COMSOC Methods in Real-World Applications

Inaugural-Dissertation

zur Erlangung des Doktorgrades
der Mathematisch-Naturwissenschaftlichen Fakultät
der Heinrich-Heine-Universität Düsseldorf

vorgelegt von

Christian Laußmann
aus Wuppertal

Düsseldorf, April 2023

aus dem Institut für Informatik
der Heinrich-Heine-Universität Düsseldorf

Gedruckt mit der Genehmigung der
Mathematisch-Naturwissenschaftlichen Fakultät der
Heinrich-Heine-Universität Düsseldorf

Berichterstatter:

1. Prof. Dr. Jörg Rothe

2. Prof. Dr. Dorothea Baumeister

Tag der mündlichen Prüfung:
04. Juli 2023

Acknowledgements

I thank my dear mother Dagmar who raised me to be a decent and independent man. I also thank my father Tim for introducing me to science, opening my eyes for nature, and keeping me curious from childhood. Without you two, your strength, and your restless help in hard times this thesis would have been impossible. Thanks to my brother Dominik for the discussions with you, which have expanded my world view. Dear Julia, thanks for your mental support during the last stressful weeks that I have been working on this thesis.

I thank my advisor Jörg and my mentor Doro for the trust they put in me when supporting my fast-track doctorate. You always helped me with good advice and gave me confidence in my work and abilities. The many productive discussions I had with each of you were crucial to push my ideas forward. Without these discussions this work would not have been possible.

Further, I also want to thank my colleagues Linus, Robin, and Anna. Even during COVID restrictions and months of home office you integrated me in the department, and helped me uncountable times with all my questions. I highly enjoyed my coffee, lunch breaks, and the many intense discussions with you. I thank you also for proofreading this thesis, and giving me highly useful feedback!

Special thanks go also to our secretary Claudia. Thank you for always having my back when it came to formalities and applications, so that I was able to keep focus on research.

I also thank my co-authors for the professional and constructive joint work. Further, I thank Jérôme Lang for helping me and my co-authors with countless discussions on the design of frameworks and axioms.

Finally, thanks to the Deutsche Forschungsgemeinschaft (DFG) for supporting large parts of this thesis under grant RO-1202/21-1.

Der erste Schluck aus dem Becher der Naturwissenschaft macht atheistisch, aber auf dem Grunde des Bechers wartet Gott!

$\sim$ Werner Heisenberg

Vires in numeris.

Abstract

Computational social choice (COMSOC) is a research area focussed on the design and analysis of algorithms for collective decision tasks. In this thesis, we focus on one sub-area of the field: voting. We study four (near) real-world applications of voting. First, we show how voting can be used in networks to find central individual nodes, or sets of nodes. We show that network science, and voting theory overlap, and each can be informed by the other. Second, we study participatory budgeting. This is an application of voting where voters vote on projects they would like to fund, while external conditions such as time and cost limits must be satisfied. Participatory budgeting is not new to literature. However, we introduce a twist: project costs, and/or project durations are uncertain. This introduces additional difficulties to the planning process. The third part of this thesis deals with strategic campaigns in apportionment elections where an external agent tries to influence the election by bribing/convincing voters to change their vote. We show that computing optimal bribery actions is generally easy with an efficient algorithm we develop. This is arguably also a negative result as it means that apportionment elections could be susceptible to such campaigns. Thus, we also propose an extension to apportionment elections which makes the procedures computationally resistant to strategic campaigns. Lastly, we develop a new ballot format for multiwinner elections which helps voters state more sophisticated preferences with ease. More specifically, the ballots allow expressing approval, incompatibilities, substitution effects, and dependencies.

Zusammenfassung

Computational social choice (COMSOC) ist ein Forschungsbereich, der sich auf die Entwicklung und Analyse von Mechanismen zur kollektiven Entscheidungsfindung fokussiert. In dieser Doktorarbeit behandeln wir einen besonderen Bereich von COMSOC: Wahlen. Insbesondere behandeln wir vier Anwendungen von Wahlmethoden, die besonders nah an realen Anwendungen sind. Als erstes zeigen wir, wie Wahlmethoden verwendet werden können, um zentrale Knoten (oder Knotenmengen) in Netzwerken zu identifizieren. Unsere Ergebnisse legen nahe, dass Wahltheorie und Netzwerktheorie eine große gemeinsame Schnittmenge haben, und dass beide voneinander profitieren können. Als nächstes betrachten wir participatory budgeting (zu Deutsch: Bürgerhaushalte), wo Wähler über Projekte abstimmen, die in einer Stadt gebaut werden sollen. Dabei gibt es Beschränkungen bezüglich des Budgets und der Zeit, die für die Umsetzung der Projekte zur Verfügung stehen. Participatory budgeting ist nicht neu. Wir versuchen in dieser Arbeit aber, das Modell realistischer zu machen, indem wir unsichere Projektkosten und Realisierungszeiten erlauben. Durch die Unsicherheiten wird der Planungsprozess erschwert. Im dritten Teil dieser Doktorarbeit kümmern wir uns um strategische Wahlkampagnen in Parlamentswahlen. Hier versucht ein externer Agent, den Ausgang der Wahl (d.h. die Sitzverteilung) zu seinen Gunsten zu ändern, indem gezielt bestimmte Wähler überzeugt (oder bestochen) werden, ihre Stimme zu ändern. Wir entwickeln einen effizienten Algorithmus, der optimale Wahlkampagnen berechnen kann. Dies ist nicht unbedingt ein positives Ergebnis, da es möglicherweise die Manipulation einer Wahl erleichtert. Wir stellen daher auch eine kleine Erweiterung für parlamentarische Wahlsysteme vor, die die Berechnungskomplexität solcher Kampagnen signifikant erschwert. Zuletzt entwickeln wir ein neuartiges Stimmzettelformat, welches die Stimmabgabe bei komplizierteren Präferenzen in Komiteewahlen erleichtert. Unser Format ermöglicht die Angabe und Auswertung von u.a. Abhängigkeiten und Inkompatibilitäten zwischen Kandidaten.

Contents

Abstract **v**

1 Introduction **1**

1.1 Computational Complexity . . . 1

1.1.1 Computation Models . . . 2

1.1.2 Complexity . . . 9

1.2 Voting . . . 16

1.3 Central Problems and Questions . . . 19

1.4 Structure . . . 26

1.5 Notation . . . 26

2 Network Centrality Through Voting Rules **27**

2.1 Summary . . . 27

2.2 Introduction . . . 27

2.3 Centrality in Networks . . . 29

2.3.1 Centrality Indices . . . 29

2.3.2 Group Centrality And Node Selectors . . . 30

2.4 Voting in Networks . . . 32

2.5 Single-Winner Voting-Based Centrality Indices . . . 32

2.5.1 Axiomatic Relations . . . 33

2.5.2 Satisfaction Approval Voting . . . 34

2.5.3 Borda . . . 35

2.5.4 Copeland . . . 36

2.6 Multiwinner Voting-Based Node Selectors . . . 38

2.7 Experiments . . . 42

2.8 Conclusions . . . 44

2.9 Publication . . . 45

2.10 Personal Contribution . . . 45

3 Participatory Budgeting under Uncertainty **47**

3.1 Summary . . . 47

3.2 Introduction . . . 47

3.3 Preliminaries . . . 49

3.4 No Uncertainty . . . 52

3.5 Uncertain Duration . . . 53

3.6 Uncertain Cost . . . 53

3.6.1 Best-Effort Algorithms . . . 57

3.7 Everything Uncertain 60
3.8 Conclusions 62
3.9 Publication 63
3.10 Personal Contribution 63

4 Strategic Campaigns in Apportionment **65**
4.1 Summary 65
4.2 Introduction 65
4.3 Apportionment 66
4.3.1 Computing Support Allocations 67
4.3.2 Apportioning Seats 68
4.4 Bribery 69
4.5 Classical Top-Choice Mode 70
4.6 The Second-Chance Mode 76
4.7 Conclusions 78
4.8 Publication 79
4.9 Personal Contribution 80

5 Designing More Expressive Ballots for Multiwinner Elections **81**
5.1 Summary 81
5.2 Introduction 81
5.3 Preliminaries 83
5.4 Bounded Approval Ballots 83
5.5 Scoring with Bounded Approval Ballots 85
5.5.1 Axiomatic Analysis 87
5.5.2 The Perfect Scoring Rule 92
5.6 Expressiveness Comparison to Approval Ballots 94
5.7 GoodVotes Web-Application 96
5.8 Conclusions 98
5.9 Publication 99
5.10 Personal Contribution 99

6 Discussion **101**

Bibliography **103**

Appendix **109**
1 Code of Centrality Experiments 109
2 Code of Best-Effort Experiments 115
3 Omitted Figures of Best-Effort Experiments 123

Erklärung **129**

CHAPTER 1

Introduction

This thesis touches several areas of computer science. The two most important areas that shine through in every chapter are *computational complexity* and *voting*. These two shall be introduced in this chapter. Topics that are only relevant for specific chapters (e.g. networks, bribery, budgeting, etc.) will be introduced in the respective chapters. Further, this chapter describes the central problems this thesis deals with (Section 1.3).

The goal of this introduction is twofold. First, for the non-specialist reader this introduction provides easily understandable explanations of the two mentioned topics, supported with several examples. Second, it provides a deeper insight in the relations of the various topics this thesis deals with.

1.1 Computational Complexity

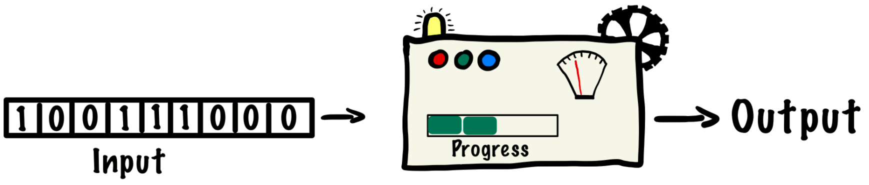
For a more detailed introduction to computational complexity we refer to the books by Arora and Barak [1], Rothe [74], [76], and the lecture by Rothe [75]. This section is based on these works, whereby we simplify the definitions and formalism as much as possible, and add several illustrative examples. It should be noted that the content of this section is more or less common knowledge in computer science. So the reader who is already familiar with theoretical computer science can continue with Section 1.2.

A central question of theoretical computer science is *'how hard is it to actually compute something, and what capabilities of a machine (or algorithm) are necessary to do so?'* It may seem that for this thesis mainly the first part of the question is relevant. We are interested in how hard it is to compute something at several places in the thesis. However, the first and second part of the question are actually inextricably linked. You cannot know how hard it is to tackle a problem if you don't know what tools are available. To give an illustrative example, we can consider digging a well with a teaspoon to be significantly harder (if not impossible) than digging the well with a backhoe. As we will see, computation machines have different abilities. Some are more on the 'teaspoon-end', and some are on the much more sophisticated 'backhoe-end'.

We will focus on the second part of the question first: *what algorithmic model is needed to solve problems in this thesis at all?* Even if this is of minor relevance for the problems of the later chapters, in Section 1.1.1 we make a short excursion into the theory of computability. We try to be as formal as necessary, but abstract away from formal details where possible. With what we learn there, we can then adequately define and understand the first part of the question in Chapter 1.1.2: *how hard is it to solve the problems addressed in this thesis?*

1.1.1 Computation Models

Before we define the first formal computation models (in the form of theoretical machines), let us first think about the general process of computation. What do we understand as *computation*? Computation tasks can be very different. We can compute the value of a mathematical function. As another computational task, we want to verify if a file is properly formatted. Also, video editing, compiling a program, encrypting files, and brewing coffee are computation tasks (at least for modern fancy coffee machines evaluating several sensors). What do all such tasks have in common? First, we have some *input*. Depending on the computation problem this can be a number, a file, or user and sensor input. We throw the input into a machine that is designed for dealing with that problem. Finally, we receive some *output* as the answer to the problem's question.



It makes sense to define exactly what the input may consist of. For instance, the machine above might know how to deal with the symbols 0 and 1 but gets stuck if the input contains the symbols $\triangle$ or $\spadesuit$. Just as a digital computer knows how to deal with current on (digit 1) and current off (digit 0), but has no chance to deal with an imaginary memory brick encoding everything by the colors blue, red, and green, or encoding everything by potatoes, tomatoes, apples, and bananas. It is not necessary that machines work with zeros and ones. Encoding everything with a set of colors or fruit is theoretically possible as well, as long as the machine is capable of reading these symbols.[1] We define the *alphabet* of a machine—the symbols it can read—as a set Σ. Let Σ^* be the set of all sequences of symbols from Σ, that is, the set of all possible inputs for the machine. In case of the previous machine we have $\Sigma = \{0,1\}$, so $\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, 000, \dots\}$ where λ denotes the empty input. Note that Σ^* is an infinite set, and inputs for the machine can be arbitrarily long (but finite).

Also, the output of a machine must be defined formally. This is of course directly linked to what kind of question is asked in the respective problem. Most of the time in complexity

[1]Zeros and ones are simply the easiest symbols to read and store with electronic parts.

theory it is assumed that the output can only be YES or NO (for example as in the task of verifying if a file is properly formatted). We call such a computation problem a *decision problem*. This comes from the theory of *formal languages*. We don't want to introduce it in detail. For short, a language is a subset of Σ^*. Such a subset may be finite or not. For instance, an infinite language is the set of all elements from Σ^* which contain an even number of the symbol 1. If the corresponding machine outputs YES, the input was part of the language (we say the machine *accepts* the input). Otherwise, the input was not part of the language (we say the machine *rejects*). Although this doesn't seem to be a powerful framework at first, it is possible to define a large variety of problems with formal languages because the rules whether an input is accepted or not can be arbitrary complex. For instance, we could define the language 3-MOVES-CHECKMATE which contains only chess positions where it is white's turn, and white can win in at most 3 moves regardless of what black does. Equivalently, we could define it as the following decision problem.

3-MOVES-CHECKMATE	
Given:	A chess position where it is white's turn.
Question:	Is it possible for white to win in at most 3 moves regardless of what black does?

We can define a machine to also output something other than YES or NO (more precisely, to output additional information on accepting or rejecting). Such a machine is then capable of computing not only decision problems but for instance also *optimization problems* or *search problems*. However, since there exists no common definition of the output over all computation models, we will explain how that works after defining the models. Further, in this thesis we will most of the time be interested in the complexity of decision problems, i.e., YES/NO outputs are more relevant in this work. Note that many problems which require other output than just YES or NO have a sibling problem which is a proper decision problem. For example, the problem to compute the value of $f(x) = x^5 - 3$ has a sibling decision problem which asks if the value $f(x)$ is greater than a given number y.

In order to compute something, the machine must execute some algorithm. An *algorithm* is a finite sequence of instructions. What exactly an *instruction* is, depends on what capabilities the machine has, that is, on the computation model. It is too short-sighted saying that *compute* $f(x)$ is an instruction. We need to make it more formal in order to really know what the machine can compute. In the following we define some common computation models. Note that we often use the word *computation machine* and *algorithm* interchangeably, as the machine informally is a formal model of an algorithm.

Finite State Automaton

What is the minimal requirement for an algorithm to run at all? The two most basic requirements are (1) to read the input, and (2) to react based on it. We can achieve both with instructions of the following format for an alphabet $\Sigma = \{a_1, a_2, \dots, a_n\}$. Note that this type of instruction is similar to a *switch* in many programming languages.

`Line W:` read next symbol A and

- go to program line X if $A = a_1$;
- go to program line Y if $A = a_2$;
- $\dots$;
- go to program line Z if $A = a_n$

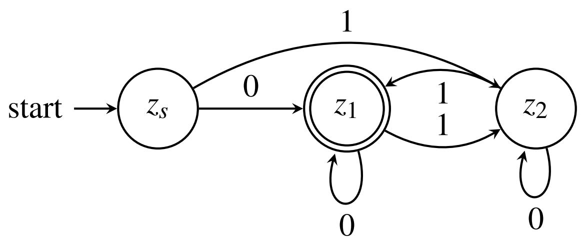
This instruction format is implemented by a *(deterministic) finite state automaton (a.k.a. deterministic finite automaton, or DFA for short).* A DFA is defined by the tuple $(\Sigma, Q, \delta, q_0, F)$ where Σ is the alphabet, Q the set of states, $\delta : Q \times \Sigma \to Q$ the transition function, q_0 the initial state, and $F \subseteq Q$ is the set of final states. Note that Q and F are finite sets (thus the name *finite state* automaton). In each step of its computation, the DFA reads the next symbol from the input, looks up the current state, and determines in which state it must change. Informally, the states correspond to a program counter which stores which program line is executed next. DFAs cannot read the input twice. It reads one symbol (from left to right) from the input in each step. The DFA accepts an input if and only if the complete input is consumed, and it stops in a final state from F. For a better visibility, we can represent such automatons graphically. Consider the following DFA which accepts non-empty inputs which contain an even number of the symbol 1.

Example:

Let $M = (\Sigma, Q, \delta, z_s, F)$ with $\Sigma = \{0, 1\}$, $Q = \{z_s, z_1, z_2\}$, $F = \{z_1\}$, and

δ	z_s	z_1	z_2
0	z_1	z_1	z_2
1	z_2	z_2	z_1

We can represent M as follows where the final states are indicated by a second outer circle, and the arrows denote to which state the DFA changes when consuming the labelled symbol.

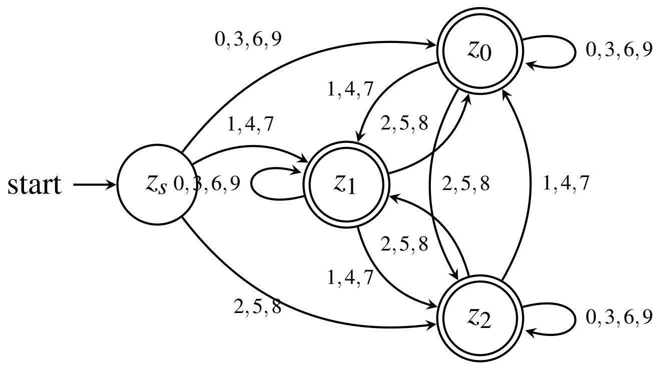

$$\begin{tikzcd}
\text{start} \arrow[r] & z_s \arrow[r, "0"] \arrow[rr, bend left, "1"] & z_1 \arrow[r, bend right, "1"'] \arrow[loop below, "0"] & z_2 \arrow[l, bend right, "1"'] \arrow[loop below, "0"]
\end{tikzcd}$$

Say we are given the input 101. The automaton M starts in z_s, reads 1, changes to z_2, then reads 0 and stays in z_2, and finally reads 1 and changes to state z_1. Since z_1 is a final state and the input is fully consumed, the automaton accepts. Given 111 as input, the automaton ends in z_2 which is not a final state, so it rejects.

Apart from some YES/NO decisions, a DFA is also capable of a very limited number of additional outputs. We can interpret the final states F as outputs. This enables us for instance to construct an automaton which computes the function $f(x) = x \mod 3$.

Example:

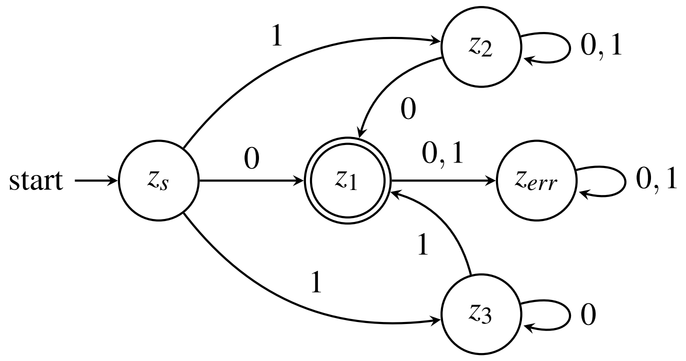
The following DFA for the alphabet $\Sigma = \{0,\dots,9\}$ computes the value $x \mod 3$. If it halts in z_0, we have $x \mod 3 \equiv 0$. If it halts in z_1, we have $x \mod 3 \equiv 1$. And if it halts in z_2, we have $x \mod 3 \equiv 2$.



A simple extension to a DFA is to introduce *nondeterminism*. The only change in a *nondeterministic finite automaton (NFA)* compared to a DFA is that the transition function is now defined as $\delta : Q \times \Sigma \to 2^Q$ where 2^Q is the powerset of Q. Informally, for each computation step there can be multiple options what to do next. Whenever there are multiple options, we call the step a *nondeterministic transition*. We can think of nondeterministic transitions as if all options are taken simultaneously. This forms a computation tree (see next example) where with each nondeterministic transition a new branch is created. The NFA accepts if and only if there exists at least one accepting path in the computation tree.

Example:

Consider the following NFA:



Say we are given 1010 as input. We can represent the automaton's computation by the following tree.

$$\begin{tikzcd}
 & & & z_2 \arrow[r] & z_2 \arrow[r] \arrow[dr] & z_2 \\
 & & z_2 \arrow[ur] \arrow[dr] & & & z_1 \\
1010 \arrow[r] & z_s \arrow[ur] \arrow[ddr] & & z_1 \arrow[r] & z_{err} \arrow[dr] & \\
 & & & & & z_{err} \\
 & & z_3 \arrow[dr] & & & \\
 & & & z_3 \arrow[r] & z_1 \arrow[r] & z_{err}
\end{tikzcd}$$

There is one accepting path in the computation tree (the one that ends with z_1). Thus, the input 1010 is accepted.

It is well known that DFAs and NFAs can accept exactly the same formal languages. Obviously, a language accepted by a DFA can also be accepted by a NFA, since each DFA is a valid NFA which makes no use of its nondeterminism. Further, by a construction due to Rabin and Scott [72], often referred to as the *powerset construction*, we know that also for each NFA N we can create a DFA which accepts the same language as N (although this DFA may sometimes have exponentially more states than N).

However, both DFAs and NFAs are rather weak computation machines. For instance, they will fail to decide the following language over $\Sigma = \{0,1\}$ already: $L_1 = \{0^n 1^n \mid n > 0\}$ where a^n is the n-times concatenation of $a \in \Sigma$. That is, it cannot even decide if an input which starts with a sequence of zeros and ends with a sequence of ones has the same number of ones and zeros (compare Rothe [75, p. 41]). The reason they fail is that they cannot memorize how many ones and zeros were consumed already. Well, they can memorize that to some extent by storing this using the states. But since the set of states is finite, it is not possible to store arbitrary numbers there. This works only if $n < k$ for some constant $k \in \mathbb{N}$.

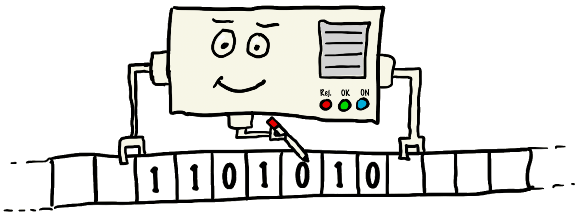
Turing Machine

The simplest type of memory we can add to an automaton is a stack. Things written to the stack are *pushed* on top of the stack; reading from the stack means to *pop* the top element from the stack. By adding a stack to an NFA we get a *pushdown automaton (PDA)*. There exist also *deterministic pushdown automatons (DPDA)*.[2] PDAs can due to their simple memory

[2]Other than with DFAs/NFAs, deterministic PDAs can decide only a strict subset of the languages a nondeterministic PDA can decide. See Rothe [75, p. 85] for an example.

decide more sophisticated languages than DFAs and NFAs. Nevertheless, most languages are still undecidable by a PDA. This is because the pop operation destroys information.

It turns out that this problem can be solved by introducing a second stack to which popped symbols from the first are pushed, and vice versa, symbols popped from the second stack are pushed to the first. This way we lose no information anymore. A machine like this is called a *Turing machine*, named after its inventor Alan Turing [85]. An equivalent and more common and intuitive definition of a Turing machine is as follows. The machine has a finite set of states and an unlimited working tape divided in cells to write symbols on and read symbols from (one symbol per cell).[3] The machine can only read one cell at a time. To see the other cells, the tape can be moved one cell left or right in each computation step. The input can be given as a separate tape (then the working tape is empty initially), or it is given directly on the working tape (starting at the position the machine reads in its first step). Both definitions are equivalent in terms of what languages the machine can accept, so for simplicity we assume that the input is given on the working tape.



The machine can read from the working tape, and then decides what to do based on its current state. It can (all in one step) change state, write on the tape (or leave it as is), and move the tape left or right (or leave it as is). At the end of the computation the machine can either enter an accepting state to answer YES, or it ends in a rejecting state to answer NO. Entering an accepting or rejecting state will immediately end the computation—other than the previous machine types DFAs, NFAs, DPDAs, and PDAs. Note that these machines had no need for explicit rejecting states. This is because their computation ended automatically when the input was fully consumed. This is not the case for Turing machines, as they can read the input back and forth. Actually, Turing machines can also run forever in infinite loops.[4] This makes it necessary to have states which indicate the end of a computation.

By the way, with the end of the computation the Turing machine can not only answer YES or NO. By writing an answer on the tape, it can output arbitrary complex computation results such as numbers, chess positions, or even encoded images. One just has to define how the output is formatted or interpreted.

[3]The machine can also have multiple working tapes, but essentially this only increases efficiency by factor two, and has no effect on what the machine can compute (see Rothe [74, p. 24]).

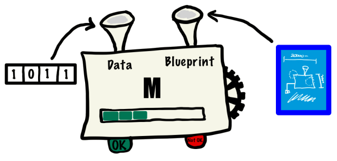
[4]There exist also definitions of Turing machines where an infinite loop entirely replaces the rejecting state, i.e., either the machine answers YES, or it doesn't answer at all (see e.g. the book by Rothe [74, p. 25]).

Determinism and nondeterminism can also be distinguished for Turing machines. A *nondeterministic Turing machine (NTM)* can have multiple options what to do next, i.e., it can have branches in its computation similar to the NFA in Example 1.1.1—but instead of just having multiple succeeding states it can also be nondeterministic with regard to the movement of the tape or which symbol to write on it. It accepts if and only if there is at least one accepting path. A *deterministic Turing machine (DTM)* is not capable of nondeterministic branches. However, it is just as powerful as a nondeterministic Turing machine because a DTM can simulate the execution of a NTM. Therefore, it just has to run every path in the computation tree of the NTM in sequence until it finds an accepting path (entering a rejecting state is replaced by starting the computation of the next path). The problem is that (unless P = NP, a question we will discuss later), in general we need up to exponentially more computation steps to simulate a NTM with a DTM.

According to the Chruch thesis, Turing machines are capable of computing every function that is *effectively calculable* (see [75, p. 115] for a historic context). Due to the lack of a formal definition of what is 'effectively calculable', the thesis remains unproven. However, it was proven for every formal algorithmic model ever proposed that Turing machines are at least as powerful. But can they compute everything? Surprisingly, they cannot. The following example is known as the *Halting Problem* (see [1, p. 22], [75, p. 155]), and shows that even Turing machines (but also all other algorithmic models) have only a limited computation power.

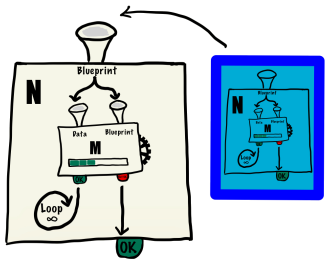
Example:

As mentioned earlier, Turing machines can run in infinite loops. Thus, an important problem is that as long as a Turing machine doesn't halt (accepting or rejecting the input), we don't know if it still computes or is simply stuck and will never give us an answer. It would be nice to have a machine which—given the blueprint[a] of another Turing machine as well as an input for this machine—decides whether the machine on the blueprint will ever halt if it is given that input. Imagine such a machine—let's call it M—exists.



If M answers 'OK', the machine from the blueprint will eventually halt on the input. If M answers 'Not OK', the machine from the blueprint runs forever.

Now consider that we have another machine N which receives a blueprint of a Turing machine, copies it, and puts both copies into M. If the result is 'OK', N enters an infinite loop, otherwise it answers 'OK'.



Let's see what happens when we give N its own blueprint. Both copies are given into M which now simulates what happens when N receives N as input—exactly what we have done. If M comes to the conclusion that N halts on input N, it answers 'OK' which makes N never halt. Vice versa, if M comes to the conclusion that N does not halt on input N, it answers 'Not OK' which makes N halt. This paradox shows that M cannot exist, i.e., it is impossible to compute for every given Turing machine whether it halts on a specific input.

[a]Note that such a blueprint can easily be encoded to fit on an input tape by assigning it a Gödel number, a process named after the mathematician Kurt Gödel.

Summary

As we see, the computation model (i.e., the abilities of the machine) has a huge impact on what the machine can compute. Memory plays a crucial role. With the highly limited memory of DFAs/NFAs, we are unable to count. Some counting ability is available when we have a stack. However, a stack loses information when we pop symbols from it, which limits the ability of a machine using it. Turing machines with their unrestricted memory are finally able to compute everything that intuitively is computable. Also, determinism and nondeterminism can make a difference in what the machine is capable of computing. While for finite state automata, as well as for Turing machines, the determinism is not a restriction, for pushdown automata it makes a difference in terms of what they can compute (Rothe [75, p. 85]).

In the following we will only deal with DTMs and NTMs. DTMs are close to what our modern computers are. The only difference is that DTMs have unlimited storage due to their unlimited tape. However, modern computers have enough storage for reasonable instance sizes, so DTMs are a sufficiently good model. NTMs are (to the state of current research) impossible to build. Although modern computers are capable of parallelization, we cannot consider them as NTMs because an NTM's computation tree can branch arbitrary often. Not even a high-tech chip with 32 or more computing cores can compete with that. But as we have seen, DTMs are just as powerful as NTMs. So it should not be a problem that we have only DTMs at hand, right? Well, in the next section we will see that although both are capable of computing the same type of problems, DTMs are probably much, much slower.

1.1.2 Complexity

Computational complexity is a measure how many resources are required to solve a problem algorithmically. Regarding Turing machines, the resources of interest are *time* (computation steps) and *space* (cells on the working tape). In this work we are only interested in the resource *time*. However, note that a Turing machine can never use more space than it performs computation steps because it requires a step to write something to a cell. Thus, time complexity is always an upper bound to space complexity. For a deterministic Turing machine which is given an instance of a problem we define the *time requirement* to be the number of computation steps it performs until its computation is done (i.e., until it accepts or rejects). We assume that a deterministic Turing machine always halts. For a nondeterministic Turing machine we define the time requirement to be the number of computation steps on the shortest accepting path if such a path exists. If the nondeterministic machine never halts, the time requirement is undefined.

Resource requirement of a machine or algorithm usually depends on the input size. Suppose you want to check a graph property such as whether it is bipartite.[5] It is easy to imagine that the more edges and nodes the graph has (i.e., the larger the input), the longer it takes to verify if it is bipartite. In fact, the resource requirement of an algorithm for this problem can become arbitrarily high by giving it an arbitrarily large graph. Essentially this holds for most reasonable algorithms. Thus, it makes sense to define the resource requirement of a specific algorithm as a function of the input size to make the resource requirements of different algorithms comparable. Further, it might be that an algorithm is very fast on some instances, and slow on others. When we speak of the resource requirements of an algorithm we always refer to the maximum resource requirement (on worst-case instances) in relation to the input size. Note that we usually don't want to know the maximum resource requirement in relation to the input size exactly, but rather how fast it grows asymptotically. Thus, it is common to use *big O notation* introduced by Landau [56, p. 31]. Writing $f(x) \in \mathcal{O}(g(x))$ means that $f(x)$ grows asymptotically not faster than $g(x)$. Formally,

$$f(x) \in \mathcal{O}(g(x)) \iff \limsup_{x\to\infty} \frac{|f(x)|}{g(x)} < \infty.$$

However, computational complexity of a *problem* should not depend on resource requirement of just one specific algorithm that solves it. Maybe we were just too stupid to find a better algorithm. Later, when someone finds a better algorithm, the complexity of the problem would change. To avoid this, we define the complexity of a problem to be the (maximum) resource requirement of the best algorithm[6] that solves it. Note that for determining the complexity of a problem we need to show (1) that there exists an algorithm which solves the problem with at most the specified resources, and (2) that there exist no algorithm solving the problem with less resources. This means that finding an algorithm to solve a problem always provides just an upper bound on the complexity of a problem. It remains to show that there is also no better algorithm—this is often the harder task.

Complexity Classes

Often, we want to classify a problem's complexity by a *complexity class*. Complexity classes are sets of problems which can be solved with at most a given resource requirement. The

[5]Informally, bipartiteness means that we can partition the set of nodes of a graph into two sets such that edges exist only between the two sets but not within a set.

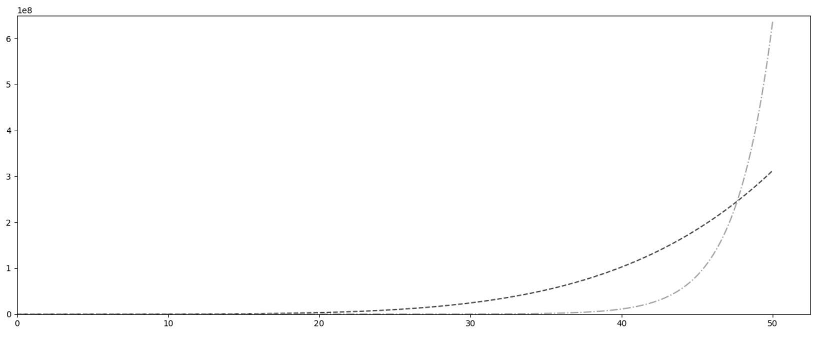
[6]Best algorithm in the sense of *having the lowest resource requirements*.

complexity classes relevant for this thesis will be introduced in the following.

When we want to compute something, (time-) efficiency plays a crucial role. We don't want to wait arbitrarily long before getting an answer when executing an algorithm. It is common to say that a problem can be solved *efficiently* if it can be solved by a deterministic Turing machine in polynomial time. For decision problems, these problems are captured in the complexity class P.

Definition 1. *A decision problem Q belongs to the complexity class P if and only if there exists a deterministic Turing machine M and a polynomial* $f : \mathbb{N} \to \mathbb{N}$ *such that for every instance I of Q machine M always gives a correct answer and always requires at most* $f(|I|)$ *computation steps, where* $|I|$ *is the input length, i.e., the number of tape cells occupied by I.*

Problems outside P are considered *intractable*. This is because these problems require a deterministic Turing machine to run for an exponential (in the input size) number of computation steps. While it is still possible that for a small input size both, polynomial and exponential time are tractable, the growth of an exponential function is asymptotically so much faster that it overtakes every polynomial for larger input sizes. Consider the following example of a polynomial $f(x) = x^5$ in comparison to the exponential function $g(x) = 1.5^x$. Even though $f(x)$ (the dashed dark line) is larger than $g(x)$ (the light dash-dotted line) for smaller x, at some point $g(x)$'s growth literally explodes and overtakes $f(x)$.



Note that polynomial runtime doesn't really mean that we can solve a problem fast in practice. An algorithm which runs in $\mathcal{O}(x^5)$ cannot be considered really fast. But it is still more efficient than the exponential time algorithm for larger instances. Nevertheless, one shouldn't be too dogmatic with distinguishing between efficient and inefficient by polynomial and exponential runtime as one can of course construct an algorithm with a huge polynomial runtime (such as $\mathcal{O}(x^{100})$) that is slower than typical exponential runtime algorithms on every reasonable instance size (compare Rothe [76, p. 71]). However, for the large majority of problems which can be solved in polynomial time, the polynomial runtime of the best known algorithm has a relatively small degree, and can thus be executed rather fast [76, p. 72].

Until now, we have only talked about deterministic Turing machines. But for this thesis, problems that can be decided by a nondeterministic Turing machine in polynomial time are interesting, too. The class of these problems is called NP (for *nondeterministic polynomial time*). In practice, talking about nondeterministic algorithms is rather unintuitive. Thus, we present a commonly used equivalent definition for the class NP.

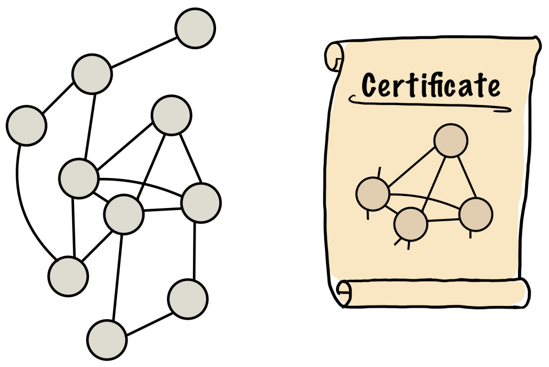
Definition 2. *A problem A belongs to the complexity class NP if and only if there exists a deterministic Turing machine M taking an instance I of the problem A and a certificate X as input such that*

1. *M runs in polynomial time, and*
2. *if and only if I is a* YES *instance for problem A, there exists a certificate X of polynomial length (w.r.t. I) such that M answers* YES *given I and X as inputs.*

Informally, a problem belongs to NP if and only if for every YES instance I we can provide an easily verifiable certificate X witnessing that I is a YES instance. This definition captures exactly the class of problems that can be decided by a nondeterministic Turing machine in polynomial time because the nondeterministic Turing machine can nondeterministically guess a certificate in polynomial time, and then verifies it in polynomial time (or finds that it is invalid). To illustrate how certificates work, consider the CLIQUE problem.

CLIQUE	
Given:	A graph $G = (V,E)$ and an integer K.
Question:	Is the clique number $\omega(G)$ at least K? That is, is there a set of nodes $V' \subseteq V$ with $\lvert V' \rvert \geq K$ where every node $v \in V'$ has an edge to every other node in V', i.e., for all $v,w \in V', v \neq w$ holds $(v,w) \in E$

If there is a clique V' of size at least K in the graph, we can provide the vertices of V' as a certificate. This certificate has only polynomial length w.r.t. the graph G (it is actually even smaller than G), and is verifiable in polynomial time by checking if all nodes specified in the certificate are indeed adjacent to each other. Thus, CLIQUE belongs to NP.



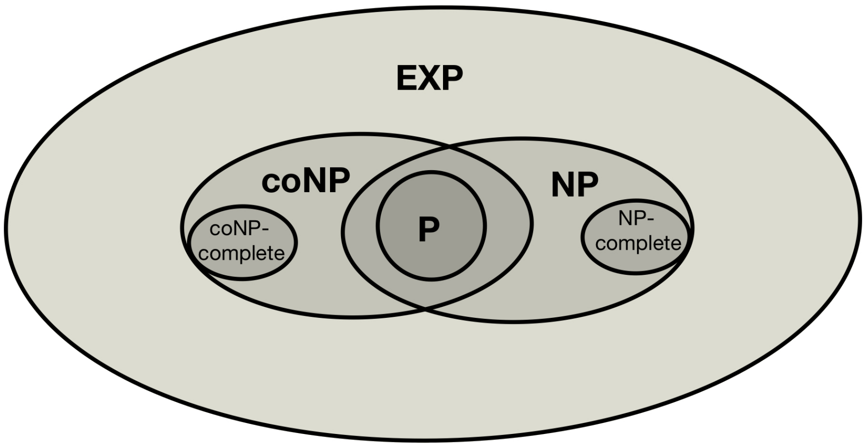


Figure 1.1: Relations of the complexity classes described in this chapter. It is known that P is a proper subset of EXP. However, the other inclusions are believed to be strict, but it is not known whether they are.

We can analogously define the complement class of NP, namely coNP, which contains those problems for which every NO instance can be easily verified with a certificate. The class coNP is not relevant in the other chapters, so we don't formally define it. However, note that CLIQUE does not belong to coNP (unless NP = coNP) because it is impossible to find a polynomial length certificate that a graph has no clique of size at least K. It is not sufficient to show that there exists *one* set of nodes of size K which is *not* a clique. We would have to show that *all* sets of nodes of size at least K are no cliques—but this requires a too large certificate since there are exponentially many subsets of nodes.

NP and coNP both are subsets of EXP—the class of problems which can be solved by a deterministic Turing machine in exponential time (recall that DTMs can simulate NTMs). Further, P is a subset of both NP and coNP. For problems in P the certificate can simply be empty, because a Turing machine can instead of verifying the certificate in polynomial time simply compute the solution in polynomial time. We illustrate the relations of the classes P, NP, coNP, and EXP in Figure 1.1. Note that there are also two subsets called *NP*- and *coNP-complete*. These sets of problems will be relevant in the next subsection. Further, note that we omitted many interesting complexity classes such as the *polynomial hierachy* and the *boolean hierachy*. These classes are not relevant for this thesis. However, the interested reader can learn more about these complexity classes and hierarchies in the book by Rothe [74].

Reductions

In the following we assume P $\neq$ NP (see the next subsection for an explanation). For showing that a problem is in P it is sufficient to provide a deterministic algorithm solving the problem in polynomial time. Similarly, for showing that it is in NP, it is sufficient to provide a nondeterministic algorithm solving the problem in polynomial time, or equivalently, showing that one can verify YES instances deterministically in polynomial time given a certificate. However, since the distinction of NP and P is so important, it is insufficient to just show that the problem is in NP. It is important to also show that it is either in P, or that it is not in P. Here, the class of NP-complete problems comes into play. The class of NP-complete problems is a subset of NP, containing *the hardest problems* in NP. Informally, this is captured as follows. A problem A cannot be easier than a problem B if we can easily translate an instance of B into an instance of A such that A gives us the same answer to the translated instance as B would to the original instance. In other words, if we can easily solve instances of B using an algorithm for A, problem B can be at most as hard as A. This process is called a *reduction*.

Definition 3. *We say a problem B is polynomial-time many-one reducible to a problem A (written as $B \leq_m^p A$) if and only if there is a function f computable by a deterministic Turing machine in polynomial time such that $f(x)$ is a* YES *instance for A if and only if x is a* YES *instance for B.*

The idea is that within NP, deterministic polynomial extra time requirement is acceptable, as it won't change the complexity. Note that the *polynomial-time many-one reduction* is transitive in the sense that if $B \leq_m^p A$ and $C \leq_m^p B$ then also $C \leq_m^p A$. This is easy to see: since $C \leq_m^p B$ we can translate an instance of C in deterministic polynomial time into an instance of B, and since $B \leq_m^p A$, we can then take another deterministic polynomial time computation for translating it into an instance of A. All in all, this can still be done in deterministic polynomial-time. Note that for other complexity classes (e.g. for subclasses of P) the polynomial-time many-one reduction is too rough. For those classes exist different reductions [76, p. 91].

Definition 4. *A problem A is* hard *for a complexity class $\mathcal{C}$ w.r.t. a specific reduction (here: polynomial-time many-one reduction) if and only if every problem from $\mathcal{C}$ can be reduced to A. Additionally, if A belongs to $\mathcal{C}$, we say A is* complete *for $\mathcal{C}$.*

So the class of NP-complete problems contains the problems to which all problems from NP can be reduced by polynomial-time many-one reductions. If we can ever solve one of those problems with a polynomial-time algorithm, we can also solve every other problem from NP in polynomial time. Note that all problems which are NP-complete can also be polynomial-time many-one reduced to each other. This yields the following statement.

Corollary 1. *Let A be a problem we want to show to be NP-complete. Let B be a problem we already know is NP-complete. Then A is NP-complete if and only if (1) A is in NP, and (2) B is polynomial-time many-one reducible to A (i.e., $B \leq_m^p A$).*

We will use the statement above several times in this thesis to show that some problems we define are indeed hard to solve. More specifically, by showing that a problem is NP-complete we can show that there exists no efficient deterministic algorithm to solve it. As we will see, sometimes this is a desirable result (e.g. because it makes manipulations hard), and sometimes it is an undesirable result (e.g. we would like to compute something, but it is too resource-intensive). However, for reductions we first need an NP-complete problem to reduce from. We now introduce the HITTING SET problem (HS for short), which was shown to be NP-complete by Karp [50]. We will use HS for showing the NP-completeness of problems in this thesis.

HITTING SET	
Given:	A set $U = \{u_1, \dots, u_p\}$, a collection $S = \{S_1, \dots, S_q\}$ of nonempty subsets of U, and an integer K.
Question:	Is there a hitting set $U' \subseteq U, |U'| \leq K$, i.e., a set U' such that $U' \cap S_i \neq \emptyset$ for each $S_i \in S$?

For HITTING SET instances, we assume that $1 \leq K \leq \min\{p,q\}$. Otherwise, the problem would be trivial: For $K \leq 0$ the subset U' must be empty, which renders every intersection with it empty. For $K \geq \min\{p,q\}$ we could either take the whole set U as U', or at least one item from each S_i into U'. In both cases, all intersections are always trivially nonempty.

P vs. NP

The *P vs. NP question* is one of the most prominent and important open problems in computer science. The question is whether P and NP are actually the same classes, or whether NP is a proper superset of P, i.e., that there are really problems which are in NP (solution is easily verifiable with a certificate) but not in P (solution can be computed efficiently). While it is commonly assumed that NP is a proper superset of P, no one was able to prove it, yet. But at the same time no one was ever able to find an efficient algorithm for an NP-complete problem—which would prove P and NP to be the same class. Note that to show P = NP it is indeed sufficient to show for at least one NP-complete problem that it is in P. This is because every problem from NP can, by definition, be reduced to this problem, and since this problem is in P, all problems from NP would be, too.

The implications of the P vs. NP question reach very far. If someone is ever able to prove the equality of P and NP, the complexity-theoretic parts of this thesis would become more or less irrelevant. However, the implications of P = NP reach much further. A lot of complexity-theoretic works of the last decades would be pointless. For example, the security of RSA relies on the assumption that INTEGER FACTORIZATION is in NP but not in P.

Should we now worry that soon someone proves P = NP? Well, the problem is very old. In the context of cryptography, already in 1955 John Nash conjectured that the time to break a cipher grows exponentially in the key length.[7] This is essentially the P vs. NP question.

[7] *"The most direct computation procedure would be for the enemy to try all 2^r possible keys one by one. [...] Now my general conjecture is as follows: For almost all sufficiently complex types of enciphering [...] the mean key computation length increases exponentially with the length of the key."* ~ John Nash [67]

We can view the key as a certificate; given the key, decryption is easy, but without the key deciphering takes exponential time. Note that this holds for every reasonable cryptography method, thus, they are all vulnerable if P = NP holds. After Nash's conjecture, many people stated the problem independently in various contexts. Thousands of people tried to prove either the equality or the inequality of P and NP, but they all failed.And there is a high incentive not to fail: The *Clay Mathematics Institute* pledged a one million USD prize for solving the P vs. NP question.[8] In case P = NP, we can be sure that intelligence agencies would pay even better for an efficient algorithm to break RSA and other cryptography systems. Intelligence agencies would probably keep the algorithm secret, but it is unlikely that they could keep it secret for long. So, *if* a solution was found already, we can expect that everyone in computer science (and most other people, too) would have heard about it already. And since so many people are (yet) unsuccessfully trying to answer the P vs. NP question for more than 60 years, it does not seem to be a simple question where we can expect an answer soon.

1.2 Voting

Let us now introduce the second major part this thesis will deal with: Voting!

Voting can be seen as a special case of cooperative game theory. We don't want to introduce cooperative game theory in detail here (see the book by Rothe [77] for more details). But from a bird's eye view, cooperative game theory is about *agents* from which some or all have *preferences* about coalition structures (that is, collections of subsets of the agents). The goal is then to select some admissible coalition structure as the *outcome*. This is why we sometimes refer to such problems as *preference aggregation*. What exactly an *admissible outcome* is, and what *types of agents* we have, characterizes what specific type of preference aggregation problem we have. Figure 1.2 provides an overview of some relations in the field. To illustrate the variety of problems, we explain some of them informally.

Roommates: The agents have preferences over each other. An admissible outcome is a coalition structure where each agent is in exactly one coalition, and this coalition has a given size. In other words, the goal is to assign each agent to a room such that the capacity of the rooms are not exceeded and no places are left. The problem was introduced by Gale and Shapley [40]. An efficient algorithm was developed by Irving [47].

Hospital-Residents: In this problem (which is called *College-Admissions* in the original paper by Gale and Shapely [40]) we have two types of agents, and both have preferences. On the one hand, we have the residents (students), who have preferences about the hospitals (or colleges) they do their internship at. On the other hand, we have the hospitals (colleges), which are concerned about the residents (students) who do their internship with them. An admissible outcome is a coalition structure where each resident is in exactly one coalition with a hospital, and each hospital is in exactly one coalition with (multiple) residents. Further, the coalition size is never greater than the

[8]See the full list of problems at claymath.org/millennium-problems.

capacity of the hospital. In other words: each resident is assigned to a hospital, and no hospital has more residents than it can handle.

Fair Division of Indivisible Items: Again, we have two types of agents—but this time active and passive ones, i.e., some have preferences, and some not. The *active agents* have preferences over the passive agents (which are usually called *items*) they want to have in their coalition. An admissible outcome is a coalition structure where each coalition contains at most one active agent and all coalitions are disjoint. In other words: the agents are assigned a subset of the available items, and no item is shared. One of the first studies is due to Brams et al. [19].

Participatory Budgeting: In participatory budgeting we have *projects* (the passive agents which have no preferences), and *voters* who have preferences about the projects. Admissible are those coalition structures, where all projects are in one of two coalitions: winners (i.e., projects which are funded) or losers. Further, projects are associated with a cost, and there is an upper bound on the total cost (called *budget*) the winner coalition may have. For a historical background we refer to Cabannes [26]. For a recent literature review in the light of computational social choice, see Aziz and Shah [4].

Multiwinner Voting: In this special case of participatory budgeting, all 'projects' (which are usually called *alternatives* or *candidates* here) have cost 1. The budget is the size of the winner coalition. In other words, we elect a *committee* of a given size. For a survey on multiwinner voting we refer to the work by Kilgour [53] and the book by Lackner and Skowron [55].

Single-Winner Voting: This is the special case of multiwinner voting where we have a budget of exactly one, i.e., we search for a *single* winner. Historically, this was the first subarea of voting theory that computational social choice paid major attention to, and there exists lots of literature. For an overview we refer to the chapter by Zwicker [90].

Apportionment: This is a special case of multiwinner voting, where the candidates are grouped in *parties*. Voters have only preferences over parties and don't care which of the party members join the committee. Also, multiple members from a party can join the committee. This is a common voting mode for parliamentary elections. Although this type of elections is used for centuries, computational social choice hasn't paid a lot of attention to it yet. A recent study by Brill et al. [25] shows the relation of multiwinner voting and apportionment.

We see that preference aggregation problems are quite diverse. Not only do they differ in the number of agent types we have, and what exactly they have preferences about. But they also differ in what an outcome looks like, and what constraints are put on it. The problems we are interested in—that is, voting related problems—stand out as the preference aggregation problems where we have one type of agents (called voters) who have preferences about another type of agents (called alternatives, candidates, or projects, depending on the application), and the outcome differentiates the second type of agents into two categories (winners/losers, or implemented/not implemented projects).

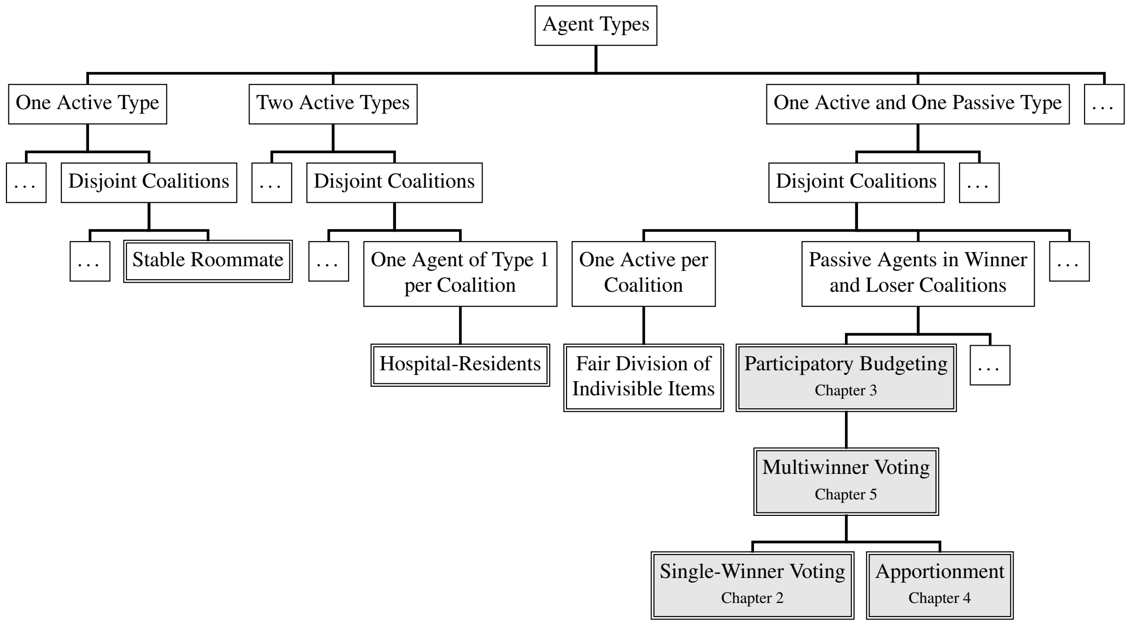

Agent Types
One Active Type
Two Active Types
One Active and One Passive Type
...
...
Disjoint Coalitions
...
Disjoint Coalitions
Disjoint Coalitions
...
...
Stable Roommate
...
One Agent of Type 1 per Coalition
One Active per Coalition
Passive Agents in Winner and Loser Coalitions
...
Hospital-Residents
Fair Division of Indivisible Items
Participatory Budgeting Chapter 3
...
Multiwinner Voting Chapter 5
Single-Winner Voting Chapter 2
Apportionment Chapter 4

Figure 1.2: Taxonomy of preference-aggregation problems from a *game theoretic* perspective based on a talk by Lang [57]. Topics covered in this thesis are colored gray.

As described, we can define each voting problem through cooperative game theory. However, the framework of cooperative game theory is much more powerful than we need for these very specific problems, and the taxonomy is only to illustrate the relations between all the preference aggregation problems. Thus, usually in literature (see e.g. [53], [55],[84], [90]) a simplified definition is used. We provide the common definitions below, and refine or modify them in the later chapters if needed.

Let us begin with the simplest case: *single-winner voting*. Let $\mathcal{E} = (\mathcal{A}, \mathcal{V})$ be a *(single-winner) election* where $\mathcal{A}$ is the set of *alternatives* which are up for election, and $\mathcal{V}$ the preferences of the voters. A *(resolute) single-winner voting rule* is a function f which maps an election to an alternative from $\mathcal{A}$. We call the candidate $f(\mathcal{E})$ the *winner* w.r.t. voting rule f. If a single-winner voting rule can result in more than one winner (that is, it gives us a set of candidates as winners), we call it *irresolute*. However, in practice such rules are made resolute by applying some tie-breaking mechanism, because in usual applications of single-winner voting we are interested in finding a *single* winner, instead of multiple ones.

This is different in *multiwinner voting*. A *(multiwinner) election* $\mathcal{E} = (\mathcal{A}, \mathcal{V}, K)$ consists of alternatives $\mathcal{A}$ and preferences $\mathcal{V}$ just as a single-winner election. But additionally we are given a *committee size* $K \in \mathbb{N}^+$. A *(resolute) multiwinner voting rule* is a function f which maps a (multiwinner) election to a size-K subset of $\mathcal{A}$ (called the *winning committee*). Just as in single-winner voting, we can have irresolute multiwinner voting rules which return a set of winning committees. Note that there exists also literature regarding multiwinner voting rules which choose committees of variable size (see e.g. Brandl and Peters [22]). That is, one does not specify the committee size in advance. However, this is not considered in this thesis.

For *apportionment elections* a variety of definitions exists in literature. It can be defined via approval-based multiwinner voting (see the work by Brill et al. [25]), or via vote distributions (see Bredereck et al. [23]). However, in this work we use a definition closer to multiwinner and single-winner elections. As an *apportionment instance* we are given $\mathcal{E} = (\mathcal{A}, \mathcal{V}, \kappa)$. The alternatives $\mathcal{A}$ are usually called *parties* in this setting, and the committee size κ is called the *seat count*. An *apportionment method* takes such an instance and produces a *seat allocation* $\alpha : \mathcal{A} \to \mathbb{N}$ for which $\sum_{p \in \mathcal{A}} \alpha(p) = \kappa$. That is, for each party it determines the number of seats that are allocated to that party, while ensuring that all seats are allocated and no seat is allocated to two parties at the same time.

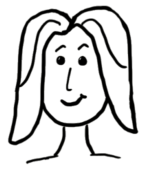
A *participatory budgeting* instance $\mathcal{E} = (\mathcal{A}, \mathcal{V}, K, c)$ generalizes a multiwinner voting instance by adding a *cost* function $c : \mathcal{A} \to \mathbb{N}$ which associates each *project* from $\mathcal{A}$ with a cost. Further, the committee size $K \in \mathbb{N}$ is now called *budget* and marks the maximum amount of money we can spend to implement projects. A *budgeting method* takes the instance and maps it to a *feasible bundle* $B \subseteq \mathcal{A}$, i.e., for B must hold $\sum_{p \in B} c(p) \leq K$.

1.3 Central Problems and Questions

In this section we want to introduce the main problems to which solutions we contribute in this thesis. Previously, as you may have noticed already, we were talking a lot about *preferences* without actually defining what a preference is. In fact, it is hard to tell what exactly a preference really is. Consider the following example of a single-winner election.

Example:

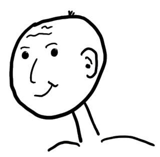
Alberto, Beatrix, Chris, Dave and Eve plan their joint vacation. They want to travel to the same location and spend time together. After a first brainstorming they discuss the shortlisted alternatives *Barcelona, Paris, Munich, Zurich* and *Lake Garda*.



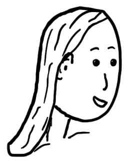
I really like swimming. So Barcelona and Lake Garda are good for me. However, I hate hiking and mountains, so I strongly disagree Zurich.

Beatrix

I like art museums. Paris would be great. Next best would be Munich. My third choice would be Barcelona, and the fourth Lake Garda. And I really dislike Zurich.



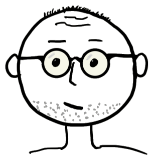
Alberto



I've been to Barcelona once, and I cannot really recommend it. Please, let us go somewhere else.

Eve

What? You don't like Barcelona? It is at least three times as good as Munich or Paris, and twice as good as Lake Garda!



Dave

On a meta-level, a preference is some internal evaluation by the respective voter for the available alternatives. However, in our formal model we need concrete data. What shines through the discussion in the previous example are three main types of *formal preferences* which we will refer to as *ballot types* (see also Zwicker [90], Procaccia and Rosenschein [70]).

Dichotomous/Trichotomous Ballots: Beatrix and Eve categorize the alternatives in two or three groups. Alternatives are either liked (often called *approved*), disliked, or neutral. For Beatrix and Eve dichotomous or trichotomous ballots would be best to express their preferences formally.

Ordinal Ranks: Alberto's preference is a little more complicated than Beatrix' and Eve's. He ranks the alternatives by how much he likes them. This can formally be expressed by an ordinal rank:

$$\text{Paris} \succ \text{Munich} \succ \text{Barcelona} \succ \text{Lake Garda} \succ \text{Zurich}$$

Sometimes in literature also *weak ranks* are allowed (see e.g. Mercik [62]). These are rankings which may include ties between alternatives. For instance, Alberto could find that Paris and Munich are equally good.

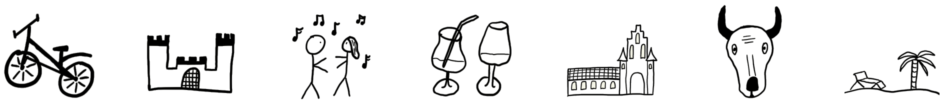
Cardinal Ballots: Dave provides even more information than Alberto. He wants to express that Barcelona is twice as good as Lake Garda and three times as good as Paris and Munich. This cannot be captured by ordinal ranks or trichotomous ballots. However, cardinal ballots would allow Dave to assign each alternative a numerical value with which he could easily express his preference. For instance, he could assign Barcelona the value 6, Lake Garda the value 3, and Munich and Paris each the value 2.

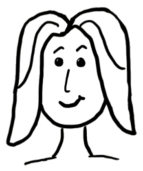
While cardinal ballots are the most expressive types of preferences, they are used rarely. This is mainly because the cognitive burden on casting such a ballot is very high for a human (though it might be possible for artificial agents [70]). Especially human voters (but to some degree also artificial intelligent agents) can very easily tell which alternatives they like, hate, or don't care about, and with a little more effort they can rank alternatives by how much they like them. But assigning each alternative an exact value is very hard. Dave might be able to do this for a very limited number of alternatives, but the more alternatives there are, the higher will the cognitive effort be.

As we see, for single-winner voting preference expression is already not trivial. This becomes even more difficult in multiwinner voting and participatory budgeting. We illustrate the complications with the following example.

Example:

Our five friends finally decide for Barcelona. For each of the three full days they spend there they want to do one joint activity. In the guidebook the following seven proposals are made: Cycling, visiting an old castle, dancing salsa, having drinks, visiting a cathedral, watching a bullfight, and lie on the beach.

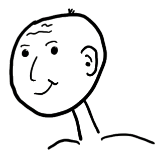




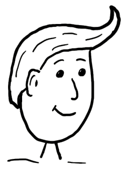
Definitely we should have drinks and go to the beach. But whatever we do the third day, please let us not support bullfights.

Beatrix

Sightseeing for one day sounds good. We could either go to the cathedral or the castle. But the other days we should have some fun dancing or at the beach.



Alberto



Sightseeing sounds great but I think visiting the cathedral makes only sense if we also visit the castle because they are historically linked. The third day I would like to go dancing salsa.

Chris

Once again, Beatrix' preference can be expressed by trichotomous ballots: she likes drinks and going to the beach, hates bullfights, and doesn't care about the others. For Alberto and Chris trichotomous ballots are not sufficient. Actually, not even the very expressive cardinal ballots would suffice. This is because in their preferences the alternatives are not independent. Alberto likes the cathedral and the castle, but doesn't want to visit both. For Chris this is exactly the opposite: visiting one without learning about the historical background of the other makes no sense. One way we can handle this is to allow everyone to express his or her preference through a ballot not on all alternatives but on all 3-elementary subsets of alternatives. However, this makes the ballot huge, and it is unrealistic that voters take the high cognitive effort to express their preference through such a ballot. This leads us to the first central problem which will be addressed in this thesis.

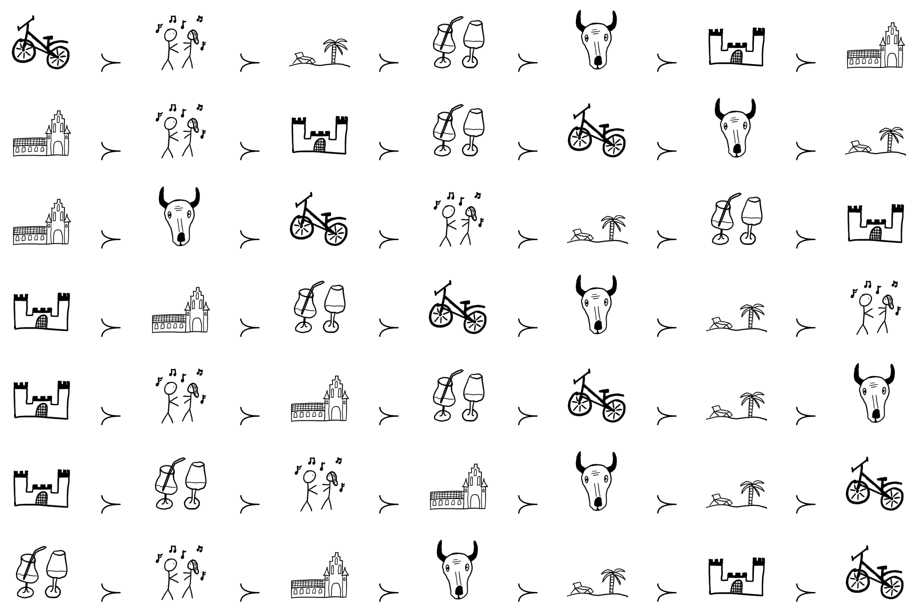
Problem 1. *We have seen that some voters want to express more complicated things than others. At the same time, some voters do not want to take a lot of cognitive effort to cast their ballot. What is the right ballot format to use for a specific application? To this end, we have to trade off between low cognitive effort, and best possible expressiveness of the ballots.*

Chapter 5 addresses Problem 1 by proposing and evaluating a new ballot format for multiwinner elections. This ballot format would allow Alberto and Chris to express their preference in Example 1.3 with low cognitive effort.

It is clear that the selection of an aggregation method depends on the ballot format because the method has to be designed to handle the ballot type. But what the best choice for an aggregation method is depends also on the intended application. Already for single-winner elections it is debatable which alternative should be the winner.

Example:

Consider the following preferences.



Several reasonable arguments how to elect the winner lead to very different results.

- The **castle** should win because it has more first positions than every other alternative. This is the argument behind the *Plurality voting rule*.
- The **drinks** should win because they are never at the last position; no one is really against it. This argument is captured by the *Veto voting rule*.
- **Dancing Salsa** is the best alternative because it wins in direct comparison against every other alternative. That is, 4 voters prefer dancing to visiting the castle but only 3 prefer the castle; 4 prefer dancing to drinks but only 3 prefer drinks; 4 prefer dancing to the cathedral but only 3 prefer the cathedral; and so on. This is the idea behind the *Condorcet voting rule*.
- The **cathedral** should win because it has the lowest average position in the preferences. That is, in average it has position $\frac{21}{7} = 3$ while the castle has average position $\frac{25}{7} \approx 3.57$, the drinks have position $\frac{24}{7} \approx 3.43$, and dancing $\frac{22}{7} \approx 3.14$. This corresponds to the *Borda voting rule* which gives the best candidate $n-1$ points, the second $n-2$ points, and so on. Then it selects the alternative with the most points as winner.

Also, with the other ballot formats it is debatable what the best alternative is. Given trichotomous ballots, should we rather select the alternative with the most likes, or the alternative with the least dislikes (similar to Plurality vs. Veto)? Given cardinal ballots, should we focus on maximizing the sum of cardinal values over all voters, or try to choose an alternative such that the worst-off voter still has the highest possible benefit from the winner? This is why a variety of single-winner rules have emerged over the years (see [58, 90] for an overview over single-winner voting rules). In multiwinner voting even more considerations come into play (compare [28]). For example, it might be important that as many voters as possible find at least one member of the committee acceptable. This applies for instance to selecting dishes to serve at a banquet. In political elections it is important that the committee is representative for the voters (i.e., for each $\frac{1}{K}$ of the voters with a similar preference one alternative is in the committee). When we shortlist candidates for a job interview, we want that candidates who are equally preferred by the voters (e.g. receive equal amount of approval) are treated equally. Further, in shortlisting selected candidates should not be unselected when we increase the committee size. Finally, in participatory budgeting it is often considered more *fair* if a proportional amount of money is spent on the satisfaction of each voter [69]. This leads us to the second central problem of this thesis.

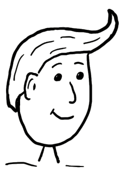
Problem 2. *Where can we use which voting rule best, and which voting rule is best for a given application? To this end, we have to study properties and behavior of voting rules, and identify which properties and behavior is important or suitable for a specific application.*

In Chapter 2 we study a novel application of voting rules: network centrality, and the selection of influential sets of nodes in a network. This relates to Problem 2 as we identify relevant, desirable properties in network science, and argue which voting rules solve the problem best (and better than existing methods from network science).

The following example illustrates another central problem we deal with in this thesis.

Example:

When the friends discussed on the activity proposals in Example 1.3, there was actually one proposal missing. Chris knows that there is also a leisure park in Barcelona. However, he hasn't made this proposal on purpose.



I want to go to the cathedral, the castle and dance Salsa. I know that Alberto, Dave and Eve want to go dancing Salsa, too. And both Alberto and Dave find the cathedral and the castle interesting. Thus, at the moment it is rather likely that we will visit the cathedral, the castle, and dance Salsa. However, there is also this leisure park, and I know if I propose it, everyone except for me will like it, so we will go there instead of visiting the cathedral.

Chris

What Chris does here is called *electoral control* in literature (see Faliszewski and Rothe [35] for an overview). On purpose, he removes (or holds back) one alternative from the election because he knows that this alternative makes the outcome worse for him. There are also control types where alternatives are added, or voters are removed from/added to the election. Further, a common problem is *bribing* of voters, that is, voters are paid to change their vote [33]. We refer to the collection of all those attempts to change the election outcome as *electoral fraud*.

In Example 1.3 the action to not propose the leisure park was rather obvious. However, sometimes there are much more difficult cases, where only by adding or removing a specific set of alternatives a successful control is possible. This may impose a high computational complexity. As mentioned in the previous subchapter about complexity theory, high computational complexity can be a positive, or negative result. Specifically electoral fraud is a good example where high computational complexity it is a positive result, as it makes fraud attempts more difficult. This yields the third problem of this thesis.

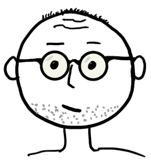
Problem 3. *As soon as we use preference aggregation, there will always be fraud. Agents can try to influence the election in multiple ways. High computational complexity can prevent malicious agents from fraud attempts, making the preference aggregation method more resistant against fraud. How resistant are preference aggregation methods? Can we make them more resistant?*

We will work on this problem by determining the computational complexity of several types of fraud in apportionment elections in Chapter 4. Also, the mentioned two-stage procedure for apportionment elections we propose in Chapter 4 will prove to increase computational fraud resistance.

Previously, we were assuming that all required information is available. But is that realistic? Consider the following example.

Example:

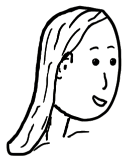
After a long discussion Alberto, Beatrix, Chris, Dave, and Eve decide to dance Salsa, have drinks, and go to the beach—one joint activity for each day of their three days stay. Just when they arrive in Barcelona it starts to rain.



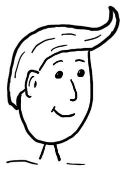
Oh, damn. My phone says it will rain the next days with high probability, too. It looks very bad for our beach trip...

Dave

Well, at least we can have the drinks indoors. And even though dancing on the streets is nicer, we'll find a bar where we can also dance Salsa indoors.



Eve



We can visit the castle or the cathedral the third day. That's also indoors. But we have to decide that right now because we have to preorder the tickets two days in advance.

Chris

Should they now buy tickets for the cathedral, or hope for better weather to go to the beach? If they don't buy tickets and it continues to rain, they must spend the third day at the hotel. But if they buy tickets and the weather becomes better, they either lost money for the tickets, or they go to the cathedral even though the beach would be a better option.

This example illustrates that in reality we often have uncertainty about available data. This leads to the fourth and last central problem we address in this thesis.

Problem 4. *In our theoretical models all information is available. However, in reality—and we want our methods to be used in reality—information is usually incomplete or uncertain. How can we deal with such incomplete information in preference aggregation?*

This thesis contributes significantly to a deeper understanding, and possible solution for this problem. In Chapter 3 we study uncertainty in the project cost and durations in participatory budgeting. We show how to deal with this uncertainty algorithmically, and we show what limitations every algorithm inherently has in this setting.

1.4 Structure

Most results presented in the following chapters were published already, or are currently under review at a conference or journal. All those (published, or unpublished) papers are joint work with my co-authors. To account for this, I decided to structure the subsequent chapters as follows. At the beginning of each chapter I provide a short summary of the topic studied in the chapter. This should help the reader to quickly identify what the chapter is about. Next, there will be a detailed introduction to the topic, followed by definitions and results. The chapters are independent of each other and should be comprehensible after reading this introduction. Whenever more definitions are required, they are given in the respective chapter. Note that some results in this thesis are not yet included in any papers. Further, some results from published papers are not included in this thesis, because I have not contributed significantly to these specific results. However, not all results presented here are completely my work. To make clear what contributions to both, the paper and the chapter, are my work, each chapter concludes with a description (1) where some results have been published, and (2) what exactly in the paper or chapter are my contributions.

1.5 Notation

Throughout this thesis, we denote the set $A \setminus \{B\}$ by A_{-B} for a set A and an element B. Further, we write $[i]$ as a shorthand for $\{1,\dots,i\} \subset \mathbb{N}$ for an integer $i \geq 1$, and $[i,j]$ for $\{i,\dots,j\} \subset \mathbb{N}$ for integers i,j with $0 \leq i \leq j$.

CHAPTER 2

Network Centrality Through Voting Rules

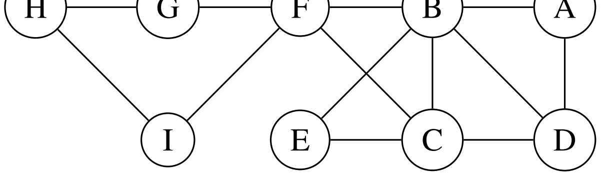
2.1 Summary

Network centrality is the science of identifying the most central or important nodes (or sets of nodes) in a network. We study how single- and multiwinner voting rules can be used in network science to identify central nodes (or sets of nodes) in a network. To this end, we first define preferences for the nodes in the network based on their network relations (here: shortest path distances). We then reformulate common single- and multiwinner voting rules to deal with these preferences and either rank the nodes by importance (centrality ranking), or construct a central set of nodes in the network (so-called *node selectors*). Finally, we reformulate reasonable properties and axioms from voting theory for network science, and show which of these properties are satisfied by traditional centrality measures and voting-based centrality measures.

2.2 Introduction

Centrality indices are used to measure the individual centrality of nodes as a quantity, and to eventually rank the nodes by their indices. *Node selectors* return a fixed-size group of nodes which is the most central according to the node selector. There exist a variety of centrality indices and node selectors in literature [54, 79, 89, 31, 39]. Often they are designed for very specific applications, and there is no universal centrality measure or node selector which works for every application. Let us look at the following example.

Example:



> Consider the network above. The question that motivates centrality indices is, *how important is e.g. node B in the network compared to the other nodes?* When the network represents e.g. a friendship network, we might find B very important as B has many friends. However, if the network represents a street map, we might find F much more important than B because when F is removed, the network is no longer connected.
>
> The question behind node selectors is *what set of nodes (of size e.g. two) is most central?* One answer could be $\{B, C\}$ because both have many neighbors. However, as noted by Everett and Borgatti [31] the centrality of a group of nodes is not necessarily the sum, or average of the individual centrality, and indeed we can argue that B and C have basically the same neighborhood, thus are not much more central than individually. So maybe it is better so select $\{B, G\}$ because together their neighborhood covers almost the whole network. The latter could be interesting when the goal is to maximize the influence in the network (see e.g. Kempe et al. [52])

Voting is a very powerful tool from social choice theory. Research has paid a lot of attention to it over the past years (see [3] for more background). As mentioned in the introduction of this thesis, in voting we have a set of candidates (or alternatives) and a list of preferences over these candidates, submitted by the voters. Thereby, preferences can have manifold forms such as approval ballots or ordinal ranks. We can then use *single-winner voting rules* to determine which candidate wins the election, or *multiwinner voting rules* to determine a fixed-size set of candidates which are then called the *winning committee*. Internally, most single-winner voting rules are actually *social welfare functions* which provide weak rankings of the candidates or alternatives which represent the support a candidate has among the voters. So one can either select the best candidate, or see how good one candidate is compared to another. Note that the goal behind centrality indices is basically the same: make candidates (nodes) comparable by how much support they have among the voters (respectively, how important each node is for the other nodes in the network). A similar relation holds also for multiwinner voting and node selectors.

The correspondences between single-winner/multiwinner voting and centrality indices/node selectors is very interesting. The fields network science and voting theory have only been connected in a few studies yet. Thereby, most literature linking the fields focuses rather on studying how social networks could impact, or improve voting (see the survey by Grandi [45]). For example, *delegative voting*, a.k.a. *liquid democracy* or *interactive democracy* [24], describes voting where voters can either vote on an issue themselves, or delegate their vote to a friend (i.e. a neighbor in the network) who is called the proxy voter (see e.g. the work by Boldi et al. [15]). Boldi et al. [15] mention that one can interpret the number of delegated votes a proxy voter has as centrality index. However, as far as we know the literature regarding the use, or adaption of voting rules as centrality measures is rather sparse. Wilkinson [88] analyzes an application of the voting rule due to Schulze [80] to find the important actors in a terrorist network that changes over time. Further, Zhang et al. [89] proposed the VoteRank node selector which (as we will see later) is very similar to the *sequential proportional approval voting* multiwinner rule. In this chapter we will significantly extend the knowledge of the relations between voting and network centrality.

2.3 Centrality in Networks

We define a *network* as an undirected connected graph $G = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes, and $\mathcal{E}$ the set of edges. Note that the definitions in this chapter can easily be extended to weighted graphs. By $N(i) = \{j \in \mathcal{N} : \{i, j\} \in \mathcal{E}\}$ we denote the neighborhood of i, and by $N[i] = N(i) \cup \{i\}$ its closed neighborhood. By $\delta(a,b)$ we denote the shortest path distance between a and b. Note that the distance is symmetric, and since we assume the graph to be connected, the distance is finite.

2.3.1 Centrality Indices

A *node index* c is a mapping $c : \mathcal{N} \to \mathbb{R}$. Often it is not the quantities themselves that are of interest, but only the induced ranking of nodes. For a node index c, we write $i \succcurlyeq_c j$ if $c(i) \geq c(j)$, $i \succ_c j$ if $c(i) > c(j)$, and $i \sim_c j$ if $c(i) = c(j)$ (i.e., a higher index is better). But which node indices deserve to be called *centrality indices*? This is of course a substantive question with—unfortunately—no definite answer. Obviously, a centrality index should respect the common intuition that "central" nodes are connected more strongly, and more directly to the other nodes. But it is hard to capture this intuition in a formal definition. However, as a minimum requirement we can require a node index to be *invariant under graph automotphisms* (i.e., the index depends solely on the network relations of a node, and not on the name), and to satisfy the *vicinal preorder* which is defined as follows.

Definition 1 (Vicinal Preorder)**.** *The* vicinal preorder *is a reflexive and transitive, but not necessarily complete or antisymmetric order which for $i, j \in \mathcal{N}$ contains $i \succcurlyeq j$ if and only if $\delta(s,i) \leq \delta(s,j)$ for all $s \in \mathcal{N} \setminus \{i, j\}$.*

The *vicinal preorder* ensures that i is considered at least as central as j if for every other node the shortest path to i is no longer than its shortest path to j. Note that in the unweighted networks we consider, the above falls down to the usual definition in terms of neighborhood inclusion where $i \succcurlyeq j$ iff $N[i] \supseteq N(j)$ as defined by Foldes and Hammer [38]. However, to make our results more easily generalizable for weighted networks, and because of the way we define preferences based on distances later, it is more convenient to use distances rather than the neighborhood inclusion. We can now define centrality formally.

Definition 2 (Centrality)**.** *Any ranking of nodes that is invariant under automorphisms and respects the vicinal preorder is a* centrality ranking. *A node index which induces a centrality ranking is called* centrality index.

To give an impression of what centrality indices are, we next present some of the most common centrality indices (see Koschützki et al. [54] for an overview). We will later compare them to the new voting-based centrality indices that we propose in Section 2.5.

Degree centrality. The degree of a node $i \in \mathcal{N}$ is the cardinality of its neighborhood. Thus, a node is central according to *degree centrality* if it has many neighbors.

$$c_{\text{deg}}(i) = |N(i)|$$

Closeness centrality. To take indirect links to other nodes into account, *closeness centrality* considers shortest-path distances $\delta(s,t)$:

$$c_{\text{clo}}(i) = \frac{1}{\sum_{t \in \mathcal{N}_{-i}} \delta(i,t)}.$$

Betweenness centrality. In Example 2.2 we argued that F is very central because it is a critical connection point. This intuition is captured by betweenness centrality. Let $\sigma(s,t)$ denote the number of shortest paths from s to t and $\sigma(s,t|i)$ the number of those shortest paths also passing through a brokering node i. Then *betweenness centrality* is given by

$$c_{\text{bet}}(i) = \sum_{s,t \in \mathcal{N}} \frac{\sigma(s,t|i)}{\sigma(s,t)}.$$

Schoch and Brandes [79] have shown that the rankings obtained from standard centrality indices such as degree, closeness, and betweenness centrality respect the vicinal preorder.

2.3.2 Group Centrality And Node Selectors

While centrality indices are used to differentiate nodes by their individual centrality, the purpose of *group centrality indices* is to rate how central a set of nodes is. As mentioned in the introduction, while it is easy to create a group centrality index out of a centrality index by summing up the individual indices of the nodes in a set, such a group centrality index is not suitable in many settings because it ignores redundancy of nodes. Everett and Borgatti [30, 31] propose group centrality indices which extend the intuition of classical centrality indices.

Group degree centrality [31]. The group degree centrality of a set $X \subseteq \mathcal{N}$ is the number of nodes in $\mathcal{N} \setminus X$ which are connected to at least one node from X.

$$c_{\text{g-deg}}(X) = |\{t \in \mathcal{N} \setminus X \mid \exists s \in X : \{s,t\} \in \mathcal{E}\}|$$

Group (minimum) [1] closeness centrality [31]. The distance $\delta(a,X)$ of a node a to a set of nodes X is the minimum shortest path distance of a to any node in X. The group closeness

[1]Everett and Borgatti [31] also mention scenarios where it could be more realistic to define the distance of a node a to a set X as average, or maximum distance of a to the nodes in X.

centrality of a set X is the inverse[2] of the sum of all distances from nodes in $\mathcal{N} \setminus X$ to X.

$$c_{\text{g-clo}}(X) = \frac{1}{\sum_{t \in \mathcal{N} \setminus X} \delta(t, X)}.$$

Group betweenness centrality [31]. Let $\sigma(s,t)$ denote the number of shortest paths from s to t and $\sigma(s,t|X)$ the number of those shortest paths also passing through at least one node $i \in X$. Then *group betweenness centrality* is given by

$$c_{\text{g-bet}}(X) = \sum_{s,t \in \mathcal{N} \setminus X} \frac{\sigma(s,t|X)}{\sigma(s,t)}.$$

Note that the group centrality indices above yield the same centrality index for sets of size one as the respective centrality index for single nodes. Thus, they can be considered proper generalizations. However, note that the group degree index and the group betweenness index are not monotonic, i.e., supersets of X may have lower centrality than X. For example, the group degree index shrinks when a node is added to X which was a direct neighbor of X, but has no further neighbors. Similarly, when nodes s and t have all their shortest paths running through X, and we add s, or t to X, the centrality of X can shrink. Thus, they are not suitable to compare different-size sets of nodes. While there are fixes to this, it doesn't really matter for our study as we consider only fixed-size sets anyway.

In the applications we have in mind, we want to find a set of nodes of a given size K which is very central, i.e. has a high group centrality index. Such applications include viral marketing strategies (see the related work section of Leskovec et al. [60]), or facility location problems [49]. It is often not practical to rate all subsets of nodes, and then pick the best size-K set since there are just so many subsets.[3] *Node selectors* are used to *construct* a set of nodes of a given size K, $1 \leq K \leq |\mathcal{N}|$, that collectively are as central as possible (what 'central' means depends on the intended application). That is, a node selector takes the network and the number $K \in \mathbb{N}$ as input, and returns a subset of $\mathcal{N}$ of size exactly K. They are thus the constructive approach to group centrality whereas group centrality indices are metrics.

Note that for many node selectors computing the node set is very difficult. For example, a node selector aiming at computing a set which is optimal according to group degree, or group closeness is computationally intractable because the problem is closely related to the problem of deciding whether there is a dominating set of given size—a problem which is well known to be NP-complete. We will later present some own node selectors based on multiwinner voting rules. Although they do not aim at maximizing group degree, or group closeness directly (they have different interesting properties as we will see), we will also compare in experiments how good they can approximate the group degree and group closeness optimum.

[2]Note that we use the inverse to maintain that higher values mean better centrality.

[3]For n nodes we have $\binom{n}{K}$ subsets. This number becomes extremely large for larger n and K.

2.4 Voting in Networks

The point of our study is to propose new centrality indices and node selectors based on voting rules. However, voting rules take preferences and candidates as input while centrality indices and node selectors receive a network as input. Thus, we first have to discuss how networks can be transformed into preferences and candidates. We propose an interpretation of the network as nodes having preferences over the other nodes. That is, nodes are both, voters and candidates, at the same time. The preferences can be derived from the network structure. More precisely, a node's preference over the other nodes depends on its distances to them. In correspondence to the absence of self-loops in the network, we assume that nodes do not have preferences for themselves.

As discussed in the introduction to this thesis, preferences can have manifold forms. In this chapter we focus on two of the most common types: approval ballots which allow each node (in the role of a voter) to approve as many other nodes as desired, and ordinal ranks where each node submits a linear order over the other nodes. We relax the requirements for ordinal ranks for the purpose of this paper and allow ties, because they appear very often in networks (especially in unweighted networks).

Definition 3 (Approval Ballots in Networks)**.** *Each node $v \in \mathcal{N}$ approves all nodes up to a given threshold distance γ in the network:* $\text{app}(v) = \{i \in \mathcal{N}_{-v} : \delta(i,v) \leq \gamma\}$.

An interpretation is that approved nodes are in an acceptable distance (i.e., reachable via a sufficiently short path) from the node approving them. This could be of interest in e.g. facility location problems. Note that due to the undirected networks, $\text{app}(v)$ is also the set of nodes (as voters) which approve of v.

Definition 4 ((Weak) Ordinal Ranks in Networks)**.** *A node v prefers a over b, denoted as $a \succ_v b$, if $\delta(v,a) < \delta(v,b)$, and is indifferent between a and b, denoted as $a \sim_v b$, if $\delta(v,a) = \delta(v,b)$. The* position *of node a in the preference of v is* $\text{pos}_v(a) = 1 + |\{x \in \mathcal{N}_{-v} : x \succ_v a\}|$.

So the voter node simply ranks the candidate nodes by their shortest-path distance. Note that in case of ties in v's preference, some nodes will share a position. Thus, the position of a node in the preference is usually different from the actual shortest path distance.

2.5 Single-Winner Voting-Based Centrality Indices

In this section, we start developing new centrality indices based on voting rules and the preferences we derived as just described. But before that, we briefly want to mention how voting theory and network science relate from an axiomatic perspective.

2.5.1 Axiomatic Relations

There are several literature reviews on axioms in single-winner voting we can recommend to the interested reader for a more detailled description of the axioms [3, 90, 13].

To start with, we want to mention some axioms which have immediate counterparts in network science. First, there is the invariance under automorphisms which relates to *neutrality* (where, informally, all candidates are treated equally) and *anonymity* (where, informally, all voters are treated equally). Voting rules usually satisfy both axioms for good reasons. Who would like to use a voting rule where some candidates, or voters have a disadvantage just because of their name, or position in the profile? All voting rules we discuss later do guarantee anonymity and neutrality, so they are also invariant under automorphisms.[4] Second, we want to note the relation between the vicinal preorder and the *Pareto criterion*. The criterion states that no candidate a should win an election if there exists another candidate b who is preferred to or considered to be indifferent with a by *every* voter and strictly preferred to a by at least one voter. Recall that the vicinal preorder states that $b \succcurlyeq a$ if $\delta(s,b) \leq \delta(s,a)$ for all $s \in \mathcal{N} \setminus \{a,b\}$ which is a little less strict but in the same spirit as the Pareto criterion.

Now we consider an axiom from voting theory which (as far as we know) has no counterpart yet in network science—but it is perfectly reasonable to consider it in network science. The axiom we are talking about is called *Condocet criterion*. But to define it we first need the following definition. In voting, it is common to say that candidate *a pairwise dominates* candidate b if a majority of voters prefers a over b. We reformulate this notion in terms of network relations as follows.

Definition 5 (Pairwise Domination in Networks)**.** *Node a* dominates *node b, denoted as* $a \gg b$*, if and only if* $|\{v \in \mathcal{N} \setminus \{a,b\} : \delta(v,a) < \delta(v,b)\}| > |\{v \in \mathcal{N} \setminus \{a,b\} : \delta(v,a) > \delta(v,b)\}|$.

That is, a dominates b when a (relative) majority of nodes is strictly closer (thus, better connected) to a than to b. Nodes which have the same distance to both count for neither of the two (or equivalently for both). Now the Condorcet criterion states that a candidate (or node) should win the election if it dominates every other candidate (or node). Voting rules satisfying the Condorcet criterion are called *Condorcet-consistent*. Analogously, we can define what a Condorcet-consistent centrality ranking is.

Definition 6 (Condorcet-Consistent Centrality Ranking)**.** *A centrality ranking is* Condorcet-consistent *if it satisfies the following. Whenever there exists a node x which pairwise dominates all other nodes, x is ranked strictly more central than any other node.*

One can easily argue that the Condorcet-criterion is very desirable in a lot of applications for network centrality. However, we have to mention that the so-called *Condorcet-winner*—a

[4]However, it is well-known by Arrow's Impossibility Theorem [2] that a dictatorship rule (which is obviously not anonymous) is the only possibility to achieve a combination of other desirable properties.

node which dominates every other node—doesn't always exist. This is known as the famous *Condorcet paradox* in social choice theory. Figure 2.1 shows an example of the paradox in networks. We study Condorcet-consistent voting rules later in Section 2.5.4.

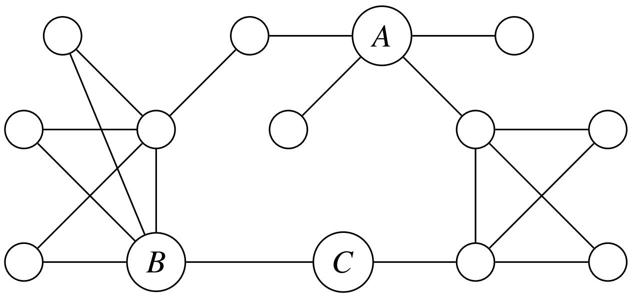


Figure 2.1: The Condorcet paradox: There are 6 nodes which strictly prefer A to B but only 5 strictly prefer B. Further, there are 5 nodes which strictly prefer B over C but only 4 strictly prefer C. Finally, 5 nodes prefer C to A but only 4 strictly prefer A. Thus, we have $A \gg B \gg C \gg A$.

2.5.2 Satisfaction Approval Voting

The first class of rules we can derive new centrality measures from are approval rules (see Laslier and Sanver [58] for an introduction). The advantage is here that these rules require no modification; they work out of the box with the approval ballots we defined in Section 2.4.

The simplest approval rule is simply called *approval* and just sums up the approvals for each node. Note that for a threshold distance of $\gamma = 1$, it is equivalent to degree centrality. It is easy to see that approval also induces a centrality ranking according to Definition 2 since if every node in the network has a shortest path to a at least as short as to b, every node which approves b must also approve a. But we promised to derive *really new* centrality measures, and not only new variants of degree centrality. A particularly interesting and more sophisticated approval rule is *satisfaction approval voting (SAV)*, proposed by Brams and Kilgour [20]. In SAV, voter v contributes a score of $\frac{1}{|\text{app}(v)|}$ to each candidate in $\text{app}(v)$, i.e., v's total weight of 1 is uniformly distributed among all candidates approved by v. These scores induce a centrality index.

Theorem 1. *SAV induces a centrality index.*

Proof. According to Definition 2, we need to show that the ranking induced by SAV respects the vicinal preorder and is invariant under automorphisms.

It is easy to see that SAV is invariant under automorphisms. We now show that it respects the vicinal preorder. Suppose $i \succcurlyeq j$ in the vicinal preorder for some nodes i, j. Since $\delta(s,i) \leq \delta(s,j)$ for all $s \in \mathcal{N} \setminus \{i,j\}$, we know that each node in $\mathcal{N} \setminus \{i,j\}$ that approves of j also approves of i and contributes exactly the same value to the scores of both. Further, each node that j approves is approved by i. It follows that $|\text{app}(i)| \geq |\text{app}(j)|$ and thus j contributes at least as much value to the score of i than vice versa. Hence, $c_{\text{SAV}}(i) \geq c_{\text{SAV}}(j)$. □

To receive a high SAV score, it is important to be close to many nodes which have not many other connections. Figure 2.2 shows an example where this behavior may be interesting. According to degree centrality, nodes A and B are identically central. However, if A is removed, the network is still connected. If B is removed, it is not. Thus, it is a reasonable intuition that B is more central than A—and SAV results in exactly that.

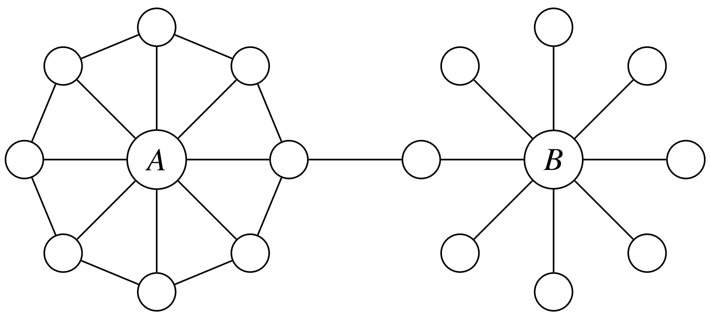


Figure 2.2: For $\gamma = 1$, node A receives a total SAV score of $\frac{7}{3} + \frac{1}{4}$, whereas B receives an SAV score of $7 + \frac{1}{2}$. Thus, B is more central according to SAV centrality.

2.5.3 Borda

Positional scoring rules are a class of voting rules which take profiles of strict ordinal ranks as input, and assign the candidates a score which depends on the positions of the candidates in the ranks. For example, the prominent Borda count adds 0 to the score of a candidate for every voter where the candidate is at last position, 1 for the second but last position, 2 for the third but last, and so on. This leads to the common formulation of the Borda count via the scoring vector $(n-1, n-2, \ldots, 0)$. So intuitively, for each voter the candidate receives 1 point for each candidate it beats.

This is more difficult when preferences have ties as in our setting. Fortunately, Gärdenfors [42] proposed a formulation of the Borda count which is equivalent (in the ranking, not in the exact scores) to the scoring vector $(n-1, n-2, \ldots, 0)$ with strict rankings, but also preserves the intuition in rankings with ties. One counts how many candidates i is preferred to and subtracts the number of candidates the voter prefers to i:

$$c_{\text{bor}}(i) = \sum_{v \in \mathcal{N}_{-i}} |\{a \in \mathcal{N}_{-v} \mid i \succ_v a\}| - |\{a \in \mathcal{N}_{-v} \mid a \succ_v i\}|.$$

Theorem 2. *Borda count induces a centrality ranking.*

Proof. It is easy to see that Borda is invariant under automorphisms. We now show that the ranking respects the vicinal preorder.

Suppose $i \succcurlyeq j$ in the vicinal preorder. Since $\delta(s,i) \le \delta(s,j)$ for all $s \in \mathcal{N} \setminus \{i,j\}$, we know that for each $s \in \mathcal{N} \setminus \{i,j\}$, it holds that $|\{a \in \mathcal{N}_{-s} \mid i \succ_s a\}| \ge |\{a \in \mathcal{N}_{-s} \mid j \succ_s a\}|$ and $|\{a \in \mathcal{N}_{-s} \mid a \succ_s i\}| \le |\{a \in \mathcal{N}_{-s} \mid a \succ_s j\}|$. Thus, it holds that $|\{a \in \mathcal{N}_{-s} \mid i \succ_s a\}| - |\{a \in \mathcal{N}_{-s} \mid a \succ_s i\}| \ge |\{a \in \mathcal{N}_{-s} \mid i \succ_s a\}| - |\{a \in \mathcal{N}_{-s} \mid a \succ_s i\}|$, i.e., from each $s \in \mathcal{N} \setminus \{i,j\}$ node i receives at least as many points as node j.

Now we only have to ensure that i receives at least as many points from j than vice versa. For each node $s' \in \mathcal{N} \setminus \{i,j\}$ with $\delta(j,s') < \delta(j,i)$, it holds that $\delta(i,s') < \delta(j,i) = \delta(i,j)$. So each node that j prefers to i will also be preferred by i over j. It follows that $|\{a \in \mathcal{N}_{-j} \mid i \succ_j a\}| \ge |\{a \in \mathcal{N}_{-i} \mid j \succ_i a\}|$. Conversely, for each node $s'' \in \mathcal{N} \setminus \{i,j\}$ with $\delta(i,s'') > \delta(i,j)$, it holds that $\delta(j,s'') > \delta(j,i)$. Thus, $|\{a \in \mathcal{N}_{-j} \mid a \succ_j i\}| \le |\{a \in \mathcal{N}_{-i} \mid a \succ_i j\}|$, so j contributes at least as much to the score of i than vice versa. $\square$

At the first look, Borda seems to be very similar to closeness centrality. However, Figure 2.3 shows the substantial differences between Borda and closeness, but also betweenness.

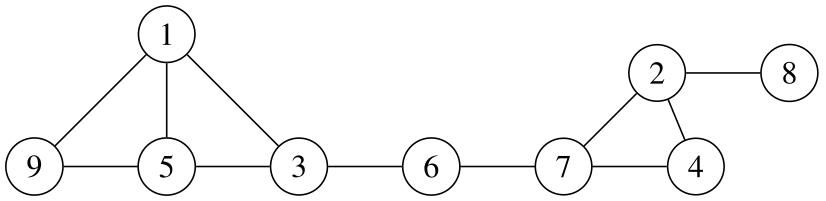


Figure 2.3: Example where closeness, betweenness, and Borda all lead to different rankings. Closeness centrality induces the centrality ranking $6 \succ 3 \sim 7 \succ 1 \sim 2 \sim 5 \succ 4 \succ 9 \succ 8$, betweenness yields $6 \succ 3 \sim 7 \succ 2 \succ 1 \sim 5 \succ 4 \sim 8 \sim 9$, whereas the Borda ranking is $7 \succ 6 \succ 3 \succ 2 \succ 5 \sim 1 \succ 4 \succ 9 \succ 8$.

2.5.4 Copeland

Previously, we have introduced the notion of a Condorcet winner and Condorcet-consistency. We now want to introduce one common Condorcet-consistent voting rule which can be used as a centrality index. While there are several prominent examples for Condorcet-consistent voting rules, such as the rules due to Banks [8], or Slater [82], we decided to study Copeland's rule [27] as it is efficiently computable, whereas most others were shown to be NP-hard. The *Copeland score of a candidate a* is defined by

$$c_{\text{cop}}(a) = |\{x \in \mathcal{N} : a \gg x\}| - |\{x \in \mathcal{N} : x \gg a\}|.$$

Theorem 3. *Copeland induces a centrality ranking.*

Proof. It is easy to see that the Copeland scores are invariant under automorphisms. We show that the ranking induced by Copeland also respects the vicinal preorder

Consider any pair $i \succcurlyeq j$ in the vicinal preorder, i.e., $\delta(s,i) \leq \delta(s,j)$ for every $s \in \mathcal{N} \setminus \{i,j\}$. We show two properties: (A) i cannot be dominated by j, and (B) every node dominated by j will also be dominated by i. (A) and (B) imply $c_{\text{cop}}(i) \geq c_{\text{cop}}(j)$.

By the definition of dominance, we can ignore the votes of i and j in the pairwise comparison of i and j. For every other node s, we have $\text{pos}_s(i) \leq \text{pos}_s(j)$, so j cannot dominate i, which proves statement (A).

Consider any node $t \in \mathcal{N} \setminus \{i,j\}$ that is dominated by j, i.e., $|\{v \in \mathcal{N} \setminus \{t,j\} : \text{pos}_v(j) < \text{pos}_v(t)\}| > |\{v \in \mathcal{N} \setminus \{t,j\} : \text{pos}_v(j) > \text{pos}_v(t)\}|$. We want to show that i also dominates t: $|\{v \in \mathcal{N} \setminus \{t,i\} : \text{pos}_v(i) < \text{pos}_v(t)\}| > |\{v \in \mathcal{N} \setminus \{t,i\} : \text{pos}_v(i) > \text{pos}_v(t)\}|$. Since $\text{pos}_s(i) \leq \text{pos}_s(j)$ for every $s \in \mathcal{N} \setminus \{i,j\}$, every voter counting for j also counts for i and no voter with $\text{pos}_s(j) = \text{pos}_s(t)$ (i.e., no voter counting neither for j nor t) can count against i. In particular, no additional node $v \in \mathcal{N} \setminus \{i,j,t\}$ counts against i. We only have to check that if i counts for j (that is, $\text{pos}_i(j) < \text{pos}_i(t)$), then also j counts for i ($\text{pos}_j(i) < \text{pos}_j(t)$). By the vicinal preorder, each node that j prefers to i will also be preferred by i over j. Thus $\text{pos}_j(i) \leq \text{pos}_i(j)$, which proves (B). □

Figure 2.4 shows that closeness and betweenness centrality are not Condorcet-consistent. This makes the Copeland centrality index—as far as we know the first Condorcet-consistent centrality index ever proposed—an interesting subject to future studies.

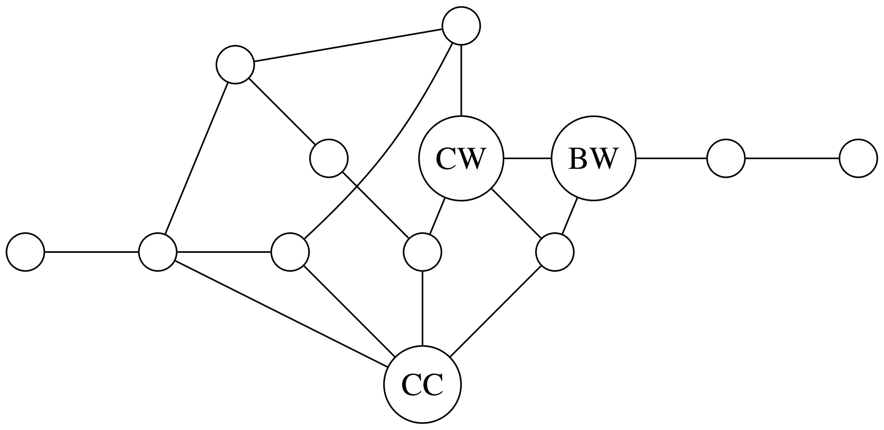


Figure 2.4: Closeness centrality and betweenness centrality do not satisfy the Condorcet criterion for centrality rankings: The closeness centrality winner (CC) and the betweenness centrality winner (BW) is not the Condorcet winner (CW).

2.6 Multiwinner Voting-Based Node Selectors

We now consider the use of multiwinner voting rules as node selectors. Important for many multiwinner voting rules is to be *representative*. For example, they are designed to avoid that when 3 candidates should be elected, and 51% approve candidates a, b, c, the committee $\{a, b, c\}$ wins regardless what the other 49% of voters want. There are various definitions of what exactly "representative" means. One of the most common definitions is for approval voting rules: *justified representation* (see Aziz et al. [7]). We formulate it for networks.

Definition 7 (Justified Representation)**.** *Let $G = (\mathcal{N}, \mathcal{E})$ be a network and γ a given number. A set $X \subseteq \mathcal{N}$ with $|X| = K$ satisfies justified representation w.r.t. γ if there is no set $Z \subseteq \mathcal{N} \setminus X$ of size $|Z| \geq \frac{n-K}{K}$, for which $\bigcap_{i \in Z}\{j \in \mathcal{N}_{-i} \mid \delta(i, j) \leq \gamma\} \neq \emptyset$, and for all $i \in Z$ holds $\{j \in \mathcal{N}_{-i} \mid \delta(i, j) \leq \gamma\} \cap X = \emptyset$. A node selector satisfies justified representation w.r.t. γ, if every set it selects satisfies justified representation.*

Thereby γ is the maximum reachable distance for nodes, i.e., the distance in which the voter nodes approve other nodes as mentioned in Section 2.4. Informally, for X to satisfy justified representation there must exist no group Z which deserves an own "representative" in X due to its size, and in which all nodes in Z can agree on one representative, but no node in Z has a representative in X. Justified representation might be an interesting property to study for facility location problems.

One approval rule that guarantees the outcome to satisfy justified representation (see [7] for the proof) is *proportional approval voting (PAV; first proposed by Thorvald Thiele)*. In PAV, it is assumed that each voter v has a satisfaction score $s_v(X) = 1 + 1/2 + 1/3 + \cdots + 1/j$ for a committee X, where j is the number of candidates from X approved by v. This follows the intuition that a voter's satisfaction grows with the number of approved candidates in the committee, but the increment in satisfaction becomes the smaller, the more candidates this voter already approves in the committee. PAV can thus be used as both, a group centrality index, or a node selector (by searching for the set of nodes with the highest satisfaction). It is known that determining the committee with the highest PAV score is NP-complete [6].

Sequential proportional approval voting (SPAV) is efficiently computable, and sometimes referred to as an approximation of PAV (where PAV is used as node selector/multiwinner rule). We present the generalization w-SPAV [7] with a vector $w = (w_1, \ldots, w_K)$. It works in rounds and has remarkable similarities to VoteRank [89]. In the first round, every voter has a voting ability of w_1, i.e., each node (as a voter) adds w_1 to the score of each neighbor. A node (as a candidate) with the highest score is added to the winning committee W (and is removed as a candidate from the further election). In the subsequent rounds, every voter has a voting ability of w_{j+1}, where j is the number of candidates the voter approves in W. The usual definition of SPAV is obtained by the vector $w = (1, \frac{1}{2}, \ldots, \frac{1}{K+1})$. VoteRank [89] can be defined by the vector $w = (1, \max\{0, 1-c\}, \max\{0, 1-2c\}, \ldots)$ for some constant c which is usually set to the inverse average degree of the network (for details on other constants see Zhang et al. [89]). The only difference to w-SPAV is that elected nodes can still vote on other nodes in w-SPAV but not in VoteRank. A particularly interesting case is $(1,0,\dots,0)$-SPAV (sometimes referred to as *GreedyAV*), because it also guarantees outcomes that satisfy justified representation [7] which is not guaranteed for SPAV and VoteRank.

Theorem 4. *The node selectors based on PAV and GreedyAV satisfy justified representation w.r.t. any γ, whereas SPAV and VoteRank do not satisfy justified representation for $\gamma = 1$.*

Proof. By the work of Aziz et al. [7] it is known that the voting rules PAV and GreedyAV satisfy justified representation. Since each node approves exactly the nodes within range γ, it is clear that also the node selectors satisfy justified representation.

For SPAV and VoteRank consider the following counter example which is based on the counterexample for multiwinner voting by Aziz et al. [7]. Let G be a network with

$$\begin{aligned}
\mathcal{N} =& \{c_1,\dots,c_{11},v_1,\dots,v_{1199}\},\\
\mathcal{E} =& \{\\
&\{v_i,c_1\},\{v_i,c_2\}, && \text{for } i \in \{1,\dots,81\} && (2.1)\\
&\{v_i,c_1\},\{v_i,c_3\}, && \text{for } i \in \{82,\dots,162\} && (2.2)\\
&\{v_i,c_2\}, && \text{for } i \in \{163,\dots,242\} && (2.3)\\
&\{v_i,c_3\}, && \text{for } i \in \{243,\dots,322\} && (2.4)\\
&\{v_i,c_4\},\{v_i,c_5\}, && \text{for } i \in \{323,\dots,403\} && (2.5)\\
&\{v_i,c_4\},\{v_i,c_6\}, && \text{for } i \in \{404,\dots,484\} && (2.6)\\
&\{v_i,c_5\}, && \text{for } i \in \{485,\dots,564\} && (2.7)\\
&\{v_i,c_6\}, && \text{for } i \in \{565,\dots,644\} && (2.8)\\
&\{v_i,c_7\},\{v_i,c_8\}, && \text{for } i \in \{645,\dots,683\} && (2.9)\\
&\{v_i,c_7\},\{v_i,c_9\}, && \text{for } i \in \{694,\dots,742\} && (2.10)\\
&\{v_i,c_7\},\{v_i,c_{10}\}, && \text{for } i \in \{743,\dots,791\} && (2.11)\\
&\{v_i,c_8\}, && \text{for } i \in \{792,\dots,887\} && (2.12)\\
&\{v_i,c_9\}, && \text{for } i \in \{888,\dots,983\} && (2.13)\\
&\{v_i,c_{10}\}, && \text{for } i \in \{984,\dots,1079\} && (2.14)\\
&\{v_i,c_{11}\}, && \text{for } i \in \{1080,\dots,1199\} && (2.15)\\
&\{v_i,v_{i+1}\}, && \text{for } i \in \{1,\dots,1198\} && (2.16)\\
&\}.
\end{aligned}$$

We want to select $K = 10$ nodes. With $\gamma = 1$ nodes $v_1,\dots,v_{1199}$ are approved by 2 to 3 nodes each, c_1,c_4 receive 162 approvals, c_2,c_3,c_5,c_6 receive 161, c_7 gets 147, c_8,c_9,c_{10} receive 145, and c_{11} receives 120 approvals.

Consider SPAV first. SPAV will select c_1 and c_4 in the first two rounds (note the neighborhoods are distinct) which reduces the voting ability of node blocks 2.1, 2.2, 2.5, 2.6 to $\frac{1}{2}$. Thus, c_2,c_3,c_5,c_6 have now only a score of 120.5, so that c_7 is selected next. This in turn reduces the scores of c_8, c_9, c_{10} to 120.5 (see blocks 2.9, 2.10, 2.11). Since $c_2, c_3, c_5, c_6, c_8, c_9$, and c_{10} have distinct neighborhoods, selecting each of them has no influence on the score of the others. They have a higher score than c_{11} (and $v_1, \dots, v_{1199}$) so they will all be selected before c_{11}. Due to $K = 10$ node c_{11} cannot be selected. However, c_{11} has $120 \geq \frac{11+1199-K}{K} = \frac{1200}{10}$ voter nodes (block 2.15) who do not approve of any of $c_1, \dots, c_{10}$. This is a contradiction to justified representation.

For VoteRank note that the average degree in the network is more than 2. Thus, the inverse degree is smaller than $\frac{1}{2}$, i.e., whenever voting abilities are reduced to $\frac{1}{2}$ in SPAV, they are reduced to some value greater than $\frac{1}{2}$ in VoteRank. It follows that $c_1, \dots, c_{10}$ will still be selected before c_{11}. Since $c_1, \dots, c_{10}$ only contribute to the scores of $v_1, \dots, v_{1199}$ which are not selected anyway, it also does not change the outcome that selected nodes cannot vote anymore in VoteRank. In conclusion, VoteRank results in the same selection of nodes which does not satisfy justified representation. □

Beyond approval-based rules there are also multiwinner rules for ordinal ranks. One of the most prominent rules is *single transferable vote (STV)*. There exist several slightly different definitions of STV in the literature. We present the one studied by Elkind et al. [29]. Adapted to networks it works as follows.

We start by initializing the committee of selected nodes $W = \emptyset$, and setting $C = V = \mathcal{N}$ where C stands for nodes in the role of candidates, and V for the role of voters. The latter is necessary because we sometimes remove a node as candidate but not as voter. Until $|W| = K$, we repeat: Set the quota to $q = \lfloor |V|/K+1 \rfloor$.[5] Determine a node (candidate) w with the highest plurality score $\rho(w) = |\{v \in V_{-w} : \text{pos}_v(w) = 1\}|$.

- If $\rho(w) \geq q$, we set $W = W \cup \{w\}$ and remove q random (voter) nodes from V who have w at first position, and we remove w from V. For all remaining preferences, we remove w and update the positions of the other (candidate) nodes accordingly (i.e., every c with $w \succ c$ improves by one position). Then we set $C = C \setminus \{w\}$.
- If $\rho(w) < q$, we choose a plurality loser l (i.e., a node with the lowest plurality score). We remove l from all preferences, update the positions of the other (candidate) nodes, and set $C = C \setminus \{l\}$. Note that l is not removed as a voter.

Another interesting criterion in the analysis of multiwinner rules is committee-monotonicity. We define it for node selectors as follows.

Definition 8 (Committee Monotonicity). *A node selector is* committee-monotonic *iff for every given network and every budget K, $1 \leq K < |\mathcal{N}|$, the following holds: If a is in a winning committee of size K chosen by the node selector, then the node selector puts a also into a winning committee of size $K+1$.*

[5]Note that Elkind et al. [29] use a quota that is higher by 1. We assume that nodes implicitly also like themselves. In consistency with [29], we also remove a selected node from $\mathcal{N}$, so in fact our quota is $q+1$.

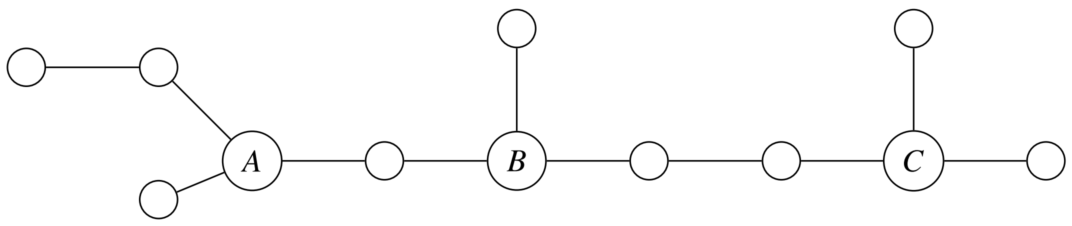


Figure 2.5: A network showing that committee-monotonicity is not always an advantage: The set $\{B\}$ is good for $K = 1$, but $\{A,C\}$ is good for $K = 2$.

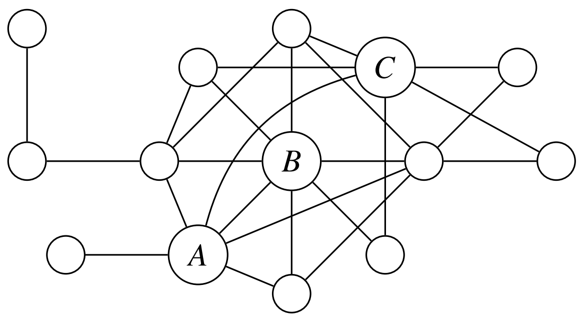


Figure 2.6: A network showing that PAV is not committee-monotonic. PAV selects the set $\{B\}$ for $K = 1$, but the set $\{A,C\}$ for $K = 2$.

Committee-monotonicity should not be confused with monotonicity in group centrality indices, which mean that the index of a set of nodes should not decrease when a node is added to the set. In multiwinner voting, committee-monotonicity is considered an advatage, or disadvantage depending on the application (see Elkind et al. [28]). In networks, however, we can easily argue that committee-monotonicity is rather a disadvantage. Consider the network in Figure 2.5 for example. When we ask the node selector for a size-1 set, certainly $\{B\}$ is a good option. However, a good size-2 set is $\{A,C\}$ rather than any set with B because such sets tend to be good for one half of the network but bad for the other.

STV is not committee-monotonic (even if we fix a tie-breaking rule), and it does exactly what we described: select $\{B\}$ for $K = 1$, and $\{A,C\}$ for $K = 2$. Figure 2.6 shows that PAV is not monotonic either. Monotonic node selectors can in fact be characterized. The following is the network correspondence to a result from Elkind et al. [28] for multiwinner voting rules.

Theorem 5. *If tie-breaking is fixed, a node selector is committee-monotonic if and only if it can be obtained from a node index by selecting the best K nodes according to that index.*

Proof. Selecting the best K nodes according to a centrality index is clearly monotonic as long as the tie-breaking rule is fixed. For the other direction, consider the following method to obtain a centrality index from a monotonic node selector: Ask for a committee X_1 of size $K = 1$ and assign the index value $n-1$ to the node in X_1. For each i in $\{2,\dots,n\}$, ask for a committee X_i of size $K = i$ and assign the index value $n-i$ to the node in $X_i \setminus X_{i-1}$. □

As a consequence, degree, closeness, betweenness, and all other centrality indices (including those based on voting rules) lead to committee-monotonic node selectors. It turns out that also w-SPAV is committee-monotonic, so PAV and STV are the only non-monotonic node selectors based on multiwinner rules we presented. Note that the node-selector versions of group closeness, group betweenness, and group degree are not committee-monotonic, too.

Theorem 6. *For fixed tie-breaking, w-SPAV (and thus, also VoteRank, SPAV, and GreedyAV) is committee-monotonic.*

Proof. Each of the K iterations of w-SPAV does not depend on K but only on the previous iterations, i.e., only on which w_i each voter contributes to the scores based on which nodes are already approved in W. Due to the fixed tie-breaking rule, all previous iterations are identical on each call, thus the previously selected nodes in W are also the same. □

2.7 Experiments

To round up this chapter, we now test the node selectors introduced in Section 2.6 on some random networks. As mentioned earlier, two metrics for group centrality are group closeness and group degree. Although the node selectors we proposed are not specifically designed for this task, we want to find out how good they behave with respect to these metrics. A more detailed study will certainly be necessary in the future (especially with more samples, different graph types, and different metrics), but the following two experiments should already provide a good indication which node selectors are of high interest. We also test the VoteRank algorithm by Zhang et al. [89] in the experiments for a comparison. The code for the experiments is available in the appendix.

In our first experiment we wanted to assess how close to the optimal group centrality values (i.e., group closeness, or group degree) the centrality values of the sets selected by the different node selectors are. For this purpose, we generated 200 random Watts-Strogatz small world networks [87] with 60 nodes. These networks consist of a ring of nodes where each node is connected with its closest neighbors (in our samples, with the 6 closest neighbors), and with some probability (here: 5%) each edge is rewired to a random other node. The name *small world* is due to the fact that in those networks the diameter and average distance between nodes is relatively small even for large networks.[6] For each graph, we compute the minimal possible group distance (respectively, maximal possible group degree) a set of size K can have, i.e., the optimum. Afterwards, we let each node selector compute a set of nodes of size K, and compute the difference in group distance (respectively, group degree) of this set to the optimum. The results are given in Tables 2.1 and 2.2. The best results (i.e.,

[6]In a famous experiment Stanley Milgram [63] showed that the same properties apply also to real social networks. In his experiment he selected random people all over the US, and asked them to forward a letter to a friend who is most likely to know a given target person. The surprising result was that most letters reached the target person within 5 hops, and no letter needed more than 10.

	$K=2$			$K=3$			$K=4$		
	lq	avg	uq	lq	avg	uq	lq	avg	uq
VoteRank	0.07	0.21	0.31	0.05	0.13	0.18	**0.04**	**0.08**	**0.12**
STV	**0.02**	**0.11**	**0.16**	**0.04**	**0.09**	**0.14**	**0.04**	0.09	**0.12**
GreedyAV	0.07	0.21	0.31	0.05	0.15	0.19	0.05	0.10	0.14
SPAV	0.09	0.22	0.31	0.07	0.16	0.23	0.05	0.11	0.16
Closeness	0.22	0.36	0.47	0.32	0.47	0.58	0.38	0.53	0.66
Degree	0.14	0.33	0.43	0.18	0.33	0.42	0.18	0.31	0.39

Table 2.1: Results of the first experiment. Lq shows the lower quartile, avg the average, and uq the upper quartile for the difference between the optimal set distance, and the one achieved by the node selectors over 200 sample Watts-Strogatz networks.

	$K=2$			$K=3$			$K=4$		
	lq	avg	uq	lq	avg	uq	lq	avg	uq
VoteRank	0	**0.01**	0	0	**0.11**	0	0	**0.20**	0
STV	0	0.59	1	0	1.00	2	0	1.15	2
GreedyAV	0	0.17	0	0	0.33	1	0	0.65	1
SPAV	0	0.10	0	0	0.40	0	0	0.77	1
Closeness	2	3.64	5	5	7.24	9	9	11.12	14
Degree	0	1.03	2	1	2.82	5	2	4.45	7

Table 2.2: Results of the first experiment. Lq shows the lower quartile, avg the average, and uq the upper quartile for the difference between the optimal set degree, and the one achieved by the node selectors over 200 sample Watts-Strogatz networks.

lowest difference to optimal) are marked in boldface. We can clearly see that STV outperforms all other node selectors in group distance for $K = 2$ and $K = 3$, and is only slightly behind VoteRank for $K = 4$. The good performance of STV is no surprise, of course: STV is designed for ordinal preferences which in our setting encode for distances, while VoteRank, GreedyAV, and SPAV consider only direct neighbors. This in turn gives the latter three methods a huge advantage when the selected set should have many neighbors. VoteRank is by far the best in this metric. However, also GreedyAV and SPAV perform much better than STV. Finally, note that closeness and degree centrality (*not* in the group centrality version) perform very bad. This underlines once again that node selectors need more sophisticated design than just selecting the individually best K nodes, and voting rules adapted as node selectors successfully achieve this.

Note that computing the optimal solution is computationally hard. Thus, in our first experiment we are limited to networks with only 60 nodes. The same experiment is simply not realizable with sufficiently high sample rate on larger networks (the experiment takes already several hours in the current version). However, note that our node selectors are far more

	$K = 2$		$K = 3$		$K = 4$		$K = 10$	
Graph Size	500	1000	500	1000	500	1000	500	1000
VoteRank	14.5%	3.5%	6.5%	2.5%	5.0%	3.5%	2.0%	1.0%
STV	**79.5%**	**90.0%**	**89.0%**	**92.0%**	**88.0%**	**93.0%**	**93.0%**	**96.5%**
GreedyAV	12.5%	6.5%	4.5%	4.5%	5.0%	2.0%	3.0%	2.0%
SPAV	14.5%	4.5%	4.5%	4.0%	5.5%	4.0%	2.5%	1.0%

Table 2.3: Results for the second experiment. The percentage is the fraction of our 200 sample graphs where the respective node selector performs best in terms of the group closeness.

	$K = 2$		$K = 3$		$K = 4$		$K = 10$	
Graph Size	500	1000	500	1000	500	1000	500	1000
VoteRank	**100%**	**100%**	**99.5%**	**99.5%**	**99.5%**	**100%**	**96.5%**	**96.0%**
STV	14.5%	17.5%	7.0%	6.0%	6.5%	2.5%	6.0%	0.5%
GreedyAV	99.5%	99.5%	96.5%	97.0%	95.0%	97.5%	78.0%	81.5%
SPAV	98.0%	100%	98.0%	98.5%	94.5%	97.5%	69.0%	78.0%

Table 2.4: Results for the second experiment. The percentage is the fraction of our 200 sample graphs where the respective node selector performs best in terms of the group degree.

efficient, and take only few seconds to compute even on significantly larger networks. Thus, we conduct a second experiment with a different setup and larger graphs.

In our second experiment we take Watts-Strogatz networks once again. This time the networks are significantly larger with 500, or $1,000$ nodes each. Since we cannot compute the optimum values on such large networks, we just count for each node selector on how many of our sample networks it yields the best set (among the node selectors in our test). When multiple selectors yield an equally good set, we count it as a victory for all of them. Note that comparisons of the type 'how much better is X compared to Y on this graph' don't really make sense when we don't know the optimal values. So we decided to just count as described above. The results are shown in Tables 2.3 and 2.4. The second experiment confirms the results from our first experiment: STV performs very good in terms of group closeness, while VoteRank is usually the best choice for group degree.

2.8 Conclusions

In this chapter we showed that network science and social choice overlap in the field of network centrality and voting theory. Not only are some axioms similar, or interchangeable;

we even identified further axioms from social choice—the Condorcet criterion and justified representation—which could be very interesting for network science.

We developed new centrality indices and node selectors based on voting rules. The Copeland centrality index based on the same name voting rule is (as far as we know) the only Condorcet-consistent centrality index yet. This makes it a very interesting candidate for future studies in network science when the intended application seems to benefit from Condorcet-consistency. Another very interesting candidate for future studies is the multiwinner voting based node selector STV. In our experiments it often produced sets of nodes with a very good group closeness (i.e., the average distance from nodes outside the set to the most proximate node in the set is very small). It will be interesting to test STV in the future in other network science applications. For example, STV might improve viral marketing. To this end, we propose testing it on the SIR infection model as Zhang et al. [89] did with VoteRank.

We further want to point out a result which goes quite in the opposite direction than the results so far: Voting theory might also be influenced by network science. We showed that the VoteRank method [89] is very similar to the class of w-SPAV multiwinner voting rules. The fact that VoteRank outperformed two of the most common w-SPAV rules GreedyAV and SPAV could lead to new insights in voting theory. A future direction of research could be to translate VoteRank back to multiwinner voting, and test whether the rule yields better results than GreedyAV and SPAV in voting, too.

2.9 Publication

Parts of this chapter appeared as extended abstract at the *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)* in 2022.

U. Brandes, C. Laußmann, and J. Rothe. "Voting for Centrality (Extended Abstract)". In: *International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2022

2.10 Personal Contribution

The idea of linking single-winner voting rules and centrality indices is due to Ulrik Brandes. Linking group centrality and node selectors to multiwinner voting is my idea. The theoretical results in Sections 2.5.2, 2.5.4 (published as Theorem 3.2 in [21]), and 2.6 were proven by me with support from Jörg Rothe. Further, I did the entire experimental study in Section 2.7. The results on positional scoring rules (Section 2.5.3 in this work, and further results in [21] not included in this chapter) were proven by Ulrik Brandes. The writing of the publication [21] was conducted in equal parts by Ulrik Brandes, Jörg Rothe, and me.

CHAPTER 3

Participatory Budgeting under Uncertainty

3.1 Summary

In this chapter, we introduce a new model for participatory budgeting, which captures situations, where the cost and durations of projects underlies some uncertainty in advance. Further, we are also given a time constraint on the total project execution time and the total implementation time. In the first part of the chapter, we define properties which are arguably desirable for a budgeting method in this scenario. We show that, whenever cost is uncertain, there exists no perfect method because some properties are incompatible. This is why we design *best effort budgeting methods* for scenarios where cost is uncertain. Although these methods cannot guarantee to satisfy *all* properties, they guarantee a maximal subset of properties, and do their best to satisfy also the other properties.

3.2 Introduction

Let us start with an example.

Example:

Chris travels through Italy. One evening he visits the pub of the village *Budgètta*. While drinking his red wine he hears the people at the next table talking about a participatory budgeting campaign.

> *"I have to say, I liked the idea of refurbishing our historical craftsman museum. Our city's tradition in handcrafting canoes must be credited!"*

sais the old man who wears an odd hat with two black feathers in it. The woman next to him responds nodding.

> *"Absolutely! But the price was 50% higher than estimated. Even worse: the project was delayed by 3 months! Now there isn't enough money left to fund the new roof for the train station. The major could have planned that better."*

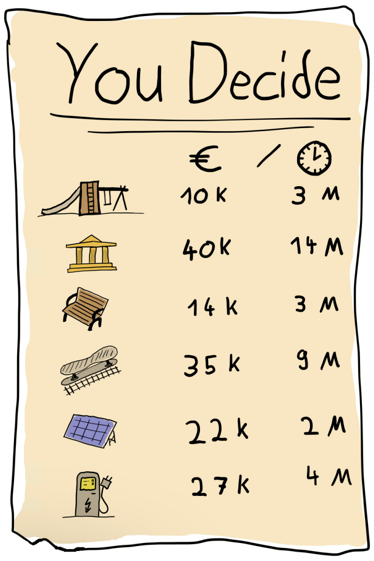
A young man in a gray suit on the other side of the table agrees.

> *"You are so right, Maria! Everyone knew that there was some risk that the refurbishing becomes far more expensive. I would have implemented the solar panels on public buildings and the electric vehicle charging stations first. At least we knew relatively precisely how much they cost."*

The old man is gesturing wildly with his hands. His long beard shakes while he talks with a much louder voice than before.

> *"Come on. The museum was the most liked project in the whole campaign. It makes absolutely sense to implement it. And look: it could also have been cheap, and you would have gotten your solar panels afterwards."*

Later that evening while Chris walks home tipsy from the red wine, he finds an old poster at a wall which advertises the participatory budgeting campaign. Chris reads it, and asks himself how such campaigns could be planned better.



The poster above shows a usual setting of participatory budgeting. Projects are associated with cost and time, and we have both, cost and time limits, for the whole campaign. A campaign manager then ask the voters which projects should be implemented, and based on these votes decides which subset of projects will be implemented.

However, what Chris heard in the pub tells us that the scenario on the poster is not very realistic. The museum took more time, and costs much more than written on the poster. That is, the exact cost and duration of a project are usually only known after finishing it. We can also observe this in many real-world projects. For Germans, projects like the Berlin airport *BER*, the *Elbphilharmonie* concert hall in Hamburg, and the infrastructure project *Stuttgart 21* come into mind. All of these projects exceeded their time limit and budget by huge amounts. But cost explosions and heavy delays also happen on municipal level. Thus, such uncertainty should also be considered in participatory budgeting.

A common way to accommodate this uncertainty is to *estimate* cost and duration. These estimates are usually submitted by the company, or person, who proposes the project, or the project manager collects offers from companies. But estimations come with problems. Not only are estimates often inaccurate. They can also be strategic (and manipulated). For instance, on the one hand, if the estimated cost is far too high, the company will probably not get the contract. On the other hand, if it is far too low, the company is unable to realize the project. In the latter case, either the company gets more money from the campaign manager, or the company is bankrupt, and leaves an unfinished project behind (which will also require the campaign manager to pay another company to finish it). To some degree this encourages companies to submit too optimistic offers, while companies which plan with some safety margin are punished. Thus, we think a better approach is to provide a range, and a probability distribution for both, cost and duration. Then, the campaign manager can include this uncertainty in the planing process. We summarize the problem types (and where they are discussed) in the following table.

	Cost known	**Cost uncertain**
Duration known	Regular PB (Section 3.4)	Section 3.6
Duration uncertain	Section 3.5	Section 3.7

The contributions of this chapter are mainly the ones in Section 3.6, whereas the other sections are mainly a literature review and explanation of the problems.

3.3 Preliminaries

Let $A = \{a_1, \dots, a_m\}$ be the set of *projects* and each subset $B \subseteq A$ is a *bundle*. We are given a time limit $\tau \in \mathbb{N}^+$ and a budget $\ell \in \mathbb{N}^+$. The projects are evaluated by a set of *voters* $V = \{v_1, \dots, v_n\}$. Each voter v approves a subset of projects denoted by $\text{app}_v \subseteq A$. By $s_v(B) = |B \cap \text{app}_v|$ we denote the *satisfaction* of voter v with bundle B, i.e. the number of items in B approved by v. We define $s(B,V) = \sum_{v \in V} s_v(B)$ as the *total satisfaction* of all voters (we omit V when it is clear from the context). We assume $s(\{a\}) > 0$ for all $a \in A$, i.e., each project is approved by at least one voter.

Projects are also associated with cost and durations. In the following sections we distinguish the cases where none, one, or both of them are uncertain. In the typical participatory budgeting setting [84, 90, 4] both, cost and durations are known.[1] This is modelled by the cost function $c : A \to \mathbb{N}^+$ and the duration function $\delta : A \to \mathbb{N}^+$. We assume $c(a) \leq K$ and $\delta(a) \leq \tau$ for all projects $a \in A$. In case duration is uncertain, we are given the tuple $\widetilde{\delta} = (\delta_{min}, \delta_{max}, \delta, \delta_p)$, where $\delta_{min} : A \to \mathbb{N}^+$ and $\delta_{max} \to \mathbb{N}^+$ provide the lower and upper bounds on the project's duration, while $\delta \to \mathbb{N}^+$ models the exact duration. It holds that $\delta(a) \in [\delta_{min}(a), \delta_{max}(a)]$. By $\delta_p(a,x)$ we denote the probability that project $a \in A$ takes at most time $x \in \mathbb{N}^+$ to finish. Overloading notation, by $\delta_p(B,x)$ for a set B we denote the probability that every project from B takes time at most x. We assume in accordance with $\delta(a) \leq \tau$ that $\delta_{max}(a) \leq \tau$ for all projects $a \in A$. In case cost is uncertain, we are given $\widetilde{c} = (c_{\min}, c_{\max}, c, c_p)$, where $c_{\min} : A \to \mathbb{N}^+$ and $c_{\max} : A \to \mathbb{N}^+$ model lower and upper bounds on the project's costs, while $c : A \to \mathbb{N}^+$ are the exact costs. Note that $c(a) \in [c_{\min}(a), c_{\max}(a)]$. By $c_p(a,x)$ we denote the probability that project $a \in A$ costs at most $x \in \mathbb{N}^+$. Again, overloading notation, by $c_p(B,x)$ we denote the probability that all projects from the set B together costs at most x. We assume in accordance with $c(a) \leq \ell$ that $c_{\max}(a) \leq \ell$ for all projects $a \in A$. We abuse notation by denoting $c(B) = \sum_{a \in B} c(a)$ as the cost of a bundle B (analogously with $c_{\min}$ and $c_{\max}$).

A budgeting scenario E is given by a tuple $(A, V, c, \ell, \delta, \tau)$ where c is replaced by $\widetilde{c}$ when cost is uncertain, and δ is replaced by $\widetilde{\delta}$ when duration is uncertain. A budgeting scenario is processed by an *online budgeting method*. Such a method works in discrete time steps, and successively builds a *budgeting log* $L : A \to \mathbb{N}_0 \cup \{\perp\}$ representing at which time step a project has been started, where $\perp$ denotes that the project will not be realized at all. That is, formally the output of an online budgeting method $\mathcal{R}(E)$ is a budgeting log. We further define the set of *realized* projects for a budgeting log L as $R(L) = \{a \in A \mid L(a) \neq \perp\}$. Finally, for every $t \in [\tau]$, let $U(L,t) = \{a \in A \mid L(a) \leq t < L(a) + \delta(a)\}$ be the *running yet unfinished* projects, and $F(L,t) = \{a \in A \mid L(a) + \delta(a) \leq t\}$ the *finished* projects.

Depending on what uncertainties occur in the scenario, the budgeting method has limited access to the cost function c, or the project durations δ. In case of uncertain cost, if $L(a) = t^*$, then $c(a)$ is available only after the project has been implemented, i.e. at time step $t^* + \delta(a)$. Obviously, the decision made at step t^* is fixed and may not be revised when more information is available. Similarly, when time is uncertain, the online budgeting method has access to δ only after the project is finished, i.e., at time step $t^* + \delta(a)$. Note that this can be particularly difficult for the method, since it doesn't even know when additional information becomes available. An *offline* budgeting method has access to the exact cost function c and the exact durations δ at any time.

A budgeting log, and thus also an online budgeting method, has rather weak requirements. For instance, it is allowed in a budgeting log to start arbitrarily many projects simultaneously, even if they will certainly exceed the budget limit; or to start projects so late that they cannot be completed in time. These issues are undesirable and should be avoided. Thus, we now introduce axioms to describe the properties of online budgeting methods.

[1]In regular PB the time limit is usually not given, because it can be ignored as we see in the next section.

Definition 1. *Let E be a budgeting scenario and L be a budgeting log with respect to E. L satisfies the following axioms if respective conditions are met.*

Risk-free (RF): *The budget may never be exceeded. Formally, $c(R(L)) \leq \ell$.*

Punctuality (PU): *Every realized project finishes within the given time limit. Formally, for all $a \in A$ it holds that either $L(a) = \perp$ or $L(a) + \delta(a) \leq \tau$.*

Exhaustiveness (EX): *There should be no project, which could have been implemented even with maximum cost without breaking feasibility. Formally, for $B = R(L)$ and every $a \in A \setminus B$, it holds that $c(B) + c_{\max}(a) > \ell$.*

α-Risk-assessment (α-RA): *A (set of) project(s) may only be started if the probability for exceeding the budget limit is at most α. Formally, given $\alpha \in [0,1)$, it holds that a set of projects S may only be started at time t if $c_p(U(L,t) \cup S, \ell - c(F(L,t))) \geq 1 - \alpha$.*

ψ-Delay-risk (ψ-DR): *A (set of) project(s) may only be started if the probability for exceeding the time limit is at most ψ. Formally given $\psi \in [0,1)$, a set of projects S may only be started at time t if $(\prod_{a \in U(L,t)} \delta_p(a, \tau - L(a))) \cdot \delta_p(S, \tau - t) \geq 1 - \psi$.*

κ-Limitation (κ-LI): *The budget limit may not be exceeded by a factor greater than κ. Formally, $c(R(L)) \leq \kappa\ell$.*

A budgeting method $\mathcal{R}$ satisfies some axiom χ if $\mathcal{R}(E)$ satisfies χ for every budgeting scenario (regardless of the tie-breaking method).

Note that 0-risk-assessment, 1-limitation, and the risk-free property coincide. Further, punctuality and 0-delay-risk coincide. More generally we can say that α-RA and κ-LI are relaxations of RF, and that ψ-DR is a relaxation of PU.

Risk-assessment and limitation can be interpreted as follows. The client has $(\kappa - 1)\ell$ extra money as a security, or loan option which should be used only if absolutely necessary. With a good risk-assessment (i.e. small α) it is improbable that the security is ever touched. Exhaustiveness has two interpretations. First, voters naturally expect that approved projects are realized if there is money left to do so safely. Second, it is common that the budget of a department may be reduced in the next period if it is not completely spent. Not all axioms make sense for all types of uncertainty. That is, when cost is known in advance, it makes no sense to analyze α-risk-assessment or κ-limitation. Analogously, it makes no sense to analyze ψ-delay-risk when durations are known. One might be tempted to analyze κ-limitation even when cost is known. The definition would actually work fine. However, it is still not useful as without uncertainty it is just equivalent to increasing the budget limit. It should really only be considered together with α-risk-assessment because we want to model that the budget should not be exceeded with high probability, but if we have bad luck, it is at least not exceeded by more than the factor κ.

Independent of the above properties we want to maximize the satisfaction of the voters with the outcome. One key metric for the analysis of online optimization algorithms is the worst-case ratio between a solution found by an online algorithm and an optimal (satisfaction maximizing) solution with complete knowledge. This factor is known as *competitive ratio* (CR) (see Fiat and Woeginger [37]).

Definition 2 (Competitive Ratio (CR)). *An online budgeting method $\mathcal{R}$ is σ-competitive (σ-CR) if there is a constant $\Delta \in \mathbb{R}$, such that for every $E \in \mathcal{E}$ and $\mathcal{B}_\ell = \{B \subseteq A \mid c(B) \leq \ell\}$ it holds that $s(R(\mathcal{R}(E))) + \Delta \geq \frac{1}{\sigma} \max_{B \in \mathcal{B}_\ell} s(B)$.*

Note that there exist offline algorithms which compute an optimal solution, and simultaneously are exhaustive, risk-free, and punctual. The simplest of such algorithms just checks for every subset $B \subseteq A$ whether (1) $c(B) \leq \ell$, and (2) whether there exists no project $p \in A \setminus B$ such that $c(p) + c(B) \leq \ell$. The first condition guarantees that implementing the bundle is risk-free, and the second condition guarantees exhaustiveness. Note that punctuality is always guaranteed since $\delta(a) \leq \tau$ for each $a \in A$, and all projects can be started simultaneously at time 0. From the bundles which pass both checks we can select the bundle with the highest satisfaction (with arbitrary tie-breaking).

3.4 No Uncertainty

Let us first consider a scenario where we have no uncertainty in both cost and durations. That is, the budgeting scenario E is given by a tuple $(A, V, c, \ell, \delta, \tau)$. We call this the *classical participatory budgeting* because it corresponds to participatory budgeting as it is used in reality, and well studied in literature (see Aziz and Shah [4] for an overview). Note that in this scenario offline and online budgeting methods are equivalent, as they both have access to the same information. Thus, the following remark.

Remark 1. *When exact cost and duration, are known in advance, there exists an online budgeting method which is exhaustive, risk-free, punctual, and has a competitive ratio of* 1.

Note that in typical definitions—as in the formal participatory budgeting framework for approval-based preferences introduced by Faliszewski and Talmon [84] and extended to irresolute budgeting rules by Baumeister et al. [11]—durations of projects, and time limits are usually omitted. This is because for each project a holds $\delta(a) \leq \tau$, i.e., each project can be implemented when we start it at time step 0 (formally, $L(a) = 0$). And since we know all information (particularly the cost) in advance, we can simply compute which projects we want to implement, and then start them simultaneously at time step 0.[2] Thus, the whole scenario can be simplified even more. We don't even need the complicated notion of a budgeting log—we can just directly provide the set of realized projects as solution.

[2]The budgeting log allows us to also start projects later, but there is really no reason to do so.

3.5 Uncertain Duration

Now we introduce uncertainty about the *durations* of projects. Therefore, each project is associated with minimum duration, maximal duration, exact duration, and a probability distribution over the durations. As mentioned, this information is provided by the tuple $\widetilde{\delta} = (\delta_{min}, \delta_{max}, \delta, \delta_p)$. Since the exact cost of each project is known, for each bundle we can still compute whether it is exhaustive, and whether it fits in our budget. We can even determine whether it is optimal w.r.t. the satisfaction of the voters. Due to $\delta_{max}(a) \leq \tau$ for all projects $a \in A$, we can also guarantee punctuality by starting all projects at time step 0. Thus, introducing uncertainty in the project durations doesn't really change anything.

Remark 2. *When exact costs are known, but durations are uncertain in advance, there still exists an online budgeting method which is exhaustive, risk-free, punctual, and has a competitive ratio of* 1.

However, note that the problem becomes much more interesting now when projects have to be implemented sequentially. Reasons can be limited parallelization capabilities (e.g. we have only three workers, so only three projects can run in parallel), or dependencies between the projects (e.g. we can independently decide on refurbishing the museum, and on installing solar panels, but if we want both, we first have to refurbish the museum because parts of the roof will be replaced; see e.g. Rey et al. [73]). When project durations are uncertain, and some projects must be implemented sequentially, it is indeed an interesting question which projects to implement at all, and how to schedule them in order to minimize risk for exceeding the time limit (see ψ-delay-risk). Similar problems are studied in scheduling and project planning literature where tasks have to be scheduled while respecting their dependencies and resource requirements. Such problems have been studied with uncertain project durations e.g. by Ma et al. [61] and Moradi and Shadrokh [66]. Further, we want to mention the work by Vaziri et al. [86], who analyze project planning where the time for tasks is uncertain and can be influenced by the resources allocated to that task. Note however, that scheduling and project planning is different from participatory budgeting since in participatory budgeting we only select and implement a *subset* of projects while in scheduling and project planning we have to finish *all* tasks.

3.6 Uncertain Cost

Let us now consider the case where durations are known, but cost is uncertain. Gomez et al. [44] present a participatory budgeting model considering uncertainty for both, cost and satisfaction. In contrast to ours, their model is purely stochastic (a set of projects is feasible if its expected cost is within the budget limit). Further, in their model projects don't have durations and are implemented all at the same time. Similar considerations were studied in KNAPSACK literature, which is closely related to classical participatory budgeting. In

KNAPSACK, we are given items associated with value and weights, and try to maximize the value of a set of projects without breaking a weight limit (see Kellerer et al. [51] for an introduction to the problem). It has been studied under uncertain weights by Monaci et al. [64, 65]. They aim to find solutions which perform well even if the exact weights turn out to be unfavorable. The KNAPSACK variant by Goerigk et al. [43] allows for querying the exact weight of a fixed number of items in order to find a good solution when weights are uncertain. However, KNAPSACK does not incorporate durations and time limits as we do.

In our model we are given the uncertain cost functions by the tuple $\widetilde{c} = (c_{\min}, c_{\max}, c, c_p)$. Now, it is not that simple anymore to precompute bundles which are exhaustive and risk-free. It is easy to see that for a risk-free bundle we have to work with the upper cost limits provided by $c_{\max}$. But this can result in a lot of money being left in the end which potentially violates exhaustiveness. Of course, we could wait for the projects to finish, and if money is left we implement further projects (assuming their maximum cost) until the bundle is exhaustive. However, this certainly brakes punctuality at some point. We can conclude that there is one major incompatibility in this setting. *We cannot be punctual, risk-free, and exhaustive at the same time.* The following two results show that we can even strengthen this result further as it holds for every reasonable relaxation of RF.[3]

Theorem 1. *The following incompatibilities hold when cost is uncertain.*

1. *For any fixed $\alpha < 1$, no online budgeting method simultaneously satisfies α-risk-assessment, punctuality, and exhaustiveness.*
2. *For any fixed $\kappa < m$, no online budgeting method simultaneously satisfies κ-limitation, punctuality, and exhaustiveness.*

Proof. Let us start with the first claim. Consider a budgeting scenario with an odd budget limit $\ell \geq 3$, and the projects $A = \{a_1, a_2, a_3, \ldots\}$. Each project $a_i \in A$ has minimum cost $c_{\min}(a_i) = {}^{(\ell-1)}/_2$, maximum cost $c_{\max}(a_i) = {}^{(\ell+1)}/_2$, and takes time $\delta(a_i) = \tau$. Note that a set of two projects exceeds the budget limit if and only if both projects have maximum cost. Let each project have maximum cost with probability greater than $\sqrt{\alpha}$. Due to α-risk-assessment a budgeting method can only start one project at the first time step, say a_1. By punctuality it is impossible to start another project. Since $c(a_1) = {}^{\ell}/_2$ is possible, there are instances for which $c(a_1) + c_{\max}(a_2) \leq \ell$, thus exhaustiveness is violated.

Now focus on the second claim. Consider $E \in \mathcal{E}$ with projects $A = \{a_1, \ldots, a_m\}$, $m \geq 3$ and a budget of $\ell > m$. Each project $a_i \in A$ takes time $\delta(a_i) = \tau$ to realize, has cost $c(a_i) = 1$, and maximum cost $c_{\max}(a_i) = \ell - m + 1$. By exhaustiveness, all projects must be realized, since for each $a_i \in A$ holds $c(A \setminus \{a_i\}) + c_{\max}(a_i) = \ell$. Further, by punctuality all projects must be started simultaneously. However, this decision has to be made without knowing the exact cost. Let $E' \in \mathcal{E}$ be equivalent to E, except for that each project has the maximum cost of

[3]Note that for $\kappa \geq m$ we can simply start every project at time step 0 since $c_{\max}(a) \leq \ell$ holds for every $a \in A$. This way we achieve κ-limitation, exhaustiveness, and punctuality at the same time. However, this cannot be considered a reasonable relaxation for risk-free.

$\ell - m + 1$ as exact cost. An online budgeting method that implements all projects to satisfy exhaustiveness and punctuality might end up spending $c_{\max}(A) = m \cdot (\ell - m + 1)$. Thus, to start all projects, it cannot be better than $\frac{m \cdot (\ell - m + 1)}{\ell} = \left(m - \frac{m^2 + m}{\ell}\right)$-limited. By choosing ℓ large, we can approach m to any fixed value $\kappa < m$. $\square$

Now we know that we cannot achieve exhaustiveness and punctuality in combination with any reasonable risk-assessment, or limitation. But can we achieve at least every two of them? Indeed, this is possible.

Theorem 2. *The following algorithms are possible when only cost is uncertain.*

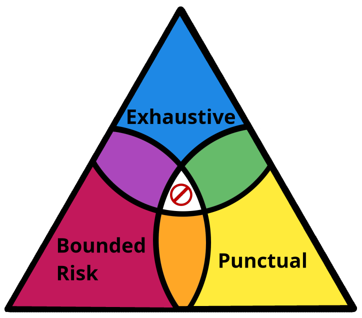
1. *There exists an algorithm which is risk-free, and simultaneously punctual.*
2. *There exists an algorithm which is risk-free, and simultaneously exhaustive.*
3. *There exists an algorithm which is punctual, and simultaneously exhaustive.*

Proof. The first claim is proven by the following algorithm. We start the project with the highest voter satisfaction, say a_1, which has by definition maximum cost of at most ℓ. To achieve punctuality, we stop now.

For the second claim, we first start the project with the highest voter satisfaction, and then every time a project finishes we start another project which can be safely added without exceeding the budget (i.e., assuming maximum cost). If no such project exists anymore, we achieved exhaustiveness, and stop.

The algorithm that proves the third claim, simply starts all projects at the first time step. Note that the set of all projects is guaranteed to be exhaustive. Since $\delta(a) \leq \tau$ for all projects $a \in A$, the algorithm is punctual. $\square$

We can illustrate our results so far by the following *triangle of participatory budgeting under uncertainty*. According to Theorem 1, for every reasonable bound risk it is impossible to achieve the other two, exhaustiveness and punctuality, simultaneously. However, every combination of two of the three is possible according to Theorem 2. Note the similarity to the well-known principle in project management: *fast, good, cheap; pick two*.



Note that the first two algorithms in the proof of Theorem 2 are m-competitive, since $s(A) \leq m \cdot s(\{a_1\})$. Further, the third algorithm is even 1-competitive. However, the third algorithm is certainly not usable at all since it completely ignores risk. But can an algorithm with proper risk management guarantee any better competitiveness than the first two algorithms?

Theorem 3. *The following holds when only cost is uncertain. For any online budgeting method that satisfies α-risk-assessment for a fixed $\alpha < 1$, the competitive ratio is in $\Omega(m)$. If $c_{\max}(a) = c_{\min}(a) + 1$ holds for all $a \in A$, the competitive ratio is in $\Omega(2)$.*

Proof. Consider $E \in \mathcal{E}$ with $A = \{a_1, a_2, a_3, \ldots, a_m\}$ and a set of voters, such that each project $a_i \in A$ yields the same (additive) satisfaction $s(a_i) = \lambda \in \mathbb{N}^+$. Let $\ell = m$, $c_{\min}(a_i) = 1$ and $c_{\max}(a_i) = m$ for every $a_i \in A$. Further, for each a_i we set the probability that a_i costs exactly m to α (thus, $c_p(a_i, \ell - 1) = 1 - \alpha$). An online algorithm with α-risk-assessment cannot start more than one project at the same time because for every pair of projects $a_i \neq a_j$ it holds $c_p(\{a_i, a_j\}, \ell) \leq c_p(a_i, \ell - 1) \cdot c_p(a_j, \ell - 1) = (1-\alpha)^2 < 1 - \alpha$. Thus it starts at most one project, for example a_1. Revealing $c(a_1) = m$ and $c(a_i) = 1$ for $i \in [2, m]$, an offline algorithm may select the optimal solution $B = A \setminus \{a_1\}$ with $s(B) = \lambda \cdot (m-1)$, while the online algorithm yields a satisfaction of $s(\{a_1\}) = \lambda$. Note that since a_1 already consumed all the budget, the online algorithm cannot start any further projects anymore. Overall we deduce a competitive ratio of $(m-1) \in \Omega(m)$.

For bounded uncertainty by $c_{\max}(a) = c_{\min}(a) + 1$ for all $a \in A$, we can use a similar argument. Let $\ell \geq 2$ be an even number, $A = \{a_1, a_2, a_3, \ldots, a_m\}$ with $c_{\min}(a_i) = \ell/2$ and $c_{\max}(a_i) = \ell/2 + 1$. Again, we assume equal utility of λ for every project. We set $c_p(a_i, \ell/2) = 1 - \alpha$, so the probability that two projects can be implemented within ℓ is $(1-\alpha)^2 < 1 - \alpha$. Let an online budgeting method implement a_1 first (due to α-RA it cannot implement two projects). Revealing $c(a_1) = \ell/2 + 1$ and $c(a_2) = c(a_3) = \ell/2$, the optimal solution is $\{a_2, a_3\}$, yielding a competitive ratio of $\frac{2\lambda}{\lambda}$. □

We can even show that the $\Omega(2)$-bound on the competitive ratio is tight.

Theorem 4. *The following holds when only cost is uncertain. If the uncertainty on the cost is bounded by a small factor c^*, that is, $c_{\max}(a) - c_{\min}(a) < \frac{c_{\max}(a')}{m} = c^*$ for all $a, a' \in A$, there is a 2-competitive, risk-free method satisfying either punctuality or exhaustiveness.*

Proof. We use the optimal offline method to retrieve an optimal bundle B, assuming the lower bound cost for each project, i.e. $\sum_{b \in B} c_{\min}(b) \leq \ell$.
Case 1: If $\sum_{b \in B} c_{\max}(b) \leq \ell$, we are done.
Case 2: Otherwise, since implementing B may exceed the budget limit, we remove the least valuable project $a \in B$ and implement $B' = B \setminus \{a\}$ at the first time step. It holds that $\ell \geq \sum_{b \in B} c_{\min}(b) \geq \sum_{b \in B} c_{\max}(b) - |B| c^* \geq \sum_{b \in B} c_{\max}(b) - c_{\max}(a) = \sum_{b \in B'} c_{\max}(b)$, due to $|B| c^* \leq |B| \cdot \frac{c_{\max}(a)}{m} \leq c_{\max}(a)$. On the other hand, since a is the least valuable project in B, the satisfaction with B' is at least $s(B') \geq s(B) \cdot \frac{|B'|}{|B|} = s(B) \cdot \frac{|B|-1}{|B|}$. The worst competitive

ratio of two is achieved if $|B| = 2$, since $|B| = 1$ is already covered by case 1. We now start B' at time step 0 which guarantees punctuality. To achieve exhaustiveness, we wait for the projects in B' to finish, and then successively start other projects (which even with maximum cost don't exceed the budget limit) until the set of realized projects is exhaustive. Note that in both cases there is no risk for exceeding the budget limit. □

3.6.1 Best-Effort Algorithms

The results so far are rather disappointing. We have seen that there is no online budgeting method that guarantees all the desirable properties when cost is uncertain. And even if some properties are compatible, the competitiveness is often rather bad in worst case. However, this is only a worst-case analysis so far. In this section, we design "best effort" algorithms which trade off the desirable properties, and aim for a good average competitiveness.

First, we propose the online budgeting method *"Best Effort Exhaustiveness" (BEE)* which trades exhaustiveness against punctuality, risk-assessment, and limitation. That is, the method guarantees punctuality, α-risk-assessment, and κ-limitation for given τ, α, κ, but not exhaustiveness—which it cannot guarantee according to Theorem 1. However, it tries to be as exhaustive as possible. The algorithm generalizes a common greedy algorithm for Knapsack (see Kellerer et al. [51]) to our setting. Essentially, we rank projects by their expected cost per value ratio. That is, projects with low expected cost, but high satisfaction for the voters are ranked high. Then, favoring the highest ranked projects, we start as many projects as possible without violating risk-assessment, limitation, and punctuality. Whenever projects finish, we start new projects following the same system. We provide a formal description of the algorithm in Algorithm 1, whereby $c_{exp}(a)$ we denoted the expected cost of project a, which can be computed from the probabilistic cost function c_p. Due to technical reasons we use sampling to narrow c_p for sets of projects.[4]

With an extension of the BEE algorithm, we get the *Best Effort Punctuality (BEP)* algorithm (provided in Algorithm 2) which trades punctuality against exhaustiveness, risk-assessment, and limitation. Here, we take the result of BEE and make it exhaustive while exceeding the time limit as little as possible.

Experimental Setup

To put the best effort algorithms to test, we use real datasets from the Participatory Budgeting Library (PABULIB, see Stolicki et al. [83]). These datasets include approval ballots from the voters as well as the corresponding projects with known cost but without durations. We

[4]Computing c_p for sets of projects precisely, i.e., the composition probability distribution of several discrete probability distributions, is computationally difficult in general. Our sampling approach just determines whether starting a set of projects is likely to violate α-risk-assessment. To this end, we draw $1,000$ costs at random from all projects in the set, and start the set only if in less than $\alpha \cdot 1,000$ cases we exceed the budget limit. The code is available in the appendix.

Algorithm 1 Best Effort Exhaustiveness (BEE)

```
$t^* \leftarrow 0$
$u(a) \leftarrow \frac{s(\{a\})}{c_{exp}(a)} \quad \forall a \in A$   $\{$Rating by expected value per cost$\}$
$L(a) \leftarrow \perp \quad \forall a \in A$
while $t^* \leq \tau$ do
    $Y \leftarrow \{a \in A \setminus (U(L,t^*) \cup F(L,t^*)) \mid \delta(a) \leq \tau - t^*\}$
    while $Y \neq \emptyset$ do
        let $a \in Y$ be the project with maximum $u(a)$
        remove $a$ from $Y$
            $\{$Ensure $\alpha$-RA, and $\kappa$-LI$\}$
        if $c_{\max}(a) \leq \kappa\ell - c(F(L,t^*)) - c_{\max}(U(L,t^*))$ and $c_p(U(L,t^*) \cup \{a\}, \ell - c(F(L,t^*))) \geq 1 - \alpha$ then
            $L(a) \leftarrow t^*$ $\{$start $a\}$
        end if
    end while
    update $t^*$ to the next time step where a project finishes
end while
return $L$
```

modify the datasets to fit our uncertainty scenario as follows. For each project we draw a duration between 1 and 10 uniformly at random. The uncertainty is randomized for each project a in two stages; first we draw the *spread* $x \geq 0$ at random; then we set $c_{\min}(a)$ to a uniform random value between $c(a) - x$ and $c(a)$, and $c_{\max}(a) = c_{\min}(a) + x$. We set the probability distribution of the cost to be uniform between $c_{\min}(a)$ and $c_{\max}(a)$. We run 200 samples, where for each sample we randomize new uncertainty to the cost, but the duration as well as the exact cost is always the same. We use different distributions for the spread to model scenarios where the exact cost can be well approximated or not: **High Spread** means a normal distribution with mean $\mu = 0.5 \cdot c(p)$ and variance $\sigma = 0.25 \cdot c(p)$ for each project p. By **Medium Spread** we denote a normal distribution with $\mu = 0.2 \cdot c(p)$ and $\sigma = 0.1 \cdot c(p)$. Finally, by **Low Spread** we mean a normal distribution with $\mu = 0.1 \cdot c(p)$ and $\sigma = 0.05 \cdot c(p)$. Note that spreads have to be natural numbers. So we round the numbers, and in the very unlikely case that a spread is negative, we set it to 0.

We use three metrics to evaluate the algorithms' performance. Given the budgeting log L computed by an online budgeting method, first of all we measure the satisfaction of the voters. However, since online budgeting methods are allowed to exceed the budget according to κ-limitation, for higher κ and α the expected satisfaction can be higher than the satisfaction of the "optimal" bundle (i.e., the offline algorithm). To circumvent this issue we introduce the *satisfaction ratio*

$$sr(L) = \frac{(1-\alpha)s(B_1) + \alpha s(B_2)}{s(R(L))},$$

whereby B_1 is the optimal bundle for a budget limit ℓ and B_2 the optimal bundle for a budget limit $\kappa\ell$. To measure exhaustiveness, we compute how many projects maximally could have been realized in addition to the already realized projects. That is, the size of the largest set $X \subseteq A \setminus R(L)$ of not realized projects with $c_{\max}(X) \leq \ell - c(R(L))$. So, the closer we get to 0, the closer to complete exhaustiveness are we. For measuring punctuality, we simply compute by how much we exceed the time limit.

Algorithm 2 Best Effort Punctuality (BEP)

```
$L = \mathrm{BEE}()$    {Get budgeting log from BEE}
let $t^*$ be the time when the last project in $L$ finishes
$Y \leftarrow \{a \in A \setminus (U(L,t^*) \cup F(L,t^*)) \mid c_{\max}(a) \leq \ell - c(F(L,t^*)) - c_{\max}(U(L,t^*))\}$
while $Y \neq \emptyset$ or $U(L,t^*) \neq \emptyset$ do
    while $Y \neq \emptyset$ do
        let $Y' \subseteq Y$ be the projects $a$ with minimum $\delta(a)$
        let $a \in Y'$ be the project with maximum $c_{exp}(a)$
        remove $a$ from $Y$
            {ensure $\kappa$-LI and $\alpha$-RA:}
        if $c_{\max}(a) \leq \kappa\ell - c(F(L,t^*)) - c_{\max}(U(L,t^*))$ and $c_p(U(L,t^*) \cup \{a\}, \ell - c(F(L,t^*))) \geq 1 - \alpha$ then
            $L(a) \leftarrow t^*$    {start $a$}
        end if
    end while
    update $t^*$ to the next time step where a project finishes
    $Y \leftarrow \{a \in A \setminus (U(L,t^*) \cup F(L,t^*)) \mid c_{\max}(a) \leq \ell - c(F(L,t^*)) - c_{\max}(U(L,t^*))\}$
end while
return $L$
```

Results

The results for the dataset *Warsaw Ursynów, 2021* from the PABULIB are given in Figures 3.1 and 3.2. The budgeting campaign captured by that dataset had a budget of roughly 5 million, and roughly $10,000$ voters approved their preferred projects out of a set of 105 projects, with cost ranging from $2,500$ to roughly 1 million, with an average cost of $232,973.6$. In our experiment the average project duration was 5.83 time units. We provide the results for several other datasets in the appendix.

Comparing the levels of uncertainty, it is not surprising that average values for satisfaction ratio and punctuality worsen with increasing uncertainty. However, for exhaustiveness this is not generally the case: with high uncertainty the exhaustiveness often is better than with medium uncertainty. Probably this is because of the higher maximum costs, which result in fewer projects which can safely be implemented with the remaining budget. Looking at our results in more detail, we see in the left column of Figure 3.1 that the BEE-Algorithm performs much better in terms of both exhaustiveness and satisfaction ratio if we give it more time. So if we have not too tight time limits (note that there are projects that already have a duration of 10), BEE is almost exhaustive in the average case even with medium to high uncertainty, and the satisfaction ratio comes close to 1. In the middle column of Figure 3.1, we see that limitation plays a similar role. Up to some point a higher κ brings BEE closer to exhaustiveness, and increases satisfaction. Yet, increasing κ even further does not seem to have a big impact. This indicates that to some degree tighter time limits can be realized without big quality loss by providing more backup funds, and vice versa. Risk-assessment plays almost no role regarding exhaustiveness and satisfaction unless α is almost 0. This is shown in the right column of Figure 3.1. Although BEE with slightly higher time limit is close to exhaustive, real exhaustiveness is hard to achieve as the high unpunctuality of the BEP-Algorithm shows in Figure 3.2. Recall that an average project has a duration of 5.83. Hence, even for low uncertainty, through all α and κ the time limit is exceeded (in most cases) by a multiple. Higher uncertainty makes this effect even more dramatic. The satisfaction ratio for BEP is by construction of the algorithm at least as high as of BEE

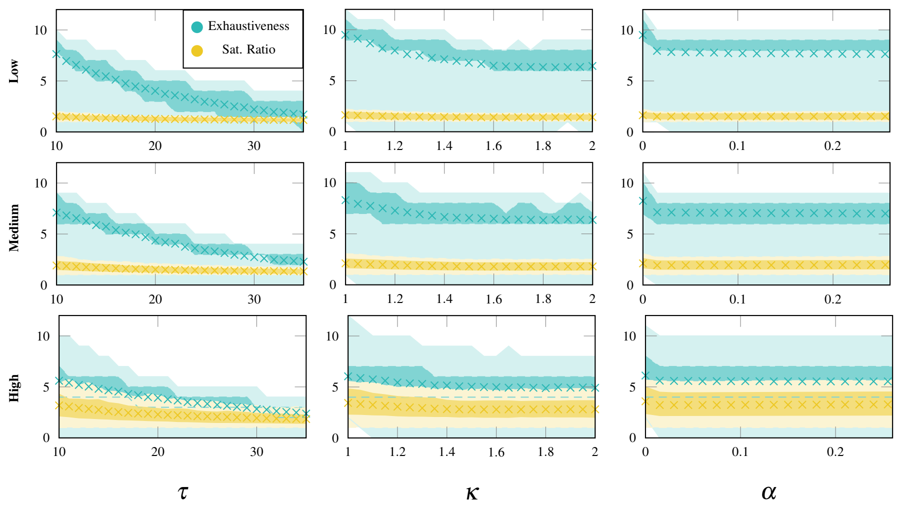


Figure 3.1: Experimental analysis of BEE with dataset Warsaw Ursynów, 2021 and different levels of uncertainty. The cross marks show the average, dark colored areas the upper and lower quartile and light colored areas show the maximum and minimum values over 200 samples of the same dataset with different minimum and maximum costs. **Left:** Fixed parameters $\alpha = 0.05, \kappa = 1.2$ and a variety of time limits. **Middle:** Fixed parameters $\alpha = 0.05, \tau = 10$ and a variety of κ-limitations. **Right:** Fixed parameters $\tau = 10, \kappa = 1.2$ and a variety of α-risk-assessments.

(because BEP calls BEE), and the additional projects increase the satisfaction even further. For a better visibility, we omitted the satisfaction ratio in Figure 3.2.

3.7 Everything Uncertain

Finally, we now want to discuss the problems and questions that arise when both cost, and durations are uncertain. That is, now we are given uncertain cost $\widetilde{c} = (c_{\min}, c_{\max}, c, c_p)$, and uncertain durations $\widetilde{\delta} = (\delta_{min}, \delta_{max}, \delta, \delta_p)$.

First, note that all properties that were incompatible in our uncertain-cost study remain incompatible when additionally durations are uncertain. That is, α-risk assessment, punctuality, and exhaustiveness cannot be simultaneously satisfied by any online budgeting method, and the same holds for κ-limitation, punctuality, and exhaustiveness. This holds even for the relaxation ψ-delay-risk. The proof for Theorem 1 also proves the following result.

Remark 3. *The following incompatibilities hold if both cost and durations are uncertain.*

1. *For any fixed $\alpha < 1$ and $\psi < 1$, no online budgeting method simultaneously satisfies α-risk-assessment, ψ-delay-risk, and exhaustiveness.*
2. *For any fixed $\kappa < m$ and $\psi < 1$, no online budgeting method simultaneously satisfies κ-limitation, ψ-delay-risk, and exhaustiveness.*

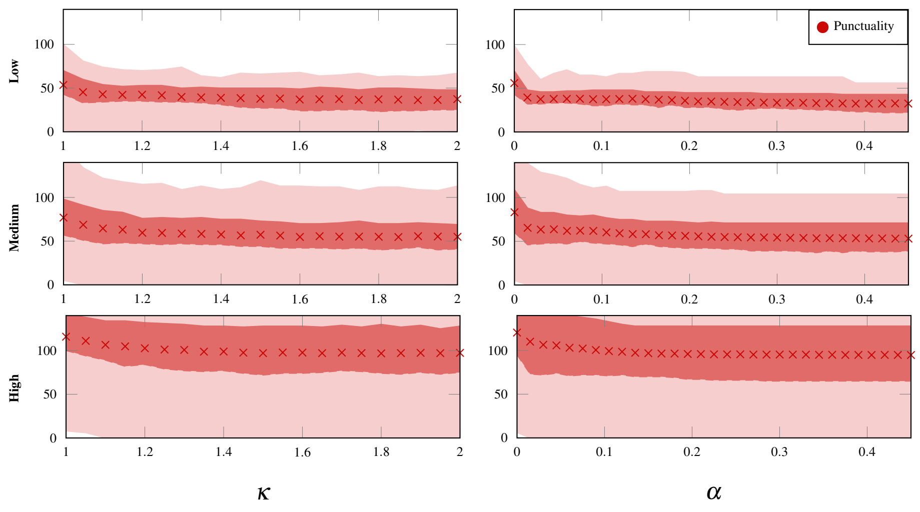


Figure 3.2: Experimental analysis of BEP with dataset Warsaw Ursynów, 2021 and different levels of uncertainty. The cross marks show the average, dark colored areas the upper and lower quartile and light colored areas show the maximum and minimum values over 200 samples of the same dataset with different minimum and maximum costs. **Left:** Fixed parameters $\alpha = 0.05, \tau = 10$ and a variety of κ-limitations. **Right:** Fixed parameters $\tau = 10, \kappa = 1.2$ and a variety of α-risk-assessments.

Interestingly, every two of risk-free, punctuality (and thus also ψ-delay-risk), and exhaustiveness are still possible. Since every project has a maximum duration of τ, the algorithms presented in the proof of Theorem 2 work even when durations are unknown.

Remark 4. *The following algorithms are possible even if cost and durations are uncertain.*

1. *There exists an algorithm which is risk-free, and simultaneously punctual.*
2. *There exists an algorithm which is risk-free, and simultaneously exhaustive.*
3. *There exists an algorithm which is simultaneously punctual, and exhaustive.*

Further, the bounds provided in Theorems 3 and 4 remain valid.

So does the additional uncertainty in the durations change anything at all? Well, not in the worst case analysis. However, it will certainly be a challenge for best-effort algorithms. Let us illustrate the problem with the following example.

Example:

Consider the Projects, cost and time intervals, and approval counts provided in the table below. The budget is $\ell = 60$K, and the time limit $\tau = 18$ months.

Project	Cost	Time	Approval
🪑	15K – 17K	3M – 4M	50
🔌	25K – 31K	4M – 6M	40
🏛️	30K – 53K	12M – 18M	70
🛝	7K – 9K	3M – 4M	15
🛹	30K – 45K	8M – 13M	30
▦	21K – 28K	2M – 3M	40

A simple approach would be to start the museum and the park benches simultaneously accepting the risk of exceeding the budget. But we can improve our risk management by implementing projects sequentially as we learned in Section 3.6. We could start the museum first, and when it finishes we can decide whether to implement the park benches, or the playground, depending on the remaining budget. But note that other than in Section 3.6 we don't know the exact durations of the projects. So this second approach works only if we are lucky, and the museum is implemented in relatively short time. But there is yet another approach: We start the museum. If it finishes within 14 months, and it costs at most 43K, we implement the park benches afterwards. If it finishes within 14 months, and it costs between 44K and 51K, we implement the playground afterwards. And finally, if it doesn't finish within 14 Months, we can either wait another month accepting the risk for exceeding the time limit when starting another project afterwards, but potentially gaining information on the exact price of the museum; or implementing the park benches, or the playground right now accepting the risk for exceeding the budget.

The third approach essentially moves the problems of the first two approaches to the future as we run into the same problem not in the first month, but after month 14. But in many cases the museum will finish earlier. Thus, we will often avoid this problematic decision when using the third approach. So the third approach is certainly superior to the first two. But an online budgeting method needs to be very sophisticated to design such strategies for potentially dozens of projects. We leave the design of such algorithms for future work.

3.8 Conclusions

We introduced a framework for participatory budgeting where knowledge of either the exact cost of projects, or their exact duration, or both is limited. That is, before a project finishes, there is only information on minimum and maximum cost/duration, as well as a probability distribution. The exact cost/duration becomes available only when a project is finished. We further introduced axioms which we think a reasonable budgeting method for this scenario (we call them *online budgeting methods*) should satisfy, and analyzed which axioms (and combinations of them) are satisfiable by such methods. It turns out that while uncertain durations alone are easy to handle, uncertain cost alone is already difficult for an online budgeting method. We established results on what combinations of axioms can, or cannot be achieved. We further developed and experimented with online budgeting methods which achieve a maximal subset of axioms while doing their best to also satisfy the remaining axiom in average case instances (so-called *best-effort budgeting methods*). While the sets of axioms that work together remain the same when additional to the cost uncertainty also durations are uncertain, we conjecture that best-effort budgeting methods for this scenario have to be much more sophisticated. However, this remains as future work yet. Another direction for future research could be to study dependencies between the resources time and cost. For instance, with longer durations, the cost increases (e.g., salaries for workers have to be paid longer). But it could also be the other way around: if one is willing to pay more, it is possible to speed up the project by e.g. employing more workers (see Vaziri et al. [86]).

3.9 Publication

Parts of this chapter (mainly Section 3.6) appear together with further results at the *International Joint Conference on Artificial Intelligence* in 2022.

D. Baumeister, L. Boes, and C. Laußmann. “Time-Constrained Participatory Budgeting Under Uncertain Project Costs”. In: *International Joint Conferences on Artificial Intelligence*. 2022

3.10 Personal Contribution

The axioms in [10] were developed by Dorothea Baumeister, Linus Boes, and me in equal parts. For this chapter I extended the list of axioms for uncertainty in the durations. Theorems 1, 2, 3, and 4 (which are merged from Theorems 3, 4, 6, 7, and 8, and Observation 5 in [10]) were proven by Linus Boes and me in equal parts. The best-effort budgeting methods and experiments (which were also briefly introduced in [10]) were developed, executed, and analyzed by me. The remainder of this chapter (particularly all parts about uncertain durations) was developed by me, and is not included in [10]. The writing of the publication [10] was conducted in equal parts by Dorothea Baumeister, Linus Boes, and me. I reused a few text passages from [10] in this chapter which were not completely written by me.

Chapter 4

Strategic Campaigns in Apportionment

4.1 Summary

Probably the largest elections among humans are parliamentary elections. Voters elect representatives into a parliament, who then discuss problems and decide for solutions. One widely used way of electing a parliament is through *party-list proportional representation apportionment methods* (or *apportionment* for short). Here, voters vote for parties instead of directly for candidates, and the seats in the parliament are apportioned to the parties according to their share of votes. Often, there is also a so-called *legal electoral threshold*, i.e., each party must receive a minimum share of the total votes in order to gain seats at all.

Strategic campaigns are attempts to convince specific voters to change their vote for another party. We model strategic campaigns as BRIBERY problems. In this chapter, we show that most apportionment methods are vulnerable to BRIBERY regardless of whether a threshold is imposed or not. These are bad news, since a malicious person can easily compute optimal campaigns. However, we will also show that by slightly modifying the apportionment process, we can increase the complexity of BRIBERY significantly, which provides more resistance to strategic campaigns. Meanwhile, it is not much more effort for the voters to vote in this modified setting, and computation of the seat allocation remains easy.

4.2 Introduction

When it comes to parliamentary elections, voters usually don't vote on individual candidates but on parties. Among these parties the fixed number of seats in the parliament is apportioned. An *apportionment method* is an algorithm that apportions the seats in a parliament to the parties. Often the goal is to apportion the seats in a way that the fraction of seats in the parliament a party gets is as close as possible to the fraction of voters who voted for that party. This ensures that the parliament is representative for the voters' opinions, can then efficiently discuss topics, and decide laws in the name of the voters. Pukelsheim [71] provides an overview of apportionment methods in European countries and describes the process based on real election data. Gallagher and Mitchell [41] give another very detailed explanation of voting rules used world-wide.

Many countries using apportionment extend the basic procedure by a so-called *electoral threshold*—a minimum fraction of votes a party must receive to get any seats at all. In Germany, for instance, a party must receive at least 5% of the total votes before it gets any seats at all (even though 'mathematically' each of the 736 seats represents about 0.13% of the voters). By reducing the number of parties in parliament (see [68] for a study on mechanical and psychological effects), thresholds are important for the government to quickly form, and for the parliament to allow efficient decision-making. But a disadvantage of the threshold is that potentially many voters are not represented in the parliament because they were supporting a party that did not make it above the threshold. However, we don't want to discuss these advantages and disadvantages of thresholds in this work. We want to find out to what extent thresholds can be exploited in *strategic campaigns*.

In strategic campaigns, an external agent intents to change the election outcome in his favor by convincing (e.g. by advertisement) a limited number of voters to change their vote. That is, an external agent seeks to change a limited number of votes in order to either ensure a party he supports receives at least ℓ seats in the parliament (constructive case), or to limit the influence of a party he despises by ensuring it receives no more than ℓ seats (destructive case). In literature, this problem is usually referred to as BRIBERY. The research line on the computational complexity of BRIBERY (which is also referred to as *strategic campaigns*), was initiated by Faliszewski et al. [33]. The complexity of bribery has been studied for a wide range of voting rules, as surveyed by Faliszewski and Rothe [35]. Most of the time bribery was only studied for single-winner voting rules. In this chapter we study the complexity of BRIBERY for apportionment methods with threshold. Bredereck et al. [23] only recently initiated the study of bribery in apportionment elections. They show that an optimal strategic campaign for apportionment elections *without* a threshold can be computed in polynomial time. A similar result was shown by Güney [46]. Their studies are closely related to the first part of this chapter, and the algorithms we provide in Section 4.5 are based on algorithms Bredereck et al. [23] developed. Especially in the light of today's possibilities to process enormous amounts of data from e.g. social networks (which allow to predict the voting behavior of individuals and to target them with individualized advertising), it becomes very important to understand how hard it is to target elections with strategic campaigns, i.e. to know how effective campaigns can be and how easy it is to find optimal campaigns.

4.3 Apportionment

An apportionment instance $I = (\mathcal{P}, \mathcal{V}, \tau, \kappa)$ consists of the set of *m parties* $\mathcal{P}$, a list of *n votes* $\mathcal{V}$ over the parties in $\mathcal{P}$, a threshold $\tau \in \mathbb{N}$ and the seat count $\kappa \in \mathbb{N}$. We assume κ is smaller than $|\mathcal{V}|$. In reality, κ will be orders of magnitude smaller than $|\mathcal{V}|$ since a parliament is supposed to be a small representation of the citizens. Each vote in $\mathcal{V}$ is a strict ranking of the parties from most to least preferred, and we write $A \succ_v B$ if voter v prefers party A to B. We omit the subscript v when it is clear from the context. We refer to the most preferred party of a voter v as v's *top choice*. Relative thresholds (as they occur much more often in reality, e.g. the 5% in Germany) can easily be converted to an absolute threshold, as in our definition.[1]

Next, let us look at how the apportionment instance is processed to apportion the seats to the parties. Note that in order to respect the threshold, it must hold that whenever less than τ voters have a party P as their top choice, P must receive 0 seats in the end. Since we want to keep the processing of the threshold separated from the actual apportionment procedures as they are described in literature, we process apportionment instances in two consecutive steps as illustrated by the figure below.

$$\begin{tikzcd}
I = (\mathcal{P},\mathcal{V},\tau,\kappa) \arrow[r, "\text{Top-Choice or Second-Chance}"'] & \sigma \arrow[r, "\text{Apportionment Method}"'] & \alpha \\
\text{Apportionment Instance} & \text{Support Allocation} & \text{Seat Allocation}
\end{tikzcd}$$

In the first step we compute a *support allocation* σ. The support allocation informally describes how many voters count as supporters for each party. In the second step, the seats are apportioned to the parties according to their support. The electoral threshold will be applied in the first step. We will introduce two methods how to deal with the threshold in a second. But it is important to note at this point that we require $\sigma(P) = 0$ whenever less than τ voters have a party P as their top choice. By this requirement we ensure that no changes (compared to literature) to the actual apportionment procedures used in the second step are needed. They work *out of the box* regardless what threshold is used, and how we deal with it.

4.3.1 Computing Support Allocations

In this work we consider two methods (which we refer to as *modes*) of computing the support allocation. Both result in different support allocations if and only if at least one party is below the electoral threshold. This is because of the different ways the two modes deal with votes for parties below the threshold. By our requirement, it is clear that—in both modes—holds $\sigma(P) = 0$ whenever less than τ voters have a party P as their top choice. However, it is not defined what to do with the votes which have P as their top choice. In most real elections these votes are currently dropped. That is, the support for each party P is exactly the count of votes where P is the top choice—except when this count is less than τ; then the support for P is 0. We refer to this mode as the *top-choice mode*, as only top-choices are considered supporters. We illustrate the procedure with the following example.

[1] Since strategic campaigns as we define them later do only change the votes but never the vote count, we can use relative and absolute thresholds interchangeably, indeed. Note however, that this doesn't hold for fraud types which change the vote count (e.g. CONTROL).

Example:

Consider $\tau = 4$, $\mathcal{P} = \{A,B,C,D,E\}$, any κ and

$$\begin{aligned}\mathcal{V} = (3 \quad &\times \quad A \succ B \succ C \succ E \succ D,\\ 2 \quad &\times \quad C \succ E \succ B \succ A \succ D,\\ 8 \quad &\times \quad D \succ C \succ E \succ B \succ A,\\ 6 \quad &\times \quad B \succ A \succ C \succ E \succ D).\end{aligned}$$

Clearly A, C and E don't make it above the threshold. In the top-choice mode votes for them (i.e., the first two blocks) are ignored. Thus, the final support allocation is $\sigma(A) = \sigma(C) = \sigma(E) = 0$, $\sigma(B) = 6$, and $\sigma(D) = 8$.

When we consider the example above once again, it is easy to argue that while D has the most support, party B seems to be more liked by the most voters. Knowing that A and C don't make it above the threshold, the first two voter groups might regret that they haven't voted for B instead. Both find D to be the worst party while B is their second or third most preferred party, and B made it above the threshold. Inspired by this observation, we introduce the *second-chance mode*. In the second-chance mode we first find the set $\mathcal{P}_{\geq\tau}$ of parties which make it above the threshold (i.e., where at least τ voters have the respective party as top choice). Then, we let each voter count as one supporter for the party in $\mathcal{P}_{\geq\tau}$ they prefer the most. In the example above we have $\mathcal{P}_{\geq\tau} = \{B,D\}$. The first two voter groups now count as supporters for B as this is their most preferred party in $\mathcal{P}_{\geq\tau}$. The third group counts for D, and the last group for B, just as before. This results in the support allocation $\sigma(A) = \sigma(C) = \sigma(E) = 0$, $\sigma(B) = 11$, and $\sigma(D) = 8$.

4.3.2 Apportioning Seats

Now that we have computed the support allocation either way, we can determine the seat allocation by employing a so-called *apportionment method*. Such methods take the support allocation σ and the seat count κ as input, and compute the seat allocation $\alpha : \mathcal{P} \to \{0,\ldots,\kappa\}$ satisfying $\sum_{A\in\mathcal{P}} \alpha(A) = \kappa$. We focus on the *largest-remainder method (LRM)* and the class of *divisor sequence apportionment methods*, including, for example, the D'Hondt method (a.k.a. Jefferson's method), and the Sainte-Laguë method (a.k.a. the Webster method).

A *divisor sequence method* is characterized by a strict monotonic increasing function $f : \mathbb{N}_{\geq 1} \to \mathbb{N}_{\geq 1}$. Thereby $f(i)$ is the i-th divisor. D'Hondt, for instance, is characterized by the function $f_{\mathrm{DH}}(x) = x$, and Sainte Laguë is characterized by $f_{\mathrm{SL}}(x) = 2x - 1$. For each party $P \in \mathcal{P}$, we then compute the list

$$\left[\frac{\sigma(P)}{f(1)}, \frac{\sigma(P)}{f(2)}, \ldots, \frac{\sigma(P)}{f(\kappa)}\right]$$

using these divisors. Then, we go through the lists of all parties to find the highest κ values. Ties are broken by some tie-breaking mechanism. Each party receives one seat for each of its list values that is among the κ highest values. Let us illustrate the process in the following example for Sainte Laguë.

Example:

Suppose we apportion $\kappa = 4$ seats to the parties $\mathcal{P} = \{P_1, P_2, P_3\}$, and are given the following support allocation: $\sigma(P_1) = 1024$, $\sigma(P_2) = 817$, and $\sigma(P_3) = 610$. We compute the following lists (values are rounded to one decimal place):

$$\begin{array}{lllll} P_1 : & [\mathbf{1024}, & \mathbf{341.3}, & 204.8, & 146.3], \\ P_2 : & [\ \mathbf{817}, & 272.3, & 163.4, & 116.7], \\ P_3 : & [\ \mathbf{610}, & 203.3, & 122, & 87.1]. \end{array}$$

We highlight the $\kappa = 4$ highest values in boldface. Party P_1 receives two seats, parties P_2 and P_3 receive one seat each.

The *Largest-Remainder Method (LRM)* is not based on a function computing divisors. It first computes the total support $n' = \sum_{P \in \mathcal{P}} \sigma(P)$ (which is exactly the number of voters except when votes are ignored in the top-choice mode because a party is below the threshold). We then compute the fair share $\text{fs}(P_i) = \kappa \cdot \frac{\sigma(P_i)}{n'}$ which, informally, is the number of seats the party would receive in a perfectly proportional parliament. Since this number is usually not an integer, we now have to round the fair shares of the parties. First, each party $P_i \in \mathcal{P}$ receives a *lower quota* of $\text{lq}(P_i) = \lfloor \text{fs}(P_i) \rfloor$ seats, i.e., the integer part of $\text{fs}(P_i)$. Usually we now have some seats left over. Say we have s seats left, then each of the s parties with the largest *remainder* $\text{rem}(P_i) = \text{frac}(\text{fs}(P_i))$ (where $\text{frac}(x) = x - \lfloor x \rfloor$ denotes the non-integer part of a real number x) receive one additional seat (i.e., their fair share is rounded up).

In Example 4.3.2 LRM would work as follows. First we compute $n' = 1024 + 817 + 610 = 2451$. We now compute the fair shares $\text{fs}(P_1) = \kappa \cdot \frac{1024}{2451} \approx 1.67$, $\text{fs}(P_2) = \kappa \cdot \frac{817}{2451} \approx 1.33$, and $\text{fs}(P_3) = \kappa \cdot \frac{610}{2451} \approx 0.99$. Thus, P_1 and P_2 receive a lower quota of 1 seat, and P_3 no seat for now. However, 2 seats are left, so P_3 and P_1 with their largest remainders receive one additional seat each.

4.4 Bribery

By strategic campaigns we mean that an external agent strategically convinces a limited number of voters in such a way that the election outcome is changed in a by the agent desired way. We model strategic campaigns as the following BRIBERY decision problem (compare Bredereck et al. [23] and Faliszewski et al. [33]).

$\mathcal{R}$-BRIBERY	
Given:	An apportionment instance $(\mathcal{P}, \mathcal{V}, \tau, \kappa)$, a distinguished party $P^* \in \mathcal{P}$, and integers ℓ, $1 \leq \ell \leq \kappa$, and K, $0 \leq K \leq \lvert\mathcal{V}\rvert$.
Question:	Is there a successful campaign, that is, is it possible to make P^* receive at least ℓ seats using apportionment method $\mathcal{R}$ by changing at most K votes?

We also consider a destructive version of the problem where the external agent wants to limit the influence of a distinguished party.

$\mathcal{R}$-DESTRUCTIVE-BRIBERY	
Given:	An apportionment instance $(\mathcal{P}, \mathcal{V}, \tau, \kappa)$, a distinguished party $P^* \in \mathcal{P}$, and integers ℓ, $0 \leq \ell < \kappa$, and K, $0 \leq K \leq \lvert\mathcal{V}\rvert$.
Question:	Is there a successful campaign, that is, is it possible to make P^* receive at most ℓ seats using apportionment method $\mathcal{R}$ by changing at most K votes?

In both the constructive and the destructive cases, we assume tie-breaking in the apportionment methods to be to the advantage of P^* (i.e., in a tie, P^* gets the seat). Note that the encoding of τ, ℓ, and κ does not matter since they are bounded by $\lvert\mathcal{V}\rvert$.

The analysis of the complexity of the problems above is important to understand how realistic it is that an external agent significantly changes the election outcome by individualized advertisement or direct payments to the voters. Such actions are usually expensive, so we can assume that the agent has only capabilities to change a very limited amount of votes. Thus, the agent has to be certain that his goal can be achieved, and he has to be careful how to spend the budget for maximal effect. Note that when the problems above are intractable for an apportionment method $\mathcal{R}$, computing an optimal strategic campaign is intractable, too.

4.5 Classical Top-Choice Mode

We start with a complexity analysis of $\mathcal{R}$-BRIBERY and $\mathcal{R}$-DESTRUCTIVE-BRIBERY in the classical top-choice mode of apportionment as defined in Section 4.3.1. Before we present our main result regarding the top-choice mode in Theorem 1, let us first present the following lemma and understand its implications.

Lemma 1. *The following two statements hold for all divisor sequence methods and for the Largest Remainder Method.*

1. *Adding voters to support a distinguished party P^* can never make P^* lose seats. Further, removing voters from P^* can never make P^* gain seats.*

2. *Adding voters to support another party than P^* cannot increase the number of seats for P^* by more than if we add the same number of supporters to P^* directly. Conversely, removing supporters from another party than P^* cannot decrease the number of seats for P^* by more than if we remove the same number of supporters directly from P^*.*

Proof. Le us first prove the lemma for **divisor sequence methods**.

The first claim is easy to see. Increasing (decreasing) the support for P^* increases (decreases) all values in the list of P^* while keeping the values of the other parties untouched. Thus, when a value in the list of P^* was one of the κ highest seats before, it still is (in the destructive case, when a value wasn't one of the highest, it remains so).

For the second claim, note that changing the vote counts for other parties than P^* results in no changes of the list values of P^* but may increase the list values of the other parties. Thus, the other parties cannot lose seats at all, and P^* cannot get more seats than before. On the other hand, adding the same vote counts to P^* can (by the first claim) never make P^* lose seats. The case of removing voters is treated analogously.

The proof for **LRM** is a little more involved since adding or removing votes changes n and thus the fair shares of all parties.

For the first claim, note that $\kappa \cdot \frac{\sigma(P^*)-x}{n'-x} < \kappa \cdot \frac{\sigma(P^*)}{n'} < \kappa \cdot \frac{\sigma(P^*)+x}{n'+x}$ for every positive x. That is, the fair share decreases when we remove supporters, but increases when we add supporters. However, for other parties the fair share decreases when n' increases, and vice versa. Thus, P^* receives at least as many seats as before when supporters for P^* are added, and cannot gain seats when supporters are removed.

For the second claim, note it is in fact possible (although rare) that P^* receives more seats than before by adding a voter to *another* party than P^* due to the changed remainders with the changed value of n. Party P^*'s remainder may be decreased a little less than the remainder of a much bigger party. However, note that we would achieve the same n by adding the same voters to P^* instead of to the other party. In that case, P^*'s fair share is even higher, so P^* will not receive fewer seats than before. An analogous argumentation applies to the converse case since removing voters from P^* reduces P^*'s seat count at least as much as removing voters from another party. □

Lemma 1 is so important because it guarantees that in the constructive case it is optimal to bribe K voters of other parties to vote for P^* instead, and in the destructive case it is optimal to take K voters away from P^* and bribe them to vote for another party instead. Armed with this knowledge, we can now prove our main result of the section.

Theorem 1. *Let $\mathcal{R}$ be a divisor sequence method or the Largest Remainder Method. Then $\mathcal{R}$-*BRIBERY *and $\mathcal{R}$-*DESTRUCTIVE-BRIBERY *are in P.*

Let us start with the proof for divisor sequence methods.

Proof (Divisor Sequence Methods). We prove that Algorithm 3 decides the constructive case correctly in deterministic polynomial-time, and Algorithm 4 does for the destructive case.

Algorithm 3 Deciding whether bribery is possible (divisor sequence methods).

```
Input: $\mathcal{P}$, $\mathcal{V}$, $\tau$, $\kappa$, $P^*$, $K$, $\ell$
  $K \leftarrow \min\{n - \sigma(P^*), K\}$
  if $\sigma(P^*) + K < \tau$ then
      return No
  end if
  compute $\gamma$ {$\gamma[P][x]$ is the minimum vote count we must remove from P to ensure that P receives exactly x seats before $P^*$ gets $\ell$ seats assuming $P^*$ gets exactly K additional votes.}
  initialize table tab with $\kappa - \ell$ columns and $m$ rows,
     where tab[0][0] $\leftarrow 0$ and the other entries are $\infty$.
  let $o : \{1, \dots, |\mathcal{P}_{-P^*}|\} \to \mathcal{P}_{-P^*}$ be an ordering

  for $i \leftarrow 1$ to $|\mathcal{P}_{-P^*}|$ do
      for $s \leftarrow 0$ to $\kappa - \ell$ do
          for $(x, \text{cost}) \in \gamma[o(i)]$ do
              if $s - x \geq 0$ then
                  $tmp \leftarrow$ tab[$i-1$][$s-x$] + cost
                  if $tmp <$ tab[$i$][$s$] then
                      tab[$i$][$s$] $= tmp$
                  end if
              end if
          end for
      end for
  end for

  for $s \leftarrow 0$ to $\kappa - \ell$ do
      if tab[$|\mathcal{P}_{-P^*}|$][$s$] $\leq K$ then
          return Yes
      end if
  end for
  return No
```

We begin with the **constructive** case. Algorithm 3 runs in polynomial time. This is easy to see once we describe how γ is computed, since the rest of the algorithm consists of loops which obviously run in polynomial-time. As commented, $\gamma[P][x]$ gives the minimum number of votes that have to be removed from party P so that P receives only x seats before P^* receives the ℓ-th seat (assuming P^* has exactly K additional votes in the end—which is optimal according to Lemma 1). Computing the γ values works with a binary search for the jumping points of the function ϕ, which is defined as the number of seats a party with y votes receives before P^* receives ℓ seats (again, assuming P^* has exactly K additional votes in the end). That is, assuming P^* has p^* votes in the end, we have $\phi(y) = 0$ if $y < \tau$ or $\frac{y}{f_D(1)} \leq \frac{p^*}{f_D(\ell)}$, and $\phi(y) = \max\{k \mid \frac{y}{f_D(k)} > \frac{p^*}{f_D(\ell)}\}$ otherwise, where f_D is the characteristic function of the respective divisor sequence method.

We now show the correctness of the algorithm. Setting K to the minimum of $n - \sigma(P^*)$ and K is necessary to ensure that we never remove more voters from parties in $\mathcal{P}_{-P^*}$ than allowed or exist. Next, if P^* cannot reach the threshold, we must answer No since P^* can then never receive any seat at all. In the middle part of the algorithm, we fill a table. For each i, $1 \leq i \leq |\mathcal{P}_{-P^*}|$, and each s, $0 \leq s \leq \kappa - \ell$, the cell tab[$i$][$s$] contains the minimum number of votes needed to be moved away from parties $P_1, \dots, P_i$ such that $P_1, \dots, P_i$ receive s seats in total before P^* is assigned its ℓ-th seat (again, assuming P^* has exactly K additional votes in the end). The values are computed dynamically from the previous row to the next row. This is possible because the seats that parties $P_1, \dots, P_i$ receive in total before P^* is assigned its ℓ-th seat are exactly the sum of the number of seats the parties receive individually before P^* receives its ℓ-th seat. Further, since this number can be computed directly by comparing the divisor list of the party with the divisor list of P^* (i.e., the ϕ function of each party is independent of other parties' support) the required bribery budget is also exactly the sum of the individual bribes.

Finally, in the end, we check if in the last row there exists a value of at most K. If this holds for, say, cell $\text{tab}[|\mathcal{P}_{-P^*}|][s]$, then we correctly answer YES because there do exist bribes that do not exceed K and ensure that the other parties receive at most s seats before P^* is assigned its ℓ-th seat. Since there are κ seats available, and the other parties get $s \leq \kappa - \ell$ seats before P^* receives the ℓ-th seat, P^* will indeed receive its ℓ-th seat. However, if all cells of the last row contain a value greater than K, the given budget does not suffice to ensure that the other parties receive at most $\kappa - \ell$ seats before P^* receives its ℓ-th seat. Thus, the other parties receive at least $\kappa - \ell + 1$ seats in this case, which leaves at most $\ell - 1$ seats for P^*, so we correctly answer NO.

Algorithm 4 Deciding whether destructive bribery is possible (divisor sequence methods).

```
Input: $\mathcal{P}$, $\mathcal{V}$, $\tau$, $\kappa$, $P^*$, $K$, $\ell$
  $K \leftarrow \min\{\sigma(P^*), K\}$
  if $\sigma(P^*) - K < \tau$ then
      return YES
  end if
  compute $\gamma$ {$\gamma[P][x]$ is the minimum vote count we need to add to P to ensure that P receives exactly x seats before $P^*$ gets $\ell + 1$ seats assuming $P^*$ loses exactly K votes.}
  initialize table tab with $\kappa - \ell$ columns and $m$ rows,
     where tab[0][0] $\leftarrow$ 0 and the other entries are $\infty$.
  let $o : \{1, \dots, |\mathcal{P}_{-P^*}|\} \to \mathcal{P}_{-P^*}$ be an ordering

  for $i \leftarrow 1$ to $|\mathcal{P}_{-P^*}|$ do
      for $s \leftarrow 0$ to $\kappa - \ell$ do
          for $(x, \text{cost}) \in \gamma[o(i)]$ do
              if $s - x \geq 0$ then
                  $tmp \leftarrow \text{tab}[i-1][s-x] + \text{cost}$
                  if $tmp < \text{tab}[i][s]$ then
                      $\text{tab}[i][s] = tmp$
                  end if
              end if
          end for
      end for
      for $s \leftarrow 0$ to $\kappa - \ell$ do
          if $\text{tab}[i-1][s] \leq K$ and $s + \phi(\sigma(o(i)) + K - \text{tab}[i-1][s]) \geq \kappa - \ell$ then
              return YES
          end if
      end for
  end for
  return NO
```

Now we turn towards the **destructive** case. It is easy to see that Algorithm 4 runs is polynomial-time just as Algorithm 3. This time, γ and ϕ are defined slightly different: We define $\phi(y)$ as the number of seats a party with y votes receives before P^* is assigned its $(\ell + 1)$-th seat (assuming P^* loses K votes which is optimal by Lemma 1). Further, we define $\gamma[P][x]$ as the minimum number of votes we need to add to party P such that it receives at least x seats before P^* is assigned its $(\ell+1)$-th seat. We can compute γ in polynomial-time using binary search for the jumping points in ϕ (which itself can be computed similarly as in the constructive case).

We now show the correctness of the algorithm. Setting K to the minimum of $\sigma(P^*)$ and K is necessary to ensure that we never remove more voters from P^* than allowed or exist. Next, if P^* will not reach the threshold after removing K supporters, we can immediately answer YES since P^* doesn't receive any seat at all. In the middle part of the algorithm, we fill a table. For each i, $1 \leq i \leq |\mathcal{P}_{-P^*}|$, and each s, $0 \leq s \leq \kappa - \ell$, the cell $\text{tab}[i][s]$ contains the minimum number of votes needed to be moved towards parties $P_1, \dots, P_i$ such that $P_1, \dots, P_i$ receive s seats in total before P^* is assigned its $(\ell+1)$-th seat (assuming P^* loses exactly K votes in the end). The values are computed dynamically from the previous row to the next row which is possible by the same argument as in the constructive case. At the end of each iteration we check if there is a possibility with the remaining budget to gain at least $\kappa - \ell$ seats for $\mathcal{P}_{-P^*}$ before P^* gains the $(\ell+1)$-th seat. If so, we answer YES since we can add at most K voters to the other parties such that not enough seats are left for P^* to get the $(\ell+1)$-th seat. If this was possible in no iteration, we must answer NO since with our budget it was never possible for $\mathcal{P}_{-P^*}$ to take enough seats away from P^*. □

Let us now show the proof of Theorem 1 for the Largest Remainder method.

Proof (Largest Remainder Method). For LRM we can use very similar algorithms as for divisor sequence methods. However, a few changes in the design are necessary. First, since LRM depends on the total support count (i.e., number of voters for parties above the threshold), we have to pay attention when pushing a party below the threshold (or raising a party above the threshold). This will change the total support count, and thus all fair shares. Therefore, we will basically run the algorithm once where we push no party below the threshold, once where we push a total of one party below the threshold, once where we push a total of two parties below the threshold, and so on (analogously with bringing parties above the threshold in the destructive case). The second major difference to the algorithm for divisor sequences is that a table needs only to be filled to check whether P^* can receive a remainder seat. The lower quota for P^* is fixed by the total support count already. We thus fill the table if and only if the remainder seat matters (either because P^* needs it in the constructive case, or because P^* must not get it in the destructive case).

We begin with the detailed proof for the **constructive** case. Algorithm 5 runs in polynomial time. This is easy to see once we describe how γ is computed, since the rest of the algorithm consists of loops which obviously run in polynomial-time (note that the while-loop runs only until no further party can be pushed below the threshold, i.e., at most $|\mathcal{P}| - 1$ times). As commented, $\gamma[P][x]$ gives the minimum number of votes that have to be removed from party P so that P receives only x seats before P^* receives its remainder seat (assuming P^* has exactly K additional votes in the end). Thereby, γ excludes all possibilities to push P below the threshold, i.e., for $x = 0$ is undefined. This is important to let us control how many parties are above the threshold. Computing the γ values works with a binary search for the jumping

Algorithm 5 Deciding whether bribery is possible (LRM).

```
Input: $\mathcal{P}$, $\mathcal{V}$, $\tau$, $\kappa$, $P^*$, $K$, $\ell$
  $K \leftarrow \min\{n - \sigma(P^*), K\}$
  if $\sigma(P^*) + K < \tau$ then
      return NO
  end if
  while True do
      update $n$ to match the number of voters for parties still above the threshold (budget expended already counts for $P^*$)
      if $P^*$ receives at least $\ell$ seats by lower quota then
          return YES
      end if
      if $P^*$ receives exactly $\ell - 1$ seats by lower quota then
          compute $\gamma$ {$\gamma[P][x]$ is the minimum vote count we must remove from P to ensure that P receives only x seats before $P^*$ gets the remainder seat.}
          initialize table tab with $\kappa - \ell$ columns and $m$ rows,
             where tab[0][0] $\leftarrow$ 0 and the other entries are $\infty$.
          let $o : \{1, \dots, |\mathcal{P}_{-P^*}|\} \to \mathcal{P}_{-P^*}$ be an ordering
          for $i \leftarrow 1$ to $|\mathcal{P}_{-P^*}|$ do
              for $s \leftarrow 0$ to $\kappa - \ell$ do
                  for $(x, \text{cost}) \in \gamma[o(i)]$ do
                      if $s - x \geq 0$ then
                          $tmp \leftarrow \text{tab}[i-1][s-x] + \text{cost}$
                          if $tmp < \text{tab}[i][s]$ then
                              $\text{tab}[i][s] = tmp$
                          end if
                      end if
                  end for
              end for
          end for
          for $s \leftarrow 0$ to $\kappa - \ell$ do
              if $\text{tab}[|\mathcal{P}_{-P^*}|][s] \leq K$ then
                  return YES
              end if
          end for
      end if
      if there is a party which can be pushed below the threshold with the remaining budget then
          push the smallest such party below the threshold, and update the budget accordingly
      else
          return NO
      end if
  end while
```

points of the function ϕ, which is defined as the number of seats a party with y votes receives before P^* receives its remainder seat (again, assuming P^* has exactly K additional votes in the end). That is, $\phi(y)$ is the lower quota of a party with y support whenever the remainder of a party with y support is lower than the remainder of P^*, or else the lower quota plus one.

We now show the correctness of the algorithm. Setting K to the minimum of $n - \sigma(P^*)$ and K is necessary to ensure that we never remove more voters from parties in $\mathcal{P}_{-P^*}$ than allowed or exist. Next, if P^* cannot reach the threshold, we must answer NO since P^* can then never receive any seat at all. As mentioned, we now repeat the following first with no party pushed below the threshold, and then again pushing one additional party (the smallest) at a time below the threshold until we cannot push any further parties below the threshold. This ensures that we cover all possible number of parties above the threshold, and thus all possible values for n (the total support). For each iteration we update n accordingly to ensure the fair shares of all parties can be computed correctly. In case P^* receives at least ℓ seats already by lower quota, we can safely answer YES because with the current n the distinguished party is already guaranteed its ℓ-th seat. Only if P^* receives exactly $\ell - 1$ seats by lower quota, we start to fill a table as we did with the divisor sequence methods. The proof of correctness for this part of the algorithm works analogously as for the divisor sequence methods, and is therefore omitted. If the last table row contains a value of at most K, we know that it is possible to take voters away from the other parties such that they receive at most $\kappa - \ell$ seats before P^* gets a remainder seat. So P^* will receive the remainder seat which results in ℓ seats for P^* in total. However, if all cells of the last row contain a value greater than K, we must try to push an additional party below the threshold. If this is possible, we do so, and repeat the while-loop. Otherwise, we can answer NO because with all possible n the distinguished party was unable to receive ℓ seats.

For the **destructive** case Algorithm 5 can be adapted in the same way as Algorithm 3 was adapted for the destructive case for divisor sequence methods. This time within the while-loop we answer YES if P^* receives at most $\ell - 1$ seats by lower quota, and we fill the table if and only if P^* receives exactly ℓ seats by lower quota (here we try to prevent P^* from receiving the remainder seat). □

By tracing back through the tables one can easily extract the information from which party how many votes have to be removed (constructive case), or to which party how many votes have to be added (destructive case). Note, however, that this number might not add up to K but can be smaller. In bribery, we assume votes to be moved but neither added nor deleted. Since we assume in the algorithms (according to Lemma 1) that exactly K votes are added for P^* (constructive), or removed from P^* (destructive), we might need to remove (constructive) or add (destructive) some additional votes for some random party other than P^*. This way we ensure that the number of votes added to P^* matches the number removed from other parties in the constructive case, and vice versa in the destructive case. Note further that computing optimal campaigns can also be done efficiently by binary search for the highest (smallest) value for ℓ where constructive (destructive) bribery is still possible.

4.6 The Second-Chance Mode

Efficient algorithms as developed in the previous section make it very simple for a campaign manager to exert influence on the election outcome. By applying the algorithms to real election data, one can observe that a very small budget is sufficient to change the election outcome seriously. The interested reader is encouraged to take a look at the experimental results in [59]. This being said, it can be considered quite dangerous for a democracy when optimal campaigns can be computed efficiently. Therefore, it would be of great advantage if there was some modification to the usual apportionment setting that makes the computation of optimal campaigns intractable.

In this section we analyze the *second-chance mode* of voting in apportionment elections which turns out to provide computational resistance against strategic campaigns. Instead of just focusing on specific apportionment methods, we generalize our results to the class of majority-consistent apportionment methods.

Definition 1. *Let $A \in \mathcal{P}$ be the party with the highest support (with arbitrary tie-breaking if necessary) in a given support allocation σ. We call an apportionment method* majority-consistent *if no party in $\mathcal{P}$ with less support than A receives more seats than A.*

Informally, a majority-consistent apportionment methods assigns the party with highest support the most seats. This is certainly a reasonable criterion for apportionment methods, and all prominent apportionment methods (including divisor sequence methods and LRM) satisfy it. For this huge class of apportionment methods we can show the following result.

Theorem 2. *For each majority-consistent apportionment method $\mathcal{R}$, $\mathcal{R}$-*BRIBERY *and $\mathcal{R}$-*DESTRUCTIVE-BRIBERY *are NP-complete in the second-chance mode.*

Proof. The membership to NP is easy to see for both problems: given the list of votes after applying bribery (the certificate) we can easily verify that no more than K votes were changed, and that now P^* receives at least (or at most) ℓ seats. We show NP-hardness.

Let us start with the proof for $\mathcal{R}$-BRIBERY. Let $(U,S,K) = (\{u_1,\dots,u_p\},\{S_1,\dots,S_q\},K)$ be an instance of HITTING SET with $q \geq 4$. In polynomial time, we construct an instance of $\mathcal{R}$-BRIBERY with parties $\mathcal{P} = \{c,c'\} \cup U$, a threshold $\tau = 2q+1$, $\ell = 1$ desired seat, $\kappa = 1$ available seat, and the votes

$$\begin{aligned}
\mathcal{V} = (4q+2 \text{ votes} \quad & c \succ \cdots, \\
4q+K+2 \text{ votes} \quad & c' \succ \cdots, && (4.1)\\
\text{for each } j \in [q], 2 \text{ votes} \quad & S_j \succ c' \succ \cdots, && (4.2)\\
\text{for each } i \in [p], q-\gamma_i \text{ votes} \quad & u_i \succ c \succ \cdots, && (4.3)\\
\text{for each } i \in [p], q-\gamma_i \text{ votes} \quad & u_i \succ c' \succ \cdots), && (4.4)
\end{aligned}$$

where $S_j \succ c'$ means that each element in S_j is preferred to c', but we do not care about the exact order of the elements in S_j. Further, $2\gamma_i$ is the number of votes from group (4.2), in which u_i is at the first position. That is, it is guaranteed that each party u_i is in exactly $2\gamma_i + (q-\gamma_i) + (q-\gamma_i) = 2q < \tau$ votes at first position—thus failing the threshold. Party c is first in the preference of $4q+2 \geq \tau$ voters, and c' in the preference of $4q+K+2 \geq \tau$ voters. Note that the voters in groups (4.2) and (4.4) use their second chance to vote for c', and those in group (4.3) use it to vote for c. It follows that currently c' receives exactly $2q+K$ more support than c, and thus wins the seat. We now show that we can make the distinguished party $P^* = c$ win the seat by bribing at most K voters if and only if there exists a hitting set of size at most K.

($\Leftarrow$) Suppose there exists a hitting set $U' \subseteq U$ of size exactly K (if $|U'| < K$, it can be padded to size exactly K by adding arbitrary elements from U). For each $u_i \in U'$, we bribe one voter from group (4.1) to put u_i at their first position. These u_i now each receive the $2q+1$ top choices required by the threshold, i.e., they participate in the further apportionment process. Note that each u_i can receive a support of at most $4q+1$, so no u_i can win the seat against c or c'. Groups (4.3) and (4.4) do not change the support difference between c and c' and thus can be ignored. However, since U' is a hitting set, all $2q$ voters in group (4.2) now vote for a party in U' instead of c', reducing the difference between c and c' by $2q$. Further, we have bribed K voters from group (4.1) to not vote for c', which reduces the difference between c and c' by another K votes. Therefore, c and c' now have the same support, and since we assume tie-breaking to prefer c, party c wins the seat.

(⇒) Suppose the smallest hitting set has size $K' > K$. That is, with only K elements of U we can hit at most $q-1$ sets from S. It follows that by bribing K voters from group (4.1) to vote for some $u_i \in U$ instead of c', we can only prevent up to $2(q-1)$ voters from group (4.2) to use their second chance to vote for c'. Thus, we reduce the difference between c and c' by at most $2(q-1)+K$, which is not enough to make c win the seat. Now consider that we do not use the complete budget K on this strategy (to bribe voters of group (4.1)), but only $K'' < K$. Note that by bringing only K'' parties from U above the threshold, we can only hit up to $2(q-1-(K-K''))$ sets from S. So the difference between c and c' is reduced by at most $2(q-1-(K-K''))+K''$ using this strategy. However, we now have a budget of $K-K''$ left to bribe voters, e.g., from group (4.1), to vote primarily for c without bringing any additional u_i above the threshold. It is easy to see that we will only reduce the difference between c and c' by at most $2(K-K'')$ with this strategy as, in the best case, c gains one supporter and c' loses one with a single bribery action. Thus, we cannot reduce the difference between c and c' by more than $2(q-1-(K-K''))+K''+2(K-K'') = 2(q-1)+K''$ with this mixed strategy. For each $K'' \leq K$, it holds that $2(q-1)+K'' < 2q+K$. Therefore, if there is no hitting set of size at most K, we cannot make the distinguished party c win against c'.

The proof for $\mathcal{R}$-DESTRUCTIVE-BRIBERY works analogously by exchanging c and c'. However, due to the tie-breaking we have to give c' one additional vote. This results in the following votes.

$$\begin{aligned}
\mathcal{V} = (4q+K+2 \text{ votes} \quad & c \succ \cdots, \\
4q+3 \text{ votes} \quad & c' \succ \cdots, \\
\text{for each } j \in [q], 2 \text{ votes} \quad & S_j \succ c \succ \cdots, \\
\text{for each } i \in [p], q-\gamma_i \text{ votes} \quad & u_i \succ c \succ \cdots, \\
\text{for each } i \in [p], q-\gamma_i \text{ votes} \quad & u_i \succ c' \succ \cdots)
\end{aligned}$$

By an analogous argumentation as in the constructive case, it holds that we can make c lose the seat if and only if there exists a hitting set of size at most K. □

4.7 Conclusions

We developed efficient algorithms to decide whether strategic campaigns (bribery) for apportionment elections with thresholds can be successful. The algorithms can also be used to compute an exact plan which voters to bribe, and how to change their votes, in order to gain the maximum possible number of additional seats for a distinguished party (or make it lose the maximum possible number of seats in the destructive case). Although this is a positive result from the perspective of a campaign manager, the efficient computability of (optimal) strategic campaigns is a negative result in terms of apportionment methods being vulnerable to such attacks. In the light of online profiles and social networks we use in our everyday life, predicting votes and convincing specific voters to change their votes using individualized advertisement becomes more and more easy. So strategic campaigns will become more and more a threat.

As one countermeasure we introduced the second-chance mode of voting, where voters of parties below the threshold get a second chance to vote. It turns out that the second-chance mode makes computing strategic campaigns intractable. Thus, it provides more resistance to strategic campaigns. Further, note that the second-chance mode could resolve criticism many people put up against thresholds. A voter who thinks his/her most preferred party is likely to not make it above the threshold will probably vote for one of the bigger parties instead—against the true preference. And voters who still vote for small parties might end up unrepresented in the parliament. The second-chance mode encourages voters to vote also for smaller parties if it is their true preference, as it gives these voters a second chance if the small party doesn't make it above the threshold. We suggest studying the extent to which voters' satisfaction with parliament increases when they use the second-chance mode.

As further future work, we propose to study more sophisticated cost functions for convincing voters to change their vote (e.g., *distance bribery* as studied by Baumeister et al. [12]). That is, it might cost more to change e.g. a left-wing vote to a right-wing vote than to a medium-left vote. This models strategic campaigns even more realistically. Further, results from Faliszewski et al. [36] show that structured preferences such as single-peaked preferences can decrease the complexity of *electoral control*, a research field which is related to strategic campaigns as we study them. It would thus be interesting to see if the second-chance mode still protects the election from strategic campaigns when preferences are single-peaked. In case the campaigns can be efficiently computed then, one can check whether the intractability remains for *nearly* single peaked preferences. Faliszewski et al. [34] showed that this can increase the complexity of electoral control again.

4.8 Publication

Together with additional experimental results and some results from my master thesis, the results from this chapter were submitted to the *32nd International Joint Conference on Artificial Intelligence (IJCAI)* in January, 2023.[2]

[2]By the time this thesis is published, the submitted paper was rejected and resubmitted to another conference.

C. Laußmann, J. Rothe, and T. Seeger. *Apportionment with Thresholds: Strategic Campaigns Are Easy in the Top-Choice But Hard in the Second-Chance Mode*. submitted to the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023) in January, 2023

4.9 Personal Contribution

The idea of studying strategic campaigns in apportionment with electoral threshold was developed by me together with Tessa Seeger. Theorem 1 (corresponding to Theorem 1 in [59] and to a proof in the appendix of that paper) was proven by me, and I developed the algorithms. I thank Niclas Boehmer and Martin Bullinger for their helpful comments on the LRM algorithms at a *Schloss Dagstuhl* seminar in 2021. The proof that the second-chance mode increases computational complexity (Theorem 2 in this chapter, and Theorem 2 in [59]) was conducted by me. Tessa Seeger and Jörg Rothe fixed some minor issues in the proofs. The experiments in [59] (which are not part of this thesis) were conducted by me and Tessa Seeger in joint work. The writing of the paper under review [59] was conducted in equal parts by Jörg Rothe, Tessa Seeger, and me. I reused a few text passages from [59] in this chapter which were not completely written by me.

This chapter is related with my master thesis. I proved a preliminary version of Theorem 1 before starting my master thesis. This result was submitted to a conference during the time I worked on my master thesis, and got rejected. This is why it is referenced in my master thesis, and some results in my master thesis build on it. Further, I proved several additional results during my master thesis which are now included as Theorem 3 in [59] (which is not part of this thesis). To state the difference between my master thesis and this chapter clearly: In this chapter we study only strategic campaigns in the model of bribery. My master thesis deals with electoral control—a similar yet different model of influencing the election outcome. In electoral control we don’t change votes, but we add/delete votes, or parties.

Chapter 5

Designing More Expressive Ballots for Multiwinner Elections

5.1 Summary

Throughout this thesis, we talk a lot about *preferences* of voters. But what are these preferences, actually? In fact, this is not easy to answer. Voters might in one setting just find things either good or bad, and in another setting they have precise utilities for each option. When it comes to multiwinner voting, we have even more complex preferences as the combinations of alternatives might influence the utility of each other. In our opinion, some of the most often occurring preference types for preferences in multiwinner voting are unconditional approvals, substitution effects, incompatibilities between alternatives, and dependencies (in the sense of mutual requirement). In this chapter, we develop a ballot format which is very easy to use for the voters, and allows them to express the four mentioned preference types.

5.2 Introduction

Approval ballots allow a voter to expresses which alternatives he/she likes, e.g., by checking boxes on the ballot. This ballot format is often used—mainly because it is so simple for the voters to understand. However, approval ballots are only fully expressive when voters in a single-winner election have dichotomous preferences. Especially in the light of multiwinner voting it is highly questionable whether approval ballots can sufficiently express the voters' true preferences. Consider the following example.

Example:

Beatrix, Chris, and Dave plan their joint weekend vacation. They want to select one activity for each day (Saturday and Sunday).

> *"I want to do sports on one day. Generally I like swimming, hiking, and climbing. We should do one of the three, but no matter what, the other day we should relax or maybe visit a museum"*, says Beatrix.

As we see, Beatrix is satisfied with swimming, hiking, or climbing alone, but once we select more than one she finds it too exhaustive and is less satisfied than with one activity alone. We say the preference includes an **incompatibility**. This is only one example of a preference which cannot be expressed properly with an approval ballot. Let's sketch another preference that is likely to occur.

> **Example:**
>
> For her birthday party, Eve asks what food she should prepare for the buffet.
>
> > *"I like potatoes and fries. And I also like your fancy noodle salad and self-made bread, Eve! However, you can choose between the potatoes and fries. I like both but one of them is sufficient"*, says Alberto.

Alberto finds both potatoes and fries nice, but he doesn't care whether only potatoes, only fries, or both are served. In any case he has the same degree of satisfaction. We say Alberto's preference includes **substitution** effects. But his preference also includes **simple approval**. He likes the noodle salad and the bread—independent of each other.

> **Example:**
>
> Dave is also a guest at Eve's party. He also has a preference for the buffet.
>
> > *"I pretty much like your self-made bread! However, I cannot eat bread alone. You will also have to serve your famous herb butter."*

Dave's preference includes a **dependency**. That is, herb butter alone is just as useless as bread alone. Only the combination of the two will satisfy Dave.

As we see, some preferences can be adequately expressed by approval ballots, but other common preferences cannot. Of course, Dave can approve bread and herb butter—but there is no way he can state that he only likes them if both are selected. Beatrix can approve swimming, hiking, and climbing—but there is a risk that then two or all three activities are selected, which she didn't want. She could also strategically approve only swimming—but this is also not ideal as it decreases the overall approval for hiking and climbing, so that in the end it might be that no sport at all is selected. To account for such preferences, in this chapter we develop an easy-to-use ballot format, the *bounded approval ballots* with which a voter can easily express all the four preference types (and also combinations of them). Thereby, bounded ballots consist of *bounded sets*, which are essentially approval ballots with three additional numbers: the lower bound (to model dependencies), the saturation point (for the substitution statements), and the upper bound (for the incompatibilities). We then study how they can be aggregated, and what axiomatic properties an aggregation method can guarantee under which conditions. We further illustrate with examples, by how much bounded ballots (with suitable aggregation methods) can make an election outcome better. Finally, we also present a web-application which makes use of bounded ballots to allow voters to vote in elections in a convenient way.

Note that there exist other approaches to account for more complex preferences in literature. Conditional preferences allow a voter to express a preference conditioned by the status of a given variable. The most prominent proposals of such ballot formats are due to Barrot and Lang [9], Boutilier et al. [17], and Booth et al. [16]. Similar ballots have been proposed for combinatorial auctions (see [18, 78]). However, these ballots are rather complicated compared to the very simple approval ballot format. Regarding the context of participatory budgeting, Jain et al. [48] propose to partition the project set in categories, and define the interaction type (e.g. substitution) for each partition. This is very different from our approach as they fix the partitions and interaction types for all voters, and we allow voters to do this on their own. Thus, voters can ‘partition’ the projects as they like. Fairstein et al. [32] generalized the work by Jain et al. [48] so that voters can now partition the projects as they like. However, they consider mostly substitution effects. Further, the goals in participatory budgeting are often different from the goals in the context of multiwinner voting we consider.

5.3 Preliminaries

A multiwinner election consists of a set of m alternatives (or candidates) $\mathcal{A} = \{a_1, \dots, a_m\}$, a vote profile $\mathfrak{B} = (\boldsymbol{B}_1, \dots, \boldsymbol{B}_n)$ which is a list of ballots $\boldsymbol{B}_i$ of n voters $\mathcal{N} = \{1, \dots, n\}$, and an integer $K \in \{1, \dots, m\}$ which gives the committee size. We denote by $\mathcal{C}_K = \{\pi \subseteq \mathcal{A} \mid |\pi| = K\}$ the set of all K-sized committees. The outcome of an irresolute multiwinner election is a set of winning committees $\{\pi_1, \pi_2, \dots\} \subseteq \mathcal{C}_K$.

We use $\oplus$ to denote the concatenation between two lists. The subtraction of list B from list A will be denoted through $A \ominus B$ (where for each element in B the first occurrence of the element in A is removed). We sometimes omit the brackets around a list of length one.

5.4 Bounded Approval Ballots

A vote profile consists of ballots. The format of these ballots can vary depending on what method for aggregation should be used. The common Approval ballots consist of a set of alternatives. But we want to develop a ballot format which is similarly simple as approval ballots, but provides the ability to express more sophisticated preferences. The following is our proposal for such a ballot format.

Definition 1 (Bounded Approval Sets and Ballots)**.** *Le $\mathcal{A}$ be the set of alternatives. A* bounded (approval) set *is a tuple $B^j = \langle A^j, \ell^j, s^j, u^j \rangle$ such that $A^j \subseteq \mathcal{A}$ and $\ell^j, s^j, u^j \in \mathbb{N}$ with $1 \leq \ell^j \leq s^j \leq u^j \leq |A^j|$. Thereby ℓ^j denotes the lower bound, s the saturation point, and u the upper bound. A* bounded (approval) ballot *$\boldsymbol{B}_i$, for voter $i \in \mathcal{N}$, is a list $\boldsymbol{B}_i = (B_i^1, \dots, B_i^p)$ of bounded sets.*

That is, a bounded approval set contains a set of approved alternatives (just as an approval ballot) accompanied by three numbers ℓ^j, s^j, and u^j. Thereby the three numbers can be interpreted as follows. At least ℓ^j but no more than u^j have to be selected from the set A^j; while after s^j alternatives have been selected the voter will not enjoy any additional satisfaction from selecting more alternatives. The idea is to encode our initial modelling goals in the bounded approval sets as follows.

- *Standard approval ballots* can be expressed by setting $\ell^j = 1$, $s^j = u^j = |A^j|$: the more alternatives from A^j the better.
- *Incompatibilities* can be expressed by bounded sets with an upper bound $u^j = 1$: Selecting multiple alternatives from A^j is not desired by the voter because these alternatives are incompatible, but selecting one is desirable.
- *Substitution* can be expressed by bounded sets with $\ell^j = s^j = 1$ and $u^j = |A^j|$: Selecting one alternative from A^j is desired, but additional alternatives are substitutes.
- *Dependencies* can be expressed by bounded sets where $\ell^j = |A^j|$: All alternatives from A^j rely on each other, and are only useful for the voter if all of them are present.

As we see, for a voter it is not much effort to encode his/her preference. Approval ballots are basically still valid (with simple reformatting), and incompatibilities, substitution effects, and dependencies require the voter also only to set the bounds correctly (which, as we will see in the web-application later, is also not difficult with assistance of a good interface). Note that it is also possible to encode mixtures of the four preference types into a bounded set. For instance, the set $\langle\{a,b,c,d,e\},2,4,5\rangle$ encodes that at least 2 of a,b,c,d,e have to be selected (dependency), from 2 to 4 the satisfaction increases (simple approval), and from 4 on the satisfaction remains the same (substitution). These special cases, however, could in some cases put a higher cognitive burden on the voters, as it is hard to assist the voter with a simple computer interface here. In any case, most voters will be fine with the four basic types already. Further, a bounded approval ballot can consist of one or more bounded approval sets. This way the voter can state e.g. substitution effects for one set of alternatives, and a dependency for another at the same time. This enables the voter to express a lot.

To illustrate the mapping of preferences to bounded ballots, consider the example from the beginning again. Alberto likes potatoes and fries, but he considers them to be substitutes. This corresponds to the bounded set $\langle\{\text{potatoes},\text{fries}\},1,1,2\rangle$. Further, he likes noodle salad and the bread independent of each other. This corresponds to the bounded set $\langle\{\text{noodle salad},\text{bread}\},1,2,2\rangle$. His bounded ballot would then be

$$(\langle\{\text{potatoes},\text{fries}\},1,1,2\rangle,\langle\{\text{noodle salad},\text{bread},1,2,2\}\rangle).$$

5.5 Scoring with Bounded Approval Ballots

Having just bounded ballots alone doesn't help us solving the overall problem: computing an outcome, i.e., a size-K committee. Thus, we have to find methods to aggregate the bounded ballots. We now provide different *scoring functions* which map profiles and committees to real numbers. Committees with the highest score win.

Definition 2 (Scoring Function)**.** *A scoring function score is a function mapping a bounded approval ballot $\boldsymbol{B}$ and a committee π to a real value $score(\boldsymbol{B}, \pi)$. We extend scoring functions to profiles s.t. for every profile $\mathfrak{B}$, $score(\mathfrak{B}, \pi) = \sum_{\boldsymbol{B} \in \mathfrak{B}} score(\boldsymbol{B}, \pi)$.*

Of course, a scoring function for aggregating bounded approval ballots is useless if it doesn't respect what bounded approval ballots encode. We conduct a detailed axiomatic study in the next section. However, to provide an intuition how scoring functions could be build, let us first consider a bounded ballot consisting of only one bounded set $B^j = \langle A^j, \ell^j, s^j, u^j \rangle$. For a committee π, we want scoring functions to behave as depicted in Figure 5.1.

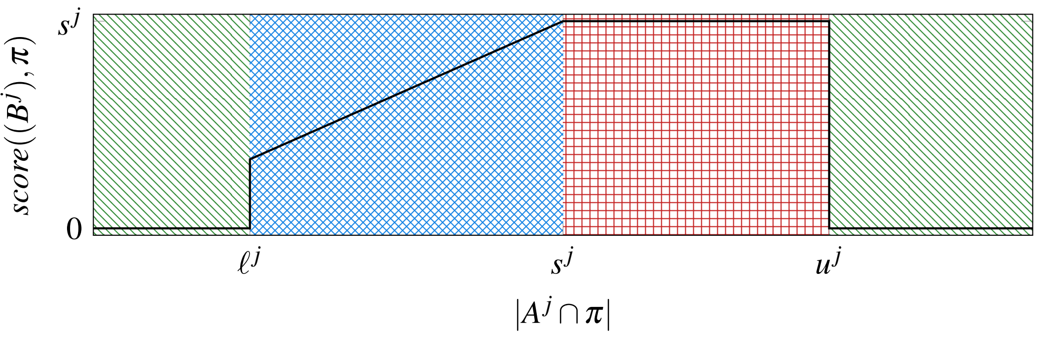


Figure 5.1: Intended behavior of scoring functions for a single bounded set.

The 🟩 area represents that π violates a dependency or incompatibility stated in B^j. Thus, π should score 0 (the voter has no satisfaction from π). The 🟦 area represents that each element in $\pi \cap A^j$ is independently approved according to B^j. That is, each element contributes to the total score. Finally, the 🟥 area represents that the elements in π show substitution effects according to B^j. In this area, no additional satisfaction (and thus no additional score) is achieved when more alternatives from A^j are in π. To conveniently treat regular approval ballots, in the 🟦 area we want the score to be exactly $|A^j \cap \pi|$, i.e., the number of approved alternatives in π. Note that the diagram in Figure 5.1 only shows how the whole committee π scores, but not how the individual alternatives in the committee score. However, it is convenient to distribute the points equally among the alternatives in $|A^j \cap \pi|$. That is, in the 🟩 area each alternative scores 0, in the 🟦 area each alternative scores 1, and in the 🟥 area each alternative scores $\frac{s^j}{|A^j \cap \pi|}$. We formalize this in the function ϕ.

$$\phi(B^j, \pi) = \begin{cases} 1 & \text{if } \ell^j \leq |A^j \cap \pi| \leq s^j \\ \frac{s^j}{|A^j \cap \pi|} & \text{if } s^j < |A^j \cap \pi| \leq u^j \\ 0 & \text{otherwise.} \end{cases}$$

At this point we introduce some useful notation: for a ballot $\boldsymbol{B}$ and $a \in \mathcal{A}$, let $\boldsymbol{B}_{|a} = \{B^j \in \boldsymbol{B} \mid a \in A^j\}$ be the bounded sets involving a. Further, in correspondence to the notation for simple approval ballots, instead of $\pi \cap \left(\bigcup_{B^j \in \boldsymbol{B}} A^j\right)$ we write $\pi \cap \boldsymbol{B}$ for short.

The following scoring function *total score* applied to a bounded ballot consisting of a single bounded set gives us exactly the behavior of the diagram in Figure 5.1.

Definition 3. *The* total score *function is defined as*

$$score_{tot}(\boldsymbol{B}, \pi) = \sum_{a \in \pi \cap \boldsymbol{B}} \sum_{B^j \in \boldsymbol{B}_{|a}} \phi(B^j, \pi).$$

Bounded ballots can of course contain more than one bounded set. When designing scoring functions, it is important that they deal with such ballots in a convenient way, too. Let us first consider a relatively simple case. We say a bounded ballot is *non-overlapping* when all bounded sets in the ballot are pairwise disjoint (i.e., for B^i, B^j holds $A^i \cap A^j = \emptyset$). The bounded sets in such a ballot are then completely independent. We could even treat them as if they were all from different voters—just as in regular approval voting it doesn't make a difference whether one voter approves a, b and another voter c, d, or a single voter approves a, b, c, d.[1] We want scoring functions to treat non-overlapping ballots like this, too: The score of the ballot for a committee π should be the sum of the scores of each bounded set for π. Note that $score_{tot}$ treats non-overlapping ballots exactly like this.

However, as soon as not all bounded sets in a ballot are disjoint, it is not exactly clear how a scoring function should behave. Say $a \in \pi$ is contained in three bounded sets B^1, B^2, B^3 within a single bounded ballot. In one set we have $\phi(B^1, \pi) = 0$ because the lower bound is not met. In the next set we have $\phi(B^2, \pi) = 1$ because $|A^2 \cap \pi|$ is exactly between lower bound and saturation point. Finally, in B^3 we have a substitution effect, so $\phi(B^3, \pi) = 0.75$. What is now the correct score contribution of a? In $score_{tot}$ the score contribution of a would be 1.75. Arguably, this is not so good because it behaves very different from what we expect in approval voting (alternatives which are not approved score 0, and approved alternatives score 1). Common operators like average, minimum, and maximum can be considered to avoid this. First, let us consider the average score function.

[1]Note that this makes a difference in variants of approval voting which aim at proportionality, such as *satisfaction approval voting* or *proportional approval voting*. However, this is not what we aim for here. We want to generalize regular approval voting.

Definition 4. *The* average score *function is defined as*

$$score_{avg}(\boldsymbol{B},\pi) = \sum_{a\in\pi\cap\boldsymbol{B}} \frac{1}{|\boldsymbol{B}_{|a}|}\cdot\sum_{B^j\in\boldsymbol{B}_{|a}} \phi(B^j,\pi).$$

Here, the score for a would be the average of 0, 1, and 0.75, that is, approximately 0.58. This seems quite natural since what the voter wants to express is that in some sense a dependency is violated, but at least a is good in combination with the elements from B^2 and B^3. So a should score something for B^2 and B^3, but not too much because one dependency is not met. However, one can just as well argue that the voter wants a to score 0 because one dependency is not met, or to score 1 because it is sufficient if in at least one combination of alternatives, i.e., in B^3, the alternative a is fully approved. This is captured by the following two scoring functions maximum score and minimum score.

Definition 5. *The* maximum score *function and* minimum score *function are defined as*

$$\begin{aligned} score_{min}(\boldsymbol{B},\pi) &= \sum_{a\in\pi\cap\boldsymbol{B}} \min\left(\phi(B^j,\pi)\mid B^j\in\boldsymbol{B}_{|a}\right)\\ score_{max}(\boldsymbol{B},\pi) &= \sum_{a\in\pi\cap\boldsymbol{B}} \max\left(\phi(B^j,\pi)\mid B^j\in\boldsymbol{B}_{|a}\right). \end{aligned}$$

Note that all presented scoring functions are equivalent (i.e., yield the same score) for every non-overlapping bounded ballot.

In the next subsection we will analyze the four scoring functions axiomatically. But before that, we want to mention that for all four scoring functions determining the best committee cannot be done in polynomial-time, unless P = NP. This is because we can simulate the (approval version of the) Chamberlin–Courant rule with them by submitting a single bounded set per voter, where each bounded set B^j has $s^j = 1$ and $u^j = |A^j|$. The observation then follows from the fact that Chamberlin–Courant winner determination is NP-hard [81].

5.5.1 Axiomatic Analysis

We now extend the rather simple modeling goal from Figure 5.1 with several other axioms which capture how scoring functions for bounded approval ballots should behave to be convenient. A summary of the axiomatic analysis can be found in Table 5.1. To start with, we present an axiom which ensures that ballots are treated in an *approval-like* fashion.

Definition 6 (Approval Adequacy)**.** *A scoring function score satisfies* approval adequacy *if for every ballot* $\boldsymbol{B}$ *and committee* π *the two conditions hold:*

1. $score(\boldsymbol{B},\pi) \leq |\pi\cap\boldsymbol{B}|$;
2. $score(\boldsymbol{B},\pi) = |\pi\cap\boldsymbol{B}|$ *whenever* $\ell^j \leq |A^j\cap\pi| \leq s^j$ *for all* $B^j\in\boldsymbol{B}$.

Informally, the score for an item being in a committee should be between 0 and 1 ("not approved" to "fully approved"), and if there is no reason (incompatibility, substitution, or dependency) not to fully approve an item, the score should be indeed 1 for that item. We will now see that this modelling goal is already violated by one of our scoring functions.

Theorem 1. *The scoring functions $score_{min}$, $score_{max}$, and $score_{avg}$ satisfy approval adequacy while $score_{tot}$ violates it.*

Proof. Let us first consider $score_{tot}$. It is easy to see that whenever an alternative appears in multiple bounded sets within a ballot, the alternative can score more than 1 which violates the first condition of approval adequacy. As an example, consider the ballot $\boldsymbol{B} = (\langle\{a_1,a_2\},1,2,2\rangle, \langle\{a_1,a_3\},1,2,2\rangle)$ for which holds $score_{tot}(\boldsymbol{B},\{a_1\}) = 2$.

For the other scoring functions, note that (for each alternative in $\pi \cap \boldsymbol{B}$) they select either a minimum of values from the ϕ-function, a maximum, or the average. Since $0 \leq \phi(B^j,\pi) \leq 1$ for all bounded sets $B^j \in \boldsymbol{B}$, and all bundles π, also the score for each alternative must be between 0 and 1, so the total score is at most $|\pi \cap \boldsymbol{B}|$. So they all meet the first condition of approval adequacy. For the second condition, note that whenever $\ell^j \leq |A^j \cap \pi| \leq s^j$ for all $B^j \in \boldsymbol{B}$, we have $\phi(B^j,\pi) = 1$ for all $B^j \in \boldsymbol{B}$. The minimum, maximum, and average of these $\phi(B^j,\pi)$ is 1. Thus, each alternative in $\pi \cap \boldsymbol{B}$ scores 1 which results in a total of $|\pi \cap \boldsymbol{B}|$. $\square$

Next, we define two axioms enforcing that a violated incompatibility or dependency—when added to the ballot—should not increase the score. Note that we allow it to decrease the score because if an alternative appears also in other bounded sets where lower and upper bound are not violated, it might be reasonable that the total degree of approval for the alternative is lower with a violated bounded set. This is exactly the consideration behind $score_{avg}$ and $score_{min}$. However, we also do not require it to strictly decrease the score, because of our consideration behind $score_{max}$.

Definition 7 (Incompatibility Adequacy)**.** *A scoring function score satisfies* incompatibility adequacy *if for every $A \subseteq \mathcal{A}$, and all ballots $\boldsymbol{B}$ and $\boldsymbol{B}' = \boldsymbol{B} \oplus \langle A,1,1,1\rangle$, the following holds:*

- *$score(\boldsymbol{B},\pi) \leq score(\boldsymbol{B}',\pi)$ for every π with $|\pi \cap A| = 1$;*
- *$score(\boldsymbol{B},\pi) \geq score(\boldsymbol{B}',\pi)$ for every π with $|\pi \cap A| \neq 1$.*

Theorem 2. *All four scoring functions satisfy incompatibility adequacy.*

Proof. Let $\boldsymbol{B}' = \boldsymbol{B} \oplus \langle A,1,1,1\rangle$ be the ballot with added incompatibility.

(Case 1) Assume $|\pi \cap A| \neq 1$. Note that $\phi(\langle A,1,1,1\rangle,\pi) = 0$. For $score_{tot}$ it is now clear that by adding $\langle A,1,1,1\rangle$ to the ballot the score cannot increase. For $score_{avg}$ note that for each $a \in \pi \cap A$, $|\boldsymbol{B}'_{|a}| = 1 + |\boldsymbol{B}_{|a}| > |\boldsymbol{B}_{|a}|$ holds, *i.e.*, the normalization factor decreases. This decrement together with $\phi(\langle A,1,1,1\rangle,\pi) = 0$ results in a decreased score contribution of a and since for all alternatives $b \in \pi \setminus A$ the contribution is unchanged, we have

$score_{avg}(\boldsymbol{B},\pi) \geq score(\boldsymbol{B}',\pi)$. For $score_{min}$, due to $\phi(\langle A,1,1,1\rangle,\pi) = 0$ we have zero score contribution in $\boldsymbol{B}'$ for each $a \in \pi \cap A$. Again, since for all alternatives $b \in \pi \setminus A$ the contribution is unchanged, the score cannot increase. For $score_{max}$, the score doesn't change at all, since adding a bounded set to the ballot for which the ϕ-value is zero cannot change the maximum over the ϕ-values.

(Case 2) Assume that $\pi \cap A = \{a\}$ for some $a \in \mathcal{A}$. For $score_{tot}$ it is easy to see that the score can never decrease when adding a bounded set to a ballot. For $score_{min}$, due to $\phi(\langle A,1,1,1\rangle,\pi) = 1$ the minimum ϕ-value for a cannot decrease. Since for all alternatives $b \in \pi \setminus A$ the contribution is unchanged, the score cannot decrease. For $score_{max}$, due to $\phi(\langle A,1,1,1\rangle,\pi) = 1$ alternative a now contributes 1 to the score. Again, since for all alternatives $b \in \pi \setminus A$ the contribution is unchanged, the score cannot decrease. For $score_{avg}$ we distinguish three cases. (1) If $\boldsymbol{B}_{|a} = \emptyset$, then clearly $score(\boldsymbol{B}',\pi) = score_{avg}(\boldsymbol{B},\pi) + 1 > score_{avg}(\boldsymbol{B},\pi)$. (2) If $\boldsymbol{B}_{|a} \neq \emptyset$ and for all $B^j \in \boldsymbol{B}_{|a}$, $\ell^j \leq |A^j \cap \pi| \leq s^j$ holds. Then clearly $score_{avg}(\boldsymbol{B},\pi) = score(\boldsymbol{B}',\pi)$. (3) Finally, assume $\boldsymbol{B}_{|a} \neq \emptyset$ but for some $B^j \in \boldsymbol{B}_{|a}$ either $|A^j \cap \pi| < \ell^j$ or $s^j < |A^j \cap \pi|$. Let $\alpha = \sum_{B^j \in \boldsymbol{B}_{|a}} \phi(B^j,\pi)$ and $\beta = |\boldsymbol{B}_{|a}|$. By assumption, $\alpha < \beta$ holds. Note that α/β is the contribution of a to the score of ballot $\boldsymbol{B}$. Moreover, $\alpha+1/\beta+1$ is the contribution of a to the score of $\boldsymbol{B}'$. Since for any $0 \leq \alpha < \beta$ we have $\alpha/\beta < \alpha+1/\beta+1$, we immediately obtain $score_{avg}(\boldsymbol{B},\pi) < score(\boldsymbol{B}',\pi)$. □

Definition 8 (Dependency Adequacy)**.** *A scoring function score satisfies* dependency adequacy *if for every $A \subseteq \mathcal{A}$, and all ballots $\boldsymbol{B}$ and $\boldsymbol{B}' = \boldsymbol{B} \oplus \langle A,|A|,|A|,|A|\rangle$ holds:*

- *$score(\boldsymbol{B},\pi) \leq score(\boldsymbol{B}',\pi)$ for every π with $A \subseteq \pi$;*
- *$score(\boldsymbol{B},\pi) \geq score(\boldsymbol{B}',\pi)$ for every π with $A \not\subseteq \pi$.*

Theorem 3. *All four scoring functions satisfy dependency adequacy.*

Proof. The argumentation is analogous to the proof for incompatibility adequacy. This time, of course, we consider a ballot $\boldsymbol{B}' = \boldsymbol{B} \oplus \langle A,|A|,|A|,|A|\rangle$ with added dependency. □

Beside encoding approval, incompatibilities, and dependencies, one of our main goals was to allow voters to encode substitution effects. By our next axiom we want to find out whether our scoring functions treat such ballots properly.

Definition 9 (Substitution Adequacy)**.** *A scoring function score satisfies* substitution adequacy *if for every ballot $\boldsymbol{B}$ and committee π for which there exists an alternative $a^\star \in \mathcal{A} \setminus \pi$ such that for all bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$, it is the case that $s^j \leq |A^j \cap \pi| \leq u^j - 1$, we have $score(\boldsymbol{B},\pi) = score(\boldsymbol{B},\pi \cup \{a^\star\})$.*

Informally, if according to all bounded sets an item $a^\star$ is considered a substitute w.r.t. π, then adding $a^\star$ to π should neither increase, nor decrease the score. This property, unfortunately, is only satisfied by one scoring function.

Theorem 4. *The scoring functions $score_{min}$, $score_{max}$, and $score_{avg}$ fail substitution adequacy. Only $score_{tot}$ satisfies this property.*

Proof. Consider $score_{avg}$ first. Let $\mathcal{A} = \{a_1, a_2, a_3\}$, $\pi = \{a_1, a_2\}$, and $\boldsymbol{B} = (\langle\{a_1, a_2\}, 1, 2, 2\rangle, \langle\{a_1, a_3\}, 1, 1, 2\rangle)$. Note that a_3 is a proper substitute according to substitution adequacy, i.e., it fulfills the conditions required for $a^\star$. On the one hand, we have $score_{avg}(\boldsymbol{B}, \pi) = 2/2 + 1/1 = 2$. On the other hand, for $\pi' = \{a_1, a_2, a_3\}$, we have $score_{avg}(\boldsymbol{B}, \pi') = \frac{1+1/2}{2} + 1/1 + \frac{1/2}{1} = 9/4$, which is more than π's score. Thus, the substitute changes the score.

For $score_{max}$ the same ballot results in $score_{max}(\boldsymbol{B}, \pi) = 1 + 1 = 2$ while $score_{max}(\boldsymbol{B}, \pi') = 1 + 1 + 1/2 > 2$.

For $score_{min}$ the ballot $\boldsymbol{B} = (\langle\{a_1, a_2\}, 1, 1, 2\rangle, \langle\{a_1, a_3\}, 1, 1, 2\rangle)$ results in $score_{min}(\boldsymbol{B}, \pi) = 1/2 + 1/2 = 1$ while $score_{min}(\boldsymbol{B}, \pi') = 1/2 + 1/2 + 1/2 > 1$.

Finally, consider $score_{tot}$. Let $\boldsymbol{B}$ be a bounded approval ballot, π a committee, and $a^\star \in \mathcal{A} \setminus \pi$ an alternative as in the definition of substitution adequacy. Let $B = \langle A, \ell, s, u\rangle$ be an arbitrary bounded set from $\boldsymbol{B}$ such that $a^\star \in A$. By the definition of $a^\star$, we know that $s \leq |A \cap \pi| \leq u - 1$. Hence, the contribution of B to $score_{tot}(\boldsymbol{B}, \pi)$ is s^j. Now, for $\pi' = \pi \cup \{a^\star\}$ we have $s + 1 \leq |A \cap \pi'| \leq u$. Hence, the contribution of B to $score_{tot}(\boldsymbol{B}, \pi')$ is also s^j. This applies to any bounded set including $a^\star$. Since the contributions of sets which don't include $a^\star$ are also unchanged, we have $score_{tot}(\boldsymbol{B}, \pi) = score_{tot}(\boldsymbol{B}, \pi')$. □

Apart from the four central axioms which describe our modelling goals there are further axioms which ensure convenient behavior. The first one states that adding a bounded set which does not conflict with a committee π should not decrease the score of π.

Definition 10 (Ballot-Size Monotonicity)**.** *Let $\boldsymbol{B}$ be a ballot and π a committee. A scoring function score satisfies* ballot-size monotonicity *if for every bounded set $B = \langle A, \ell, s, u\rangle$ such that $\ell \leq |A \cap \pi| \leq u$, we have $score(\boldsymbol{B}, \pi) \leq score(\boldsymbol{B} \oplus B, \pi)$.*

Unfortunately, ballot-size monotonicity is only satisfied by half of our scoring functions.

Theorem 5. *Ballot-size monotonicity is satisfied by $score_{max}$ and $score_{tot}$, but failed by $score_{min}$, and $score_{avg}$.*

Proof. Note that by adding a bounded set B^j to a ballot the score under $score_{max}$ and $score_{tot}$ cannot decrease. Thus, both satisfy ballot-size monotonicity.

For $score_{avg}$ consider the following counter example. Let $\mathcal{A} = \{a_1, a_2, a_3\}$, $\pi = \{a_1, a_2\}$, and $\boldsymbol{B} = (\langle\{a_1, a_2\}, 1, 2, 2\rangle)$. Observe that we have $score_{avg}(\boldsymbol{B}, \pi) = 2$. Now, consider the bounded set $B = \langle\{a_1, a_2, a_3\}, 1, 1, 3\rangle$ for which holds $\ell = 1 \leq |\{a_1, a_2, a_3\} \cap \pi| \leq u = 3$. Since $score_{avg}(\boldsymbol{B} \oplus B, \pi) = \frac{1+1/2}{2} + \frac{1+1/2}{2} = 3/2 < 2$, ballot-size monotonicity is not satisfied.

For $score_{min}$ a similar counter example works. Let $\mathcal{A} = \{a_1, a_2, a_3\}$, $\pi = \{a_1, a_2\}$, and $\boldsymbol{B} = (\langle\{a_1, a_2, a_3\}, 1, 2, 2\rangle)$. Here, we have $score_{min}(\boldsymbol{B}, \pi) = 2$. Consider the bounded set $B = \langle\{a_1, a_2\}, 1, 1, 2\rangle$. Now we have $score_{min}(\boldsymbol{B} \oplus B, \pi) = 1 < 2$. □

The next axiom, ballot-splitting monotonicity, says that expressing an equivalent statement with one big ballot, or several smaller ones, should result in the same score.

Definition 11 (Ballot-Splitting Monotonicity)**.** *A scoring function score satisfies* ballot-splitting monotonicity *if for every committee π and every ballot $\boldsymbol{B}$ for which there exists a bounded set $B^{j^\star} \in \boldsymbol{B}$ such that $\ell^{j^\star} \leq |A^{j^\star} \cap \pi| \leq s^{j^\star}$, then, for $\boldsymbol{B}' = (\boldsymbol{B} \ominus B^{j^\star}) \oplus (\langle\{a\}, 1, 1, 1\rangle \mid a \in A^{j^\star} \cap \pi)$, we must have $score(\boldsymbol{B}, \pi) = score(\boldsymbol{B}', \pi)$.*

Ballot-splitting monotonicity is satisfied by all scoring functions we introduced.

Theorem 6. *All four scoring functions satisfy ballot-splitting monotonicity.*

Proof. By the definition of ballot-splitting monotonicity, we replace a bounded set B^j with $|A^j \cap \pi| = q$ such that $\phi(B^j, \pi) = 1$ by q one-elementary bounded sets $B^j_1, \dots, B^j_q$ each with $\phi(B^j_i, \pi) = 1$ where each of these bounded sets contains exactly one element from $A^j \cap \pi$. For $score_{tot}$ this means removing q points from the score, and afterwards adding q points. Further, this operation is neutral on the minimal and maximal values for ϕ for all alternatives, meaning that the score does not change in $score_{min}$ and $score_{max}$. Finally, for $score_{avg}$ note that the number of bounded sets where a is contained in doesn't change for any $a \in \mathcal{A}$ by this operation. So overall splitting the ballot has no effect on the normalization factor. Further, each $a \in A^j \cap \pi$ loses, and gains one point by the operation. Thus, also in $score_{avg}$ the score remains unchanged. □

Finally, score monotonicity requires the score not to decrease when adding an alternative to the committee which doesn't conflict with any bounded set.

Definition 12 (Score Monotonicity)**.** *A scoring function score satisfies* score monotonicity *if for every ballot $\boldsymbol{B}$ and committee π for which there exists an alternative $a^\star \in \mathcal{A} \setminus \pi$ such that for all bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$ it is the case that $\ell^j \leq |A^j \cap \pi| \leq u^j - 1$, we have that $score(\boldsymbol{B}, \pi) \leq score(\boldsymbol{B}, \pi \cup \{a^\star\})$.*

Just as substitution adequacy, unfortunately, score monotonicity is only satisfied by $score_{tot}$.

Theorem 7. *The scoring functions $score_{min}$, $score_{max}$, and $score_{avg}$ fail score monotonicity. Only $score_{tot}$ satisfies this property.*

score$_{\mathbf{x}}$	*min*	*max*	*avg*	*tot*
Approval Adequacy ♣	✓	✓	✓	✗
Substitution Adequacy ♣	✗	✗	✗	✓
Incompatibility Adequacy	✓	✓	✓	✓
Dependency Adequacy	✓	✓	✓	✓
Ballot-Size Mon.	✗	✓	✗	✓
Ballot-Split. Mon.	✓	✓	✓	✓
Score Mon.	✗	✗	✗	✓

Table 5.1: Summary of our axiomatic analysis. ♣ indicates the impossibility result.

Proof. We first prove that $score_{tot}$ satisfies score monotonicity. Let $a^\star$ be the alternative described in the definition, i.e., for all bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$ it is the case that $\ell^j \leq |A^j \cap \pi| \leq u^j - 1$. Note that for bounded sets where $a^\star$ is not a member of nothing is changed. For bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$ with $\ell^j \leq |A^j \cap \pi| \leq s^j - 1$ it is immediate that the score contributions of alternatives in $A^j \cap \pi$ are the same in $score_{tot}(\boldsymbol{B}, \pi)$ and $score_{tot}(\boldsymbol{B}, \pi \cup \{a^\star\})$, and $a^\star$'s contribution counts on top. In bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$ with $s^j \leq |A^j \cap \pi| \leq u^j - 1$ (i.e., where $a^\star$ is a substitute) we know that the contribution of B^j to $score_{tot}(\boldsymbol{B}, \pi)$ is s^j. This contribution is unchanged in $score_{tot}(\boldsymbol{B}, \pi \cup \{a^\star\})$. Thus, we can conclude that $score_{tot}(\boldsymbol{B}, \pi) \leq score_{tot}(\boldsymbol{B}, \pi \cup \{a^\star\})$, and score monotonicity is satisfied.

Now consider the other scoring functions. Let $\mathcal{A} = \{a_1, a_2, a_3\}$, $\pi = \{a_2, a_3\}$, and $\boldsymbol{B} = (\langle\{a_1, a_2\}, 1, 1, 2\rangle, \langle\{a_1, a_3\}, 1, 1, 2\rangle)$. Note that a_1 fulfills the conditions required for $a^\star$ in the definition of score monotonicity. We have $score_{avg}(\boldsymbol{B}, \pi) = score_{min}(\boldsymbol{B}, \pi) = score_{max}(\boldsymbol{B}, \pi) = 2$. However, $score_{avg}(\boldsymbol{B}, \pi \cup \{a_1\}) = \frac{1/2 + 1/2}{2} + 1/2 + 1/2 < 2$, $score_{min}(\boldsymbol{B}, \pi \cup \{a_1\}) = \frac{1}{2} + 1/2 + 1/2 < 2$, and $score_{max}(\boldsymbol{B}, \pi \cup \{a_1\}) = \frac{1}{2} + 1/2 + 1/2 < 2$. This shows that score monotonicity is not satisfied by the other scoring functions. □

5.5.2 The Perfect Scoring Rule

As we see in Table 5.1, none of the scoring rules we proposed is perfect. Regarding the monotonicity axioms, $score_{tot}$ is the only scoring function which satisfies all axioms. However, $score_{tot}$ doesn't satisfy approval adequacy. Arguably, approval adequacy is a very important (if not the most important) axiom since our initial goal was to extend approval ballots. Due to the violated approval adequacy, a voter can cast a ballot in a way that an alternative he likes has an arbitrarily high score. This is absolutely not in the spirit of approval voting. Unfortunately, all scoring functions in our list which satisfy approval adequacy fail substitution adequacy—which is also a very important modeling goal. But can we actually design a scoring function which satisfies all our modeling goals? Unfortunately, we cannot.

Theorem 8. *There exists no scoring function that satisfies approval adequacy and substitution adequacy simultaneously.*

Proof. Suppose there exists a scoring rule *score* satisfying approval adequacy and substitution simultaneously. Throughout the proof, let $\mathcal{A} = \{a_1, a_2, a_3\}$, and

$$\boldsymbol{B} = (\langle\{a_1,a_3\},1,1,2\rangle, \langle\{a_2,a_3\},1,1,2\rangle).$$

For $\pi_1 = \{a_3\}$ approval adequacy implies $score(\boldsymbol{B}, \pi_1) = 1$. Note that a_1 is a suitable substitute for π_1 by the definition of substitution adequacy. Thus, for $\pi_1' = \{a_1, a_3\}$ must hold $score(\mathfrak{B}, \pi_1') = score(\mathfrak{B}, \pi_1) = 1$ in order for *score* to satisfy substitution. Interestingly, alternative a_2 is a suitable substitute for π_1'. Thus, for $\pi_1'' = \{a_1, a_2, a_3\}$ substitution entails that $score(\mathfrak{B}, \pi_1'') = score(\mathfrak{B}, \pi_1') = 1$. Consider now the committee $\pi_2 = \{a_1, a_2\}$. Approval adequacy on $\boldsymbol{B}$ and π_2 implies that $score(\boldsymbol{B}, \pi_2) = 2$. Alternative a_3 is a suitable substitute here, thus, for $\pi_2' = \{a_1, a_2, a_3\}$ we should have $score(\mathfrak{B}, \pi_2') = score(\mathfrak{B}, \pi_2) = 2$. Since $\pi_2' = \pi_1''$, the contradiction is immediate. □

This result is quite bad as it prevents us from modeling what we had in mind in the first place. However, we can still fix this issue at least partially.

The first solution is to simply not allow overlapping ballots. Recall that all scoring functions are equivalent on non-overlapping ballots. Thus, all their good properties are combined, i.e., they satisfy all axioms. Consequently, we can state the following theorem (which would just as well work with $score_{avg}$, $score_{min}$, or $score_{max}$ due to their equivalence).

Theorem 9. *Whenever each ballot is non-overlapping, the scoring function $score_{tot}$ satisfies all axioms we defined.*

This solution restricts voters in what they can express. However, it is not always clear what a voter wants to express with an overlapping ballot anyway. For instance, what is the interpretation of the following ballot from the proof of Theorem 4: $\boldsymbol{B} = (\langle\{a_1,a_2\},1,2,2\rangle, \langle\{a_1,a_3\},1,1,2\rangle)$? The naive interpretation is that a_1 and a_2 are approved, and that a_1 and a_3 are substitutes. But when we think about it, also the following interpretation is possible: a_1, a_2 are approved, and a_3 is only desired when a_1 is not selected, but the voter doesn't care about a_3 if a_1 is already selected. While both interpretations seem to be very similar, they completely change the score of the set $\{a_1, a_2, a_3\}$. The first interpretation corresponds to that a_1 is not fully approved because a_1 and a_3 'share' their approval. However, the second interpretation corresponds to that a_1 is fully approved, and a_3 contributes nothing to the score, i.e., is not approved. In the proof of Theorem 4 we see that the two interpretations can result in different final scores. To conclude, restricting ballots to not overlap might be not a major problem for the expressiveness, as it is often unclear what an overlapping ballot means anyway. Overlapping ballots could even increase the cognitive burden on the voters, because they have difficulties casting a ballot which actually expresses what they want to say.

The second solution is to remove the saturation point from the ballots (or, equivalently set it always to the upper bound). We thus drop one of our modeling goals, i.e., substitution, but at least we now have a scoring function satisfying all other properties.

Theorem 10. *The scoring rule $score_{max}$ satisfies all axioms when for each bounded set B^j in the ballot holds $s^j = u^j$.*

Proof. Since $score_{max}$ satisfied all axioms but score monotonicity and substitution adequacy already on the unrestricted ballots, we only have to show that it now satisfies the remaining two properties. Thereby, substitution adequacy is now trivially satisfied, as there no longer exist substitutes.

By the definition of score monotonicity, whenever there exists an alternative $a^\star \in \mathcal{A} \setminus \pi$ such that for all bounded sets $B^j \in \boldsymbol{B}_{|a^\star}$ holds $\ell^j \leq |A^j \cap \pi| \leq u^j - 1$, the score must not decrease by adding $a^\star$ to π. Due to the restricted ballots, we have $\ell^j \leq |A^j \cap (\pi \cup \{a^\star\})| \leq s^j$ for all $B^j \in \boldsymbol{B}_{|a^\star}$. Thus, for each of these bounded sets the ϕ-value is 1, which cannot reduce any maximum values. Since bounded sets which don't contain $a^\star$ are unaffected when we add $a^\star$ to the committee, the total score cannot decrease, i.e., score monotonicity is satisfied. □

This solution might in some cases be the less restrictive one. However, we cannot really consider it to be a good solution either. First, it doesn't allow expressing substitution effects, and second, it doesn't really resolve the problem of ballot interpretation. Consider, for example, the following ballot: $\boldsymbol{B} = (\langle\{a_1, a_2, a_3\}, 2, 3, 3\rangle, \langle\{a_2, a_4\}, 2, 2, 2\rangle)$. Does $\{a_1, a_2\}$ now score 2, that is, are a_1 and a_2 fully approved? Or did the voter mean that a_2 helps make a_1 useful, but in order to also achieve satisfaction from a_2 also a_4 must be selected?

All in all it seems that the best solution is the first one—requiring ballots to not overlap. We can then satisfy all axioms, avoid difficult ballot interpretations, reduce the cognitive burden on the voters, and still allow for a significant improvement in expressiveness compared to simple approval ballots.

5.6 Expressiveness Comparison to Approval Ballots

We now want to illustrate in how far bounded ballots are more expressive than simple approval ballots. Therefore, we present two examples where elections result in tremendously better outcomes for the voters when they use bounded approval ballots instead of simple approval ballots. This holds even if voters cast their approval ballot strategically. However, we assume voters do not communicate (otherwise they can trivially achieve optimal outcomes anyway). The examples aren't formal measures for expressiveness. Such measures (e.g. *distortion*) do not really make sense here anyway, as bounded approval ballots are already not fully expressive. The examples really only serve as an illustration what is possible with bounded approval ballots, and not with approval ballots.

Example:

(Pure Substitution) Assume that for every voter i, their preferences are defined such that there exists a set of alternatives $A_i \subseteq \mathcal{A}$ for which i is unsatisfied whenever $\pi \cap A_i = \emptyset$ and fully satisfied as soon as $\pi \cap A_i \neq \emptyset$. Note that i's preferences can easily be expressed by a single bounded set $\langle A_i, 1, 1, |A_i| \rangle$. Now, if voter i were asked to submit a standard approval ballot, the only reasonable ballot to submit would be A_i.

Let the number of voters n be such that $n-1$ is divisible by K, and let $\mathcal{A} = \{a_1, \ldots, a_{k^2}\}$. Consider the profile $\mathfrak{B}$ of bounded ballots in which ${}^{n-1}/_{k}$ voters submit $\langle \{a_1, \ldots, a_k\}, 1, 1, k \rangle$, ${}^{n-1}/_{k}$ voters submit $\langle \{a_{k+1}, \ldots, a_{2k}\}, 1, 1, k \rangle$, and so on. For the last voter group which submits $\left\langle \{a_{(k^2-k+1)}, \ldots, a_{k^2}\}, 1, 1, k \right\rangle$, we add one additional voter. If standard approval ballots were used, the first group of voters would approve $\{a_1, \ldots, a_k\}$, the second group $\{a_{k+1}, \ldots, a_{2k}\}$, and so on. Overall, all alternatives would be approved by the same number of voters, except for $a_{(k^2-k+1)}, \ldots, a_{k^2}$ which receive one approval more. Thus, if we were to select a committee of size K, $\{a_{(k^2-k+1)}, \ldots, a_{k^2}\}$ would be selected using standard approval ballots. This fully satisfies the last voter block, but no other voters. In the case of bounded approval ballots, we have $score(\mathfrak{B}, \{a_{(k^2-k+1)}, \ldots, a_{k^2}\}) = {}^{n-1}/_{k} + 1$, but $score(\mathfrak{B}, \{a_k, a_{2k}, \ldots, a_{k^2}\}) = n$ for any of the scoring functions we proposed. Thus, the committee $\{a_k, a_{2k}, \ldots, a_{k^2}\}$ will be chosen with bounded approval ballots, thus satisfying *all* voters.

The example above shows that already for preferences incorporating only approval and substitution, it is possible that only a fraction of the voters that could be fully satisfied are satisfied in simple approval voting. This becomes even worse with incompatibilities.

Example:

(Pure Incompatibility) Assume that for every voter i, their preferences are defined such that there exists a set of alternatives $A_i \subseteq \mathcal{A}$ for which i is unsatisfied whenever $|\pi \cap A_i| \neq 1$ and fully satisfied otherwise. Note that voter i's preferences can be expressed by a bounded set $\langle A_i, 1, 1, 1 \rangle$.

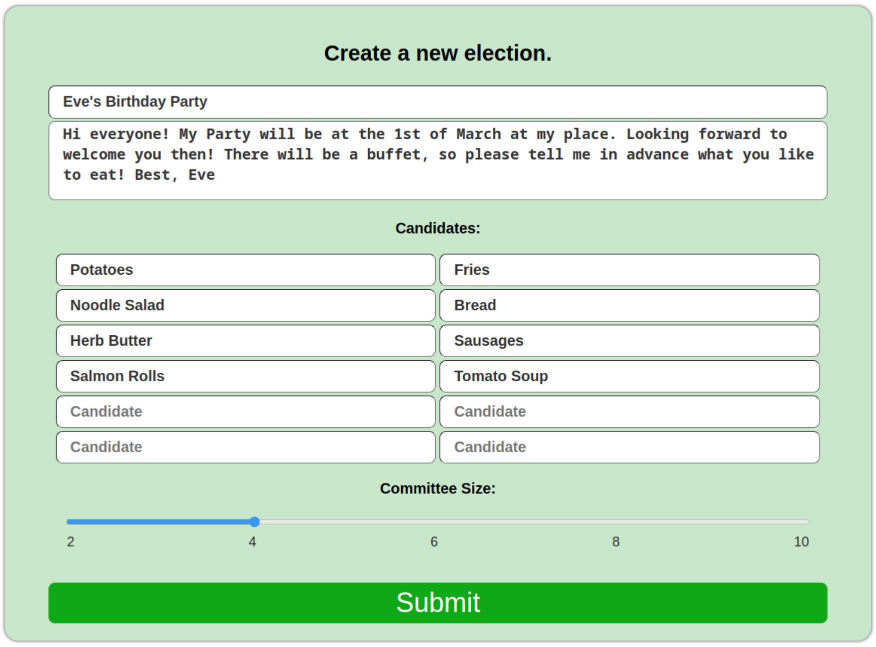
Let $n \geq 3$, $K = 2$, and $\mathcal{A} = \{a, b, c, d\}$. Assume that the first $n-1$ voters submit the ballot $\langle \{a, b\}, 1, 1, 1 \rangle$, and the last voter submits $\langle \{c, d\}, 1, 1, 1 \rangle$. It is clear that according to each of our scoring functions the committees maximizing the social welfare are $\{a, c\}$, $\{a, d\}$, $\{b, c\}$, or $\{b, d\}$. Each of them fully satisfies *all* voters. Under standard approval ballots it is reasonable to assume that the first $n-1$ voters would submit either $\{a\}, \{b\}$, or $\{a, b\}$, and the last one either $\{c\}, \{d\}$, or $\{c, d\}$. Then, unless the first $n-1$ voters all approve of only a or only b (which is unlikely if communication is impossible), the committee $\{a, b\}$ would maximize the social welfare. Note that it satisfies no voter at all.

As voters cannot express incompatibilities in approval ballots, it is possible that all voters dislike the outcome, but there exists an outcome fully satisfying every voter. This massive difference comes solely from the bit of extra information in the bounded ballots.

5.7 GoodVotes Web-Application

In the previous section we developed and analyzed a ballot format which (with a few limitations) allows voters to express more complex preferences than with approval ballots without major increment of cognitive effort. Probably, a question that came into the mind of the reader several times while reading this chapter is whether it is really so little cognitive effort for the voter to cast bounded ballots. For a mathematician, or a computer scientist, submitting sets and bounds in the correct format is certainly not a big deal. But for most people, the format alone might already raise the cognitive effort. Simple approval voting is so easy because the ballot format is so simple in reality. The voter doesn't submit a properly formatted set of candidates; the voter just checks the checkboxes of all approved candidates. So the missing component in our study is an easy way to let voters cast their ballots without requiring complicated formats. We propose a web-based application which guides voters through the process of casting bounded approval ballots. This should significantly reduce the cognitive burden of casting bounded approval ballots.

The application *GoodVotes* is written in the *Python* programming language with help of the *Flask* framework to handle the HTTP requests. The source code is available in the GitHub repository github.com/claussmann/GoodVotes, and can be downloaded for free. Please note that it is yet only a prototype of an application which could be used in real elections. While it is fully functional, it lacks proper security assessment, database integration, and is only optimized for the Chrome browser family. However, it shows how voters can be guided through the process of casting bounded ballots with ease. In the following we show some screenshots from the application, and explain how it works from the voter's perspective.



Create a new election.

Eve's Birthday Party

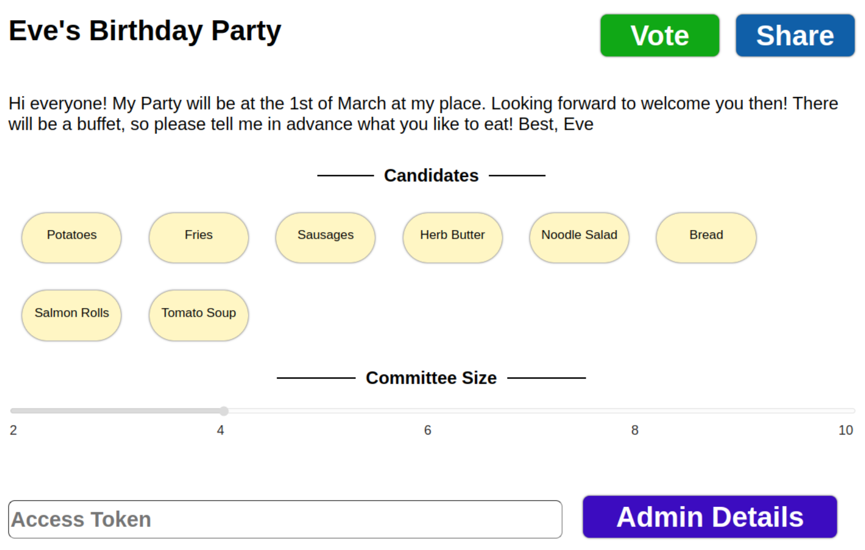
Hi everyone! My Party will be at the 1st of March at my place. Looking forward to welcome you then! There will be a buffet, so please tell me in advance what you like to eat! Best, Eve

Candidates:

Potatoes	Fries
Noodle Salad	Bread
Herb Butter	Sausages
Salmon Rolls	Tomato Soup
Candidate	Candidate
Candidate	Candidate

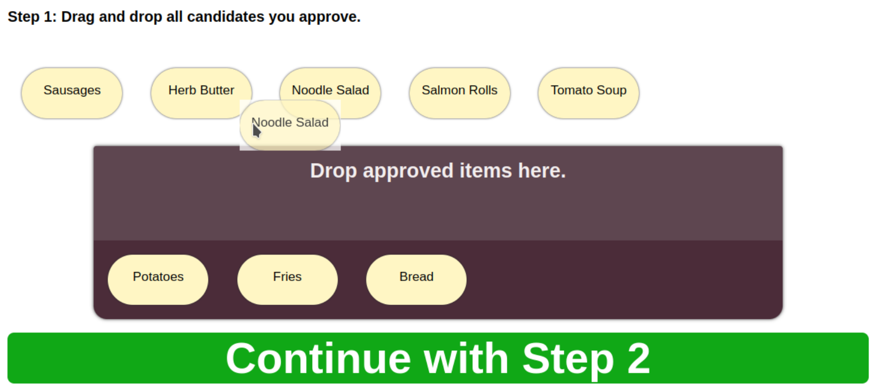
Committee Size: 4

The very first step in the process is for the election chair to create the election. In our example from the beginning of this chapter, Eve creates the election what food to serve for the buffet. She enters all candidates (here: food), and adds a title and description, so that every voter knows what he/she votes for. Finally, she sets the committee size. After clicking ‘submit’, Eve can send an invitation link to her guests (or the guests can use the search bar to find Eve’s election) who then see all the information on Eve’s election. There is also additional information and control options on this page which are only accessible via the password/token Eve received when creating the election. This includes ending or deleting the election, a live view how many voters voted already, and what is currently the best committee.

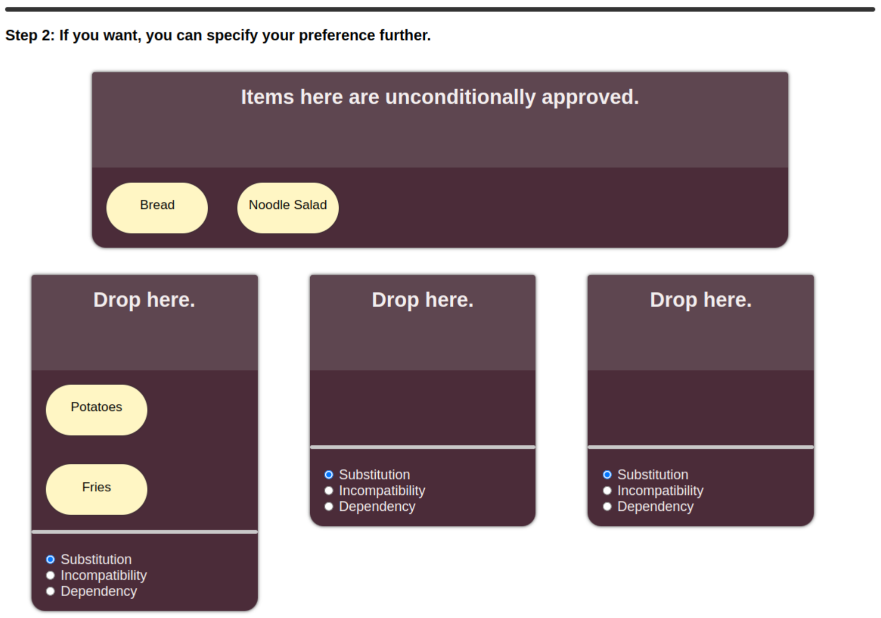

Eve's Birthday Party

Hi everyone! My Party will be at the 1st of March at my place. Looking forward to welcome you then! There will be a buffet, so please tell me in advance what you like to eat! Best, Eve

When a voter clicks on the ‘Vote’ button, she is guided through a two stage voting process. In the first step, she has to select all alternatives she generally approves of. That is, all alternatives which later appear in the bounded ballots.


Step 1: Drag and drop all candidates you approve.

Finally, in the second step the voter can further specify her preference. To this end the voter drags and drops approved items into bins. For each bin the voter can state whether the items in this bin are simply approved, are substitutes, depend on each other, or are incompatible.


Step 2: If you want, you can specify your preference further.

Items here are unconditionally approved.

Note that the application doesn't allow arbitrary bounded sets. In particular the bounded sets in a ballot are disjoint. However, as we proved in our axiomatic analysis overlapping ballots lead to many problems. So it is reasonable to avoid these problems in the application.

5.8 Conclusions

Our goal in this chapter was to find an approval-like ballot format which is more expressive than simple approval, but remains simple enough to use for the voters. Specifically, we wanted to allow voters to easily express substitution effects, incompatibilities, and dependencies between alternatives in a committee. These three (and mixtures of them) arguably are the most often appearing preference types in the context of approval voting. With *bounded approval ballots* we proposed a ballot format which can encode for simple approval, as well as for substitution effects, dependencies, and incompatibilities.

We proposed several scoring functions to evaluate such ballots. However, we ended with an impossibility result which shows that not all the modeling goals we had in mind are compatible in general. Approval-like behavior, and properly treating substitution effects at the same time is impossible. We have shown two ways to circumvent this problem. The most promising one is to forbid overlapping ballots, i.e., ballots in which an alternative is part of multiple bounded sets at the same time. Even with this restriction bounded ballots are much more expressive than simple approval ballots, and can significantly improve the election outcome for the voters as we illustrated in Section 5.6. Finally, in Section 5.7 we presented a web-application called *GoodVotes* which allows the voter to cast bounded ballots with ease, and without requiring knowledge of the underlying ballot format.

5.9 Publication

The results in this chapter together with further results are accepted, and will be presented at the *22nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*. The code for the web-application is available at github.com/claussmann/GoodVotes.[2]

D. Baumeister, L. Boes, C. Laußmann, and S. Rey. "Designing Expressive Preferences in Multiwinner Voting". In: *International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2023

5.10 Personal Contribution

The idea for bounded approval ballots and the scoring functions, was developed jointly in equal parts by Dorothea Baumeister, Linus Boes, Simon Rey, and me. The axiomatic analysis in Section 5.5.1 (corresponding to Theorems 3–6 in our AAMAS paper [14]) was conducted in equal parts by Simon Rey and me. The impossibility result in Theorem 8 (corresponding to Theorem 8 in [14]) was proven by Simon Rey. The two possibilities to escape of the impossibility result (Theorem 9 corresponding to Theorem 9 in our AAMAS paper [14] and Theorem 10) were developed by Simon Rey and me in equal parts. The examples in Section 5.6 (corresponding to Section 4.2 in [14]) were developed by me. The writing of the publication [14] was conducted in equal parts by Dorothea Baumeister, Linus Boes, Simon Rey, and me. I reused a few text passages from [14] in this chapter which were not completely written by me. Finally, the application *GoodVotes* was designed and programmed by me.

[2]By the time this thesis is published, GoodVotes was renamed to GoodVoteX. In its current version, I maintain and develop GoodVoteX together with my colleague Paul Nüsken.

Chapter 6

Discussion

In this thesis we analyzed methods from *computational social choice* in the light of real-world applications. Network centrality has applications to viral marketing, infrastructure planning, and many more. Especially in the light of social networks this research area becomes increasingly important. Designing voting methods, convenient yet expressive ballots, and protecting elections from fraud is also very important in everyday life. Democratization is progressing everywhere. We are asked for our opinion more and more. Therefore, it is crucial that we design good procedures, and also understand their properties and potential vulnerabilities. In the introduction to this thesis we presented four central problems (Section 1.3). To conclude this thesis, we now briefly want to review in how far we addressed them.

Problem 1 issued that voters might have (and want to express) relatively complicated preferences, but the language for expressing these should at the same time not be too cognitive challenging. In Chapter 5 we certainly moved a big step forward to solving this problem. While bounded approval ballots are not fully expressive, they allow expressing the most common preference types in multiwinner voting, i.e., substitution effects, incompatibilities, dependencies, and regular approval. At the same time, they are not really cognitive challenging. In fact, approval ballots remain valid, so the lazy voter can just submit such a ballot. With the web-application presented in Chapter 5 it is even more convenient for the voters to make use of the most important features of bounded approval ballots.

Problem 2 concerns where which voting rules could be used, and where which properties are important. We leveraged knowledge regarding this question in Chapter 2 by showing that voting rules can be used as centrality indices and node selectors. Many properties of voting rules which are well studied in social choice literature find application, or have counterparts in network science. Some voting based centrality indices, or node selectors have desirable properties that common methods from network science fail. For example, Condorcet-consistency is currently only satisfied by the Copeland centrality index which we defined based on the same name voting rule. Further, from the experiments we also learned that the path of influence could also work in the other direction: VoteRank is closely related to the class of w-SPAV rules, but it performs better than those rules in our experiments on networks. Maybe results like this can also help to tweak multiwinner voting rules.

In Problem 3 we raised the question in how far preference aggregation is susceptible to fraud attempts by malicious agents, and how it can be protected. We extended the existing literature (mostly on single-winner rules) by showing the computational vulnerability of apportionment methods to bribery-like fraud in Chapter 4. We also addressed the second part of the question by proving that the second-chance mode (which requires only very little change in the procedure) protects apportionment elections from such fraud by making it computationally intractable to compute optimal actions.

Finally, in Problem 4 we pointed out that in real-world the perfect information required by many social choice procedures might be unavailable. We addressed this problem in Chapter 3 where we introduced a more realistic framework for participatory budgeting. In participatory budgeting projects have to be implemented, and it is well known that projects sometimes turn out to take longer, or cost more than expected. We showed that under these conditions there exists no ideal method. However, our best-effort methods achieve reasonably good results with the limited information they have.

This thesis aims at building a bridge from theory to practice. Although we were able to build this bridge to some degree in each chapter, it is still a long way before results from this thesis are widely used in real-world applications. We believe the most promising next step is to improve the *GoodVotes* application from Chapter 5. After all, a theoretical concept—even a really good one—will only be broadly accepted if it is easy to use for common people. By making concepts usable through web applications or apps, there are good chances that many people will use it. A good example are online poll services such as *Doodle*, *Bitpoll*, or *Nuudel*. Many people use such services just because they are very easy. But from time to time, everyone experiences the situation that one cannot really express what one really wants through such 'ballots'. We believe it is time to use COMSOC methods in real-world by making them as accessible for the voters as Doodle and others are.

Bibliography

[1] S. Arora and B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

[2] K. Arrow. *Social Choice and Individual Values*. Vol. 12. Yale university press, 2012.

[3] K. Arrow, A. Sen, and K. Suzumura, eds. *Handbook of Social Choice and Welfare*. Vol. 2. North-Holland, 2011.

[4] H. Aziz and N. Shah. "Participatory Budgeting: Models And Approaches". In: *Pathways Between Social Science and Computational Social Science: Theories, Methods, and Interpretations* (2021), pp. 215–236.

[5] H. Aziz and N. Shah. "Participatory Budgeting: Models and Approaches". In: *Pathways between Social Science and Computational Social Science: Theories, Methods and Interpretations*. Ed. by T. Rudas and P. Gábor. Springer, 2021, pp. 215–236.

[6] H. Aziz et al. "Computational Aspects of Multi-Winner Approval Voting". In: *Proceedings of the 14th international conference on autonomous agents and multiagent systems (AAMAS)*. IFAAMAS, 2015, 107–115.

[7] H. Aziz et al. "Justified Representation in Approval-Based Committee Voting". In: *Social Choice and Welfare* 48.2 (2017), pp. 461–485.

[8] J. Banks. "Sophisticated Voting Outcomes and Agenda Control". In: *Social Choice and Welfare* 1.4 (1985), pp. 295–306.

[9] N. Barrot and J. Lang. "Conditional and Sequential Approval Voting on Combinatorial Domains". In: *Proceedings of the 25th International Joint Conference on Artificial Intelligence*. 2016, pp. 88–94.

[10] D. Baumeister, L. Boes, and C. Laußmann. "Time-Constrained Participatory Budgeting Under Uncertain Project Costs". In: *International Joint Conferences on Artificial Intelligence*. 2022.

[11] D. Baumeister, L. Boes, and T. Seeger. "Irresolute Approval-based Budgeting". In: *Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems*. IFAAMAS, 2020, pp. 1774–1776.

[12] D. Baumeister, T. Hogrebe, and L. Rey. "Generalized Distance Bribery". In: AAAI Press, 2019, pp. 1764–1771.

[13] D. Baumeister and J. Rothe. "Preference Aggregation by Voting". In: *Economics and Computation. An Introduction to Algorithmic Game Theory, Computational Social Choice, and Fair Division*. Ed. by J. Rothe. Springer Texts in Business and Economics. Springer-Verlag, 2015. Chap. 4, pp. 197–325.

[14] D. Baumeister et al. “Designing Expressive Preferences in Multiwinner Voting”. In: *International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2023.

[15] P. Boldi et al. “Voting in Social Networks”. In: *Proceedings of the 18th ACM Conference on Information and Knowledge Management*. ACM Press, Nov. 2009, pp. 777–786.

[16] R. Booth et al. “Learning Conditionally Lexicographic Preference Relations”. In: *Proceedings of the 19th European Conference on Artificial Intelligence*. 2010, pp. 269–274.

[17] C. Boutilier et al. “Reasoning With Conditional Ceteris Paribus Preference Statements”. In: *Proceedings of the 15th Annual Conference on Uncertainty in Artificial Intelligence*. 1999, pp. 71–80.

[18] Craig Boutilier and Holger H. Hoos. “Bidding Languages for Combinatorial Auctions”. In: *Proceedings of the 17th International Joint Conference on Artificial Intelligence*. 2001, pp. 1211–1217.

[19] S. Brams, P. Edelman, and P. Fishburn. “Fair Division of Indivisible Items”. In: *Theory and Decision* 55 (2003), pp. 147–180.

[20] S. Brams and D. Kilgour. “Satisfaction approval voting”. In: *Voting Power and Procedures*. Springer, 2014, pp. 323–346.

[21] U. Brandes, C. Laußmann, and J. Rothe. “Voting for Centrality (Extended Abstract)”. In: *International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, 2022.

[22] F. Brandl and D. Peters. “An Axiomatic Characterization of The Borda Mean Rule”. In: *Social choice and welfare* 52.4 (2019), pp. 685–707.

[23] R. Bredereck et al. “Strategic Campaign Management in Apportionment Elections”. In: ijcai.org, July 2020, pp. 103–109.

[24] M. Brill. “Interactive Democracy”. In: *Proceedings of the 17th International Conference on Autonomous Agents and Multiagent Systems*. IFAAMAS, July 2018, pp. 1183–1187.

[25] M. Brill, J. Laslier, and P. Skowron. “Multiwinner Approval Rules as Apportionment Methods”. In: *Journal of Theoretical Politics*. AAAI Press, Feb. 2017, pp. 414–420.

[26] Y. Cabannes. “Participatory Budgeting: A Significant Contribution to Participatory Democracy”. In: *Environment and urbanization* 16.1 (2004), pp. 27–46.

[27] A. Copeland. *A reasonable social welfare function*. unpublished. 1951.

[28] E. Elkind et al. “Properties of Multiwinner Voting Rules”. In: *Social Choice and Welfare* 48.3 (2017), pp. 599–632.

[29] E. Elkind et al. “What Do Multiwinner Voting Rules Do? An Experiment over the Two-Dimensional Euclidean Domain”. In: *Proceedings of the 31st AAAI Conference on Artificial Intelligence*. AAAI Press, Feb. 2017, pp. 494–501.

[30] E. Everett and S. Borgatti. “Extending centrality”. In: *Models and Methods in Social Network Analysis* 35.1 (2005), pp. 57–76.

[31] M. Everett and S. Borgatti. “The Centrality of Groups and Classes”. In: *Journal of Mathematical Sociology* 23.3 (1999), pp. 181–201.

[32] R. Fairstein, R. Meir, and K. Gal. “Proportional Participatory Budgeting with Substitute Projects”. In: *arXiv preprint arXiv:2106.05360* (2021).

[33] P. Faliszewski, E. Hemaspaandra, and L. Hemaspaandra. “How Hard Is Bribery in Elections?” In: 35 (2009), pp. 485–532.

[34] P. Faliszewski, E. Hemaspaandra, and L. Hemaspaandra. “The Complexity of Manipulative Attacks in Nearly Single-peaked Electorates”. In: *Proceedings of the 13th conference on theoretical aspects of rationality and knowledge*. 2011, pp. 228–237.

[35] P. Faliszewski and J. Rothe. “Control and Bribery in Voting”. In: *Handbook of Computational Social Choice*. Ed. by F. Brandt et al. Cambridge University Press, 2016. Chap. 7, pp. 146–168.

[36] P. Faliszewski et al. “The Shield That Never Was: Societies With Single-peaked Preferences Are More Open to Manipulation And Control”. In: *Proceedings of the 12th Conference on Theoretical Aspects of Rationality and Knowledge*. 2009, pp. 118–127.

[37] A. Fiat and G. Woeginger. *Online Algorithms: The State of the Art*. Vol. 1442. Springer, 1998.

[38] S. Foldes and P. Hammer. “The Dilworth Number of a Graph”. In: *Annals of Discrete Mathematics* 2 (1978), pp. 211–219.

[39] L. Freeman. “Centrality in Social Networks: Conceptual Clarification”. In: *Social Networks* 1 (1979), pp. 215–239.

[40] D. Gale and L. Shapley. “College Admissions And The Stability of Marriage”. In: *The American Mathematical Monthly* 69.1 (1962), pp. 9–15.

[41] M. Gallagher and P. Mitchell, eds. *The Politics of Electoral Systems*. Oxford University Press, 2005.

[42] P. Gärdenfors. “Positionalist Voting Functions”. In: *Theory and Decision* 4.1 (1973), pp. 1–24.

[43] M. Goerigk et al. “The Robust Knapsack Problem with Queries”. In: *Computers & Operations Research* 55 (2015), pp. 12–22.

[44] J. Gomez, D. Insua, and C. Alfaro. “A Participatory Budget Model Under Uncertainty”. In: *European Journal of Operational Research* 249.1 (2016), pp. 351–358.

[45] U. Grandi. “Social choice and social networks”. In: *Trends in Computational Social Choice*. Ed. by Ulle Endriss. AI Access, 2017. Chap. 9, pp. 196–184.

[46] E. Güney. “A Mixed Integer Linear Program for Election Campaign Optimization under D’Hondt Rule”. In: *Proceedings of the Annual International Conference of the German Operations Research Society*. Springer, 2017, pp. 73–79.

[47] R. Irving. “An Efficient Algorithm For The ’Stable Roommates’ Problem”. In: *Journal of Algorithms* 6.4 (1985), pp. 577–595.

[48] P. Jain, K. Sornat, and Nimrod N. Talmon. “Participatory Budgeting with Project Interactions”. In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence*. 2020, pp. 386–392.

[49] O. Kariv and L. Hakimi. “An Algorithmic Approach to Network Location Problems. I: The p-centers”. In: *SIAM Journal on Applied Mathematics* 37.3 (1979), pp. 513–538.

[50] R. Karp. “Reducibility Among Combinatorial Problems”. In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. Plenum Press, 1972, pp. 85–103.

[51] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer, 2004.

[52] D. Kempe, J. Kleinberg, and É. Tardos. “Maximizing The Spread of Influence Through a Social Network”. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, pp. 137–146.

[53] M. Kilgour. “Multi-Winner Voting”. In: *Studies of Applied Economics* 36.1 (2018), pp. 167–180.

[54] D. Koschützki et al. “Centrality Indices”. In: *Network Analysis: Methodological Foundations*. Ed. by U. Brandes and T. Erlebach. Springer, 2005, pp. 16–61.

[55] M. Lackner and P. Skowron. *Multi-Winner Voting With Approval Preferences*. Springer Nature, 2023.

[56] E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. available at `https://archive.org/details/handbuchderlehre01landuoft/page/30/mode/2up`. Leipzig B.G. Teubner, 1909.

[57] J. Lang. *Coalition Formation Games Span All of Social Choice! Towards a taxonomy.* Talk at Dagstuhl Seminar 21331 on Coalition Formation Games. 2021.

[58] J. Laslier and M. Sanver, eds. *Handbook on Approval Voting*. Springer, 2010.

[59] C. Laußmann, J. Rothe, and T. Seeger. *Apportionment with Thresholds: Strategic Campaigns Are Easy in the Top-Choice But Hard in the Second-Chance Mode*. submitted to the 32nd International Joint Conference on Artificial Intelligence (IJCAI 2023) in January, 2023.

[60] J. Leskovec, L. Adamic, and B. Huberman. “The Dynamics of Viral Marketing”. In: *ACM Transactions on the Web (TWEB)* 1.1 (2007), 5–es.

[61] W. Ma et al. “Resource-Constrained Project Scheduling Problem With Uncertain Durations And Renewable Resources”. In: *International Journal of Machine Learning and Cybernetics* 7.4 (2016), pp. 613–621.

[62] J. Mercik. “Voting Procedures With A Priori Incomplete Individual Profiles”. In: *Multiperson decision making models using fuzzy sets and possibility theory* (1990), pp. 242–251.

[63] S. Milgram. “The Small World Problem”. In: *Psychology today* 2.1 (1967), pp. 60–67.

[64] M. Monaci and U. Pferschy. “On the Robust Knapsack Problem”. In: *SIAM Journal on Optimization* 23.4 (2013), pp. 1956–1982.

[65] M. Monaci, U. Pferschy, and P. Serafini. “Exact Solution of the Robust Knapsack Problem”. In: *Computers & Operations Research* 40.11 (2013), pp. 2625–2631.

[66] H. Moradi and S. Shadrokh. “A Robust Scheduling For The Multi-Mode Project Scheduling Problem With a Given Deadline Under Uncertainty of Activity Duration”. In: *International Journal of Production Research* 57.10 (2019), pp. 3138–3167.

[67] J. Nash. *John Nash in a letter to the NSA*. `https://www.nsa.gov/portals/75/documents/news-features/declassified-documents/nash-letters/nash_letters1.pdf`. 1955.

[68] M. Pellicer and E. Wegner. “The Mechanical And Psychological Effects of Legal Thresholds”. In: *Electoral Studies* 33 (2014), pp. 258–266.

[69] D. Peters, G. Pierczyński, and P. Skowron. “Proportional Participatory Budgeting With Additive Utilities”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 12726–12737.

[70] A. Procaccia and J. Rosenschein. “The Distortion of Cardinal Preferences in Voting”. In: *Cooperative Information Agents X: 10th International Workshop*. Springer. 2006, pp. 317–331.

[71] F. Pukelsheim. *Proportional Representation*. Springer, 2017.

[72] M. Rabin and D. Scott. “Finite Automata and Their Decision Problems”. In: *IBM journal of research and development* 3.2 (1959), pp. 114–125.

[73] S. Rey, U. Endriss, and R. de Haan. “Designing Participatory Budgeting Mechanisms Grounded in Judgment Aggregation”. In: *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*. Vol. 17. 1. 2020, pp. 692–702.

[74] J. Rothe. *Complexity Theory and Cryptology*. Springer, 2005.

[75] J. Rothe. *Informatik IV Theoretische Informatik*. available at `https://ccc.cs.uni-duesseldorf.de/~rothe/INFO4/main.pdf`. lecture notes. 2019.

[76] J. Rothe. *Komplexitätstheorie und Kryptologie*. Springer, 2008.

[77] J. Rothe et al. *Economics and Computation*. Ed. by J. Rothe. Vol. 4. Springer, 2015.

[78] T. Sandholm. “Algorithm for Optimal Winner Determination in Combinatorial Auctions”. In: *Artificial Intelligence* 135 (2002), pp. 1–54.

[79] D. Schoch and U. Brandes. “Re-Conceptualizing Centrality in Social Networks”. In: *European Journal of Applied Mathematics* 27.6 (2016), pp. 971–985.

[80] M. Schulze. “A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method”. In: *Social Choice and Welfare* 36.2 (2011), pp. 267–303.

[81] P. Skowron and P. Faliszewski. “Chamberlin–Courant Rule with Approval Ballots: Approximating the MaxCover Problem with Bounded Frequencies in FPT Time”. In: *Journal of Artificial Intelligence Research* 60 (2017), pp. 687–716.

[82] P. Slater. “Inconsistencies in a Schedule of Paired Comparisons”. In: *Biometrika* 48.3–4 (1961), pp. 303–312.

[83] D. Stolicki, S. Szufa, and N. Talmon. “Pabulib: A Participatory Budgeting Library”. In: *arXiv preprint arXiv:2012.06539* (2020).

[84] N. Talmon and P. Faliszewski. “A Framework for Approval-based Budgeting Methods”. In: *Proceedings of the 33rd AAAI Conference on Artificial Intelligence*. Vol. 33. AAAI Press, 2019, pp. 2181–2188.

[85] A. Turing. “On Computable Numbers, With an Application to The Entscheidungsproblem”. In: *J. of Math* 58.345-363 (1936), p. 5.

[86] K. Vaziri, P. Carr, and L. Nozick. “Project Planning for Construction Under Uncertainty with Limited Resources”. In: *Journal of Construction Engineering and Management* 133.4 (2007), pp. 268–276.

[87] D. Watts and S. Strogatz. “Collective Dynamics of ’Small-World’ Networks”. In: *nature* 393.6684 (1998), pp. 440–442.

[88] K. Wilkinson. *Analysis of a Voting Method for Ranking Network Centrality Measures on a Node-Aligned Multiplex Network*. Tech. rep. AFIT-ENS-MS-18-M-170. Wright-Patterson Air Force Base, Ohio, USA: Air Force Institute of Technology, 2018.

[89] J. Zhang et al. “Identifying a Set of Influential Spreaders in Complex Networks”. In: *Scientific Reports* 6 (2016).

[90] W. Zwicker. “Introduction to the Theory of Voting”. In: *Handbook of Computational Social Choice*. Ed. by F. Brandt et al. Cambridge University Press, 2016. Chap. 2, pp. 23–56.

Appendix

1 Code of Centrality Experiments

We here provide the code for the experiments in Section 2.7.

Listing 6.1: **Centrality.py — Implementations of Centrality Indices, and Node Selectors**

```
from abc import ABC, abstractmethod
from random import shuffle, sample, random, choice
import networkx as nx
import itertools


class OrdinalPreference(list):
    def __init__(self, owner, iterable_pref = None):
        self.owner = owner
        if iterable_pref != None:
            super().__init__(iterable_pref)

    def remove_candidate(self, candidate):
        for pos in self:
            if candidate in pos:
                pos.remove(candidate)
                if len(pos) == 0:
                    self.remove(pos)
                break

    def get_predecessors_of(self, candidate):
        predecessors = set()
        for pos in self:
            if candidate in pos:
                return predecessors
            else:
                predecessors = predecessors.union(pos)
        raise ValueError("Candidate not in preference: %s" % str(candidate))

    def get_successors_of(self, candidate):
        successors = set()
        found = False
        for pos in self:
            if found:
                successors = successors.union(pos)
            if candidate in pos:
                found = True
        if not found:
            raise ValueError("Candidate not in preference: %s" % str(candidate))
        return successors

    def compare(self, a, b):
        if self.owner == a or self.owner == b:
            return 0
        for pos in self:
            if (a in pos) and (b not in pos):
                return 1
            if (a in pos) and (b in pos):
                return 0
            if (a not in pos) and (b in pos):
                return -1

    def __eq__(self, other):
        if type(other) != type(self):
            return False
        if other.owner != self.owner:
            return False
        return super().__eq__(other)

    def __setitem__(self, pos, nodes):
        if type(nodes) == set:
            super().__setitem__(pos, nodes)
        else:
            raise ValueError("Must be a set: %s" % str(nodes))
    def append(self, nodes):
        if type(nodes) == set:
            super().append(nodes)
        else:
            raise ValueError("Must be a set: %s" % str(nodes))


class ApprovalPreference(set):
    def __init__(self, owner, iterable_pref = None):
        self.owner = owner
        if iterable_pref != None:
            super().__init__(iterable_pref)

    def __eq__(self, other):
        if type(other) != type(self):
            return False
        if other.owner != self.owner:
            return False
        return super().__eq__(other)



def get_ordinal_preferences_from_graph(graph):
    ret = list()
    shortest_paths = nx.shortest_path_length(graph, source=None, target=None)
    for start, distances in shortest_paths:
        nodes_remaining = set(graph.nodes)
        nodes_remaining.remove(start)
        max_distance = max(distances.values())
        pref = OrdinalPreference(start)
        ret.append(pref)
        for d in range(1, max_distance + 1):
            nodes_at_pos = {node for node in distances if distances[node] == d}
            pref.append(nodes_at_pos)
            nodes_remaining = nodes_remaining.difference(nodes_at_pos)
        if len(nodes_remaining) > 0:
            pref.append(nodes_remaining)
    return ret

def get_approval_preferences_from_graph(graph):
    ret = list()
    for node in graph.nodes:
        ret.append(ApprovalPreference(node, graph.neighbors(node)))
    return ret


############################################################
############################################################
############################################################

class NodeSelector(ABC):
    def __init__(self, graph):
        self.graph = graph

    @abstractmethod
    def select_committee(self, number_of_nodes):
        pass

class CentralityIndex(NodeSelector, ABC):
    def __init__(self, graph):
        self.indices = self.__get_indices__(graph)
        super().__init__(graph)

    def get_winners(self):
        max_val = max(self.indices.values())
        winners = [n for n in self.indices.keys() if self.indices[n] == max_val]
        return winners

    def select_committee(self, number):
        ret = set()
        nodes_left = list(self.indices.keys())
        for x in range(number):
            max_key = max(nodes_left, key=self.indices.get)
            max_val = self.indices[max_key]
            max_keys = [n for n in nodes_left if self.indices[n] == max_val]
            select = choice(max_keys)
            ret.add(select)
            nodes_left.remove(select)
        return ret

    @abstractmethod
    def __get_indices__(self, graph):
        pass

class GroupCentralityIndex(NodeSelector, ABC):
    def __init__(self, graph):
        super().__init__(graph)

    def select_committee(self, number):
        ret = set()
        best = 0
        for com in itertools.combinations(self.graph.nodes, number):
            index = self.get_committee_index(self.graph, com)
            if index > best:
                ret = set(com)
                best = index
        return ret

    @abstractmethod
    def get_committee_index(self, graph, committee):
        pass

#-----------------------------------------------------------------
#                    Voting Based Centrality
#-----------------------------------------------------------------

class VoteRank_Centrality(NodeSelector):
    name = "VoteRank"

    # NetworkX provides also an implementation. But here we use random tie-breaking.
    def select_committee(self, x):
        reduction_constant = self.__average_degree__()
        voting_ability = {n:1 for n in self.graph.nodes}
        selected = set()
        for i in range(x):
            scores = {n:0 for n in self.graph.nodes}
            for n in self.graph.nodes:
                if n not in selected:
                    scores[n] = sum([voting_ability[neigh] for neigh in self.graph.neighbors(n)])
            max_val = max(scores.values())
            max_keys = [n for n in self.graph.nodes if scores[n] == max_val and n not in selected]
            select = choice(max_keys)
            selected.add(select)
            voting_ability[select] = 0
            for neighbor in self.graph.neighbors(select):
                voting_ability[neighbor] -= reduction_constant
                if voting_ability[neighbor] < 0:
                    voting_ability[neighbor] = 0

        return selected

    def __average_degree__(self):
        sum = 0
        for n in self.graph.nodes:
            sum += len(list(self.graph.neighbors(n)))
        return sum / len(self.graph.nodes)


class STV_Centrality(NodeSelector):
    name = "STV"

    def select_committee(self, x):
        if x > len(self.graph.nodes):
            x = len(self.graph.nodes)
        pref_rem = get_ordinal_preferences_from_graph(self.graph)
        can_rem = list(self.graph.nodes)
        winners = set()
        while len(winners) < x:
            q = len(pref_rem)//(x+1)
            best_c, best_score, worst_c, worst_score = self.__plurality_win_los__(pref_rem, can_rem)
            if best_score >= q:
                winners.add(best_c)
                self.__remove_q_preferences_voted_for_c__(pref_rem, q, best_c)
                self.__remove_preference_of__(pref_rem, best_c)
                self.__remove_c_from_all_preferences__(pref_rem, best_c)
                can_rem.remove(best_c)
            else:
                can_rem.remove(worst_c)
                self.__remove_c_from_all_preferences__(pref_rem, worst_c)
        return winners

    @staticmethod
    def __plurality_win_los__(preferences, candidates):
        scores = dict()
        for c in candidates:
            scores[c] = 0
        for pref in preferences:
            for n in pref[0]:
                scores[n] += 1
        best_score = max(scores.values())
        winner = choice([n for n in scores if scores[n] == best_score])
        worst_score = min(scores.values())
        loser = choice([n for n in scores if scores[n] == worst_score])
        return (winner, scores[winner], loser, scores[loser])

    @staticmethod
    def __remove_q_preferences_voted_for_c__(preferences, q, c):
        voted_for = [p for p in preferences if c in p[0]]
        rem = sample(voted_for, q)
        for i in rem:
            preferences.remove(i)

    @staticmethod
    def __remove_preference_of__(preferences, c):
        for pref in preferences:
            if pref.owner == c:
                preferences.remove(pref)
                break

    @staticmethod
    def __remove_c_from_all_preferences__(preferences, c):
        for pref in preferences:
            pref.remove_candidate(c)


class SPAV_Centrality(NodeSelector):
    name = "Seq-PAV"

    def select_committee(self, x):
        satisfaction = dict()
        for node in self.graph.nodes:
            satisfaction[node] = 1
        candidates = list(self.graph.nodes)
        shuffle(candidates)
        selected_set = set()
        while len(selected_set) < x:
            best_node = -1
            best_score = -1
            for c in candidates:
                score = 0
                for v in self.graph.neighbors(c):
                    score += 1/satisfaction[v]
                if score > best_score:
                    best_score = score
                    best_node = c
            selected_set.add(best_node)
            for neighbor in self.graph.neighbors(best_node):
                satisfaction[neighbor] += 1
            candidates.remove(best_node)
        return selected_set


class GreedyAV_Centrality(NodeSelector):
    name = "Greedy-AV"

    def select_committee(self, x):
        satisfaction = dict()
        for node in self.graph.nodes:
            satisfaction[node] = 1
        candidates = list(self.graph.nodes)
        contribution = {v:1 for v in self.graph.nodes}
        shuffle(candidates)
        selected_set = set()
        while len(selected_set) < x:
            scores = {c:sum(contribution[v] for v in self.graph.neighbors(c)) for c in candidates}
            best_node = max(candidates, key= lambda x:scores[x])
            selected_set.add(best_node)
            for neighbor in self.graph.neighbors(best_node):
                contribution[neighbor] = 0
            candidates.remove(best_node)
        return selected_set

#-----------------------------------------------------------------
#                    STANDARD Centrality
#-----------------------------------------------------------------

class Degree_Centrality(CentralityIndex):
    name = "Degree"

    def __get_indices__(self, graph):
        if isinstance(graph, nx.DiGraph):
            return nx.in_degree_centrality(graph)
        return nx.degree_centrality(graph)

class Group_Degree_Centrality(GroupCentralityIndex):
    name = "Group Degree"

    def get_committee_index(self, graph, committee):
        return nx.group_degree_centrality(graph, committee)

class Closeness_Centrality(CentralityIndex):
    name = "Closeness"

    def __get_indices__(self, graph):
        return nx.closeness_centrality(graph)

class Group_Closeness_Centrality(GroupCentralityIndex):
    name = "Group Closeness"
    def get_committee_index(self, graph, committee):
        return nx.group_closeness_centrality(graph, committee)
```

Listing 6.2: **Experiment1.py — Implementation of First Experiment**

```
from Centrality import *
import networkx as nx
import random
import time


##########
n = 60
samples = 200
budgets = (2,3,4)
##########

def main():
    node_selectors = (VoteRank_Centrality, STV_Centrality, GreedyAV_Centrality,
                      SPAV_Centrality, Degree_Centrality, Closeness_Centrality)
    graphs = generate_samples()
    experiment(graphs, node_selectors)

def generate_samples():
    graphs = list()
    graphtype = nx.watts_strogatz_graph
    k = 6
    p = 0.05
    print("Using %s() for sample generation"%str(graphtype))
    print("Parameters: n = %d  k = %d  p = %.2f"%(n,k,p))
    for s in range(samples):
        graph = graphtype(n, k, p)
        while not nx.is_connected(graph):
            graph = graphtype(n, k, p)
        graphs.append(graph)
    return graphs

def experiment(graphs, selectors):
    print("\033[92mHow Good is Selected Set in Average Distance/Neighborhood Size? \033[0m")

    def __set_distance_avg__(committee, graph):
        total = 0
        for node in graph.nodes-committee:
            distances = [nx.shortest_path_length(graph, source=node, target=x) for x in committee]
            total += min(distances)
        return total / (len(graph.nodes) - len(committee))

    def __neighborhood_size__(committee, graph):
        total = 0
        for node in graph.nodes-committee:
            distances = [nx.shortest_path_length(graph, source=node, target=x) for x in committee]
            if min(distances) == 1:
                total += 1
        return total

    for budget in budgets:
        print("\033[94mSet Size:\033[0m  \033[1m %d\033[0m"%(budget))
        results_dist = {method : list() for method in selectors}
        results_deg = {method : list() for method in selectors}
        i = 0

        for graph in graphs:
            start = time.time()

            selected_cl = Group_Closeness_Centrality(graph).select_committee(budget)
            assert len(selected_cl) == budget
            opt_dist = __set_distance_avg__(selected_cl, graph)

            selected_deg = Group_Degree_Centrality(graph).select_committee(budget)
            assert len(selected_deg) == budget
            opt_deg = __neighborhood_size__(selected_deg, graph)

            ex_time = time.time() - start

            for method in selectors:
                selected = method(graph).select_committee(budget)
                assert len(selected) == budget
                results_dist[method].append(__set_distance_avg__(selected, graph) - opt_dist)
                results_deg[method].append(opt_deg - __neighborhood_size__(selected, graph))

            print("est. time: %.1f minutes"%((samples-i)*ex_time/60), end="\r", flush=True)
            i += 1
        print("                              ")
        print(" "*18 + "Distance (diff to opt)        Degree (diff to opt)")
        print(" "*18 + "  lq    avg    uq    max         lq    avg    uq    max")
        for method in selectors:
            results_dist_current = sorted(results_dist[method])
            results_deg_current = sorted(results_deg[method])
            dist_lq = results_dist_current[int(samples/4)]
            dist_uq = results_dist_current[int(3*samples/4)]
            dist_max = results_dist_current[-1]
            dist_avg = sum(results_dist_current)/samples
            deg_lq = results_deg_current[int(samples/4)]
            deg_uq = results_deg_current[int(3*samples/4)]
            deg_max = results_deg_current[-1]
            deg_avg = sum(results_deg_current)/samples
            print("\033[95m%s:"%method.name +
                " "*(18-len(method.name)) +
                "\033[1m %.2f   %.2f   %.2f   %.2f           %d    %.2f    %d   %d\033[0m"
                % (dist_lq, dist_avg, dist_uq, dist_max, deg_lq, deg_avg, deg_uq, deg_max))
        print()
        print()
############################################################
main()
```

Listing 6.3: **Experiment2.py — Implementation of Second Experiment**

```
from Centrality import *
import networkx as nx
import random
import time


##########
n = 500
samples = 200
budgets = (2,3,4,10)
##########

def main():
    node_selectors = (VoteRank_Centrality, STV_Centrality,
                      GreedyAV_Centrality, SPAV_Centrality)
    graphs = generate_samples()
    experiment(graphs, node_selectors)

def generate_samples():
    graphs = list()
    graphtype = nx.watts_strogatz_graph
    k = 6
    p = 0.05
    print("Using %s() for sample generation"%str(graphtype))
    print("Parameters: n = %d  k = %d  p = %.2f"%(n,k,p))
    for s in range(samples):
        graph = graphtype(n, k, p)
        while not nx.is_connected(graph):
            graph = graphtype(n, k, p)
        graphs.append(graph)
    return graphs

def experiment(graphs, selectors):
    print("\033[92mHow Often is Selected the Best in Average Distance/Neighborhood Size?\033[0m")

    def __set_distance_avg__(committee, graph):
        total = 0
        for node in graph.nodes-committee:
            distances = [nx.shortest_path_length(graph, source=node, target=x) for x in committee]
            total += min(distances)
        return total / (len(graph.nodes) - len(committee))

    def __neighborhood_size__(committee, graph):
        total = 0
        for node in graph.nodes-committee:
            distances = [nx.shortest_path_length(graph, source=node, target=x) for x in committee]
            if min(distances) == 1:
                total += 1
        return total

    for budget in budgets:
        print("\033[94mSet Size:\033[0m  \033[1m %d\033[0m"%(budget))
        results_dist = {method : 0 for method in selectors}
        results_deg = {method : 0 for method in selectors}
        i = 0

        for graph in graphs:
            start = time.time()
            best_dist = list()
            best_deg = list()
            best_dist_val = n
            best_deg_val = 0

            for method in selectors:
                selected = method(graph).select_committee(budget)
                assert len(selected) == budget
                dist = __set_distance_avg__(selected, graph)
                deg = __neighborhood_size__(selected, graph)
                if dist < best_dist_val:
                    best_dist = list()
                    best_dist.append(method)
                    best_dist_val = dist
                elif dist == best_dist_val:
                    best_dist.append(method)
                if deg > best_deg_val:
                    best_deg = list()
                    best_deg.append(method)
                    best_deg_val = deg
                elif deg == best_deg_val:
                    best_deg.append(method)

            for method in best_dist:
                results_dist[method] += 1
            for method in best_deg:
                results_deg[method] += 1

            ex_time = time.time() - start
            i += 1
            print("est. time: %.1f minutes"%((samples-i)*ex_time/60), end="\r", flush=True)

        print("                                    ")
        print(" "*18 + "Best Distance in X Cases           Best Degree in X Cases")
        for method in selectors:
            print("\033[95m%s:"%method.name +
                " "*(18-len(method.name)) +
                "\033[1m %.1f %%                          %.1f %% \033[0m" %
                (100*results_dist[method]/samples, 100*results_deg[method]/samples))
        print()
        print()
############################################################
main()
```

2 Code of Best-Effort Experiments

We here provide the code for the experiments in Section 3.6.1.

Listing 6.4: **LimitationVarible.py — Limitation is Variable**

```
#!/usr/bin/env pypy3
from Core import *
from Shared import *
import sys
import random
from pathlib import Path

print("Experimenting with variable Limitation")

# usage: ./LimitationVariable.py DATA SAMPLES TLIMIT ALPHA UNCERTAINTY={NONE, LOW, MEDIUM, HIGH}
pabulib_file = sys.argv[1]
sample_count = int(sys.argv[2])
tlimit = int(sys.argv[3])
UNCERTAINTY = sys.argv[5]
alpha_risk = float(sys.argv[4])

dataset_name = pabulib_file.rsplit("/", 1)[1].replace(".pb", "")
Path("Results/BEE/%s/%s"%(dataset_name, UNCERTAINTY)).mkdir(parents=True, exist_ok=True)
Path("Results/BEP/%s/%s"%(dataset_name, UNCERTAINTY)).mkdir(parents=True, exist_ok=True)
out_file_sat_bee = open("Results/BEE/%s/%s/LimitSat.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_sat_bep = open("Results/BEP/%s/%s/LimitSat.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_ex_bee = open("Results/BEE/%s/%s/LimitEx.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_pu_bep = open("Results/BEP/%s/%s/LimitPu.dat"%(dataset_name, UNCERTAINTY), "w")
print(pabulib_file, file=out_file_sat_bee)
print(pabulib_file, file=out_file_sat_bep)
print(pabulib_file, file=out_file_ex_bee)
print(pabulib_file, file=out_file_pu_bep)
print("tlimit: %d   alpha: %.2f   MIN_DURATION: %d MAX_DURATION: %d    samples: %d" %
    (tlimit, alpha_risk, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_sat_bee)
print("tlimit: %d   alpha: %.2f   MIN_DURATION: %d MAX_DURATION: %d    samples: %d" %
    (tlimit, alpha_risk, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_sat_bep)
print("tlimit: %d   alpha: %.2f   MIN_DURATION: %d MAX_DURATION: %d    samples: %d" %
    (tlimit, alpha_risk, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_ex_bee)
print("tlimit: %d   alpha: %.2f   MIN_DURATION: %d MAX_DURATION: %d    samples: %d" %
    (tlimit, alpha_risk, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_pu_bep)

blimit, projects, votes = read_scenario(pabulib_file)
utilities = {p : get_util_satisfaction(votes, p) for p in projects}
limitations = [1 + 0.05*i for i in range(21)]
```

```
#####################################
# Now follows the actual experiment
#####################################

# Finding optimal solutions.
opt = dict()
for k_lim in limitations:
    select_no_surplus = optimal_planning(projects, tlimit, blimit, utilities)
    select_with_surplus = optimal_planning(projects, tlimit, blimit * k_lim, utilities)
    opt[k_lim] = (1-alpha_risk) * sum(utilities[p] for p in select_no_surplus)
    opt[k_lim] += alpha_risk * sum(utilities[p] for p in select_with_surplus)

ratios_bee = {k_lim : [] for k_lim in limitations}
ratios_bep = {k_lim : [] for k_lim in limitations}
ex_val = {k_lim : [] for k_lim in limitations}
pu_val = {k_lim : [] for k_lim in limitations}

for s in range(sample_count):
    print("Sample %d of %d" % (s+1, sample_count), end='\r')
    resample_uncertainty(projects, UNCERTAINTY)

    for k_lim in limitations:
        # Best effort exhaustiveness
        log = BEE(projects, tlimit, blimit, k_lim, alpha_risk, utilities)
        selection = log.get_projects()
        util_online = sum(utilities[p] for p in selection)

        ratios_bee[k_lim].append(satisfaction_perf(opt[k_lim], util_online))
        ex_val[k_lim].append(exhaustiveness_perf(projects, selection, tlimit, blimit, utilities))

        # Best effort punctuality
        log = BEP(projects, tlimit, blimit, k_lim, alpha_risk, utilities, BEE_log = log)
        selection = log.get_projects()
        util_online = sum(utilities[p] for p in selection)

        ratios_bep[k_lim].append(satisfaction_perf(opt[k_lim], util_online))
        pu_val[k_lim].append(punctuality_perf(log, tlimit))
print()

#####################################################
# This is only for correctly formatted output to file
#####################################################

print("klimit best lq avg uq worst", file=out_file_sat_bee)
print("klimit best lq avg uq worst", file=out_file_sat_bep)
print("klimit best lq avg uq worst", file=out_file_ex_bee)
print("klimit best lq avg uq worst", file=out_file_pu_bep)

for k_lim in limitations:
    # Best effort exhaustiveness
    current_ratios_bee = sorted(ratios_bee[k_lim])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(k_lim, min(current_ratios_bee),
        current_ratios_bee[sample_count//4], sum(current_ratios_bee)/sample_count,
        current_ratios_bee[3*sample_count//4], max(current_ratios_bee)), file=out_file_sat_bee)
    current_exhaustiv = sorted(ex_val[k_lim])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(k_lim, min(current_exhaustiv),
        current_exhaustiv[sample_count//4], sum(current_exhaustiv)/sample_count,
        current_exhaustiv[3*sample_count//4], max(current_exhaustiv)), file=out_file_ex_bee)

    # Best effort punctuality
    current_ratios_bep = sorted(ratios_bep[k_lim])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(k_lim, min(current_ratios_bep),
        current_ratios_bep[sample_count//4], sum(current_ratios_bep)/sample_count,
        current_ratios_bep[3*sample_count//4], max(current_ratios_bep)), file=out_file_sat_bep)
    current_punctuality = sorted(pu_val[k_lim])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(k_lim, min(current_punctuality),
        current_punctuality[sample_count//4], sum(current_punctuality)/sample_count,
        current_punctuality[3*sample_count//4], max(current_punctuality)), file=out_file_pu_bep)

out_file_sat_bee.close()
out_file_sat_bep.close()
out_file_ex_bee.close()
out_file_pu_bep.close()
```

Listing 6.5: **RiskVarible.py — Risk Assessment is Variable**

```
#!/usr/bin/env pypy3
from Core import *
from Shared import *
import sys
import random
from pathlib import Path

print("Experimenting with variable Risk Assessment")

# usage: ./RiskVariable.py DATA SAMPLES KAPPA TLIMIT UNCERTAINTY={NONE, LOW, MEDIUM, HIGH}
pabulib_file = sys.argv[1]
sample_count = int(sys.argv[2])
```

```
k_limitation = float(sys.argv[3])
UNCERTAINTY = sys.argv[5]
tlimit = int(sys.argv[4])

dataset_name = pabulib_file.rsplit("/", 1)[1].replace(".pb", "")
Path("Results/BEE/%s/%s"%(dataset_name, UNCERTAINTY)).mkdir(parents=True, exist_ok=True)
Path("Results/BEP/%s/%s"%(dataset_name, UNCERTAINTY)).mkdir(parents=True, exist_ok=True)
out_file_sat_bee = open("Results/BEE/%s/%s/RiskSat.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_sat_bep = open("Results/BEP/%s/%s/RiskSat.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_ex_bee = open("Results/BEE/%s/%s/RiskEx.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_pu_bep = open("Results/BEP/%s/%s/RiskPu.dat"%(dataset_name, UNCERTAINTY), "w")
print(pabulib_file, file=out_file_sat_bee)
print(pabulib_file, file=out_file_sat_bep)
print(pabulib_file, file=out_file_ex_bee)
print(pabulib_file, file=out_file_pu_bep)
print("tlimit: %d   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (tlimit, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_sat_bee)
print("tlimit: %d   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (tlimit, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_sat_bep)
print("tlimit: %d   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (tlimit, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_ex_bee)
print("tlimit: %d   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (tlimit, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_pu_bep)


blimit, projects, votes = read_scenario(pabulib_file)
utilities = {p : get_util_satisfaction(votes, p) for p in projects}
alphas = [0.015*i for i in range(31)]


####################################
# Now follows the actual experiment
####################################

# Compute optimal solutions for each alpha
opt = dict()
select_no_surplus = optimal_planning(projects, tlimit, blimit, utilities)
select_with_surplus = optimal_planning(projects, tlimit, blimit * k_limitation, utilities)
for alpha_risk in alphas:
    opt[alpha_risk] = (1-alpha_risk) * sum(utilities[p] for p in select_no_surplus)
    opt[alpha_risk] += alpha_risk * sum(utilities[p] for p in select_with_surplus)

ratios_bee = {alpha_risk : [] for alpha_risk in alphas}
ratios_bep = {alpha_risk : [] for alpha_risk in alphas}
ex_val = {alpha_risk : [] for alpha_risk in alphas}
pu_val = {alpha_risk : [] for alpha_risk in alphas}

for s in range(sample_count):
    print("Sample %d of %d" % (s+1, sample_count), end='\r')
    resample_uncertainty(projects, UNCERTAINTY)

    for alpha_risk in alphas:
        # Best effort exhaustiveness
        log = BEE(projects, tlimit, blimit, k_limitation, alpha_risk, utilities)
        selection = log.get_projects()
        util_online = sum(utilities[p] for p in selection)

        ratios_bee[alpha_risk].append(satisfaction_perf(opt[alpha_risk], util_online))
        ex_val[alpha_risk].append(exhaustiveness_perf(projects, selection, tlimit, blimit, utilities))

        # Best effort punctuality
        log = BEP(projects, tlimit, blimit, k_limitation, alpha_risk, utilities, BEE_log = log)
        selection = log.get_projects()
        util_online = sum(utilities[p] for p in selection)

        ratios_bep[alpha_risk].append(satisfaction_perf(opt[alpha_risk], util_online))
        pu_val[alpha_risk].append(punctuality_perf(log, tlimit))
print()

#####################################################
# This is only for correctly formatted output to file
#####################################################

print("alpha best lq avg uq worst", file=out_file_sat_bee)
print("alpha best lq avg uq worst", file=out_file_sat_bep)
print("alpha best lq avg uq worst", file=out_file_ex_bee)
print("alpha best lq avg uq worst", file=out_file_pu_bep)

for alpha_risk in alphas:
    # Best effort exhaustiveness
    current_ratios_bee = sorted(ratios_bee[alpha_risk])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(alpha_risk, min(current_ratios_bee),
        current_ratios_bee[sample_count//4], sum(current_ratios_bee)/sample_count,
        current_ratios_bee[3*sample_count//4], max(current_ratios_bee)), file=out_file_sat_bee)
    current_exhaustiv = sorted(ex_val[alpha_risk])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(alpha_risk, min(current_exhaustiv),
        current_exhaustiv[sample_count//4], sum(current_exhaustiv)/sample_count,
        current_exhaustiv[3*sample_count//4], max(current_exhaustiv)), file=out_file_ex_bee)
    # Best effort exhaustiveness
    current_ratios_bep = sorted(ratios_bep[alpha_risk])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(alpha_risk, min(current_ratios_bee),
        current_ratios_bee[sample_count//4], sum(current_ratios_bee)/sample_count,
        current_ratios_bee[3*sample_count//4], max(current_ratios_bee)), file=out_file_sat_bep)
    current_punct = sorted(pu_val[alpha_risk])
    print("%.3f %.3f %.3f %.3f %.3f %.3f" %(alpha_risk, min(current_punct),
        current_punct[sample_count//4], sum(current_punct)/sample_count,
        current_punct[3*sample_count//4], max(current_punct)), file=out_file_pu_bep)

out_file_sat_bee.close()
out_file_sat_bep.close()
out_file_ex_bee.close()
out_file_pu_bep.close()
```

Listing 6.6: **TlimitVarible.py — Time Limit is Variable**

```
#!/usr/bin/env pypy3
from Core import *
from Shared import *
import sys
from pathlib import Path
import random

print("Experimenting with variable Time Limit")

# usage: ./TlimitVariable.py DATA SAMPLES KAPPA ALPHA UNCERTAINTY={NONE, LOW, MEDIUM, HIGH}
pabulib_file = sys.argv[1]
sample_count = int(sys.argv[2])
k_limitation = float(sys.argv[3])
UNCERTAINTY = sys.argv[5]
alpha_risk = float(sys.argv[4])

dataset_name = pabulib_file.rsplit("/", 1)[1].replace(".pb", "")
Path("Results/BEE/%s/%s"%(dataset_name, UNCERTAINTY)).mkdir(parents=True, exist_ok=True)
out_file_sat = open("Results/BEE/%s/%s/TimeLimitSat.dat"%(dataset_name, UNCERTAINTY), "w")
out_file_ex = open("Results/BEE/%s/%s/TimeLimitEx.dat"%(dataset_name, UNCERTAINTY), "w")
print(pabulib_file, file=out_file_sat)
print(pabulib_file, file=out_file_ex)
print("alpha: %.2f   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (alpha_risk, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_sat)
print("alpha: %.2f   limit: %.2f   MIN_DURATION: %d MAX_DURATION: %d   samples: %d" %
    (alpha_risk, k_limitation, MIN_DURATION, MAX_DURATION, sample_count), file=out_file_ex)


blimit, projects, votes = read_scenario(pabulib_file)
utilities = {p : get_util_satisfaction(votes, p) for p in projects}
max_project_length = max(p.duration for p in projects)
max_time = 35 # this is the maximum time in the experiment

####################################
# Now follows the actual experiment
####################################

# Finding optimal solutions.
select_no_surplus = optimal_planning(projects, max_project_length, blimit, utilities)
select_with_surplus = optimal_planning(projects, max_project_length, blimit * k_limitation, utilities)
opt = (1-alpha_risk) * sum(utilities[p] for p in select_no_surplus)
opt += alpha_risk * sum(utilities[p] for p in select_with_surplus)

ratios = {i : [] for i in range(max_project_length, max_time+1)}
ev_val = {i : [] for i in range(max_project_length, max_time+1)}

for s in range(sample_count):
    print("Sample %d of %d" % (s+1, sample_count), end='\r')
    resample_uncertainty(projects, UNCERTAINTY)

    for tlimit in range(max_project_length, max_time+1):
        budgeting_log = BEE(projects, tlimit, blimit, k_limitation, alpha_risk, utilities)
        selection = budgeting_log.get_projects()
        util_online = sum(utilities[p] for p in selection)

        ratios[tlimit].append(satisfaction_perf(opt, util_online))
        ev_val[tlimit].append(exhaustiveness_perf(projects, selection, tlimit, blimit, utilities))
print()

#####################################################
# This is only for correctly formatted output to file
#####################################################

print("tlimit best lq avg uq worst", file=out_file_sat)
print("tlimit best lq avg uq worst", file=out_file_ex)

for tlimit in range(max_project_length, max_time+1):
    current_ratios = sorted(ratios[tlimit])
    print("%d %.3f %.3f %.3f %.3f %.3f" %(tlimit, min(current_ratios),
        current_ratios[sample_count//4], sum(current_ratios)/sample_count,
        current_ratios[3*sample_count//4], max(current_ratios)), file=out_file_sat)
    current_exhaustiv = sorted(ev_val[tlimit])
    print("%d %.3f %.3f %.3f %.3f %.3f" %(tlimit, min(current_exhaustiv),
        current_exhaustiv[sample_count//4], sum(current_exhaustiv)/sample_count,
        current_exhaustiv[3*sample_count//4], max(current_exhaustiv)), file=out_file_ex)

out_file_sat.close()
out_file_ex.close()
```

Listing 6.7: **Shared.py — Shared Library Functions**

```
import random
from Core import *

MIN_DURATION = 1
MAX_DURATION = 10

"""
Read a budgeting scenario from a PABULIB File.
Returns tuple(budgetlimit, projects, votes)
"""
def read_scenario(path):
    random.seed(12345)

    projects = set()
    votes = list()
    file = open(path, 'r')

    meta = dict()
    line = file.readline().replace("\n","")# skip first line
    line = file.readline().replace("\n","")
    while line != "PROJECTS":
        meta[line.split(";")[0]] = line.split(";")[1]
        line = file.readline().replace("\n","")

    line = file.readline().replace("\n","")
    line = file.readline().replace("\n","") # Skip two lines
    while line != "VOTES":
        data = line.split(";")
        random_duration = random.randint(MIN_DURATION, MAX_DURATION)
        projects.add(Project(int(data[0]), data[4], int(data[1]), random_duration))
        line = file.readline().replace("\n","")

    line = file.readline().replace("\n","")
    line = file.readline().replace("\n","") # Skip two lines
    while line:
        data = line.split(";")
        approvals = [int(id) for id in data[4].split(",")]
        votes.append(ApprovalVote(approvals))
        line = file.readline().replace("\n","")
    file.close()
    budget = int(meta["budget"])
    return (budget, projects, votes)

"""
return the ratio opt/online
"""
def satisfaction_perf(util_opt, util_online):
    if util_online == 0:
        if util_opt == 0:
            return 1
        else:
            return float('inf')
    else:
        return util_opt / util_online

"""
Count how many project at most could be simultaneously realized even with max cost
"""
def exhaustiveness_perf(projects, selected, tlimit, blimit, utilities):
    not_realized = sorted(
        [p for p in projects if p not in selected and p.duration <= tlimit and utilities[p] > 0],
         key = lambda p:p.upper_cost)
    number_could_be_added = 0
    b_left = blimit - sum(p.real_cost for p in selected)
    for p in not_realized:
        if p.upper_cost < b_left:
            number_could_be_added += 1
            b_left -= p.upper_cost
    return number_could_be_added

"""
Return by how much the time limit is exceeded
"""
def punctuality_perf(budget_log, tlimit):
    if budget_log.when_finishes_last_project() <= tlimit:
        return 0
    return budget_log.when_finishes_last_project() - tlimit

"""
Assign each project a new uncertainty. Does not change the real cost!
```

```
"""
def resample_uncertainty(projects, uncertainty):
    expected_value = 0.2 # Expected value is 20% of the project cost.
    variance = 0.1  # Variance is 10% of project cost. That is, ~70% of the projects have
                    # a spread of 10% to 30% and 95% have a spread between 0% and 40%.
                    # Negative spreads are impossible.
    if uncertainty == "HIGH":
        expected_value = 0.5
        variance = 0.25
    if uncertainty == "MEDIUM":
        expected_value = 0.2
        variance = 0.1
    if uncertainty == "LOW":
        expected_value = 0.1
        variance = 0.05
    for p in projects:
        if uncertainty == "NONE":
            p.set_random_uncertainty(0)
        else:
            project_spread = random.gauss(expected_value*p.real_cost, variance*p.real_cost)
            project_spread = int(max(0, project_spread))
            p.set_random_uncertainty(project_spread)
```

Listing 6.8: **Core.py — Algorithms and required Datatypes**

```
from typing import *
import random

#################################################################
######    Data Objects                              #############
#################################################################

class Project:
    def __init__(self, id:int, name:str, cost:int, duration:int):
        if cost < 1:
            print("WARN: Cost cannot be 0. Setting to 1.")
            cost = 1
        self.id = id
        self.name = name
        self.real_cost = cost
        self.duration = duration
        self.lower_cost = cost
        self.upper_cost = cost
        self.spread = 0

    """
    Sets a random uncertainty around the real cost.
    Usually max_cost - min_cost should be exactly the spread, but it
    is possible that min_cost was cropped at cost 1.
    """
    def set_random_uncertainty(self, spread:int):
        lower_deviation = random.randint(0, spread)
        upper_deviation = spread - lower_deviation
        self.lower_cost = max(1, self.lower_cost - lower_deviation)
        self.upper_cost += upper_deviation
        self.spread = self.upper_cost - self.lower_cost

    """
    Returns the probability that this project costs exactly cost
    """
    def probability_cost_exact(self, cost:int) -> float:
        if cost > self.upper_cost or cost < self.lower_cost:
            return 0
        return 1/(self.spread + 1)

    """
    Returns the probability that this project costs at most cost
    """
    def probability_cost_max(self, cost:int) -> float:
        if cost < self.lower_cost:
            return 0
        return 1/(self.spread + 1) * (cost - self.lower_cost + 1)

    """
    Returns the probability that this project costs at lest cost
    """
    def probability_cost_min(self, cost:int) -> float:
        if cost > self.upper_cost:
            return 0
        return 1/(self.spread + 1) * (self.upper_cost - cost + 1)

    """
    Returns the probability that this project costs between the two values (included)
    """
    def probability_cost_between(self, costLo:int, costUp:int) -> float:
        if costLo < self.lower_cost:
            costLo = self.lower_cost
        if costUp > self.upper_cost:
            costUp = self.upper_cost
        return 1/(self.spread + 1) * (costUp - costLo + 1)

    """
    Returns a random cost for this project according to the probability distribution
    for the costs. Thus, the expected value of this function is the expected value
    for the cost of this project.
    """
    def draw_random_cost(self) -> int:
        return random.randint(self.lower_cost, self.upper_cost)

    """
    Returns the expected cost for this project (that is, the expected value of
    the probability distribution).
    """
    def expected_cost(self) -> int:
        return self.lower_cost + (self.upper_cost-self.lower_cost)//2

class ProjectLog(dict):
    def start(self, projects, time):
        if isinstance(projects, Project):
            assert projects not in self.keys(), "Projects cannot be started twice"
            self[projects] = time
        else:
            for p in projects:
                assert p not in self.keys(), "Projects cannot be started twice"
                self[p] = time

    """
    Returns all projects that are finished by that time
    """
    def get_finished_projects_until(self, time:int) -> set:
        return {a for a in self.keys() if self[a] + a.duration <= time}

    """
    Returns all projects that are not yet finished but already started at this time
    """
    def get_running_projects_at(self, time:int) -> set:
        return {p for p in self.keys() if self[p] + p.duration > time}

    """
    Returns the time step at which the next project finishes
    """
    def when_finishes_next_project(self, current_time:int):
        running = self.get_running_projects_at(current_time)
        if len(running) > 0:
            return min(self[p]+p.duration for p in running)
        return -1

    """
    Returns the time step at which the last project finishes
    """
    def when_finishes_last_project(self):
        if len(self.keys()) > 0:
            return max(self[p]+p.duration for p in self)
        return -1

    """
    Returns the set of all projects that have been started
    """
    def get_projects(self) -> set:
        return set(self.keys())

class ApprovalVote:
    def __init__(self, approval:set):
        self.approvals = approval

    """
    Satisfaction is the number of approved projects in the outcome.
    """
    def satisfaction(self, projects) -> int:
        if isinstance(projects, Project):
            if projects.id in self.approvals:
                return 1
            return 0
        return len([p for p in projects if p.id in self.approvals])

"""
Returns the sum of approvals for this project
"""
def get_util_satisfaction(votes: Iterable, projects) -> int:
    return sum([vote.satisfaction(projects) for vote in votes])

##################################################################
######    Project Planning Algorithms                #############
##################################################################

"""
This algorithm finds the optimal solution in terms of utility.
"""
```

```
def optimal_planning(projects:Iterable, tlimit:int, blimit:int, util:dict) -> set:
    total_utility = sum(util.values())
    projects = tuple(p for p in projects if p.duration <= tlimit)
    table = [[float('inf') for u in range(total_utility + 1)] for p in range(len(projects) + 1)]
    table[0][0] = 0
    for p in range(1, len(projects) + 1):
        project_utility = util[projects[p-1]]
        project_cost = projects[p-1].real_cost
        for u in range(total_utility + 1):
            table[p][u] = min(table[p-1][u], table[p-1][u-project_utility] + project_cost)
    for u in range(total_utility, 0, -1):
        if table[-1][u] <= blimit:
            total_cost = 0
            total_utility = 0
            ret = set()
            for p in range(len(projects), 0, -1):
                project_utility = util[projects[p-1]]
                project_cost = projects[p-1].real_cost
                if table[p][u - total_utility] != table[p-1][u - total_utility]:
                    ret.add(projects[p-1])
                    total_cost += project_cost
                    total_utility += project_utility
            assert total_cost == table[-1][u]
            assert total_utility == u
            assert total_utility == sum(util[p] for p in ret)
            return ret
    return set()


"""
This algorithm aims at maximizing utility while preserving punctuality, alpha-RA and k-limitation.
It does not guarantee exhaustiveness but does its best effort.
"""
def BEE(projects:Iterable, tlimit:int, blimit:int, k_lim:float, alpha:float, util:dict) -> ProjectLog:
    projects = [p for p in projects if p.duration <= tlimit and util[p] > 0]
    log = ProjectLog()
    rating = {p : util[p]/p.expected_cost() for p in projects} # rating by utility per cost
    T = 0

    while T < tlimit:
        Y = {p for p in projects if p.duration <= tlimit - T} - log.get_projects()
        b_spent = sum(p.real_cost for p in log.get_finished_projects_until(T))
        while len(Y) > 0:
            project = max(Y, key = lambda x:rating[x])
            Y.remove(project)
            b_running = sum(p.upper_cost for p in log.get_running_projects_at(T))
            if project.upper_cost <= k_lim*blimit - b_spent - b_running:
                running = log.get_running_projects_at(T)
                running.add(project)
                if __is_ok_with_risk__(running, blimit - b_spent, alpha):
                    log.start(project, T)

        T = log.when_finishes_next_project(T)
        if T == -1:
            break
    return log


"""
This algorithm aims at maximizing utility while preserving exhaustiveness, alpha-RA and k-limitation.
It does not guarantee punctuality but does its best effort.
"""
def BEP(projects:Iterable, tlimit:int, blimit:int, k_lim:float, alpha:float, util:dict, BEE_log = None) -> ProjectLog:
    projects = [p for p in projects if p.duration <= tlimit and util[p] > 0]
    if BEE_log != None:
        log = BEE_log
    else:
        log = BEE(projects, tlimit, blimit, k_lim, alpha, util)
    T = log.when_finishes_last_project()

    # Now ensure exhaustiveness
    while True:
        b_spent = sum(p.real_cost for p in log.get_finished_projects_until(T))
        avail = blimit - b_spent - sum(p.upper_cost for p in log.get_running_projects_at(T))
        Y = {p for p in projects if p not in log.get_projects() and p.upper_cost <= avail}

        if len(Y) == 0 and log.when_finishes_next_project(T) == -1:
            break
        while len(Y) > 0:
            min_duration = min(p.duration for p in Y)
            project = max([p for p in Y if p.duration == min_duration], key=lambda x:x.expected_cost())
            Y.remove(project)
            b_running = sum(p.upper_cost for p in log.get_running_projects_at(T))
            if project.upper_cost <= k_lim*blimit - b_spent - b_running:
                running = log.get_running_projects_at(T)
                running.add(project)
                if __is_ok_with_risk__(running, blimit - b_spent, alpha):
                    log.start(project, T)

        # Note: At least one project has been started because projects in Y are
        # defined as projects which fit even with maximum cost. Thus, the following
        # will increase T, thus not producing an infinite loop.
        T = log.when_finishes_next_project(T)
    return log


"""
Uses sampling to figure out how likely the set of projects exceeds the budget limit.
If the probability is higher than alpha, returns False.
"""
def __is_ok_with_risk__(projects:set, limit:int, alpha:float) -> float:
    max_cost = sum(p.upper_cost for p in projects)
    min_cost = sum(p.lower_cost for p in projects)
    if max_cost <= limit:
        return True
    if min_cost > limit:
        return False
    if min(p.probability_cost_min(limit - (max_cost - p.upper_cost)) for p in projects) <= alpha:
        return True

    samples = 1000
    exceedings = 0
    for i in range(samples):
        if sum(p.draw_random_cost() for p in projects) > limit:
            exceedings += 1
    return exceedings < samples * alpha
```

3 Omitted Figures of Best-Effort Experiments

We now provide the omitted results from the best-effort experiments in Section 3.6.1.

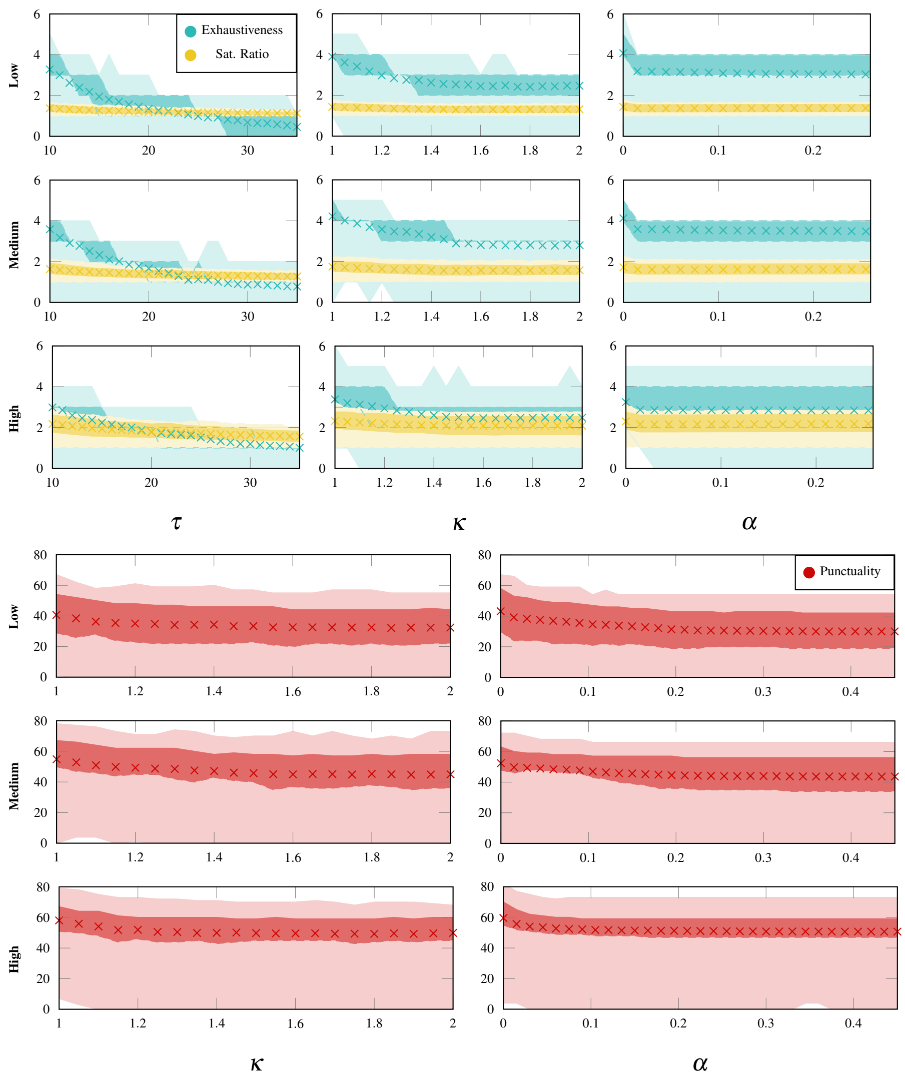


Figure 1: Experimental analysis with dataset Warsaw Ursynów Wysoki Północny 2018.

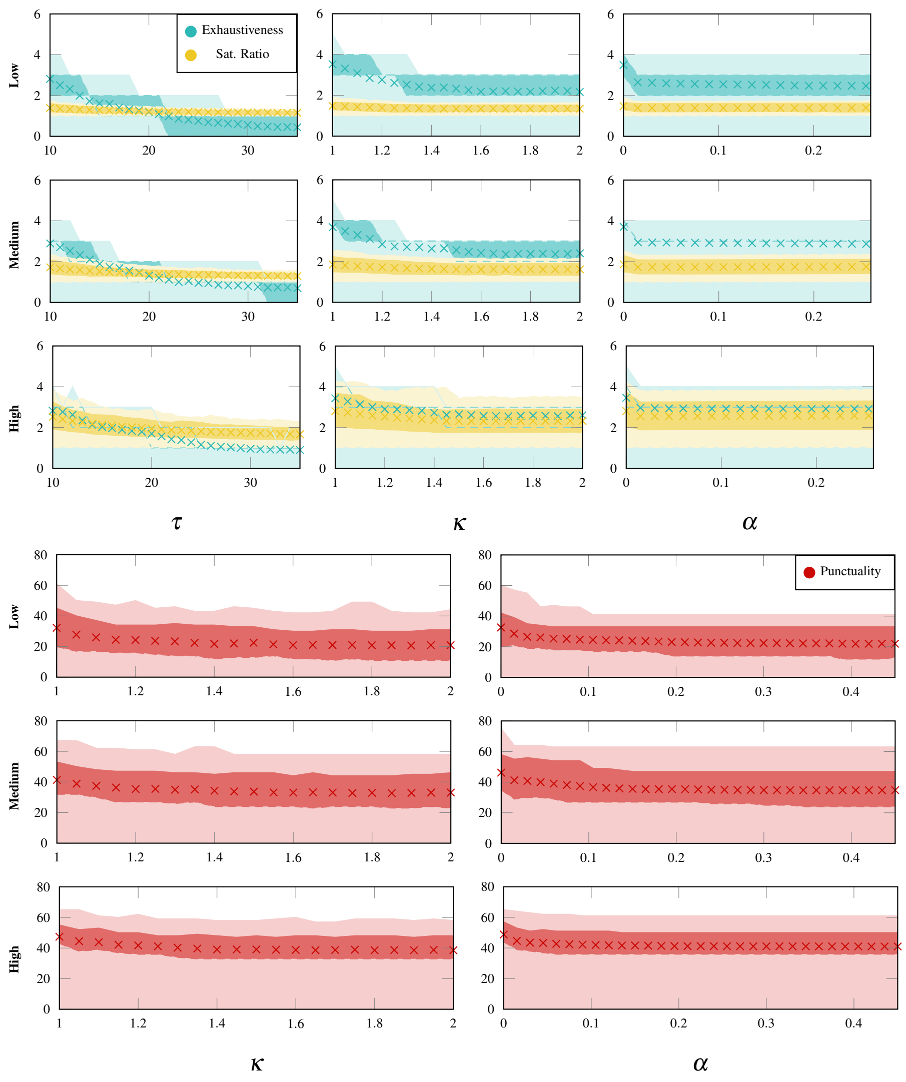


Figure 2: Experimental analysis with dataset Warsaw Ursynów 2019.

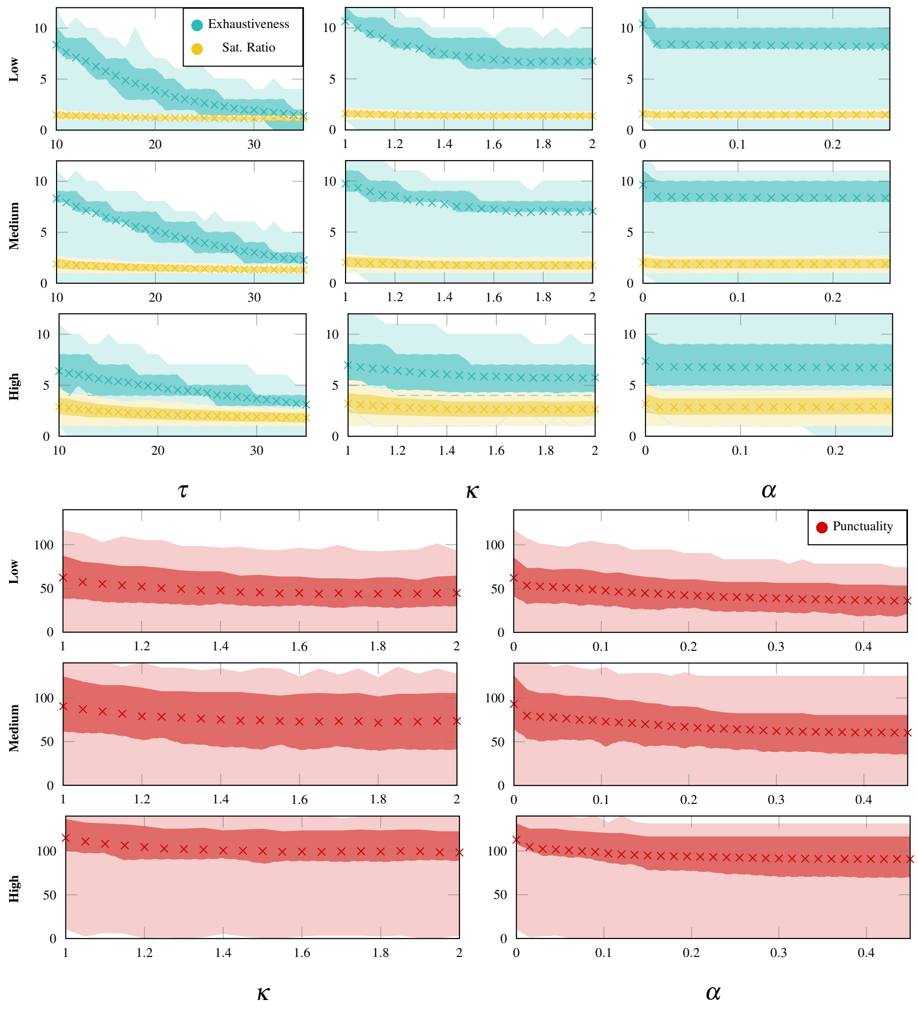


Figure 3: Experimental analysis with dataset Warsaw Ursynów 2020.

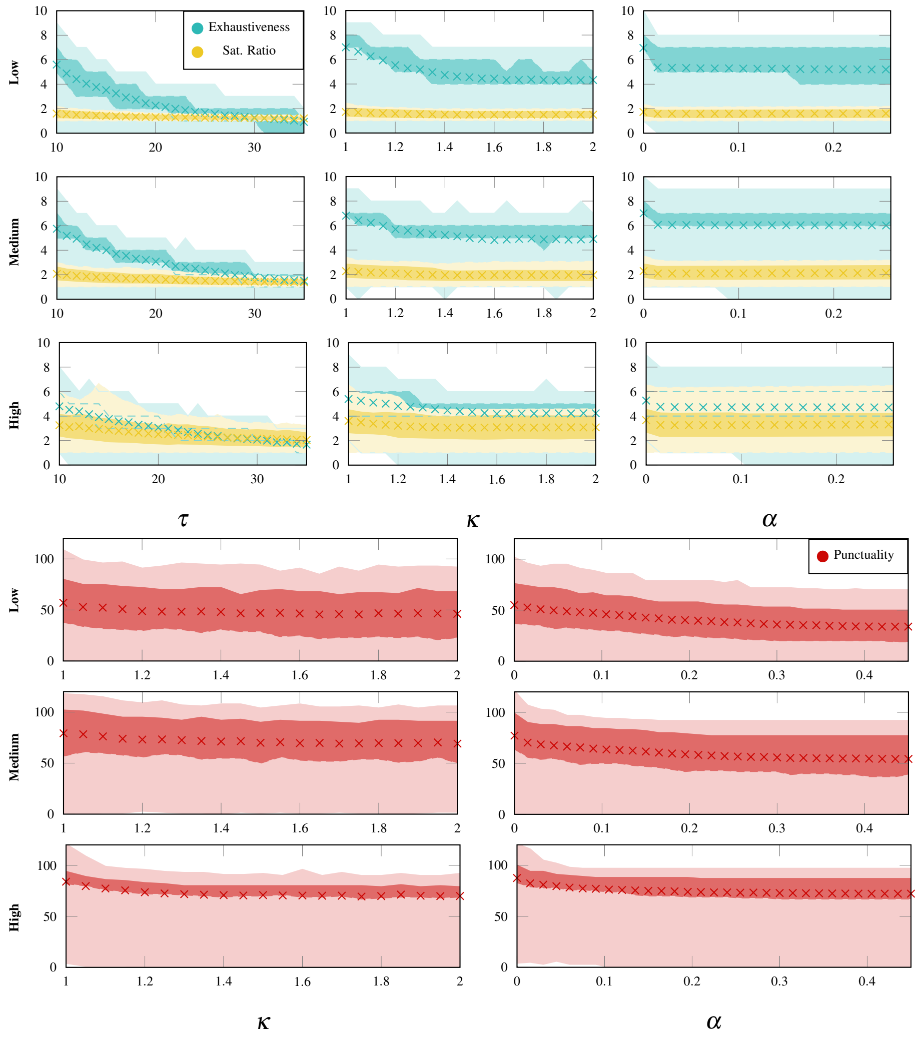


Figure 4: Experimental analysis with dataset Warsaw Praga-Południe 2021.

Erklärung

Eidesstattliche Erklärung
entsprechend §5 der Promotionsordnung vom 15.06.2018

Ich versichere an Eides Statt, dass die Dissertation von mir selbständig und ohne unzulässige fremde Hilfe unter Beachtung der „Grundsätze zur Sicherung guter wissenschaftlicher Praxis an der Heinrich-Heine-Universität Düsseldorf“ erstellt worden ist.

Des Weiteren erkläre ich, dass ich eine Dissertation in der vorliegenden oder in ähnlicher Form noch bei keiner anderen Institution eingereicht habe.

Teile dieser Dissertation wurden bereits in Form von Konferenzbeiträgen veröffentlicht, zur Publikation angenommen oder zur Begutachtung eingereicht. Die vollständigen Zitate der jeweiligen Schriften, sowie meine Anteile an diesen Schriften werden auf den Seiten 45, 63, 80, und 99, angegeben.

Ort, Datum	Christian Laußmann