = BRS_C1_1(depth-1, numPlayers-1, false, alpha, beta);
            board.undoMove(aMove);

            if (curScore > maxScore) {
                bestMove = aMove;
                maxScore = curScore;
            }
            if (maxScore > alpha) {
                alpha = maxScore;
            }
            if (alpha >= beta) {
                break; // beta cutoff
            }
        }
        return maxScore;
    }
    else if ( (curDepth + 1) % numplayers != 0 || appliedBMOMAlready ) {
        // standard min node
        minScore = +Infinity;
        foreach(Move aMove in generateAllMoves())
        {
            board.doMove(aMove);
            curScore = BRS_C1_1(depth-1, nextMaxNodeCount-1,
                appliedBMOMAlready | (aMove == bestMoveOrderingMove),
                alpha, beta);
            board.undoMove(aMove);

            if (curScore < minScore) {
                bestMove = aMove;
                minScore = curScore;
            }
            if (minScore < beta) {
                beta = minScore;
            }
            if (beta <= alpha) {
                break; // alpha cutoff
            }
        }
        return minScore;
    }
    else {
        // min node, but only best move ordering move to check
        board.doMove(bestMoveOrderingMove);
        curScore = BRS_C1_1(depth-1, numPlayers-1, false, alpha, beta, true);
        board.undoMove(bestMoveOrderingMove);
    }
}
```

Algorithm A.8: Pseudocode for $\text{BRS}_{C-1,1}$

Appendix B

Mathematical proves

B.1 Complexity of $\mathbf{BRS}_{C-1,1}$

Proposition: For all $n \geq 3$ the following is true

$$b^{n-1} - (b-1)^{n-1} \leq nb^{n-2}; b \geq 1 \tag{B.1}$$

Base case: For $n = 3$ the following statement holds.

$$\begin{aligned} b^2 - (b-1)^2 &\leq 3b \\ b^2 - b^2 + 2b - 1 &\leq 3b \\ 2b - 1 &\leq 3b \\ b &\geq -1 \end{aligned}$$

Induction step: Assume that the statement holds for $n = k$

$$b^{k-1} - (b-1)^{k-1} \leq kb^{k-2} \tag{B.2}$$

such that it is also true for $n = k + 1$

$$b^k - (b-1)^k \leq (k+1)b^{k-1}$$

$$\begin{aligned} b^k - (b-1)^k &= b \times b^{k-1} - (b-1) \times (b-1)^{k-1} \\ &= b \times b^{k-1} - b \times (b-1)^{k-1} + (b-1)^{k-1} \\ &= b \times (b^{k-1} - (b-1)^{k-1}) + (b-1)^{k-1} \\ &\overset{B.2}{\leq} b \times (kb^{k-2}) + (b-1)^{k-1} \\ &= kb^{k-1} + (b-1)^{k-1} \\ &\leq kb^{k-1} + b^{k-1} \\ &= (k+1)b^{k-1} \quad \square \end{aligned}$$