Economics & Informatics
Erasmus School of Economics
Erasmus University Rotterdam

Master Thesis

Weighted Voting Game Design Heuristics

Author:
Allard Kleijn
303118

Supervisor:
Dr. Yingqian Zhang

Co-reader, TU Delft:
Dr. Tomas Klos

Co-reader:
Dr. Adriana Gabor

April 26, 2012

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Whenever multiple stakeholders with conflicting opinions need to make a decision some form of voting is often employed. Commonly voting is done by counting the number of votes in favor and checking if this meets a certain threshold, which is often set to the majority. A voting protocol in which every stakeholder has an equal vote can be highly desirable, for example in political elections, but is less suited for other situations.

Consider the Council of the European Union, for example, it would not make sense that a large country with many inhabitants like Germany has the same power as a small country with much fewer inhabitants like Luxembourg have an equal vote. For these situations a weighted voting game (WVG) is much more desirable.

In a WVG there is a predefined quota and each stakeholder (often referred to as 'player' in WVGs) is assigned a weight. Whenever the sum of the weights of the players in a coalition is larger than or equal to the quota that coalition is referred to as 'winning', and 'losing' otherwise. On first sight it might appear that a player with a larger weight often is more influential, but this is not necessarily correct.

One definition describes the power of a player as the number of cases in which he can turn a losing coalition into a winning coalition. We can describe the coalitions as binary numbers, where each position refers to a player. If the coalition contains a player the value at that position is 1 and 0 otherwise. For example, a coalition 011 contains players 2 and 3, but not player 1. Now consider the following example: a WVG with 3 players and the quota is set at 50, players one and two have weight 49 and player three has weight 2, represented by $[50 : 49, 49, 2]$. The list of possible winning coalitions are: 111, 110, 101 and 011. Removing player 3 from winning coalitions 101 and 011 will result in a losing coalition. For players 1 and 2 there are also two cases for which removing the player from the coalition results in a losing coalition. All players are crucial in two out of four cases and therefore share equal power, while the weight of player 3 is $24,5$ times smaller than that of the other players.

Ideally, one should be able to compute a player's power for any given WVG.

We review our previous example once more: $[50 : 49, 49, 2]$. It quickly becomes clear that determining the power from a given weight distribution is complicated. This is referred to as the forward problem.

Consider our example again: $[50 : 49, 49, 2]$. Now consider we are not interested in finding the power distribution for this weight distribution, but we are interested in finding the weights distribution that yields $[50 : 49, 49, 2]$ as a power distribution. In other words, what vector of weights do we require that yields a desired distribution of power? This we refer to as the inverse problem. Although the forward problem and inverse problem are related, they should be considered as different problems. Consider the example of the European Council once more, if we use the number of inhabitants per country as the desired power distribution and we are able to compute the weights and quota that yields the given power distribution the European Council is able to vote with the 'right' distribution, given the definition of power provided.

De Keijzer [3] designed an algorithm for the inverse problem. He takes a target power distribution for all agents and finds the WVG with the closest matching power index by enumerating all weighted voting games for n players and computing the power in each game. Multiple studies on local search algorithms have also been performed, but the algorithm designed by De Keijzer is the only known method that guarantees an exact solution. This algorithm enumerates WVGs by starting with a single WVG and expanding it. A game can be the 'parent' and/or the 'child' of another game, building a tree structure. De Keijzer implemented both a breadth-first and depth-first approach to generate this tree. In this research we are interested in designing heuristics for the selection of the order in which games are expanded. The goal of using these heuristics is finding a 'good' solution 'faster', which we refer to as error convergence. Note that 'good' and 'faster' are relative concepts, therefore we test error convergence between two or more methods. Additionally, we test the influence of certain parameters on the effectiveness of the heuristics.

1.1 Contributions

In this paper we provide the following contributions:

- We provide extensions to the algorithm designed by De Keijzer.
- We provide an performance evaluation of the aforementioned extensions.
- We provide an evaluation of the influence of different parameters on the extensions
- We provide statistical support that indicates one of the extensions, using the correct parameters, provides greater error convergence than De Keijzer's original algorithm.

1.2 Outline

This thesis is divided into seven chapters. Chapter 2 provides a formal problem definition. Firstly, we will define a weighted voting game. Secondly, we will discuss different existing power indices, namely Banzhaf and Shapley-Shubik. Thirdly, we will provide a formal definition of the problem. Fourthly, we will discuss the representation of games. Chapter 3 discusses related work in the fields of the weighted voting game design problem and tree search techniques. Chapter 4 focuses on the algorithm designed by De Keijzer, as it is the base on which the heuristics are built. Chapter 5 discusses various designs for search heuristics, which can be applied to the field of WVG design. Chapter 6 provides an overview of the experiments and results performed with the heuristics. Chapter 7 concludes this thesis, with a discussion and our ideas for future work.

Chapter 2

Problem definition

This chapter gives some preliminary concepts that are vital in the understanding of the problem discussed in this thesis. Additionally, we state the problem with current approaches and a method to overcome this.

2.1 Preliminaries

In order to understand what Weighted Voting Games are we must first define some other types of games.

A cooperative game is a pair (N, v), where N is a finite set of players. Coalitions are subsets of N and v is known as the gain function or characteristic function, where $v : 2^N \to \mathbb{R}^{\geq 0}$ every coalition to a non-negative real number. Therefore, v describes the collective payoff the players of a certain coalition obtain by cooperating.

N is referred to as the grand coalition, which contains all players.

A subset of cooperative games are the simple games. In a simple game every coalition S is either winning or losing. Therefore the characteristic function can be defined as $v : 2^N \to \{0, 1\}$, where $v(S) = 1$ is a winning coalition and $v(S) = 0$ is a losing coalition.

Monotone games are a subset of simple games, defined by an additional characteristic. In a monotone game a coalition that is winning can not be made losing by adding players to it.

Weighted voting games (WVGs) are monotone games which have a weighted form. In these games every player has a given weight and a quota determines if a coalition is winning or losing. Note that not all monotone games have a weighted form. Therefore, all weighted voting games are monotone games, however not all monotone games are weighted voting games.

Canonical weighted voting games (CVWGs) are a specific type of weighted voting games. A WVG is considered an canonical weighted voting game if and only if the weights vector is non-increasing. This implies that the weight of player i has a smaller or equal weight when compared to the weight of player

$i - 1$.

A minimal winning coalition (MWC) is a winning coalition in which no player can be removed without making it a losing coalition. Similarly, a maximal losing coalition is a losing coalition to which no player can be added without making it a winning coalition.

Right truncations are operations that can be performed on a canonical coalition of players. An ith right truncation removes the i highest-numbered players from the coalition. Consider the following example: let C be a 4 player coalition 1011, where every position of this binary number indicates if that player is in the coalition. In this case the coalition contains players 1,3 and 4. The 2^{nd} right truncation of C is 1000.

2.2 Representation of games

The types of voting games presented in the previous section can be represented in a number of ways. These representations offer a method to describe voting games. Note that a coalition is described as a binary number with a length equal to the number of players. Each position describes whether the coalition contains the player at that position. A coalition containing players 1,3 and 5 out of 6 possible players would therefore look as follows: 101010.

- The winning coalition form can be used to represent simple games, which lists all winning coalitions. Let W be the set of winning coalitions then $W = \{100, 010, 001\}$ could be an example winning coalition form of a simple game.

- The losing coalition form is similar to the winning coalition form. It lists all the losing coalitions for a simple game. The set of losing coalition is denoted by L.

- The minimal winning coalition form can be used to represent monotonic games. This representation lists the minimal winning coalitions. The set of minimal winning coalitions is denoted by $W\min$.

- The maximal losing coalition form, similarly to the minimal winning coalition form, lists the maximal losing coalitions. Note that both minimal winning coalition form and maximal losing coalition form only fully describe monotonic games, not simple games. The set of maximal losing coalitions is denoted by $L\max$.

- The weighted form can be used to represent weighted voting games in the following format $[q : w_1, ..., w_n]$, where q is the quota and $w_1, ..., w_n$ denote the weights for players 1 through n.

2.3 Power indices

As already discussed in Chapter 1 the weight of a player does not equal its power. The power of a player is computed using a power index. Throughout the literature different power indices have been proposed. Two commonly used indices are: the Banzhaf index and the Shapley-Shubik index. We will discuss these power indices in this section.

2.3.1 Banzhaf index

The Banzhaf index is one of the commonly used power indices. It considers in how many cases a player is able to transform a losing coalition into a winning coalition by joining it. This value, called the raw Banzhaf index, is not easily comparable due to its unnormalized form. Therefore the normalized Banzhaf index is used whenever a comparison between two or more Banzhaf indices is required.

$$\beta_i = \frac{\beta'_i}{\sum_{j=1}^{n} \beta'_j} \quad . \tag{2.1}$$

where

$$\beta'_i = |\{S \subseteq N - \{i\} \, | v(S) = 0 \wedge v(S \cup \{i\}) = 1\}|$$

and

$\beta'_i =$ Raw Banzhaf Index
$\beta_i =$ Normalized Banzhaf Index

2.3.2 Shapley-Shubik index

Whereas in the Banzhaf index the order in which players cast their vote is irrelevant, the Shapley-Shubik index concerns itself with sequential coalitions. The Shapley-Shubik index can be computed by considering all possible winning sequential coalitions for N players and determining the pivotal player for each coalition. The pivotal player is the player which turns a losing coalition into a winning coalition. Consider p as the number of times player P is pivotal, then the Shapley-Shubik index for player P is $\frac{p}{n!}$.

2.4 Problem definition

As mentioned in Chapter 1 we are interested in the inverse or 'weighted voting game design' problem: how can we design weighted voting games so that we can find the WVG that matches the closest with a given target index?

Different heuristics have been proposed to overcome this problem, but none guarantee an exact solution. However, De Keijzer's solution enumerates all canonical weighted voting games for a given number of players, therefore guaranteeing an exact solution. Although De Keijzer designed his algorithm to be efficient, it still runs in exponential time in the input. De Keijzer's algorithm builds a tree of possible solutions in a breadth-first manner, by expanding the previous 'generation' of WVGs. Note that we can not decrease the runtime if we use De Keijzer's solution without sacrificing the guarantee of an exact solution. However, we can combine De Keijzer's exact solution with heuristics to decide on which WVG in the tree to expand first, which could possibly lead to finding the optimum in a shorter time.

We state Hypothesis 2.1, which is crucial for the design of all of the heuristics proposed in Chapter 5. We can support our hypothesis with the way De Keijzer's algorithm works. Each game (the root excepted) is constructed by expanding its parent, therefore we expect similar performance between a parent and child game. By using the heuristics in Chapter 5 we seek to find an optimal solution in the smallest amount of time. Additionally, we are interested in the optimal settings for the parameters of some of these heuristics and how these change for different number of players.

Hypothesis 2.1 *There exists a relation between the distance to target of one node and the distance to target of its children.*

Chapter 3

Related work

This chapter describes the related work in two different fields. The first section concerns itself with research in the field of weighted voting games design. The second section describes different existing techniques in the field of searching a tree structure. We are interested in these techniques, because we expand De Keijzer's existing algorithm, which enumerates WVGs by building a tree structure.

3.1 Inverse problem: WVG design

In this study we focus on the inverse problem, in which a desired target vector is given and an attempt is made to find the WVG with the minimal distance to the target vector. There are only a few concrete algorithms that attempt to solve this problem known to us. One approach by Fatima et al. [4] is concerned with finding a weighted voting game in weighted representation for the Shapley-Shubik index. This algorithm uses one of two update rules to iteratively update a vector of weights. The authors prove that by applying these rules the Shapley-Shubik index for each agent can not get worse. Therefore, it is an anytime algorithm. Aziz et al. [1] propose an algorithm for finding a WVG with the Banzhaf index. It is largely similar to [4] in that it updates a vector of weigths in order to approach the target power index. However, due to the *generating function method* [2] the authors use the weighted representation is limited to integers, in contrast to the approach in [4], in which the output may have rational weights. A different approach is used by De Keijzer et al. [3]. The authors propose an algorithm that enumerates the set of canonical weighted voting games completely. Therefore this approach is guaranteed to deliver an exact solution. We discuss De Keijzer's solution in-depth in Chapter 4.

3.2 Tree search methods and techniques

A tree is a specific type of acyclic hierarchical structure in which the specific elements are represented as 'nodes'. A node may represent any type of data, but note that in our application we consider nodes to represent WVGs. A tree has exactly one root node, which has no parent. Each non-root node has zero to many children and exactly one parent. A node which has no children is referred to as a leaf node. Nodes are interconnected using 'edges'. An edge indicates a parent-child relation, but may also hold data, such as the distance between the nodes. Note that in our application edges only represent the parent-child relation.

Many algorithms and techniques have been designed for finding a specific node of interest in the tree. We can make two clear distinctions in this field. Either the location of the node of interest is known beforehand or it is not. When the node is known beforehand we are commonly interested in the shortest path to it, where the edges contain some value on the distance between nodes. This process, often referred to as path-finding, may seem straightforward, but it is a complicated problem. When the location of the desired node is not known beforehand we are interested in finding it as fast as possible. Usually, some characteristic of the nodes is used to make a decision on which path to follow. In some of these cases it is possible to eliminate an entire branch from the search process, because of a characteristic of a single node that lies at the top of this branch. Note that in our application we do not know the location of the node of interest beforehand, moreover we do not know what the desired node is until the search process is complete. However, we can calculate the distance between the weight distribution of a WVG (node) and the desired power distribution. This provides us with a performance measure for each node, on which we can base decisions in terms of the order in which to evaluate nodes. The following subsections discuss various specific tree search techniques.

3.2.1 Depth-first and Breadth-first search

Depth-first search and breadth-first search are the two most basic forms of unordered tree searching. First we will discuss depth-first search (DFS). In DFS the first node that is selected is the root node. The first child of the root is selected, after which the first child of this node is selected, until a leaf node or the node of interest is found. When the node of interest is reached the search terminates. When a leaf node is found the technique will process the nearest ancestor with unexplored children. Basically, DFS completely searches through a single branch, until a leaf node is found. After that other branches are explored.

Breadth-first search (BFS) could be considered the opposite of DFS. First the root node (level 0) is selected, after which all children (level 1) are processed. This is followed by the selection of all their children (level 2). In this way BFS processes the nodes of a tree layer by layer. BFS is a general algorithms and the order in which the nodes in each layer are considered is not defined.

Neither of these two search methods is considered better, depending on the data one can perform better than the other. For example, data in which the node of interest is in the bottom of the tree DFS is likely to perform better. A tree in which the node of interest is in the top nodes BFS is likely to perform better. It should be noted that BFS is considered to use more memory than DFS, because BFS has to store pointers for level $l + 1$ when processing nodes at level l.

3.2.2 Backtracking and back-jumping

Backtracking is a general algorithm that can be applied to tree searching, where each branch node in the tree represents a choice. When the algorithm encounters a node that is less feasible than a different node that path is (temporarily) abandoned and the algorithm 'backtracks' to the parent of the less feasible node to evaluate a different branch. Back-jumping could be considered to be part of backtracking technology, however backtracking can only track back along one edge in the tree whereas back-jumping may jump up multiple levels.

3.2.3 Greedy algorithms

Greedy algorithms are straightforward and shortsighted algorithms usually based on a simple heuristic. At every step the algorithm selects the most promising of the feasible solutions. A common example is that of given change, in terms of money, in the smallest amount of coins. The options for algorithm are the different coins with different values. At every step the algorithm selects the coin with the largest value (most promising solution) from the set of coins that are not greater than the total amount of change that should be returned (feasible solutions). Let us consider a change of \$0.37 and we have 25¢, 10¢, 5¢ and 1¢ coins to return. A change giving algorithm would select 25¢ in the first iteration, as it the largest amount that is smaller than the required change. In the next iteration the feasible solutions are: 10¢, 5¢ and 1¢, as 25¢ is larger than \$0.12. So the algorithm selects 10¢. In the last two iterations the algorithm selects 1¢ resulting in \$0 required change.

Greedy algorithms are easy to design and implement and depending on the problem can perform very well. However, many problems can not be solved using greedy algorithms, due to their vulnerability to getting stuck in local optima.

Chapter 4

De Keijzer's WVG Designer

This chapter describes the work of De Keijzer, providing insight in his approach for the weighted voting game design problem. As mentioned in Chapter 1 De Keijzer attempts to solve the inverse problem. He considers a desired target vector and finds the game(s) with the lowest distance from the target, rather than trying to compute a priori power of the agent. Note that this is an exact approach, whereas other approaches do not guarantee an exact solution. It should be noted that De Keijzer's implementation we use for this study contains a bug. De Keijzer updated his implementation, but it was completed after we performed our experiments.

4.1 Crucial insights

De Keijzer discovers a number of crucial insights that can be used for a WVG design algorithm. The first crucial insight concerns canonical weighted voting games. De Keijzer limits his algorithm to canonical weighted voting games only, because there are infinitely many weighted representations of each WVG. Secondly, De Keijzer discovers that the MWC list at rank i can be generated fairly efficiently from the MWC list at rank $i-1$, if we only consider CWVGs. Each rank is a 'generation' of games, which means if we consider all games at rank $i-1$ to be the parent games we can generate the child games at rank i.

4.2 Process

The process of enumerating the WVGs for n agents in De Keijzer's solution is as follows. First, we generate an n-agent WVG with no MWCs, this is $rank0$. At this point generating the set of games with i MWCs works as follows: we obtain each set of maximal losing coalitions for each game with $i-1$ MWCs. For each maximal losing coalition C in each set we consider if there is a right truncation of C, which is an MWC. If so, C is added to the list of MWCs, so that list represents a new WVG. De Keijzer tests if the game is a WVG by solving a linear program. Note that this process creates a tree structure as games are created from their predecessors. We'll use an example to clarify this approach.

Consider Figure 4.1, which demonstrates the complete WVG tree for 4 players. Each node describes a game, labeled with the MWC most recently added to the list of MWCs as a binary number. For example, 1100 is the MWC with players 1 and 2, but without players 3 and 4. The process starts with the root node, which contains no MWCs. Since this is a game where it is not possible to achieve a MWC the implication is that the maximal losing coalition contains all players (1111). In this step we consider every right-truncation of MLCs and check if they are MWCs from which we can construct WVGs. The $i-^{th}$ right-truncation is achieved by removing the i highest numbered players from the coalition. Therefore all right-truncation of the MLC are: 1111, 1110, 1100, 1000 and 0000. Now we need to consider for each coalition if it is a MWC. If this is the case the coalition can be added to the list of MWCs, forming a new WVG. Note that the list of MWCs for the root is empty. This is fairly easy for these coalitions: for coalition 1110 an example weight distribution could be $[7; 3, 2, 2, 1]$, however this can become complicated, therefore De Keijzer solves a linear program to test for MWCs. Considering Figure 4.1 again we should note that all right-truncations are indeed MWCs, as they are added as children of the root. Let us consider one more example: the expansion of 1000. Since player 1 is able to achieve a MWC on its own the MLC contains all other players (0111). The possible right-truncations for this coalition are: 0111, 0110, 0100 and 0000. There is already a game with 0000 in the tree so this game is not produced as child of 1000. We will explain duplicates checking further in the next paragraph. The coalitions: 0111, 0110 and 0100 are MWCs and each one is added to a list of MWCs, therefore as a child of 1000. Obtaining the list of MWCs for a specific node from the tree can be performed by taking the shortest path from the specific node to the root, each node that is passed is a MWC in the list.

Although this approach is correct, it may generate duplicates for $n \geq 4$. Keep in mind that a duplicate in a graph will cause the duplicate node to have at least 2 incoming edges and it creates a cycle. Since a cycle is not allowed in a tree structure, and maintaining a tree structure is desirable for searching techniques, we need to check whether a game is already generated. De Keijzer uses an effective duplicates-check. Consider a canonical WVG G which we created by adding C to the list of MWCs. Now we can sort the list in ascending order and consider for each coalition C' that occurs before C in this sorted list, when we remove C' from the list of MWCs, if G is still a CVWG. In this case we do not output G, because it has already been processed before. Let us consider an example for the duplicates check: in Figure 4.1 the dotted line denotes a duplicate (at node 0110), this is game G. The list of MWCs can be obtained by traversing the tree towards the node, therefore the list of MWCs is 0110, 1010 and 1100. If we want to expand this game we need to add MWC 1001, coalition C, to the list of MWCs, therefore we obtain the following sorted list: 0110, 1001, 1010 and 1100. We can identify coalition 0110 as C', which occurs before C in the sorted list. If removing C' yields a WVG than we do not output

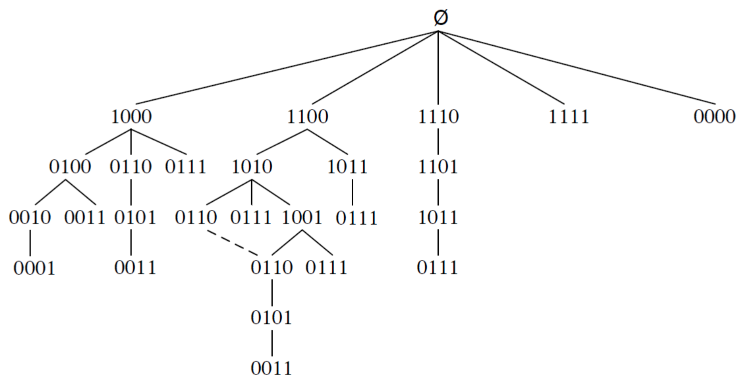


Figure 4.1: The complete tree of WVGs for 4 players

game G. De Keijzer solves a linear program to test if G is a WVG, however we can see from the tree that G is a WVG as there is a WVG with the list of MWCs after removal, namely the third game from the right in the third layer, excluding the root.

Chapter 5

Search algorithms

This chapter describes the heuristics designed to improve the error convergence of the Weighted Voting Game Design problem. Although depth-first search and breadth-first search are performed on existing trees in tree search theory, in this chapter we will define a general method that builds trees from the root node down. In addition we define some heuristics that determine the order in which the tree is constructed. Note that the presented algorithm and heuristics are based on Hypothesis 2.1.

Hybrid Directional Tree Building (HDTB) is a construct we define in this thesis, which combines depth-first search and breadth-first search, but instead of using it as a means to search an existing tree it is used to build the tree from the root node down. First we define two sets of nodes: the visible nodes V and the available nodes A. The visible nodes are all the nodes that the algorithm knows about at a given point. The available nodes are a subset of V limited by a heuristic to force a degree of local search, therefore $A \subseteq V$. The HDTB algorithm selects one to many nodes from the set of available nodes that are considered 'most interesting', based on some heuristic. We refer to this as set E, for which we can define: $E \subseteq A \subseteq V$. The children that can be derived from E are referred to as set C. Using this we can define how the visible nodes are updated: if V_{begin} contains the visible nodes at the beginning of an iteration then $V_{end} = (V_{begin} \backslash E) \cup C$. HDTB is a general algorithm, which can be combined with different heuristics to specify the selection of nodes (determine the elements in E) and limitation of available nodes (determine the elements in A). Algorithm 1 shows the pseudo code for the general HDTB algorithm i.e. it is not combined with any heuristics. Note that the methods HEURISTICS_SELECT(Q) and HEURISTICS SORT(Q, S) perform an operation that is dependent on the used heuristic. For each heuristic in the following sections we will define how each of these operations is performed.

Algorithm 1 Hybrid Directional Tree Building Algorithm (HDTB). The input is a target power index $\vec{p}$.

```
1: $Q \leftarrow r$    ▷ The search starts with the root node that corresponds to the canonical WVG with 0 MWC
2: while ($V \neq \emptyset$) do
3:     $V \leftarrow \emptyset$
4:     $N_c \leftarrow$ HEURISTICS_SELECT($Q$) ▷ Based on the heuristic, determine the number of candidate nodes to consider from $Q$
5:     for $i = 1 \ldots N_c$ do
6:         $u \leftarrow$ DEQUEUE($Q$)    ▷ Dequeue the node at the head of $Q$
7:         Output game($u$)    ▷ Use De Keijzer's algorithm to output the game at $u$ (if valid)
8:         Output $d_u$    ▷ Use De Keijzer's algorithm to compute the distance value between game($u$) and $\vec{p}$
9:         for every $u$'s child node $v$ do
10:            $V \leftarrow v$    ▷ Put $v$ in the visible nodes set $V$
11:        end for
12:    end for
13:    $Q \leftarrow$ HEURISTICS_SORT($Q, V$)    ▷ Based on the adopted heuristic, determine for nodes in $Q$ and $S$ how to sort them in the updated queue
14: end while
```

5.1 Greedy HDTB

Greedy Hybrid Directional Tree Building is the simplest variant of HDTB. This algorithm starts by selecting the root node, for which all the children are constructed. For all children the normalized Banzhaf index is computed and compared against the target index. The best performing child is used as parent from which to create the next generation, the other children are placed in a fifo structure (first in, first out), in descending order of performance. The newly considered parent node is used to generate the next generation of children. Once again the best performing child is selected, the others are placed in the fifo structure, this process continues until a leaf node is reached. When this occurs the first node on the stack is popped, considered as parent node, has its children checked for performance, other children are placed on the stack and the best performing child is used for the next generation. When a leaf node is reached and the stack is empty the stop condition is reached. Figure 5.1 shows a snapshot of the greedy HDTB algorithm in process. We can see how node 1 has been processed, which placed node 2 and 3 on the stack, one by one in descending order of performance. In the next iteration node 2 is on top and therefore processed next, placing its children on the stack. Node 5 is processed at which point we reach the state of the snapshot. Note the poor performance on this example tree, as the algorithm will completely process the subtree with node 2 as root first, while the subtree with node 3 as root contains the global optimum.

The method HEURISTICS SORT(Q, V) performs no operation on the visible nodes as it is a fifo structure, except for placing the top node on the queue. Therefore, Q contains a single node, which is selected by HEURISTICS_SELECT(Q). The available nodes A always only contains the top node of

V in this heuristic.

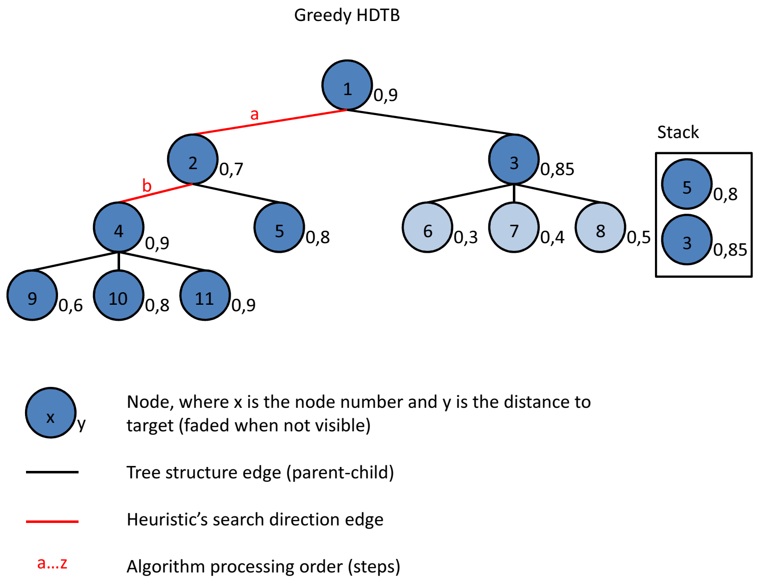
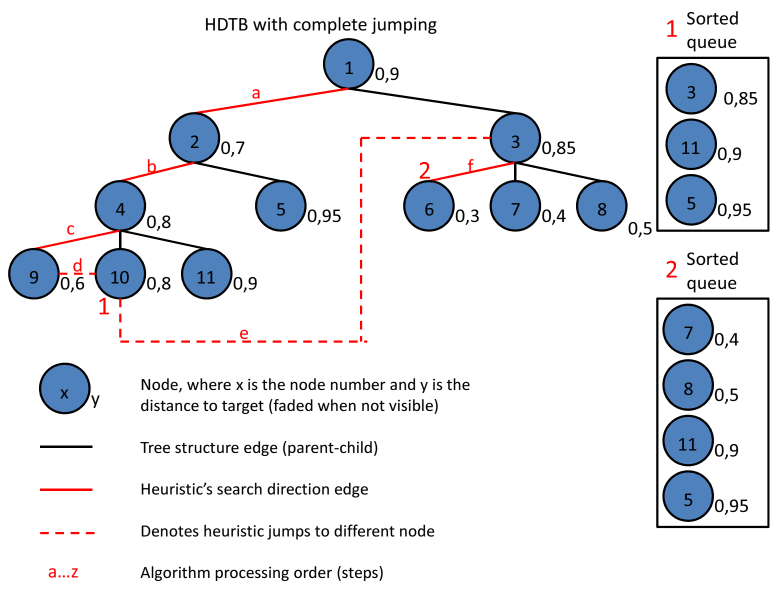

Greedy HDTB
a
b
1 0,9
2 0,7
3 0,85
4 0,9
5 0,8
6 0,3
7 0,4
8 0,5
9 0,6
10 0,8
11 0,9
Stack
5 0,8
3 0,85
x y
Node, where x is the node number and y is the distance to target (faded when not visible)
Tree structure edge (parent-child)
Heuristic's search direction edge
a…z
Algorithm processing order (steps)

Figure 5.1: A snapshot of the greedy HDTB algorithm

5.2 HDTB with complete jumping

While greedy HDTB selects the best performing node locally in a set of sibling, HDTB with complete jumping considers performance globally. The HDTB with complete jumping algorithm considers all visible nodes and selects the one with the best performance in each iteration. Note that this allows the algorithm to back-jump, as well as jump forward in the tree. Also note that although this heuristic provides global search, its search space is still limited to the visible nodes. Figure 5.2 shows two snapshots of the HDTB with complete jumping algorithm at different points of the process. Note that this HDTB variant uses a queue, which is sorted in each iteration. As mentioned we present two cases, denoted by the red numbers 1 and 2. In case 1 we see the best available (or visible) node is node 3, which requires the algorithm to jump. In case 2 the algorithm has performed two additional iterations, including the jump to node 3. We can see node 5 and node 11 are still on the bottom of the queue, due to their performance.

The method HEURISTICS SORT(Q, V) sorts the available nodes on performance, in this heuristic V and A are identical sets, and places the node with the highest performance on the queue Q, since there is only one node on the queue HEURISTICS_SELECT(Q) selects this node.



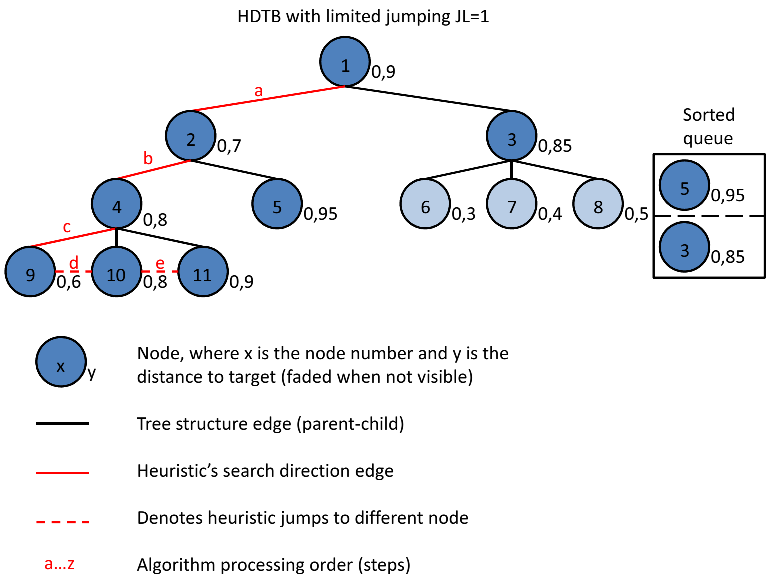
HDTB with complete jumping
1
Sorted queue
2
Sorted queue
a
b
c
d
e
f
0,9
0,7
0,85
0,8
0,95
0,3
0,4
0,5
0,6
0,9
Node, where x is the node number and y is the distance to target (faded when not visible)
Tree structure edge (parent-child)
Heuristic's search direction edge
Denotes heuristic jumps to different node
a...z
Algorithm processing order (steps)

Figure 5.2: Snapshots of the HDTB with complete jumping algorithm

5.3 HDTB with limited jumping

Where greedy HDTB and HDTB with complete jumping present two extreme cases (completely myopic or completely global search, respectively), limited jumping offers a trade-off between local and global search. The maximum jump distance is determined by a parameter, which defines the search space. This approach offers more flexibility, but the effectiveness is likely dependent on the value for the maximum jump distance. Note that we measure distance between 'layers' or 'generations' in the tree, not distance in number of edges. We believe this makes more sense in this case, since the HDTB algorithms are based on the assumption that there is some relation between the performance of the parent node and the child node. Allowing jumping per layer more easily allows jumping to a different branch if the current branch is considered less interesting. Figure 5.3 illustrates the HDTB with limited jumping algorithm. Note that this example is similar to Figure 5.2, however due to the limited jump level (JL=1), the algorithm is only able to jump one layer above or below its current node. The sorted queue is separated into two sections: the first section contains the available nodes, both sections combined contain the visible nodes. Although node 3 has a better performance, the algorithm selects node 5, due to the JL restriction.

The method HEURISTICS SORT(Q, V) first moves the nodes within the maximal JL from the visible nodes to the available nodes. The list of available nodes is sorted and the highest performing node placed on the queue. HEURISTICS_SELECT(Q) selects this node. A special case exists for HDTB with limited jumping: when $A = \emptyset$ the top node from V is placed in A, in order to overcome an infinite loop. Note that V is unsorted, therefore there is no indication of the performance of this node.



HDTB with limited jumping JL=1
1 0,9
a
2 0,7
3 0,85
b
4 0,8
5 0,95
6 0,3
7 0,4
8 0,5
c
9 0,6
d
10 0,8
e
11 0,9
Sorted queue
5 0,95
3 0,85
x y
Node, where x is the node number and y is the distance to target (faded when not visible)
Tree structure edge (parent-child)
Heuristic's search direction edge
Denotes heuristic jumps to different node
a…z
Algorithm processing order (steps)

Figure 5.3: A snapshot of the HDTB with limited jumping algorithm

5.4 BestX and WorstX

BestX and WorstX (BX and WX for short) are extra parameters we introduce for HDTB algorithms with jumping. Rather than selecting the best known solution in the current situation, the best x and worst x are processed, where x can be any integer value. Greedy HDTB is unsuitable for these parameters as the fifo structure does not allow searching for the best or worst candidate nodes. Note that all discussed HDTB algorithms have BestX=1 and WorstX=0 by default. The BX and WX parameters are introduced to increase the search space. This is due to two observations made during preliminary experimentation: the optimal solution is often preceded by one of the worst solutions and HDTB with jumping performs much like greedy HDTB in some situations, likely due to a too small search space. The first observation led to the introduction of WorstX, the second to the introduction of BestX. Figure 5.4 demonstrates the HDTB with limited jumping algorithm with a BX parameter of two. Note how the algorithm extends node 2 and node 3 and places all children on the queue. After sorting node 6 and node 7 are considered the best options. At this point the algorithm has found the global optimum, however there is no way of knowing this beforehand, therefore the algorithm continues processing nodes until the queue is empty, i.e. all nodes have been processed. These parameters can change the operation of the methods HEURISTICS SORT(Q, V) and HEURISTICS_SELECT(Q). HEURISTICS SORT(Q, V) separates and sorts the available nodes, once again: for HDTB with complete jumping V and A are identical. The BestX and WorstX candidate nodes are selected from the available nodes and placed on the queue. Note that when $BX + WX \geq |A|$, Q is identical to A i.e. all available nodes are placed on the queue. HEURISTICS_SELECT(Q) returns $|Q|$, where $1 \leq |Q| \leq BX + WX$, depending on the number of available nodes. The BX and WX parameters do not influence the special case for HDTB with limited jumping. When $A = \emptyset$ the top node from V is placed in A, not the BX and WX nodes.

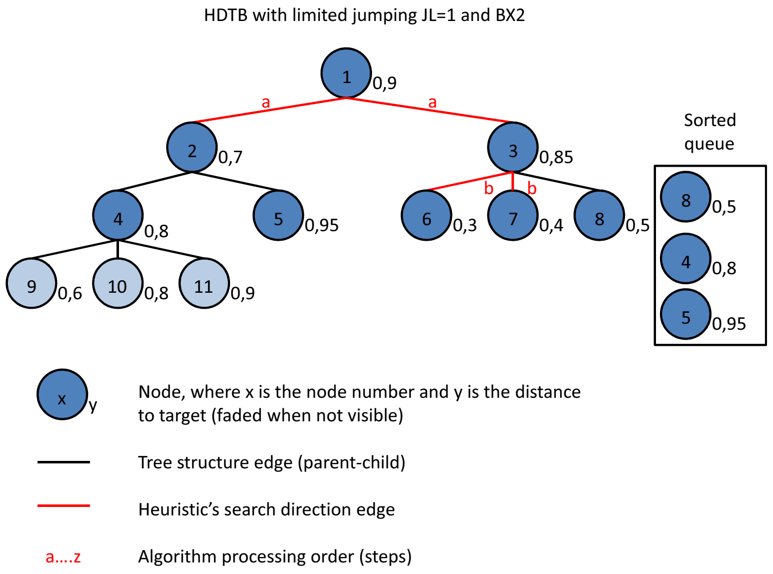

HDTB with limited jumping JL=1 and BX2
1 0,9
a
a
2 0,7
3 0,85
b
b
4 0,8
5 0,95
6 0,3
7 0,4
8 0,5
9 0,6
10 0,8
11 0,9
Sorted queue
8 0,5
4 0,8
5 0,95
x y
Node, where x is the node number and y is the distance to target (faded when not visible)
Tree structure edge (parent-child)
Heuristic's search direction edge
a….z
Algorithm processing order (steps)

Figure 5.4: A snapshot of the HDTB with limited jumping with BX parameter algorithm

Chapter 6

Experimental Evaluation

In order to determine the performance and characteristics of the designed algorithms we design a number of experiments. A distinction can be made between two sets of experiments. The first set is oriented towards an explorative overview of the performance of the algorithms and is used for gaining insight in the mechanics and determining the setup for the second set of experiments. The second set provides a much more in-depth view of the performance of the algorithms with a wide range of parameters. We will first discuss the experimental setup for both sets of experiments, followed by their results. In order to properly interpret the experimental results we need some details and definitions, which apply to all experiment sections. It should be noted that every CVWG tree contains at least one optimum and the theoretical limit is the number of games, i.e. all games are optima. Additionally, the location of the optimum (or optima) can not be guaranteed during runtime. Furthermore, we measure the distance to the target vector of each node in the tree (each WVG) using the euclidean distance. We define the time to optimum of a heuristic to mean the number of games that need to be processed to reach the optimum, not the time it takes. The terms performance and time to optimum will both be used and have the same meaning. The presented experiments are largely performed on random target vectors. This target vector is created by taking n, where n is the number of players, random values between 0 and 1, which we refer to as R_n. For every R_n we do $\log(R_n) * -1$ after which all values are normalized to sum up to exactly 1. The values are sorted in descending order to form a target vector with non-increasing weights. Finally, we provide Table 6.1, which provides an overview of the number of games for 1 to 8 players.

6.1 Explorative experiments

Since it is difficult to make proper assumptions regarding the performance of different algorithms with different parameter settings we devise a set of experiments that allows us more insight. This set of experiments' main focus is on

Players	Number of games
1	3
2	5
3	10
4	27
5	119
6	1083
7	24162
8	1353220

Table 6.1: Number of games per number of players

comparing the performance of different algorithms and parameters. Note that we perform a limited number of experiments in this section, therefore we do not draw conclusions, but rather compare the results to our intuition to select a candidate heuristic for extensive testing. The most interesting cases for us are where one heuristic has a shortcoming (poor performance) where another heuristic performs better. Although this does not provide us with statistical data we are able to identify if cases exist for which some heuristics perform better.

6.1.1 Experimental setup

For the first set of experiments a number of target vectors are designed, as well as three randomly generated vectors, for every number of players. The designed target vectors are aimed at specific extreme cases to evaluate the ability of the algorithms to handle these. The first specifically designed target vector has one half as the first element and each successive element is divided by two. An exception exist for the last element, which is equal to the previous element. For 5 players the target vector would look as follows: (1/2, 1/4, 1/8, 1/16, 1/16). The second specifically designed target vector contains one value for the first half of players and another value for the second half. For an odd number of players the first group has $\frac{N-1}{2}$ highest numbered players, the second group has the other players. A target vector for 5 players would look as follows: (3/10, 3/10, 2/15, 2/15, 2/15).

During preliminary experiments it is noted that cases with 4 players offer little interesting data. Due to their relatively small trees the algorithms reach an optimal solution early in their process, therefore there's is little difference in performance. For this reason, all cases with 4 players are left out of the explorative experiments. In addition, cases with 8 players or more are also left out, due to runtime. This limits the cases for the explorative experiments to 5, 6, and 7 players.

Greedy HDTB is not included in this set of experiments due to its poor performance compared to De Keijzer's algorithm, which is consistent with intuition on this short-sighted heuristic. HDTB with limited jumping is the most promising and versatile heuristic, especially in combination with BestX and WorstX. Therefore the selected algorithms for the explorative experiments are:

ID	Target vector
tv1	(0.5, 0.25, 0.125, 0.0625, 0.0625)
tv2	(0.3, 0.3, 0.1333, 0.1333, 0.1333)
tv3	(0.4224, 0.2392, 0.1374, 0.1097, 0.0912)
tv4	(0.5471, 0.2838, 0.0919, 0.0432, 0.0340)
tv5	(0.3634, 0.2550, 0.1872, 0.1024, 0.0921)

Table 6.2: Target vectors for experiments with 5 players

De Keijzer's, HDTB with jump-level=maximum depth of the tree, HDTB with jump-level=8 and BX=3, HDTB with jump-level=8, BX=3 and WX=3. These heuristics are abbreviated to JL=max, JL=8, BX3 JL=8 and BX3WX3 JL=8, occasionally prefixed with HDTB.

6.1.2 Experiment results

Using the setup described in the previous section we can generate experiment results. Each paragraph discusses the experiment results for each number of players, starting with 5 players. For each number of players the performance of the four algorithms for five target vectors is reviewed. A table displays the time to first optimum, which defines the performance, in values relative to the total number of games. Additionally, we show a graph to illustrate the error convergence and some of the algorithm's characteristics. Again we should stress that we are interested in identifying single cases for which one heuristic performs better than others, not in proving significant differences in performance.

5 players First, we will discuss the results for experiments with 5 players. These experiments use the target vectors from Table 6.2. The individual elements of the target vectors are rounded at 4 decimals in the table for display purposes, the actual values offer greater precision. It should be noted that target vectors "tv1" and "tv2" are the designed target vectors, whereas the rest are randomly generated. The results are displayed in Table 6.3 and we can make a few observations. First, for each target vector all other algorithms outperform De Keijzer's. Secondly, there is quite a difference between the performance of BX1 JL=max compared to the two with BX3 for tv1. This likely indicates that a lack of search space caused BX1 JL=max to follow a less promising search path. Third, we can see similar performance for BX3 JL=8 and BX3WX3 JL=8 for target vectors: tv1, tv4 and tv5. For tv2 BX3 JL=8 is more than twice as fast as BX3WX3 JL=8. For tv3 this is almost the other way around. Finally, we can observe that BX1 JL=max, BX3 JL=8 and BX3WX3 JL=8 offer similar performance on average.

6 players This paragraph discusses the results for 6 players. Table 6.4 shows the target vectors used for these experiments. Using these target vectors we generated the results shown in Table 6.5. Considering this table we can make a few observations. First, in all cases there is an heuristic that outperforms De

Target vectors	Methods			
	De Keijzer's	BX1 JL=max	BX3 JL=8	BX3WX3 JL=8
tv1	27,97	31,36	6,78	11,02
tv2	65,25	14,41	16,95	34,75
tv3	53,39	32,20	34,75	21,19
tv4	27,97	3,39	6,78	11,02
tv5	77,97	40	39,83	36,44
Average	50,51	23,05	21,02	22,88

Table 6.3: Results for experiments with 5 players (5 players have 119 WVGs)

Keijzer's algorithm. Additionally, in 3 out of 5 cases all algorithms outperform De Keijzer's algorithm. This could be random chance, however it is possible that the HDTB algorithms have trouble coping with the much larger search space (note that the tree is more than 9 times larger for 6 players, compared to 5 players). This could lead to the algorithms following an incorrect branch. Additionally, it should be noted that BX3WX3 JL=8 is the only algorithm that is never outperformed by De Keijzer's algorithm in these cases. Secondly, it seems BX1 JL=max is pretty much hit or miss in these cases. This could, again, indicate that its limited search space can cause the algorithm to follow an incorrect path down for some time before jumping back. This is reflected in the average performance. Thirdly, we can observe small difference in performance between BX3 JL=8 and BX3WX3 JL=8, with BX3 JL=8 usually having a slightly better performance. This could indicate that the optimum is not found as a child of one of the worst games. Since BX3WX3 JL=8 also needs to extend the worst 3 games we can understand how it has a slightly higher time to optimum in most cases. Finally, tv4 is an interesting case as all algorithms seem to have difficulty finding the optimum. However, BX3WX3 JL=8 clearly outperforms the other HDTB algorithms. This would indicate that the optimum is a child of a poorly performing game, which the introduction of the WX parameter allows the algorithm to overcome. Close inspection of the data reveals that BX3 JL=8 makes the 'mistake' in the first generation of games, the root excepted. It considers two children interesting and expands these branches far down, while both optima lie in a different branch.

Figure 6.1 shows an example of the error convergence graph for tv1 of 6 players for all algorithms. We can see some characteristics for each algorithm. De Keijzer's algorithm clearly has a rather random search direction, there is no clear pattern in the graph. BX1 JL=max and BX3 JL=8 demonstrate a pattern quite typical to them. The algorithm quickly selects good games, as it reaches optimum or a dead end it selects relatively bad games until it can expand a more promising path. At this point we can see a drop in distance to target. BX3WX3 JL=8 also demonstrates a typical pattern, starting with large oscillations, which get smaller over time. This makes sense as the best and worst games are selected, where early on more extreme cases are available. Note how around 970 games BX3 JL=8 needs to expand the worst game in order to reach an optima, while BX3WX3 JL=8 does not suffer from this problem.

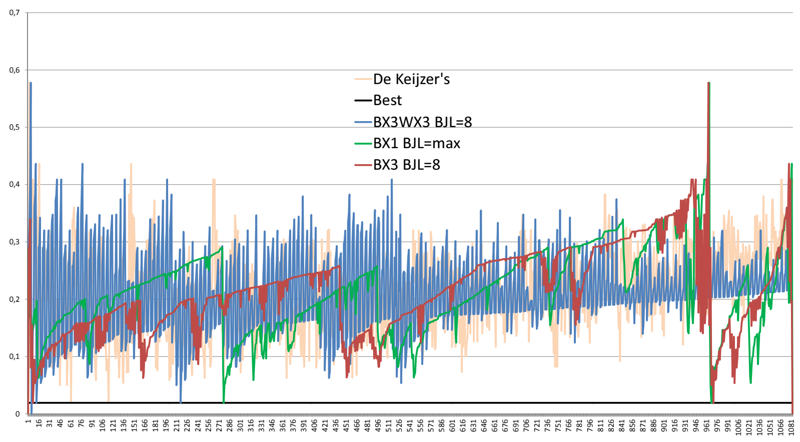


Figure 6.1: Example error convergence for 6 players

ID	Target vector
tv1	(0.5, 0.25, 0.125, 0.0625, 0.0313, 0.0313)
tv2	(0.25, 0.25, 0.25, 0.0833, 0.0833, 0.0833)
tv3	(0.3827, 0.2389, 0.1667, 0.1307, 0.0468, 0.0342)
tv4	(0.4425, 0.2122, 0.1997, 0.0869, 0.0451, 0.0136)
tv5	(0.4359, 0.3572, 0.1206, 0.0714, 0.0089, 0.0059)

Table 6.4: Target vectors for experiments with 6 players

7 players This paragraph discusses the results for experiments with 7 players. The target vectors used for these experiments are shown in Table 6.6. The results for experiments with 7 players are shown in Table 6.7. We can observe a number of things. First, De Keijzer's algorithm is outperformed by all other algorithms in 3 out of 5 cases. Secondly, we see that the performance of BX1 JL=max is becomes quite poor for 7 players. Thirdly, we see a similar performance between BX3 JL=8 and BX3WX3 JL=8 for 3 out of 5 cases, with BX3 JL=8 performing better. Although for one case (tv2) BX3 JL=8 has almost half the time to optimum, the absolute difference in number of games is relatively low (less than 300 out of 24162 possible games). The cases for which BX3WX3 JL=8 outperforms BX3 JL=8 are much more interesting. For tv4 it performs more than 11 times as fast with an absolute difference of more than 11000 games. For tv5 the difference is smaller, but still well over 4000 games.

Target vectors	Methods			
	De Keijzer's	BX1 JL=max	BX3 JL=8	BX3WX3 JL=8
tv1	5,72	25,67	0,74	1,20
tv2	8,49	0,46	1,02	1,85
tv3	22,62	8,03	1,75	2,86
tv4	20,13	46,54	38,69	17,82
tv5	4,34	0,37	1,11	1,29
Average	12,26	16,21	8,66	5,00

Table 6.5: Results for experiments with 6 players (6 players have 1083 WVGs)

ID	Target vector
tv1	(0.5, 0.25, 0.125, 0.0625, 0.0313, 0.0156, 0.0156)
tv2	(0.2143, 0.2143, 0.2143, 0.0893, 0.0893, 0.0893, 0.0893)
tv3	(0.2691, 0.2399, 0.1606, 0.1117, 0.0854, 0.0732, 0.0606)
tv4	(0.5388, 0.1366, 0.1353, 0.1070, 0.0485, 0.0263, 0.0075)
tv5	(0.3096, 0.2519, 0.1327, 0.0978, 0.0900, 0.0758, 0.0422)

Table 6.6: Target vectors for experiments with 7 players

Finally, we can observe the average performances, for which BX3WX3 JL=8 is clearly the best performing algorithm for 7 players, all round.

6.1.3 Experiments conclusion

Finally, we can draw some conclusions based on the explorative experiments. First, we can conclude that BX1 JL=max can be hit or miss in terms of performance, especially with a larger number of players. Secondly, the Keijzer's algorithm performs the worst on average. This is obvious considering its brute force approach. It should be noted that this algorithm was never designed with performance in mind and proving this is not the goal of these experiments, rather the algorithm is included as a benchmark. Thirdly, we can conclude that BX3 JL=8 performs well in general, but suffers a dramatic increase in time to optimum when the optima are children of bad performing games. Finally, we can conclude that, considering these cases, BX3WX3 JL=8 is the most interesting candidate for further experimentation. Although outperformed at some occasions its performance on average and ability to overcome the problems of using the BX parameter only, make it a more interesting candidate for the in-depth

Target vectors	Methods			
	De Keijzer's	BX1 JL=max	BX3 JL=8	BX3WX3 JL=8
tv1	1,02	30,74	0,05	0,08
tv2	57,55	8,10	1,39	2,69
tv3	80,91	19,94	20,98	21,98
tv4	4,04	49,54	50,24	4,43
tv5	55,24	39,72	43,36	25,19
Average	39,75	29,61	23,20	10,87

Table 6.7: Results for experiments with 7 players (7 players have 24162 WVGs)

experiments than any other tested algorithm.

6.2 In-depth experiments

By considering the results of the explorative experiments we can determine on which elements we would like to focus. Since the runtime of some experiments is long we need to limit some variables. In the following sections we provide an overview of the experimental setup and the results. The main focus of this set of experiments is on determining the influence of BX and JL on performance and determining the ideal range of BX and JL for a given number of players.

6.2.1 Experimental setup

The number of target vectors for these experiments is determined at 100 for each number of players. This offers a reasonably sized sample with a matching runtime.

The range of players for this set of experiments is set to 4-7 players. We do not exclude cases with 4 players as we did with the explorative experiments, due to the shift in focus. The in-depth experiments are more focused on the relation between the performance and the combination of BX and JL. As we have no insight in this matter at this point there is no reason to exclude cases with 4 players. We do exclude 8+ players due to runtime limitations.

In order to limit runtime we consider the most interesting candidate, HDTB with limited jumping, BX and WX. By design this should be the most reliable candidate heuristic, the addition of the WX parameter should offer better reliability in general, although sometimes at a cost of performance. Our explorative experiments appear to support this intuition.

Although the addition of WX seems to provide a greater reliability when finding an optimum, we do fix it for this set of experiments to limit the number of combinations and therefore runtime. However, setting WX to a single value for all number of players seems illogical. Therefore, we set the value for WX to $WX = 2^{N-4}$, where N is the number of players. Note that this equation is only used for $4 \leq N \leq 8$. The fixing of the WX implies that the remaining parameters for testing are BX and jump-level (JL). The range for JL is fairly intuitive, as the minimum for the algorithm is 1 and the maximum is the maximum depth of the tree. For cases with 7 players we limited the maximum JL to 17, in order to decrease runtime. The value range for BX is less intuitive than JL. By looking at the tree we determined the range to be the same as the JL, with an exception for 4 players, which has maximum BX=10 instead of BX=6.

6.2.2 Experiment results

This section describes the results for the in-depth experiments and is set up as follows: each paragraph describes the results for a fixed number of players with varying values for BX and JL on 100 target vectors. The results for each number of players are displayed by at least a 3-dimensional graph. This graph may be accompanied by 2-dimensional representation when required. Note that both graphs provide absolute values for the time to optimum.

4 players This paragraph describes the results for 4 players. The range of values for BX and JL are both set to 1-10 and all combinations are run on 100 target vectors. Figure 6.2 shows the average time to optimum for all cases. Immediately, it is clear that the value for JL has little or no influence on the performance of the algorithm, with a small exception up to BX4 and up to JL=3. We can see there is no influence from JL, apart from the aforementioned exception, because the performance is not affected on the y-axis. We notice the lowest time to optimum (highest performance) is at BX2 $JL \geq 2$ and the worst performance is at BX1 JL=2.

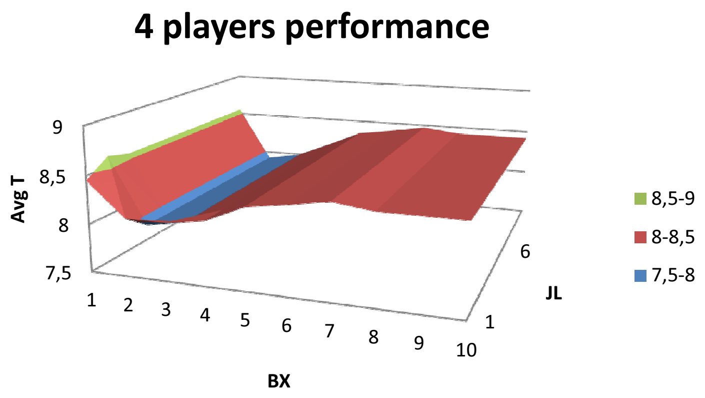


Figure 6.2: 4 players performance overview (4 players have 27 WVGs)

5 players This paragraph describes the results for 5 players. Both the value ranges for BX and JL are set to 1-15 and again all combinations are tested on 100 target vectors. In Figure 6.3 we can see the results for these experiments. We notice a similar pattern to that in the results for 4 players, JL seems to offer little to none influence on the performance. We can observe the optimal settings at BX4 $JL \geq 4$, however at BX5 the performance is similar. The worst results are at BX1 JL=5, with BX1 $JL \geq 4$ and BX15 $JL \geq 1$ providing similar results.

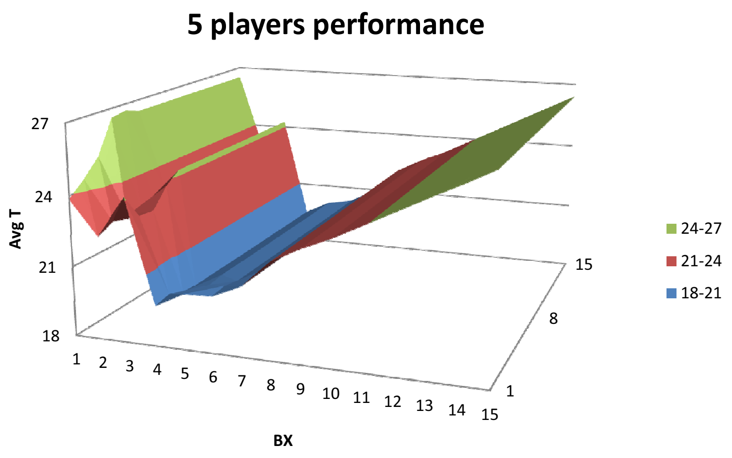


Figure 6.3: 5 players performance overview (5 players have 119 WVGs)

6 players This paragraph describes the results for 6 players. The ranges for BX and JL are set to 1-20 and all combinations are tested on 100 target vectors. Figure 6.4 provides some interesting results. Immediately, we can observe the influence of JL is greater, around $1 \leq JL \leq 10$. However, at higher JL settings the influence is reduced to near zero. Interestingly, the optimal settings are at BX5 JL=3, however similar results can be found at BX10 $JL \geq 10$, which might be more reliable, due to the fact that surrounding parameter settings provide similar performance, which does not hold for BX5 JL=3.

7 players This paragraph describes the results for 7 players. In order to limit runtime the maximum JL is set at 17, while the maximum BX is set at 35 (equal to maximum possible JL on 7 players). All combinations are run on 100 target vectors to maintain a decent sample size. Figure 6.5 displays the results using these settings, note that this graph is rotated compared to similar ones in order to provide a better overview. Note the influence of JL, which was near absent for cases with less than 7 players. Also note how there seems to be an area with lower time to optimum (higher performance) following a curve from BX2 JL=2

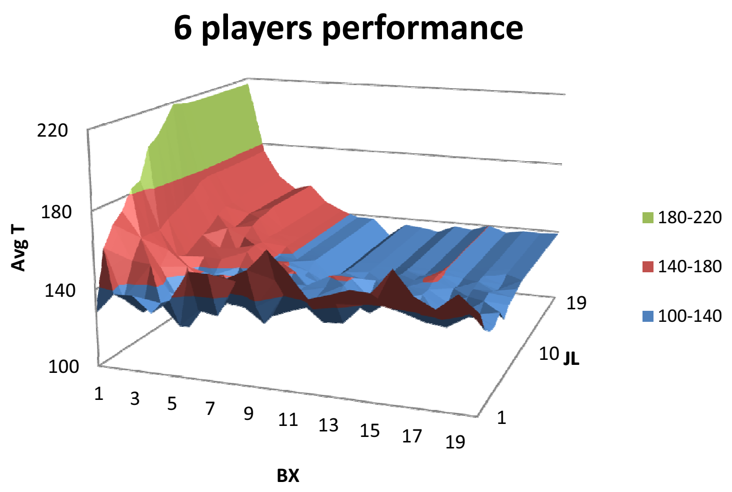


Figure 6.4: 6 players performance overview (6 players have 1083 WVGs)

to BX35 JL=7, where performance is higher near the beginning. The optimal average performance is found at BX5 JL=3, closely followed by BX27 JL=8, while the worst average performance is found at BX1 JL=16.

6.2.3 Heuristic relative effectiveness

The experiment results seem to indicate that the heuristic has a relatively higher performance for a larger number of players. Considering the relative time to optimum, expressed in relation to the total number of games, we can compare the performance of the heuristic on different number of players. For each number of players we select the case with the fastest time to optimum, the slowest, the average over all cases and the Median. Note that each of these performance measures is averaged over 100 target vectors. Also note that our data does not contain the complete range of possible BX and JL settings for each number of players, therefore our results and conclusions only apply to this dataset. Table 6.8 describes what we refer to as the heuristic's relative effectiveness. We can clearly see a decrease in time to optimum (increase in performance) for the optima cases for a larger number of players. A similar decrease can be seen for the worst cases, however this does not hold for 7 players. This could be due to the range of BX and JL settings for 7 players. However, it is also possible that the time to optimum varies more for 7 players. Considering the Median case and

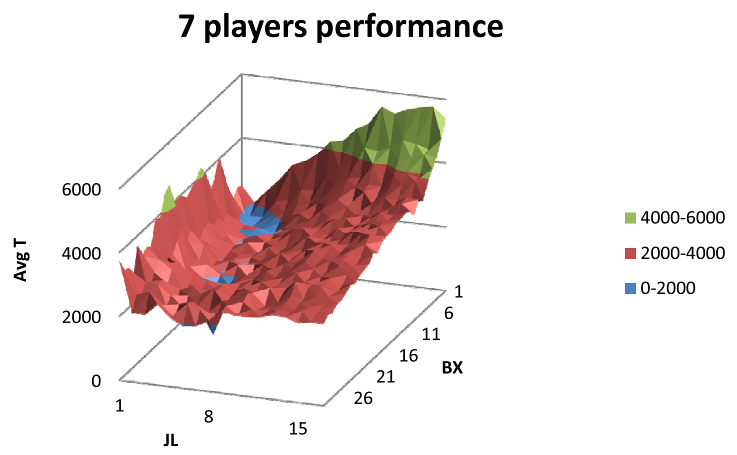


Figure 6.5: 7 players performance overview (7 players have 24162 WVGs)

Players	Optimum	Worst	Median	Average all cases
4	32,44%	37%	31,15%	34,54%
5	17,78%	24,08%	24,08%	20,87%
6	8,96%	16,55%	12,79%	10,44%
7	6,80%	23,70%	10,59%	11,25%

Table 6.8: Heuristic relative effectiveness

the average over all cases we don't see a similar decrease in performance as from 6 to 7 players in the worst case. Based on the Median case and the average over all cases we can only conclude similar performance. We can conclude that the relative effectiveness of the algorithm increases with a larger number of players, provided we use settings that provide average or better performance.

6.3 Comparative experiments

Chapter 6.2 provides us with the influence of the BX and JL parameters on performance of HDTB with limited jumping. However, all heuristics are designed to provide an increase in performance on De Keijzer's original algorithm. As we only have the details for HDTB with limited jumping we focus on this heuristic for comparative experiments against De Keijzer's algorithm.

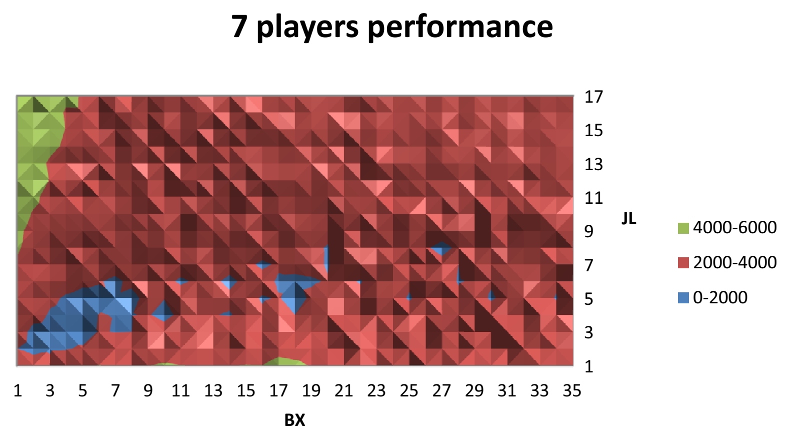


Figure 6.6: 7 players performance overview (7 players have 24162 WVGs)

Settings	Performance	Corresponding BX	Corresponding JL
Optimal	7,92	2	5
Worst	8,61	1	2
Median	8,31	9	5
Average	8,42	5	5

Table 6.9: 4 players comparative settings

6.3.1 Experimental setup

As the performance of the heuristic varies greatly with different BX and JL parameters we need to select settings that offer different levels of performance. To this end we select the optimal settings, the worst settings, the median, and a case that is closest to the average performance over all cases. Tables 6.9, 6.10, 6.11 and 6.12 contain the settings, their performance (rounded at two decimals) and the corresponding BX and JL settings. Note that the WX parameter is fixed at the same level as the in-depth experiments. For each setting we compare the performance against De Keijzer's algorithm on 100 random target vectors. By means of a paired one-tailed T-test we check the results for statistical significance.

Settings	Performance	Corresponding BX	Corresponding JL
Optimal	21,16	5	9
Worst	28,66	15	7
Median	28,66	15	7
Average	24,83	3	1

Table 6.10: 5 players comparative settings

Settings	Performance	Corresponding BX	Corresponding JL
Optimal	97	7	8
Worst	179,2	1	9
Median	101,56	9	16
Average	113,10	17	5

Table 6.11: 6 players comparative settings

6.3.2 Experiment results

Using the previously described settings we can compare the performance for the heuristic with the performance of De Keijzer's algorithm. The worst case for 7 players is the only case for which the P-value is larger than 0,05, namely 0,4534. The average P-value for all other cases is $5,10 * 10^{-5}$. From this we can conclude that for all cases, except the worst case for 7 players, HDTB with limited jumping, BX and WX parameters performs significantly better than De Keijzer's original algorithm. It should be noted that we draw no conclusions for WX values, BX and JL settings outside the ranges defined in the in-depth experiments.

Settings	Performance	Corresponding BX	Corresponding JL
Optimal	1643,48	27	8
Worst	5725,63	1	16
Median	2386,55	2	5
Average	2718,19	33	3

Table 6.12: 7 players comparative settings

Chapter 7

Conclusion and future work

This chapter presents our conclusion, as well as our ideas for future work.

7.1 Conclusion

In this section we draw five main conclusions on our research, which are accompanied by theories, which we do not substantiate, that can be used in our future work. Firstly, we have determined the ideal ranges for the parameters BX and JL for 4 to 7 players. However, these values are only determined in combination with a fixed value for WX, we can not draw any conclusions for different values of WX. An interesting observation here is the relative similar ideal ranges for BX and JL between experiments with 4, 5, 6 or 7 players. By intuition it would make sense that progressively larger values for BX and JL are required for a larger tree, which is not the case. In the case of BX it seems the heuristic is able to make a fairly 'good' decision on which path to take high in the tree and therefore has no need for a larger search space. In the case of the JL setting, keeping the algorithm relatively short-sighted appears to be an advantage. This could indicate that the algorithm selects a 'good' path relatively early and only needs to jump a couple of levels when a branch is found less promising. Both of these theories could support our hypothesis, however they do not provide any proof. For JL the ideal settings range from 3 to 5, while the BX settings range from 2 to 5 for 4-7 players.

Secondly, we provide statistical support that one of our heuristics, HDTB with limited jumping, BX and WX, provides an improvement on the error convergence compared to De Keijzer's original algorithm for 4 to 7 players, given we do not use the worst settings for 7 players. Once more it should be noted that WX is fixed in our experiments and different values for WX could greatly change the outcome, both positively and negatively.

Thirdly, we conclude that the outcome of our experiments provides support for our hypothesis. However, we can not accept our hypothesis, neither can we falsify it.

Fourthly, we conclude that, given the range of 4 to 7 players, the relative effectiveness of HDTB with limited jumping improves for a larger number of players. This means that the heuristic is able to find a optimum faster in relation to the total number of games for n players, than for $n-1$ players.

Finally, we conclude that there is statistical support that HDTB with limited jumping provides a better performance than De Keijzer's original algorithm, except for worst settings on 7 players, given the range of BX and JL settings we use.

7.2 Future work

In our research we have been forced to make decisions due to time restrictions. However, we are interested in the further investigation of a number of options. First, we are interested in a more in-depth investigation of the relation between the performance of one node and its children. For example, by calculating the distance between the performance and its children for every node. Secondly, we are interested in a performance evaluation in which we vary the value for the WX parameter. Finally, we are interested in designing and testing other heuristics, which can be tested against the current heuristics.

Bibliography

[1] H. Aziz, M. Paterson, and D. Leech. Efficient algorithm for designing weighted voting games. In *Proceedings of the IEEE Computer Society, 11th IEEE International Multitopic Conference*, 2007.

[2] J. Bilbao, J. Fernández, A. Losada, and J. López. Generating functions for computing power indices efficiently. *TOP: An Official Journal of the Spanish Society of Statistics and Operations Research*, 8(2):191–213, December 2000.

[3] B. de Keijzer. On the design and synthesis of voting games : exact solutions for the inverse problem. Master's thesis, Delft University of Technology, 2009.

[4] S. S. Fatima, M. Wooldridge, and N. R. Jennings. An anytime approximation method for the inverse Shapley value problem. In *Proceedings of the Seventh International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008)*, pages 935–942, Estoril, Portugal, May 2008.