FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Cloud Computing Techniques for Winner Determination in Computational Social Choice

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Mag. Julia Theresa Csar, Bakk.
Matrikelnummer 00601791

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Univ.Prof. Dr. Reinhard Pichler

Diese Dissertation haben begutachtet:

Priv.-Doz. Dr. Markus Endres

Jun.-Prof. Dr. Gábor Erdélyi

Wien, 19. Juni 2018

Julia Theresa Csar

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Cloud Computing Techniques for Winner Determination in Computational Social Choice

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Mag. Julia Theresa Csar, Bakk.
Registration Number 00601791

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr. Reinhard Pichler

The dissertation has been reviewed by:

Priv.-Doz. Dr. Markus Endres	Jun.-Prof. Dr. Gábor Erdélyi

Vienna, 19th June, 2018

Julia Theresa Csar

Erklärung zur Verfassung der Arbeit

Mag. Julia Theresa Csar, Bakk.
1120 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Juni 2018

Julia Theresa Csar

Acknowledgements

First I want to thank my advisor Prof. Reinhard Pichler for his guidance to a completely new reserach area, his patients and his great support.

I would like to thank my co-authors Martin Lackner, Emanuel Sallinger and Vadim Savenkov and my colleagues for all nice discussions, lunch breaks and all the support over the last years while working on my dissertation. Especially I would like to thank Andreas Pfandler and Sebastian Skritek, not only for proof-reading parts of this thesis but also for their great support. As well I would like to thank Wolfgang Fischl who has been my friendly office-roommate for quite some time.

Last but not least I would like to thank my friends and family who never lacked to support me, believed in me at all times and encouraged me at the right moments.

This work was supported by the Austrian Science Fund projects (FWF):P25518-N23 and (FWF):Y698.

Kurzfassung

Diese Dissertation beschäftigt sich mit der Entwicklung von Algorithmen in Cloud-basierten Systemen für Probleme im Bereich des "Computational Social Choice". Computational Social Choice beschäftigt sich unter anderem mit Methoden um die Gewinner einer Wahl zu bestimmen. Eine solche Wahl ist dadurch gekennzeichnet, dass die Stimmen in Form von Präferenzlisten bekannt gegeben, oder durch einen automatisierten Mechanismus generiert werden. Das ist zum Beispiel in Online Diensten, wie Streaming Services oder Shops der Fall, wenn Nutzer aus mehreren Möglichkeiten eine Wahl treffen können. In so einem Szenario wären zum Beispiel die beliebtesten Lieder einer Gruppe von Nutzern von Interesse. Methoden für die Identifikation von Gewinnern wurden bereits einige entwickelt, allerdings sind die dazugehörigen Algorithmen für deutlich kleinere Wahlen ausgelegt. Um die Verarbeitung von so großen Wahlen möglich zu machen wird in dieser Arbeit der Grundstein gelegt um BigData Technologien auf dieses Problem anzuwenden. BigData Technologien setzen voraus, dass eine Parallelisierung zu einem gewissen Grad möglich ist. Komplexitätstheorie eröffnet die Möglichkeit ein Problem auf Parallelisierbarkeit zu analysieren. Deshalb werden in dieser Arbeit bereits bekannte Komplexitätsresultat diskutiert und diese um neue Resultate ergänzt. Weiters werden zu einigen Methoden aus dem Bereich des Computational Social Choice Algorithmen entwickelt. Diese Algorithmen basieren auf den Prinzipien der Programmierparadigmen Pregel und MapReduce, welche beide für die Verarbeitung von großen Datenmengen in sogenannten Cloudsystemen konzipiert wurden. Die in dieser Arbeit vorgestellten Algorithmen werden hinsichtlich ihrer Performanz theoretisch analysiert. Weiters werden die Algorithmen implementiert und sowohl an künstlichen als auch an realen Datensätzen evaluiert. Die Implementierung der Algorithmen ist open-source verfügbar.

Abstract

This work deals with the development of cloud-computing algorithms for problems of winner determination in computational social choice. In winner determination in computational social choice we are concerned with determining the winner(s) of an election. In the considered elections the votes are given as preference profiles, i.e. a ranking of candidates, by a voter or by an automated process. For example such scenarios include preference data generated by online services, e.g. streaming services of music or movies, or online shops. In such systems the user has to choose among many candidates and this action is interpreted as a vote, i.e. the users express their preferences. Such online services result in very large elections, but the developed algorithms for methods in computational social choice are usually designed for much smaller settings. To make it possible to apply the methods of computational social choice to such large elections it is necessary to adapt new technologies. Such new technologies have been devised in other areas dealing with huge data sets and include parallel computation and cloud computing techniques. For developing algorithms suited for parallel computation in cloud computing environments the programming paradigms MapReduce and Pregel have been proposed. This work aims at developing algorithms for winner determination by using these programming paradigms. In this thesis, first some problems of winner determination in computational social choice are analysed regarding to their parallelizability. Known complexity results are summarized and extended by new results. Based on this investigation new cloud computing algorithms for several methods in computational social choice are proposed. The algorithms are analysed with regard to common performance measures and are further evaluated by an experimental study. The resulting source code is available as open-source.

Contents

Part I

Introduction and Preliminaries

CHAPTER 1

Introduction

1.1 Motivation

When using online services we are constantly engaging in actions that create lots of user specific data. The generated data and its usage is a major topic in today's discussions. Online stores or music and movie streaming services are used in every day life frequently. The behaviour of users in such systems is monitored and analysed with regard to many aspects. Naturally, one might be interested in the most preferred item, e.g. the most liked movie by a certain group of users. This situation can be modelled as an election, where the choice of watching a movie or listening to a song is interpreted as a vote of the user and the available items are the candidates in the election. A vote in such an election might be directly derived from the selection made by a user or from aggregated data of a group of users. Such aggregated preferences are for example derived from a ranking of the most viewed or most frequently purchased items during a given time period.

Finding the winner or winning set of candidates in an election is not as trivial as it might look at first sight. In fact, the development of methods for the selection of the winner(s) is a very old research topic. Fair winner determination is not only of interest in political elections, but also in other situations of joint decision making. When voters provide a ranking of several candidates, it is not always clear, which candidate should be selected. For example in a very small election of three candidates (A, B and C) and three voters, the following votes are obtained: $A > B > C$, $A > B > C$, $B > C > A$. (The vote $A > B > C$ was cast twice.) One might argue that candidate A is the winner of this election, since it has been ranked first in the most votes, although it has been ranked last by the third voter. An the other hand candidate B has never been ranked last, therefore it might be the better choice. A different approach would be to select both candidates A and B as a set of tied winners. As you can see, even in such a small election it is not clear what the best way of decision making is.

Methods for selecting a winner from candidates in elections can become very complex and there is a vivid research area – social choice – dealing with the problem of proposing voting rules, with different axiomatic properties and related topics. Such methods do not only get complex in the sense of the fairness of the selection, but also when it comes to computability. The field of computational social choice deals with the problem of determining the computational complexity of methods in social choice and with the development of algorithms for those. Many algorithms have been developed for selecting the winner or the winning set of candidates in an election. The methods of computational social choice are usually applied to elections of a relatively small size, e.g. political elections, where the number of candidates is in the range of hundreds and voters might be up to millions. But in the scenarios where we automatically generate lists of preferences, i.e. in online services, the number of alternatives or candidates easily goes up to several thousands and the number of voters seems to be rather unlimited. The computational complexity of these problems becomes especially important when we are dealing with such large elections. Not all of these methods are feasible in reasonable time when working with such big elections and it becomes necessary to make use of new techniques for algorithm design and computation. Such new techniques for dealing with big data sets are devised in other data centered research areas and include parallel computation or cloud computing techniques.

Cloud computing technologies are based on the idea of connecting many independent computing resources and combine their computing power in a high performance cluster. These systems are also referred to as shared nothing architectures, since the computing nodes do not share any resources, but are connected over a fast network. Cloud computing clusters offer the main advantage of scaling out instead of the scaling up. This means, that instead of adding additional resources to one computing entity (node), additional nodes can be added to the system. The new computing nodes are not required to have the same hardware configuration as existing ones. This concept adds much more flexibility to such high performance computing solutions, because many computing entities with differing characteristics can work together and the system can be extended or scaled down as needed.

To leverage the computing power in a cloud computing environment it is necessary to design algorithms in a specific way. For this purpose several programming frameworks have been proposed. The two most prominent of those are MapReduce and Pregel, which are both based on the Bulk Synchronous Processing model proposed by [Val90]. MapReduce has first been proposed in [DG08] and is based on the general idea of splitting data by mapping it to key-value pairs and perform computations on independent subsets of the data. The key of such a key-value pair determines which computing entity, i.e. reduce task, is receiving this data value. This splitting enables the system to distribute the work load among the computing entities efficiently. The reduce tasks work independently of each other and perform a computation on all received values and write the result back to the (distributed) file system. In some cases several such MapReduce rounds might be needed. This method of algorithm design is especially suited when it comes to

batch processing, but might not be the best choice when the data shows a more complex structure, or if the computation needs more iterations. For example, in many big data tasks we are dealing with graph-based data and for this purpose a new programming paradigm – Pregel – was proposed. Pregel is used for graph based computation in such cloud computing environments and has first been proposed in [MAB+10]. A Pregel algorithm is often described as a vertex-centric computation, since each vertex forms its own independent computing entity which can be distributed on the cluster. The vertices communicate with each other in this system and perform their computations independently. The communication and computation is performed synchronously.

The first open-source software solution implementing MapReduce was Apache Hadoop. Hadoop is an open source project and has been readily extended in the last years. Now it contains much more than only MapReduce, but consists of a whole ecosystem of services including a distributed file system and a resource manager. Cloud computing frameworks on top of Hadoop are developing very quickly and one of the most recent advancements was the development of Spark. The software Spark provides many more methods for data heavy computations in cloud computing systems. It does not only come with a machine learning library but also provides the possibility of graph based computations including Pregel procedures by the package GraphX. Although Spark and Hadoop contain a lot of optimization techniques and develop at a high rate, the basic principle of MapReduce or Pregel form the heart of each program running in the system and therefore efficient algorithms use the concepts of those paradigms.

An important aspect of algorithm design is the theoretical performance of the algorithm. For the performance analysis of algorithms designed using MapReduce or Pregel several important computation cost factors have to be considered. Those are especially communication cost and memory usage, besides the actual computation time. While the computation time heavily depends on the underlying cloud infrastructure, the programming paradigms provide us with the possibility of formulating algorithms independently of the underlying concrete cloud system and perform further theoretical analyses of other factors of performance, such as communication cost and data replication rate.

Applying the methods of social choice to large scale elections (as created by streaming services or online stores) makes it necessary to develop algorithms for those methods, which are suited for the computation in a cloud based infrastructure. Extending the methods of computational social choice for the use of cloud computing techniques lets us face many new challenges. First of all not all methods of computational social choice are equally well suited for parallelization and for the computation in cloud computing environments. To determine whether a problem can be parallelized the methods of computational complexity can be used. In computational complexity one seeks to categorize problems into complexity classes by their computational properties. Depending on the complexity class of a problem, we know whether an efficient parallel algorithm might exist and for the problems known to be parallelizable, efficient cloud computing algorithms can be designed.

1.2 Problem Statement

Large scale election data is created in different types of online services. Data of that size is often managed in cloud computing environments, but the current methods of computational social choice do not take advantage of the cloud computing system. It is necessary to develop new algorithms and techniques to deal with large data sets and therefore "moving the methods of computational social choice to the cloud".

The aim of this thesis is to develop cloud computing algorithms for winner determination in large scale elections. For this purpose existing methods are analysed regarding their parallelizability. Further, for the parallelizable methods cloud computing algorithms are developed and tested for their applicability.

In a nutshell, the goal of this work is to bridge the gap between the current state of big data technologies and the methods of computational social choice.

1.3 Related Work

The research areas most closely related to this work are computational social choice and algorithm design for cloud computing frameworks.

Computational social choice

The development of algorithms and methods for preference aggregation and winner determination is a major topic in computational social choice. Preference aggregation aims at combining a set of rankings to one universal ranking, wheras winner determination aims at selecting a winner or several winners from the set of candidates in the election. For many of the voting rules and scenarios important in the area, efficient algorithms have been devised [DKNS01, San02, BGN10, BBF10, LPR+12, CKKP14].

Many recent papers in the direction of algorithm development and complexity analysis focus on NP-hard voting rules, i.e. voting rules that cannot be solved in polynomial time unless P = NP. One example of such a rule is the Kemeny rule, where a lot of work has been done on studying practical algorithms [CDK06, BBN14, SvZ09, AM12]. The work [DKNS01] focuses on developing algorithms for rank aggregation for ranking data on the web. Their focus also lies on the Kemeny rule and their research focusses on finding heuristic algorithms which provide good rankings, with respect to specific criteria.

Other NP-hard voting rules are the STV rule and the ranked-pairs rule, under the assumption that no fixed-tie breaking rule is used [CRX09, BF12]. Recent work by [JSW+17] has considered the NP-hard variants of STV and ranked pairs and established fast algorithms for these problems. In this work the STV rule and the ranked-pairs method are discussed, under the assumption that there is a fixed tie-breaking order. The Schulze method [Sch11] is a relatively young method of rank aggregation and widely used in group decision of political groups and open-source projects. Similarities and differences of the ranked pairs and Schulze method in the context of strategic voting have been considered in [PX12].

In [MFG12] an empirical study on large election data was performed and they highlight that there is a lack of support for existing statistical models which are used to generate synthetic election data. In this work this problem is encountered during the experimental evaluation.

Algorithm Design for Cloud Computing Frameworks

The theoretical frameworks for algorithm design used today are based on the work of [Val90], where the bulk synchronous programming model was proposed. Later this model has been adapted to fulfill the current needs and MapReduce was proposed by Google [DG08]. Pregel [MAB+10] is a further adaptation for graph based algorithms. For the design of algorithms, efficiency is of course a crucial topic. Therefore several concepts of theoretical performance analysis have been proposed.

For designing MapReduce algorithms some approaches for minimizing the replication of data in join computation have been used in [AU10, ASSU13]. Replication of data occurs when the data values are needed at several reduce tasks during the computation. The proposed algorithms in [AU10] are optimal in the sense that they minimize the replication rate and also the lower and upper bounds of communication cost for the proposed algorithms for join computations are derived. The structure of the underlying data also plays a huge role in algorithm performance and it might be useful to consider data characteristics in algorithm optimization. This can be observed by comparing the solutions for optimizations of MapReduce algorithms for join computations presented in [AU10, BKS13] and [BKS14]. Whereas [AU10, BKS13] rely on the assumption of a uniform distribution of data values, for skewed data a different algorithm is proposed in [BKS14]. The methods used for analysing the performance of MapReduce algorithms in those works are relevant to this thesis.

In [YCX+14] many Pregel based graph algorithms have been proposed and the concept of *Balanced Practical Pregel Algorithms (BPPA)* has been introduced. BPPA algorithms have linear space usage, linear communication cost, linear computation cost and use at most a logarithmic number of rounds (for more details see Section 2.3.2. Many efficient Pregel Algorithms, analyzed by the same performance measures, are presented in [SW14]. [YCX+14] and [SW14] also aim at developing efficient Pregel algorithms and they introduce many methods and techniques that can be applied for algorithm development. They propose algorithms for many common graph related problems such as identifying strongly connected components. Their notion on efficient Pregel algorithms is a good guideline for Pregel algorithm design and where an inspiration for the algorithm presented in this thesis.

1.4 Methodological Approach

To reach the aim of this work several methods are used. For the theoretical analysis of problems we use the methods of computational complexity theory. Further, the

development of new algorithms is done by making use of the cloud computing paradigms MapReduce and Pregel. The resulting cloud computing algorithms are analysed regarding several performance measures defined by current research in algorithm development for cloud computing environments. The practicality of the algorithms is shown by experimental evaluation.

This defines the following methodological work-flow, throughout this thesis:

1. Analyse winner determination problems by using methods of complexity theory. Existing complexity results are used as a starting point for this analysis, which are then further extended by new results. The focus of our theoretical complexity analysis lies on the parallelizability of the problems.

2. Develop new algorithms suited for a cloud-computing environment and analyse the theoretical performance of our algorithms.
   For the development of cloud computing algorithms the programming paradigms MapReduce and Pregel are used. Despite the differences in the way they abstract from the underlying cloud system, both paradigms aim at the same architecture and thus share the same principal ideas (distribute the data onto the independent computing entities in the system and perform computations separately). As a result, similar performance measures apply to both of them. In general, the following performance measures are of interest:

   - *number of rounds or supersteps* – the number of rounds the algorithm needs to iterate before termination. Ideally algorithms have a constant or at most poly-log number of rounds.
   - *replication rate* – the factor by which the number of the inputs received by the parallel processes exceeds the size of the original input.
   - *computation time* – the total time needed for the whole computation. In a parallel computation, this corresponds to the time needed by the longest computation path.
   - *memory consumption* – for a more fine-grained differentiation the RAM consumption and the data size stored in the file system in between rounds of the overall computation can be distinguished.

3. Implementation of the algorithms using the cloud computing frameworks Hadoop [Bor07] and Spark [ZCF+10].

4. Experimental Evaluation of the developed algorithms on different types of input data. This includes real world data and synthetic data to show the practicality of the algorithms.

1.5 Results

Some of the results presented in this work have already been published in peer-reviewed international conferences or workshops. In the lists of results it is noted if the result is already contained in a publication and the publications are listed in Section 1.7.

Several complexity results for winner determination rules in computational social choice are the starting point. For example in [BFH09] the complexity of computing the Smith Set, Schwartz Set and Copeland Set was analyzed. Those existing results are extended by new complexity results. In particular, in this work the following new results, are shown:

- **Theorem 1**: The Schulze Winner Determination problem is NL-complete – published in [CLP18].
- **Theorem 2**: The Ranked Pairs Winner Determination problem is P-complete – published in [CLP18].
- **Theorem 3**: The STV Winner Determination problem is P-complete – published in [CLPS17b].
- **Theorem 4**: The STV Winner Determination problem can be solved in $\mathcal{O}(m + \log(n))$ space – published in [CLPS17b].

While we show the NL-completeness of the Schulze winner determination problem, which means that it is suited for parallelization we show for the Ranked-Pairs winner determination that it is P-complete and therefore inherently sequential. For the STV winner determination method the P-hardness of this problem is shown and the *paraL*-membership for the parameterization by the number of candidates in an election (but with unrestricted number of voters) is proved. Under this parameterization a MapReduce algorithm for the STV winner determination method is developed in the next step.

The results of the complexity analysis are further used to develop cloud computing algorithms using the concepts of MapReduce and Pregel. In particular, the following algorithms are proposed:

- MapReduce algorithm for computing scoring rules based on the preference profile – published in [CLPS16].
- MapReduce algorithm for computing the weighted majority graph from the preference profile – published in [CLPS17b].
- MapReduce algorithm for computing the Copeland set based on the graph representation of the election.
- Mapreduce algorithm for computing the Smith set – published in [CLPS17b, CLPS17a].

- Pregel algorithm for computing the Smith set.
- MapReduce algorithm for computing the Schwartz set – published in [CLPS17b].
- Pregel algorithm for computing the Schwartz set.
- MapReduce algorithm for computing the Schulze winner.
- Pregel algorithm for computing the Schulze winner – published in [CLP18].
- Ideas for algorithms for computing the Ranked-pairs winner.
- Mapreduce algorithm for computing the STV winner – published in [CLPS17b].

Further, all those algorithms are analysed with regard to theoretical performance measures of cloud computing algorithms and are implemented. The resulting code is available as open source on Github:

- `https://github.com/theresacsar/BigVoting`
  MapReduce algorithms implemented in Java.
- `https://github.com/theresacsar/CloudVoting`
  Spark library implemented in Scala, containing methods for dealing with preference data and Pregel algorithms.

The experimental evaluation is performed using different types of input data. We use both, synthetic data and real world data to show the scalability and the practicability of the algorithms. The real world data used for this work is based on data provided by the music streaming service Spotify. The top most viewed songs per day and country over the year 2017 where taken as input. This results in a real world winner determination problem with more than 10,000 candidates (songs). The experimental evaluation shows that the proposed algorithms scale very well in cloud computing environments and can deal with large data sets. The experimental evaluation was partially published in [CLPS17b] and [CLP18].

1.6 Structure of this Thesis

First of all the needed definitions and results from the areas of social choice and cloud computing technologies are summarized in Chapter 2.

The theoretical results of this thesis are all contained in Part II in two separate chapters for the complexity results and the designed algorithms. The results of the *complexity analysis of selected voting rules* are presented in Chapter 3. In Chapter 4 the *Cloud Computing Algorithms* are presented and explained in detail. All algorithms are presented not only as pseudo-code but also illustrated by example.

In Part III the *performance of the proposed algorithms* is investigated in more detail. In Chapter 5 the *theoretical performance measures* of the algorithms are summarized, compared and discussed. The results of the *experimental evaluation* are shown in Chapter 6, where it is observed that the algorithms are scaling well in practice.

Finally there is Part IV with a summary of the results, concluding remarks and directions for future work.

1.7 Publications

Some of the results presented in this thesis have already been published in the following peer-reviewed publications:

[CLP18] Theresa Csar, Martin Lackner, and Reinhard Pichler. Computing the Schulze method for large-scale preference data sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sweden, 2018*. To appear.

[CLPS17b] Theresa Csar, Martin Lackner, Reinhard Pichler, and Emanuel Sallinger. Winner determination in huge elections with MapReduce. In *Proceedings of AAAI-17,2017*.

[Csa18] Theresa Csar. CloudVoting: Analyzing Preferences using Spark and GraphX. In *Online proceedings of MPREF-18 Workshop, 2018*.

[CLPS17a] Theresa Csar, Martin Lackner, Reinhard Pichler and Emanuel Sallinger. Computational Social Choice in the Cloud. In *Proceedings of PPI17 Workshop at BTW 2017, Datenbanksysteme für Business, Technologie und Web*, 2017.

[CLPS16] Theresa Csar, Martin Lackner, Reinhard Pichler and Emanuel Sallinger. Winner determination in huge elections with MapReduce. In *Proceedings of the 10th Multidisciplinary Workshop on Advances in Preference Handling*, 2016.

[CPSS15] Theresa Csar, Reinhard Pichler, Emanuel Sallinger, and Vadim Savenkov. Using statistics for computing joins with map reduce. In *Alberto Mendelzon Workshop 2015*.

CHAPTER 2

Preliminaries

In this chapter an introduction to the used methods from computational social choice and cloud computing techniques used in this thesis is given. First the basics of computational social choice including methods for winner determination are explained and defined. Further an introduction to the used complexity classes is given. The cloud computing programming paradigms MapReduce and Pregel are explained and the used cloud computing frameworks (Hadoop, Spark and GraphX) are introduced.

2.1 Computational Social Choice and Winner Determination

Computational social choice is a huge research area dealing with joint decision making (for an introduction to these topics see [CELM07]). In this thesis, we are focussing on the problem of *winner determination* in situations where several voters provide a ranking of candidates. This can be approached from two directions – on the one hand there is *voting theory* which is concerned with reaching a common decision and on the other hand *preference aggregation*, where all provided preference rankings are aggregated.

The area of voting theory is concerned with the development of methods to define the winning candidate or a winning set of those candidates. There are many methods for such joint decision making of groups of entities, i.e. groups of people, or some automated processes (for an overview of methods see [BBH16]).

First the input data to problems of computational social choice is discussed in Section 2.1.1, followed by an introduction to the used methods for winner determination in this thesis in Sections 2.1.2- 2.1.6. Further the computational complexity aspects of the winner determination problems are discussed (Section 2.2).

2.1.1 Election Data

Before going into detail on the used methods of winner determination, we define the problem instances and give an introduction to the used terminology. Given a set of alternatives (or candidates) A of size m and a set of voters $N = \{1, \dots, n\}$, each voter provides his or her preference over the candidates in A as a preference relation or vote. This preference relation of a voter i is referred to by $\succeq_i$. Whereas $\succeq_i$ contains a weak order on A and the preference relation $\succ_i$ denotes the strict part of $\succeq_i$. We assume that votes are strict partial orders, i.e., reflexive, antisymmetric and transitive binary relations. The full preference profile P is the collection of all votes $P = (\succeq_1, \dots, \succeq_n)$. An alternative $a \in A$ strictly dominates an alternative $b \in A$ if there are more votes ranking a before b in the preference profile P. This means, a strictly dominates b if $|\succ_i \in P : a \succ_i b| > |\succ_i \in P : b \succ_i a|$. Weak domination of candidates is defined analogously, by replacing $>$ by $\geq$. From the preference profile the dominance relation $D(P) \subseteq A \times A$ can be created by using the definition of weak or strict dominance. The *strict or weak dominance graph* is a directed graph denoted as $D_\succ$ or $D_\succeq$ where the vertices of the graph are the candidates A of the election and there is an edge from candidate a to candidate b if a strictly (or weakly) dominates b, i.e. $D_\succ = (A, E_P)$ with $E_P = \{(a, b) \in A^2 \colon \text{if } a \text{ strictly dominates } b\}$. We will simply use D to denote the weak or strict dominance relation, if the variant is clear from the context or not important to the specific case. From the definition of $D_\succ$ follows that the strict dominance relation is asymmetric, since it is not possible that a strictly dominates b and b strictly dominates a at the same time. Further $D_\succ$ is irreflexive, because no candidate can dominate itself [BFH09].

In some cases we are not only interested in the position of a candidate in the votes but also in the number of voters ranking candidate a before candidate b. Therefore, we define the *majority margin* of two candidates $a, b \in A$ as the number of votes preferring a over b minus the number of votes preferring b over a. The *majority margin* over a preference profile P is defined as $\mu_P(a, b) = |i \in N : a \succ_i b| - |i \in N : b \succ_i a|$.

The weighted directed graph with the candidates as vertices and weighted (directed) edges between all candidates with a majority margin greater than 0 is called the *weighted majority graph* of P. This means, if candidate a dominates b then there is an edge from a to b with weight $\mu_P(a, b)$ in the weighted majority graph $\mathcal{W}_P$. Therefore, $\mathcal{W}_P$ has the same directed edges as the corresponding strict dominance graph, but with the respective majority margin as weight. The weighted majority graph of P is denoted as $\mathcal{W}_P = (A, E_P, \mu'_P)$ with $E_P = \{(a, b) \in A^2 \colon \mu_P(a, b) > 0\}$. The restriction of μ_P to E_P is denoted as μ'_P, i.e. only keeping weights larger than 0.

Note, that there are two relevant dimensions that influence the size of the problem instance: the number of candidates m and the number of votes n. In Table 2.1 the used notations are listed.

A	set of candidates (alternatives)
$N = \{1, \ldots, n\}$	set of voters
$\succeq_i$	the preference (vote) of voter i
$\succ_i$	the strict part of $\succeq_i$
$P = \{\succeq_1, \ldots, \succeq_n\}$	the set of votes; the full preference profile
a strictly dominates b	if $\mid \succ_i \in P : a \succ_i b \mid > \mid \succ_i \in P : b \succ_i a \mid$
$D_\succ(P)$	the strict dominance relation
$D_\succeq(P)$	the weak dominance relation
$\mu_P(a,b) = \mid i \in N : a \succ_i b \mid - \mid i \in N : b \succ_i a \mid$	majority margin of a and b
$E_P = \{(a,b) \in A^2 \colon \mu_P(a,b) > 0\}$	the edges in the weighted majority graph
μ'_P	restriction of μ_P to E_P
$\mathcal{W}_P = (A, E_P, \mu'_P)$	weighted majority graph

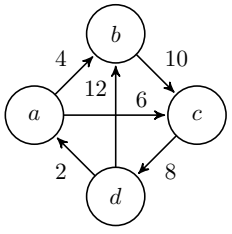
Table 2.1: Notation

Example 1 *Consider the election with four candidates $A = \{a, b, c, d\}$ and a preference profile containing 30 votes given in Figure 2.1a (originally by Schulze [Sch03]). The first column in the table gives the number of voters casting the preference relation in the second column as vote.*

There are 17 *votes ranking a before b and* 13 *votes ranking b before a in the preference profile P. Therefore a dominates b and the corresponding weighted edge in the weighted majority graph has weight* 4*, i.e. $\mu_P(a,b) = 4$. The other majority margins are computed analogously and the resulting weighted majority graph can be found in Figure 2.1b. In this example there are no ties and therefore the strict and the weak dominance graph coincide. The strict and weak dominance graphs have the same directed edges as the weighted majority graph, but without weights.* ◇

$\#Votes$	Preference Relation
3	$a \succ c \succ d \succ b$
5	$a \succ d \succ b \succ c$
4	$b \succ a \succ c \succ d$
5	$b \succ c \succ d \succ a$
2	$c \succ a \succ d \succ b$
5	$c \succ d \succ a \succ b$
2	$d \succ a \succ b \succ c$
4	$d \succ b \succ a \succ c$

(a) Full Preference Profile P



(b) Weighted Majority Graph $\mathcal{W}_P$

Figure 2.1: The Full Preference Profile and the Weighted Majority Graph in Example 1

2.1.2 Winner Determination and Rank Aggregation

In Winner Determination we are interested in finding a winner or a winning set of candidates of an election. We refer to methods which define how to select a winner as winner determination rules, social choice rules or social choice functions. Probably one of the most natural approaches of winner determination is to consider pairwise comparisons and to declare a candidate to be the winner if it strictly dominates all other candidates. Such a candidate is called a *strong Condorcet winner*, and is unique if it exists. In some cases, e.g. if there is a cycle in the dominance graph, there is no Condorcet winner in the election.

It is possible that there are several *weak* Condorcet winners. A *weak Condorcet winner* is a candidate that dominates or ties all other candidates. The Condorcet method is a choice rule that selects all weak Condorcet winners as the winning set of the election.

Winner determination rules are categorized by the type of input used (Fishburn's classification [Fis77]). The discussed rules are either based on the dominance relation (or dominance graph), weighted majority graph or the preference profile. The social choice functions based on the dominance relation D are referred to as C1-functions. This means that C1-functions only consider the relation if a candidate defeats another candidate in a pairwise comparison, but no further information is used for winner determination. Therefore adding additional votes, that do not alter the dominance graph, does not effect the outcome of the choice rule. C2-functions take the weighted majority margin as input, i.e. C2-functions are based on the weighted majority graph $\mathcal{W}_P$. So called C3-functions use strictly more information than the weighted majority graph. They require the full preference profile, and often yield computationally hard winner determination problems.

The corresponding Winner Determination problems based on a social choice rule $\mathcal{R}$ are defined as follows:

$\mathcal{R}$ Winner Determination (C1-function)	
Instance:	dominance relation D, candidate c
Question:	Is c a winner in D according to rule $\mathcal{R}$?

$\mathcal{R}$ Winner Determination (C2-function)	
Instance:	weighted majority graph $\mathcal{W}_P = (A, E_P, \mu'_P)$, candidate c
Question:	Is c a winner in $\mathcal{W}_P$ according to rule $\mathcal{R}$?

$\mathcal{R}$ Winner Determination (C3-function)	
Instance:	preference profile P, candidate c
Question:	Is c a winner in the election given by P according to rule $\mathcal{R}$?

Some social choice functions can also be used to create an aggregated ranking. The problem of rank aggregation is defined as follows:

$\mathcal{R}$ Rank Aggregation	
Instance:	a preference profile P.
Question:	Find a linear order of candidates that is optimal with respect to $\mathcal{R}$.

The winner determination rules used in this thesis are discussed in the remaining subsections: Next we discuss positional scoring rules (very simple voting rules, where scores are assigned to the candidates by the position in the votes). Then we proceed with $C1$-functions, which are based on the strict or weak dominance graph as input. In particular we discuss the Smith Set and the Schwartz Set, which both follow a very similar definition, but differ on the type of dominance graph they use - the Smith Set uses the weak dominance graph as input and the Schwartz Set the strict dominance graph. Next the Schulze method and the ranked-pairs method are discussed, which are based on the weighted majority graph $\mathcal{W}_P$ and are therefore C2-functions. The Schulze method is originally a method for rank aggregation. Last but not least we review the C3-function Single Transferable Vote (STV).

2.1.3 Positional Scoring Rules

The most commonly known and probably most intuitive positional scoring rule is the plurality rule. The plurality rule selects the candidate as winner, which is ranked first in most votes. In general a positional scoring rule is taking the preference profile P as input and assigns a score to each candidate dependent on its position in each vote. The final score of a candidate is the sum over its scores. This can be modelled by providing a scoring vector, which gives the scores assigned to each position in the vote. For example for the plurality rule the scoring vector is $s = (1, 0, 0, \ldots, 0)$, i.e. the only candidate receiving a point is the candidate ranked first in each vote. Another frequently used scoring rule is the Borda scoring rule. For a vote of length m the scoring vector is $s = (m-1, m-2, ..., 0)$. This version of the Borda method is also called *asymmetric Borda score*. For other definitions of the Borda scoring rules see [BBH16].

The k-Approval rule assigned as score of 1 to the candidates ranked at the first k positions and 0 otherwise. The plurality rule is a special case of the k-approval rule, where $k = 1$. There are many different types of positional scoring rules, in Table 2.2 some scoring vectors are listed.

Rule	Scoring Vector
Plurality	$(1, 0, 0, \ldots, 0)$
Anti-plurality	$(0, 1, 1, \ldots, 1, 1)$
k-Approval	$(1, 1, 1, \ldots, 1, 0, 0, \ldots, 0)$
Borda	$(m-1, m-2, ..., 0)$

Table 2.2: Scoring Vectors for several Positional Scoring Rules [BBH16].

Candidate	Borda	Plurality	2-Approval
a	**53**	8	**16**
b	46	**9**	13
c	41	7	15
d	48	6	**16**

Table 2.3: Positional Scores in Example 1.

It might come by surprise that even such simple voting rules lead to different results in a simple election. In Example 1 one such election is shown and the results of the applied scoring rules can be found in Table 2.3.

Example 1 continued. *For the preference profile given in Figure 2.1a positional scoring rules are computed. The computation of the Borda Scores works as follows: The first entry of the preference profile states that the preference list* $a \succ c \succ d \succ b$ *was given by three voters. The number of candidates in the preference relation is* 4 *and therefore the scoring vector for the Borda method is* $s = (3, 2, 1, 0)$*. Therefore candidate a receives three times* 3 *points for its position in the first preference relation in Figure 2.1a. When looking at the other preference relations the total Borda score can be calculated and therefore the Borda score of a is* $Borda_P(a) = 3*3+5*3+4*2+5*0+2*2+5*1+2*2+4*2 = 53$*. The calculation of the Borda scores of the other candidates works analogously. The scores by Borda rule and some other scoring rules for the candidates in the preference profile P can be found in Table 2.3. Each scoring rule then selects the candidate with the highest score as winner. It can be observed that already for such a small election with only four candidates the scoring rules are not selecting the same candidate as winners. Whereas Borda would select candidate a, Plurality chooses candidate b and 2-Approval is undecided for candidates a and d. In Table 2.3 the scores of the respective winners are printed bold.* ◇

2.1.4 Winner Determination Methods based on the Dominance Relation (C1-Functions)

There are many choice sets based on structural graph properties of the dominance graph. One example is the Copeland Set, which is based on the number of incoming and outgoing edges of the dominance graph. There are also social choice functions depending on other structural graph properties. We will discuss two closely related sets: the Smith set and the Schwartz set.

So-called $C1$-functions are based solely on the dominance graph and are not effected by any changes in the underlying preference profile, that do not alter the dominance graph. Before discussing such choice rules in more detail we will first introduce the *Condorcet Winner* and the important properties: *Condorcet-consistency* and *Smith-consistency.*

Condorcet Winner and Property Condorcet-consistency

The *Condorcet Winner* is the candidate that defeats all other candidate in the election in a pairwise majority sense, i.e. the Condorcet Winner is dominating all other candidates. In some elections there is no Condorcet winner, but if there exists one it is unique. In some cases the dominance relation might contain a cycle. If this happens, then there is no Condorcet winner. This phenomenon is also referred to as the *Condorcet's voting paradox*.

The property *Condorcet consistency* is used to describe voting rules. A voting rule is Condorcet consistent if it selects the Condorcet winner as the unique winner, if there is one.

Property: Smith-Consistency

The subset $A_1 \subseteq A$ of candidates of the total set of candidates A is said to be Smith-consistent if all candidates in A_1 dominate all candidates in $A \setminus A_1$. The smallest set satisfying this property is called the Smith set and will be discussed later.

Copeland Scores and Copeland Set

Other then the positional scoring rules introduced before, the Copeland scores are based on the structure of the dominance graph. In particular, the Copeland Score of a candidate a is the difference of the number of candidates that are dominated by a and the number of candidates that dominate a. In the dominance graph this can easily be computed by counting the number of incoming and outgoing edges and computing the difference. The candidate with the highest score is then selected as the Copeland Winner. In other words the *Copeland Winner* is the candidate winning most pairwise majority contests [BBH16].

The Copeland Scores [BBH16] are defined as $\text{CopelandS(a)} = |\{b|a \succ b\}| - |\{b|b \succ a\}|$.

The *Copeland Set* is the set of all candidates with the maximum Copeland score[Cop51]. The Copeland set is always non empty and it is straight-forward to see, that the Copeland Set is Condorcet-consistent, but not necessarily Smith-consistent. Since a candidate in the Copeland Set might be dominated by an outside candidate.

Smith Set

The Smith set is the (unique) minimal set of candidates that dominate all outside candidates in the weak dominance graph. We define the Smith Set via dominant sets. A dominant set A_1 is a nonempty set of candidates (A_1 is a subset of A), that dominate all candidates outside of A_1, i.e. each candidate in A_1 dominates all candidates in $A \setminus A_1$ [BBH16]. Obviously, each candidate contained in A_1 has a larger Copeland score than a candidate in $A \setminus A_1$. Therefore, the Smith set is a superset of the Copeland Set.

For finding an algorithm for computing the Smith set a different definition, based on strongly connected components in the weak dominance graph, is useful. The Smith set

can also be defined as the unique undominated strongly connected component in the weak dominance graph [BFH09]. For the computation of the strongly connected components in graphs many algorithms have been developed, e.g. Tarjan's algorithm or a Pregel based algorithm presented in [YCX+14].

Schwartz Set

The Schwartz set is very similar to the Smith set, but it is based on the strict dominance graph, whereas the Smith Set is based on the weak dominance graph. Therefore, the Smith Set and the Schwartz Set differ in their treatment of ties [BFH09]. The Schwartz set is defined as the union of all minimal sets (strongly connected components) that are not dominated by outside candidates in the strict dominance graph [BFH09]. Further [BFH09] show that the Schwartz Set is always a subset of the Smith Set.

Example 1 continued. *The weighted majority graph of the Example can be seen in Figure 2.1b. The candidates a and d both have a Copeland score of* 1 *and form the Copeland Set. There is no Condorcet Winner in this election. The Schwartz and Smith Set are identical, because the strict and the weak dominance graph coincide. Further, both sets contain all candidates, since the dominance graph is one single strongly connected component.* ◇

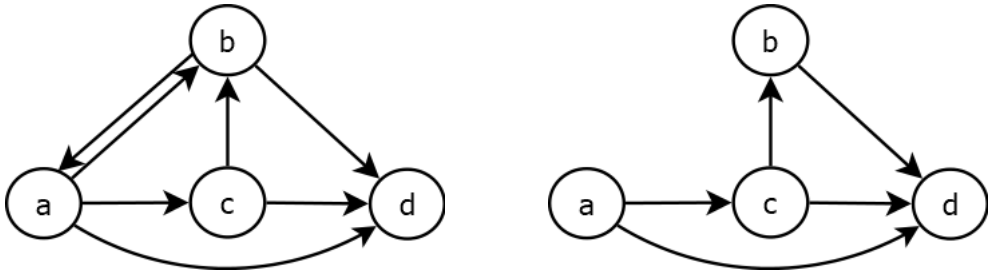
Example 2 *An election with four candidates* $\{a, b, c, d\}$ *and a preference profile* P *with six votes is given. The preference profile* P *is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

This preference profile P *results in the weak dominance graph shown in Figure 2.2a and the strict dominance graph shown in Figure 2.2b. The strict dominance relation is* $D_{\succ} = \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$ *and the weak dominance relation is* $D_{\succeq} = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$.

Condorcet Winner. *There is no strict Condorcet Winner, but candidate a is a weak Condorcet Winner.*

Copeland Scores. *The Copeland Scores can easily be computed from the strict dominance relation* $D_{\succ}$. *The Copeland Scores of the candidates are: CopelandScore(a) = 2, CopelandScore(b) = 0,CopelandScore(c) = 1,CopelandScore(d) = -3. Therefore, candidate a is the Copeland Winner with score* 2 *and a is the only candidate in the Copeland Set.*



(a) Weak Dominance Graph (b) Strict Dominance Graph

Figure 2.2: Dominance Graphs for Example 2

Smith Set. *The set of candidates $\{a, b, c\}$ is the smallest undominated set in the weak dominance graph, i.e. $\{a, b, c\}$ is the Smith Set. The weak dominance graph only contains two strongly connected components. Those are $\{a, b, c\}$ and $\{d\}$.*

Schwartz Set. *The Schwartz Set only contains one candidate, that is a. Candidate a is the only undominated candidate (and the only undominated strongly connected component) in the strict dominance graph. In this example, each candidate in the strict dominance graph forms its own strongly connected component.* ◇

2.1.5 Winner Determination Methods based on the Weighted Majority Graph (C2-Functions)

In this work two C2-functions are discussed: the Schulze method and the ranked-pairs method.

Schulze Method

The Schulze Method [Sch03, Sch11] is originally a method for preference aggregation, this means that the Schulze method does not only return a winner, but returns an aggregated preference profile over the whole set of candidates. The candidate ranked first in the aggregated preference is the Schulze winner; but it is not necessary to compute the full ranking to find the Schulze winner. The *Schulze method* [Sch11] is a refinement of the Schwartz method and always selects a candidate contained in the Schwartz set as winner. It is a so called C2-function and therefore uses the weighted majority graph as input. Its definition depends on *widest paths* in the weighted majority graph. In a weighted majority graph (A, E, μ) a path $(x_1, \ldots, x_k)$ has *width* α if $\min_{i \in \{1,\ldots,k-1\}} \mu(x_i, x_{i+1}) = \alpha$. A *widest path from a to b* is a path from a to b of maximum width. We use $p(a, b)$ to denote the width of such a path. According to the Schulze method, an alternative a beats alternative b if there is a wider path from a to b than from b to a, i.e., if $p(a, b) > p(b, a)$. An alternative a is a Schulze winner if there is no alternative b that beats a. It is guaranteed that such a candidate exists but it is not necessarily unique. The Schulze method can be used to compute an aggregated ranking, then the outcome is defined

	a	b	c	d
a	-	6	6	6
b	2	-	10	8
c	2	8	-	8
d	2	12	10	-

Table 2.4: The widest paths in Example 1

by the relation $(a, b) \in R$ if and only if $p(a, b) \geq p(b, a)$. It can be shown that R is a weak order [Sch11]. The Schulze winners are exactly the top-ranked alternatives in R. Note, that also the scoring rules presented in the beginning of the section can be used to aggregate a ranking over all candidates.

Ranked-Pairs

We define the *ranked pairs method* [Tid87] subject to a fixed tie-breaking order T, which is a linear order of the candidates. The ranked pairs method creates a ranking of alternatives, starting with an empty relation R. All pairs of candidates are sorted according to their majority margin and ties are broken according to T. Then, pairs of candidates are added to the relation R in the sorted order (starting with the largest majority margin). However, a pair is omitted if it would create a cycle in R. The final relation R is a ranking of all alternatives and the top-ranked alternative is the ranked-pairs winner (subject to T).

Example 1 continued. *The preference profile P of the election given in Example 1 results in the weighted majority graph displayed in Figure 2.1b. Table 2.4 shows the widest paths between any two vertices. The unique Schulze winner is candidate a, having a widest path of width 6 to every other candidate, whereas all incoming widest paths to vertex a have width 2. The unique ranked-pairs winner is d (independently of the chosen tie-breaking order). It is obtained by inspecting the edges in descending order of weights and retaining an edge only if it does not cause a cycle. We thus retain $(d, b), (b, c)$, omit (c, d), and retain $(a, c), (a, b), (d, a)$. Hence, d is the only vertex without incoming edge in the resulting directed acyclic graph.* ◇

2.1.6 More Winner Determination Methods (C3-Functions)

Winner determination methods which use more information than the weighted majority graph, are referred to as C3-functions. These voting rules are often observed to be NP-complete. In this thesis we only discuss the C3-function single transferable vote (STV) rule.

Single Transferable Vote (STV)

The winner determination method "single transferable vote" (STV) is based on the full preference profile as input and is defined as follows: In each round the candidate that

3	$a \succ c \succ b$
5	$a \succ b \succ c$
4	$b \succ a \succ c$
5	$b \succ c \succ a$
2	$c \succ a \succ b$
5	$c \succ a \succ b$
2	$a \succ b \succ c$
4	$b \succ a \succ c$

(a) Preference Profile after deleting d

3	$a \succ b$
5	$a \succ b$
4	$b \succ a$
5	$b \succ a$
2	$a \succ b$
5	$a \succ b$
2	$a \succ b$
4	$b \succ a$

(b) Preference Profile after deleting c

Figure 2.3: STV Method by Example

is ranked first in the least number of votes is removed from each vote. Therefore, after $m-1$ rounds only one candidate remains and this candidate is selected as the winner. In case of ties we assume that a tie-breaking order is given.

Example 1 continued. *We calculate the winner by the STV method from the preference profile given in Figure 2.1a. The candidate ranked first in the least votes is identified, which is d (ranked first in 6 votes) and is deleted from the profile. The resulting new preference profile is shown in Figure 2.3a. The next candidate to remove is candidate c, since it only appears in* 7 *votes in the first position. In the remaining preference profile shown in Figure 2.3b, candidate a is ranked first in* 17 *votes and b in* 13 *votes, therefore b is removed and a is selected as the STV winner.* ◇

2.2 Computational Complexity of Social Choice Rules

We use the methods of computational complexity theory to determine whether a parallel algorithm for a winner determination problem might exist. Tractability in the form of P-membership is often considered as the optimal outcome of a complexity analysis. Analogously, in parameterized complexity theory, one aims at showing FPT-membership, i.e., fixed-parameter tractability. However, in order to establish high parallelizability of a problem, we have to look inside P and FPT. A more fine grained view is needed to distinguish between P-hard problems (which are considered as *inherently sequential* [Joh90]) and problems which actually lie in some class inside P. A problem is considered as *parallelizable* if it can be solved on polynomially many processors in poly-log time, i.e., in time $\mathcal{O}(\log(n)^i)$ for some $i \geq 0$. NC is the class of all such parallelizable decision problems.

The formal definition of NC is usually via Boolean circuits. A family $\{B_n \colon n \geq 1\}$ of Boolean circuits is log-space uniform if there is a deterministic Turing machine which, when given n, constructs B_n in space $\mathcal{O}(\log n)$. Then we can define $\mathsf{NC} = \bigcup_{i \geq 0} \mathsf{NC}^i$, where NC^i is the class of problems that can be decided by a log-space uniform class of Boolean circuits having size $\mathcal{O}(n^{\mathcal{O}(1)})$ and depth $\mathcal{O}(\log(n)^i)$. If we allow Boolean circuits

of unbounded fan-in, then we get the classes AC^i. Therefore the class AC^0 is the class of problems solvable by uniform constant-depth Boolean circuits with unbounded fan-in and a polynomial number of gates [BFH09]. The class TC^0 additionally allows for treshold gates. Treshold gates output true if and only if the number of true inputs exceeds a certain treshold.

The complexity classes used in this work are known satisfy in the following inclusions:

$$AC^0 \subset TC^0 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{NC} \subseteq \mathsf{P} \subseteq \mathsf{NP} \tag{2.1}$$

Problems contained in the classes AC^0, TC^0, L, NL and NC are considered as highly parallelizable.

The space complexity class L (log-space) is closely related to these circuit-classes. Problems in L can be solved with logarithmic space and membership in L can be seen as evidence that a problem is parallelizable as it can be computed in $\log^2$ time with a polynomial number of parallel processors.

2.3 Cloud Computing Techniques

To deal with ever growing datasets in various data centric applications many of new methods have been developed. Cloud computing technologies are built upon using a cloud based infrasture, i.e. shared nothing architectures. For implementing software and to formulate algorithms on such cloud-based infrastructure various programming frameworks were proposed. In particular MapReduce [DG08] and Pregel [MAB+10]; both frameworks are special cases of the bulk synchronous processing model (BSP) [Val90].

In this section the theoretical background for the design of Algorithms using MapReduce and Pregel is introduced. In the following section, Section 2.4, the software and computing frameworks commonly used in cloud computing environments are reviewed.

2.3.1 MapReduce

MapReduce was first introduced by [DG08] for distributed batch processing. MapReduce algorithms are based on the concept of splitting problem instances into small parts (not necessarily disjoint) and performing computations on those parts using independent computation nodes.

In particular MapReduce algorithms work in three phases: (1) map, (2) shuffle and (3) reduce phase. In the map phase, the input data is read from the file system and the read values are mapped to an arbitrary number of key-value pairs. Then, in the shuffle phase, those key-value pairs are distributed among the reduce tasks depending on the key. In the reduce phase, each worker performs a computation on the received key-value pairs and writes the result back to the file system.

Introduction by example. The concept of MapReduce is best illustrated by an example. We compute the Borda scores of a small election using a simple MapReduce algorithm. The algorithm is illustrated in Figure 2.4. The set of candidates is $A = \{a, b, c\}$ and a valid vote is for example $a > b > c$. The used scoring vector for the Borda scoring rule is $s = (2, 1, 0)$. Therefore, the candidate ranked first receives 2 points, the second 1 point and the last 0 points. In the map-phase each preference relation is read and mapped to key-value pairs. The vote $a > b > c$ results in the key-value pairs: (key=a,value=2),(key=b,value=1) and (key=c,value=0). The key-value pairs with value 0 can be omitted, since they are not relevant for the result. In the shuffle phase, all those key-value pairs are assigned to reduce tasks by their key, such that all scores of a candidate end up at the same reduce task. In the reduce phase each reduce task just sums up all the scores for the given candidate. Then the reducers write their calculated Borda scores back to the file system.

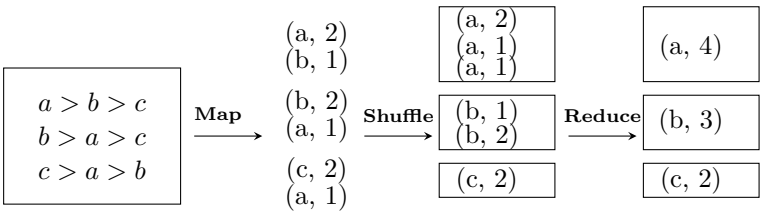


Figure 2.4: Calculation of scores by the Borda scoring rule.

Performance Analysis of MapReduce algorithms

There is a lot of research on analyzing the performance of MapReduce algorithms [ASSU13, LRU14] designed for cloud computing environments. In such cloud computing environments the computing nodes are connected by a network and therefore the network capacity can quickly become a barrier. That is why communication cost is an important factor. The *total communication cost* is denoted as *tcc*. It measures the amount of data that is moved to the reduce tasks during the shuffle phase. The *tcc* is closely related to the *replication rate* (*rr*) and the replication rate is the average number of key-value pairs that are produced by one input value. Therefore the *rr* measures the proportion of the *tcc* to the input size. The *number of keys* used for the computation is referred to by #keys and is a measure of parallelization. The more keys are used the more parallelization is possible. Note, that the number of keys is not necessary identical to the number of reduce tasks. Usually several keys can be processed by one reduce tasks. The assignment of keys to reduce tasks can be done by a simple hash function and can be used to tune the algorithm. Cloud computing frameworks discussed in Section 2.4 have some methods for load balancing and performance optimization implemented.

It is often necessary to chain several MapReduce computations and therefore the number of *MapReduce rounds* is also an important performance measure. The number of rounds is referred to as #rounds.

As another time measure we will study the *wall clock time* (*wct*), which measures the maximum time consumed by a single computation path in the parallel execution of the algorithm. Since the computations performed by the map and reduce tasks are very simple, the predominant cost factor is the size of the input and output of the map and reduce tasks. Therefore, we identify the wall clock time with the maximum number of input and output data items (i.e., the number of emit() and return() statements in the algorithms) in any computation path.

Performance Analysis of a MapReduce Algorithm by Example. Analysing the simple MapReduce algorithm for Borda scores described above gives the following result:

Performance: *The Borda scores for a given preference profile P can be computed in MapReduce with the following characteristics: $rr = 1$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m$, $wct \leq n + 1$, and $tcc \leq m(n + 1)$.*

Argumentation: *The algorithm takes only one MapReduce round ($\#\mathsf{rounds} = 1$). Each entry in the preference list results in one key-value pair, therefore the replication rate is 1 (rr=1). The candidates are used as key, such that we have $\#\mathsf{keys} = m$ keys. Each reduce task receives at most n values (one per vote) and outputs exactly 1 value. From this follows that $wct \leq n + 1$ and $tcc \leq m(n + 1)$.* ◇

2.3.2 Pregel

In many big data applications we are dealing with large graphs, such as social networks or linked data. For such data instances the MapReduce programming paradigm is not that well suited. Pregel [MAB+10] was developed to optimize the computation of the commonly known page-rank graph algorithm, for ranking web search results. Pregel algorithms are often described as vertex-centric computations. Each vertex acts as its own computation entity. Vertices send messages to other vertices, store their own vertex information and perform computations. The whole Pregel computation works in several supersteps. A superstep consists of three phases: (1) vertices send messages to other vertices, (2) vertices process the data and perform computation, (3) vertices can be set inactive. After receiving messages, a vertex performs its vertex program, in which the information stored at the vertex can be changed. Inactive vertices do not send any messages, and a vertex can be set to inactive/active by itself. If a vertex does not receive any messages, it is set inactive automatically and can be reactivated by messages in following supersteps. Pregel programs terminate as soon as all vertices are inactive.

Performance Considerations for Pregel Algorithms

There are very similar performance consideration for Pregel algorithms as for Mapreduce. The communication cost and the usage of space plays an important role. In particular we are interested in the following measures: space usage, communication cost, computation cost and number of supersteps (rounds). The *space usage* refers to the space of storage each

vertex requires to store its vertex information. The *computation cost* is the computation cost of the vertex program, i.e. the computation performed by each vertex, and the communication cost refers to the size of the messages sent by each vertex in each superstep. The concept of balanced practical pregel algorithms (BPPA) has been introduced by [YCX+14] and defined that a BPPA algorithm has the following properties: linear space usage, linear communication cost, linear computation cost and uses at most a logarithmic number of rounds [YCX+14]. For many problems it is difficult to find algorithms that meet these performance guarantees.

2.4 Cloud Computing Frameworks

2.4.1 Hadoop

Apache Hadoop (`hadoop.apache.org/`) was originally programmed as an open source implementation of MapReduce and is implemented in Java. It quickly developed to a huge ecosystem of applications and is now much more than 'just' MapReduce. It comes together with highly developed resource management tools and the hadoop distributed file system (HDFS) as a central part.

2.4.2 Spark

Spark[1] is a cloud computing framework built on top of Hadoop and answers to many weaknesses of Hadoop, when it comes to interactive datanalysis. Spark is widely applied to many data-intensive tasks and has first been introduced in 2010 by [ZCF+10]. The core concept of the Spark engine are the Resilient Distributed Datasets (`RDD`). `RDD`s are distributed and read-only collections of objects. In contrast to Hadoop, where the data has to be written back to the file system after each map task, the data can be held in memory on the cluster. The ability of loading data into memory for reusing it in following steps gains enormous speed to iterative computations and makes interactive analysis possible. A quickly developing optimization engine is working in the background to make this possible. `RDD`s are distributed and replicated across the nodes in the cluster, which does not only speed up the computation but also provides adata safety in the case one partition of an `RDD` is lost. The partitioning can be configured, but also internal Spark processes optimize the process.
A very important concept of `RDD`s is their lazy evaluation. This causes that the `RDD` is created only when the data is needed and not before. Therefore Spark tries to minimize the size of datasets read into memory and can create an optimized execution plan before even distributing the data on the worker nodes. There are usually two ways of creating an `RDD`: either byloading data from a file or by parallelizing an already existing collection of objects. The second approach is usually not applicable since this would require to load the whole dataset in memory on one machine before distributing it.
RDDs allow for two types of methods: actions and transformations. *Transformations*

[1] `https://spark.apache.org/`

cause a new RDD to be created. Because of the lazy-evaluation this new RDD is not created before the data is actually needed. The computation happens only when an *action* is called. Actions always cause the Spark engine to perform a computation [KKWZ15, KW17].

2.4.3 GraphX

GraphX [XGFS13][2] is a library for Spark containing many methods for computations with graphs. Besides basic graph operators and algorithms, GraphX also provides an Pregel API. GraphX also provides methods to optimize the partitioning of the Graph on the cluster in order to avoid huge communication costs and to speed up the computation time.

[2] https://spark.apache.org/graphx/

Part II

Complexity Results and Cloud Computing Algorithms

CHAPTER 3

Complexity of Voting Rules

The first step in developing successful algorithms is to get a thorough understanding of the problem at hand. The goal of this theoretical complexity section is to determine the properties and the structure of considered problems and extending known results. In [BFH09] the complexity of the Smith Set, Schwartz Set and Copeland Set was analyzed. Those methods are all methods for selecting a winning set of candidates from an election. The results show that the winner determination problem in the Smith Set is AC^0-complete, the selection of the Schwartz set is NL-complete and the Copeland set is TC^0-hard [BFH09]. Based on these results cloud computing algorithms are proposed and analyzed theoretically with regard to several performance measures in Chapter 4.

For several other voting rules new complexity results are derived. In particular the STV rule, ranked-pairs method and the Schulze method are investigated. The STV und ranked-pairs winner determination proplems are both known to be NP-hard under the assumption that no tie-braking rule is given[CRX09, BF12]. In this work, for the STV rule the P-hardness of this problem is shown and further the $paraL$-membership under the parameterization by the number of candidates in an election (but with unrestricted number of voters) is proved. Under this parameterization a MapReduce algorithm for the STV rule is developed in Chapter 4.

For the ranked-pairs method we obtain a similar result and show that it is P-complete. Problems which are P-complete are considered as inherently sequential [Joh90].

For the Schulze Winner Determination problem, we show, that it is NL-complete and therefore suited for parallel computation.

The theorems together with their proofs presented in this section are:

- **Theorem 1**: The Schulze Winner Determination problem is NL-complete.
- **Theorem 2**: The Ranked Pairs Winner Determination problem is P-complete.
- **Theorem 3**: The STV Winner Determination problem is P-complete.
- **Theorem 4**: The STV Winner Determination problem can be solved in $\mathcal{O}(m + \log(n))$ space.

3.1 The Schulze Winner Determination Problem

Theorem 1 *The* Schulze Winner Determination *problem is* NL*-complete.*

Recall from Section 2.1 that the Schulze-Winner-Determination problem selects a candidate as winner if the outgoing widest paths to all other vertices, are stronger than the incoming widest paths.

In [BFH09] it is shown that any rule that selects winners from the Schwartz set—as Schulze's method does—and that breaks ties among candidates according to a fixed order is NL-hard to compute. Our result is similar, but shows NL-hardness without requiring a tie-breaking order. In this hardness proof, we construct the weighted majority graph $\mathcal{W}_P = (A, E, \mu')$ with even integer weights and with E being an asymmetric relation, i.e, if $(a, b) \in E$ then $(b, a) \notin E$. It follows from McGarvey's Theorem [McG53] that such weighted majority graphs can be obtained even from preference profiles containing only $\sum_{e \in E} \mu(e)$ many linear orders.

Of course, the results presented in this Chapter also hold if preference profiles (and not the weighted majority graph) are given as input.

Proof 1 *Our NL-membership proof for the Schulze-Winner Determination Problem is based on the following two related problems* $\mathrm{WP}_{\geq}$ *and* $\mathrm{WP}_{>}$*, where we ask if, for given vertices* s, t *and width* w*, a path from* s *to* t *of width* $\geq w$ *or* $> w$*, respectively, exists. Formally, we study the following problems:*

Existence-of-Wide-Paths $\mathrm{WP}_{\geq}$ / $\mathrm{WP}_{>}$	
Instance:	*a weighted graph* $\mathcal{G}$*, vertices* s, t*, weight* $w \in R$
Question:	*Does there exist a path from* s *to* t *of width* $\geq w$ *(in case of the* $WP_{\geq}$*-problem) or of width* $> w$ *(in case of the* $WP_{>}$*-problem)?*

It is straightforward to verify that both problems $\mathrm{WP}_{\geq}$ *and* $\mathrm{WP}_{>}$ *are in* NL*: guess one vertex after the other of a path from* s *to* t *of length at most* $|V|$ *and check for any two successive vertices that they are connected by an edge of weight* $\geq w$ *or* $> w$*, respectively.*

Since $\mathrm{NL} = \mathsf{co\text{-}NL}$ *by the famous Immerman-Szelepcsényi Theorem, the co-problems of* $\mathrm{WP}_{\geq}$ *and* $\mathrm{WP}_{>}$ *are also in* NL*. We can thus construct a non-deterministic Turing machine (NTM) for the Schulze winner determination problem, which loops through all candidates* $c' \neq c$ *and does the following:*

- *guess the width* w *of the widest path from* c *to* c' *(among the weights of the edges in* $\mathcal{W}_P$*);*
- *solve the* $\mathrm{WP}_{\geq}$ *problem for vertices* c, c' *and weight* w*: check that there exists a path from* c *to* c' *of width* $\geq w$*;*
- *solve the co-problem of* $\mathrm{WP}_{>}$ *for* c', c *and weight* w*: check that there is* no *path from* c' *to* c *of width* $> w$*;*

The correctness of the non-deterministic Turing machine is immediate. Moreover, by the above considerations on the problems $\mathrm{WP}_{\geq}$*,* $\mathrm{WP}_{>}$ *and their co-problems, the NTM clearly works in log-space time.*

Note that if we want to check if c *is the* unique *Schulze winner, then we just need to replace the third step above by a check that there is no path from* c' *to* c *of weight* $\geq w$*; in other words, we solve the co-problem of* $\mathrm{WP}_{\geq}$*. Again, the overall non-deterministic Turing machine clearly works in log-space.*

We prove the NL-hardness of the Schulze Winner Determination problem by reduction from the NL*-complete Reachability problem. Let* $(\mathcal{G}, a, b)$ *be an arbitrary instance of Reachability with* $\mathcal{G} = (V, E)$ *and* $a, b \in V$*. We construct a weighted majority graph* $\mathcal{W} = (V', E', \mu)$ *from* $\mathcal{G}$ *as follows and choose* a *as the distinguished candidate:*

- *First, remove from* $\mathcal{G}$ *all edges of the form* (v, a) *and* (b, v) *for every* $v \in V$*, i.e., all incoming edges of* a *and all outgoing edges from* b*.*
- *For every pair of symmetric edges* $e_1 = (v_i, v_j)$ *and* $e_2 = (v_j, v_i)$*, choose one of these edges (say* e_1*) and introduce a "midpoint", i.e., add vertex* w_{ij} *to* V *and replace* e_1 *by the two edges* (v_i, w_{ij}) *and* (w_{ij}, v_j)*.*
- *For every vertex* u *different from* b*, introduce a new vertex* r_u *and edges* (b, r_u)*,* (r_u, u)*.*
- *Now there is exactly one incoming edge of* a*, namely* $e = (r_a, a)$*. Define the weight of this edge as* $\mu(e) = 2$ *and set* $\mu(e') = 4$ *for every other edge* $e' \neq e$*.*

It is easy to verify that a *is a Schulze winner in* $\mathcal{W}$ *(actually, it is even the unique Schulze winner), if and only if there is a path from* a *to* b *in* $\mathcal{G}$*. To see this, first observe that*

there is a path from a to b in $\mathcal{W}$ if and only if there is one in $\mathcal{G}$. Hence, if there is a path from a to b, then the widest path from a to any vertex in $\mathcal{W}$ is 4. Conversely, all paths from any vertex to a must go through edge (r_a, a) and, therefore, have width at most 2. On the other hand, if there is no path from a to b, then a cannot be a winner, since b indeed has a path to a (via r_a) and, therefore, in this case, b is definitely preferred to candidate a according to the Schulze method.

3.2 The Ranked Pairs Winner Determination Problem

Theorem 2 *The* Ranked Pairs Winner Determination *problem is* P*-complete.*

P-membership in case of the ranked pairs method is obvious (and well-known). It remains to prove the P-hardness.

Proof 2 *The P-hardness proof is by reduction from the classical* P*-complete Boolean-Formula-Evaluation (BFE) problem: given a Boolean formula ϕ with variables in X and truth assignment I on X, does $I \models \phi$ hold?*

Consider an arbitrary instance (ϕ, X, I) of the BFE problem. Without loss of generality, we may assume that (1) each variable in ϕ occurs exactly once in ϕ and (2) the only connective in ϕ is nor*. We construct an instance of the Ranked Pairs Winner Determination Problem $(\mathcal{W}, v)$ with the weighted majority graph $\mathcal{W} = (A, E_P, \mu)$ as follows:*

Consider the parse tree $\mathcal{T}$ of ϕ, where each node of $\mathcal{T}$ corresponds to a subexpression of ϕ. In particular, the leaf nodes correspond to the variables in ϕ. Now let $\{g_1, \ldots, g_m\}$ denote the inner nodes of $\mathcal{T}$ in some top-down *order, i.e., whenever g_i is an ancestor of g_j, then $i < j$ holds. Moreover, let $X = \{x_1, \ldots, x_\ell\}$ and let $\{g_{m+1}, \ldots, g_{m+\ell}\}$ denote the corresponding leaf nodes in $\mathcal{T}$. Then we set*

$$A = \{g_0\} \cup \{g_1, \ldots, g_m, g_{m+1}, \ldots, g_{m+\ell}\} \cup \{h_1, \ldots, h_m\}.$$

By slight abuse of notation, we use $g_1, \ldots, g_{m+\ell}$, to denote (1) candidates/vertices in $\mathcal{W}$, (2) nodes in $\mathcal{T}$, and (3) subexpressions of ϕ. The vertices $g_0, h_1, \ldots, h_m$ are new symbols. We say that "g_j is the parent of g_i", if g_j is the parent of g_i in the parse tree $\mathcal{T}$. In addition, we define g_0 as the "parent of g_1".

To define the set of edges E together with weight function μ, let $\alpha \in \{1, \ldots, m + \ell\}$; we distinguish two cases:

Case 1: Suppose $\alpha \in \{1, \ldots, m\}$, i.e., g_α is an inner node of $\mathcal{T}$. Let g_i be the parent of g_α and, for $\alpha \neq 1$, let g_j be the parent of g_i. Then E contains the following edges:

$e_0 = (g_j, g_\alpha)$ *with* $\mu(e_0) = 4\alpha$, *for every* $\alpha \neq 1$.

$e_1 = (g_\alpha, h_\alpha)$ *with* $\mu(e_1) = 4\alpha - 1$.

$e_2 = (h_\alpha, g_i)$ *with* $\mu(e_2) = 4\alpha - 2$.

$e_3 = (g_i, g_\alpha)$ *with* $\mu(e_3) = 4\alpha - 3$.

Case 2: Suppose $\alpha \in \{m+1, \ldots, m+\ell\}$, *i.e.,* g_α *is a leaf node in* $\mathcal{T}$ *(corresponding to some variable* x_γ*). Again, let* g_i *denote the parent of* g_α *and let* g_j *denote the parent of* g_i*. Then* E *contains* $e_0 = (g_j, g_\alpha)$ *with* $\mu(e_0) = 4m + 2\gamma$*. Moreover, if* $I(x_\gamma) = true$*, then* E *contains* $e_1 = (g_\alpha, g_i)$*. Conversely, if* $I(x_\gamma) = false$*, then* E *contains* $e_1 = (g_i, g_\alpha)$*. In either case, we set* $\mu(e_1) = 4m + 2\gamma - 1$.

These are all the edges in E*. For the sake of readability, we use the integer interval from 1 to* $4m + 2\ell$ *as weights (with weight 4 missing, since* g_1 *has no grand-parent). The weights could be easily made even by multiplying all weights by* 2*. Finally, we choose* g_1 *as the distinguished vertex for which we have to decide whether it is a ranked pairs winner.*

The intuition of this construction is as follows: in the ranked pairs method, we inspect the edges in descending order of their weights (note that all edges in $\mathcal{W}$ *have unique weights) and check if the current edge may be added to the relation* R *(now considered as a DAG) without producing a cycle. Then* R *contains either the edge* (g_j, g_i) *or a path from* g_i *to* g_j *(in case* g_i *is an inner node of* $\mathcal{T}$*, this is the path* $g_i \to h_i \to g_j$*; in case* g_i *is a leaf node, this is the edge* (g_i, g_j)*). The crucial property of our construction (which can be proved by structural induction on the nodes of the parse tree* $\mathcal{T}$*) is that the existence of a path from* g_i *to* g_j *encodes* $I \models g_i$ *(recall that we identify nodes in the parse tree with the corresponding subexpressions of* ϕ*), whereas the existence of the edge* (g_j, g_i) *in* R *encodes* $I \not\models g_i$.

In particular, for g_1 *with parent* g_0 *this means that if* $I \not\models g_1$*, then* R *contains the edge* (g_0, g_1) *and* g_1 *is clearly not a winner. Conversely, suppose that* $I \models g_1$ *holds and let* $g_1 = g_\alpha \,\mathsf{nor}\, g_\beta$*. Of course,* $I \models g_1$ *means that* $I \not\models g_\alpha$ *and* $I \not\models g_\beta$*. Hence,* R *contains no incoming edge to* g_1 *at all. Hence, in this case,* g_1 *is a winner (actually even the unique winner) independently of the chosen tie-breaking order.*

The construction is illustrated by Example 3.

Example 3 *Consider formula* $\phi = x_1 \,\mathsf{nor}\, (x_2 \,\mathsf{nor}\, x_3)$ *and assignment* I *with* $I(x_1) = I(x_2) = true$ *and* $I(x_3) = false$*. The parse tree* $\mathcal{T}$ *of* ϕ *has 2 inner nodes* (g_1, g_2) *corresponding to the two occurrences of* nor *and 3 leaf nodes* (g_3, g_4, g_5) *for the 3 variables. The graph* $\mathcal{W}$ *resulting from our problem reduction and the DAG* R *resulting from applying the ranked pairs method are displayed in Figure 3.1, where* $\mathcal{W}$ *contains all edges shown in the figure while* R *contains only the edges with solid lines.*

The edges to the leaf nodes x_1, x_2 *and from leaf node* x_3 *directly encode the truth value of these variables in* I*. For each inner node* g_α *(for* $\alpha \in \{1, 2\}$*) with parent* g_i*, the graph* $\mathcal{W}$ *contains the triangle* g_α, h_α, g_i*. Depending on the truth value of* g_α*, either edge* (h_α, g_i)

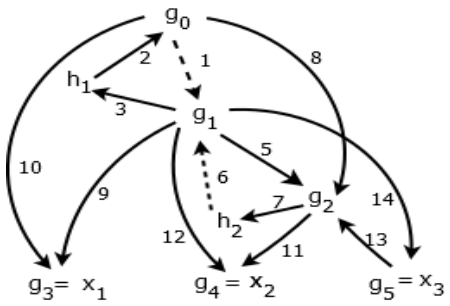


Figure 3.1: Graph $\mathcal{W}$ and relation R of Example 3 for Boolean formula $\phi = x_1 \,\mathsf{nor}\, (x_2 \,\mathsf{nor}\, x_3)$ and assignment I with $I(x_1) = I(x_2) = \text{true}$ and $I(x_3) = \text{false}$.

is retained in R (if $I \models g_\alpha$) or edge (g_i, g_α) (if $I \not\models g_\alpha$). Formula ϕ evaluates to true for the assignment I and g_1 is indeed the unique winner in the election represented by $\mathcal{W}$.

3.3 The STV Winner Determination Problem

In this section the Single Transferable Vote (STV) rule is studied as one example of a voting rule that allows for only limited parallelization. Recall from Chapter 2.1 that STV is defined as follows: Every voter provides a total order of all m candidates. In each round the candidate that is ranked first in the least number of votes is removed from each vote. The remaining candidate after $m - 1$ rounds is the winner. In case of ties we assume that a tie-breaking order is given. In the following we will show that STV is in general difficult to parallelize effectively.

Theorem 3 STV-Winner *is* P*-complete.*

Proof 3 *The proof of* P*-hardness is by reduction from the well-known* P*-hard* Boolean Circuit Evaluation *problem. Let an arbitrary instance of this problem be given by a Boolean circuit $\mathcal{C}$. We may assume that $\mathcal{C}$ has the following form [GHR95]:*

- *$\mathcal{C}$ has gates $g_1, \ldots, g_n$;*
- *the gates $g_1, \ldots, g_k$ with $1 \leq k < n$ are the input gates, i.e., each g_i with $1 \leq i \leq k$ is either of type $\top$ or $\bot$;*
- *all non-input gates (i.e., g_i with $k + 1 \leq i \leq n$) are either $\wedge$-gates or $\vee$-gates;*
- *for every i in $\{k + 1, \ldots, n\}$, g_i receives its input from some gates g_α and g_β with $\alpha, \beta < i$;*
- *g_n is the output gate.*

From this, we construct an instance (C, V, c_n) *of the* STV Winner Determination *problem, where the set of candidates* C *and the set of votes* V *are defined as follows:*

$C = \{c_1, \bar{c}_1, c_2, \bar{c}_2, \ldots, c_n, \bar{c}_n\}$.

For the definition of V*, it is convenient to introduce the following notation: Let* $D = \{d_1, \ldots, d_j\} \subseteq C$*, s.t. either* $\{c_n, \bar{c}_n\} \subseteq D$ *or* $\{c_n, \bar{c}_n\} \cap D = \emptyset$*. Then we write*

$$d_1 \succ d_2 \succ \cdots \succ d_j$$

as a short-hand for the total order

$$d_1 \succ d_2 \succ \cdots \succ d_j \succ e_{j+1} \succ e_{j+2} \succ \cdots \succ e_n,$$

where $\{e_{j+1}, e_{j+2}, \ldots, e_n\} = C \setminus D$ *is defined as follows: If* $\{c_n, \bar{c}_n\} \subseteq D$ *holds, then the* e_i*'s are arranged in arbitrary order. If* $\{c_n, \bar{c}_n\} \cap D = \emptyset$ *holds, then* $e_{j+1} = c_n$*,* $e_{j+2} = \bar{c}_n$*, and the remaining* e_i*'s are arranged in arbitrary order.*

The intuition here is that c_n *or* $\bar{c}_n$ *serve as "stop elements", i.e., our definition of the instance of* STV-Winner *will guarantee that one of* c_n *and* $\bar{c}_n$ *must be the winner. Hence, the order of the candidates after* c_n *and* $\bar{c}_n$ *in any vote is irrelevant, since they will never come to first place no matter in which order the candidates different from* c_n *and* $\bar{c}_n$ *get eliminated.*

We now define the set of vertices $V = V_1 \cup V_2 \cup V_3$ *as follows:*

V_1 *contains* $8n^3 + 2n^2 \cdot (n-1)$ *votes, namely for every* $i \in \{1, \ldots, n\}$*:*
$4n^2 + 4n \cdot (i-1)$*-times the vote* $c_i \succ \bar{c}_i$
$4n^2 + 4n \cdot (i-1)$*-times the vote* $\bar{c}_i \succ c_i$

V_2 *contains* n *votes, namely for every* $i \in \{1, \ldots, n\}$*:*

[Case 1.] If g_i *is an input gate of type* $\top$ *or an* $\vee$*-gate, then* V_2 *contains the vote* $c_i \succ \bar{c}_i$*.*

[Case 2.] If g_i *is an input gate of type* $\bot$ *or an* $\wedge$*-gate, then* V_2 *contains the vote* $\bar{c}_i \succ c_i$*.*

V_3 *contains* $4(n-k)$ *votes, namely for every* $i \in \{k+1, \ldots, n\}$*:*

[Case 1.] If g_i *is a an* $\vee$*-gate, which takes its input from* g_α, g_β *with* $\alpha < \beta$*, then* V_2 *contains*
2-times the vote $c_\beta \succ c_\alpha \succ \bar{c}_i \succ c_i$
2-times the vote $\bar{c}_\beta \succ c_\beta$

[Case 2.] If g_i *is a an* $\wedge$*-gate, which takes its input from* g_α, g_β *with* $\alpha < \beta$*, then* V_2 *contains*
2-times the vote $\bar{c}_\beta \succ \bar{c}_\alpha \succ c_i \succ \bar{c}_i$
2-times the vote $c_\beta \succ \bar{c}_\beta$

For the correctness of our problem reduction, we have to show that c_n is the STV-winner of the election (C, V) if and only if the Boolean circuit $\mathcal{C}$ (i.e., the output gate g_n) evaluates to true.

First recall that according to the STV-rule, the winner is determined in $2n-1$ rounds, such that in each round one candidate is eliminated, namely the one that is ranked in first place (among the remaining candidates) by the smallest number of votes. For every $j \in \{0, \ldots, 2n-1\}$, let C_j denote the set of remaining candidates after j rounds. To prove the above equivalence, it suffices to prove the following two claims:

Claim 1. *For every $j \in \{0, \ldots, n\}$, C_j has the following properties:*

for every $i \in \{1, \ldots, j\}$, exactly one of c_i and $\bar{c}_i$ is in C_j:
$c_i \in C_j$ if and only if gate g_i evaluates to true and
$\bar{c}_i \in C_j$ if and only if gate g_i evaluates to false;

for every $i \in \{j+1, \ldots, n\}$, both c_i and $\bar{c}_i$ are in C_j.

Claim 2. *For every $j \in \{0, \ldots, n-1\}$, C_{n+j} has the following properties:*

for every $i \in \{1, \ldots, j\}$, neither c_i nor $\bar{c}_i$ is in C_{n+j};
for every $i \in \{j+1, \ldots, n\}$, exactly one of c_i and $\bar{c}_i$ is in C_j (namely the one which was in C_n according to Claim 1).

From these two claims, the correctness of our reduction follows immediately. Indeed, by Claim 2, the set C_{2n-1} of candidates remaining after $2n-1$ rounds is a singleton consisting of either c_n or $\bar{c}_n$. By Claim 1, it is c_n if and only if g_n evaluates to true (and it is $\bar{c}_n$ if and only if g_n evaluates to false).

Proof of Claim 1. *We proceed by induction on j. For $j = 0$, the claim holds trivially. So suppose that the claim holds for some $j \in \{0, \ldots, n-1\}$. We have to show that it also holds for $j+1$. To this end, we divide Claim 1 in two parts: (1) First we show that in the $(j+1)$-st round, either c_{j+1} or $\bar{c}_{j+1}$ is eliminated from C_j. (2) We will then show that c_{j+1} is retained if gate g_{j+1} evaluates to true and $\bar{c}_{j+1}$ is retained if gate g_{j+1} evaluates to false.*

(1) For the first part of the proof, we may assume by the induction hypothesis that the set of candidates C_j retained after j steps has the form $\{d_1, d_2, \ldots, d_j\} \cup \{c_{j+1}, \bar{c}_{j+1}, \ldots, c_n, \bar{c}_n\}$, where d_i is either c_i or $\bar{c}_i$.

First inspect the votes in V_1: in the original votes $c_i \succ \bar{c}_i$ and $\bar{c}_i \succ c_i$ with $i \leq j$, candidate d_i is now in first place. This is at least $8n^3$-times the case. All candidates c_i and $\bar{c}_i$ with $i \geq j+1$ are exactly $4n^2 + 4n \cdot (i-1)$-times in first place. In particular, each of the two candidates c_{j+1} and $\bar{c}_{j+1}$ occurs exactly $4n^2 + 4n \cdot j$-times in first place.

Now inspect the votes in V_2. For every $i \leq n$, at least one of c_i or $\bar{c}_i$ is still present in C_j. Hence, over all votes in V_2, each candidate of C_j (i.e., this applies in particular to c_{j+1} and $\bar{c}_{j+1}$) occurs at most once in first place.

Finally, recall that V_3 contains in total $4(n-k)$ votes. Hence, even if one of c_{j+1} and $\bar{c}_{j+1}$ happens to be in first place in all of these votes, there are clearly no more than $4(n-1)$ votes with this property (since we have $k \geq 1$ for the number k of input gates). Summing up all possible occurrences of c_{j+1} or $\bar{c}_{j+1}$ in first place of some vote, we get $(4n^2 + 4n \cdot j) + 1 + 4(n-1) = 4n^2 + 4n \cdot (j+1) - 3$ as a first upper bound. By $j \leq n-1$, we can transform this expression into $8n^2 - 3$ as a second upper bound.

Now consider the number of occurrences in a vote in V_1 alone for each $d_i \in C_j$ and each $c_i, \bar{c}_i$ with $i \geq j+2$. As we have observed above, there are at least $8n^2$ occurrences of every $d_i \in C_j$ in first place, which is greater than the second upper bound on the possible occurrences of c_{j+1} or $\bar{c}_{j+1}$ in first place. Moreover, each $c_i, \bar{c}_i$ with $i \geq j+2$ has at least $4n^2 + 4n \cdot (i-1)$ occurrences in first place. For $i \geq j+2$, this means that there are at least $4n^2 + 4n \cdot (j+1)$ such occurrences, which is greater than the first upper bound derived above for c_{j+1} or $\bar{c}_{j+1}$. Hence, in any case, each $d_i \in C_j$ and each $c_i, \bar{c}_i$ with $i \geq j+2$ has more occurences in first place than c_{j+1} and $\bar{c}_{j+1}$ and, therefore, one of c_{j+1} and $\bar{c}_{j+1}$ must be eliminated in round $j+1$.

(2) For the second part of the proof, we have to check which of c_{j+1} and $\bar{c}_{j+1}$ is deleted in round $j+1$. As observed above, in V_1 both have precisely the same number of occurrences in first place, namely $4n^2 + 4n \cdot (i-1)$. Now, if g_{j+1} is an input gate of type $\top$ or an $\vee$-gate, then we have the vote $c_{j+1} \succ \bar{c}_{j+1}$ in V_2 and no further vote with c_{j+1} or $\bar{c}_{j+1}$ in first place. If g_{j+1} is an input gate of type $\bot$ or an $\wedge$-gate, then we have the vote $\bar{c}_{j+1} \succ c_{j+1}$ in V_2 and again no further vote with c_{j+1} or $\bar{c}_{j+1}$ in first place.

Finally, look at the votes in V_3. Consider the votes $c_\beta \succ c_\alpha \succ \bar{c}_i \succ c_i$ and $\bar{c}_\beta \succ c_\beta$ each included twice in V_3 if g_i is a an $\vee$-gate, which takes its input from g_α, g_β with $\alpha < \beta < i$. Likewise, we have votes $\bar{c}_\beta \succ \bar{c}_\alpha \succ c_i \succ \bar{c}_i$ and $c_\beta \succ \bar{c}_\beta$ twice each in V_3 if g_i is a an $\wedge$-gate, which takes its input from g_α, g_β with $\alpha < \beta < i$.

Clearly, g_{j+1} may play the role of such a gate g_β providing input to some other gate g_i with $j+1 < i$. By the induction hypothesis, since we assume $\alpha < \beta$, c_α may be present in C_j or not. In any case, in these 4 votes introduced into V_3 for gate g_i, each of c_{j+1} and $\bar{c}_{j+1}$ has precisely 2 occurrences in first place in V_3 after j rounds. Hence, if g_{j+1} is an input gate (i.e., $j+1 \leq k$), then, in V_3, the candidates c_{j+1} and $\bar{c}_{j+1}$ have the same number of occurrences in first place after j rounds. Hence, the winner between c_{j+1} and $\bar{c}_{j+1}$ only depends on the votes in V_2. That is, we get the desired equivalence: g_{j+1} evaluates to true if and only if input gate g_{j+1} is of type $\top$ if and only if after j rounds, c_{j+1} has in total more occurrences in first place than $\bar{c}_{j+1}$ (namely by a margin of just 1) if and only if c_{j+1} is retained in C_{j+1} while $\bar{c}_{j+1}$ is deleted in round $j+1$.

It remains to consider the case that g_{j+1} is not an input gate. By the above consideration, in the 4 votes added to V_3 for some g_i with $i \neq j+1$, either both c_{j+1} and $\bar{c}_{j+1}$ have two occurrences in first place or both have no such occurrence. We therefore only need to consider the 4 votes introduced in V_3 for non-input gate g_i with $i = j+1$. We treat the cases of $\vee$-gates and $\wedge$-gates separately.

Case 1. *First consider the case that g_{j+1} is an $\vee$-gate which receives its input from some gates g_α, g_β with $\alpha < \beta < j+1$. We have the following chain of equivalences: The two original votes of the form $c_\beta \succ c_\alpha \succ \bar{c}_{j+1} \succ c_{j+1}$ are turned into $\bar{c}_{j+1} \succ c_{j+1}$ after j rounds if and only if both c_β and c_α have been deleted in the first j rounds if and only if (by the induction hypothesis) both gates g_β and g_α evaluate to false if and only if the $\vee$-gate g_{j+1} evaluates to false. In other words, in V_3, $\bar{c}_{j+1}$ has two more occurrences in first place than c_{j+1} if and only if g_{j+1} evaluates to false, while $\bar{c}_{j+1}$ and c_{j+1} have the same number of occurrences in first place in V_3 if and only if g_{j+1} evaluates to true. Combining this with the above observation that, in V_2, c_{j+1} has exactly one more occurrence in first place than $\bar{c}_{j+1}$, we get the desired equivalence, namely: g_{j+1} evaluates to true if and only if after j rounds, c_{j+1} has in total more occurrences in first place than $\bar{c}_{j+1}$ (namely by a margin of just 1) if and only if c_{j+1} is retained in C_{j+1} while $\bar{c}_{j+1}$ is deleted in round $j+1$.*

Case 2. *The case that g_{j+1} is an $\vee$-gate which receives its input from some gates g_α, g_β with $\alpha < \beta < j+1$ is similar. Our chain of equivalences now looks as follows: The two original votes of the form $\bar{c}_\beta \succ \bar{c}_\alpha \succ c_{j+1} \succ \bar{c}_{j+1}$ are turned into $c_{j+1} \succ \bar{c}_{j+1}$ after j rounds if and only if both $\bar{c}_\beta$ and $\bar{c}_\alpha$ have been deleted in the first j rounds if and only if (by the induction hypothesis) both gates g_β and g_α evaluate to true if and only if the $\wedge$-gate g_{j+1} evaluates to true. In other words, in V_3, c_{j+1} has two more occurrences in first place than $\bar{c}_{j+1}$ if and only if g_{j+1} evaluates to true, while c_{j+1} and $\bar{c}_{j+1}$ have the same number of occurrences in first place in V_3 if and only if g_{j+1} evaluates to false. Together with V_2, we again get desired equivalence, namely: g_{j+1} evaluates to true if and only if after j rounds, c_{j+1} has in total more occurrences in first place than $\bar{c}_{j+1}$ (namely by a margin of just 1) if and only if c_{j+1} is retained in C_{j+1} while $\bar{c}_{j+1}$ is deleted in round $j+1$.* ◇

Proof of Claim 2. *Again we proceed by induction on j. For $j = 0$, we have $C_{n+j} = C_n$. Hence, Claim 2 follows directly from Claim 1. So suppose that Claim 2 holds for some $j \in \{0, \ldots, n-2\}$. We have to show that it also holds for $j+1$. By the induction hypothesis, the set of candidates C_{n+j} has the form $\{d_{j+1}, d_{j+2}, \ldots, d_n\}$, where d_i is either c_i or $\bar{c}_i$. We have to show that in round $n+j+1$, the candidate d_{j+1} gets eliminated, i.e., candidate d_{j+1} has the least number of occurrences in first place among the candidates in C_{n+j}.*

We first observe that in V_1 alone, every d_i with $i \geq j+1$ has exactly $8n^2 + 8n \cdot (i-1)$ occurrences in first place. For $i \geq j+2$, we get the lower bound $8n^2 + 8n \cdot (j+1)$ on the occurrences in first place in V_1 and, hence, also in V.

Now compute the number of occurrences of d_{j+1} in first place. In V_1, we get exactly $8n^2 + 8n \cdot j$ occurrences. By $|V_2| = n$ and $|V_3| < 4n$, we get the upper bound $8n^2 + 8n \cdot j + n + 4n = 8n^2 + 8n \cdot (j+1) - 3n < 8n^2 + 8n \cdot (j+1)$. Hence, d_{j+1} is indeed the candidate that is deleted in round $n+j+1$. ◇

As argued above, the correctness of the P*-hardness proof follows immediately from the correctness of Claims 1 and 2.*

The next theorem shows that only the number m of candidates is the source of P-completeness and the number n of voters is not an obstacle to parallelization.

Theorem 4 STV-Winner *can be solved in* $\mathcal{O}(m + \log(n))$ *space.*

Proof 4 *We require the following variables to be kept in memory: the current vote under consideration* ($i \in \{1, \ldots, n\}$)*, the current candidate* c_j ($j \in \{1, \ldots, m\}$)*, the current score of candidate* c_j ($s \in \{1, \ldots, n\}$)*, the minimum score of the preceding candidates* $c_1, \ldots, c_{j-1}$ ($t \in \{1, \ldots, n\}$)*, the candidate having this minimum score* $c_{j'}$ ($j' \in \{1, \ldots, n\}$) *and the set of candidates that have already been removed* $A \subseteq C$*. The algorithm starts with* $i = 1$*,* $j = 1$*,* $j' = 0$*,* $s = 0$*,* $t = +\infty$ *and* $A = \emptyset$*. We repeat the following steps* $m - 1$ *times: For every candidate* $c_j \notin A$ *we compute the score of* c_j *by verifying for each vote* V_i *whether* c_j *is the highest ranking candidate not contained in* A*; if yes, we increase* s *by* 1*. If* s *is smaller than* t *(that is, we assume lexicographic tie-breaking), we set* $t \leftarrow s$ *and* $j' \leftarrow j$*. Once this has been done for every candidate, we add candidate* $c_{j'}$ *to* A*, and set* $i = 1$*,* $j = 1$*,* $j' = 0$*,* $s = 0$ *and* $t = +\infty$*. The remaining candidate after* $m - 1$ *iterations is the winner. The space requirements of this algorithm are* $\log(n)$ *for the variables* i, s, t*,* $\log(m)$ *for the variables* j, j' *and* m *for the set* A*.*

From the perspective of classical complexity theory, we have shown that the STV Winner Determination problem is contained in the complexity class L, if we fix m to a constant. Problems in L can be solved with logarithmic space and membership in L can be seen as evidence that a problem is parallelizable as it can be computed in $\log^2$ time with a polynomial number of parallel processors. Theorem 4 can also be seen from the perspective of parameterized space complexity [EST14]. The result translates to a para-L membership proof for the STV-Winner Determination problem with parameter m. para-L membership requires that the problem can be solved in $\mathcal{O}(f(m) + \log(n))$ space for some computable function f. Note that para-L containment is a stronger result than L membership for fixed m since the latter would also hold, for instance, for a space complexity of $\mathcal{O}(m \cdot \log(n))$.

CHAPTER 4

Cloud Computing Algorithms

In this section the developed cloud computing algorithms are presented. For the design of the Algorithms we took the complexity results from the preceding chapter into account. Those results give us insight which problems are parallelizable. We derive several cloud computing algorithms for winner determination methods: positional scoring rules, copeland scores, Smith set, Schwartz set, the Schulze method and the STV rule.

The outline of this section is as follows: first the calculation of positional scoring rules based on the preference profile is discussed. Next the transformation of a preference profile to a graph representation is explained. Following this preparation we discuss algorithms for voting rules for winner dermination problems based on the dominance graph and weighted majority graph. The chapter finishes with a MapReduce algorithm for the STV Winner Determination and some algorithm ideas for the ranked-pairs method. For the discussed winner determination problems we propose a Pregel based and a MapReduce based algorithm, if it makes sense to do so. Escpecially for problems which are not graph based, a Pregel algorithm is not proposed.

The presented cloud computing algorithms are:

- MapReduce Algorithm for calculating positional scoring rules
- MapReduce Algorithm for computing the weighted majority graph from the preference profile
- MapReduce Algorithm for computing the Copeland Set
- Mapreduce Algorithm for computing the Smith Set
- Pregel algorithm for computing the Smith Set
- MapReduce Algorithm for computing the Schwartz Set

- Pregel Algorithm for computing the Schwartz Set
- MapReduce Algorithm for computing the Schulze Method
- Pregel Algorithm for computing the Schulze Method
- Mapreduce Algorithm for computing the STV rule

The properties of all proposed algorithms are summarized and discussed in Chapter 5 and the experimental evaluation is presented in Chapter 6.

4.1 Positional Scoring Rules

The Winner Determination problem for positional scoring rules takes a preference profile P containing the preferences of all voters as input.

	Winner Determination under scoring rule $\mathcal{S}$
Instance:	a preference profile P containing m votes, scoring vector s of length m, candidate c
Question:	Does c have the maximum score of all candidates under scoring rule $\mathcal{S}$?

In Section 2.3.1 the MapReduce algorithm for computing Borda scores has already been explained and discussed. The MapReduce algorithm for computing Borda scores is illustrated in Figure 2.4 and the computational properties of computing Borda scores are discussed. Computing other positional scoring rules, which are based on a scoring vector, is done with a very similar aglorithm; only the scoring vector has to be adapted accordingly.

We do not propose a Pregel algorithm for positional scoring rules, since the computation is based on the original preference profile and not on a graph representation.

4.1.1 MapReduce Algorithm for Positional Scoring Rules

The MapReduce algorithm for general scoring rules with a scoring vector s of length m is shown in Algorithm 4.1.1. First every single vote is mapped to key-value pairs using the procedure `MapScores` (shown in Algorithm 4.1.2). The key of the resulting key-value pairs is equal to the candidates id and the value is the corresponding score, depending on the position of the candidate in the vote. For example, the vote $a \succ b \succ c$ under the scoring vector $s = (2, 1, 0)$ would result in the key-value pairs: (key=a, value=2), (key=b,value=1) and (key=c, value=0). Key-value pairs with a value equal to 0 can be left out, since they are not influencing the outcome. The key-value pairs are then shuffled to the reduce tasks (all key-value pairs with the same key end up at the same reduce

task). The reduce tasks then just sum up all received values per key to compute the scores. This is done by using the procedure `ReduceSum` shown in Algorithm 4.1.3.

To keep the notation simple the actual parameters in the procedure calls are omitted, but they are usually clear from the context. For the very first map procedure, the input is always the original input, i.e. the problem instance – for the scoring rules this is the preference profile as a list of preferences. The input to each reduce task corresponds to the output produced by the preceding map task, grouped by key.

Algorithm 4.1.1: Computing Positional Scores

Input: Preference profile P containing m votes and scoring vector s
Output: Candidates together with their score

```
1 MapScores
2 ReduceSum
```

Algorithm 4.1.2: MapScores()

Input: Preference profile P containing m votes and scoring vector s
Output: key-value pairs of the form (key=candidate,value=score)

```
1 foreach vote in P do
2 |   for i=1 to length(vote) do
3 |   |   emit(key=vote[i], value=s[i])
4 |   end
5 end
```

Algorithm 4.1.3: ReduceSum()

Input: Key-value pairs(key=candidate,value=score) produced by MapScores()
Output: Sum of all received values (candidate with score)

```
1 return (key, sum(values))
```

4.2 Computing the Graph Representation

Elections are usually given as a preference profile P, containing several votes, i.e. rankings of the candidates. From the preference profile of an election the graph representation – strict or weak dominance graph or weighted majority graph – can be derived. This graph representation provides the bases for many other choice rules from computational social choice. Social choice functions using the dominance graph as input are referred to as $C1$-functions and $C2$-functions use the weighted majority graph as input[Fis77]. In this Section a MapReduce algorithm for creating the weighted majority graph from the preference profile is proposed. By a simple postprocessing step the weighted majority

graph can be transformed to the strict or weak dominance graph. Since we are only creating a graph as output of this procedure and the computation itself is not graph based, we are not proposing a Pregel algorithm but only a MapReduce algorithm.

4.2.1 MapReduce Algorithm for transforming the Preference Profile into a Graph Representation

The algorithm for creating the weighted majority graph from a preference profile P is outlined in Algorithm 4.2.1. The output of this algorithm is a list of edges with the corresponding majority margin as weight. This output can further be processed, for example into an adjacency matrix with or without weights, representing the strict or weak dominance relation respectively. This transformation is not explained in more detail in this work.

This straightforward MapReduce algorithm for computing the weighted majority graph needs one map-reduce round, with processes MapVotes and ReduceSum and a last post processing step.

The algorithm works as follows: First the `MapVotes` procedure reads all votes contained in the preference profile P and maps them to key-value pairs. For each vote the procedure emits two key-value pairs for each pair of candidates (a, b) appearing in the order $a \succ b$. The key-value pairs are of the form (key=(a, b), value=1) and (key=(b, a), value=-1).

The reduce task then perform the procedure `ReduceSum` on all received key-value pairs. Each reduce tasks gets key-value pairs of the form (key=(a, b), value= 1 or -1) and computes the sum of the values. The result is the majority margin of a and b, i.e. $\mu_P(a, b)$.

If the strict or weak dominance relation is desired as output, this simple transformation can be included in the postprocessing step.

Algorithm 4.2.1: Computing the weighted majority graph from the preference profile

Input: Preference profile P containing m votes
Output: Edges E_P with weights μ'_P

```
1 MapVotes
2 ReduceSum
3 Postprocessing
```

4.3 Copeland Set

The computation of the Copeland scores and the respective Copeland set (containing the candidates with the highest Copeland scores) is done based on the dominance relation. The calculation of the scores can be performed in a very simple MapReduce round, or by

Function MapVotes(P)

Input: Preference profile P containing m votes
Output: key-value pairs of the form (key=(a,b),value=1) and (key=(b,a),value=-1) for each vote, if candidate a is ranked before candidate b

```
1 foreach vote in P do
2     for i=1 to length(vote)-1 do
3         for j=i+1 to length(vote) do
4             emit(key=[vote[i], vote[j]], value=1);
5             emit(key=[vote[j], vote[i]], value=-1);
6         end
7     end
8 end
```

Function ReduceSum

Input: key-value pairs of the form (key=(a,b),value=1 or -1) produced by MapVotes
Output: Sum of all received values (majority margin)

```
1 return (k, sum(values))
```

using graph functions. Again, there is no Pregel algorithm but there is a graph based procedure. Depending on the distributed computation framework the dominance graph may already be stored in a distributed file system and even simple graph operations are performed in parallel. This depends highly on the system and therefore there a no theoretical performance measures given. In such a graph-based environment the Copeland score of a candidate is then simply the difference of outdegree (number of outgoing edges) and indegree (number of incoming edges), e.g. GraphX has those functions implemented and performs them in parallel depending on the chosen partitioning of the graph.

4.3.1 MapReduce Algorithm for Computing the Copeland Scores

The computation of the Copeland scores takes as input the dominance relation D. With slight abuse of notation we also refer to the adjacency matrix of the dominance graph as D. The overall algorithm is shown in Algorithm 4.3.1. Intuitively, a MapReduce algorithm for computing the Copeland scores creates the key-value pairs (key=a, value=1) and (key=b, value=-1) for each entry $D[a, b] = 1$ in the adjacency matrix of the dominance graph, i.e. if a dominates b. In the algorithm presented in this section this is realized by emitting the whole column or row, where the columns are multiplied by -1.

As input the adjacency matrix of the dominance graph (or simply denoted as the dominance matrix) is used and the procedures `MapRows` and `MapColumns` map the

rows and columns to the corresponding key-value pairs. The notations $D[-, i]$ and $D[i, -]$ refer to the i^{th}-column or i^{th}-row respectively. The columns are multiplied by -1, since they are representing incoming edges. The reducers then simply have to sum up the values to calculate the Copeland score. This is done by using the procedure `ReduceVectorsSum`.

For determining the Copeland set the candidates with the highest Copeland score can be selected in a simple postprocessing phase.

Algorithm 4.3.1: Copeland Scores

Input: Dominance Relation D
Output: Candidates with Copeland scores

```
1 MapRows;
2 MapColumns;
3 ReduceVectorsSum;
```

Algorithm 4.3.2: MapRows()

Input: Dominance Relation D
Output: (key=i, value = D[i,-])

```
1 for i=1 to m do
2 |   emit(key=i, value=D[i, −]);
3 end
```

Algorithm 4.3.3: MapColumns()

Input: Dominance Relation D
Output: (key=i, value = (-1) · D[-,i])

```
1 for i=1 to m do
2 |   emit(key=i, value=(−1) · D[−, i]);
3 end
```

4.4 Smith Set

Recall the definition of the Smith set from Section 2.1. Given a set of candidates A, a preference profile P and the resulting weak dominance graph $D_{\succ} = (A, E_P)$. A candidate a is in the Smith set if and only if for every candidate $b \in A$ there is a path from a to b in the weak dominance graph [BFH09]. A naive algorithm would therefore compute the full transitive closure in order to identify the candidates in the Smith set. However, [BFH09] proof a Theorem that allows us to compute the Smith set by knowing only the paths of length 2 and 3 in the weak dominance graph. In particular, they show that in

Algorithm 4.3.4: ReduceVectorsSum()

Input: key-value pairs: (key= candidate, 1 or -1)
Output: (key=candidate, value = CopelandScore)

```
1 ReduceVectorsSum(key k, list vectors);
2 result = 0;
3 for v in vectors; i=1 to m do
4 |   result = result + v[i];
5 end
6 return (k, result);
```

the weak dominance graph a vertex t is not reachable from a vertex s if and only if there exists a vertex v, with the following properties: (1) paths of length 3 starting from v do not reach any other vertices than the paths of length 2, (2) s is reachable by v on a path of length 2, but t is not. More formally there exists a vertex $v \in A$ such that $D^2_{\succeq}(v) = D^3_{\succeq}(v)$, $s \in D^2_{\succeq}(v)$, and $t \notin D^2_{\succeq}(v)$. Where $D^k_{\succeq}(v)$ denotes all vertices reachable from vertex v by a path of length at most k. All such vertices v and all vertices in the sets $D^2_{\succeq}(v)$ can be excluded from the Smith Set and the remaining vertices in the graph are the winning set – the Smith set. Therefore, it is sufficient to compute paths of length 2 and length 3 to identify the candidates included in the Smith Set. We use this property for the MapReduce algorithm.

For the Pregel algorithm we use the following definition of the Smith Set: The Smith set is the unique undominated strongly connected component in the weak dominance graph [BBH16]. The presented Pregel algorithm uses this property to determine the Smith set by adapting an algorithm for finding strongly connected components.

In this section first the MapReduce algorithm and then the Pregel algorithm for computing the Smith set is described.

4.4.1 MapReduce Algorithm for Computing the Smith Set

For identifying the candidates in the Smith set it is necessary to compute paths of length 2 and 3 in the weak dominance graph, as discussed before (Section 4.4). Therefore, we construct a MapReduce algorithm that computes the reachable candidates iteratively. Intuitively, in each round the MapReduce algorithm combines the known paths ending and starting in each vertex. As stated above we only need to know paths of length at most 3 and therefore we construct a MapReduce algorithm taking three rounds. The result of the MapReduce computation are all candidates which are *not* part of the Smith Set. Therefore a last post-processing step is needed to identify the candidates, which form the Smith set. The overall computation is shown in Algorithm 4.4.1.

For storing information about the vertices we use the data type `VertexWritable`. `VertexWritable` consists of three sets storing the vertices which are known to be have incoming paths to or outgoing paths from the considered vertex v. The sets stored in

Algorithm 4.4.1: Smith Set MapReduce

Input: Weak Dominance Graph $D_{\succeq}$
Output: Smith Set

```
1 FirstMap
2 ReduceVertex
3 MapVertex
4 ReduceVertex
5 MapVertex
6 ReduceComplement
7 ComputeSmithSet
```

`VertexWritable` are updated in each MapReduce round. In particular, the three sets stored in the `VertexWritable` of vertex v are:

- The set *v.new* contains all newly found reachable vertices from v, i.e. all vertices that are reachable on a path of length k.
- The set *v.old* contains all vertices that are reachable from v on a path of length at most k.
- The set *v.reachedBy* contains all vertices known to reach v on a path of length at most k.

The known paths in MapReduce round i have length of at most $k = 2^{i-1}$ and the VertexWritable of v in round i contains the following sets of vertices:

$$v.new = \{a \in A \mid \exists v \mapsto a \text{ of length } k \wedge \nexists v \mapsto a \text{ of length } < k\} \tag{4.1}$$

$$v.old = \{a \in A \mid \exists v \mapsto a \text{ of length } < k\} \tag{4.2}$$

$$v.reachedby = \{a \in A \mid \exists a \mapsto v \text{ of length } \leq k\} \tag{4.3}$$

A path from a to b in the weak dominance graph is denoted as $a \mapsto b$.

Another central data type used in this Algorithm is the type `VertexSetWritable`. This data type consists of a set of vertices plus a *mode*. The mode can take one of the values {'old', 'new', 'reachedBy'}. The data type is used for the values of the produced key-value pairs in the reduce phase.

The whole MapReduce algorithm works as follows. As input the adjacency matrix D of the weak dominance graph is given. The first Map procedure is needed to create the desired data structure from the input data. For the i-th row $D[i, -]$ in the dominance matrix, `FirstMap` emits a pair $(key = i, value = (set, \text{'new'}))$ with $set = \{j \mid D[i, j] = 1\}$. In other words, all vertices reached by an outgoing edge from i are emitted as part of the set in the `VertexSetWritable` with mode = `new`, which is used as the value in the key-value

pair. Additionally `FirstMap` also emits the values $(key = j, value = (\{i\}, \text{'reachedBy'}))$ for each vertex j which has an incoming edge to i in the weak dominance graph. The procedure `ReduceVertex` (shown in Algorithm 4.4.4 combines all received VertexSetWritables to a new VertexWritable. After the FirstMap procedure the input to each reduce task consists of one VertexSetWritable with mode `new` and several VertexSetWritables, each containing only one vertex, with mode `reachedby`. The VertexWritable of vertex v produced by `ReduceVertex` (in this first round) contains the received set with mode='new', the union of all 'reachedBy' values as the set `v.reachedBy` and an empty set as `v.old`. The `ReduceVertex` procedure is shown in Algorithm 4.4.4. After the first MapReduce round (FirstMap + ReduceVertex) each VertexWritable contains the information of all adjacent vertices (candidates).

This first MapReduce round is needed to create the starting values from the dominance graph for the following MapReduce computation. The pseudo code for the `FirstMap` procedure is shown in Algorithm 4.4.2.

Algorithm 4.4.2: FirstMap(D)

Input: Adjacency Matrix D
Output: key-value pairs of the form (key=VertexID,value=VertexSetWritable)

```
1 foreach row with index i in D do
2 |   emit(key=i,value=({j | D[i,j] = 1}, 'new'))
3 |   for j=1 to length(row) do
4 |   |   if row[j]==1 then
5 |   |   |   emit(key=j,value= ({i}, 'reachedBy')
6 |   |   end
7 |   end
8 end
```

Algorithm 4.4.3: MapVertex(VertexWritable)

Input: VertexWritable v of vertex i
Output: key-value pairs of the form (key= VertexID,value=VertexSetWritable)

```
1 emit(i,VertexSetWritable(v.new, mode='old'))
2 emit(i,VertexSetWritable(v.old, mode='old'))
3 emit(i,VertexSetWritable(v.reachedBy,mode='reachedBy'))
4 for r in v.reachedBy do
5 |   emit (r, VertexSetWritable(v.new, mode='new'))
6 end
7 for n in v.new do
8 |   emit(n,VertexSetWritable(v.reachedBy, mode='reachedBy'))
9 end
```

Algorithm 4.4.4: ReduceVertex(key=i,value=VertexSetWritable)

Input: key i, list of VertexSetWritable
Output: VertexWritable

```
1  new = ∅
2  old = ∅
3  reachedBy = ∅
4  for (set, mode) in input-list do
5  |  if mode = 'old' then
6  |  |  old = old ∪ set
7  |  end
8  |  if mode = 'new' then
9  |  |  new = new ∪ set
10 |  end
11 |  if mode = 'reachedBy' then
12 |  |  reachedBy = reachedBy ∪ set
13 |  end
14 end
15 new = new \ old
16 return (i, VertexWritable(old, new, reachedBy))
```

In the following rounds the `FirstMap` procedure is replaced by the `MapVertex` procedure. The `MapVertex` function (shown in Algorithm 4.4.3) receives the `VertexWritables` as input and produces new key-value pairs. The `VertexWritable` v is mapped to the following key-value pairs:

- The vertices previously stored in the set `v.new` are emitted with mode `'old'`, such that the `ReduceVertex` function knows, that they have previously been known to be reached by v, i.e. there is a path of length $< k$. Those vertices are going to be part of the set `v.old` in the output of the following `ReduceVertex` procedure.
- The vertices stored in `v.old` are going to stay in `v.old` in the next round. So `v.old` is emitted with mode 'old'.
- For each vertex in `v.new` a `VertexSetWritable` with the set `v.reachedBy` and mode 'reachedBy' is emitted.
- For each vertex in `v.reachedBy` a `VertexSetWritable` with the set `v.new` with mode 'new' is emitted.

The purpose of the MapReduce procedures is to combine the known incoming and outgoing paths of each vertex to new (longer) paths; i.e. the known paths in round i have length $\leq 2^{i-1}$.

`ReduceVertex` (Algorithm 4.4.4) receives values of type `VertexSetWritable` for a given vertex v and calculates the new `VertexWritable` data structure for v: the union of all input sets with mode 'reachedBy' is assigned to the set *reachedBy*. The union of the input sets with mode 'old' is assigned to the set *old*. The set *new* should only contain vertices newly found in this round to be reachable by v. Therefore, we first combine all input sets with mode 'new' and then set *new* to *new* $\setminus$ *old*.

After the last call of `MapVertex` in Algorithm 4.4.1 the information of all paths of length at most 4 is stored at each vertex. We can use this information to exclude all vertices v where $v.reachedBy$ contains vertices not contained in $v.new \cup v.old$. The remaining vertices are part of the SmithSet. This post processing can be done by using the procedures `ReduceComplement` and `ComputeSmithSet`. Thus by doing this we checked whether $D^2_{\succeq}(v) = D^4_{\succeq}(v)$ holds, which is equivalent to checking $D^2_{\succeq}(v) = D^3_{\succeq}(v)$. Recall the Theorem mentioned at the beginning of the section – if we find such a vertex with $D^2_{\succeq}(v) = D^3_{\succeq}(v)$ and $D^2_{\succeq}(v) \subset A$ we can exclude v and all vertices reachable by v from the possible winners, i.e. we exclude v and $v.new \cup v.old$ from the Smith Set. Thus, the `ReduceComplement` function returns vertices which are *not* in the Smith Set – the complement of the Smith set. In the last procedure `ComputeSmithSet` the Smith set is derived using this information.

MapReduce Algorithm for computing the Smith Set by Example

Example 4 *(Example 2 continued.)*

An election with four candidates $\{a, b, c, d\}$ *and a preference profile* P *with six votes is given. The preference profile* P *is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

This preference profile P *results in the weak dominance graph shown in Figure 4.1.*

The output values produced in each step are shown in Table 4.1. The entries in '(2) ReduceVertex' are the output of the first ReduceVertex call after FirstMap. Therefore this row contains the starting values of each vertex; i.e. the sets `new` *and* `reachedBy` *contain only adjacent vertices. '(3) MapVertex' shows all key-value pairs produced in the following map process. The key-value pairs are of the form (key=destination, value=(Set,mode)). The key-value pairs containing the information that at vertex can reach itself, or is reached by itself are omitted, e.g. the key-value pair (b, ({b}, 'rB')) is left out. In the shuffle phase the key-value pairs are then grouped by their key (destination) vertex and the values are processed in the reduce phase. The output resulting from the reduce phase is shown in '(4) ReduceVertex', i.e. the new* `VertexSetWritables` *with the updated information.*

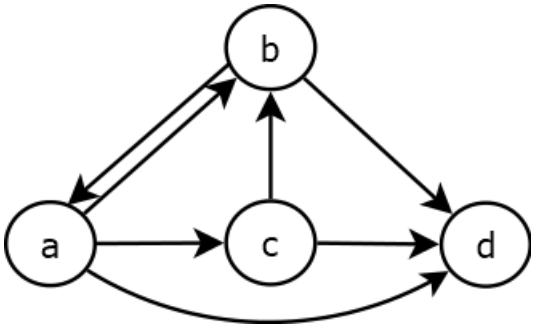


Figure 4.1: Weak Dominance Graph for Example 4

At this point the information about all paths ≤ 2 is stored in the `VertexWritables`*. The steps '(5) MapVertex' and '(6) ReduceVertex' are part of the last MapReduce round. The resulting* `VertexSetWritables` *in '(6) ReduceVertex' contain all paths of length ≤ 4. It remains to perform the post-processing step, where we check for each vertex v if (1) v.old contains all other vertices and (2) if v.new is empty. If v.new is empty, this means that $D_{\succ}(v)^2 = D_{\succ}(v)^4$. We identify d as the only vertex, where this is the case. Thus we exclude d from the SmithSet and if there where any vertices contained in old, we would exclude them too. The remaining vertices {a,b,c} form the Smith Set.* ◇

Step	a	b	c	d
(2) ReduceVertex	new = $\{b, c, d\}$ old = $\emptyset$ rB = $\{b\}$	new =$\{a, d\}$ old = $\emptyset$ rB = $\{a, c\}$	new = $\{b, c\}$ old = $\emptyset$ rB = $\{a\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$
(3) MapVertex	(a,($\{b, c, d\}$,'old')) (a,({b},'rB')) (c, ({b},'rB')) (d, ({b},'rB')) (b, ({b,c,d},'new'))	(b,($\{a, d\}$,'old')) (b,({a,c},'rB')) (a, ({a,c},'rB')) (d, ({a,c},'rB')) (A, ({a,d},'new')) (c, ({a,d},'new'))	(c,($\{b, c\}$,'old') (c,({a},'rB')) (b, ({a},'rB')) (d, ({a},'rB')) (a, ({b,d},'new'))	(d,($\{a, b, c\}$,'rB')
(4) ReduceVertex	new = $\emptyset$ old = {b,c,d} rB = {b,c}	new = $\{c\}$ old = $\{a, d\}$ rB = $\{a, c\}$	new = $\{a\}$ old = $\{b, d\}$ rB = $\{a, b\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$
(5) MapVertex	(a,({b,c,d},'old')) (a,({b,c},'rB'))	(b,({a,d,c},'old')) (b,({a,c},'rB')) (c, ({a,c},'rB')) (a, ({c},'new'))	(c,({a,b,d},'old')) (c,({a,b},'rB')) (a, ({a,b},'rB')) (b, ({a},'new'))	(d,({a,b,c},'rB'))
(6) ReduceVertex	new = $\emptyset$ old = {b,c,d} rB = {b,c}	new =$\emptyset$ old = $\{a, c, d\}$ rB = $\{a, c\}$	new = $\emptyset$ old = $\{a, b, d\}$ rB = $\{a, b\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$

Table 4.1: Computing the Smith Set by Example.

4.4.2 Pregel Algorithm for computing the Smith Set

For the Pregel algorithm for computing the Smith set we adapt an algorithm for finding strongly connected components presented in [YCX+14]. The algorithm for finding strongly connected components is based on propagating the *id* of each vertex in forward and backward direction along the edges in the graph. Each vertex then only remembers the smallest number it received for each direction. This procedure is called forward and backward min label propagation. This results in the situation that the vertices contained in the same strongly connected component are labelled with the same forward and backward min-labels. We adapt this algorithm for our purpose to identify the unique undominated strongly connected component in the weak dominance graph of an election, i.e. the Smith set.

As input the weak dominance graph $D_{\succ}$ is given. Each vertex has as vertex id v_{id} and the forward and backward labels (s, t) together with a *status* as value. The label s (source) is the forward min-label and t (target) the backward min-label. The *status* can be *unknown*, *notSmith* or *Smith*. At the beginning of the algorithm the *status* of each vertex is initialized with *unknown*. The label s stands for the smallest id of a source of a path that reaches v in the weak dominance graph. Similarly t stands for the smallest id of all vertices (targets) that can be reached by v. To give some intuition of the algorithm: For example after the forward and backward min-label propagation the labels $(s = 2, t = 1)$ of a vertex v mean that the vertex with the smallest *id* that can reach v is the vertex with id 2 and the vertex with the smallest *id* reachable from v has id 1. From this follows, that the vertex with $id = 1$ cannot reach v, because otherwise the forward level of v was equal to 1 too. From this we know that the vertex with $id = 1$ is dominated by v and we know that the vertex with $id = 1$ cannot be part of the Smith Set. So we set the status of this vertex to *notSmith*. We also know that all vertices that are reachable by the vertex with id=1 are not part of the Smith set either (because they are dominated by v too). Those vertices can be easily identified, because they have the forward label $s = 1$.

For a vertex v with the forward and backward min-labels (s, t) we distinguish the following cases:

- If $s < t$ we know that the there is a path from s to v but no path from v to s, i.e. s strictly dominates v. Therefore v can be excluded and the status of v is set to `notSmith`.
- If $s > t$: there exists a path from v to t but there is no path from t to v. Therefore t and all vertices with forward label t are strictly dominated by v and can be excluded, i.e. their status is set to `notSmith`.
- $s == t$: the vertex is still a possible candidate for the Smith set and its status remains `unknown`.

Further inference by the labels can be done by looking at adjacent vertices. The vertices u and v are connected by the edge $u \to v$ in the weak dominance graph and have the labels $(u.s, u.t)$ and $(v.s, v.t)$. The following cases can be distinguished:

- $u.s < u.t$: u and v are both excluded from the Smith Set (see case above).
- $u.s = v.s \wedge u.t = v.t$: the vertices are both candidates for the Smith set and their status remains `unknown`.
- $u.s > v.s$: vertex $v.s$ can reach v but not u. Therefore v, $v.s$ and all other vertices with forward label $v.s$ cannot be in the Smith Set and their status is set to `notSmith`.
- $u.s < v.s$: not possible, since there is an edge (u, v).
- $u.t < v.t$: there is a path from u to t but v cannot reach t, therefore v is excluded from the Smith Set. Further all other vertices with backward-label $v.t$ are excluded.
- $u.t > v.t$: not possible, since there is an edge (u, v).

Following from these observations we formulate the following algorithm outlined in Algorithm 4.4.1. At the beginning of the computation all vertices are initialized with status='unknown' and their id as forward and backward label $(s = id, t = id)$.

In the while-loop the preprocessing procedure sets the labels of vertices that are already known to be `notSmith` to $(s = \infty, t = \infty)$ (in the first iteration there are no vertices with status `notSmith`). This labelling causes the vertices to act as gateways for the labels of other vertices, since they do not propagate their own id as forward or backward label. In this way only 'unknown' vertices are checked, whether they are in the Smith set. In the `Preprocessing` procedure all vertices with status 'unknown' send their labels to the adjacent vertices in the messages ('forward',s) in outgoing direction and ('backward',t) in incoming direction. Those are the first messages sent in the Pregel computation. The `Forward-Backward-Propagation` realizes the forward and backward min-label propagation and the (s, t) labels of the vertices under the current initialization are computed. This procedure is the actual 'pregel heart' of the computation and is shown in Algorithm 4.4.3. If a vertex receives a message with a forward or backward label smaller than its current label, it updates its label. Vertices only send messages if their label has changed. Otherwise the status is set to 'inactive'. Recall, that Pregel algorithms stop as soon as all vertices are 'inactive'. This happens as soon as the forward and backward labels converge to their final value under the current initialization (the (s,t) values set in the preprocessing). The `post processing` (Algorithm 4.4.4) then checks all labels and changes the status of a vertex, if it can be excluded from the possible candidates of the Smith set. The following cases are checked:

- mark all vertices v with $v.s < v.t$ as not in Smith Set, i.e. set $v.status = notSmith$.

- if $\exists v : v.s > v.t$: mark vertex t and all vertices with forward label t as not in Smith Set, i.e. $t.status = notSmith$ and $\forall u \in A$ with $u.t = t$ set $u.status = notSmith$.

- for each edge (u, v) in the graph check if $u.s > v.s$. If this is the case then vertex $v.s$ can reach v but not u. Therefore v, $v.s$ and all other vertices with forward label $v.s$ cannot be in the Smith Set (set their status to `notSmith`).

- for each edge (u, v) in the graph check if $u.t < v.t$. If this is the case then vertex u can reach $u.t$ but v cannot reach $u.t$. Therefore v, $v.t$ and all other vertices with backward label $v.t$ cannot be in the Smith Set (set their status to `notSmith`).

The only case when no status of a vertex is changed, is when all labels (backward and forward) of the vertices with status *unknown* are identical. From this follows that only one single SCC is left in the computation and this SCC is the unique undominated SCC, i.e. the Smith Set. So we can select all remaining 'unknown' vertices as the Smith Set.

Algorithm 4.4.1: SmithSet($D_{\geq}$)

```
1 Initialisation-of-vertices;
2 while the number of vertices with status = 'unknown' changes do
3 |  PreProcessing;
4 |  Forward-Backward-Propagation;
5 |  PostProcessing;
6 end
7 Output vertices with status 'unknown';
```

Algorithm 4.4.2: PreProcessing()

```
1  if v.status = 'unknown' then
2  |  s = v.id;
3  |  t = v.id;
4  |  foreach outgoing edge (v, u) do
5  |  |  send ('forward', s) to vertex u;
6  |  end
7  |  foreach incoming edge (u, v) do
8  |  |  send ('backward', t) to vertex u;
9  |  end
10 else
11 |  s = ∞;
12 |  t = ∞;
13 end
```

Algorithm 4.4.3: Forward-Backward-Propagation

```
1  foreach received value (d, i) do
2  |  if d = 'forward' ∧ i < s then
3  |  |  s = i;
4  |  else if d = 'backward' ∧ i < t then
5  |  |  t = i;
6  end
7  if s has changed then
8  |  foreach outgoing edge (v, u) do
9  |  |  send ('forward', s) to vertex u;
10 |  end
11 end
12 if t has changed then
13 |  foreach incoming edge (u, v) do
14 |  |  send ('backward', t) to vertex u;
15 |  end
16 end
17 if s and t remain unchanged then
18 |  set v inactive;
19 end
```

Pregel Algorithm for Computing the Smith Set by Example

Example 5 *(Example 4 continued.) An election with four candidates $\{a, b, c, d\}$ and a preference profile P with six votes is given. The preference profile P is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

This preference profile results in the weak dominance graph shown in Figure 4.1.

Initialization and Preprocessing: *All vertices start with status unknown and the s and t labels are initialized with the id of the corresponding vertex. In the first row of Table 4.2 the ids can be observed.*

Forward Min-Label Propagation: *Recall that a pregel procedure always works in supersteps, where each superstep consists of two steps: (1) the active vertices send messages to each other and (2) the vertices process the received data. In this Pregel procedure each active vertex sends its s label along the outgoing edges to the adjacent vertices. Then each vertex updates its label with the minimum s label it received. If a vertex did not change its label in such a step it gets inactive and the computation stops*

Algorithm 4.4.4: PostProcessing for vertex v

```
1  if v.s < v.t then
2  |  v.status = 'notSmith';
3  else if v.s > v.t then
4  |  set status of vertex with id v.t to 'notSmith';
5  foreach incoming edge (u, v) do
6  |  get (u.s, u.t) from vertex u;
7  |  if u.s > v.s then
8  |  |  v.status = 'notSmith';
9  |  |  foreach vertex a with a.s = v.s do
10 |  |  |  a.status = 'notSmith';
11 |  |  end
12 |  end
13 |  if u.t < v.t then
14 |  |  v.status = 'notSmith';
15 |  |  foreach vertex a with a.t = v.t do
16 |  |  |  a.status = 'notSmith';
17 |  |  end
18 |  end
19 end
```

when all vertices are inactive. Since s has the smallest label (1) and it can reach all of the other vertices, all s labels are set to 1 *after only one step (as can be observed in Table 4.2).*

Superstep	a	b	c	d
Init	1	2	3	4
(1)	1	1	1	1

Table 4.2: Forward (s) labels in Example 5

Backward Min-Label Propagation: *The pregel procedure starts with each vertex sending each t label in backward direction to the neighbours. Then each vertex sets its t label to the minimum t value it received. Let us inspect vertex b in more detail: In the first superstep vertex b sends its label ($t = 2$) to vertex c. It further receives two messages - from vertex a it receives label* 1 *and from vertex D it receives label* 4*. Thus vertex b updates its t label to* 1*.*

Vertex c received the labels 2 *and* 4 *from vertices b and d respectively and changes its label to* 2*. In the next step vertex b passes his label ($t = 1$) on to vertex c.*

Vertex d did not receive any messages in the backward propagation, because it has no outgoing edges.

Superstep	a	b	c	d
Init	1	2	3	4
(1)	1	1	2	4
(2)	1	1	1	4

Table 4.3: Backward (t) labels in Example 5

Postprocessing. *Now we check the s and t labels to identify candidates, that cannot be in the Smith set. Vertices a, b and c have the same labels ($s = 1, t = 1$ and no further reasoning can be done on their status. Vertex d has the labels $s = 1$ and $t = 4$. Therefore we know that vertex a can reach d, but there is no path in the other direction. The status of d is therefore changed to notSmith.*

Start of the next iteration of the while-loop.

Preprocessing: *The vertices $\{a, b, c\}$ are initialized with their ids as s and t labels, since their status is still 'unknown'. Vertex d is initialized with the labels ($s = \infty, t = \infty$), since it is already excluded from the possible Schulze winners.*

Forward and Backward Min-Label propagation. *In the beginning only vertices a, b and c are sending initial messages to their neighbours. Again the labels are propagated and after this procedure the vertices a, b and c have the label ($s = 1, t = 1$). Vertex d has the label ($s = 1, t = \infty$).*

Postprocessing. *In the post processing no status is changed, since d was already known to not be in the Smith set.*

Finished. *Since no status was changed to computation stops and the Smith set is $\{a, b, c\}$.* ◇

4.5 Schwartz Set

Recall the definition of the Schwartz Set given in Section 2.1.2. Given a preference profile P and the corresponding strict dominance graph $D_{\succ}$, candidate a is in the Schwartz set if and only if for every candidate b there is a path from a to b, whenever there is a path from b to a [BFH09, Lemma 4.5]. For the Smith Set we could make use of a result obtained by [BFH09] to only compute paths of length 3. For the Schwartz set this optimization is not possible and we have to compute the whole transitive closure.

4.5.1 MapReduce Algorithm for computing the Schwartz Set

The algorithm for computing the Schwartz Set follows the same principles as the MapReduce algorithm for computing the Smith set (Section 4.4), but there are several important differences. First of all, for the Schwartz set it is necessary to compute the whole transitive closure, whereas for the Smith set the paths of length 3 where sufficient and further the post processing differs.

The overall structure of the Algorithm is given in Algorithm 4.5.1. In contrast to the algorithm for computing the Smith set, the strict dominance graph is used as input. The procedures `FirstMap` (Algorithm 4.4.2), `ReduceVertex` (Algorithm 4.4.4) and `MapVertex` (Algorithm 4.4.3 are the same procedures as used for computing the Smith set.

The central data types in this algorithm are again the `VertexWritable` and `VertexSetWritable`. Recall that a `VertexWritable` consists of three sets of vertices storing information on incoming and outgoing paths for a given vertex v as follows:

- the set *old* stores all vertices that have been found previously to be reachable from v;
- the set *new* stores all vertices that have been found in the last map-reduce round to be reachable from v;
- the set *reachedBy* stores all vertices known to reach v;

The data type `VertexSetWritable` contains sets of vertices together with a mode. The mode can take one of the values 'old', 'new', and 'reachedBy'.

The function `ReduceVertex` always outputs all vertices as `VertexWritables` and this data is used as input to the next MapReduce round; or to the `PostProcessing` step.

The starting values for the `VertexWritables` are created by the procedures `FirstMap` and `ReduceVertex`. After execution of those procedures on the adjacency matrix of the strict dominance matrix, the `VertexWritables` of each vertex contain only the information of directly adjacent vertices, i.e. paths of length 1.

The while-loop then iterates until all vertices converge, i.e. the set $v.new$ is empty for each vertex v. At this point the full transitive closure of the strict dominance graph is stored in the `VertexWritables`. In the post processing step we only have to compare the set $v.old$ and $v.reachedby$ to determine the vertices contained in the Schwartz set. In particular, a vertex v is contained in the Schwartz set if the set of vertices reachable by v ($v.old$) is a superset of the set of vertices that reach v ($v.reachedBy$); i.e., we have to check if $v.reachedby \subseteq v.old$ holds.

Example 6 *(`Example 4 continued`) An election with four candidates $\{a, b, c, d\}$ and a preference profile P with six votes is given. The preference profile P is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

Algorithm 4.5.1: Schwartz Set

Input: Strict Dominance Graph $D_{\succ}$
Output: Schwartz Set

```
1 FirstMap
2 ReduceVertex
3 while there exists a vertex with new ≠ ∅ do
4 |   MapVertex
5 |   ReduceVertex
6 end
7 ComputeSchwartzSet
```

This preference profile P results in the strict dominance graph shown in Figure 4.2. The output values of each procedure in the whole MapReduce computation are shown in Table 4.4. The entries in the section '(2) ReduceVertex' show the result of the first MapReduce round. At this point only the adjacent vertices are saved in the sets 'reachedBy' and 'new'. In the following MapReduce round the process '(3) MapVertex' outputs new key-value pairs. For vertex a the only key-value pair (a,({c,d},'old')) is emitted. For vertex b more is happening: first the set 'new'={b} is emitted with label 'old' and the the set 'rB' is processed, as follows: From the information stored in the sets 'b.new' and 'b.reachedBy' we know that there is an outgoing path from b to d and an incoming path from c to b. By emitting the key-value pairs (d,({c},'rB')) and (c,({d},'new') those paths are going to get connected in the following reduce phase. The output of the reduce phase is shown in '(4) ReduceVertex'. The whole computation stops as soon as all sets 'new' are empty. In the post processing only the condition reachedBy ⊆ old has to be checked. If this condition holds, that vertex is part of the Schwartz set. Therefore we have only one vertex in the Schwartz Set, namely vertex a. ◇

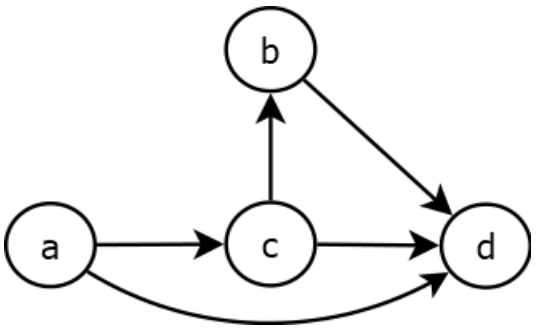


Figure 4.2: Strict Dominance Graph for Example 6

4.5.2 Pregel algorithm for computing the Schwartz Set

The Schwartz set is defined as the union of all minimal undominated sets of vertices in the strict dominance graph [BFH09]. If all strongly connected components (SCCs)

Step	a	b	c	d
(2) ReduceVertex	new = $\{c, d\}$ old = $\emptyset$ rB = $\emptyset$	new =$\{d\}$ old = $\emptyset$ rB = $\{c\}$	new = $\{b, d\}$ old = $\emptyset$ rB = $\{a\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$
(3) MapVertex	(a, ($\{c, d\}$,'old'))	(b, ($\{d\}$,'old')) (b, ($\{c\}$,'rB')) (d, ($\{c\}$,'rB')) (c, ($\{$d$\}$,'new'))	(c, ($\{b, d\}$,'old')) (c, ($\{a\}$,'rB')) (b, ($\{$a$\}$,'rB')) (d, ($\{$a$\}$,'rB')) (a, ($\{$b,d$\}$,'new'))	(d, ($\{a, b, c\}$,'rB'))
(4) ReduceVertex	new = $\{b\}$ old = {c,d} rB = $\emptyset$	new = $\emptyset$ old = $\{d\}$ rB = $\{a, c\}$	new = $\emptyset$ old = $\{b, d\}$ rB = $\{a\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$
(5) MapVertex	(a,({b,c,d},'old'))	(b,({d},'old')) (b,({a,c},'rB'))	(c,({b,d},'old')) (c,({a},'rB'))	(d,({a,b,c},'rB'))
(6) ReduceVertex	new = $\emptyset$ old = {b,c,d} rB = $\emptyset$	new =$\emptyset$ old = $\{d\}$ rB = $\{a, c\}$	new = $\emptyset$ old = $\{b, d\}$ rB = $\{a\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{a, b, c\}$
(7) $rB \subseteq old$	`true`	`false`	`false`	`false`

Table 4.4: Computing the Schwartz Set by Example

are known, the Schwartz Set is easily found by forming the union of all undominated SCCs. Note, that in contrast to the Smith Set, the Schwartz Set is based on the strict dominance graph. In the strict dominance graph several undominated strongly connected components can occur, whereas in the weak dominance graph the undominated scc is always unique.

We use the same basic ideas as for the pregel algorithm for computing the Smith Set in Section 4.4.2. But there are very important differences in the post processing step, where it has to be considered that it is possible to have several undominated SCCs in the strict dominance graph.

Each vertex stores its vertex id v_{id} and the pair (s, t) and a status as value, where s is the forward min-label and t the backward min-label. Further, the status of a vertex can be 'unknown', 'Schwartz' or 'notSchwartz'. All vertices are initialized with status 'unknown'. In the while-loop the preprocessing procedure sets the labels of vertices that are already known to be `notSchwartz` to $(s = \infty, t = \infty)$, all vertices with 'unknown' are initialized with their *id*s as s and t label. In the first iteration all vertices have status 'unknown'. The forward and backward min-label propagation form the heart of the algorithm. The forward min-label s corresponds to the smallest *id* of all vertices, which are known to reach v, i.e. there exists a path from source s to v. The backward min-label t is the smallest *id* of all vertices that are reachable by v, i.e. there is a path from v to target t.

In the `PostProcessing` step (illustrated in Algorithm 4.5.1) vertices which cannot be in the Schwartz set are identified as follows:

- mark all vertices v with $v.s < v.t$ as not in the Schwartz Set, i.e. set $v.status =$

notSchwartz.

- if $\exists v : v.s > v.t$: mark vertex t and all vertices with backward label t as not in Schwartz Set, i.e. $t.status = notSchwartz$ and $\forall u \in A$ with $u.s = t$ set $u.status = notSchwartz$.

Algorithm 4.5.1: Post processing for vertex v

```
1 if v.s < v.t then
2 |  v.status = 'notSchwartz';
3 else if v.s > v.t then
4 |  set status of vertex v.t to 'notSchwartz';
5 |  foreach vertex u with u.s = v.t do
6 |  |  u.status = 'notSchwartz';
7 |  end
```

The while-loop continues as long as the number of vertices with status 'unknown' decreases. All remaining vertices with status 'unknown' are then selected as the Schwartz Set.

Pregel algorithm for Computing the Schwartz Set by Example

Example 7 *(Example 6 continued.) An election with four candidates $\{a, b, c, d\}$ and a preference profile P with six votes is given. The preference profile P is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

Initialization: *all vertices start with status unknown and the s and t labels are initialized with the id of the corresponding vertex.*

Forward Min-Label Propagation: *Recall that a pregel procedure always works in supersteps, where each superstep consists of two steps: (1) the active vertices send messages to each other and (2) the vertices process the received data. In this Pregel procedure each active vertex sends its s label along the outgoing edges to the neighbours. Then each vertex updates its label with the minimum s label it received. If a vertex did not change its label in such a step it gets inactive and the computation stops when all vertices are inactive. Since a has the smallest label (1) and it can reach all of the other vertices, all s labels are set to 1 after only two steps(as can be observed in Table 4.5).*

Backward Min-Label Propagation: *The pregel procedure starts with each vertex sending each t label in backward direction to the neighbours. Then each vertex sets its*

Superstep	a	b	c	d
Init	1	2	3	4
(1)	1	2	1	1
(2)	1	1	1	1

Table 4.5: Forward labels (s) in Example 7

t label to the minimum t value it received. In the first superstep vertex b sends it label ($t = 2$) to vertex c. It further receives one message from vertex d with label 4. Thus vertex b keeps its t label as 2, but c updates its label to 2.

Vertex c received the labels 2 and 4 from vertices b and d respectively and changes its label to 2. In the next step vertex b passes his label ($t = 1$) on to vertex c. On the other hand vertex d never received a message, because it has no outgoing edges.

Superstep	a	b	c	d
Init	1	2	3	4
(1)	1	2	2	4

Table 4.6: Backward labels (t) in Example 7

Post processing. *Now we check the s and t labels to identify candidates, that cannot be in the Schwartz set. For vertices b, c and d we observe that they have labels $s < t$. Therefore they are dominated by a vertex and are excluded from the Schwartz set. Candidate a is the only vertex remaining and therefore the only candidate in the Schwartz set.* ◇

4.6 Schulze Method

The Schulze Method is a method for preference aggregation. It is a so-called $C2$ function since it is using the weighted majority graph $\mathcal{W}_P$ as input. For the MapReduce algorithm we represent the weighted majority graph by the adjacency matrix M, with the majority margin as values. The entry $M[i,j] = k$ in the matrix would mean that the edge (i,j) in the weighted majority graph $\mathcal{W}_P$ has weight k and equivalently that the majority margin of i and j is k, i.e., $\mu_P(i,j) = k$.

4.6.1 MapReduce Algorithm for Computing the Schulze Winner

To find the Schulze Winner it is necessary to know the widest paths connecting all vertices in the weighted majority graph $\mathcal{W}_P$. For the computation of the Schwartz set we proposed an Algorithm in Section 4.5, which computes the full transitive closure in the strict dominance graph as an intermediate result. For the Schulze Winner the transitive closure alone is not sufficient, but we need it together with the width of the widest path connecting the vertices. We can adapt the Schwartz Set algorithm in order to find all widest paths in the weighted majority graph.

In this MapReduce algorithm we use the data types `WidestPathsWritables` and `WidestPathsSetWritables`. The data type `WidestPathsWritables` stores the targets and sources of all incoming and outgoing known widest paths, together with the width in the sets *new*, *old* and *reachedBy*. The information about outgoing widest paths is stored in the sets *new* and *old*; the set *new* only contains newly found widest paths and the set *old* contains widest paths that have already been found in a previous round. The incoming widest paths is stored in the set *reachedBy*. We do not store the whole path, but the *id* of the source or target vertex is stored, together with the maximum known weight of all paths connecting these vertices.

The `WidestPathsWritable` of vertex v therefore consists of the following three sets:

- the set *old* stores all ids of vertices, that have been found previously to be reachable from v, together with the width of the widest known path;
- the set *new* stores all ids of vertices that were found in the last map-reduce round to be reachable from v, together with the width of the path. Further, if a wider path to a vertex is found, then the corresponding vertex id and the weight are stored in this set too.
- the set *reachedBy* stores all vertices known to reach a, together with the width of the widest known path;

The data type `WidestPathsSetWritables` contains a *set* of widest paths together with a mode. The mode is either 'old', 'new', or 'reachedBy'. The *set* of widest paths is a set of vertex ids together with the weight of the found paths, i.e. a set of tuples of the form (id,weight).

The overall algorithm is outlined in Algorithm 4.6.1.

After the procedure `FirstWidthMap` the input to each reduce task consists of one WidestPathSetWritable with mode `new` and several WidestPathSetWritables, each containing only one vertex, with mode `reachedBy`. The procedure `ReduceWidestPath` (shown in Algorithm 4.6.4 combines all received WidestPathSetWritables to a new WidestPathWritable. The WidestPathWritable produced by `ReduceVertex` (in this first round) contains the received set with mode=*new*, the union of all *reachedBy* values as the set `reachedBy` and an empty set as `old`. The information contained in these sets after this first round is equivalent to all outgoing and incoming edges in the weighted majority graph. Therefore only 'widest' paths of length 1 are known at this point. The `ReduceVertex` procedure is shown in Algorithm 4.4.4.

The operators $\cup_w$ and $\setminus_w$ in the `ReduceWidthVertex` procedure consider the vertex labels together with their width, i.e. tuples (id, w). The operator $\cup_w$ only keeps one entry per vertex id, that is the entry with the maximum width. The operation $new \setminus_w old$ causes the result to only include vertices (together with the width of the newly found path) if there is no stronger path contained in *old*, i.e. $new \setminus_w old = \{(n, n.w) \in new \mid \nexists (n, w) \in old \text{ with } w \geq n.w\}$.

The procedure `MapWidestPaths` combines the widest incoming and the widest outgoing paths of each vertex and emits the corresponding key-value pairs.

The while-loop continues until there is no new wider path found, i.e. the set *new* of each vertex is empty. At this point we know all widest paths in the weighted majority graph. In the post processing step (`FindSchulzeWinner`) we only have to compare the set *old* and *reachedBy* to determine if all outgoing paths are wider than the incoming paths. If this is the case, the corresponding vertex is a Schulze Winner.

Algorithm 4.6.1: Schulze Winner

Input: Adj. Matrix M of the weighted majority graph
Output: Schulze Winner

```
1 FirstWidthMap
2 ReduceWidestPaths
3 while there exists a vertex with new ≠ ∅ do
4 |   MapWidestPaths
5 |   ReduceWidestPaths
6 end
7 FindSchulzeWinner
```

Algorithm 4.6.2: FirstWidthMap(M)

Input: Adjacency Matrix M of the weighted majority graph
Output: key-value pairs of the form (VertexID,WidestPathsSetWritable)

```
1 foreach row with index i in M do
2 |   emit(key=i,value=({(j, row[j]) | row[j] > 0}, 'new'))
3 |   for j=1 to length(row) do
4 |   |   if row[j]>0 then
5 |   |   |   emit(key=j,value= ({(i, row[j])}, 'reachedBy')
6 |   |   end
7 |   end
8 end
```

Algorithm 4.6.3: MapWidestPaths(WidestPathsWritable)

Input: WidestPathsWritable v
Output: key-value pairs of the form (VertexID,WidestPathsSetWritable)

```
1 emit(key=i,value=(v.new, mode='old'))
2 emit(key=i,value=(v.old, mode='old'))
3 emit(key=i,value=(v.reachedBy,mode='reachedBy'))
4 for (r,r_w) in v.reachedBy do
5 |   for (n,n_w) in v.new do
6 |   |   emit (key=r, value=((n,min(n_w, r_w)), mode='new'))
7 |   |   emit(key=n,value=((r,min(n_w, r_w)),mode='reachedBy'))
8 |   end
9 end
```

Example: MapReduce Algorithm for Computing the Schulze Winner

Example 8 *(Example 4 continued) An election with four candidates* $\{a, b, c, d\}$ *and a preference profile* P *with six votes is given. The preference profile* P *is as follows:*

$$\begin{aligned} P = \{&a \succ b \succ d \succ c,\ b \succ a \succ c \succ d,\\ &a \succ c \succ d \succ b,\ c \succ b \succ d \succ a,\\ &a \succ c \succ b \succ d,\ c \succ b \succ a \succ d\} \end{aligned}$$

This preference profile P *results in the weighted majority graph shown in Figure 4.3. The output values of each procedure in the whole MapReduce computation are shown in Table 4.7. The entries in the section '(2) ReduceVertex' show the result of the first MapReduce round. At this point only the adjacent edges together with the weights are saved in the sets 'reachedBy' and 'new'. In each round also the set 'reachedBy' and 'old'*

Algorithm 4.6.4: ReduceWidestPaths(key=i,value=VertexSetWritable)

Input: key i, list of WidestPathsSetWritable
Output: WidestPathWritable

```
1  new = ∅
2  old = ∅
3  reachedBy = ∅
4  for (set, mode) in input-list do
5  |  if mode = 'old' then
6  |  |  old = old ∪_w set
7  |  end
8  |  if mode = 'new' then
9  |  |  new = new ∪_w set
10 |  end
11 |  if mode = 'reachedBy' then
12 |  |  reachedBy = reachedBy ∪_w set
13 |  end
14 end
15 new = new \_w old
16 return (i, VertexWidthWritable(old, new, reachedBy))
```

are emitted by each vertex. Due to space reasons this is omitted in Table 4.7. The whole computation stops as soon as all sets 'new' are empty. In the postprocessing only path widths saved in the sets 'old' and 'reachedBy' have to be compared. In (6) ReduceVertex in Table 4.7 we can observe, that vertex b is reached by a on a widest path with weight 2*, but b cannot reach a. Therefore, b is not a Schulze Winner. The only vertex, where all outgoing paths are stronger than the incoming paths is vertex a and therefore a is the unique Schulze Winner in the example.* ◇

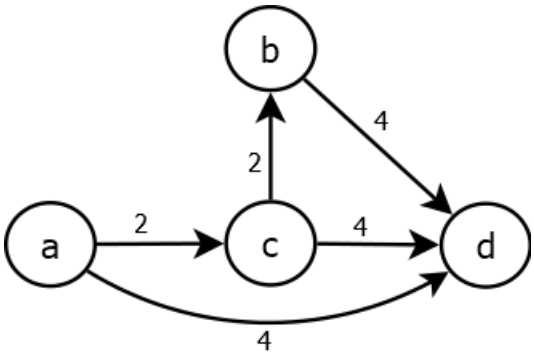


Figure 4.3: Weighted Majority Graph for Example 8

Step	a	b	c	d
(2) ReduceVertex	new = $\{(c,2),(d,4)\}$ old = $\emptyset$ rB = $\emptyset$	new =$\{(d,4)\}$ old = $\emptyset$ rB = $\{(c,2)\}$	new = $\{(b,2),(d,4)\}$ old = $\emptyset$ rB = $\{(a,2)\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{(a,4),(b,4),(c,4)\}$
(3) MapVertex	(a, ($\{(c,2),(d,4)\}$,'old'))	(b, ($\{(d,4)\}$,'old')) (b, ({(c,2)},'rB')) (d, ({(c,2)},'rB')) (c, ({(d,2)},'new'))	(c, ($\{(b,2),(d,4)\}$,'old')) (c, ({(a,2)},'rB')) (b, ({(a,2)},'rB')) (d, ({(a,2)},'rB')) (a, ({(b,2),(d,2)},'new'))	(d,($\{(a,4),(b,4),(c,4)\}$,'rB'))
(4) ReduceVertex	new = $\{(b,2)\}$ old = {(c,2),(d,4)} rB = $\emptyset$	new =$\emptyset$ old = $\{(d,4)\}$ rB = $\{(a,2),(c,2)\}$	new = $\emptyset$ old = $\{(b,2),(d,4)\}$ rB = $\{(a,2)\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{(a,4),(b,4),(c,4)\}$
(5) MapVertex	(a,({(c,2),(d,4),(b,2)},'old'))	(b,($\{(d,4)\}$,'old')) (b,($\{(a,2),(c,2)\}$,'rB'))	(c,($\{(b,2),(d,4)\}$,'old')) (c,({(a,2)},'rB'))	(d,($\{(a,4),(b,4),(c,4)\}$,'rB'))
(6) ReduceVertex	new = $\emptyset$ old = {(b,2),(c,2),(d,4)} rB = $\emptyset$	new =$\emptyset$ old = $\{(d,4)\}$ rB = $\{(a,2),(c,2)\}$	new = $\emptyset$ old = $\{(b,2),(d,4)\}$ rB = $\{(a,2)\}$	new = $\emptyset$ old = $\emptyset$ rB = $\{(a,4),(b,4),(c,4)\}$
(7) Schulze Winner	`true`	`false`	`false`	`false`

Table 4.7: Computing the Schulze Winner by Example

4.6.2 Pregel Algorithm for computing the Schulze Winner

In this section, we present our Pregel-based algorithm for determining the Schulze winners for a given weighted majority graph $\mathcal{W} = (A, E, \mu)$; we use $p(a, b)$ to refer to the width of the widest path connecting a to b. A straightforward algorithm would consist in computing the path width for all pairs of vertices in the graph. So in particular, for every pair (a, b) of vertices in $\mathcal{W}$ the widths $p(a, b)$ and $p(b, a)$ are computed. Such an algorithm would require us to store a linear amount of information for each vertex v, i.e. at each vertex v the information $p(v, a)$ for every $a \in A$ is stored. This contradicts the philosophy of Pregel algorithms which aim at keeping the local information at each vertex small [YCX+14].

The Schulze Winner is always a subset of the Schwartz set, i.e. the set of all undominated vertices. This relationship is obvious from the definition. The Schwartz set is the union of all undominated strongly connected components (SCCs) and therefore an algorithm using this idea would first compute the Schwartz set and then only compute the widest paths for pairs of vertices in the undominated SCCs. This approach has the disadvantage that the set of pairs to be considered may still be very large. Moreover, the computation of the SCCs only makes use of a part of the available information since it is not considering the weights in the graph.

The idea of our new Pregel-style algorithm is therefore, to use the methods for computing SCCs and adapt them in order to include the information of the weighted edges – the majority margin. The forward/backward propagation of minimum vertex-ids, which has already been used in the Sections 4.4 and 4.5, forms the heart of the algorithm. It is adapted such that it additionally propagates the widths of the paths and utilises the weight information to prune the search space as soon as possible. Further, the local information stored at each vertex is guaranteed to be small.

Recall from Section 2.3 that Pregel Algorithms are often described as 'think-like-a-vertex' algorithms. This comes from the fact that each vertex acts as an independent

entity. This means that the vertices are distributed among the nodes of the cluster and the computations at each vertex can be performed in parallel. The vertices exchange information by sending messages to each other along the edges of the graph. One superstep of a Pregel Algorithm consists of the following two steps: sending messages, and executing the vertex program. Vertices can be set inactive either by their vertex program, or automatically if they did not receive a message by any other vertex. The Pregel computation stops when there are no active vertices left. One important performance measure of Pregel algorithms is the size of the local information saved at each vertex. The proposed Pregel algorithm for computing the Schulze Method is guaranteed to have small local information at each vertex. The overall structure of our algorithm is given in Algorithm 4.6.1.

Algorithm 4.6.1: SchulzeWinner(W_P)

Input: Weighted Majority Graph W_P
Output: Schulze Winner

```
1 Initialisation-of-vertices;
2 while there exists a vertex v with v.status = 'unknown' do
3 |   Preprocessing;
4 |   Forward-Backward-Propagation;
5 |   Postprocessing;
6 end
7 Output vertices with status 'winner';
```

We assume that each vertex v is assigned an unique id $v.id \in \{1, \dots m\}$. Moreover, each vertex v has a *status* which may take one of three possible values {'winner', 'loser', 'unknown'} to express that v is a Schulze winner, not a Schulze winner, or if we do not know yet, respectively. Additional information stored at each vertex includes the fields s, t, ws, wt, and scc, whose meaning will be explained below, as well as information on the adjacent edges together with their weights.

In Initialisation-of-vertices, we determine for each candidate v the maximum weight of all incoming and outgoing edges and set the status accordingly: if $\max_{a \in A} \mu(a, v) > \max_{a \in A} \mu(v, a)$, then we know for sure that there exists a vertex c (namely the one with $\mu(c, v) = \max_{a \in A} \mu(a, v)$) which is preferred to v by the Schulze method. Hence, in this case, we set $v.status =$ 'loser'; otherwise we set $v.status =$ 'unknown';

The goal of each iteration of the while-loop in Algorithm 4.6.1 is to compute for every vertex v the ids s (= source) and t (= target) which are the minimum ids among all vertices with *status* = 'unknown' such that there is a path from s to v and from v to t. Moreover, we also determine the weights ws and wt of the widest paths from s to v and from v to t. Termination is guaranteed since in each iteration at least one vertex changes its status from 'unknown' to 'loser' or 'winner'. The experimental evaluation (see Chapter 6) shows that the algorithm terminates very fast: on real-world data, typically even a single iteration of the while-loop suffices. Preliminary experiments with synthetic

data show that the while-loop is executed less than 10 times for instances with 10.000 candidates.

In Preprocessing, shown in Algorithm 4.6.2, we initialise the fields (s, ws, t, wt) of all vertices. For every vertex v with $v.status =$ 'unknown', we set $v.s = v.t = v.id$ and $v.ws = v.wt = \infty$. Thus, initially, the minimum id of vertices to reach v and reachable from v is the id of v itself. Of course, this path from a vertex to itself has arbitrarily big width (although this does not matter in the sequel; we could have assigned any value to ws and wt). We send the information on v as a source (resp. target) to its adjacent vertices via outgoing (resp. incoming) edges. For vertices with status different from 'unknown', we set $v.s = v.t = \infty$ and $v.ws = v.wt = 0$. This allows such a vertex v to pass on vertex ids from other sources and targets but it prevents v from passing on its own id.

Algorithm 4.6.2: Preprocessing()

```
1  if v.status = 'unknown' then
2  |  s = v.id; ws = ∞;
3  |  t = v.id; wt = ∞;
4  |  foreach outgoing edge (v, u) with weight w do
5  |  |  send ('forward', s, w) to vertex u;
6  |  end
7  |  foreach incoming edge (u, v) with weight w do
8  |  |  send ('backward', t, w) to vertex u;
9  |  end
10 else
11 |  s = ∞; ws = 0;
12 |  t = ∞; wt = 0;
13 end
```

The Forward-Backward-Propagation is the actual 'Pregel heart' of the computation. The other procedures work in parallel too, but they do not use the Pregel Computation API. Algorithm 4.6.3 realizes the forward and backward propagation as a Pregel procedure. For each vertex v, we determine (1) the minimum source-id s together with the maximum width ws of paths from s to v and (2) the minimum target-id t together with the maximum width wt of paths from v to t. We thus analyse each received message (d, u, w) consisting of a direction d (with possible values 'forward' and 'backward'), vertex id u, and width w. In case of 'forward' direction, we have to check if we have found a source u with a yet smaller id than the current value s. If so, we update $v.s$ and $v.ws$ accordingly. If the received vertex-id u is equal to the current value of $c.s$, we have to update $c.ws$ in case the received value w is greater than $c.ws$ (i.e., from the same source we have found a path of greater width).

Messages in 'backward' direction are processed analogously, resulting in possible updates of the target-id t and/or the width wt of paths from v to t. After all messages have been

Algorithm 4.6.3: Forward-Backward-Propagation

```
1  foreach received value (d, u, w) do do
2  |  if d = 'forward' then
3  |  |  if u < s then
4  |  |  |  s = u;
5  |  |  |  ws = w;
6  |  |  else if u = s then
7  |  |  |  ws = max(ws, w)
8  |  |  ;
9  |  else if d = 'backward' then
10 |  |  if u < t then
11 |  |  |  t = u;
12 |  |  |  wt = w;
13 |  |  else if u = t then
14 |  |  |  wt = max(wt, w);
15 end
16 if (s, ws) has changed then
17 |  foreach outgoing edge (v, c) with weight w do
18 |  |  send ('forward', s, min(ws, w)) to vertex c;
19 |  end
20 if (t, wt) has changed then
21 |  foreach incoming edge (c, v) with weight w do do
22 |  |  send ('backward', t, min(wt, w)) to vertex c;
23 |  end
24 if the labels were not changed then
25 |  set v inactive;
```

processed, we propagate the information on new source id s (resp. target id t) and/or increased width of paths from s to v (resp. from v to t) to all adjacent vertices of v in forward (resp. backward) direction. Forward-Backward-Propagation terminates when no more messages are pending.

In Postprocessing, as shown in Algorithm 4.6.4, we use two crucial properties of source and target ids, which are inherited from the SCC computation in [YCX+14]: First, if for a vertex v, we have $v.s = v.t$, then the set of vertices u with the same source/target id (i.e., $u.s = u.t = v.s$) forms the SCC of v. Second, if for two vertices v and u, we have $v.s \neq u.s$ or $v.t \neq u.t$, then v and u belong to two different SCCs.

In Algorithm 4.6.4, we first compare, as for the algorithm for computing the Schwartz Set in Section 4.5, for each vertex v the values of s and t: if $v.s < v.t$, then v is reachable from s but s is not reachable from v. Hence, v is a loser. If $v.s > v.t$, then t is reachable from v but v is not reachable from t. Hence, t is a loser. Note that setting the status of t

Algorithm 4.6.4: Postprocessing for vertex v

```
1  if v.s < v.t then
2  |  v.status = 'loser';
3  else if v.s > v.t then
4  |  set status of vertex v.t to 'loser';
5  else if v.s = v.t then
6  |  v.scc = v.s;
7  |  if v.ws > v.wt then
8  |  |  v.status = 'loser';
9  |  else
10 |  |  set status of vertex v.s to 'loser';
11 |  end
12 foreach incoming edge (u, v) do
13 |  get (u.s, u.t) from vertex u;
14 |  if v.s ≠ u.s or v.t ≠ u.t then
15 |  |  if v.s = v.t then
16 |  |  |  foreach vertex c with c.scc = v.s do
17 |  |  |  |  set status of vertex c to 'loser';
18 |  |  |  end
19 |  |  else if v.s ≠ v.t then
20 |  |  |  v.status = 'loser';
21 |  end
22 end
23 if v.scc = c and v.status = 'unknown' then
24 |  v.status = 'winner';
25 end
```

is done by a subroutine whose details are omitted here. Finally, if $v.s = v.t$, then (as recalled above) we have found the SCC of v. As in [YCX+14], we use the minimum id of the vertices in an SCC to label the SCC. If the width of the path from s (which is equal to t by our case distinction) to v is greater than from v to s, then v is a loser (since s is preferred to it). In the opposite case, s is a loser.

In the next step in Algorithm 4.6.4, we compare the values of (s, t) of each vertex v with the values of $(u.s, u.t)$ of all vertices u with an incoming edge (u, v). If $u.s \neq v.s$ or $u.t \neq v.t$, then v and u are in different SCCs. By the existence of the edge (u, v), this means that there can be no path from v to u (otherwise v and u would be in the same SCC). Hence, u is preferred to v according to the Schulze method. Moreover, if $v.s = v.t$, then we have found the SCC of v. In this case, u is preferred to all vertices in this SCC.

Further, suppose that we have found some SCC such that the vertex v with minimum id in this SCC has not been identified as a loser by any of the above cases. In particular,

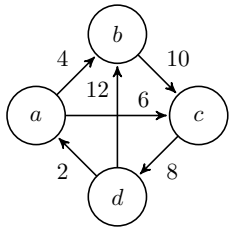
this means that none of the vertices in this SCC has an incoming edge from outside the SCC and, moreover, the SCC cannot contain a vertex u with $p(u,v) > p(v,u)$. In this case, we may mark vertex v as a winner. It is now also clear that at least one vertex must change its status from 'unknown' to either 'loser' or 'winner' in every execution of Algorithm 4.6.4 and, therefore, in every iteration of the while-loop of Algorithm 4.6.1.

Finally, we remark that by iterating the algorithm k times and continuously removing Schulze winners, it is straight-forward to compute a top-k ranking according to Schulze.

4.6.3 Computing the Schulze Winner by Example

Example 9 *Consider the weighted majority graph displayed in Figure 4.4 (originally by Schulze [Sch03]). The table in Figure 4.4 shows the widest paths between any two vertices. The unique Schulze winner is candidate a, having a path of width 6 to every other candidate, whereas all incoming paths to vertex a have width* 2.

The weighted majority graph in Figure 4.4 is strongly connected. Hence, the set of Schwartz winners is the entire SCC $\{a, b, c, d\}$.



	a	b	c	d
a	–	6	6	6
b	2	–	10	8
c	2	8	–	8
d	2	12	10	–

Figure 4.4: A weighted majority graph and its widest paths

We go through the algorithm for computing the Schulze Winner step-by-step:

Initialisation-of-vertices: *For each candidate we determine the maximum weight of all incoming and outgoing edges and set the status accordingly. The result can be seen in Table 4.8. After this initialisation step only candidates a and b remain as possible winners, with status 'unknown'.*

v	v.id	$\max_{u \in A} \mu(u,v)$	$\max_{u \in A} \mu(v,u)$	status
a	1	6	2	unknown
b	2	10	12	loser
c	3	8	10	loser
d	4	12	8	unknown

Table 4.8: Initialisation of Vertices

Start of the While Loop: *First for each vertex the fields (s,ws,t,wt) are initialised, based on their status.*

Superstep	v	v.id	v.status	$v.s$	$v.ws$	$v.t$	$v.wt$
PreProc	a	1	unknown	1	∞	1	∞
PreProc	b	2	loser	∞	0	∞	0
PreProc	c	3	loser	∞	0	∞	0
PreProc	d	4	unknown	4	∞	4	∞
1	a	1	unknown	1	∞	1	∞
1	b	2	loser	**1**	**4**	∞	0
1	c	3	loser	**1**	**6**	**4**	**8**
1	d	4	unknown	4	∞	**1**	**2**
2	a	1	unknown	1	∞	1	∞
2	b	2	loser	1	4	**4**	**8**
2	c	3	loser	1	6	**1**	**2**
2	d	4	unknown	**1**	**6**	1	2
3	a	1	unknown	1	∞	1	∞
3	b	2	loser	1	**6**	**1**	**2**
3	c	3	loser	1	6	1	2
3	d	4	unknown	1	6	1	2
4	a	1	unknown	1	∞	1	∞
4	b	2	loser	1	6	1	2
4	c	3	loser	1	6	1	2
4	d	4	unknown	1	6	1	2
PostProc	a	1	winner	1	∞	1	∞
PostProc	b	2	loser	1	6	1	2
PostProc	c	3	loser	1	6	1	2
PostProc	d	4	loser	1	6	1	2

Table 4.9: Vertices during the Computation

Only the vertices with status='unknown' initialise their attributes s and t with their id, vertices with a different status use ∞ as starting value. Such that in the further computation we only compute reachability and widest paths from and to 'unknown' vertices and other vertices only act as passageways. The resulting input to the first iteration of the while-loop is shown in Table 4.9 in the first 4 rows.

In the first step (PreProc) only the vertices a and d send messages. Vertex a has two outgoing edges to the vertices b and c and one incoming edge from d. Vertex a therefore sends the message ('forward',1,4) to vertex b, where 1 is the id of a and 4 is the minimum of its ws attribute and the weight of the edge connecting a and b (i.e. $min(a.ws, \mu(a,b)) = min(\infty, 4) = 4$). This message informs b that a can reach it along a path with weight 4. The edge connecting a and c has weight 6, therefore a sends the message (forward, 1, 6) to vertex c. Along the incoming edge, vertex a sends a message to d, with the information ('backward', 1,2); telling vertex d that there is a path from d to a with weight 2.

Superstep	sender	receiver	message
PreProc	a	b	('forward', 1, 4)
PreProc	a	c	('forward', 1, 6)
PreProc	a	d	('backward', 1, 2)
PreProc	d	a	('forward', 4, 2)
PreProc	d	b	('forward', 4, 12)
PreProc	d	c	('backward', 4, 8)
1	b	c	('forward', 1, 4)
1	c	d	('forward', 1, 6)
1	c	a	('backward', 4, 8)
1	c	b	('backward', 4, 8)
1	d	c	('backward', 1, 2)
2	b	a	('backward', 4, 8)
2	b	d	('backward', 4, 8)
2	c	a	('backward', 1, 2)
2	c	b	('backward', 1, 2)
2	d	a	('forward', 1, 6)
2	d	b	('forward', 1, 6)
3	b	a	('backward', 1, 2)
3	b	d	('backward', 1, 2)
3	b	c	('forward', 1, 6)

Table 4.10: Messages sent during the computation

Vertex d sends messages along its incoming and outgoing edges to the adjacent vertices (outgoing: a and b, incoming: c).

The vertices b and c do not send any messages, but they are going to receive messages, update their statuses and send messages in the next superstep.

Next each vertex combines the information contained in the received messages. Vertex a received only one message ('forward', 4,2). Since the id 4 *is larger than the* $a.s = 1$ *attribute, the information saved at vertex a is not changed. Vertex b receives two 'forward' messages: ('forward', 1,4) and ('forward',4,12). The received messages are compared to its s attribute (*$b.s = \infty$*) and the information is updated accordingly. Since (*$1 < 4 < \infty$*) the attribute b.s is set to* 1 *and because of the third attribute in the message ('forward',1,4) b.ws is set to* 4*. Now vertex b knows that the vertex with the smallest ID, that can reach b (within a path length of* 1*) has id* 1*(=a.id) and that the widest path known from a to b has width* 4*. The update procedure of the information at vertex c and d works in the same way. After updating the attributes the vertices again start with sending messages. Only vertices which updated attributes send messages. Following Algorithm 4.6.3 the messages sent over the overall computation are shown on top of Table 4.10.*

Each vertex processes its received messages independently and updates its vertex information. Then new messages are sent to the adjacent vertices. The first column of

Table 4.9 is the number of the superstep of the backward-forward-propagation algorithm. The attributes changing in the corresponding step are formatted bold.

In Table 4.10 all messages sent in each step can be found. Only vertices with updated attribute values send messages. In Table 4.9 it is interesting to observe, that in superstep 3 *the attribute ws of vertex b is changed, but the attribute s is not changed. This is because the smallest path from a to b has length* 1*, but the widest path has length* 3 *and is therefore discovered in the third superstep. The messages sent after the third superstep do not yield any changes in the attributes of the vertices.*

The Postprocessing procedure following Algorithm 4.6.4 leads to the change of the status of d since $s = t = 1$ *(id of vertex a), and the path from a to d is wider than the path from d to a. This can be read from the Table, where ws is larger than wt. In Table 4.9 for the last superstep it can be oberved that all vertices have vertex a as s and t. For each vertex the incoming widest path from a is larger than the outgoing path to a (*$ws > wt$*) and therefore a is the Schulze Winner.*

The Postprocessing changes the status of vertex a to 'winner'.

In this example we only need one iteration of the While-Loop and 4 Supersteps in the Pregel procedure to identify the Schulze Winner. ◇

4.7 Ranked-Pairs Rule

In Chapter 3 it is shown that the winner determination problem by the ranked-pairs method is P-complete and therefore inherently sequential [Joh90]. We will sketch some algorithm ideas for this problem.

As stated in Section 2.1 the ranked pairs method is computed the following way: the edges are sorted by their weight and starting from the heaviest edge one by one is added, but only if it does not create a cycle in the graph.

Therefore, adding edges that cannot be part of a cycle is safe and consistent with the method. Some edges can be identified which are definitely **not** part of a cycle:

- all incoming edges of a vertex with outdegree of 0
- all outgoing edges of a vertex with indegree of 0
- edges connecting vertices in different strongly connected components

Further, the edges within one SCC cannot interfere with edges in other SCCs. The central problem of this method is the detection of cycles. The Schwartz-Set MapReduce algorithm can be adapted to detect cycles: If a vertex a receives as input the key-value pair $(a, ('a', 'rB'))$, i.e. gets the information that it is reached by itself, then there is a cycle in the graph.

We sketch the following algorithm:

- Detect SCCs – for example by using the min-label algorithm proposed in [YCX+14]
- Add all edges (u, v) satisfying at least one of the following conditions to the result graph $G(V, E)$ and add the vertices u and v to the set V.
  - v has no outgoing edges
  - u has no incoming edges
  - u and v are not in the same SCC
- For each SCC $i = \{1, \ldots, l\}$ create a graph $G_i(V_i, E_i)$ with the set $V_i = \emptyset$ of vertices and $E_i = \emptyset$ of edges.
- Compute the ranked-pairs method for each SCC_i separately by repeating the following steps until no edges are left in the SCC_i.
  Start with the heaviest edge (u,v) in SCC_i: if $u \in V_i \wedge v \notin V_i$ or $u \notin V_i \wedge v \in V_i$: add (u,v) to E_i and $\{u, v\}$ to V_i; otherwise: check if a cycle is produced by adding (u, v) to G_i. If 'yes' delete the edge, otherwise add it to G_i
- Add all sets of vertices V_i and edges E_i to the result graph G.
- Select the root of G as winner.

4.8 STV Rule

The single transferable voting rule is based on the preference profile P as input. The candidate with the lowest plurality score is removed from all votes in the preference profile in each round. Then the plurality scores of the remaining candidates in the new preference profile are calculated. Again, the candidates with the lowest score is removed. This is repeated until only one candidate is left and this candidate is the winner.

4.8.1 MapReduce Algorithm for computing the Winner based on the STV rule

The MapReduce algorithm for computing the winner according to the STV rule maintains the set of already excluded candidates and controls the iteration through $m-1$ MapReduce rounds, such that each round excludes one more candidate. The MapReduce rounds are used to compute the plurality score on the current set of votes. The preference profile itself is not changed, but the map procedure gets the information of which candidates have been excluded and uses this to assign the scores. The reduce task then simply sum up the values to compute the plurality score of each candidate.

The basic idea of the algorithm is to use $m - 1$ rounds and exclude one candidate per round. The overall algorithm is shown in Algorithm 4.8.1. During the map phase, the highest-ranking not-yet-excluded candidate of each vote is sent to the corresponding

reduce task, which simply counts the number of received messages. Each reduce task is responsible for the score of one candidate. The map and reduce procedures are shown in Algorithm 4.8.3 and Algorithm 4.8.2. The next round starts with the exclusion list extended by the lowest scoring candidate (subject to tie-breaking). Clearly, this algorithm is impractical for large m as it requires $m-1$ rounds. However, for small m, this algorithm can be considered feasible – which matches exactly the theoretical claims of Theorems 3 and 4.

Algorithm 4.8.1: STV

Input: Preference profile P containing m votes
Output: STV Winner

```
1 excl_candidates=∅;
2 for 1 to m-1 do
3 |  MapSTV;
4 |  ReduceSum;
5 |  excl_candidates += candidate with the lowest sum;
6 |  # use tiebreaking order if needed;
7 end
8 return the single candidate not contained in excl_candidates;
```

Algorithm 4.8.2: MapSTV()

Input: Preference profile P containing m votes and the set excl_candidates
Output: key-value pairs of the form (key=candidate,value=score)

```
1 foreach vote in P do
2 |  for i=1 to length(vote) do
3 |  |  if vote[i] is not in excl_candidates then
4 |  |  |  emit(key=vote[i], value=1)
5 |  |  |  break;
6 |  |  end
7 |  end
8 end
```

Algorithm 4.8.3: ReduceSum()

Input: Key-value pairs(key=candidate,value=score) produced by MapSTV()
Output: Sum of all received values (candidate with score)

```
1 return (key, sum(values))
```

Part III

Theoretical Performance and Experimental Evaluation

CHAPTER 5

Theoretical Performance Guarantees of the proposed Algorithms

5.1 Performance Guarantees for Our MapReduce Algorithms

MapReduce algorithms for various problems in computational social choice have been described in Chapter 4. In this section the performance guarantees for the Algorithms are derived. The derivation of the performance guarantees by theorems and proofs can be found in the following subsections, and a summary is provided in Section 5.1.8. For an introduction to the performance measures of MapReduce algorithms see Section 2.3.1. The size of a problem instance and therefore the computational performance of the problems depends on the number of votes n in the election and the number of candidates m.

5.1.1 Performance of the MapReduce Algorithm for Positional Scoring Rules

Proposition 5 *The scores of candidates, according to a scoring rule with scoring vector s, in an election given by a preference profile P can be computed in MapReduce with the following characteristics: $rr = 1$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = n$, $wct \leq m$, and $tcc \leq mn$.*

Proof 5 *The algorithm takes only one MapReduce round ($\#\mathsf{rounds} = 1$). Each entry in a vote of the preference profile results in one key-value pair, therefore the replication rate is 1 ($rr = 1$). The candidates are used as keys, from this follows that we have m keys.*

Each reduce task receives at most n values (one per vote) and outputs exactly 1 value. From this we know that $wct \leq n + 1$ and $tcc \leq m(n + 1)$. ◇

5.1.2 Performance of the MapReduce Algorithm for transforming the Preference Profile into a Graph Representation

Proposition 6 *The MapReduce algorithm for computing the weighted majority graph has the following characteristics: $rr \leq m - 1$, $nr = 1$, $nk = m^2$, $wct \leq n + 1$, and $tcc \leq m^2(n + 1)$.*

Proof 6 *There are m^2 reducers (one for every pair (a, b) of candidates). Each reducer receives at most n values (one key-value pair from each vote) and outputs the sum of these values, i.e. the majority margin of a and b. From this follows that $wct \leq n + 1$.*

Each vote has a maximum length of m and therefore results in a maximum number of $2 \cdot \binom{m}{2} = m(m - 1)$ key-value pairs. Therefore the upper bound of the replication rate is $rr \leq m - 1$. ◇

5.1.3 Performance of the MapReduce Algorithm for Computing the Copeland Scores

Proposition 7 *The Copeland scores can be computed by a MapReduce algorithm with the following characteristics: $rr = 2$, $\#\mathsf{rounds} = 1$, $\#\mathsf{keys} = m$, $wct \leq 2m + 2$, and $tcc \leq 2m^2 + 2m$.*

Proof 7 *Algorithm 4.3.1 takes one map-reduce round, therefore $\#\mathsf{rounds} = 1$. There are m reducers (each corresponding to one candidate, i.e., one row or column in the dominance matrix). Each reducer receives as input $2m$ values, because each reduce task receives exactly one column and one row of the dominance matrix. Note, that each entry in the dominance matrix is sent to exactly 2 reduce tasks. We thus have a replication rate of 2 ($rr = 2$). Each reduce task then produces one pair (candidate, score) as output. In total, we thus have $wct \leq 2m + 2$ and $tcc \leq 2m^2 + 2m$.* ◇

5.1.4 Performance of the MapReduce Algorithm for Computing the Smith Set

Proposition 8 *Algorithm 4.4.1 for computing the Smith set has the following characteristics: $rr \leq 2m + 1$, $\#\mathsf{rounds} = 3$, $\#\mathsf{keys} = m$, $wct \leq 6m^2 + 8m$ and $tcc \leq 6m^3 + 8m^2$.*

Proof 8 *The number of MapReduce rounds is 3 ($\#\mathsf{rounds} = 3$). For the replication rate we inspect the total number of key-value pairs received by each reducer. In the first MapReduce round the replication rate is trivially ≤ 2. Since each entry in the dominance*

matrix is mapped to at most 2 *key-value pairs. In the following two rounds each reducer receives one set with mode = 'old', at most* m *sets with mode = 'new' and at most* m *sets with mode = 'reachedBy'. Each of those sets has a maximal possible length of* m*. This gives a total amount of at most* $2m^3 + m^2$ *vertices. Since we started from a matrix of size* m^2 *the total replication rate is* $\leq 2m + 1$*. The number of keys is equal to the number of candidates* m*.*

The amount of information received by each reducer in each round is bounded by $2m^2 + m$*, namely linearly many VertexSetWritables with modes 'new' and 'reachedBy' and one VertexSetWritable with mode 'old'. The amount of information sent by each ReduceVertex process is bounded by* $2m$*. Since 'old' and 'new' are disjoint sets, the total size of the three sets ('old', 'new' and 'reachedBy') cannot be greater than* $2m$*. The output of ReduceComplement in the last map-reduce round is bounded by* m*. In total, for the 3 map-reduce rounds, we thus have* $wct \leq 6m^2 + 8m$ *and* $tcc \leq 6m^3 + 8m^2$*.* ◇

5.1.5 Performance of the MapReduce Algorithm for Computing the Schwartz Set

Proposition 9 *Algorithm 4.5.1 for computing the Schwartz set has these characteristics:* $rr \leq 2m + 1$*,* $\#\mathsf{rounds} = \lceil \log_2 m \rceil + 1$*,* $\#\mathsf{keys} = m$*,* $wct = (2m^2 + 3m)(\lceil \log_2 m \rceil + 1)$*, and* $tcc \leq 2m^3 + 3m^2(\lceil \log_2 m \rceil + 1)$*.*

Proof 9 *The number of iterations of the while loop is bounded by* $\lceil \log_2 m \rceil$*. Together with the first map-reduce round, we thus get the upper bound* $\#\mathsf{rounds} \leq \lceil \log_2 m \rceil + 1$*.*

Inspecting the vertex-sets emitted by the mapper in each round, we observe that each reduce task receives one set with mode = 'old', up to m *sets with mode = 'new' and up to* m *sets with mode = 'reachedBy'. Hence, the* m^2 *entries in the input dominance matrix may give rise to at most* $2m^3 + m^2$ *vertices communicated in total to the* m *reduce tasks in each round. We thus get* $rr \leq 2m + 1$*.*

The number of keys $\#\mathsf{keys}$ *is bounded by the number* m *of candidates.*

The total size of all sets reachedBy or new that may ever be emitted is bounded by the maximum number of possible combinations (r, i, n)*, where* r *stands for an entry in the set reachedBy,* n *stands for an entry in the set new and* i *is an arbitrary vertex, i.e. the number of combinations is* m^3*.*
Together with the set of mode = 'old', the total amount of information received *by the reducers is* $\leq 2m^3 + m^2(\lceil \log_2 m \rceil + 1)$*. For the total amount of information written as* output *from the reduce tasks and serving as input to the mappers, we have the upper bound* $(\lceil \log_2 m \rceil + 1) \cdot 2 \cdot m^2$*, i.e., in each round, each of the* m *reduce tasks returns* 3 *VertexSetWritables of total size* $\leq 2m$*, since the sets 'old' and 'new' are disjoint. In total, we thus have* $tcc \leq 2m^3 + 3m^2(\lceil \log_2 m \rceil + 1)$*.*

For an upper bound on the wall clock time wct we have to multiply the number of rounds with an upper bound on the information received and sent by each reduce task in each round. We thus get $wct \leq 2m^2 + 3m(\lceil \log_2 m \rceil + 1)$. ◇

5.1.6 Performance of the MapReduce Algorithm for Computing the Schulze Winner

For the MapReduce algorithm for computing the Schulze winner, shown in algorithm 4.6.1, the following performance guarantees hold. Note, that the performance guarantees are identical to the performance measures derived for the MapReduce algorithm for computing the Schwartz Set in Section 4.5.

Proposition 10 *Algorithm 4.6.1 for computing the Schulze winner has these characteristics:* $rr \leq 2m+1$, $\#\mathsf{rounds} = \lceil \log_2 m \rceil + 1$, $\#\mathsf{keys} = m$, $wct = (2m^2+3m)(\lceil \log_2 m \rceil + 1)$, *and* $tcc \leq 2m^3 + 3m^2(\lceil \log_2 m \rceil + 1)$.

Proof 10 *The number of iterations of the while loop is bounded by* $\lceil \log_2 m \rceil$. *Together with the first map-reduce round, we thus get the upper bound* $\#\mathsf{rounds} \leq \lceil \log_2 m \rceil + 1$.

Inspecting the vertex-sets emitted by the mapper in each round, we observe that each reduce tasks receives one set with mode = 'old', up to m sets with mode = 'new' and up to m sets with mode = 'reachedBy'. Hence, the m^2 *entries in the input adjacency matrix may give rise to at most* $2m^3 + m^2$ *tuples communicated in total to the m reduce tasks in each round. We thus get* $rr \leq 2m + 1$.

The number of keys $\#\mathsf{keys}$ *is bounded by the number m of candidates.*

The total size of all sets reachedBy or new that may ever be emitted is bounded by the maximum number of possible combinations (r, i, n), *where r stands for an entry in the set reachedBy, n stands for an entry in the set new and i is an arbitrary vertex, i.e. the number of combinations is* m^3.
Together with the set of mode = 'old', the total amount of information received *by the reducers is* $\leq 2m^3 + m^2(\lceil \log_2 m \rceil + 1)$. *For the total amount of information written as* output *from the reduce tasks and serving as input to the mappers, we have the upper bound* $(\lceil \log_2 m \rceil + 1) \cdot 2 \cdot m^2$, *i.e., in each round, each of the m reduce tasks returns* 3 *WidestPathsWritables of total size* $\leq 2m$, *since the sets 'old' and 'new' are disjoint. In total, we thus have* $tcc \leq 2m^3 + 3m^2(\lceil \log_2 m \rceil + 1)$.

For an upper bound on the wall clock time wct we have to multiply the number of rounds with an upper bound on the information received and sent by each reduce task in each round. We thus get $wct \leq 2m^2 + 3m(\lceil \log_2 m \rceil + 1)$. ◇

Problem	Input	Input Size	#keys	rr
Scoring rules	total orders	$\mathcal{O}(mn)$	m	1
STV	total orders	$\mathcal{O}(mn)$	m	1
Dom. graph	partial orders	$\mathcal{O}(nm^2)$	m^2	m
Schwartz set	dom. graph	$\mathcal{O}(m^2)$	m	$2m+1$
Smith set	dom. graph	$\mathcal{O}(m^2)$	m	$2m+1$
Copeland set	dom. graph	$\mathcal{O}(m^2)$	m	2
Schulze	w. maj. graph	$\mathcal{O}(m^2)$	m	$2m+1$

Table 5.1: Summary of performance characteristics of our MapReduce algorithms.

5.1.7 Performance of the MapReduce Algorithm for Computing the STV Winner

Proposition 11 *For computing STV, we obtain the following characteristics:* $rr = 1$, $\#\mathsf{rounds} = m-1$, $\#\mathsf{keys} \leq m$, $wct \leq (m-1)(n+1)$, *and* $tcc \leq \frac{(m+2)(m-1)}{2} \cdot (n+1)$.

Proof 11 *There are* $m-1$ *rounds. For every total order in the preflist of each voter, the mapper only emits 1 value (namely score 1 for the top-candidate in this total order). There is no duplication of values and therefore the replication rate is* 1 *(*$rr \leq 1$*). Each reducer receives at most* n *values and outputs 1 value. Thus* $wct \leq (m-1) \cdot (n+1)$.

In each round, the number of keys (and, hence, the number of reduce tasks) is bounded by the number of not yet excluded candidates. That is, we start with m *reducers in the first round and end up with* 2 *reducers in the* $(m-1)$*-st round. The total number of reduce tasks is therefore* $\frac{(m+2)(m-1)}{2}$ *and we have* $tcc \leq \frac{(m+2)(m-1)}{2} \cdot (n+1)$. ◇

5.1.8 Summary of Performance Guarantees for Our MapReduce Algorithms

In this section we derived performance guarantees for all proposed MapReduce Algorithms. In Table 5.1 and Table 5.2 all performance guarantees of the proposed algorithms are summarized.

Input type and size. The algorithms in Table 5.1 are ordered by the type of input instance required. The algorithms for the positional scoring rules and for the STV winner determination require total orders as input. Total orders always have size $m \cdot n$, since we have n votes ranking m candidates. In the proposed MapReduce algorithms a graph is always represented as an adjacency matrix, where the entries in the matrix take either 0 and 1 in the dominance graph or the majority margin in the weighted majority graph as values. Therefore the input size of all problems using a graph representation as input is $\mathcal{O}(m^2)$, which is the size of the matrix.

Problem	**#rounds**	**wct**	**tcc**
Scoring rules	1	$n+1$	$m(n+1)$
STV	$m-1$	$(m-1)(n+1)$	$\frac{(m+2)(m-1)}{2} \cdot (n+1)$
Dom. graph	1	$n+1$	$m^2(n+1)$
Schwartz set	$\lceil \log_2 m \rceil + 1$	$\mathcal{O}(m^2 \log m)$	$\mathcal{O}(m^3)$
Smith set	3	$6m^2 + 8m$	$6m^3 + 8m^2$
Copeland set	1	$2m + 2$	$2m^2 + 2m$
Schulze	$\lceil \log_2 m \rceil + 1$	$\mathcal{O}(m^2 \log m)$	$\mathcal{O}(m^3)$

Table 5.2: Summary of performance characteristics of our MapReduce algorithms.

Number of keys. Most of the proposed algorithms use one key per candidate; only the algorithm for creating the graph representation uses m^2 keys. This is because we have to consider all possible pairs of candidates in the computation.

Replication rate. The replication rate varies a lot among the different algorithms. Scoring rules and STV winner determination have a replication rate of 1, since each vote is considered as a whole and no complex splitting of the data is required. For creating the graph representation it is necessary to first split each vote into all pairs of candidates. This results in a replication rate of m. The computation of the Schwartz Set, Smith Set and Schulze method are based on a special data structure and are solved by very similar algorithms, for which the replication rate is bounded by $2m + 1$.

Number of MapReduce rounds. The algorithms for computing the scoring rules, the graph representation, the Smith set and the Copeland set have a fixed number of MapReduce rounds. The algorithm for STV Winner Determination is $m - 1$ since only one candidate can be excluded per round. Note, that this is as expected from the derived complexity result in Chapter 3, since the STV Winner Determination method is shown to be in paraL.
For the Winner Determination problems by the Schwartz set and the Schulze method the number of rounds is bounded by $\lceil \log_2 m \rceil + 1$, since the whole transitive closure has to be computed, or in the case of the Schulze method all widest paths have to be found. In practice the bound is usually not reached and the computation stops after much fewer rounds; this is also observed in the experimental evaluation in Chapter 6.

Wall clock time. The wall clock time of the algorithms for computing the positional scoring rules and the graph representation is $n + 1$ and therefore only depends on the number of votes. The algorithm for computing the STV winner results in a wall clock time of $(m - 1)(n + 1)$. Therefore it depends on both, the number of votes and the number of candidates. The wall clock time of the remaining algorithms only depends on the number of candidates.

Total communication cost. The total communication costs of the algorithms for the winner determination by $C1$- and $C2$-functions depend on the number of candidates m. For the other methods: positional scoring rules, STV method and dominance graph; the total communication cost also depends on the number of votes. The total communication cost is cubic in the number of candidates for the Schwartz Set, Smith Set and the Schulze method.

5.2 Performance Guarantees for Our Pregel Algorithms

In Chapter 4 Pregel algorithms for the winner determination problems by the following methods have been proposed: Smith Set, Schwartz Set and Schulze Method. All of those algorithms use the central Pregel procedures of the forward and backward min-label propagation presented in [YCX+14]. The forward and backward propagation are both so called BPPAs (Balanced Practical Pregel Algorithms) with $\mathcal{O}(\delta)$ supersteps [YCX+14]; δ refers to the diameter of the graph. This means, that the algorithms have linear space usage, linear computation cost and linear communication cost.

The number of iterations for identifying all strongly connected components in the graph by min-label propagation is bounded by the longest path in the condensation of the input graph [YCX+14]. The condensation is the directed acyclic graph resulting from contracting each strongly connected component into one vertex. Unfortunately we cannot give any tighter bounds on the number of iterations for our algorithms. Therefore the number of iterations for the computation of the Smith set, the Schwartz set and the Schulze method is bounded by the number of strongly connected components of the input graph. In the following, we argue that the algorithms are not expected to reach the bound. The Pregel algorithms for computing the Smith or Schwartz set take as input the weak or strict dominance graph, respectively. The condensation resulting from the weak dominance graph has only one root, but the condensation of the strict dominance graph may have several roots. This property is implicitly used in the post processing procedures of the algorithms and is important to the following performance considerations. Recall, that the Schwartz set is the union of all undominated SCCs in the strict dominance graph and the Smith set is the unique undominated SCC in the weak dominance graph.

We provide some observations on the performance behaviour of the Pregel algorithms:

Performance of the Pregel algorithms for computing the Schwartz and the Smith Set. The (weak or strict) dominance graph D is given as input. The graph D has l strongly connected components, denoted by $\{SCC_1, SCC_2, \ldots, SCC_l\}$, therefore the resulting condensation C of D has l vertices and we are interested in finding the root(s) of the directed acyclic graph C.

The vertex v with the smallest id ($v.id$) in the graph is contained in SCC_i. The set S (sources) contains all SCCs, which have a path to SCC_i and the set T (targets) contains all SCCs, that are reachable from SCC_i. Note, that S, T, and SCC_i are disjoint sets.

After the forward and backward min-label propagation the vertices have the following labels (the notation $*$ refers to an arbitrary vertex id):

$$label(w) = (w.s, w.t) = \begin{cases} (*, v.id) & \text{if } w \in S \\ (v.id, v.id) & \text{if } w \in SCC_i \\ (v.id, *) & \text{if } w \in T \\ (*, *) & otherwise. \end{cases}$$

Vertices that are not connected to SCC_i can only occur if the strict dominance graph was taken as input, i.e. vertices of this type only exist in the computation of the Schwartz set. The postprocessing procedures of the algorithms cause us to exclude all vertices contained in T, i.e. their status is set to 'notSmith' or 'notSchwartz'. If S is the empty set, then SCC_i is an undominated strongly connected component and is identified as a winning set; i.e., SCC_i is either the Smith set or a part of the Schwartz set. If S is not empty, than SCC_i is excluded from the possible winners in the postprocessing step. In the next iteration the vertices in this SCC are not initialized and therefore do not propagate their *id*s as forward and backward labels in the pregel procedure. In this next iteration the vertex with the smallest *id* labelling the remaining vertices, is therefore a different one than in the preceding iteration.
In the worst case we exclude only one SCC per iteration. Therefore we have the worst case number of rounds in the while-loop of l.

Performance of the Pregel algorithm for computing the Schulze method. For computing the Schulze method the weighted majority graph $\mathcal{W}_P$ is used as input. A major difference to the algorithms for computing the Smith set and the Schwartz set is, that we can eliminate many vertices from the set of possible winners in a preprocessing step. In this preprocessing step we check the weights of the outgoing and incoming edges of each vertex and exclude a vertex if the maximum weight of all outgoing edges is smaller than the maximum weight of all incoming edges. For the further computation we observe that the weighted majority graph has the same edges (with positive weight) as the strict dominance graph. Note, that the Schulze Winners are always a subset of the Schwartz set. Based on the min-label propagations we also use postprocessing steps to eliminate vertices, as for the computation of the Schwartz set. We additionally use the information on the width of the widest paths. The propagation of the information on the width of the widest paths might take more supersteps than the propagation of the min-labels, since the widest path might be longer than the shortest path connecting two vertices. Nevertheless the theoretical performance boundaries of the Pregel procedure remain the same, as for the min-label propagation without weights. Despite the adaptation of the min-label propagation to maintain weights it is still a BPPA algorithm taking $\mathcal{O}(\delta)$ supersteps.
In the postprocessing step the additional information on the widest paths is used to exclude vertices from the possible Schulze winners. We therefore expect the Schulze

Algorithm to take much fewer iterations of the while-loop, than for the computation of the Schwartz set; although we cannot provide a tighter theoretical upper bound.

CHAPTER 6

Experimental Evaluation

In this chapter the experimental evaluation of the proposed algorithms is documented. In Chapter 5 the theoretical performance guarantees of all Algorithms were discussed and we observed that the Schwartz Set and the Schulze Method are the computationally most expensive methods. Therefore, we focus in the experimental evaluation on the Schwartz set and on the Schulze method. First the experimental evaluation of the MapReduce algorithm is discussed. For this evaluation we only used synthetic data. The evaluation of the Pregel algorithm is discussed next, where we used real-world data and synthetic data.

6.1 MapReduce Algorithm for Computing the Schwartz Set

The MapReduce algorithms presented in this thesis are implemented in Java on top of Hadoop. The source code is available opensource on GitHub[1]. The MapReduce algorithms for computing the Schwartz set, the Smith set and the Schulze method are very similar. By the theoretical performance guarantees of the algorithms we observed that the Schwartz Set and the Schulze Method are the computationally most expensive methods. Therefore, we focus on the Schwartz set, to show that the MapReduce algorithm is practicable and that it scales well in a cloud computing environment. The scalabilty is very important to ensure that the approach is suitable for even larger problem instances and it fits the architecture of the cloud computing environment. From the property of scalability it follows, that given larger and larger problem instances, one can achieve reasonable computation times for determining the winners of these elections by choosing an appropriate number of computation nodes.

[1] `https://github.com/theresacsar/BigVoting`

m Candidates	$10m$ Edges	$m^2/10$ Edges
1,000	10,000	100,000
3,000	30,000	900,000
5,000	50,000	2,500,000
7,000	70,000	4,900,000

Table 6.1: Number of Edges of the Synthetic Graphs

m Candidates	$10m$ Edges	$m^2/10$ Edges
1,000	0.02	0.2
3,000	0.007	0.2
5,000	0.004	0.2
7,000	0.002	0.2

Table 6.2: Density of the Synthetic Graphs

Test Environment. For the experiments the environment provided by Amazon as the Amazon Elastic Compute Cloud (EC2)[2] is used. It is possible to utilize an EC2 cluster with many different configurations – ne can choose among various types of computing nodes, with different hardware resources. The nodes in an EC2 cluster are called instances, and we utilize instances of the type `m3.xlarge` for our experiments. The `m3.xlarge` instances have the following configuration: 4 virtual CPUs, 15 GiB RAM, 2 x 40 GB Instance Storage and high network performance [3]. Note, that using different types of instances doesn't reflect on the general result of scalability of our algorithms. Up to 128 of such instances are used for the evaluation. The times we report in this section are measured from the start of the first MapReduce round to the end of the final MapReduce round.

Synthetic Datasets. To show practicality and scalability, synthetic datasets of varying size are utilized. We use the DigraphGenerator [SW16] to randomly generate dominance graphs with m candidates and $10m$ or $m^2/10$ edges. In the experimental analyses it can be observed that the number of edges is an important factor for the performance of the algorithms. Therefore, two different types of graphs with different densities are used. The generated random datasets have $m = \{1000, 3000, 5000, 7000\}$ candidates. The resulting number of edges are shown in Table 6.1. For each number of candidates, five different graph instances for the "$10m$ edges"-graphs and for the "$m^2/10$ edges"-graphs are generated. Only one dataset per number of candidates is used to limit the cost incurred by the experiments. The density ($\frac{2|E|}{|V|(|V|-1)}$) of the used graphs can be seen in Table 6.2.

Computation Time of the Schwartz set. Run times for computing the Schwartz set using the MapReduce algorithm as described in Section 4.5 are shown in Figure 6.1

[2] `https://aws.amazon.com/de/ec2/`

[3] `https://aws.amazon.com/ec2/previous-generation/?nc1=h_ls`

and Figure 6.2. Figure 6.1 shows the results for the sparser datasets with $10m$ edges. The number of candidates ranges from 1,000 to 7,000 and we used $1 + 2$ to $1 + 32$ EC2 instances (1 name node instance and x worker instances). A timeout of 60 minutes was used for this experiment. It can be observed that the time incurred falls below 20 minutes once $1 + 16$ EC2 instances are used, and below 15 minutes for $1 + 32$ instances.

The times for the denser ($m^2/10$ edges) graphs is shown in Figure 6.2. In comparison to the sparser graphs we use much more, up to $1 + 128$, EC2 instances. As Figure 6.2 shows, the runtimes are higher for the denser graphs, but the general picture remains the same. A timeout of 90 minutes was used for this experiment. We see that the time incurred falls below 40 minutes for most inputs once $1 + 64$ EC2 instances are used and below 20 minutes once $1 + 128$ EC2 instances are used.

Most importantly, we see that the implementation scales: We observe that utilizing a larger number of EC2 instances significantly decreases the computation time. Note that the main memory consumption used in the EC2 instances remains $\mathcal{O}(m)$, i.e., relative to the number of candidates, and not $\mathcal{O}(m^2)$. This is essential for scalability, as otherwise main memory size would become a hidden limit for scalability, not obvious in Figures 6.1 and 6.2.

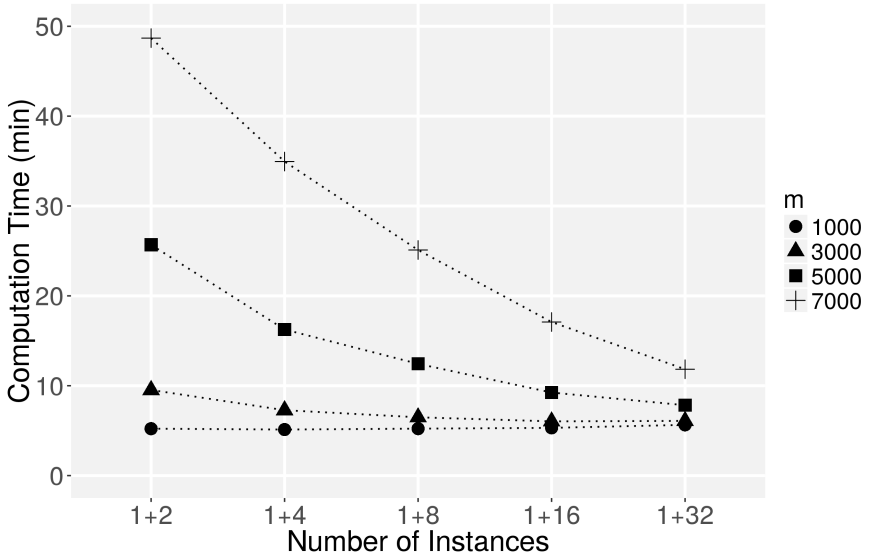


Figure 6.1: Time for the computation of the Schwartz Set with $10m$ edges using up to $1 + 32$ instances.

In this evaluation, the focus is on the Schwartz set as it was the most complex and theoretically most difficult one of the MapReduce algorithm proposed in this thesis – thus showing the overall limits of the approach. We demonstrate with our experiments that our algorithms indeed benefit from an increase in parallelization, i.e., a significant decrease in the run time can be observed when the number of processors (instances) is

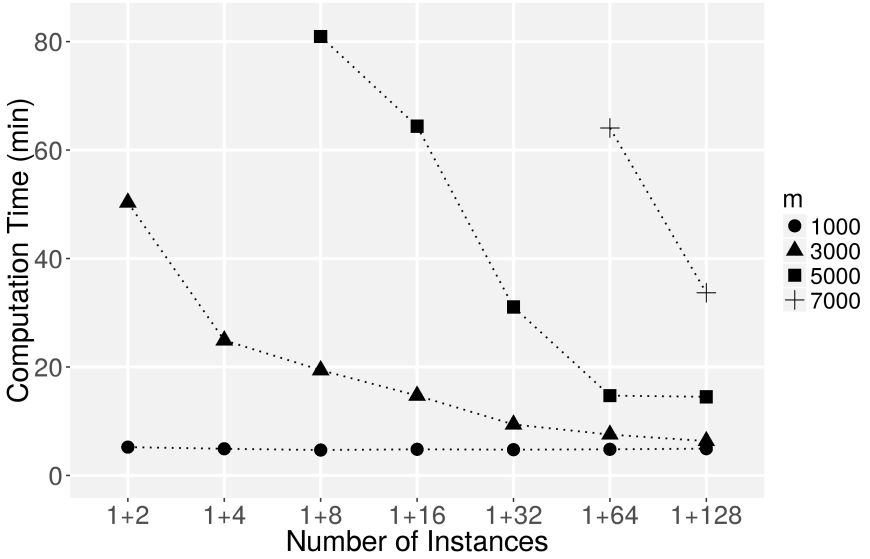


Figure 6.2: Time for the computation of the Schwartz Set with $m^2/10$ edges using up to $1 + 128$ instances.

increased.

6.2 Pregel Algorithm for Computing the Schulze Winner

The Schulze Method is the most involved of the proposed Pregel algorithms. We therefore perform experimental evaluation of the Algorithm for the Schulze method presented in Section 4.6. For this purpose we test them against actual real-world large-scale data sets. We use the ranking data provided by the online music streaming service Spotify. The Spotify ranking data[4] of 2017 consists of daily top-200 music rankings for 53 countries. First these rankings are transformed to the preflib dataformat (soi) [MW13]. This is done by considering the ranking of each day and country as a single voter, and then generate the corresponding weighted majority graph. The weighted majority graph in .soi format is then used as input for the Pregel Schulze algorithm. The experiments are based on four data sets generated from the spotify data: Global150, Global200, Europe150, and Europe200, which are derived from the daily top-150 or respectively top-200 charts of all available or only the European countries. We do not take into account the number of listeners in each country, since this information is not available. This information could easily be included by giving the votes/rankings by the countries corresponding weights.

In Table 6.3, we provide an overview of these four data sets. All four weighted tournament graphs are dense — the density calculated as $\frac{2|E|}{|V|(|V|-1)}$ is between 0.94 and 0.97 for all spotify graphs. Datasets referred to as *global* contain rankings from all countries available,

[4] `https://spotifycharts.com/regional`

Table 6.3: Spotify data sets

	candidates m	voters n	edges	after preproc.
Europe150	9,698	7,481	44.1M	11 undecided
Europe200	12,250	7,481	70.7M	12 undecided
Global150	14,187	15,553	94.9M	8 undecided
Global200	18,407	15,553	159.6M	9 undecided

where datasets *europe* only contain rankings from European countries. We also vary the length of used votes - e.g. we only use the first 150 or 200 songs from each vote. This means, the dataset `Europe150` contains as votes all top 150 rankings of the available European countries. We note that the Spotify data sets used here are significantly larger than any instances available in the PrefLib database [MW13].

Our Schulze algorithm is implemented in the Scala language. Furthermore, we use the GraphX library[5], which is built on top of Spark [ZCF+10], an open-source cluster-computing engine. GraphX provides a Pregel API, but is slightly more restrictive than the Pregel framework. In particular, in GraphX, only messages to adjacent vertices can be sent, while other Pregel implementations allow messages to be sent to arbitrary vertices. The source code of our implementation is available as open source on GitHub [6].

We ran our experiments on a Hadoop cluster with 18 nodes (each with an Intel Gold 5118 CPU, 12 cores, 2.3 GHz processor, 256 GB RAM, and a 10Gb/s network connection). To better observe the scalability of our algorithm, we restricted the number of cores and nodes (details follow).

6.2.1 Results

Our experiments show that our algorithm scales very well with additional computational resources: both an increase in nodes and in cores per node significantly sped up the computation. We refer the reader to Figure 6.3 for an overview of run-times for 1,2,3 and 4 nodes with up to 1,2,4 and 8 cores each. Therefore, we have a total number of cores of 1, 2, 3, 4, 6, 8, 12, 16, 24 and 32 in the experiments. On the x-axis of this chart we show the total number of cores, i.e., the number of nodes times the number of cores per node. For x-values with multiple interpretations we show the best runtime. This is always the configuration with most cores per node, since using another node costs more computing power. Note, that the differences between an increase in nodes or cores is almost negligible. Furthermore, our implementation manages to compute Schulze winners of all Spotify data sets within very reasonable time: with 4 nodes each using 8 cores, the data sets could be handled in less than 6.5min.

[5] `https://spark.apache.org/graphx/`
[6] `https://github.com/theresacsar/CloudVoting`

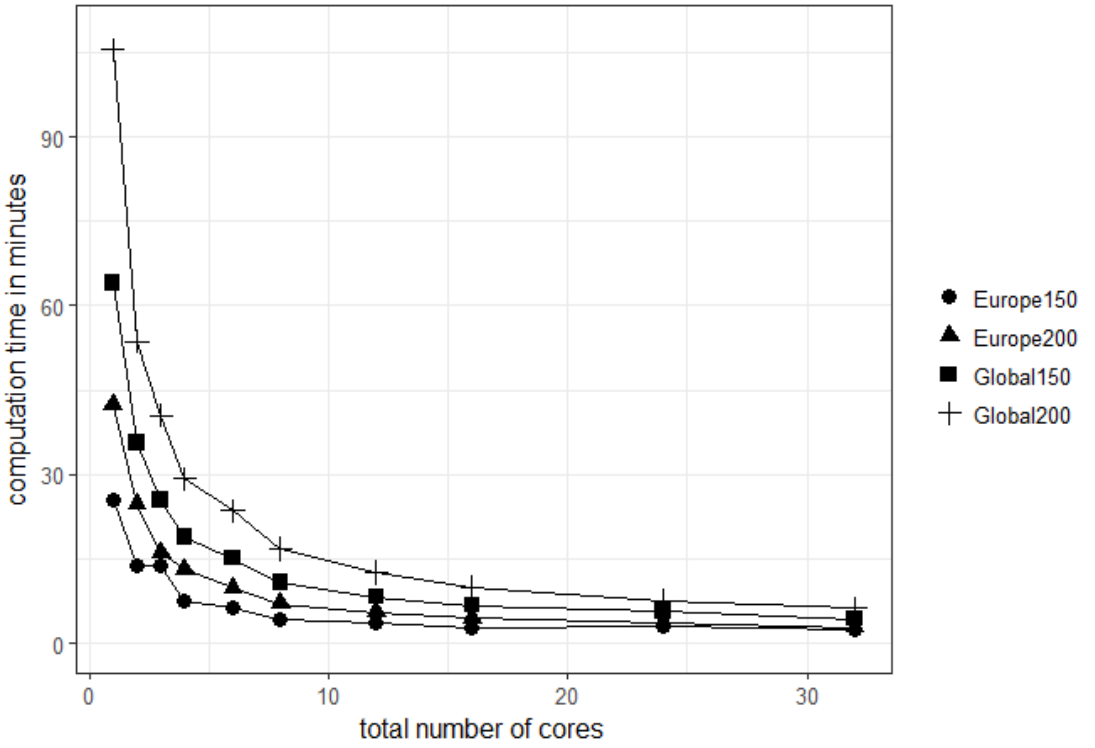


Figure 6.3: Runtime required for computing Schulze winners.

To the best of our knowledge the original, sequential algorithm [Sch11] based on the classical Floyd–Warshall algorithm is the only published algorithm for computing the Schulze winners. We compare the run-times of our implementation to this originally proposed algorithm. These two algorithms differ not only in their capability of parallelization, but also in that our algorithm only returns Schulze winners whereas the original algorithm returns a full ranking of candidates. Due to our focus on Schulze winners, we were able to include the many optimisations described in Section 4.6. We observe, that our algorithm is faster than the original algorithm even without parallelization (1 node with 1 core): Our algorithm requires with this configuration 26min/42min/64min/105min for the the Europe150/200 and Global150/200 data sets, respectively. In contrast, the original algorithm (also implemented in Scala) requires 71min/143min/221min for the Europe150/200 and Global150 data sets; it did not terminate in reasonable time for the Global200 data set. As mentioned before, this comparison is not completely fair due to the different output, but shows the impact of the optimisations in our algorithm.

Synthetic Data. We also performed experiments with low-density graphs (based on synthetic data). We observed that for these graphs the number of undecided candidates decreases slower with each iteration of the forward-backward propagation, in contrast to the Spotify data sets where even after preprocessing very few undecided candidates

Dataset	Vertices	Edges	Density	Time (s)	Rounds	μ	σ
Dataset1	1,000	71,065	0.14	271	8	5	1.30
Dataset2	5,000	518,070	0.04	220	6	4	1.30
Dataset3	5,000	1,621,807	0.13	207	3	5	1.30
Dataset4	10,000	2,216,769	0.04	263	8	5	1.30
Dataset5	10,000	7,532,099	0.15	432	4	6	1.30
Dataset6	10,000	12,915,325	0.26	441	4	7	1.30

Table 6.4: Synthetic Datasets and runtime of the Pregel Algorithm for Computing the Schulze method

remained (cf. Table 6.3). The computation of the spotify data sets takes at most 2 rounds, whereas for the synthetic datasets we observe, that they take much longer to converge.

For the experiments with the randomly generated datasets we did not restrict the used resources on the cluster so Spark can choose the optimal number of used nodes and cores. The total number of available nodes is 18, but the Spark used at most 6 nodes, so we never exceeded the maximum of cores used in the evaluation of the spotify data. The random data is generated using the function GraphGenerators.logNormalGraph(μ, σ) provided by GraphX. The chosen parameters, observed computation time and the number of rounds are shown in Table 6.4. For the MapReduce algorithms we observed in the preceding section that the density is an influential factor. This seems to not be the case for the Pregel algorithms. For the Spotify real world datasets the number of undecided candidates after the preprocessing was significantly lower than the number observed in the evaluation of the synthetic datasets. Further, the number of undecided candidates decreases much slower than in the real world data.

Part IV

Conclusion

CHAPTER 7

Conclusion

7.1 Summary and Discussion

This thesis deals with the computational challenges arising from selecting winners from large elections. The focus lies on the design of cloud computing algorithms for selected methods of winner determination in computational social choice. Complexity results on winner determination methods – namely the single tranferable vote (STV) rule, ranked-pairs rule and the Schulze method – are derived to show whether they are suitable for parallel computation. The NL-completeness result for the Schulze Winner Determination problem ensured us, that efficient parallel execution is possible. Further, it is shown that the ranked-pairs methods is P-complete and therefore inherently sequential [Joh90]. For the STV Winner Determination problem it is shown, that it is solvable in $\mathcal{O}(m + \log(n))$ space. Thus, a MapReduce algorithm under this parametrization is proposed in Section 4.8.1.

For other winner determination methods – namely the Schwartz set, the Smith set, the Schulze method, positional scoring rules and the Copeland scores – cloud computing algorithms are designed using the programming paradigms MapReduce and/or Pregel. For $C1$- and $C2$-functions, which take the dominance graph or the weighted majority graph as input, MapReduce algorithms as well as Pregel algorithms are proposed. The graph representation itself intuitively calls for a Pregel algorithm, but the representation of the input graph as an adjacency matrix allows us to efficiently apply MapReduce algorithms. The other winner determination methods are based on the preference profile as input and only MapReduce algorithms are designed for this type of methods, since the Pregel computation framework is only applicable to graph based computations.

The scalability and practicality of the proposed cloud computing algorithms is shown by experimental evaluation.

MapReduce Algorithms. All proposed MapReduce algorithms are analysed with regard to their theoretical performance measures. Performance guarantees for replication rate, communication cost, wall clock time and the number of MapReduce rounds are proposed and proved. We observe that the algorithms vary in their performance guarantees and the predominant factor is either the number of votes or the number candidates; dependent on the respective method.

The experimental evaluation of the MapReduce algorithms shows that the number of edges or the density of the graphs is a very important factor for the performance of the algorithms. This yet needs to be further investigated.

Pregel Algorithms. For the following winner determination methods Pregel algorithms have been proposed: Smith set, Schwartz set, and the Schulze method.

The reason for the fact that there are more MapReduce algorithms than Pregel algorithms designed is that Pregel algorithms can only be used for graph based computations. The proposed Pregel algorithms are all making use of the idea of forward and backward min-label propagation in the graph. This method is then adapted to the specific problem. Besides the tuning of the Pregel procedure many logical considerations are comprised in the preprocessing and postprocessing procedures in the respective algorithms. The considerations vary depending on the problem statement and aim at shrinking the problem space very quickly.

In the experimental evaluation it is observed that the reduction of the problem space (reducing the number of possible candidates) works very well for the Schulze method.

Implementation. The implementation of the algorithms results in two open-source projects: `BigVoting` and `CloudVoting`.

`BigVoting` (`https://github.com/theresacsar/BigVoting`) contains the MapReduce algorithms implemented in Java on Hadoop.

`CloudVoting` (`https://github.com/theresacsar/CloudVoting`) is a Spark library containing many methods necessary for working with election data. The programming language used is Scala and the implementation makes use of the Spark library GraphX for the graph based computations. The package `CloudVoting` contains many functions for working with preference data: (1) functions for reading and writing election data in the widely used PrefLib format, (2) methods for converting the representation of election data (e.g. convert a weighted majority graph to a weak dominance graph), (3) functions for computing scores based on various scoring rules, (4) the Pregel algorithms presented in this thesis.

Experimental Evaluation. For the experimental evaluation the most involved algorithms are chosen as representatives. The evaluation shows that the algorithms scale very well and that they are practical on real-world data. In particular we observed, that the algorithms perform better on real world data, than on the generated synthetic data.

Concluding Remarks. In this work, we have presented several cloud computing algorithms for winner determination in computational social choice. The MapReduce algorithms are implemented using the programming language Java on Hadoop and the Pregel algorithms are implemented using the programming language Scala on top of Spark. The Pregel algorithm also make use of the Spark library GraphX. Frameworks such as Spark are quickly developing and offer more possibilities for algorithm optimization. For example for the graph based algorithm it is possible to use graph partitioning techniques for distributing the graph in the cluster and further speed up the computation. Spark also makes it possible to combine Pregel based computations with MapReduce computations and sequential operations seamlessly. From this follows that the actual performance of the algorithms highly depends on the implementation and the underlying framework and infrastructure on the used cluster.

Nonetheless, it remains necessary to formulate the algorithms using cloud computing paradigms. Only the correct implementation makes it possible for the underlying framework to efficiently execute the computation in the cloud system. The proposed algorithms are expected to be used in future implementations and other software packages combined with framework specific optimizations.

7.2 Open Issues and Directions for Future Work

The application of cloud computing techniques for methods in computational social choice is a new research area. Therefore, we encountered many possible new directions and open issues for future work.

Other Problems in Computational Social Choice. There are many other voting rules that can be investigated for their parallelizability. The winner determination problem by the Kemeny rule is NP-hard[BITT89, HSV05] and thus is unlikely to allow for practical parallel computation. However, heuristic algorithms [DKNS01, DK04] for computing the winners based on the Kemeny rule might be parallelizable.

Computational social choice is also dealing with other problems besides winner determination. Such topics include committee selection, judgement aggregation and problems of fair division. It would be interesting to investigate the parallelizability of problems in those areas.

Generate Random Preference Profiles. During the experimental evaluation we experienced that the generation of large synthetic datasets is a no-trivial task. Also in another studies on large election data (e.g. [MFG12]) it was highlighted that there is a lack of support for existing statistical models which are used to generate synthetic election data. We conclude, that there seems to be a need to gain more insight on the statistical properties of election data.
The most widely used method for generating election data is the Mallows model. The available implementations for random data generation by the Mallows model are not

suited very well for large scale election data generation, i.e. we experienced that the Mallows model reaches its limits for elections with more than 10,000 candidates. Creating new methods for the random data generation or devising new efficient algorithms for already existing methods, would be very beneficial for future work on large scale election data sets. Especially under the aspect that it is often difficult to obtain large scale real-world election data.

Experimental Evaluation. It would be interesting to perform larger experiments to identify characteristics in election graphs that influence the performance. In the experimental section it has already been observed, that the performance of MapReduce is influenced by the number of edges (or the density of the graph) and for the Pregel algorithms the number of strongly connected components has been identified as a highly influential factor. Generating large scale election data and performing experiments on those could give valuable insights on the behaviour of those algorithms and to identify graph properties influencing the performance.

Framework Specific Optimizations. The implementations of the algorithms can be tuned to the specific underlying framework. For example for the Pregel algorithm several methods of graph partitioning can be used to optimize the performance. Partitioning of data is relevant to algorithms running on Spark or Hadoop too.

Further Algorithm Optimization. The algorithms can be further optimized by applying other optimization approaches. For example the assignment of *id*s to the vertices is currently done at random, but it would be more beneficial to assign the ids by some scoring function, e.g. the Borda scores. With this approach the algorithms would converge faster.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AM12] Alnur Ali and Marina Meilă. Experiments with kemeny ranking: What works when? *Mathematical Social Sciences*, 64(1):28–40, 2012.

[ASSU13] Foto N. Afrati, Anish Das Sarma, Semih Salihoglu, and Jeffrey D. Ullman. Upper and lower bounds on the cost of a map-reduce computation. *Proceedings of the VLDB Endowment*, 6(4):277–288, 2013.

[AU10] Foto N Afrati and Jeffrey D Ullman. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology*, pages 99–110. ACM, 2010.

[BBF10] Yoram Bachrach, Nadja Betzler, and Piotr Faliszewski. Probabilistic possible winner determination. In *Proceedings of AAAI-10*, 2010.

[BBH16] Felix Brandt, Markus Brill, and Paul Harrenstein. Tournament solutions. In Felix Brandt, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel Procaccia, editors, *Handbook of Computational Social Choice*. Cambridge University Press, 2016.

[BBN14] Nadja Betzler, Robert Bredereck, and Rolf Niedermeier. Theoretical and empirical evaluation of data reduction for exact kemeny rank aggregation. *Autonomous Agents and Multi-Agent Systems*, 28(5):721–748, 2014.

[BF12] Markus Brill and Felix Fischer. The price of neutrality for the ranked pairs method. In *Proceedings of AAAI-12*, 2012.

[BFH09] Felix Brandt, Felix Fischer, and Paul Harrenstein. The computational complexity of choice sets. *Mathematical Logic Quarterly*, 55(4):444–459, 2009.

[BGN10] Nadja Betzler, Jiong Guo, and Rolf Niedermeier. Parameterized computational complexity of dodgson and young elections. *Information and Computation*, 208(2):165–177, 2010.

[BITT89] John Bartholdi III, Craig A Tovey, and Michael A Trick. Voting schemes for which it can be difficult to tell who won the election. *Social Choice and Welfare*, 6(2):157–165, 1989.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2013, New York, NY, USA - June 22 - 27, 2013*, pages 273–284, 2013.

[BKS14] Paul Beame, Paraschos Koutris, and Dan Suciu. Skew in parallel query processing. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 212–223. ACM, 2014.

[Bor07] Dhruba Borthakur. The hadoop distributed file system: Architecture and design. *Hadoop Project Website*, 11(2007):21, 2007.

[CDK06] Vincent Conitzer, Andrew Davenport, and Jayant Kalagnanam. Improved bounds for computing Kemeny rankings. In *Proceedings of AAAI-06*, pages 620–626, 2006.

[CELM07] Yann Chevaleyre, Ulle Endriss, Jérôme Lang, and Nicolas Maudet. A short introduction to computational social choice. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 51–69. Springer, 2007.

[CKKP14] Ioannis Caragiannis, Christos Kaklamanis, Nikos Karanikolas, and Ariel D Procaccia. Socially desirable approximations for Dodgson's voting rule. *ACM Transactions on Algorithms (TALG)*, 10(2):6, 2014.

[CLP18] Theresa Csar, Martin Lackner, and Reinhard Pichler. Computing the schulze method for large-scale preference data sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, Stockholm, Sveden*, 2018.

[CLPS16] Theresa Csar, Martin Lackner, Reinhard Pichler, and Emanuel Sallinger. Winner determination in huge elections with mapreduce. *10th Multidisciplinary Workshop on Advances in Preference Handling*, 2016.

[CLPS17a] Theresa Csar, Martin Lackner, Reinhard Pichler, and Emanuel Sallinger. Computational social choice in the clouds. *Datenbanksysteme für Business, Technologie und Web (BTW 2017)-Workshopband*, 2017.

[CLPS17b] Theresa Csar, Martin Lackner, Reinhard Pichler, and Emanuel Sallinger. Winner determination in huge elections with MapReduce. In *Proceedings of AAAI-17*, 2017.

[Cop51] Arthur H Copeland. A reasonable social welfare function. Technical report, mimeo, 1951. University of Michigan, 1951.

[CPSS15] Theresa Csar, Reinhard Pichler, Emanuel Sallinger, and Vadim Savenkov. Using statistics for computing joins with map reduce. In *CEUR Workshop Proc*, volume 1378, pages 69–74, 2015.

[CRX09] Vincent Conitzer, Matthew Rognlie, and Lirong Xia. Preference functions that score rankings and maximum likelihood estimation. In *Proceedings of IJCAI-09*, pages 109–115, 2009.

[Csa18] Theresa Csar. Cloudvoting: Analyzing preferences using spark and graphx. *11th Multidisciplinary Workshop on Advances in Preference Handling*, 2018.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DK04] Andrew Davenport and Jayant Kalagnanam. A computational study of the kemeny rule for preference aggregation. In *Proceedings of AAAI-04*, volume 4, pages 697–702, 2004.

[DKNS01] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of WWW-01*, pages 613–622. ACM Press, 2001.

[EST14] Michael Elberfeld, Christoph Stockhusen, and Till Tantau. On the space and circuit complexity of parameterized problems: Classes and completeness. *Algorithmica*, 71(3):661–701, 2014.

[Fis77] Peter C Fishburn. Condorcet social choice functions. *SIAM Journal on applied Mathematics*, 33(3):469–489, 1977.

[GHR95] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.

[HSV05] Edith Hemaspaandra, Holger Spakowski, and Jörg Vogel. The complexity of Kemeny elections. *Theoretical Computer Science*, 349(3):382–391, 2005.

[Joh90] David S. Johnson. A catalog of complexity classes. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A)*, pages 67–161. 1990.

[JSW+17] Chunheng Jiang, Sujoy Sikdar, Jun Wang, Lirong Xia, and Zhibing Zhao. Practical algorithms for computing stv and other multi-round voting rules. In *Proceedings of EXPLORE-17*, 2017.

[KKWZ15] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. *Learning spark: lightning-fast big data analysis*. " O'Reilly Media, Inc.", 2015.

[KW17] Holden Karau and Rachel Warren. *High Performance Spark*. " O'Reilly Media, Inc.", 2017.

[LPR+12] Jérôme Lang, Maria Silvia Pini, Francesca Rossi, Domenico Salvagnin, Kristen Brent Venable, and Toby Walsh. Winner determination in voting trees with incomplete preferences and weighted votes. *Autonomous Agents and Multi-Agent Systems*, 25(1):130–157, 2012.

[LRU14] Jure Leskovec, Anand Rajaraman, and Jeffrey D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.

[MAB+10] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of SIGMOD-10*, pages 135–146. ACM, 2010.

[McG53] David C McGarvey. A theorem on the construction of voting paradoxes. *Econometrica*, 21(4):608–610, 1953.

[MFG12] Nicholas Mattei, James Forshee, and Judy Goldsmith. An empirical study of voting rules and manipulation with large datasets. *Proceedings of COMSOC*, page 59, 2012.

[MW13] Nicholas Mattei and Toby Walsh. PrefLib: A library for preferences http://www.preflib.org. In *Proceedings of ADT-13*, pages 259–270, 2013.

[PX12] David C Parkes and Lirong Xia. A complexity-of-strategic-behavior comparison between schulze's rule and ranked pairs. In *Proceedings of AAAI-12*, 2012.

[San02] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial intelligence*, 135(1):1–54, 2002.

[Sch03] Markus Schulze. A new monotonic and clone-independent single-winner election method. *Voting matters*, 17(1):9–19, 2003.

[Sch11] Markus Schulze. A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36(2):267–303, 2011.

[SvZ09] Frans Schalekamp and Anke van Zuylen. Rank aggregation: Together we're strong. In *Proceedings of ALENEX-2009*, pages 38–51, 2009.

[SW14] Semih Salihoglu and Jennifer Widom. Optimizing graph algorithms on pregel-like systems. *Proceedings of VLDB-14*, 7(7):577–588, 2014.

[SW16] Robert Sedgewick and Kevin Wayne. *Algorithms (Fourth edition deluxe).* Addison-Wesley, 2016.

[Tid87] T Nicolaus Tideman. Independence of clones as a criterion for voting rules. *Social Choice and Welfare*, 4(3):185–206, 1987.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[XGFS13] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[YCX+14] Da Yan, James Cheng, Kai Xing, Yi Lu, Wilfred Ng, and Yingyi Bu. Pregel algorithms for graph connectivity problems with performance guarantees. *Proceedings of VLDB-14*, 7(14):1821–1832, 2014.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10:10–10, 2010.

Curriculum Vitae

Mag. Julia Theresa Csar, Bakk.

Homepage http://www.dbai.tuwien.ac.at/staff/csar/
Email csar@dbai.tuwien.ac.at

Date of Birth December 5th, 1987—Wels, Austria

Education

2011–2013 MAG.RER.SOC.OEC Master in Statistics, University of Vienna
2006–2011 BAKK. RER. SOC. OEC Bachelor in Statistics, University of Vienna
2006 MATURA Higher School Certificate at BG/BRG Dr. Schauerstraße 9, Wels

Working Experience

2014–today PRAEDOC UNIVERSITY ASSISTANT in the Database and Artificial Intelligence Group at the Technical University of Vienna. Besides doing research my responsibilities are also in teaching.

SS 2017 Teaching the course "Grundzüge Relationaler Datenbanken" (Databasesystems) at the Statistics Department of the University of Vienna.
SS 2014 Adjunct Professor for the Undergraduate Semester Course: MATH 1410/51 Introductory College Mathematics (Spring Semester) at the Webster University Vienna.

SS 2012 Tutor for the course "Evaluation und Assessment im Bildungsbereich" at the University of Vienna
WS 2011/12 Tutor for the course "System- und Modelltheorie" at the University of Vienna

2008–2010 PROJECT ASSISTANT at the University of Vienna, Institute of Scientific Computing
LOGI.DIAG—Test Driven Development and Condition Monitoring in Automated Systems
Development and implementation of a statistical compression algorithm that can be used for real time analysis and preventive maintenance. Implementation of functions used for condition monitoring.

Peer-reviewed Publications

2018 *Computing the Schulze Method for Large-Scale Preference Data Sets*
Theresa Csar, Martin Lackner, Reinhard Pichler
(to appear) IJCAI-18, Stockholm, Sweden

2018 *CloudVoting: Analyzing Preferences using Spark and GraphX*
Theresa Csar, 11th MPREF Workshop @ AAAI-18, New Orleans, USA

2017 *Computational Social Choice in the Cloud*
Theresa Csar, Martin Lackner, Reinhard Pichler, Emanuel Sallinger
PPI17 Workshop @ BTW 2017, Stuttgart, Germany

2017 *Winner Determination in Huge Elections with MapReduce*
Theresa Csar, Martin Lackner, Reinhard Pichler, Emanuel Sallinger
AAAI-17, San Francisco, USA

2016 *Winner Determination in Huge Elections with MapReduce*
Theresa Csar, Martin Lackner, Reinhard Pichler, Emanuel Sallinger
10th MPREF Workshop @ IJCAI-16, New York City, USA

2015 *Using Statistics for Computing Joins with MapReduce*
Theresa Csar, Reinhard Pichler, Emanuel Sallinger, Vadim Savenkov
Alberto Mendelzon Workshop, Lima, Peru

2010 *Prototyping Predictive Maintenance Tools with R*
Erich Neuwirth, Theresa Csar
R User Conference, Gaithersburg, Maryland, USA

Code projects available as open source

CloudVoting Spark library for Computational Social Choice (Scala)
`https://github.com/theresacsar/CloudVoting`

BigVoting MapReduce Algorithms for Computational Social Choice (Java)
`https://github.com/theresacsar/BigVoting`

Master's and Bachelor's Theses

2013 Master's Thesis: *Calculation of the transition density of allele frequencies in probabilistic models in population genetics*

2011 Bachelor's Thesis: *Measurements for Hurricane Intensity*
2011 Bachelor's Thesis: *Vorhersage des Risikos einer Rezidiverkrankung von Thrombosepatienten*
English title: risk estimation for recurrence of thrombosis disease