Participatory Budgeting with Conflicts and Partial Information

Candidate Number: 1062418

Word count: 17144 (Online LaTeX Editor Overleaf word count)

A thesis submitted for the degree of

MSc in Advanced Computer Science

Trinity 2022

Abstract

Participatory budgeting describes a form of election in which citizens can inform their government about their preferences for how they feel a public budget should be spent. In this dissertation, we study a new form of participatory budgeting election in which projects conflict with each other over a one-dimensional interval. We provide a polynomial time algorithm for this scenario that optimises utilitarian welfare, and show that an implementation of the algorithm is efficient enough to be useful in practice. Furthermore, we show that generalising the model to allow each project to have multiple possible locations in which it could be implemented makes the problem of utilitarian welfare optimisation $\mathcal{NP}$-hard.

We also study multiwinner voting and participatory budgeting with partial information, which could be used in the scenario where eliciting the preferences of a whole electorate is infeasible. We define the notion of approximate representation for forms of justified representation and provide bounds on the sample size of a population required to provide approximate representation. Finally, we simulate such a sampling process on a real-world election instance and provide evidence for the practicality of such a process.

Contents

List of Figures

Abbreviations & Frequently Used Notation

$\mathcal{A}$	:	A voting algorithm.
A	:	The approval function of voters to sets of candidates.
B	:	A ballot.
b	:	The budget of a participatory budgeting election.
$\mathcal{C}_q$	:	The set of q-cohesive groups in a multiwinner voting election.
C	:	The set of candidates/projects in an election.
cost	:	The cost function for a participatory budgeting election.
E	:	An election.
G	:	A group of voters.
k	:	The number of candidates to elect in a multiwinner voting election.
ℓ	:	The location function for projects in a participatory budgeting with interval conflicts election.
MPBIC	:	Multi-instance Participatory Budgeting with Interval Conflicts.
MWV	:	MultiWinner Voting.
m	:	The number of candidates/projects in an election.
n	:	The number of voters in an election.
PB	:	Participatory Budgeting.
PBIC	:	Participatory Budgeting with Interval Conflicts.
$\mathcal{P}(X)$	:	The power set of the set X.
p	:	A candidate of an election.
$\mathcal{R}$	:	A fairness property.
T	:	A subset of candidates of an election.
$\mathcal{V}(E)$	:	The set of valid winning sets of an election.
V	:	A set of voters in an election.
v	:	A single voter in an election.
W	:	A winning set of an election.

Chapter 1

Introduction

Participatory budgeting (PB) is a democratic process in which citizens can inform the government about their preferences for how a public budget should be spent. PB was first developed in Brazil by the Brazilian Workers Party in the 1980s and fully implemented in 1989 in the city of Porto Alegre (Shah, 2007), and is now used in more than 1,500 cities around the world (Ganuza and Baiocchi, 2012). PB has had many positive impacts on the cities where it has been implemented. Due to the fact that more citizens are aware of how resources are distributed during a PB election, PB tends to increase the transparency of the governments that implement it and reduce corruption (Shah, 2007). However, the benefits of PB extend beyond just democratic health; studies have shown that increased implementation of PB is strongly correlated with reductions in infant mortality in Brazil (Touchton and Wampler, 2020). This is mainly because PB leads to improvements in infrastructure, particularly health and wastewater infrastructure, in poorer areas of cities that otherwise would not have received investment (Gonçalves, 2014). Clearly then, PB can be a useful tool in a city government's toolbox.

The process in each city differs, but it can generally be broken down into the following steps (Aziz and Shah, 2021):

1. The city is divided into smaller districts, and the budget for each district is decided.
2. Residents share and discuss ideas through meetings and forums to come up with proposals for projects to be completed in their district.
3. The proposals are developed into feasible projects by focus groups and experts, and estimates of the costs of the projects are given.
4. Eligible citizens vote on the final projects to be implemented.
5. Votes are counted and a final budget allocation is formed.

To achieve each of these steps in a way that ensures that citizens are represented fairly in the outcome poses interesting, but not necessarily computational, challenges. However, some of these challenges can be tackled computationally. For example, the problem of redistricting, that is, splitting a city into smaller districts in a way that is fair and not subject to gerrymandering, has been approached algorithmically from many different perspectives (Magleby and Mosesson, 2018; Fifield et al., 2020; Cohen-Addad et al., 2018, 2021). In this dissertation, we shall primarily focus on the computational aspects of the final two steps.

1.1 Contributions

The contributions of this dissertation are as follows:

- PB with project conflicts
  - We generalise the standard PB model to allow projects to conflict with each other on a one-dimensional interval;

  - We provide a pseudo-polynomial time algorithm and a polynomial time algorithm that output election results that maximise utilitarian welfare;
  - We implement the algorithms in Python 3 and test the performances of the algorithms against random and real-world election data;
  - We generalise the model further to allow projects to have multiple possible locations. We prove that the conflict graphs of these problems have unbounded treewidth, and finally prove that the problem of utilitarian welfare optimisation in this scenario is $\mathcal{NP}$-hard.

- PB with partial information
  - Within the context of MWV, we generalise three common notions of representation, and provide a general upper bound on the population sample size required to provide approximate representation;
  - Within MWV, we provide tighter upper bounds on the sample sizes necessary to provide approximate Justified Representation, and approximately optimal utilitarian welfare;
  - Within the context of PB, we provide upper bounds on the necessary sample sizes to provide Extended and Proportional Justified Representation;
  - We simulate taking samples of voters on a real-world PB election to show that the bounds we derive are empirically correct, and to show that the sampling method could be useful in practice.

1.2 Related work

The theoretical problem of PB has been extensively studied (Aziz and Shah, 2021). Much of the theory of PB was first developed within the context of multiwinner voting (MWV) (Faliszewski et al., 2017). The concept of Justified Representation and Extended Justified Representation was first developed by Aziz et al. (2017), with Proportional Justified Representation being developed later by Fernández et al. (2017). Other types of fairness properties exist; for example, Full Justified Representation (Peters et al., 2020) is an even stronger form of justified representation and, indeed, implies Extended Justified Representation. Peters et al. (2020) also discuss generalised utility ballots rather than just approval ballots, while Aziz and Lee (2021) examine fairness properties for ordinal preference ballots.

PB with general conflicts has also previously been studied. Jain, Sornat and Talmon (2021) discuss general interactions between projects, where projects can be substitutes for each other or where projects complement each other so that a project should only be implemented if the complement is also implemented. Jain, Sornat and Talmon (2021) show that, in general, the problem of utilitarian welfare maximisation in this context is $\mathcal{NP}$-complete. The authors later discussed PB where projects are divided into categories that each have their own budget requirement, in addition to the overall budget requirement (Jain, Sornat, Talmon and Zehavi, 2021). They similarly show $\mathcal{NP}$-completeness of the utilitarian welfare maximisation problem in this setting.

Voting for candidates with locations associated with them is studied in the field of facility location. For example, Elkind et al. (2022) investigate fairness within facility location, and consider cases where two facilities cannot be placed at the same location. Chan et al. (2021) provide a survey of voting mechanism design for facility location problems.

Maximising utilitarian welfare in the standard PB context is equivalent to the 0-1 knapsack problem, of which there is abundant literature (Martello and Toth, 1990; Kellerer et al., 2004). Pferschy and Schauer (2009) discuss the knapsack problem where items can conflict with each other, and provide algorithms for two special cases of the conflict graph, which we shall examine in more detail in Section 3.1.1. The specific problem of maximising welfare within standard PB has been studied before by Fluschnik et al. (2019), where they discuss maximising utilitarian welfare, egalitarian welfare, and Nash welfare.

Partial information within voting theory has been studied extensively, but primarily from the perspective of election manipulation given partial information (Dey et al., 2018; Erdélyi and Reger, 2016). Hazon et al. (2012) studied the evaluation of election results under partial information in a single-winner election, and discuss computing the probability of a particular candidate winning given only partial information. Kalech et al. (2011) also explore practical rules for voting under partial information, where in this setting, only partial information about each voter's preferences is known, but every voter is considered.

The method discussed in Chapter 4 involves random sampling of the population. Sampling techniques have been extensively studied (Cochran, 1977; Wu and Thompson, 2020), and in particular, methods for deriving upper bounds on the necessary sample size based on the desired statistical power (Henry, 1990) and standard error (Land and Zheng, 2010) have also been studied . Finally, the method we construct can be thought of as a form of (ε, δ) approximation scheme (Mitzenmacher and Upfal, 2005), albeit in some cases, we obtain a non-polynomial approximation scheme.

1.3 Outline

- In Chapter 2, we introduce the theoretical foundations of MWV and PB.
- In Chapter 3, we generalise the PB model to allow projects to be placed along an interval. We shall show that utilitarian welfare optimisation can be solved in polynomial time in this new context, implement the algorithm in Python 3, and test the performance of the algorithm against random and real-world election data. We shall then further generalise the model to the scenario where projects have multiple possible locations at which they could be placed. We show that despite restrictions on the number of possible locations for each project, and the number of other projects overlapping with any given project location, the conflict graphs of these problems have unbounded treewidth. Indeed, we finally show that the problem of utilitarian welfare optimisation is $\mathcal{NP}$-hard in this scenario.
- In Chapter 4, we discuss taking a random sample of the population and polling only the sample voters, instead of assuming that we have full information about the whole population. We prove upper bounds on the population sample size necessary to provide approximate representation within MWV and PB contexts. We shall also simulate the sampling method on a real-world PB election to show that the bounds hold for a real election, and to show that the sampling method could prove useful in practice.
- In Chapter 5, we shall discuss the contributions made and identify further open problems.

Chapter 2

Preliminaries

We shall start by establishing the theoretical foundations of PB.

2.1 Voting models

Much of the theoretical development of PB stems from the study of MWV, so we shall first define MWV and explore some of the properties of MWV before turning to PB.

Definition 2.1. (MWV election)
Let

- $V = \{1, \ldots, n\}$ a set of voters;
- $C = \{x_1, \ldots, x_m\}$ a set of candidates or projects;
- $k \in \mathbb{N}$ a total number of candidates to elect;
- $A : V \to \mathcal{P}(C)$ an approval function that maps voters to the set of candidates they approve.

Then $E = (V, C, k, A)$ is an *MWV election*. The primary task associated with an MWV election is to find a set $W \subseteq C$ of projects that are selected to be completed with $|W| = k$. We say that W is *valid* if $|W| = k$.

An MWV election occurs, for example, in committee elections, where we have a fixed number k of committee members that we wish to vote in. In the above model, and throughout this dissertation, we shall focus on the case where voters' preferences are given through an approval ballot. There are other ways in which voters can express their preferences; for example, ordinal ballots allow voters to specify ranked preferences rather than just providing two levels of preference. Similarly, utility ballots allow each voter to indicate exactly how much personal utility they would gain from each candidate. The extra information about voters' true preferences would allow us to compute election results that match the population's preferences more closely in general; however, the extra information also adds extra complexity to the model and to the fairness properties that we shall define below. Therefore, we shall use approval ballots throughout the dissertation, but it is worth noting that results such as Theorem 3.14 could likely be adapted to the case of full utility ballots.

We can extend the MWV election model to allow the candidates to have different costs associated with them, which gives us the PB election model.

Definition 2.2. (PB election)
Let

- $V = \{1, \ldots, n\}$ a set of voters;
- $C = \{x_1, \ldots, x_m\}$ a set of candidates or projects;
- $\text{cost} : C \to \mathbb{R}_+$ a cost function that takes each project to a cost value;
- $b \in \mathbb{R}_+$ a total budget available for implementing projects;
- $A : V \to \mathcal{P}(C)$ an approval function that maps voters to the set of candidates that they approve.

Then $E = (V, C, \text{cost}, b, A)$ is a *PB election*. The primary task associated with a PB election is to find a set $W \subseteq C$ of projects that are selected to be completed, with $\sum_{p \in W} \text{cost}(p) \leq b$. We again say that W is *valid* if $\sum_{p \in W} \text{cost}(p) \leq b$.

We can see that every MWV election is actually a special case of a PB election; if we set $b = k$ and $\text{cost}(p) = 1$ for all $p \in C$.

Remark 2.3. We shall slightly abuse the notation and define $A(G) = \bigcap_{v \in G} A(v)$ for a group of voters $G \subseteq V$. $A(G)$ is the set of candidates that every voter in G approves. Furthermore, for $T \subseteq C$, we define $\text{cost}(T) = \sum_{p \in T} \text{cost}(p)$ so that $\text{cost}(T)$ is the total cost of implementing all projects in T.

Definition 2.4. (Valid set)
For an MWV or a PB election E, let $\mathcal{V}(E) = \{W \subseteq C : W \text{ is valid}\}$. Explicitly, for an MWV election, we have $\mathcal{V}(E) = \{W \subseteq C : |W| = k\}$ and for a PB election, we have $\mathcal{V}(E) = \{W \subseteq C : \text{cost}(W) \leq b\}$.

Now that we have defined the elections, we shall now discuss some desirable properties for W (the result of an election) and how to achieve those properties for W.

2.2 Fairness properties

There are potentially up to 2^m different possible choices for W in PB elections, and $\binom{m}{k}$ choices in MWV elections, but typically we wish to select W in a way that ensures some fairness property holds of W. To select a winning set, we will use an election algorithm.

Definition 2.5. (Election algorithm)
Let $\mathcal{A}$ be an algorithm that takes an election over a set of candidates C as input, and produces a subset of C as output. Let E be an election over candidates C. Then we say $W \subseteq C$ is the *result*, or the *winning set*, of election E using algorithm

$\mathcal{A}$ if the output of $\mathcal{A}$ on the input E is W, and $W \in \mathcal{V}(E)$, so W is a valid result. We shall use the notation $\mathcal{A}(E) = W$.

Definition 2.6. (Fairness properties)
$\mathcal{R}$ is a fairness property if, for each election E, there exists a set $\mathcal{R}(E) \subseteq \mathcal{V}(E)$. Let E be an election and W the result of election E using algorithm $\mathcal{A}$, so $W = \mathcal{A}(E)$. We say that W satisfies the fairness property $\mathcal{R}$ if $W \in \mathcal{R}(E)$. We say $\mathcal{A}$ *guarantees* $\mathcal{R}$ if for every election E, $\mathcal{A}(E) \in \mathcal{R}(E)$.

Nothing in the above two definitions guarantees any intuitive sense of fairness for W. For example, for PB elections, we could define $\mathcal{R}(E) = \{\emptyset\}$, and $\mathcal{A}$ as the algorithm that outputs the constant empty set. The empty set is always a valid project set since it has cost 0, and $\mathcal{A}$ guarantees $\mathcal{R}$. Of course, we would like to define fairness properties that encapsulate intuitive ideas of fairness. We shall now define some common fairness properties that we will use later.

Definition 2.7. (Utilitarian welfare optimisation)
The utilitarian welfare optimisation fairness property is defined by

$$\mathcal{R}(E) = \text{argmax}_{W \in \mathcal{V}(E)} \sum_{v \in V} |A(v) \cap W|.$$

In words, W satisfies *utilitarian welfare optimisation* if it achieves the maximum possible sum of the number of approved candidates in the winning set per voter, compared to all other valid winning sets.

For MWV, utilitarian welfare can be achieved in polynomial time: sort the projects by number of approvals and select the best k projects. In the standard PB model, utilitarian welfare optimisation is actually a special case of the 0-1 knapsack problem.

Definition 2.8. (0-1 knapsack problem)
Suppose that we are given a set of r items, where item i has weight $w_i \geq 0$ and value v_i, and a weight limit/capacity w. The *knapsack problem* is defined as the problem of assigning to each variable z_i a value in $\{0, 1\}$ such that $\sum_{i=1}^{r} z_i v_i$ is maximised, subject to $\sum_{i=1}^{r} z_i w_i \leq w$.

If we define the items to be the candidates, the weight of each item to be the cost of the candidate, and the value of the item to be the number of people that approve of the candidate, then we can see that utilitarian welfare optimisation is a special case of the knapsack problem. The decision version of the knapsack problem (deciding whether $\sum_{i=1}^{r} z_i v_i \geq t$ is achievable or not) is weakly $\mathcal{NP}$-complete (Martello and Toth, 1990). However, we can construct a polynomial time algorithm to solve the problem in the PB election scenario because the utilities v_i are polynomially bounded by the input, as we shall show in Section 2.3.

We can define other fairness properties. Let us first focus on fairness properties specific to MWV, and then generalise them to PB. The following fairness properties will be defined in terms of cohesive groups of voters.

Definition 2.9. (MWV cohesive group)
We say a group of voters $G \subseteq V$ is *q-cohesive* (with respect to election E) for some $1 \leq q \leq k$ if $|G| \geq \frac{qn}{k}$ and $|A(G)| \geq q$. Let

$$\mathcal{C}_q = \{G \subseteq V : |G| \geq \frac{qn}{k} \text{ and } |A(G)| \geq q\}$$

denote the set of q-cohesive groups for election E, if E is unambiguous in the context, or we shall use $\mathcal{C}_q(E)$ if necessary.

A q-cohesive group is a set of voters who all approve of some set of q candidates and who together make up at least $\frac{q}{k}$ proportion of V. Informally, this group G represents a set of voters who, if they were "sufficiently underrepresented" by a winning set, could make a convincing case that they deserve to be represented by at least q members of the winning set, and could propose a set of q candidates on which they would all agree.

Definition 2.10. (MWV Justified Representation (MWV-JR))
MWV-JR (Aziz et al., 2017) is a fairness property defined by

$$\text{MWV-JR}(E) = \{W \in \mathcal{V}(E) : \forall G \in \mathcal{C}_1, \exists v \in G, |A(v) \cap W| \geq 1\}.$$

If W satisfies Justified Representation, then there is no group of voters that all agree on a single candidate and is large enough to be represented by a candidate, but where no voter in the group currently has any candidate they approve in the winning set.

Definition 2.11. (MWV Proportional Justified Representation (MWV-PJR))
MWV-PJR (Fernández et al., 2017) is a fairness property defined by

$$\text{MWV-PJR}(E) = \left\{ W \in \mathcal{V}(E) : \forall 1 \leq q \leq k, \forall G \in \mathcal{C}_q, |W \cap \bigcup_{v \in G} A(v)| \geq q \right\}.$$

Definition 2.12. (MWV Extended Justified Representation (MWV-EJR))
MWV-EJR (Aziz et al., 2017) is a fairness property defined by

$$\text{MWV-EJR}(E) = \{W \in \mathcal{V}(E) : \forall 1 \leq q \leq k, \forall G \in \mathcal{C}_q, \exists v \in G |W \cap A(v)| \geq q\}.$$

MWV-EJR guarantees that there is no cohesive group of voters of size $\geq q\frac{n}{k}$, such that every voter has fewer than q candidates in the winning set that they approve of. Intuitively, every voter in this group isn't as represented as they should be given that they belong to a large enough group that all agree on q candidates. Similarly, MWV-PJR guarantees that there is no cohesive group such that there are fewer than q members of the winning set that any of the voters in the group approve. We find that if a winning set satisfies MWV-EJR then it also satisfies MWV-PJR, and if it satisfies MWV-PJR then it satisfies MWV-JR. To see this, first consider that if we had a q-cohesive group such that

$$|W \cap \bigcup_{v \in G} A(v)| < q,$$

then there certainly cannot exist a voter in G such that $|W \cap A(v)| \geq q$. Similarly, if we had a 1-cohesive group with $|A(v) \cap W| = 0$ for every voter, then

$$|W \cap \bigcup_{v \in G} A(v)| = 0 < 1.$$

To generalise these notions to the PB setting, we first need to change the definition of a cohesive group to take into account the cost of the candidates. The new definitions of a cohesive group and PB-EJR were formalised by Peters et al. (2020).

Definition 2.13. (PB Cohesive group)
Define a group $G \subseteq V$ of voters as T*-cohesive* for $T \subseteq C$ (with respect to election E) if $|G| \geq \frac{n}{b} cost(T)$, and $T \subseteq A(G)$. Let

$$\mathcal{C}_T = \left\{ G \subseteq V : |G| \geq \frac{n}{b} \text{cost}(T) \text{ and } T \subseteq A(G) \right\},$$

similar to before.

We can then use this definition of a cohesive group in our definitions.

Definition 2.14. (PB Extended Justified Representation (PB-EJR))
PB-EJR is a fairness property defined by

$$\text{PB-EJR}(E) = \{W \in \mathcal{V}(E) : \forall T \subseteq C, \forall G \in \mathcal{C}_T, \exists v \in G, |W \cap A(v)| \geq |T|\}.$$

For PB-EJR, we use the cost of the projects that the voters approve of to define the minimum size of the cohesive group, rather than just the number of projects. As an example, if the members of a group agree on 3 projects that together would use 100% of the budget, then this group needs to constitute a larger proportion of the population to justify implementing the projects compared to 3 projects that together use 5% of the budget.

To call both MWV-EJR and PB-EJR by a similar name is justified: on MWV elections, these properties are equivalent.

Lemma 2.15. *In MWV elections,* MWV-EJR *and* PB-EJR *are equivalent.*

Proof. Let $E = (V, C, \text{cost}, b, A)$ be a PB election that is also a MWV election, so $b = k$ and $\text{cost}(p) = 1$ for every candidate $p \in C$. First, suppose that W satisfies PB-EJR. Let G be an MWV q-cohesive group. Then $|G| \geq \frac{qn}{k}$ and $|A(G)| \geq q$. Let $T \subseteq A(G)$ with $|T| = q$. Then $|G| \geq \frac{qn}{k} = \frac{n}{k}\sum_{p \in T} \text{cost}(p)$ so G is a PB T-cohesive group. Therefore, there exists a voter $v \in G$ with $|A(v) \cap W| \geq |T| = q$, so G satisfies MWV-EJR.

Now suppose that W satisfies MWV-EJR. Let G be a PB T-cohesive group. Let $q = |T|$. Then $|G| \geq \frac{n}{b}\sum_{p \in T} \text{cost}(p) = \frac{n}{b}|T| = \frac{qn}{k}$. Also, since $T \subseteq A(G)$, we have that $|A(G)| \geq q$, and so G is an MWV q-cohesive group. Since G satisfies MWV-EJR, there exists $v \in G$ such that $|A(v) \cap W| \geq q = |T|$, so G satisfies PB-EJR. ■

PB-PJR has been defined similarly before by Los et al. (2022).

Definition 2.16. (PB Proportional Justified Representation (PB-PJR))
PB-PJR is a fairness property defined by

$$\text{PB-PJR}(E) = \{W \in \mathcal{V}(E) : \forall T \subseteq C, \forall G \in \mathcal{C}_T, |\bigcup_{v \in G} A(v) \cap W| \geq |T|\}.$$

Similarly to the MWV case, we also find that MWV-PJR is equivalent to PB-PJR on MWV elections (Los et al., 2022), and we see that PB-EJR implies PB-PJR; if there were some T-cohesive group with $|\bigcup_{v \in G} A(v) \cap W| < |T|$, then there cannot exist a voter v with $|A(v) \cap W| \geq |T|$.

A definition of PB-JR is harder to construct. This is because in the definition of these fairness properties for the PB setting, we use cohesive groups for subsets rather than for values of q. For MWV elections, each candidate can be considered to be a uniform unit of the total budget of k candidates. We then consider only cohesive groups of size $n \times \frac{1}{k}$, so groups that make up at least a whole unit proportion of the population need representation, where here, representation means that at least one voter of the group must be satisfied with the winning set by at least by candidate. In the PB setting however, the concept of a unit of the budget is less clear. Aziz et al. (2018) consider a unit of the budget as $\min_{p \in C} \operatorname{cost}(p)$, so a group that agrees on a candidate must have a size of at least $\frac{n \min_{p \in C} \operatorname{cost}(p)}{b}$ to be guaranteed representation. They provide the following definition, which they call StrongBJR-L.

Definition 2.17. (PB Justified Representation (PB-JR))
PB-JR is a fairness property defined by

$$\text{PB-JR}(E) = \Bigg\{ W \in \mathcal{V}(E) : \forall G \subseteq V,$$
$$\left[\left(|A(G)| \geq 1 \text{ and } |G| \geq \frac{n \min_{p \in C} \operatorname{cost}(p)}{b} \right) \implies \exists v \in G, A(v) \cap W \neq \emptyset \right] \Bigg\}.$$

However, Aziz et al. (2018) also show that under this definition, the existence of a set of projects that satisfies PB-JR is not guaranteed. Therefore, we shall focus on PB-EJR and PB-PJR in this dissertation, since the existence of a winning set that satisfies them is guaranteed to exist due to the fact that the Method of Equal Shares (described in Section 2.3) always finds a winning set that satisfies PB-EJR.

2.3 Election algorithms

Finally, we will look at some examples of election algorithms. For utilitarian welfare optimisation, we have a pseudo-polynomial time algorithm that can solve the 0-1 knapsack problem. Recall that as input, we are given a set of r items, where the item i has weight $w_i \geq 0$ and value v_i, and a weight limit w. The dynamic programming algorithm constructs a table indexed by index i, for $0 \leq i \leq r$ and by weight y, for $0 \leq y \leq w$. The entry in the table at position (i, y) will store the best possible value achievable from the first i of r items, and with a weight limit of y (instead of w). We can iteratively build the table T. First, we place 0 in the table entries with $i = 0$ or $y = 0$, since we can achieve no value if we use none of the items or if we have a weight limit of 0. Then, for each value $1 \leq i \leq r$ in increasing order and for each $1 \leq y \leq w$, we can construct the table entry

$$T[i, y] = \begin{cases} T[i-1, y] & \text{if } w_i > y \\ \max(T[i-1, y], T[i-1, y-w_i] + v_i) & \text{otherwise} \end{cases}.$$

With a filled table, we can then work backwards from the entry $(r, w) = (i, y)$ to reconstruct the winning set; if $T[i-1, y] < T[i-1, y-w_i] + v_i$, then we know that item i is included, so we can then update the remaining weight limit to $y = y - w_i$, and otherwise we do not include item i. We reduce i by 1, and repeat. We continue until we reach $i = 0$. The algorithm runs in pseudo-polynomial time, since it runs in $\mathcal{O}(rw)$, where r is polynomial in the size of the input, but w could be exponential in the size of the input.

Instead, we can index T by $0 \leq i \leq r$ and by utility $0 \leq y \leq \sum_{i=0}^{r} v_i$, and store the minimum weight capacity required to achieve a utility of at least y. We can similarly compute this table:

$$T[i, y] = \min(T[i-1, y], w_i + T[i-1, \max(y - v_i, 0)]),$$

and we start with $T[0, y] = \infty, T[i, 0] = 0$. This table can be computed in $\mathcal{O}(r \sum_{i=0}^{r} v_i)$, which is again pseudo-polynomial time if we are given just the v_i's as input. We can also compute the winning set in polynomial time, similarly to before. However, when applied to PB, we are provided with each voter's full preferences as input, and in this case $v_i = |\{v \in V : x_i \in A(v)\}|$. So, here we have that $\sum_{i=0}^{r} v_i \leq nm$ which is polynomial in the size of the input. For more details on solving knapsack problems using dynamic programming, see the textbook (Kellerer et al., 2004).

Let us now look at algorithms for achieving JR, PJR, and EJR. Two of the earliest algorithms designed to provide representation for MWV are Proportional

Approval Voting (PAV) and Phragmén's method. PAV is simple; we iterate over all possible valid sets W and compute its score

$$s(W) = \sum_{v \in V} \sum_{i=1}^{|A(v) \cap W|} \frac{1}{i}.$$

We then select the winning set with the highest score. PAV satisfies MWV-EJR (Aziz et al., 2017), but runs in potentially exponential time, since there are $\binom{m}{k}$ valid sets W. PAV can be generalised to the PB setting by instead considering all valid sets W for the PB election. However, generalising in this way means that PAV no longer satisfies PB-EJR, or even PB-PJR (Los et al., 2022), and still runs in potentially exponential time. Phragmén's method on the other hand satisfies MWV-PJR and PB-PJR (but not MWV-EJR) (Los et al., 2022), and can be computed in polynomial time. Phragmén's method can be thought of as a continuous process where each voter is provided with a fixed rate income of money over time. As soon as some group of voters who all approve of a project p which is not already in W can afford the cost of p, p is added to W and the voters who approve p get their balance reset to 0. This continues until a project is selected that would make the total cost of W exceed the budget.

The Method of Equal Shares (on approval ballots), otherwise known as Rule X, guarantees PB-EJR, as well as another fairness property called priceability (Peters et al., 2020). The method works as follows (Peters and Skowron, 2019): start with an empty winning set W and allocate \$1 to each voter. We want each voter to "pay" for the projects they approve of and that are implemented. Let $\$P_v(p)$ be the amount that the voter v pays for project p, initialised at 0. Let $\$P_i(W) = \sum_{p \in W} \$P_i(p)$ be the amount that the voter v already pays for the projects in W. For $r \geq 0$, say that a project p is r-affordable if

$$\sum_{v : p \in A(v)} \min(\$r, \$1 - \$P_v(W)) \geq \$\text{cost}(p).$$

The sum $\sum_{v:p\in A(v)} \min(\$r, \$1 - \$P_v(W))$ is the total amount of money that the voters of project p would pay for a given r, where we need to take the minimum of r with $1 - P_v(W)$ to account for the case where voter v does not have $\$r$ left, and so instead they will simply pay all the money they have left, which is $\$1 - \$P_v(W)$. If no project is r-affordable for any r, terminate and output W, otherwise select the project p that is r-affordable for the lowest r to be put into W, and set $\$P_v(p) = \min(\$r, \$1 - \$P_v(W))$ for all voters v with $p \in A(v)$. This algorithm runs in polynomial time, and since it guarantees PB-EJR, it also guarantees PB-PJR and MWV-(E/P/)JR. Therefore, we can use Rule X as the algorithm $\mathcal{A}$ in all the main theorems in Chapter 4. We shall now move on to use these definitions and concepts in the following chapters.

Chapter 3

Participatory budgeting with conflicts

First, we shall consider the problem of PB when we have conflicts between projects. By conflicts between projects, we mean that some subsets of projects are unable to be implemented together in the same winning set. Standard PB can be thought of as already having some project conflicts; any subset of the projects whose total cost is greater than the budget cannot be implemented in a winning project set. Project conflicts arise naturally in different ways. Multiple projects may carry out the same general function, and as such, at most one project of each function may need completing. For example, we could have multiple different plans to build a park in the city, but it has been decided that at most one park should be built (perhaps the city only has the ongoing budget to maintain a single park). In this case, the park projects would conflict with each other.

Conflicts between projects could also occur due to implementation workload. If all the projects that won in the election were a single governmental department's responsibility, *i.e.* all projects were to be implemented by the Parks and Recreation department, this may be too much extra workload for the department. In this way, we may define the Parks and Recreation projects to be conflicting in such a way that we set a maximum limit on the number of those projects that can be implemented.

In this dissertation, however, I shall focus on one-dimensional location conflicts. For the moment, we make some assumptions about the situation we wish to model:

1. Once implemented, a project occupies a certain space;
2. The space that a project occupies is part of a one-dimensional interval;
3. The space a project occupies on the interval is itself an interval, so every project is contiguous on the larger interval;
4. Every project proposal proposes a single specific location, rather than a set or range of possible locations in which it can be implemented in;
5. No two projects can occupy the same point.

Assumption 1, as we shall see in Section 3.1.3.1, can be removed to allow us to also consider projects without location, and we study Assumption 4 in more detail in Section 3.2 to show that removing this assumption makes the problem of utilitarian welfare optimisation intractable.

To illustrate this idea, see Figure 3.1. Here, we can see a road at the bottom and some proposed public works. From left to right, we have: new public toilets (WC), a new shopping centre, a new DIY workshop, a new bus shelter, a new park, a new town square with a fountain, and a new multi-storey car park. The specific location along the road matters; we can see that the proposed DIY workshop would take up the same space along the road as the park, the shopping centre, and the bus shelter. If we implement the DIY workshop, we can then no longer implement the park, the shopping centre, or the bus shelter.

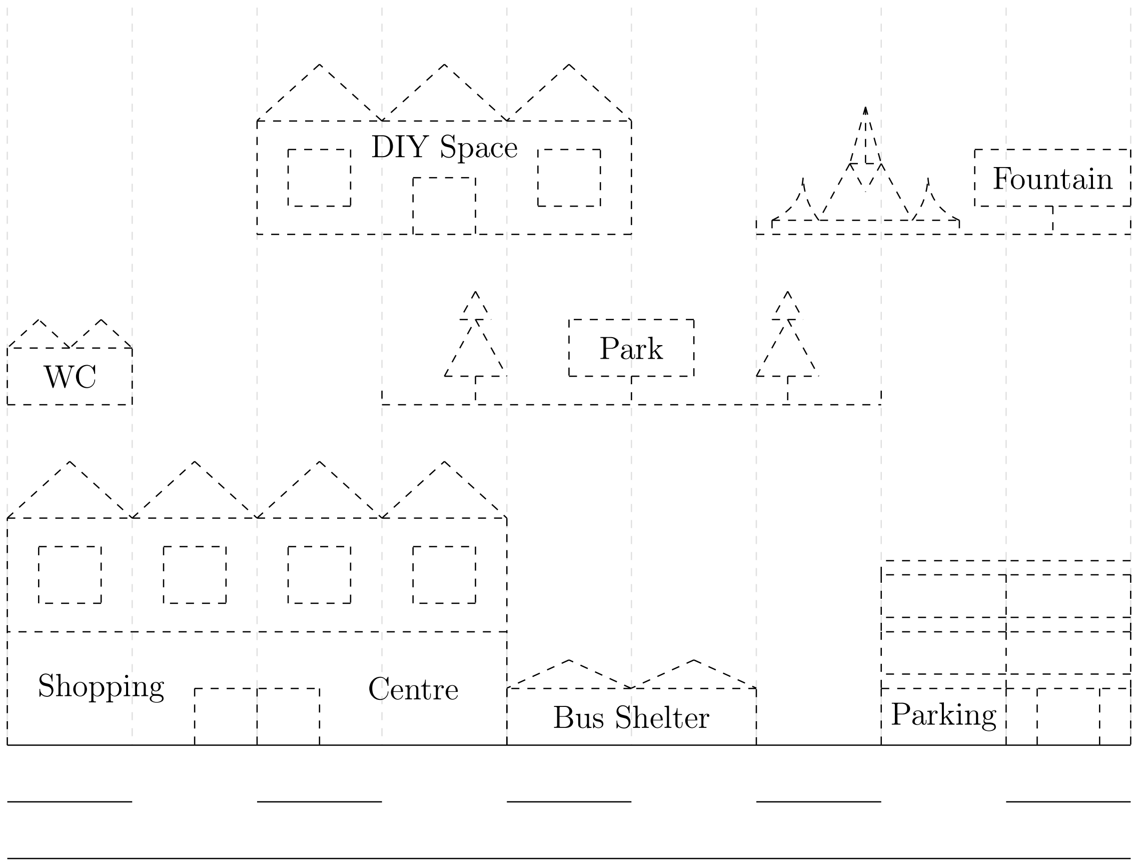


Figure 3.1: An example of how the new participatory budgeting model could be used to plan the public amenities along a road.

3.1 Conflicts on the one-dimensional interval

Let us start by modifying the standard PB model defined in Definition 2.2.

Definition 3.1. (PB with interval conflicts (PBIC) election)

Define $\mathcal{I} := \{[s, e) : 0 \leq s < e \leq 1\}$. A *PB with interval conflicts (PBIC) election* $E = (V, C, \text{cost}, b, A, \ell)$ is a standard PB election that additionally has a function $\ell : C \to \mathcal{I}$. We interpret $\ell(x_i) = [s, e)$ as meaning that project x_i would occupy $[s, e)$ on the interval. In addition to requiring that valid winning sets be within budget, we require that for W to be valid, we need that for any $p_1, p_2 \in W$, with $p_1 \neq p_2$, it holds that $\ell(p_1) \cap \ell(p_2) = \emptyset$. We therefore define

$$\mathcal{V}(E) = \{W \subseteq C : \text{cost}(W) \leq b \text{ and } \forall p_1, p_2 \in W, p_1 \neq p_2 \implies \ell(p_1) \cap \ell(p_2) = \emptyset\}.$$

Remark 3.2. We say that x_i occupies $[s, e)$ instead of $[s, e]$ so that we can place projects right next to each other, which makes examples easier to demonstrate, but practically does not make much difference to the computations, since otherwise we could subtract a small ε "buffer" to the ends of each project to ensure they do not overlap. We say that s is the start of the interval and e is the end of the interval.

Definition 3.3. (Conflict graph of a PBIC election)
Let E be a PBIC election with candidate set C and location function ℓ. Let

$$F = \{(x_i, x_j) \subseteq C^2 : x_i \neq x_j \text{ and } \ell(x_i) \cap \ell(x_j) \neq \emptyset\}.$$

Then we define the *conflict graph* of E to be the graph $G = (C, F)$.

We shall turn our attention to fairness properties for this new model. First, fairness properties other than welfare make less sense in this new context. Even MWV-JR, a relatively weak form of fairness, may not always be satisfiable. In contrast, in a standard MWV election, a winning set W that satisfies MWV-JR always exists (Aziz et al., 2017). Consider the following election:

- $V = \{1, \ldots, n\}$;
- $C = \{x_1, x_2, \cdots, x_{2n}\}$;
- $\text{cost}(x) = 1$ for all $x \in C$;
- $b = 2n$;
- $A(i) = \{x_i, x_{n+i}\}$ for $i \in V$;
- $\ell(x) = [0, 1)$ for all $x \in C$.

This is an MWV election, since the cost of each candidate is the same. Only one of the $2n$ candidates is electable since they overlap on the interval; however, $\{v\}$ forms a 1-cohesive group for x_v and x_{n+v} for every $v \in V$. This is because for both $p = x_v$ and $p = x_{n+v}$, if a 1-cohesive group agrees on p then the minimum group size is $\frac{n}{2n} = \frac{1}{2}$, so we only need a single voter to approve of p. So, if a winning set satisfies JR, we would need $|A(v) \cap W| = |\{x_v, x_{n+v}\} \cap W| > 0$ for all v, and so we would need to implement at least n different projects, which is not possible.

In fact, for any fairness property $\mathcal{R}$ defined for PB elections, we can prove that generalising $\mathcal{R}$ to PBIC elections does not make it easier to find winning sets that satisfy $\mathcal{R}$.

Theorem 3.4. *Let $\mathcal{R}$ be a fairness property defined for PB elections. Let X be the problem of finding a winning set W for a PB election $E = (V, C, cost, b, A)$ that satisfies $\mathcal{R}(E)$ if it exists and let Y be the problem of finding a winning set W' for a PBIC election $E' = (V', C', cost', b', A', \ell)$ that satisfies*

$$\mathcal{R}(E') = \mathcal{R}((V', C', cost', b', A')) \cap \mathcal{V}(E')$$

if it exists. Then problem Y is at least as hard as problem X under Karp reductions.

Proof. Suppose we are given a PB election $E = (V, C, \mathrm{cost}, b, A)$. Label each of the m members of C by x_i for $0 \leq i < m$. Let $\ell(x_i) = [\frac{i}{m}, \frac{i+1}{m})$. This gives us a PBIC instance $E' = (V, C, \mathrm{cost}, b, A, \ell)$. No project in C overlaps with any other project, so since $\mathcal{V}(E) = \mathcal{V}(E')$, we have that $\mathcal{R}(E) = \mathcal{R}(E')$ and so problem Y is at least as hard as problem X. ■

Elkind et al. (2022) consider a fairness property called Maximal Justified Representation (MJR), specifically designed for voting in MWV elections under conflicts.

Definition 3.5. (Maximal Justified Representation (MWV-MJR))
Let $E = (V, C, \text{cost}, b, A, \ell)$ be an election. Let

$$\mathcal{N} = \{G \subseteq V : |G| \geq \frac{n}{k} \wedge A(G) \neq \emptyset\}$$

be the set of 1-cohesive groups of E. Let the group-representing set of candidate x_i be $\mathcal{N}_i = \{G \in \mathcal{N} : x_i \in \bigcup_{v \in G} A(v)\}$. Then the fairness property is defined by

$$\text{MWV-MRJ}(E) = \left\{ W \in \mathcal{V}(E) : \forall W' \in \mathcal{V}(E), \bigcup_{x_i \in W} \mathcal{N}_i \not\subset \bigcup_{x_j \in W'} \mathcal{N}_j \right\}.$$

Elkind et al. (2022) show that under no conflicts, MWV-MJR is equivalent to MWV-JR, and that with conflicts, an MWV-MJR winning set always exists. However, in general, it is $\mathcal{NP}$-complete to find such a set, and specialising to conflicts on the interval does not immediately appear to make the problem easier.

We now turn our attention to welfare optimisation. We shall first show that egalitarian welfare optimisation is at least as hard as finding a minimum set cover, even for just MWV elections.

Definition 3.6. (Egalitarian welfare optimisation)
The egalitarian welfare optimisation fairness property is defined as

$$\mathcal{R}(E) = \text{argmax}_{W \in \mathcal{V}(E)} \min_{v \in V} |A(v) \cap W|.$$

In words, W satisfies *egalitarian welfare optimisation* if it achieves the maximum possible minimum utility over all voters, compared to all other valid winning sets.

Definition 3.7. (Minimum set cover)
Let U be a finite set of items, $C \subseteq \mathcal{P}(U)$ a collection of subsets of U, and $j \leq |C|$. The minimum set cover problem (Garey and Johnson, 1991) is defined as deciding whether there exists some $C' \subseteq C$ with $|C'| \leq j$ such that for every element $x \in U$, there exists some $c \in C'$ such that $x \in c$.

The minimum set cover problem is known to be $\mathcal{NP}$-complete (Garey and Johnson, 1991). We shall now show that the problem of deciding if an egalitarian welfare of at least 1 is achievable is at least as hard as deciding the minimum set cover problem.

Theorem 3.8. *The problem of deciding whether there exists a winning set W for MWV election E such that the egalitarian welfare achieved by W is at least 1 is at least as hard as the minimum set cover problem under Karp reductions.*

Proof. Let $\mathcal{R}$ be the egalitarian welfare optimisation fairness property. Let X be a collection of subsets of Y, and $j \leq |X|$. We shall construct a MWV election E such that X contains a cover for Y of size j or less if and only if for $W \in \mathcal{R}(E)$, $\min_{v \in V} |A(v) \cap W| \geq 1$. Let $V = Y, C = X, k = j, A(v) = \{c \in C : v \in c\}$ and $E = (V, C, k, A)$. Let $W \in \mathcal{R}(E)$. If $\min_{v \in V} |A(v) \cap W| \geq 1$, then $|A(v) \cap W| \geq 1$ for all $v \in V$. We have $|W| = k = j$ and for every member v of $V = Y$ there is some member p of W such that $p \in A(v)$ and so $v \in p$, so X contains a cover for Y of size j or less.

Now suppose that X contains a cover for Y of size j or less. There exists a subset $X' \subseteq X$ with $|X'| \leq j$ such that every element of Y belongs to some member of X'. Let $T \supseteq X'$ be some set with $|T| = j$. Then

$$\min_{v \in V} |A(v) \cap T| \geq \min_{v \in V} |A(v) \cap X'| = \min_{v \in Y} |\{c \in X' : v \in c\}| \geq 1.$$

Therefore $\max_{T \in \mathcal{V}(E)} \min_{v \in V} |A(v) \cap T| \geq 1$ and so for $W \in \mathcal{R}(E)$,

$$\min_{v \in V} |A(v) \cap W| \geq 1.$$

■

Therefore, if we had a polynomial time algorithm that could find a winning set W that achieves egalitarian welfare optimisation for an election E, we could decide if E has egalitarian welfare of at least 1 and so hence this algorithm could be used to decide the minimum set cover problem. Therefore, unless $\mathcal{P} = \mathcal{NP}$, no polynomial time algorithm exists.

3.1.1 Utilitarian welfare optimisation

We shall now look at utilitarian welfare optimisation. Utilitarian welfare optimisation with conflicts between pairs of projects is equivalent to knapsack with conflicts:

Definition 3.9. (Knapsack with conflicts)
Suppose that we are given the following:

- a set $I = \{y_1, \ldots, y_r\}$ of r items, where item y_i has weight $w_i \geq 0$ and value v_i;
- a weight limit w;
- a graph $G = (I, E)$ called the conflict graph.

The *knapsack with conflicts problem* is defined as the problem of assigning each variable x_i a value in $\{0, 1\}$ such that $\sum_{i=1}^{r} x_i v_i$ is maximised, subject to $x_i + x_j \leq 1$ for each $\{y_i, y_j\} \in E$ and to $\sum_{i=1}^{r} x_i w_i \leq w$ (in other words, for two items that have an edge between them, at most one of them is in the knapsack).

The equivalence can be seen for an election E by setting $I = C, w = b$, and G as the conflict graph of E. The knapsack with conflicts problem has been studied before by Pferschy and Schauer (2009). Knapsack with conflicts clearly generalises the standard 0-1 knapsack, just by taking the edge set $E = \emptyset$, and so is weakly $\mathcal{NP}$-hard. We also see that we can reduce the independent set problem to knapsack with conflicts: for a graph (V, E), let $I = V$, with $w_i = 0$ and $v_i = 1$, and let $w = 0$. Then by solving this knapsack with conflicts problem, we can find the maximum independent set. So, this shows that the knapsack with conflicts problem is strongly $\mathcal{NP}$-hard.

Pferschy and Schauer (2009) provide two pseudo-polynomial time algorithms for special cases of the conflict graph, one for the case where the conflict graph is chordal, and one where the conflict graph has bounded treewidth. First, we shall define these terms and then show that the chordal graph algorithm would work in the PBIC setting.

Definition 3.10. (Chordal graph)
A graph $G = (V, E)$ is *chordal* if every induced cycle $G[S]$ (an induced subgraph that is a simple cycle) has exactly 3 vertices.

Definition 3.11. (Treewidth)
A *tree decomposition* of a graph $G = (V, E)$ is a tree $T = (X, E')$ where $X \subseteq \mathcal{P}(V)$ is such that

1. $\bigcup_{S \in X} S = V$, so every vertex in V appears in some $S \in X$;
2. For any edge $(a, b) \in E$ there exists a vertex S of T such that both a and b are in S, so for every edge there is some vertex $S \in X$ that captures the edge relation;
3. If a vertex $x \in V$ is in both S and S' for S, S' vertices of the tree, then x is in every vertex of the tree T that lies on the (unique) path from S to S'.

The *width* of the tree decomposition is defined as $\omega(T) = \max_{S \in X} |S| - 1$. Let $\tau(G)$ be the set of all tree decompositions of G. The *treewidth* of G is defined as $\min_{T \in \tau(G)} \omega(T)$.

The conflict graphs of PBIC problems are exactly interval graphs, and we can show that interval graphs are chordal.

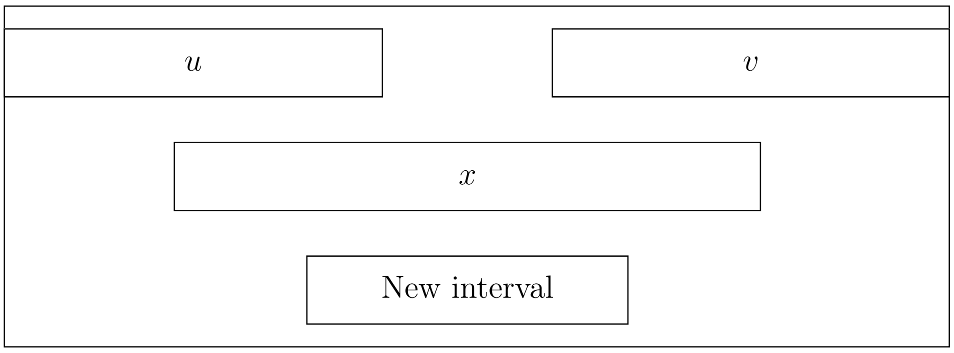


Figure 3.2: An illustration of the positioning of u, x, v, and the new interval.

Definition 3.12. (Interval graph)

Let $G = (V, E)$ be a graph. Then G is an interval graph if there exists a set

$$S = \{S_i : S_i \text{ is an interval on the real line and } 1 \leq i \leq |V|\}$$

and a bijection $f : V \to S$ such that $(a, b) \in E \iff f(a) \cap f(b) \neq \emptyset$. In words, it is a graph with vertices that can be interpreted as intervals and edges between intervals that overlap.

Lemma 3.13. *Interval graphs are chordal.*

Proof. Suppose that there exists a t-cycle subgraph with no chord for $t > 3$ in an interval graph. Let x be some vertex of the cycle, and let u, v be its two neighbours. Then u and v do not have an edge between them, so on the interval they do not overlap. Without loss of generality, say that the end of the interval u is less than the start of the interval v, so u comes strictly before v on the interval. Then x overlaps both of them, so we must have that the start of x is before the end of u and the end of x is after the start of v. See Figure 3.2 for an illustration.

Consider the other $t - 3$ vertices of the cycle. If we contract all the edges between these vertices, we get a 4-cycle. Contracting the edges is equivalent to taking the union of the intervals together into one single interval, with the start being the leftmost start of all the intervals and the end being the rightmost end of all the intervals. This new large interval conflicts with both u and v, so we also see that its start is before the end of u and the end of the new interval is after the start of v. But then we necessarily see that the new large interval will overlap with x between the end of u and the start of v. Therefore, there exists an edge between the new contracted vertex and x. But then that means that there existed an edge between x and one of the $t-3$ contracted vertices which are not neighbours of x, so there exists a chord in this cycle. ■

The algorithm for chordal graphs presented by Pferschy and Schauer (2009) runs in $\mathcal{O}\left((n+m)P^2\right)$ where the authors use n as the number of items, m as the number of edges in the conflict graph, and P as the sum of the values of the items. In our case, we have m items, at most m^2 edges, and $P \leq nm$, so the algorithm would run in $\mathcal{O}\left((m+m^2)(nm)^2\right) = \mathcal{O}(n^2m^4)$. This runs in polynomial time with respect to our input, since we assume that we are given the ballot of each voter individually as input.

However, we can achieve optimal utilitarian welfare in $\mathcal{O}(nm^2)$ as we shall see. First, we will discuss a pseudo-polynomial time algorithm, and then optimise it to achieve the polynomial time algorithm.

Theorem 3.14. *There exists a pseudo-polynomial time algorithm that guarantees optimal utilitarian welfare.*

Proof. See Algorithm 1 for the pseudocode of the algorithm. The algorithm operates similarly to the standard pseudo-polynomial time algorithm for knapsack described in Section 2.3. We create a dynamic programming table T where in $T[j][y]$ we shall store the maximum utility that we can achieve using the first j projects, where we order the projects by the rightmost point of the interval of each project, with a maximum cost limit of y. The main change from the standard knapsack algorithm is how we generate $T[j][y]$ from previously generated table entries. Note the step "Find preceding non-conflicting index t of j". Here, we find the largest index t of P such that the end of $P[t]$' is before the start of $P[j]$'. We let $t = 0$ if there is no preceding index. To find the largest index t of the project that precedes j without conflicting with it, we can compute these values for each j at the beginning of the algorithm and store them in a map, so when we need to find t in the main body of the algorithm, we can find the value in a map in $\mathcal{O}(1)$.

Now, we show that the algorithm is correct. We proceed by induction. Clearly, the maximum utility achievable from the first 0 projects is 0; the only possible achievable set of projects is the empty set, regardless of the weight limit. Now, suppose that for all $0 \leq i < j$ and $0 \leq x \leq y$, we have that $T[i][x]$ is the maximum utility that we can achieve using the first i projects with a maximum cost limit of x. We wish to compute $T[j][y]$. Consider an optimal set of projects that achieve the maximum possible utility using the first j projects with a maximum cost limit of y.

Either this set contains $P[j]$ or it does not. If not, then this optimal set is the same as the optimal set for projects up to $j-1$ with the same weight limit of w, so in this scenario we would have $T[j][y] = T[j-1][y]$ by our inductive hypothesis with $i = j-1, x = y$. Note that if the cost of $P[j]$ is greater than the weight limit, we know that the optimal set cannot include $P[j]$. Now, consider the case where the set does contain $P[j]$. Let t be the largest index of P such that the end position of $P[t]$ is before the start of $P[j]$, and 0 if it doesn't exist. Since $P[t+1]$'s end is after $P[j]$'s start, we have that every $P[i]$'s end is after $P[j]$'s start for $t+1 \leq i \leq j-1$, since P is ordered by end position. Also, since $P[j]$ has the rightmost end point of all projects up to j, $P[i]$'s end is between $P[j]$'s start and end for $t+1 \leq i \leq j-1$. Therefore, we know that all of the projects from $t+1$ to $j-1$ overlap with $P[j]$, and none of the projects up to t overlap with $P[j]$. Therefore, in this case, we know that the optimal project set is $P[j]$ union with the optimal set for the first t projects, but with cost limit $y - \text{cost}(P[j])$, since we have used some of the capacity y by including $P[j]$ in the set. Therefore,

$T[j][y] = T[t][y - \text{cost}(P[i])] + v[P[j]]$ where $v[P[j]]$ is the utility gained from implementing $P[j]$. Now, if we do not know whether $P[j]$ should be included in the optimal set for j, y, we can just take the maximum of the two cases, and so

$$T[j][y] = \max(T[j-1][y], T[t][y - \text{cost}(P[j])] + v[P[j]].$$

Once we have computed the table values for the maximum achievable utility, we can recover the corresponding set using a similar argument. From table entry $T[j][y]$, if we have that $T[j-1][y] < T[t][y - \text{cost}(P[j])] + v[P[j]]$, then clearly we are better to add project $P[j]$ to the optimal set for the projects up to project t using the weight limit $y - \text{cost}(P[j])$, which we can generate inductively, similar to above. See Algorithm 1 for the pseudocode of this section of the algorithm.

Finally, the runtime of the algorithm is pseudo-polynomial. It takes $\mathcal{O}(nm)$ to compute v, $\mathcal{O}(m \log m)$ to compute P, and $\mathcal{O}(b)$ to setup the table. The main loop then takes $\mathcal{O}(m(\log(m) + b))$. Finally, computing the winning set takes $\mathcal{O}(m \log(m))$. Overall then, the algorithm runs in

$$\mathcal{O}(m(n + \log(m) + b)).$$

n and m are polynomial in the size of the input, since we will assume that we are given A as a table of voters to ballots, however, b is not polynomial in the size of the input but is polynomial in the numeric value of the input. ■

```
input : $E = (V, A, C, \mathrm{cost}, b, \ell)$, a PBIC election.
$T \leftarrow$ an $(m+1) \times (b+1)$ array, initialised with null values in each
  position, 0-indexed;
for $a \in V$ do
    for $x \in A(a)$ do
        $v[x] \rightarrow v[x] + 1$;
    end
end
$P \leftarrow C$ sorted by rightmost location on interval (1-indexed);
for $y$=0 to $b$ do
    $T[0][y] \leftarrow 0$;
end
for $j$=1 to $m$ do
    Find preceding non-conflicting index $t$ of $j$;
    for $y$=0 to $b$ do
        if $\mathit{cost}(P[j]) > y$ then
            $T[j][y] \leftarrow T[j-1][y]$;
        end
        else
            $T[j][y] \leftarrow \max(T[j-1][y], T[t][y - \mathrm{cost}(P[j])] + v[P[j]]$;
        end
    end
end
$X \leftarrow \emptyset$;
$j \leftarrow m$;
$y \leftarrow b$;
while $j > 0$ do
    Find preceding non-conflicting index $t$ of $j$;
    if $\mathit{cost}(P[j]) \leq y$ and $T[j-1][y] < T[t][y - \mathit{cost}(P[j])] + v[P[j]]$ then
        $X \leftarrow X \cup \{P[j]\}$;
        $j \leftarrow t$;
        $y \leftarrow y - \mathrm{cost}(P[j])$;
    end
    else
        $j \leftarrow j - 1$;
    end
end
output: $X$
```

Algorithm 1: Maximising utilitarian welfare in a PBIC setting in pseudo-polynomial time.

We can optimise Algorithm 1 to be fully polynomial. Rather than indexing the dynamic programming table by weight, we can index it by utility and store the minimum weight limit required to achieve that utility. We can also optimise the implementation of the algorithm by generating table entries "on demand" rather than generating the entire table. We can do this by keeping a stack of table entries that we need to generate to compute the final answer. When we pop a table entry x from the stack, we check that the prerequisite table entries have already been computed. If not, we put x back on the stack and additionally add the uncomputed prerequisite entries on top of x on the stack as well.

Theorem 3.15. *There exists a polynomial time algorithm that solves the PBIC problem optimally with respect to utilitarian welfare.*

Proof. See Algorithm 2 for the pseudocode. The key points in the proof of correctness follow in a similar fashion to the proof of correctness of the pseudo-polynomial time algorithm. We find the maximum utility in the m'th row that has a minimum required weight limit $\leq b$ by means of a binary search, where we generate the value in the table on demand, and then compare it with the budget to choose if we should continue searching for a lesser or greater achieved utility value. The use of a binary search works here because the table values will be sorted. If $u \leq v$, the weight limit required to achieve a utility of u will be at most the weight limit w required to achieve a utility of v.

We then construct the corresponding project set similar to before; when we decide if project j should be in the set, we compare the weight limits required to achieve a utility of u from the first j projects where the project set includes and excludes project j. Since both options achieve a utility of u, we should choose the option of least weight. Since we choose the option of least weight at each stage, we select the set of least weight overall that achieves a utility of *upper*. From the correctness of the dynamic programming table, the minimum weight set that achieves utility *upper* has weight at most b, so we get an optimal set that is valid.

The runtime is polynomial in the size of the input: we compute v in $\mathcal{O}(nm)$ and set up T in $\mathcal{O}(m^2 n)$ (since $U_{\max} \leq nm$). We compute P in $\mathcal{O}(m \log m)$. The table generation takes time $\mathcal{O}(nm^2 + \log(nm))$; we have at most $\mathcal{O}(nm^2)$ table generation steps and at most $\mathcal{O}(\log(nm))$ binary search steps. We also take $\mathcal{O}(m)$ steps to recover the set X. Therefore, in total, the algorithm runs in $\mathcal{O}(nm^2)$ steps, which is polynomial in the size of the input. ■

```
input : $E = (V, A, C, \mathrm{cost}, b, \ell)$, a PBIC election.
Compute $v[x]$ for every $x \in C$ as before;
$U_{\max} \leftarrow \sum_{x \in C} v[x]$;
$T \leftarrow$ an $(m+1) \times (U_{\max}+1)$ array, initialised with null values in each
  position, 0-indexed;
$T[0][u] \leftarrow \infty$ for all $1 \leq u \leq U_{\max}$; $T[i][0] \leftarrow 0$ for all $0 \leq i \leq m$;
$P \leftarrow C$ sorted by rightmost location on interval (1-indexed);
$lower \leftarrow 0$, $upper \leftarrow U_{\max}$;
while $lower < upper$ do
    $current\_u \leftarrow \lfloor \frac{lower+upper+1}{2} \rfloor$;
    stack$\leftarrow [(m, current\_u)]$;
    while stack is not empty do
        $(j, u) \leftarrow$ stack.pop();
        Find preceding non-conflicting index $t$ of $j$;
        $needed\_util \leftarrow \max(0, u - v[P[j]])$;
        if $T[t][needed\_util]$ is not null and $T[j-1][u]$ is not null then
            $T[j][u] = \min(\mathrm{cost}(P[j]) + T[t][needed\_util], T[j-1][u])$
        else
            Push $(j, u)$ back to the stack, then push $(t, needed\_util)$ and
              $(j-1, u)$ to the stack if table entries are null respectively;
        end
    end
    if $T[m][current\_u] > b$ then
        $upper \leftarrow current\_u - 1$;
    else
        $lower \leftarrow current\_u$;
    end
end
$X \leftarrow \emptyset, j \leftarrow m, u \leftarrow$ upper;
while $j > 0$ do
    Find preceding non-conflicting index $t$ of $j$;
    $needed\_util \leftarrow \max(0, u - v[P[j]])$;
    if $T[j-1][u] > cost(P[j]) + T[t][needed\_util]$ then
        $X \leftarrow X \cup \{P[j]\}$;
        $j \leftarrow t, u \leftarrow needed\_util$;
    else
        $j \leftarrow j - 1$;
    end
end
output: $X$
```

Algorithm 2: Maximising utilitarian welfare in a PBIC setting in polynomial time.

3.1.2 Empirical testing

The correctness of the algorithms was confirmed for small elections by iterating over all valid winning sets and ensuring the winning set output by the algorithms had maximal utility with respect to the valid sets. To test the practical performance of the algorithm, 2 versions of the algorithm were implemented. The first version of the algorithm generates a $(m+1) \times (b+1)$ dynamic programming table on-demand, which we shall call standard interval knapsack. This algorithm works similarly to the algorithm in Algorithm 1, but with the on-demand stack demonstrated in Algorithm 2. The Python code for this version is provided in Appendix A.1.3. The second version is the algorithm described in Algorithm 2 where we instead use a $(m+1) \times (U_{\max}+1)$ dynamic programming table with on-demand generation, which we shall call reverse interval knapsack. The Python code for this version is provided in Appendix A.1.4.

The algorithms were implemented using Python 3.10. Although the algorithms could have been implemented in any modern programming language, Python 3 was chosen for the following reasons:

- Quality of existing libraries and built-in data types. Python 3 has an extensive number of inbuilt data types that have been utilised in this project, such as sets and dictionaries. The library support for Python is also excellent.

- Existing familiarity with Python 3. Previous experience with Python helped significantly, as commonly used features and optimisations were previously known to the author. Python is also a widely used language and so is also likely to be supported for future work.

- Ease of testing. Python has excellent profiling and interactive debugging tools, which allow for easier bug fixing and testing.

To benchmark the code, an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) r6a.large instance was created and used to remotely run the benchmarking code. This instance had 16GiB memory, a sustained clock speed of 3.6GHz, and used Amazon Linux 2 (Kernel 5.10) as its operating system. This instance type was chosen because it most closely resembled the author's personal laptop, which had been used for informal testing.

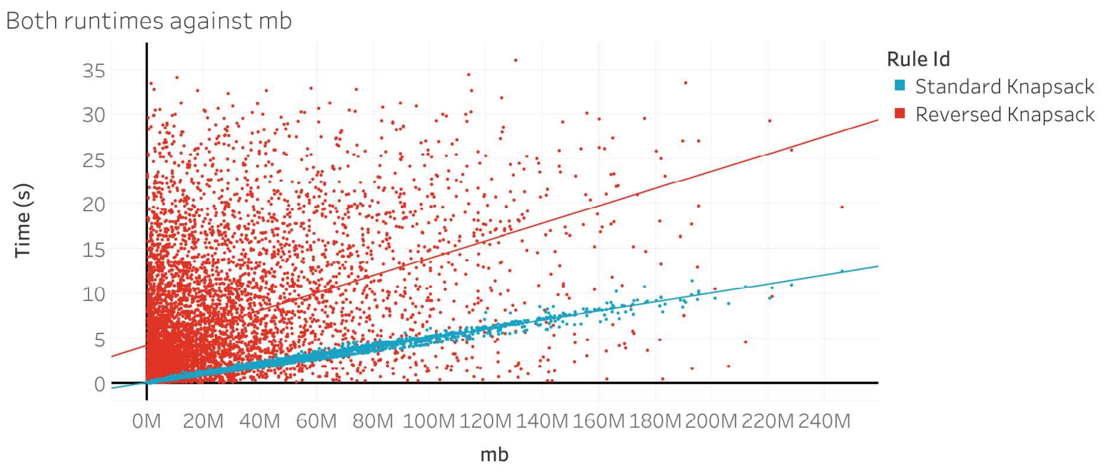
To begin, we shall look at the algorithm performance on randomly generated elections. To generate the artificial data, random elections were created with the number of voters uniformly distributed between 1,000 and 300,000, and the number of projects uniformly distributed between 3 and 70. The location data for the random elections is generated by selecting a size for the project between 0 and 1, and then a random start location on the interval for the project such that the project still remains wholly within the interval. The costs of the projects were distributed uniformly at random between 5,000 and 100,000. The budget is then chosen uniformly between 1 times the average cost of a single project and the sum of the costs of all projects. The graphs in Figure 3.3 illustrate the runtime of these versions.

We make the following observations:

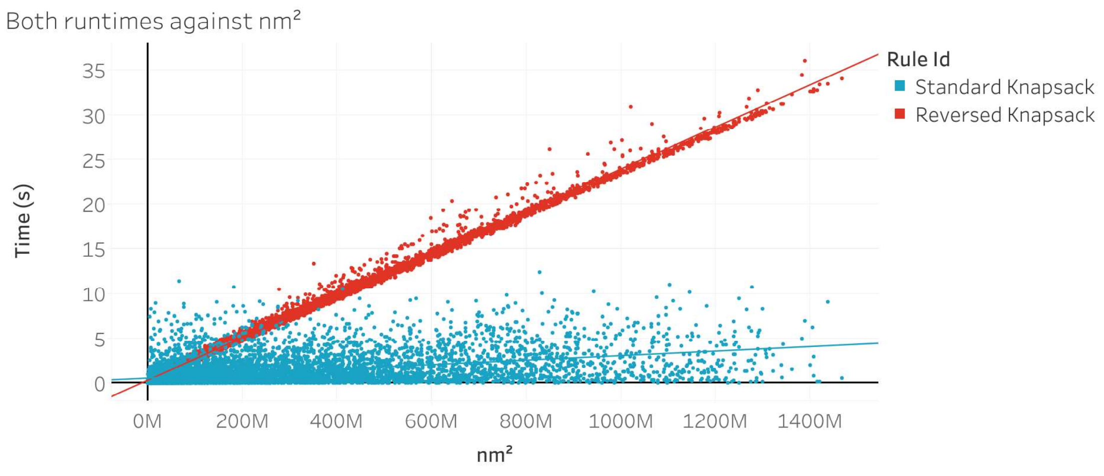
- The practical runtime of the standard interval knapsack algorithm is highly correlated with mb, suggesting that the practical runtime is $\mathcal{O}(mb)$ as predicted. The trend line for the standard knapsack time is

$$t = 4.997 \times 10^{-8} mb + 1.990 \times 10^{-2}.$$

  The R-squared value for the trend line of standard knapsack runtime against mb is 0.9967.



(a) The runtimes of both algorithms, plotted against the value of mb for each election.



(b) The runtimes of both algorithms plotted against the value of nm^2 for each election.

Figure 3.3: Runtimes of both the standard and reversed knapsack algorithms compared to both mb and nm^2 on random data.

- The reverse interval knapsack algorithm's practical runtime is highly correlated with nm^2, suggesting that the practical runtime is indeed $\mathcal{O}(nm^2)$ as predicted. The trend line for the reverse knapsack time is

$$t = 2.359 \times 10^{-8} nm^2 + 2.820 \times 10^{-1}.$$

  The R-squared value for the trend line of reverse knapsack is 0.9923.

- The runtime of the reverse interval knapsack is only weakly correlated with mb, and similarly the runtime of the standard interval knapsack is only weakly correlated with nm^2. The R-squared value for the trend line for the runtime of reverse knapsack against mb is 0.2062, and the R-squared value for the trend line for the runtime of standard knapsack against nm^2 is 0.2018.

- The standard knapsack algorithm generally tends to perform better than the reversed knapsack algorithm on random data.

We then test the standard and reverse interval knapsack algorithms on partially real data. To test the practical performance of the algorithms, Pabulib data (Stolicki et al., 2020) was used as input to the algorithms. The data available within Pabulib is real-world PB election data, primarily from Poland. At the time of testing, there were 458 different elections available within Pabulib. However, these elections do not have location data associated with the projects. For testing, the location data for the projects is randomly generated in the same way as for the random data. This location generation process was repeated 10 times for each election to obtain 10 test elections for each real election, for a total of 4580 test elections.

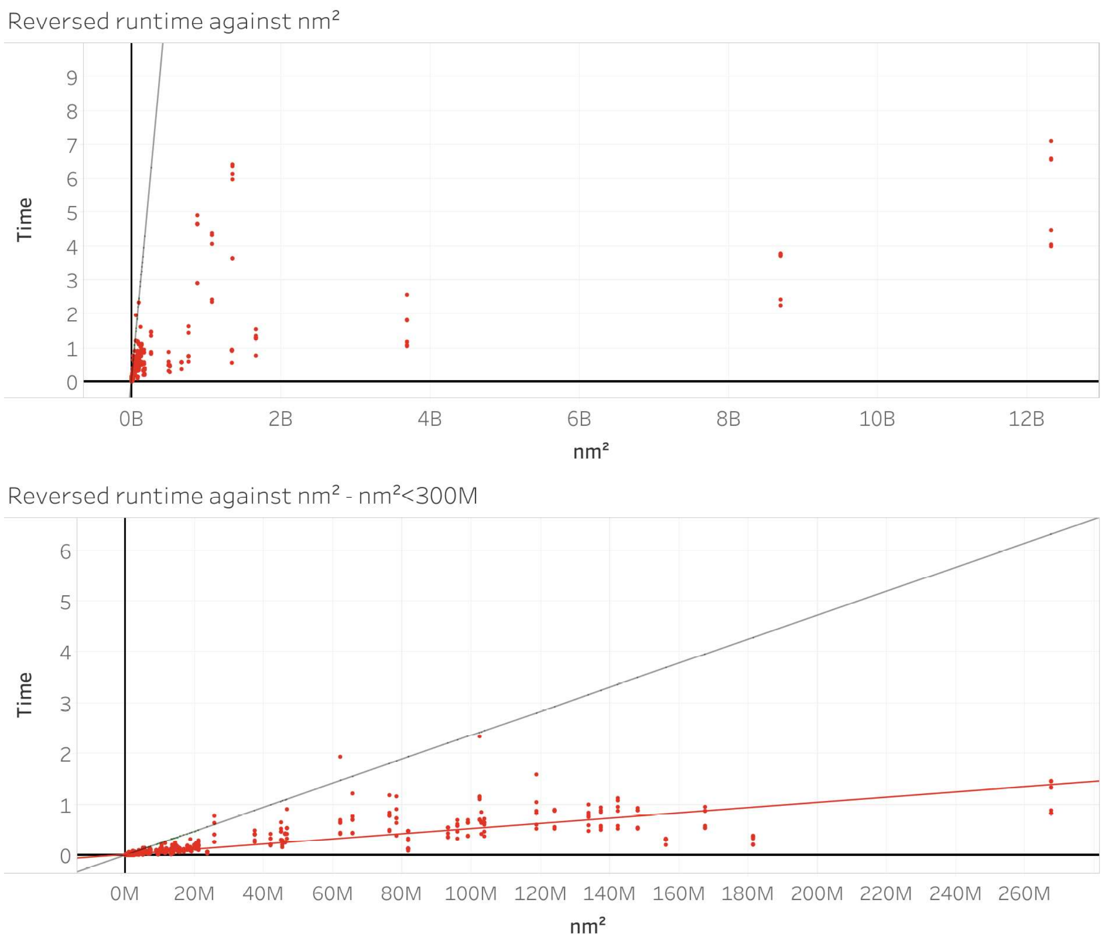
From Figure 3.4, we can make the following observations:

- We can see that the relationship between runtime and nm^2 is less clear for the reversed knapsack algorithm compared to the random data, but that the runtime of the algorithm on real data is significantly less than on random data. On the election with the highest value of $nm^2 \approx 1.232 \times 10^{10}$, the longest time it took was 7.133s, while (extrapolating and assuming that the trend line relationship still holds up to >12 billion) we would expect it to take $\approx$ 615.7s. Taking into account all data, we get a weak trend line (not shown) with R-squared of 0.4746, but this trend line is "weighed down" by the low runtimes of the high nm^2 elections. Considering only the elections with $nm^2 < 300$ million, we get a trend line of $t = 5.122 \times 10^{-9} nm^2 + 1.490 \times 10^{-2}$ with an R-squared of 0.7063.

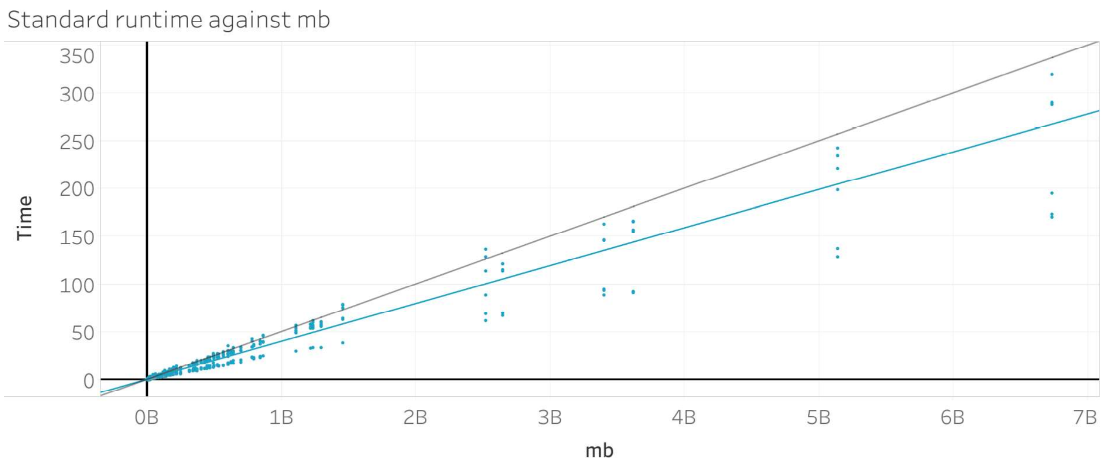
- We can see that the relationship between runtime and mb for the standard knapsack algorithm on real data is stronger than the relationship between the reversed knapsack runtime and nm^2, and that the algorithm performs similarly well on real data compared to random data. We get a trend line of $t = 3.971 \times 10^{-8}mb + 4.550 \times 10^{-2}$ with an R-squared of 0.9380.

- We can see that the standard knapsack algorithm performs significantly worse on real data compared to the reversed knapsack algorithm. This contrasts with the performance of the two algorithms on random data. On average, the standard knapsack algorithm took 42.01 times longer on the same election.

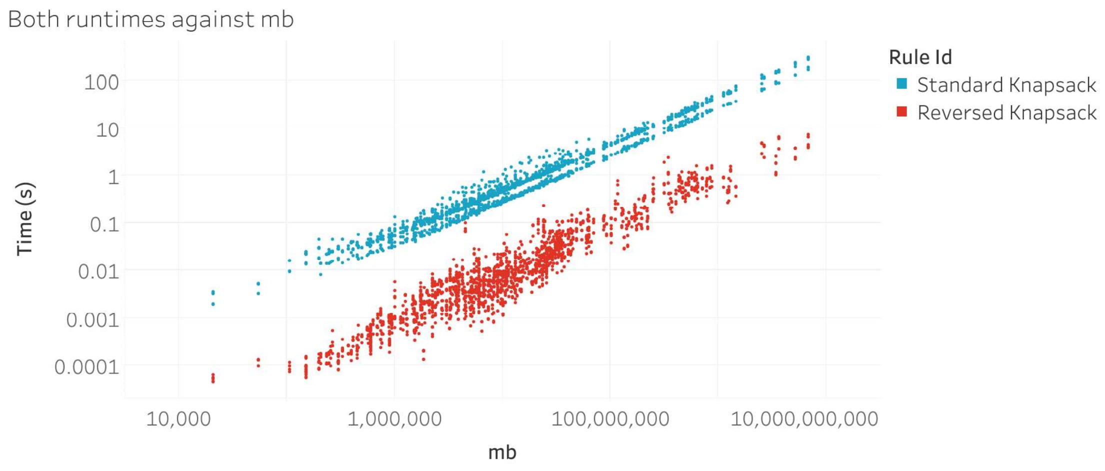
Overall, the practical performance of the reversed knapsack algorithm is significantly better than the standard knapsack algorithm on (partially) real-world elections, and the runtime is well within what would be acceptable for deployment within a government.



(a) The runtime of the reversed knapsack algorithm against nm^2. The trend line obtained from the random data for the reversed knapsack algorithm is shown in grey.



(b) The runtime of the standard knapsack algorithm against nm^2. The trend line obtained from the random data for the standard knapsack algorithm is shown in grey.



(c) Runtimes of both the standard and reversed knapsack algorithms on log-log plot. Log-log plot used due to wide variation in order of magnitudes of data.

Figure 3.4: Runtimes of both the standard and reversed knapsack algorithms compared to mb and nm^2 on real data.

3.1.3 Extensions

Now we shall discuss some extensions of the model.

3.1.3.1 Projects without location

We can also allow some projects to have no specific location associated with them. For example, we may have a project to improve the frequency of household rubbish collection within the city, in which case implementing this project does not take up physical space. To include these types of project into consideration alongside the projects with locations, we can extend the total interval to the left and place the non-located projects into this extension before the located projects in such a way that none of the non-located projects overlap with any of the other projects. The location knapsack algorithm will then function identically to the standard knapsack algorithm on these non-located projects, building up the dynamic programming table in the same fashion. Then once we have reached the projects with locations, the algorithm functions as required.

3.1.3.2 Multiple intervals

If we have multiple independent intervals (*i.e.* there are no project conflicts between projects on different intervals) that share the same total budget, then we can also solve this case by appending one interval after another so that no project from one interval overlaps with another project in a different interval. Then, by running the algorithm on this larger interval, we get the desired result.

3.1.3.3 Temporal conflicts

This model could also be applied to temporal conflicts on the interval. In this scenario, $\ell(x)$ could represent the time in which the project will be implemented rather than the space. An example of such a scenario would be a community garden whose members wish to vote on the crops to be grown in the garden throughout the year. Certain crops can only be grown at certain times of the year and different crops may have different maturity periods. The cost of seeds and maintenance for each crop may also differ, so each crop may have different costs. In this scenario, we can use the above model to vote on the crops to grow and maximise the approval of the voters. This model assumes that we can only implement one project at any point on the interval, so if we can implement multiple projects at the same time (*i.e.* in the example above, if we can grow different plants at the same time), then this model will not necessarily find an optimal solution in that scenario.

3.2 Multiple possible project locations

We now turn our attention to a modification of the model in which projects can have multiple possible locations at which they could be implemented, instead of one single specific location.

Definition 3.16. (Partition)
A set $\pi = \{\rho_1, \dots, \rho_r\}$ is a *partition* of the set C if:

- $\emptyset \notin \pi$,
- $\bigcup_{i=1}^{r} \rho_i = C$,
- and $\forall 1 \leq i, j \leq r, [\rho_i \cap \rho_j = \emptyset]$.

We say that ρ_i is a *part* of π.

Definition 3.17. (Multi-instance PB with interval conflicts (MPBIC) election)
A *multi-instance PB with interval conflicts (MPBIC) election*

$$E = (V, C, \mathrm{cost}, b, A, \ell, \pi)$$

is a PBIC problem which additionally has a partition π of C such that $\forall \rho \in \pi$ and $\forall x, y \in \rho$, we have $\mathrm{cost}(x) = \mathrm{cost}(y)$, and for all voters $v \in V$, we have

$x \in A(v) \iff y \in A(v)$. We interpret the candidates as instances of different projects ρ, so π represents the set of projects, and each instance in ρ is a possible location for the project ρ. Let $\text{part}(p)$ be the part ρ of π with $p \in \rho$. For MPBIC elections we define valid winning sets as

$$\mathcal{V}(E) = \{W \in \mathcal{V}(E') : \forall p_1, p_2 \in W, p_1 \neq p_2 \implies \text{part}(p_1) \neq \text{part}(p_2)\},$$

where $E' = (V, C, \text{cost}, b, A, \ell)$ and so $\mathcal{V}(E')$ is the set of valid winning sets of E as interpreted just as a PBIC election. We require that in the winning set, no two winning instances are of the same project type.

Remark 3.18. Clearly, every PBIC is an MPBIC: if we let $\pi = \{\{x\} : x \in C\}$, every project type has only one instance.

Definition 3.19. (Conflict graph of a MPBIC election)
Let E be an MPBIC election with a candidate set C, location function ℓ, and partition π. Let

$$F_1 = \{(x_i, x_j) \subseteq C^2 : x_i \neq x_j \text{ and } \ell(x_i) \cap \ell(x_j) \neq \emptyset\},$$

$$F_2 = \{(x_i, x_j) \subseteq C^2 : x_i \neq x_j \text{ and } \exists \rho \in \pi, \{x_i, x_j\} \subseteq \rho\}.$$

Then we define the *conflict graph* of E to be the graph $G = (C, F_1 \cup F_2)$.

MPBIC elections cannot, in general, be solved by Algorithm 1 or Algorithm 2. Consider the example in Figure 3.5. The superscript denotes the project type, and the subscript denotes the instance number of that project type. The instances sorted by endpoint are in order $P = [I_1^1, I_1^2, I_2^2, I_2^1]$. In the last stage of the algorithms, we will decide whether I_2^1 is included in the optimal solution or not. If it is not, then the optimal utility is the same as the utility achievable from the first 3 instances. However, if it is, then the optimal utility is the utility we would get from I_2^1, plus the utility we would achieve from just I_1^2, but *not including* I_1^1, since

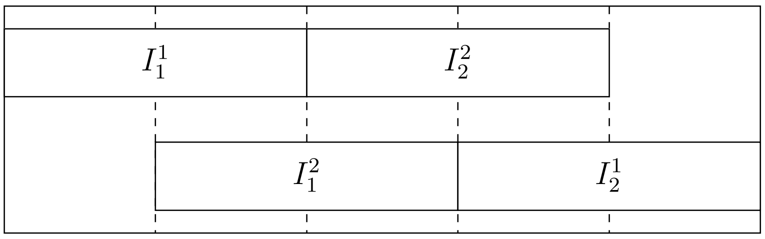


Figure 3.5: An example MPBIC problem where the algorithms in Section 3.1 fail.

otherwise we would have a conflict between the two instances of Project 1. In this way, allowing projects to have multiple possible locations breaks the linearity of project conflicts. By this, we mean that it breaks the property that for every instance I, every instance J that has its endpoint to the left of I's endpoint and conflicts with I exists within a contiguous subarray of P (the instances sorted by endpoint) that ends at the project before I in P. In the example in Figure 3.5, the instances that conflict with I_2^1 are both I_2^2 and I_1^1, so the conflicting instances do not form a contiguous subarray of P.

Utilitarian welfare optimisation in the above setting is again a form of conflicted knapsack problem. The conflict graphs generated are interval graphs where, additionally, the vertices of the graph have been partitioned and where edges are added to make the induced subgraph of each partition a complete graph.

As stated before, the pseudo-polynomial time algorithms demonstrated by Pferschy and Schauer (2009) deal with two special cases of the conflict graph, one for the case where the conflict graph is chordal, and one where the conflict graph has bounded treewidth. We shall now show that the conflict graphs generated when we allow multiple locations per project fall into neither of these two classes.

First, we present a straightforward proof that the conflict graphs in MPBIC problems are not chordal.

Lemma 3.20. *The conflict graphs of MPBIC problems are, in general, not chordal.*

Proof. Consider the example in Figure 3.5. The conflict graph is a simple 4-cycle. The only induced cycle is the 4-cycle which has more than 3 vertices, so the

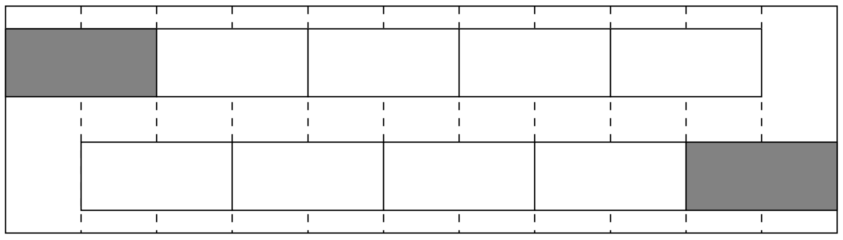


Figure 3.6: An example MPBIC problem that demonstrates how to create arbitrarily large non-chordal conflict graphs.

conflict graph is not chordal. Arbitrary g-cycle conflict graph MPBIC problems can be generated from g instances that overlap in a "bricklaying" pattern, with the first and last instance being of the same project type. See Figure 3.6 for an example. ■

We shall now show that conflict graphs generated with multiple possible locations per project also do not have bounded treewidths, except in the very limited case outlined in Lemma 3.29. To prove this, we need to define a family of graphs that we will show later exist as graph minors in the conflict graphs.

Definition 3.21. (Graph minor)
Let G be a graph. Then H, a graph, is a *graph minor* of G if H can be formed from G through a sequence of edge contractions, edge deletions, and vertex deletions of G.

Definition 3.22. (Hexagonal grid graph)
A *t-hexagonal grid graph* is a graph $H = (V, E)$ constructed as follows. Let $V^{i,j} := \{v_r^{i,j} : 1 \leq r \leq 6\}$ for $0 \leq i, j < t$. Identify the following vertices as the same (where they exist):

$$v_1^{i,j} = v_3^{i-1,j} = v_5^{i,j+1} \text{ for even } j;$$

$$v_2^{i,j} = v_4^{i,j+1} = v_6^{i+1,j+1} \text{ for even } j;$$

$$v_1^{i,j} = v_3^{i-1,j} = v_5^{i-1,j+1} \text{ for odd } j;$$

$$v_2^{i,j} = v_4^{i-1,j+1} = v_6^{i,j+1} \text{ for odd } j.$$

Let

$$E^{i,j} = \{\{v_s^{i,j}, v_r^{i,j}\} : s = r + 1 \mod 6\}.$$

Finally, let $V = \bigcup_{0 \le i,j < t} V^{i,j}$, and let $E = \bigcup_{0 \le i,j < t} E^{i,j}$. See Figure 3.8 for an illustration of the construction of the graph, and Figure 3.7 for an overall illustration of an 8-hexagonal grid graph. We shall refer to subgraph $H^{i,j} = (V^{i,j}, E^{i,j})$ as *hexagon* (i, j). We say that a hexagon is *internal* if every vertex in $V^{i,j}$ is identified with 2 other vertices of other hexagons.

We shall define a square grid in a similar way. Note that for other definitions of a t-square grid, t may refer to the number of vertices within each row and column of vertices rather than the number of squares in each row and column of squares, so a t-square grid here may be referred to as a $(t + 1)$-square grid elsewhere. We use this definition to be consistent with our definition of a hexagonal grid graph.

Definition 3.23. (Square grid graph)
A *t-square grid graph* is defined as a graph $S = (V, E)$ constructed as follows. Let $V^{i,j} := \{v_r^{i,j} : 1 \le r \le 4\}$ for $0 \le i, j < t$. Identify the following vertices as the same (where they exist):

$$v_1^{i,j} = v_2^{i-1,j} = v_3^{i-1,j+1} = v_4^{i,j+1}.$$

Let

$$E^{i,j} = \{\{(v_s^{i,j}, v_r^{i,j}) : s = r + 1 \mod 4\}\}.$$

Then let $V = \bigcup_{0 \le i,j < t} V^{i,j}$, and let $E = \bigcup_{0 \le i,j < t} E^{i,j}$. See Figure 3.9 for an illustration of the construction of the square grid. We shall refer to subgraph $S^{i,j} = (V^{i,j}, E^{i,j})$ as *square* (i, j). We say that a square is *internal* if every vertex in $V^{i,j}$ is identified with 3 other vertices of other squares.

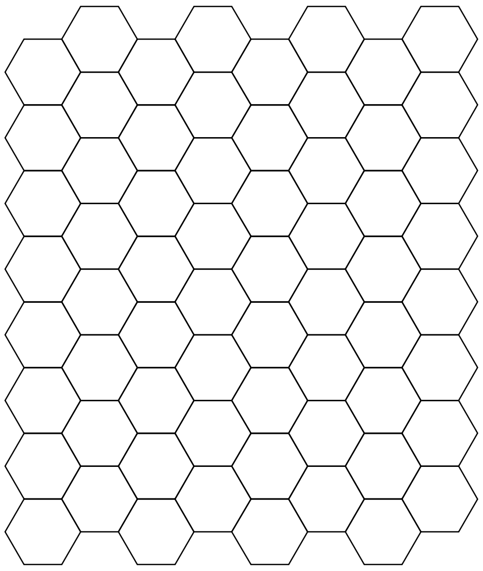


Figure 3.7: $t \times t = 8 \times 8$ hexagonal grid.

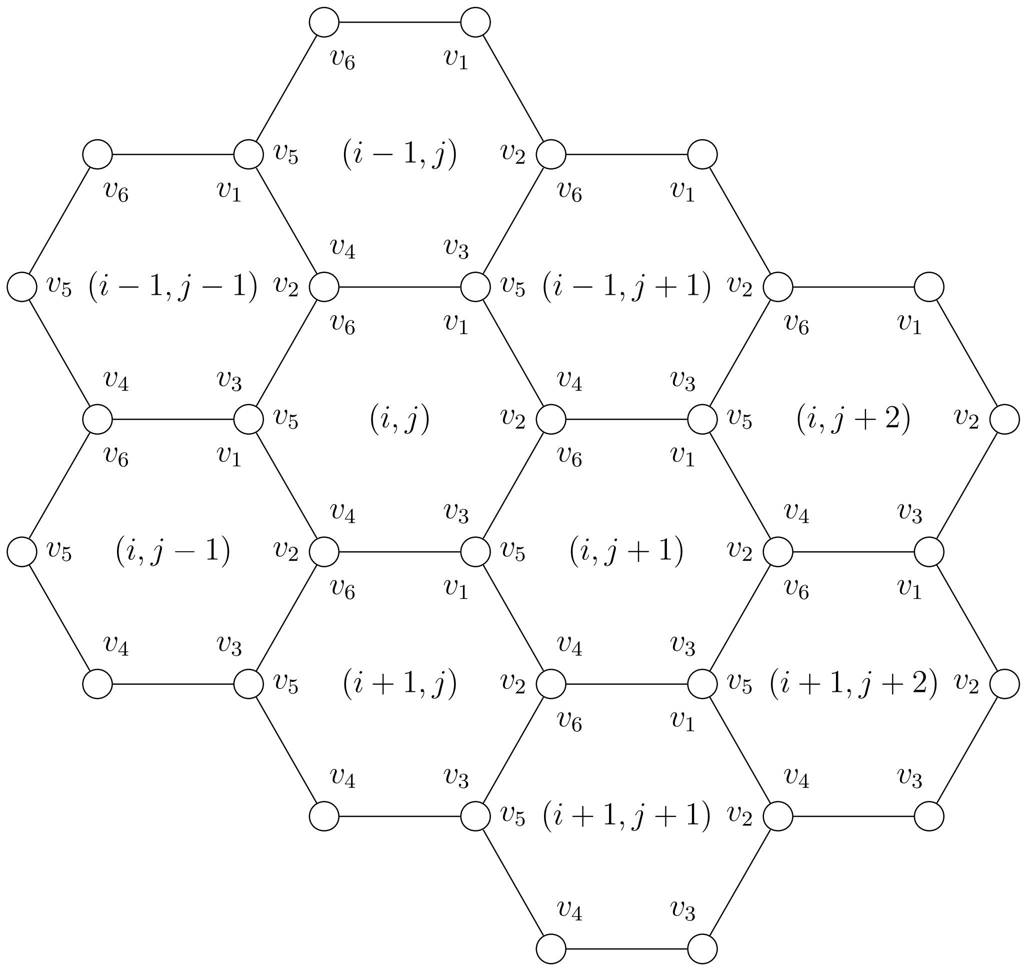


Figure 3.8: Illustration of the construction of the hexagon grid. Here, j is odd.

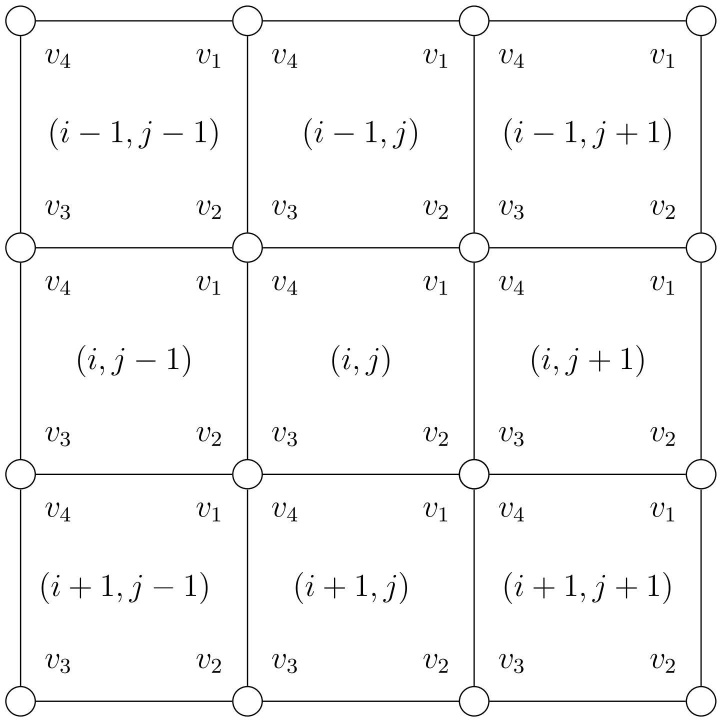


Figure 3.9: Illustration of the construction of the square grid.

Lemma 3.24. *A t-hexagonal grid has $2t(t+2)$ vertices and $3t^2+4t-1$ edges.*

Proof. Consider each column of hexagons, where column j is defined as the induced subgraph $H[\cup_{i=1}^{t} V^{i,j}]$. The concatenation of a column of hexagons at one end of the grid would add $2t+1$ new vertices to the graph, where we identify the new concatenated vertices to be consistent with Definition 3.22. A graph of a single column of hexagons would have $5t+1$ vertices, and we can think of the grid as a single column of hexagons with $t-1$ columns sequentially concatenated to it. Therefore, the total number of vertices in the t-grid is $5t+1+(t-1)(2t+1) = 2t^2-t-1+5t+1 = 2t^2+4t = 2t(t+2)$ vertices.

Now the number of edges in the grid is less than $6t^2$ since each hexagon is associated with at most 6 edges to the graph and there are t^2 hexagons; however, some of these edges are counted twice. Every edge that belongs to two hexagons is counted twice, and the only edges that are not counted twice are the edges along the exterior of the grid. Every column with $1 \leq j < t-1$ has four edges on the exterior of the grid, and the two columns at the end of the grid have $2t+3$ edges each that are on the exterior. Therefore, there are $2 \times (2t+3) + 4(t-2) = 8t-2$ exterior edges. If we add this to $6t^2$, we guarantee that we count each edge exactly twice. Therefore, the total number of edges in the graph is $\frac{6t^2+8t-2}{2} = 3t^2+4t-1$ edges. ■

Now, we shall use the square grid to obtain a lower bound on the treewidth of the hexagonal grid. Cygan et al. (2015) provide the following lemma.

Lemma 3.25. *The treewidth of a t-square grid is $t+1$.*

Proof. See Exercise 7.37 in the textbook (Cygan et al., 2015). ■

From this lemma, we can obtain the following lower bound on the treewidth of a t-hexagonal grid.

Lemma 3.26. *The treewidth of a t-hexagonal grid is at least $t+1$.*

Proof. Within the t-hexagonal grid, for every hexagon $H^{i,j}$, contract the edges $(v_1^{i,j}, v_2^{i,j})$ and $(v_5^{i,j}, v_6^{i,j})$ if j is odd, and the edges $(v_2^{i,j}, v_3^{i,j})$ and $(v_4^{i,j}, v_5^{i,j})$ if j is even. Note that for an internal hexagon $H^{i,j}$, each edge e of $H^{i,j}$ we contract also belongs to another adjacent hexagon $H^{x,y}$, but if j is odd, then y will be even (and vice versa), so the contraction of e in $H^{x,y}$ will be the same as the contraction of e in $H^{i,j}$. We only contract the edge once. Referring to Figure 3.8 helps to demonstrate this.

The remaining graph is a $(t+1)$-square grid graph. Every hexagon $H^{i,j}$ becomes a square $S^{i,j}$; we contract exactly 2 edges in each hexagon $H^{i,j}$. Without loss of generality, assume j is odd. For a hexagon with odd j, we contract $(v_1^{i,j}, v_2^{i,j})$ and $(v_5^{i,j}, v_6^{i,j})$, and for hexagons $H^{i-1,j}$ and $H^{i+1,j}$ we do not contract the edges they share with $H^{i,j}$. For the other 4 hexagons that share edges with $H^{i,j}$, $j+1$ and $j-1$ are both even, so those hexagon's contracted edges exactly correspond to the contracted edges for $H^{i,j}$. This holds because of the identification of the vertices between adjacent hexagons, so for example $(v_1^{i,j}, v_2^{i,j}) = (v_5^{i-1,j+1}, v_4^{i-1,j+1})$. This can be seen in Figure 3.8. Therefore, each odd j hexagon has exactly two contracted edges, and the same holds for even j hexagons. For internal hexagons, this is equivalent to contracting the edges shared between adjacent hexagons $H^{i,j}, H^{x,y}$ where both $i \neq x$ and $j \neq y$.

For odd j, relabel the vertices:

$$v_{1/2}^{i,j} \to s_1^{i,j};$$

$$v_3^{i,j} \to s_2^{i,j};$$

$$v_4^{i,j} \to s_3^{i,j};$$

$$v_{5/6}^{i,j} \to s_4^{i,j}.$$

For even j, relabel:

$$v_1^{i,j} \to s_1^{i,j};$$

$$v_{2/3}^{i,j} \to s_2^{i,j};$$

$$v_{4/5}^{i,j} \to s_3^{i,j};$$

$$v_6^{i,j} \to s_4^{i,j}.$$

We use e.g. $v_{1/2}$ because vertices v_1 and v_2 are now identified as the same vertex due to the edge between v_1 and v_2 being contracted. The edges of each square are preserved, since we contract two edges in each hexagon; we just need to show that every vertex is identified with the correct vertices.

For odd j, $j+1$ is even, and so by using the above relabelling, and the identification of the vertices in the hexagonal grid, we have that

$$s_1^{i,j} = v_1^{i,j} = v_3^{i-1,j} = s_2^{i-1,j};$$

$$s_1^{i,j} = v_1^{i,j} = v_5^{i-1,j+1} = s_3^{i-1,j+1};$$

$$s_1^{i,j} = v_2^{i,j} = v_6^{i,j+1} = s_4^{i,j+1}.$$

For even j, $j+1$ is odd, and we have

$$\begin{aligned}
s_1^{i,j} &= v_1^{i,j} = v_3^{i-1,j} = s_2^{i-1,j};\\
s_1^{i,j} &= v_1^{i,j} = v_3^{i-1,j} = v_2^{i-1,j} = v_4^{i-1,j+1} = s_3^{i-1,j+1};\\
s_1^{i,j} &= v_1^{i,j} = v_5^{i,j+1} = s_4^{i,j+1}.
\end{aligned}$$

Therefore, every hexagon $H^{i,j}$ becomes square $S^{i,j}$, so the t-square grid is a graph minor of the t-hexagonal grid. The treewidth of the t-square grid is $t+1$ by Lemma 3.25, and so by the paper (Bodlaender and Koster, 2011), Lemma 14, which states that the treewidth of a graph is at least as large as the treewidth of any of its graph minors, we find that the treewidth of the t-hexagonal grid graph is at least $t+1$. ■

Now that we have shown that the t-hexagonal grid graph has a treewidth of $t+1$, we will show that we can construct a family of MPBIC elections where hexagonal grid graphs exists as graph minors of the conflict graphs of the elections.

Theorem 3.27. *There exists a family of MPBIC elections where each project can have at most 2 possible instances, and each instance can overlap with at most 2 other instances, such that the treewidths of the conflict graphs of the elections in this family is unbounded.*

Proof. Let $t \in \mathbb{N}$. We shall construct an MPBIC problem that will have the t-hexagonal grid as a graph minor of its conflict graph. This election will have $m = 6t(t+2)$ instances in total, with every project having 2 instances each (so each project has 2 possible locations), and every instance will overlap with at most 2 other instances.

Let H be a t-hexagonal grid graph. H has $2t(t+2)$ vertices by Lemma 3.24. Divide the interval into $2t(t+2)$ distinct and non-overlapping sections and index each section by a vertex of H, so we say that the section corresponding to v starts

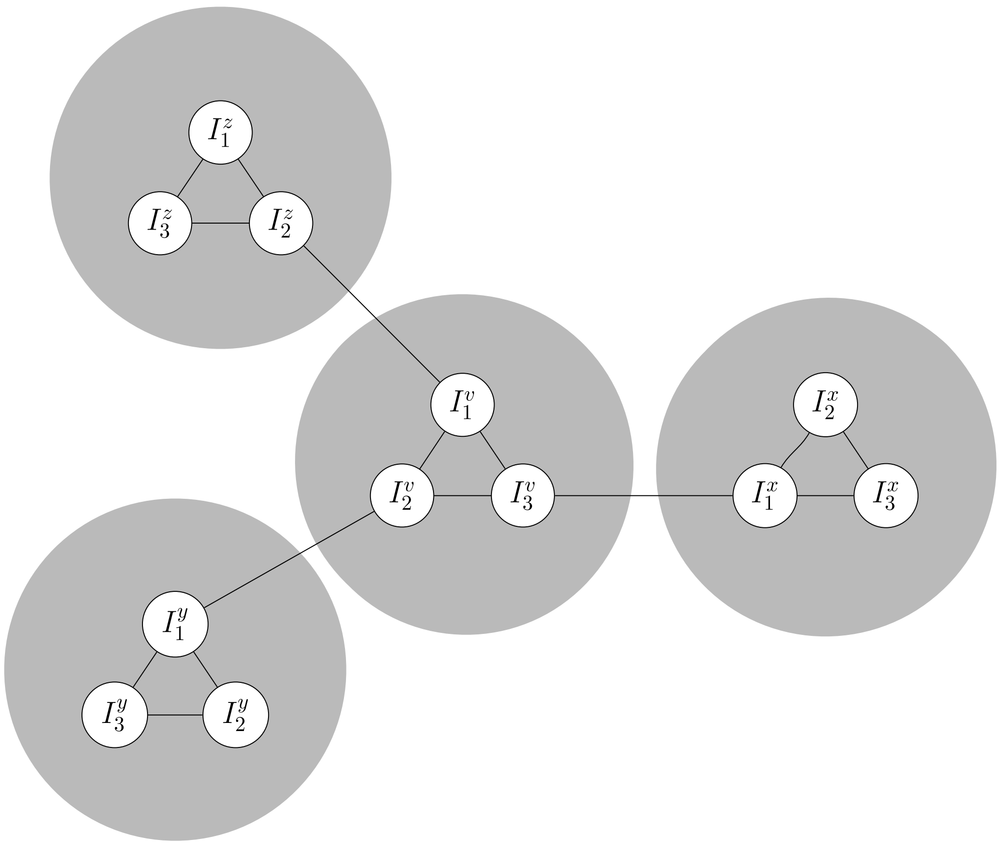


Figure 3.10: A "close up" of a neighbourhood of the conflict graph, with each small white vertex corresponding to an instance of the election, and each large grey vertex corresponding to a section of the interval, and a vertex of the hexagonal grid.

at $s(v)$ and ends at $e(v)$. For each vertex v of H, we shall construct 3 instances, I_1^v, I_2^v, I_3^v. These instances shall be placed at the v'th section of the interval, and so will occupy $[s(v), e(v))$. For each edge (u, v) in H, we will select instances I_i^v and I_j^u that have not already been selected and identify them as the same project type, so we add a part $\{I_i^v, I_j^u\}$ to π. Since each vertex of H has degree at most 3, there will always exist an unselected instance for each vertex when we need to select an instance for an edge.

A subsection of the conflict graph is shown in Figure 3.10 as an example. Every instance I conflicts with exactly the instances placed in the same section, and with the instance that is of the same project type as I. In the conflict graph, if we contract the edges corresponding to the conflict among instances placed in the same section, we get exactly the t-hexagonal grid graph. From Lemma 3.26, the treewidth of the hexagonal grid is at least $t + 1$ and again by the paper (Bodlaender and Koster, 2011), Lemma 14, the treewidth of the conflict graph is at least $t + 1 = \mathcal{O}(\sqrt{m})$. ■

We can prove a similar result for the case when each instance can overlap with at most 1 other instance.

Theorem 3.28. *There exists a family MPBIC elections where each project can have at most 3 possible instances, and each instance can overlap with at most 1 other instance, such that the treewidths of the conflict graphs of the elections in the family are unbounded.*

Proof. The proof is similar to the proof of Theorem 3.27. We allow each project to have at most 3 instances each, and for each instance to overlap with at most 1 other instance. The vertices of the hexagonal grid now correspond to projects, so the triangle formed at each vertex of the hexagon will be derived from the conflict subgraph of the 3 instances of the same project type. We divide the interval into $3t^2+4t-1$ sections, one for each edge of the hexagonal grid. We index the vertices of the hexagonal grid by project types, so for a vertex v, we can refer to project type v. To derive the edges (x, y) of the hexagonal grid, we place at each section an instance of project type x and an instance of project type y. Because the maximum degree of the hexagonal grid graph is 3, we will always have enough instances of a project to do this. Therefore, we can similarly find the t-hexagonal grid minor within the conflict graph, and so the treewidth of the conflict graph is at least $t + 1 = \mathcal{O}(\sqrt{m})$. ■

We have therefore shown that, for both of the cases above, the treewidths of the graphs can be unbounded. Therefore, the bounded treewidth algorithm we discussed earlier will not run in polynomial time in the size of the input election, as the runtime is exponential in the treewidth. For the sake of completeness, we note that the final case remaining is trivial.

Lemma 3.29. *If we allow at most 2 different locations per project and allow every instance to overlap with at most 1 other instance, we can solve the problem of utilitarian welfare optimisation in polynomial time.*

Proof. Every instance can now only conflict with at most 2 other instances; the other instance of the same project type and the instance it overlaps with. So, every vertex in the conflict graph has degree at most 2, and so every component of the conflict graph is either a simple path or a simple cycle.

The paths have treewidth 1 since they are trees. The cycles have treewidth at most 2; see Exercise 7.8(d) of the textbook (Cygan et al., 2015), but a proof is provided here for completeness. Select some vertex x in the cycle and order the edges $e_1, \dots, e_r$ by traversing the cycle, starting with an edge adjacent to x. Create a tree decomposition by creating a path with vertices labelled $\{x\} \cup e_i$ for each $1 \leq i \leq r$ in order. The maximum size of each label is 3, every vertex and edge appears in some label, and any vertex appears only in connected labels; x is in every label, and any other vertex only appears in the two labels corresponding to its two edges, which are connected to each other in the tree decomposition.

We can then create a tree decomposition of the whole conflict graph by selecting an arbitrary component's tree decomposition to start, and arbitrarily connecting the other tree decompositions to the starting decomposition, using a single edge to ensure that the resultant decomposition is a tree. The resultant tree decomposition is, in fact, a tree decomposition by the fact that each component is disconnected from every other component. This tree decomposition still has treewidth at most 2, since each label still has at most 3 vertices in it. We then apply the bounded treewidth algorithm provided by Pferschy and Schauer (2009) to this tree decomposition. This procedure runs in $\mathcal{O}(m^3n^2)$ time, which is polynomial in the size of the input election. ■

Upon a final wider search of the literature, the paper (Spieksma, 1999) was uncovered. Whilst it was not directly identified in the initial literature search with the relevant search terms, it has proven to contain much relevant information about the MPBIC problem studied here. Spieksma (1999) explores a similar problem within interval scheduling. The problem they formulate is as follows: given n k-tuples of intervals

$$\{(I_1^1, I_2^1, \ldots, I_k^1), (I_1^2, I_2^2, \ldots, I_k^2), \ldots, (I_1^n, I_2^n, \ldots, I_k^n)\}$$

on the real line, we must select as many intervals as possible such that at any point on the real line, no two selected intervals intersect, and no two selected intervals are in the same tuple. Spieksma (1999) cites from personal communication that the problem of deciding whether there exists a solution with at least t intervals selected is $\mathcal{NP}$-complete for $k \geq 2$, and further shows that the problem of maximising the number of intervals does not have a PTAS unless $\mathcal{P} = \mathcal{NP}$, for any $k \geq 2$. In particular, this means that solving the problem exactly cannot be done in polynomial time, as otherwise this polynomial time algorithm could be used as a PTAS. The utilitarian welfare optimisation problem for an MPBIC election is a more general form of the problem formulated here: let

- $V = \{1, \ldots, n\}$;
- $C = \{I_y^x : 1 \leq x \leq n, 1 \leq y \leq k\}$;
- $\text{cost}(p) = 1$;
- $b = n$;
- $A(v) = \{I_y^v : 1 \leq y \leq k\}$;

- $\ell(I) = I$;
- $\pi = \{\{I_y^x : 1 \leq y \leq k\} : 1 \leq x \leq n\}$.

Then we obtain an MPBIC election where the problem of maximising utilitarian welfare is equivalent to the interval scheduling problem. This is because the value of each project here is 1, so we are trying to just maximise the number of projects implemented. The cost of each project is also 1, and the budget is n, so the budget will never limit the number of projects implemented. Therefore, utilitarian welfare optimisation for an MPBIC election also does not admit a PTAS unless $\mathcal{P} = \mathcal{NP}$, and similarly the decision problem of deciding if a utilitarian welfare value of at least t is achievable is $\mathcal{NP}$-hard.

Chapter 4

Participatory budgeting with partial information

A government does not know the preferences and wishes of the citizens it represents unless it has elicited those preferences. Elections are a vital part of the democratic process, as they provide a structured way for all eligible citizens to express their preferences to their government. However, they can be expensive and logistically challenging to conduct for the government. In addition to eliciting voter preferences through elections, governments also employ polls and surveys to get a broad idea of the views of the electorate. In this chapter, we shall consider the scenario in which we do not have full information about the preferences of all voters. Instead, we shall only be allowed to sample a subset of voters to elicit their preferences, and then run an election algorithm on the subset of preferences to obtain a winning set. We shall provide bounds on the number of citizens we are required to sample to achieve certain fairness properties with respect to the whole population.

Of course, it is unreasonable to expect us to be able to achieve perfect representation with probability 1 without sampling the whole population. If we consider the problem of achieving an MWV-JR winning set, there may be some 1-cohesive group such that all members approve only a single candidate p, and such that the group has size exactly $\frac{n}{k}$. Then any MWV-JR winning set must contain p, however, if we take a sample of $n-1$ voters where the excluded voter is a member of this group (which can occur with probability $\frac{n}{k}$), and then run an MWV-JR algorithm on the sampled voters' preferences, we will not necessarily achieve an MWV-JR winning set, since $\frac{n}{k} - 1 < \frac{n-1}{k}$. Instead, we can try to achieve approximate representation with high probability. The problem then becomes finding the number of voters we need to sample to provide approximate representation for the whole population, given values for the desired probability of error and precision of the approximation.

4.1 Multiwinner voting with partial information

First, we shall focus only on the MWV scenario.

Definition 4.1. Let $G \subseteq V$. Define $\mathcal{B}(G) = \{A(v) : v \in G\}$ as the set of ballots B such that there exists a voter v in G with $A(v) = B$. Define $G^{\uparrow}$ as the set of voters for which there exists some voter in G with the same ballot type, so $G^{\uparrow} = \{v \in V : A(v) \in \mathcal{B}(G)\}$. Note that if G is a q-cohesive group, then so is $G^{\uparrow}$.

Definition 4.2. (Subelection)
For an election $E = (V, C, k, A)$ with a set of voters V and an approval function A, for a subset $S \subseteq V$, define a *subelection* $E[S] = (S, C, k, A')$ where A' is A restricted to the domain S.

We will also extend our definition of a cohesive group, so that we can capture the idea of approximate representation.

Definition 4.3. ((q, ε)-cohesive group)
We say a group G is (q, ε)-*cohesive* (for election E) for some $1 \leq q \leq k$ if $|G| \geq (1+\varepsilon)\frac{qn}{k}$ and $|A(G)| \geq q$. Define

$$\varepsilon\text{-}\mathcal{C}_q(E) = \{G \subseteq V : |G| \geq (1+\varepsilon)\frac{qn}{k} \text{ and } |A(G)| \geq q\}$$

for $1 \leq q \leq k$ and $\varepsilon > 0$ as the set of (ε, q)-cohesive groups of E.

Remark 4.4. Clearly $0\text{-}\mathcal{C}_q(E) = \mathcal{C}_q(E)$. For $\varepsilon_1, \varepsilon_2 \geq 0$ with $\varepsilon_1 \leq \varepsilon_2$, we have $\varepsilon_1\text{-}\mathcal{C}_q(E) \subseteq \varepsilon_2\text{-}\mathcal{C}_q(E)$.

Definition 4.5. (Monotonic justified representation)
Let $\mathcal{R}$ be a fairness property. Then $\mathcal{R}$ is a form of *monotonic justified representation* if for every $k \geq 1$, there exists $1 \leq U \leq k$ such that for every MWV election E to select k candidates, for every $1 \leq q \leq U$, and for every $T \in \mathcal{V}(E)$, there exists a family of sets of voters $\mathcal{G}(E, T, q) \subseteq \mathcal{C}_q(E)$ such that

$$\mathcal{R}(E) = \{W \in \mathcal{V}(E) : \forall 1 \leq q \leq U, \mathcal{C}_q(E) = \mathcal{G}(E, W, q)\}$$

and where $\forall 1 \leq q \leq U, \forall G \in \mathcal{C}_q(E), \forall T \in \mathcal{V}(E)$, we have:

- $G \notin \mathcal{G}(E, T, q) \implies G^{\uparrow} \notin \mathcal{G}(E, T, q)$;
- $\forall S \subseteq V, \left[G^{\uparrow} \notin \mathcal{G}(E, T, q) \implies G^{\uparrow} \cap S \notin \mathcal{G}(E[S], T, q)\right]$.

For a form of monotonic justified representation $\mathcal{R}$, we can then define a fairness property $\varepsilon\text{-}\mathcal{R}$, where

$$\varepsilon\text{-}\mathcal{R}(E) = \{W \in \mathcal{V}(E) : \forall 1 \leq q \leq U, \varepsilon\text{-}\mathcal{C}_q(E) \subseteq \mathcal{G}(E, W, q)\}.$$

We shall now motivate these definitions. Monotonic justified representation attempts to capture the idea that cohesive groups deserve representation, where the precise idea of what it means for a group to be represented is different for each fairness property. The family of groups of voters $\mathcal{G}(E, T, q)$ is the set of q-cohesive groups that would be represented if T was selected as a winning set. Therefore $W \in \mathcal{R}(E)$ if and only if every cohesive group is represented with W. The first additional constraint, $G \notin \mathcal{G}(E, T, q) \implies G^{\uparrow} \notin \mathcal{G}(E, T, q)$ for $G \in \mathcal{C}_q(E)$, captures the idea that if G is not represented by T, but is q-cohesive, then simply making the group bigger without changing the type of ballots present in the group won't make the group represented. This allows us to upper bound the size of an unrepresented group by assuming that every voter with a ballot type present in G is in G. The second additional constraint, $\forall S \subseteq V, \big[G^{\uparrow} \notin \mathcal{G}(E,T,q) \implies G^{\uparrow} \cap S \notin \mathcal{G}(E[S],T,q)\big]$, states that if $G^{\uparrow}$ is not represented, then there is no subelection in which the restriction of $G^{\uparrow}$ to that subelection is represented with respect to the subelection. If we assume T satisfies $\mathcal{R}$ with respect to the subelection $E[S]$, this will allow us to show that the restriction of $G^{\uparrow}$ to the set S must be too small to form a cohesive group for $E[S]$, and thus allow us to make conclusions about the size of G.

Now, we shall motivate ε-$\mathcal{R}$. First, note that $\mathcal{R}(E) \subseteq \varepsilon\text{-}\mathcal{R}(E)$. For $\mathcal{R}$ a form of monotonic justified representation, suppose that for an election E it holds that W satisfies ε-$\mathcal{R}$ but not $\mathcal{R}$. Then there is some group G that is q-cohesive, such that $G \notin \mathcal{G}(E,W,q)$. Therefore, G cannot be (q,ε)-cohesive since W satisfies ε-$\mathcal{R}$, and so if G were (q,ε)-cohesive, we would have that $G \in \mathcal{G}(E,W,q)$. Therefore, $\frac{qn}{k} \leq |G| < \frac{qn}{k}(1+\varepsilon)$. This means that if a $\frac{\varepsilon}{1+\varepsilon}(< \varepsilon)$ proportion of the group decide to change their preferences away from the groups preferences, then this group is no longer cohesive. Therefore, if W satisfies ε-$\mathcal{R}$ but not $\mathcal{R}$, the only cohesive groups that W does not represent are somewhat unstable, and so in this way, W approximately represents the cohesive groups of the election.

Lemma 4.6. MWV-JR, MWV-PJR *and* MWV-EJR *are all forms of monotonic justified representation.*

Proof. Let E be an MWV election. For MWV-JR and MWV-EJR, let

$$\mathcal{G}(E,T,q) = \{G \in \mathcal{C}_q(E) : \exists v \in G \text{ such that } |A(v) \cap T] \geq q]\}.$$

Then for $\mathcal{R} =$ MWV-JR, we have $U = 1$, and $\mathcal{R} =$ MWV-EJR, we have $U = k$. By Definitions 2.10 and 2.12, we have $W \in \mathcal{R}(E)$ iff for every $1 \leq q \leq U$ and every q-cohesive group G, $G \in \mathcal{G}(E,W,q)$. Now, let $1 \leq q \leq U$ and let G be a q-cohesive group with $G \notin \mathcal{G}(E,T,q)$. Then we have $\forall v \in G, |A(v) \cap T| < q$. So, let $v \in G^{\uparrow}$. There exists a voter $u \in G$ such that $A(v) = A(u)$. But $|A(u) \cap T| < q$ so $|A(v) \cap T| < q$ for any $v \in G^{\uparrow}$, and so we have $G^{\uparrow} \notin \mathcal{G}(E, T, q)$. Now, if $G^{\uparrow} \notin \mathcal{G}(E, T, q)$, then $\forall v \in G^{\uparrow}, |A(v) \cap T| < q$, but then certainly $\forall v \in G^{\uparrow} \cap S$, $|A(v) \cap T| < q$. So $G^{\uparrow} \cap S \notin \mathcal{G}(E[S], T, q)$, regardless of whether $G^{\uparrow} \cap S$ is large enough to form a q-cohesive group for $E[S]$. Therefore MWV-JR and MWV-EJR are both monotonic justified representations.

For MWV-PJR, let

$$\mathcal{G}(E, T, q) = \left\{ G \in \mathcal{C}_q(E) : \left| \bigcup_{v \in G} A(v) \cap T \right| \geq q \right\}.$$

Then for $U = k$, by Definition 2.11, we have W satisfies MWV-PJR(E) iff for every $1 \leq q \leq U$, for every q-cohesive group G with respect to E, $G \in \mathcal{G}(E, W, q)$. Let $1 \leq q \leq U$ and let G be a q-cohesive group with respect to E. Suppose $G \notin \mathcal{G}(E, T, q)$. We have $|\bigcup_{v \in G} A(v) \cap T| < q$. So then $\bigcup_{v \in G} A(v) = \bigcup_{v \in G^{\uparrow}} A(v)$ since for every $v \in G^{\uparrow}$, there exists a voter $u \in G$ such that $A(v) = A(u)$. Therefore, $|\bigcup_{v \in G^{\uparrow}} A(v) \cap T| < q$ and $G^{\uparrow} \notin \mathcal{G}(E, T, q)$. Also, if $|\bigcup_{v \in G^{\uparrow}} A(v) \cap T| < q$, then $|\bigcup_{v \in G^{\uparrow} \cap S} A(v) \cap T| < q$, so we have that if $G^{\uparrow} \notin \mathcal{G}(E, T, q)$ then

$$G^{\uparrow} \cap S \notin \mathcal{G}(E[S], T, q),$$

regardless of whether $G^{\uparrow} \cap S$ is large enough to form a q-cohesive group for $E[S]$. Therefore, MWV-PJR is also a form of monotonic justified representation. ■

Now that we have fully defined monotonic justified representation, and shown that MWV-JR, MWV-PJR, and MWV-EJR are all forms of monotonic justified representation, we shall now provide a general upper bound on the sample size required for approximate representation for forms of monotonic justified representation. First, let us define the hypergeometric distribution.

Definition 4.7. (Hypergeometric distribution)

The *hypergeometric distribution* is a probability distribution with parameters N, the population size, K, the number of success states of the population, and n, the number of draws. For X distributed with a hypergeometric distribution, the probability mass function of X is given by

$$\Pr(X = k) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}},$$

for $0 \leq k \leq \min(K, n)$. We require $0 \leq K \leq N$ and $0 \leq n \leq N$. We can interpret X as being the number of success states in a random draw of n states from a whole population of N, where the number of success states in the whole population is K. It models sampling without replacement.

Theorem 4.8. *Let $\mathcal{R}$ be a form of monotonic justified representation. Let E be an MWV election. Suppose that we have an algorithm $\mathcal{A}$ that guarantees $\mathcal{R}$, and we can sample uniformly (without replacement) from V. Then we can construct a sampling process that achieves ε-$\mathcal{R}$ for E with probability at least $1 - \delta$ for any $\varepsilon, \delta > 0$, using a number of samples polynomial in $\ln \frac{1}{\delta}$, $\frac{1}{\varepsilon}$, and k, and independent of n, but potentially exponential in m.*

Proof. Let $\mathcal{R}$ be a form of monotonic justified representation. Let $k \geq 1$ and let U be as defined in the definition of monotonic justified representation. Let $E = (V, C, k, A)$ be some MWV election to select k candidates. Let $\alpha(G) = |\mathcal{B}(G)|$, and let

$$Q(T) = \{G \subseteq V : \exists 1 \leq q \leq U[G \in \varepsilon\text{-}\mathcal{C}_q(E) \text{ and } G \notin \mathcal{G}(E, T, q)]\};$$

$$\alpha = \max_{T \in \mathcal{V}(E)} \max_{G \in Q(T)} \alpha(G).$$

$\alpha(G)$ is the number of different ballots present in group G. $Q(T)$ is the set of (ε, q)-cohesive groups such that $G \notin \mathcal{G}(E, T, q)$. So then α is an upper bound of the number of ballots present in (ε, q)-cohesive groups that are not represented by winning set T, over all possible valid sets T.

Let $B \subseteq C$ be a ballot type. Let $z_X(B) = |\{v \in X : A(v) = B\}|$ be the number of voters in the set $X \subseteq V$ who vote with ballot type B. Let

$$f = |\{B \subseteq C : z_V(B) > 0\}|$$

be the number of different types of ballots present in the population. Trivially, $f \leq 2^m$. The sampling process is as follows:

1. Take a uniform sample S without replacement from V of size $s \geq \frac{k^2\alpha^2 \ln\left(\frac{f}{\delta}\right)}{2\varepsilon^2}$;
2. Apply $\mathcal{A}$ to the election $E[S]$ to get a winning set of candidates W.

We shall now show that this achieves ε-$\mathcal{R}$ for E with probability at least $1-\delta$ for any $\varepsilon, \delta > 0$. For every $B \subseteq C$, $z_S(B)$ is distributed with a hypergeometric distribution with population size n, $z_V(B)$ success states, and with a draw size of s. Let $t = \frac{\varepsilon}{k\alpha}$. Hoeffding (1963) provides a tail bounds on the hypergeometric distribution, so we get

$$\Pr\left(z_S(B) \leq \left(\frac{z_V(B)}{n} - t\right) \cdot s\right) \leq e^{-2t^2 s}.$$

By the union bound, we have that

$$\Pr\left(\text{for some } B \subseteq C,\, z_S(B) \leq \left(\frac{z_V(B)}{n} - t\right) \cdot s\right) \leq f e^{-2t^2 s}.$$

For the moment, suppose that we have that for every $B \subseteq C$,

$$z_S(B) > \left(\frac{z_V(B)}{n} - t\right) \cdot s,$$

so $z_V(B) < n\left(\frac{z_S(B)}{s} + t\right)$. Let G be some (q, ε)-cohesive group for some $1 \leq q \leq U$.

Suppose to the contrary that $G \notin \mathcal{G}(E, W, q)$. We have that

$$\begin{aligned}
|G| &\leq \sum_{B \in \mathcal{B}(G)} z_V(B) && (4.1)\\
&< \sum_{B \in \mathcal{B}(G)} n \cdot \left(\frac{z_S(B)}{s} + t \right) \\
&= nt|\mathcal{B}(G)| + \frac{n}{s} \sum_{B \in \mathcal{B}(G)} z_S(B) \\
&< nt\alpha + \frac{n}{s} q \frac{s}{k} && (4.2)\\
&\leq n \frac{\varepsilon}{k\alpha} \alpha + q \frac{n}{k} \\
&\leq \varepsilon q \frac{n}{k} + q \frac{n}{k} \\
&= (1 + \varepsilon) q \frac{n}{k}
\end{aligned}$$

which is a contradiction since G is (q, ε)-cohesive so $|G| \geq (1 + \varepsilon) q \frac{n}{k}$, so we must have $G \in \mathcal{G}(E, W, q)$.

Equation (4.1) holds because G certainly cannot contain more voters than all voters who vote with the same type of ballot as some voter in G. Equation (4.2) holds as follows: we have

$$\sum_{B \in \mathcal{B}(G)} z_S(B) = |\{v \in S : A(v) \in \mathcal{B}(G)\}| = |G^{\uparrow} \cap S|.$$

We have that $G \notin \mathcal{G}(E, W, q)$, and so by the two additional constraints on monotonic justified representation, $G^{\uparrow} \notin \mathcal{G}(E, W, q)$ and $G^{\uparrow} \cap S \notin \mathcal{G}(E[S], W, q)$. Since $\mathcal{A}$ guarantees $\mathcal{R}$, we have that $\mathcal{A}(E[S]) \in \mathcal{R}(E[S])$, and so for any $1 \leq q \leq U$ and q-cohesive group G' with respect to $E[S]$, $G' \in \mathcal{G}(E[S], W, q)$. We have that $G^{\uparrow} \cap S \notin \mathcal{G}(E[S], W, q)$, so $G^{\uparrow} \cap S$ is not a q-cohesive group for $E[S]$. But we have $|A'(G^{\uparrow} \cap S)| \geq |A(G)| \geq q$ (assuming $G^{\uparrow} \cap S \neq \emptyset$), so we must have that $|G^{\uparrow} \cap S| < \frac{qs}{k}$. If $G^{\uparrow} \cap S = \emptyset$, we again have $|G^{\uparrow} \cap S| < \frac{qs}{k}$. Therefore, if for every $B \subseteq C, z_S(B) > \left(\frac{z_V(B)}{n} - t \right) \cdot s$, then we achieve ε-$\mathcal{R}$.

So then the probability that we do not achieve ε-$\mathcal{R}$ is at most

$$\leq fe^{-2t^2 s} = fe^{-2\frac{s\varepsilon^2}{k^2\alpha^2}} = fe^{-2\frac{\frac{k^2\alpha^2\ln\left(\frac{f}{\delta}\right)}{2\varepsilon^2}\varepsilon^2}{k^2\alpha^2}} = \delta.$$

So therefore, we achieve ε-$\mathcal{R}$ with probability at least $1-\delta$. ■

We can provide bounds for α in the case of MWV-JR, MWV-PJR and MWV-EJR.

Lemma 4.9. *For* MWV-JR, MWV-PJR*, we have that* $\alpha \leq 2^{m-k-1}$. *For* MWV-EJR*, we have that* $\alpha \leq 2^{m-1}$.

Proof. Let E be an MWV election and $T \in \mathcal{V}(E)$. Here, take $\mathcal{G}(E,T,q)$ to be relative to the fairness property considered. First, consider MWV-JR, and a $(1,\varepsilon)$-cohesive group $G \notin \mathcal{G}(E,T,1)$. Then for a voter to be in this group, their ballot must approve $A(G)$ and disapprove of T. $|A(G)| \geq 1$ and $|T| = k$, so the number of different ballots that satisfy this is at most $2^{m-(k+1)}$ since the approval and disapproval of $k+1$ of the candidates is fixed.

Now, consider MWV-PJR and a (q,ε)-cohesive group $G \notin \mathcal{G}(E,T,q)$. Let $X = \bigcup_{v\in G} A(v) \cap T$. Then, for a voter to be in G, they must approve every candidate in $A(G)$ and disapprove every candidate in $T \setminus X$. We have that $|T \setminus X| \geq k-(q-1)$, and so there are at most $2^{m-(q+k-(q-1))} = 2^{m-k-1}$ such ballots.

Finally, consider MWV-EJR and a (q,ε)-cohesive group $G \notin \mathcal{G}(E,T,q)$. Every voter approves of $A(G)$ and approves of less than q of the winning set T. We want to count the number of different ballots that could possibly exist in G. Consider a voter v and say that the voter approves exactly r' of the candidates in T for $|A(G)\cap T| \leq r' \leq q-1$. Then the voter approves $r = r' - |A(G)\cap T|$ of the candidates in $T \setminus A(G)$. There are $\binom{|T\setminus A(G)|}{r}$ ways to select r candidates in

$T \setminus A(G)$ to approve, and we know then that all other candidates in $T \setminus A(G)$ are disapproved by v. We have then fixed the approval/disapproval of

$$|A(G)| + r + (|T \setminus A(G)| - r) = |A(G)| + |T \setminus A(G)| = |A(G) \cup T|$$

of the candidates, and the rest of the candidates can be approved or disapproved freely without affecting G's membership of $\mathcal{G}(E, T, q)$. Therefore, the number of ballots in G is at most

$$\begin{aligned}
\sum_{r=0}^{q-1-|A(G)\cap T|} \binom{|T \setminus A(G)|}{r} 2^{m-|T\cup A(G)|} &\leq \sum_{r=0}^{q-1} \binom{k}{r} 2^{m-|T\cup A(G)|} \\
&< 2^{m-|T\cup A(G)|} \sum_{r=0}^{k-1} \binom{k}{r} \\
&\leq 2^{m-(k+q-(q-1))}(2^k - 1) \\
&< 2^{m-1}
\end{aligned}$$

■

We can in fact provide a better bound on the number of people we need to sample for MWV-JR.

Lemma 4.10. *Let E be an MWV election. Suppose that we have an algorithm $\mathcal{A}$ that guarantees* MWV-JR*, and we can sample uniformly (without replacement) from V. Then we can construct a sampling process that guarantees ε-*MWV-JR *for E with probability at least $1 - \delta$ for any $\varepsilon, \delta > 0$, using a number of samples polynomial in $\ln \frac{1}{\delta}$, $\frac{1}{\varepsilon}$, m, and k, and independent of n.*

Proof. The sampling process is as follows:

1. Take a uniform sample S without replacement from V of size
$$s \geq \frac{k^2 \ln\left(\frac{m\binom{m-1}{k}}{\delta}\right)}{2\varepsilon^2};$$
2. Apply $\mathcal{A}$ to the election $E[S]$ to get a winning set of candidates W.

Let $D_T(p) = \{v \in V : p \in A(v), A(v) \cap T = \emptyset\}$ be the set of voters who approve of a candidate p and do not approve any candidate in $T \in \mathcal{V}(E)$. For $p \in T$, we have $D_T(p) = \emptyset$. Similarly to above, we have that for every fixed $p \in C, T \subseteq C \setminus \{p\}$, $|D_T(p) \cap S|$ is distributed with a hypergeometric distribution with population size n, $D_T(p)$ success states, and s draws. Let $t = \frac{\varepsilon}{k}$. Again by the paper (Hoeffding, 1963), we have a tail bound on the distribution,

$$\Pr\left(|D_T(p) \cap S| \leq \left(\frac{|D_T(p)|}{n} - t\right) \cdot s\right) \leq e^{-2t^2 s}.$$

For $p \in T$ we have $|D_T(p)| = 0$ with certainty, so

$$\Pr\left(|D_T(p) \cap S| \leq \left(\frac{|D_T(p)|}{n} - t\right) \cdot s\right) = 0$$

in this case. So by the union bound, we have that

$$\Pr\left(\text{for some } T \subseteq C, p \in C,\ |D_T(p) \cap S| \leq \left(\frac{|D_T(p)|}{n} - t\right) \cdot s\right) \leq m\binom{m-1}{k} e^{-2t^2 s}.$$

Suppose for the moment that we do not have the above event, so for all $T \subseteq C, p \in C, |D_T(p) \cap S| > \left(\frac{|D_T(p)|}{n} - t\right) \cdot s$. Then certainly for W, we have that $|D_W(p) \cap S| > \left(\frac{|D_W(p)|}{n} - t\right) \cdot s$ for every $p \in C$. Then let G be a $(1, \varepsilon)$-cohesive group. Suppose to the contrary that no voter in G approves of some project in W (so considering MWV-JR as a form of monotonic justified representation, $G \notin \mathcal{G}(E, W, 1)$). Then for all $p \in A(G)$, we have $G \subseteq D_W(p)$. So therefore, for any arbitrary $p \in A(G)$, we have

$$\begin{aligned} |G| &\leq |D_W(p)| \\ &< n \cdot \left(\frac{|D_W(p) \cap S|}{s} + \frac{\varepsilon}{k}\right) \\ &< n \cdot \left(\frac{\frac{s}{k}}{s} + \frac{\varepsilon}{k}\right) \\ &= \frac{n}{k}(1 + \varepsilon). \end{aligned} \tag{4.3}$$

Equation (4.3) holds; if $D_W(p) \cap S = \emptyset$, then $|D_W(p) \cap S| < \frac{s}{k}$. Otherwise, if $D_W(p) \cap S$ is a 1-cohesive group with respect to $E[S]$, we would have that there exists some voter $v \in D_W(p) \cap S$ with $|A(v) \cap W| \geq 1$, since $\mathcal{A}$ guarantees MWV-JR, and so $W \in$ MWV-JR$(E[S])$. However, if $|A(v) \cap W| \geq 1$, then $v \notin D_W(p)$, since v does not approve of any candidate in W, so therefore by contradiction, $D_W(p) \cap S$ cannot be 1-cohesive. We have that $A(D_W(p) \cap S) \supseteq A(D_W(p)) \supseteq \{p\}$, and so we must have that $|D_W(p) \cap S| < \frac{s}{k}$.

Therefore, we have that $|G| < \frac{n}{k}(1+\varepsilon)$, and so G is not $(1, \varepsilon)$-cohesive. Therefore since we have derived a contradiction, it must be the case that some voter in G approves of some candidate in W, and so if for all $T \subseteq C, p \in C$,

$$|D_T(p) \cap S| > \left(\frac{|D_T(p)|}{n} - t\right) \cdot s,$$

then we achieve ε-MWV-JR. The probability that we have for all $T \subseteq C$ and for all $p \in C$, $|D_T(p) \cap S| > \left(\frac{|D_T(p)|}{n} - t\right) \cdot s$ is at least $1 - m\binom{m-1}{k}e^{-2t^2 s}$. We have that

$$m\binom{m-1}{k}e^{-2t^2 s} \leq m\binom{m-1}{k}e^{-2\varepsilon^2/k^2 \frac{k^2 \ln\left(\frac{m\binom{m-1}{k}}{\delta}\right)}{2\varepsilon^2}} = \delta$$

and so we achieve ε-MWV-JR with probability $\geq 1 - \delta$. ■

Remark 4.11. The quantity $\ln\left(\frac{m\binom{m-1}{k}}{\delta}\right) = \ln\left(m\binom{m-1}{k}\right) + \ln\frac{1}{\delta}$ is polynomial in k and m; by Stirling's bounds on factorials, we have $m\binom{m-1}{k} \leq \left(\frac{(m-1)e}{k}\right)^k m$, so $\ln\left(m\binom{m-1}{k}\right) \leq \ln m + k \ln\left(\frac{e(m-1)}{k}\right) = \mathcal{O}(k \log m)$.

We shall finally look at approximately maximising utilitarian welfare within the MWV context.

Theorem 4.12. *Let E be an MWV election. Suppose that we know that the average proportion of candidates that voters approve of (out of the m total candidates) is at least r. Suppose that we have an algorithm $\mathcal{A}$ that guarantees exact utilitarian welfare optimisation, and we can sample uniformly (without replacement) from*

V. Then we can construct a sampling process that achieves an ε approximation of the optimal utilitarian welfare for E with probability at least $1-\delta$ for any $\varepsilon, \delta > 0$, using a number of samples polynomial in $\ln \frac{1}{\delta}$, $\frac{1}{\varepsilon}$ and $\frac{1}{r}$, logarithmic in m, and independent of k and n.

Proof. The sampling process is the same as previously, with sample size $s \geq \frac{2\ln\frac{2m}{\delta}}{\varepsilon^2 r^2}$. Let W be the result of this process. Let $D(p) = \{v \in V : p \in A(v)\}$ for $p \in C$. Let **OPT** be the maximum possible utilitarian welfare achievable by a valid winning set W^*. We have that the total number of approvals given across all projects is at least rmn. So, the top k most approved candidates must receive in total at least krn approvals by the pigeonhole principle, so we have $\mathbf{OPT} \geq krn$.

Let $X_p = |D(p) \cap S)|$ where S is the random sample set taken in the sampling process. Then X_p is distributed with a hypergeometric distribution with population size n, $|D(p)|$ success states, and s draws. By the paper (Hoeffding, 1963), we again have that

$$\Pr\left(\left|\frac{X_p}{s} - \frac{|D(p)|}{n}\right| \geq t\right) \leq 2e^{-2t^2 s}.$$

Similar to before, we then have that

$$\Pr\left(\text{For some } p \in C,\ \left|\frac{X_p}{s} - \frac{|D(p)|}{n}\right| \geq t\right) \leq 2me^{-2t^2 s}.$$

Let $t = \frac{r\varepsilon}{2}$. With probability at least $1 - 2me^{-2t^2 s}$, we see that for all $p \in C$, $\left|\frac{X_p}{s} - \frac{|D(p)|}{n}\right| < t$, so if this event holds, we have

$$\begin{aligned}
\sum_{v \in V} |A(v) \cap W| &= \sum_{p \in W} |D(p)| \\
&\geq \sum_{p \in W} \frac{n}{s}\left(X_p - ts\right) \\
&= -tnk + \frac{n}{s}\sum_{p \in W} X_p \\
&\geq -tnk + \frac{n}{s}\sum_{p \in W^*} X_p \qquad (4.4)
\end{aligned}$$

$$
\begin{aligned}
&\geq -tnk + \frac{n}{s} \sum_{p \in W^*} \frac{s}{n} \left(|D(p)| - ts \right) \\
&\geq -2tnk + \sum_{p \in W^*} |D(p)| \\
&\geq \mathbf{OPT} - 2tnk \\
&= \mathbf{OPT} - 2\frac{r\varepsilon}{2} nk \\
&= \mathbf{OPT} - \varepsilon rnk \\
&\geq \mathbf{OPT} \left(1 - \varepsilon \right).
\end{aligned}
$$

Here, Equation (4.4) is true because W was chosen to maximise the utilitarian welfare with respect to the subelection $E[S]$, and $\sum_{p \in W} X_p$ is the total utility obtained from W in $E[S]$, so it must be at least as much as the utility obtained from W^* with respect to $E[S]$.

So, if we have that for all $p \in C$, $\left| \frac{X_p}{s} - \frac{|D(p)|}{n} \right| < t$, then we achieve an ε approximation of the optimal utilitarian welfare. We have that the probability of this not occurring is at most

$$\leq 2me^{-2t^2 s} = 2me^{-\frac{r^2 \varepsilon^2 s}{2}},$$

and so since $n \geq \frac{2 \ln \frac{2m}{\delta}}{\varepsilon^2 r^2}$, we have that

$$2me^{-\frac{r^2 \varepsilon^2 s}{2}} \leq 2me^{-\frac{r^2 \varepsilon^2 \frac{2 \ln \frac{2m}{\delta}}{\varepsilon^2 r^2}}{2}} = 2me^{-\ln \frac{2m}{\delta}} = \delta.$$

Therefore, we achieve an ε approximation of optimal utilitarian welfare with probability at least $1 - \delta$. ■

Remark 4.13. We can make the modest assumption that every voter approves of at least one candidate to get $r \geq \frac{1}{m}$. This gives us a sampling bound of $s \geq \frac{2m^2 \ln \frac{2m}{\delta}}{\varepsilon^2}$.

4.2 Participatory budgeting with partial information

We shall now show similar bounds for the standard PB scenario without conflicts.

Definition 4.14. For an election $E = (V, C, \text{cost}, b, A)$ with a voter set V and an approval function A, for a subset $S \subseteq V$, define a *subelection*

$$E[S] = (S, C, \text{cost}, b, A')$$

where A' is A with domain restricted to S.

First, we look at PB-EJR, as defined in Definition 2.14. We will similarly extend our definition of a T-cohesive group.

Definition 4.15. ((T, ε)-cohesive group)
We say a group G is (T, ε)*-cohesive* with respect to election E for some $T \subseteq C$ if $|G| \geq (1+\varepsilon)\frac{\text{cost}(T)n}{b}$ and $T \subseteq A(G)$. Let ε-$\mathcal{C}_T$ be the set of (T, ε)-cohesive groups.

We then define ε-PB-EJR and ε-PB-PJR similarly to the definition of PB-EJR and PB-PJR from Definition 2.14 and Definition 2.16 with (T, ε)-cohesive groups substituting in for T-cohesive groups. Explicitly,

$$\varepsilon\text{-PB-EJR}(E) = \{W \in \mathcal{V}(E) : \forall T \subseteq C, \forall G \in \varepsilon\text{-}\mathcal{C}_T, \exists v \in G, |W \cap A(v)| \geq |T|\};$$

$$\varepsilon\text{-PB-PJR}(E) = \{W \in \mathcal{V}(E) : \forall T \subseteq C, \forall G \in \varepsilon\text{-}\mathcal{C}_T, |\bigcup_{v \in G} A(v) \cap W| \geq |T|\}.$$

We then get the following theorem.

Theorem 4.16. *Let E be a PB election. Suppose we have an algorithm $\mathcal{A}$ that guarantees* PB-EJR*, and we can sample uniformly (without replacement) from V. Let $\mu = \min_{p \in C} \text{cost}(p)$. Then we can construct a sampling process that achieves* ε-PB-EJR *for E with probability at least $1 - \delta$ for any $\varepsilon, \delta > 0$, using a number of samples polynomial in $\ln \frac{1}{\delta}, \frac{1}{\varepsilon}$, and $\frac{b}{\mu}$, and independent of n, but potentially exponential in m.*

Proof. Let

$$\alpha = \max\{|\mathcal{B}(G)| : G \text{ is } (T,\varepsilon)\text{-cohesive and } \forall v \in G, |A(v) \cap W| < |T|\}.$$

Let $z_X(B) = |\{v \in X : A(v) = B\}|$ be the number of voters in the set $X \subseteq V$ who vote with ballot type B. Let $f = |\{B \subseteq C : z_V(B) > 0\}|$. The process is as follows:

1. Take a uniform sample S without replacement from V of size $s \geq \frac{\alpha^2 b^2 \ln \frac{f}{\delta}}{2\varepsilon^2\mu^2}$;
2. Apply $\mathcal{A}$ to the set of ballots collected from S to get a winning set of candidates W.

We shall now show that this guarantees ε-PB-EJR with probability at least $1-\delta$ for any $\varepsilon, \delta > 0$. For every $B \subseteq C$, $z_S(B)$ is again distributed with a hypergeometric distribution with population size n, $z_V(B)$ success states, and s draws. By the same argument as before, we have that

$$\Pr\left(\text{for some } B \subseteq C,\, z_S(B) \leq \left(\frac{z_V(B)}{n} - t\right) \cdot s\right) \leq f e^{-2t^2 s}.$$

Let $t = \frac{\varepsilon\mu}{\alpha b}$. For the moment, suppose that for every $B \subseteq C$, $z_S(B) > \left(\frac{z_V(B)}{n} - t\right) \cdot s$, so $z_V(B) < n\left(\frac{z_S(B)}{s} + t\right)$. Let G be some (T,ε)-cohesive group. Suppose to the contrary that for every voter $v \in G$ we have $|A(v) \cap W| < |T|$. We have that

$$\begin{aligned}
|G| &\leq \sum_{B \in \mathcal{B}(G)} z_V(B) \\
&< \sum_{B \in \mathcal{B}(G)} n\left(\frac{z_S(B)}{s} + t\right) \\
&= nt|\mathcal{B}(G)| + \frac{n}{s}\sum_{B \in \mathcal{B}(G)} z_S(B) \\
&< nt\alpha + \frac{n}{s}\frac{s}{b}\mathrm{cost}(T) \qquad (4.5) \\
&\leq n\frac{\varepsilon\mu}{\alpha b}\alpha + \frac{n}{b}\mathrm{cost}(T) \\
&\leq n\frac{\varepsilon \mathrm{cost}(T)}{b} + \frac{n}{b}\mathrm{cost}(T) \\
&= (1+\varepsilon)\frac{n}{b}\mathrm{cost}(T)
\end{aligned}$$

which is a contradiction since G is (T,ε)-cohesive so $|G| \geq (1+\varepsilon)\frac{n}{b}\text{cost}(T)$, so we must have that there exists some voter $v \in G$ with $|A(v) \cap W| \geq |T|$. Equation (4.5) holds by similar arguments to Theorem 4.8; we have that

$$\sum_{B \in \mathcal{B}(G)} z_S(B) = |G^{\uparrow} \cap S|.$$

If for every voter in G, we have that $|A(v) \cap W| < T$, then for every voter v in $G^{\uparrow}$, $|A(v) \cap W| < |T|$ and so for every voter $v \in G^{\uparrow} \cap S, |A(v) \cap W| < |T|$. Since $\mathcal{A}$ guarantees PB-EJR, $W = \mathcal{A}(E[S]) \in \text{PB-EJR}(E[S])$. Therefore, if $G^{\uparrow} \cap S$ were T-cohesive for $E[S]$, we would have that there exists some voter v in $G^{\uparrow} \cap S$ such that $|A(v) \cap W| \geq |T|$. Therefore, $G^{\uparrow} \cap S$ is not T-cohesive for $E[S]$. But $T \subseteq A(G) \subseteq A(G^{\uparrow} \cap S)$, so we have $|G^{\uparrow} \cap S| < \frac{\text{cost}(T)s}{b}$. Therefore since we derived a contradiction, for every (T,ε)-cohesive group G, there exists a voter $v \in G$ with $|A(v) \cap W| \geq |T|$. So if for every $B \subseteq C, z_S(B) > \left(\frac{z_V(B)}{n} - t\right) \cdot s$, then we achieve ε-PB-EJR. The probability that it is not the case that for every $B \subseteq C, z_S(B) > \left(\frac{z_V(B)}{n} - t\right) \cdot s$ is at most

$$\leq fe^{-2t^2 s} = fe^{-2\frac{\varepsilon^2\mu^2}{\alpha^2 b^2}s} = fe^{-2\frac{\varepsilon^2\mu^2}{\alpha^2 b^2}\frac{\alpha^2 b^2 \ln\frac{f}{\delta}}{2\varepsilon^2\mu^2}} = \delta,$$

so therefore we achieve ε-PB-EJR with probability at least $1-\delta$. ■

A sampling process for ε-PB-PJR can be achieved similarly.

Corollary 4.17. *Let E be a PB election. Suppose that we have an algorithm $\mathcal{A}$ that guarantees* PB-PJR*, and we can sample uniformly (without replacement) from V. Let $\mu = \min_{p \in C}(\text{cost}(p))$. Then we can construct a sampling process that achieves ε-*PB-PJR *for E with probability at least $1-\delta$ for any $\varepsilon, \delta > 0$, using a number of samples polynomial in $\ln\frac{1}{\delta}, \frac{1}{\varepsilon}$, and $\frac{b}{\mu}$, and independent of n, but potentially exponential in m.*

Proof. By a similar proof, we can achieve identical bounds on s. This time, we assume to the contrary that there exists a (T, ε)-cohesive group such that $|\bigcup_{v \in G} A(v) \cap W| < |T|$. In this case, we still have $\sum_{B \in \mathcal{B}} z_S(B) < \frac{s}{b}\text{cost}(T)$: if $|\bigcup_{v \in G} A(v) \cap W| < |T|$, then $|\bigcup_{v \in G^\uparrow} A(v) \cap W| < |T|$ and $|\bigcup_{v \in G^\uparrow \cap S} B \cap W| < |T|$. Also, $T \subseteq A(G) = A(G^\uparrow) \subseteq A(G^\uparrow \cap S)$. Since $\mathcal{A}$ guarantees PB-PJR,

$$W = \mathcal{A}(E[S])) \in \text{PB-PJR}(E[S]).$$

Therefore, if $G^\uparrow \cap S$ formed a T-cohesive group for $E[S]$ we would have

$$|\bigcup_{v \in G^\uparrow \cap S} B \cap W| \geq |T|,$$

so it cannot be a T-cohesive group and so since $T \subseteq A(G^\uparrow \cap S)$, we have that $|G^\uparrow \cap S| < \frac{s}{b}\text{cost}(T)$. Finally, $\sum_{B \in \mathcal{B}} z_S(B) = |G^\uparrow \cap S|$. ■

4.2.1 Empirical Results

We now look at evaluating these bounds in relation to a real instance from Pabulib (Stolicki et al., 2020). We shall attempt to answer 2 questions:

- In practice, do the proven sampling bounds actually achieve the ε, δ representation?
- If so, how tight are the bounds?

We shall focus on a specific case of PB election in which every voter approves of exactly one project, which we shall call a single-approval election. We focus on this case because of the 458 elections available on Pabulib at the time of testing, 142 of the 458 elections (31.0%) were single-approval elections, and in particular, 13 of the 20 largest elections by number of voters were single-approval elections, so this election format is widely used in practice. Under this assumption, a T-cohesive group can only exist for $|T| = 1$, since for any group G, $|A(G)| \leq 1$. T-cohesive groups, then, are subsets of the set of voters who approve of the single project in T. W satisfies PB-EJR if for every T-cohesive group G, there exists a voter $v \in G$ such that $|A(v) \cap W| \geq |T|$, but $|T| = 1$ and $A(v) = T$ for all $v \in G$. So W satisfies PB-EJR if for any $\{p\}$-cohesive group, $p \in W$ but this is equivalent to having for all $p \in C$,

$$|\{v \in V : \{p\} = A(v)\}| \geq \frac{n}{b}\text{cost}(p) \implies p \in W,$$

since any $\{p\}$-cohesive group is a subset of $\{v \in V : \{p\} = A(v)\}$.

From this we can make two observations. First, we can construct a very simple polynomial time algorithm to achieve PB-EJR in this setting. We iterate over the projects p, and if the number of voters that approve p is at least $\frac{n}{b}\text{cost}(p)$, then we add p to W. The resultant W is, in a sense, minimal. That is to say that any winning set W' satisfies PB-EJR if and only if $W \subseteq W'$ since adding additional projects to W does not affect whether or not it satisfies PB-EJR, and every candidate p of W must be included in W', since the group of all voters who approve p forms a cohesive group of sufficient size. Second, in this scenario we have significantly lower bounds on f and α. We know that we have at most m different ballots present in the population, and $\alpha = 1$.

Let us focus on a specific real-world election. We shall look at the election titled "Poland Wrocław 2015 From 500" within Pabulib. This data corresponds to the 2015 PB election held in the Polish city of Wrocław. In this election, the projects were split by cost into 3 groups: the set of projects costing less than 150,000 PLN, the set of projects costing between 150,000 PLN and 500,000 PLN, and the set of projects costing at least 500,000 PLN (Bednarska-Olejniczak and Olejniczak, 2016). The total budget for the whole election was 20,000,000 PLN, but for our purposes, we shall assume that the budget is only for allocation to the projects costing at least 500,000 PLN, so we shall treat this project price group as a standalone election. In this election, 171 projects were considered, a total of 127,773 voters voted, and using this price group, we have that the minimum project cost is 530,000 PLN. This election is a single-approval election (each voter could only vote for at most one project in each price group), so for our sampling bounds, we can use $\alpha = 1$ and $f = m = 171$. We shall use $\delta = 5\%$ for demonstration and analyse the results we achieve for varying values of ε.

Using the above election parameters, we will choose samples of size

$$s = \left\lceil \frac{20000000^2 \ln(\frac{171}{0.05})}{2\varepsilon^2 500000^2} \right\rceil = \left\lceil \frac{800 \ln(3420)}{\varepsilon^2} \right\rceil$$

for appropriate values of ε. We see that for $\varepsilon < \sqrt{\frac{800\ln(3420)}{n}} \approx 0.2257$, the theoretical sample size we need is actually larger than the total population size. The smallest value of ε we shall use will be 0.25 and we shall increase ε by 0.05 up to $\varepsilon = 5$. For each value of ε, we shall generate 900 samples and apply our basic EJR algorithm to the sample to get a winning set W. Given W, we can then use the whole population to calculate the simulated error value ε^* achieved. That is to say, for each project $p \notin W$, we can check if the number of voters of p is enough for the voters of p to form a cohesive group, and if so, we can then compute

$$\varepsilon_p = \frac{b|G|}{\text{cost}(p)n} - 1$$

where $G =$ is the set of voters of p, to get the true error value for each project, and then $\varepsilon^* = \max_{p \in C \setminus W} \varepsilon_p$ which is the overall true error value.

The minimal single-approval EJR algorithm we derived is useful for evaluating the sampling bounds we have achieved. Compare the winning set W produced by this algorithm with the winning set W' produced by some other PB-EJR algorithm on the same sample of the population S. We have that $W \subseteq W'$, and so if W provides ε^*-PB-EJR for the whole population, then W' certainly does. However, W' may be able to provide ε'-PB-EJR with $\varepsilon' < \varepsilon^*$. Clearly $\max_{p \notin W'} \varepsilon_p \leq \max_{p \notin W} \varepsilon_p$, but we could have a strict inequality if

$$\arg\max_{p \notin W} \varepsilon_p \subseteq W'.$$

Therefore, when we analyse the ε^* we obtain from the real election, we should keep in mind that using a different PB-EJR algorithm could provide better bounds, and so the ε^* that we obtain is a worst-case value.

We shall now answer the two questions we started with. The first question has a straightforward answer: the sampling bounds do indeed achieve the corresponding δ, ε representation for each value of ε we test. In fact, there was no sample in which $\varepsilon^* > \varepsilon$, so in the experiment, we achieve ε with probability 1, so $\delta = 0$. This then leads to our second question. The fact that no sample ever failed to provide its corresponding ε representation suggests that our bounds are quite loose.

In Figures 4.1 to 4.4, we see 4 graphs for $\varepsilon^* = 0, 5, 15, 25\%$, respectively. The graphs plot the probability that the winning set of the sample does not achieve ε-PB-EJR for the whole population against (log) sample size. We use a logarithmic axis because of the difference in order of magnitudes between the sample sizes. Four points in particular are annotated on each graph corresponding to sample sizes of 689, 1787, 8014, and 92635. These sample sizes are the minimum sample sizes (of the sizes tested) required to achieve $25, 15, 5$, and 0% ε^* in practice, respectively, with probability at most 5%.

However, these sample sizes correspond to a guaranteed $\varepsilon = 290, 180, 85$, and 25% given by our bounds, respectively, and so we can see that the bounds are extremely loose. Furthermore, we had that for $\varepsilon^* = 30\%$, even a sample size of $s = 232$, corresponding to a guaranteed $\varepsilon = 500\%$, provided 30%-PB-EJR with failure probability of 0% in the tests. We can see then that in practice, sampling can produce approximate representation with a relatively small sample size. Certainly in this example, we can achieve 15%-PB-EJR with less than 1.4% of the population sampled, but since the sample bounds we show do not depend on the total population size, it is reasonable to expect the required proportion of the population to be sampled to only decrease for larger populations.

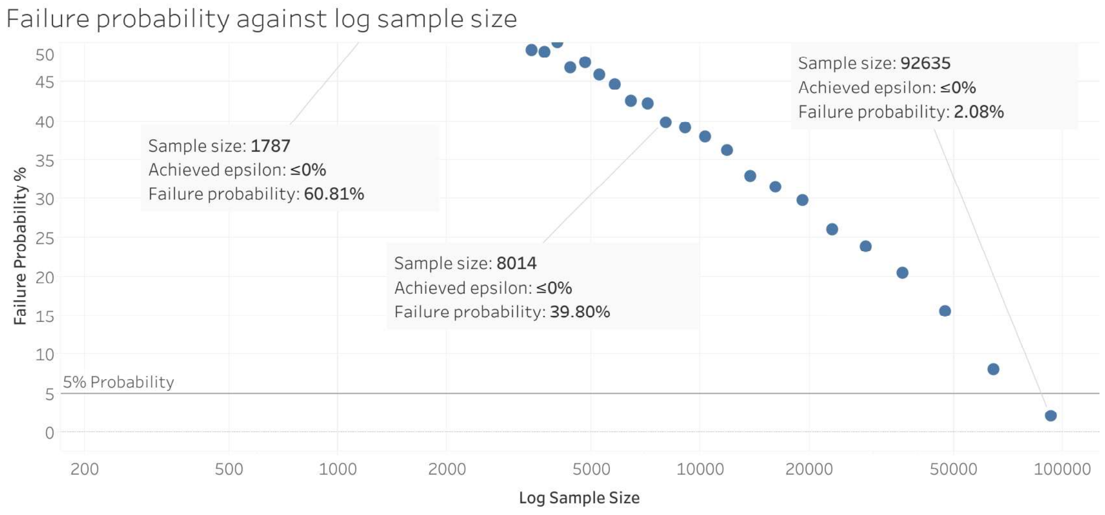


Figure 4.1: Simulated failure probability against log sample size with $\varepsilon = 0\%$. Only $s = 92635$ provided exact representation with probability at most 5%.

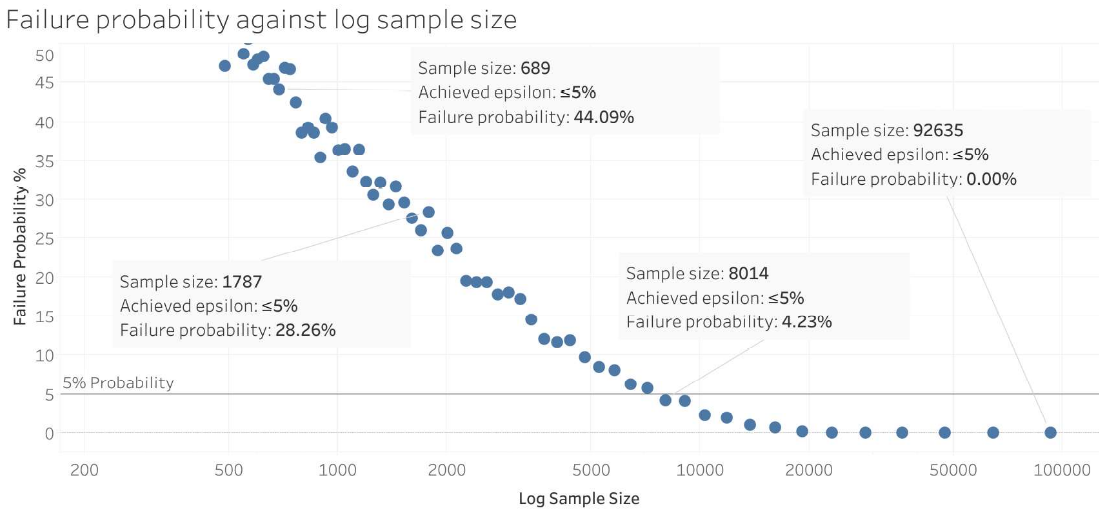


Figure 4.2: Simulated failure probability against log sample size with $\varepsilon = 5\%$. The smallest sample size to provide 5% representation with failure probability at most 5% is $s = 8014$.

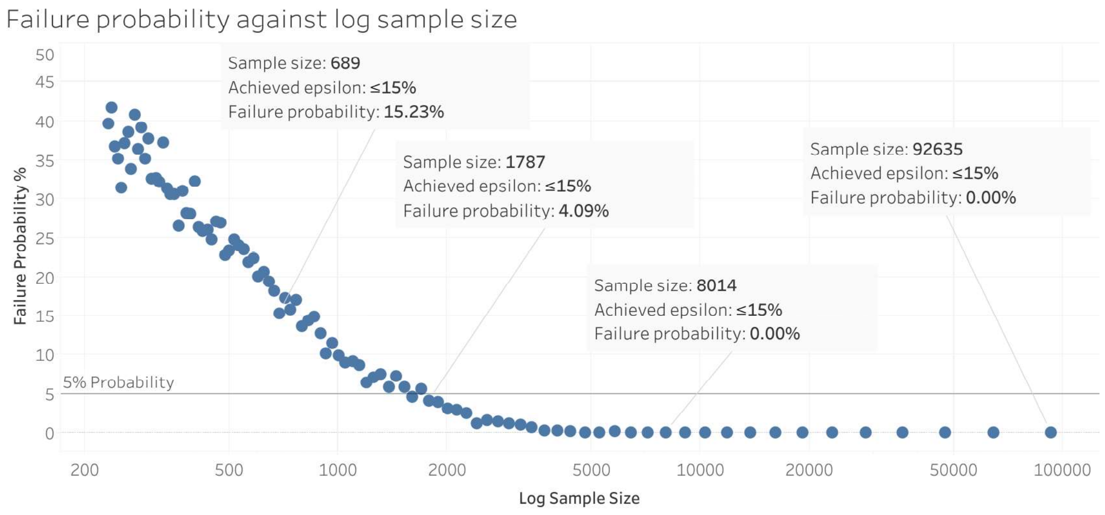


Figure 4.3: Simulated failure probability against log sample size with $\varepsilon = 15\%$. The smallest sample size to provide 15% representation with failure probability at most 5% is $s = 1787$.

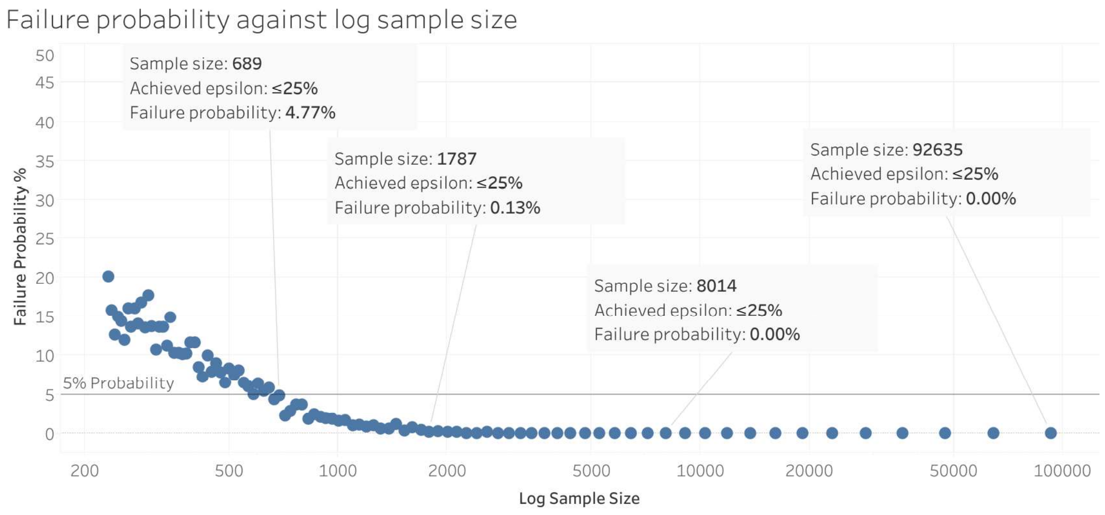


Figure 4.4: Simulated failure probability against log sample size with $\varepsilon = 25\%$. The smallest sample size to provide 25% representation with failure probability at most 5% is $s = 689$.

Chapter 5

Conclusion

5.1 Summary and evaluation

Initially in Chapter 2, we discussed the MWV and PB theory, including the underlying voting models, fairness properties for each type of election, and existing algorithms that can guarantee these fairness properties. In Chapter 3, we then moved on to discuss an extension of the PB model in which projects are assigned some position on a one-dimensional interval, such that any winning set of the election cannot include two projects that overlap on this interval. We showed in Section 3.1 that guaranteeing any fairness property for this model is at least as hard as providing it for the standard model and that some fairness properties for the standard model are unsatisfiable for the interval model. We then went on to show in Section 3.1.1 that there exists a pseudo-polynomial time algorithm, and then a fully polynomial time algorithm, for utilitarian welfare optimisation in the interval PB model. The algorithms were then implemented and tested in Python 3 against both random and partially real-world data, and in Section 3.1.2, it was demonstrated that the practical performance of the algorithms matched the proven runtimes, and showed the applicability of the algorithms to real-world elections. In the interval model, we then went on to show in Section 3.2 that if we allow each project to have a set of possible locations on the interval that it could be implemented at, then the conflict graph of the election has unbounded treewidth, even with a bounded number of locations per project and a bounded number of overlaps per project location. In fact, we showed that the problem of utilitarian welfare maximisation in this setting is $\mathcal{NP}$-hard.

In Chapter 4, we then investigated the scenario in which we obtain partial information about the electorate's preferences through random sampling. We first discussed the MWV scenario in Section 4.1. A subclass of fairness properties called monotonic justified representation was defined, and it was shown that MWV-JR, MWV-PJR, and MWV-EJR are all forms of monotonic justified representation. We then provided a general approximation scheme for forms of monotonic justified representation, so that for any $\varepsilon, \delta > 0$, we can give the sample size required to achieve an ε approximation of representation with probability at least $1 - \delta$. We then proved a tighter bound for MWV-JR and also provided a bound for utilitarian welfare optimisation. Similar techniques were then used to prove bounds on the required sample size to provide approximate PB-EJR and PB-PJR representation for the PB scenario in Section 4.2. Finally, using real-world election data taken from Pabulib (Stolicki et al., 2020), we simulated taking samples of the voters in this election, and in Section 4.2.1, we gave evidence for the empirical correctness of the bounds proven and showed the practical applicability of the methods derived.

5.2 Further open problems

There are several different possible directions for future work, which we shall now discuss.

5.2.1 Higher dimensional project conflicts

In Chapter 3, we consider only conflicts between projects that overlap in the one-dimensional interval. However, we could further extend this to the scenario where projects are instead represented by hyperrectangles of dimension n in $[0, 1]^n$, where two projects conflict if they overlap in this space. The two-dimensional case has applications for PB; we could consider the land in a city as the $[0, 1]^2$ space, and the projects to be rectangular plots of land in the city, with the objective of achieving some fairness property subject to no two projects using the same piece of land. This is at least as hard as the one-dimensional case since we can always embed any one-dimensional PBIC election into a higher-dimensional space. Furthermore, for the two-dimensional case and above, the conflict graph is no longer necessarily an interval graph, or even chordal: we can construct a simple 4-cycle as shown in Figure 5.1. This means that we cannot apply most of the algorithms discussed in Section 3.1.1. The algorithm for bounded treewidth conflict graphs presented by Pferschy and Schauer (2009) is not immediately ruled out, but intuitively it feels unlikely that higher-dimensional conflict graphs have bounded treewidth, even under strict conditions on the degree of each project in the conflict graph.

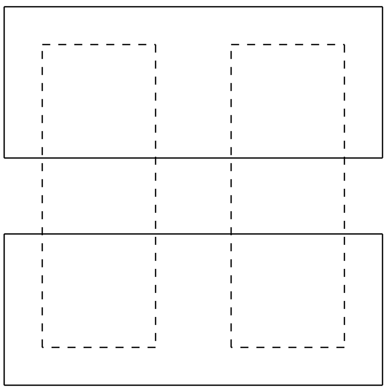


Figure 5.1: An illustration of the projects of a two-dimensional PBIC election where the conflict graph is not chordal. Each rectangle corresponds to a project's location in $[0, 1]^2$.

5.2.2 Allowing multiple projects at a particular point on the interval

For certain settings, we may want to allow projects to overlap on the interval. If we consider the scenario discussed in Section 3.1.3.3, we may be able to have multiple projects "active" at the same time. Perhaps we have h allotments that can grow different crops at each time, so in this case we would allow at most h different projects to be active at any point on the interval instead of just 1. We could extend this further and define each project as having some numerical "load" value which potentially differs for different projects. A valid winning set would then require that the sum of the loads of the winning projects at each time point be at most some capacity value, in addition to requiring that the total cost of the projects be within the budget. The literature of interval scheduling with multiple machines may be useful here.

5.2.3 Improved sampling bounds

The empirical evaluation of the sampling bounds derived in Section 4.2.1 suggests that tighter bounds may be possible, so that we can guarantee a smaller value of ε for any sample size s and failure probability δ. There are tighter bounds for the hypergeometric distribution, but they are harder to manipulate mathematically. For example, Chvátal (1979) provides a tighter bound in terms of the Kullback-Leibler divergence

$$D(a||b) = a\log\frac{a}{b} + (1-a)\log\frac{1-a}{1-b}.$$

Using this bound, we can prove that

$$\Pr\left(z_S(B) \leq \left(\frac{z_V(B)}{n} - t\right)s\right) \leq e^{-sD\left(\frac{z_V(B)}{n}-t||\frac{z_V(B)}{n}\right)}.$$

This can be shown to be at most e^{-2st^2}, and so is a strictly tighter bound than what we achieve; however, the Kullback-Leibler divergence is not easily manipulable in the contexts that we use it in, such as Theorem 4.8. However, for a specific election, if we have reasonable bounds on $\frac{z_V(B)}{n}$, then we could compute bounds on $D\left(\frac{z_V(B)}{n} - t || \frac{z_V(B)}{n}\right)$ that could be used to potentially provide improved sample size bounds.

5.3 Personal reflection

I would like to conclude by making some final remarks on my own personal experience. This dissertation has challenged me greatly; for example, the process of deriving the general sampling bounds for monotonic justified representation in the context of MWV elections was particularly demanding. The derivation of the bounds in Section 4.1 involves bounding the error between the number of voters who vote with ballot type B within the whole population and within the sample, for every ballot type B present in the whole population. However, in the development of this dissertation, I attempted to bound the error for many other groups of voters to obtain a similar proof of a much tighter bound, with little success. To overcome this, I realised that instead of aiming directly for a tight bound, if I instead prove a general looser bound, I may then be able to prove a tighter bound from that foundation. This led to the development of Theorem 4.8, and then the tighter bound shown in Lemma 4.10. The challenges I have tackled have led to discoveries that I personally have found interesting, and so I have found the process of developing this dissertation extremely rewarding. Completing this dissertation has been an enjoyable experience for me, and an excellent opportunity for me to further explore the concepts introduced to me in the "Computational Game Theory" course, and apply the probabilistic techniques taught to me in the "Probability and Computing" course to other areas of computer science. Research is very rarely truly completed, and Section 5.2 reflects the possible future directions of the work I have begun, but I am happy with the theoretical work and the results I have presented in this dissertation.

Appendix A

Source code extracts

This chapter contains an extract of the core code described in the dissertation. Some code, such as code to read and write Pabulib files, has not been included, but can be found at `http://github.com/DrewSpringham/ParticipatoryBudgeting`.

A.1 PB with conflicts on interval

A.1.1 `election_instance.py`

Listing A.1: Implementation of PBIC projects.

```python
class Project:
    """
    Defines a locational project
    """
    def __init__(self, start, size, cost):
        """
        Creates a locational project
        :param start: Start position on the interval
        :param size: The size (width) of the project
        :param cost: The cost of the project
        """
        self.start = start
        self.size = size
        self.cost = cost

    def __repr__(self):
        return '{start}/{end}/{cost}'.format(start=self.start, end=self.end, cost=self.cost)

    @property
    def end(self):
        return self.start + self.size
```

Listing A.2: Implementation of PBIC election.

```python
class Election:
    _approvals_by_project = None

    def __init__(self, voters, projects, approvals, election_id=None, budget=1):
        """
        Creates an election
        :param election_id: An identifier for the election. Optional, but useful for testing
        :param voters: A set of voters
        :param projects: A set of Projects
        :param approvals: A dict, indexed by voters and containing set of Projects as values
        :param budget: The budget of the election
        """

        self.approvals = approvals
        self.voters = voters
        self.projects = projects
        self.budget = budget
        #If we don't have an id, create a random 64 length string
        if election_id is None:
            self.election_id = ''.join(random.choice(string.ascii_letters) for i in range(64))
        else:
            self.election_id = election_id

    @property
    def approvals_by_project(self):
        """
        Generates a dictionary that takes projects to the total number of voters that approve it. Dictionary is cached, so only generated once
        :return: dictionary that takes projects to the total number of voters that approve it.
        """
        if self._approvals_by_project is not None:
            return self._approvals_by_project
        else:
            total_project_approval = defaultdict(int)
            # Iterate over voters and add one to the count of each project the voter approves
            for v in self.voters:
                for p in self.approvals[v]:
                    total_project_approval[p] += 1
            self._approvals_by_project = total_project_approval
            return total_project_approval
```

A.1.2 `helpers.py`

Listing A.3: Implementation of correctness testing functions.

```
    def verify_outcome(E, W):
    """
    Check that a winning set if valid for E
    :param E: An election
    :param W: A subset of projects of E
    :return: Bool indicating if W is valid for E
    """
    # if W is over budget, it is not valid
    if sum([p.cost for p in W]) > E.budget:
        return False
    # For each pair of projects, check they dont overlap. Could be more efficient in this
    #but don't need to be
    for p in W:
        for q in W:
            if p != q:
                # Projects overlap iff one's start exists within the others interval
                if q.start <= p.start <= q.end or p.start <= q.start <= p.end:
                    return False
    return True


def check_optimality(E, W):
    """
    Brute force check the utilitarian welfare optimality of W
    :param E: An election
    :param W: A winning set for E
    :return: if W is utilitarian welfare optimal
    """
    max_approvals = 0
    # For every possible subset of projects
    for s in powerset(E.projects):
        # If the subset would be a valid outcome
        if verify_outcome(E, s):
            # Check that it does not achieve more utility than our winning set
            total_approvals = sum([E.approvals_by_project[v] for v in s])
            if total_approvals > max_approvals:
                max_approvals = total_approvals
    achieved_value = sum([E.approvals_by_project[p] for p in W])
    return max_approvals == achieved_value
```

Listing A.4: Implementation of computing the preceding project in a list ordered by end position.

```
def compute_preceding_projects(P):
    """
    Compute a map from projects to the index of the project that lies wholly to the left of it
    :param P: The set of projects sorted by end point
    :return: A dictionary from projects to index of project that lies wholly to the left of it
    """
    m = len(P)
    prec_projects = {}
    for j in range(m + 1):
        found = False
        # Could make this more efficient with a binary search since P is sorted by end point,
        #but don't need to. Iterate backwards from project j until we find the project that lies
        #wholly to the left of it, so that projects end is before project i's start
        for k in range(j - 1, -2, -1):
            if P[k].end < P[j - 1].start:
                prec_projects[j] = k
                found = True
                break
        # If we don't find it, no project lies to the left of it, so we assign it index -1
        if not found:
            prec_projects[j] = -1
    return prec_projects
```

A.1.3 `standard_interval_knapsack.py`

Listing A.5: Implementation of standard knapsack dynamic programming table generation.

```python
def interval_knapsack_table(E: Election):
    """
    Compute the knapsack table for E using the standard knapsack table format: rows
    representing how many projects we are considering, columns representing weight limit
    :param E: The election
    :return m: the (partial) knapsack table, with sufficient entries filled to compute optimal project set
    """
    # number of projects
    n = len(E.projects)
    # Projects asc. sorted by end point
    P = list(sorted(E.projects, key=lambda p: p.end))
    W = E.budget
    # set up table
    m = [[None for _ in range(W + 1)] for _ in range(n + 1)]
    # If we consider no projects, the best utility we can achieve is 0, so weight is irrelevant
    for w in range(W + 1):
        m[0][w] = 0
    # We compute table entries on demand instead of the whole table, so we set up a stack
    # we wish to compute the (n,W) entry, as this corresponds to finding the best utility from all n projects and weight limit of W
    to_compute = [(n, W)]
    # We compute for every project p the project q whose endpoint is furthest to the right of p and not overlapping with p
    prec_project = compute_preceding_projects(P)
    # We shall stop once we have computed all table entries we need to compute
    while len(to_compute) > 0:
        i, w = to_compute.pop()
        proj_index = i - 1
        proj = P[proj_index]
        wi = proj.cost
        # prec_index is the index of P of the precding project
        prec_index = prec_project[i]
        # if, for this weight limit w, we could ever include the i'th project in the set, the optimal value is the same as not considering project i
        if wi > w:
            # we may not have already computed this value, so we'll put the current (i,w) back on the stack and add the computation of (i-1,w) on the stack
            if m[i - 1][w] is None:
                to_compute.append((i, w))
                to_compute.append((i - 1, w))
            else:
                m[i][w] = m[i - 1][w]
        else:
            util = E.approvals_by_project[proj]
            if m[i - 1][w] is not None and m[prec_index + 1][w - wi] is not None:
                m[i][w] = max(m[i - 1][w], util + m[prec_index + 1][w - wi])
            else:
                # We can't compute (i,w) yet because some computation we depend on has not been computed, so we add it back to the stack so we can come back to it later
                to_compute.append((i, w))
                if m[i − 1][w] is None:
                    to_compute.append((i − 1, w))
                if m[prec_index + 1][w − wi] is None:
                    to_compute.append((prec_index + 1, w − wi))
    return m
```

Listing A.6: Implementation of reversed knapsack set generation from the dynamic programming table.

```
def from_table(E, P, T):
    """
    Gets the project set from the partially filled dynamic programming table
    :param E: An election
    :param P: The project set of P ordered by end point
    :param T: The partially filled dynamic programming table for E
    :return:
    """
    proj_set = set()
    i = len(E.projects)
    w = E.budget
    #at each iteration, we look if we should include item i to the project set given our current
 weight limit w
    while i > 0:
        wi = P[i − 1].cost
        prec_projects = compute_preceding_projects(P)
        k = prec_projects[i] + 1
        # if the weight of project i is greater than our remaining weight limit, then we can't
add it, so we then look at project i−1. Also, if the utility we get from not adding it and using
all items up to i−1 is more than we get from adding it and only using items up to k, we won't
 add it
        if w < wi or T[i − 1][w] > T[k][w − wi] + E.approvals_by_project[P[i − 1]]:
            i = i − 1
        # if we can add it and we get more utility from adding it, we'll add it, and update
the weight limit
        else:
            proj_set.add(P[i − 1])
            i = k
            w = w − wi

    return proj_set


def interval_knapsack_projects(E):
    """
    Create the optimal project set for election E
    :param E: An election
    :return: A subsset of projects of E that optimise utiliatrian welfare for E
    """
    #Generate the dynamic programming table
    T = interval_knapsack_table(E)
    P = list(sorted(E.projects, key=lambda p: p.end))
    proj_set = from_table(E, P, T)
    return proj_set
```

A.1.4 `reverse_interval_knapsack.py`

Listing A.7: Implementation of reversed knapsack dynamic programming table generation.

```
def interval_knapsack_table_reversed(E: Election):
    """
    Creates the reverse knapsack dynamic programming table
    :param E: An election
    :return T,upper: the dynamic programming table T, and upper which is the max utility
    achievable with budget b
    """
    m = len(E.projects)
    max_total_approvals = sum([E.approvals_by_project[p] for p in E.projects])
    # Projects sorted by end point
    P = list(sorted(E.projects, key=lambda p: p.end))
    b = E.budget
    # set up table
    T = [[None for _ in range(1 + max_total_approvals)] for _ in range(m + 1)]
    # Compute dict indexed by projects p of last project that lies wholly before p on interval
    prec_projects = compute_preceding_projects(P)
    # need infinite weight limit to achieve any utility with no items
    for u in range(1 + max_total_approvals):
        T[0][u] = math.inf
    # can achieve o utility with any number of projects
    for i in range(m + 1):
        T[i][0] = 0
    lower_bound = 0
    upper_bound = max_total_approvals
    # binary search style loop
    while lower_bound < upper_bound:
        r = (lower_bound + upper_bound + 1) // 2
        to_compute = [(m, r)]
        # Computing table elements on demand, so we can get the value of T[m][r]
        while len(to_compute) > 0:
            i, u = to_compute.pop()
            # Value of item i
            vi = E.approvals_by_project[P[i - 1]]
            # weight of item i
            wi = P[i - 1].cost
            # the index of last project that lies wholly to left of project i (-1 if doesn't exist)
            t = prec_projects[i]
            # if we include project i, to make utility of u, the rest of the projects must
            # contribute u-vi, 0 if u-vi<0
            needed_util = max(u - vi, 0)
            # if we've already compute the table elements we need, we can compute T[i][u]
            if T[t + 1][needed_util] is not None and T[i - 1][u] is not None:
                T[i][u] = min(wi + T[t + 1][needed_util], T[i - 1][u])
            else:
                # we need to compute prerequisite table entries
                to_compute.append((i, u))
                if T[t + 1][needed_util] is None:
                    to_compute.append((t + 1, needed_util))
                if T[i - 1][u] is None:
                    to_compute.append((i - 1, u))
        # given T[m][r], we compare to the budget to see if we have spare capacity to achieve
        #more utility, or if we are over capacity
        if T[m][r] > b:
            upper_bound = r - 1
        else:
            lower_bound = r
    return T, upper_bound
```

Listing A.8: Implementation of reversed knapsack set generation from the dynamic programming table.

```
def from_table_reversed(E, P, T, u):
    """
    Generates the set of projects that achieves u within E's budget, given a sufficiently filled dyn. prog. table
    :param E: An election
    :param P: The projects of E ordered by end point
    :param T: The reverse knapsack dynamic programming table for E
    :param u: the optimal utility we have found that we can achieve for E
    :return proj_set: A subset of E.projects that achieves utility u within E's budget
    """
    i = len(E.projects)
    proj_set = set()
    while i > 0:
        wi = P[i - 1].cost
        vi = E.approvals_by_project[P[i - 1]]
        prec_projects = compute_preceding_projects(P)
        t = prec_projects[i] + 1
        needed_util = max(u - vi, 0)
        #If item i is not in an optimal set, then adding item i mean that we would need a larger weight limit to reach the same utility, so therefore if we add item i and the weight limit we need from the remaining items is not more than the weight limit we require from the first i-1 items, then item i is in the optimal set
        if T[i - 1][u] < wi + T[t][needed_util]:
            i = i - 1
        else:
            proj_set.add(P[i - 1])
            i = t
            u = needed_util
    return proj_set


def interval_knapsack_projects_reversed(E):
    """
    Generates the optimal winning set for election E
    :param E: An election
    :return proj_set: The optimal project set for E
    """
    # Generate the dynamic programming table
    T, best_util = interval_knapsack_table_reversed(E)
    P = list(sorted(E.projects, key=lambda p: p.end))
    # Get the project set from the table
    proj_set = from_table_reversed(E, P, T, best_util)
    return proj_set
```

A.1.5 `random_instances.py`

Listing A.9: Implementation of the random instance generation.

```
def random_project_size_then_start(min_cost, max_cost):
    """
        Create a random project by selecting a random size and then start point
        :param min_cost: A minimum cost of the project
        :param max_cost: A maximum cost of the project
        :return: A random project with the parameters given
        """
    size = random.random()
    start = random.uniform(0, 1 - size)

    if min_cost == max_cost:
        cost = min_cost
    else:
        cost = random.randint(min_cost, max_cost)
    return Project(start, size, cost)


def random_project(min_cost, max_cost):
    return random_project_size_then_start(min_cost, max_cost)


def random_approvals(voters, projects):
    """
    Generate a random approval dict given the voters and projects
    :param voters: Set of voters
    :param projects: Set of projects
    :return: Random approval function
    """
    approvals = {}
    for v in voters:
        num = random.randint(1, len(projects))
        approvals[v] = random.sample(projects, num)
    return approvals


def random_instance(N, p):
    """
    Generates a random election with at N voters and p projects
    :param N: Number of voters for the election
    :param p: Number of projects for the election
    :return: An election with those parameters
    """
    voters = [i for i in range(1, N + 1)]
    projects = []
    for i in range(p):
        #generate p random projects
        projects.append(random_project(5000, 100000))
    approvals = random_approvals(voters, projects)
    #find average cost of project
    average_cost = sum([P.cost for P in projects]) / p
    #select the budget randomly to be between the average cost of a single project and the sum of all the costs
    budget = ceil(average_cost * random.uniform(1, p))
    E = Election(voters, projects, approvals, None, budget)
    return E
```

Listing A.10: Implementaton of the random instance correctness checking.

```
def random_instances(k, min_N, max_N, min_p, max_p):
    """
    Generate k random election
    :param k: The number of elections to generate
    :param min_N: The minimum number of voters in each election
    :param max_N: The max. number of voters in each election
    :param min_p: The min. number of projects in each election
    :param max_p: the max. number of porjects in each election
    """
    for i in range(k):
        N = random.randint(min_N, max_N)
        p = random.randint(min_p, max_p)
        yield random_instance(N, p)

def random_check(N, p, rule):
    """
    Verify that the elction rule is correct on a random instance
    :param N: The number of voters in the random instance
    :param p: The number of project in the random instance
    :param rule: The rule to test
    """
    E = random_instance(N, p)
    ik_result = rule(E)

    if not check_optimality(E, ik_result):
        pickle.dump(E, open("bad_instance.p", "wb"))
        raise ValueError("Random checks failed, check pickle file.")


def random_checks(k, rule, min_N=50, max_N=1000, min_p=5, max_p=15):
    """
    Test an election rule k times for correctness against random instances
    :param k: The number of tests to run
    :param rule: The election rule
    :param min_N: The min number of voters in the test
    :param max_N: the max number of voters in the test
    :param min_p: The min number of projects in the test
    :param max_p: The max number of porjects in the test
    """
    for _ in tqdm(range(k)):
        N = random.randint(min_N, max_N)
        p = random.randint(min_p, max_p)
        random_check(N, p, rule)

def main():
    random_checks(1000, interval_knapsack_projects)
```

A.1.6 `real_instance.py`

Listing A.11: Implementation of the real election data correctness testing.

```python
def real_instances(up_to=None):
    """
    Generator for real instances
    :param up_to: An optional limiter to limit the number of instances to return
    """
    directory = "../tests/pb_files_loc/"
    files = os.listdir(directory)
    if up_to is not None:
        files = files[:up_to]
    for filename in tqdm(files):
        f = os.path.join(directory, filename)
        print(f)
        # checking if it is a file
        if os.path.isfile(f):
            E = convert_to_election(f)
            yield E


def real_checks(rule):
    for E in real_instances():
        ik_result = rule(E)
        if not check_optimality(E, ik_result):
            raise ValueError(f"Real checks failed, check file {E.election_id}")


def main():
    real_checks(interval_knapsack_projects_reversed)
```

A.1.7 `benchmarking.py`

Listing A.12: Implementation of benchmarking of election rules.

```python
def generate(data_source, rules):
    """
    Benchmarks the time election rules take for different elections, and saves the data
    :param data_source: A string of which data source to use. Either "random" or "real"
    :param rules: a list of tuples of rules and rule ids
    """
    elections = {'election_id': [], 'voters': [], 'projects': [], 'budget': []}
    times = {'election_id': [], 'rule_id': [], 'time': []}
    if data_source == 'random':
        source = random_instances(10, 1000, 300000, 3, 70)
    elif data_source == 'real':
        source = real_instances()
    else:
        raise ValueError("Unknown data source!")
    try:
        print("Starting testing")
        for n, E in enumerate(source):

            election_id = E.election_id
            print(f"Testing election {n}")
            elections['election_id'].append(election_id)
            elections['voters'].append(len(E.voters))
            elections['projects'].append(len(E.projects))
            elections['budget'].append(E.budget)
            for (rule, rule_id) in rules:
                # We time how long it takes for a rules to compute a winning set
                start = timeit.default_timer()
                ik_result = rule(E)
                end = timeit.default_timer()
                time = end - start
                print(f"Finished on rule {rule_id}")
                times['election_id'].append(election_id)
                times['rule_id'].append(rule_id)
                times['time'].append(time)
    finally:
        #We save the data to a csv file for external analysis
        election_frame = pd.DataFrame(elections)
        res_frame = pd.DataFrame(times)
        election_frame.to_csv(f'bench_elections_{data_source}.csv', mode='a', index=
False, header=False)
        res_frame.to_csv(f'bench_results_{data_source}.csv', mode='a', index=False,
header=False)
```

A.2 Sampling

A.2.1 `basic_single_ejr.py`

Listing A.13: Implementation of the basic single approval election EJR algorithm.

```python
def basicejr(E: Election):
    """
    Generates the minimal EJR project set for a single approval election
    :param E: A single approval election
    :return W: A minimal EJR winning set for E
    """
    if not is_single(E):
        raise ValueError("Election needs to be single")
    # For each project, check if the number of people that approve is sufficiently large such that exclduing the
    # project would make the project set no longer EJR
    W=[p for p in E.projects if E.approvals_by_project[p] >= p.cost / E.budget * len(E.voters)]
    return set(W)
```

A.2.2 `sampling.py`

Listing A.14: Implementation of helper functions for sampling simulation.

```python
def is_single(E):
    """
    Check if an election is a single approval election
    :param E: An election
    :return: If the election is a single approval election
    """
    single = True
    # Check that every voter only votes for a single candidate
    for v in E.voters:
        if len(E.approvals[v]) != 1:
            single = False
    return single


def check_EJR_single_approval(E, W):
    """
    For an single approval election, compute the true EJR error of the project set W
    :param E: An election
    :param W: A winning set of projects
    :return true_eps: true_eps which is the max value of eps over each project p such that p
is not in W and the number of voters of p is (1+eps)(cost(p)*n/b)
    """
    true_eps = 0
    for p in E.projects:
        G_size = E.approvals_by_project[p]
        req_size = p.cost * len(E.voters) / E.budget
        if G_size >= req_size and p not in W:
            true_eps = max(true_eps, G_size / req_size - 1)
    return true_eps


def create_subelection(E, s):
    """
    Create a new election derived from E with a sample of the voters of size s
    :param E: An election
    :param s: A sample size of the voters
    :return: A new election derived from E
    """
    S = set(random.sample(list(E.voters), s))
    subapprovals = E.approvals.copy()
    # Remove all voters not in the sample S
    for v in E.voters:
        if v not in S:
            del subapprovals[v]
    subelection = Election(S, E.projects, subapprovals, None, E.budget)
    return subelection
```

Listing A.15: Implementation of sampling simulation.

```
def test_election(E, source_name, runs, max_eps, eps_step_per_unit, delta):
    elections = {'election_id': [], 'voters': [], 'projects': [], 'budget': [], 'min_cost': []}
    results = {'election_id': [], 'rule_id': [], 'run_id': [], 'delta': [], 'eps': [], 'real_eps': []}
    #wrap in try so we can save current results at any timr
    try:
        #only works if election is single
        if is_single(E):
            min_cost = min([p.cost for p in E.projects]) + 0.0001
            elections['election_id'].append(E.election_id)
            elections['voters'].append(len(E.voters))
            elections['projects'].append(len(E.projects))
            elections['budget'].append(E.budget)
            elections['min_cost'].append(min_cost)

            m = len(E.projects)
            for eps_i in range(max_eps * eps_step_per_unit):
                #eps_i runs from 0 to max_ eps* number of steps per unit of epsilon, so to get a value of eps we transform
                eps = (eps_i + 1) / eps_step_per_unit
                #get sample size from bound
                s = ceil(E.budget ** 2 * log(m / delta) / (2 * eps ** 2 * min_cost ** 2))
                # only try to sample if sample size is less than whole population
                if s <= len(E.voters):
                    print(f"EPSILON: {eps}\n")
                    print(f"SAMPLE SIZE {s}")
                    #run this configuration runs number of times
                    for _ in trange(runs):
                        subelection = create_subelection(E, s)
                        W = basicejr(subelection)
                        real_eps = check_EJR_single_approval(E, W)
                        results['election_id'].append(E.election_id)
                        results['rule_id'].append("BasicJR")
                        #choose random id for run to make it easier to select a specific run
                        #in analysis
                        r_id=''.join(random.choice(string.ascii_letters) for _ in range(64))
                        results['run_id'].append(r_id)
                        results['delta'].append(delta)
                        results['eps'].append(eps)
                        results['real_eps'].append(real_eps)
    finally:
        #save the results once we finish or error (or keyboard interupt)
        election_frame = pd.DataFrame(elections)
        res_frame = pd.DataFrame(results)
        election_frame.to_csv(f'elections_{source_name}.csv', mode='a', index=False, header=False)
        res_frame.to_csv(f'results_{source_name}.csv', mode='a', index=False, header=False)


def main(source_name, runs, max_eps, eps_step_per_unit, delta):
    f = "../tests/pb_files/poland_wroclaw_2015_from-500.pb"
    if source_name == "real":
        source = real_instances()
    elif source_name == "only":
        source = [convert_to_election(f)]
    elif source_name == "random":
        source = random_instances(50, 1000, 300000, 3, 70)
    else:
        raise ValueError("Source name not defined")
    for E in source:
        test_election(E, source_name, runs, max_eps, eps_step_per_unit, delta)
```

Bibliography

Aziz, H., Brill, M., Conitzer, V., Elkind, E., Freeman, R. and Walsh, T. (2017), 'Justified representation in approval-based committee voting', *Social Choice and Welfare* **48**(2), 461–485.

Aziz, H. and Lee, B. E. (2021), Proportionally representative participatory budgeting with ordinal preferences, *in* 'Proceedings of the 35th AAAI Conference on Artificial Intelligence', pp. 5110–5118.

Aziz, H., Lee, B. E. and Talmon, N. (2018), Proportionally Representative Participatory Budgeting: Axioms and Algorithms, *in* 'Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems', pp. 23–31.

Aziz, H. and Shah, N. (2021), Participatory Budgeting: Models and Approaches, *in* 'Pathways Between Social Science and Computational Social Science: Theories, Methods, and Interpretations', Springer International Publishing, Cham, pp. 215–236.

Bednarska-Olejniczak, D. and Olejniczak, J. (2016), Participatory budget of Wrocław as an element of Smart City 3.0 concept, *in* 'Proceedings of the 19th International Colloquium on Regional Sciences', pp. 760–766.

Bodlaender, H. L. and Koster, A. M. (2011), 'Treewidth computations II. Lower bounds', *Information and Computation* **209**(7), 1103–1119.

Chan, H., Filos-Ratsikas, A., Li, B., Li, M. and Wang, C. (2021), Mechanism Design for Facility Location Problems: A Survey, *in* 'Proceedings of the 30th International Joint Conference on Artificial Intelligence', pp. 4356–4365.

Chvátal, V. (1979), 'The tail of the hypergeometric distribution', *Discrete Mathematics* **25**(3), 285–287.

Cochran, W. G. (1977), *Sampling Techniques*, 3 edn, Wiley, London.

Cohen-Addad, V., Klein, P. N., Marx, D., Wheeler, A. and Wolfram, C. (2021), On the Computational Tractability of a Geographic Clustering Problem Arising in Redistricting, *in* 'Proceedings of the 2nd Symposium on Foundations of Responsible Computing', p. 3:1–3:18.

Cohen-Addad, V., Klein, P. N. and Young, N. E. (2018), Balanced Centroidal Power Diagrams for Redistricting, *in* 'Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems', pp. 389–396.

Cygan, M., Fomin, F. V., Kowalik, L., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M. and Saurabh, S. (2015), Treewidth, *in* 'Parameterized Algorithms', Springer International Publishing, Cham, chapter 7, pp. 151–244.

Dey, P., Misra, N. and Narahari, Y. (2018), 'Complexity of manipulation with partial information in voting', *Theoretical Computer Science* **726**, 78–99.

Elkind, E., Li, M. and Zhou, H. (2022), Facility Location With Approval Preferences: Strategyproofness and Fairness, *in* 'Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems', pp. 391–399.

Erdélyi, G. and Reger, C. (2016), Possible Bribery in k-Approval and k-Veto Under Partial Information, *in* 'Proceedings of the 17th International Conference on Artificial Intelligence: Methodology, Systems, and Applications', pp. 299–309.

Faliszewski, P., Skowron, P., Slinko, A. and Talmon, N. (2017), Multiwinner voting: A new challenge for social choice theory, *in* 'Trends in Computational Social Choice', AI Access, chapter 2, pp. 27–47.

Fernández, L. S., Elkind, E., Lackner, M., García, N. F., Arias-Fisteus, J., Basanta-Val, P. and Skowron, P. (2017), Proportional Justified Representation, *in* 'Proceedings of the 31st AAAI Conference on Artificial Intelligence', pp. 670–676.

Fifield, B., Higgins, M., Imai, K. and Tarr, A. (2020), 'Automated Redistricting Simulation Using Markov Chain Monte Carlo', *Journal of Computational and Graphical Statistics* **29**(4), 715–728.

Fluschnik, T., Skowron, P., Triphaus, M. and Wilker, K. (2019), Fair Knapsack, *in* 'Proceedings of the 33rd AAAI Conference on Artificial Intelligence', pp. 1941–1948.

Ganuza, E. and Baiocchi, G. (2012), 'The Power of Ambiguity: How Participatory Budgeting Travels the Globe', *Journal of Deliberative Democracy* **8**(2), 12–30.

Garey, M. R. and Johnson, D. S. (1991), *Computers and intractability : a guide to the theory of NP-completeness*, W.H. Freeman, New York.

Gonçalves, S. (2014), 'The Effects of Participatory Budgeting on Municipal Expenditures and Infant Mortality in Brazil', *World Development* **53**, 94–110.

Hazon, N., Aumann, Y., Kraus, S. and Wooldridge, M. (2012), 'On the evaluation of election outcomes under uncertainty', *Artificial Intelligence* **189**, 1–18.

Henry, G. T. (1990), *Practical sampling*, Sage Publications, Newbury Park.

Hoeffding, W. (1963), ‘Probability Inequalities for Sums of Bounded Random Variables’, *Journal of the American Statistical Association* **58**(301), 13–30.

Jain, P., Sornat, K. and Talmon, N. (2021), Participatory Budgeting with Project Interactions., *in* ‘Proceedings of the 29th International Joint Conference on Artificial Intelligence’, pp. 386–392.

Jain, P., Sornat, K., Talmon, N. and Zehavi, M. (2021), Participatory Budgeting with Project Groups, *in* ‘Proceedings of the 30th International Joint Conference on Artificial Intelligence’, pp. 276–282.

Kalech, M., Kraus, S., Kaminka, G. A. and Goldman, C. V. (2011), ‘Practical voting rules with partial information’, *Autonomous Agents and Multi-Agent Systems* **22**(1), 151–182.

Kellerer, H., Pferschy, U. and Pisinger, D. (2004), *Knapsack Problems*, 1 edn, Springer Berlin, Heidelberg.

Land, K. C. and Zheng, H. (2010), Sample Size, Optimum Allocation, and Power Analysis, *in* ‘Handbook of survey research’, 2 edn, Emerald, Bingley, chapter 7, pp. 199–219.

Los, M., Christoff, Z. and Grossi, D. (2022), Proportional Budget Allocations: Towards a Systematization, *in* ‘Proceedings of the 31st International Joint Conference on Artificial Intelligence’, pp. 398–404.

Magleby, D. B. and Mosesson, D. B. (2018), ‘A New Approach for Developing Neutral Redistricting Plans’, *Political Analysis* **26**(2), 147–167.

Martello, S. and Toth, P. (1990), *Knapsack problems : algorithms and computer implementations*, Wiley, Chichester.

Mitzenmacher, M. and Upfal, E. (2005), *Probability and Computing*, Cambridge University Press, Cambridge.

Peters, D., Pierczyński, G. and Skowron, P. (2020), 'Proportional Participatory Budgeting with Cardinal Utilities'.
URL: *arxiv.org/abs/2008.13276v1*

Peters, D. and Skowron, P. (2019), 'Proportionality and the Limits of Welfarism'.
URL: *arxiv.org/abs/1911.11747*

Pferschy, U. and Schauer, J. (2009), 'The knapsack problem with conflict graphs', *Journal of Graph Algorithms and Applications* **13**(2), 233–249.

Shah, A. (2007), *Participatory Budgeting*, World Bank, Washington, D.C.

Spieksma, F. C. R. (1999), 'On the approximability of an interval scheduling problem', *Journal of Scheduling* **2**(5), 215–227.

Stolicki, D., Szufa, S. and Talmon, N. (2020), 'Pabulib: A Participatory Budgeting Library'.
URL: *arxiv.org/abs/2012.06539*

Touchton, M. and Wampler, B. (2020), 'Public engagement for public health: participatory budgeting, targeted social programmes, and infant mortality in Brazil', *Development in Practice* **30**, 681–686.

Wu, C. and Thompson, M. E. (2020), *Sampling theory and practice*, Springer, Cham.