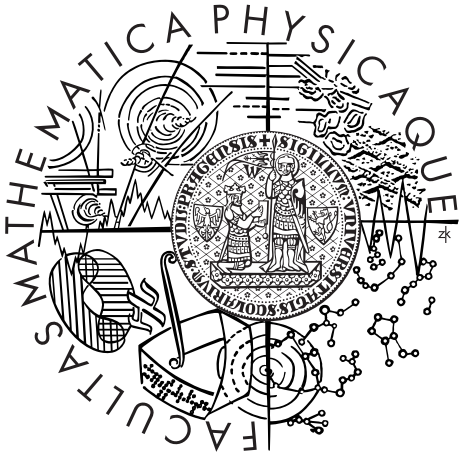
Charles University in Prague
Faculty of Mathematics and Physics

SVOČ 2013



Pavel Veselý

Artificial intelligence in abstract 2-player games

Department of Applied Mathematics

Supervisor of the thesis: RNDr. Tomáš Valla, Ph.D.

Prague 2013

Abstract

Tzaar is an abstract strategy two-player game, which has recently gained popularity in the gaming community and has won several awards. There are some properties, most notably the high branching factor, that make Tzaar hard for computers. We developed Waltz, a strong Tzaar-playing program, using enhanced variants of Alpha-beta and Proof-number Search based algorithms. After many tests with computer opponents and a year of deployment on a popular board-gaming portal, we conclude that Waltz can defeat all available computer programs and even strong human players. In this thesis we describe Waltz and its performance. Also, as a general contribution, we propose some enhancements of Alpha-beta and Proof-number Search, that can be useful in other domains than Tzaar.

This work originated as author's bachelor thesis and continued as a paper written with Tomáš Valla and submitted to the 8th international conference on Computer and Games (CG2013).

Contents

Introduction

From the beginning of the computer era people are working on game playing programs. The first challenge was to create a program that can win against the top human in Chess. In 1996 Garry Kasparov, being currently the best Chess player, lost a game against the computer Deep Blue, but still won the whole match consisting of six games. Nowadays the best computer programs for Chess are much stronger than the best people.

In the meantime, algorithms in Chess programs were adapted to other games. They were not always successful against professional players, for example in the game Go, best human players are still much better than any program. For these games new algorithms were invented and the progress in this field is still going on.

Tzaar is a relatively new game, that was invented by Kris Burm and published in 2007. It is a part of the Project GIPF, a set of six abstract strategy two-player games on a hexagonal grid (not always the same). Some computer programs for Tzaar are available, but nobody has done a research on the game properties of Tzaar and on algorithms appropriate for Tzaar (up to my best knowledge). The game GIPF, the first in the Project GIPF, has already been studied, see [25].

Several properties, most notably the high branching factor (see Section 1.4), make Tzaar a hard game to play for computers. Therefore, together with the Tzaar popularity, writing a strong Tzaar playing program is a challenge. We address this challenge by developing Waltz,[1] a strong program able to defeat all other Tzaar programs that we are aware of, and also—which is more important—match up with and defeat even strong human players.

We have installed several playable "robots" on the popular board-gaming portal Boitejeaux.net [28], where some very strong players are playing. The details about Waltz performance against both computer and human opponents can be found in Chapter 4.

The algorithms employed in Waltz are based on Alpha-beta pruning and Proof-number Search (PNS), together with many enhancements, see Chapter 3.2 for more details. Both algorithms work on a *game tree* in which every node corresponds to a position and the root node is the position for which we want to find the best move. A node N is a son of a node P if and only if there is a move from the position in the node P to the position corresponding to the node N. Note that one position can be more than once in the tree.

We chose and tuned these algorithms and their enhancements after numerous statistical experiments and play-outs with other Tzaar playing programs, humans, and different versions of Waltz.

We also propose these new enhancements: Randomized Alpha-beta, Heuristic Weak PNS, and Alpha-beta and Depth-first PNS for lost positions (see Sections 2.1.3, 2.2.5 and 2.3), which we consider to be a general contribution of this thesis.

The thesis is organized as follows. In the first chapter we present the game Tzaar together with its properties, strategies and existing programs for playing Tzaar against an artificial intelligence. Chapter 2 covers appropriate algorithms for Tzaar in general way, namely Alpha-beta and Proof-number Search with their

[1] The name stands for the recursive acronym *Waltz ALgorithmic TZaar*.

enhancements. The next chapter describes their implementation in the domain of Tzaar with details of Tzaar specific heuristics. In the fourth chapter we test the algorithms experimentally and show how enhancements and parameters of the algorithms help making the search quicker, and how is Waltz successful against other programs and more importantly against people.

The source code of Waltz, the thesis, and other information can be downloaded from our website [23].

1. Tzaar Game

In this chapter we introduce rules of the game Tzaar and discuss its game properties. Popularity of Tzaar and some strategic notes on playing are mentioned. We also take a look at other programs for playing Tzaar against an artificial intelligence.

1.1 Tzaar Rules

Tzaar is a modern abstract strategy two-player game, bearing a distant similarity to Checkers in some sense. Abstract means that the game does not have a theme. It also has full information, i.e., both players know all information about the current position in the game, and there is no randomness. White player and black player take turns, white has the first turn.

The board for Tzaar is hexagonal and consists of 30 lines that makes 60 intersections. There is a missing intersection in the center of the board. In the starting position there are 30 white and 30 black pieces. Each color has pieces of three types: 6 are *Tzaars,* 9 are *Tzarras* and 15 are *Totts.* The pieces lie on the intersections of the lines. See Fig. 1.1 for illustration.

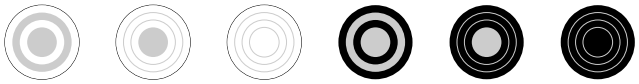


Figure 1.1: Tzaar pieces

The starting position has one piece at each intersection. The placement could be random or players can use a fixed starting position which is defined in the official rules [10] and shown in Figure 1.2.

Pieces can form *stacks,* that means, towers of pieces of the same color. In the beginning, all stacks on the board have height one. A stack is one entity, thus it cannot be divided into two stacks.

Each player's turn consists of two moves. There is an exception in the very first turn of the white player, as his turn consists only of the first move. The first move of each turn must be a *capture.* The player on turn moves one of his stacks along a line to an intersection with an opponent's stack. A stack cannot jump over other stacks and over the center of the board, only a jump over arbitrary number of empty intersections is allowed. No stack may end the jump on an empty intersection. Captured stack must have height at most the height of the capturing stack. Captured pieces leave the board.

The second move of a turn can be another capture move, or a stacking move, or a pass move. *Passing* means that the player on turn does not move with any stack, so nothing changes and the other player is on turn. During the *stacking move* the player jumps with his stack on some other stack of his color. The height of the resulting stack is the sum of both stacks heights. The type of the resulting stack is the same as the type of the stack which jumped, i.e., the type of the resulting stack is determined by the piece on the top, There is no restriction on

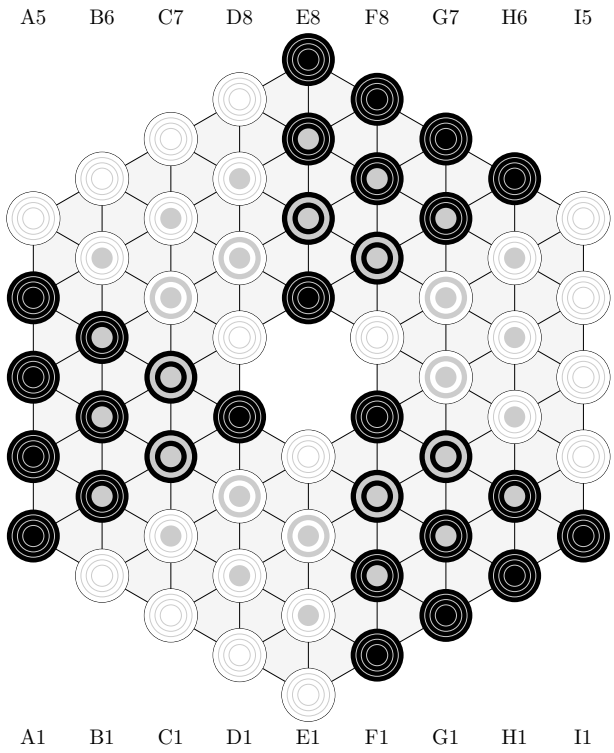


Figure 1.2: Fixed starting position

heights and types of stacks which can do a stack move, for example a Tzaar stack with height 1 can jump on a Tott stack with height 8.

A player loses when the last stack of one of the three types is captured, or if he cannot capture in the first move of his turn. However he can make his last possible capture and then pass or stack. Only visible pieces, i.e., pieces on the top of stacks, count for the presence of three types.

A draw is not possible in Tzaar. Quite surprisingly, according to the rules, a player can "commit suicide" (lose) by stacking his piece on the last stack of a piece type.

Tournament version of the game begins with an empty board. The players take turns in which they put a piece on an empty intersection on the board. The order of placing the pieces is arbitrary. After putting all pieces on the board, all intersections are occupied and the game starts like an usual game. In this thesis we deal only with games without this placement phase, i.e., not int the tournament version. The reasons are that the tournament version cannot be played on board-gaming portals like Boiteajeux.net [28], or BoardSpace.net [3] and that I was not able to think up a strategy, or evaluation function for Waltz that is not random.

1.2 Tzaar History

Tzaar, originally written as TZAAR, is a relatively new game. It was invented by Kris Burm and published in 2007. It is a part of the Project GIPF, a set of six abstract strategy two-player games. Tzaar replaced TAMSK, the second game of the project, and no other game was added to the project after Tzaar, since it has

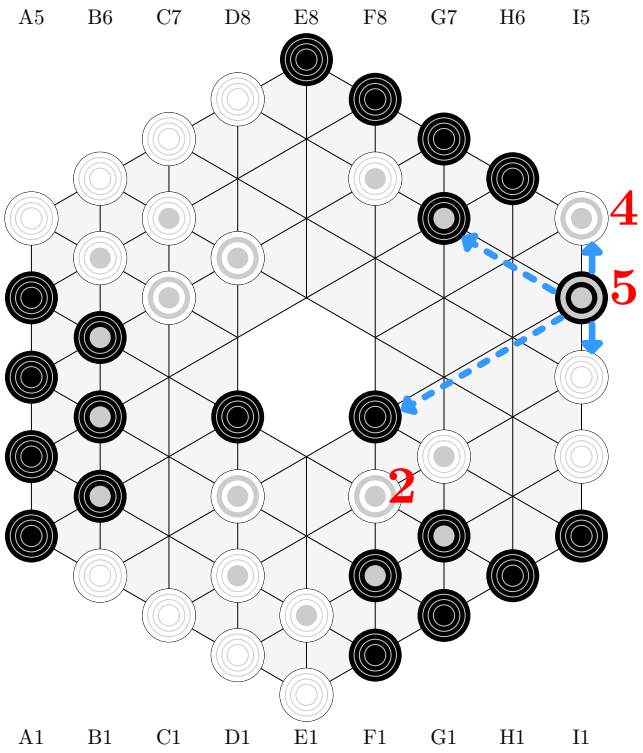


Figure 1.3: The Tzaar board with a sample position. The possible moves of the black Tzaar stack in the second move of a turn are marked by arrows, the dashed arrows represent stacking moves and the numbers denote the stack heights greater than one.

now six games as Kris Burm intended.

Despite being so young, Tzaar has won quite a lot of awards, most notably the Games Magazine's award "Game of the Year 2009" [29]. In 2008 it has been honoured "Spiel des Jahres" Recommendation [30] and earned nominated to "2008 International Gamers Award – General Strategy: Two-players" [2], "2008 Golden Geek Best 2-Player Board Game Nominee" [1] and "Nederlandse Spellenprijs 2008" [5].

Tzaar is also highly rated by the gaming community, for example on the popular server BoardGameGeek.com it has the second highest rating among abstract games (more than 1 300 users have voted up to March 31, 2013) [6].

1.3 Strategies

In this section we discuss some strategies how to play Tzaar. The strategies are based on playing with human and computer opponents (also other than Waltz) and there are some positions where they do not hold. The official Tzaar page has some strategy tips too [11].

In the second move of a turn a player has three possibilities: capture, stack, or pass. In most situations the stacking is the most reasonable, because it makes one of stacks more powerful and more safe against opponent's stacks. The other reason is that the opponent loses capturing possibilities, thus it is more likely that he will run out of captures and lose in the endgame.

Capturing again (so called *double capture move*) is appropriate if the opponent is running out of pieces of a type (he should not have a high stack of that type), or if a high stack can be captured. Height of a double captured stack had better be more than two, because by capturing stacks of size two, one can lose capturing possibilities. More importantly, double capturing two pieces with height only one leads mostly to to a loss because of no capturing possibilities.

Passing move occur only rarely during a game. It is worth doing only in the endgame when stacking is not possible, or would result in a loss. See Figure 1.3 for an example of such a position.

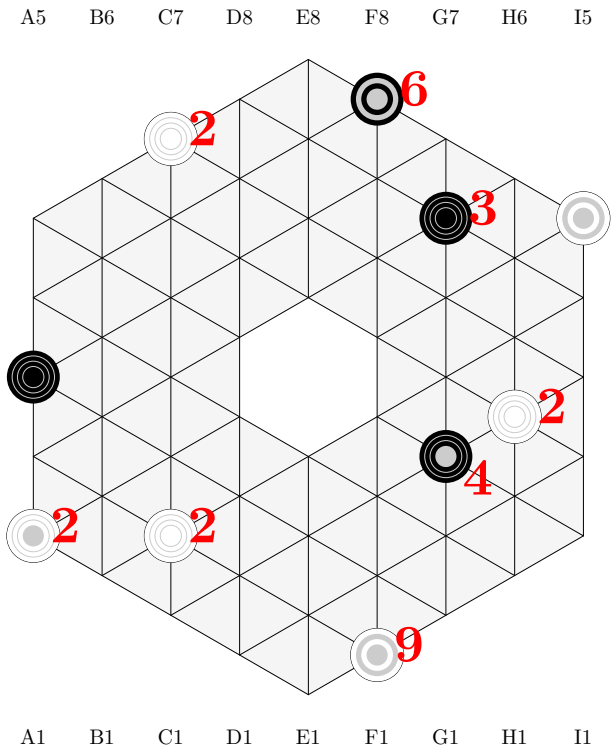


Figure 1.4: In this position, black player is on turn. After the last black Tzaar stack captures white Tzaar piece in the right corner, the only move not leading to a loss for black is the pass move.

We may also observe that black player has a little advantage, because he is stacking first. Hence he can often threaten his opponent by chasing his stacks.

The first move is always a capture move, but it often depends which type of piece is captured. It is mostly the best to capture a type of piece which from that the opponent does not have a high stack, and secondarily from which there are not many stacks. In a typical game a player starts creating a stack of Tzaar, thus it is probably convenient to capture Tzarras.

There are generally two strategies how to stack. The first is creating one high stack that is powerful and that can capture all opponent's stacks or that forces opponent to raise his highest stack (when a player has a little lower stack than his opponent). The second strategy is based on creating more lower stacks, mostly of size two, but it is safer when some of them have height at least three.

It is not known to me which strategy is better, probably the first for black player and the second for white player. With the first strategy a player can quite easily threaten or even capture small opponent's stacks, but with the second it

is sometimes impossible for the opponent to create a new stack (it would be captured immediately) and the opponent can lose because of it. The second strategy is more reasonable in the endgame, since it decreases the opponent's capturing possibilities. Having a stack much higher than all other opponent's stacks is only a little advantageous in the endgame.

These strategy observations were mostly about material; now we discuss some positional strategy features. One important positional advantage is that high stacks can move easily to any direction and thus they threaten large part of the board. We can conclude that it is better to have high stacks in the middle of the board, not on the border. Moreover from the middle of the board a stack can nearly always escape from a threat by a higher stack. The worst position is in one of the six corners. It is also good to limit possibilities to move for opponent's high stacks.

When a stack is isolated, i.e., it cannot be captured and it cannot capture or stack (there are no other pieces on the same lines as the isolated stack), the type of that piece is safe. Hence the player cannot run out of it which is a great advantage. But isolating a high stack is not good, because the stack cannot be used for capturing opponent's stacks.

As the game is coming to the end it is useful for a player to limit opponent's possibilities of captures and also prepare own capture possibilities. Most importantly, limiting moving possibilities of an opponent's high stack is advantageous.

Moving pieces (stacks of height one) from the middle of the board is less important and useful only before the endgame—otherwise the opponent can jump through them and maybe double-capture a high stack.

1.4 Game Properties of Tzaar

Before creating a computer AI for playing Tzaar, it is convenient to know game properties. According to Heule and Rothkrantz [14], and Allis [7], we try to estimate the state space and game tree complexity of Tzaar. Then we show some other properties of Tzaar, e.g., branching factor in different parts of the game, the number of starting positions, and the number of positions in the endgame with a fixed number of stacks.

First we count the *maximum height of a stack*. Observe that before a stacking move creating a stack of height h, the opponent must have captured at least $h-1$ pieces (at least one capture before stacking to height two). There should be two pieces of another types present and there are 30 pieces of each color, so the maximum stack height is 14 because of 13 captures before stacking to the height 14 and two other pieces visible.

The *state space complexity* is the number of game positions reachable from any starting position. We bound it from above by the sum of the number of positions with a fixed number of stacks on the board. There are $\binom{60}{v}$ different choices of fields for stacks, where v is the number of free fields on the board Let k be the sum of heights of all white stacks on the board, and analogously ℓ for the black color. Let there be s white stacks, so the number of black stacks is $60-v-s$. Then k is bounded by the number of necessary captures before v free fields are on the board.

The number of different stack heights for s stacks with k white pieces is $\binom{k-1}{s-1}$; the number of different choices of fields for white pieces is $\binom{60-v}{s}$ and 3^s is the number of different types of white stacks. Similar formulas holds for black player. We note that two positions that differ only in types of pieces inside stacks (not on the top) are equivalent. This gives us the upper bound on the number of possible states:

$$\sum_{v=1}^{55}\binom{60}{v}\sum_{k=2}^{30-\lfloor\frac{1}{4}v\rfloor}\sum_{s=2}^{\min(k,58-v)}\binom{60-v}{s}\binom{k-1}{s-1}3^s\cdot\sum_{\ell=60-v-s}^{30-\lfloor\frac{1}{4}v\rfloor}\binom{\ell-1}{59-v-s}3^{60-s-v}$$

$$\doteq 9.17\cdot 10^{57}$$

Let us now take symmetries into account. The position can have 6 equivalent rotations. The position may also have 6 isomorphic mirrors by 6 axes (between opposite corners of the board and between centers of opposite sides). Mirroring twice by any two axes results in a rotated position, thus there are 12 isomorphic positions. This decreases the upper bound of the state space complexity to $7.64\cdot 10^{56}$. (we divided the value by 12 simply, because using Burnside's Lemma does not change at least the first 10 digits).

We can observe that this is the number of positions which can be reached from all starting positions altogether, but some positions (and maybe most of them) can be obtained from more than one initial position. Note that positions in the placement phase of the tournament version are already counted.

The *game tree complexity* is defined as the number of leaf nodes in the minimum solution tree of the initial position(s), which is usually roughly counted as the average branching factor (according to the depth) to the power of the average game length in turns. Since pass moves are played rarely we can estimate that in every turn of a player, except for the first turn of white player, two stacks disappear from the board —one captured and one captured, or stacked. There should be at least 5 stacks in the final position, thus the upper bound on the number of turns in a game in which players do not pass is 28.

The *branching factor*, i.e., the number of possibilities how a player can play in one turn, depends on the starting position. The fixed starting position has the maximum branching factor around 5 500, but there are starting positions with the branching factor up to 10 000. We count positions reachable by two possible ways only once, otherwise the branching factor can be 14 000. During the game, the branching factor is decreasing as the pieces are captured or stacked. Table 1.1 provides a summary of the minimum, maximum and average branching factor according to the number of stacks on the board, computed statistically from real play-outs.

The game tree complexity is usually estimated by multiplying the average branching factor for each turn. For Tzaar we get approximately 10^{79}.

In the tournament version of the game, the search tree is even much more larger because of the placement phase. The first player has 60 possibilities where to put his first piece, the second player has 59 possibilities, the first then has 58 ... Thus the game tree of the placement phase has 60! leaves (without taking symmetries into account) and the game tree complexity of the tournament version of the game is $60!\cdot 10^{79}\doteq 10^{161}$.

Stacks	59	55	49	43	37	31	25	19	13	9	7	6
Positions	470	464	456	460	446	427	399	338	170	61	15	7
Min	4961	3962	2732	1732	906	403	139	35	1	1	1	1
Max	9933	7651	6007	4235	2986	2078	1073	476	114	21	4	2
Avg	7497	5965	4463	2971	1978	1117	562	203	37	7	1	1

Table 1.1: Minimum, maximum and average branching factor according to the number of stacks on the board. The table contains also the number of positions from which the values were obtained. We take positions reachable by two possible ways only once. We sampled positions from real games at BAJ [28].

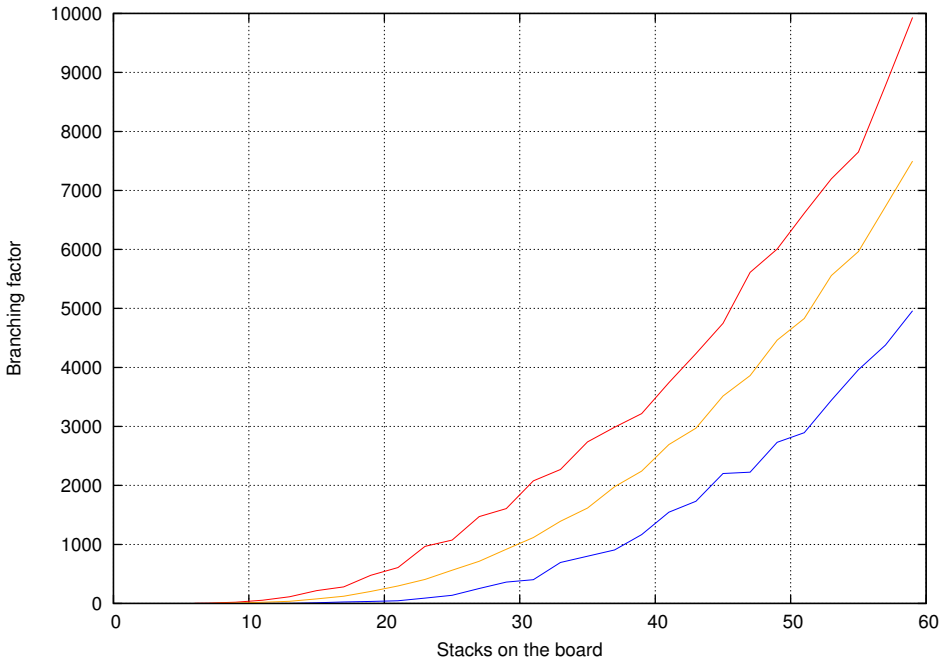


Figure 1.5: Maximum (red), average (yellow) and minimum (blue) branching factor according to the number of stacks on the board.

The number of different starting positions is the number of permutations with repetitions:

$$\frac{60!}{(15! \cdot 9! \cdot 6!)^2} \doteq 7.13 \cdot 10^{40}$$

There are again 12 symmetries which results in $5.94 \cdot 10^{39}$ different starting positions.

It is also handy to know the number of endgame positions. We count the number of positions with six stacks of different types or colors—if there are two pieces of the same color and type, the position is won by one of players. We observe that the number of positions with more than six stacks is higher. The number of positions with exactly six stacks is the number of different choices of six fields on the board multiplied by the number of permutations of six stacks and the number of different stack sizes for each piece type and for each player. The maximum sum of stack sizes for a player is 16, because there should be a capture before each stacking. These observations give us the following formula:

$$\binom{60}{6} \cdot 6! \cdot \left(\sum_{i=3}^{16} \binom{i-1}{2} \right)^2 = 1.13 \cdot 10^{16}$$

where i is the sum of stack heights for one player.

After taking symmetries into account, we get $9.42 \cdot 10^{14}$ different positions with six different stacks.

1.5 Existing Tzaar Programs with Artificial Intelligence

Up to March 2013 there are several Tzaar programs with an artificial intelligence (AI) available on the Internet: HsTZAAR [24], TZ1 [21], students' and teacher's works at the Department of Mathematical Sciences at the University of Alaska [31] and robots on BoardSpace.net [3].

The *HsTZAAR* program [24] published as an open source project is created in Haskell. It is developed from the older program *htzaar*. It offers graphical interface and saving and loading games. There are four levels of an AI opponent. The second best plays quite reasonable moves. The best AI opponent is very slow (in the beginning of a game it thinks more than a few minutes in each move depending on the hardware). The first two levels are very quick, but still they play reasonable moves.

TZ1 [21] is written in Java and also has a graphical interface. The AI opponent plays very bad moves and can be beaten by intermediate players easily.

There are four available students' and teacher's works on University of Alaska's site [31]: *Mockinator*, *Mockinator++* (improved Mockinator), *BiTzaarBot* (with algorithm inspired by the Monte-Carlo Tree Search) and *GreensteinTzaarAI*. They are able to connect to the Daedalus Game Manager [4] which has graphic user interface for Tzaar and which maintain games between programs. GreensteinTzaarAI won a tournament between these programs [31].

Robots on BoardSpace.net [3] run under Java. The Dumbot is the weakest robot there, but Smartbot and Bestbot also do not play well and experienced players are able to win against them nearly every time. They even do not stack the first time they can. Additionally Bestbot can think in some positions for more than half an hour (depending on the hardware).

For a comparison between these programs and Waltz see Section 4.2.2. We note that it is not possible for us to test Waltz against BoardSpace.net robots and TZ1 automatically.

2. Search Algorithms for Board Games

This chapter describes existing algorithms for searching for the best move in two-player strategic games without randomness and with a complete information, for example, Chess, Go, or Tzaar. In these games, one theoretical method for searching for the best move is generating the whole game tree for a given position. We want to find a winning strategy, that is, a move resulting in a position from which all opponent's moves leads to a winning position for us, i.e., for each such position there exists our move leading to a position from which all opponent's moves lead to a winning position for us—with recursively the same definition.

This strategy can be found, if there is any, by the *Minimax algorithm* that we describe in Section 2.1.1. Searching the whole game tree would result in the best possible play in winning positions (the move leading to a win would be always found), but the problem is that game trees for most board games including Chess, Go and Tzaar are too large to be searched completely in a reasonable time. Only a very small part of the game tree can be searched within a given time, that is, up to several minutes.

We describe basically two different algorithms, Alpha-beta and Proof-number Search, together with some of their enhancements. Note that there are other algorithms, like Dependency-based search [7] and Lambda search [22], but they are not suitable in the domain of Tzaar as shown in Section 3.2. Monte Carlo Tree Search [8, 17] may be a good choice for Tzaar, but it was not tried.

2.1 Alpha-beta Algorithm

Alpha-beta algorithm is an extension of the Minimax algorithm by pruning non perspective branches of the game tree, so we describe Minimax first and then the Alpha-beta pruning. There are many enhancements for Alpha-beta, most of them are for making pruning as effective as possible. We describe only the most common and appropriate for Tzaar.

2.1.1 Minimax

The base of the Minimax algorithm is a depth-first search on the game tree in which the root node is the position for which we want to find the best move. Because of a time limit the tree is searched only to a given depth. The searched part of the game tree is called the *search tree*.

A *value* is assigned to each node in the search tree and the node evaluation is done recursively. [1] There are two players which we name *Max* and *Min* and let Max be on turn in the root position of the tree. Max wants the value of his nodes to be as high as possible and Min as low as possible.

Leaves are evaluated directly, final positions in which the game ends obtain $+\infty$ if won by Max, $-\infty$ if won by Min, and zero in the case of a tie. Other leaf

[1] We sometimes use a *value of a move* that means simply a value of the position after executing the move.

positions have value counted by a heuristic *evaluation function* that returns value higher than zero in positions better for Max player, approximately 0 in balanced positions and less than zero in positions in which Min has an advantage. It should hold that the better the position for Max, the higher the value is and vice versa for Min. Our evaluation function for Tzaar is described in Section 3.5.

Nodes inside the tree including the root node count the value from values of their sons in the tree. When Max is on turn, a node obtains the maximum of values of his sons, for Min it is the minimum. The best move from the root position leads to the son that has the same value as the root.

Pseudocode of Minimax is in Algorithm 1. The function returns the best possible value and move for the player on turn. Note that ∞ is a constant that should be more than any value which the evaluation function can return.

Algorithm 1 Minimax algorithm

```
1: function MINIMAX(position, depth, onTurn)
2:     if depth = 0 or endOfGame(position) then                ▷ leaf node
3:         return (evaluate(position), emptyMove)
4:     end if
5:     if onTurn = Max then
6:         bestValue ← −∞ − 1
7:         bestMove ← emptyMove                    ▷ Try all moves of player Max
8:         for all m in generateMoves(position, Max) do:
9:             (value, move) ← minimax(executeMove(m), depth − 1, Min)
10:            if value > bestValue then
11:                bestValue ← value
12:                bestMove ← m
13:            end if
14:        end for
15:        return (bestValue, bestMove)
16:    end if
17:    if onTurn = Min then
18:        bestValue ← ∞ + 1
19:        bestMove ← emptyMove                    ▷ Try all moves of player Min
20:        for all m in generateMoves(position, Min) do:
21:            (value, move) ← minimax(executeMove(m), depth − 1, Max)
22:            if value < bestValue then
23:                bestValue ← value
24:                bestMove ← m
25:            end if
26:        end for
27:        return (bestValue, bestMove)
28:    end if
29: end function
```

In this Minimax implementation, there should be two different formulas for counting the value in each internal node, one for each player. To have only one formula, *Negamax* adjustment is used. A player on turn is always Max, i.e., the evaluation function returns high values when the player on turn has an advantage.

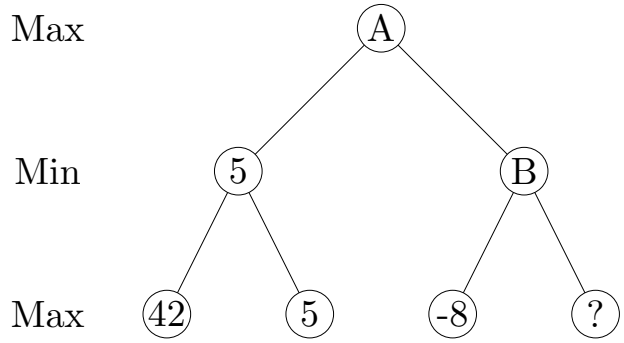


Figure 2.1: Example of a search tree. The position ? need not to be searched, since the position B has value at most -8 and thus the root position have value 5.

The maximum of values of sons is always counted. The value is multiplied by -1 between search tree levels.

The time complexity of this algorithm is $\mathcal{O}(b^d)$ where b is the average branching factor and d is the search depth. The space complexity is $\mathcal{O}(d)$ because of a stack for remembering nodes on the path to leaves.

2.1.2 Alpha-beta Pruning

The *Alpha-beta pruning* (or the Alpha-beta algorithm) is an important improvement of the Minimax algorithm. It is based on the observation that mostly not all nodes have to be searched to obtain a value of the root position. For example see a search tree in Figure 2.1.

This observation can be generalized. While searching the tree we remember variables *alpha*, the best achievable value for the player Max, and *beta*, the best achievable value for Min. The best achievable means that whatever opponent's moves are, the value is guaranteed—it can be better for the player, but not worse. At the beginning *alpha* is $-\infty$ (Max can lose the game) and *beta* is $+\infty$ (Min can lose too).

In the position where Max is on turn, variable *alpha* is updated and if it exceeds *beta*, searching the subtree of the node is interrupted. This interruption is called a *cutoff*. The similar with *alpha* and *beta* swapped holds for positions of Min. Thus *alpha* is always strictly less than *beta*.

In Algorithm 2 we present a pseudocode of Alpha-beta pruning together with Negamax adjusted—we count always maximum, values are multiplied by -1 between levels, and *alpha* and *beta* are swapped. Note that the function is returning only the value, but not the best move for simplicity.

The worst-case complexity of Alpha-beta is the same as for Minimax—when nodes are examined from the worst to the best. But if they are searched from the best to the worst, pruning is done as much as possible and in every second level of the search tree (odd, or even depending on who has the advantage) only one move has to be examined. Hence the time complexity of the algorithm is $\mathcal{O}(b^d)$ and $\Omega(b^{d/2})$. For more elaborated analysis of the algorithm and proof of its correctness see the article of Knuth and Moore [16].

Next we propose a new function for choosing randomly a move that is only a little worse than the best move. Then some enhancements for the Alpha-beta

Algorithm 2 Alpha-beta algorithm

```
1: function alphaBeta(position, depth, onTurn, alpha, beta)
2:     if depth = 0 or endOfGame(position) then                ▷ leaf node
3:         return evaluate(position)
4:     end if
5:     bestValue ← −∞ − 1
6:     for all m in generateMoves(position, onTurn) do:        ▷ Try all moves
7:         value ← − alphaBeta(executeMove(m), depth−1, opponent(onTurn),
       −beta, −alpha)
8:         if value > bestValue then
9:             bestValue ← value
10:        end if
11:        if value > alpha then
12:            alpha ← value
13:        end if
14:        if alpha ≥ beta then                                ▷ Pruning (cutoff)
15:            break
16:        end if
17:    end for
18:    return bestValue
19: end function
```

algorithm are presented. See Section 4.1.1 how the enhancements have effect on the duration of the search in the domain of Tzaar.

2.1.3 Randomized Alpha-beta

When the computer plays many times without randomness, regularly repeating patterns in its strategy can be observed and exploited by the opponent. Hence a randomized strategy is needed. On the other hand, randomization should not result in silly moves. The goal is to choose a move with a value not far from the value of the best move, i.e., with a value at least $bestValue - margin$ for a given constant *margin*. We call such moves *good-enough*. We propose an algorithm for finding all good-enough moves using a modification of Alpha-beta.

The search differs from Alpha-beta in the root node, for other nodes the deterministic Alpha-beta is executed. For the recursive call of Alpha-beta we have to set $beta := -bestValue + margin + 1$ instead of $-bestValue$, otherwise false good-enough moves may be found. The reason is as follows. Suppose that the best move is searched first and there are some other good enough moves. While searching these moves in the second level of the tree from the root, *alpha* is $bestMoveValue$ (best value that Max can achieve) and *beta* is updated. For example it can happen that *beta* changes to $bestMoveValue - 5$ and the search in this node is pruned, so move is considered to be good enough, but the real value of the move is $bestMoveValue - 100$ and the move is in fact not good enough.

In the root node we collect all moves together with their value into a list. After searching all moves, the list is filtered to contain only good enough moves. Among these nodes the result is chosen randomly uniformly. This leads to Algorithm 3 executed for the root node. For clarity Alpha-beta is now not in the Negamax

form. Note that parameters *alpha* and *beta* are useless in the root node, so we omit them. One can add a break to the function when a winning move is found.

Algorithm 3 Randomized Alpha-beta algorithm

```
1: function RandomizedAlphaBeta(position, depth, onTurn)
2:     bestValue ← −∞
3:     allMoves ← ∅
4:     for all m in generateMoves(position, onTurn) do:
5:         value ← −alphaBeta(executeMove(m), depth − 1,
   opponent(onTurn), −∞, −bestValue + margin + 1)  ▷ Note a change in beta
6:         Add (m, value) to the list allMoves
7:         if value > bestValue then
8:             bestValue ← value
9:         end if
10:    end for
11:    goodEnough ← select moves with value ≥ bestValue − margin from
   allMoves
12:    return uniformly random move from goodEnough
13: end function
```

Up to my knowledge nobody has done choosing random moves using Alpha-beta in this way, so this algorithm for choosing random moves via the Alpha-beta search is a contribution of this thesis.

2.1.4 Iterative Deepening

Iterative deepening (ID) is simply running the Alpha-beta algorithm first to depth 1, then to depth 2, 3, ... Searching one turn of a player deeper lasts in most games much more than the previous search, so the deepest search takes nearly the whole time of all searches, hence ID does not slow down the search. It is used to estimate how deep can Waltz search within a time limit. The implementation of this estimation for Tzaar is briefly described in Section 3.4.

2.1.5 Transposition Table

Transposition table (TT) is in fact a hash table for storing information about positions. In many games it happens quite often that two positions can be reached by two different move sequences. In these cases storing some information about searched positions comes in handy.

There are some differences between the Transposition Table and a usual hash table. Since the whole position representation is large, we cannot store the position in TT. Instead of it we generate a big enough hash value for the position (64 bits are mostly sufficient) and only a part of the hash value is used as an *index* in the table, i.e., the position in the array in which TT is stored. Thus it is useful to have the size of TT 2^k for a constant k.

Since the number of searched positions is very high (up to billions), there can occur two types of collisions: two different positions obtain the same hash value—this is called *type-1 error*—or two positions obtain the same index in TT—*type-2 error*.

Type-1 error is a lot more critical than type-2 error, so the hash value range should be very high and the hash function has to uniformly and randomly give values to the positions. This error is rare for most games with a good hash function and it is tested only by trying to play saved moves from TT in the position. It is usually not tested in another way, since the test would consume too much memory and would slow down the search.

Type-2 error occur quite often, because the size of TT is limited by the memory and usually does not exceed millions. This error can be found by comparing the whole hash value stored in TT

The problem with type-2 error occurs when we want to save a position into TT and there is another position saved on the index. If we just delete the position that was on the index, we may rewrite a search result that was counted for a long time by a result calculated quickly. Thus we use a *replacement scheme Two big*. In this scheme we count how many positions were searched to obtain a value of a position and store it to TT. On each index there are two positions in the table. Newly inserted position to TT is always stored and when there are already two positions under one index, we delete the one with fewer nodes searched to obtain its value. For more replacements schemes and comparison of them in the domain of Chess see [9].

In whole for each position we want to save to TT the hash value, the counted value, the best move, the search depth and the number of nodes examined when searching the position. The counted value can be one of three types: *lower bound* when the value is lower than *alpha*, *upper bound* when the value is bigger than *beta* (the search was pruned in the position), or *exact value* otherwise. The type of the value is also saved.

When we want to search a position in depth $d > 0$, we first look into TT whether it was searched to the sufficient depth. If so and the type of the value is exact value, the position need not be searched. Otherwise it is searched and we also update the bound *alpha* or *beta* when the type of the value is lower bound, or upper bound, respectively. After searching a position we save the result into TT, i.e., the value, the type of the value and the best move, together with some information about the search.

The only theoretical thing left around TT is the hash function. For board games *Zobrist hashing* [32] is usually used. For each possible combination of figure and field on the board (including empty fields), a random value in the range of hash values is generated. For example for Tzaar a random value is generated for each combination of a stack type, color, height and a field. The hash value of a position is xor of values of all combinations of a figure and a field on the board. The advantage of the Zobrist hashing is that it can be counted incrementally, i.e., after a move the value is changed by xor with what has disappeared from the board and what has appeared.

2.1.6 Move Ordering

The crucial thing about Alpha-beta algorithm is to have moves in the order from the best to the worst. Since we do not know which move is the best, we have to try some heuristics.

The first and probably the most important heuristic comes from using the

Iterative Deepening and the Transposition Table. Suppose we are not in the first iteration of ID, then we have already searched internal nodes of the search tree and stored for each of them the best move in the previous iteration of ID. The best move can be changed in the next iteration, since we search the position deeper, but we can use the stored move as the first. This heuristic is often called the *Hash Move* (the move is stored in TT) or the *Principal Variation Move* (it was the best in the previous iteration of ID).

If the search is not pruned by searching the Hash Move (if it exists), we have to try another moves and we want to do it in a good order. One way how to do it is generating moves already in a good order, the other way is first generate all moves, assign them heuristic value and sort them according to the value. We use the latter.

2.1.7 History Heuristic

The *History Heuristic* is a dynamic method for making the Move Ordering better. It is based on counting how many times each move caused a cutoff (the search was interrupted in the current node), e.g., by using quadratic array with one coordinate as the source field of the move and the other coordinate as the destination field. Moves which caused a cutoff many times should be tried before the other moves, since they likely cause a cutoff again. One possible implementation is to add the number of cutoffs to the heuristic value for sorting generated moves.

Another important heuristic in many Chess programs, called the *Killer Move*, is trying a few moves that caused a cutoff in another nodes in the same depth of the search tree. It is similar to the History Heuristic.

2.1.8 NegaScout

The *NegaScout algorithm* comes with a following idea. We restrict ourselves, set *alpha* close to *beta* and try to search a move. If we still obtain a value at most *alpha*, or at least *beta* (thus a cutoff in the second case), we do not need to search the move again without the restriction. Otherwise the re-search of the move is needed.

The restriction is that we set *beta* temporarily to $alpha + 1$. Since the range between *alpha* and *beta* is called *window*, this restriction is called *null window* (the window is as narrow as possible). We can observe that the search with the null window is often much quicker that with the full window, i.e., without restrictions. The whole NegaScout algorithm is more effective than Alpha-beta (effectiveness means number of nodes examined), but relies on a good Move Ordering. The proof of its correctness can be found in [19].

Algorithm 4 shows the pseudocode of the NegaScout.

2.1.9 Quiescence Search

Alpha-beta stands on a good leaf evaluation. Since the evaluation should be quick, it can often happen that a leaf obtain a high value, but the value change dramatically with the next move (which is not examined). This problem is called *horizon effect*.

Algorithm 4 NegaScout algorithm

```
1: function NEGASCOUT(position, depth, onTurn, alpha, beta)
2:     if depth = 0 or endOfGame(position) then                    ▷ leaf node
3:         return evaluate(position)
4:     end if
5:     bestValue ← −∞ − 1
6:     beta2 ← β                                  ▷ Full window for the first move
7:     for all m in generateMoves(position, onTurn) do:            ▷ Try all moves
8:         value ← -negaScout(executeMove(m), depth − 1, opponent(onTurn),
   −beta2, −alpha)                                                 ▷ Null window search
9:         if value > alpha and value < beta and m is not the first move then
10:            value ← -negaScout(executeMove(m), depth − 1,
   opponent(onTurn), −beta, −alpha)                                ▷ Full window re-search
11:        end if
12:        if value > bestValue then
13:            bestValue ← value
14:        end if
15:        if value > alpha then
16:            alpha ← value
17:        end if
18:        if alpha ≥ beta then                                    ▷ Pruning (cutoff)
19:            break
20:        end if
21:        beta2 ← α + 1                                           ▷ Update beta2
22:    end for
23:    return bestValue
24: end function
```

To resolve the problem, Quiescence Search is used. In every leaf a restricted search is run and the restriction is that we use only moves which can change value dramatically, like capturing a high stack in Tzaar. The leaves of the search tree are *quiet*, i.e., there is no move that changes the value much. For more information see e.g. [20].

2.2 Proof-number Search

Proof-number Search (PNS) is a best first search algorithm for finding the winning strategy in game trees developed by Allis [7]. In a sufficient time and memory it can decide whether a given position is winning for us, or for our opponent (supposing we are on turn). Ties can be viewed as a loss for us. For simplicity and their non-existence in Tzaar we do not mention ties in the following description.

During the search we have a part of the game tree stored in the memory, we call this part an *expanded tree*. A node in the tree can have three values: *true* if it is won for us, *false* if it is won for the opponent and *unknown* otherwise. The tree is viewed as an AND/OR tree, i.e., an internal node is OR when we are on turn, or AND when our opponent is on turn and the root node is an OR node. OR means that we can choose any move leading to a win, but in opponent's AND

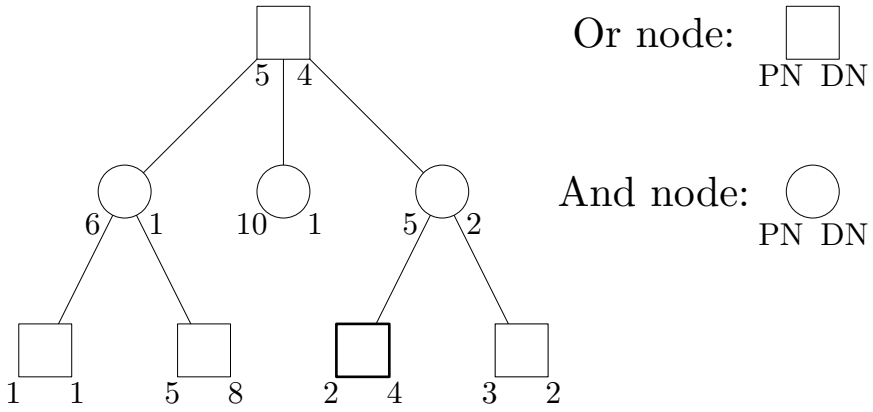


Figure 2.2: Example of a part of an AND/OR tree with proof and disproof numbers. Circles denote AND nodes and squares denote OR nodes. The highlighted OR node is the most proving node in this tree.

nodes we must examine all his moves to be sure the node is winning for us. Note that the algorithm in general works also on arbitrary AND/OR trees [7], but we deal only with game trees.

We want to find a proof in the tree that corresponds to a winning strategy for us, or disproof when there is no winning strategy. The tree can also be viewed as a very huge formula for which we want to decide whether it is true or not.

The main idea of the algorithm is to look for the shortest proof and disproof simultaneously. We store a *proof number* and a *disproof number* for each node in the expanded tree. The proof number (PN) of a node N is the minimum number of nodes that we have to prove to obtain the value true in the node N. The disproof number (DN) is similarly the number of nodes we have to disproof to have the value false in the node N.

In leaves of the tree, nodes obtain proof and disproof numbers straightforwardly: PN is 0 and DN is ∞ for a leaf with value true (no node has to be proved to prove the node, and the node cannot be disproved), PN is ∞ and DN is 0 for a false leaf and both PN and DN are one for an unknown leaf.

For an internal OR node, PN is minimum of PNs of its children, since we can choose where to move to win, and DN is the sum of its children. When we sum DNs and there is ∞ between the children, the resulting DN is also ∞. For an internal AND node the computation is done vice versa: PN is the sum of children's PNs and DN is the minimum of children's DNs. We can see that an OR node has the value true if and only if PN is 0 and DN is ∞, and vice versa holds for an AND node. For an example see Figure 2.2.

The PNS algorithm can be basically described by a loop done until a root node has the value unknown. In the loop, three steps are executed:

1. find an unknown leaf node and expand it, i.e., add all its children to the expanded tree,

2. assign values unknown, or true, or false to the children,

3. update proof and disproof numbers in the tree.

The second step is obvious to implement and the third step is done by updating

PNs and DNs on the path from the expanded node to the root node (in this order) as PNs and DNs of the other nodes were not changed.

The choice of a node to expand is done according to the searching for the shortest proof and disproof. We want to find a leaf node for which it holds that proving it decrements PN and disproving it decrements DN of the root node. Such a node is called *the most proving node* (MPN). We select it by walking on a path from the root until we get to a leaf and find MPN. In OR nodes we continue to a child with the lowest PN, in AND nodes we go to a child with the lowest DN. For an explanation of such a selection see [7].

Note that the algorithm runs the best on trees which are not uniform, for example when one player repeatedly threats the other player who has only a few possible moves that do not lead to a quick loss. So the algorithm often finds a winning strategy that is in fact a long sequence of threats and the length of the strategy in moves may be far bigger than the length of the shortest winning strategy.

The algorithm can be simplified in a way similar to the Negamax algorithm. Let all nodes be OR and the only thing we have to do is swapping PN and DN between tree levels. See a pseudocode of the Depth-first Proof-number Search in Algorithm 5.

The main disadvantage of this algorithm is that it has the whole search tree in the memory. Many improvements that have lower memory requirements were invented, see [13] for a survey. We describe an adjustment of PNS that is up to my knowledge the most used nowadays.

2.2.1 Depth-first Proof-number Search

Depth-first Proof-number Search (DFPNS) is PNS adjusted to the depth-first search and can be implemented using one recursive function. In the memory we store only the path from the root to the current MPN, the other nodes already searched are in the Transposition Table (TT) similar to the one used in Alpha-beta. Note that nodes in TT can be also deleted in the case of a collision.

We also postpone the update of PN and DN of nodes upper in the search tree while MPN is still in their subtree. We do it by using thresholds on PN and DN. When the search selects a child, it sets the thresholds such that when PN or DN is at least the corresponding threshold, MPN is not in the child's subtree.

We show a formula for counting the thresholds for a child of an OR node. Let tpn and tdn be thresholds on PN and DN of the current OR node, $dn1$ be DN of the child with the lowest PN between children, $pn2$ be the second lowest PN and $sumDN$ be the sum of all children's DNs. Then the new threshold on PN is the minimum of tpn and $pn2 + 1$ and the threshold on DN is $tdn - sumDN + dn1$.

Note that the child with the minimum PN has MPN in its subtree (or it is already MPN), so the search goes to it. For an AND node the computation is done vice versa. The thresholds for the root node are set to ∞.

The Transposition Table plays important role in this algorithm. When we count non-trivially PN and DN of a node and MPN is not in the subtree of the node, we store PN and DN for that node to TT. When we are searching between children of a node (to count the minimum of PNs and the sum of DNs), we look for child's PN and DN first to TT, if the child is not a final position. If PN and

DN for the node are not in TT, we set PN and DN to 1.

Algorithm 5 shows the pseudocode for DFPNS adjusted similarly to Negamax.

Note that there are some differences to the pseudocode of DFPNS in [12], e.g., we do not need to call function `dfpns` for final positions and we can cut off the search immediately when we find a win.

We can also observe that DFPNS searches the tree in the same order as PNS when TT is big enough and no node is deleted from it.

There are some properties of DFPNS that can be adjusted to get a more effective algorithm. The first is the initialization of unknown leaves to more appropriate numbers than one, the second is better setting the thresholds for the selected child and the third is the way of counting disproof numbers in an OR node.

Next we describe most of DFPNS enhancements that were invented—the others are not so important for Tzaar, since they are dealing with problems not occurring in Tzaar. We show computations only for OR nodes, because the pseudocode in Algorithm 5 is modified in the Negamax way, i.e., every node counts PN and DN like an OR node, and one can change formulas for AND nodes straightforwardly.

We note that there is another problem in some games called *Graph History Interaction* (GHI). It occurs when the current state of a position depends on a few or even all positions that were in the game recently. The problem makes use of the Transposition Table harder. Since this is not the case of Tzaar, we do not describe any method how to deal with GHI. For a solution see e.g. [15].

2.2.2 Evaluation Function Based PNS

Evaluation Function Based PNS is an enhancement based on the better initialization of unknown leaves. It was originally proposed by Schadd and Winands [26] for PNS, but it can be easily added to DFPNS.

Since mostly more than one node has to be searched to determine the value, we want to take into account the branching factor (or at least an estimate of it) and also who has an advantage—that is estimated by the evaluation function. We use a step function

$$step(value) = \begin{cases} 1 & \text{if } value \geq t \\ 0 & \text{if } -t < value < t \\ -1 & \text{if } value \leq -t \end{cases}$$

where t is a threshold parameter corresponding to a value that indicates player's high advantage, i.e., the player is likely going to win. Using this function we initialize the proof and disproof numbers for an AND leaf node with this formulas:

$$pn = m \cdot (1 + b \cdot (1 - step(value))),$$
$$dn = 1 + a \ \cdot (1 + step(value))$$

where m is number of moves from the child (or an estimate of it) and a, b are parameters. Initialization for an OR leaf is done vice versa. (We present initialization of an AND leaf, because the initialization is always used in an OR node in the program.)

Algorithm 5 DFPNS algorithm

```
1: procedure DFPNS(node, onTurn, tpn, tdn)
2:     minPN ← ∞                                          ▷ Minimal PN
3:     dn1 ← 0                          ▷ DN of the child with the minimal PN
4:     minPNnode ← null                      ▷ Node with the minimal PN
5:     pn2 ← ∞                                ▷ The second minimal PN
6:     sumDN ← 0                                          ▷ Sum of DNs
7:     loop until break
8:         for all m in generateMoves(node, onTurn) do    ▷ Try all moves
9:             child ← executeMove(m)
10:            pn ← 1                    ▷ Default values for unknown leaves
11:            dn ← 1
12:            if endOfGame(node) then
13:                if winner = onTurn then                    ▷ Our win
14:                    addNodeToTT(node, 0, ∞)      ▷ PN is 0 and DN is ∞
15:                    return                ▷ No need to search anything else
16:                else                                       ▷ We lose
17:                    pn ← ∞
18:                    dn ← 0
19:                end if
20:            else
21:                ttEntry ← lookupInTT(child)
22:                if ttEntry ≠ null then          ▷ Already searched node
23:                    pn ← ttEntry.dn             ▷ Note swapping PN and DN
24:                    dn ← ttEntry.pn
25:                end if
26:            end if
27:            if sumDN < ∞ then
28:                sumDN ← sumDN + dn
29:            end if
30:            if pn < minPN then
31:                pn2 ← minPN
32:                minPN ← pn
33:                dn1 ← dn
34:            else if pn < pn2 then
35:                pn2 ← pn
36:            end if
37:        end for
38:        if minPN ≥ tpn or sumDN ≥ tdn then             ▷ Test thresholds
39:            addNodeToTT(node, minPN, sumDN)
40:            break
41:        end if
42:        ntpn ← tdn − sumDN + dn1              ▷ Swapping PN and DN again
43:        ntdn ← min(tpn, pn2 + 1)
44:        dfpns(child, opponent(onTurn), ntpn, ntdn)
45:    end loop
46: end procedure
```

2.2.3 $1 + \varepsilon$ Trick

The problem which leads to using the *1 + ε Trick* is switching between subtrees where the most proving node (MPN) is. The worst case is when two children of an OR node have nearly the same proof numbers and their subtrees cannot be stored in Transposition Table together. Then it can happen that proof numbers of these children increase repeatedly only by one, making the algorithm to switch between their subtrees very often and recalculate PNs and DNs of some nodes many times. Lew and Pawlewicz [18] proposed following solution of the problem.

To decrease the number of switches between subtrees we change the setting of the threshold on PN for a child of an OR node to the minimum of tpn and $pn2 \cdot (1 + \varepsilon) + 1$ for a constant $\varepsilon > 0$. The same formula with disproof numbers instead of proof numbers is used in AND nodes.

2.2.4 Weak PNS

As the positions often occur more than once in the game, the state space is described by a directed acyclic graph instead of a tree. Then DFPNS suffers from the *double-counting problem*, when the proof number of a position contains the proof number of another position more than once.

The problem can be resolved by modifying the summation of disproof numbers in OR nodes and proof numbers in AND nodes. Weak PNS [12] proposes taking the maximum disproof number and adding the number of children minus one. Another solution to this problem is described by Kishimoto [15].

2.2.5 Heuristic Weak PNS

We propose a new enhancement based on Weak PNS and the evaluation function. We modify counting disproof numbers in OR nodes in a way similar to Evaluation Function Based PNS. The idea of using evaluation function is also briefly mentioned by Kishimoto [15] when comparing Weak PNS to algorithms described there.

We define the step function similar to the one in Evaluation Function Based PNS:

$$\text{step}(value) = \begin{cases} 2 & \text{if } value \geq t, \\ 1 & \text{if } -t < value < t, \\ 0 & \text{if } value \leq -t, \end{cases}$$

where t is a parameter corresponding to a value that indicates a player's high advantage. We compute the disproof number (DN) as:

$$dn = maxDN + h(m-1)\,\text{step}(value)$$

where $maxDN$ is the maximum DN among children, m is the number of moves and $h > 0$ is a constant.

Now we discuss why this modification behaves better than Weak PNS. When the player on turn has a big advantage and $value \geq t$, we probably have to search many nodes to disprove the node. Since the player is likely going to win, DN is ∞ with a high probability. We can thus set DN to $maxDN + 2h(m-1)$. In the case of a balanced position, we count DN nearly the same as in Weak PNS, only

with the factor h. Because of this, the parameter h should be close to 1. When the player on turn is in a bad position, we likely do not need to search many positions to disprove the node, so DN is set to $maxDN$.

2.2.6 Dynamic Widening

Dynamic Widening is the technique that also tries to avoid the double-counting problem and overestimation of DN or PN in the way similar to Weak PNS. It was invented by Yoshizoe [27]. In an OR node we sum only DN of the J best children instead of summing all children's DN. The best means that we take the first J children in the increasing order by PN. This leads to the formula:

$$dn = \sum_{i=1}^{J} dn_i$$

where dn_i is the disproof number and pn_i is the proof number of i-th child and $pn_1 \leq pn_2 \leq \cdots \leq pn_n$. The parameter J can be a constant or it can depend on the number of children.

2.3 Alpha-beta and DFPNS in Lost Positions

Suppose we are not in a final position and Alpha-beta has found out that the position is lost, or DFPNS has found a disproof. Since a human opponent may overlook his win, we want to make a move that is still good despite the lost position. We also know nothing about our opponent, thus the playing heuristic in lost positions should be general.

One possible solution is to create as much chaotic position on the board as possible and hope that the opponent gets confused. However, this is a very domain-dependent and truly hard to define heuristic.

More general solution is to play a move that leads to a loss after the maximal possible number of moves. By doing that, it is more probable that the opponent makes a mistake. Using the Iterative Deepening in Alpha-beta, we just choose the best move in the last iteration where Alpha-beta has not found out that the position is lost.

Let D be the depth of the deepest loss in the search tree. Looking for D using DFPNS is also possible, but it may overlook a loss in depth less than D, since it does not necessarily search every position in the depth less than D. Anyway, we may count the maximal losing depth and the minimal winning depth similarly to the counting of proof and disproof numbers.

A won leaf obtains the maximal losing depth ∞ and the minimal winning depth 0 and vice versa for a lost leaf. The maximal losing depth of an unknown leaf is 3 and the minimal winning depth is ∞ (both are the worst case). An internal OR node has the minimal winning depth the minimum from maximal losing depths of his children plus one, and the maximal losing depth the maximum of the minimal winning depths of his children plus one.

Using this approach we are able to find a lower bound on the maximal losing depth of the root node and do the corresponding move.

3. Algorithms on the Domain of Tzaar

In this chapter we show how Waltz is implemented. First we describe some implementation details, mainly our representation of the Tzaar board.

In Section 1.4 we discussed properties of the game Tzaar, mainly the large branching factor due to two moves in a turn of each player and short games which last typically up to 28 turns of a player. According to these properties we discuss which algorithms are suitable for Waltz. Their implementation details and domain dependent heuristics follows.

3.1 Implementation of Tzaar Board

First of all we show some implementation basis of Waltz. The Tzaar board is a hexagon with five fields on each side, but we cannot simply store it in the memory as a hexagon. Thus it is sloped to be fit in the quadratic array `board` of size 9×9. In Waltz one linear array of size 81 is used instead of quadratic array.

Example of the array containing the fixed starting position (array members are separated by spaces):

```
-1   1   1   1   1 100 100 100 100
-1  -2   2   2   2  -1 100 100 100
-1  -2  -3   3   3  -2  -1 100 100
-1  -2  -3  -1   1  -3  -2  -1 100
 1   2   3   1 100  -1  -3  -2  -1
100  1   2   3  -1   1   3   2   1
100 100  1   2  -3  -3   3   2   1
100 100 100  1  -2  -2  -2   2   1
100 100 100 100 -1  -1  -1  -1   1
```

The number 100 stands for a field outside the board and also for the field in the center of the board. An empty field with no pieces has 0. Other numbers stand for different types of pieces: 1 is a white Tott, 2 is a white Tzarra, 3 is a white Tzaar and black pieces have the same numbers multiplied by -1.

In the array `board` a player can move in six directions: horizontally left, or right, vertically up, or down, and diagonally right and down, or left and up. The other diagonal directions (right and up, left and down) are not possible.

Heights of stacks are in another array of the same size and the same format. In these arrays the field usually denoted by A1 is in the left upper corner (on the index 0) and field I5 on the index 80.

Note that during the search there is only one position stored in the memory.

The program remembers also some additional information. The evaluation function and the other algorithms use some positional and material information, e.g., the hash value, the highest stacks of each type and the zone of control. This can be counted statically, i.e., for each position anew, but it would slow down the

search very much. Hence it is counted incrementally, i.e., the value is changed when a move is executed.

The zone of control determines how many pieces of a certain type can be captured by one move, no matter who is on turn. It is used by the evaluation function and for determining whether a player on turn has lost because of no possible captures—the zone of control of all opponent's piece types is zero in this case.

Details about implementation and the program can be found in the documentation, see Appendix A. The implementation of the algorithms is described in Section 3.3.

3.2 Algorithms for Tzaar

Since the game tree properties differ in the middle game and in the endgame, we discuss these parts of the game separately. Also we choose between knowledge based methods and brute force methods.

Some Chess computer programs use an opening database technique, i.e., tables with many different openings, to play quicker and better in the beginning of the game. In Tzaar a lot of games start with a random starting position and the number of starting positions is very large, exactly $5.94 \cdot 10^{39}$ (see Section 1.4). Hence building an opening database is not possible. Even if we consider only one starting position, e.g., the fixed starting position, there are many reasonable moves—it could be one fourth of all moves, thus the opening database for only the first six turns of a player would be very large. The opponent also can make Waltz fall out of the book doing a weaker move and this weaker move would not result in a position much better for Waltz if it is done early in the game. Instead of the opening database we use random Alpha-beta proposed in Section 2.1.3.

An endgame database is not reasonable too, since the number of positions with only six pieces is $6.22 \cdot 10^{14}$ which is too much. These positions can also be quickly solved by a simple Minimax search and the number of positions with more than six pieces is even larger. The opening and endgame databases are knowledge based method. According to the presented arguments together with the state space and game tree complexities and conclusions of Heule and Rothkrantz [14], we conclude that knowledge based methods are not applicable in Tzaar. Hence Waltz have to use a brute force search.

Opening has the largest branching factor, but it is not very different from the middle game. Before the endgame a player (attacker) mostly cannot capture defender's high stack, or even win in a few moves by a threat sequence. From the observation in Section 1.3 defender can escape with his stack from most of threats easily and there are often more different possibilities how to do it. Thus we can conclude that algorithms based on threats would be ineffective during the opening and middle game, therefore Proof-number Search (PNS) is used only during the endgame.

The only algorithm left that is known to me is Minimax with the Alpha-beta pruning, we call this algorithm Alpha-beta. In Waltz we use Alpha-beta with several enhancements, namely:

- Transposition Table (TT): Used for storing moves from the previous shal-

lower search (the Principal Variation Move, PVM) and also because some positions can be reached by a few different move sequences.

- Iterative Deepening (ID): Implemented because of time estimation (how deep may the engine search), and because of PVM.
- domain specific Move Ordering (MO): Done by heuristically assigning values to moves and sorting moves according to these values. See Section 3.6.
- History Heuristic (HH): Only for the first move of a turn.
- NegaScout (NS): To quickly find cutoff nodes.
- Randomized Alpha-beta: A new enhancement we propose.

See Section 4.1.1 for information how each enhancement makes the search quicker.

From important enhancements of the Alpha-beta algorithm the Quiescence Search was not implemented. The main reasons are that in most positions there is no move which changes the value much and that searching for it would probably mean to generate all moves which would cost a lot of time. Instead of that, the concept of the zone of control is used in the evaluation function. Killer Move heuristic was also not implemented.

In the endgame the branching factor is not so high and threat sequences occur more frequently. There are also fewer solutions to threats, therefore threats limit the branching factor and the Proof-number Search (PNS) can be sometimes more effective than the Alpha-beta search.

PNS as proposed by Allis [7] is very memory consuming and search trees in Tzaar endgames are too large to fit in the memory. For these reasons we use the Depth-first Proof-number Search (DFPNS) with the following enhancements:

- Move Ordering: The same as in Alpha-beta.
- Evaluation Function Based PNS (EFB PNS) [26]: Heuristic initialization of leaves using the evaluation function.
- $1 + \varepsilon$ Trick [18]: To avoid frequent jumping of the search across the tree.
- Weak PNS (WPNS) [12] and Dynamic Widening (DW) [27]: To suppress overestimation of proof and disproof numbers.
- Heuristic Weak PNS (HW PNS): A new enhancement we propose.
- Time estimation: How many nodes can DFPNS visit within a given time—at first a certain number of nodes is visited and then the number of nodes to visit is estimated.

See Section 4.1.4 for evaluation how each enhancement improves the search.

There are some other algorithms searching for the best move in board games. For the Lambda Search [22], I was not able to think up how to determine quickly the order of a threat, thus it is not implemented. Since there is mostly more than one possibility how to escape from a threat, we conclude that the Dependency-based Search [7] is not suitable for Tzaar. The Monte Carlo Tree Search [8, 17] is probably worth trying for Tzaar, but it is not implemented at all.

3.3 Implementation of the Algorithms

The main difference between Tzaar and Chess is that in Tzaar there are two moves in each turn of a player (except for the first turn of white player). Hence for simplicity moves are generated separately for the first and the second move of a turn and they are executed and reverted also separately. In Alpha-beta and DFPNS functions there are two nested loops, one for the first moves and the other for the second moves which are generated after doing the first move.

Also the depth in the search tree in Alpha-beta and DFPNS is counted in moves, not turns of a player. The program has often time to search the tree with the Alpha-beta algorithm to the depth 5 (2 and half turns of a player), but not to the depth 6 (3 turns) because of the large branching factor. Hence the Iterative Deepening increases search depth by one move. In DFPNS a maximal depth in which a player lose (see Section 2.3) is counted similarly—the depth is the number of moves.

Again for simplicity, searching functions assume that the first turn of white player consists of two moves as the other turns. This is in fact not true, but it does not result in worse play and does not slow down the search much.

The Transposition Table (TT) differs for Alpha-beta and DFPNS. TT for both algorithms stores for each position the hash value and the number of nodes searched to obtain the value of a position, i.e., visited nodes in the subtree of the position. Alpha-beta saves into TT the best moves, the counted value with the value type (exact value, lower bound, or upper bound) and the depth to which the position was searched. DFPNS saves the proof and disproof numbers, the minimal winning depth and the maximal losing depth.

The Alpha-beta pruning enhancements are implemented straightforwardly with the exception of the History Heuristic. The History Heuristic is counted only for the first move of a turn, because the array with both moves would have four dimensions (from and to field for the first and the second move) and thus would be very large—precisely $81^4 = 43\,046\,721$.

Note that the Principal Variation Move stored in TT from the previous iteration of ID is tried before generating moves and if it leads the cutoff, we do not need to generate moves at all.

Enhancements for DFPNS are implemented straightforwardly too. DFPNS uses the heuristic Move Ordering described in Section 3.6 which, quite surprisingly, helps to solve some positions, see Section 4.1.3.

The tournament version of Tzaar is not implemented, because it is also not implemented on servers where Tzaar is played and I do not know which strategy Waltz should follow in the placement phase.

3.4 Search Time Estimation

The program has limited time to search. For Alpha-beta the Iterative Deepening (ID) is used to search to the maximal depth that can fit into the time limit. To estimate the duration of searching to the next depth the duration of the last iteration of ID is multiplied by an expected branching factor.

The expected branching factor is lower than the real branching factor because of pruning. It depends on the last move done in leaves of the search tree, i.e.,

whether it was the first, or the second move of a player, and on the number of stacks on the board. Figure 3.1 shows the multipliers for both moves of a turn.

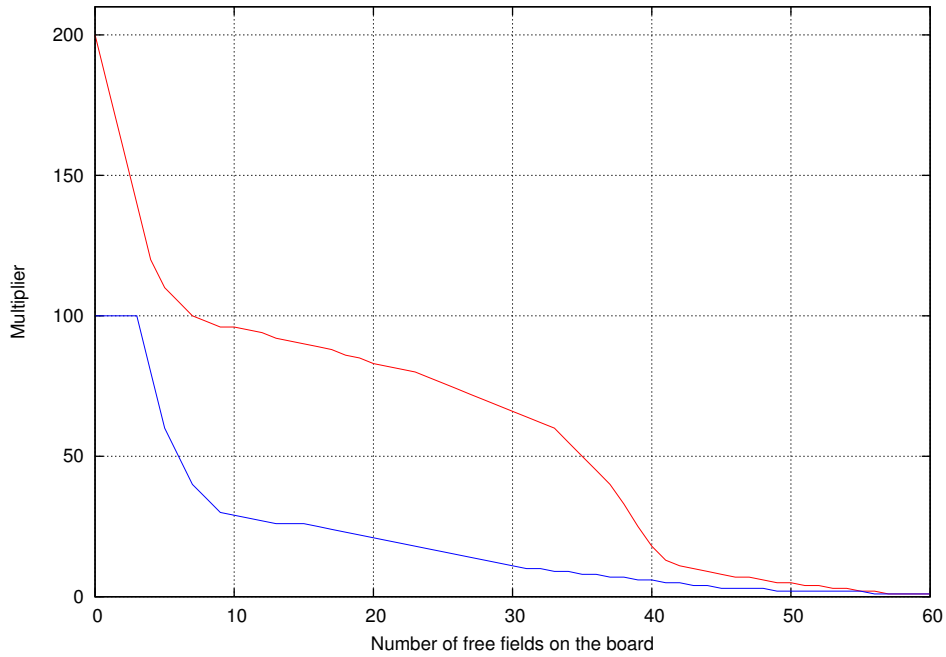


Figure 3.1: Graph of multipliers for ID for the first move (blue) and the second move (red) according to the number of free fields.

DFPNS is usually run until it finds a proof, or a disproof. To fit the search into the time limit it is stopped after searching a certain number of nodes. To estimate how many nodes can DFPNS search in a given time, at first a certain number of nodes is visited[1]. According to the duration of searching them, the maximal number of nodes to search within the given time is counted.

3.5 Evaluation Function

The evaluation of a position in Tzaar is used both by the Alpha-beta search and DFPNS. In positions with a positive value, white player has an advantage (∞ is a win), and vice versa for the black player. Balanced positions, i.e., positions where no player has an advantage, obtain number near to zero. We use basically this formula:

$$\begin{aligned}\text{eval}(\textit{position}) = {} & \text{material}(\textit{position}, \text{White}) + \text{positional}(\textit{position}, \text{White}) \\ & - \text{material}(\textit{position}, \text{Black}) - \text{positional}(\textit{position}, \text{Black})\end{aligned}$$

The material value for a player is the sum of values of player's stacks:

$$\text{material}(\textit{position}, \textit{player}) = \sum_{\substack{s \text{ is a stack} \\ \text{of } \textit{player}}} \text{heightValue}(s) \cdot \text{countValue}(s)$$

[1]For example it can be 100 000, but it depends also on the hardware.

The material value is more important in the first half of the game. The material value together with some positional information is counted incrementally (when a move is executed or reverted), other positional features evaluated by the function material are counted statically for each leaf node that is not won by a player.

The function heightValue grows rapidly for heights less than four, then stays nearly the same and decreases for stacks higher than eight. The reason is that instead of building a very high stacks a player can build more lower stacks, which is usually better. Values of this array are shown in Figure 3.2. The function countValue is inversely proportional to the number of stacks with the same piece type as the stack s. Its values are illustrated in Figure 3.3.

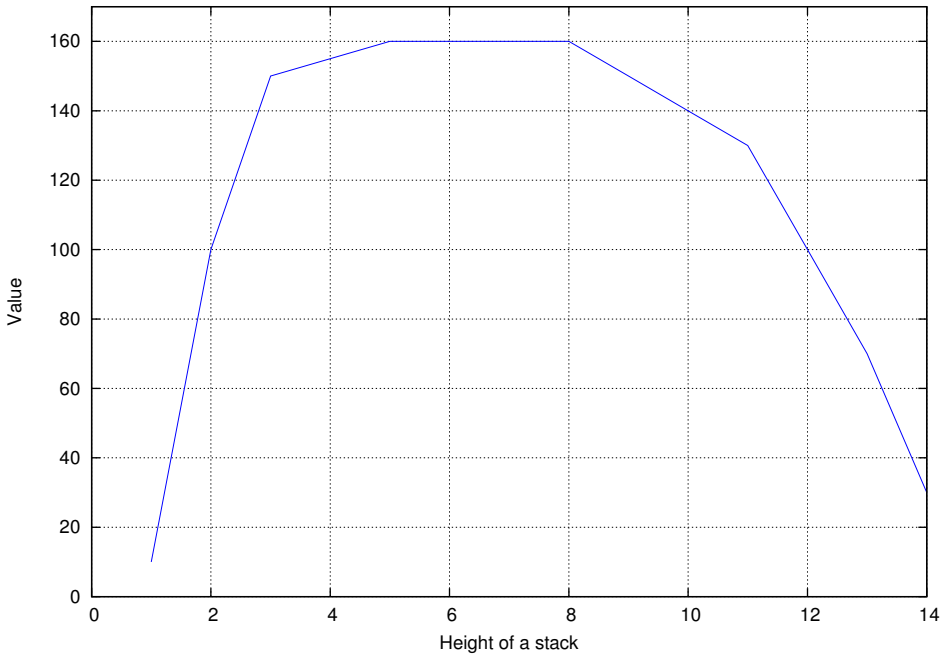


Figure 3.2: The function heightValue: the value of a stack according to its height.

For the positional value the Zone of Control (ZOC) is maintained. It determines how many stacks of a certain type can be captured by one move, no matter who is on turn. It is used also for determining whether a player on turn has lost because of no possible captures—the zone of control of all opponent's piece types is zero in this case.

The positional value for a player is roughly the sum of these bonuses:

- 20 000 000 for an immediate threat: The player is on turn and he can capture all stacks of an opponent's piece type (the player can win).
- 1 000 for a threat, when the player is not on turn.
- 1 000–200 000 if the opponent has few possible captures.
- Value of ZOC: $\sum_{\substack{\text{opponent's} \\ \text{piece type } t}} \text{stacksInZOC}(t) \cdot (1 - \text{count}(t)/\,\text{initialCount}(t))$
- 25 000 for each player's piece type that is "secure"—the player has a stack higher than all stacks of his opponent—and 100 000 if all types are secure.

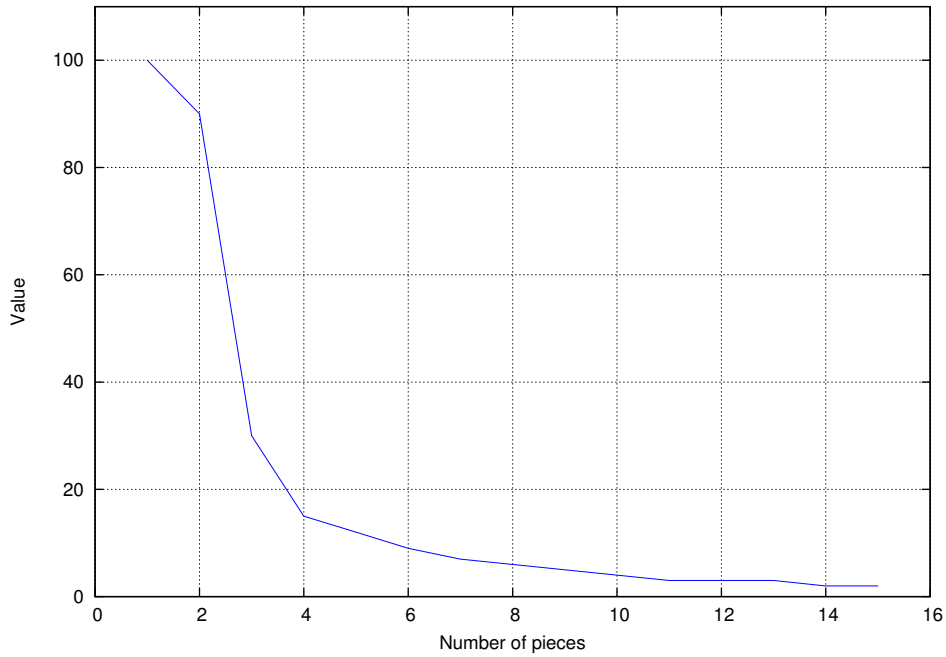


Figure 3.3: The function countValue: the value of a stack according to the number of pieces of the same type.

- 50 000 for stacks with height at least 2 of all types.
- 50 000 if an opponent's valuable stack can be captured.
- 1 000–100 000 for a high stack that cannot move.
- 10–25 for high stacks not on the margin of the board and −30 for a stack in the corner, since a high stack has only a little number of possibilities how to move in such positions.

We remark that the most of the evaluation function was done by inventing features and setting constants intuitively according to the observations done in Section 1.3 and then playing with Waltz. We also tested different versions of the evaluation function by numerous play-outs.

3.6 Move Ordering

The Move Ordering is done straightforwardly by generating moves into an array, assigning them a heuristic value and simply sorting them by the value using the QuickSort algorithm. The less the sort value is, the better the move should be and the sooner is executed. Moves for a player on turn are generated separately for the first and the second move of a turn and assigning a value to the moves also differs for the first and the second move.

The first move is always a capture. Capturing a stack of a type that occurs on the board quite often, i.e., there are relatively many visible pieces of that type, is not very advantageous, thus the sort value is linearly dependent on the count of stacks of the captured type—the multiplier is constant *CaptureCountMult* with the default value 20.

The height of the captured stack is also taken into account and it is more significant than the count of stacks of the stack type. The dependence on the height grows very fast with the height less than six and then it is approximately linear, see Figure 3.4.

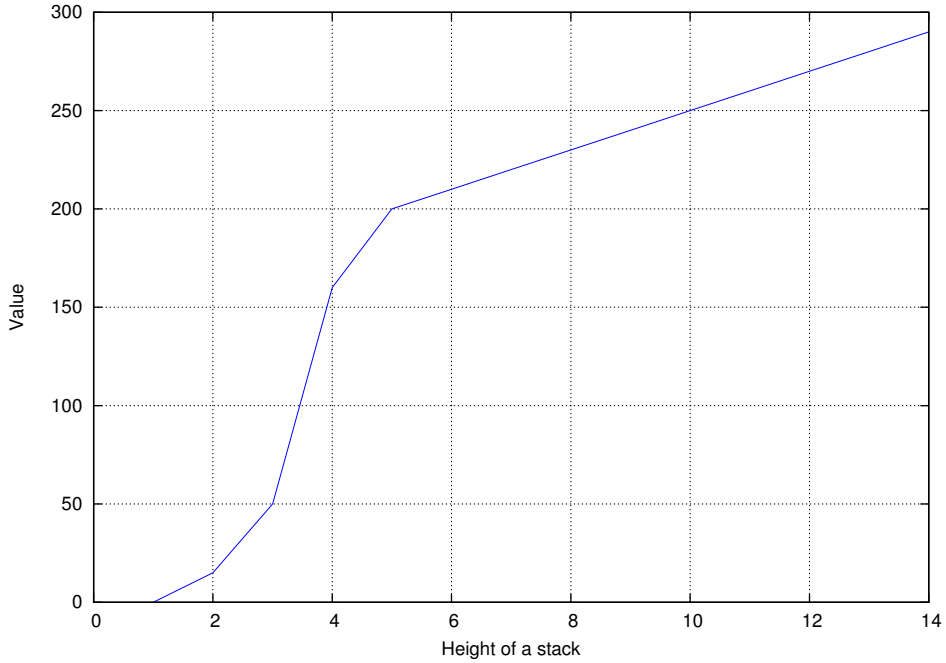


Figure 3.4: The function heightValueOfCapture value of a stack according to its height in the Move Ordering.

For the first move the History Heuristic is also taken into account. The number of prunes caused by a move, precisely by the move with the same fields from and to, is multiplied by the constant *HistoryPruneMult* (the default value is 20) and subtracted from the value.

This leads to a formula that assigns values to capture moves:

$$\begin{aligned}\text{captureMove}(move, pos) = {} & \text{heightValueOfCapture}(move) \\ & - \mathit{CaptureCountMult} \cdot \text{countType}(pos, \text{capturedType}(move)) \\ & - \mathit{HistoryPruneMult} \cdot \text{numberOfPrunes}(move))\end{aligned}$$

The second move can be a capture too, but often it is a stacking move, and rarely a pass move. From the observation in Section 1.3 stacking is mostly the best choice and stacking moves should be done before capturing a stack with the height one. Capturing stacks with height at least three is mostly better than doing a stack move in the second move of a turn. Passing is reasonable only in the endgame and it should be tried at last.

For a capturing move the sort value is counted in the same way as for the first move. For a stacking move we use this formula:

$$\begin{aligned}\text{stackMove}(move, pos) = {} & \mathit{StackBonus} \\ & + \mathit{StackCountMult} \cdot \text{countType}(pos, \text{stackingType}(move))\end{aligned}$$

In other words the value is linearly dependent on the count of stacks of the same type as the resulting stack has and the multiplier is the constant *StackCountMult* (default value is only 3). Since capturing stack types with small count on the board can often lead to a win, the constant *StackBonus* (with default value 6) is added to the value of a stacking move.

Different values of constants are tested in Section 4.1.2.

3.7 Versions of Waltz

As shown in Chapter 4, Waltz is able to play on the level of best players on Boiteajeux.net. Thus for beginner and intermediate players we want to have versions that are not so strong. We created four versions: beginner, intermediate, expert, and unbeatable. We describe the versions and which algorithms they use.

The expert and unbeatable versions have all features mentioned above (Alpha-beta to the depth according to the time estimation, DFPNS, the best evaluation function ...). They only differ in the time limit¿ the expert version has 30 seconds and the unbeatable version has 300 seconds[2]. For the first three turns the Randomized Alpha-beta proposed in Section 2.1.3 is used with margin 20 (and without the History Heuristic and the NegaScout), then we run Alpha-beta without randomness, but with all enhancements. When the number of stacks is at most 23, DFPNS is used. If it does not find a solution, Alpha-beta is called also with the time limit of 30, or 300 seconds. If DFPNS search disproves the position, we call Alpha-beta to find a move leading to the deepest loss.

The intermediate version does not use DFPNS and it has only some features of Alpha-beta based algorithms (like the best evaluation function, but not the History Heuristic and the NegaScout). It searches only to the depth limited by two and half turns, i.e., five moves, and the Randomized Alpha-beta is used in the whole game (the margin is again 20).

The beginner version is similar to the intermediate, but more dumb. It also does not use DFPNS, the Negascout and the History Heuristics. Moreover, it searches to the depth of only two turns, i.e., four moves, the margin for randomly selecting between the best moves in Randomized Alpha-beta is 5 000, and most importantly it has a very simple evaluation function which consists only of the incremental part of the best evaluation function (also with worse constants). Thus the evaluation is based mostly on the material part.

[2]The time limits are occasionally a little exceeded due to search time underestimation.

4. Waltz's Results

In this chapter we show how our implementation of the algorithms described in Chapter 3.2 is successful. First we give results how the enhancements of the algorithms and their parameters make Waltz quicker. The next section shows results against other programs that are available on the Internet and that are briefly described in Section 1.5 In the last section we give results against human opponents on Boiteajeux.net (BAJ). From the results we can conclude that Waltz is able to play on the level of best human players on BAJ server. We can also see that this program is now very likely the best available AI for Tzaar.

4.1 Experiments with Waltz

This section shows the results of search speed optimization. We measure the runtime for various combinations of enhancements for the search algorithms and for different values of parameters for these enhancements. Then we choose the enhancements and values that were best in experiments.

For parameter tuning and measuring the runtime we use two sets of Tzaar positions which can be downloaded from our website [23]. The first set, we call it *MidSet*, consists of 200 middle game positions with exactly 41 stacks on the board. It is intended for testing Alpha-beta. We got the positions from Waltz's games on BAJ.

For experimenting with DFPNS we have a set of 713 endgame positions with less than 27 stacks on the board, we call it *EndSet*. We took these positions from Waltz's games with strong and intermediate players on BAJ. This set contains both easy positions (Waltz solves them quickly) and hard positions (neither DFPNS, nor Alpha-beta are able to find a solution within a minute). Some positions are won for the player on turn, some are won for the other player. We remark that overall most positions are solved by DFPNS either in a time less than a second, or in time much more than one minute (which means practically that they cannot be solved by DFPNS).

In following sections we show tables with search durations in seconds for different enhancements or parameter settings for Alpha-beta and DFPNS. A time in a table is a sum of durations of searching each position in one of the test sets. Note that when we test the influence of a constant on the search duration, other constants have their default values.

For automatically doing experiments, a *testing program* was used. For each position in the given set it starts the search and then reads the search duration from the output (without the time spent on loading a position and saving results of the search).

The parameter or enhancement setting is compiled in Waltz itself, but the algorithm used for the search and the time limit can be specified via command line parameters. The testing program outputs the sum of search durations of all positions in the set.

We performed the tests on a Dual-Core AMD Opteron 2216 server with 64 GiB of memory, but we used only one of its cores and a small part of memory.

4.1.1 Alpha-beta Enhancements

Each middle game position in the test set was run to depth of 3 turns of a player, i.e., 6 moves. The depth of 6 moves was chosen, because it is often hard to search—the search duration for that depth is mostly above 30 seconds, but it only rarely exceeds two minutes, thus the depth is feasible for Waltz. Depth 5 is usually done within a second in the middle game and, on the other hand, depth 7 would last at least 30 times longer than depth six, thus it is mostly not feasible for Waltz in the middle game.

Table 4.1 shows the efficiency of the Alpha-beta enhancements. Constants used by the algorithms were set according to another statistical experiments (see default values of constants in the next section). Each *MidSet* position was searched to the depth of 3 turns, i.e., 6 moves. The depth of 6 moves was chosen, because it is often hard to search—the search duration for that depth is mostly above 30 seconds, but it only rarely exceeds two minutes, thus the depth is feasible for Waltz. Depth 5 is usually searched within a second in the middle game and, on the other hand, searching to depth 7 would last at least 30 times longer than depth six, thus it is mostly not feasible for Waltz in the middle game.

Enhancements	Duration [s]
All enhancements (ID, PVM, MO, HH, NS)	3 800
Without NegaScout (ID, PVM, MO, HH)	4 134
Without History Heuristic (ID, PVM, MO, NS)	8 906
Without NegaScout and History Heuristic (ID, PVM, MO)	9 282
Randomized Alpha-Beta with ID, PVM and MO	9 803
Only Principal Variation Move and Iterative Deepening	22 712
No enhancement (even without ID)	36 764
Only Iterative Deepening	37 586
Only Move Ordering and Iterative Deepening	38 189

Table 4.1: Duration of the Alpha-beta search with various enhancements combinations.

From these experiments we may conclude that it is best to use all Alpha-beta enhancements, i.e., the Transposition Table with the Principal Variation (PVM), the Iterative Deepening (ID), the Move Ordering (MO), the History Heuristic (HH) and the NegaScout (NS). We observe that PVM and HH are the most important. Also note that NegaScout without HH is only a little better than not using NegaScout at all, i.e., only using ID, PVM and MO.

The 7th and 8th rows of the table show how much time the previous iterations of ID cost (in this case it is the search to depths from 1 to 5), since using only ID without PVM is useful only for the time estimation.

4.1.2 Alpha-beta Enhancements Parameters

In this section we show how constants have influence on the Alpha-beta search duration. The only enhancements with parameters are the Move Ordering, the History Heuristic and the Transposition Table. The tests were done on *MidSet* with the depth limit of 6 moves, i.e. the same way as in the previous section.

For the Move Ordering, there are three important constants in the formulas in Section 3.6: *StackBonus* (the default value is 6, Table 4.2), *StackCountMult* (the default value is 3, Table 4.3), and *CaptureCountMult* (the default value is 20, Table 4.4). From various constant settings, we can see the importance of the Move Ordering on the search duration.

StackBonus	0	3	6	9	12	15	21	30
Duration [s]	3892	3865	3875	3847	3843	3812	3814	3817

StackBonus	39	51
Duration [s]	3846	3860

Table 4.2: Duration of the Alpha-beta search with different values of the constant *SortStackBonus*.

From Table 4.2 we observe that setting *SortStackBonus* too high or too low, i.e., taking capture moves earlier or later in the order, increases the search duration.

StackCountMult	1	2	3	4	5	6	7
Duration [s]	3924	3862	3852	3825	3774	3765	3749

StackCountMult	8	10
Duration [s]	3741	3796

Table 4.3: Duration of the Alpha-beta search with different values of the constant *SortStackCountMult*.

Table 4.3 shows that taking moves creating piece types with more stacks later decreases the search time.

CaptureCountMult	1	2	4	6	8	10	15
Duration [s]	5285	4682	4190	3892	3809	3749	3759

CaptureCountMult	18	20	25	30
Duration [s]	3806	3852	3874	3913

Table 4.4: Duration of the Alpha-beta search with different values of the constant *SortCaptureCountMult*.

The History Heuristic is also involved in the Move Ordering. Durations for different values of the multiplier that determines the influence of HH in the Move Ordering formula for the first move of a player are shown in Table 4.5. Quite surprisingly, the best value is 5 and it is smaller than the best value for the constant *SortCaptureCountMult* which is approximately 15. The default value was 20 before the experiments.

Table 4.6 shows the influence of the size of the Transposition Table on the search duration. We observe that the size of 2^{18} is sufficient for searching for searching to depth 6 and setting the size higher does not help much. Note that there is often much time spent on memory allocation for higher sizes of TT—with the size of 2^{24} the search took 3845 s while for 2^{25} it took 3906 s. The default value is 2^{19}.

HistoryPruneMult	3	5	10	15	20	25	30
Duration [s]	3827	3821	3845	3876	3886	3899	3919

Table 4.5: Duration of the Alpha-beta search with different values of the constant $SortHistoryPruneMult$.

Binary logarithm of TT size	12	13	14	15	16	17	18
Duration [s]	4360	4176	4064	4003	3977	3911	3883

Binary logarithm of TT size	19	20	21	22	23	24	25
Duration [s]	3844	3841	3833	3838	3847	3845	3906

Table 4.6: Duration of the Alpha-beta search with different sizes of the Transposition Table represented by binary logarithms.

4.1.3 DFPNS Enhancements

Table 4.7 shows the importance of enhancements for DFPNS. Note that there is nearly no difference between Dynamic Widening (DW), Weak PNS (WPNS) and Heuristic Weak PNS (HW PNS), and that one single enhancement is still not enough. Surprisingly, sorting moves heuristically using the same algorithm as in Alpha-beta is useful. We ran the tests on *EndSet* positions with the time limit of 60 seconds.

Enhancements	Solved (out of 713)
HW, ET and EFB	484
WPNS, ET and EFB	484
DW, ET and EFB	480
HW, ET and EFB without sorting moves	465
Only $1 + \varepsilon$ Trick	343
Without enhancements	336
Only Heuristic Weak	289
Only Evaluation Function Based	289

Table 4.7: Results of the DFPNS search with different enhancements.

4.1.4 DFPNS Enhancements Parameters

Now we show how the constants of DFPNS enhancements have influence on the solvability. The constants were already set empirically and by doing similar experiments. The experiments are mostly done with Heuristic Weak PNS, $1 + \varepsilon$ Trick, and Evaluation Function Based DFPNS (unless otherwise stated). We used the positions from *EndSet* and the time limit of 60 seconds again.

We experimented with the size of the Transposition Table (TT), the results are shown in Table 4.8. The default size was 2^{20} and when we changed it to 2^{25}, the number of solved positions increased only by 4. On the other hand, the size at least 2^{19} is needed.

For the $1 + \varepsilon$ Trick, only the value of constant ε is important. Table 4.9 shows the number of solved positions for different values of the constant ε. The default

Binary logarithm of TT size	18	19	20	21	22	23	25
Solved (out of 713)	467	482	484	485	488	487	488

Table 4.8: Results of the DFPNS search with various sizes of Transposition Table represented by binary logarithms.

value was 0.125, but from the experiments we can conclude that best values are surprisingly at least 4. Note that function defined by these values is not convex opposed to functions in the other tables.

Value of ε	64	32	16	8	4	2	1	0.5	0.25
Solved (out of 713)	487	487	487	487	487	486	485	483	483

Value of ε	0.125	0.0625
Solved (out of 713)	484	476

Table 4.9: Results of the DFPNS search with various values of the constant ε.

There are three constants in Evaluation Function Based DFPNS called A, B (multipliers in the step function for proof and disproof numbers) and T (the threshold). Results for different values of these constants are shown in tables 4.10 (the default value of A is 10), 4.11 (the default value of B is 10), and 4.12 (the default value of T is 50 000 000). We can see that the best value for T is very high (half of the value of a win), A should be very low (less than 10) and B should can be anything higher than approximately 20.

EFB DFPNS constant A	1	3	5	8	10	12	15	20
Solved (out of 713)	484	484	484	482	484	471	471	468

Table 4.10: Results of the DFPNS search with various values of the constant A for EFB DFPNS.

EFB DFPNS constant B	1	3	5	8	10	12	15	20	50
Solved (out of 713)	296	397	443	470	484	483	483	484	484

Table 4.11: Results of the DFPNS search with various values of the constant B for EFB DFPNS.

EFB DFPNS constant T	10^5	10^6	10^7	10^8	10^9
Solved (out of 713)	293	351	387	484	485

Table 4.12: Results of the DFPNS search with various values of the constant T for EFB DFPNS.

We also tried this enhancement with a step function that have more steps, but it did not help to increase the number of solved positions.

Heuristic Weak PNS has two parameters: the threshold T for the step function and the multiplier H. From the experiments we observed that the best values are $h = 1$ and $t \geq 10^6$—the value of a position in which a player has a significant advantage. We remark that the less H is, the algorithm behave more

like Weak DFPNS without the step function. We tried this enhancement with a step function that have more steps, but it did not help to increase the number of solved positions.

Heuristic Weak DFPNS constant H	1	2	3	5
Solved (out of 713)	484	444	374	301

HW DFPNS constant T	10^4	10^5	10^6	10^7	10^8	10^9
Solved (out of 713)	468	474	484	484	484	484

Table 4.13: Search results with different values of the constant H and T for the Heuristic Weak DFPNS.

As we can see from Table 4.7 the Dynamic Widening has nearly the same solvability as Heuristic Weak DFPNS (both used with the $1 + \varepsilon$ Trick and the Evaluation Function Based enhancement). We experimented with the the constant J (the number of disproof numbers taken into account in increasing order by proof numbers of their nodes) and observed that J should be less than 10 and that the best value is 3. Table 4.14 shows the search duration for different values of the constant J

Dynamic Widening constant J	2	3	4	5	7	8	10	15
Solved (out of 713)	481	483	480	480	480	480	467	457

Table 4.14: Results of the DFPNS search with different values of the constant J for the Dynamic Widening.

4.1.5 DFPNS versus Alpha-beta in Endgames

DFPNS was designed to find long winning strategies where the player can force his opponent to have only a little number of possible moves. We tried DFPNS on Tzaar endgames, although Tzaar has relatively high branching factor even in endgames (see Section 1.4) and it is hard to guess for DFPNS which move is worth trying. On the other hand, the player can sometimes force his opponent to have a small number of moves.

To decide whether to use Alpha-beta or DFPNS in endgames we ran statistical experiments. Using the best possible setting of constants in DFPNS, it solved 495 out of 713 positions on *EndSet*. Then we tried Alpha-beta (with all enhancements) and it solved 506 positions. There are 20 positions which DFPNS solved and Alpha-beta did not, so DFPNS is meaningful to use in Waltz.

Thus in the program we try to use DFPNS first (for a part of the time limit) and if it does not succeed because of the time limit, we call Alpha-beta. To deal with lost positions that were disproved by DFPNS we call Alpha-beta on them too, see Section 2.3 for theoretical details.

4.2 Playing with Other Programs and People

In this section we show how is our program successful against human and artificial opponents. We let Waltz play on a game server against people and in a program

against other programs for playing Tzaar.

4.2.1 Different Robot Versions against Each Other

To check whether the Waltz versions are set well, they played games against each other in the program called *Arena*. The program manages many games between two versions of Waltz.[1] Each game starts in the fixed starting position and it is random who is white.

The beginner version was beaten by the intermediate version 33 times and won 5 times, the intermediate version was beaten by the expert version 21 times and won 4 times and the expert version was defeated by the unbeatable version 19 times and won 6 times. From these results we conclude that the strength of robot's versions increases according to names of the versions.

4.2.2 Results against Computer Opponents

We tested Waltz against other existing programs for playing Tzaar that are available up to March 6, 2013: HsTZAAR [24] and programs of teacher and students from University of Alaska [31] ((Mockinator, Mockinator++, BiTzaarBot and GreensteinTzaarAI). See Table 4.15 for the results.

Program	Wins	Losses	Note
HsTZAAR	479	121	With algorithm `pscout_full_4` on 4 cores.
HsTZAAR	65	18	With algorithm `pscout_full_5` on 5 cores.
GreensteinTzaarAI	342	53	We could not set the time limit.
BiTzaarBot	196	5	The time limit was 40 seconds.
Mockinator++	83	2	The time limit was 40 seconds.
Mockinator	82	1	The time limit was 40 seconds.

Table 4.15: Results of Waltz against other Tzaar-playing programs. Wins and losses are counted from the Waltz's point of view.

During the tests, Waltz had time limit of 30 seconds. Each game started on a random starting position. We performed tests with HsTZAAR on Intel Xeon ES-1620 server with 64 GiB of memory and tests with the other programs on a AMD Turion II P560 Dual-Core notebook with 4 GiB of memory.

There is also program TZ1 and robots on BoardSpace.net [3] available, but due to their design it is not possible to run automatic play-outs between them and Waltz. Based on some games that we played with these programs we believe that Waltz can win most of play-outs against them. Robots on BoardSpace.net win only rarely against an experienced player as one can see from their results.

4.2.3 Results on Boiteajeux.net with Human Opponents

To test Waltz against people we chose the game server Boiteajeux.net (BAJ) [28]. More than 35 turn-based games including Tzaar are played there in the way similar to *play by email* (a player send moves to his opponent by an email and then waits for the answer). Instead of emails, an HTML interface is provided.

[1] It was also used for deciding which parameters are better for the evaluation function.

For each game, an ELO rating is counted. For a win a player obtains some ELO points according to his and opponent's ELO and his opponent loses the same number of points (it is always at least one point). New player receives ELO 1 500. Robots exist on BAJ only for a few games and Waltz is the only one for Tzaar there.

There are two main reasons why to choose BAJ for the robot. Mainly a lot of people play Tzaar there—23 871 Tzaar games were finished on BAJ from October 31, 2008 to March 6, 2013. The second is that robot's communication with the server can be done by a simple HTTP client (it is checking games where the robot is on turn and then playing moves). The other possibility instead of BAJ is BoardSpace.net which run under Java, so one cannot create an HTTP client. Java robot would be also slow.

We created four different versions of Waltz: for beginners, for intermediate players, for experts and "unbeatable" that are described in section 3.7.

Now we describe how Waltz was successful against human opponents on BAJ.

We released Waltz in the expert version on March 20, 2012 under username *Pauliebot* and it was under development until April 4, 2012. After that we made only minor updates, mostly improving the evaluation function. On April 24, 2012 we released the other versions of Waltz (usernames are *PauliebotBeginner*, *PauliebotMedium* and *PauliebotUnbeatable*). New versions of Waltz are announced on BAJ forum for Tzaar.

ELOs of players and other data in this section were up to the date April 30, 2013. The results can be found on Boiteajeux.net server using a search in the left menu. To play with the robot, add the robot's username in the field Guests on the page for creating new games.

The beginner robot with username *PauliebotBeginner*, ELO 1 556 and 230 finished games is probably the most popular, because it is challenging even for intermediate players. For example rupelboom (ELO 1 827) won against it 14 times and lost 3 times, but tchako (ELO 1 813)lost with the robot 8 times. Beginners, i.e., players with only a few finished Tzaar games, mostly do not win against the beginner robot. Thus the robot might have to play weaker, but I was not able to think out how to do it without making serious mistakes and still playing on the level of real beginners. The depth to which the robot searches is discussable.

The intermediate robot *PauliebotMedium* (ELO 1 776) played only 70 games. For example rupelboom (ELO 1 827) won all five games against the intermediate robot, PhilDakota (ELO 1 701) won eight games and lost six games and surprisingly, tchako (ELO 1 813) won two times and lost four times. Thus the robot is challenging for intermediate players (ELO around 1 800), but it can be sometimes defeated by tchako who is losing with the beginner robot. The latter is probably caused by a different robot's strategy—tchako can defeat the intermediate's one, but not the beginner's one.

The expert version played 157 games so far.[2] It won 117 of them and it is the 14th best Tzaar player with ELO 2 083. Most important results of the expert version are in of Table 4.16. We conclude that the expert version played on the level of best players on BAJ, but sometimes intermediate players were able to defeat Waltz.

[2]Some of these games were played for testing purposes.

Player	Rank	ELO	Wins	Losses	Note
SlowBrain	1st	2 467	1	3	He is far better than other players.
Gambit	2nd	2 272	2	4	Two games lost during development.
Paulie	3rd	2 220	8	3	Waltz mostly wins in offline games.
evrardmoloic	17th	2 062	1	1	He was 2nd in the summer 2012.
PhilDakota	68th	1 701	14	3	
mat76	72th	1 692	19	1	
Gregg	101th	1 618	14	5	

Table 4.16: Some results of the expert version. Wins and losses are counted from the Waltz's point of view.

The unbeatable Waltz version has ELO 2 072, the 15th highest, and played 76 games from which it won 46 games.[3] The most important results of the unbeatable version are in Table 4.17. The 9 wins against SlowBrain are a great success because SlowBrain is far better than other players. From these results we conclude that more time to search helps Waltz to play better. This was confirmed also by games between the unbeatable and the expert versions where the unbeatable won 65 of 99 games.

Player	Rank	ELO	Wins	Losses
SlowBrain	1st	2 467	9	18
Paulie	3rd	2 220	1	3
mnmr	5th	2 184	0	1
Zeichner	9th	2 143	1	0
Talisac	11th	2 123	3	0
azazhel	26th	1 931	8	2

Table 4.17: Some results of the unbeatable versions. Wins and losses are counted from the Waltz's point of view.

The most frequently appearing reason why Waltz lost some games on BAJ was the loss of the last stack of Tzarras. We observed that in two or three last turns of these lost games Waltz had no chance to create a stack of Tzarras which could not be captured by the opponent—Waltz was probably not aware of such an opponent's trap soon enough. Another bad thing in Waltz's behavior during these games was losing quite high stacks (size 3, 4, or even 5) during the middle game.

We thus tried to improve the evaluation function to avoid these problems. The version with the enhanced evaluation function won 136 and lost 102 games against the version with the old evaluation function. On March 4, 2013 we released the version with enhanced evaluation function.

[3]Some of these games were against other Waltz versions—this was done to increase robot's ELO, otherwise strong players would not want to play against an opponent with low ELO.

Conclusion and Future Work

As we observed in Chapter 4 we created a robot that is able to play on the level of best players and that can defeat all other available programs. We conclude that the Alpha-beta algorithm is quite successful in the domain of Tzaar and maybe in the domains of games that have large branching factor and a good evaluation function is not so hard to implement (unlike Go).

We also tested the Depth-first Proof-number Search (DFPNS) on endgame positions that have mostly still quite a big number of possible moves. We have found that performance of this algorithm is similar to Alpha-beta in endgames—sometimes DFPNS solves endgame positions quicker and sometimes Alpha-beta is quicker. The latter is caused by the high branching factor and the fact that there are often not enough forcing moves in Tzaar.

We dealt with two problems specific for playing with people. We proposed Randomized Alpha-beta, a new algorithm for randomly choosing a good enough move, and we discussed how to play in lost positions. We also gave a new enhancement for DFPNS called Heuristic Weak PNS which was developed from Weak PNS.

There are plenty of things left for future work and research. One can always adjust the evaluation function together with other constants in the program. Maybe there are positional or material properties that should be added to the evaluation function to make it even better. Also the implementation of algorithms can be made better, for example we do not probably need to generate all moves every time, but we can maintain a list of possible moves for each player and modify it according to moves that are played. From the algorithmic view there are many Alpha-beta enhancements invented mostly for Chess that were not tested in the domain of Tzaar, e.g., maybe the Quiescence search can help.

DFPNS has performance similar to Alpha-beta in endgames, but it is left to future research to adjust DFPNS for games with a large branching factor, probably using some heuristics like the Move Ordering. The new algorithm Heuristic Weak PNS turned out not to be very successful for Tzaar, but in another domains such as Othello and Shogi, it may help.

The Monte Carlo Tree Search (MCTS) is probably the most promising approach, and we consider it to be the next direction where we would like to move Waltz's development. This algorithm is very successful for Go (the best Go programs use it) and Go has also a big branching factor as Tzaar. Hence MCTS is worth trying for Tzaar, although in the endgame it can be better to try DFPNS or Alpha-beta that can solve the position deterministically.

Another direction lies in parallelizing Waltz's algorithms, which is a natural step we would like to try—both DFPNS and Alpha-beta can be made parallel.

Acknowledgments

I would like to thank my advisor Tomáš Valla for showing me the Project GIPF, particularly Tzaar, for weekly meetings where we discussed our work on Waltz and for his help with creating and submitting a paper about Waltz to the conference Computer Games 2013. Jitka Novotná and Pavel Dvořák were also

on the weekly meetings and they helped me with some fresh ideas. Petr Baudiš and Jan Hric had a seminar about game algorithms during the winter semester 2011/2012 where I learned about state-of-art algorithms for board games.

I also would like to thank the Department of Applied Mathematics for a place and computation time on a server on which Waltz runs and on which the experiments took place. Waltz plays with people on the french game server Boiteajeux.net despite the fact that robots are not "officially supported" there. Some players on this server tried to defeat Waltz, a few of them more than several times, and thus they gave me a valuable feedback on Waltz's strength.

Bibliography

[1] 2008 golden geek best 2-player board game nominee. BoardGameGeek [cit. 2012-07-13], http://boardgamegeek.com/boardgamehonor/8763/2008-golden-geek-best-2-player-board-game-nominee

[2] 2008 nominees - general strategy. International Gamers Awards [cit. 2012-07-13], http://www.internationalgamersawards.net/winners-and-nominees/nominees/2008-nominees

[3] Boardspace.net. [cit. 2012-07-13], http://www.boardspace.net/

[4] Daedalus game manager. Google Code, http://code.google.com/p/daedalus-game-manager/

[5] Nederlandse spellenprijs - historie. [cit. 2012-07-13], http://www.spellenprijs.nl/historie.html

[6] TZAAR, BoardGameGeek. [cit. 2012-07-13], http://boardgamegeek.com/boardgame/31999/tzaar

[7] Allis, V.: Searching for Solutions in Games and Artificial Intelligence. Ph.D. thesis, University of Limburg, Maastricht, The Netherlands (1994)

[8] Baudiš, P.: MCTS with Information Sharing. Master's thesis, Faculty of Mathematics and Physics, Charles University in Prague (2011)

[9] Breuker, D.M., van den Herik, H.J., Uiterwijk, J.W.H.M.: Replacement schemes and two-level tables. ICCA Journal pp. 175–180 (1996)

[10] Burm, K.: Tzaar rules. GIPF project, http://www.gipf.com/tzaar/rules/rules.html

[11] Burm, K.: Tzaar strategy. GIPF project, http://www.gipf.com/tzaar/strategy/strategy.html

[12] Hashimoto, J., Hashimoto, T., Iida, H., Ueda, T.: Weak proof-number search. In: Proceedings of the 6th international conference on Computers and Games. pp. 157–168. Springer-Verlag, Berlin, Heidelberg (2008)

[13] van den Herik, H.J., Winands, M.: Proof-Number Search and Its Variants, pp. 91–118. Springer, Berlin, Heidelberg (2008)

[14] Heule, M.J.H., Rothkrantz, L.J.M.: Solving games: Dependence of applicable solving procedures. Sci. Comput. Program. 67(1), 105–124 (2007)

[15] Kishimoto, A.: Dealing with infinite loops, underestimation, and overestimation of depth-first proof-number search. In: Fox, M., Poole, D. (eds.) Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010. AAAI Press, Atlanta, Georgia, USA (2010)

[16] Knuth, D.E., Moore, R.W.: An analysis of alpha-beta pruning. Artificial Intelligence 6(4), 293–326 (1975)

[17] Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4212, pp. 282–293. Springer (2006)

[18] Pawlewicz, J., Lew, Ł.: Improving depth-first pn-search: $1 + \varepsilon$ trick. In: Proceedings of the 5th international conference on Computers and games. pp. 160–171. CG'06, Springer-Verlag, Berlin, Heidelberg (2007)

[19] Reinefeld, A.: An improvement to the scout tree-search algorithm. ICCA Journal pp. 4–14 (1983)

[20] Schrüfer, G.: A strategic quiescence search. ICCA Journal 12(1), 3–9 (1989)

[21] Schwagereit, J.: Tz1: A program to play tzaar. [cit. 2012-07-13], `http://www.johannes-schwagereit.de/tz1/`

[22] Thomsen, T.: Lambda-search in game trees – with application to go. In: ICGA Journal. pp. 203–217. Springer (2001)

[23] Valla, T., Veselý, P.: Waltz, `http://kam.mff.cuni.cz/~vesely/tzaar/`

[24] Vasconcelos, P.: HsTZAAR, `http://www.dcc.fc.up.pt/~pbv/stuff/hstzaar/`

[25] Wentink, D.: Analysis and Implementation of the game Gipf. Master's thesis, Universiteit Maastricht (2001)

[26] Winands, M.H.M., Schadd, M.P.D.: Evaluation-function based proof-number search. In: Proceedings of the 7th international conference on Computers and games. pp. 23–35. CG'10, Springer-Verlag, Berlin, Heidelberg (2011)

[27] Yoshizoe, K.: A new proof-number calculation technique for proof-number search. In: Proceedings of the 6th international conference on Computers and Games. pp. 135–145. CG '08, Springer-Verlag, Berlin, Heidelberg (2008)

[28] Boiteajeux board-gaming portal, `http://www.boiteajeux.net/`

[29] GAMES game awards. Games Magazine, `http://www.gamesmagazine-online.com/gameslinks/archives.html#2009awards`

[30] Spiel des jahres, awarded games 2008, `http://www.spiel-des-jahres.com/cms/front_content.php?idart=925`

[31] Tzaar - ai game project for 2011, `http://www.math.uaa.alaska.edu/~afkjm/cs405/tzaar/`

[32] Zobrist, A.L.: A new hashing method with application for game playing. ICCA Journal 13(2), 69–73 (1990)

List of Figures

List of Tables

List of Abbreviations

- AI — Artificial Intelligence
- BAJ — Boiteajeux.net
- DFPNS — Depth-first Proof-number Search
- DGM — Daedalus Game Manager
- DN — Disproof number
- DW — Dynamic Widening
- EFB DFPNS — Evaluation Function Based Depth-first Proof-number Search
- GHI — Graph History Interaction
- HH — History Heuristic
- HW — Heuristic Weak
- ID — Iterative Deepening
- MCTS — Monte Carlo Tree Search
- MO — Move Ordering
- NS — NegaScout
- MPN — Most Proving Node
- PNS — Proof-number Search
- PN — Proof number
- PV — Principal Variation
- TT — Transposition Table

A. Documentation of Waltz

Waltz consists of the library for searching for best moves (described in Section A.3), the program `tzaar` that loads the board from a file and starts the search (Section A.2) and the Python web client for communication with Boiteajeux.net (Section A.1). For simple building of the project, a Makefile is provided.

The source code of Waltz can be downloaded from our website [23].

A.1 Python Web Client for Boiteajeux.net

Web client for communicating with the website Boiteajeux.net is written in Python. It downloads a page from BAJ and looks for a game in which Waltz is on turn. When such a game is found the client downloads the page with board and calls the function `PlayGame`.

Function `PlayGame` has a parameter page containing HTML code of the page with board. From the HTML code it parses a position, converts it to the board representation (described in Section A.2.1) and saves it to a file. Then it calls the program `tzaar` and waits until the computation is done. Finally it loads best moves from a file and executes them on BAJ (via HTTP POST requests).

The client also looks for an invitation for playing—one can start a game with Waltz by creating a new game and adding the robot's username in the field Guests. The client sends email to a given recipient when an exception occurs, and every day after midnight it sends statistics of searches done during the day.

Note that before using the client for BAJ, we have to configure it. That means, inside the Python code, fill the robot's username and password, an email where to send error messages and daily outputs, and directories where is Waltz located and where to save positions and outputs.

A.2 Program `tzaar`

The program `tzaar` written in C is quite simple. Given a file with a position in Tzaar it initializes arrays and variables with a board representation that are in the library. Then it calls function `GetBestMove` in `lib` for searching for best moves and saves returned moves to a file.

The files and other options are given via command line arguments:

- `-a N` or `--ai N` — set the algorithm number `N` (see lib.h). The default AI number is in constant `MAINAI` in lib.h (it is AI used by the expert robot).
- `-b FILE` or `--bestmove=FILE` — search for best moves in a position stored in `FILE` and then save best moves into this file. An input file format is described in Section A.2.1 and output file format in Section A.2.2. This is a required option.
- `-e FILE` or `--execute=FILE` — execute best moves after searching for them and then save the position to `FILE`. Default is not to save any position after the search.

- `-t SECONDS` or `--timelimit=SECONDS` — set the time limit of the search to `SECONDS`. The default value is in the constant `AITIMELIMIT` (30 seconds).
- `-h` or `--help` — print usage.

A.2.1 Board Representation and Input Files

Now we describe a board representation and then the format of input and output files.

The Tzaar board is a hexagon with 5 fields on each side, but we cannot simply store it in the memory as a hexagon. Thus it is sloped to be fit in the quadratic array 9×9. In the library, one linear array of size 81 is used instead of a quadratic array.

Example of the array containing fixed starting position (array members are separated by spaces):

```
-1   1   1   1   1  100 100 100 100
-1  -2   2   2   2  -1  100 100 100
-1  -2  -3   3   3  -2  -1  100 100
-1  -2  -3  -1   1  -3  -2  -1  100
 1   2   3   1  100 -1  -3  -2  -1
100  1   2   3  -1   1   3   2   1
100 100  1   2  -3  -3   3   2   1
100 100 100  1  -2  -2  -2   2   1
100 100 100 100 -1  -1  -1  -1   1
```

The number 100 stands for a field outside the board and also for the field in the middle of the board. Empty fields with no piece have number 0. Other numbers stand for different types of pieces:

- 1 is a white Tott,
- 2 is a white Tzarra,
- 3 is a white Tzaar,
- -1 is a black Tott,
- -2 is a black Tzarra,
- -3 is a black Tzaar.

In this array, a player can move in six directions: horizontally left, or right, vertically up, or down, and diagonally right and down, or left and up. The other diagonal directions (right and up, left and down) are not possible.

The heights of stacks are in another array of the same size. Fields outside the board and empty fields have stack height zero. Example for the fixed starting position:

```
1 1 1 1 1 0 0 0 0
1 1 1 1 1 1 0 0 0
1 1 1 1 1 1 1 0 0
```

```
1 1 1 1 1 1 1 1 0
1 1 1 1 0 1 1 1 1
0 1 1 1 1 1 1 1 1
0 0 1 1 1 1 1 1 1
0 0 0 1 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

In the `lib`, there is also an array with names of fields (`"-"` is a field outside the board):

```
"A1", "B1", "C1", "D1", "E1", "-" , "-" , "-" , "-" ,
"A2", "B2", "C2", "D2", "E2", "F1", "-" , "-" , "-" ,
"A3", "B3", "C3", "D3", "E3", "F2", "G1", "-" , "-" ,
"A4", "B4", "C4", "D4", "E4", "F3", "G2", "H1", "-" ,
"A5", "B5", "C5", "D5", "-" , "F4", "G3", "H2", "I1",
"-" , "B6", "C6", "D6", "E5", "F5", "G4", "H3", "I2",
"-" , "-" , "C7", "D7", "E6", "F6", "G5", "H4", "I3",
"-" , "-" , "-" , "D8", "E7", "F7", "G6", "H5", "I4",
"-" , "-" , "-" , "-" , "E8", "F8", "G7", "H6", "I5"
```

The input file for the program is a sequence of numbers separated by some whitespace characters:

- first a player on turn is specified, i.e., robot's color. Number 1 stands for white, -1 for black,
- then there is the array with the board representation—81 integers with types of pieces, or 0 as an empty field, or 100 as a field outside the board,
- the last is the array with stack heights (81 nonnegative integers).

Example with comments after '%' (they should be deleted before using the file):

```
1 % our robot is white player

% pieces on the board
 1  -1  -1  -1  -1  100 100 100 100
 1   2  -2  -2   2   3  100 100 100
 1   0  -3   0  -3   0   1  100 100
 0   0   0   0   0   0   2   1  100
 0  -3   0   0 100  0   3   0   1
100 -1   0  -3   0  -1  -3   2  -1
100 100 -2   0   0   1  -3  -2  -1
100 100 100  0   2   2   2  -2  -1
100 100 100 100  1   1   1   1  -1

% stack heights
1 1 1 1 1 0 0 0 0
1 1 1 1 1 3 0 0 0
1 0 1 0 1 0 1 0 0
```

```
0 0 0 0 0 0 1 1 0
0 3 0 0 0 0 1 0 1
0 1 0 1 0 1 1 1 1
0 0 3 0 0 2 1 1 1
0 0 0 0 1 1 1 1 1
0 0 0 0 1 1 1 1 1
```

Note that in the format it does not matter on whitespace characters (spaces, new lines and tabs). The file with numbers separated by a single space and no new lines will is loaded successfully.

A.2.2 Output File with Best Moves

Now we describe how best moves are saved in the output file. On the first line, there is the first move of a turn. It is saved as the name of the field from which a player moves a stack, and the name of the field where the player captured an opponent's stack (the names are separated by a space).

On the second line, there is an integer specifying what is the second move. Number -2 stands for no move (in the case of the first turn of white player, or win after the first move), -1 stands for a pass move, 0 for a stacking move, and 1 for a capture. In the last two cases, names of two fields follow, the first is the field from which a stack is moved, and the second is the field to which the stack is moved.

On the third line, there is some information about the search, namely the search duration in seconds (with three decimal places) and the returned value.

Example of a capture move and a stacking move:

```
G4 C6
0 F2 F1
33.303 33319
```

Example of a capture and a pass move (the returned value 2 000 000 000 means that Waltz is going to win):

```
H2 D1
-1
2.742 2000000000
```

A.2.3 Module `main`

The module `main` contains function `main` which parses command line arguments using the getopt library and calls `ProcessPosition` which loads position. Then it calls `GetBestMove` in `lib` and finally it saves the result.

A.2.4 Module `saveload`

The module `saveload` contains functions for loading and saving positions in the format described in Section A.2.1 and a function for saving best moves in the format described in Section A.2.2. Function `SaveWholePosition` save all information about the position, including counts of pieces according to a type, the zone of control, but this function is not used by Waltz.

A.3 Library `lib`

The library `lib` contains the computation part of the program and it is also written in C (standard C99). For searching for best moves the algorithms Alpha-beta and Depth-first Proof-number Search (DFPNS) are implemented. There are also auxiliary functions for generating, executing and reverting moves. It is possible to choose between different versions of the algorithms, for example there are versions for beginners and intermediate players.

A.3.1 Arrays and Fields for Position Properties

For the current position representation there are arrays `board` and `stackHeights` of size 81 with the same format as described in Section A.2.1. The variable `player` is 1 when white is on turn and -1 when black is on turn. Since the turn of a player consists of two moves, the variable `moveNumber` determines which phase of a turn is: 1 is the first phase (a capture move) and 2 is the second (capture, stack, or pass).

In the variable `turnNumber`, there is the number of turns from the root position of the search (starting 1)—this is different from the number of turns in the whole game which is not known to Waltz. Moves are stored in the array `history`.

There are some variables and arrays for storing information about the current position that can be counted directly from arrays `board` and `stackHeights`, but that would be very slow. The array `counts` contains the number of stacks alive (visible) for each piece type, the variable `stoneSum` is the sum of stacks on the board and it is used by the Iterative Deepening.

The variable `value` is the value of the current position determined by the evaluation function or by a search. The variable `materialValue` is counted by the part of the evaluation function that is counted incrementally when executing or reverting moves. The variable `hash` contains the value of the Zobrist hash function for the current position and it is also counted incrementally.

The array `highestStack` of the size equal to the number of piece types contains the height of the highest stack for each type. For incrementally maintaining this array library uses the quadratic array `countsByHeight` that contains the number of pieces for each type and height.

The array `zoneOfControl` contains for each piece type how many pieces of that type can be captured with one move. This is used in the evaluation function and for determining whether a player has any possible captures when he is on turn and it is his first move. The array is updated also incrementally using the array `threatenByCounts` of the size 81 which contains for each field how many stacks can capture a stack on this field.

A.3.2 Module `lib`

The module `lib` is the main module of the library. It contains definitions of structures, types, macros, global arrays and variables used throughout the library (some of them are described in the previous section).

The function `GetBestMove` starts the search according to the chosen algorithm and does the time estimation via the Iterative Deepening for the Alpha-beta based

algorithms and for DFPNS via the estimation of the maximal number of nodes that can be searched.

In the header file, types and basic macros are defined first. Constants for properties of the game, types of AI (algorithms), and AI settings follows. The structure for a move, global variables, and arrays are defined at the end.

A.3.3 Module `moves`

The module `moves` contains functions for generating, executing and reverting moves. There are also support functions for deallocating memory (free a single move or a linked list of moves), determining whether someone won in the current position, updating the zone of control after executing or reverting a move and converting between a field index and a field name (for example the field on index 3 has name D1).

Most of functions in this module are optimized to be as fast as possible, because they are called many times during the search. Note that the functions for generating moves are used for the first and the second move of a turn separately.

In the header file there are constants for the Move Ordering and arrays for possible directions that are used in the functions for generating moves.

A.3.4 Module `init`

The module `init` has functions for initializing arrays and variables with information about the current position. The counting of the hash value, the material value (the part of the value of a position that is counted incrementally during the search) and the zone of control is implemented here. Function `InitBoard` prepares starting position according to the parameter `setup` (random, or fixed) and calls other initialization functions, but it is not used by our robot.

Functions in this module are not optimized to be fast, because they are not called during the search, only before it. The values of the parameter `setup` and the fixed starting position are defined in the header file.

A.3.5 Module `alphabeta`

The module `alphabeta` contains functions that implement the Alpha-beta pruning algorithm with its enhancements, functions for working with the Transposition Table (TT) and static evaluation functions.

There are different Alpha-beta functions with different enhancements used:

- `AlphaBeta` — simple Alpha-beta with storing positions to TT,
- `AlphaBetaPV` — Alpha-beta with TT and the Principal Variation Move (PV),
- `AlphaBetaPVMO` — Alpha-beta with TT, PV and the heuristic Move Ordering (MO),
- `AlphaBetaMO` — Alpha-beta only with the Move Ordering (without TT),

- `AlphaBetaPVMORandom` — random Alpha-beta proposed in Section 2.1.3 with TT, PV and MO. It should be called only on the root of the search tree.

- `AlphaBetaPVMONegascout` — Alpha-beta with TT, PV, MO and Negascout,

- `AlphaBetaPVMOHistory` — Alpha-beta with TT, PV, MO and the History Heuristic,

- `AlphaBetaPVMOHistoryNegascout` — Alpha-beta with TT, PV, MO, the History Heuristic and the Negascout,

- `AlphaBetaPVMOBeginner` — Alpha-beta with TT, PV, MO and the beginner static evaluation function,

- `AlphaBetaPVMORandomBeginner` — random Alpha-beta with TT, PV, MO and the beginner static evaluation function. It should be called only on the root of the search tree.

The Alpha-beta enhancement Iterative Deepening is implemented in the module `lib`. The Transposition Table use the replacement scheme Two Big.

A.3.6 Module `pns`

The module `pns` contains the implementation of the Depth-first Proof-number Search (DFPNS) with some enhancements and the Transposition Table (TT) used by DFPNS (it differ from TT used by Alpha-beta).

There are different DFPNS functions with different enhancements used:

- `dfpns` — simple DFPNS without enhancements,

- `dfpnsEpsTrick` — DFPNS with the $1 + \varepsilon$ Trick,

- `weakpns` — Heuristic Weak DFPNS (one can modify it to Weak DFPNS easily),

- `dfpnsEvalBased` — Evaluation Function Based DFPNS,

- `dfpnsWeakEpsEval` — Heuristic Weak DFPNS with the $1+\varepsilon$ Trick and the Evaluation Function Based enhancement,

- `dfpnsDynWideningEpsEval` — DFPNS with the Dynamic Widening, the $1 + \varepsilon$ Trick and Evaluation Function Based enhancement,

Constants for the enhancements are in the header file. The Transposition Table for DFPNS also use the scheme Two Big.

A.3.7 File `hashedpositions.h`

This header file contains data for the Zobrist hashing. There is a three dimensional array `HashedPositions` of unsigned 64bit integers (type thash) that contains random numbers for each combination of a field, a piece (or an empty field) and a stack height that can occur on the board. Impossible combinations have value zero.

The array is indexed in this way: `HashedPosition[field][pieceType][stackHeight]` where `field` is an index in the array `board` and it is in range from 0 to 80, `pieceType` is the value from the array `board` plus three (the value ranges from -3 to 3) and `stackHeight` is the height of a stack if there is any, or zero otherwise. Note that for fields outside the board, the values are not used.