CM3203 Final Year Project · Final Report

The Community Knapsack: Exploring Optimisation Algorithms For Combinatorial Participatory Budgeting

Supervisor:	**Author:**	**Moderator:**
[redacted]	Fin Bignold-Jordan	[redacted]
-	[redacted]	-

Abstract

Participatory budgeting is a democratic approach to decision-making in which citizens or residents of municipalities and communities directly vote on the allocation of public resources to projects and proposals. The combinatorial model of participatory budgeting, in which projects must either be fully or not at all funded, is widely solved using an arbitrarily bad greedy approach which fails to utilise the resources maximally. This project aims to improve participatory democracy by investigating, designing and evaluating optimisation algorithms to identify more accurate, feasible alternatives to this greedy approach, and implement these in a budget allocation library for e.g., budget decision-makers, researchers and students to solve their own problems.

Acknowledgements

I would like to express my gratitude to my supervisor, [redacted], for his invaluable expertise and mentorship throughout the course of this project. I am especially grateful for his patience during our weekly progress meetings and his guidance during taxing periods of time. Thank you for making this project possible.

Contents

Table of Acronyms

Report Acronyms

Acronym	Meaning
PB	Participatory Budgeting
CPB	Combinatorial Participatory Budgeting
BKP	Binary Knapsack Problem
MBKP	Multi-Dimensional Binary Knapsack Problem
FKP	Fractional Knapsack Problem

Algorithm Acronyms

Acronym	Meaning
BRF	Brute Force
MEM	Memoization
DYP	Dynamic Programming
BRB	Branch & Bound
ABR	Approximate Branch & Bound
GRE	Greedy
RAG	Ratio Greedy
FPA	Fully Polynomial-Time Approximation Scheme
SIA	Simulated Annealing
GEN	Genetic Algorithm
CBC	Integer Linear Programming Solver (Branch & Cut)

Table of Notation

The notation listed here is globally defined and used throughout the report, although additional notation may be defined locally in certain sections and proofs.

Notation	Meaning
B	The d resources, capacities or budgets available in the problem.
P	The set of n projects, items or candidates available in the problem.
V	The set of m voters or agents who submit preferences over the candidates.
u	A function providing the utility a voter derives from a candidate.
c	A function providing the weights or costs of each candidate.
v	A function providing the overall value of each candidate.
x	The decision variables for the inclusion of candidates in the solution.

1 Introduction

1.1 Introduction & Motivation

Participatory budgeting (PB) is a democratic approach to decision-making in which citizens or residents of municipalities and communities directly vote on the allocation of public resources to projects and proposals. PB was first introduced in Porto Alegre, Brazil in 1989 in response to authoritarian governing and social inequality [14], and has since become a global success with implementations recorded in many countries across every continent [15]. It is a field of great innovation, especially from the standpoint of local government and democracy, creating opportunities to empower, engage and educate residents to foster a more inclusive and vibrant civil society [37]. There are many different forms and implementations of PB, but each process tends to follow the same general steps, including dividing the region into sub-regions or neighbourhoods, holding meetings for residents to deliberate over and propose projects, the vetting, formalisation and refinement of these projects; and finally, the voting and budget allocation stage in which the residents vote over which projects should be funded using the resources [2].

This project is focused on the final stages of the process in the *combinatorial* model of participatory budgeting (CPB), in which each project must either be fully or not at all funded, and the way in which budgets or resources should be allocated given a set of voter utilities or preferences over the projects. The most natural and common approach in this model is to attempt to maximise the overall voter utility in the allocation, also known as *maximising the utilitarian welfare*, using the greedy algorithm, which involves sorting the projects by number of votes and fully funding them until any of the budgets or all of the projects have been exhausted [2, 34, 36]. Unfortunately, this approach may yield arbitrarily bad solutions that do not fully maximise voter utilities, often satisfying fewer participants overall.

When maximising the utilitarian welfare, CPB is reducible to the one and multi-dimensional binary knapsack problems (BKP, MBKP), and thus it is $\mathcal{NP}$-hard and, unless $\mathcal{P} = \mathcal{NP}$, cannot be solved exactly in polynomial time. This makes achieving optimality in budget allocation in this setting a difficult problem. The motivation behind this project is to improve participatory democracy by improving budget utilisation and thus increasing voter satisfaction. Our approach is to investigate and design exact and approximate optimisation algorithms to identify more accurate, feasible alternatives to the greedy approach, and implement these in a Python-based budget allocation library for evaluation purposes and e.g., budget decision-makers, researchers and students to solve their own instances.

1.2 Report & Section Overview

The background and preliminaries for this report and project are provided in Sect. 2, where we introduce the one and multi-dimensional binary knapsack problems (`BKP`, `MBKP`) and the fractional knapsack problem (`FKP`), and demonstrate the greedy algorithm for these problems. We perform a literature review into participatory budgeting, briefly considering the various models before delving into approaches to preference elicitation, vote aggregation and budget allocation. We identify how `CPB` is reducible to `BKP` and `MBKP` when maximising the utilitarian welfare, and describe the limitations of the greedy algorithm in budget allocation and participatory democracy.

Sect. 3 is dedicated to the investigation and design of our alternative exact and approximation algorithms, namely dynamic programming, branch-and-bound and greedy approaches, a simulated annealing and genetic algorithm approach, and finally a mixed integer linear programming solver [20]. This section is primarily focused on describing how the algorithms converge on optima, analysing their performance and their differences when solving one (`BKP`) and multi-dimensional (`MBKP`) problems.

The requirements, design, implementation and testing of our Python-based budget allocation library is detailed in Sect. 4. The library wraps around the algorithms for evaluation purposes and to provide a simple interface for e.g., budget decision-makers, researchers or students to solve their own `CPB` problems. We visualise the library at a high-level, justify the design decisions, describe the implementation and discuss the unit testing to verify the behaviour and functionality of the code.

Sect. 5 contains an evaluation of the algorithms and the library: the algorithms are tested and evaluated by visualising and comparing the run-times and total value of the allocations they find over randomly generated and real-world data [39]. We consider how our algorithms perform against the greedy approach, and make judgements on their suitability to certain problem characteristics. The library is evaluated through the completion of the requirements, the suitability of the design and from the perspective of usability for our prospective users.

The report and project is concluded in Sect. 6 where we summarise our work and results by drawing our final conclusions on the algorithms and our library, discuss differences between the initial plan at the beginning of this semester and the end product described here, and finally consider the limitations of the project and list specific and general propositions for future work in this field of study.

2 Background & Preliminaries

2.1 Knapsack Problems

2.1.1 Binary Knapsack Problem (BKP)

The binary knapsack problem (BKP) is a classic problem in the field of combinatorial optimisation [32]. An instance of this problem has a set of items, each characterised by a numeric weight and value, and a knapsack with some weight capacity. The objective is to find the combination of items to place in the knapsack that has the highest overall value and whose summed weights do not exceed the capacity. Naturally, there is then a finite and discrete number of feasible allocations to explore, and thus BKP is a combinatorial optimisation problem.

More formally, we can consider an instance of BKP as a four-tuple (B, P, c, v), where B is the capacity of the knapsack and thus $B \in \mathbb{N}_0$, P is the set of candidate items such that $P = \{1, \ldots, n\}$, where n is the total number of items and $n \in \mathbb{N}_0$, and c and v are the weight and value functions defined $c : P \to \mathbb{N}$ and $v : P \to \mathbb{N}_0$, which provide the weight c_p and value v_p of each item $p \in P$ respectively. Finally, we introduce a vector of decision variables $\vec{x} = (x_1, \ldots, x_n)$ such that x_p considers the inclusion of item p in the knapsack, where $x_p \in \{0, 1\}$ and $x_p = 1$ if and only if item p is in the knapsack. An integer linear program for the problem is provided in (1).

Binary Knapsack Problem (BKP)

$$\begin{aligned} \text{maximise} \quad & \sum_{p \in P} x_p \cdot v_p \\ \text{subject to} \quad & \sum_{p \in P} x_p \cdot c_p \leq B \\ & x_p \in \{0, 1\} \quad \forall p \in P \end{aligned} \tag{1}$$

A simple demonstration of BKP and an example of the greedy approach finding a sub-optimal solution is provided in Ex. 1. The greedy approach here is essentially the same as for MBKP and thus CPB, and in all these problems it may produce arbitrarily bad solutions by prioritising individually highly-valued items, thus blocking combinations of individually lower-valued items that may sum to a higher overall value. Despite this limitation, the greedy algorithm remains attractive where it is intuitive, simple and fast, and solving the problem by e.g., brute force is intractable where there are precisely 2^n possible solutions for each problem.

Example 1. Consider an instance with capacity $B = 100$ and $n = 3$ items whose weights and values are defined $c = (100, 50, 40)$ and $v = (20, 15, 10)$. The greedy approach would pick the highest-valued item, i.e., project 1 with a value of 20 and a weight of 100, thus exhausting the entire capacity and returning $\vec{x} = (1, 0, 0)$. However, items 2 and 3 have an overall value of 25 and an overall weight of 90. Clearly, $25 > 20$ and $90 \leq 100$ and thus $\vec{x} = (0, 1, 1)$ is the optimal solution. The greedy approach blocks the optimal solution by prioritising and including items with higher individual values.

`BKP` is in the set of $\mathcal{NP}$-hard problems [21], implying that it cannot be solved exactly in fully polynomial time. A proof of this known result is included in the appendix under Thm. 1. However, `BKP` is specifically classed as a *weakly* $\mathcal{NP}$-hard problem, meaning that it *can* be solved in *pseudo*-polynomial time through a dynamic programming approach. A simple description of this complexity is that the run-time is bounded polynomially by the magnitude or value of an input, but when represented in binary is exponential in the number of bits. An algorithm of this order will typically be able to solve small and medium instances, but will become too slow for very large inputs.

2.1.2 Multi-Dimensional Binary Knapsack Problem (`MBKP`)

The multi-dimensional binary knapsack problem (`MBKP`) is a generalisation of `BKP` which accommodates an arbitrary number of capacities as opposed to a single capacity. We refer to each capacity as being in a *dimension* of the knapsack, and say that every item has one weight in each dimension. A solution is invalid if *any* of the capacities are exceeded by the included items. More formally, we extend our four-tuple (B, P, c, v) such that $B = (B_1, \ldots, B_d)$ and $c : P \to \mathbb{N}^d$, where d is the number of dimensions and $d \in \mathbb{N}$. We say that $B_j \in \mathbb{N}_0$ and $c_{p,j} \in \mathbb{N}$ for all $p \in P$ and $j = 1, \ldots, d$, where B_j is the capacity in the j^{th} dimension and $c_{p,j}$ is the weight of some p in the j^{th} dimension. An integer linear program for the problem is provided in (2).

Multi-Dimensional Binary Knapsack Problem (`MBKP`)

$$\begin{aligned} \text{maximise} \quad & \sum_{p \in P} x_p \cdot v_p \\ \text{subject to} \quad & \sum_{p \in P} x_p \cdot c_{p,j} \leq B_j \quad \forall j = 1, \ldots, d \\ & x_p \in \{0, 1\} \quad \forall p \in P \end{aligned} \tag{2}$$

A simple demonstration of MBKP and an example of the corresponding greedy approach finding a sub-optimal solution is provided in Ex. 2. The greedy approach is essentially the same for this problem as for BKP, where the algorithm terminates once *any* of the capacities or *all* of the items have been exhausted. The multi-dimensional problem has the same number of combinations of possible solutions, i.e., 2^n, however, the additional constraints typically reduce the number of those solutions that are valid.

> **Example 2.** Consider an instance with capacity $B = (10, 20)$ and $n = 3$ items whose weights and values are defined $c = ((5, 10), (10, 2), (5, 5))$ and $v = (10, 20, 15)$. The greedy approach picks the item with the highest value, i.e., project 2 with a value of 20 and weights $(10, 2)$ in dimensions one and two, thus exhausting the capacity in dimension one and returning $\vec{x} = (0, 1, 0)$. However, items 1 and 3 have an overall value of 25 and an overall weight of $(10, 15)$. Clearly, $25 > 20$, $10 \leq 10$ and $15 \leq 20$ and thus $\vec{x} = (1, 0, 1)$ is the optimal solution. The greedy approach blocks optimality by including items with higher individual values.

Naturally, where BKP is essentially a case of MBKP in which $d = 1$, MBKP is also in the set of $\mathcal{NP}$-hard problems [32]. A proof of this known result is included in the appendix under Thm. 2. However, unlike BKP, MBKP is a *strongly* $\mathcal{NP}$-hard problem and thus cannot be solved in fully *or pseudo*-polynomial time. Thus, the run-time of any algorithm that can solve MBKP exactly must be bounded by an exponential in the number of items or number of constraints.

2.1.3 Fractional Knapsack Problem (FKP)

The fractional knapsack problem (FKP) is a continuous knapsack problem in which we may include *fractions* of items in the knapsack. More formally, we define our decision variables $x_p \in [0, 1]$ for all $p \in P$, such that $x_p > 0$ if and only if item p is included in the knapsack to some degree. An integer linear program for FKP is shown in (3).

Fractional Knapsack Problem (FKP)

$$\begin{aligned} \text{maximise} \quad & \sum_{p \in P} x_p \cdot v_p \\ \text{subject to} \quad & \sum_{p \in P} x_p \cdot c_p \leq B \\ & x_p \in [0, 1] \;\; \forall p \in P \end{aligned} \tag{3}$$

This problem is not directly relevant to CPB[1], however, FKP $\in \mathcal{P}$ and can be solved exactly by a polynomial time greedy algorithm [22]. More specifically, we sort the items by their value-to-cost ratio, fully include as many items as possible in this order until we find an item that cannot fully fit, and then include the fraction of that item that *can* fit in the knapsack. This approach is demonstrated in Ex. 3.

> **Example 3.** Consider an instance with capacity $B = 10$ and $n = 3$ items whose weights and values are defined $c = (4, 4, 5)$ and $v = (10, 8, 15)$. The optimal greedy algorithm for FKP computes their value-to-cost ratio by taking v_p/c_p for each $p \in P$ and thus we have $(2.5, 2, 3)$. The items with the highest ratios are included first, i.e., items 1 and 3, and then our remaining capacity is 1. The first item that cannot be included is item 2 with a weight of 4, and as such we include $1/4^{th}$ of item 2 to yield the allocation $\vec{x} = (1, 1/4, 1)$ with an overall value of 27 and an overall weight of 10.

This algorithm inspires two approaches for BKP and MBKP and thus CPB, namely ratio greedy and branch-and-bound, where in the former we use the value-to-cost(s) ratio and fully include as many items as possible in that order rather than taking a fraction of the first one that cannot fit, and in the latter we use this algorithm to find an upper bound or promise of each intermediate solution to improve our search strategy. These approaches are detailed in Sect. 3.

2.2 Participatory Budgeting (PB)

2.2.1 Literature Review

In their paper *Participatory Budgeting: Models and Approaches*, Aziz and Shah provide an excellent mathematical formulation of participatory budgeting problems [2], including a list of the key parameters involved in representing them. We provide a brief summary of these parameters below, adapted from their work and modified to suit our approach in addressing the problem.

Similarly to our knapsack problems, we have a set of *budgets* or *resources* $B \in \mathbb{N}_0^d$, where d is the number of dimensions and $d \in \mathbb{N}$, a set of *projects* or *proposals* $P = \{1, \ldots, n\}$, where n is the number of projects and $n \in \mathbb{N}_0$, and a *cost* function $c : P \to \mathbb{N}^d$ providing the cost(s) for each project $p \in P$. There is no intrinsic value for each project, and thus there is no natural value function v, but instead a set of *voters* $V = \{1, \ldots, m\}$ who vote over the projects, where m is the number of voters and

[1] Maximising the utilitarian welfare under the divisible model of PB, in which projects can be partially funded, can be reduced to FKP in the same way that CPB can be reduced to BKP and MBKP.

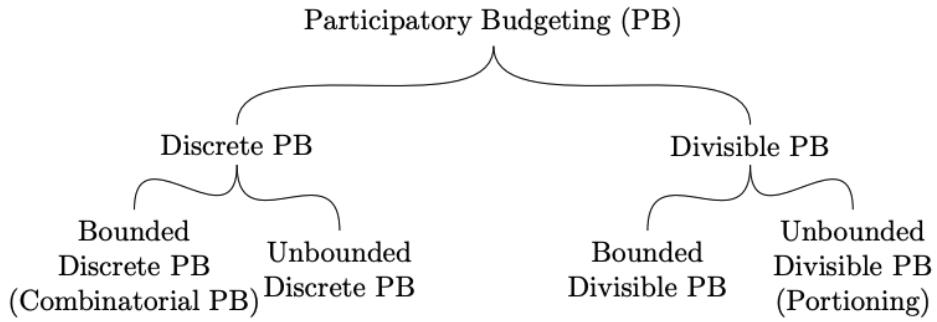

Figure 1: A taxonomy of participatory budgeting models from *Participatory Budgeting: Models and Approaches* [2].

$m \in \mathbb{N}_0$. These votes may be formulated as a cardinal *utility* function $u : V \to \mathbb{N}_0^n$ when voters submit implicit or explicit scores over the projects, or as an ordinal preference relation $\succ_i$ for each $i \in V$ when voters submit preference rankings over the projects. Finally, we introduce a set of *possible degrees of funding* $\mathcal{X}_p$ and a vector of decision variables $\vec{x} = (x_1, \ldots, x_n)$ such that x_p is the *actual degree of funding* for each $p \in P$ and thus $x_p \in \mathcal{X}_p$.

Aziz and Shah provide a taxonomy of key participatory budgeting models. This is shown in Fig. 1. It defines two primary types of model: discrete and divisible participatory budgeting, and four sub-models: bounded discrete or combinatorial, unbounded discrete, bounded divisible and unbounded divisible [2].

This project focuses on the discrete or combinatorial model of participatory budgeting (`CPB`), which is "perhaps the most widely studied and applied model of `PB`", and involves projects either being "fully implemented or not implemented at all" [2]. We say that in this model $\mathcal{X}_p = \{0, 1\}$ such that $x_p \in \{0, 1\}$ equivalently to the binary knapsack problems. An intuitive example in this setting is where the projects are proposals for bridges over a river cutting through an administrative region. The residents vote over which bridges should be built, where bridges must either be fully or not at all implemented. However, we are careful to note that `CPB` is applicable for most types of projects, including those which are theoretically divisible.

The remaining models are not in the scope of this project or report. However, a brief summary is provided here for the sake of completeness. The general discrete model permits projects with a discrete number of stages of implementation such that $\mathcal{X}_p \subseteq \mathbb{N}_0$, whilst the unbounded case allows for an infinite number of these stages, i.e., $\mathcal{X}_p = \mathbb{N}_0$. On the other hand, the general divisible model permits projects that can

be partially implemented such that $\mathcal{X}_p \subseteq \mathbb{R}_+$. The bounded and unbounded divisible models are defined similarly to the discrete models, i.e., $\mathcal{X}_p = [0, 1]$ in the bounded case and $\mathcal{X}_p = \mathbb{R}_+$ in the unbounded case.

A typical PB process is a short-term exercise lasting one year or two at most [6]. The community, i.e., residents, are involved throughout the whole process, often proposing projects themselves, before they are planned and vetted for the voting stage [2]. There are a number of important considerations in this voting stage, namely preference elicitation, ballot design and vote aggregation or budget allocation.

There are many ways that voters can express preferences or utilities over a set of projects. This project will consider four main types of voting: approval, cumulative, scoring and ordinal [39]. The most simple and common approach is approval voting, where "each voter is allowed to cast a single vote for each as of many candidates as they wish" [41], and thus $u_{i,p} \in \{0, 1\}$ for every voter $i \in V$ over each project $p \in P$. In other words, each voter i implicitly scores a project with one vote if they approve it, or zero votes otherwise. The number of approvals a voter can make may be capped in practice, e.g., [23], but this is relatively unimportant for our reduction.

In cumulative voting, each voter is allowed at most t votes and can choose how to distribute them across projects. We say that in this setting u_i is an n-dimensional vector such that $\|u_i\| \leq t \in \mathbb{N}_0$. In words, the number of votes distributed by each $i \in V$ over P must not exceed t. This rule is often employed in corporate voting, where the amount of votes a shareholder receives directly corresponds to the number of shares they own [9]. However, in this report we will make the assumption that t is some fixed constant for *all* voters such that they have equal say in the allocation.

The score-based or scoring voting method is similar to cumulative voting, except the constraint on the number of votes to be distributed is relaxed: "each voter submits a ballot vector that holds the scores that she gives to each of the candidates" [13]. More formally, we let $\mathcal{I}$ be the set of possible scores where $\mathcal{I} \subseteq \mathbb{N}_0$, and say that $u_{i,p} \in \mathcal{I}$ for all voters $i \in V$ over each project $p \in P$. In words, each voter gives each project a score in the set $\mathcal{I}$ without limits on the distribution or total of those scores.

Finally, the ordinal voting method involves residents submitting their preferences over the projects as a ranking from most to least preferred, where the voter is assumed to be indifferent between any projects that are not ranked. Mathematically, we say that each voter has a weak order preference relation $\succ_i$ over P where $\succ$ is the strict part and $\sim$ is the indifference part [2]. For example, given $P = \{1, 2, 3, 4\}$, and a ranking $\succ_i = 4 \succ 3 \succ 2 \sim 1$, we say that voter i prefers project four to three, and three to two

and one. This implies that project four is also preferred to two and one, and that the voter is indifferent between projects one and two. Previously, we discussed that voters and their preferences replace the value function v from our knapsack problems. However, the problem with ordinal voting is the lack of obvious numeric value to associate with the position of each project in the ranking, and thus the way projects should be valued under this rule. This project will adopt a modified Borda count approach for this conversion as described in Sect. 2.2.2.

After the voting stage, the individual votes must be aggregated in some way. The topic of vote aggregation is "perhaps the most well-studied aspect of the entire participatory budgeting process" [2], and in the literature there are many strategies to find allocations that satisfy various fairness axioms. The most relevant of these in this report is *welfare maximisation*, which combines the individual utility functions of voters into a societal utility function, and finds an allocation which maximises the value of this function. Fluschnik et al. describe three of these functions for a multi-agent knapsack problem: individually-best or utilitarian, diverse and fair [19].

This project focuses on the individually-best or utilitarian welfare function, which is defined in relation to our parameters as $\sum_{p \in P} \sum_{i \in V} x_p u_{i,p}$ to be maximised subject to the resource constraints. In words, we find the allocation which has the highest overall utility or number of votes without exceeding any of the resources. As detailed in Sect. 2.2.2 and 2.2.3, maximising this welfare function under `CPB` is reducible to `BKP` and `MBKP`, and through the greedy algorithm is the function that is used most often in practice. Unfortunately, [19] shows that this rule does not take into account fairness-related issues, e.g., it may leave some voters without any of their preferred projects funded. However, this is mostly out of scope for this report under the assumption that democracies maximising the utilitarian welfare do indeed wish to find an allocation with the highest overall utility, rather than some form of fairer allocation with a lower overall utility.

The diverse and fair social welfare functions, as well as the axiomatic approach [2], are alternative vote aggregation methods which satisfy certain fairness criteria. For example, the diverse function is defined in relation to our parameters as $\sum_{i \in V} \max_{p \in P}(x_p u_{i,p})$ to be maximised subject to resource constraints, and essentially makes the assumption that each voter cares only about their most preferred project. The different aggregation methods may yield very different solutions, and designing, implementing and comparing algorithms for these would certainly be an interesting area of future work.

2.2.2 Combinatorial Participatory Budgeting (CPB)

The maximisation of the utilitarian welfare under the combinatorial model of participatory budgeting (CPB), in which we aim to find an allocation that maximises the number of votes by fully or not at all funding projects, is reducible to MBKP and thus BKP in the one-dimensional case. Firstly, an integer linear program for CPB is shown in (4).

Combinatorial Participatory Budgeting (CPB)

$$\begin{aligned} \text{maximise} \quad & \sum_{p \in P} \sum_{i \in V} x_p \cdot u_{i,p} \\ \text{subject to} \quad & \sum_{p \in P} x_p \cdot c_{p,j} \leq B_j \quad \forall j = 1, \ldots, d \\ & x_p \in \{0, 1\} \quad \forall p \in P \end{aligned} \tag{4}$$

CPB (4) is mathematically very similar to BKP (1) and MBKP (2), and the reduction is as simple as setting the value $v_p = \sum_{i \in V} u_{i,p}$ for each project $p \in P$. The utility function u is naturally defined in the approval, cumulative and scoring voting methods, and thus the reduction takes polynomial $\mathcal{O}(mn)$ time, i.e., m steps per n projects. However, the ordinal preference case is more difficult where we must convert to utility values.

The most obvious conversion approach is to use the Borda count, in which "the last preference cast should receive one point, the voter's penultimate ranking should get two points, and so on." [16]. This method works effectively when all the preference relations have strict parts of the same length. For example, given $P = \{1, 2, 3\}$ and a preference relation $\succ_{i_1} = 3 \succ 1 \succ 2$, the corresponding utility vector is clearly $u_{i_1} = (2, 1, 3)$. However, fairness issues arise when the preference relations have strict parts of different length. For example, given preferences with only one project in the ranking, e.g., $\succ_{i_2} = 1 \succ 3 \sim 2$, we might choose to give the indifference part zero points and start the count at the strict part such that $u_{i_2} = (1, 0, 0)$. However, this only gives i_2 one vote for their most preferred project whilst i_1 was given three.

This project adopts a slight modification of this approach, in which we give votes counting down from n from the most to least preferred projects in the strict part and give zero votes to the indifference part, thus giving each position in the ranking the same number of votes. For example, $\succ_{i_2}$ would be converted to $u_{i_2} = (3, 0, 0)$. This runs in polynomial $\mathcal{O}(mn)$ time, i.e., m steps per n projects, and thus reducing CPB (4) to BKP (1) or MBKP (2) given ordinal preferences still runs in $\mathcal{O}(mn + mn) = \mathcal{O}(mn)$ time.

2.2.3 Real-World Budget Allocation

We have introduced BKP and MBKP and demonstrated the sub-optimal, arbitrarily bad greedy algorithm for both of these problems, and given a reduction from CPB to BKP in the one-dimensional case and to MBKP in the multi-dimensional case. Naturally, the greedy algorithm is then defined equivalently for CPB as exemplified in Ex. 4.

> **Example 4.** Consider an instance with budget $B = 100$, $n = 3$ projects whose costs are defined $c = (100, 50, 40)$ and $m = 3$ voters whose utilities are defined $u = ((1, 1, 0), (1, 1, 1), (1, 0, 1))$, forming the values $v = (3, 2, 2)$. The greedy algorithm fully funds project 1 exhausting the budget and returning $\vec{x} = (1, 0, 0)$ with an overall value of 3. However, funding projects 2 and 3 would have an overall value of 4 and an overall cost of 90. Clearly, $4 > 3$ and $90 \leq 100$ and thus $\vec{x} = (0, 1, 1)$ is the optimal solution.

The use of arbitrarily bad allocations produced by the greedy algorithm may have adverse impacts on the effectiveness of participatory budgeting processes. More specifically, the allocations produced by the algorithm will often use the available budgets or resources less effectively, often funding fewer projects and satisfying fewer voters overall.

There is little in the literature as to why democracies choose to employ the greedy algorithm for budget allocation. However, one could theorise that it is the simplicity and the speed of the approach that makes it the obvious choice. For example, it would be relatively easy to follow the steps in e.g. some spreadsheet software, even with limited technical expertise. The algorithm is fast from a computational standpoint, running in polynomial $\mathcal{O}(n \log n)$ time in the one-dimensional case and $\mathcal{O}(nd \log n)$ time in the multi-dimensional case, assuming that sorting takes $\mathcal{O}(n \log n)$ time.

The existing research in participatory budgeting from the perspective of computer science is generally concerned with identifying vote aggregation methods which satisfy various fairness axioms, e.g. [1, 18]. These theoretical approaches may be too complex to implement in practice, further increasing the attractiveness of the greedy approach. For example, maximising a welfare function or finding rules to satisfy axioms are non-trivial tasks, and budget decision-makers may lack the required expertise.

Furthermore, there is little to be found in the literature in regard to techniques to achieve optimality when maximising the utilitarian welfare in CPB, and seemingly very few code implementations of allocation techniques for e.g., budget-decision makers,

researchers or students to solve their own problems[2]. There is a large amount of research into and implementations of algorithms for `BKP`, e.g., [31], and a limited number of algorithms proposed for `MBKP`, e.g., [30]. However, we aim to design our own algorithms where possible, implement them in an open-source budget allocation library and evaluate the approaches in the context of `CPB` using randomly generated and real-world data [39], rather than from a broader, more general perspective.

[2]There are some existing proprietary and open-source tools to run participatory budgeting instances, e.g., [8, 38], but these are generally end-to-end tools with long set-up processes.

3 Optimisation Algorithms

This section is dedicated to the explanation, design and analysis of our optimisation algorithms for the one-dimensional (BKP) and multi-dimensional (MBKP) problems in the context of our reduced CPB problem. The exact algorithms sub-section describes the set of algorithms implemented which are guaranteed to find the optimal solution at the cost of speed, whilst the approximation algorithms sub-section describes the set of algorithms implemented which should theoretically run much faster at the cost of optimality. The implementation and code for each of these algorithms can be found in the community_knapsack/solvers/ directory in the library attached to the submission.

3.1 Exact Algorithms

3.1.1 Brute Force (BRF)

A brute force algorithm (BRF) is a naive approach to solving a problem in which all of the possible solutions are exhausted before the *best* one is output or returned. The one and multi-dimensional problems both have precisely 2^n possible allocations, and thus we must exhaust all of these in the brute force algorithm and output the one with the highest overall value that does not exceed any of the budgets or resources.

The set of all allocations for CPB in both the one and multi-dimensional case is equivalent to the power set of the projects, 2^P. For example, given $P = \{1, 2, 3\}$, the set of possible allocations is $2^P = \{\{\}, \{1\}, \ldots, \{2, 3\}, \{1, 2, 3\}\}$, and thus there are 2^n possible allocations to generate and consider. Our brute force algorithm generates these iteratively by using n-length bit string representations of numbers zero through $2^n - 1$, where each bit string can be seen as a vector of decision variables for the problem. For example, given $n = 3$, we generate bit strings 000, 100, 010, 001, 110, 101, 011 and 111. These are unique and thus exhaustive of all possible allocations where each bit p is the decision variable value for x_p. The value of each allocation is simply calculated by taking the sum of $x_p \cdot v_p$, whilst the cost of the allocation is calculated by taking the sum of $x_p \cdot c_p$ in the one-dimensional case and $x_p \cdot c_{p,j}$ in the multi-dimensional case.

The generation and iteration of 2^n possible allocations clearly takes exponential time in the number of projects n, specifically $\mathcal{O}(2^n)$. The one-dimensional case is dominated by the exponential part, and thus the linear cost and value calculations in n are insignificant. However, the multi-dimensional case requires d cost calculations, i.e., one in each dimension, and thus the overall time complexity in this case is $\mathcal{O}(2^n \cdot d)$, i.e., d steps per 2^n allocations.

BRF is not expected to perform well for anything other than problems with a very small

number of projects, i.e., it is guaranteed to take exponential time in n. Nevertheless, it plays a crucial role as a *worst-case* benchmark for our remaining algorithms. Furthermore, this naive approach paves the way to more efficient algorithms that optimise this process to avoid exhausting every allocation such as memoization and branch-and-bound.

3.1.2 Memoization (`MEM`)

A memoization algorithm (`MEM`) is a top-down dynamic programming approach that stores or caches results to avoid computing the same sub-problem in a recursive algorithm more than once. The dynamic program is well-defined for the one-dimensional (`BKP`) case [17] and the recurrence relation for this program is shown in (5).

$$\texttt{opt}(p, b) = \begin{cases} 0 & \text{if } p = 0 \\ \texttt{opt}(p-1, b) & \text{if } c_p > b \\ \max(\texttt{opt}(p-1, b), \texttt{opt}(p-1, b-c_p) + v_p) & \text{otherwise} \end{cases} \tag{5}$$

The recurrence relation essentially describes a recursive brute force algorithm that finds for any sub-problem $\texttt{opt}(p, b)$ the maximum value achievable with the first p projects and b budget, such that $\texttt{opt}(n, B)$ returns the optimal value. At any particular sub-problem, one branch is created which considers the funding of p if $c_p \leq b$, and thus that branch has a lower budget b, and another which considers not funding p. The maximum value of these two branches is then propagated up the recursion tree. This process *can* generate up to 2^n nodes, although this is typically much lower where the second recursive case prunes branches where project p cannot be funded. An example of the recursion tree generated by this brute force algorithm is shown in Fig. 2.

The recursive brute force approach still takes $\mathcal{O}(2^n)$ time in the worst-case, where each sub-problem computation creates two new sub-problems until the base case is reached and we cannot guarantee pruning. However, unlike the iterative brute force approach this upper bound is not guaranteed and thus we may expect it to be faster, although the overhead associated with recursive calls may offset this benefit.

The sub-problem results of the recursive brute force algorithm can now be *memoized* or *cached* to avoid re-computation. There are precisely $n + 1$ possible values of p, i.e., $\{0, \ldots, n\}$, and $B + 1$ possible values of b, i.e., $\{0, \ldots, B\}$, and thus the set of all possible function calls is $\{0, \ldots, n\} \times \{0, \ldots, B\}$. We assume that for any reasonably-sized instance, $n \cdot B < 2^n$, and thus it must be that we are repeating the computation of sub-problems. The sub-problem results are stored in a matrix or a hash table, such that at most $n \cdot B$ recursive calls are made and thus our overall time complexity is $\mathcal{O}(n \cdot B)$

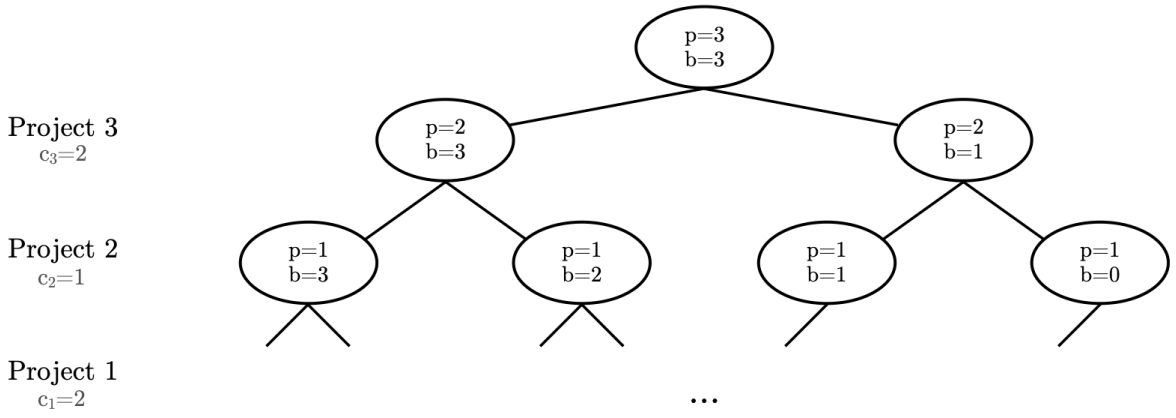

Figure 2: A simple example showing the exponential branching process of recursive brute force and the pruning of nodes where projects cannot fit.

in the worst-case, which is pseudo-polynomial in the magnitude of the budget.

There is seemingly no defined memoization or dynamic programming recurrence relation or algorithm for the multi-dimensional (`MBKP`) case in the literature or online. As such, we have designed a generalisation of (5) in which $\texttt{opt}(p, b)$ finds the maximum value achievable with the first p projects and b budgets or resources, such that b is a vector or tuple of budgets. The first recursive case occurs when $c_{p,j} > b_j$ for *any* $j = 1, \ldots, d$, whilst otherwise in the second recursive case *all* of the budgets in b must be updated before recursion, i.e., $b_j - c_{p,j}$ for *all* $j = 1, \ldots, d$. There are then $n \cdot B_1 \cdot \ldots \cdot B_d$ possible sub-problems, and thus we say that the overall time complexity in the multi-dimensional case is exponential in the number of dimensions, i.e., $\mathcal{O}(n \cdot \max(B)^d)$.

We expect this algorithm to scale fairly well as the size of the problem increases, and certainly much better than the brute force approach in both the one and multi-dimensional cases, although we would expect the latter to perform worse overall even for an identical problem due to the additional work involved with permitting an arbitrary number of dimensions. Additionally, we expect that the algorithm will perform very effectively when the project costs are high relative to the budget, i.e., where the first recursive case is able to prune inclusion or funding branches when e.g., $c_p > b$. Finally, it is important to recognise that this algorithm may also pose issues such as stack overflow errors, where the stack memory is exceeded by the number of recursive calls.

3.1.3 Dynamic Programming (`DYP`)

A bottom-up dynamic programming algorithm (`DYP`) similarly uses a recurrence relation but computes the result of all sub-problems in order from the base cases up

rather than computing the required sub-problems from the top down. This approach is very well-defined for the one-dimensional (`BKP`) case, e.g., [24, 40], and computes *all* the sub-problems in recurrence relation (5) to avoid the issues associated with recursion.

The one-dimensional case defines a matrix (or similarly, a hash table) `opt` with dimensions $(n+1) \times (B+1)$ initialised with zeroes in every address. The base cases are then implicitly defined in this matrix such that when $p = 0$ there is no value. The algorithm iterates through all possible sub-problems and stores the result such that for every $p \in \{1, \ldots, n\}$ we compute the best allocation with each $b \in \{0, \ldots, B\}$. The recursive cases are now essentially matrix look-up cases, where it is guaranteed by the ordering of the iteration that for any sub-problem the recurrence cases must have been computed. For example, given $p = 1$, the recursive cases will return zero as they are base cases, and so $\texttt{opt}(1, b) \leq v_1$ for all b such that all the recursive cases for $p = 2$ have been computed, etc., until the optimal solution is reached and stored in $\texttt{opt}(n, B)$.

As discussed, there is seemingly no defined dynamic programming recurrence relation or algorithm for the multi-dimensional (`MBKP`) case in the literature or online. As such, we can continue to use our generalisation of (5), although it is programatically difficult to use iteration with an arbitrary number of dimensions in this context. The solution devised was to generate the product of the sets of possible values, e.g., we compute the product of $\{0, \ldots, n\} \times \{0, \ldots, B_1\} \times \cdots \times \{0, \ldots, B_d\}$, and sort this by the budgets or resources such that each is considered sequentially for each project, ensuring that the sub-problems are available in the matrix or hash table `opt`.

The run-time complexity of the bottom-up dynamic programming algorithm is bounded identically to the memoization algorithm. However, the former uses iteration to compute *all* of the possible sub-problems and is thus guaranteed to take $n \cdot B$ steps in the one-dimensional case and $n \cdot \prod_{j=1}^{d} B_j$ steps in the multi-dimensional case, written as $\mathcal{O}(n \cdot B)$ and $\mathcal{O}(n \cdot \max(B)^d)$ respectively. However, the iterative approach circumvents the run-time overhead associated with recursive functions, and can process large instances without running the risk of exhausting the stack memory.

We expect that our bottom-up dynamic programming algorithm will perform similarly or perhaps slightly better than our memoization algorithm in the one-dimensional case except when the project costs are high relative to the budget and the latter can prune the search space efficiently. The multi-dimensional case is less certain where we must generate an exponential number of sub-problems in the number of dimensions, which likely performs very slowly in practice compared to e.g., nested loops.

3.1.4 Branch-&-Bound (BRB)

A branch-and-bound algorithm (BRB) is an approach to solving optimisation problems which involves breaking a problem down into sub-problems and then eliminating them when they cannot lead to the optimal solution. This approach is fairly well-defined for the one-dimensional (BKP) case, e.g., [29, 33], and involves dividing or *branching* into sub-problems considering the inclusion or exclusion of one project from an allocation, and pruning or *fathoming* these once we know they cannot lead to the optimal allocation.

The *branching* part of the algorithm can be visualised as a search tree as shown in Fig. 3. The tree is made up of levels of nodes, where the nodes represent individual possible allocations. At each level l, we consider including or excluding project l from each allocation by branching into two new nodes, and if any of these exceed the budget they are immediately fathomed and thus no longer considered.

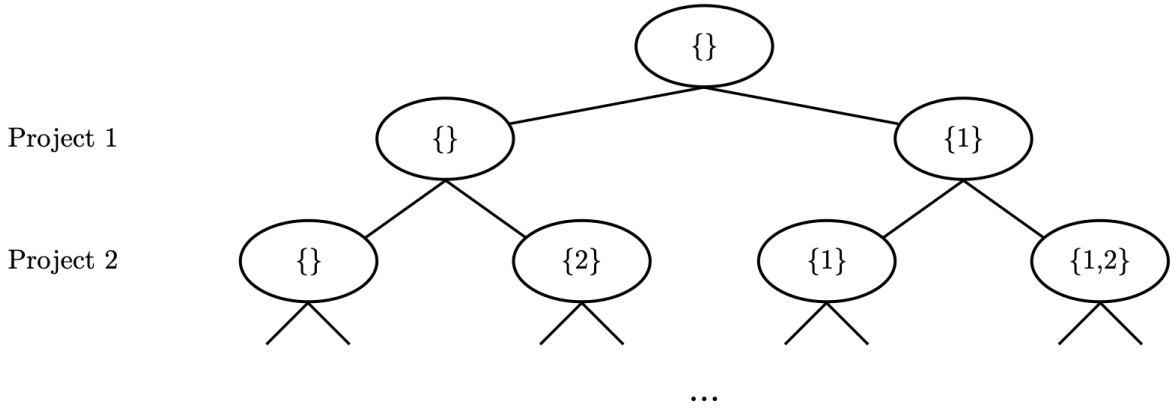

Figure 3: A simple example of the branching process in the branch and bound algorithm, which considers including or excluding a project at each level.

The *bounding* part of the algorithm attempts to estimate the potential of each allocation node as it is generated by solving a version of the problem with relaxed constraints such that it becomes polynomially solvable. More specifically, the constraint that each decision variable $x_p \in \{0, 1\}$ is relaxed such that $x_p \in [0, 1]$, and the maximum value achievable with the remaining projects and budget is calculated. The solution to the relaxed problem is called the *bound* for each node and indicates the potential with regard to optimality, and if the bound of a node is lower than the best value found at any stage in the algorithm then it is fathomed and thus no longer considered.

The relaxation in the one-dimensional (BKP) case forms FKP, which as discussed is optimally solvable in polynomial time by the greedy algorithm. Intuitively, the value

of the optimal solution to the relaxed problem must be equal to or higher than the optimal value of the original problem, and thus branch-and-bound is guaranteed to find the optimal solution in the one-dimensional case.

The relaxation in the multi-dimensional (`MBKP`) case seemingly cannot be solved exactly by a greedy algorithm and must instead be solved by e.g., the simplex algorithm [11]. We design and implement an alternative *approximate* branch-and-bound scheme which generalises the greedy algorithm for `FKP` to d dimensions by sorting the projects by their value-to-*costs* ratio, i.e., dividing the value of each project by the sum of all their costs. Unfortunately, this bounding algorithm is sub-optimal and thus our branch-and-bound algorithm is an approximation in the multi-dimensional case.

The worst-case run-time complexity of our branch-and-bound algorithm is equivalent to brute force, i.e., $\mathcal{O}(2^n)$ in the one-dimensional case and $\mathcal{O}(2^n \cdot d)$ in the multi-dimensional case, where there are no guarantees on the number of nodes that are fathomed and thus up to 2^n nodes or allocations are explored during the process. However, a large amount of fathoming or pruning will undoubtedly lead to a much faster output, i.e., $\Omega(n \log n)$ and $\Omega(nd \log n)$ in the best case, and we expect this will likely be affected by the magnitude of the costs relative to the budget similarly to memoization.

Ultimately, we expect branch-and-bound to perform relatively well, although this will likely be dependent on the characteristics of the problem and thus it is not guaranteed to output the optimal allocation in reasonable time. However, we can assume that in any case it will be fairly fast for problems with a small number of projects and dimensions. The multi-dimensional case is an approximation, and it is difficult to estimate the quality of the solutions without knowing how the generalised bounding technique performs, although we expect that it will be a feasible approximation scheme.

3.1.5 Integer Programming Solver (`CBC`)

An integer programming solver is a tool or software, e.g., [20, 26], that typically uses advanced optimisation techniques to solve integer programming problems. These often advertise their capability to solve difficult problems very quickly, and thus we make use of an open-source solver, namely `COIN-OR CBC` [20] through the `PuLP` [35] Python library for our one (1) and multi-dimensional (2) problems.

The `CBC` integer programming solver uses a branch-and-cut algorithm to find the optimal solution. The branch-and-cut technique is essentially an optimisation of branch-and-bound in which the constraints of our relaxed problems, e.g., `FKP`, are incrementally tightened to yield a more accurate bound for each node. As a result, more nodes can be fathomed earlier thus leading to optimality in fewer overall steps. However, we assume

that there are still no guarantees on the fathoming of nodes, and that the worst-case time complexity is then $\mathcal{O}(2^n)$ and $\mathcal{O}(2^n \cdot d)$ in the one and multi-dimensional cases.

The solver is written in `C++`, which is between ten and one hundred times faster than Python [3], and called by `PuLP`. Unfortunately, this will not lead to a very fair comparison of the algorithms in our evaluation, however the scalability of the solver may still indicate relative performance as the number of projects and dimensions increase. We expect that the branch-and-cut algorithm will likely be faster than *our* exact algorithms in any case due to the advanced optimisation techniques and the years of experience invested into the tools, although we would similarly expect it to slow down for large problems, giving rise to the need for speed and therefore our approximation algorithms.

3.2 Approximation Algorithms

3.2.1 Ratio Greedy (`RAG`)

The ratio greedy algorithm is an improvement to our defined greedy approach and is inspired by the optimal greedy algorithm for the fractional knapsack problem. The approach is well-defined for the one-dimensional (`BKP`) case, e.g., [25], and the intuition is to sort the projects by their value-to-cost ratios rather than just their values, such that the attractiveness of projects with high relative costs is reduced and thus the process blocks fewer projects that may lead to the optimal solution.

Similarly to our bounding technique in branch-and-bound, we can generalise this approach to the multi-dimensional (`MBKP`) case by sorting the projects by their value-to-*costs* ratio, i.e., the value of each project divided by the sum of all of its costs. As discussed, this technique is not optimal for the fractional multi-dimensional case, and so there is some uncertainty with regard to approximation performance in this setting.

The time complexity of ratio greedy is equivalent to the greedy algorithm, i.e., $\mathcal{O}(n \log n)$ in the one-dimensional case and $\mathcal{O}(nd \log n)$ in the multi-dimensional case, where the only additional work of computing the ratio is insignificant. As such, we would expect it to perform very similarly to the greedy algorithm with regard to run-time, although we expect that the ratio will yield much better approximations of the optimal solution in the one-dimensional case where less blocking will occur, and possibly better approximations in the multi-dimensional case.

3.2.2 Fully Polynomial-Time Approximation Scheme (`FPA`)

A fully polynomial-time approximation scheme (`FPA`) is an algorithm which finds approximations of optimisation problems in strictly polynomial time. Previously, we

discussed that the one-dimensional (BKP) problem is weakly $\mathcal{NP}$-hard and investigated two pseudo-polynomial dynamic programming approaches. However, the one-dimensional case also admits a fully polynomial-time approximation scheme by scaling the parameters to be within a polynomial factor of the number of projects n, thereby eliminating the pseudo-polynomial part. This approach is well-defined in the literature, e.g., [7, 31].

The approximation scheme scales the parameters and then solves the problem using a dynamic programming approach. However, the scaling process loses precision of these parameters and scaling down the budget or costs could lead to allocations which actually cost more than the available budget. As such, we must introduce and implement a new dynamic program whose recurrence relation is shown in (6).

$$\texttt{opt}(p, g) = \begin{cases} 0 & \text{if } g \leq 0 \\ \infty & \text{if } p = 0 \text{ and } g > 0 \\ \min(\texttt{opt}(p-1, g), \texttt{opt}(p-1, g-v_p) + c_p) & \text{otherwise} \end{cases} \tag{6}$$

The intention behind each $\texttt{opt}(p, g)$ sub-problem is to find the minimum weight possible to achieve at least value g with the first p projects such that after computing all possible sub-problems, i.e., all possible values $g \in \{0, \ldots, \sum_{p \in P} v_p\}$ for all projects $p \in P$, the optimal value must have been stored. This approach is less intuitive than our original recurrence relation (5), but still results in a pseudo-polynomial time complexity in the maximum value, i.e., $\mathcal{O}(n \cdot n \cdot \max(v)) = \mathcal{O}(n^2 \cdot \max(v))$, which can be scaled without the risk of finding allocations that over-spend the available budget.

The scaling process works by introducing an accuracy parameter $\epsilon \in [0, 1]$, computing a scaling factor $\theta = \epsilon \cdot \max(v)/n$ and setting $v_p = \text{round}(v_p/\epsilon)$ for all $p \in P$. The output value is then multiplied by θ to scale it to original size. The accuracy parameter ϵ decides how vastly each value is scaled, and thus the accuracy and speed of the algorithm. For example, a very large $\epsilon = 0.99$ is equivalent to saying that we will accept a solution that is within 99% of the optimal solution, where the scaled values are much smaller enabling the process to run quickly at the cost of accuracy, and vice versa for a very small ϵ.

The scaled run-time complexity for the fully polynomial-time approximation scheme is then reduced from $\mathcal{O}(n^2 \cdot \max(v))$ to $\mathcal{O}(n^3/\epsilon)$, where $\max(v)$ is divided by θ and $\theta = \epsilon \cdot \max(v)/n$, and thus the algorithm runs in fully polynomial time in the number of projects. The implementation in this project sets $\epsilon = 0.5$ by default to balance between optimality and run-time, and thus we expect it to perform much better than the dynamic programming approaches with regard to speed. Generally, we expect that the solutions will be better than 50% of the optimal value, but accept that these could

be lower than the greedy algorithm in the worst-case.

Lastly, we note that the multi-dimensional (`MBKP`) case does not admit a fully polynomial-time approximation scheme where it is strongly $\mathcal{NP}$-hard. As such, we do not design or implement an algorithm of this nature in this report or in our library. However, the concept of scaling down the problem parameters may still apply in these settings to form a fast, accurate approximation scheme that is more effective than the greedy algorithm and would be an interesting area of future work.

3.2.3 Simulated Annealing (`SIA`)

A simulated annealing algorithm is a local search approach that uses probabilistic techniques to converge on optima, inspired by the physical process of annealing [28]. This optimisation approach involves tracking a current solution and then moving between feasible solutions or allocations in search of the global optimum, where we can only move to a solution that is within the *neighbourhood* of our current solution at any time.

The neighbourhood of some solution $\vec{x}_{now}$ is the set of all feasible solutions that can be reached by transforming $\vec{x}_{now}$ in some predefined way. We define our neighbourhood as the feasible solutions that can be reached from $\vec{x}_{now}$ by flipping a single decision variable in $\vec{x}_{now}$, such that a single unfunded project becomes funded or vice versa. For example, from $\vec{x}_{now} = (1, 0, 1)$, we could move to $(1, 1, 1)$ or $(0, 0, 1)$, but not $(1, 1, 0)$. An important point here is that this neighbourhood may have local optima, e.g., if $(1, 1, 0)$ is the optimal solution but no neighbours of $\vec{x}_{now}$ have a higher value, then $\vec{x}_{now}$ is a local optimum and we need to go through a worse solution to reach the global optimum.

The simulated annealing parameters control the run-time and approximation accuracy of the algorithm. These are the initial temperature T_{start}, the temperature length T_{length}, the cooling ratio f and some stopping criterion, which we choose to be the stopping temperature T_{stop}. The general premise of the annealing process is to perform T_{length} loops per temperature T, where $T = T_{start}$ initially before being cooled after each set of T_{length} iterations, i.e., $T = T \cdot f$, until $T \leq T_{stop}$.

The algorithm is initialised with two variables x_{now} and x_{best} to track the current and best solutions respectively, where x_{now} is initially the empty allocation and $x_{best} = x_{now}$. A neighbouring allocation is generated for each iteration $1 \dots T_{length}$, and is immediately discarded or skipped if it is invalid, i.e., exceeds any of the resources. The current allocation $x_{now} = x_{next}$ immediately if the value of x_{next} is greater than the value of x_{now}, and $x_{best} = x_{next}$ if it is greater than the value of x_{best}. Otherwise, $x_{now} = x_{next}$ with probability $e^{-\Delta E/T}$, where ΔE is the difference in values. The temperature affects the probability of worse solutions being selected and thus this probability is decreased

as the temperature is reduced by the cooling ratio. This allows the process to escape local optima in neighbourhoods earlier in the process, whilst avoiding moving away from good solutions towards the end of the process.

The time complexity of the simulated annealing algorithm is dependent on the number of projects and the simulated annealing parameters in the one-dimensional (`BKP`) case, and the number of projects, simulated annealing parameters and dimensions in the multi-dimensional (`MBKP`) case. More specifically, the neighbour generation takes $\mathcal{O}(n)$ and $\mathcal{O}(nd)$ time, and generation is performed continuously T_{length} times until $T \leq T_{stop}$. The time complexity can be defined by letting q be the number of times T must be cooled before the algorithm stops where initially $T = T_{start}$ such that $T_{start} \cdot f^q \leq T_{stop}$. We can rearrange for q such that $q \leq \log(T_{stop}/T_{start})/\log(f)$ and our overall worst-case time complexity is $\mathcal{O}(n \log(T_{stop}/T_{start})/\log(f))$ in the one-dimensional case or $\mathcal{O}(nd \log(T_{stop}/T_{start})/\log(f))$ in the multi-dimensional case.

The performance of this algorithm largely depends then on the problem size and the parameters, and we have defined these to be $T_{start} = 1.0$, $T_{length} = 50000$, $f = 0.9$ and $T_{stop} = 0.5$ by default based on trial-and-error testing during the implementation of the algorithm. These appeared to give reasonable approximations to both one and multi-dimensional problems. However, where these parameters are fixed and do not scale with the problem size, we see that the algorithm becomes less effective as the number of projects increases where the exponential number of allocations require a very large number of iterations. This may be especially apparent in the multi-dimensional case where more allocations are invalid and thus discarded or skipped.

3.2.4 Genetic Algorithm (`GEN`)

A genetic algorithm is a global search approach that uses probabilistic techniques to converge on optima, inspired by the process of natural evolution or equivalently survival of the fittest [27]. This optimisation approach involves maintaining a population of chromosomes representing possible solutions and evolving this population towards the optimal solution through fitness-based selection, crossover and mutation.

A chromosome is defined as a string of binary numbers called *genes* which represent a solution, and each chromosome has a *fitness* or value. The one and multi-dimensional problems are easily encoded by setting a chromosome to be a possible allocation whose genes are decision variables, such that each gene represents a single project and each chromosome is n-bits long, and whose fitness is simply the overall value of the allocation or zero if the allocation is invalid. A population of σ of these chromosomes is maintained at any time, where σ is a parameter of the algorithm.

The algorithm works by first generating σ empty allocation chromosomes[3], and then performing e generations or evolutions of the population, where e is a parameter. The convergence towards the optimal solution is lead by fitness-based selection and results in the population containing increasingly fit allocations or chromomes, and as such the output of the algorithm is the best chromosome in the population after e evolutions.

The evolution process starts by selecting two parent chromosomes α and β from the population based on fitness values, i.e., chromosomes with a higher fitness have a higher chance of being selected. The chromosomes α and β are then set to be the offspring and are possibly crossed-over with some probability P_{cross}, in which a random point between $0 \dots n$ is selected and the genes of α and β are swapped after this point, where P_{cross} is a parameter of the algorithm. Each chromosome is then possibly mutated with some probability P_{mut}, in which a random gene is selected and flipped, where P_{mut} is another parameter of the algorithm. Lastly, the possibly crossed-over and mutated offspring α and β replace their parents in the population.

The time complexity of the genetic algorithm is dependent on the number of projects, σ and e in the one-dimensional (`BKP`) case, and the number of projects, σ, e and the number of dimensions in the multi-dimensional (`MBKP`) case. More specifically, the population generation and each of fitness-based selection, crossover and mutation run in $\mathcal{O}(n)$ and $\mathcal{O}(nd)$ time, and the algorithm performs e evolutions and creates σ offspring in each evolution, and thus the time complexity of the algorithm is $\mathcal{O}(e\sigma n)$ in the one-dimensional case and $\mathcal{O}(e\sigma nd)$ in the multi-dimensional case.

The expected performance of our genetic algorithm largely depends on the problem size and the parameters, and we have defined these to be $\sigma = 200$, $e = 100$, $P_{cross} = 0.8$ and $P_{mut} = 0.3$ by default based on trial-and-error testing during the implementation of the algorithm. These appeared to give good approximations to both one and multi-dimensional problems. However, similarly to our simulated annealing algorithm, where these parameters are fixed and do not scale with the problem size, we see that the algorithm becomes less effective as the number of projects increase where the exponential number of allocations require very large numbers of evolutions. Additionally, we expect that the algorithm becomes less effective as the dimensions increase where the constraints may make it difficult to evolve to valid solutions.

[3]The chromosomes in genetic algorithms are often randomly generated, although we found this to be less effective where often the population only contained invalid solutions therefore rendering fitness-based selection useless.

4 Budget Allocation Library

4.1 Objectives & Prerequisites

4.1.1 Aims & Requirements

The overall aim of our budget allocation library is to provide a platform to maximise the utilitarian welfare of one and multi-dimensional combinatorial participatory budgeting problems using our exact and approximation algorithms. The target audience is firstly ourselves, where we use the library to evaluate our algorithms over a variety of randomly generated and real-world problems [39], and secondly budget decision-makers, researchers or students who may wish to compute allocations for their own instances without the need for extensive knowledge in the field of optimisation or computer science.

The basic *must-have* requirements of the library then are to provide a simple application programming interface to input one and multi-dimensional `CPB` problems, select one of the available algorithms and solve the problem using it to receive an allocation including the list of projects funded, the overall value or number of votes and the overall cost(s). The library must implicitly reduce the problem to `BKP` or `MBKP`, and run the selected algorithm on this reduction to obtain an allocation.

The simplicity element of the library is handled through our design, as well as our decision to use Python as the programming language and foundation of the library. Python is an intuitive and accessible language that is generally effortless to set up and write, has a straightforward package installation process, and thus supports the use of our platform by those with varying levels of technical expertise. We do assume however that users have at least a basic understanding of the language such that they can import the library, input problems and solve them as required.

The input of one and multi-dimensional problems has many viable approaches, however, we choose to adopt and recommend a `csv`-based data format defined by Stolicki et al. [39], which is already being used to store data from more than seven hundred previous real-world participatory budgeting instances. Additionally, the library supports the random generation of problems for our evaluation and general testing purposes, and the manual input of data in the simplest case.

Finally, the algorithm selection and solving elements of the library allow users to select a one-dimensional algorithm for one-dimensional (`BKP`) problems, or a multi-dimensional algorithm for multi-dimensional (`MBKP`) problems, and pass a timeout value to terminate the solving process after a certain number of seconds. The latter feature is particularly important for our evaluations, and prevents algorithms running infinitely when solving

intractable instances, which is particularly important for our exact or optimal techniques.

4.1.2 Parsing Real-World Data

The primary and recommended method of inputting participatory budgeting data into the library is through the `csv`-based data format described by Stolicki et al. [39], hereafter referred to as the `.pb` data format. The format supports the approval, cumulative, scoring and ordinal voting methods described in Sect. 2.2, and contains the projects and their costs, the voters and their preferences over the projects, and metadata such as `country` and `district` which is less important for this project.

```
META
key;value
country;Poland
unit;Warszawa
subunit;Ursynow
instance;2019
district;Ursynow
num_projects;58
num_votes;7683
budget;2000000
vote_type;approval
rule;greedy
...
PROJECTS
project_id;cost;votes
659;20000;3349
1024;175000;3216
1061;149000
...
VOTES
voter_id;vote
16;640,643,1554
140;643,1534,1619
147;184,641,1072
...
```

Figure 4: An example of the `csv`-based `.pb` data format defined by Stolicki et al. [39] to represent data from participatory budgeting problems.

Fig. 4 provides an example of the data format, which includes the problem metadata, the project proposals and the voters and their utilities or preferences over the projects. This particular example has fifty-eight projects, a budget of two million and uses approval voting such that the `vote` of each voter is simply a comma-separated list of the projects

that they approve. The cumulative, ordinal and scoring voting types have slightly different layouts and are not shown here but are all supported by our library.

One problem identified with this format is that there is no obvious support for multiple dimensions in the specification, and all of the previous instances recorded appear to be one-dimensional. Our library must permit both one and multi-dimensional problems, and as such we extend the data format to allow the `budget` field in the metadata and the `cost` field in the projects list to be a comma-separated list of costs in each dimension. However, whilst this is of benefit to future users, it is not hugely relevant to the remainder of this report where there are no multi-dimensional real-world instances to evaluate in the `.pb` format.

4.2 Design & Approach

4.2.1 Activity Overview

The aims and requirements give rise to the activity diagram for the library provided in Fig. 5, which shows the flow from creating a problem to obtaining a budget allocation. The general idea is that the library should expose three methods or classes to encode a problem for solving: a manual or hard-coded method, a parsed method as discussed and a random generation method to generate problems within user-specified bounds.

These methods then create either a one or multi-dimensional problem object which can be solved by selecting an algorithm and optionally specifying a timeout value in seconds. As discussed, the optional timeout value enables the user to specify how long the selected algorithm should run for in seconds before the library terminates the process and returns the empty allocation, and if no timeout is specified the algorithm will run until it completes. This is a useful feature where some of the algorithms could run for an essentially infinite amount of time for very large problems, and thus prevents the entire program from having to be forcibly terminated when an algorithm is taking too long.

Finally, the algorithm returns the allocation found by the solving process, or the empty allocation if the run-time exceeded the specified timeout. The allocation should include a list of the project ids funded, the overall value and cost(s) of these, as well as the run-time of the algorithm and whether or not it is an approximation of the optimal solution. Naturally, the overall cost of the allocation, or cost in each dimension in the multi-dimensional case, will not exceed the budget(s) in any dimension.

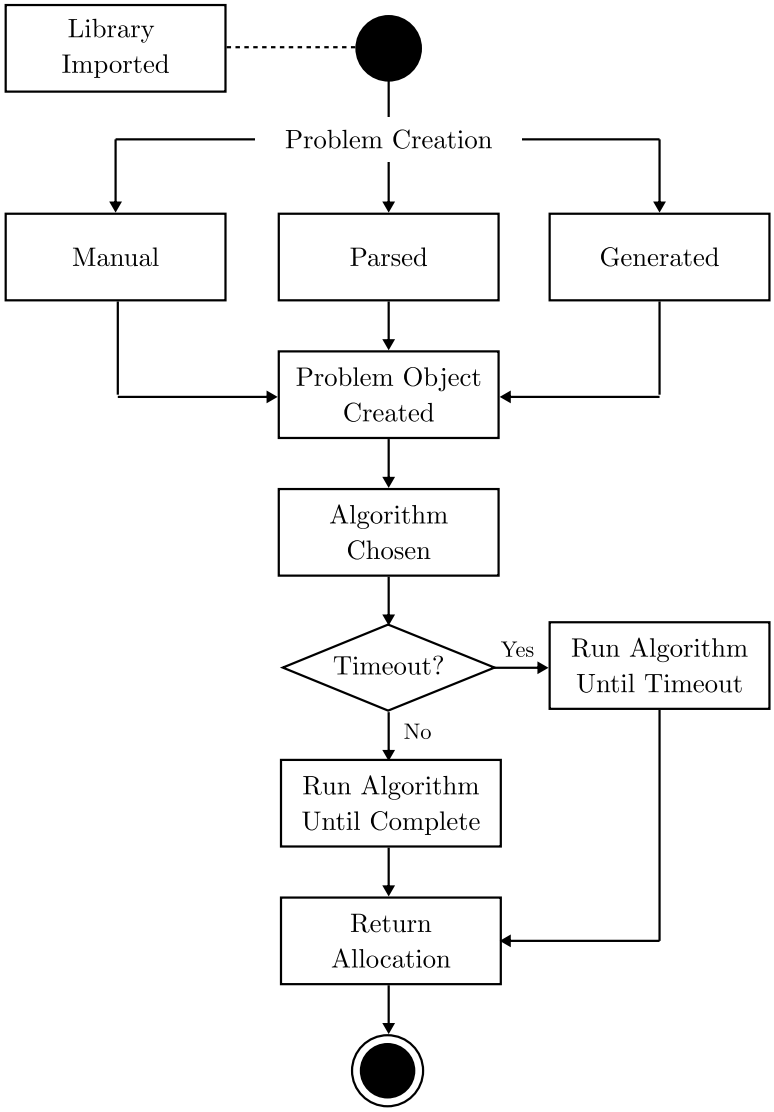

Figure 5: A high-level activity diagram showing the steps from importing the library to returning an allocation.

4.2.2 Class Overview

The library follows an object-oriented design and wraps around the algorithm functions to provide a simple interface to create and solve problems. Fig. 6 and 7 present the class structure and visualise the interaction between objects[4], where the former shows the classes available to create problems, and the latter shows the classes involved in solving problems. These diagrams are not an *exact* representation of the implementation in practice, but provide an overview of the design and functionality of the system.

`PBSingleProblem` and `PBMultiProblem` are objects representing one and multi-dimensional problems respectively, and encapsulate the problem data including the budget or budgets, projects, costs, voters and utilities. These classes can be directly instantiated through their constructors by passing in the data for each of the displayed fields, thus satisfying our requirement to permit manual problem creation.

The `PBParser` class parses files in the `.pb` data format into `PBSingleProblem` and `PBMultiProblem` objects. The parser is constructed with the path to the file, `file_path`, and the problem objects can be obtained through the `single_problem()` and `multi_problem()` methods. These implicitly call the `parse()` to parse the contents of the file. One possible drawback of this design is that the user must know whether they have a one or multi-dimensional problem to obtain the correct object and algorithms, and this is mitigated through the documentation and by allowing the `multi_problem` method to work for one-dimensional problems, although the algorithms will likely be slower where they must accommodate an arbitrary number of dimensions.

The `PBGenerator` class randomly generates `PBSingleProblem` and `PBMultiProblem` objects within some user-specified bounds. The generator object is constructed with a value to `seed` the random generator such that generated problems are reproducible. The `generate_single_problem` and `generate_multi_problem` methods generate one and multi-dimensional problems respectively within user-specified bounds for each parameter. For example, passing `projects_bound=(20,100)` to one of the methods will generate a problem with at least twenty but no more than one hundred projects. The input of the functions are very similar, except that in the multi-dimensional case the budget and cost bounds must be provided for *each* dimension.

The `PBSingleProblem` and `PBMultiProblem` classes extend the abstract `PBProblem` class, which factors out the implicit reduction of the voter utilities to project values at construction and the `solve` method to avoid code duplication in the individual problem classes. The input for the `solve` method is a `PBSingleAlgorithm` or `PBMultiAlgorithm`

[4]The class diagram been split into two to more clearly represent the design and structure of the library.

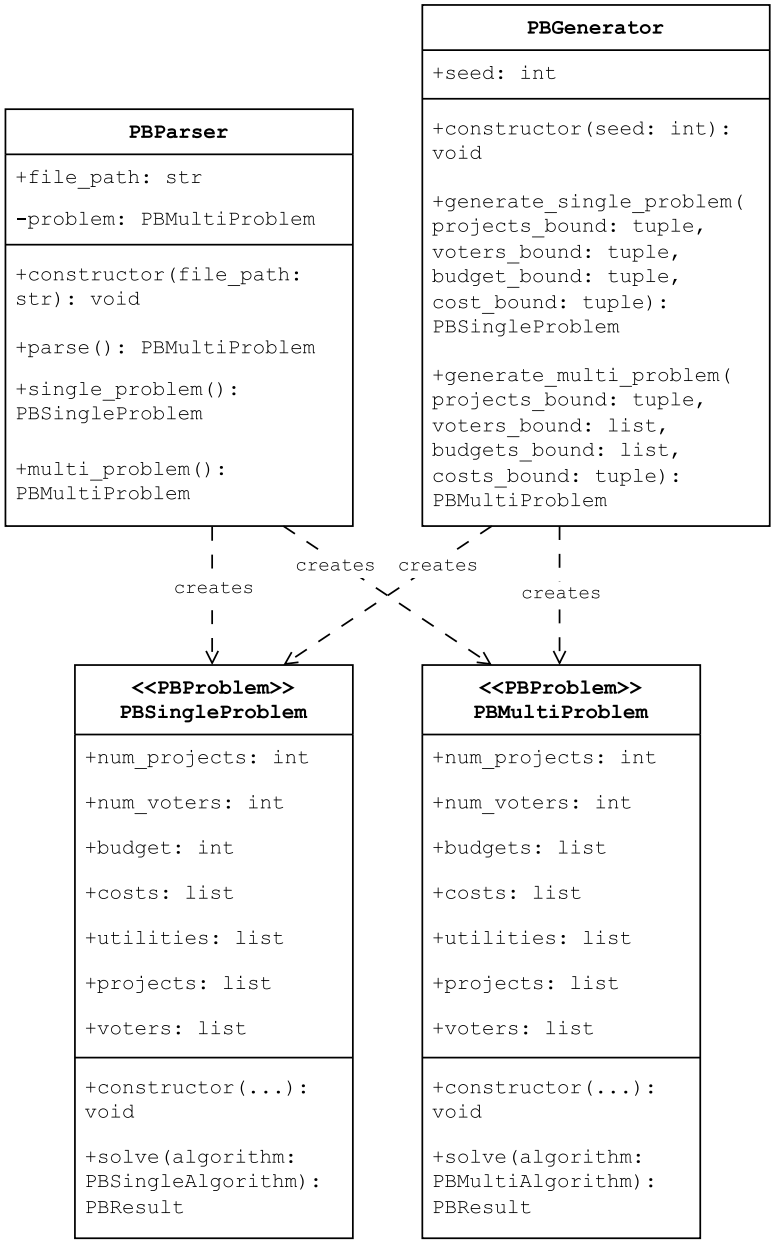

Figure 6: A class or object diagram presenting the classes and methods available to create problem objects through manual entry, parsing or random generation.

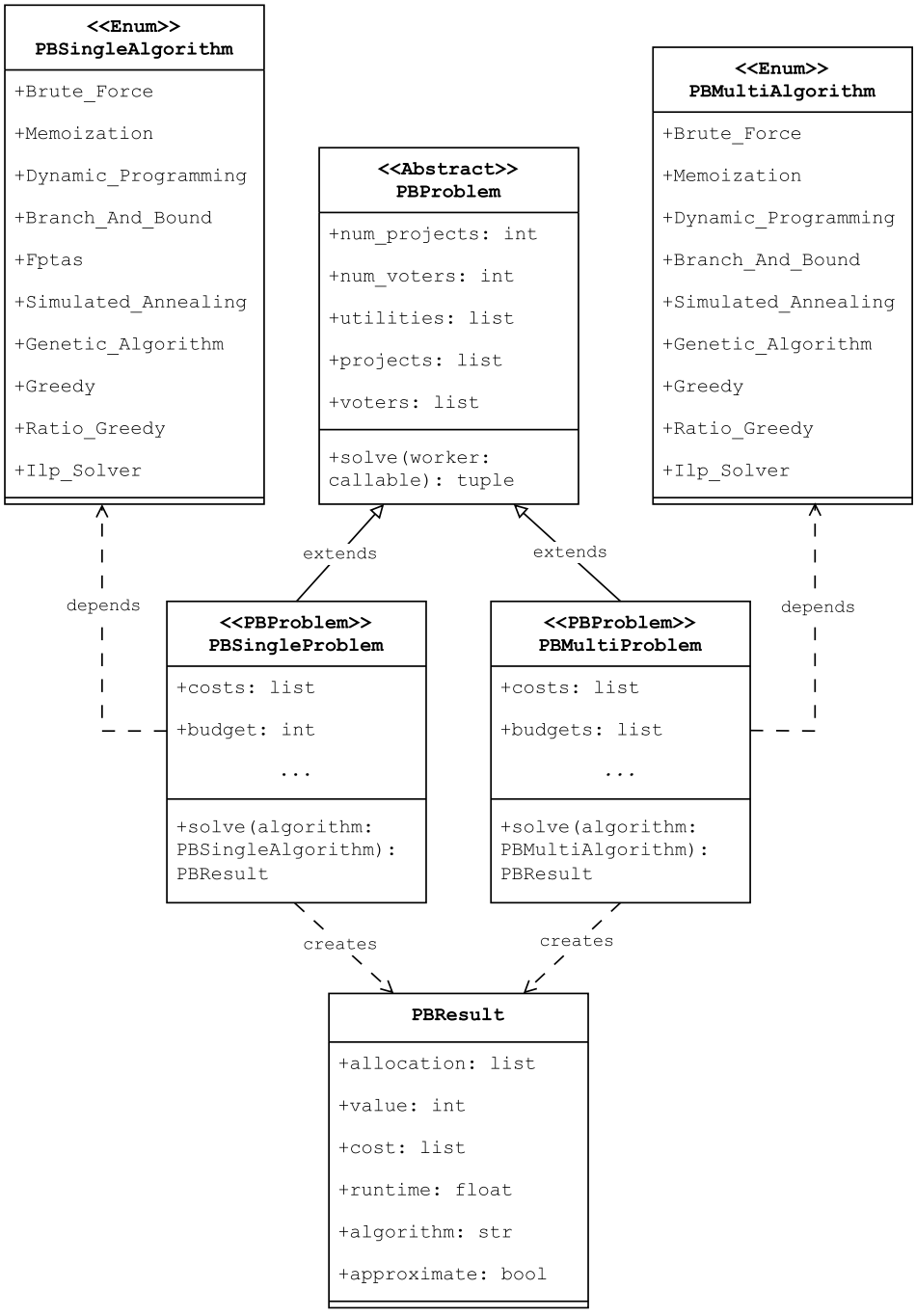

Figure 7: A class or object diagram presenting the class structure and methods available to solve `PBSingleProblem` or `PBMultiProblem` objects and obtain allocations.

enumerator value representing the `algorithm` to use to solve the problem and an optional `timeout` value in seconds determining when the algorithm should be terminated. The methods then submit a locally-defined worker function to the super class which spawns and handles a thread to run the algorithm through the worker function and terminate it after the specified number of seconds.

Finally, the `solve` methods return a `PBResult` object containing the allocation computed by the algorithm, or the empty allocation if the run-time exceeded the timeout. As expected, this object contains a list of funded project ids, the overall value and cost(s) of those funded projects, the run-time and name of the algorithm, and whether or not the algorithm is exact or an approximation scheme.

4.2.3 Algorithm Modules

The one and multi-dimensional algorithms are written as stand-alone functions in modules which the `solve` function calls based on the passed `PBSingleAlgorithm` or `PBMultiAlgorithm` enumerator value. Fig. 8 shows the structure of these modules, where the exact algorithms are accessed from the `exact` module and the approximation algorithms are accessed from the `approximate` module.

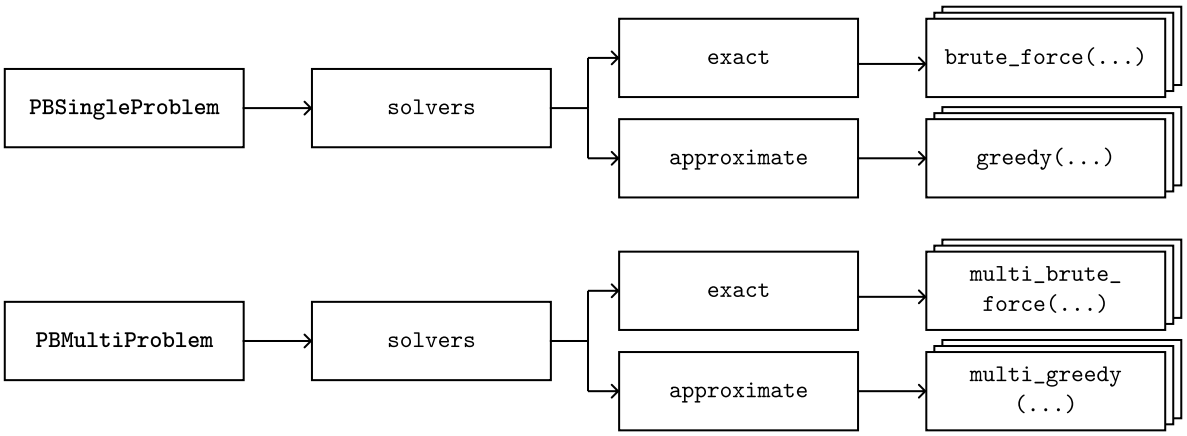

Figure 8: A diagram showing the algorithm modules and functions accessed by the `PBSingleProblem` and `PBMultiProblem` classes.

The naming of the functions is based on the algorithm for the one-dimensional functions, e.g., `greedy` for the greedy algorithm or `dynamic_programming` for the dynamic programming algorithm, whilst the multi-dimensional functions are prefixed with `multi`, such that we call e.g., `multi_greedy` or `multi_dynamic_programming`. The former functions always accept three parameters: a single `budget`, an array of `costs`

and an array of `values`, whilst the latter functions always accept an *array* of `budgets`, a two-dimensional array of `costs` and an array of `values`. Additionally, the approximation algorithms `FPA`, `GEN` and `SIA` optionally accept their own parameters, such as ϵ as the `accuracy` parameter or T_{length} as the `temperature_length` parameter.

Generally, we do not expect or necessarily recommend that these functions are called directly where the problem classes handle the reduction and solving of the problem implicitly. However, we recognise that the classes and design of the library does not support the modification of algorithm-specific parameters through the `solve` function to avoid overloading or complicating the interface. As such, more technically experienced users such as researchers or students may choose to import and call the functions from the library directly to fine-tune the performance of the approximation algorithms.

4.3 Implementation

This section describes and provides code excerpts for important and interesting parts of the library, such as the solving the process, the parsing of the `.pb` data format and the random generation of one and multi-dimensional problems. The entire codebase of the library can be found in the `community_knapsack/` directory of the attached submission.

4.3.1 Problem Classes

`PBProblem`, `PBSingleProblem` and `PBMultiProblem` are the most fundamental classes in the library. The construction of these classes is relatively self-explanatory, and mainly involves validating and storing the input in the object. However, the constructor also handles the reduction or aggregation of the individual voter utilities into values for each of the projects through the `aggregate_utilities` function shown in Fig. 9. As discussed, this involves simply looping through each of the voters and taking the sum of their utilities for each project.

```python
def aggregate_utilitarian(num_projects: int, utilities: Sequence[Sequence[int]]) -> List[int]:
    values: List[int] = [0] * num_projects

    for vid, utility in enumerate(utilities):
        if len(utility) != num_projects:
            raise ValueError(f'Voter {vid} has utilities for {len(utility)} projects but expected utilities '
                             f'for {num_projects} projects.')
        for pid, u in enumerate(utility):
            values[pid] += u

    return values
```

Figure 9: A code excerpt showing a function to aggregate the voter utilities into project values by summing the votes for each project.

The `solver` methods in these classes are more complex where the individual problem objects submit a worker function to run the algorithm, and a new thread must be created to run this worker in order to terminate the process once the timeout is exceeded. Fig. 10 and 11 provide code excerpts from the `worker` function in the `PBSingleProblem` class and the `solve` method in the abstract `PBProblem` class.

```
def _worker(self, algorithm: PBSingleAlgorithm, values: List[int], result_queue: mp.Queue) -> None:
    allocation, value = algorithm(self.budget, self.costs, values)
    result_queue.put((allocation, value, sum(
        self.costs[idx] for idx in allocation
    )))
```

Figure 10: A code excerpt showing the `worker` function defined in the `PBSingleProblem` class to run the specified one-dimensional algorithm.

```
# Create multiprocessing queue to receive allocation:
result_queue: mp.Queue = mp.Queue()

# Start the algorithm process and wait until the timeout is reached:
process: mp.Process = mp.Process(target=self._worker, args=(algorithm, self.values, result_queue))
process.start()
process.join(timeout if timeout >= 0 else None)

# Terminate the process if it is still running and return the empty allocation:
if process.is_alive():
    process.terminate()
    warnings.warn(f'The {algorithm.name} algorithm did not finish within the {timeout} second timeout limit. '
                  f'Try increasing the timeout or using a different algorithm (such as an approximation '
                  f'scheme).')
    return PBResult([], 0, 0, timeout, algorithm.name, algorithm.is_approximate())
```

Figure 11: A code excerpt from the `solve` method defined in the `PBProblem` class that creates a thread and runs the worker function.

The `worker` function in practice is actually a method overridden by the sub-classes such that it can be called by the `solve` method in the super-class yet still access the budget and cost data. As specified in the design, this avoids duplicating the code in the separate classes. The worker is then run on a separate thread through the built-in `multiprocessing` module, and stores the algorithm output in a queue to be accessed after the algorithm finishes running. If a `timeout` value is specified, the main thread will begin running again no more than `timeout` seconds after starting the algorithm and will terminate the worker and return the empty allocation if it has not completed.

Another noteworthy implementation detail from the problem classes and solving process is the use of higher-order functions in the enumerator values of `PBSingleAlgorithm` and `PBMultiAlgorithm`, where the value of each field is the corresponding algorithm

```
class PBSingleAlgorithm(_PBAlgorithm):
    BRUTE_FORCE = (solvers.exact.brute_force,)
    """A very slow but exact algorithm that enumerates every possible allocation and returns the best (optimal) one.
    This is likely too slow and is very rarely applicable."""

    MEMOIZATION = (solvers.exact.memoization,)
    """A relatively slow pseudo-polynomial, exact algorithm that improves upon the brute force algorithm for a faster
    result."""
```

Figure 12: A code excerpt showing the use of higher order functions in the `PBSingleAlgorithm` enumerator where the values are algorithm functions.

function as demonstrated in Fig. 12. The enumerator values are then used directly in the `worker` function as shown in Fig. 10 to avoid long if-else clauses for all of the possible value cases thus promoting cleaner and extensible code.

4.3.2 Parsing Problems

The `PBParser` class adapts a piece of code provided by Stolicki et al. [39] to parse files following the `.pb` data format and form `PBSingleProblem` and `PBMultiProblem` objects. The general approach to this is to parse all these files as multi-dimensional problems and then convert these to one-dimensional problems when `single_problem()` is called by capturing only the first dimension of budgets and costs. This strategy avoids code duplication where we may otherwise parse the two types of problem separately, and thus improves the readability of the class and methods.

The majority of the `parse()` function involves using the `csv` package to read the file and then verifying whether the input is in the correct format. One important feature of this method is the conversion from approval, cumulative, scoring and ordinal voting to utility values over the projects. As discussed, the approval, cumulative and scoring methods are straight-forward where we set the number of votes or points that each voter gives to a project as their utility for that project. However, the ordinal voting method must use our modified Borda count approach from Sect. 2.2.2 to derive utilities from the preferences or ranking of each voter.

Fig. 13 provides an excerpt from a `vote_to_utility` function which converts the vote format from the `.pb` file to a list of utilities over the projects for each voter. This function takes a list of project ids, `votes`, and a list of `points` under the cumulative or scoring voting methods. The approval case simply gives each project one utility for each project in `votes`, whilst the cumulative and scoring cases give `points` utility to each corresponding project in `votes`.

Fig. 14 provides an excerpt from the `ordinal_to_utility` function which converts

```
utility: List[int] = [0] * num_projects

# Approval voting gives all projects in `votes` a utility of one:
if vote_type == 'approval':
    for vote in votes:
        utility[vote] = 1
    return utility

# Cumulative and scoring gives projects in `votes` their points:
if vote_type in ('cumulative', 'scoring'):
    for idx, vote in enumerate(votes):
        utility[vote] = points[idx]
    return utility

# Ordinal voting uses the modified Borda count:
if vote_type in 'ordinal':
    return ordinal_to_utility(num_projects, votes)

return utility
```

Figure 13: A code excerpt showing the conversion of the approval, cumulative, scoring and ordinal voting methods to voter utilities over the projects.

```
# Use the Borda count, but limit the maximum utility to max_vote_length
# if it is specified, or num_projects otherwise:
utility: List[int] = [0] * num_projects
curr_vote: int = min(max_vote_length, num_projects)

for preference in votes:
    utility[preference] = curr_vote
    curr_vote -= 1

return utility
```

Figure 14: A code excerpt showing the modified Borda count approach to derive voter utilities over the projects from a preference ranking.

the preference rankings to utilities via our modified Borda count approach. In this case, we assume that the projects in `votes` are ordered from most to least preferred, and assign utilities by counting down from `num_projects` or `max_vote_length` whilst giving all unranked projects zero utility. The `max_vote_length` parameter is accepted by the function and becomes the maximum utility when we know that no voter has ranked more than that number of projects and thus all positions will receive the same utility.

4.3.3 Generating Problems

The `PBGenerator` class creates `PBSingleProblem` and `PBMultiProblem` objects by randomly generating each parameter between some user-specified bounds. This imple-

mentation of this class is mostly straight-forward, although the way in which utilities are generated for each voter is noteworthy and important for our evaluation. In practice, we limit the generations to approval voting, i.e., where each voter can vote at most once for each project, and give each project a *popularity* rating to attempt to simulate the natural preferences of voters in real-world instances.

```
utilities: List[List[int]] = []
probabilities: List[float] = [min(max(self._random.gauss(0.5, 0.2), 0.2), 0.8) for _ in range(num_projects)]

for voter in range(num_voters):
    utility: List[int] = [0] * num_projects
    for project in range(num_projects):
        if self._random.random() < probabilities[project]:
            utility[project] = 1
    utilities.append(utility)

return utilities
```

Figure 15: A code excerpt showing the generation of approval utilities for each voter over the projects where the projects have popularity ratings.

These popularities, named `probabilities` in the implementation, are generated using the built-in `random.gauss` function to form a normal distribution of popularities such that a small selection of the projects are either very popular or unpopular, whilst the majority are somewhere in the middle. The intuition then is to iterate through each project for each voter, and then have the voter possibly approve each project with a probability equal to its popularity, such that they have a high chance of voting for popular projects and a low chance of voting for unpopular projects, etc.

Overall, it is difficult to simulate possibly large, multi-agent scenarios realistically, and as such randomly generated instances may not perfectly represent real-world scenarios. However, we can generally assume based on our analysis that the algorithms are unlikely to be dependent on the distribution or magnitude of the utilities, or as a result whichever voting method was used, and therefore we are confident in the use of randomly generated problems in our evaluation.

4.3.4 Algorithm Functions

The research, design and implementation of the algorithms took perhaps the most time out of this project. Unfortunately, the breadth of algorithms implemented makes it difficult to discuss them all fully in this report. Nevertheless, this section aims to provide an overview of their implementation in the library.

As discussed, the algorithm functions share similar signatures, where in the one-

```python
def memoization(budget: int, costs: List[int], values: List[int]) -> Tuple[List[int], int]:
    pass
def multi_memoization(budgets: List[int], costs: List[List[int]], values: List[int]) -> Tuple[List[int], int]:
    pass
```

Figure 16: A demonstration of the function signatures for the one and multi-dimensional algorithms.

```python
def fptas(budget: int, costs: List[int], values: List[int], accuracy: float = 0.5) -> Tuple[List[int], int]:
    pass
```

Figure 17: A demonstration of the function signatures that optionally accept additional parameters to tune and control the algorithm.

dimensional case they accept a single `budget`, and an array of `costs` and another of `values`, and in the multi-dimensional case they accept an array of `budgets` and a two-dimensional array of `costs` for each dimension and an array of `values`. Fig. 16 shows a one and multi-dimensional function signature for the memoization algorithm. The approximation algorithms that have their own parameters also *optionally* accept these parameters as input. Fig. 17 shows the function signature for the fully polynomial-time approximation scheme, where the accuracy parameter is defined to be 0.5 by default.

The return value or output of the algorithm functions is a tuple containing the allocation found as a list of the funded projects and the overall value or number of votes. The `PBSingleProblem` and `PBMultiProblem` objects then convert the funded projects to a list of project ids and compute the overall cost using this output before constructing a `PBResult` object and returning it to the user.

The algorithms are defined in their own files, e.g., `greedy.py` or `brute_force.py`, which often contain multiple functions that work together to perform the processes described in Sect. 3. The aim of this is to move towards the single-responsibility principle and promote cleaner code, rather than having one function which performs all of the various tasks involved in finding the allocation. For example, `genetic.py` contains functions for generating the population, selection, crossover and mutation. However, these files all contain one main function which runs the processes and returns an allocation as described, and the `solvers.exact` and `solvers.approximate` modules expose these functions such that the individual files do not need to be imported, and only the relevant functions are available.

Additionally, the one and multi-dimensional algorithms sometimes share very similar core processes, with only a small part of the algorithm varying where e.g., we must

```python
# Checks if an allocation exceeds the budget:
is_invalid: Callable[[Allocation], bool] = \
    lambda allocation: allocation.cost > budget

# Send to annealing function and return result:
return __simulated_annealing(
    costs,
    values,
    initial,
    is_invalid,
    initial_temperature,
    temperature_length,
    cooling_ratio,
    stopping_temperature
)
```

(a) The definition of the predicate for the one-dimensional algorithm and the submission to the base function.

```python
# Generate a neighbouring allocation and skip it if it
# is invalid:
neighbour: Allocation = \
    current.neighbour(values, costs)

if is_invalid(neighbour):
    continue
```

(b) The calling of the predicate from the base function to check whether the neighbour generated is invalid.

Figure 18: A code excerpt showing the submission of a locally-defined function to the base function to avoid code duplication in the simulated annealing algorithm.

check if an allocation exceeds any of the budgets. We mitigate this using higher-order functions by defining a base function which runs the process, and then submit the problem dependent task as a higher-order function to the base function. Fig. 18 provides an example of this in `simulated_annealing.py`, where the only difference between the one and multi-dimensional algorithms is the verification of generated neighbours. The individual functions pass a predicate to the base function to verify the neighbour, and the base function uses this to decide whether or not to skip it.

Overall, the individual implementations of the algorithms closely follow the descriptions provided in Sect. 3. As previously mentioned, the code for these can be found in the `community_knapsack/solvers` directory of the attached submission, and the functions can be imported directly through the library.

4.3.5 Documentation

The library and algorithms have been extensively documented using type hints, `docstrings` and comments to support potential users to create and solve problems and to enhance the readability and extensibility of the code. The type hints and `docstrings` apply to the exposed classes, methods and functions of the library such that integrated development environments and code editors provide relevant information and input suggestions and guidelines. For example, Fig. 19 provides a screen capture of the information provided when constructing a `PBSingleProblem` object.

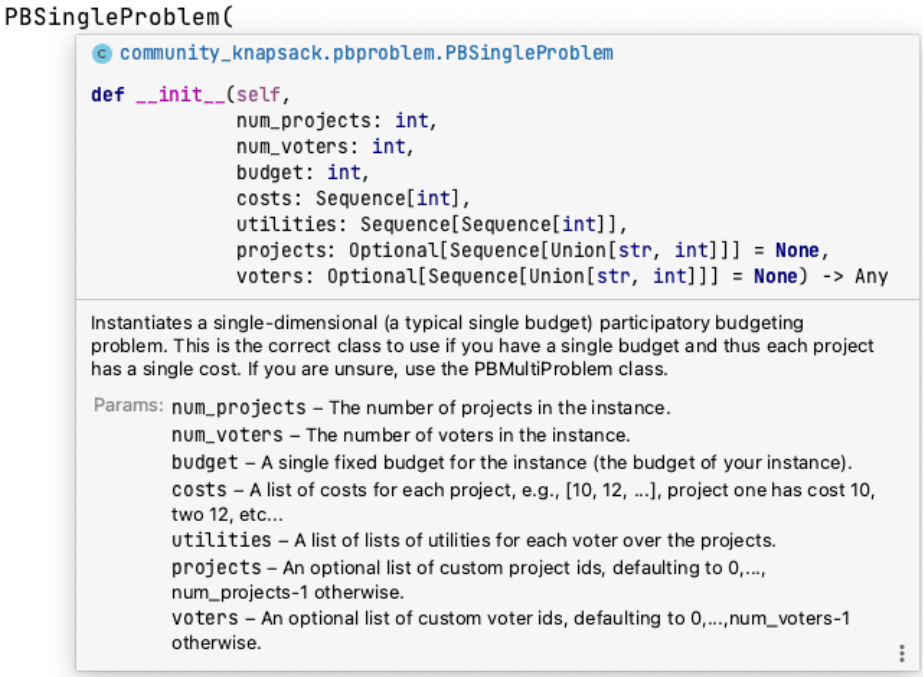

Figure 19: A screen capture showing the available documentation when hovering over the `PBSingleProblem` class in the PyCharm IDE.

The comments document the intuition and operation of the public and private processes in the system from the generator, parser and solver to the individual exact and approximation algorithms. These can be seen in the screen captures throughout this section, and in the source code in the attached submission. These comments do not appear to users who import the library, but enhance comprehension of the code and provide the foundations for extending the library and algorithms.

4.4 Unit Testing

The library has over one hundred unit tests written using the `pytest` package to test and verify the classes, methods, functions and algorithms. These ensure that the software is working correctly and as expected, and that no explicit or implicit errors occur when creating and solving problems. The full test implementation can be found in the `community_knapsack/tests` directory of the attached submission.

The individual algorithms are tested through a series of one and multi-dimensional binary knapsack problem data [4, 5], where the exact algorithms must match the optimal value exactly and the approximation algorithms must be within 30% of the optimal solution. Unfortunately, one limitation of these particular tests is that the instances are

all relatively small and thus do not provide full coverage. However, our evaluations in Sect. 5 show that the algorithms are indeed working as expected.

The `PBParser` class is tested by passing dummy and test files in the `.pb` data format to ensure that syntactical or value errors in the files result in the library throwing errors, that the parameters of the problem are parsed correctly, and that the voter preferences are converted to a list of utilities over the projects for each voter as expected. On the other hand, the `PBGenerator` class is tested by ensuring that errors are thrown when the user-specified parameter bounds are invalid, and that passing a seed to the generator results in the predictable output of problem objects.

The `PBProblem`, `PBSingleProblem` and `PBMultiProblem` classes are tested by ensuring that the data throws errors and warnings when the problem parameters are input incorrectly, e.g., when the number of cost dimensions does not match the number of budget dimensions, or when there are more costs than projects, and that the solving processes return the expected allocations for manually defined and computed instances similarly to the tests for the individual algorithm functions.

Overall, a large portion of these tests are dedicated to the guards which verify the input parameters of the constructors and methods of our classes. These ensure that warnings and errors are thrown for invalid inputs to prevent users from finding possibly bad allocations. This is particularly important in the parsing case, where it may be easy to make mistakes or cause errors when converting data into the `.pb` data format.

An important point to note here is that whilst the library does guard against the values of the inputs, it does not generally guard against their data types. There are several reasons for this decision, including the fact that Python is a dynamically typed language, the library provides type hints in the documentation, and that we assume users have at least basic understanding of the language. Furthermore, the library is still expected and appears to throw errors for nearly all erroneous data types where the values cannot be manipulated properly in e.g., reduction or solving.

4.5 Example Usage

The library can be used as per the activity diagram in Fig. 5, i.e., by importing it, creating a problem manually, or through parsing or random generation, and then selecting an algorithm to solve the problem and obtain an allocation. A basic demonstration of parsing a `.pb` file is provided in Fig. 20, whilst other examples can be found in the `community_knapsack/examples` directory of the attached submission.

The library is imported through the `import community_knapsack` syntax, or alter-

```
from community_knapsack import PBParser, PBSingleProblem, PBSingleAlgorithm
parser: PBParser = PBParser('../resources/pabulib/poland_warszawa_2017_aleksandrow.pb')
problem: PBSingleProblem = parser.single_problem()
print(problem.solve(PBSingleAlgorithm.BRANCH_AND_BOUND))
```

Figure 20: A code excerpt showing how the library is imported and used to parse a `.pb` file and obtain an allocation using branch-and-bound.

natively through the `from community_knapsack import ...` syntax. This provides access to all of the described classes, as well as the `solvers` module. The `PBParser` object is constructed with a path to a file in the `.pb` data format as described, and a one-dimensional problem is obtained from the parser. Finally, the problem is solved using the branch-and-bound algorithm and the output is printed to the console.

Fig. 21 provides a screen capture of the `PBResult` object output by the `solve` function. As earlier specified, this includes a list of the funded project ids, the overall value and cost of the problem, and the name, run-time and optimality of the utilised algorithm.

```
PBResult(allocation=['1206', '720', '2592'], value=303, cost=107950,
runtime=381.709792, algorithm='BRANCH_AND_BOUND', approximate=False)
```

Figure 21: A screen capture of the console output showing the returned `PBResult` object containing the allocation for the problem.

5 Evaluation & Results

This section covers the evaluation of the run-time performance and allocation efficacy of the exact and approximation algorithms for both randomly generated and real-world combinatorial participatory budgeting problems, and the evaluation of the library through the completion of the requirements, the suitability of the design and from the perspective of usability for our prospective users.

5.1 Generated Evaluation

This sub-section covers the one and multi-dimensional algorithm evaluations over randomly generated problems with varying parameters. These experiments were carried out in `.ipynb` notebook files which used our library to generate the problems and run the algorithms, and the `matplotlib` library to plot the results. These files can be found in the `community_knapsack/evaluations` directory of the attached submission.

The problems were created through the `PBGenerator` object constructed with a seed for reproducible output, although in any case the results of each experiment are stored in `.json` files where the experiments were very slow, often taking between five and thirty minutes. This was partially due to waiting for time limits where the algorithms were intractable for certain problems.

Finally, we note that the parameters of the `FPA`, `SIA` and `GEN` algorithms are set to their defaults as specified in Sect. 3 and repeated again here: the `FPA` accuracy parameter $\epsilon = 0.5$, the `SIA` parameters $T_{start} = 1.0$, $T_{length} = 50000$, $f = 0.9$ and $T_{stop} = 0.5$, and the `GEN` parameters $\sigma = 200$, $e = 100$, $P_{cross} = 0.8$ and $P_{mut} = 0.3$. These are not tuned in this evaluation, and tuning these parameters and possibly adapting the algorithms to scale with the size of the problem is an interesting piece of future work for this project.

5.1.1 One-Dimensional Algorithms

Our choice of problem parameter bounds for these experiments was based on the aforementioned existing real-world participatory budgeting instance data from [39], where it appeared that many problems had a fairly small number of projects and a relatively large budget. For example, many instances appear to have between forty to one hundred projects, with some having far fewer or far more, and budgets generally ranging from the hundreds of thousands and into the millions.

The first experiment involves observing the run-time and values of the algorithms as the number of projects n is increased from 5 to 250 with an increment of 5. The budget is fixed at $B = 100{,}000$, and the cost for each project is randomly generated between

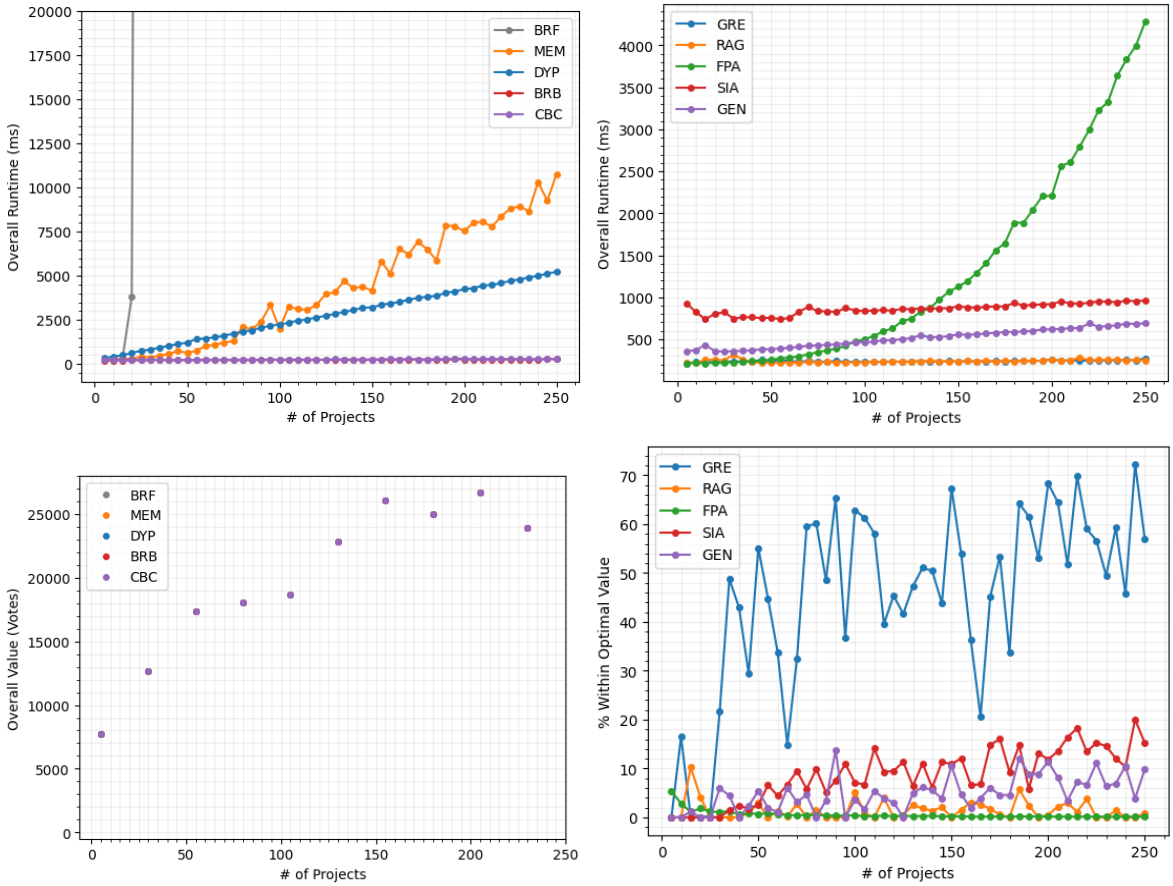

Figure 22: The run-times and values of the exact and approximate one-dimensional algorithms as the number of projects increases from 5 to 250.

5,000 and 50,000. A total of $m = 3{,}000$ voters are simulated who submit approval votes over the projects. The algorithms were run once on each generated problem with a timeout of 60 seconds and the results from the experiment can be seen in Fig. 22.

The top and bottom left plots visualise the run-times and values of the exact algorithms. We can see that all of the values produced by these are the same, indicating that they are all working correctly and as expected. The brute force algorithm is clearly the worst performing in regard to run-time where it grows exponentially in the number of projects, likely rendering it obsolete for most real-world problems. The run-times of the memoization and dynamic programming approaches scale linearly in the number of projects as we would expect, although the former appears to be less efficient. This is likely due to the aforementioned overhead with recursive calls. Finally, the run-times of the branch-and-bound algorithm and integer programming solver appear unaffected

by the number of projects and generally run very quickly implying suitability to `CPB`. However, we will see that the branch-and-bound algorithm is highly dependent on the magnitude of the costs relative to the budget, and very small relative costs can lead to intractable run-times.

The top and bottom right plots visualise the run-times and values of the approximation algorithms, where the values are represented by their percentage within the optimal solution, and thus lower is better. The greedy approaches are the fastest of the approximation algorithms and generally scale very efficiently with the number of projects. However, the widely adopted greedy algorithm is clearly arbitrarily bad with some solutions deviating over 70% from the optimal solution. On the other hand, the ratio greedy algorithm appears to produce much better results that are typically within $0-10\%$ of the optimal solution. This immediately suggests that ratio greedy successfully prevents the blocking faults of the typical greedy approach and thus that it is a much better choice for the one-dimensional problem.

The fully polynomial-time approximation scheme scales fairly sharply with the number of projects as we might expect given the cubic time complexity of the algorithm. However, it is much faster than the alternative dynamic programming approaches and finds very accurate solutions that are typically within $0-2\%$ of optimality in this experiment. This suggests it is a very good candidate for the one-dimensional problem, and perhaps even better than ratio greedy provided that the number of projects is not extremely large, which could render the algorithm intractable.

Finally, the simulated annealing and genetic algorithms appear to run very efficiently and produce reasonable approximations of the optimal solution, where in this experiment the former is within $0-20\%$ of optimality and the latter is within $0-10\%$. However, whilst we expect a linear trend in the run-time as the number of projects increase, the approximations computed by the algorithms also appear to scale away from optimality. This is likely due to their fixed parameters and the exponential number of *possible* solutions, where too few of these are generated and explored to successfully converge on the global optimum.

Our second experiment involves observing the run-time and values of the algorithms as the budget B is increased from 100,000 to 1,000,000 with an increment of 20,000. The number of projects is fixed at $n = 50$, and the cost of each project is randomly generated between 5,000 and 50,000. A total of $m = 3{,}000$ voters are simulated who submit approval votes over the projects. The algorithms were run once on each problem with a timeout of 120 seconds. The results from the experiment can be seen in Fig. 23.

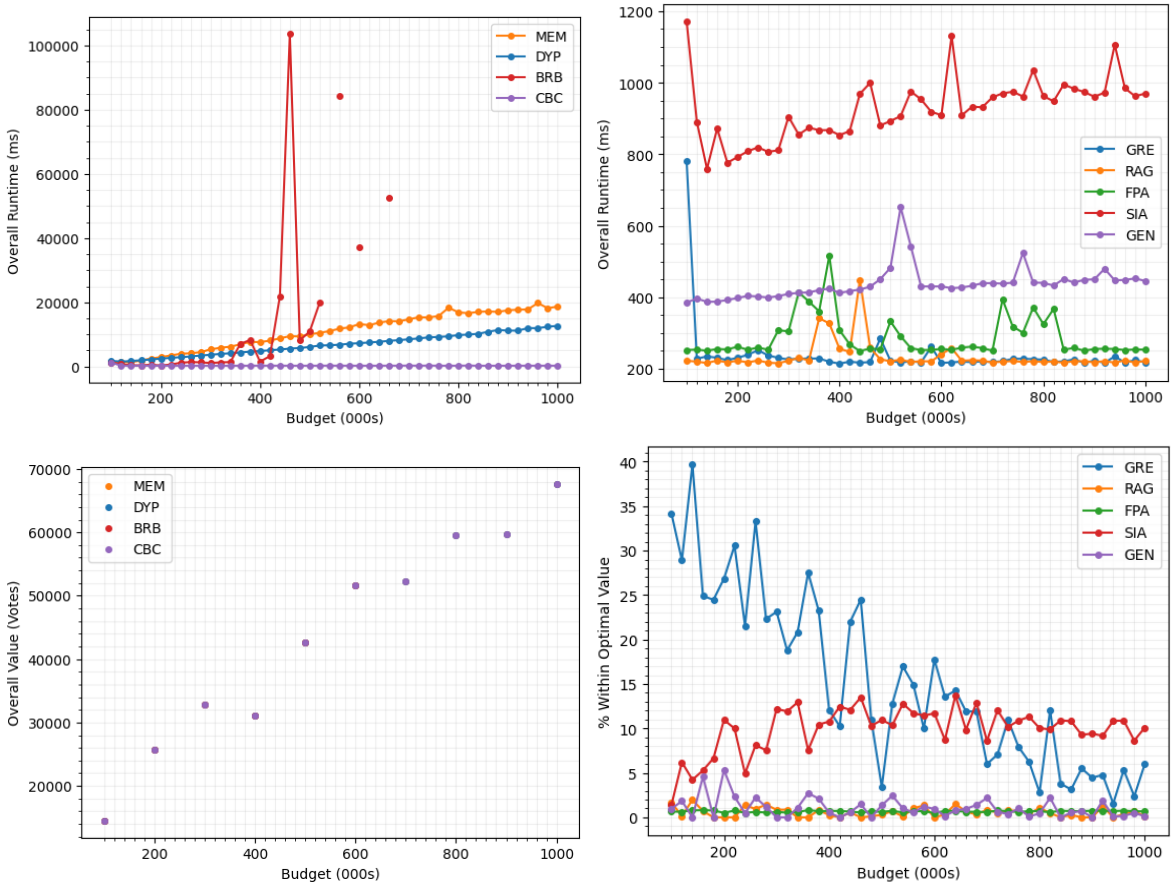

Figure 23: The run-times and values of the exact and approximate one-dimensional algorithms as the budget increases from 100,000 to 1,000,000.

The left-hand side shows the run-times and values of the exact algorithms, excluding brute force which did not finish within the time limit for any of the problems. The memoization and dynamic programming algorithms performed very similarly to our previous experiment and scale linearly as the budget increases. However, the branch-and-bound algorithm exhibits a dramatic increase in run-time as the budget increases, and was generally unable to finish within the time limit as the budget exceeded 600,000. As seen in the next experiment, this is due to the magnitude of the costs relative to the budget, rather than just the budget alone where this ratio affects the ability of the algorithm to fathom nodes early in the process. The benefits of the advanced optimisation techniques employed by the integer programming solver become apparent here where it exhibits no change in run-time as the budget increases, thus suggesting it is not affected by the budget or cost-to-budget ratio.

The right-hand side shows the run-times and values of the approximation algorithms, where the values are represented by their percentage within the optimal solution. The run-times of our two greedy approaches appear unaffected by the magnitude of the budget and run very quickly in this experiment. However, whilst the ratio greedy algorithm still appears to find solutions that are within $0 - 10\%$ of the optimal solution, the typical greedy approach interestingly exhibits improvement as the budget increases. A possible explanation for this phenomenon is that as the costs become smaller relative to the budget, fewer blocks can occur because more projects can fit in the allocation, and thus it is easier to find a good solution.

The run-time and accuracy of the fully polynomial-time approximation scheme and the genetic algorithm appear to be unaffected by the magnitude of the budget. These both appear to be finding very good solutions within $0 - 5\%$ of the optimal solution as seen when $n = 50$ in the previous experiment. This suggests that these algorithms are good options for real-world problems when the budget is high.

The run-time of the simulated annealing algorithm appears to exhibit a slight linear trend as the budget increases, and the accuracy appears to decrease at the beginning of the experiment before flattening off to be within $10 - 15\%$ of the optimal solution. The former may be due again to the cost-to-budget ratio, perhaps where more solutions are valid as the costs become lower relative to the budget, and so fewer are discarded during the annealing process and thus the algorithm explores more solutions overall. This may also explain the latter phenomenon, where there are many more *valid* solutions that the algorithm cannot explore due to the nature of the fixed parameters, and thus it cannot converge on the global optimum.

Our third and final experiment for the one-dimensional algorithms involves observing their run-time and values as the magnitude of the costs change relative to the budget. More specifically, we increase the minimum cost bound from 5,000 to 25,000 with an increment of 1,000 where the maximum cost bound is fixed at 50,000. The number of projects and budget are fixed at $n = 50$ and $B = 500{,}000$, and a total of $m = 3{,}000$ voters are simulated who submit approval votes over the projects. The algorithms were run once on each problem with a timeout of 120 seconds. The results from the experiment can be seen in Fig. 24.

The left-hand side shows the run-times and values of the exact algorithms excluding again brute force where it did not finish within the time limit. The memoization algorithm exhibits a decreasing run-time as the minimum cost increases, presumably where more pruning is able to take place, and further evaluations not shown in this report revealed that once the costs become very high relative to the budget it performs

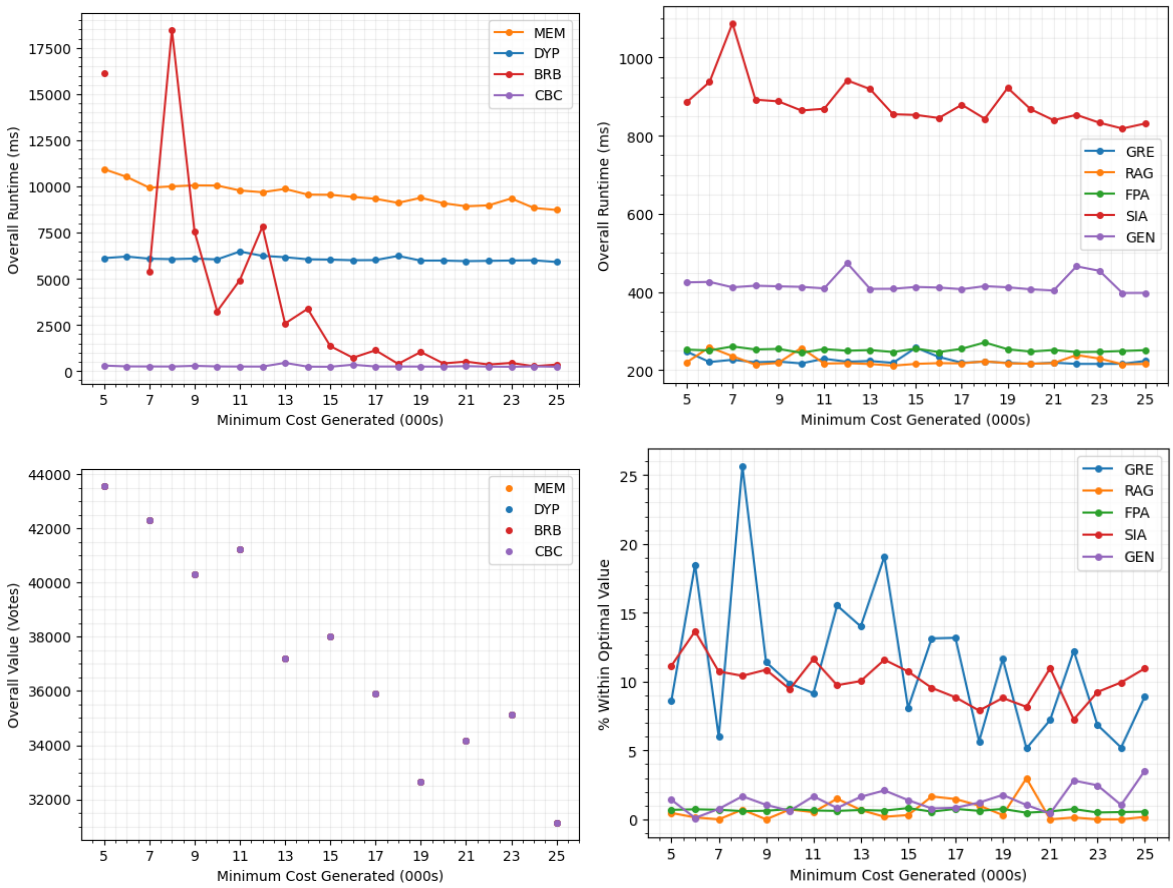

Figure 24: The run-times and values of the exact and approximate one-dimensional algorithms as the minimum cost generated increases from 5,000 to 25,000.

much more efficiently than the dynamic programming algorithm, which does not appear to be affected by the costs. The run-time of the branch-and-bound algorithm decreases rapidly as the minimum cost increases, thus proving that it is highly dependent on the cost-to-budget ratio in each problem. This suggests that it is generally only useful in real-world scenarios where the costs are high relative to the budget, where otherwise it is intractable and unlikely to finish in a reasonable time frame. Lastly, the integer programming solver is unaffected by these costs and appears to be the best choice for all the problems explored in these evaluations.

The right-hand side shows the run-times and values of the approximation algorithms. The run-times of the greedy algorithms and the accuracy of the ratio greedy algorithm appear unaffected by the increasing costs, whereas the accuracy of the typical greedy approach appears to be increasing with the minimum cost generated. This is slightly

contradictory to our previous result, in which we said that smaller costs lead to a higher accuracy. However, further analysis reveals that this is likely due to the difference between the minimum and maximum bounds decreasing where fewer blocks can occur. This is another special case of the parameters in which the typical greedy approach turns out to be fairly accurate.

Similarly to the previous experiment, the fully polynomial-time approximation scheme and the genetic algorithm appear fairly unaffected by the magnitude of the costs or the cost-to-budget ratio. As speculated, the simulated annealing algorithm does indeed appear to be less effective in both run-time and accuracy when the magnitude of the costs is low relative to the budget, and we can see that these improve as the minimum cost generated increases. However, these changes in performance are not hugely significant and thus this algorithm is not necessarily unsuitable in this setting.

Overall, the experiments for the one-dimensional algorithms have shown that the integer programming solver is the most effective method to maximise the utilitarian welfare under one-dimensional `CPB`. The solver did not exhibit any changes in performance as the parameters scaled, and ran very quickly on every problem. We previously noted that we expected it to be more effective than our exact algorithms, especially because it is written in `C++`, although we did not expect it to scale so effectively or necessarily undercut the run-times of the approximation schemes.

The other exact algorithms still appear to be viable, and may exhibit better performance when written in a faster language such as `C++`. The memoization and dynamic programming approaches clearly both scale linearly in the number of projects and budget, and therefore we generally expect fairly significant run-times for large problems with hundreds of projects and budgets in the millions. The memoization and branch-and-bound algorithms proved to be very effective when the costs of the projects are high relative to the budget, although branch-and-bound was intractable for low relative costs rendering this unsuitable for general budget allocation where this ratio is unknown.

Our developed approximation algorithms appeared to perform well across the problems, and generally ran faster overall than *our* exact algorithms. As expected, the greedy approach was shown to be poor in practice with regard to optimality, generally producing the worst result for each problem. The ratio greedy and fully polynomial-time approximation scheme appeared to yield the most accurate allocations, with the latter perhaps performing slightly better at the cost of run-time where it scales significantly as the number of projects increase. The genetic algorithm also performed well, although it was generally the second slowest algorithm and produced less accurate results as the number of projects increased. Finally, the simulated annealing algorithm was slightly dis-

appointing, often taking the longest time to complete and generally producing solutions that were fairly distant from the optimal solution.

5.1.2 Multi-Dimensional Algorithms

Our choice of problem parameter bounds for these experiments are based on the existing real-world data from [39], although are slightly more arbitrary where no multi-dimensional instances are available. We make the assumption then that in realistic instances there would only be a small number of possible dimensions, e.g., no more than ten, and that the magnitude of each budget may vary where it may not involve money.

The first experiment involves observing the run-times and values of the exact and approximation algorithms as the number of dimensions d is increased from 1 to 10. The budget B_j in each dimension is randomly generated between 1,000 and 100,000, and the costs in each dimension are randomly generated between 5% and 40% of the budget. The number of projects is fixed at $n = 50$, and there are a total of $m = 3000$ voters who submit approval votes over the projects. The algorithms were run once on each problem with a timeout of 120 seconds and the results can be seen in Fig. 25.

The left-hand side shows the run-times and values of the exact algorithms, excluding brute force where it did not finish within the time limit. The dynamic programming algorithm appears to be very ineffective for multi-dimensional problems, and did not finish within the time limit for any $d > 1$, showing that generating all of the possible sub-problems was unfortunately an ineffective strategy. Surprisingly, our memoization algorithm appears to perform extremely well given multiple dimensions, and even improves as the number of dimensions increase, possibly where the costs are high relative to the budget, thus enabling the algorithm can prune branches immediately if even one of the budgets is exceeded in any dimension. The integer programming solver again appears very effective and exhibits no change as the number of dimensions increases.

The right-hand side shows the run-times and values of the approximation algorithms. The greedy approaches both appear unaffected by the number of dimensions and run very quickly across all the problems. However, they now both exhibit arbitrarily bad allocation behaviour suggesting that our adaptation of the ratio greedy algorithm is unfortunately ineffective for multi-dimensional problems. However, our approximate branch-and-bound (`ABR`) scheme which harnesses ratio greedy generally appears to have a higher accuracy with solutions between $0 - 10\%$, runs very quickly and appears unaffected by the number of dimensions, although similarly to `BRB` this is likely to be affected by the magnitude of the costs relative to the budget.

The run-times of the simulated annealing and genetic algorithms both appear to

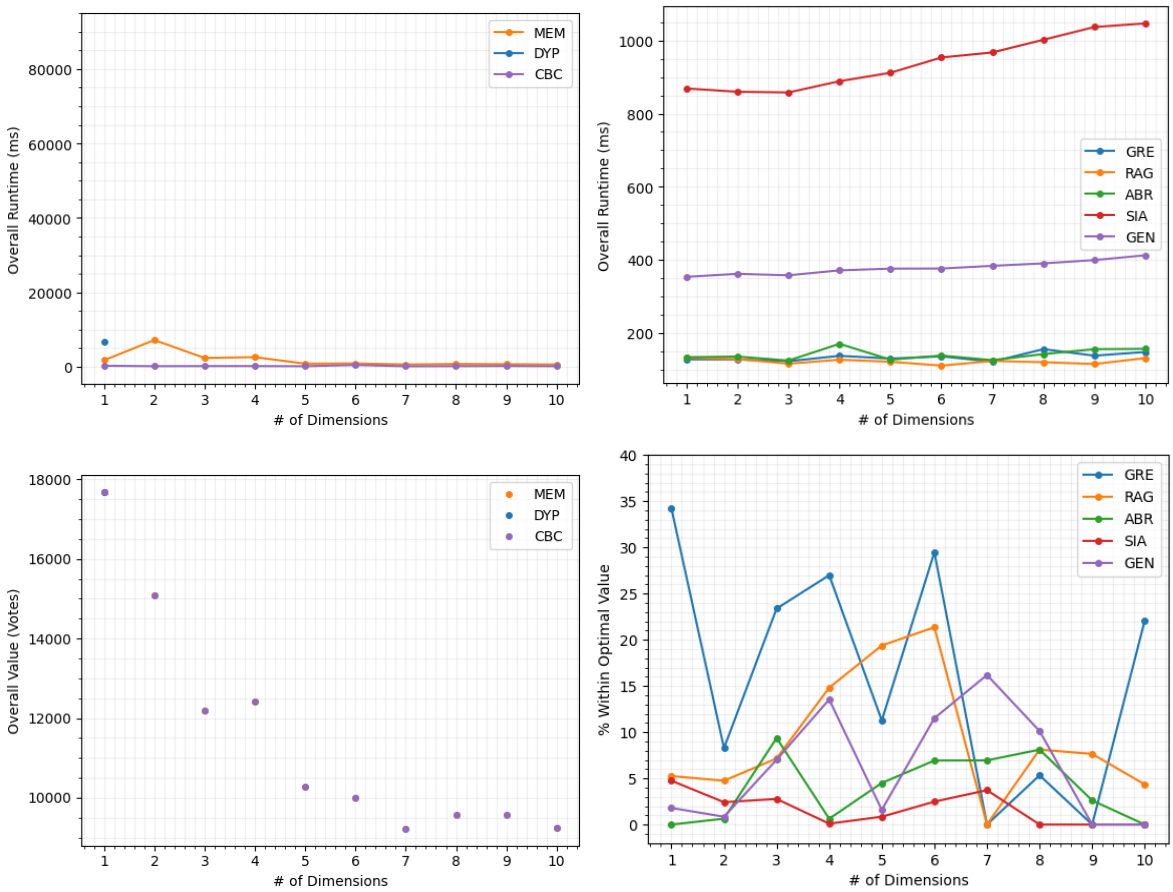

Figure 25: The run-times and values of the exact and approximate multi-dimensional algorithms as the number of dimensions increases from one to ten.

increase slightly as the number of dimensions increase, and this is likely due simply to the additional work involved with computing the cost of each allocation in d dimensions. The former algorithm appears to produce the best results overall, whilst the latter has much more variation in accuracy. This may be because more of the possible allocations become invalid as the number of dimensions increase, and the simulated annealing algorithm simply discards or skips invalid allocations, whilst the genetic algorithm gives them a fitness of zero and keeps them in the population.

Our next experiment involves identifying the dependence of our exact and approximate multi-dimensional algorithms on the cost-to-budget ratio by observing their run-times and values as the maximum cost generated increases from 10 to 50% of the budget in each dimension. The budget B_j in each dimension is randomly generated between 1,000 and 100,000, and the minimum cost generated in each dimension is 1% of the budget.

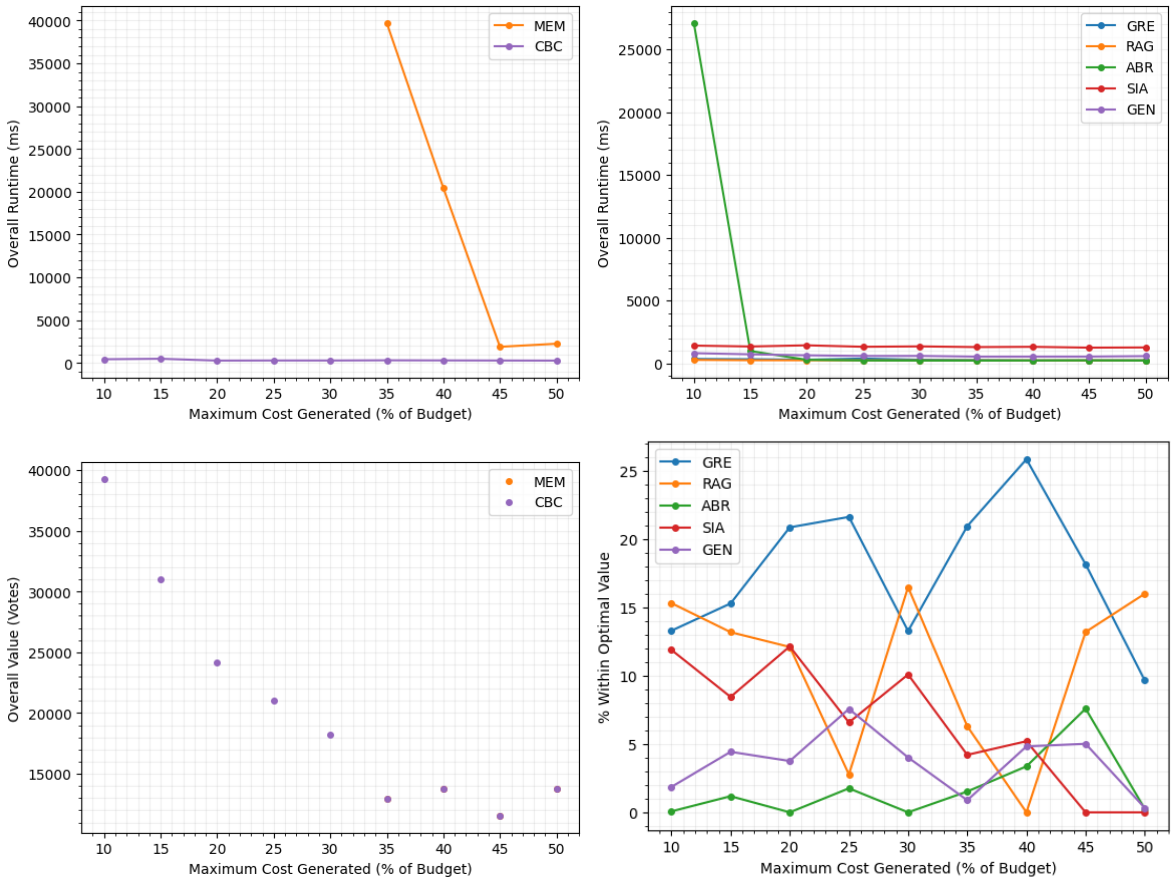

Figure 26: The run-times and values of the exact and approximate multi-dimensional algorithms as the maximum cost generated increases from 10% to 50% of the budget in each dimension.

The number of projects is fixed at $n = 50$, and there are a total of $m = 3000$ voters who submit approval votes over the projects. The algorithms were run once on each problem with a timeout of 60 seconds and the results can be seen in Fig. 26.

The left-hand side shows the run-times and values of the exact algorithms, excluding brute force and dynamic programming where they did not finish within the time limit. The memoization algorithm is clearly intractable then when the magnitude of the costs are low relative to the budget in *all* dimensions. However, if even *one* of the dimensions have high relative costs, then as discussed the search space can still be pruned effectively and the algorithm should run very quickly. As before, the integer programming solver appears unaffected by the magnitude of the costs or the cost-to-budget ratio.

The right-hand side shows the run-times and values of the approximation algorithms.

The results are fairly similar to those discussed in the previous experiment and in the one-dimensional evaluations. For example, we can see that the simulated annealing algorithm converges on the global optimum more effectively when the costs increase relative to the budgets. The most significant change here is that our approximate branch-and-bound approach scales similarly to our exact branch-and-bound algorithm in the one-dimensional case, where it becomes intractable for very small relative costs. However, similarly to memoization, if even one of the dimensions has high relative costs then the algorithm should be able to fathom nodes effectively and produce allocations in reasonable time.

Our final experiment in the multi-dimensional case evaluates the run-times and values of the exact and approximate algorithms as the number of projects increases from 5 to 250 when the problem has multiple dimensions. The number of dimensions is fixed at $d = 3$, the budget B_j in each dimension is randomly generated between 1,000 and 100,000, and the costs in each dimension are randomly generated between 5% and 40% of the budget. There are a total of $m = 3000$ voters simulated who submit approval votes over the projects. The algorithms were run once on each problem with a timeout of 60 seconds and the results can be seen in Fig. 27.

The left-hand side shows the run-times and values of the exact algorithms, excluding brute force and dynamic programming where they did not finish within the time limit. The run-time of the memoization algorithm grows rapidly as the number of projects increase in multiple dimensions due to the number of possible sub-problems increasing exponentially. Unfortunately, it did not finish within the time limit when $n > 115$ in this experiment, perhaps suggesting that it is not very effective for larger problems unless at least dimension has very high relative costs. Lastly, the integer programming solver was once again unaffected and ran very quickly for all of the problems.

The right-hand side shows the run-times and values of the approximation algorithms. Similarly to our previous results, the greedy and ratio greedy approaches did not appear to slow down as the number of projects increased, although both appeared to be arbitrarily bad in their solutions. The approximate branch-and-bound algorithm also appeared to perform well as the number of projects increased where it is generally more dependent on the cost-to-budget ratio, and whilst it is more accurate than the ratio greedy approach, it still appears to produce worse allocations as the projects increase suggesting this bounding method is relatively ineffective in practice.

Finally, both the run-time and the accuracy of the simulated annealing and genetic algorithms appear to worsen with the number of projects due to the additional work required to generate and validate neighbouring allocations or offspring, and the fixed

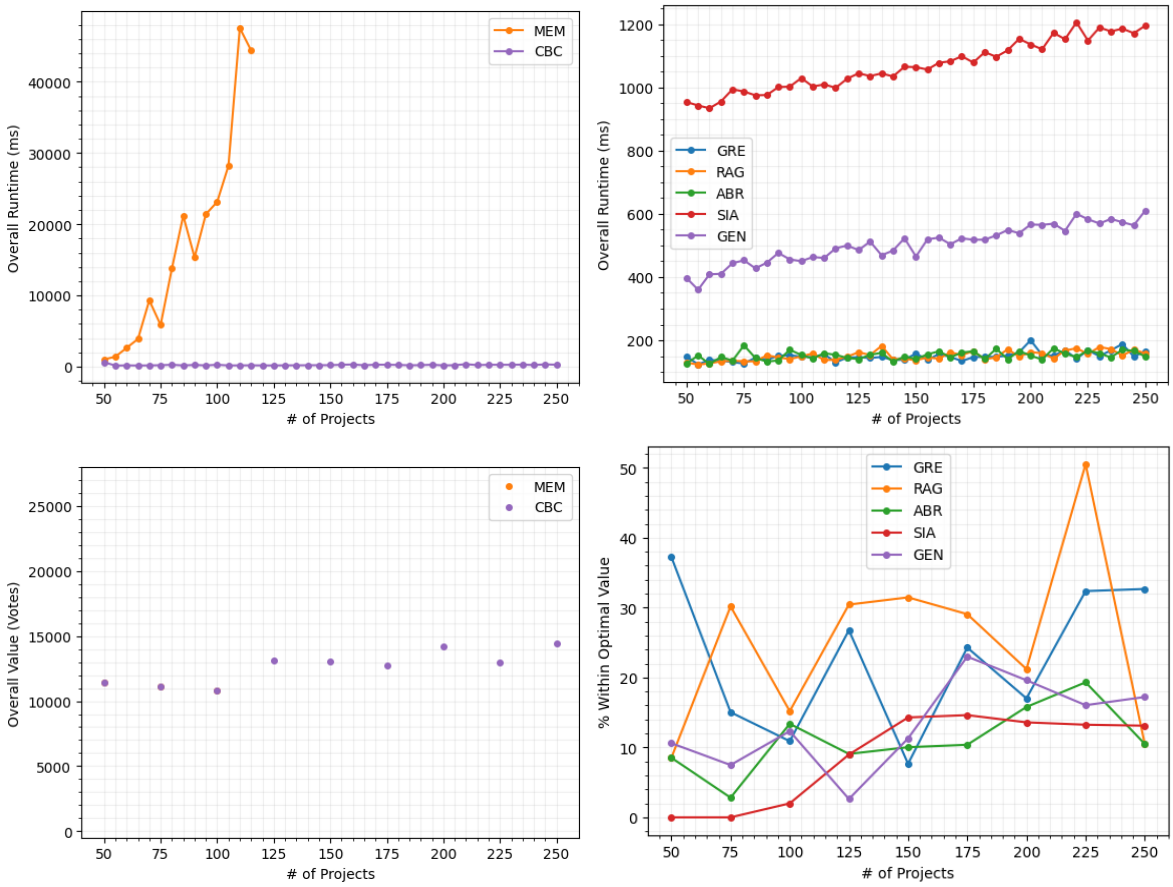

Figure 27: The run-times and values of the exact and approximate multi-dimensional algorithms as the number of projects increases from 5 to 250.

parameters preventing the algorithms from exploring enough solutions to successfully converge on the global optimum. However, these algorithms do appear to produce reasonable approximations of the optimal solution fairly quickly, and the growth of their run-times is not hugely significant suggesting suitability to real-world budget allocation.

Overall, the integer programming solver is clearly the dominant algorithm for the one and multi-dimensional problems, seemingly even amongst the approximation algorithms, and is therefore the best choice for democracies to maximise the utilitarian welfare under `CPB`. The only other viable exact algorithm in the multi-dimensional case is memoization, which appears to perform well when at least one dimension has high costs relative to the budget, or when the number of projects is very small, but otherwise may exhibit an exponential run-time.

The approximation algorithms are slightly disappointing in the multi-dimensional case, with the typical greedy approach sometimes outperforming our ratio greedy algorithm, and all the algorithms becoming less effective as the number of projects increase. Our approximate branch-and-bound algorithm is likely the best choice when the costs are high relative to the budget, or otherwise simulated annealing or our genetic algorithm can be used to find seemingly reasonable approximations within a number of seconds.

5.2 Real-World Evaluation

The real-world evaluation for this project involves investigating the run-time performance and allocation efficacy of our exact and approximate one-dimensional algorithms over previous real-world problems. These have been obtained from Pabulib [39], a participatory budgeting library storing data from instances around Europe in the `.pb` data format. Table 1 shows the six problems we have selected with varying numbers of projects and sizes of budget, and incorporates problems that use approval, cumulative and ordinal voting. Unfortunately, the library does not appear to store any scoring voting instances, perhaps suggesting that it is uncommon in practice.

ID	Name	Voting Type	# Projects	Budget
PT19	poland_warszawa_2019_targowek-fabryczny-elsnerow-i-utrata.pb	Approval	17	400,000
FR19	france_toulouse_2019_.pb	Cumulative	30	1,000,000
PW21	poland_warszawa_2021_wlochy.pb	Approval	49	1,412,893
PO22	poland_krakow_2019_nowa-huta.pb	Ordinal	63	1,523,200
PU23	poland_warszawa_2023_ursus.pb	Approval	72	2,427,140
PT20	poland_warszawa_2020_targowek.pb	Approval	88	4,072,457

Table 1: A table showing the id, name, voting type, number of projects and budget of the Pabulib [39] instances to be evaluated.

Our exact and approximate one-dimensional algorithms were run once for each of these problems with a sixty second time limit. The run-times and values of these can be seen in Fig. 28, where the top two charts display the run-times of the exact and approximation algorithms respectively, whilst the bottom two charts display the values of the approximation algorithms.

As we would expect, the brute force algorithm was only able to solve the first problem with $n = 17$ within the time limit, the memoization and dynamic programming approaches scale polynomially with the number of projects and the magnitude of the budget, and the integer programming solver was able to solve all the problems very

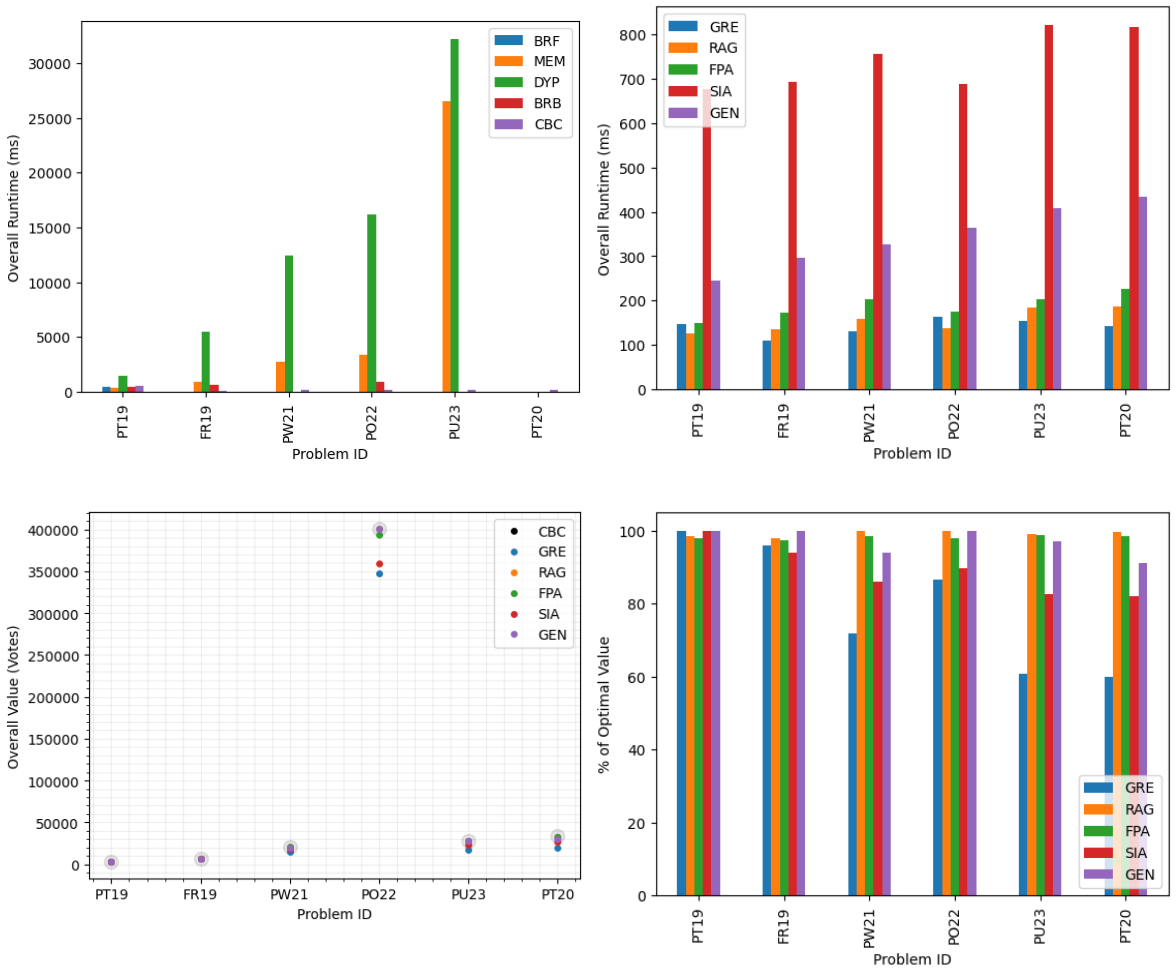

Figure 28: The run-times and values of the algorithms over a set of previous real-world from Pabulib [39] within a sixty second time limit.

quickly. The memoization algorithm appears to be outperforming dynamic programming which would suggest high relative costs enabling effective pruning of the search space. However, the branch-and-bound algorithm does not finish within the time limit for some of these instances, which contradicts our results from our previous evaluations.

Further testing showed that the branch-and-bound algorithm requires *all* or *most* of the costs to be high relative to the budget, and if even a small number of projects have low costs then the algorithm becomes intractable. The projects with small costs require further generation and expansion of nodes to begin fathoming nodes, where the bounds are higher than the best value found so far for a longer portion of the process, thus increasing the overall run-time significantly.

Our approximation algorithms appear to perform similarly to our previous evaluations. They all finish running within one second, and for the most part provide very good approximations of the optimal solution with the exception of the greedy algorithm and arguably simulated annealing as the problem sizes increase. As expected, the ratio greedy and fully polynomial-time approximation scheme are the most effective, with the former showing dominance over the latter through slightly better run-times and allocations. The genetic algorithm is not far behind, and appears to produce good approximations in reasonable time.

Overall, all of our one-dimensional algorithms appear to be better than the widely adopted greedy approach providing that they are tractable and finish in a reasonable time frame, and thus they are all viable alternatives to improve participatory democracy. Unfortunately, there are seemingly no real-world multi-dimensional `CPB` data sets available, but we would assume that our multi-dimensional algorithms would exhibit a similar result, perhaps with the exception of our adapted ratio greedy approach.

5.3 Library Evaluation

The library appears to meet the requirements and aims to a relatively high standard. The interface was simple and easy to use for both the randomly generated and real-world problems during the evaluation, and we expect that it will work well for other users, including budget decision-makers, researchers and students provided they have some basic programming knowledge. However, there are inevitably issues and possible problems with any software, and this section aims to discuss and address limitations identified with our library during the implementation, testing and evaluation.

Generally, the library is very simple to use, possibly only requiring four of five lines of code when parsing a `.pb` file as shown in Sect. 4.5. This is excellent from a usability perspective and satisfies one of our requirements, but sacrifices the possibility of e.g., modifying the parsing or generation process, or tuning the algorithms without changing the library itself. The latter example is particularly problematic where the approximation algorithms that have their own specific parameters, such as simulated annealing, can typically be tuned to obtain better solutions or reduce the run-time.

An issue that may impact usability is the requirement to *select* an algorithm from the `PBSingleAlgorithm` or `PBMultiAlgorithm` enumerator classes. More specifically, the names of the algorithms are not particularly intuitive and do not indicate their possible run-time, optimality or their suitability to any specific problem parameters. This has been mitigated through extensive documentation for each of the fields in the enumerator, and in the algorithm functions themselves, however one possible improvement could be

to have the problem classes automatically choose an algorithm based on the parameters and the results from our evaluation.

We recognise that the `PBGenerator` class, although unlikely to be used in practice, is fairly limited and may not accurately represent real-world scenarios. This issue is naturally dependent on the aims of the actor when using the library, e.g., it is unlikely to cause issues for general testing or experimentation, but will be unrealistic if trying to simulate real projects or voters and their thoughts and decisions. Of course, the library was not implemented with this goal, but it is nevertheless important to consider the possible implications of this class in practice, and thus the documentation clearly states that the generator does not produce realistic problems.

The algorithm evaluation has revealed that the integer programming solver is much more effective than our algorithms, and this may be partly due to the choice of Python for our programming language. As discussed, our algorithms would be much faster if written in `C++` [3], and this may have been an oversight given that speed is of the essence in optimisation. One important piece of future work then is re-writing these algorithms in a faster language, calling them from the Python library and then re-evaluating their performance. Hopefully, we would observe that all of our algorithms are able to solve the problems in our evaluations in much shorter time frames.

One specific implementation issue that was identified during the evaluation is that the timeout process increases the run-times of the algorithms. More specifically, the entire problem object is sent to the new thread in practice, and thus the run-time becomes linear in the size of the problem. Fig. 29 demonstrates this behaviour as the number of voters and thus the number of utilities stored by the object increases. Luckily, this does not particularly affect our evaluation where the run-times of *all* the algorithms scale identically because they are all processing the same problems. However, our algorithms may be slower overall with a very large number of voters.

Finally, although all of our unit tests pass implying that the library is working correctly, there is some concern towards the coverage of these tests and the possible existence of bugs in the software. For example, whilst the library generally performed very well during the evaluations, one problem identified was that occasionally the processes would not terminate after the specified time limit. This seemed to specifically be a problem with the multi-dimensional dynamic programming algorithm, perhaps indicating that the generation of all possible sub-problems could not be halted properly. Additionally, the algorithms are not tested on a very wide range of problems, and although the above evaluation indicates that they work well, there may be issues with e.g., very large problems, or problems with untypical characteristics. There could also

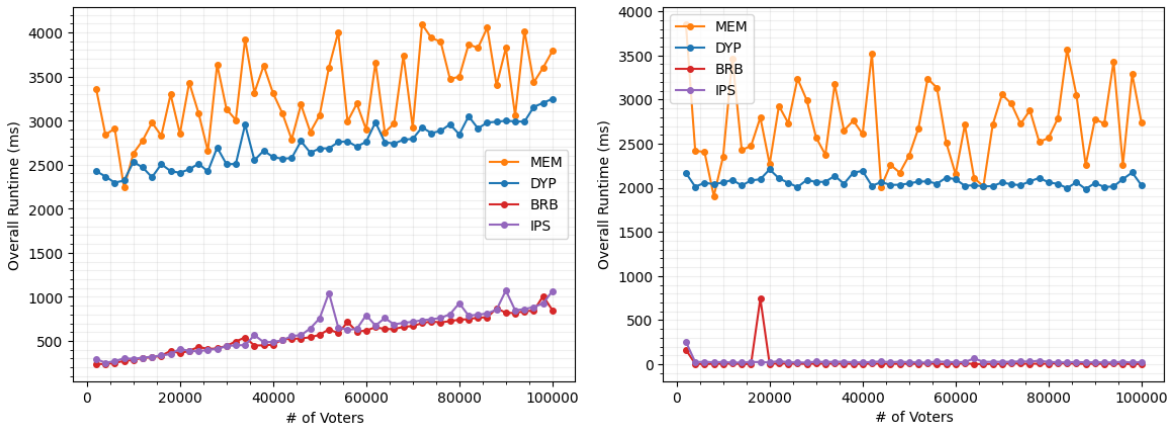

Figure 29: A comparison of the run-times of the exact algorithms as the number of voters increases when using the library (left) and when using the algorithm functions directly (right).

be cases where the library accepts invalid inputs, such as an invalid `.pb` file, and does not throw an error immediately or worse yet may return an invalid allocation without warning the user that a problem has occurred.

5.4 Results & Summary

Our evaluation has investigated the run-time and allocation efficacy of our exact and approximation algorithms and the effectiveness of our library in delivering a platform to run these algorithms. We found that the integer programming solver is the most effective algorithm to maximise the utilitarian welfare under one and multi-dimensional combinatorial participatory budgeting, where it is extremely fast and always returns the optimal allocation. This is the approach that we recommend to replace the widely adopted greedy algorithm to improve budget allocation and participatory democracy.

Unfortunately, this was not an algorithm that we designed or implemented ourselves, and we found that our own approaches were less effective in comparison. The memoization algorithm appeared to be the second best exact approach in the one and multi-dimensional case, generally finding solutions to small and medium-sized problems and scaling fairly predictably in the number of projects, dimensions and budget. The dynamic programming algorithm was similarly effective in the one-dimensional case, although was unable to solve any multi-dimensional problems. The branch-and-bound algorithm is very effective when all or most of the project costs are high relative to the budget, but is less predictably performant and is therefore difficult to recommend in the general case. Lastly, the brute force algorithm was ineffective for nearly all problems

and should not be used in practice.

The approximation algorithms appeared to perform well in the one-dimensional case, with the ratio greedy algorithm and fully polynomial-time approximation scheme producing the most accurate results and generally finishing within a number of seconds. However, the ratio greedy algorithm lost dominance in the multi-dimensional case and sometimes produced worse results than the typical greedy approach. The simulated annealing and genetic algorithms performed *reasonably* in both cases, and although they were slower overall than the other algorithms they still generally produced better solutions than the greedy approach in a number of seconds. Lastly, the approximate branch-and-bound scheme appeared to give good results and was very fast when all or most of the project costs are high relative to the budget in at least one dimension, but this is again less predictable and therefore difficult to recommend.

An important note here is that these algorithm evaluations are not completely conclusive. Further testing and experimentation, as well as rewriting the algorithms in a faster language, will help us to better understand and compare the specific use cases and problem characteristics that each algorithm is suited towards. More specifically, we did not explore all the different possible combinations or magnitudes of the problem parameters, or special cases of these such as e.g., single-peaked preferences, equal project costs, etc. Additionally, we did not identify when the integer programming solver begins to become less effective, which would help us to understand when our approximation algorithms might become necessary. However, we do suspect that is going to be much higher than any reasonably sized `CPB` problem, and we are fairly confident that the integer programming solver is the most effective strategy.

Finally, we found that our library was overall very effective when generating, parsing and solving problems using the algorithms, and that it outputs errors and warnings as expected when provided erroneous input. We discussed a number of limitations identified during the implementation, testing and algorithm evaluation phases, including possible usability issues for our target audience, implementation issues and concern towards the coverage of the unit tests. However, we are generally confident that the library works as expected and can be harnessed by e.g., budget decision-makers, researchers or students to solve their own combinatorial participatory budgeting instances.

6 Conclusion

6.1 Report Summary

To summarise, this report has introduced background on three important knapsack problems, namely BKP, MBKP and FKP, explored participatory budgeting and approaches to preference elicitation, vote aggregation and budget allocation in CPB, and introduced methods to reduce CPB to BKP and MBKP. We have investigated and designed several exact and approximation algorithms for these problems, namely dynamic programming, branch-and-bound and greedy approaches, a simulated annealing and genetic algorithm approach, as well as an integer linear programming solver, and implemented these in a budget allocation library. We evaluated these algorithms to identify their accuracy and feasibility, and compared them against the widely adopted greedy approach. And lastly, we evaluated our library to identify its effectiveness in providing a platform for budget allocation to improve participatory democracy.

As per our original aims, we are confident that we have identified more accurate, feasible alternatives to the widely adopted greedy approach, with the integer programming solver being the most effective of these overall. This algorithm appears to obtain an optimal solution in seconds for small, medium and large CPB problems, and does not appear affected by specific characteristics of the parameters. Generally speaking, this result was fairly unexpected and was perhaps an oversight at the beginning of the project, where it was assumed that no exact algorithm would be able to solve very large instances in reasonable time, and that we would therefore be reliant on approximation algorithms. In any case, most of our developed approaches appear to produce better solutions than the greedy algorithm provided their run-times are not intractable, and hence most of them would be better suited to maximising the utilitarian welfare under combinatorial participatory budgeting.

Finally, based on our experience and findings during the evaluation we are also confident in the usability and effectiveness of our budget allocation library for more general audiences, and we hope that our library may be useful for researchers and students in the fields of e.g., combinatorial optimisation or computational social choice, who may be able to use the library not only to solve one and multi-dimensional CPB problems, but also more general BKP and MBKP instances using existing well-defined algorithms and our novel approaches, such as our simulated annealing and genetic algorithms, or our generalisations of memoization and branch-and-bound to the multi-dimensional case.

6.2 Initial Plan Differences

The initial plan for this project was slightly different to the end result produced and described in this report. Firstly, whilst we are still focused on participatory budgeting (specifically *combinatorial* participatory budgeting), we have implemented a Python library rather than a web application as previously planned. The latter may have been more suitable for e.g., budget decision-makers to solve problems, however would have ultimately been a more time-consuming process that may have affected the breadth of algorithms explored and our ability to evaluate our results as effectively. Furthermore, the library can now be used as a framework or platform from which web or desktop applications can be built in future work.

Secondly, we have designed and implemented more algorithms than initially expected, such as memoization, the integer programming solver, and a fully polynomial-time approximation scheme, and furthermore adapted these algorithms where possible to the multi-dimensional problem. The latter is particularly interesting where there are seemingly few algorithms in the literature or online for this problem, and this enabled us to generalise our library and provide results for a wider breadth of `CPB` problems.

Overall, as per the initial plan we designed and implemented exact and approximation algorithms for combinatorial participatory budgeting, and performed a critical evaluation into the algorithms and compared them against the widely adopted greedy approach.

6.3 Future Work

There are many possible aspects of future work in this project, including improvements and extensions to our library, algorithms and our results that can further the field of budget allocation in participatory democracy. This section suggests and describes future work proposals discovered throughout the course of this twelve-week project:

6.3.1 Project Improvements

- *Rewriting The Algorithms:* As discussed, the optimisation algorithms are written in Python, which is generally considered a slow, high-level language [3]. The speed of optimisation algorithms is naturally very important, and thus rewriting them in e.g., `C` or `C++` and calling them from the library would likely result in much faster run-times and a fairer evaluation against the integer programming solver.
- *Improving The Algorithms:* The algorithms generally performed better overall than the greedy algorithm. However, these can likely be improved further. For example, using a different method to generate the sub-problems in multi-dimensional dynamic

programming could lead to tractable run-times, or re-designing the fully polynomial-time approximation scheme, simulated annealing and genetic algorithms such that their specific parameters scale with the size of the problem could lead to better general performance that does not decrease with the size of the problem.

- *Further Algorithm Evaluations:* The algorithm evaluations in this report are not completely conclusive, and there are many more experiments that can be run to better understand their performance given certain problem characteristics. For example, this might include additional project cost distributions, the impact of the voting method used, their performance over very large instances or the tuning of the parameters of the approximation algorithms. This may help us to understand e.g., when the integer programming solver becomes less effective and thus when approximation algorithms are required.
- *Better Problem Generation:* The generator class and the one and multi-dimensional problems generated are unlikely to be reflective of real-world problems. This class could be improved to simulate e.g., realistic voting patterns, the ordinal, cumulative and scoring voting methods, or more realistic project cost distributions, etc. This would facilitate a more complete algorithm evaluation and enable general users to test the library, or perhaps even test their own developed allocation techniques.
- *Further Library Testing:* The coverage of the unit tests is not complete, and further input and error validation and algorithm tests would provide confidence that the library can be fully trusted to produce valid allocations, or time-out as expected, for any size of parsed, generated or manually defined problems.

6.3.2 Project Extensions

- *Additional Algorithms:* The most obvious extension to the library is to implement additional optimisation algorithms, e.g., a scaled dynamic programming scheme or an exact branch-and-bound algorithm for the multi-dimensional problem, or general approximation algorithms such as particle swarm or tabu search. These would give users a wider breadth of options and we may find that these produce better results than our developed approaches.
- *Automatic Solving:* One problem with our library highlighted in the evaluation is the lack of intuitiveness when selecting an algorithm. A possible solution to this is to have the library automatically choose the algorithm based on the size and parameters of each problem. This may be of less importance where the integer programming solver is so effective, but may be useful if further research identified limitations and thus the need for alternative approaches.

- *Welfare Functions:* The design and implementation of algorithms to maximise other welfare functions, such as the diverse function [19], or perhaps algorithms that satisfy certain fairness axioms would enable users to obtain allocations with varying degrees of fairness. These allocations could then be compared and judged by budget decision-makers, or perhaps even be voted on by the community and residents themselves in a novel stage of the `PB` process.
- *Comparison Tools:* An interesting extension could be to write a set of comparison and visualisation tools for the library to display e.g., the budget utilisation or the number of voters satisfied, such that different allocations found by e.g., the approximation algorithms, could be judged and compared by budget decision-makers or alternatively by the residents and community.
- *Web Application:* As discussed, the library can be used as a basis for a web or desktop application which provides a more intuitive interface to allocate budgets. The data could be entered manually or through a `.pb` file, and an allocation could be found without having to write any code. This would undoubtedly improve the accessibility and usability of the library for budget decision-makers, and as a distant goal could even be extended to a platform to manage the entire process from end-to-end in a commercial or non-profit context.

References

[1] Aziz, H., Lee, B., & Talmon, N. (2017). Proportionally Representative Participatory Budgeting: Axioms and Algorithms. *arXiv preprint arXiv:1711.08226*. [Accessed 26/04/23].

[2] Aziz, H., & Shah, N. (2021). *Participatory Budgeting: Models and Approaches*. Springer. [Accessed 13/03/2023].

[3] Bales, R. (2022). C++ vs Python: Full Comparison. https://history-computer.com/c-vs-python-2/. [Accessed 28/04/23].

[4] Beasley, J. E. (n.d.). *OR-Library: Multi-Dimensional Knapsack*. http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html [Accessed 16/03/23].

[5] Burkardt, J. (n.d.). *Data For The 01 Knapsack Problem*. https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html [Accessed 16/03/23].

[6] Cabannes, Y. (2004). Participatory Budgeting: A Significant Contribution to Participatory Democracy. *Environment and urbanization*, *16*(1), 27–46. [Accessed 14/04/2023].

[7] Chekuri, C. (2009). The Knapsack Problem. https://courses.engr.illinois.edu/cs598csc/sp2009/lectures/lecture_4.pdf. [Accessed 14/04/2023].

[8] Cobudget: Make Ideas And Money Flow. (n.d.). https://cobudget.com/. [Accessed 27/04/23].

[9] Cole Jr, A. T. (1949). Legal and Mathematical Aspects of Cumulative Voting. *SCLQ*, *2*, 225. [Accessed 24/04/23].

[10] Cormen, T. H. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press. [Accessed 16/04/2023].

[11] Dantzig, G. B., Orden, A., Wolfe, P., et al. (1955). The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*, *5*(2), 183–195. [Accessed 07/05/23].

[12] De Biasi, M. (2014). Multi-Dimensional Knapsack Strongly NP-Complete. https://cstheory.stackexchange.com/questions/21865/multidimensional-knapsack-strongly-np-completeanswer-21867.. [Accessed 20/03/23].

[13] Dery, L., Tassa, T., Yanai, A., & Zamarin, A. (2021). A Secure Voting System for Score Based Elections. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2399–2401. [Accessed 24/04/23].

[14] de Sousa Santos, B. (1998). *Participatory Budgeting in Porto Alegre: Toward a Redistributive Democracy* (Vol. 26). SAGE Publications, Inc. [Accessed 14/04/2023].

[15] Dias, N., Enríquez, S., & Júlio, S. (2019). *Participatory Budgeting World Atlas 2019*. Oficina. https://www.pbatlas.net/pb-world-atlas-2019.html. [Accessed 21/03/2023].

[16] Emerson, P. (2013). The Original Borda Count and Partial Voting. *Social Choice and Welfare*, *40*(2), 353–358. [Accessed 24/04/23].

[17] Explained, C. S. T. (2021). *A Dynamic Program for the Knapsack Problem*. https://www.youtube.com/watch?v=whiQReoF_9w [Accessed 28/04/23].

[18] Fain, B., Goel, A., & Munagala, K. (2016). The Core of the Participatory Budgeting Problem. *Web and Internet Economics: 12th International Conference, WINE 2016, Montreal, Canada, December 11-14, 2016, Proceedings 12*, 384–399. [Accessed 26/04/23].

[19] Fluschnik, T., Skowron, P., Triphaus, M., & Wilker, K. (2019). Fair Knapsack. *Proceedings of the AAAI Conference on Artificial Intelligence*, *33*(01), 1941–1948. [Accessed 24/04/23].

[20] Forrest, J., & Lougee-Heimer, R. (2005). CBC User Guide. https://www.coin-or.org/cbc/ [Accessed 25/04/23].

[21] Garey, M. R. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. [Accessed 16/04/2023].

[22] Ge, R. (2019). Lecture 8: Greedy Algorithm II. https://courses.cs.duke.edu/spring19/compsci330/lecture8scribe.pdf. [Accessed 18/04/23].

[23] Goel, A., Krishnaswamy, A. K., Sakshuwong, S., & Aitamurto, T. (2015). Knapsack Voting. *Collective Intelligence*, *1*. [Accessed 26/04/23].

[24] Goodrich, M. T., & Tamassia, R. (2015). Algorithm Design and Applications. *363*.

[25] Gupta, A. (2005). Dynamic Programming. https://www.cs.cmu.edu/afs/cs/academic/class/15854-f05/www/scribe/lec10.pdf. [Accessed 25/04/23].

[26] Gurobi Optimisation: The leader in decision intelligence technology - gurobi optimization. (n.d.). https://www.gurobi.com/. [Accessed 28/04/23].

[27] Holland, J. H. (1992). Genetic Algorithms. *Scientific American*, *267*(1), 66–73. [Accessed 28/04/23].

[28] Kirkpatrick, S., Gelatt Jr, C. D., & Vecchi, M. P. (1983). Optimization By Simulated Annealing. *science*, *220*(4598), 671–680. [Accessed 28/04/23].

[29] Kolesar, P. J. (1967). A Branch and Bound Algorithm for the Knapsack Problem. *Management Science*, *13*(9), 723–735. [Accessed 28/04/23].

[30] Mansini, R., & Speranza, M. G. (2012). Coral: An Exact Algorithm For The Multidimensional Knapsack Problem. *INFORMS Journal on Computing*, *24*(3), 399–415. [Accessed 27/04/23].

[31] Martello, S., & Toth, P. (1987). Algorithms For Knapsack Problems. *North-Holland Mathematics Studies*, *132*, 213–257. [Accessed 27/04/2023].

[32] Martello, S., & Toth, P. (1990). Knapsack Problems: Algorithms and Computer Implementations. [Accessed 16/04/2023].

[33] Parmar, L. (2019). Knapsack Problem (Branch and Bound Approach). https://medium.com/@leenancyparmar1999/knapsack-problem-branch-and-bound-approach-1fdab6d9a241. [Accessed 28/04/23].

[34] Peters, D. (2022). *Participatory Budgeting: A Survey*. Université Paris Dauphine–PSL. https://www.dominik-peters.de/slides/dagstuhl-pb-survey.pdf. [Accessed 14/04/2023].

[35] PuLP · PyPI. (n.d.). https://pypi.org/project/PuLP/. [Accessed 28/04/23].

[36] Rey, S., & Maly, J. (2023). The (Computational) Social Choice Take on Indivisible Participatory Budgeting. *arXiv preprint arXiv:2303.00621*. [Accessed 14/04/2023].

[37] Shah, A. (2007). *Participatory Budgeting*. World Bank Publications. [Accessed 13/04/2023].

[38] Stanford Participatory Budgeting Platform. (n.d.). https://pbstanford.org/. [Accessed 27/04/23].

[39] Stolicki, D., Szufa, S., & Talmon, N. (2020). Pabulib: A Participatory Budgeting Library. [Accessed 14/04/2023].

[40] Terh, F. (2019). How To Solve The Knapsack Problem With Dynamic Programming. *Medium*. https://medium.com/@fabianterh/how-to-solve-the-knapsack-problem-with-dynamic-programming-eb88c706d3cf. [Accessed 28/04/23].

[41] Weber, R. J. (1995). Approval Voting. *Journal of Economic Perspectives*, *9*(1), 39–49. [Accessed 24/04/23].

Appendix

Theorem 1. The binary knapsack problem (BKP) is $\mathcal{NP}$-hard, in $\mathcal{NP}$ and thus $\mathcal{NP}$-complete.

Proof. Consider a decision version of BKP (B, P, T, c, v) in which we ask whether a subset $P' \subseteq P$ exists such that $\sum_{p \in P'} c_i \leq B$ and $\sum_{p \in P'} v_i \geq T$. A subset P' can clearly be verified in polynomial time, i.e., by computing $\sum_{p \in P'} c_i$ and $\sum_{p \in P'} v_i$, and thus BKP $\in \mathcal{NP}$. The subset-sum problem (SSP) [10] is a known $\mathcal{NP}$-hard problem and can be reduced to BKP in polynomial time. SSP is defined as a pair (A, k) and a question as to whether there exists some $A' \subseteq A$ such that $\sum_{a \in A'} a = k$. To accomplish this reduction, construct a decision BKP instance such that $v_i = c_i = A_i \ \forall i \in A$ and $B = T = k$. The verification result for this instance is the same as for SSP. A Yes result for the reduced problem implies that $\sum_{i \in A'} c_i \leq B$ and $\sum_{i \in A'} v_i \geq V$. However, given our reduction, we also have $\sum_{i \in A'} A_i \leq k$ and $\sum_{i \in A'} A_i \geq k$, which implies that $\sum_{i \in A'} A_i = k$. Conversely, a No result implies $\sum_{i \in A'} A_i \neq k$. Therefore, any result for the reduced problem is identical to the result for SSP. Thus, we conclude that BKP is $\mathcal{NP}$-hard, in $\mathcal{NP}$ and hence $\mathcal{NP}$-complete. □

Theorem 2. The multi-dimensional binary knapsack problem (MBKP) is $\mathcal{NP}$-hard, in $\mathcal{NP}$ and thus $\mathcal{NP}$-complete.

Proof. Consider a decision version of MBKP (B, P, T, c, v) in which we ask whether a subset $P' \subseteq P$ exists such that $\sum_{p \in P'} c_{p,j} \leq B_j$ for all $j = 1, \ldots, d$ and $\sum_{p \in P'} v_p \geq T$. A subset P' can clearly be verified in polynomial time, i.e., by computing $\sum_{p \in P'} c_{p,j}$ for all $j = 1, \ldots, d$ and $\sum_{p \in P'} v_p$ and thus MBKP $\in \mathcal{NP}$. The exact cover by three sets problem (X3C) [21] is a known $\mathcal{NP}$-hard problem and can be reduced to MBKP in polynomial time[5]. X3C is defined as a pair (U, C) where $U \subseteq \mathbb{N}$, $|U| \equiv 0 \pmod 3$ and $C = \{(a, b, c)_i \mid a, b, c \in U \text{ and } i \in \mathbb{N}_0\}$, and a question as to whether there exists some subset $C' \subseteq C$ such that every member of U is contained by exactly one triple of C'. To accomplish this reduction, construct a decision MBKP instance such that $B = (1, 1, \ldots, 1) \in \mathbb{N}^{|U|}$, $P = \{1, \ldots, |C|\}$, $T = |U|/3$. We say that c_p returns a vector such that $c_{p,j} = 1$ if $U_j \in C_p$ or 0 otherwise for all $j = 1, \ldots, d$ and that $v_p = 1$ for all $p \in P$. The verification result for this instance is the same as for X3C. A Yes result to the reduced problem implies $\sum_{p \in P'} c_{p,j} \leq B$, thus no more than one $p \in P'$ has a weight in any dimension j, and that $\sum_{p \in P'} v_i \geq T$, meaning that at least T items have been selected. Each item has exactly three weights, and so it must be then that all the capacities are exhausted exactly and thus we have found some subset $C' \subseteq C$ such

[5]The inspiration for this proof came from a response to a stackoverflow.com thread about the strong $\mathcal{NP}$-completeness of MBKP [12].

that every member of U is contained by exactly one triple of C'. Conversely, a `No` result implies that no such subset could be found. We can then conclude that `MBKP` is $\mathcal{NP}$-hard, in $\mathcal{NP}$ and hence $\mathcal{NP}$-complete. □