COMPETITIVE PLAY IN STRATEGO

A.F.C. Arts

Master Thesis DKE 10-05

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
M.P.D. Schadd, M.Sc
D.L. St-Pierre, M.Sc

Maastricht University
Department of Knowledge Engineering
Maastricht, The Netherlands
March 2010

In loving memory of my father

Preface

This master thesis was written at the Department of Knowledge Engineering at Maastricht University. The subject is the board game Stratego. In this thesis we investigate the relative unknown algorithms called Star1, Star2, StarETC.

I would like to thank the following people for their help and support. First of all, my supervisors Dr. Mark Winands and Maarten Schadd, MSc. for their guidance, insight and support. They have helped me with ideas, problems and writing this thesis. Second, the people at *Airglow Studios* for providing the idea of a thesis about Stratego. And I wish to thank the other thesis committee members for reading this thesis.

Finally, I would like to thank my family and friends for their support.

Sander Arts
Maastricht, March 2010

Summary

The focus of the thesis is the analysis and implementation of an artifical player for the game Stratego. Stratego is a deterministic imperfect-information board game for two players. The problem statement of this thesis is: *How can we develop informed-search methods in such a way that programs significantly improve their performance in Stratego?* To answer this problem statement the state-space and game-tree complexity is calculated. For the development of the program, the following backward pruning techniques are implemented: α-β pruning, iterative deepening, the history heuristic, transposition tables and the *-MINIMAX algorithms such as STAR1, STAR2 and STARETC. A forward pruning technique called multi-cut is tested. Additionally the evaluation function uses features as described by De Boer (2007).

It is shown that Stratego is a complex game when compared to other games, such as chess and Hex. The game-tree complexity of 10^{535} exceeds the game-tree complexity of Go. The state-space equals 10^{115}. The use of STAR1, STAR2 and STARETC improve the reachable search depth compared to the EXPECTIMAX algorithm that is normally used in games with non-determinism. EXPECTIMAX with the HISTORY HEURISTIC, α-β and TRANSPOSITION TABLES is able to prune 74.06%. The results show that STAR1 and STAR2 can improve the node reduction by 75.18% compared to EXPECTIMAX with α-β, HISTORY HEURISTIC and TRANSPOSITION TABLES. The *-MINIMAX algorithms get their advantage from pruning in chance nodes. The performance gain gives the artificial player a possibility to search deeper than an artificial player without a *-MINIMAX algorithm. The MULTI-CUT algorithm does not perform significant better. The best performance of multi-cut is a score percentage of 53.3% in self play. In the worst case the score percentage is 46.69%.

The conclusion of the thesis is that informed-search techniques improve the game playing of an artificial player for Stratego. However, because of the high complexity of the game, a intermediate or expert level of Stratego is hard to realize.

Samenvatting

Het doel van deze thesis is de analyse en implementatie van een computer speler voor het spel Stratego. Stratego is een deterministisch onvolledig-informatie bordspel voor 2 spelers. De hoofdvraag van deze thesis is: *Hoe kunnen we geïnformeerde zoektechnieken ontwikkelen zodat de prestatie van een programma significant verbeterd in Stratego?* Om deze vraag te beantwoorden worden de toestandsruimte-complexiteit en de zoekboom-complexiteit berekend. Voor de ontwikkeling van het programma zijn meerdere *backward pruning* technieken gebruikt: α-β snoeïng, ITERATIVE DEEPENING, de HISTORY HEURISTIEK, TRANSPOSITIE TABELLEN en de *-minimax algoritmen zoals STAR1, STAR2 en STARETC. Er is een *forward pruning* techniek gebruikt, namelijk MULTI-CUT. Ook zijn er nog evaluatie functie eigenschappen getest zoals beschreven door De Boer (2007).

Het blijkt dat Stratego een complex spel in vergelijking met andere spellen, zoals schaak en Hex. De zoekboom-complexiteit van 10^{535} is groter dan die van Go. De toestandsruimte-complexiteit is 10^{115}. Het gebruik van STAR1, STAR2 en STARETC verbeteren de zoekdiepte vergeleken met het expectimax-algoritme dat normaal gebruikt wordt in spellen met onvolledige informatie of kansknopen. EXPECTIMAX met de HISTORY HEURISTIEK, α-β en TRANSPOSITIE TABELLEN kan tot 74.06% van de knopen snoeien. De resultaten laten zien dat STAR1 met STAR2 de reductie kan verbeteren met 74.18% in vergelijking met EXPECTIMAX met α-β snoeïng, HISTORY HEURISTIEK en TRANSPOSITIE TABELLEN. De *-MINIMAX algoritmen halen hun voordeel bij het snoeien van kansknopen. De prestatiewinst geeft de computer speler de kans dieper te zoeken dan een computer speler zonder *-MINIMAX. Het programma met MULTI-CUT presteert niet significant beter. Het beste resultaat, in spellen tegen zichzelf, is een scorings-percentage van 53.3%. In het slechtste geval is het scorings-percentage 46.69%.

De conclusie van deze thesis is dat geïnformeerde zoektechnieken het spel van een computer speler voor Stratego verbeteren. Hoewel, door de hoge complexiteit van het spel is het lastig om een speler van een gemiddeld of hoog niveau voor Stratego te maken.

Algorithms

List of Algorithms

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This chapter introduces the subject of the thesis, Stratego. A historical overview of the game is given in Section 1.2. Section 1.3 discusses the related work done in this area. The research questions of this thesis are presented in Section 1.4. Finally, 1.5 gives an outline of this thesis.

1.1 Games and AI

Games in different forms have fascinated people all over the world since civilization emerged. Board games come in all different kind of forms. For example, if chance is involved or how much information the player has. Games make it possible for people to challenge their intellect against other humans.

With increasing computational power humans attempted to let computers play games. Chess was one of the first games that received attention from science. The first persons to describe a possible chess program were Shannon (1950) and Turing (1953). Research on abstract games such as chess has led to the chess engine Deep Blue, which defeated the World Champion of chess Kasparov in 1997, (Hsu, 2002). Computers improved their play in several games since the first chess program (Van den Herik, Uiterwijk, and Van Rijswijck, 2002) appeared. Some games have already been completely solved, such as *Connect-Four*, (Allis, 1988), *Checkers* (Schaeffer, 2007) and *Fanorona* (Schadd *et al.*, 2008). Moreover, researchers investigated also less deterministic games. Stochastic (non-deterministic) games, such as Backgammon that included an element of chance and Poker, a game which includes chance and imperfect information.

1.2 Stratego[1]

The game of Stratego is a board game for two players. The players each have 40 pieces with military ranks that remain hidden for the opponent. The pieces can move around the board and attack opponent's pieces. The player whose Flag is captured loses the game. The rank of a piece is only revealed when attacked or attacking.

Stratego was created by Mogendorff during World War II. It registered as a trademark in 1942 by the Dutch company Van Perlestein & Roeper Bosch NV. In 1958 the license was granted to Hausemann an Hötte. The first version of Stratego was distributed by Smeets and Schippers in 1946 (United States Court, 2005). In 1961 the game was sublicensed to Milton Bradley, which was acquired by Hasbro in 1984, and first published in 1961 in the United States.

The game is popular in the Netherlands, Belgium, Germany, Britain, the United States of America and Ukraine. Since 1997, there is a world championship of Stratego every year. In 2007, 44 people participated in this championship and since 2007 the USA Stratego Federation holds the Computer Stratego World Championship (USA Stratego Federation, 2009).

[1]STRATEGO is a registered trademark of Hausemann & Hötte.

1.3 Related Work

Some research has been done for the game of Stratego. There are several publications which describe their research about this topic. The first publication is by Stengård (2006) who has done research in a minimax-based approach of Stratego and created an algorithm called *P-E-minimax*. This algorithm implements a null-move like technique to calculate if a move results in a similar position. The evaluation function used is unknown.

The second publication about Stratego is by De Boer (2007), former (human) World Champion Stratego. De Boer focuses on Stratego board set-ups and uses plans to play the game instead of search techniques. Like most of the current Stratego bots, this implementation does not use minimax-based algorithms but is based upon a more human-like approach (USA Stratego Federation, 2009). By selecting a strategy (attacking a piece, defending a piece, etc) the algorithm creates short-term plans to realise its goals. However, this implementation heavily depends on the evaluation function.

A multi-agent implementation has been done by Treijtel (2000) & Ismail (2004). These publications approach Stratego using a rule based agent system to which assigns scores to the generated moves of the pieces. The main focus of these publication lies on the implementation of the agents and their behaviour.

Furthermore there is a B.Sc thesis about opponent modelling in Stratego by Stankiewicz (2009) and Stankiewicz and Schadd (2009). This thesis researches the modelling of the opponent by examening the moves of the opponent and tries to discover the rank of the pieces by using these observations.

Another B.Sc. thesis is by Mohnen (2009) which implements domain-dependent knowledge of Stratego in the evaluation function. The evaluation function is used to see if better domain knowledge results in a higher number of α-β cutoffs.

The technique ChanceProbCut is applied in Stratego by Schadd, Winands, and Uiterwijk (2009). This papers describes a forward-pruning technique in expectimax which reduces the size of the search tree significantly.

The final paper on Stratego is by Schadd and Winands (2009) where an adapted version of Quiesence Search is applied

A game similar to Stratego is the game *Si Guo Da Zhan (The Great Battle of Four Countries)* (Xia, Zhu, and Lu, 2007). This game is intended for four players. Each player has an army with pieces of different ranks, similar as in Stratego.

Kriegspiel (Ciancarini and Favini, 2007) is a chess game where the location of the opponent's pieces is unknown. The standard chess set-up stays unchanged. However, the player will need to guess or estimate the location of the opponent's pieces.

1.4 Problem Statement and Research Questions

In this thesis we investigate the relevance of informed search to improve computer game-play. The following problem statement guides the research:

Problem Statement. *How can we develop informed-search methods in such a way that programs significantly improve their performance in Stratego?*

To understand the complexity of Stratego we can use the state-space and the game-tree complexity as measure and define Stratego in the game-space (Allis, 1994). The position of Stratego in the game-space indicates if the solvability of the game and informs us if we can use a knowledge-based or search-based approach (Van den Herik *et al.*, 2002).

Research Question 1. *What is the complexity of Stratego?*

Several informed search techniques exist in order to improve the game play. The most famous one is the minimax-based α-β search technique. Unfortunately, α-β is only applicable for deterministic perfect-information games. A derivative of minimax has been developed to address stochastic games called expectimax (Michie, 1966). It has been successfully been applied in the stochastic game of Backgammon (Hauk, Buro, and Schaeffer, 2004). A direct successor of expectimax is the group of algorithms called *-minimax (Ballard, 1983), which is able to prune substantial parts of the search tree.

The characteristics of Stratego indicate that *-MINIMAX might be a successful approach for Stratego. This leads to our second research question:

Research Question 2. *To what extent can *-minimax techniques be utilized in order to realize competitive game play in Stratego?*

1.5 Outline of the thesis

- Chapter 1 explains the subject of the thesis and presents the research questions.
- Chapter 2 introduces the basics of Stratego, such as the pieces, the game board, the rules and strategies for the game.
- Chapter 3 gives an insight in the complexity of Stratego and answers the first research question. This chapter also compares the complexity of Stratego with other games.
- Chapter 4 explains the search techniques and enhancements used in the implementation of Stratego.
- Chapter 5 gives an insight into the different aspects of the evaluation function, such as piece value, information value and strategic positions.
- Chapter 6 lists the tests and their results. These results will answer the second research question.
- Chapter 7 concludes the thesis and answers the research questions.

Chapter 2

The Game of Stratego

In Section 2.1 the rules of Stratego will be discussed. In Section 2.2 the notation of Stratego will be introduced. It will also look at strategies and knowledge in Stratego in Section 2.3.

2.1 Rules

The rules of the game are described by Hasbro (2002). The goal of the game is to capture the opponent's flag. The detailed rules are given below.

Start of the Game

Stratego is played on a 10×10 board with two lakes of size 2×2 in the middle of the board. The players place their 40 pieces with the back of the piece toward the opponent in a 4×10 area. One player takes the blue pieces, called the blue player, and the other player takes the red pieces, called the red player. The movable pieces are divided in ranks (from the lowest to the highest): Spy, Scout, Miner, Sergeant, Lieutenant, Captain, Colonel, Major, General, Marshal. The player also has two unmovable piece types, the Flag and the Bomb. Table 2.1 shows the number of pieces for each player. After the pieces of both players are placed on the board, the red player makes the first move. A player can either choose to move a piece or attack an opponent's piece.

1 Flag	6 Bombs
1 Spy	8 Scouts
5 Miners	4 Sergeants
4 Lieutenants	4 Captains
3 Colonels	2 Majors
1 General	1 Marshal

Table 2.1: Available Stratego pieces for each player

Movement

- Alternately the Red player and the Blue player moves.
- Pieces are moved one square at the time to orthogonally-adjacent vacant squares. The Scout is an exception to this rule, as explained below.
- The lakes in the center of the board contain no squares, therefore a piece can neither move into nor cross the lakes.
- Only one piece can occupy a square.
- Each turn only one piece may be moved.

- Scouts can be moved across any number of vacant squares either forward, backward or sideways.
- The player must move or attack in his turn.

Attack

- A player can attack if a player's piece is orthogonally adjacent to a square occupied by an opponent's piece.
- A Scout can attack from any distance in a straight line provided that the squares between itself and the attacked piece are vacant.
- It is not required for a player to attack an adjacent opponent's piece.
- The piece with the lowest rank loses the attack and will be removed from the board. If the attacking piece wins, it will be moved on to the square it attacked. If the attacking piece loses no piece will be moved.
- If a piece attacks an opponent's piece of the same rank, both pieces are removed from the board.
- The Spy defeats the Marshal if it attacks the Marshal. Otherwise the Spy will lose the attack.
- Every movable piece, except the Miner, that attacks the Bomb will lose the attack and will be removed from the board. When a Miner attacks a Bomb, the Bomb will lose and will be removed from the board. The Miner now moves to the square which the Bomb occupied.
- The Bomb and Flag cannot attack.

Two-squares rule

To avoid repetition a player's piece may not move back and forth on the same two-squares in a row for more than five times. This is called the two-squares rule. If the player moves a new piece the two-squares rule is interrupted and starts again for the new piece. The actions of the opponent do not influence the two-squares rule for the player. If the player moves to a square other than the previous square then the two-squares rule is interrupted and the two-squares rule applies to the current and previous square.

In case of the Scout the rules are extended. The first of the five moves set the range where the two-squares rule will apply. Where a normal piece is only allowed to move five times between two-squares, the Scout is only allowed to move five times within the range of his first move.

Chasing rule

An additional tournament rule defined by the *International Stratego Federation* (2009) is the chasing rule. This rule prevents the players to end up in a continuous chase of two pieces by non-stop threatening and evading. The threatening piece may not play a move such that the game ends up in a state that has been visited already during the chase. The only exception occurs when a threatening piece moves back to the previous square as long is does not violate the two-squares rule.

End of the game

The game ends when the Flag of one of the players is captured. The player whose Flag is captured loses the game. If one player is not able to move or attack, this player forfeits the game. If both players cannot move or attack, the game is a draw.

2.2 Notation

There is no official notation in the game of Stratego. Several notations are available but rely on perfect information of the initial piece set-up of both players. We will use a chess-like notation for Stratego.

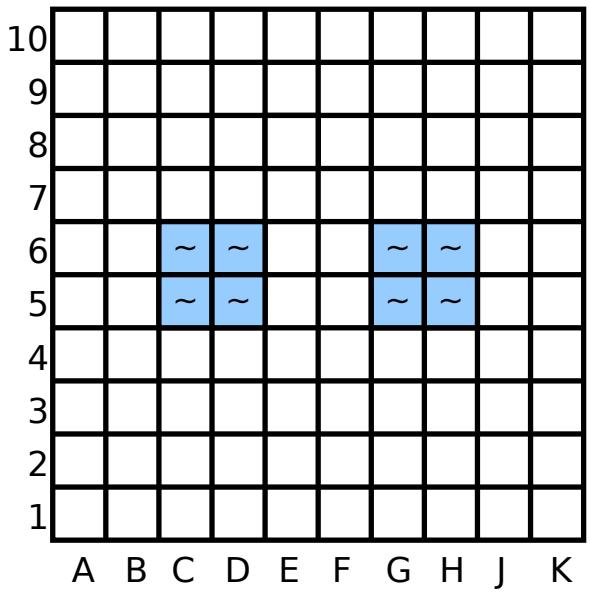


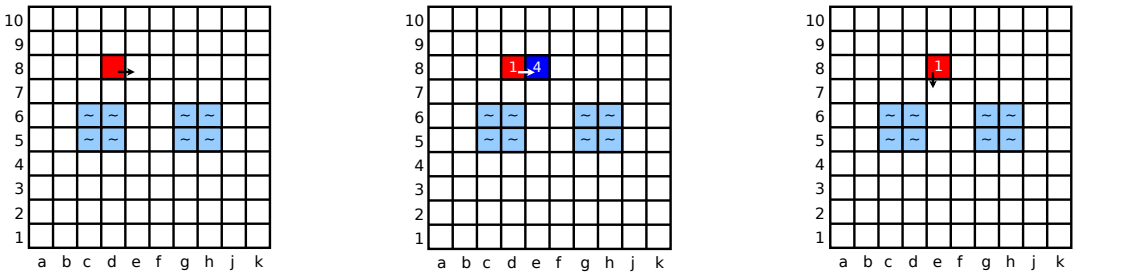
Figure 2.1: Stratego board with coordinates

We first define the game board in a chess form, see Figure 2.1, where the ten files are labeled A to K. To prevent confusion between the letters I and J, the letter I has been skipped in the notation. The ten ranks are numbered starting at 1. The last rank is labeled 0 instead of 10.

A move will be indicated with the origin and the destination of the piece separated by a hyphen. E.g. a move of a piece from square **d8** to square **e8** is notated as: **d8-e8**, see Figure 2.2a.

Attack moves will be denoted with "x" instead of a hyphen between the moves. An additional feature in Stratego would be the revealing of the pieces. The ranks will be notated in a numerical form for convenience from high to low starting with the number 1 for the Marshal. Only the Spy will be written down as "S", the Bomb as "B" and the Flag as "F". The rank of the piece will be placed after the square and a "#". An attack of the Marshal at square **d8** towards a piece at square **e8**, which is occupied by a Colonel will be written down in the following notation: **d8#1xe8#4**, see Figure 2.2b.

Moves of pieces from which the rank is known will indicate the rank in a similar form, e.g., a Marshal moving from **e8** to **e7** will be notated as **e8#1-e8**, see Figure 2.2c.



(a) Normal move: **d8-e8**

(b) Attack move: **d8#1xe8#8**

(c) Normal move with known piece: **e8#1-e7**

Figure 2.2: Notation of moves in Stratego

2.3 Strategies and Knowledge

This section will give an overview of human strategies and knowledge in the game of Stratego. The setup of the pieces and the play of the pieces have a great influence on the game.

2.3.1 Flag Position

The goal of the game is to capture the opponent's Flag. Therefore it is important to guard the Flag against the opponent's pieces. The main strategy used is to place the Flag adjacent to the edge or in the corners. In this way the Flag can only be reached by two or three directions. When a player places Bombs around the Flag the opponent can only reach the Flag by using a Miner.

A player could create a diversion by placing a similar structure guarding a non-important piece along the backline. This requires the player to play more moves to find the opponent's Flag.

2.3.2 Miner

When the opponent's Flag is surrounded by Bombs the player must make sure it is still able to capture the Flag by removing the Bombs using his Miners. To prevent a loss on forehand some of the player's Miners have to remain on the board until the possible location of the Flag has been discovered.

2.3.3 The Spy

The only movable piece that the Spy is able to defeat is the Marshal and the Marshal is the only movable piece that can defeat the General. If the General is defeated by the Marshal, the Marshal is known and the Spy can try to defeat the Marshal. To defeat the Marshal as quick as possible the Spy would need to be close to the General, preferable an adjacent square to attack immediately after the defeat of the General. However, if the General would travel around the board with an adjacent unknown piece, this piece is most likely the Spy.

2.4 Other Versions

There are a few variations on Stratego. There is a Stratego game made for 4 players, where each player has 21 pieces and all 4 players play on a 9×9 board, excluding the setup squares. A similar variant exists for 3 and 6 players.

Hertog Jan Stratego, specially developed for the beer brewery *Hertog Jan*, has 12 pieces on an 10×10 board with similar rules. But it has 12 custom pieces, such as an Archer, a piece that can attack over a small distance and the Duke, which represents the Flag but can move.

A 3D Stratego, called *Stratego Fortress* which includes multiple levels of board and trap doors and is set in medieval atmosphere. Players can build a custom fortress where they need to hide a treasure instead of the Flag.

These days Stratego comes in different forms, besides the original Stratego, there have been themed variations, such as a *Star Wars*-theme, *Lord of the Rings*-theme and many more.

Chapter 3

Complexity Analysis

In this chapter we investigate different complexity measures for Stratego and compare the results to other games. In Section 3.1 we look at the complexity of choosing a board setup, in Section 3.2 we will analyze the state-space complexity of the game and in Section 3.3 we will analyze the game tree complexity, (Allis, 1994).

3.1 Board-setup Complexity

A player is able to place his own 40 pieces on the board. The player places his pieces in a 4×10 area. We can calculate the number of possible board setup configurations in Stratego by calculating all the combination to place the 40 pieces:

$$\binom{40}{1,1,2,3,4,4,4,5,8,1,6,1} = \frac{40!}{(1!)^4 2!3!(4!)^3 5!8!6!} \tag{3.1}$$

By excluding mirrored setups the number of different board setups equals $5^{33} \approx 10^{23}$. Compared to the complexity of other games as in Table 3.1 we can see that the initial possible game states of Stratego exceeds the complexity of games such as checkers.

3.2 State-space Complexity

The state-space complexity is the collection of all reachable board configurations a game can have. This number is difficult to compute exactly, therefore we compute an upper bound by counting all the legal positions in Stratego, this also includes unreachable positions. There is a difference between the concept of an illegal configuration and an unreachable position. An illegal configuration in Stratego would be a board where both flags are missing or a board where the two flags are placed next to each other. A legal but an unreachable position is a board configuration where all the lanes are blocked by bombs but a piece has passed the bombs.

As a result of variable empty spaces and removed pieces we can use a similar but more complex formula than Formula 3.1 for the computation of the state-space complexity. To calculate this upper bound of the state-space complexity we need to summate over 24 variables. Every variable is one of the twelve either blue or red pieces. Every variable can vary between zero and the maximum amount of pieces of that type in the game. For every possible combination we need to calculate a formula similar as Formula 3.1:

$$\binom{92}{\#\ \textit{of free positions}, \#\ \textit{red flags}, \#\ \textit{red bombs}, \ldots, \#\ \textit{blue flags}, \ldots, \#\ \textit{blue marshals}} \tag{3.2}$$

However because we are dealing with variables in this formula we need to summate over the different variables as indicated by Formula 3.3. This Formula takes into account that Bombs may only be placed within the 4×10 start area and the fact that both players need their Flag to be on the game board. The computation was done using hashed values to decrease the computation time.

$$40 \times 40 \times \left(\sum_{\substack{red\\bombs}}^{y} \sum_{\substack{blue\\bombs}}^{z} \frac{39!}{(39-y)! \times y!} \times \frac{39!}{(39-z)! \times z!} \right) \times \sum_{\substack{red\\spies}}^{i} \sum_{\substack{red\\scouts}}^{j} \cdots \sum_{\substack{blue\\generals}}^{w} \sum_{\substack{blue\\marshals}}^{x} \tag{3.3}$$
$$\frac{(90-y-z)!}{(90-y-z-i)! \times i!} \times \frac{(90-y-z-i)!}{(90-y-z-i-j)! \times j!} \times \cdots \times \frac{(90-y-z-i-j-\cdots-w)!}{(90-y-z-i-j-\cdots-w-x)! \times x!}$$

The number of free squares depends on the number of pieces on the board. There are 92 (where 2 of them are always occupied by both the Flags) available squares minus the number of pieces on board. The upper bound of the state-space of Stratego is 10^{115}.

3.3 Game-tree complexity

Based on approximately 4,500 games from the Gravon Stratego (2009) database which did not result in a forfeit, the average number of moves in a Stratego game is calculated. This average is 381.134 ply, a ply is a turn for one player.

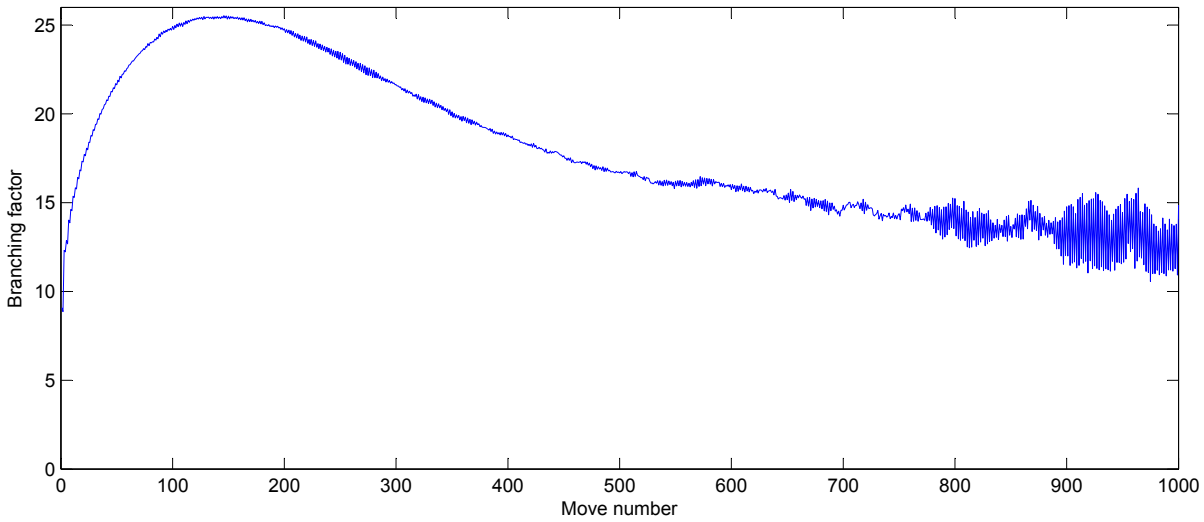


Figure 3.1: Average branching factor of Stratego per move number

The branching factor in the beginning of a Stratego game is approximate 23 and slowly decreases to 15 towards the end of the game. Figure 3.1 shows the average branching factor per move. These values are based upon the same games from Gravon Stratego. At around 150 moves the branching factor reaches its peak with an average of 25. Then the branching factor almost linearly decreases towards 15. After 700 moves the branching factor increases again and fluctuates. Because there are fewer games that last longer than 700 moves the significance of these branching factors decreases.

The average branching factor of Stratego is approximately 21.739. And the game has an average of 30.363 chance nodes where each chance node has an average branching factor of 6.823. The game-tree upper bound is the branching factor to the power of the average game length. In other words $21.739^{381.134} \times 6.823^{30.363} \approx 10^{535}$. The game-tree complexity is independent of the starting position.

3.4 Stratego Compared to Other Games

In Section 3.2 and 3.3 we computed the state-space complexity (10^{115}) for Stratego and the game-tree complexity (10^{535}). In Table 3.1 Stratego is compared to other games (Van den Herik *et al.*, 2002) and it is clear that Stratego is a complex game. The game-tree complexity is large because of the high game

Game	State-space	Game-tree
Nine Men's Morris	10^{10}	10^{50}
Pentominoes	10^{12}	10^{18}
Awari	10^{12}	10^{32}
Kalah(6,4)	10^{13}	10^{18}
Connect-Four	10^{14}	10^{21}
Domineering (8×8)	10^{15}	10^{27}
Dakon-6	10^{15}	10^{33}
Checkers	10^{21}	10^{31}
Othello	10^{28}	10^{58}
Qubic	10^{30}	10^{34}
Draughts	10^{30}	10^{54}
Chess	10^{46}	10^{123}
Chinese Chess	10^{48}	10^{150}
Hex (11×11)	10^{57}	10^{98}
Shogi	10^{71}	10^{226}
Renju (15×15)	10^{105}	10^{70}
Go-Moku (15×15)	10^{105}	10^{70}
Stratego	10^{115}	10^{535}
Go (19×19)	10^{172}	10^{360}

Table 3.1: Complexity of several games

length (381 ply). The high game-tree complexity requires us to use more search based approaches to play the game of Stratego.

Chapter 4

Search

This chapter introduces the search techniques used to find a best possible move in Stratego. Section 4.1 till Section 4.6 will introduce basic techniques such as MINIMAX, ITERATIVE DEEPENING SEARCH, α-β PRUNING and NEGAMAX. Section 4.7 will introduce the family of *-MINIMAX algorithms developed by Ballard (1983). Furthermore the TRANSPOSITION TABLE and the enhancements for *-MINIMAX is discussed in Section 4.8 Finally we will discuss move ordering techniques in Section 4.9 and the forward-pruning technique MULTI-CUT in Section 4.10.

4.1 Introduction

The decisions that a player can take during the course of a game can be represented as a tree, where the current position is the root node. When the player has multiple moves available the tree branches. If a node succeeds another node that node is called a child node. Terminal positions (i.e., positions where the game has ended) have no branches and are called leaf nodes. The computer can now back up from that leaf node until it finds a branch that was not visited yet. In this way the computer creates a game tree with branches and leaf nodes.

When the player finds the result of a leaf node (win or loss) it can backup the value to the parent branch and eventually to the root node. It is now possible to create a path to a leaf node with the best results.

This becomes more complicated when two players are involved in the game because the second player also wants to win. Therefore the second player will try to make the first player lose the game. This means that, in the game-tree, the second player will choose a branch that results in a loss.

However, in practice it is infeasible to create the whole game tree because of its size. To compensate this, the players will evaluate their position in the game tree. This evaluation function, depending on its correctness, can forecast whether the current branch is a heading towards a win, loss or draw.

4.2 Minimax

Computers can use the *minimax* algorithm (Von Neumann and Morgenstern, 1944) to deal with a game tree. The algorithm searches the game tree to find the best possible move. The first player will try to maximize the score, and is called the MAX player, while the second player, the MIN player, tries to minimize the score. The algorithm will recursively alternate between the MAX player and the MIN player until it reaches a terminal node. The game tree will be searched in a *depth-first* way meaning that the algorithm will search the deepest unvisited branch until a terminal node is reached. The value of the terminal node is then returned to the parent node. When all the branches of a node have been visited, the player can now choose the best branch from that node and return the value of that branch to the parent node again up to the root node.

As stated before, the size of the game tree is usually too large to search through the complete game tree. Instead of searching the game tree until the MINIMAX algorithm finds a terminal node, it is possible to end the search when it has reached a certain depth. This is called a *depth-limited* search. If the algorithm has reached a certain depth without encountering a terminal node it cannot back up a terminal value

(win, loss or draw). Therefore it needs to evaluate its current position using an evaluation function that estimates the value of the reached node. This value will then be backed up instead of the terminal value. The optimal game or *principal variation* is the game where the MAX player always selects the move with the highest evaluation value and the MIN player the move with the lowest evaluation value.

An example of such a search tree limited to depth 3 can be seen in Figure 4.1. The top node is the root node and the bottom nodes are the leaf nodes. The MAX player will explore the left most branch. Now the MIN player explores a branch, again the left most branch. This continues until a leaf node is found. The MAX player finds a leaf node with a value of 6. Next, the MAX player explores the second branch and finds the value -7. The MAX player will choose the move with the highest value, $\text{MAX}(6, -7) = 6$. The value of 6 is returned to the MIN player and the MIN player will explore the second branch. Again, the MAX player will choose the highest value of the two leaf nodes, $\text{MAX}(7, -8) = 7$. The MIN player chooses the node with the lowest value $\text{MIN}(7, 6) = 6$. This process will be repeated until all the children of the root node are explored. Eventually the best move will give a value of 6.

The pseudo code of MINIMAX can be found in Algorithm 1.

Algorithm 1 MINIMAX algorithm

```
function minimax(depth, board)
value = max-value(depth, board)
return value
```

```
function max-value(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
max = −∞
for all moves do
  doMove()
  value = min-value(depth − 1, board)
  undoMove()
  if value > max then
    max = value
  end if
end for
return max
```

```
function min-value(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
min = ∞
for all moves do
  doMove()
  value = max-value(depth − 1, board)
  undoMove()
  if value < min then
    min = value
  end if
end for
return min
```

4.3 Iterative Deepening Search

In advance it is not known how much time the search will cost. ITERATIVE DEEPENING SEARCH (Russell and Norvig, 1995) used in combination with depth-first search makes it possible to find the best search

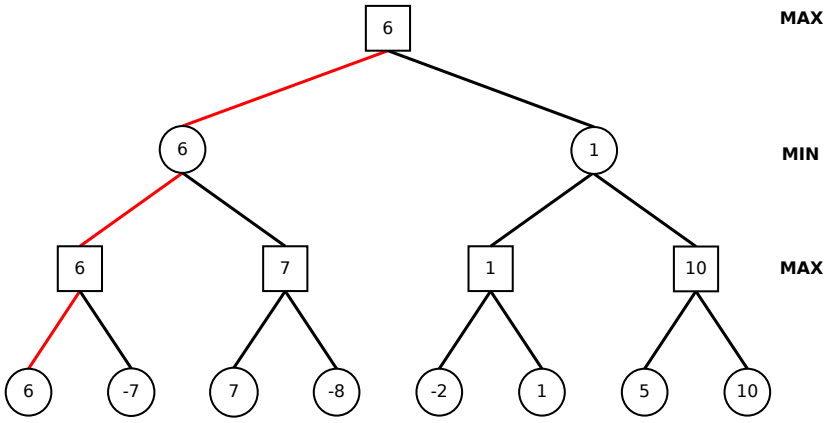


Figure 4.1: Game tree generated by the MINIMAX algorithm with corresponding values of the nodes

depth within a predetermined amount of time.

The algorithm will increase the search depth until it runs out of time. It starts a search with a depth limit of 1. If the search is finished and there is time left the limit is increased by a certain amount (e.g. one). When there is no time left for the search, it backs up the best move found so far. ITERATIVE DEEPENING is also known as an *anytime algorithm* because it has a best move at anytime.

At first this search method seems wasteful, because it visits several states multiple times. However, the majority of the nodes are at depth the maximum search depth. The nodes generated at depths lower than the maximum search depth are a small fraction of the total number of nodes generated during the search.

The nodes at the maximum search depth d are only generated once during a search. All the nodes with a depth lower than d are generated multiple times. Nodes at depth d - *1* are generated twice, first when the maximum depth is d - *1* and second when the maximum depth equals d. Nodes at depth d - *2* are generated tree times, this continues until the root, which are generated d times. The total number of nodes generated at depth d with a branching factor of b, as defined by Russell and Norvig (1995) is $(d)b + (d-1)b^2 + \cdots + (1)b^d$. For $d = 5$ and $b = 10$ the total number of nodes generated is $50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$. If we compare this to a depth-first search with a complexity of b^d. This generates a total amount of $10+100+1,000+10,000+100,000 = 111,110$ nodes.

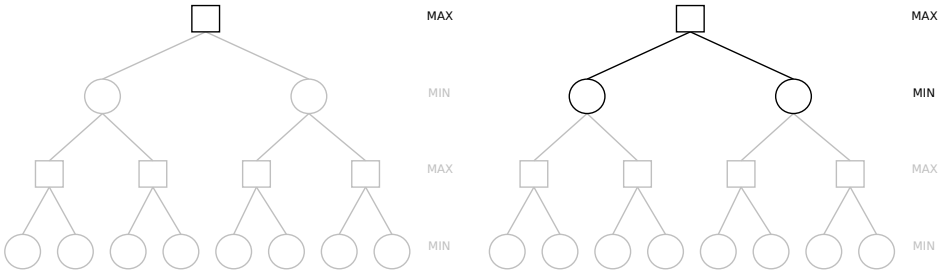
Without search enhancements such as *ordering* of the nodes (Section 4.9) and *pruning* (Section 4.5) ITERATIVE DEEPENING performs worse than a *depth-first* search. With search enhancements ITERATIVE DEEPENING is able to select the optimal path found so far to continue its search at a new depth. This eventually leads to nodes that do not need to be visited anymore and a lower number of generated nodes.

A visual representation of ITERATIVE DEEPENING can be seen in Figure 4.2. Figure 4.2a depicts a situation where the algorithm finished the first iteration and only the root node is visited. In Figure 4.2b the search method has now finished the second iteration, the algorithm can now sort the child node of the root nodes and possible prune a child node. The algorithm continues until it runs out of time or when all the nodes are visited.

4.4 Negamax

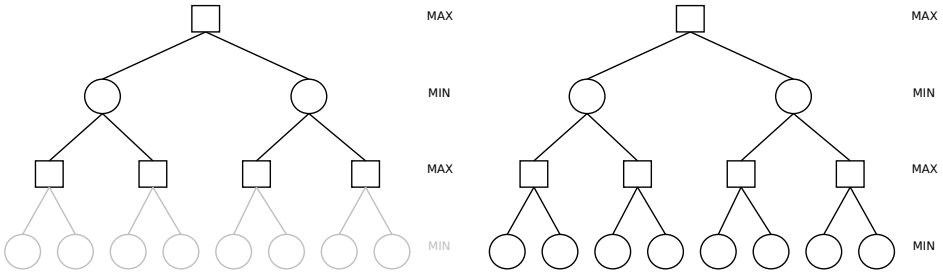
A variation on the MINIMAX algorithm is the NEGAMAX (Knuth and Moore, 1975) algorithm. Both the MIN player and the MAX player choose their optimal line of play. The MIN and MAX player can be interchanged with respect to the line of play when the values are negated.

Suppose the MAX player starts the search. The MIN player will make a move on the second turn. If we interchange the MIN player and MAX player on ply two we only have to negate the values to achieve the same result. However, the original MAX player has become the MIN player now. On the next ply we



(a) depth 0

(b) depth 1



(c) depth 2

(d) depth 3

Figure 4.2: ITERATIVE DEEPENING until ply 3

can interchange the players and negate the values again to compensate this problem. This process can be repeated until the end of the search.

In practice this is a simplified version of the MINIMAX algorithm where both players use the same code to choose the optimal line of play. For coding purposes this a more simple and efficient manner to implement the MINIMAX algorithm.

The pseudo code example is shown in Algorithm 2. By negating the outcome for each player we can reduce the amount of code needed to retrieve the same result. This results in better maintainable code.

Algorithm 2 Negamax algorithm

```
function negamax(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
max = −∞
for all moves do
  doMove()
  value = −negamax(depth − 1, board)
  undoMove()
  if value > max then
    max = value
  end if
end for
return max
```

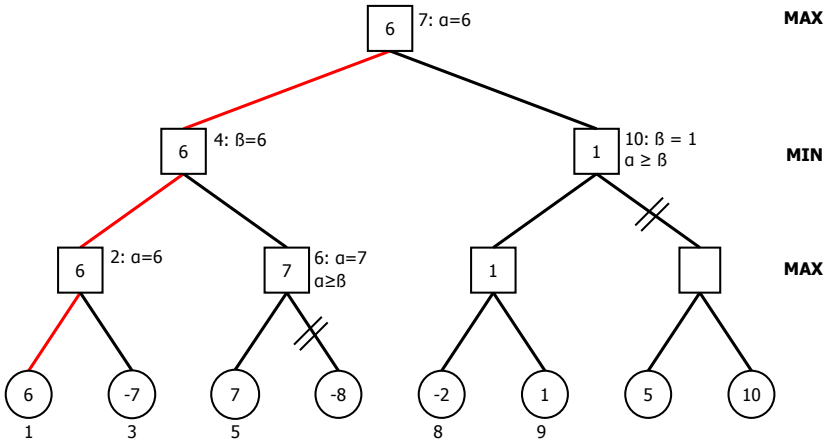


Figure 4.3: α-β pruning in action. The principal variation is red

4.5 α-β Search

A widely used technique for backwards-pruning nodes which are not interesting for the player is the α-β enhancement for NEGAMAX (Knuth and Moore, 1975).

If we take another look at the game-tree in Figure 4.1, it is possible to prune some nodes that will not influence the results of the game-tree. The fourth leaf node with a value of -8 is irrelevant for the result. We will show why, when the third leaf node is found the MAX player can play a move with value 7. Now we substitute the value of the fourth leaf node with x. The value of the MIN node can be computed by $\text{MIN}(6, \text{MAX}(7, x))$. This means that the MAX player will play a move with a value of at least 7. The value of 6 that the MIN player found by exploring the first branch is always a better move for the MIN player then. Therefore the MIN player does not need to know the value of node x to determine his value and this branch can be pruned.

A similar situation occurs at the second branch of the root node. The MAX player chooses to play the leaf node with the value 1. This value is returned to the MIN player before searching the second branch of the MIN player. Again we substitute the value of the second branch with a variable, y. The value of the root node can be computed by $\text{MAX}(6, \text{MIN}(1, y))$. The MIN player will play a move with a value of at most 1. However the MAX player can play a move with a value of 6 at the root. Therefore the second branch of the MIN player does not need to be explored and can be pruned. A visual representation can be seen in Figure 4.3

In practice the minimum feasible value for the MAX player is stored in α and the maximum feasible value for the MIN player is stored in β. These values are passed down in the search method.

The pseudo code of the NEGAMAX implementation of α-β pruning is shown in Algorithm 3. A similar variant exists for MINIMAX however it will take more code to produce the same result.

4.6 Expectimax

To deal with chance nodes we cannot use the standard NEGAMAX algorithm, instead the EXPECTIMAX (Michie, 1966) algorithm can be used. The method is similar to NEGAMAX in the sense that it visits all nodes in the game tree. The difference is that EXPECTIMAX deals with chance nodes in addition to MIN and MAX nodes.

To define the value of a chance node we cannot simply select the highest or lowest value from one of the successors. Instead an expected value has to be calculated. Every child node has a weight equal to the probability that the child node occurs. The EXPECTIMAX value of a node is the weighted sum of the successors' values:

$$E(x) = \sum P_i \times E(i) \tag{4.1}$$

Algorithm 3 Negamax algorithm with α-β pruning

```
function negamax(depth, board, α, β)
if depth = 0 or terminal then
  return evaluate()
end if
for all moves do
  doMove()
  α = max(α, −negamax(depth − 1, board, −β, −α))
  undoMove()
  {beta cutoff}
  if α ≥ β then
    return α
  end if
end for
return α
```

Algorithm 4 The negamax formulation of the EXPECTIMAX method

```
function negamax(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
max = −∞
for all moves do
  doMove()
  if chanceEvent() then
    value = −expectimax(depth − 1, board)
  else
    value = −negamax(depth − 1, board)
  end if
  undoMove()
  if value > max then
    max = value
  end if
end for
return max
```

```
function expectimax(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
sum = 0
for all chance events do
  doChanceEvent(board, event)
  sum+ = probability(event) × −search(depth, board)
  undoChanceEvent(board, event)
end for
return sum
```

here $E(i)$ is the evaluation value of node i and P_i is the probability of node i. Algorithm 4 shows an example of EXPECTIMAX used in a NEGAMAX implementation.

Figure 4.4 shows an example of the EXPECTIMAX algorithm (Hauk, 2004). The MAX player encounters a chance node. There are six possible (uniform) chances nodes, node B till G. The MIN player will choose the node with the lowest value. Therefore node B will get the value -5, node C will get -10, etc.

In the next step we multiply the chance of every node, $\frac{1}{6}$, with the value of the node and summate

the outcomes. This results that the value of node A equals −3.5.

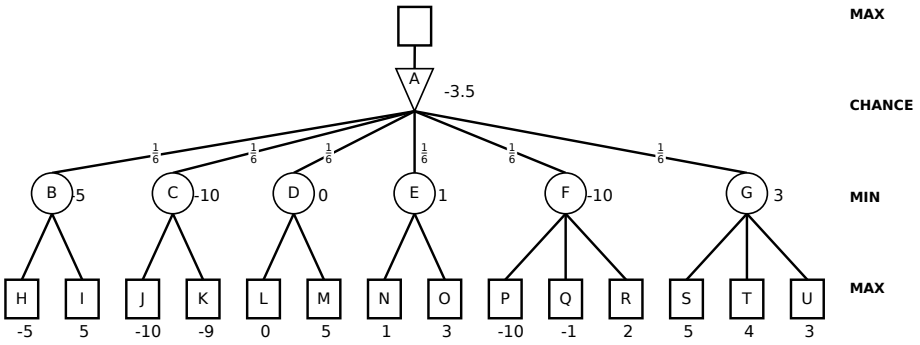


Figure 4.4: An EXPECTIMAX tree with uniform chances

4.7 *-Minimax Algorithms

EXPECTIMAX is a suitable algorithm to handle chance events. However the NEGAMAX algorithm is able to prune a great number of nodes using the α-β enhancement. We would like a similar technique for EXPECTIMAX but we cannot know the value of a chance node before we have visited all the children nodes. However, if the upper and lower bound on the game values are known and the α and β values of the parent node were passed down it is possible to calculate if the theoretical possible value for the chance node falls outside the bounds. Ballard (1983) introduced an α-β enhancement for EXPECTIMAX called *-MINIMAX. These type of nodes are referred to as *nodes by Ballard. The procedure for the standard MIN and MAX nodes will stay unchanged.

Ballard has proposed two versions of this algorithm, called STAR1 and STAR2. These will be discussed in Section 4.7.1 and 4.7.2. An extended version called STAR2.5 is a refined version of the Star2 algorithm and will be discussed in Section 4.8.2.

4.7.1 Star1

This section will introduce the idea of α-β pruning in chance nodes. With every new piece of information in a chance node, the theoretical bounds of the chance node can be narrowed until a cutoff occurs or all child nodes are visited. The STAR1 algorithm will first be explained using uniform chances and afterwards non-uniform chances will be introduced.

As an example, consider an evaluation function which is bounded by the interval [-10, 10]. The current chance node has four children (with equal probability) of which two have been visited. This situation is depicted in Figure 4.5. We can calculate the lower bound of the chance node , namely $\frac{1}{4}(5+3+-10+-10) = -3$. The upper bound of the chance node is $\frac{1}{4}(5+3+10+10) = 7$. If an α value ≥ 7 was passed down to the chance node a pruning can take a place. Even in the best possible case the value of the complete chance node will not influence the principal variation. Similarly, if a β value ≤ -3 was passed down, a pruning occurs.

The lower and upper bound of the evaluation function are denoted as L and U. $V_1, V_2 \cdots V_n$ are the values of the N successors of the current chance node, whose ith successor is being searched.

The lower bound of the chance node is

$$\frac{(V_1 + V_2, \cdots + V_{i-1}) + V_i + L \times (N - i)}{N} \tag{4.2}$$

and the upper bound is

$$\frac{(V_1 + V_2, \cdots + V_{i-1}) + V_i + U \times (N - i)}{N} \tag{4.3}$$

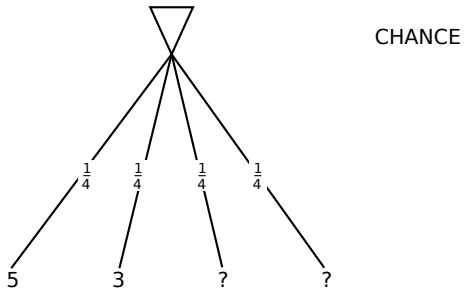


Figure 4.5: A chance node

When the α and β values are passed down from the parent node it is possible to get a pruning. Like in the α-β enhancement for NEGAMAX a cutoff occurs when the upper bound of the chance node is less or equal to α.

$$\frac{(V_1 + V_2, \cdots + V_{i-1}) + V_i + U \times (N - i)}{N} \leq \alpha \tag{4.4}$$

or if the lower bound is larger or equal to β.

$$\frac{(V_1 + V_2, \cdots + V_{i-1}) + V_i + L \times (N - i)}{N} \geq \beta \tag{4.5}$$

This creates a possibility of an α-β-like cutoff in the chance node. STAR1 gives also the possibility to calculate a new α and β for child nodes of a chance node. This method can be generalized for all chance nodes. This cutoff will not influence the principal variation. By rewriting Formulas 4.4 and 4.5 a new window for the child node is calculated which can trigger a cutoff in the child node. If we denote A_i as the lower bound of the ith successor then

$$A_i = N \times \alpha - (V_1 + V_2 \cdots V_{i-1}) - U \times (N - i) \tag{4.6}$$

where α is the same α as passed down to the chance node. For the upper bound, denoted as B_i we have

$$B_i = N \times \beta - (V_1 + V_2 + \cdots V_{i-1}) - L \times (N - i). \tag{4.7}$$

Again β stays unchanged.

Initially the value of A_1 is $N \times \alpha - U \times (N - 1)$ or $N \times (\alpha - U) + U$ and the value for B_1 equals $N \times \beta - L \times (N - 1)$ of $N \times (\beta - L) + L$. The upcoming values of A and B can computed using Formula 4.6 and Formula 4.7. Ballard uses an iterative method to update the values of A and B after calculating the initial values.

$$A_{i+1} = A_i + U - V_i \tag{4.8}$$

$$B_{i+1} = B_i + L - V_i \tag{4.9}$$

The implementation of these formulas in pseudo-code can be seen in Algorithm 5. The values A and B are reused every iteration and therefore have no index i. AX and BX represent the values that will be passed down to the children. These values are limited by the bounds of the evaluation function.

We take a look at the example in Figure 4.6. The node A is entered with a lower bound (α) of 3 and an upper bound (β) of 4. We will write this as $[3, 4]$. The game values vary between $L = -10$ and $U = 10$. When the first child of A is searched we calculate the upper and lower bound of the node. α equals $\frac{1}{3}(3 \times L) = L$ and the β value equals $\frac{1}{3}(3 \times U) = U$, this means that the bounds for child B are $[-10, 10]$. After searching node B, we find a value of 2. If we move on to the second child, C, we can calculate the range of values for node A, a lower bound of $\frac{1}{3}(2 + 2 \times L) = \frac{1}{3} \times -18 = -6$ and an upper bound of $\frac{1}{3}(2 + 2 \times U) = \frac{1}{3} \times 22 = 7\frac{1}{3}$. This does not result in a cutoff because -6 is not greater or equal

Algorithm 5 The negamax formulation of the STAR1 algorithm with uniform chances

```
function search(depth, board)
if depth = 0 or terminal then
  return evaluate()
end if
max = −∞
for all moves do
  doMove()
  if chanceEvent() then
    α = max(α, −star1(depth − 1, board, −β, −α))
  else
    α = max(α, −negamax(depth − 1, board, −β, −α))
  end if
  undoMove()
  if α ≥ β then
    return α
  end if
end for
return α
```

```
function star1(depth, board, α, β)
if depth = 0 or terminal then
  return evaluate()
end if
N = generateChanceEvents();
A = N × (β − U) + U
B = N × (α − L) + L
vsum = 0

for i = 1, i ≤ N, i++ do
  AX = max(A, L)
  BX = min(B, U)
  doChanceEvent(board, event)
  value = −search(depth, board, −BX, −AX)
  undoChanceEvent(board, event)
  vsum+ = v

  if v ≤ A then
    vsum+ = U × (N − i)
    return vsum/N
  end if
  if v ≥ B then
    vsum+ = L × (N − 1)
    return vsum/N
  end if

  A+ = U − value
  B+ = L − value
end for
return vsum/N
```

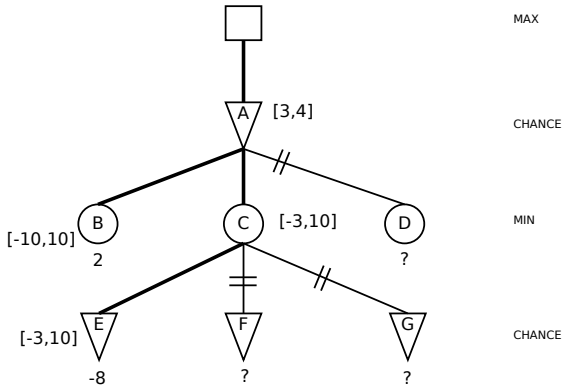


Figure 4.6: A chance node

to 4 and $7\frac{1}{3}$ is not smaller or equal to 3. Before visiting node C we calculate the new α and β values we will pass down using Formula 4.6,

$$A_i = 3 \times 3 - 2 - 10 = -3 \tag{4.10}$$

$$B_i = 3 \times 4 - 2 + 10 = 20 \tag{4.11}$$

The β value for node C is 20, however the highest possible value of a leaf node is 10. Therefore the β value will be set to 10 instead of 20. The node C will be passed down a window of $[-3, 10]$.

The first node visited is the node E, which returns a value of -8. Since $-8 \leq -3$ this will cause a cutoff at node C. In node A we check again if a cutoff has occurred by using Formula 4.4 and Formula 4.5

$$\frac{(2 - 8 + 10)}{3} \leq \alpha = 1\frac{1}{3} \leq 3 \tag{4.12}$$

It is clear that $1\frac{1}{3} \leq 3$ and there will also be a cutoff at node A. The upper bound is not calculated because it is not important anymore. We cannot return the exact value of node A because node D is pruned. The original α value is returned as the value of node A. This ensures that node A will automatically be pruned at a lower depth.

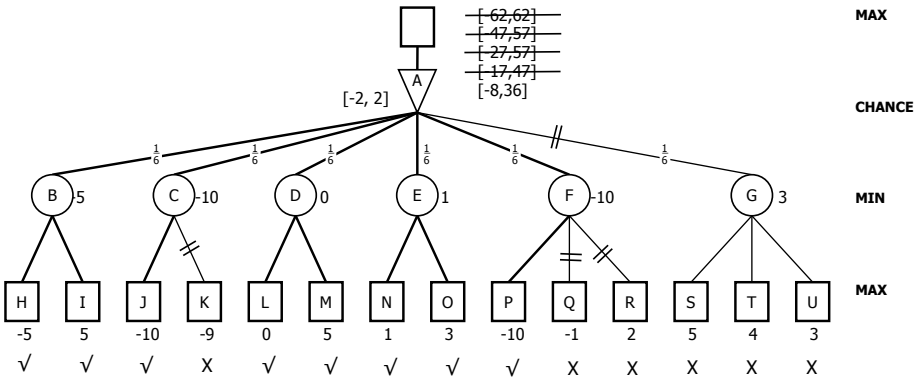


Figure 4.7: The Star1 algorithm in action

Figure 4.7 depicts the EXPECTIMAX tree example of Figure 4.4, now visited by the STAR1 algorithm. The evaluation function is still bounded by the bounds $[-10, 10]$. The node A is entered with a window of $[-2, 2]$. The bounds for the first node, node B, are computed using Equations 4.6 and 4.7: $A_1 = 6 \times -2 - 10 \times (6 - 1) = -62$ and $B_1 = 6 \times 2 - -10 \times (6 - 1) = 62$. The values of A_1 and B_1 are limited to the bounds of the evaluation function, therefore the node B will be entered with the bounds $[-10, 10]$. Node B returns the value -5 and the new window for node C can be calculated using Formula 4.8 and 4.9: $A_2 = -62 + 10 + 5 = -47$ and $B_2 = 62 - 10 + 5 = 57$. Again the window will be bounded by the evaluation function. When the search continues with the other children, the windows are updated for every child accordingly. Up until node F the windows will fall outside the bounds of the evaluation function. Node F will be entered with a window of $[-8, 36]$. When node P is visited the nodes Q and R will be pruned because the value of node P, -10 falls outside the window of $[-8, 36]$. This will also trigger a cutoff in node A, which has a window of $[-2, 2]$ and therefore node G will also be pruned. In Figure 4.4 it was clear that node A has the value -3.5 and indeed falls outside the window $[-2, 2]$.

Up to known the theory and examples assume that the probabilities of each child are uniform. In practice this is not always the case therefore non-uniform probabilities are introduced into the STAR1 algorithm.

Formula 4.2 and Formula 4.3 can be rewritten to calculate the new lower and upper bound of the chance nodes. Here, P_i is the probability of node i occurring:

$$\frac{(P_1 \times V_1 + P_2 \times V_2, \cdots + P_{i-1} \times V_{i-1}) + P_i \times V_i + L \times (1 - P_1 - P_2 - \cdots - P_i)}{P_i} \tag{4.13}$$

and

$$\frac{(P_1 \times V_1 + P_2 \times V_2, \cdots + P_{i-1} \times V_{i-1}) + P_i \times V_i + U \times (1 - P_1 - P_2 - \cdots - P_i)}{P_i}. \tag{4.14}$$

Formula 4.6 and Formula 4.7 can be rewritten to calculate the values for A_i and B_i:

$$A_i = \frac{\alpha - U \times (1 - P_1 - P_2 - \cdots - P_i) - (P_1 \times V_1 + P_2 \times V_2 + \cdots + P_{i-1} \times V_{i-1})}{P_i} \tag{4.15}$$

where α is the same α as passed down to the chance node. For the upper bound, denoted as B_i

$$B_i = \frac{\beta - L \times (1 - P_1 - P_2 - \cdots - P_i) - (P_1 \times V_1 + P_2 \times V_2 + \cdots + P_{i-1} \times V_{i-1})}{P_i}. \tag{4.16}$$

Again these formulas can be computed incrementally. To simplify the equations, $Y = (1 - P_1 - P_2 - \cdots - P_i)$ can be substituted where Y can be computed incrementally, $Y_{i+1} = Y_i - P_{i+1}$ and $Y_1 = 0$. Secondly, $X = (P_1 \times V_1 + P_2 \times V_2 + \cdots + P_{i-1} \times V_{i-1})$ is replaced, which can also be computed incrementally, $X_{i+1} = X_i + P_i \times V_i$ and where $X_1 = 0$. By using X and Y it is possible to rewrite in a compact manner the incremental update of A and B:

$$A_i = \frac{\alpha - U \times Y_i - X_i}{P_i}, B_i = \frac{\beta - L \times Y_i - X_i}{P_i}. \tag{4.17}$$

4.7.2 Star2

The STAR1 algorithm is an enhancement for EXPECTIMAX but still needs to visit a large number of nodes to realize a cutoff. This is a result of the fact that STAR1 assumes the worst-case scenario about unseen children's value. STAR1 does not and cannot use any information about the structure of the game tree. Therefore Ballard (1983) has also introduced the STAR2 algorithm.

In most search trees of board games the order of play is always the same, a MIN player follows a MAX player and vice versa. Games involving chance events can add an additional step between the MIN player and the MAX player, a chance event. Ballard (1983) refers to this type of trees as *regular *-minimax trees*. In NEGAMAX a MIN players follows a MAX player. STAR2 can use this information to improve the lower bound before searching the children. Instead of assuming that the child has a value equal to the lower bound or upper bound of the evaluation function, a more accurate lower bound could be calculated by searching a k number of successors of the child node. This will improve the search window because it is not likely that all the values of the child's successors equal the lower and upper bound values, the

terminal values of the evaluation function. This is called the *probing* phase of the algorithm. If k is equal to 0 than STAR2 has no effect.

The effectiveness of the probing phase depends on the selection of the successor that will be probed. This can be done in several ways, e.g. selecting a random move, the first generated move. Using move ordering as described in Section 4.9 will most likely return the best move to probe. However, even a random move can reduce the search window because it has a high probability that it is not equal to the lower or upper bound of the evaluation function. In the worst case STAR2 causes a search overhead.

The probed values for each child node will be stored in the array W and will be used to check for cutoffs and calculate the new initial bounds. For simplicity it is assumed that the probabilities are uniform. The Formula 4.5 is modified in such a way that instead of using the lower bound, L , the probed values are used:

$$\frac{(V_1 + V_2 + \cdots + V_{i-1}) + V_i + (W_{i+1} + \cdots + W_N)}{N} \geq \beta. \tag{4.18}$$

The values of W_1 to W_{i-1} can be substituted by the real values of the corresponding nodes. In practice it is easier to retain the values $W_1, \ldots, W_{i-1}$ and replace them with the values $V_1, \ldots, V_{i-1}$.

Because of the probing phase is not possible to calculate the values for B using 4.3. Therefore the value of B will be calculated using the following formula:

$$B_i = N \times \beta - (V_1 + V_2 + \cdots + V_{i-1}) - (W_{i+1} + \cdots + W_N). \tag{4.19}$$

The value of B is initialized in a similar manner as done in STAR1 by adjusting Formula 4.19. $B_1 = N \times \beta - (W_1 + W_2 + \cdots + W_N)$ and can be updated incrementally by $B_{i+1} = B_i + W_i - V_i$.

Algorithm 6 shows the pseudo-code of the STAR2 algorithm. The probe method is a simple method that selects k number of children at random or by the means of move ordering and searches them. Another possibility is to adapt the standard search method to deal with probing.

In Figure 4.8 (Hauk, 2004) the STAR2 algorithm visits the search tree from Figure 4.4. By only visiting the probed nodes, STAR2 creates a cutoff before searching all the nodes of the child. Eventually the search window becomes $[-8, 62]$ or $[-8, 10]$ when the window is bound to the lower and upper bound. When node F returns the value -10 this value falls outside the window $[-8, 62]$ and a cutoff occurs.

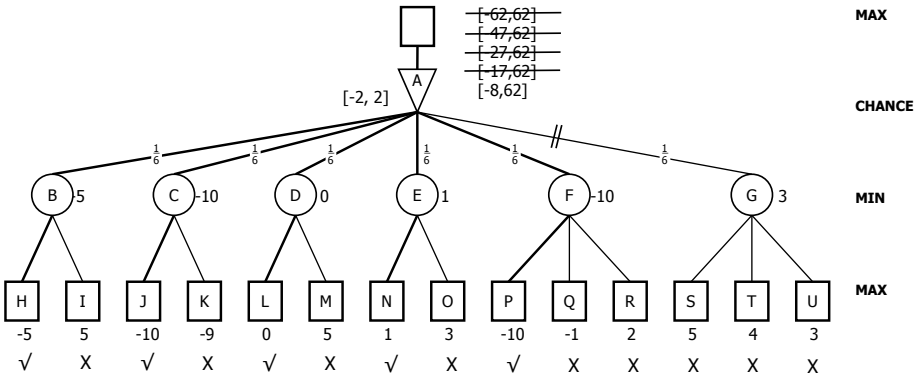


Figure 4.8: The STAR2 algorithm in action

For the probing phase only the calculation for B needs to be updated. This can be done iteratively:

$$B_i = \frac{\beta - W_i - X_i}{P_i} \tag{4.20}$$

where $W_i = (W_{i+1} + \cdots + W_N)$, the sum of the unvisited nodes not yet probed.

Algorithm 6 The negamax formulation of the STAR2 algorithm with uniform chances

```
function star2(depth, board, α, β)
if depth = 0 or terminal then
  return evaluate()
end if
N = generateChanceEvents();
W[N] = array()
A = N × (β − U) + U
B = N × (α − L) + L
AX = max(A, L)
vsum = 0

for i = 1, i ≤ N, i++ do
  BX = min(B, U)
  value = −probe(depth, board, −BX, −AX)
  vsum+ = value
  W[i] = value
  if value ≥ B then
    vsum+ = L × (N − 1)
    return vsum/N
  end if
  B+ = L − value
end for
for i = 1, i ≤ N, i++ do
  B+ = W[i]
  AX = max(A, L)
  BX = min(B, U)
  doChanceEvent(board, event)
  value = −search(depth, board, −BX, −AX)
  undoChanceEvent(board, event)
  vsum+ = v

  if value ≤ A then
    vsum+ = U × (N − i)
    return vsum/N
  end if
  if value ≥ B then
    vsum+ = L × (N − 1)
    return vsum/N
  end if

  A+ = U − value
  B− = value
end for
return vsum/N
```

4.8 Transposition Tables

During a search identical board positions can occur, called transpositions. Instead of searching the board position again it is possible to store the result of the previous search in a so-called *transposition table* (Greenblatt, Eastlake, and Crocker, 1967 and Slate and Atkin, 1977). The transposition table stores the board position, using a generated key with information of the board, such as the best move and the value of the best move. To prevent storing the complete board the transposition table stores a 64-bit *hash value* of the board position using the *Zobrist hash* (Zobrist, 1970). This will be discussed in Section 4.8.1. In Section 4.8.2 an improved version of the STAR2 algorithm will be presented using transposition tables.

4.8.1 Zobrist Hashing

Zobrist hashing (Zobrist, 1970) is a method that generates a hash value for a game board. For each piece at each square the hash method needs a unique value. In Stratego there are 12 ranks and 3 possible states of every piece: not known and not moved; not known and moved; and known. There are 92 reachable squares on the game board. This gives a total of $12 \times 3 \times 92 = 3{,}312$ unique 64-bit values. These values will be generated at the initialization of the computer player using a random generator.

The game hash can change on four occasions, piece placement, piece movement, piece removal and change of player.

1. On piece placement, when players add the pieces to the game board, the new hash value is computed by a XOR operation on the board with the hash value of a certain piece at a certain square.

2. On piece removal the same procedure is executed. The hash value is computed by a XOR operation of the hash value of a certain piece at a certain position with the current hash value.

3. When a piece moves from a square to another square, the piece is first removed from the old position and replaced at the new position. It is similar to a piece removal and a piece placement.

4. Every time the players change turns, the hash will be updated and thus keeping track of each players turn.

Initially the empty board hash has the value 0. The advantage of the XOR operation is that it is a bitwise operation and therefore a fast operation for a machine.

The computed hash value now serves as the key for the associated board position in the transposition table. Instead of using the complete 64 bit key, the first n bits of the hash value are used as the key, the *hash index*. The remaining bits, the *hash key*, are used to discriminate between different positions mapping the same hash index.

Components of the table

A *transposition table* will need the following components to be effective (Breuker, 1998):

key the remaining bits of the hash value

move the best move found so far for the current position

score the score associated with the best move

flag defines the type of score: lower bound, upper bound or an exact value

depth the relative depth at which the best move was found

During a search the positions are retrieved from the transposition table before searching the board position and afterwards when the information of the position will be stored in the transposition table.

If the position exists in the transposition table before the position is searched the algorithm can check if the information is useful, this depends on the *flag* and the *depth*.

Algorithm 7 Negamax with transposition tables

```
function negamax(depth, board, α, β)
oldα = α
retrieve(board, depth)
{if the position is not found, TTdepth will be -1}
if TTdepth ≥ depth then
  if flag = EXACTVALUE then
    return score
  else if flag = LOWERBOUND then
    α = max(α, TTvalue)
  else if flag = UPPERBOUND then
    β = min(β, TTvalue)
  end if
  if α ≥ β then
    return α
  end if
end if
if depth = 0 or terminal then
  return evaluate()
end if
bestvalue = −∞
for all moves do
  doMove()
  value = −negamax(depth − 1, board, −β, −α)
  undoMove()
  α = max(α, value)
  if value ≥ bestValue then
    bestValue = value
  end if
  if α ≥ β then
    break
  end if
end for
if bestValue ≤ oldα then
  flag = UPPERBOUND
else if bestValue ≥ β then
  flag = LOWERBOUND
else
  flag = EXACTVALUE
end if
store(board, depth, bestMove, bestValue, flag)
return α
```

1. When the remaining search depth is lower than the search depth of the transposition table entry the information can be used, because it guarantees that the information in the transposition table gives at least the same information as a search would do. If the value is an exact value this value can be returned immediately and the position does not have to be searched anymore.

2. When the remaining search depth is lower than the search depth of the transposition table entry and the value is an upper or lower bound, the value can be used to narrow the α-β window. The α value can be adjusted if the value is a lower bound and the value is larger than α. The same holds for β when the value is an upper bound and the value is smaller than β. Furthermore a cutoff can occur when α is larger or equal to β.

3. When the remaining search depth is higher than the search depth of the transposition table entry the information in the transposition table is not useful, except the best move. The best move can be searched first and could possibly lead to an early cutoff. See also Section 4.9 for more information about move ordering.

After searching the position the information found can be stored in the transposition table. When the transposition table does not have an entry with the same key the information can be stored directly. However, when the entry already exists, called a *collision*, a choice has to be made which information is more important to preserve. Breuker, Uiterwijk, and Vd Herik (1994) have proposed several *replacement schemes*:

1. **Deep**
   In case of a collision, this scheme preserves the board position with the deepest subtree. The scheme assumes that a deeper search has resulted in more nodes visited than a shallower search and therefore gives a better estimate of the utility of the board position.

2. **New**
   In this case the transposition table will always replace the old entry with the information of the current search. This is based upon the observation that changes occur locally.

3. **Old**
   This scheme works in a similar way as the **New** replacement scheme. However this scheme will never replace any entry in the transposition table.

4. **Big**
   When the board position contains many forced moves or many cutoffs, in case of good move ordering, the depth of the search is not a good indicator of the amount of search done on the board position. Thus instead of keeping track of the depth, one can keep track of the amount of nodes visited during the search. This scheme will therefore preserve the entry with the largest subtree. A drawback is that the number of nodes visited in the subtree also has to be stored in the transposition table.

The **Deep** scheme was implemented based upon the experiments by Breuker *et al.* (1994).

4.8.2 StarETC

Ballard has proposed several enhancements for the Star2 algorithm, among them the use of a transposition table in *-minimax (Ballard, 1983). The transposition table can be used in the process of probing for values (Veness, 2006). StarETC is the stochastic variant for *Enhanced Transposition Cutoff* (ETC). The values for the chance node children may be retrieved from the transposition table and stored in the array W as described in Section 4.7.2. Instead of searching the child node's successors a bound or exact value can be retrieved from the transposition table. However, the information in the transposition table is only used when the relative depth of the table entry has at least a depth of $d - 1$. The probe phase normally returns an exact value of the child's successor. The transposition table does not only store exact values but also lower and upper bounds. To use this information, Star2 also needs to keep track of the separate lower and upper bounds. The array W, as described in Section 4.7.2, is extended to two dimensions, one for each bound. When the value of the table entry is an exact value, the upper and lower bound of W are both equal to the entry value.

The second advantage of the transposition table is a chance on a preliminary cutoff. Every time a useful table entry is found and W is updated, the lower bound of the board position is calculated using: $P \cdot W_{lower}$ and the upper bound is calculated using $P \cdot W_{upper}$.
P is an array containing the probabilities of each chance event. When the lower bound is $\leq \alpha$ or the upper bound is $\geq \beta$ a cutoff occurs.

Finally, it is possible to store extra information in the transposition table when the Star1 creates a cutoff. In the case of a fail low the position is stored as an upper bound with a value equal to A.

Algorithm 8 shows the pseudo-code for the probing enhancement for Star2. It makes use of two extra functions, *lowerbound*() and *upperbound*(), to calculate the lower and upper bound of the probed cq. retrieved values for the nodes using Formula 4.19 and a similar formula for the lower bound. Also there are two functions to calculate the successor values. Depending on the implementation of these four functions, the algorithm can handle uniform and non-uniform probabilities. The array of stored values is expanded with a separate lower and upper bound to deal with the information from the transposition table.

4.9 Move Ordering

As stated before, move ordering is important in α-β based search. It can reduce the search tree significantly. When searching it is preferable to search the best moves first because it will be more likely that they create a cutoff.

The moves can be ordered using two approaches, static or dynamic. Static move ordering happens according to some predefined rules, mostly based on human strategies and knowledge of the game. The History Heuristic (Schaeffer, 1989) and transposition table ordering are dynamic ways of ordering the moves. This means that the move ordering depends on the positions visited so far.

The moves will be weighted depending on the ordering, the possible best move will have the highest value and the worst move has the lowest value. Moves suggested by dynamic ordering will have the highest value because they have proven to be a good move in previous searches. A move that is in the transposition table will be searched first. Winning moves or capturing moves are increased with a large value ensuring a win or a capture of a piece. Next the value of the move is incremented with the value of the history heuristic. Finally the value is slightly increased or decreased with values determined by the static ordering.

4.9.1 Static Ordering

Static ordering uses information about the type of piece and the goal and destination of the move. In Stratego we have ordered the moves as follows

1. A move that results in a direct win, capturing the opponent's Flag.
2. An attacking move with a sure win.
3. A move attacking an unknown piece.
4. A move towards the opponent
5. A move sideways
6. A move backwards
7. An attacking move with a sure loss

The moves that result in a win, attack an unknown piece have a large positive influence in the move ordering. Non-capturing moves towards the opponent cause a slight increase of the value, moving away from the opponent is penalized. Finally, a clear loss results in a large decrease of the move's value.

Algorithm 8 The negamax formulation of the StarETC algorithm

```
function starETC(depth, board, α, β)
if depth = 0 or terminal then
  return evaluate()
end if
if retrieve(board, depth) == success then
  if entry = EXACTVALUE then
    return entry.value
  end if
  if entry = LOWERBOUND then
    if entry.value ≥ β then
      return entry.value
      α = max(α, entry.value)
    end if
  end if
  if entry = UPPERBOUND then
    if entry.value ≤ α then
      return entry.value
      β = min(β, entry.value)
    end if
  end if
end if

N = generateChanceEvents();
W[N] = array()

for i = 1, i ≤ N, i++ do
  if retrieve(event[i], depth − 1) == success then
    if entry = LOWERBOUND OR EXACTVALUE then
      W[i].lower = entry.value
      if lowerbound(W) ≥ β then
        store(board, depth, LOWERBOUND, lowerbound(W)
        return lowerbound(W)
      end if
    else if entry = UPPERBOUND OR EXACTVALUE then
      W[i].upper = entry.value
      if upperbound(W) ≤ α then
        store(board, depth, UPPERBOUND, upperbound(W)
        return upperbound(W)
      end if
    end if
    if entry = LOWERBOUND then
      if entry.value ≥ β then
        return entry.value
        α = max(α, entry.value)
      end if
    end if
    if entry = UPPERBOUND then
      if entry.value ≤ α then
        return entry.value
        β = min(β, entry.value)
      end if
    end if
  end if
end for
```

Algorithm continued on next page

Algorithm 8 continued

```
for i = 1, i ≤ N, i++ do
  B = successorMax(W, β)
  BX = min(B, U)
  value = −probe(depth − 1, board, −BX, −W[i].lower)
  W[i].lower = value
  if value ≥ B then
    store(board, depth, LOWERBOUND, lowerbound(W))
    return lowerbound(W)
  end if
end for
for i = 1, i ≤ N, i++ do
  A = successorMin(W, α)
  B = successorMax(W, β)
  AX = max(A, L)
  BX = min(B, U)
  doChanceEvent(board, event)
  value = −search(depth, board, −BX, −AX)
  undoChanceEvent(board, event)
  W[i].lower = value
  W[i].upper = value

  if v ≤ A then
    store(board, depth, UPPERBOUND, upperbound(W))
    return upperbound(W)
  end if
  if v ≥ B then
    store(board, depth, LOWERBOUND, lowerbound(W))
    return lowerbound(W)
  end if
end for

store(board, depth, EXACTVALUE, lowerbound(W))

return lowerbound(W)
```

4.9.2 History Heuristic

Schaeffer (1989) introduced this technique to increase the effectiveness of move ordering. When a certain move m is the best move in a certain position, possibly creating a cutoff, it has the possibility of being the best move in another similar position. Therefore it would be interesting to start the search with move m. The HISTORY HEURISTICS stores all these move in a history table, where each entry is a move with associated value. Every time a move creates a cutoff or is the best move after searching the tree, the table entry of the move is incremented with a value, e.g. 2^d, d or 1 (Winands *et al.*, 2006), where d is the depth of the search tree so far. The history table is emptied every time the search is initialized.

Based upon the literature, our implementation of Stratego updates the values in the HISTORY HEURISTIC are with the value d.

4.9.3 Transposition Table Ordering

When the transposition table has an entry of the current position visited with a higher depth, the information in the entry is used to return an exact value or adjust the lower and upper bound for the search. When the entry in the transposition table is not an exact value, the move from the entry can be searched first and might lead to a cutoff. If the search depth of the entry is lower than the search depth remaining for the current position, this information is not useful to update the lower and upper bounds but the best move of the entry may still be the best move when searched deeper and possibly creating a cutoff (Breuker, 1998). Therefore the search algorithm will visit the transposition table move first. When the transposition table entry is useful, the corresponding move will not be visited anymore.

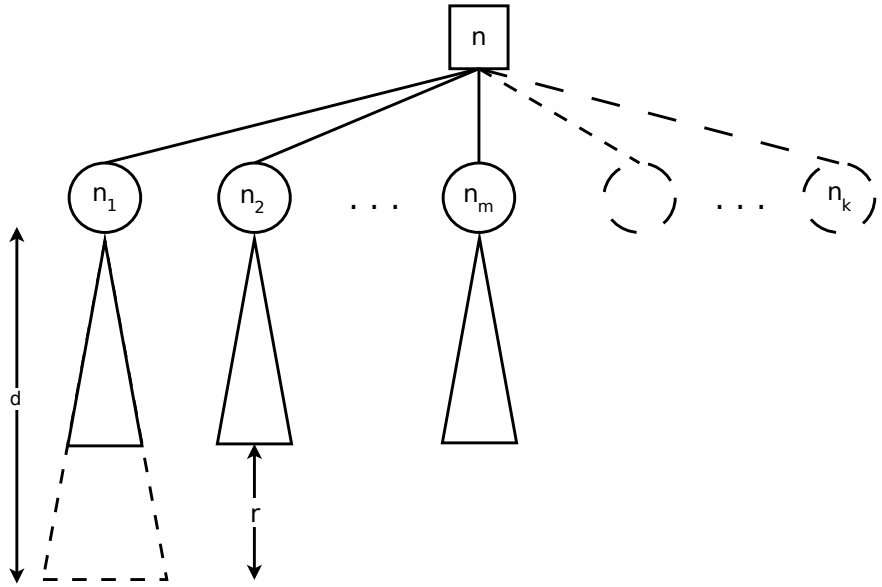


Figure 4.9: MULTI-CUT pruning

4.10 Multi-Cut

MULTI-CUT (Björnsson and Marsland, 2001) is a forward-pruning technique which may improve the playing strength of a program. It is used in a combination with α-β pruning.

The nodes of a tree, using a NEGAMAX implementation, can be identified as follows:

1. The pv-node is a node that is a part of the principal variation, e.g. the root node. At least one child of the pv-node must have a minimax value and is the next pv-node, all remaining children are then cut-nodes.

2. A child of a cut-node which has a value higher than the value of the principal variation value, becomes an all-node.

3. Every child of an all-node is a cut-node.

In α-β it is possible that the first move of a cut-node will create a cutoff and the other children do not have to be searched anymore. The idea of MULTI-CUT is that a cut-node has many successors that cause a β cutoff and therefore will be pruned. MULTI-CUT will visit some of the successors at a reduces search depth and determine if the node can be pruned in advance.

Björnsson and Marsland (2001) combines MULTI-CUT with a principal variation search and searches the principal variation first before applying MULTI-CUT. Winands *et al.* (2005) created a MULTI-CUT that is able to prune at all-nodes and shows it is safe to forward prune at all-nodes and gives a reduction of the search. This version of MULTI-CUT is implemented in our Stratego player.

In MULTI-CUT, M children of a possible cut-node are searched with a depth of $d - 1 - R$, where d is the search depth remaining and r is a search reduction. When C moves of the total M moves return a value larger than β the node is pruned in advance, where $1 \leq C \leq M$. This is depicted in Figure 4.9 (Björnsson and Marsland, 2001). The dotted area below n_1 the children after n_M are the savings that MULTI-CUT creates when a successful pruning was created. If a cutoff occurs, the node will return the β value. Otherwise, the node will be searched in full-depth. When no cutoff occurs the nodes visited by MULTI-CUT are extra overhead but could lead to move ordering information for the remaining search.

Chapter 5

Evaluation Function

A game with a full search-based approach would not need a complex evaluation function because the only outcomes the leaf nodes have is win, loss or draw. However, when a game is too complex, such as Stratego, only a search-based approach does not suffice. Therefore a combination of search and knowledge, in the form of an evaluation function, is needed.

An evaluation function can implement human knowledge and strategies of the game or computer generated information such as an end-game database. An end-game database for Stratego would be too large to create. The evaluation function of Stratego mostly depends on human knowledge.

5.1 Piece Value

An evaluation function returns a numerical value to express the utility of the current position. Therefore, a value is assigned to each position. A way to do this is by counting the material on the board.

A piece value can be static, a predetermined value that does not change over time, or dynamic, a value that changes over time. In Stratego the importance of the pieces changes over time. For example, the Spy is important when the opponent still has its Marshal in play. When the Marshal is removed from the board the Spy is only useful to capture the Flag or sacrifice the piece to discover the identity of an unknown piece. Therefore the Spy will lose a large deal of its value because it cannot use its special ability anymore, capturing the Marshal.

De Boer (2007) suggested a formula for the relative values of pieces in Stratego:

- The Flag has the highest value of all
- The Spy's value is half of the value of the Marshal
- If the opponent has the Spy in play, the value of the Marshal is multiplied with 0.8
- When the player has less than three Miners, the value of the Miners is multiplied with $4 - \#left$.
- The same holds for the Scouts
- Bombs have half the value of the strongest piece of the opponent on the board
- The value of every piece is incremented with $\frac{1}{\#left}$ to increase their importance of the piece when only a few are left

In the situation when the opponent has a Sergeant as highest rank, it does not matter for the player if he has a General or Colonel as highest rank, because they both defeat all the pieces left of the opponent. The value of General or Colonel should be relative to the number of pieces they can capture. In Figure 5.1 such situation is depicted. The blue General (the "2") is the highest ranked piece on the board. However it would as just be valuable if it was, for example, a Marshal or Major, because it would still defeat the same number pieces.

Similar is a situation when the opponent would lose his Marshal and the player's General becomes the highest rank on the game board. Initially the General would have a low value because it can be defeated

by the Marshal. When the Marshal is removed from the game board the General is undefeatable and its value needs to be changed accordingly.

A player loses if his Flag is captured and therefore the piece will be assigned a static terminal value which is larger than the bounds of the evaluation function to ensure a cutoff in case of a win or loss.

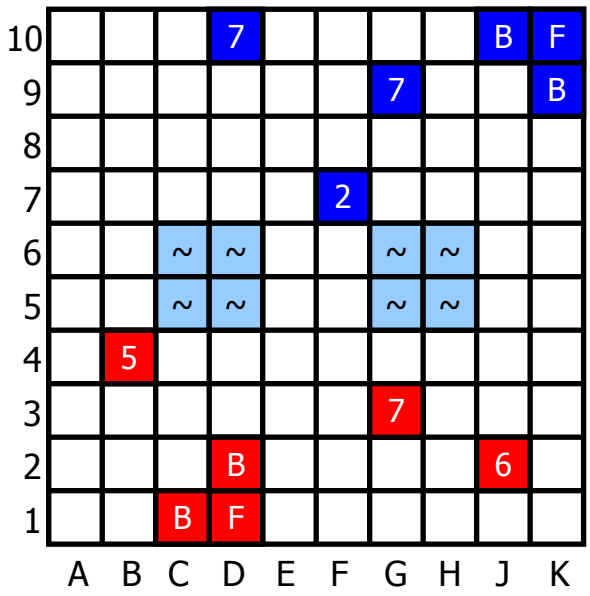


Figure 5.1: Example where the rank of the highest piece is not important

5.2 Value of Information

Initially the pieces of the opponent are unknown to the player and the other way around. Most Stratego players will try to keep the identity of their pieces unknown as long as possible (De Boer, 2007). With imperfect information the players can mislead the opponent to make a bad move. This prevents the opponent to play the best move possible. E.g. if there is a situation where the opponent has an unknown Marshal and a few low-ranked unknown pieces left. When the opponent knows which piece is the Marshal, his General could safely attack the remaining unknown pieces.

In Figure 5.2 the Red player has two unknown pieces left, one is the Marshal and the other piece is a Miner. The Blue player is now in the situation where he can attack both pieces. When the Blue player uses the Spy to attack and it shows that the attacked piece is the Marshal, the game will result in a definite win for Blue. However, if the blue Spy attacks the Miner, the Blue player loses his Spy and the Red player can easily capture one of the opponent's Miners. This results in a clear loss for Blue.

5.3 Strategic Positions

The game board of Stratego contains three small areas with a size of two by two connecting both sides of the game board, separated by water, these are called *lanes*. Every piece going to the opposite side needs to pass these lanes. In some situations it is beneficial to control these lanes, preventing the opponent to cross the board. Especially in the end game, when Miners need to remove the Bombs surrounding the Flag. A player can defend these lanes with only one piece per lane and preventing a loss. In Figure 5.3 the Red player is outnumbered and can only hope for a draw. The Red player is defending the lanes such that the Blue player cannot use its Miners to approach the Flag. Therefore the player forces a draw.

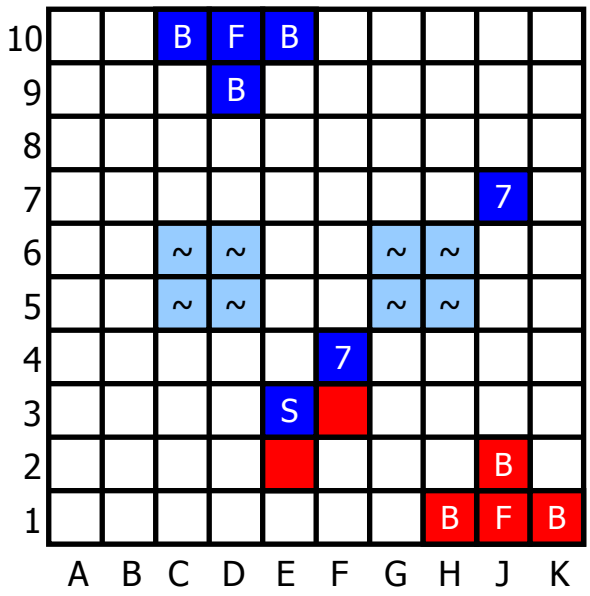


Figure 5.2: Example where the Red player has a Miner and the Marshal left. If the Blue player attacks the Marshal with the Spy he wins, otherwise the Red player wins

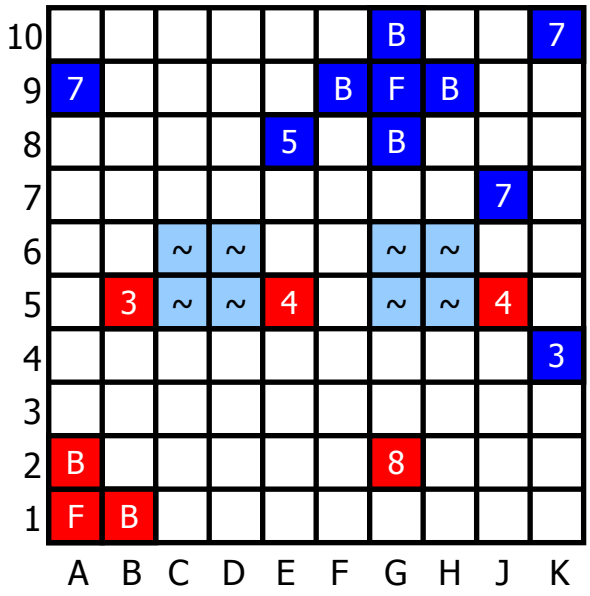


Figure 5.3: Example where the Red player controls all three lanes

5.4 Implementation

The key points mentioned in the previous section have been implemented as different independent features in an evaluation function.

- **First feature** is multiplying the value of the Marshal (both player and opponent) with 0.8 if the opponent has a Spy on the game board.
- **Second feature** multiplies the value of the Miners with $4 - \#left$ if the number of Miners is less than three.
- **Third feature** sets the value of the Bomb to half the value of the piece with the highest value.
- **Fourth feature** sets divides the value of the Marshal by two if the opponent has a Spy on the board.
- **Fifth feature** gives a penalty to pieces that are known to the opponent
- **Sixth feature** increases the value of a piece when the player has a more pieces of the same type than the opponent.

Eventually the value of a piece is multiplied with the number of times that the piece is on the board, and summated over all the pieces. The default values of each piece can be found in Table 5.1. These values are based upon the M.Sc. thesis by Stengård (2006).

Marshal	400	Sergeant	15
General	200	Miner	25
Colonel	100	Scout	30
Major	75	Spy	200
Lieutenant	50	Bomb	20
Captain	25	Flag	10000

Table 5.1: Default values of the evaluation function for each piece

Chapter 6

Experimental Results

This chapter presents the experiments, and their results, done in this research. Section 6.1 describes the experimental design, Section 6.2 shows the results of using different pruning techniques in MIN/MAX nodes. Section 6.3 measures the performance of the *-MINIMAX algorithms from Ballard (1983) and the TRANSPOSITION TABLE enhancement by Veness (2006). Section 6.4 gives an overview of the performance of evaluation function features. Finally Section 6.5 presents the experiments and results of MULTI-CUT.

6.1 Experimental Design

The implemented techniques are tested using two approaches. The first approach is the testing the performance of the technique with respect to the number of nodes visited. The test set consists of 300 board positions, where the first 100 board positions are taken from the begin game of Stratego, 100 midgame positions and 100 positions are endgame positions. See Figure 6.1 for example board configurations.

The second approach is using self play, where two programs play a match against each other. Here 11 predefined setups (see Appendix A), from which each player is appointed a board setup, are used.

6.2 Pruning in MIN/MAX Nodes

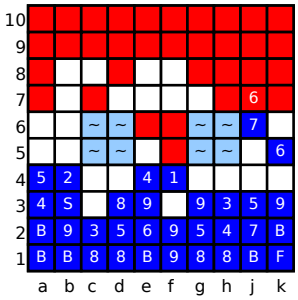
This section focusses on pruning in deterministic nodes. Section 6.2.1 shows the results of ITERATIVE DEEPENING. The effect of the HISTORY HEURISTIC is shown in Section 6.2.2. The results of the TRANSPOSITION TABLE is described in Section 6.2.3 and Section 6.2.4 shows the result of combination of the HISTORY HEURISTIC and the TRANSPOSITION TABLE. The techniques are compared to a default program using α-β pruning by default or if not mentioned otherwise. For computational reasons, all experiments are performed up to 5 ply.

6.2.1 Iterative Deepening

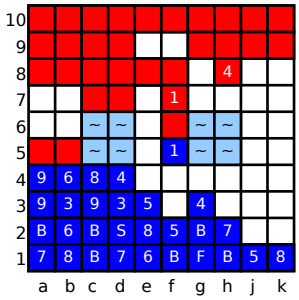
Table 6.1 shows that ITERATIVE DEEPENING (ID) without enhancements provides a little overhead, 4.10% on a 5-ply search, when compared to a fixed depth search. However, the advantage of ITERATIVE DEEPENING is that it can be stopped anytime during the search at depth n and then is able to return the best move on a $n-1$ depth search.

Ply	Without ID	With ID	Overhead
1	10,344	10,344	0.00%
2	264,264	274,608	3.91%
3	5,994,388	6,268,996	4.58%
4	215,558,850	221,827,846	2.91%
5	5,408,948,866	5,630,776,712	4.10%

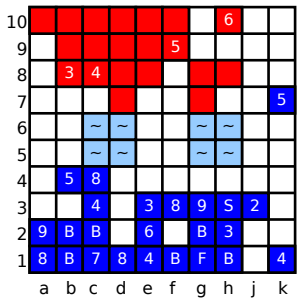
Table 6.1: Overhead of ITERATIVE DEEPENING



(a) Begin game position



(b) Midgame position



(c) Endgame position

Figure 6.1: Example positions used in the test set

6.2.2 History Heuristic

The History Heuristic (HH) sorts the nodes dynamically based upon previous search results. Stratego is a game with an average branching factor of around 30, where some moves lead to already seen board configurations and some moves do not result in progression of the game. Static move ordering is used to make an initial ordering based on domain knowledge (see Section 4.9.1. The dynamic move ordering improves the static move ordering.

The History Heuristic improves the performance of Iterative Deepening as shown in Table 6.2. The overhead of Iterative Deepening is still noticeable at ply 2 where the overhead is 3.26%. The gain of the dynamic move ordering in combination with Iterative Deepening begins at ply 3. The History Heuristic improves the node reduction up to 55.46% on a 5-ply search.

Ply	Without HH & without ID	With HH & with ID	Gain
1	10,344	10,344	0.00%
2	264,264	273,141	-3.36%
3	5,994,388	5,322,698	11.21%
4	215,558,850	141,060,175	34.56%
5	5,408,948,866	2,409,129,708	55.46%

Table 6.2: Node reduction by History Heuristic

The table reveals that the dynamic move ordering has a positive effect on top of the static ordering as described in Section 4.9. The performance of the History Heuristic increases every ply. By looking at the trend of the gain we can assume that the performance of the History Heuristic improves even more when searching deeper than 5 ply.

6.2.3 Transposition Table

The Transposition Table (TT) is able to reduce the number of nodes by keeping track of previous visited board configurations, the corresponding upper bound, lower bound and the best move. Every time an equal board configuration appears, the program can immediately retrieve the previously calculated upper and lower bound. These values are compared to the current α and β values and a cutoff can occur. If these values do not provide a cutoff, the Transposition Table can provide a best move which is searched first. In Table 6.3 shows the gain of Transposition Tables in combination with Iterative Deepening.

Ply	Without TT & without ID	With TT & with ID	Gain
1	10,344	10,344	0.00%
2	264,264	227,938	13.75%
3	5,994,388	5,001,828	16.56%
4	215,558,850	134,654,093	37.53%
5	5,408,948,866	3,375,802,619	37.59%

Table 6.3: Node reduction by a Transposition Table

The performance is already noticeable on a 2-ply search. This can be explained by the advantage of move ordering in combination with Iterative Deepening. The move ordering at the second iteration uses the information from the first iteration and is already able to prune 13.75%. By looking at the trend of the gain we can assume that the performance of the Transposition Table improves even more when searching deeper than 5 ply.

6.2.4 History Heuristic and Transposition Table

In the next experiment the Transposition Table and the History Heuristic are combined. These techniques combine their strength to perform better. The combination of Transposition Tables, the History Heuristic and Iterative Deepening is able to reduce the number of nodes by 74.06% as shown in Table 6.4.

Ply	Without TT & without HH & without ID	With TT & with HH & with ID	Gain
1	10,344	10,344	0.00%
2	264,264	224,387	15.09%
3	5,994,388	4,123,300	31.21%
4	215,558,850	87,384,008	59.46%
5	5,408,948,866	1,403,260,491	74.06%

Table 6.4: Node reduction by HH & TT

Table 6.5 shows the strength of Iterative Deepening. A 5-ply search with Iterative Deepening improves the node reduction by 32.65% when compared without Iterative Deepening. Iterative Deepening successfully uses information from lower depth searches to improve the move ordering and hence reduces the number of nodes. The effectiveness of Iterative Deepening with move ordering enhancements increases every ply. Again we may assume that the gain increases further when the search depth grows.

6.3 Pruning in Chance Nodes

Section 6.3.1 looks at the result of the Star1 algorithm. In Section 6.3.2 the result of Star2 with and without Star1 are shown. The results of the StarETC algorithm is shown in Section 6.3.3 The experiments with *-minimax make use of Iterative Deepening, the History Heuristic and the Transposition Table.

Ply	With TT & with HH & without ID	With TT & with HH & with ID	Gain
1	10,344	10,344	0.00%
2	262,796	224,387	14.62%
3	4,658,616	4,123,300	11.49%
4	100,656,880	87,384,008	13.19%
5	2,083,669,650	1,403,260,491	32.65%

Table 6.5: Node reduction using HH & TT with Iterative Deepening and without Iterative Deepening

6.3.1 Star1

The Star1 algorithm is able to backward prune chance nodes using an α-β window and information about the theoretical lower and upper bound of the evaluation function.

The Star1 algorithm is able to prune 32.54% at the first ply as can be seen in Table 6.6. This can be explained by the fact that a chance node is not a counted as a ply but the children of the chance node are. Therefore the Star1 algorithm is able to prune at a 1-ply search. On a 5-ply search Star1 reduces the node count by 90.13%.

Ply	Default	With Star1	Gain
1	10,344	6,978	32.54%
2	264,264	188,423	28.70%
3	5,994,388	1,607,955	73.18%
4	215,558,850	61,256,939	71.58%
5	5,408,948,866	533,925,113	90.13%

Table 6.6: Node reduction by Star1

6.3.2 Star2

Star2 probes 1 or more children of a node to tighten the bounds of a chance node. This also helps Star1 to prune. With a good move ordering Star2 can be quite effective. Table 6.7 presents the results of the Star2 algorithm with a probe factor of 1. At depth 2 and 3 Star2 has no significant effect on the search. The reason for this result is that Star2 probes children of nodes which do not give good information for Star2 to adjust the bounds and hence only increase the number of nodes. However, on a 5-ply search Star2 is able to prune 93.56%.

Ply	Default	With Star2 & with Star1	Gain
1	10,344	6,978	32.54%
2	264,264	190,389	27.95%
3	5,994,388	1,650,350	72.47%
4	215,558,850	36,888,491	82.89%
5	5,408,948,866	348,235,823	93.56%

Table 6.7: Node reduction using Star2 with a probing factor of 1

In Table 6.8 the Star1 and Star2 algorithm are compared. For a 1-ply search Star2 is not useable. Therefore Star2 does not provide any gain compared to Star1. The results show the overhead of Star2 on a shallow search. It also shows the gain compared to Star1 on a deeper search. The gain increases to 39.78% on a 4-ply search but drops to 34.78. A 6 ply search is likely to improve the performance of Star2 again. The Star2 algorithm is able to probe a different number of children in every node. By default Star2 probes 1 child per chance node. Depending on the quality of the move ordering, the gain increases or decreases by incrementing the probe factor.

Ply	Without Star2 & with Star1	With Star2 & with Star1	Gain
1	6,978	6,978	0.00%
2	188,423	190,389	-1.04%
3	1,607,955	1,650,350	-2.64%
4	61,256,939	36,888,491	39.78%
5	533,925,113	348,235,823	34.78%

Table 6.8: Node reduction by STAR1 and STAR2 with a probe factor of 1

In Table 6.9 we compare different probe factors. The first probe factor of 0 equals the STAR1 algorithm and has therefore no gain. The values 1 and 2 have a positive gain when compared to STAR1, 34.78% and 16.44% respectively. Higher values for the probe factor increase the node count up to 35.59%. A probe factor of 1 has the best reduction of nodes, this means that the move ordering (the HISTORY HEURISTIC and TRANSPOSITION TABLE) performs well in combination with STAR2.

Probe factor	Nodes	Gain
(Star1) 0	533,925,113	-
1	348,235,823	34.78%
2	446,157,703	16.44%
3	543,629,215	-1.82%
4	628,570,608	-17.73%
5	723,955,888	-35.59%

Table 6.9: Node reduction using different probe factor values for STAR2

6.3.3 StarETC

The STARETC algorithm makes use of information from chance nodes. STARETC stores the lower and upper bounds computed by the STAR1 and STAR2 algorithms, as similar as the TRANSPOSITION TABLE. STARETC is able to reduce the number of nodes by 93.52% as can be seen in Table 6.10.

Ply	Default	With StarETC	Gain
1	10,344	6,978	32.54%
2	264,264	190,389	27.95%
3	5,994,388	1,649,015	72.49%
4	215,558,850	36,796,717	82.93%
5	5,408,948,866	350,754,377	93.52%

Table 6.10: Node reduction using STARETC

It performs almost identical like STAR2 as can be seen in Table 6.11. The relative gain of STARETC compared to STAR2 is almost 0. On a 5-ply search there seems to be a little additional overhead when compared to STAR2. The STARETC is likely to perform better than STAR2 on a deeper search.

Overview

The results of the node reduction experiments are summarized in Table 6.12. The table compares the techniques on a 5-ply search. Each technique makes use of ITERATIVE DEEPENING. Every new experiment is extended with the previous mentioned techniques. The gain of the mean can be found in the tables previously mentioned. Worth noting is the fact that each technique improves the worst-case.

The STAR1 and STAR2 improve the node reduction compared to α-β pruning with the HISTORY HEURISTIC and TRANSPOSITION TABLE with an additional 75.18%.

Ply	Star2	StarETC	Gain
1	6,978	6,978	0.00%
2	190,389	190,389	0.00%
3	1,650,350	1,649,015	0.08%
4	36,888,491	36,796,717	0.25%
5	348,235,823	350,754,377	-0.72%

Table 6.11: Node reduction using StarETC

	Total	Mean	Median	Std. Dev.	Min	Max
Alpha-Beta	5,630,776,712	18,769,256	2,243,814.5	53,356,557.85	34	559,492,043
History Heuristic	2,409,129,708	8,030,432	716,116.0	24,486,486.38	34	229,001,460
Transposition Table	1,403,260,491	4,677,535	573,530.5	14,574,204.75	34	127,506,016
Star1	533,925,113	1,779,750	294,901.0	5,144,435.40	31	45,979,029
Star2	348,235,823	1,160,786	230,650.5	3,640,867.21	31	34,938,971
StarETC	350,754,377	1,169,181	231,281.0	3,670,205.21	31	33,607,267

Table 6.12: Number of nodes searched on 5-ply search

6.4 Evaluation Function Features

This section looks at the performance of the features as presented in Section 5.4. First, the node reduction is compared and second, the strength of the evaluation features is compared to the default evaluation function. The search function is extend with all previous mentioned techniques: Iterative Deepening, History Heuristic, Transposition Table, Star1, Star2 and StarETC.

In self play the program will play 121 games of Stratego. Each program has a search time of one second to compute a move. There are 11 predefined board setups, which can be found in Appendix B. Every game starts with a unique combination of two setups (therefore results in a total of $11^2 = 121$ games). The game ends by the game rules or after 1,200 plies.

First, we compare the node reduction of the different features compared to the default version in Table 6.13. It shows that the third feature can reduce the nodes visited by the current evaluation function with 7.84%. The fifth feature decreases the gain by -7.92%. The other features have no significant increase or decrease impact on the performance.

Extension	Nodes	Gain
None	350,754,377	-
1st feature	344,586,412	1.76%
2nd feature	351,781,359	-0.29%
3rd feature	323,241,276	7.84%
4th feature	350,270,127	0.14%
5th feature	378,522,770	-7.92%
6th feature	344,157,660	1.88%

Table 6.13: Node reduction using evaluation function features

Second, we take a look at the performance of these features in self play. For evaluating the features, every feature has been tested individually against a program without any features. The results are shown in Table 6.14. The column 'Score percentage' is the sum of the win percentage and half of the draw percentage. The features do not provide much improvement as compared to the default evaluation function. The second feature, where the weight of the Miners increases as their number on the board decreases, has the best performance with a 44.44% win rate and a 33.33 % loss rate. With the current number of tests it is not possible to state that this feature is significant better than the default evaluation function.

Extension	Win	Loss	Draw	Score percentage
1st feature	38.89 %	47.22%	13.89%	45.84%
2nd feature	44.44 %	33.33%	22.22%	55.55%
3rd feature	36.11 %	47.22%	16.67%	44.45%
4th feature	27.78 %	52.78%	19.44%	37.50%
5th feature	25.00 %	38.89%	36.11%	43.06%
6th feature	25.00 %	52.78%	22.22%	36.11%

Table 6.14: Comparison of different evaluation function features

6.5 Multi-cut

In this section the program plays 121 games similar as described in Section 6.4. Again the search time has been set to one second. In this experiment the search function, with all pruning techniques active and without any function features, is extended using MULTI-CUT.

M is the maximum number of moves that MULTI-CUT considers, C is the number of moves that result in a cut-off before MULTI-CUT prunes the complete subtree. R is the search depth reduction.

M	C	R	Wins	Loss	Draw	Score percentage
8	2	2	38.84%	42.15%	19.01%	48.36%
8	3	2	45.45%	40.15%	14.05%	52.48%
8	4	2	36.36%	42.98%	20.66%	46.69%
10	2	2	42.15%	35.54%	22.31%	53.31%
10	3	2	45.45%	39.67%	14.88%	52.89%
10	4	2	37.19%	42.98%	19.83%	47.11%
12	2	2	40.50%	39.67%	19.83%	50.42%
12	3	2	41.32%	37.19%	21.49%	52.07%
12	4	2	41.32%	41.32%	17.36%	50.00%
8	3	3	42.98%	40.40%	16.53%	51.25%
10	3	3	37.19%	42.98%	19.83%	47.11%
12	3	3	41.32%	40.50%	18.18%	50.41%

Table 6.15: Self play results using MULTI-CUT

In Table 6.5 the results are shown. The column 'Score percentage' is the sum of the win percentage and half of the draw percentage. MULTI-CUT shows some improvement over the default search method, but not significantly enough to state that MULTI-CUT is an essential improvement. The best performing parameters (M,C,R) are (*8,3,2*),(*10,2,2*),(*10,3,2*),(*12,3,2*).

Chapter 7

Conclusions

In this chapter the Research Questions and the Problem Statement, as stated in Chapter 1 are answered based upon the research done in this thesis. Section 7.1 gives answers to the Research Questions, Section 7.2 and future research ideas are given in Section 7.3

7.1 Answering the Research Questions

The first Research Question that was stated is:

Research Question 1. *What is the complexity of Stratego?*

The complexity of the state-space and the game-tree complexity of Stratego have been calculated in Chapter 3. First the complexity of the board setup is 10^{23}. Second, the calculated upperbound of the state-space complexity is 10^{115} and the game-tree complexity equals 10^{534}. The state-space complexity is an upperbound because some positions are calculated that would never be reached in practice. The game-tree complexity is calculated based upon human played Stratego games. The complexity of Stratego is one of the highest complexities in board games (See Table 3.1). Based upon these numbers it is not likely that Stratego will be solved in the near future and that a search-based approach with an evaluation function can be used.

The second Research Question was:

Research Question 2. *To what extent can *-minimax techniques be utilized in order to realize competitive game play in Stratego?*

EXPECTIMAX with the HISTORY HEURISTIC, α-β and TRANSPOSITION TABLES is able to prune 74.06%. Additionally, STAR1 and STAR2 give a node reduction of 75.18%. The *-MINIMAX algorithms get their advantage from pruning in chance nodes. Chapter 3 shows that Stratego has a relative low number of chance nodes. However an improvement in performance is quite drastic. The performance gain gives the artificial player a possibility to search deeper than an computer player without a *-MINIMAX algorithm.

7.2 Answering the Problem Statement

After answering our Research Questions it is possible to answer the Problem Statement:

Problem Statement. *How can we develop informed-search methods in such a way that programs significantly improve their performance in Stratego?*

The answer to Research Question 1 shows that Stratego is a complex game. Therefore, it is not possible to solve the game. The program will have to use informed-search techniques to be able to play Stratego.

The answer to Research Question 2 shows that informed-search methods decrease the number of nodes that need to be visited compared to EXPECTIMAX. This allows the player to search deeper in a given time frame and thus increase his potential to better evaluate search branches. However because of the relative high complexity, intermediate or even expert level game play is hard to realize.

7.3 Future Research

Most research in this thesis is based upon well-known techniques such as, EXPECTIMAX and α-β PRUNING. Although the *-MINIMAX algorithms have been developed 20 years ago there is still little research available on this topic. More research in different areas is needed to increase the further understanding of these algorithms. Perhaps it is possible to develop new techniques based upon the current research.

There is little research in the field of Stratego itself, the current papers available do not provide enough scientific insight into Stratego. Especially Stratego is a very complex game where humans still have the upper hand in playing. This gives insight into the fact that humans play games very differently than artificial players. For future research one could think of a different approach of search. Moreover more research could be done in the evaluation function for Stratego. Not only finding better weights and features but perhaps one could take into account a short-term plan or the structure in the game.

References

Allis, L.V. (1988). A knowledge-based approach to connect-four. The game is solved: White wins. M.Sc. thesis, Vrije Universiteit Amsterdam, The Netherlands. [1]

Allis, L.V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, Rijksuniversiteit Limburg, The Netherlands. [2, 9]

Ballard, B. W. (1983). The *-minimax search procedure for trees containing chance nodes. *Artificial Intelligence*, Vol. 21, No. 3, pp. 327–350. Ballard, B. [2, 13, 19, 23, 28, 39]

Björnsson, Y and Marsland, T.A. (2001). Multi-cut alpha-beta-pruning in game-tree search. *Theoretical Computer Science*, Vol. 252, Nos. 1–2, pp. 177–196. [32, 33]

Boer, V. de (2007). Invicible. A Stratego Bot. M.Sc. thesis, Technische Universiteit Delft, The Netherlands. [vii, ix, 2, 35, 36]

Breuker, D.M. (1998). *Memory versus search in games*. Ph.D. thesis, Universiteit Maastricht, The Netherlands. [26, 32]

Breuker, D.M., Uiterwijk, J.W.H.M., and Herik, H.J. vd (1994). Replacement Schemes for Transposition Tables. *ICCA Journal*, Vol. 17, pp. 183–193. [28]

Ciancarini, P. and Favini, G. P. (2007). A Program to Play Kriegspiel. *ICGA Journal*, Vol. 30, No. 1, pp. 3–24. [2]

Gravon (2009). Project Gravon. http://www.gravon.de. [10]

Greenblatt, D., Eastlake, D.E., and Crocker, S.D. (1967). The Greenblatt Chess Program. *Chess Skills in Man and Machine*, pp. 82–118, ACM, New York, NY, USA. [26]

Hasbro (2002). Stratego Rules. [5]

Hauk, T. G. (2004). Search in Trees with Chance Nodes. M.Sc. thesis, University of Alberta, Canada. [18, 24]

Hauk, T., Buro, M., and Schaeffer, J. (2004). Rediscovering *-Minimax Search. *Computers and Games* (eds. H. J. van den Herik, Y. Björnsson, and N. S. Netanyahu), Vol. 3846 of *Lecture Notes in Computer Science*, pp. 35–50, Springer. [2]

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. ISSN 0004–3702. [1, 2, 10]

Hsu, F.-H. (2002). *Behind Deep Blue: Building the Computer that Defeated the World Chess Champion*. Princeton University Press, Princeton, NJ, USA. ISBN 0691090653. [1]

International Stratego Federation (2009). Stratego Game Rules. http://www.isfstratego.com/images/isfgamerules.pdf. [6]

Ismail, M. (2004). Multi-agent stratego. B.Sc thesis, Rotterdam University, The Netherlands. [2]

Knuth, D.E. and Moore, R.W. (1975). An Analysis of Alpha Beta Pruning. *Artificial Intelligence*, Vol. 6, No. 4, pp. 293–326. [15, 17]

Michie, D. (1966). Game-Playing and Game-Learning Automata. *Advances in Programming and Non-Numerical Computation*, pp. 183–200. [2, 17]

Mohnen, J. (2009). Using Domain-Dependent Knowledge in Stratego. B.Sc thesis, Maastricht University, The Netherlands. [2]

Neumann, J. von and Morgenstern, O. (1944). *Theory of Games and Economic Behavorial Research*. Princeton University Press, Princeton, New Yersey, USA, first edition. [13]

Russell, S. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ. [14, 15]

Schadd, M. P. D. and Winands, M. H. M. (2009). Quiescence Search for Stratego. *BNAIC 2009* (eds. T. Calders, K. Tuyls, and M. Pechenizkiy), pp. 225–232, Eindhoven, The Netherlands. [2]

Schadd, M.P.D., Winands, M.H.M., Uiterwijk, J.W.H.M., Herik, H.J. van den, and Bergsma, M.H.J. (2008). Best Play in Fanorona leads to Draw. *New Mathematics and Natural Computation*, Vol. 4, No. 3, pp. 369–387. [1]

Schadd, M. P. D., Winands, M. H. M., and Uiterwijk, J. W. H. M. (2009). ChanceProbCut: Forward Pruning in Chance Nodes. *IEEE Symposium on Computational Intelligence and Games (CIG 2009)* (ed. P. L. Lanzi), pp. 178–185. [2]

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Trans. Pattern Anal. Mach. Intell.*, Vol. 11, No. 11, pp. 1203–1212. ISSN 0162–8828. [29, 32]

Schaeffer, J. (2007). Checkers Is Solved. *Science*, Vol. 317, No. 5844, pp. 1518–1522. [1]

Shannon, C. E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, Vol. 41, No. 1, pp. 256–275. [1]

Slate, J.D. and Atkin, L.R. (1977). CHESS 4.5: The Northwestern University Chess program. *Chess Skill in Man and Machine* (ed. P.W. Frey), pp. 82–118, Springer-Verlag, New York, USA. [26]

Stankiewicz, J.A. (2009). Opponent Modeling in Stratego. B.Sc thesis, Maastricht University, The Netherlands. [2]

Stankiewicz, J.A. and Schadd, M. P. D. (2009). Opponent Modeling in Stratego. *BNAIC 2009* (eds. T. Calders, K. Tuyls, and M. Pechenizkiy), pp. 233–240, Eindhoven, The Netherlands. [2]

Stengård, K. (2006). Utveckling av minimax-baserad agent för strategispelet Stratego (in Swedish). M.Sc. thesis, Lund University, Sweden. [2, 38]

Treijtel, C. (2000). Multi-Agent Stratego. M.Sc. thesis, Delft University of Technology, The Netherlands. [2]

Turing, A. M. (1953). Chess. *Faster than Thought: A Symposium on Digital Computing Machines* (ed. B. V. Bowden), pp. 286–295. Pitman, London, UK. [1]

United States Court, District of Oregon (2005). Case No. 04-1344-KI, Estate of Gunter Sigmund Elkan vs. Hasbro Inc. [1]

USA Stratego Federation (2009). Computer Stratego World Championship. http://www.strategousa.org. [1, 2]

Veness, J. (2006). Expectimax Enhancements for Stochastic Game Players. M.Sc. thesis, University of New South Wales, Australia. [28, 39]

Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J. W. H. M., and Werf, E. C. D. van der (2005). Enhanced forward pruning. *Information Science*, Vol. 175, No. 4, pp. 315–329. [33]

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. *Computers and Games, Lecture Notes in Computer Science*, pp. 262–272. [32]

Xia, Z., Zhu, Y., and Lu, H. (2007). Using the Loopy Belief Propagation in Siguo. *ICGA Journal*, Vol. 30, No. 4, pp. 209–220. [2]

Zobrist, A.L. (1970). A New Hashing Method with Application for Game Playing. Technical report, Computer Science Department, University of Wisconsin, Madison, WI, USA. [26]

Appendix A

Board Setups

The board set-ups are used in the self play experiments of 6.

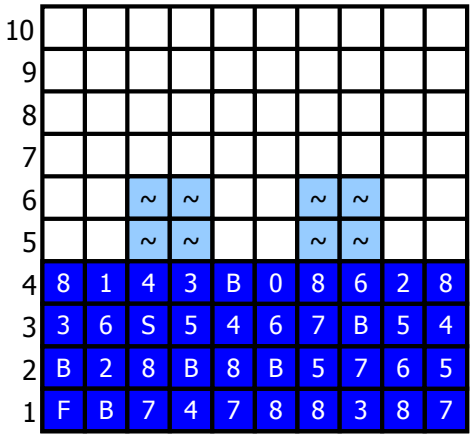


Figure A.1: Board set-up #1

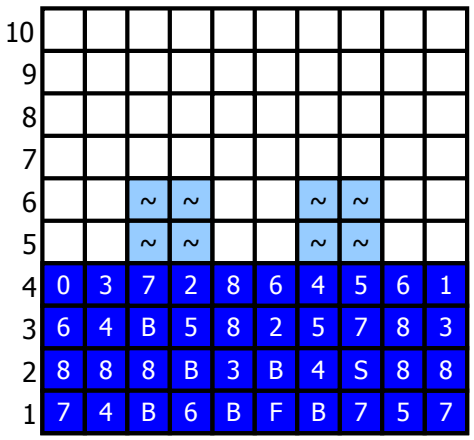


Figure A.2: Board set-up #2

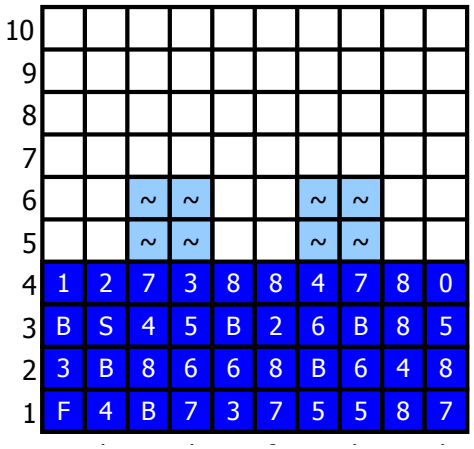


Figure A.3: Board set-up #3

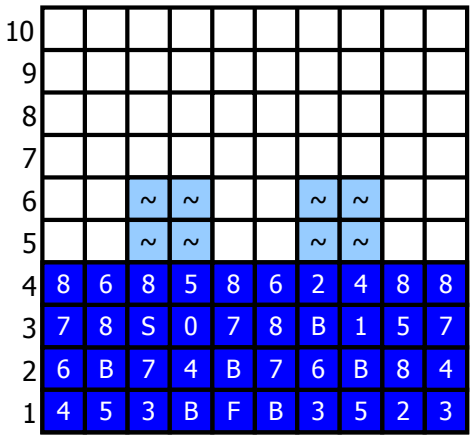


Figure A.4: Board set-up #4

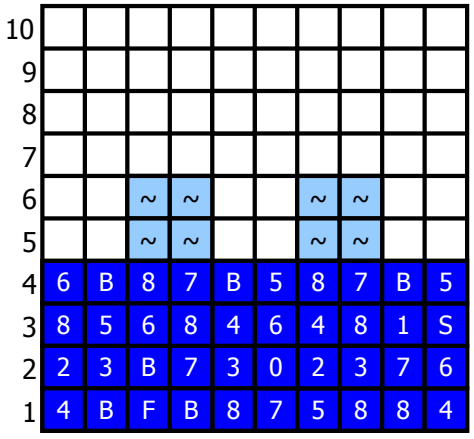


Figure A.5: Board set-up #5

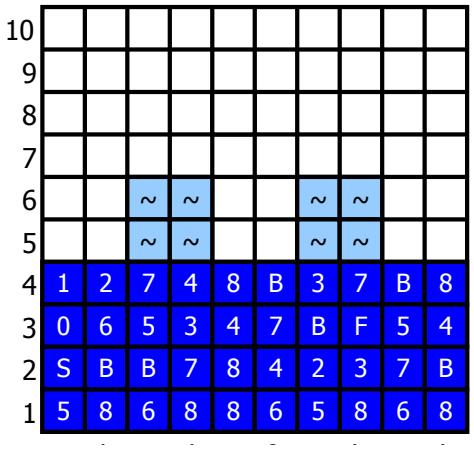


Figure A.6: Board set-up #6

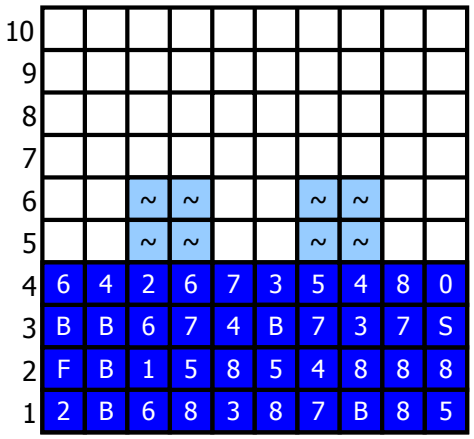


Figure A.7: Board set-up #7

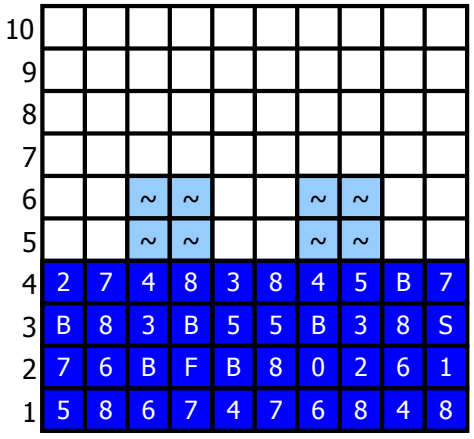


Figure A.8: Board set-up #8

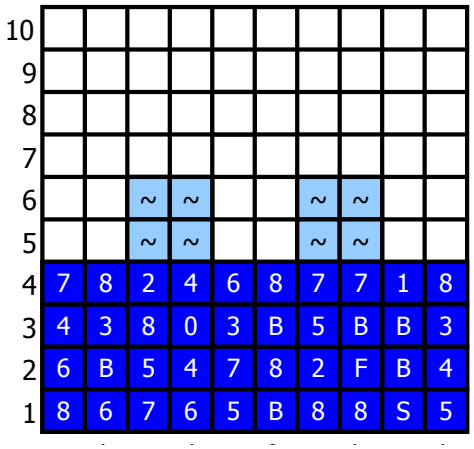


Figure A.9: Board set-up #9

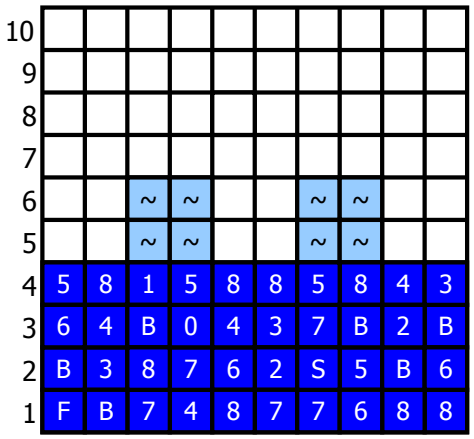


Figure A.10: Board set-up #10

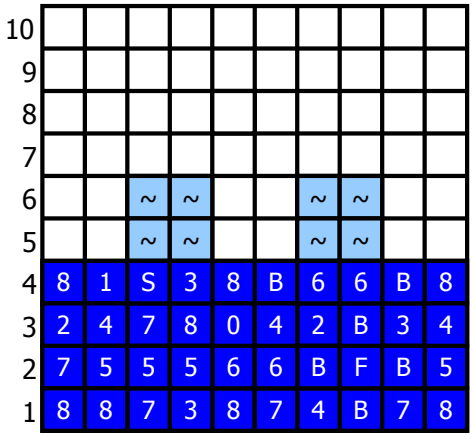


Figure A.11: Board set-up #11