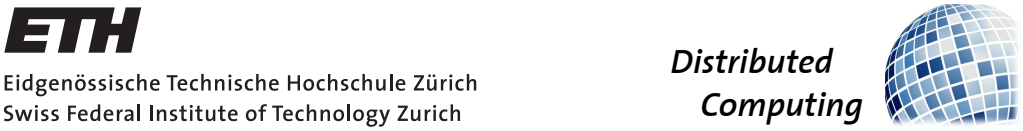

Ranking Alternatives Offline

Bachelor's thesis

Raphael Anderegg

araphael@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zrich

Supervisors:
Georg Bachmeier
Prof. Dr. Roger Wattenhofer

September 15, 2016

Acknowledgements

I want to thank my supervisor Georg Bachmeier for his help with this thesis. He provided me with all the references needed for my work.
I would also like to thank the team of the GraphStream[1] project for providing the graphstream library, which proved invaluable for debugging and implementing the logic of graphs.
Furthermore I want to thank the Gurobi Optimization[2] team for providing the linear programming solver I used.

[1] graphstream-project.org
[2] www.gurobi.com

Abstract

Rank aggregation describes how one can derive a consensus from multiple preferences over a given set of *alternatives*. Alternatives could be candidates in an election, search results ranked by search engines or businesses which are ranked by users with services like Yelp[3].
There are different *paradigms* for how the consensus should be constructed. In this thesis we will concern ourselves with the well known Kemeny and Slater rules and the FLAP[4] rule. As it is NP-hard to compute the consensus with these paradigms there has been much research on producing good heuristics to decrease computation time. We will introduce and evaluate different heuristics for the different paradigms.

[3] www.yelp.com
[4] Feedback Linear Arrangement Problem

Contents

Chapter 1

Introduction

Rank aggregation is the method of aggregating preferences over different *alternatives* of multiple parties. Alternatives could be a multitude of things, from candidates in a political race to products in an online shop. Rank aggregation is an important concept since there are many different possibilities how to form a consensus from (often times) conflicting preferences. One could start with the simple *plurality rule*. It simply ranks alternatives by the number of times they are ranked highest. In an election for example we only cast a vote for the candidate we favour the most and do not give any information about the other candidates. Unfortunately we discard most information, so although *plurality rule* is easy and fast to implement, sometimes it is not enough. Rank aggregation has gained attention because of its possible uses in online shopping systems, review and rating websites and so on. There are different paradigms of scoring rankings. The most researched are the Slater rule and the Kemeny rule. Another paradigm devised by the Distributed Computing Group at ETH is the FLAP paradigm. It is also known that it is NP-hard to compute an optimal ranking for the Slater and Kemeny paradigms. So most of the algorithms known compromise optimality to lower the execution time. In this thesis we will investigate the properties of heuristic methods developed by the writer and by other works. Aside from designing and implementing algorithms that solve the rank aggregation problem, we first have to find or produce data we can use to give our algorithms as input. Therefore this thesis presents a method to produce artificial votes.

1.1 Related work

There is a large zoo of heuristics in the literature for the Kemeny and Slater rule. A compilation of the more well-known heuristics for the Kemeny rule can be found in[1]. They compare the heuristics for different input data. They conclude that it is best to use an optimal method when possible to prevent suspicion or disagreement since the non-optimal methods could produce different results. They come to the same conclusion as this thesis that *Local search* is the most

favourable method to use in this case.
There have also been advances in finding methods for the Slater rule. One of them is described in[2]. But as we will see their approach has strong drawbacks. Most methods for the Kemeny rule can be adapted to fit with the Slater rule which we demonstrate in this thesis.
For the FLAP rule there is no literature yet because the Distributed Computing Group at ETH devised it.
There are many different ways of generating synthetic test data described in the literature. We decided to use the method described in [3].

Chapter 2

Background

2.1 Vote

A *vote* is defined by a sequence $(a_0, ..., a_{n-1})$ that is some permutation on a given set A of alternatives. The alternatives are ranked from highest to lowest, so in this case a_0 would be ranked the highest. A vote v can also be represented as a function where $v(a_i)$ is the rank of alternative a_i. A *rank aggregation function* maps a give set of votes V to a single *total ranking* r_{tot} which has the same form as a single vote but should satisfy certain properties based on the votes given to the rank aggregation function.

2.2 Majority Graph

For every two alternatives s and t, we define v_{st} to be the number of votes in V which rank s higher than t and v_{ts} the number of votes which rank t higher than s. If $v_{st} > v_{ts}$ we define $s > t$ as the *majority vote* and in turn $t < s$ as the *minority vote*.
From these building blocks we can construct a weighted graph $G = (A, A \times A, w)$ called the *preference graph* with edges e_{st} pointing from alternative s to alternative t and weight function $w(e_{st}) = v_{st}$.
We convert this preference graph to the *majority graph* by combining the respective two edges between two nodes. If there are the edges in the preference graph e_{st} and e_{ts} with $w(e_{st}) > w(e_{ts})$, then more votes prefer s to t. We denote this in the majority graph by drawing one edge e'_{st} from n_s to n_t with weight $w(e_{st}) - w(e_{ts})$ i.e. the margin by which s wins the *pairwise ranking* against t. Therefore all edges in the majority graph will have a non-negative weight. In the special case that $w(e_{st}) = w(e_{ts})$ the direction of the edge in the majority graph does not matter and the weight is 0. After an example we will show how the majority graph can be used to express different rank aggregation paradigms.

Example 2.1. Consider a case with 3 alternatives A,B,C and 4 votes (B,C,A),

(B,C,A), (C,A,B) and (C,B,A). From this we can construct the edges of the preference graph. For example $e_{AB} = 1$ because there is 1 vote (C,A,B) which prefers A to B. We can further compute $e_{BA} = 3$ because there are 3 votes (B,C,A), (B,C,A) and (C,B,A) which prefer B to A.
From this we compute the edges of the majority graph. For example we draw the edge e'_{BA} because $w(e_{BA}) > w(e_{AB})$. We then can calculate the weight $w(e'_{BA}) = w(e_{BA}) - w(e_{AB})$. Because $w(e_{BC}) = w(e_{CB})$ we can choose to draw either e'_{BC} or e'_{CB} because they have weight 0.

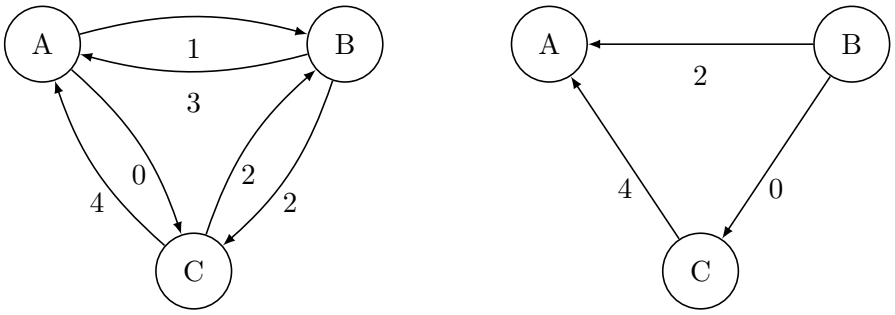

Figure 2.1: The preference graph of Example 2.1 on the left and the corresponding majority graph on the right

2.3 Kemeny rule

The first paradigm we will define is the Kemeny rule. It tries to minimize the *Kemeny score* of the total ranking, which is defined as follows. We take as input the majority graph G and a total ranking r_{tot} resulting from the rank aggregation function. For each edge e_{st} in the majority graph we add $w(e_{st})$ to the score if the total ranking ranks t higher than s. The intuition is that we pay the weight of every edge we have to invert in the majority graph to satisfy the total ranking. This means that an optimal solution regarding the Kemeny score will try to minimize the sum of all weights of inverted edges resulting from the total ranking. It is not always possible to find a total ranking with Kemeny score 0 because there may be a cycle in the majority graph and we have to break that cycle by inverting at least one edge in that cycle.

$$KemenyScore(G, r_{tot}) := \sum_{\substack{e_{st} \in E \\ r_{tot}(t) < r_{tot}(s)}} w(e_{st})$$

Example 2.2. Consider the majority graph given in Figure 2.2. The minimal Kemeny score for this graph is 2 and there are two optimal total rankings: total rankings (B,C,A) with edge e_{AB} reversed and (C,A,B) with edge e_{BC} reversed.

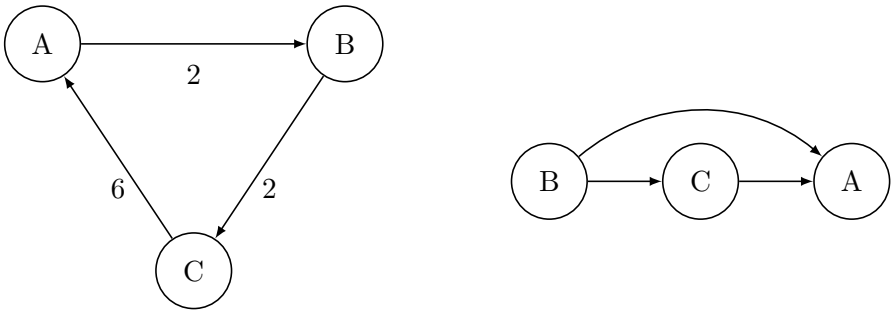

Figure 2.2: Example of a majority graph on the left and the total ranking (B,C,A) represented by a graph on the right

The next paradigm will be very similar to this one and can be seen as the Kemeny paradigm without weights.

2.4 Slater rule

The Slater rule tries to minimize the *Slater score* of a total ranking, which is defined similar to the Kemeny score. We again take the majority graph G and a total ranking r_{tot}. The only difference is that instead of adding the weight of a reversed edge e_{st} to the score we add 1 if $w(e_{st}) > 0$ and 0 if $w(e_{st}) = 0$.

$$SlaterScore(G, r_{tot}) := \sum_{\substack{e_{st} \in E \\ r_{tot}(t) < r_{tot}(s)}} w_{slater}(e_{st})$$

where $w_{slater}(e) = 1$ if $w(e) > 0$, $w_{slater}(e) = 0$ otherwise

Example 2.3. Consider the majority graph given in Figure 2.2. The minimal Slater score for this graph is 1 and there are three optimal total rankings: total rankings (B,C,A) with edge e_{AB} reversed, (C,A,B) with edge e_{BC} reversed and (A,B,C) with edge e_{CA} reversed.

2.5 FLAP rule

The Feedback Linear Arrangement Problem, in short FLAP, is the most complex paradigm of the three discussed in this thesis. The FLAP rule aims to minimize the *FLAP score* which is defined as the *upset* function that takes as arguments a majority graph G with edge set E and a total ranking r.

$$upset(G, r_{tot}) := \sum_{\substack{e_{ij} \in E \\ r_{tot}(j) < r_{tot}(i)}} w(e_{ij})(r_{tot}(i) - r_{tot}(j))$$

One should note that $r(j) < r(i)$ denotes that alternative j is ranked above i since 0 is the highest rank.
The FLAP score is similar to the Kemeny score, but the weight of an inverted edge is multiplied by how 'long' the edge is, i.e. the distance between the two alternatives in the total ranking r.

Example 2.4. Consider the majority graph given in Figure 2.3. The minimal FLAP score is 14 with an optimal total rankings (C,B,D,A) with edges e_{BC}, e_{DB} and e_{AD} reversed. In contrast the optimal Kemeny score would be 6 with optimal total ranking (B,C,A,D) with only edge e_{DB} reversed. The FLAP score with total ranking (B,C,A,D) would be 18 since alternatives B and D have a distance of 3 in (B,C,A,D) so we have to multiply $w(e_{DB})$ by 3. This example also shows that that the FLAP and Kemeny rules produce different optimal solutions.

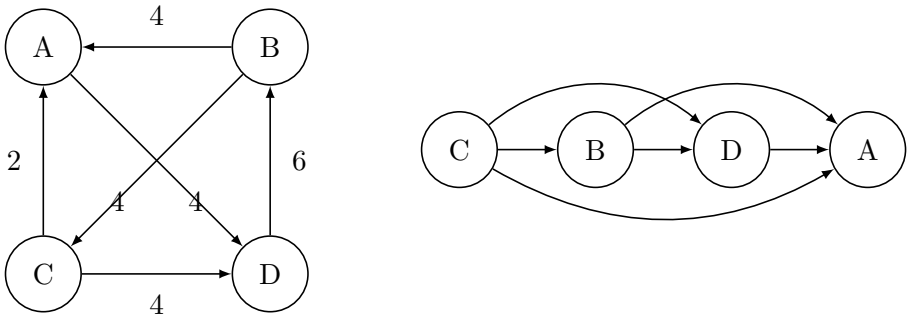

Figure 2.3: Example of a majority graph on the left and the total ranking (C,B,D,A) represented by a graph on the right

Chapter 3

Data

In this chapter we will present the data we used to test our methods on. The problem of finding or generating test data is an essential part of this thesis because, as we will see, the hardness of a problem widely varies across different *datasets*. Each dataset is a collection of votes over the same set of alternatives. So each dataset can be characterized by the number of alternatives and the number of votes.

3.1 PrefLib

PrefLib[4] is a reference library of preference data and contains most datasets that are known. On the search for datasets we came across a few datasets (for example the Sushi dataset) which we all found on PrefLib. For this reason we took all real world data from the PrefLib database. This has the advantage that PrefLib has unified data formats. There are two formats that are interesting for us. The *SOC - Strict Orders Complete List* format contains all datasets that only contain votes which rank all alternatives by a strict order. Because there are few datasets in the SOC format we also looked at the *TOC Orders with Ties - Complete List* format which contains all datasets which have ties in the votes but the votes still rank all the alternatives. To represent all the datasets available we plotted the individual datasets by their number of votes and alternatives in Figure 3.1. As we can see the datasets have a substantial restriction. There are no datasets with a high number of both vote count and alternative count.

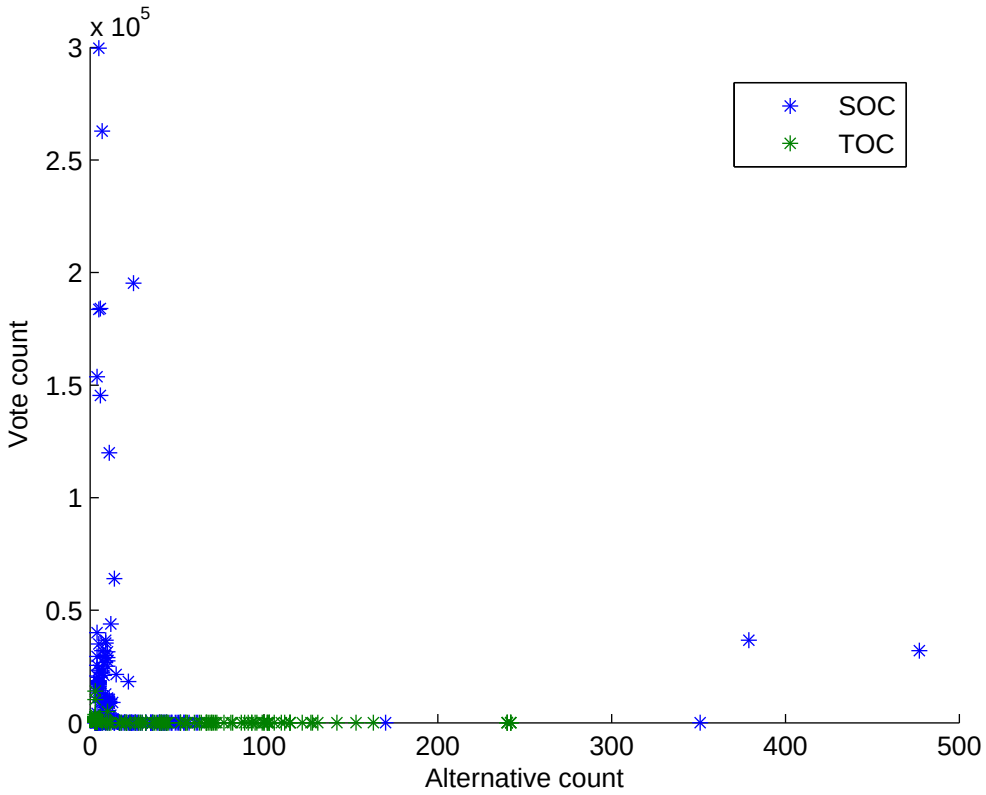

Figure 3.1: Datasets plotted by their characteristics. (i.e. vote count and alternative count)

3.2 Yelp Analysis

Yelp is a crowd-based review platform for local businesses. This means that users can rate a business with a one to five star rating system. Every rating can also include a review comment. The motivation for analysing this data is to extract datasets for our evaluation. The businesses would be the alternatives but the votes have to be extracted in an nontrivial way. We cannot use the star based ranking system because it would be to granular with only 5 preference categories. So we have to analyse the review text of which there are $1'125'457$ in the dataset we used, which was the dataset of the Yelp Dataset Challenge 2014. The dataset specifies $30'944$ business names without duplicates. We will define the analysis method we used in the next chapter.

3.3 Synthetic Vote Generation

Since we want to have a large amount of datasets where we can change the properties as needed we have to design a vote generator. We take as basis the definition given in [3]. As described before each dataset can be characterized by the alternative count and vote count. These two parameters alone are not enough as it does not describe the relation between the votes. For this reason we define the *consensus probability* $p \in [0, 1]$. It is an approximate value that describes the probability at which a vote ranks two alternative the same as some imaginary *guidance vote* which is the same for all votes. We will always choose $p \in [0, 0.5]$ for consistency reasons. Note that with consensus probability $p \leq 0.5$ and guidance vote v the resulting dataset has the same characteristics as a dataset with consensus probability $1 - p$ with the inverted guidance vote of v since the alternatives are interchangeable. Our vote generator takes an alternative count n and a consensus probability p. As guidance vote we always use the vote (a_0,a_1,a_2,...,a_n). So the vote we are constructing agrees with the guidance vote with probability p in each pairwise ranking.

Algorithm 1: Vote Generator

Input : consensus probability $p \in [0, 1]$, alternative count n
Output: one vote represented by a sequence of alternatives (a_x, a_y,...,a_z)

```
G := graph without edges and nodes representing alternatives
foreach two distinct alternatives a_i, a_j with i < j do
    if edge (a_i, a_j) or (a_j, a_i) not already in G then
        r := random number ∈ [0, 1)
        agree := r < p
        if agree then
            add edge (a_i, a_j) to G
        else
            add edge (a_j, a_i) to G
        add all resulting transitive edges to G
return G converted to the transitive sequence
```

Chapter 4

Algorithms Definitions

In this chapter we define each algorithm we implemented in this thesis. The results of these algorithms will be presented in the next chapter.

4.1 Linear Programs

These algorithms serve as a basis of comparison for the other developed heuristics. The Linear Programs have the property that they produce an optimal solution at a reasonable computation time without needing a complex implementation. For the solution to be optimal we have to run the programs as integer linear programs which means that variables can be restricted to $\mathbb{N}$. This has the implication that the program will generally have a higher computation time. A possibility for future work could be to investigate the quality of the non-optimal linear programs.

4.1.1 Linear Program for Kemeny

This linear program is *Linear Program 3* from [5]. Given is a majority graph $G = (A, E, w)$. Note that $w(e) = 0$ if $e \notin E$.

Linear Program for Kemeny

$x_{ab} \in \{0, 1\}$

minimize $\sum_{a,b \in A} w(e_{ab}) x_{ab}$

subject to

for all distinct $a, b \in A, x_{ab} + x_{ba} = 1$ (1)

for all distinct $a, b, c \in A, x_{ab} + x_{bc} + x_{ca} \geq 1$ (2)

$x_{(a,b)} = 1$ signifies that the edge from a to b exists in the total ranking. In other

words the final ranking ranks a above b. This implies that

$$x_{(a,b)} = 1 \iff x_{(b,a)} = 0$$

Which is enforced by constraints (1).
Constraints (2) ensure transitivity of the rankings. It can also be seen as the condition that for every group of 3 nodes in both directions of traversal there is at least one edge in traverse direction. In other words there are no cycles.
The objective function minimizes the sum of the weights of all edges which have to be reversed to satisfy the total ranking. This is precisely the definition of the Kemeny rule.

4.1.2 Linear Program for Slater

This linear program is very similar to the Linear Program for Kemeny. All specifications stay the same except the weight function, which is now the earlier defined w_{slater}. Again note that $w_{slater}(e) = 0$ if $e \notin E$.

Linear Program for Slater

$x_{ab} \in \{0, 1\}$

minimize $\sum_{a,b \in A} w_{slater}(e_{ab}) x_{ab}$
subject to
for all distinct $a, b \in A, x_{ab} + x_{ba} = 1$ (1)
for all distinct $a, b, c \in A, x_{ab} + x_{bc} + x_{ca} \geq 1$ (2)

The objective function minimizes the sum of all edges that reverse an edge from the majority graph in the total ranking. In contrast to the Kemeny rule, the Slater rule does not take into considerations the margin of how much a pairwise contest was won but just who was the winner.

4.1.3 Linear Program for FLAP

We introduce variables $r_0, r_2, ..., r_{n-1}$ which indicate the total ranking of the alternatives. This means that if $r_i = k$ the i'th alternative has ranking k, with $n-1$ being the highest ranking. Note that this is in contrast to the order convention in this paper, but the result can easily be inverted after the computation.
We also introduce variables $a_{i,j} \in \{0, 1\}$ with $i, j \in \{0, 1, ..., n-1\}$ The following constraints establish that no two alternatives have the same ranking.

$$\begin{array}{cccccc} a_{0,0} & a_{1,0} & a_{2,0} & a_{3,0} & ... & a_{n-1,0} \\ a_{0,0} & a_{1,1} & a_{2,1} & a_{3,1} & ... & a_{n-1,1} \\ a_{0,2} & a_{1,2} & a_{2,2} & a_{3,2} & ... & a_{n-1,2} \\ a_{0,3} & a_{1,3} & a_{2,3} & a_{3,3} & ... & a_{n-1,3} \\ a_{0,4} & a_{1,4} & a_{2,4} & a_{3,4} & ... & a_{n-1,4} \\ ... & ... & ... & ... & ... & ... \\ a_{0,n-1} & a_{1,n-1} & a_{2,n-1} & a_{3,n-1} & ... & a_{n-1,n-1} \end{array}$$

$a_{i,k} = 1$ indicates that alternative i is ranked $k'th$. To ensure that no two alternatives are ranked the same rank we demand $\forall k \leq n-1$

$$\sum_{i=0}^{n-1} a_{i,k} = 1$$

and to ensure that one alternative cannot hold two ranks we demand $\forall i \leq n-1$

$$\sum_{k=0}^{n-1} a_{i,k} = 1$$

Now we can compute the rank r_i of alternative i.

$$r_i = \sum_{k=0}^{n-1} a_{i,k} k$$

We introduce a new set of variables $d_{x,y}$ which indicate the difference between the ranking of alternative x and y. We achieve this by introducing the constraints

$$d_{x,y} \geq r_x - r_y$$

$$d_{x,y} \geq 0$$

So $d_{x,y}$ specifies how much higher x is ranked above y and is 0 if x is ranked below y. We prevent the case where x is ranked above y by adding the objective function

$$\textbf{minimize} \sum_{x,y \in \{0,1,...,n-1\}} w(e_{xy}) d_{x,y}$$

where $w_{x,y}$ is the weight of edge e_{xy} in the majority graph. If we compare the objective function to the FLAP score definition

$$upset(G, r) := \sum_{r(i)<r(j)} weight(e_{j,i})(r(j) - r(i))$$

we see that our objective function is equivalent to the upset function. If an alternative x is ranked above y in the final ranking (i.e. $r_x > r_y$, as $n-1$ is the highest ranking.) $d_{x,y} = r_x - r_y > 0$ and $d_{y,x} = 0$. Now if x wins the pairwise election against y, $w(e_{xy}) = 0$ and $w(e_{yx}) > 0$. So if x is ranked above

y in the total ranking we do not pay a price because both terms $w(e_{xy})d_{x,y}$ and $w(e_{yx})d_{y,x}$ are 0.
If we however rank y higher than x, $d_{x,y} = 0$ and $d_{y,x} > 0$. So we pay $w(e_{yx})d_{y,x}$ which is exactly what the definition of FLAP describes. Since we know the values of $w(e_{ij})$ before running the solver, we can exclude the terms where $w(e_{ij}) = 0$, essentially halving the length of the sum.

Linear Program for FLAP

$a_{i,j} \in \{0, 1\}$

minimize $\sum_{x,y \in \{0,1,\ldots,n-1\}} w(e_{xy})d_{x,y}$
subject to
$\sum_{i=0}^{n-1} a_{i,k} = 1$
$\sum_{k=0}^{n-1} a_{i,k} = 1$
$r_i = \sum_{k=0}^{n-1} a_{i,k}k$
$d_{x,y} \geq r_x - r_y$
$d_{x,y} \geq 0$

4.2 Heuristic Methods

In this section we define heuristic algorithms. They usually trade speed with optimality. By comparing the results to the results of the linear programs we can evaluate the quality of the heuristic results.

4.2.1 Copeland's Method

The Copeland method[6] is a well known rank aggregation function which simply ranks the alternatives in the majority graph by their in-degree. The intuition is, that the more pairwise victories an alternative has, the higher it should rank in the total ranking. Note that this method does not consider the magnitude of the victories. In other words it does not include the weights of the edges in the calculation. For this reason the pure Copeland method should be used for Slater, since Slater does not consider the weights as well. An adapted version for Kemeny will be presented after this one.

Algorithm 2: Copeland's method

Input : Majority graph G
Output: Nodes of G sorted by in-degree

```
1 N := nodes(G)
2 return mergesort(N)                    // mergesort on in-degree
```

4.2.2 Weighted Copeland's Method

For this algorithm we take the basic Copeland method and instead of sorting the nodes by in-degree we sort them by the added weights of the incoming edges. This change makes it more suitable for Kemeny or possibly FLAP since those paradigms also have weights in their definitions.

Algorithm 3: Weighted Copeland's method

Input : Majority graph G
Output: Nodes of G sorted by in-degree times weights

```
1 N := nodes(G)
2 return mergesort(N)      // mergesort on in-degree times weights
```

4.2.3 Borda rule

Another well known and very old rank aggregation function is the Borda rule. It uses the *Borda count* which was devised in 1770 by Jean-Charles de Borda and is calculated as follows. For each single vote the Borda count of a specific alternative is the number of alternatives that are ranked lower. So the lowest alternative would have a Borda count of 0. Furthermore the total Borda count of an alternative for all votes is the sum of all Borda counts for each vote.
We can easily read this data from the weights of the edges of the majority graph. For any alternative we compute the total Borda count as follows. In the majority graph we look at each leaving edge e_{out} and add $\frac{|V|+w(e_{out})}{2}$ where $|V|$ is the number of voters. For each entering edge e_{in} we add $\frac{|V|-w(e_{in})}{2}$.
This method should be applied to the Kemeny and possibly FLAP rule since an unweighted Borda rule for Slater would just result in the Copeland method.

Algorithm 4: Borda Rule

Input : Majority graph G
Output: Nodes of G sorted by Borda count

```
1 N := nodes(G)
2 return mergesort(N)                    // mergesort on borda count
```

4.2.4 Search with pruning

This is a slow but optimal method which searches through the solution space like a tree. It has been first described for rank aggregation in [7] where the method is called *branch and bound*. The method tries to reduce computation time by pruning branches of the tree that are guaranteed to be worse than the current best solution. It accomplishes this by keeping track of the best paradigm score it has yet seen. This works for Kemeny and Slater since we know that the score only

increases when adding edges. Unfortunately the score of FLAP is too complex since the impact of an edge on the score is not static. A possible optimization that has been implemented is to run a faster but not optimal rank aggregation function and start from the score of that solution. This allows us to prune bad search paths much earlier.
The algorithm listed in Algorithm 5 will start with an empty graph we will call `currentGraph`. We add all the nodes to `currentGraph` without the edges. We also initialize the `currentScore` with 0. Now we can optionally use a fast but inexact rank aggregation function we will call the *preSolver* function to get a better starting point. If we do we set `bestScoreYet` and `bestOrderYet` to the solution. If we start from nothing we set them 0 and *empty* respectively. Then we set up the search stack `unvisitedEdgesStack` and fill it with all edges from G.
Now that we have set up everything we can start searching the solution tree. We start with the edge on the top of the stack and pop it. We then test if we would introduce a cycle by adding the edge to `currentGraph`. If no we add the edge and then add all edges that are implied by transitivity and remove them from `unvisitedEdgesStack`. Then we update `currentScore` and test if `currentScore` is still smaller than `bestScoreYet`. If yes and `unvisitedEdgesStack` is empty we found a better solution and we can adapt `bestOrderYet` and `bestScoreYet`. Otherwise if `unvisitedEdgesStack` is not empty we go further down the recursion. After all these cases we reset `currentGraph` and `currentScore` and reinsert `edge` into `unvisitedEdgesStack`. We then try the same thing with `edge` pointing in the other direction before returning.

Algorithm 5: Search with Pruning

Input : Majority graph G, optional $preSolver()$
Output: order of nodes with the best possible paradigm score

```
1  currentGraph := nodeSet(G) // this graph will keep our progress
2  currentScore := 0
3  bestOrderYet := preSolver(G) // optional
4  bestScoreYet := score(bestOrderYet) // best score we have seen
5  unvisitedEdgesStack := edgeSet(G)
6  visit(edge ∈ unvisitedEdgesStack)
7  return bestOrderYet
8  ____________________________________
9  Procedure visit(edge)
10     do Twice
11         remove(unvisitedEdgesStack, edge)
12         scoreBefore := currentScore
13         if no path from targetNode of edge to sourceNode of edge then
               // calculated by breadth first search
14
15             currentGraph.add(edge)
16             addedTransitiveEdges := all edges that must be added to
                ensure transitivity
17             currentGraph. add(addedTransitiveEdges)
18             unvisitedEdgesStack.remove(edge)
19             update currentScore
20             if currentScore < bestScore then
21                 if unvisitedEdgesStack is empty then
22                     bestScoreYet := currentScore
23                     bestOrderYet := sorted nodes in currentGraph
24                 end
25                 else
26                     visit(edge ∈ unvisitedEdgesStack)
27                 end
28             end
               // reset changes
29             currentGraph.remove(edge, addedTransitiveEdges)
30             currentScore := scoreBefore
31             unvisitedEdges.add(edge, addedTransitiveEdges)
32         end
33         reverse(edge)// try the same thing with the edge pointing
               in the other direction
```

4.2.5 Recursion on set of similar candidates

This rank aggregation function is taken from [2] and tries to divide the majority graph into sets of *similar candidates*. First we need to give a definition of a set of similar candidates.

Definition 4.1 (Similar Candidates)**.** A subset $S \subseteq A$ is a set of similar candidates if for any $s_1, s_2 \in S$ and for any $a \in A - S$, $s_1 \to a$ if and only if $s_2 \to a$ (and hence $a \to s_1$ if and only if $a \to s_2$).

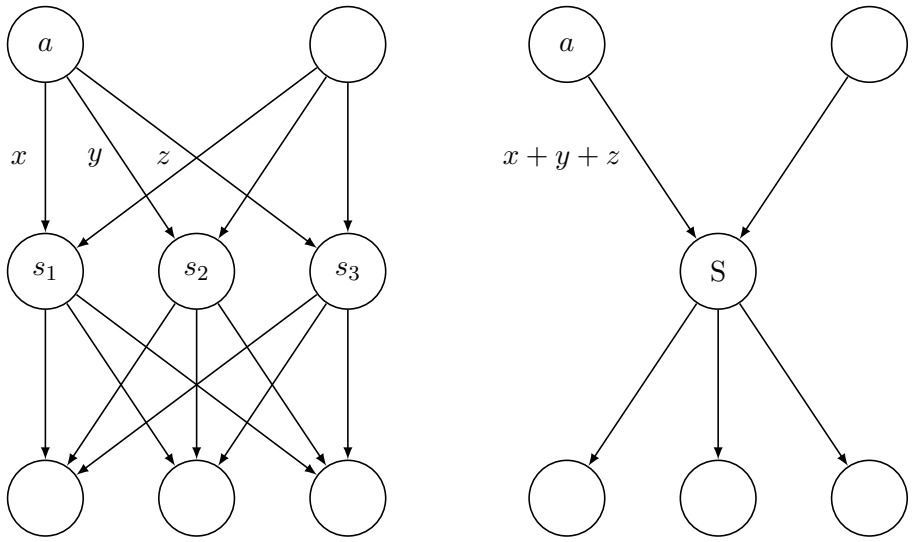

Figure 4.1: The original majority graph without the edges irrelevant to classifying s_1, s_2 and s_3 as similar on the left and after the transformation on the right with $S = (s_1, s_2, s_3)$

If these conditions hold for S in the majority graph, S is a set of similar candidates. We can see right away that there are always trivial similar sets. One is the set $S_{all} = A$ with all candidates in A and the sets containing at most one candidate. We will see that these cases are not interesting for improving computation time.
An interesting property for us is that we can transform a majority graph G with a similar set S by regarding all orderings between nodes outside and inside S as orderings with one meta-alternative representing all nodes in S. So we could solve the smaller transformed majority graph G' and the sub-graph consisting of the nodes in S which is also a majority graph and then insert the result in the result of G'. Now it becomes clear why the trivial similar sets are not interesting because they do not change the complexity of the original majority graph. Another property of this method is that we can apply it recursively until there are no sets of nontrivial similar candidates. At the base case we still have to

apply an arbitrary rank aggregation function we will call the *base case algorithm*. Furthermore we introduce a theorem from [2].

Theorem 4.2. *If S consists of similar candidates, then there exists a total ranking r_{tot} with optimal Slater score in which the alternatives in S form a (contiguous) block (that is, there do not exist $s_1, s_2 \in S$ and $a \in A - S$ such that $r_{tot}(s_1) < r_{tot}(a) < r_{tot}(s_2)$).*

This theorem shows that we do not lose optimality by reducing the majority graph with a set of similar candidates. So if the base case algorithm also is optimal with regard to Slater score, the whole recursion method will be optimal too.
The potential for computation time reduction follows from the observation that the computation time increases superlinearily with the number of alternatives. The fastest reduction of the number of alternatives would be to choose a set of similar candidates which size is closest to half of the size of the set of all alternatives. But since such a search could be costly it could be faster to just pick the first similar set of candidates we can find. For this uncertainness we tested both variants. The set of similar candidates is found by transforming the problem into a Horn satisfiability problem as described in [2].
For every alternative a_i we define a boolean variable v_i. Then, for every ordered triplet of distinct alternatives a_1, a_2, a_3, if there are either edges from a_1 to a_3 and a_3 to a_1 or edges from a_2 to a_3 and a_3 to a_1 we introduce the clause $v_1 \wedge v_2 \implies v_3$. We can now use a Theorem from [2].

Theorem 4.3. *A setting of the variable v_i satisfies all the clauses if and only if $S = \{a_i \in A : v_i = true\}$ consists of similar candidates.*

With the help of Theorem 4.3 we can identify a set of similar candidates by choosing two alternatives a_i, a_j and set their respective variables v_i, v_j to *true*. We then go through the clauses and set further variables to true if they are implied. We stop when all clauses are satisfied. By setting two variables to *true* at the start we avoid trivial sets of similar candidates of size one. We also want to avoid the trivial set of similar candidates where $S = A$. So after all clauses are satisfied, we have to check if all variables are set and if they are, throw the result away. We can repeat our search with every pair of two alternatives until we have found a desired set of similar candidates.
Theorem 4.2 does not hold with the Kemeny score, as we will show in an example.

Example 4.4. Given the majority graph in Figure 4.2 we can see that there is a nontrivial set $S = \{s_1, s_2\}$ of similar candidates. Furthermore we can see that there is a total ranking $r_{tot} = (s_1, a_2, a_1, s_2)$ with a Kemeny score of 2. We already notice that the set S is split between the other alternatives and in turn would violate theorem 4.2 with regard to the Kemeny score. On the reduced majority graph we can also see that the recursion on the recursion algorithm would have a

Kemeny score of at least 100 and would therefore violate the optimality property we have with the Slater score.

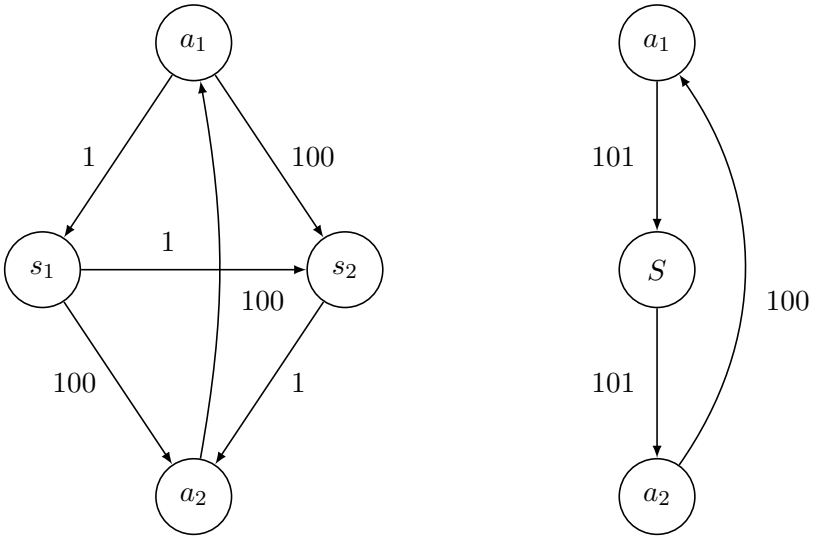

Figure 4.2: The original majority graph on the left and the reduced majority graph with similar set $\{s_1, s_2\}$

Algorithm 6: Recursion with similar candidates

Input : majority graph G, base case algorithm $baseCase()$
Output: sorted nodes based on the base case algorithm

1 $similarSet$:= optimal nontrivial set of similar candidates *or* first found nontrivial set of similar candidates
2 **if** $similarSet$ *not null* **then**
3 $subGraph := G \cap similarSet$
4 $subResult$:= solve $subGraph$ with 'Recursion with similar candidates' algorithm
5 $subNode$:= node representing the $subGraph$ in the $hyperGraph$
6 $metaGraph := (G \setminus similarSet) \cup subNode$
7 $metaResult$:= solve $metaGraph$ with 'Recursion with similar candidates set' algorithm
8 replace $subNode$ with $subResult$ in $metaResult$
9 $result := metaResult$
10 **else**
11 $result := baseCase(G)$
12 **return** $result$

4.2.6 Local search

The algorithm described in this section aims to optimize a total ranking that is already close to optimal. For each alternative it tries to find a new position to improve the score of the specific paradigm. It is best used to improve the result even further after a non-optimal method which we call the *presort algorithm* has been applied. It stops searching if it does not find a better position for any of the alternatives. This could lead to a very high computation time because we start the search at the very beginning every time we have found an improvement. The *scoreEffect(positionBefore, positionAfter)* procedure takes the edges attached to the alternative a at *positionBefore* and calculates the effect of theses edges if a were at position *positionAfter*.

Algorithm 7: Local Search

Input : majority graph G, presort algorithm $presort()$
Output: sorted nodes based on the presort algorithm $presort()$ plus local improvements

```
1  preResult := presort(G)
2  foreach alternative ∈ preResult do
3      oldScoreEffect := scoreEffect(position of alternative, position of
         alternative)
4      foreach newPosition ∈ preResult do
5          newScoreEffect := scoreEffect(position of alternative,
             newPosition)
6          if newScoreEffect < oldScoreEffect then
7              set position of candidate to newPosition
8              reset foreach counters
9              break
10 return preResult
11 ---------------------------------------------------
12 Procedure scoreEffect(positionBefore, positionAfter)
13     before := true
14     newScoreEffect := 0
15     foreach position in preResult do
16         if position ≥ positionAfter then
17             before := false
18         if position ≠ positionBefore then
19             if before then
20                 newScoreEffect := newScoreEffect+weight of edge from
                     positionBefore to position
21             else
22                 newScoreEffect := newScoreEffect+weight of edge from
                     position to positionBefore
23     return newScoreEffect
```

4.3 Other Methods

In this section we describe methods which are not rank aggregation functions but methods which are still used in this thesis.

4.3.1 Probability Estimation

As we will see, the consensus probability p of synthetic datasets greatly influences the hardness of the rank aggregation problem. The incentive for estimating p would be that we could apply this to real world datasets and therefore estimate their hardness. If we look at the definition of the consensus probability we see that it uses a guidance vote which is then discarded. So if we want to estimate p we have to come up with a workaround to estimate this guidance vote. We devised two approaches to this problem which we will define now.
The first method is called *probability estimation with alternative distance* and it takes each vote individually as guidance vote v_{guide} and compares the index of the alternatives with the indexes in the other votes. We sum up all the differences and then normalize the result with respect to the number of votes and the number of alternatives.

Algorithm 8: probability estimation with alternative distance

Input : set of votes V
Output: normalized estimation for consensus probability

```
1  if |V| < 2 then
2      return 0
3  avgerageDifference := 0
4  foreach baseVote ∈ V do
5      totalDifference := 0
6      foreach currentVote ∈ V do
7          currentDifference := 0
8          foreach alternative ∈ baseVote do
9              difference := difference + |index of alternative in baseVote
                 − index of alternative in currentVote|
10     averageDifference :=
         averageDifference + totalDifference/(|V| − 1)
11 return averageDifference/(|V|·number of alternatives^2)
```

The second method is called *probability estimation with alternative pair disagreement* and it works the same way like the first but instead of comparing the differences of the indices it counts how many pairs of alternatives are ordered

differently between v_{guide} and the other votes.

Algorithm 9: probability estimation with alternative pair disagreement

Input : set set of votes V
Output: normalized estimation for consensus probability

```
1  if |V| < 2 then
2      return 0
3  avgerageDifference := 0
4  foreach baseVote ∈ V do
5      totalDifference := 0
6      foreach currentVote ∈ V do
7          currentDifference := 0
8          foreach distinct alternative1, alternative2 ∈ baseVote do
9              if ordering of alternative1 and alternative2 in baseVote
                 differs from ordering in currentVote then
10                 currentDifference := currentDifference + 1
11         totalDifference :=
             totalDifference + currentDifference/((number of
             alternatives^2)/2− number of alternatives)
12     averageDifference :=
         averageDifference + totalDifference/(|V| − 1)
13 return averageDifference/(|V| − 1)
```

4.3.2 Yelp 'better than'-analysis

This is an automatic analysis method which uses the comments of the reviews as a data source. First we only look at reviews with the substring 'better than' in their comment. Then we look at the part after the 'better than' substring. The length of this suffix can be chosen beforehand. We then search the suffix for the name of any other business. The resulting reviews can be used as a pairwise comparison. More precisely the review is believed to rank the reviewed business higher than the business found in the suffix. It should be obvious that there will be many false positives. We will discuss the issues with this method in the next chapter.

Chapter 5

Results

In this section we will show the results our testing produced. We conducted our testing as follows. We first generated a large set of test cases with the vote generation mechanism we described earlier. More precisely we generated datasets for each consensus probability category (0, 0.1, 0.2, 0.3, 0.4, 0.5) and in each category we have the subcategory of an alternative count of 3 to 100 and in those subcategories we generated 50 different datasets. When not specified otherwise, the result graphs show the results of all datasets generated. We capped the computation time at 300 seconds and if we exceed the cap we note a computation time of -1 seconds. We abort the computation altogether if we reach an alternative count where all datasets exceed the time cap. We have normalized the vote count to 100 as the number of votes has no impact on the complexity of the majority graph. It just scales the weights of the edges but does not change the structure of the graph.

5.1 Linear Programs

These are the results of the linear programs we described earlier. We are not that interested in their running time, but rather the score the solutions produced. We will use the optimal score of the linear programs to compare to the less optimal but faster heuristics. For this reason we will not discuss the figures much in this section but just remark noteworthy characteristics. We used the Gurobi Optimizer[1] for our implementation.

5.1.1 Linear Program for Kemeny

From Figure 5.1 we can see that the computation time will increase with the number of alternatives and consensus probability. We will see that this is a characteristic of every rank aggregation function.

[1] www.gurobi.com/products/gurobi-optimizer

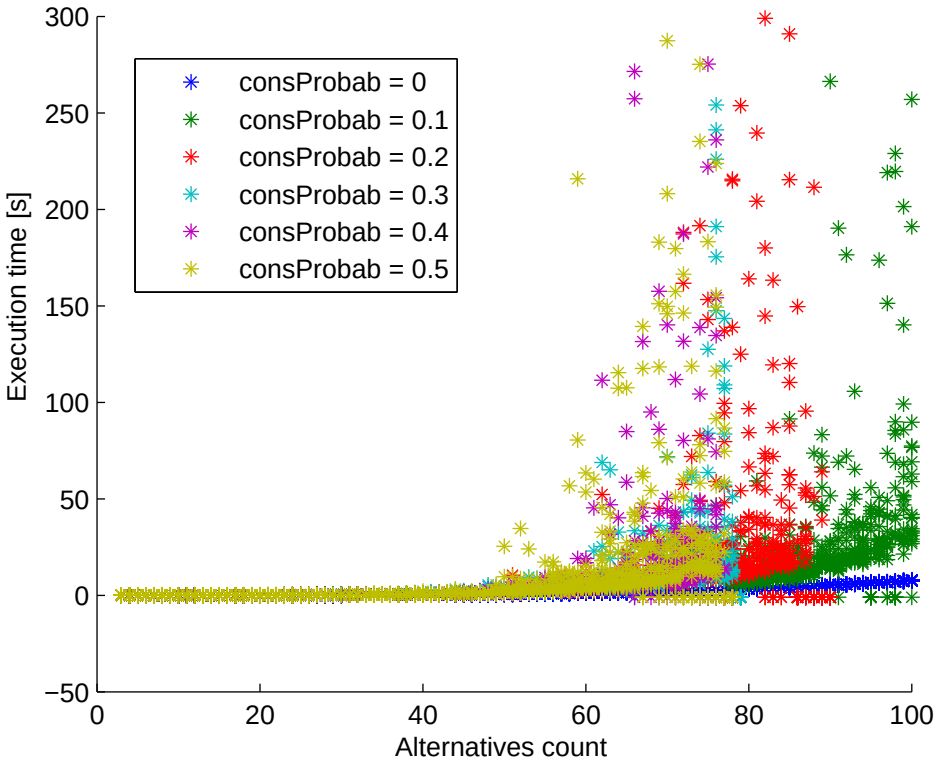

Figure 5.1: Computation time of linear program for Kemeny, 20 cases per alternative count category

5.1.2 Linear Program for Slater

The graph in Figure 5.2 is very similar to the one in Figure 5.1. The only difference is that the computation time is worse overall. This can be explained by the characteristics of linear programming and the Slater definitions. Essentially the linear programs for Slater and Kemeny have to decide in which direction the edges should point to. But in contrast to Kemeny, with Slater all the edges have the same weight. So the linear program which uses branch and bound methods does not have an indicator for which edge to start the search.

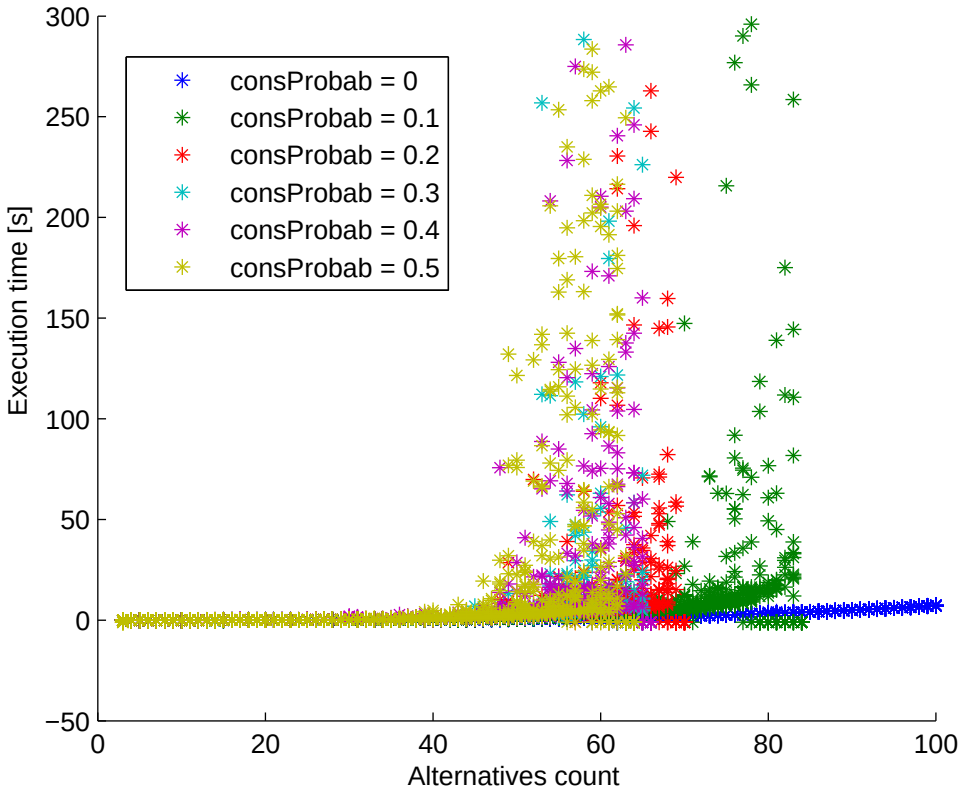

Figure 5.2: Computation time of linear program for Slater, 20 cases per alternative count category

5.1.3 Linear Program for FLAP

As we can see in Figure 5.3 the computation time of the FLAP cases explodes after just a few alternatives are introduced. For this reason we cannot compare any algorithms for the FLAP paradigm since cases with very few alternatives are not very interesting. The only category which could be evaluated are the datasets with consensus probability 0 which means that all votes always take the inverse vote of the guidance vote. In other words they are all the same, which makes the rank aggregation problem trivial.

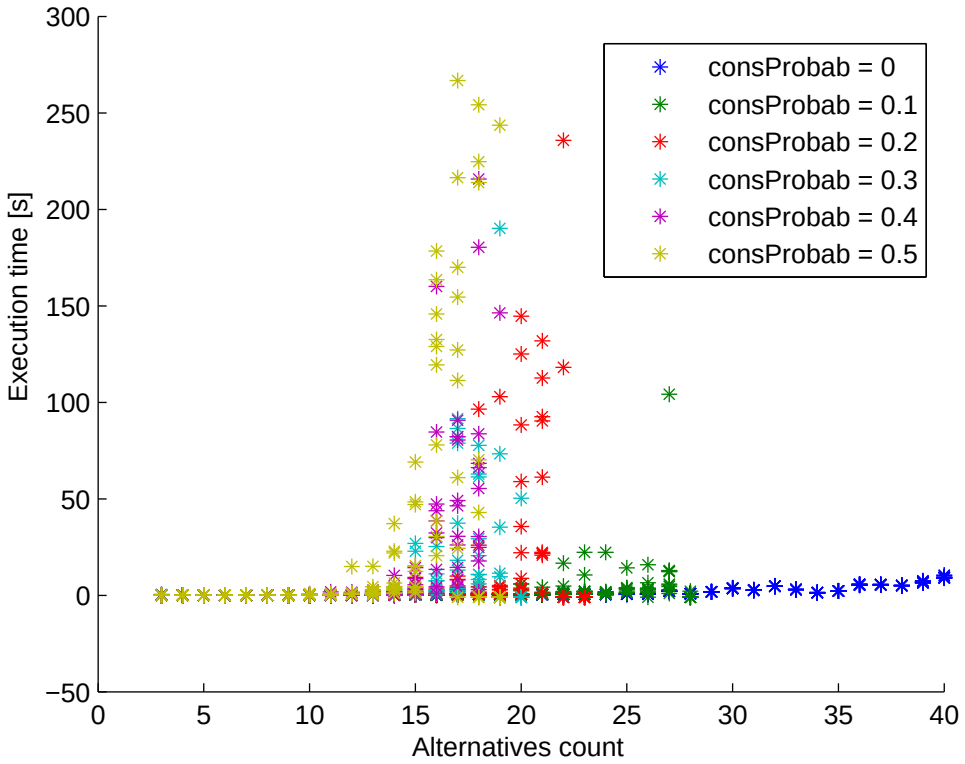

Figure 5.3: Computation time of linear program for FLAP, 10 cases per alternative count category

5.2 Heuristic Methods

In this section we show the results of the implemented heuristic methods and compare them with each other and with the linear programs.

5.2.1 Copeland's Method

Since the algorithm is very basic, the problem is solved nearly instantaneously, on the test machine specifically in 10 to 100μs. For this reason we only compare the Slater score to the optimal linear program for Slater. We have computed comparison graphs for consensus probability 0.1 to 0.5. We will only show the graphs for consensus probability 0.1 and 0.5 since the intermediary results can be interpolated. From Figure 5.4 and Figure 5.5 we can see that scorewise the Copeland method is worse than the linear program but only by a constant difference. How this relation could change for more alternatives we can only

imply, since data for higher alternative count is not available because of the exponential running time of the linear program. All in all the Copeland method is a very fast approximation algorithm and has been used as a building block in other algorithms we discussed.

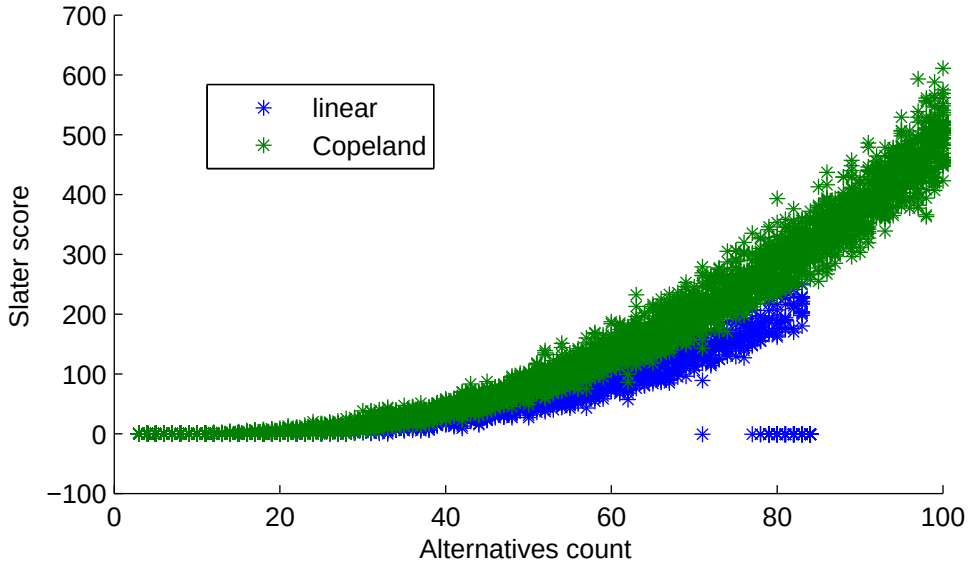

Figure 5.4: Slater score of linear program for Slater and Copeland method with consensus probability = 0.1

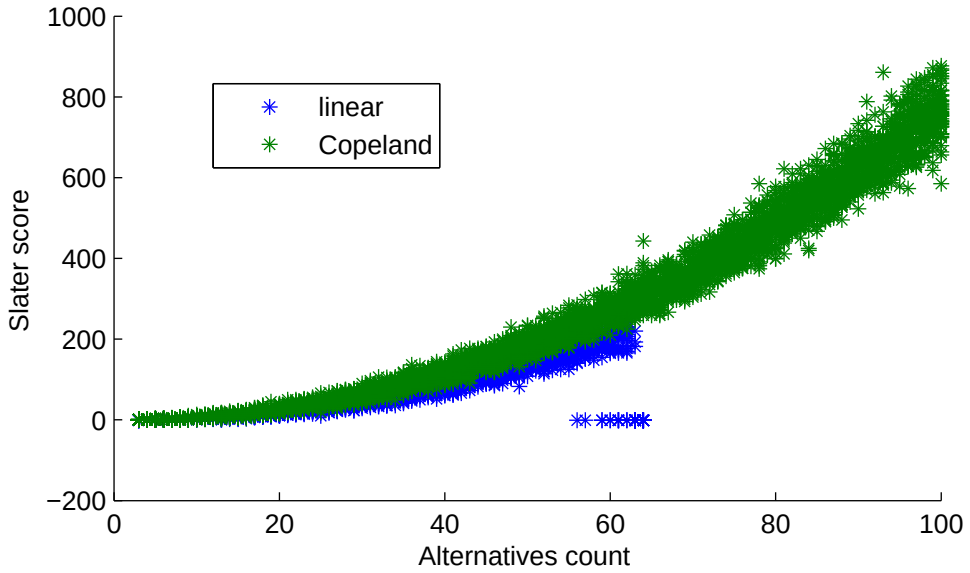

Figure 5.5: Slater score of linear program for Slater and Copeland method with consensus probability = 0.5

5.2.2 Weighted Copeland's Method

Everything that has been said about the unweighted version can also be said about the weighted Copeland method but with regard to the Kemeny score. Because the linear program for Kemeny runs faster than the Slater version we can compare the score up to a count of 100 alternatives for consensus probability 0.1. But we do not learn anything new as the constant difference of the two scores just continues as before. The results are shown in Figure 5.6 and Figure 5.7.

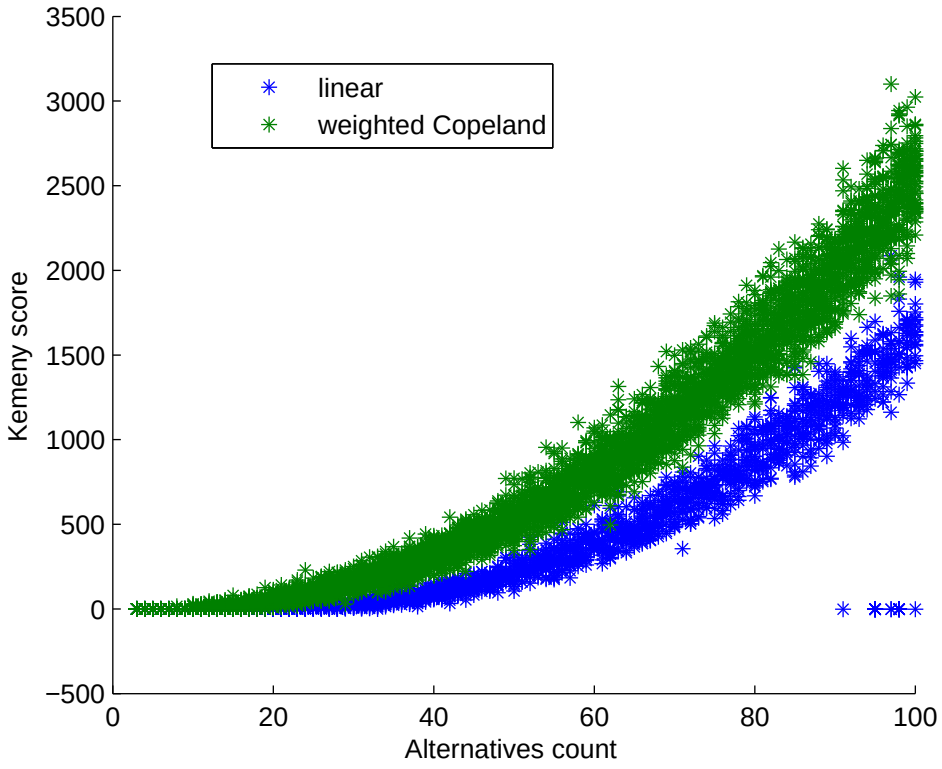

Figure 5.6: Kemeny score of linear program for Kemeny and weighted Copeland method with consensus probability = 0.1

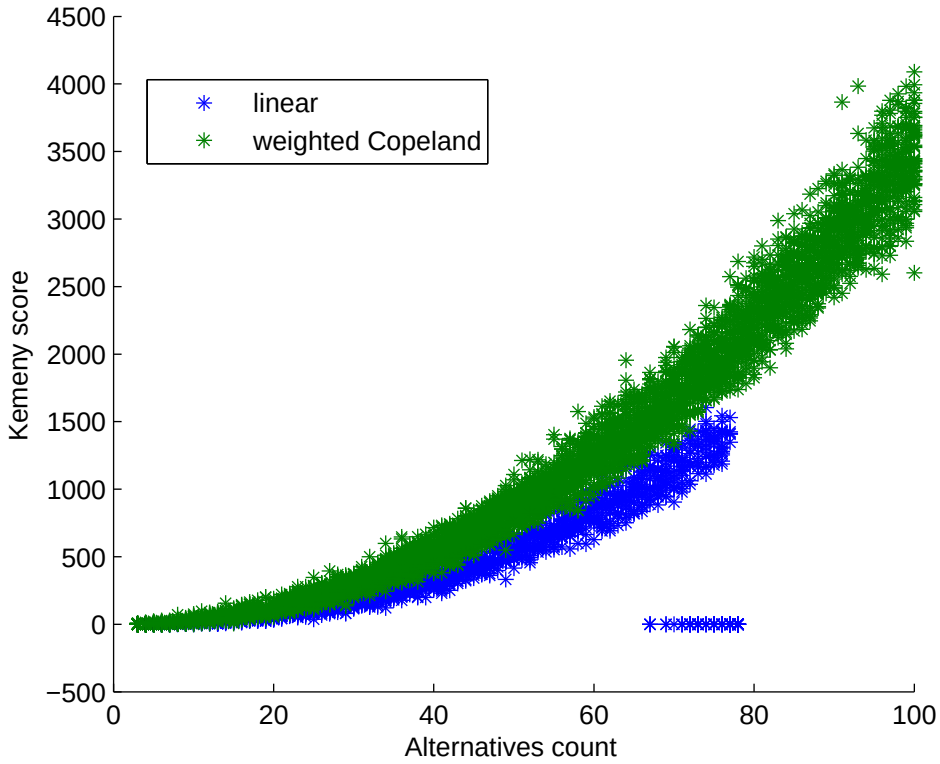

Figure 5.7: Kemeny score of linear program for Kemeny and weighted Copeland method with consensus probability = 0.5

5.2.3 Borda rule

The computation for the Borda is very fast. The time ranges from 10 to 1000μs. It is just slightly slower than the weighted Copeland method. For this reason we will compare the Kemeny score with the Copeland method in Figure 5.8 and Figure 5.9. From Figure 5.9 we can deduce that Borda and weighted Copeland are the same regarding their score. And in Figure 5.8 we see that the score of the weighted Copeland is better. Combining this with the fact that Borda is slightly slower we can deduce that the weighted Copeland method is superior to the Borda rule.

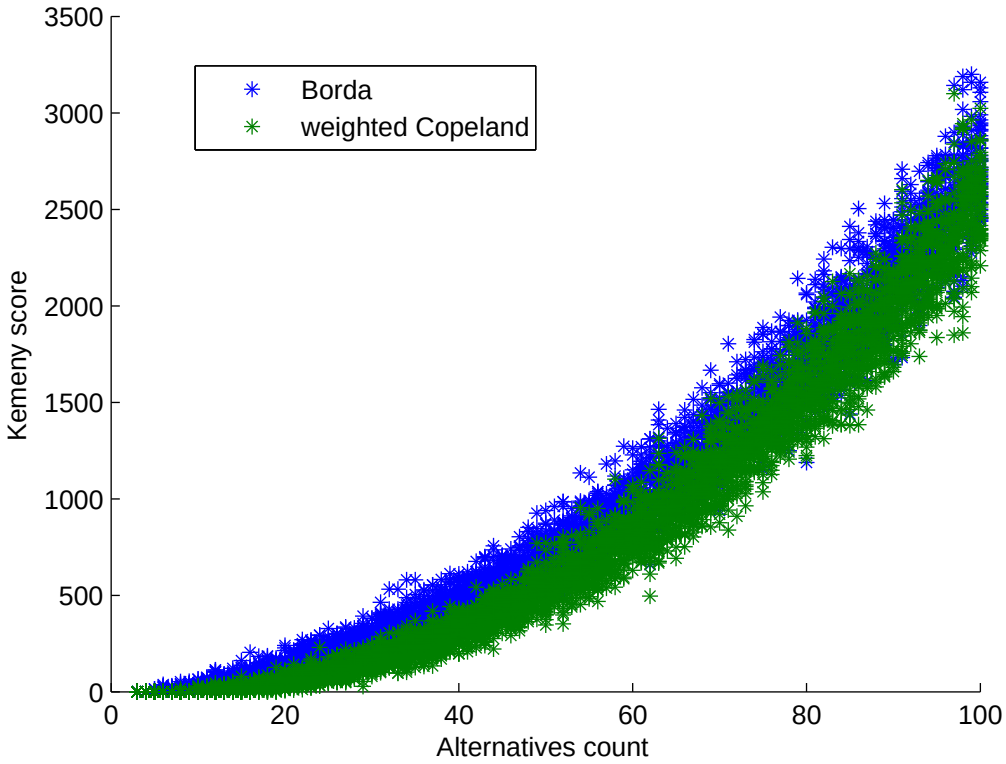

Figure 5.8: Kemeny score of Borda and weighted Copeland method with consensus probability = 0.1

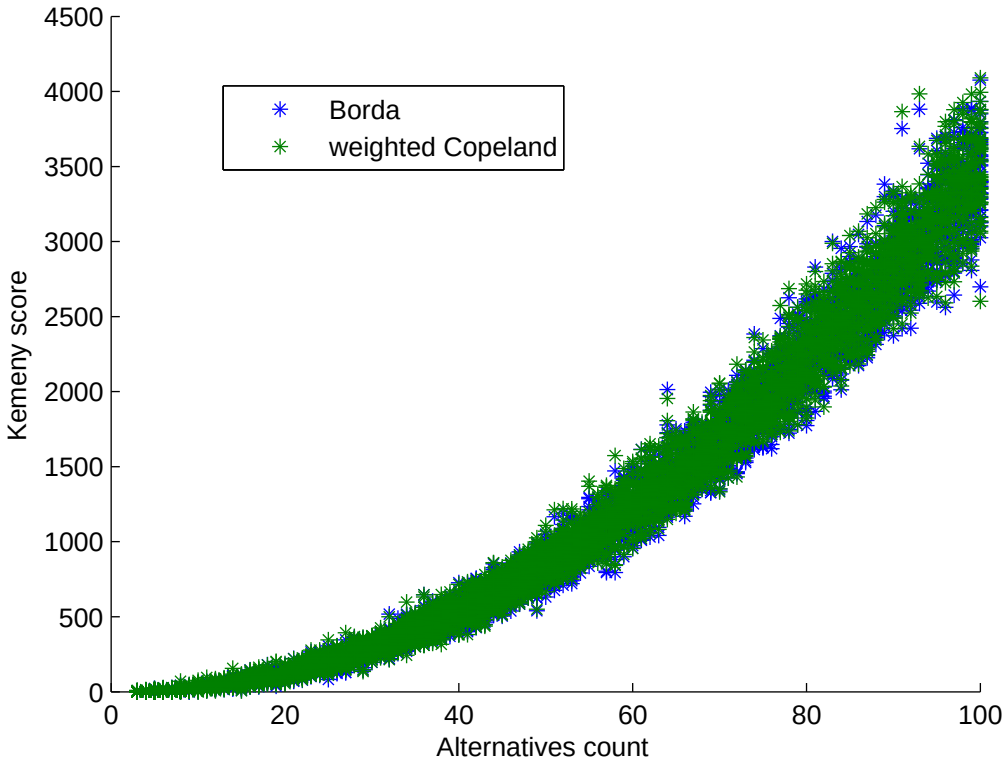

Figure 5.9: Kemeny score of Borda and weighted Copeland method with consensus probability = 0.5

5.2.4 Search with pruning

Since this algorithm searches the whole solution space we know that it produces the optimal solution we only have to look at the computation time. We decided to run our computations with the Kemeny paradigm and used the weighted Copeland method as our `preSolver` function. The results are displayed in Figure 5.10. The graph shows that a search over the solution space results in exponential running time for an NP-hard problem even when cutting off branches of the search tree like the prune method does. It is noteworthy that the datasets with consensus probability 0 are solved virtually instantaneously. The reason is that the `preSolver` function finds the already optimal solution with a Kemeny score of 0. So all other branches in the search tree get cut off because any edge that gets changed results in a score greater than 0. We also ran punctual tests on the Slater paradigm with the Copeland method as `preSolver`. The results were similar.

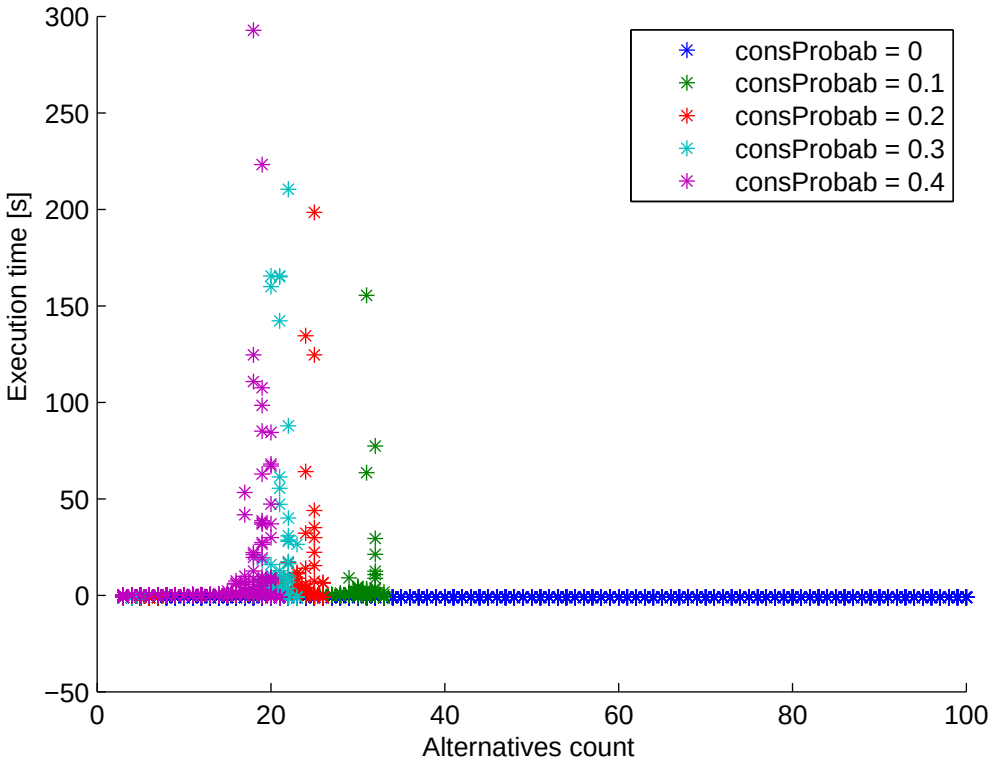

Figure 5.10: Computation time of search with pruning, 20 cases per alternative count category

5.2.5 Recursive on set of similar candidates

For our evaluation we used the Slater paradigm and the linear program for Slater for our base case algorithm. Since our base case algorithm is optimal our resulting algorithm will be optimal too. Furthermore we take the first set of similar candidates we find and do not search for the optimal set. The resulting computation times are shown in Figure 5.11. When we compare the data to the linear Slater we can see that the running time is actually worse. The reason for this is the overhead of finding the similar set of candidates.

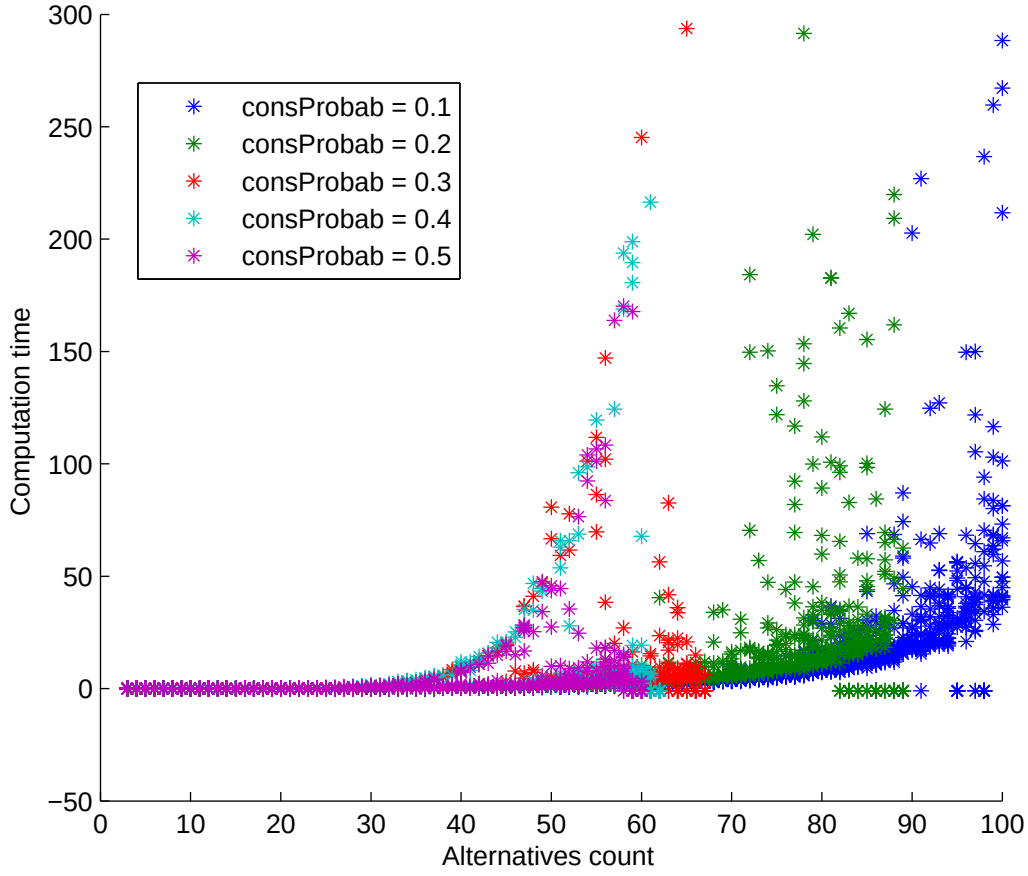

Figure 5.11: Computation time of Recursive on set of similar candidates, 20 cases per alternative count category

Next we look at the size of the optimal similar set of candidates. The motivation behind this is that we can estimate how much of a benefit we actually could gain from splitting the majority graph into these two sets. Note that the optimal set would split the majority graph in half. Figure 5.12 and Figure 5.13 show that there are no beneficial similar sets for higher alternatives counts. For higher consensus probabilities it is even worse with datasets higher than 20 alternatives having optimal sized sets of similar candidates with sizes 0 to 2. This makes it close to impossible for the algorithm to improve computation time since there are only barely beneficial sets of similar candidates for higher alternative counts. So our algorithm just spends time by searching for a similar set that is either not present or very small. For this reason we stopped investigating further since the basis of the whole method is to find good sets of similar candidates.

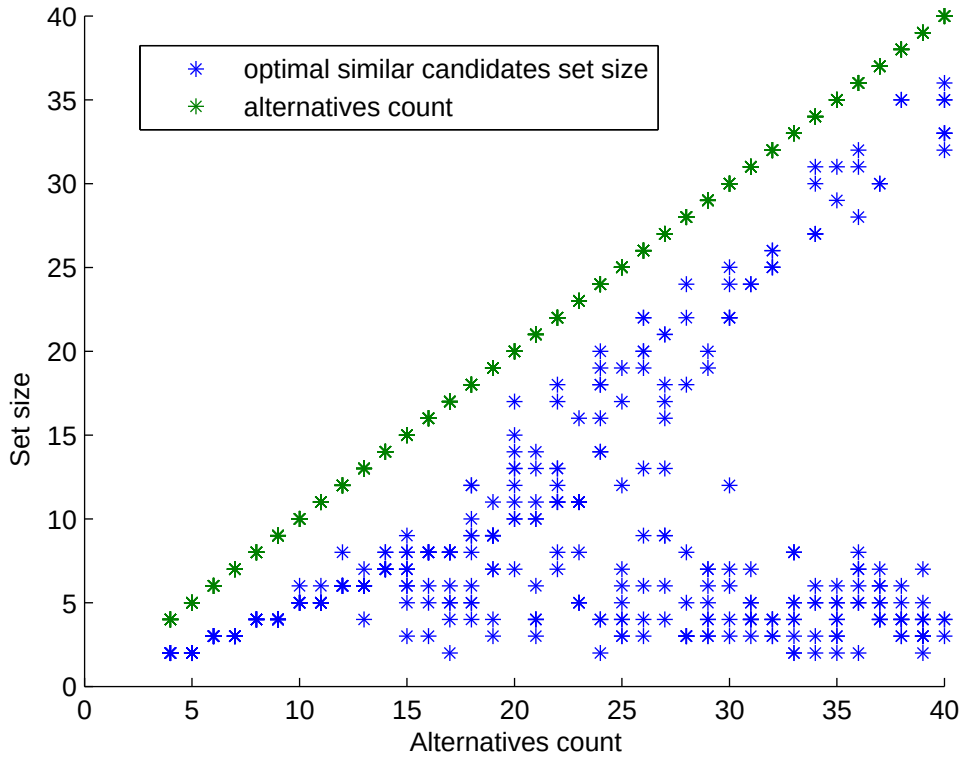

Figure 5.12: optimal set of similar candidates size, 10 cases per alternative count category, consensus probability = 0.1

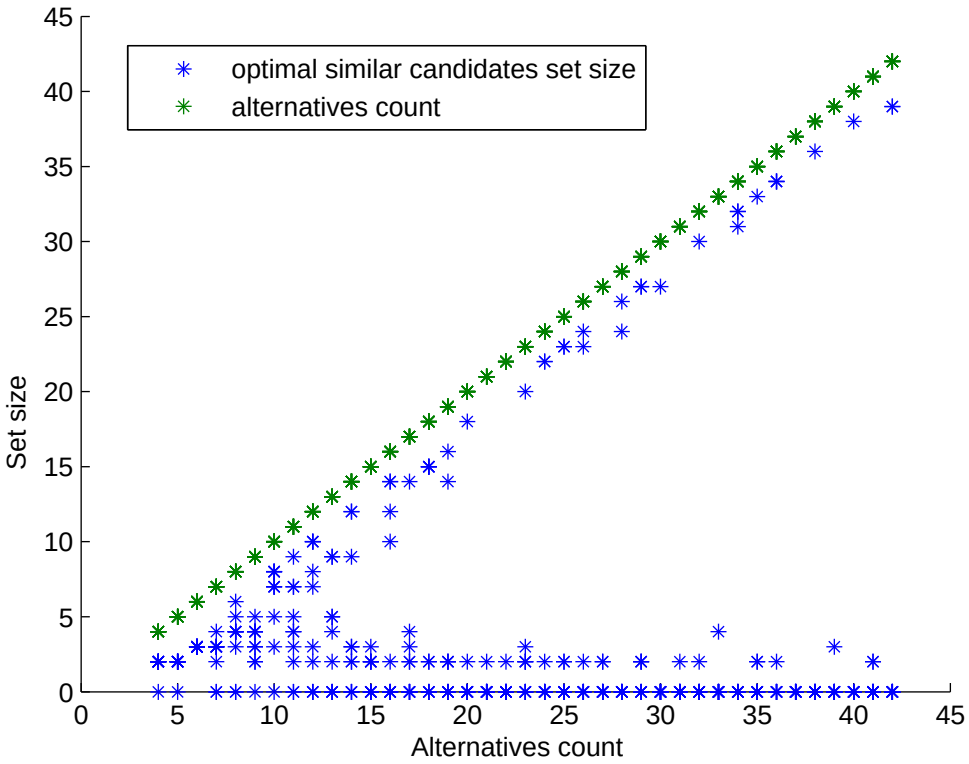

Figure 5.13: optimal set of similar candidates size, 10 cases per alternative count category, consensus probability = 0.5

5.2.6 Local search

This method is the undisputed winner of the discussed heuristics. We will first look at the Slater paradigm. We used the Copeland method first and then used local search. We show the computation times of the linear program for Slater, only Copeland and Copeland plus local search in Figure 5.14. We can see that even though local search seems like a brute force algorithm, the computation time does barely increase with higher alternatives counts even with a consensus probability of 0.5 as seen in Figure 5.15. Next we look at the Slater scores of the methods. We see in Figure 5.16 that Copeland plus local search is barely worse than the optimal linear program and a huge step up from the simple Copeland method.
Next we look at the Kemeny paradigm. This time we used the weighted Coplenad method to produce the input for the local search. Just as with Slater we can see that the local search barely uses any additional computation time. When comparing the Kemeny score we see the same result. The scores of the weighted

Copeland plus local search results are nearly optimal.
We also compared the FLAP score of the results of weighted Copeland plus local search to our results of the linear program for FLAP. The resulting Figure 5.19 should be taken with a grain of salt as the interesting alternatives counts start when the linear program breaks down. Nevertheless it could be used as an upper bound for future work.

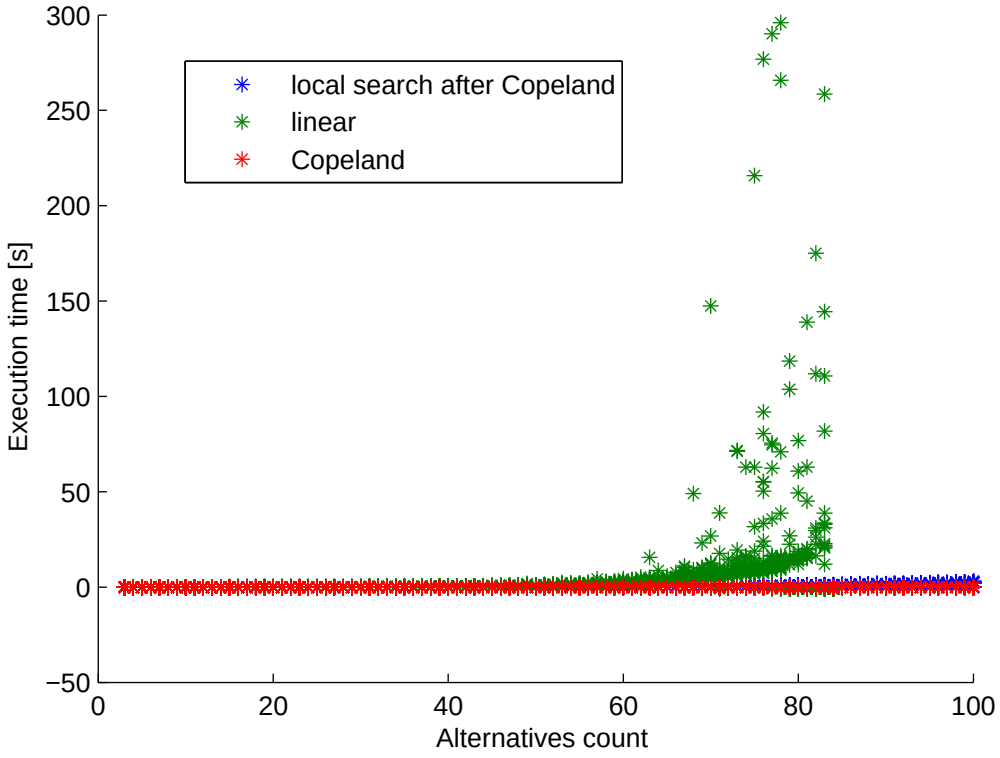

Figure 5.14: Computation time of local search after Copeland, linear program for Slater and only Copeland, consensus probability = 0.1

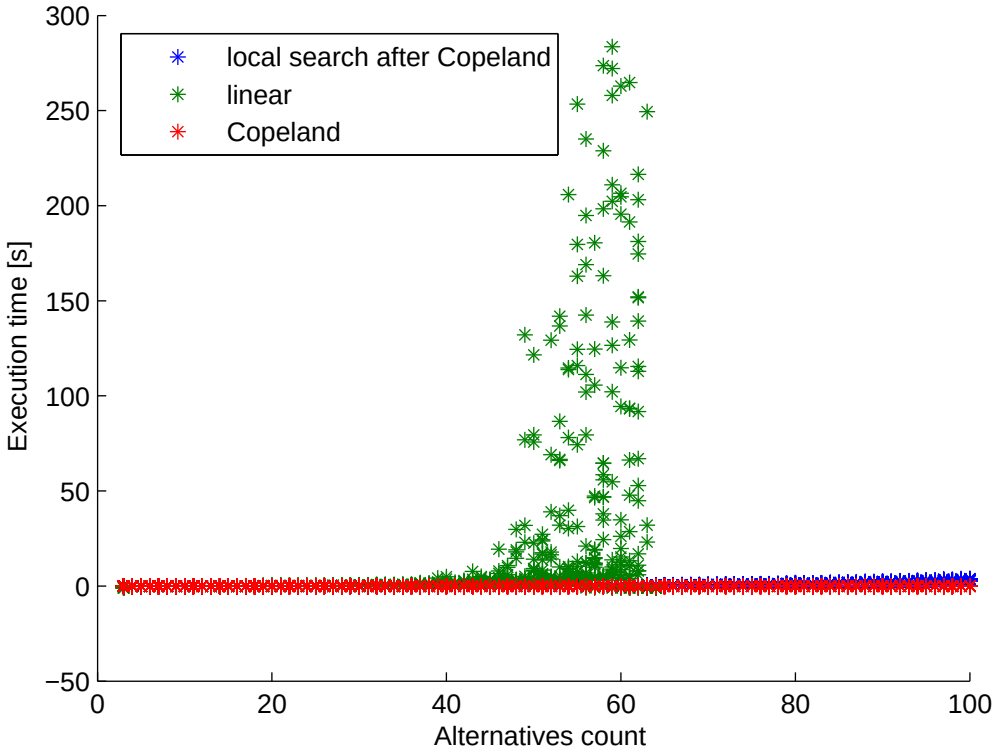

Figure 5.15: Computation time of local search after Copeland, linear program for Slater and only Copeland, consensus probability = 0.5

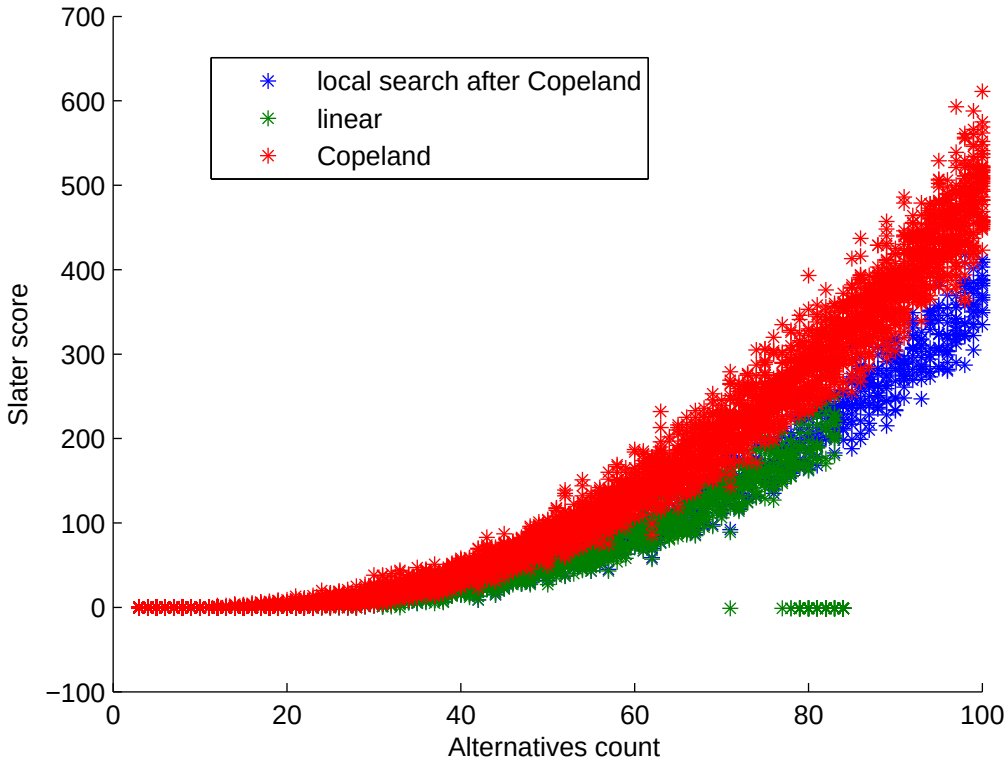

Figure 5.16: Slater score of local search after Copeland, linear program for Slater and only Copeland, consensus probability = 0.1

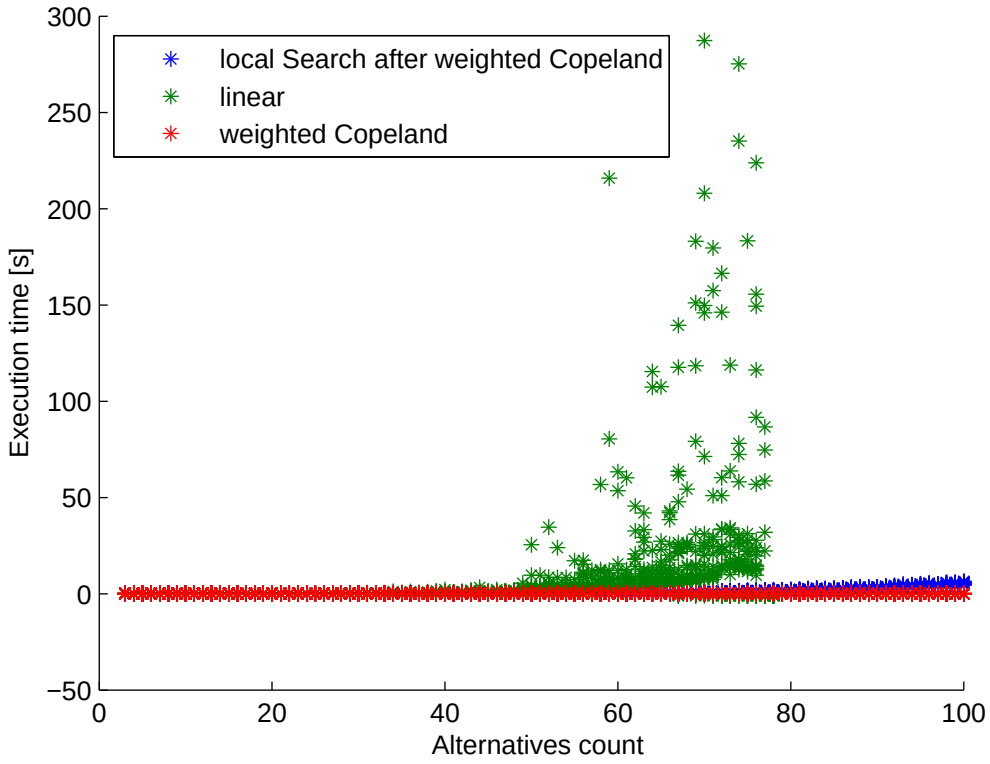

Figure 5.17: Execution time of local search after weighted Copeland, linear program for Kemeny and weighted Copeland, consensus probability = 0.5

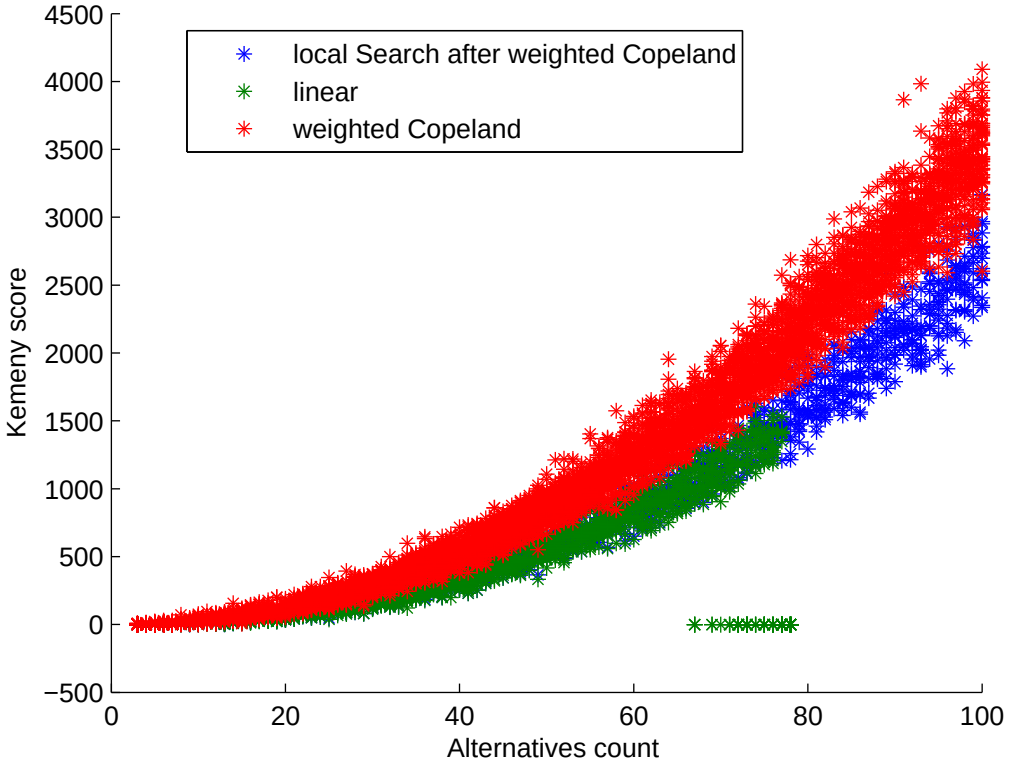

Figure 5.18: Kemeny score of local search after weighted Copeland, linear program for Kemeny and weighted Copeland, consensus probability = 0.5

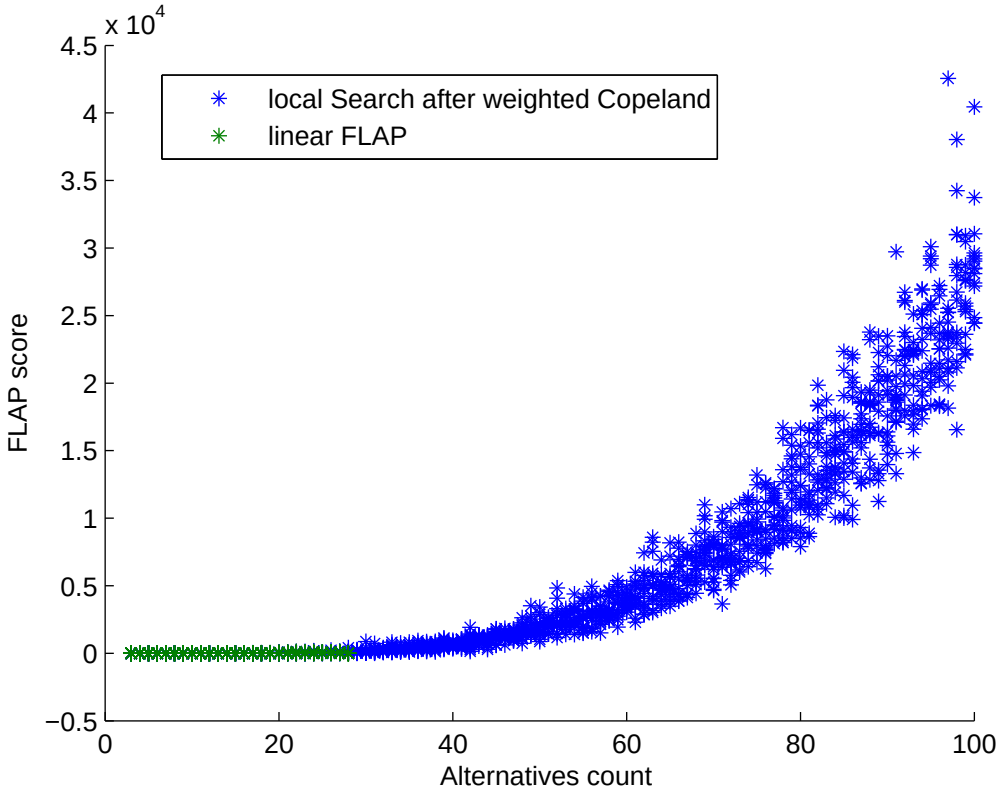

Figure 5.19: FLAP score of local search after weighted Copeland and linear program for FLAP, consensus probability = 0.1

5.3 Other Methods

5.3.1 Probability Estimation

We tested both methods by first generating datasets to cover the range of consensus probability from 0 to 0.5. We can then run the datasets through our estimation method and plot the results against the actual value. Fist we look at Algorithm 8. In Figure 5.20 we can see that our score is indicative of the consensus probability but for consensus probabilities 0.3 to 0.5 it would be hard to infer the consensus probability. We can compare this to Algorithm 9 in Figure 5.21. We can see that Algorithm 8 is the better method since the score is more indicative of the underlying consensus probability for higher values.
We then field-tested Algorithm 8 by giving it real world datasets from PrefLib and then check the results by running the dataset through the linear program for Kemeny. Unfortunately all of the 10 datasets we tested were estimated to have a consensus probability of 0 to 0.1. Nevertheless the resulting computation

time was also indicating a consensus probability between 0 and 0.1. This shows that our method of estimation the consensus probability works with small consensus probabilities. But it also shows that real world datasets typically have very uniform votes.

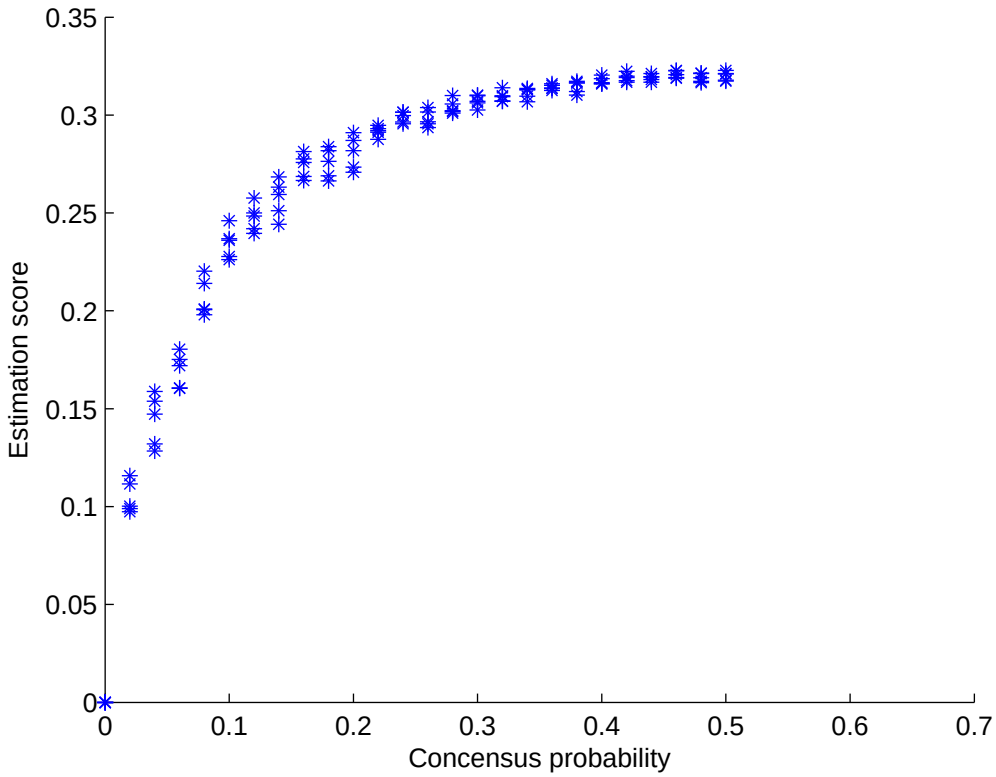

Figure 5.20: probability estimation score with alternative distance

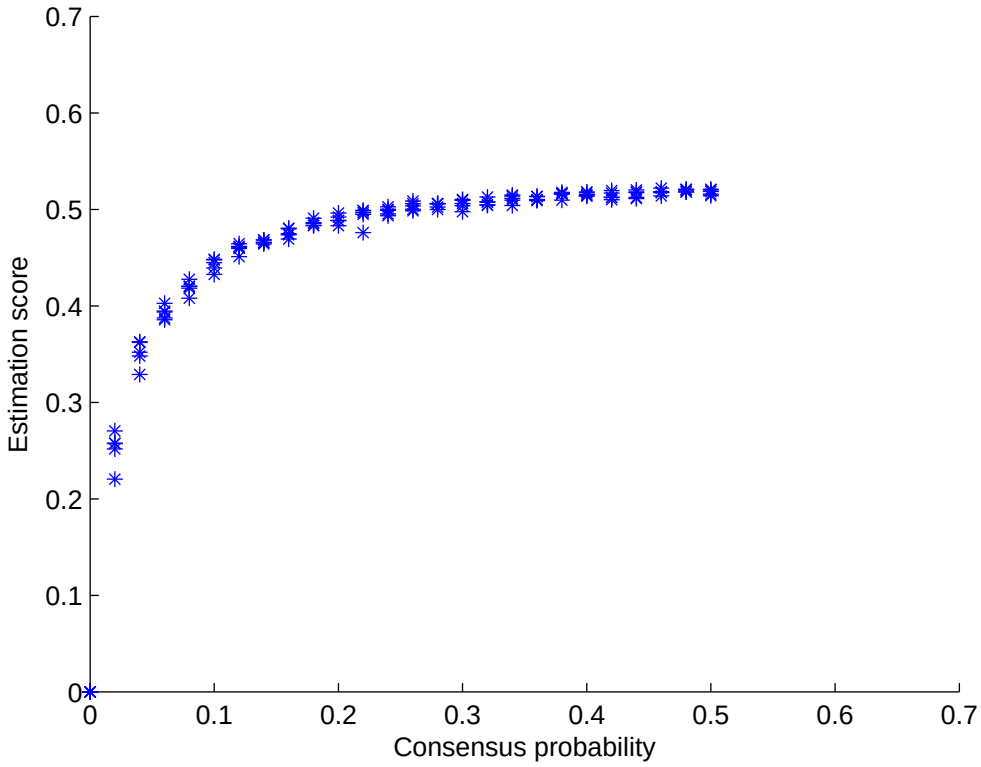

Figure 5.21: probability estimation score with alternative alternative pair disagreement

5.4 Yelp analysis

We ran the 'better-than' analysis over the 1'125'457 reviews which all include a review comment. We searched for business references in suffixes with length 40. The algorithm only returned 5'284 reviews that have the desired format, which is a very low number if we remind us that there are 30'944 distinct business names. We then manually looked at 50 random reviews to get an estimation of how many of theses reviews are false positives. Only 35 had an actual reference to another business and they all only referenced widely known businesses (e.g. Starbucks). Reasons for false positives include names for businesses which have other meanings (e.g. 'America') or irony. For these reasons the 'better than' analysis is not fit for preference data extraction since the few valid preference relations would be to a small number of widely known 'reference' businesses.
The 'better-than' analysis can be viewed as a lower bound analysis since we now know that this is the least we could get out of the give datasets. We will also provide an upper bound analysis, showing the maximal potential of the datasets

and in turn proving that the Yelp review comments are not fit for preference ranking data overall. Since we have the list of business names we can assess how many review text have a potential reference to other businesses. This is similar to the 'better-than' analysis but does not demand the substring 'better than' and searches the whole review comment for the name of another business. We also count each foreign business name in a review comment individually since one review could reference multiple other businesses. The search returned 38'408 possible preferences, which is already a low number if we consider there are 30'944 business names. We again picked 50 random reviews and found that only 6 are an actual reference. These few references have the same issues as the 'better-than' analysis. This concludes the analysis of the Yelp dataset and we conclude that the review texts cannot be used for preference ranking data extraction. There is still the possibility for extracting data from the star ranking. But the data could not be used in the context of this thesis because of the granularity of a star based system.

Chapter 6

Conclusions

6.1 Summary

We have analysed a wide variety of rank aggregation functions and evaluated their quality and efficiency for the three paradigms Kemeny, Slater and FLAP. We concluded that from the methods covered the local search algorithm with the Copeland heuristic produces the most optimal solutions with a very fast running time. We also delved into other topics like real-world data extraction and analysis. One could say that we barely scratched the surface as the topic is very broad with much work still being carried out. We will outline some ideas for future work in the next section.

6.2 Future Work

One topic for further research would be to unify the methodology of how rank aggregation paradigms are defined and then try to analyse characteristics of paradigm families. Currently their creation resembles more of an alchemy than an exact science.
An issue with the FLAP rule is that the linear program defined in this thesis is too complex for a reasonable running time for an optimal solution. Future works could either try to come up with a better linear program or another technique altogether to produce a reasonably fast optimal solver. Furthermore the FLAP problem is strongly believed to be NP-hard but there has been no proof even though it should be easy to show since Kemeny and Slater are known to be NP-hard.
During the work on this thesis there were ideas to generate additional datasets by comparing books by their sales numbers in different countries on online services like Amazon[1] but unfortunately Amazon does not openly publish their sales numbers. Nevertheless there could be great potential in automated data gathering of this sort as we have seen the limitations of currently available datasets.

[1] www.amazon.com

Bibliography

[1] Ali, A., Meil, M.: Experiments with kemeny ranking: What works when? Mathematical Social Sciences **64**(1) (2012) 28–40

[2] Conitzer, V.: Computing slater rankings using similarities among candidates. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. AAAI'06, AAAI Press (2006) 613–619

[3] Davenport, A., Kalagnanam, J.: A computational study of the kemeny rule for preference aggregation. In: Proceedings of the 19th National Conference on Artifical Intelligence. AAAI'04, AAAI Press (2004) 697–702

[4] Mattei, N., Walsh, T.: Preflib: A library of preference data HTTP://PREFLIB.ORG. In: Proceedings of the 3rd International Conference on Algorithmic Decision Theory (ADT 2013). Lecture Notes in Artificial Intelligence, Springer (2013)

[5] Conitzer, V., Davenport, A., Kalagnanam, J.: Improved bounds for computing kemeny rankings. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. AAAI'06, AAAI Press (2006) 620–626

[6] Copeland, A.H.: A 'reasonable' social welfare function. Notes from a seminar on applications of mathematics to the social sciences (1951)

[7] Meila, M., Phadnis, K., Patterson, A., Bilmes, J.A.: Consensus ranking under the exponential model. CoRR **abs/1206.5265** (2012)