A Branch-and-Bound Algorithm For Solving MMS Under Budget Constraints

Ola Kristoffer Hoff

1st July 2024

Abstract

The problem of distributing resources among groups or individuals is a common event, whether it is inheritance settlements or resource allocation in computer systems. It is often desirable that the distribution provides all parties involved with their fair share. Since the 1950s, the field of fair allocation has been studied, and in recent years, the focus on allocations of indivisible goods has increased. One popular criterion for fairness is the *maxi-min share* (MMS). Different additions to the allocation problem are researched to better model real-world scenarios. One such addition is budget constraints, which represent each person's limited capacity to receive goods. For example, giving more food to a homeless shelter than they can store is pointless since the food will go bad and be wasted; each shelter's capacity to store food can be considered budget constraints.

This allocation problem is extremely hard to solve optimally, being an NP-hard problem that grows exponentially. Methods to efficiently solve these problems so they become practical for real-world scenarios are of great interest.

In this thesis, we develop a branch-and-bound algorithm to solve the allocation problem of MMS under budget constraints. Branch-and-bound methods are not a new idea; to efficiently solve problem instances, we study the algorithm's behaviour as the properties of the problem instances change. A significant amount of data is required to achieve a statistical analysis. Due to the lack of publicly available datasets suitable for this problem, a data generation tool is developed to supply the necessary data for the analysis.

The primary contributions of this work include the implementation of a branch-and-bound algorithm tailored for MMS under budget constraints and the creation of a novel data generation tool for producing diverse problem instances. Experiments conducted using the generated datasets reveal key insights into the algorithm's performance and sensitivity to various attributes. The results demonstrate that the branch-and-bound algorithm shows consistent performance with the changes in properties of the data, indicating that the problem remains challenging regardless of the properties' values. This analysis provides a valuable understanding for future research and limitations for practical applications in fair allocation.

Sammendrag

Problemet med å fordele ressurser blant grupper eller individer er vanlig, enten det gjelder fordeling av arv eller ressursallokering i datasystemer. Det er ofte ønskelig at fordelingen gir alle involverte parter deres rettferdige andel. Siden 1950-tallet har feltet rettferdig fordeling blitt studert, og de siste årene har fokuset på fordeling av udelelige goder økt. Et populært kriterium for rettferdighet er *maxi-min share* (MMS). Forskjellige tillegg til fordelingsproblemet blir forsket på for å bedre modellere virkelige scenarioer. Ett slikt tillegg er budsjettbegrensninger, som representerer den begrensede kapasiteten hver person kan ha til å motta goder. For eksempel er det meningsløst å gi mer mat til et hjelpesenter enn de kan lagre, siden maten vil bli dårlig og bli kastet; hvert hjelpesenters kapasitet til å lagre mat kan betraktes som budsjettbegrensninger.

Dette fordelingsproblemet er ekstremt vanskelig å løse optimalt, da det er et NP-hardt problem som vokser eksponentielt. Metoder for å løse disse problemene effektivt slik at de blir praktiske for virkelige scenarioer er av stor interesse.

I denne avhandlingen utvikler vi en branch-and-bound-algoritme for å løse fordelingsproblemet med MMS under budsjettbegrensninger. Branch-and-bound-metoder er ikke en ny idé; for å løse probleminstanser effektivt studerer vi oppførselen til algoritmen når egenskapene til probleminstansene endres. For å oppnå en statistisk analyse kreves det en betydelig mengde data. På grunn av mangel på offentlig tilgjengelige datasett som passer for dette problemet, er det utviklet et verktøy for datagenerering for å tilføre den nødvendige dataen til analysen.

De viktigste bidragene fra dette arbeidet inkluderer implementeringen av en branch-and-bound-algoritme tilpasset MMS under budsjettbegrensninger, og opprettelsen av et nytt verktøy for datagenerering som produserer varierte probleminstanser. Eksperimentene utført med disse genererte datasettene avslører viktige innsikter i algoritmens ytelse og følsomhet overfor ulike egenskaper. Resultatene viser at branch-and-bound-algoritmen har stabil ytelse selv når egenskapene til dataene endres, noe som indikerer at problemet forblir utfordrende uavhengig av egenskapenes verdier. Denne analysen gir verdifull forståelse for fremtidig forskning og begrensninger for praktiske anvendelser innen rettferdig fordeling.

Acknowledgements

I want to express my sincerest gratitude to Magnus Lie Hetland, my supervisor, and Halvard Hummel, my co-supervisor, for their exceptional support and guidance throughout this project. In our weekly meetings, which always went over time, I got phenomenal insights, suggestions and a platform to discuss my ideas, which has been vital for accomplishing the final result. Additionally, I appreciate Halvard's meticulous feedback on the drafts, maintaining a comment-to-content ratio of 1:1.

I would also like to express my appreciation to my deskmate, Mattias Agentoft Eggen, for being a great sparring partner throughout the project.

Lastly, I want to express a sincere thanks to Christian Lewin, my best friend and roommate for the last five years. Christian has been a constant source of inspiration, pushing me to my best not only in academic endeavours but also in various aspects of daily life.

Contents

List of Figures

List of Tables

Preface

This is an explicit declaration of what and where I have used parts of the precursor project (TDT4051 Fordypningsprosjekt) (noted also in the text with this citation: [13]) in this thesis.

- The theory section regarding fair allocation 2.1 is partly taken directly or rewritten and added to from the project.
- The explanation of how to formulate an ILP for solving MMS is rewritten into context, 3.2.1.

Chapter 1

Introduction

In this thesis, we will study *fair allocation*. In computer science, fair allocation is an optimisation problem where, given a fairness criterion, we seek the best distribution of items/things called goods given to containers/people called agents. The goal is to maximise the criterion, the fairness notion. There are many different notions to choose from depending on what you want to model and which properties of fairness is most important, one such notion is *maxi-min share* (MMS).

MMS is an advanced form of the *cake cutting* principle [3]. A method used by parents to fairly distribute cake between children. One child cuts the cake into two pieces, and the other gets to pick the piece they want. Both children have to feel that the piece they got was fair. The second child must be happy since they got to pick the piece they preferred. The first child must be happy since they divided the cake so they would be indifferent between the pieces. MMS uses the same principle, first each agent gets to distribute the goods into bundles, and with the assumption that they will end up with the worst bundle, they maximise the worst bundle. The maximised worst bundle's value is the agent's MMS value. After all agents' MMS values have been acquired, the actual allocation of the goods can be found. An allocation where each agent gets their respective MMS value is called an MMS allocation. An MMS allocation is not always possible to obtain [11]. Instead, we can maximise the allocation to get the greatest α-MMS allocation, where α is an approximation factor of the MMS allocation obtained.

Fair allocation with MMS can model many real-world scenarios, such as an inheritance settlement where the children are the agents and the assets are the goods or distributing tasks between coworkers. The model does have its limits, for example if some of the coworkers have different positions, some might work part-time. In this scenario, the model is missing something; there is a constraint on how much work each person can get done in a given time span, which is individual for each. Incorporating constraints on the agents bundles gives the problem of *MMS under budget constraints*, each good gets a weight, and each agent a budget with a capacity which must be respected [13, 21]. The model can be generalised further such that each good's weight is subjective for the agents. Even though this model would be more flexible, the introduction of

budget constraints where the weights are objective has little research. In this thesis, we will work with *budget symmetry illusion maxi-min-share* (BSIMMS), a definition of MMS under budget constraints with objective values for the goods' weight. This problem definition is similar to the definition *multiple opinionated knapsack problem* (MOKP) only with a different maximisation criterion of MMS instead of utilitarian summation [13, 38].

1.1 Contribution

This thesis aims to increase the efficacy of solving BSIMMS by identifying attributes of the problem instance and guiding the choice of optimisations used by the developed branch-and-bound algorithm. Known techniques and optimisations will be used for the branch-and-bound algorithm and a few new methods. The performance will be compared to that of a *mixed integer programming* (MIP) solver.

Q1: How can a branch-and-bound algorithm be developed to solve BSIMMS, and how does it compare to a MIP-solver?

To answer this, we will develop a branch-and-bound algorithm for the problem of BSIMMS and compare its performance with a MIP-solver, a general-purpose method capable of solving the problem of BSIMMS [13, 16]. This comparison will help give a clear view of the practicality of the branch-and-bound algorithm developed and whether or not it should be applied in real-world scenarios. The branch-and-bound algorithm will be adopted from methods used to solve MOKP [38]. We will also introduce a few new optimisation techniques.

Q2: Can attributes of a problem instance of BSIMMS, found in polynomial time, help identify the best choice of optimisations for the branch-and-bound algorithm?

If there is a way to pick which optimisations are most likely to be the most efficient at solving the given problem instance of BSIMMS with the branch-and-bound algorithm, it could greatly reduce the processing needed. Since the problem grows exponentially, finding any indicator for which optimisations to use in polynomial time could greatly reduce the runtime. These indicators are attributes of the problem instance. We will introduce a few novel attributes that encapsulate key information about the problem instances. Additionally, we will analyse the impact of these attributes' values on the performance of the different optimisations used in the branch-and-bound algorithm to understand their effect.

Q3: How can a tool for generating problem instances, with given attributes, for budgeted fair allocation be developed?

Given the lack of publicly available data for fair allocation problems with budget constraints, we will develop a tool capable of generating problem instances with specified values for the attributes defined, applicable to any budget-constrained fair allocation problem. This tool will provide a solid foundation for creating

datasets to test and validate the branch-and-bound algorithm developed in this thesis, and it can be a valuable tool for other research that lacks access to data of this format.

In summary, this thesis will develop a problem instance generation tool and a branch-and-bound algorithm for solving the BSIMMS problem. We will analysis the algorithm's performance and compare it with the MIP-solver's. Further, we will investigate if any of the defined attributes indicate which optimisations might work well for a given problem instance.

1.2 Structure of the Thesis

The structure for the rest of the thesis is as follows: In Chapter 2, the necessary background for understanding the topics explored in this thesis will be presented. Some theories from the precursor project will also be presented that are essential for the definition of the problem of BSIMMS. The branch-and-bound algorithm will be presented in chapter 3. All the optimisations, both adapted and novel, will be presented, and then the implementation of the algorithm will be given.

Chapter 4 will present the novel attributes for instances of BSIMMS and how to generate values of these. Then, it will explain how the tool for generating problem instances works. Chapter 5 shows how the experiments were made and managed. Then, in Chapter 6, we will present the results from the experiments, which we will discuss in Chapter 7 before we present the conclusion to the thesis in Chapter 8.

Chapter 2

Theory

This chapter is meant to ensure an understanding of the notation and concepts necessary for the rest of the thesis. In Section 2.1, we will present the relevant parts of the fair allocation field, including MMS, the addition of budget constraints and the definition of BSIMMS. Then, in Section 2.2, we will present the complexity of the BSIMMS problem and show why a branch-and-bound algorithm is an interesting approach to solving the problem of BSIMMS. Section 2.4 will give a background to *linear programming* and its alternative forms and how it is related to solving BSIMMS. Lastly, in Section 2.5, we will present some related work on BSIMMS.

2.1 Fair Allocations

Fair allocation of indivisible goods is an optimisation problem where the goal is to allocate a set of goods into bundles for agents and maximise these bundles given a criterion, called a fairness notion [2, 3]. There are many fairness notions; this thesis will only use *maxi-min share* (MMS). A problem instance of fair allocation for indivisible goods is denoted as $I = (N, G, V)$, using n and m for the number of agents and goods, respectively. N is the set of agents defined by $N = [1, 2, \cdots, n]$. G is the set of goods $G = \{g_1, g_2, \ldots, g_m\}$. V is the set of value functions, one for each agent, mapping a good to a real positive value: $v_i : G \to \mathbb{R}^+, v_i \in V, i \in N$. We will use additive and monotone value functions, meaning that the values are independent of the bundle, and adding a new value will always be non-decreasing [2, 3].

All solutions to the problem are called allocations, a set of bundles: $A = (a_1, a_2, \cdots, a_n)$. We can denote the set of all possible allocations as a function: $\mathcal{P} : (N, G) \to \{A_1, A_2, \ldots, A_k\}$, $k = n^m$.

2.1.1 MMS

MMS is a fairness notion used to give a metric value to an allocation [6]. An MMS problem instance is a fair allocation problem with MMS as the maximisation criterion. Before we can find the MMS of an allocation, we need to know the MMS-value of each agent. The MMS-value of an agent i, μ_i, given the set

of all feasible allocations, is defined as such:

$$\text{MMS}_i : (I) \rightarrow \mu_i = \max_{A \in \mathcal{P}(N,G)} \min_{a \in A} v_i(a), \quad I = (N, G, V)$$

In other words, we say the agent must distribute the goods into bundles (as many as there are agents) and then is given the worst bundle, in sum value from their perspective (value function). The agent then needs to maximise the minimum bundle to get as much as possible, thus maxi-min. Once we have each agent's MMS value, we can find the MMS of an allocation.

An allocation is called an MMS allocation if all the agents get at least their (maxi-min) share: $v_i(a_i) \geq \mu_i, \forall a_i \in A, i \in N$. MMS allocations are proven not always to exist, but we have α-approximate MMS allocations: $v_i(a_i) \geq \alpha\mu_i, \forall a_i \in A, i \in N, \alpha \geq 0$. We can reformulate the problem to maximise the α value instead of just seeking an MMS allocation.

$$\alpha\text{-MMS} : (I) \rightarrow \max_{A \in \mathcal{P}(N,G)} \min_{a \in A} \frac{v_i(a)}{\mu_i}, \quad I = (N, G, V)$$

2.1.2 Budget Constraints

An extension to the fair allocation problem is budget constraints. Here, we add a weight or cost to each good and give each agent a budget with a given capacity. These budgets have to be respected; thus, the number of feasible allocations for the problem can be limited. To implement the budgets, we have to know each good's cost (weight/size) and the budget capacity for each agent. We expand our notation for an instance to such: $I = (N, G, V, s, b)$. Variables s and b are added to the instance to describe the budget constraints. The value s returns the weight of a good: $s : G \rightarrow \mathbb{R}^+$. The function b returns an agent's budget: $b : i \rightarrow \mathbb{R}^+, i \in N$. We use the shorthand b_i to denote the budget of agent i. This definition of MMS under budget constraints is the same as defined by Weidong Li and Bin Deng [21].

The function for all feasible allocations respecting the budgets is: $\mathcal{P} : (N, G, b) \rightarrow \{A_1, A_2, \ldots, A_k\}, \ k \leq n^m$. Since introducing budgets removes some feasible solutions from the unconstrained problem, we have: $\mathcal{P}(N, G, b) \subseteq \mathcal{P}(N, G)$.

2.1.3 BSIMMS

In the precursor project to this thesis [13], we explored different ways of defining MMS under budget constraints. Here, we will use the one we called *budget symmetry illusion MMS* (BSIMMS), which is the same as used in by Weidong Li and Bin Deng [21], in their paper regarding budgeted MMS. From the precursor project, we have the definition of BSIMMS in Definition 1 [13].

Definition 1. *Given an instance of budget-constrained fair allocation $I = (N, G, V, s, b)$, the BSIMMS for an agent, $i \in N$, over the set of goods, G, is γ_i.*

$$\gamma_i(I) = \max_{A \in \mathcal{P}(N,G,b')} \min_{a \in A} v_i(a), \ \ |b'| = |b|, \ b'_k = b_i, \forall b'_k \in b', b_i \in b$$

In the precursor project, we also showed that BSIMMS still has the properties of MMS for scale invariance and normalised valuations as seen in Proposition 1 and Proposition 2 [13].

Proposition 1 (Scale invariance for budget symmetry illusion [12, 13])**.** *Let $A = (a_1, a_2, \cdots, a_n)$ be an α-BSIMMS allocation for the problem instance $I = (N, G, V, s, b)$ with any value functions. For any agent $i \in N$ and any $c \in \mathbb{R}^+$, if we create an alternate instance $I' = (N, G, V', s, b)$ where i's valuations are scaled by c, i.e., $v'_i(g) := cv_i(g), \forall g \in G$, then A is still an α-BSIMMS allocation for (N, G, V', s, b).*

Proof. Let γ_i and γ'_i be agent i's BSIMMS in instance I and I', respectively. For any bundle $S \subseteq G$, we have $v'_i(S) = cv_i(S)$. Therefore, $\gamma'_i = c\gamma_i$. Let $A = (a_1, a_2, \cdots, a_n)$ be the allocation where i gets γ_i. Then, $v'_i(a_k) = cv_i(a_k) \geq c\alpha\gamma_i = \alpha\gamma'_i$, for any bundle $a_k \in A$. □

Proposition 2 (Normalised valuation for budget symmetry illusion [12, 13])**.** *For a problem instance $I = (N, G, V, s, b)$ with additive value functions, if agent i's value function satisfies:*

$$v_i(G) = \sum_{g \in G} v_i(g) = |N|,$$

then $\gamma_i \leq 1$.

Proof. For contradiction, suppose that $v_i(G) = |N|$, but $\gamma_i > 1$. Let $A = (a_1, a_2, \cdots, a_n)$ be the BSIMMS partition for i. From the definition of γ_i, $v_i(a_k) \geq \gamma_i$, $\forall a_k \in A$, so $|N| = v_i(G) \geq \sum_{a \in A} v_i(a) \geq |N|\gamma_i > |N|$, a contradiction. □

2.1.4 Nash Welfare

Nash welfare (NW) is another fairness notion [8, 9], an alternative to MMS. NW works as a criterion by maximising the product of the value each agent has for their respective bundle. In other words, it aggregates the values the agents get. This makes it so mathematically that the optimal NW solution is when all agents have equal value. Note that in the definition, only the product of non-empty bundles is aggregated, and it maximises the aggregate with a criterion of maximising the number of non-empty bundles. NW differs from MMS because it is an aggregate utilitarian method, not a proportionality one. In other words, NW cares about the collective result, while MMS only cares about the worst individual assessment.

$$NW(I) = \max_{A \in \mathcal{P}(N,G)} \prod_{a_i \in A} v_i(a_i)$$

The NW has a lot of good properties when it comes to fair allocation, it is both *envy-free up to one good* (EF1) and *Pareto optimal* (PO), these are desirable traits in fair allocations [8]. The first guarantees that no agent i can envy another agent j's bundle if the good in j's bundle that i values the most is removed. Envy is defined by i valuing j's bundle more than its own. The second property guarantees that no agent's bundles can be improved by swapping goods without another agent's bundle deteriorating.

2.2 Complexity

To understand why solving the maximising the α-BSIMMS is a difficult problem, the complexity of the problem should be studied. Finding an MMS allocation is NP-hard [19] and finding a BSIMMS allocation is equally complex [13], the definition of agents MMS-value is just changed. The budget constraints might reduce the number of feasible solutions in BSIMMS compared to MMS, but the problem remains NP-hard. Maximising α-BSIMMS is even more difficult since it's now a maximisation problem and not a classification problem. Finding an BSIMMS allocation is no longer good enough, it has to be the best α-BSIMMS allocation.

The problem scales exponentially with the number of goods and with power for the number of agents, meaning we have the number of possible allocations equal to n^m. Adding a single good increases the number of possible allocations by a factor of n; brute forcing this would turn one hour into n hours of runtime. This is a troublesome growth if we have bigger problems. Further more, there exists no better polynomial $(\frac{2}{3} - \epsilon)$-approximate budget constrained MMS algorithm unless $P = NP$. One common approach to difficult problems like this is heuristic-based algorithms, such as the branch-and-bound algorithm, which will be used in this thesis. They do a brute force search, but the order of the search is based on heuristics to find a good solution quickly, such that other methods can be used to skip explicitly searching other parts of the search space.

2.3 Branch-and-Bound

A *branch-and-bound* algorithm is a method for solving discrete optimisation problems [20]. An iterative brute force method would systematically check solution after solution and keep track of the best one. However, this method quickly becomes very time-consuming for problems that are NP-hard and grow exponentially. The problem of maximising α-BSIMMS, presented in Section 2.1.3, is NP-hard. It has n agents and m goods, which results in $(n+1)^m$ unique allocations of the goods into the agents' bundles; the plus one is for the good being given to charity. With 6 agents and 20 goods, if we checked one solution each second, we would be done when the sun dies out, in approximately 5 billion years. A more practical method is needed.

A branch-and-bound method also systematically checks solutions, but in such a way that it can skip over parts of the solution space if it determines that no better solution exists. The key to this method is how the search space is structured, and branch-and-bound has a tree structure. It has a root node, representing the empty solution where all the bundles are empty. The child nodes are always subproblems of the parent node; these are the *branching* in the method, which means that all the children together represent the solution space of the parent node. For each depth of the tree, the problem thus becomes more constrained. The tree's leaf nodes are child nodes that don't have any subproblems to create. In our context, a leaf node represents an allocation where all the goods have been placed in a bundle or charity.

The *bounding* part of the method is the intelligent elimination of suboptimal parts of the tree. By keeping track of the best solution yet, since it is a maximisation problem, we can compare subproblems to this. If an internal node of the tree, representing a subproblem, can not be determined to surpass the current best solution, then there is no need to check this subproblem. There are methods for finding an upper bound on the solution a subproblem can get, if this upper bound is smaller than the best solution yet, the subproblem has been implicitly solved, skipping the subproblem is called *pruning*.

Branch-and-bound in the context of container filling problems usually use the terminology of bins and items, where the objective is to maximise the items packed into the bins. In our context, we have the agents' bundles for bins and goods for items. There are two search space tree creation methods for the branch-and-bound method. The most intuitive way is *good-by-good*; this means that all the possible allocations of a given good are explored at each tree depth. For example, the children of the root node represent the subproblems of the good allocated to each bundle and the charity. The root node will therefore have $n+1$ child nodes, in fact all nodes (expect leaf nodes) will have $n+1$ child nodes. A path from the root node to a leaf node then describes where each good was allocated, giving the tree a depth of m.

Another, less intuitive way to construct the tree is to do it *bundle-by-bundle*. In this approach, all possible sets of goods into a given bundle are explored at each depth of the tree. In other words, all the possible bundles the first agent can get at the first depth will be represented as child nodes. Then, at the second depth, the same goes for the second agent's bundle, with respect to the goods not allocated in the parent node. This gives the search tree a depth of n; there is no need to have a charity in this case since it can be implicit as the last bundle.

To summarise the difference in the approaches to constructing the search space tree. The good-by-good approach creates subproblems for all possible placements of the given good at that depth. The bundle-by-bundle creates subproblems for all possible sets of goods the bundle at that depth can have.

2.4 Linear Programming

Linear programming (LP) is a method used to solve a wide range of problems due to its generality [16]. It is a system of constraints and an objective function minimised while respecting the constraints. This is a very well-studied field with many fast-solve methods such as the simplex and interior point methods [16]. A linear program on *standard form* looks like this [16]:

$$\min c^T x, \quad \text{s.t. } Ax = b,\ x \geq 0,\ c \wedge x \in \mathbb{R}^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

The x vector is the n variable. The c vector is the costs for the variables. Together, they make up the objective function subject to minimisation. The matrix A contains m constraints on the variables, and the vector b contains the limit of the constraint. To use a maximisation instead of minimisation in the definition, negate the cost vector c. The complexity of LP methods varies; the

most used is the simplex method, which is exponential, but some interior point methods are polynomial and thus LP is in complexity class P [16].

2.4.1 Integer Linear Programming

Sometimes, an LP is too general for the problem we are trying to model. Say we modelled the sales of products from a store. In this case, selling a fraction of a product does not make sense. This leads us to this form [16]:

$$\min c^T x, \quad \text{s.t. } Ax = b,\ x \geq 0,\ c \in \mathbb{R}^n,\ x \in \mathbb{Z}_+^n,\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

The only difference between an LP and this new form is that our variables must be integer values. This problem is therefore called *integer linear programming* (ILP), and it is *NP-hard* [16]. There is a special case of ILP where we only allow binary values for our variables (zero or one). This binary form could be achieved by adding constraints to the system such that only zero and one are allowed within the constraints; it is, however, such a common method that it has its notation [16]:

$$\min c^T x, \quad \text{s.t. } Ax = b,\ x \geq 0,\ c \in \mathbb{R}^n,\ x \in \{0,1\},\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

2.4.2 Relaxed Integer Linear Programming

Since ILP is *NP-hard*, it is not easy to solve. One method used to approximate and find solutions close to the ILP is called *relaxing* the ILP [16]. This is often done when we have a binary ILP. By allowing the constraints in a binary ILP to be real numbers, the problem becomes an LP with constraints only allowing values for the variables in the interval from zero to one. Solving the resulting LP is much faster since it can be done in polynomial time. The solution from the relaxed LP is an upper bound on the solution to the ILP [16].

In some instances, the goal is to find a feasible allocation, not just an upper bound in the binary ILP. From the solution of the LP, one can construct a feasible solution to the binary ILP by rounding off the variables to integers again, making sure the solution is feasible with respect to the other constraints.

2.4.3 Mixed Integer Programming

Mixed integer programming (MIP) is a general term for the ILP and LP problems we have discussed. It accepts a mixture of integer and real variables. This gives a flexible model widely used to solve real-world problems. The solvers are called MIP-solvers. These are well-studied and highly optimised and can be used to solve MMS and BSIMMS [13].

2.5 Related Work

A lot of work is being done in the field of fair allocation, specifically on MMS. There is also a lot of work on fair allocation with budget constraints and other constraints. However, the overlap of research on MMS under budget constraints (BCMMS) is limited. Bin Deng and Weidong Li studied this and described a

polynomial algorithm for finding $(\frac{1}{3} - \epsilon)$-approximate BCMMS allocations and proved that no polynomial algorithm for $(\frac{2}{3} - \epsilon)$-approximate can exist unless $P = NP$. They also showed that $\frac{1}{3}$-approximate BCMMS is guaranteed [21]. This guarantee was improved by Halvard Hummel to $\frac{1}{2}$-approximate BCMMS [14]. The work of the branch-and-bound algorithm was based on Jørgen Steig's branch-and-bound algorithm for MOKP derived from literature on the MKP, he did not get as far as to implement it with fairness notions, but made a foundation for further research [38].

Chapter 3

The Branch-and-Bound Algorithm

This chapter will discuss the branch-and-bound algorithm we will develop to solve the problem of maximising the α-approximate budget symmetry illusion maxi-min share, α-BSIMMS for short. In Section 2.1.1 and Section 2.1.3, we presented the problems of MMS, then BSIMMS, and lastly α-BSIMMS. There, we noted that to solve α-BSIMMS, the BSIMMS value for each agent has to be found first and that finding the BSIMMS value for an agent can be done with almost the same method as maximising the α-BSIMMS; the only difference is in the valuations of the goods. The same techniques and optimisations will be used for both, and if it affects the method or optimisation if it is BSIMMS value or α-BSIMMS we are solving for, we will note the difference explicitly.

3.1 Layout of the Solution Space

When creating a branch-and-bound algorithm for α-BSIMMS, we have two methods for structuring the solution space. One, we could explore *good-by-good*, or two, we can explore *bundle-by-bundle*. The first method creates all subproblems for which bundle a given goods will end up. The second does the same only for a bundle and not a good; it handles all possible sets of goods that a given bundle can have. After a good or bundle has been handled, all their choices have been made in the search space. The choice of structure is critical for the implementation and the applicable optimisations. We explore what these methods are and then reason for our choice. The implementation of the branch-and-bound algorithm uses the good-by-good structure; this section explores how this choice was reached.

Branch-and-Bound

The branch-and-bound method represents the solution space of the α-BSIMMS problem as a tree structure. As with all trees, we have the root node at the top. The root node represents the empty allocation, where all the agents' bundles are empty. All child nodes represent a new choice in the allocation, creating a more constrained subproblem. This branching continues until the leaf nodes are

reached; a leaf node of the tree structure represents allocations where all goods have been allocated to a bundle or charity.

The second part of the algorithm, the bounding, makes the branch-and-bound different from a brute force method. Methods for determining an upper bound for a subproblem can be used to limit the search. If the upper bound is no greater than the current best solution, we prune the search tree by cutting off the subtree. Such methods are based on heuristics, so in the worst case, branch-and-bound still has to explore every node in the search space tree. It is called pruning when we can implicitly check a subproblem [20]. We see its effect in Figure 3.1. Finding a good solution early can help prune the search space, leaving only a fraction of the possible solutions to be explored explicitly. As stated earlier, there are two options for structuring the search space tree.

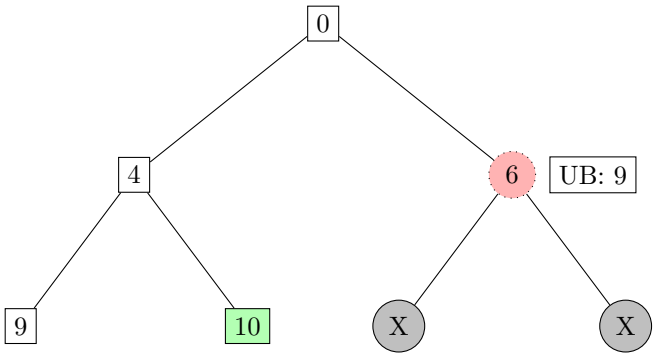


Figure 3.1: Illustration of pruning effect in branch-and-bound search tree. The rectangles are explored nodes, and the green node contains the best solution, which is 10. At the red dotted node with a value 6, the upper bound is 9. Thus, the subtree below is pruned, as illustrated with the circular grey nodes.

The good-by-good approach and the bundle-by-bundle approach. They both have properties that make different optimisations applicable, and the superior approach might depend on the problem instance to be solved. Thus, the best thing would be to implement both of these and compare their performance. We did not have the time to implement both since all the optimisations would have to be tweaked to fit the structures, and the experiments would double. For this reason, we consider both theoretically and implement the one good-by-good that shows the most significant potential.

Good-by-Good

The good-by-good approach creates a tree where a given good is allocated for each level (depth) of the tree. This means that each node will have $n+1$ children, representing giving the good to each agent and a case for giving it to charity. The depth of the tree will be a maximum of m since it ends when all the goods are allocated. Figure 3.2 shows how the search space tree would look for a case with two agents and two goods. The number of nodes in the search space tree

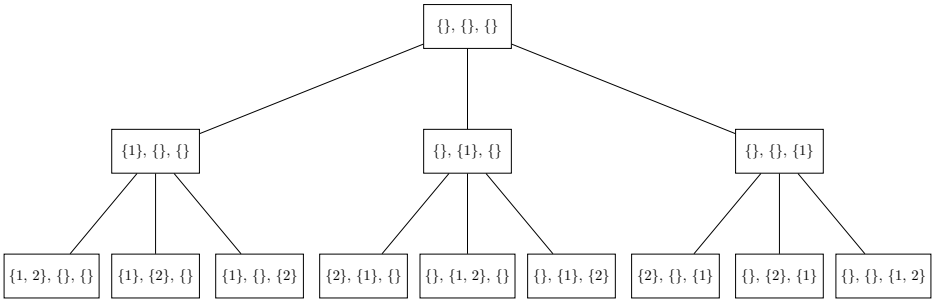


Figure 3.2: Illustration of structure for good-by-good approach with two agents and two goods. It has three bundles, two for agents and one for charity.

given the branching factor (degree), b, and depth, d, will be equal to [13]:

$$\sum_{i=0}^{d} b^i = \frac{1 - b^{d+1}}{1 - b} \text{ (geometric series)}$$

Bundle-by-Bundle

In the bundle-by-bundle approach, the level of the search space tree holds all possible bundles a given agent can get. This means the tree will have a depth equal to n instead of m. We don't need the charity explicitly if it is the last bundle to be filled since it does not affect the value of the α-BSIMMS allocation. Therefore, the depth of the search space tree is n and not $n + 1$. The number of child nodes is not a constant since it depends on which goods have already been allocated. The number of child nodes is not a constant, as it depends on the set of remaining foods. If the number of remaining goods is m', then the number of child nodes is given by

$$\sum_{r=0}^{m'} \binom{m'}{r} = 2^{m'}$$

This is the number of distinct subsets of the remaining goods. Figure 3.3 shows how the bundle-by-bundle structure looks for the same instance as in Figure 3.2, two agents and two goods. Notice that there are some duplicate nodes; when we don't allocate any goods, the node remains the same. The examples in the figures show that the good-by-good approach creates 13 nodes compared to the bundle-by-bundle approach, which creates 30 nodes.

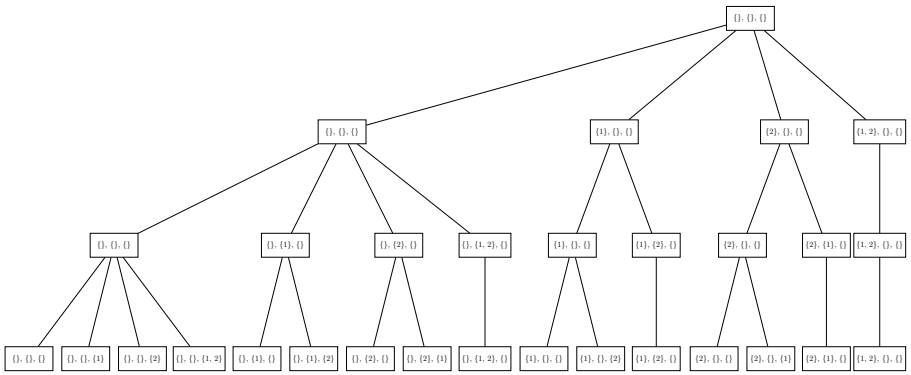


Figure 3.3: Illustration of structure for bundle-by-bundle approach with two agents and two goods. It has three bundles, two for agents and one for charity.

Choice of Search Space Structure

The choice of which structure for the branch-and-bound search space tree will be implemented directly affects the performance. The runtime can be expressed as the number of nodes explored times the time used per node. Two ways to decrease the runtime are to decrease the time used per node or decrease the number of nodes explored. The number of nodes in the search space tree differs for good-by-good and bundle-by-bundle. The good-by-good contains exactly all possible allocations and partial allocations. The bundle-by-bundle contains duplicates, and at each level of the search space tree, a duplicate node is created for the case where the bundle gets no goods. The bound-by-bound approach, however, guarantees that it can prune the last level of the search tree space representing the charity. It can also prune some of these duplicates, such as when no unallocated goods are left.

The benefits and drawbacks of the two approaches were discussed in detail in the precursor project [13]. The main point is that the good-by-good method allows for continuous improvement of the solution. Remember that the solution to α-BSIMMS is defined as the bundle with the least proportional value. The bundle-by-bundle approach always has an empty bundle until the leaf node is reached, meaning all the inner nodes of the search space tree have a value of zero. Thus, finding a new best solution only happens at the leaf nodes. The good-by-good approach is not limited like this and can use the inner nodes, partial solutions, to improve upon the best solution yet. This allows for a more aggressive pruning.

On the other hand, the bundle-by-bundle has an optimisation for limiting the number of children to explore. If a subproblem creates a bundle with a value less than the current best solution, the subproblem can be pruned. Since the least valued bundle defines α-BSIMMS solution, this bundle is an upper bound on the entire subproblem. Likewise, there is no point in adding more goods to a bundle once it surpasses the value of another bundle that has been allocated

too. This is because the current bundle will no longer define the α-BSIMMS, making further additions to the current bundle superfluous.

Our implementation uses the good-by-good method since it is much simpler to implement and use in practice. The bundle-by-bundle approach is promising, but it requires generating all the feasible bundles for an agent given the remaining goods in each subproblem. Generating all feasible bundles at once would create memory problems due to the number of bundles possible. As shown when presenting the bundle-by-bundle approach, the number of child nodes of a node is $2^{m'}$ where m' is the number of remaining goods for the subproblem. The bundle-by-bundle memory usage is bounded by $O(2^{m'})$ If a problem instance had 20 goods, it would then be $2^{20} \approx 10^6$, 1MB times the byte size of each node. The number of child nodes will double for each extra good we add to the problem, so at 30 goods, it would be a billion child nodes or 1GB times the byte size of each node. In the worst case, the empty bundle is explored first, and the next bundle has the same number of goods remaining. Continuing like this, all n agents will use n-GB times the byte size of each node in total memory. Also, it is wasteful to generate a billion bundles; if one of them turns out to prune most of them, they probably should not have been generated. An alternative is to generate them in batches; this would require a lot of bookkeeping for which bundles have been generated. Generating bundles for the bundle-by-bundle approach is a cumbersome implementation problem and, if not done correctly, will introduce a lot of overhead in the code.

The good-by-good approach can be implemented with a simple stack. All feasible subproblems can be created since only $n+1$ child nodes exist for all nodes. At the root, it will add $n+1$ nodes to the stack, and then one is taken off when explored, leaving n nodes on the stack. Then, this is repeated for each level of the search space tree; when the leaf node is explored, there are at most $n \cdot m$ nodes on the stack. With 30 goods and 30 agents, it will be 900 bytes times the number of bytes for each node. The good-by-good method is superior in terms of memory efficiency, bound by $O(nm)$. Although the bundle-by-bundle method might have worked better with the possible optimisations applicable to it, the limited time and the complexity of the base implementation made the good-by-good method the better option. We discuss this here since the choice must be clear when discussing the optimisations below. The implementation will be discussed further in Section 3.6.

3.2 Upper Bound

The first optimisation we will look into for the branch-and-bound algorithm is the *upper bound* method [20]. The upper bound method is a technique where we find an upper bound for the value of a subproblem. If this upper bound is no better than our current best solution, we prune the subproblem since it can not contain a better solution. A method for finding an upper bound has a requirement. Calculating an upper bound at each node adds a lot of overhead, so we must ensure that the pruning makes up for it. At the lowest levels of the search space tree, it becomes faster to check the entire subproblem than to calculate the upper bound; at these levels, the upper bound is skipped. There

are multiple methods for finding an upper bound to a subproblem. In this section, we will explore a couple, *linear programming* (LP) and *quick fill*. The branch-and-bound algorithm that was implemented uses only the LP method. These methods for finding an upper bound yield infeasible solutions; this is often the sacrifice in precision needed to gain the efficiency to make it worthwhile.

3.2.1 Linear Programming

One option for the upper bound function is to use a binary *Integer Linear Programming* (ILP) relaxation of the subproblem. By representing the problem of maximising α-BSIMMS as a binary ILP, the relaxed version provides an upper bound on the original binary ILP. We will first examine how to formulate the binary ILP to understand this approach. A neat trick enables us to formulate a *maximin* problem as a binary ILP.

All *linear programming* (LP), including the binary ILP, is constructed by an objective function to be maximised with respect to a set of constraints of the variables. To formulate a maximin problem, such as α-BSIMMS, which has both a maximisation and a minimisation, one must be formulated in the constraints of the binary ILP. Maximisation in the maximin is done by maximising the objective function in the binary ILP. The minimisation in the maximin is created by adding a variable z to the binary ILP and constraints to ensure that all agents' bundles have to be greater than or equal to z in value. By setting the objective function to maximise the variable z, we created a maximin in the binary ILP. Since the variable z will be maximised, and it will, by the constraints, be equal to the least valuable bundle, since all the bundles constrain it.

The value of a bundle depends on whether we solve for the BSIMMS value or the α-BSIMMS. Both are the sum of the value of the goods in the bundle, given the respective agent's value function; the α-BSIMMS value is, also normalised by dividing the value by the BSIMMS value of the respective agent. We have the definition of a binary ILP, see Section 2.4, and note that we have the maximum version of the formulation.

$$\max c^T x, \quad \text{s.t. } Lx = p,\ x \geq 0,\ c \in \mathbb{R}^n,\ x \in \{0,1\},\ b \in \mathbb{R}^m,\ A \in \mathbb{R}^{m \times n}$$

Now, we will examine how we can define each of these values to solve BSIMMS using the binary ILP.

$$x = \begin{bmatrix} x_{1,1} \\ x_{2,1} \\ \vdots \\ x_{m,1} \\ x_{1,2} \\ \vdots \\ x_{m,n} \\ z \end{bmatrix}, \quad x \in \{0,1\}^{n \cdot m + 1}$$

Here, the value $x_{j,i}$ is binary, indicating that good g_j was allocated to agent i. If a good is not allocated to any agent, it is implicitly given to charity. Remember

that we want to maximise the variable z; our objective function will, therefore, be:

$$c^T x = z$$

The cost vector c has the same structure as the variable vector x. However, to obtain the z as the only maximisation variable, it contains $m \cdot n$ zeros, then a one at the end. This ensures that only the variable z affects the objective function and is thus maximised. This gives the following structure to the cost vector c:

$$c^T = \underbrace{\begin{bmatrix} 0 & \cdots & 0 \end{bmatrix}}_{n \cdot m} \underbrace{\begin{bmatrix} 1 \end{bmatrix}}_{z}$$

Next, we define the constraints of the binary ILP. We need to construct constraints that bound z to be equal to or less than all the agents' α-values. We also need to add constraints so as not break any budget constraints and ensure that goods are given to one agent at most.

$$\mathrm{L} = \begin{array}{c} \overbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}^{n \cdot m} \quad z \\ \left[\begin{array}{ccccccccccc} s(g_1) & \cdots & s(g_m) & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 \\ 0 & \cdots & 0 & s(g_1) & \cdots & s(g_m) & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & s(g_1) & \cdots & s(g_m) & 0 \\ \hline -v'_1(g_1) & \cdots & -v'_1(g_m) & 0 & \cdots & 0 & 0 & \cdots & 0 & 1 \\ 0 & \cdots & 0 & -v'_2(g_1) & \cdots & -v'_2(g_m) & 0 & \cdots & 0 & 1 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & 0 & \cdots & 0 & -v'_n(g_1) & \cdots & -v'_n(g_m) & 1 \\ \hline 1 & \cdots & 0 & 1 & \cdots & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 1 & 0 & \cdots & 1 & 0 & \cdots & 1 & 0 \end{array}\right] \end{array}$$

Each row at the top of the constraint matrix L sums the size of all the goods an agent got, one row per agent. We have the budget corresponding to the agent in the same row in the vector p. Each row in the middle part of L sums the α-BSIMMS value the agent got and subtracts this from the value of z. In other words, these constraints ensure that all agents have at least as much normalised value as z. The value is not normalised if we solve for the BSIMMS value of an agent. The value for the normalised valuations $v'_i(g_j)$ is defined as follows; note that for solving BSIMMS, we use $v_i(g_j)$ instead:

$$v'_i(g_j) = \frac{v_i(g_j)}{\gamma_i}, \quad i \in N, \;\; g_j \in G$$

In the p vector, the corresponding middle rows are all zeros. This is because we get an equation for each i-th row as such:

$$z - \frac{v_i(a_i)}{\gamma_i} \leq 0 \Leftrightarrow z \leq \frac{v_i(a_i)}{\gamma_i}$$

Each row at the bottom of L sums up the number of times a good j has been allocated. Each row is then constructed with a 1 in every m-th column, offset

by $j-1$ for the j-th good row. Each corresponding row in the vector p has a 1 since each good can, at most, be allocated once. The complete constructed vector p is then on the following form:

$$p = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ \hline 0 \\ \vdots \\ 0 \\ \hline 1 \\ \vdots \\ 1 \end{bmatrix}, \quad p \in \mathbb{R}^{n+n+m}$$

Now, we have constructed a binary ILP for the problem. However, remember that binary ILP is an NP-hard problem. Therefore, we want the relaxed version for our upper bound function since it turns the binary ILP into an LP with complexity P. As discussed in Section 2.4, this is obtained by changing the variables in vector x from binary to being defined in the interval between zero and one, i.e.,

$$x \in \big[0,1\big]^{n \cdot m+1}$$

This LP-relaxed version of the binary ILP can be solved with various standard industry methods (i.e., the Simplex method or Interior-Point methods) [16]. Since the LP-relaxed problem is in P and the actual problem is NP-hard, we can expect it to be way quicker to solve.

Constructing and solving an LP takes much longer than just checking a leaf node's value. Even for a few hundred nodes, simply checking would probably be faster. There should be a threshold for the minimum size of a node's subtree for the upper bound to be calculated with the LP. By taking the depth of the search space tree minus the level of the subproblem in the tree, an expression for the subtree size can be found. The value for this threshold would optimally be based heuristically on how big the expected overhead of computing the upper bound is compared with the expected cost of just solving the subproblem. Having the threshold as a parameter in our experiments could have been done. Still, to limit the number of parameters in the experiments, some smaller standalone tests were executed to find a suitable value. Limiting the upper bound to nodes with at least seven goods remaining seems to be a reasonable threshold.

Some goods will already be allocated when constructing the binary ILP for a subproblem. There are two options for incorporating this into the model. First, one could set the goods that are already allocated to be allocated by adding constraints locking the respective variable, x_{ji}, to one. Second, the instance could be reduced by removing the goods from the binary ILP and adding their respective value to the l vector for the constraints summarising the agents' value. The second option is used in the implementation of the branch-and-bound algorithm.

3.2.2 Quick Fill

A second method for finding an upper bound is greedily solving the subproblem, referred to as *quick filling*. This approach trades off precision for speed. In general, there is a trade-off between speed and accuracy, meaning the faster the upper bound is calculated, the less precise it is, and thus less useful. However, imagine that a terrible path is explored; even a less precise upper bound could still determine that the path can be pruned.

Consider the method of using a surrogate budget, which combines agents' budgets into one big budget, a technique used for the *multiple knapsack problem* (MKP) [25]. However, we cannot use a surrogate budget for the α-BSIMMS problem since the solution is derived from the bundle with the least value. The objective, maximising α-BSIMMS, is not dependent on all the bundles, only the one with the least value.

In other words, an upper bound could be derived from a single bundle. If one budget is filled optimally or greedily with the remaining goods, we derive an upper bound for the subproblem from that bundle alone. This requires less overhead compared to the LP solver. The optimal filling can be done by dynamic programming of a knapsack problem instance, which would run in pseudo-polynomial time with respect to the remaining budget [17].

Suppose we choose to derive the upper bound from only one bundle; selecting which one to fill becomes crucial for the precision of the upper bound. One possibility is to instead derive the upper bound for each bundle and use the tightest upper bound. Instead of filling the bundles optimally, we could fill them greedily; this would further decrease the accuracy and increase the efficiency. An upper bound can be found by greedily filling a bundle with the highest value per weight (profit) good available until the good does not fit in the remaining budget and then allowing that good to be divisible to ensure an upper bound.

An even quicker and less precise method would be to find the most profitable good left for the agent and scale that to fit the remaining budget. When we calculate the BSIMMS value for each agent, the surrogate budget works. A surrogate budget is possible because the budgets' size and the value functions are the same, enabling us to divide the total value of the surrogate by the number of agents to derive an upper bound [13].

Ultimately, we did not have time to test these quick-fill methods. We also needed to limit the number of parameters in the experiments. Hybrid methods can also be developed, where the LP finds the upper bound at the start of the search space tree, and then when crossing a threshold in depth, the upper bound could be found by a quick fill method. This would allow for upper bounds for more subproblems, and when fewer goods remain, the error in the upper bound is reduced, making the quick-filling methods more applicable.

3.3 Bound and Bound

This section will look at a technique called *bound and bound* [25]. The bound and bound technique is an extension of the upper bound technique. In addition to computing an upper bound, the bound and bound technique computes a lower bound for the value of the subproblem. Just as the upper bound states that the solution cannot be greater, the lower bound ensures that the solution cannot be lesser. This can be used to prune subproblems as well. If the lower bound is greater than the best solution thus far, we know that solving the subproblem will yield a better solution, and we can update the current best value immediately. This allows for more pruning of subproblems with the upper bound. The bound and bound method has the property that if the lower bound and upper bound are tight (equal), that must be the solution to the subproblem. This is inferred, and we don't have to solve the subproblem explicitly. In contrast to the upper bound, the lower bound has to be derived from a feasible solution to the problem. We will look at an appropriate lower bound technique for both upper bound methods.

3.3.1 Linear Programming

When the upper bound is found with an LP, we get an upper bound and a fractional allocation that yields this bound. We could round this allocation to get a lower bound. Rounding off the allocation can be done using a very rudimentary method. Begin by finding the agent that received the largest fraction of each good from the fractional allocation. Then, give the entire good to that agent, providing that the agent's budget is respected. Repeat this process for all goods until each goods has been allocated or no agent has any remaining budget left for more goods. If any remaining goods fit into agents' budgets that did not have the largest fraction, greedily give these goods to the agents. The resulting rounded allocation will be feasible since we never break the agents' budget constraints. The rounded allocation yields a lower bound on the subproblem, derived from the fractional allocation yielding the upper bound.

3.3.2 Quick Fill

Similarly to an upper bound, we can construct a greedy quick-filling style algorithm for obtaining a lower bound on a subproblem. In contrast to the upper bound, an algorithm for a lower bound cannot focus on a single agent, but must instead try to fill all of the worst-off bundles. We consider a simple algorithm for this: For as long as possible, select the worst-off agent and give this agent the most profitable unallocated good (highest value per weight) that fits within the agent's remaining budget.

One could improve this algorithm with a technique similar to the *Yankee Swap* method by Viswanathan and Zick [41]. Yankee Swap uses Lorenz domination (optimality) to improve the allocation. If the agent has filled its budget, it can swap a set of its goods for another agent's, as long as it can ensure that the other agent's value will not decrease. Using this for subproblems where goods are already allocated, we must ensure that only the unallocated goods in the given subproblem can be swapped. A swap with another agent can be achieved

if that agent values the goods given to it equally or more than the ones it had or by adding extra goods yet to be allocated to the agent's budget. This modification adds a lot of computation, so it is unclear if the potential improvement of the lower bound is greater than the overhead.

3.4 Picking Orders

A central concept in branch-and-bound algorithms is that the faster we find a good solution, the less work we must do. Imagine that an algorithm begins traversing the tree of the solution space and then, by some stroke of luck, finds the best solution in the first leaf node. Then, almost the entire search space can be pruned, given a good enough upper bound calculation. Conversely, the solutions are explored from worst to best, and no pruning is done. These are the best and worst cases of the branch-and-bound algorithm, and the number of explored nodes can be very different. Since the runtime of the branch-and-bound algorithm is the product of the number of explored nodes and the average time per node, the difference in runtime between the best and worst case can be significant. Thus, the order in which it explores the search space is vital for performance. Two factors determine the order: the *picking order* of the agents and goods. In this section, we will discuss different picking orders. The picking order for the goods is chosen during preprocessing and remains static throughout the search. The reason behind this choice will be discussed further in Section 3.6, but it is essential to mention it here to provide a clear understanding.

Consider an example to understand where the picking orders play a role in the branch-and-bound algorithm. At each step in the algorithm, a node (subproblem) in the search space tree is considered. This node has one child node for each agent, representing the case where the respective agent gets the next good to be allocated and one state where the good goes to charity. The goods picking order, which is sorted during preprocessing, determines the good to be allocated. Thus, each level in the search space tree corresponds to a unique good. The $n+1$ child nodes will all need to be explored, but the order in which this is done can lead to better pruning. The order for the child nodes to be explored is found in the parent node. All the child nodes are created simultaneously and then sorted by the agents picking order from the parent node.

All the picking orders we present are the same for both agents and goods, with some minor differences in details, which will be pointed out. However, to give an intuitive explanation for them, they will be presented as how they work from the perspective of an order for the agents. Remember that the goods picking order will be determined in the preprocessing step and kept static, while the agents picking order will be unique to each subproblem. Also, the picking orders are deterministic and will have an inverse. If an order is sorted by maximum value, having an order sorted by minimum value is a straightforward extension. It does not always make intuitive sense, but all these are implemented in the branch-and-bound algorithm such that their behaviour can be observed and not just theorised.

3.4.1 Naive Random

The choice of not having a picking order and forgoing the overhead is an option. To avoid structured properties in the problem instance, the order of the agents and goods is shuffled. This is a cheap preprocessing step but mitigates the risk of having a bad structure. The purpose of this picking order is to make it possible to compare if the additional overhead of creating an active picking order choice is beneficial. It gives a baseline performance for the other picking orders. There are some theoretical properties with this approach. When we begin traversing the solution space, the goods are always given to the first agent in the order, then to the second agent, and so on. If this is combined with a picking order for the goods of the highest value first, the first agent will get a lot more value than any of the others, which is an unbalanced (unfair) allocation. It will not work very well to get a good solution quickly. Thus, unless all agents have similar value functions, the naive random approach cannot find an optimal solution quickly. This approach will fill the bundles left to right in the agents' picking order. Therefore, if there are not enough goods to fill all the budgets, the first solutions are very poor since the last agent possibly gets little to no goods and value. Since the α-BSIMMS is the value of the worst bundle, it will be poor. On the contrary, this approach has some benefits. At the end of the search, the bundles will then be filled from right to left, yielding a mirrored situation to the start. This means that better solutions must be found between the beginning and end of the search. These better solutions might be good enough to let the algorithm prune the end of the search.

3.4.2 Random

A very natural option for a picking order is the random order. Here, we choose a random order for the agents at each node. This guarantees neither good nor optimal results, but it is unlikely to be systemically the worst either. It gives us a $\geq \frac{1}{n+1}$ chance of choosing optimally each time. The probability is even higher if multiple choices lead to an optimal solution. Note that, in the actual implementation, the charity is forced to be the last. So, the order is not chosen completely at random; the probability of choosing optimal is a bit lower. If the subproblem is already down a sub-optimal path, it could still make the optimal choice for that subproblem. The chance of picking the optimal order the first time is slim. There might be more bad solutions than good ones, so the expected solution from a random order would probably only be somewhat decent. However, it does avoid (with a probability) the bad cases of exploring a lot of bad solutions at the start, as the method in the previous subsection does. However, this method is independent of the state of the allocation and does not leverage any of the information available to it. For example, an agent might have no goods, and the others have a lot. This information could easily be leveraged to ensure that the agent without anything would be explored first. This method, however, does not care and gives the goods with equal probability to all the agents.

3.4.3 Nash

During the precursor project, a new picking order, *Nash ordering*, was introduced and discussed [13]. Nash ordering takes inspiration from the well-known and widely used *Nash Welfare* (NW), presented in Section 2.1.4. Each agent's bundle, a_i, from the subproblems allocation, A, get a *Nash score*, NS, defined as:

$$NS(a_i) = \prod_{k \in N} v_k(a_i) \text{ where } v_k(a_i) > 0, \ \ a_i \in A$$

In other words, the Nash score for each bundle is calculated by taking the product of the total value each agent subjectively assigns to the bundle, ignoring zero valuations. The ordering is then by descending Nash score.

Applying the Nash score to the goods is a bit different; it is defined as the product of all agents' value for that good.

$$NS(g_j) = \prod_{i \in N} v_i(g_j) \text{ where } v_i(g_j) > 0, \ \ g_j \in G$$

The hypothesis is that when ordering the goods by decreasing the Nash score, there is a greater chance that the goods few agents want will end towards the back of the order. When we have a charity, it could be a correlation between a good's Nash score and its chance of being in the charity. This would increase the chance of the optimal solution being found earlier. If a set of goods goes to the charity regardless, allocating these last will allow us to find the best value earlier in the search tree. This is because adding goods to the charity does not change the value of the subproblem.

The Nash ordering is a less apparent good choice for the agents, but it could, in essence, work to ensure that we give an even amount of value to each agent from the agents' perspective as a collective. Imagine that a good that all agents value highly was given to one of the agents. The Nash score of that agent's bundle will significantly increase since everyone values that good highly. Thus, the more goods a single agent values, the more they get distributed (to other agents), the higher their Nash score and the less likely it is that the receiving agents will be explored before the given agent. Of course, this is done from all agents' perspectives and merged into a collective agreement of the order in which to explore, the Nash order.

3.4.4 Value and α-BSIMMS

One simple picking order is ordering by the goods *value*. The intuitive way to think of this ordering is by picking the maximum order first to quickly dish out as much value as possible. The minimum ordering is also implemented in the branch-and-bound algorithm. For the α-BSIMMS, we will use the normalised values.

The goods picking order by value is created by taking the sum of each agent's value on the given good. This is the same definition as the Nash score for goods,

with the sum instead of the product.

$$g_j : \sum_{i \in N} v_i(g_j)$$

The agents picking order by value based on the subjective value of their bundle. For an agent i with bundle a_i in the subproblems allocation A, we have the value of the agent defined as follows:

$$i \in N : v_i(a_i), \;\; a_i \in A$$

From the perspective of the agents' order, exploring the minimum order first is most intuitive. By applying this to the parent node, the order of exploration of the child nodes will be by which agent's bundle had the least value. We prioritise first increasing the one with the least value since it is the current bundle defining the solution's value. Exploring in this order ensures that it focuses on distributing the value evenly. Still, it does not consider the budget's capacity, which might significantly hinder it in some cases.

3.4.5 Weight

Just as one could explore by value, one can explore by the *weight* of the goods. Each good has an objective weight (in contrast to its value). Both maximum and minimum weight order will be explored. For the goods, picking order by weight is simple; order them by their weight.

$$g_j : s(g_j)$$

The sum of the weight in the agents' bundles will be used for the agents' picking order. For an agent i with bundle a_i in the subproblems allocation A, we define its weight as follows:

$$i \in N : \sum_{g_j \in a_i} s(g_j), \;\; a_i \in A$$

By ordering the goods by maximum weight, the budgets will be filled as quickly as possible; this might aid in reaching the outer feasible nodes in the search space tree, where these nodes' children are infeasible. It might be a good strategy to search for a solution along the border of the constraints.

For the agents' picking order, it aims to fill the budgets evenly by maximum weight first and will not guarantee anything for the value of the bundles. The further down the search space tree we explore, the smaller the weights become, which might be beneficial since the remaining capacities of the budgets also become smaller. It naturally balances the space left by supplying more *fine-grained* goods, which are easier to place. However, it might suffer from not correlating with the objective of maximising the α-BSIMMS; the weights and values are independent.

3.4.6 Profit

We have presented picking orders for both the value and weight of goods. Both have potential; here, we will look at the picking order of *profit* per good, which

is the good's value divided by its weight. Both value and weight are factors that will affect the bundle. We get a combination of these factors by looking at the *profit*, as with all the other picking orders, a maximum and minimum ordering for profit will be implemented in the branch-and-bound algorithm.

For the goods picking order, this is simple: we use the definition of the goods value ordering divided by the goods weight ordering; we then have the following definition:

$$g_j : \frac{\sum_{i \in N} v_i(g_j)}{s(g_j)}$$

We do the same for the agents' picking order; we take the definition of the agent's value ordering divided by its weight ordering. For an agent i with bundle a_i in the subproblems allocation A, we have the profit of the agent defined as follows:

$$i \in N : \frac{v_i(a_i)}{\sum_{g_j \in a_i} s(g_j)} = \sum_{g_j \in a_i} \frac{v_i(g_j)}{s(g_j)}, \;\; a_i \in A$$

The goods picking order will make sense for maximum profit; using the goods that have the most *bang-per-buck* first seems like a good greedy choice. For the agents picking order the minimum profit first order seems most promising. The agent with the least *bang-per-buck* is prioritised to get the next good. There is, however, no guarantee that the profit margin for the next good is any better for the given agent. This limitation is imposed by the choice to have a static ordering of the goods. If this limitation was not imposed, we could find the good with the highest profit margin for the agent in question.

3.5 Non-Naive

This section will discuss a class of optimisations we will call *non-naive*. These optimisations are not all based on heuristics as the other optimisations are, but rather on mathematical proofs that allow us to prune subproblems. We can perform multiple theory-based quick checks that prune the search space tree with minimal effort. Unless explicitly stated otherwise, every technique presented in this section has been implemented in the branch-and-bound algorithm.

3.5.1 First Solution for Finding BSIMMS

For the problem of maximising α-BSIMMS it is shown that $\alpha \geq \frac{1}{2}$ always exist [14]. Ignoring every branch with an upper bound of $< \frac{1}{2}$ allows for effective pruning from the start of the algorithm. A similar lower bound for the existence of an approximate value for the BSIMMS value of an agent is hard to find. No lower bound was found in the existing literature. Instead of mathematically finding a lower bound for this value, an alternative approach is implemented in the branch-and-bound algorithm. It finds a lower bound for the BSIMMS value of an agent in the preprocessing step using the method of greedily allocating goods as described as *quick filling* in Section 3.2.2. There is also a $\frac{2}{5}$-approximation algorithm for α-BSIMMS that can be used with identical agents [14]; this algorithm was not implemented.

3.5.2 Exceeded Proportional Share

An upper bound for BSIMMS value can be obtained by considering the proportional share. When finding the BSIMMS value for an agent, all the budgets and value functions are identical. The upper bound for the BSIMMS value is $\frac{1}{n}$th of the total value. We can see this by contradiction: if the least valued bundle has a value of $> \frac{1}{n}$ this would leave the remaining $n-1$ agents to share the remaining value $< \frac{n-1}{n}$. Even with a perfect distribution of value, they must all get $< \frac{1}{n}$ each. This would make the bundle we stated to be the least valuable, the most valuable bundle, which is a contradiction. Therefore, when exploring the search space tree, if a bundle in a subproblem already has more than $\frac{1}{n}$th of the total value, there is no need to allocate more goods to it, since it can never be the least valuable bundle defining the BSIMMS value.

The upper bound could be smaller than $\frac{1}{n}$th of the total value. If guaranteed that at least x goods will be in charity, these do not contribute to the final total value. Therefore, the maximum actual total value distributed could be found by subtracting the x least valuable goods from the total value of all goods. The upper bound would then be the $\frac{1}{n}$th proportional share of the maximum actual total value. This total value reduction is not implemented, and the method for finding x has not been developed or researched.

A similar method for pruning can be utilised when exploring the search space tree for α-BSIMMS. When creating the child nodes for a subproblem, we can check the normalised value of each agent's bundle. If an agent's bundle has more normalised value than the upper bound value found for the subproblem, the child node where that agent gets the good is superfluous to explore.

3.6 Implementation

In this section, we will look at the implementation of the branch-and-bound algorithm and how its design choices tie in with the optimisations discussed earlier in the chapter. Figure 3.4 shows a flowchart of the overall structure of the algorithm. This structure is the same for both BSIMMS and α-BSIMMS; if there are any differences, these will be discussed. The structure of this section follows the flowchart to give a step-by-step overview. We will call the algorithm finding α-BSIMMS the α function and the one finding BSIMMS-value the BSIMMS function. The code can be found in the GitHub repository here.

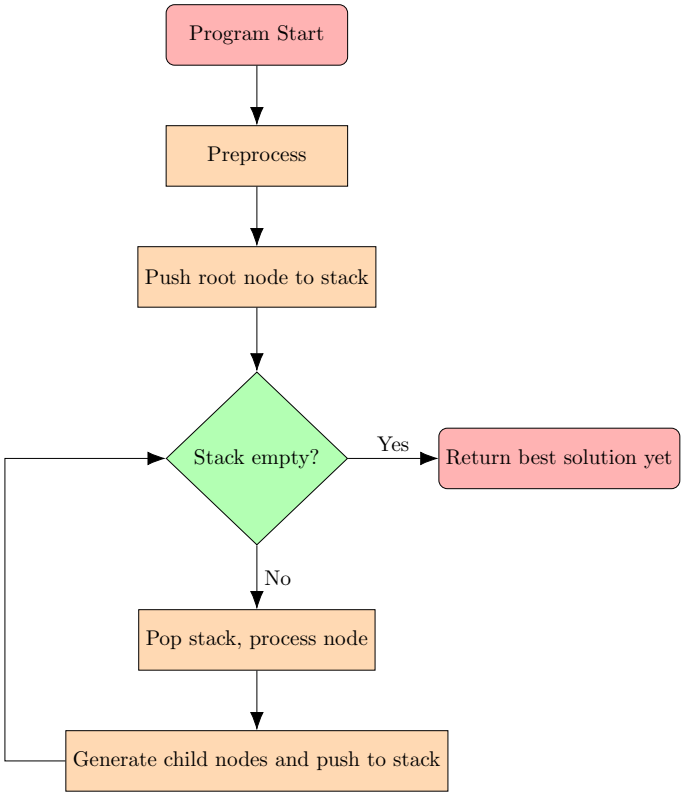


Figure 3.4: Flowchart of how the branch-and-bound algorithm works.

3.6.1 Setup

Before the algorithm is started, we process the passed command line arguments. This sets the appropriate flags and values for the optimisations to be used. It also performs a compatibility check for the arguments passed to avoid undefined or hidden behaviour. For example, a valid file path to the problem instance has to be passed. This works as a sanity check for the user and preserves the precedence of the arguments passed. This ensures the program is correct and the branch-and-bound algorithm works as the chapter presents.

3.6.2 Preprocessing

The first step of the algorithm is the preprocessing. For the α function, this step mainly finds the BSIMMS for each agent through the BSIMMS function. If an agent has a BSIMMS of zero, the agent is removed from the problem instance. This is done by a recursive call to the α function with the respective agent removed. The agent is added to the result with an empty bundle when exiting the recursive call. Recursion solves our problem elegantly and has little overhead since we have n agents limiting the depth of the call stack.

The BSIMMS function will find a feasible solution with a greedy algorithm to ensure that we have the best solution so far is greater than zero, if the non-naive optimisation is turned on.

Both functions will generate the appropriate picking order for the goods and set up all other variables and states needed before handling the first node, the empty allocation, and the root node of the search space tree.

3.6.3 Main Loop

The algorithm's main loop continues as long as we still have nodes (states) in our search stack. A state represents a node in the search tree, so an allocation. In each iteration, we pop a state of the stack and process it. The processing may generate child nodes, which are pushed onto the stack. This results in a depth-first exploration of the search tree and allows us to keep an extremely slim overhead. The memory usage scales with $O(nm)$. This simple structure provides manageable memory consumption, leaving the algorithm compute bound.

Process Current State

When processing the current state, the first thing considered is the state's α-BSIMMS (BSIMMS) value. If this value is greater than the best thus far, the best so far value is updated to the state's value. If the state is a leaf node, the processing of the state ends.

If the state is not a leaf node and the upper bound optimisations are on, the upper bound of the state will be computed if enough unallocated goods remain. Section 3.2.1 discussed the threshold of at least seven remaining unallocated goods. It constructs an LP from the state and solves it.

If the bound and bound optimisations are on, this will be handled right after the upper bound. The implementation uses the rudimentary rounding technique as described in Section 3.3.1. If the bound is tight, the subproblem is pruned, and the best solution so far is updated. If only the lower bound is greater than the best solution, the best solution will be updated, and the algorithm will continue.

Generating New States

The last step of the main loop is to generate new states. The new state for the good going to charity is pushed to the stack first. Next, new states are created for each agent receiving the next good. These states will be filtered based on whether an agent already has more than their proportional share or the upper bound of the subproblem; if so, the state is ignored (see Section 3.5.2). The states are sorted according to the agents' picking order and pushed onto the stack.

Representation of State and Allocation

This section is not directly tied to the algorithm but highlights an essential implementation aspect. The focus of the implementation was on simplicity and correctness. To achieve this, the choice to abstract away the state was made.

The only way to change the state was through well-tested and error-handling function calls, which ensured the correctness of the state. While this introduces more overhead than a bare-bones method, where the state would be altered directly, it significantly improves the development, testing, and validation processes. This consistency was invaluable for debugging edge cases encountered during experiments, thanks to the error messages implemented in the allocation object, rather than dealing with the infamous *segmentation fault* in C++ [40]. Since the algorithm uses randomised values in some instances, reproducibility is not always possible, so having a good log is essential for resolving these issues.

Chapter 4

Data Generation

This chapter will discuss the generation of the datasets used in the experiments. Different properties, which we will call attributes, of problem instances will be discussed and presented. An attribute of a problem instance is any measurable metric of the problem. The number of goods and agents are examples of attributes. Here, we will look at attributes found in polynomial time. These might provide insight into which optimisations the branch-and-bound algorithm should use to solve a particular instance of the problem. Lastly, an overview of the program created to generate these datasets will be presented.

4.1 Representativeness of Real-World Data

The objective of solving the problem of fair allocation is to solve challenges faced in real-world scenarios. Therefore, it is natural to want to test our branch-and-bound algorithm on data representing real-world cases. This is, after all, what we are trying to model the algorithm to solve. The problem is that no publicly available datasets give subjective values for a set of goods in combination with the constraints of budgets. There are multiple approaches to address the lack of accessible data. One approach is a purely generative method, which creates sets of randomised data following some given distribution. This would allow the data to cover various cases and enable a good analysis of the algorithm's behaviour for the different attributes.

A second approach to get data is to find datasets of real preferences and add the rest synthetically. The website "www.preflib.org" offers public datasets of preferences from combinatorics and elections to sports [26]. To turn preferences into the problem of budgeted fair allocation, take a dataset as follows: Use the preferences over the set of options (goods) as permutations for value functions. Generate a set of random values in a given interval with a given distribution (i.e. normal, uniform, etc.), then sort these numbers and map them to the permutation of the preference the agent has. Do this for all the agents. Now, the only thing missing is the budgets and weights. The weights can be generated like the values with different distributions. The budgets are harder to define. However, we have a definition for the interval of interest. Every agent has to be at least able to hold one good, and not all budgets should be big enough to

hold every good. In any other scenario, the agent cannot get any goods, or the budget constraints are redundant, and the problem is MMS and not BSIMMS. Thus, the interval of the budget values is known. It is hard to say which real scenario the different distributions of weights and budgets models well, so the best approach would be to use different distributions to cover multiple cases. This is a synthetically made dataset with preferences based on real-world data.

In this thesis, we will use the first approach, the generative method. The second approach is not used in this thesis due to the limited number of datasets available and the lack of a balanced variety of the problems that get covered. An analysis would be hard to do with a small dataset that is not guaranteed to be balanced.

4.2 Attributes in Datasets and Their Intervals

As discussed in Chapter 3, the branch-and-bound algorithm is only different to a brute-force search if it has good heuristics for its search, allowing it to prune subproblems. Attributes of the problem instance can be leveraged to help decide which configuration of optimisations is used in the branch-and-bound algorithm. These attributes can be found in the preprocessing of the algorithm; if found in polynomial time, this extra computation will be negligible for the overall runtime. We generate datasets with varied attributes to analyse how the algorithm responds to different values of the attributes. Creating datasets with given values for the attributes, and we see how each optimisation performs when those attributes change.

By creating a range of datasets where one attribute changes in an interval, we can isolate the effect that attribute has on the algorithm. Then, by analysis, it can be determined based on the attribute which optimisation should be used for the given attribute value. A version of the branch-and-bound algorithm could then pick the optimisation based on the values of the attributes to statistically maximise the change of picking the fastest configuration of optimisations.

In this section, we will define the attributes of the problem instances, discuss how their metric works, their intervals and how to generate them. To generate datasets in an interval for a given attribute, the attribute must be in a bound interval and have a measurable metric. Attributes should be unaffected by the problem size, defined by the number of goods and agents, and will be defined as a function mapping onto a bound interval: $\rho(i) \to [0, 1]$. This is essential for having comparable attribute values between different problem instances. Additionally, there must be some way of generating a problem instance with a given value in the interval. While it does not have to be an invertible function, getting from i_t to i_{t+1} should not be too difficult, allowing for the next step in the interval.

4.2.1 Basic Attributes

Some attributes are very basic and do not need a lot of explaining and discussion. Therefore, these are collected into one group. The basic attributes are

the number of goods, m, the number of agents, n, the total possible solutions, $(n+1)^m$, and the ratio of goods to agents, $\frac{m}{n}$. These represent themselves, and their metric is just their value. These attributes do not have an upper bounded interval since they are directly dependent on the problem size; therefore, expecting them not to be and have a bounded interval is a contradiction. Thus, these attributes do not have upper bounded intervals. They will, however, be bound by the problem instances we will create.

4.2.2 Percent of Goods Fitting in Budgets

One interesting attribute in the problem instances is the proportion of the total budget used. It can indicate how constraining the budget constraints are on the search space. If the goods take up many times the space in the budgets combined, then there will be a lot of goods in the charity. This means that the budget constraints could drastically limit the number of feasible solutions compared to the number of all allocations. This is quite a simple metric; it is defined as the sum of the weights of the goods divided by the sum of the budgets of the agents:

$$\text{Percent of goods fitting} = \frac{\sum_{g_i \in G} s(g_i)}{\sum_{i \in N} b_i}$$

The interval for this metric is only interesting for BSIMMS in the interval from $\frac{1}{n}$ to near $\frac{m}{n}$. If all agents have a budget that can fit all goods, the metric would be $\frac{1}{n}$. Anything less than this is not interesting since the problem would be reduced to MMS, not BSIMMS. If the ratio approaches $\frac{m}{n}$, the average goods weight equals the average agent's budget. Anything more is not interesting since the goods individually don't fit in the budgets. The interesting interval for this metric is thus defined as $[\frac{1}{n}, \frac{m}{n}]$. As the value approaches either bound of the interval, there will be fewer interesting cases. Thus, when choosing values in the interval, the interesting cases will be clustered more in the middle.

To set a given value of this attribute for a problem instance, assume the weights of the goods have already been determined. Calculate the total budget capacity required based on the desired attribute value from the total weight of the goods. Then, scale the budgets to match the desired total capacity.

4.2.3 Normalised Average Permutation Distance

The first novel attribute we will present is the *normalised average permutation distance* for the agents' value functions. Remember that an agent's value function is the subjective valuation of the set of goods. If we ignore the actual values, the value functions still yield a ranking of the goods for each agent. This ranking is one possible permutation of all rankings the goods can be in. Since we have m goods, there are $m!$ number of permutations. This attribute aims to give a metric to how different the agents' rankings of the goods are.

Here, we will discuss how to define a metric for the value of the agents' ranking similarity. We need to define a distance function between two permutations. The hypothesis is that the problem would act differently if the agents agree on the order of goods than if the agents disagree.

Distance Between Permutations

A function describing the distance between two permutations is needed to get the average permutation distance. The function should have increasing distances for increasing differences between permutations. A natural function for distances is the Euclidean distance (the distance between vectors) [35]. Representing the preferences as vectors can be accomplished by each value in the m-sized vector being the indices of the goods. However, this does not give a very intuitive representation of the distances. All the Euclidean distances between the preferences of three goods are shown in Table 4.1. It makes sense that the distance between "1,2,3" and "3,2,1" has the greatest distance, $\sqrt{8}$ since they are each other's inverse. However, we also see that "2,1,3" and "2,3,1" have this maximum value, but they only differ on the last two elements. Meanwhile, "1,2,3" and "1,3,2", which also only differ in the last two elements, have the shortest distance, $\sqrt{2}$. The function should not give different results based on which indices the goods get. The Euclidean distance gives the property that agents disagreeing on a pair of indices with higher differences yield greater distances than indices that are closer in value. However, we can see that the disagreement is of the same degree, and the distance should be equal, or in other words, independent of the index value itself. The Euclidean distance is, therefore, inherently flawed for this use case since it uses the indices of the goods.

	1,2,3	**1,3,2**	**2,1,3**	**2,3,1**	**3,1,2**	**3,2,1**
1,2,3	0	$\sqrt{2}$	$\sqrt{2}$	$\sqrt{6}$	$\sqrt{6}$	$\sqrt{8}$
1,3,2		0	$\sqrt{6}$	$\sqrt{2}$	$\sqrt{8}$	$\sqrt{6}$
2,1,3			0	$\sqrt{8}$	$\sqrt{2}$	$\sqrt{6}$
2,3,1				0	$\sqrt{6}$	$\sqrt{2}$
3,1,2					0	$\sqrt{2}$
3,2,1						0

Table 4.1: This table shows all the Euclidean distances between the different permutations of three goods. Only the upper half is filled in since the table is symmetric.

A distance function independent of the indices is needed. A function that shows promise is the *Kendall τ distance*, also known as the *inversion number* [10]. When talking about the inversion number, one often only refers to the distance from the *identity permutation*, which is the sorted order of the indices. Thus, talking about a single permutation having an inversion number implicitly means the distance between the permutation and the identity permutation. The Kendall τ distance always uses two permutations. These distances are however the same, the Kendall τ distance is just generalised. In the context of computer science, this distance also has a third name: *the bubble-sort distance* [10]. This name is given to it since the distance equals the number of swaps the bubble-sort algorithm has to do to turn one permutation into the other. The formal definition is the number of pairwise disagreements between two rankings in the permutations [10]. Example 1 shows how the Kendall τ distance between two permutations over three elements is found. Notice that by swapping two adjacent numbers, an adjacent transposition, the distance would change by one since only the preference between that pair has changed. This fixes the problems the

Euclidean distance had.

Example 1. *Given two preferences over three goods:* $P_1 = \{1, 2, 3\}$ *and* $P_2 = \{1, 3, 2\}$. *The preferences would be* $P_1 \rightarrow 1 > 2 > 3$ *and* $P_2 \rightarrow 1 > 3 > 2$. *The numbers are the indices of the goods, not actual values, therefore we see that agent* 2 *prefers good three to good two,* $3 > 2$. *The Kendall* τ *distance between the two agents would equal* 1 *since we count the number of pairwise disagreements. If we look at all the pairwise rankings in Table 4.2, we see that they only disagree on the ranking of the pair of goods 2 and 3.*

Pair	P_1	P_2	Disagree
1, 2	$1 > 2$	$1 > 2$	false
1, 3	$1 > 3$	$1 > 3$	false
2, 3	$2 > 3$	$3 > 2$	true

Table 4.2: Shows all pairwise rankings of two agents and whether they agree.

The Kendall τ distance will be the metric used for the distance between the permutation of two agents' value functions. Since we are not using the Euclidean distance, we can't have a Euclidean space. A more general space is a metric space, defined as a set of points and a distance function describing the distance between those points. In our context, the points will be the permutations from the value functions, and the distance function will be the Kendall τ distance. Four criteria exist for a metric space [32, p. 2].

- Zero property: $d(x, x) = 0$ Distance between the same point is zero.
- Positive property: $d(x, y) > 0$, $x \neq y$ The distance between any two distinct points is always positive.
- Symmetric property: $d(x, y) = d(y, x)$ The distance is independent of direction.
- Triangle inequality: $d(x, z) \leq d(x, y) + d(y, z)$ The shortest path is always the direct path.

The first property is trivial; all pairwise comparisons between two identical permutations would always agree; thus, the distance is zero. The positive constraint is also straightforward; since we count the number of disagreements, there is no way to get a negative number, and if there are no disagreements, they must be identical. The symmetric property is also true since order does not matter; we only care if they agree on a pair. The last property is the triangle inequality, which is a bit trickier to show. Imagine that one point defines the origin of the distances. All points of distance r from this origin would form a sphere around it with radius r, as illustrated in Figure 4.1. By definition, the Kendall τ distance is the minimum number of swaps of adjacent elements to get from one permutation to another. The distance from permutation A to the origin point, O and then to the other permutation B, can not be shorter by definition. If the distance $AO + OB$ were less than AB, it would contradict the definition of the Kendall τ distance for AB. Thus, using permutations and the Kendall τ distance is a valid metric space. From the triangle inequality, it can be derived

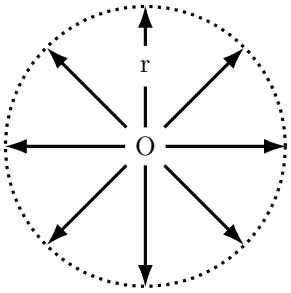


Figure 4.1: Illustration of metric space of the 2-dimensional intersection of a m-dimensional sphere. Showing the radius r from the ordered permutation, O, which is the space where all permutations with inversion number r lay.

that two permutations with one difference in their inversion numbers (Kendall τ distance from the origin point) can still have a distance between them equal to the sum of their inversion numbers. Figure 4.2, illustrates this. The horizontal line from the origin to $\frac{m(m-1)}{2}$ is the interval that defines the inversion number. The points A and B have an inversion number of r and $r+1$, respectively. On the inversion number interval, they differ in value by one and seem close together, but we can see that they are further apart in the metric space. If two permutations have the same inversion number, they might not have swapped any of the same elements in the permutation. Turing one of them into the other then requires undoing all the swaps in the first, turning it into the origin, and then doing the swaps in the second permutation.

So instead of thinking of the permutations being spaced out on the interval of the inversion number, they are spaced out on the surface of a $(m-1)$-dimensional hypersphere as depicted in Figures 4.1 and 4.2. Each distance from the origin forms a circle with a radius corresponding to a number on the inversion number interval. Imagine these circles forming latitude lines on the hypersphere. The furthest distance between two permutations with the same inversion number, r, is $2r$ since it could be two points on opposite sides of the hypersphere. Note that any permutation could be at the origin and that the latitude lines denote the set of all permutations reachable with r adjacent transpositions from the given origin.

The Kendall τ distances maximum distance does grow with the number of goods, m. Therefore, we normalise the attribute to keep the metric value independent of the number of goods. The maximum distance given m is [18]:

$$\frac{m(m-1)}{2}$$

Therefore, by dividing the distance between two permutations by the maximum, the distance is normalised and independent of the number of goods. The main reason for normalising the attribute is to eliminate a false correlation between

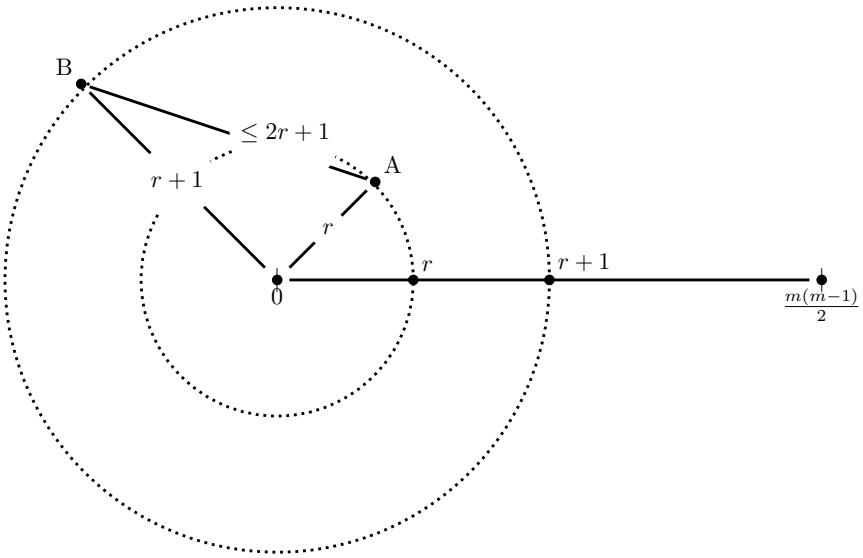


Figure 4.2: Illustration of metric space on the inversion number interval. Showing the relationship between the distance of two points A and B.

the runtime and the distance. Since there is an obvious, expected correlation between m and the runtime and m and the distance, there will be some correlation between the distance and the runtime. Normalising removes the correlation and gives a clearer picture of the actual correlation between the attribute and the runtime.

Maximum Average Permutation Distance

The attribute we are describing here is the normalised average permutation distance. We have described how we normalised the metric so as not to grow with the number of goods for the distance between pairs of permutations. However, as the number of agents increases, the maximum average distance between the permutations decreases. Here, we will describe and prove a formula for the maximum average permutation distance given the number of agents. This factor will be used to normalise the average permutation distance to make it independent of the number of agents. We always assume that the number of goods, the elements in the permutations, are greater than or equal to the number of agents. This is always the case for our algorithm; see Section 3.6.

First of we need to summarise a few properties of the metric space we have defined for this attribute:

- The distance between pairs of permutations has been normalised, so the distance between any pair of permutations, A and B, is maximum 1.
- The normalised Kendall τ distance between two points, A and B, is denoted here are the distance AB.

- The maximum distance from any permutation, A, is found by reversing the permutation [18]. Since each permutation has a unique respective permutation that yields the maximum distance, known as an *antipode* [10].
- A metric interval between two permutations A and B is defined as the set of permutations where C that $AC + CB = AB$ [10].
- Any adjacent transposition applied to a permutation A will either reduce the distance or increase the distance to any other permutation B by one. Since the preferences are binary, A and B must either agree or disagree. Thus, any preference in A must either be the same or the opposite for B; changing any preference in A adds a disagreement or removes one. Since the number of disagreements is the definition of the Kendall τ distance, this changes the distance between A and B by ± 1. Note the change in the distance by one must be normalised, so the change is $\frac{1}{\frac{m(m-1)}{2}} = \frac{2}{m(m-1)}$.

To find a general formula for the maximum average permutation distance, we begin with the trivial case of two permutations, we call these points A and B. Since AB is the only distance, maximising it yields the maximum average permutation distance. Having A and B as antipodal points makes AB maximum by definition thus the maximum average permutation distance for $n = 2$ is 1:

$$\frac{\text{Sum of distances}}{\text{\# pairs}} = \frac{AB}{C(n,2)} = \frac{1}{1} = 1$$

Before we continue, let's present a visual model to make it easier to follow the logic. Figure 4.3 shows the graph created from the permutations of four elements. Each vertex is a permutation, and the edges connect permutations which differ in one adjacent transposition, in our context, having a Kendall τ distance of one. Since we have a metric space, the length of the edge is not bound by geometry and the angles it has to other vertices. We can, therefore, imagine that we inflate the graph to form a sphere instead. The sphere will be an $(m-1)$-sphere, and the visual is more intuitive to illustrate with a 3-sphere, which we get when $m = 4$. We will refer to the visual model as the sphere. The resulting sphere is just a visual model; it bears no mathematical properties and will aid in explaining the logic. The scenario with two permutations can now be considered two points on the sphere. These points will be on opposite sides, one at each pole since they are antipodal.

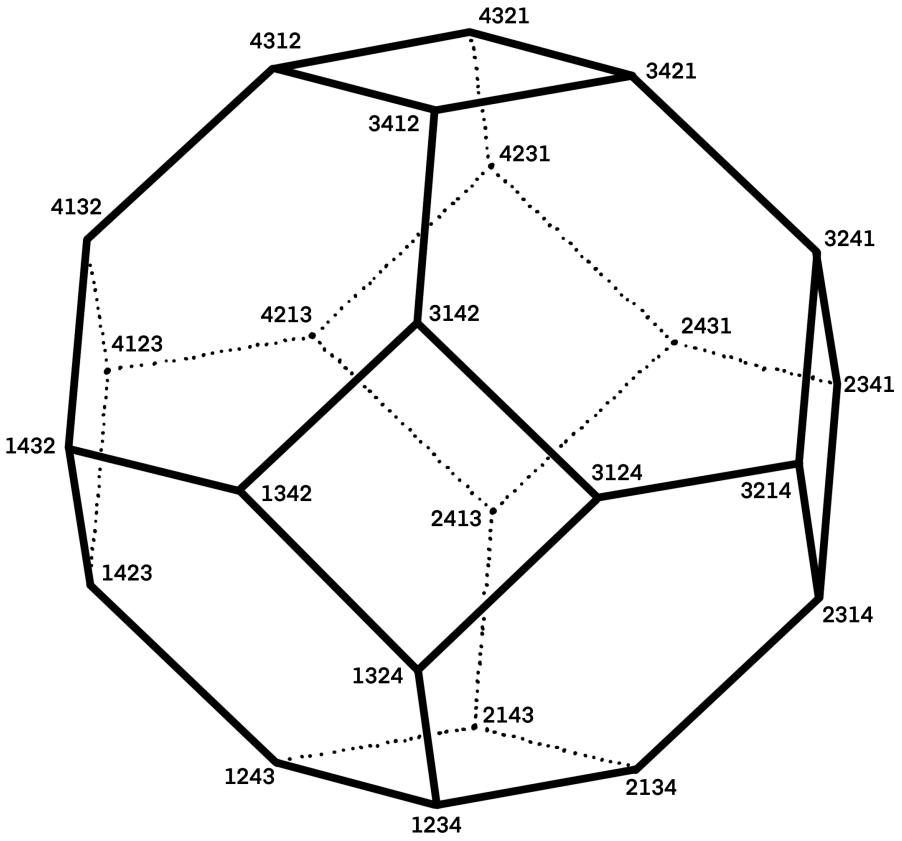


Figure 4.3: Illustration of the octahedron formed by the adjacent transposition graph of the set of permutations for $m = 4$. It is based on a figure from P. McCullagh's paper on "Models on Spheres and Models for Permutations" [27].

Now consider we have three points. We have points A and B on the north and south poles, respectively. Then, we add point C to the equator as a starting point. The formula for the maximum average distance is now:

$$\frac{AB + AC + BC}{3}$$

The sum of distances must be maximised to get the maximum out of this term. The distance AB is already maximised, so the rest of the sum, $AC + BC$, must be maximised. By the definition of the metric space, we have that $BC = CB$ (symmetric property) and that $AC + CB \geq AB$ (triangle inequality). The distance AB is known to be 1. Imagine now that we move C towards B, the moving C we denote as C'. This will take a distance $CB = \frac{x}{\frac{m(m-1)}{2}} = x'$, meaning C and B have x disagreements in preferences, where x' is x normalised. Since A and B are antipodal and only disagree, and preferences are binary, it means that for each disagreement C and B resolves, a disagreement is created for C and A. Thus, when C' and B are equal, we have $C'B = BB = 0$ (zero property). The original distance AC equals the distance from the new AC',

which equals AB minus the distance x' from the new disagreements created to achieve $B = C'$.

$$\begin{aligned} AC &= AB - x' \\ AC &= 1 - CB \\ AC + CB &= 1 \end{aligned}$$

The triangle inequality is tight, meaning that for any point C, the shortest paths between poles, A and B, can exist, including the point C. It makes intuitive sense that since A and B are in total disagreement, any change in preference for C would increase the disagreement with one and decrease it with the other. This also proves that the sum for $AC + BC$ is maximal and constant regardless of C when A and B are antipodal.

Lemma 4.2.1. *In the normalised metric space of the Kendall τ distance, any two permutations A and B that are antipodal yield a metric interval where $AC + CB = AB$ for all permutations C. This makes the metric space a antipodal metric space [10].*

We have found the maximum average permutation distance when A and B are antipodal to consider the cases where A and B are not. Imagine that A, B and C are random permutations and thus placed randomly on the sphere. Since poles are an arbitrary concept defined by any pair of antipodals, we could imagine A as the north pole. Moving B to the south pole, we create the scenario of A and B being poles. We must ensure we can reach this configuration without loss of value. Moving B south one step at a time must create a disagreement with A at each preference change. We increase AB by one (normalised) at each step. The distance AC remains constant. However, the distance BC changes, it can either increase or decrease arbitrarily based on where C is, if C is at the south pole, BC will always decrease by one at each step. So, moving from any random configuration of the three points to the configuration where A and B are antipodal is a non-decreasing transformation since the sum can only increase or stay the same. This means that any configuration of the points has an average permutation distance equal to or less than when A and B are antipodal. Thus, the average permutation distance with A and B as antipodals is the maximum average permutation distance.

Now, let's add a fourth point, D. The term for the average permutation distance is now:

$$\frac{AB + AC + AD + BC + BD + CD}{6}$$
$$\frac{AB + (AC + CB) + (AD + DB) + CD}{6}$$
$$\frac{AB + AB + AB + CD}{6}$$
$$\frac{3(AB) + CD}{6}$$

We can compact the equation by applying the same logic for the distance between point D and the antipodes A and B as when adding C. The only

distance left to maximise is CD. By reversing the order of C, making D its antipode, the distance is maximised to 1. To simplify even further, move C such that $C = A$. By keeping C and D as antipoles, D is equal to B. Moving C or D will not affect the average permutation distance since A and B are antipodal. The term is then derived as follows:

$$\frac{3(AB) + CD}{6} = \frac{3(AB) + AB}{6} = \frac{4(AB)}{6}$$
$$\frac{2(AB)}{3} = \frac{2}{3}$$

We get that the maximum average permutation distance for $n = 4$ equals $\frac{2}{3}$. Now, adding a fifth point would be the same scenario as when we added the third point, and so on. This means that the maximum average permutation distance is always found by creating the maximum number of antipode pairs. If there is an odd number of points, the last point's placement does not affect the average permutation distance.

To look at this more mathematically, we can expand the term for the average permutation distance into its components. Each distance between points, AB, is a sum of pairwise disagreements. The maximum average permutation distance is the sum of distances divided by the number of distances. Meaning that the term is a sum of sums, since addition is associative it means that the order of sums does not matter. Switching the order of the sums, we sum the sums of disagreements for each pair. Table 4.3 shows four agents' preferences over five goods. Such a table can be constructed for all instances. When finding the Kendall τ distance between A and B, we would sum up the disagreements in their columns, as we see they disagree on all pairs. When switching the order of the summations, we sum each row individually and then the sum of their sums. Each row shows the preferences for a unique pair for all the agents. The sum of each row is the number of disagreements on the pair i, j. The number of disagreements for each pair could be maximised individually to maximise the average permutation distance. Since they are independent, maximising each individually must maximise the sum of them together.

There are n agents, so a agents will prefer $i > j$ and d agents will prefer $i < j$, where $n = a + d$. The number of disagreements for a given pair equals a times d since each agent in a disagrees with each agent in d. This gives us the function $a \cdot d$ for the number of disagreements. We can find the optimal solution for a

and d as follows:

$$\begin{aligned} f(a) &= a \cdot d, \;\; n = a + d \\ f(a) &= a \cdot (n - a) \\ f'(a) &= n - 2a \\ f'(a) &= 0 \\ a &= \frac{n}{2} \\ n - d &= \frac{n}{2} \\ d &= \frac{n}{2} \\ a &= d \end{aligned}$$

Since we are dealing with positive integers, $a = d$ is not a valid solution when n is odd. The product is still greatest when a and d are as equal in value as possible, so rounding one up and the other down gives the maximum product. This means the maximum number of disagreements for a pair is found when the preferences are split as evenly as possible. Notice that this is precisely what the method of creating antipodals on the sphere does. This is also why each antipodal pair of points is independent of the others since, for each row in the table, the number of preferences over i, j in each row is balanced.

Pairs	A	B	C	D
1, 2	$>$	$<$	$>$	$<$
1, 3	$>$	$<$	$>$	$<$
1, 4	$>$	$<$	$>$	$<$
1, 5	$>$	$<$	$>$	$<$
2, 3	$>$	$<$	$>$	$<$
2, 3	$>$	$<$	$>$	$<$
3, 4	$>$	$<$	$>$	$<$
3, 5	$>$	$<$	$>$	$<$
4, 5	$>$	$<$	$>$	$<$

Table 4.3: Table showing four agents' preference over five goods.

To formalise this idea into a general function that gives the normalisation factor, the maximum average permutation distance, from the number of agents n, we generalise the formula for the maximum average distance. Let's begin with the case of n being even. Then we have two groups of $\frac{n}{2}$ points with opposite preferences. Each point has a zero distance to the $\frac{n}{2} - 1$ points on the same pole as it and a maximum distance of one to the $\frac{n}{2}$ antipode points. This is the same for all points; the total distance equals the number of unique opposite pairs. There is a pair between each point on one pole and all the points on the other pole for each point. This gives the equation:

$$\text{Sum of distances} = \frac{n}{2} \cdot \frac{n}{2}, \;\; n = 2k, \; k \in \mathbb{Z}^+$$

The number of distances equals the number of combinations of pairs: $C(n, 2)$.

This gives us the formula for the maximum average permutation distance:

$$\begin{aligned}\frac{\text{sum of distances}}{\text{number of distances}} &= \frac{\frac{n}{2}\cdot\frac{n}{2}}{C(n,2)} = \frac{\frac{n^2}{4}}{\frac{n(n-1)}{2}} \\ &= \frac{n}{2(n-1)}\end{aligned}$$

For n being odd, the same approach can be used, but there is one more point on one pole than the other. The point as split into groups of $\frac{n-1}{2}$ and $\frac{n-1}{2}+1$. So the sum of the distances becomes:

$$\text{Sum of distances} = \frac{n-1}{2}\cdot(\frac{n-1}{2}+1),\ \ n = 2k+1,\ k \in \mathbb{Z}^+$$

The maximum average permutation distance when n is odd is then:

$$\begin{aligned}\frac{\text{sum of distances}}{\text{number of distances}} &= \frac{\frac{n-1}{2}\cdot(\frac{n-1}{2}+1)}{C(n,2)} \\ &= \frac{\frac{n-1}{2}\cdot(\frac{n-1}{2}+1)}{n\cdot\frac{(n-1)}{2}} \\ &= \frac{\frac{n-1}{2}+1}{n} \\ &= \frac{\frac{n+1}{2}}{n} \\ &= \frac{n+1}{2n}\end{aligned}$$

Finally, this gives us the formula for the normalisation factor for the average permutation distance, which is the formula for the maximum average permutation distance:

$$f(n) = \begin{cases}\frac{n}{2(n-1)}, & \text{if } n = 2k,\ k \in \mathbb{N} \\ \frac{n+1}{2n}, & \text{if } n = 2k+1,\ k \in \mathbb{N}\end{cases}$$

Generating the Interval

Generating values in this interval is not an easy problem. Hand-picking permutations that give the desired normalised average permutation distance is not feasible. There are methods for sampling points that are statistically likely to give us a given value. The Mallows model is one such method [24]. It is used for similar problems and is suitable for this application [5]. The Mallows ϕ-model is defined as follows [23, 24]:

$$P(r) = P(r|\sigma,\phi) = \frac{1}{Z}\phi^{d(r,\sigma)},\ \ \phi = (0,1]$$

Here, σ is the standard permutation that we calculate our distance from, ϕ is the dispersion parameter, and Z is our normalisation value, defined as [7, 23]:

$$Z = \prod_{i=1}^{M-1}\sum_{j=0}^{i}\phi^j$$

The dispersion parameter, ϕ, can be used as the interval for generating permutations. By sampling, we can generate instances with an expected dispersion rate, giving a good spread of the permutations from the centre. One such method is the RIM sampling [23]. The algorithm is shown below in Algorithm 1.

Algorithm 1 RIM Sampling of Mallows [23]

Let $\sigma = \sigma_1 \cdots \sigma_m$ be the reference ranking and ϕ the dispersion.
Start with an empty ranking r.
for $i = 1..m$ **do**
    $d \leftarrow 1 + \phi + \cdots + \phi^{i-1}$
    for $j = i..1$ **do**
        $P \leftarrow \phi^{i-j}/d$
        if $j == 1$ **then**
            $P \leftarrow 1$ ▷ Last place to be put, must be placed there
        end if
        if $P \geq$ random $\in (0, 1]$ **then**
            Insert σ_i into r at rank position j
            break ▷ If probability did not occur continue to next rank
        end if
    end for
end for
return r

Notice that if the dispersion parameter, ϕ, is zero, the value P is one in the first iteration of the inner loop. This is because the value in the numerator of P is always to the power of zero in the first iteration, yielding one. The denominator also simplifies to one due to the zero in the base of the exponents.

$$P = \frac{\phi^{i-j}}{1 + \phi + \cdots + \phi^{i-1}} = \frac{0^0}{1 + 0 + \cdots + 0^{i-1}} = \frac{1}{1} = 1$$

The probability, P, is then a hundred per cent. This leads to the σ_i being placed into r at rank i, the same rank as it had in σ, with $\phi = 0$ no swaps will occur and $\sigma = r$.

On the other hand, when $\phi = 1$, it gets the highest spreading probability possible. P has a probability of $\frac{1}{i}$ to continue the inner loop and place σ_i at $i - 1$, the same for the next step and so on until only the first place is left, then it must be placed there.

$$P = \frac{\phi^{i-j}}{1 + \phi + \cdots + \phi^{i-1}} = \frac{1^0}{1 + 1 + \cdots + 1^{i-1}} = \frac{1}{i}$$

So the probability that the σ_k gets placed at r_1 is $\frac{1}{k^{k-1}}$.

4.2.4 Normalised Average Value Distance

Here, we will present our second novel attribute, *the normalised average value distance* for agents' value functions. Remember that the first novel attribute was the normalised average permutation distance of the agents' value functions, presented in Section 4.2.3. Imagine that two problem instances get different results in runtime, even with the same optimisations and the same permutations for the value functions. This could be attributed to the difference between the actual values of the goods. Therefore, we introduce an attribute that gives a metric for the difference in the agents' value functions. There needs to be a defined distance function between the agents' value functions.

Again, a natural choice would be the Euclidean distance using the value functions as m-dimensional vectors. However, this might introduce some problems. The distance between vectors is directly connected to the length of the vectors since vectors are in a Euclidean space [10]. We know that from Proposition 1, we can scale the value functions of agents without changing the solution to the problem; therefore, scaling should not change the distance between two value functions. Note that scaling changes the length of the vector.

By normalising the vectors (divide them by their l_2-norm), the scaling will be mitigated, and all vectors will be a distance of 1 from the origin. In this real coordinate m-space, $\mathbb{R}^m$, all the possible vectors of value functions would exist on an m-dimensional hypersphere. Due to the positive restrictions on the values, the hypersphere is bound to $\frac{1}{2^m}$ of the surface or, in other terms, limited to the $\mathbb{R}^m_{\geq 0}$ space. Since this forms a hypersphere, it would make more sense that the distance between two value functions is defined to travel through the domain, in other words, along the curve of the hypersphere and not in a direct line between them. Figure 4.4 illustrates the difference between the direct line, r, and the curved line, c. The arc c is longer than the line r. Defining the distance between points using the arches like c better encapsulates the distance between value functions. In mathematical terms, an m-sphere is considered an $(m-1)$-dimensional closed surface embedded in m-dimensional Euclidean space [1]. This further supports the notion that the distance should be limited to the closed surface.

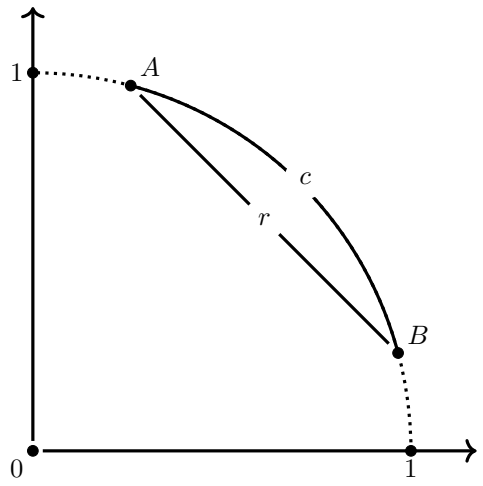


Figure 4.4: Illustration of the distance between two points on the unit hypersphere in $\mathbb{R}^m_{\geq 0}$, with $m = 2$.

The definition of the distance along the curve of the hypersphere is found by the following observation: Any two points on the surface of the hypersphere, together with the origin, form a two-dimensional plane, and the intersection between the plane and the hypersphere define a *great circle* [42]. This means that the problem of finding the distance between any two points on a hypersphere can be reduced to the problem of finding the distance between any two points on a circle with an equal radius to the hypersphere. Since the length of the vectors describing the value functions are normalised to one, this sets the radius to one. Therefore, we need to find the distance between two points on the *unit circle*. We only need the angle (in radians) between the vectors, θ, to know how much of the circle the arc covers.

$$
\begin{aligned}
d_{arc}(\vec{v}, \vec{p}) &= \text{Circles circumference} \cdot \text{Percent of circle} \\
d_{arc}(\vec{v}, \vec{p}) &= 2\pi r \cdot \frac{\theta}{2\pi}, \quad \theta = cos^{-1}\Big(\frac{\vec{v} \cdot \vec{p}}{|\vec{v}||\vec{p}|}\Big) \\
r &= |\vec{v}| = |\vec{p}| = 1, \ \text{ from normalisation} \\
d_{arc}(\vec{v}, \vec{p}) &= cos^{-1}(\vec{v} \cdot \vec{p})
\end{aligned}
$$

The distance is only defined by the angle between the vectors, as derived in the equations above. The resulting distance is known as the *cosine distance* or *great circle metric* [10]. The angle between the vectors of the value functions is a good metric for the distance since the angle between vectors is independent of the scale. When calculating the angle between two vectors, they are normalised, making the angle independent of scale. By being unaffected by scaling, the cosine distance respects Proposition 1. Also, we see in Figure 4.4, that $r \in [0, \sqrt{2}]$ and $c \in [0, \frac{\pi}{2}]$, since $\sqrt{2} < \frac{\pi}{2}$. The cosine distance yields a greater resolution for the interval to express the distance between value functions.

Figure 4.5 shows the relationship (with three goods) between the attribute of normalised average value function distance and the normalised average permutation distance (see Section 4.2.3). We see the hypersphere in $\mathbb{R}^m_{\geq 0}$, with $m = 3$. The different regions on the hypersphere represent different possible permutations the value functions can have. With three goods, there are six permutations ($3! = 6$) and six areas on the hypersphere.

Remember the case that motivated this attribute: Imagine two problem instances with the same permutations for the value functions but different values. We can now see that these must be in the same region on the hypersphere by looking at Figure 4.5. Imagine a close grouping of points on the hypersphere; there might be a difference in which optimisations work well based on where this group is. If the group is on a line separating two regions or in the middle of one region, it might give different results. These attributes aim to capture this relationship to get a clearer image of the problem instance. Do note that the Kendall τ distance between permutations would be how many jumps we need from one area to another. Jumps are only allowed across the division lines to the symmetrically mirrored area across the line. This makes sense as the lines show the area where two goods swap places in the preference, hence changing the permutation by one adjacent transposition.

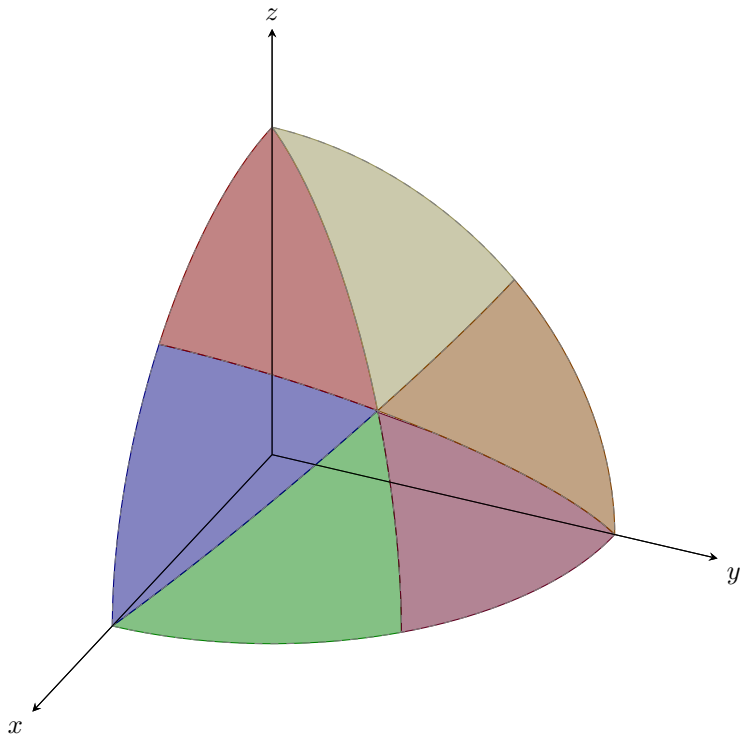


Figure 4.5: An illustration of where the areas of the different permutations of preferences are in the hypersphere in $\mathbb{R}^3_{\geq 0}$.

We must ensure this attribute is normalised and has no direct correlation to the problem size, n or m. In the use case for the branch-and-bound algorithm we developed, it turns out that this attribute is already normalised. To understand this, a few details of the algorithm must be repeated. The algorithm never solves a problem with more agents than goods. If an agent gets zero BSIMMS value, as it will when there are fewer goods than agents, the instance gets reduced to not having that agent. So the instance runs at least with $m = n$ and never with $m < n$. As long as the distance is normalised when $m \geq n$, this works fine for the algorithm. Since there are m dimensions and n points, creating n orthogonal vectors is always possible. The cosine between two orthogonal vectors is zero, thus a maximum distance of $\frac{\pi}{2}$ radians for all pairs of points. Placing all the points at the same place yields a distance of zero. Thus, the interval is defined by: $[0, \frac{\pi}{2}]$. This is independent of the problem size and does not need further normalisation.

Generating Intervals

Now that the interval for the attribute is defined, the next step is generating intervals of data. We will present the method used to generate problem instances and a second, more theoretical alternative method. The first method turned out to have an unfortunate bias in the sampling, which will be addressed.

The method used to generate value functions for a problem instance for a given average value distance was a naive novel approach shown in Algorithm 2. The algorithm was inspired by the RIM sampling, Algorithm 1. The algorithm starts by finding a random centre point on the hypersphere and then sample points nearby, controlled by the dispersion parameter, γ. It can generate invalid points, meaning it is outside of the domain $\mathbb{R}^m_{\geq 0}$; these points will be regenerated until they are valid. There is, therefore, a probability that the algorithm will run indefinitely since it can be stuck generating an invalid value indefinitely, but the probability is extremely low. In the actual implementation, the probability is even lower since each element in the random point is drawn individually, and an element is redrawn if it makes the point invalid. The points that become invalid are not just rounded off to the closest valid value because it would create a higher probability of the point being on the edge of the bounded space, $\mathbb{R}^m_{\geq 0}$.

Algorithm 2 Sampling points on $\mathbb{R}^m_{\geq 0}$ region of m-dimensional unit hypersphere

```
c ← RandomPoint()                          ▷ Random normalised point in ℝ^m_{≥0}.
P = {c}
while |P| ≠ n do                           ▷ n ≥ 2 is number of points to generate
    r ← RandomPoint(−1, 1) · γ             ▷ γ = [0, 1], is the dispersion parameter
    p ← c + r
    if p ∈ ℝ^m_{≥0} then
        P ← P ∪ {p}
    end if
end while
return P
```

Using Algorithm 2 and the dispersion parameter, one can generate clusters of value functions that, based on the dispersion parameter, are close together or far apart. The method was tested to see if it actually could generate the types of datasets wanted, which it appeared to. When analysing the data, it becomes evident that the distribution was not as first expected; this is discussed in Section 7.6.

The problem with the sampling method was that the distribution of the random points generated was expected to be uniform since it was drawn from a uniform distribution. However, when studying the method, it becomes clear that the distribution over the final placement of the nearby point is not uniform. It is also dependent on where the random centre point is placed. Figure 4.6 shows an illustration of the area where a generated point can be placed from an m-dimensional hypercube, with the centre point, c, in the middle and side lengths of 2γ. Every point in the hypercube is equally likely due to the uniform distribution, which was the idea. However, since the method takes the newly generated point and normalises it to the domain of the hypersphere, there is a non-uniform probability distribution for a given point on the hypersphere. The probability of any point, a, on the hypersphere generated by the sampling method depends on how many points in the hypercube are projected onto the hypersphere at point a. Defining the line l that goes through the origin and the point a, and finding the length of the part of l inside the hypercube yields the *frequency* of

the point a. Point a's frequency divided by the sum of the frequency for each point on the hypersphere yields the probability of the point a being generated. From the figure, it is easy to see that some points are much more likely to be generated than others, since their l lines through the hypercube are longer, thus we do not have a uniform distribution. Notice, too, that where the centre is placed also changes the probability of the points. Consider the centre point itself; when it is placed in the middle, as in Figure 4.6, it gets the maximum frequency by having its line go through the diagonal of the hypercube. If it had been closer to one of the axes, it would, at most, get l equal to 2γ. Also, the size of the hypercube is controlled by γ, and the centre probability would grow as the γ increases.

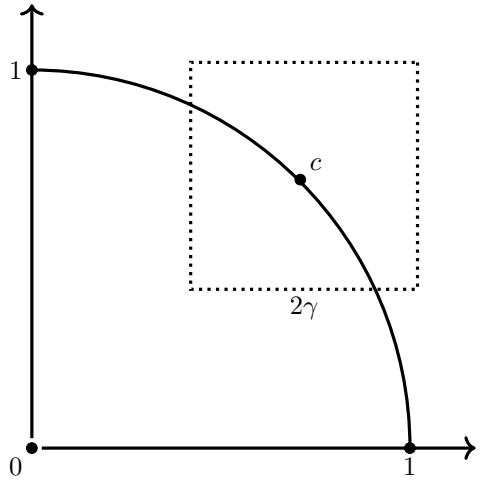


Figure 4.6: Illustration of how the area of the possible nearby points generated form a hypercube with the centre point in the middle and side lengths of 2γ.

An improvement would be to create a new vector point v equal to c and then apply two rotations. The first rotation to v would be the angle between it and the x-axis (first dimension), and the change would be uniformly drawn and controlled by γ. The second rotation would be applied afterwards to rotate it around the vector c. The second angle is also uniformly drawn. This would form a circle on the hypersphere around the centre point, with a uniform distribution. This sampling method was not implemented due to the late discovery of the bias in the first sampling method.

Another sampling method could take a different approach by generating a set of uniformly distributed points on the hypersphere and then sampling from this set. There are methods for uniformly spacing points on a unit hypersphere [36, 39]. Our proposed method here is purely theoretical and is included as an improvement to the current method due to the limitations discovered. The improved sampling method is shown in Algorithm 3. First, it generates a set of uniform points, S, in $\mathbb{R}^m_{\geq 0}$. The number of points is a factor β times more

than is needed, n. This set can be generated with a modified version of the algorithms proposed by Tashiro [39]. The sampling then gets a random centre point, c, and sorts S with respect to the cosine distance from c. Then, looping through the sorted set, with a probability of picking a point at each step. The probability decreases with the number of steps. If a point is picked, it is added to the resulting set of points, and the looping starts over. This continues until the size of the set of picked points is equal to the number of points needed, n. The decreasing probability is based on the RIM sampling, Algorithm 1. This theoretically gives the property that if the dispersion parameter, σ, is zero, all points are the same, and if it is one, all points have an equal chance of being picked at their turn.

Algorithm 3 Sampling points uniformly on $\mathbb{R}^m_{\geq 0}$ region of m-dimensional unit sphere.

Ensure: $\beta \geq 1$ ▷ The factor of pool size to draw points from
$S \leftarrow$ Generate uniform points so that the subset in $\mathbb{R}^m_{\geq 0}$ has size βn
$c \leftarrow RandomPoint()$ ▷ Random normalised point in $\mathbb{R}^m_{\geq 0}$.
$P = \{\}$
$S \leftarrow \text{Sort}(S, c)$ ▷ Sorts set with respect to distance from point c
$d \leftarrow 1 + \sigma + \cdots + \sigma^{|S|-1}$
while $|P| \neq n$ **do** ▷ $n \geq 2$ is number of points to generate
    for $j = 1..|S|$ **do**
        if $j == |S|$ **then** ▷ Last point left must be included
            $P \leftarrow \{P \cup s_j\}$
            break
        end if
        $p \leftarrow \sigma^{j-1}/d$
        if $p <$ random $\in (0, 1]$ **then**
            continue ▷ If probability did not occur continue to next point
        end if
        $P \leftarrow \{P \cup s_j\}$
        break
    end for
end while
return P

4.3 Implementation

In this section, we will discuss the details of the implementation of the program created to generate problem instances. The program can create datasets with intervals of given attributes and range constraints for the other attributes. It can create single instances with just range constraints or an arbitrarily long interval for one of the attributes. This was the vision for the tool; it makes it easy to script and generate vast amounts of different problem instances covering a multitude of scenarios. Others could use it as a standalone tool to create datasets for fair allocation problems and perhaps build upon it. Figure 4.7 shows the program's overall structure. Each step will be explained and discussed.

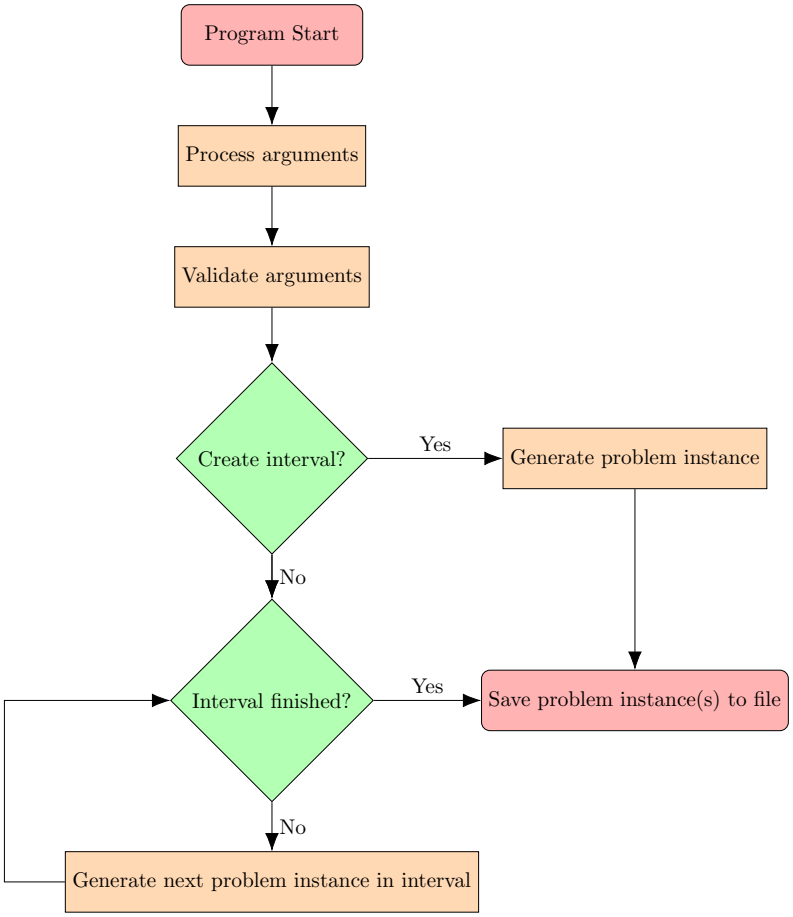


Figure 4.7: Flowchart showing the structure of the dataset generation tool.

Processing and validating arguments

The program works by supplying arguments for the attributes. Detailed documentation of the arguments can be found on the GitHub repository here. A short overview will be presented here. The arguments are as follows:

- -a : Number of agents
- -g : Number of goods
- -p : Average permutation distance
- -v : Average value distance
- -r : Ratio of goods to agents
- -b : Budget used per cent

- -d : Distribution for budgets, values or weights. Three distributions are available:
  - r : Random
  - n : Normal
  - u : Uniform
- -i : Sets which attribute (above) that will act as the interval
- -o : The output filename

All except the last three will have the form: *-X [from:to]* where the "from-to" part defines the range of the attribute. The last argument is just the file path to the output. The second to last take one of the other given attributes: *-i X[splits]*, where the range of that attribute will be the interval and will be split into "splits" number of instances. Lastly, the *-d* argument has the form: *-d [option][dist]...[option][dist]*. This means that it takes an option, which is either b (budget), v (value) or w (weight), then one of the distributions given above. An example of a run is:

./generate_dataset -o path/to/data.txt -a 2:4 -r 1:3 -p 0:100 -i p10 -d buvuwu

When first processing these arguments, there is some error handling to ensure that the format of the arguments is correct and that the values make sense. An appropriate error message is given to the user if an error occurs. This makes it easier to track faults when using the programs in scripts that will run many instances and make it easier for users. After the arguments have been parsed correctly, they are validated. This validation is a higher-order validation. There are some specific rules for each argument and some cross-compatible rules. To give some examples: for all arguments with ranges, the range is checked for validity, from a lower number to a higher (or equal) one. When the ratio argument is passed, a warning is presented to the user that if the number of goods is also passed, the number of goods will be ignored. The output file path is validated, and some extra features for creating directories, such that the file path is valid, take place, too. This happens at a different place in the code, but it was natural to state it here. Some combinations of arguments are invalid, such as setting the value distribution together with the average value distance argument. All the validation also gives the user feedback on exactly which constraint was broken.

Generating the Data

After the arguments are parsed and validated, the program generates the data. If the interval argument is passed, this is done a bit differently, but the actual generation is the same. The only difference is that it creates more instances, one file per instance, each with a given value from the interval supplied.

The generation function sets each attribute it can directly and calculates the others using the methods described in this chapter. The distributions are drawn from their respective distributions, scaled to the required size, and rounded off. Adding other attributes to this scheme should be simple. The operations are

only done if the argument is active. Simply adding and handling a new attribute would work as long as the constraints of the relationship between attributes are updated in the validation step, too.

Chapter 5

Experiments

In this chapter, we will look at and discuss the experiments. More specifically, we will look at the dataset generated, as discussed in Chapter 4. Further, we will discuss how the experiments have been managed. The design of the tools created to handle the big dataset and the data validation process will be explained.

5.1 The Dataset and Cluster

The dataset created covers as many data variants as possible. Intervals for all attributes were created using the general scientific principle of only changing one variable at a time. Each attribute has three size classes, small, medium, and big, and generally represents the number of goods and agents and expected runtime. Each interval is also created for all 27 combinations of distribution configurations; there are three distributions and three options based on distributions, thus $3^3 = 27$. In total, the dataset contains 3240 problem instances. Each optimisation configuration must be run on each problem instance to get a complete overview for the comparison. There are 403 different configurations of optimisations. This gives about 1.3 million cases to run. Some instances are expected to take vast amounts of time, so a ceiling has to be set at the runtime; in the end, this was set at half an hour. Running all the cases sequentially on a laptop would take almost 75 years. Therefore, a cluster is needed; the cluster used, IDUN Cluster [37], has a maximum capacity of 1800 nodes, which can run instances simultaneously (information about the hardware can be found in Appendix 9). The total running time would be reduced to about 15 days. However, from looking at the dataset and running some test runs, it was known that the smaller instances would finish in milliseconds, and many medium-sized instances were predicted not to need close to the full runtime either. As stated, the dataset was created so that most of it would finish, and therefore, only some of the biggest instances were expected to time out. The runtime was expected to be less than 10 days, which was well within the accepted period. The final runtime was, in the end, about 3 days, 5 days, including some extra work, like validation of the data. In the end, it was run twice, so the choice and consideration taken in this chapter were essential for completing two runs in the project's time span.

5.2 Execution and Management of Experiments

Running 1.3 million experiments is not done by hand; automation is required. Therefore, an automation system needed to be created. Here, we will look at how the problem of running and managing these experiments was solved.

5.2.1 The Run Plan

A development plan is needed before the management tool is even created. The overall run plan for validating our tool must be good. If severe faults exist, the entire experiment could be compromised and need to be redone. Since it is expected to take days to run the experiment in full, this is not acceptable. A rigorous development of the tool is needed. Therefore, a run plan was created to validate the system before running the entire experiment.

The first stage tested it on a dummy dataset. A very small dataset that did not run for long was used to check that the handling of completed jobs on the cluster was handled correctly. The second stage tested a small subset of the actual dataset. Then, in the third stage, the correctness of handling timed-out jobs was validated. This stage consisted of many sub-stages, one for each stage of the management system.

5.2.2 The Management System

Now, we will take a closer look at how the management system for running the experiment was designed and created. The naive approach of creating a job for each run needed would not work. Considering the overhead of scheduling jobs on the cluster, it would take too long and be unnecessary to create and queue 1.3 million individual jobs. With an overhead of only a second, it would be a total of 15 days of overhead, which seems excessive and wasteful. Also, there is a limit on how many jobs can be queued, as it turns out around 100k. This then posed the risk of the queue emptying overnight because the small instances would finish quickly. No mechanism on the cluster allowed a callback function on finished jobs, so the management tool would need to be run manually from time to time to keep the queue saturated with jobs so as not to lose effective running time. For this reason, many jobs had to be run in a batch together as one job. Then, if some time out, they must be rescheduled to try again with more time set aside per instance. It is important to note that instead of having an internal timer in the algorithm, the timeout of the cluster was used; a job on the cluster requests a time quota; if it runs over this time, it is stopped; this was used as time management.

How the System Works

The management is highly adapted to this particular use case and takes advantage of the dataset's structure. The dataset is structured in this format:

dataset/attribute/size/distribution/data_files[0-X].txt

The initial batched jobs will be created so each job runs all optimisations on all files in a directory. Each directory contains an interval of an attribute split into

five or ten files. This gave an initial number of 486 jobs. If the job ran to completion, all data would be accounted for, and all jobs would have been executed. The management system checks the job log files and removes everything that has been finished. If it finds a cancelled job, it moves to another directory for the timed-out jobs.

The management system then handles the timed-out jobs. It looks at each job's log file and finds all the problem instances that did not finish (or even start). There is a tiny bit of overhead here as a problem instance most likely was being worked on; it is rescheduled in full, so it has to do all the configurations of optimisations, too, even though it might have done some already. A new job is created for each of these problem instances, where the entire job is to run that single instance with all configurations of optimisations.

Almost the same thing happens for these jobs again. If they time out, we reschedule new jobs that only run the given problem instance and one configuration of optimisations. Here, the only overlap is the optimisation that was running when the job was stopped. Finally, if one of these jobs times out, a max time is set and added to the data table for that particular run.

The last problem to tackle was collecting the data from each run. Adding error handling and considering concurrency in writing results to the file seemed excessive. A simple solution was made: each instance was written to a unique file in a temporary directory for the results. Then, the management system only had to read all the files in the directory and add them to the data table sequentially, avoiding concurrency altogether.

When it came to running the experiments, a lot of practical obstacles were encountered. These were all edge cases in the management system and the branch-and-bound algorithm. The dataset worked like a fuzz test for the code. Many small fires had to be put out, leading to data loss and runtime waste. One problem discovered was the maximum size of the job queue on the cluster, as error messages began appearing close to queuing 100k jobs. This was fixed by setting an upper limit in the management system for how many jobs it would queue at once. The limit was initially sat at 50k to ensure it could run overnight without running out of jobs. The data loss was made up for with some post-run processing of the data table, which found all the jobs affected by edge cases, removed duplicates from overlapping runs, and found out which jobs were missing. This made it possible to restart all the missing jobs, and after a few iterations, a complete table of runtimes for the dataset was created. An example of how running the management tool looks can be seen in Figure 5.1. The second time the entire dataset was run, it ran without complications.

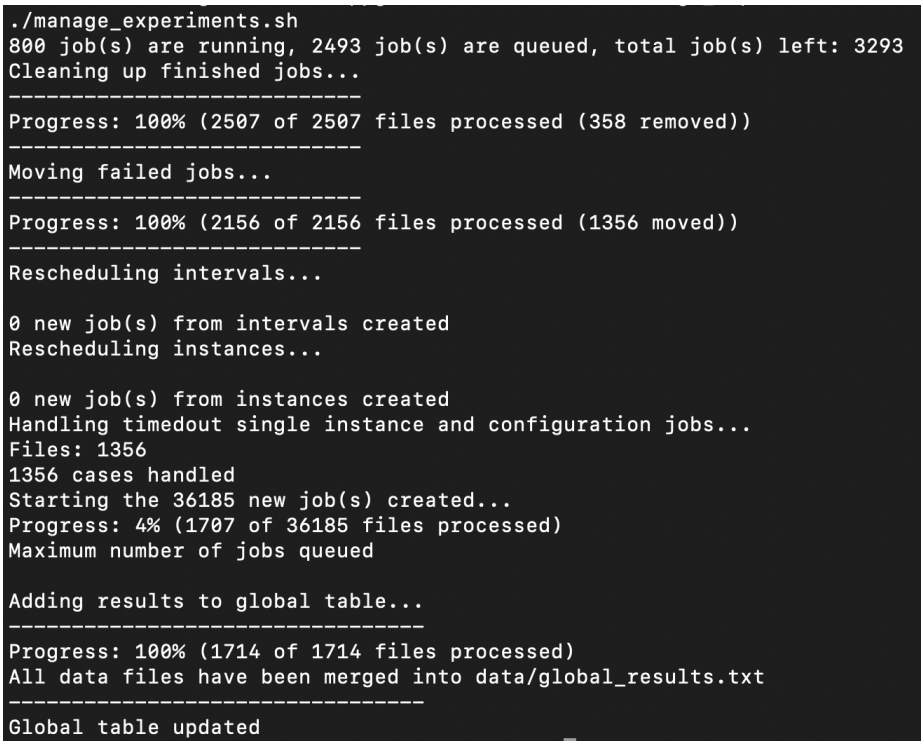


Figure 5.1: Screenshot of what a run of the management tool looks like.

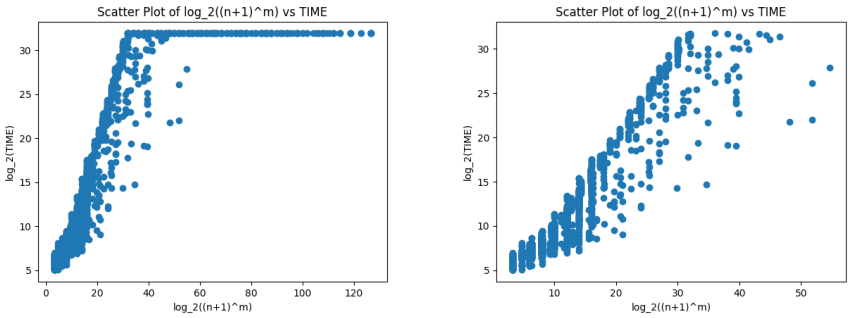
Chapter 6

Results

In this chapter, we will discuss and analyse the results of the experiments. The aim is to see if there is any relationship between the attributes we defined in Chapter 4 and the runtime of the experiments described in Chapter 5. We will present data on the relationship between attributes and optimisation using correlation, decision trees, and in-depth analysis.

6.1 Performance Analysis

A statistical predictive method must be established to discuss and compare the performance of different optimisations. A predictive model is expected to be most accurate for the naive approach since the naive approach is a brute force, making it more deterministic in running time. It will prune infeasible allocations where the budget constraints are violated, so it is not a hundred per cent deterministic in the number of nodes explored. Figure 6.1a shows the naive algorithm behaving as expected; as the problem size reaches a certain threshold, it reaches the time out and does so for increased problem sizes as well. Plotting the number of possible nodes against time shows how the problem's difficulty increases with the problem size. These values are in log_2 scale since they grow exponentially, and the log-based plot is easier to read.



(a) With timeouts.

(b) Without timeouts.

Figure 6.1: Scatter plot of $(n+1)^m$ vs. time, both log based. On the left, we have included the timeout values.

Figure 6.1a shows that the naive approach reaches the limit of what the naive approach can achieve quite quickly, at around $log_2((n+1)^m) > 40$, it is timing out quite consistently. Since it is unknown how close the timed-out instances were to solving the problem, they are ignored in further analysis. Figure 6.1b shows the timed-out instances removed.

We see that the problem does scale exponentially with the problem size. The model for predicting the time is quite simple; it is the number of nodes handled times how much time each node takes. There is also a small constant for the setup and preprocessing, which becomes negligible as the problem size increases. Therefore, the constant is excluded from the model to simplify it. First, the algorithm finds the BSIMMS value for each agent, then solves the α-BSIMMS problem with these values. Each of these has the same number of maximum explorable nodes:

$$\text{nodes} = (n+1)^m$$

Remember that the plus one is for allocating goods to the charity. Since we do this for each agent, then one last time for the actual problem, the branch-and-bound algorithm runs $n+1$ equally sized problems per instance. Adding all these together into an expression for the number of total nodes explored becomes:

$$\begin{aligned}\text{total nodes} &= (n+1)^m \cdot (n+1)\\ \text{total nodes} &= (n+1)^{m+1}\end{aligned}$$

To get a model for the runtime, we need to add a factor for how much time we spend at each node.

$$t = \alpha \cdot (n+1)^{\beta(m+1)}$$

The α term is the time spent per node. The β term allows the model to fit the data better; notice that it becomes the parameter describing the slope of the

regression line when we linearise the model. To fit the model to our data, we linearise it as such:

$$log_2(t) = log_2(\alpha) + \beta \cdot (m+1) \cdot log_2(n+1)$$
$$t' = a + bx, \;\; a = log_2(\alpha), b = \beta, x = (m+1) \cdot log_2(n+1)$$

Solving the linear regression for the last form yields a line fit to our data. A confidence interval of 95% is also added to the line to more clearly show its performance relative to others [22].

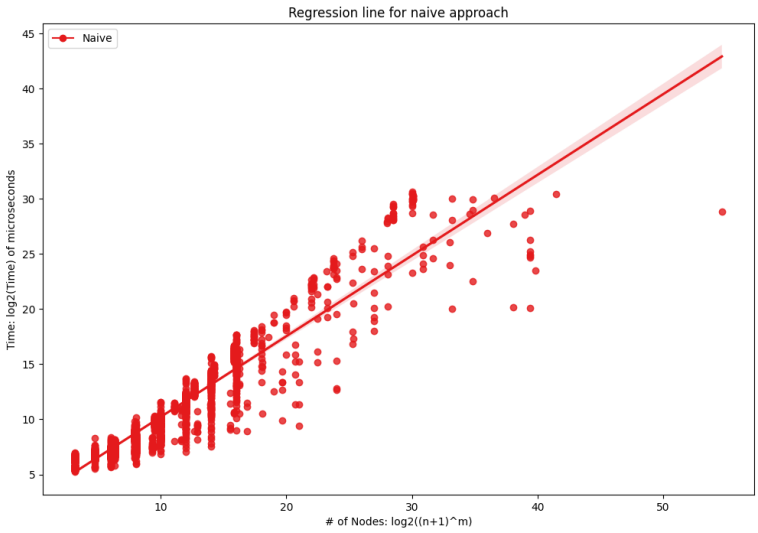


Figure 6.2: Regression model fitted to data for the naive approach.

Figure 6.2 shows the plotted line with the confidence interval. It has a slope < 1, which is not too surprising since the number of nodes we use as the parameter is expected to be an overestimate of the actual number. It is expected to be an overestimate since the budgets will (often) reduce the number of feasible nodes. A coefficient could be added to the model to try and capture this behaviour, but for the general runtime, this will suffice. The error in the number of nodes the algorithm can explore is the same for a problem instance regardless of the optimisations. The confidence interval gets wider with the problem size. This is due to the fewer runs of greater sizes that finished.

Plotting the data will be a recurring method of comparing the different optimisations in the chapter. Therefore, its structure has been emphasised so the comparisons are clearer. An important and note-worthy property of the confidence interval is that if two confidence intervals overlap, there is not a statistically significant difference between them [22]. In other words, if they do not overlap, one's performance is statistically significantly better than the others.

Figure 6.3 shows all the major optimisations, the MIP-solver and the naive ap-

proach plotted together. The major optimisations are non-naive, upper bound and bound-and-bound. The data points are non-overlapping, meaning that the optimisation configurations with both non-naive and upper bound are not included in the plot. We can see that the MIP-solver is the best choice for solving the problem when the problem is of a size greater than 12 on the log scale equivalent to about 4000 nodes. This could, for example, be a problem size of 3 agents and 8 goods. The MIP-solver outperforms the branch-and-bound algorithm by orders of magnitude, up to 2^{15} times faster, for bigger problem sizes, and it scales better in the size range used in the experiments.

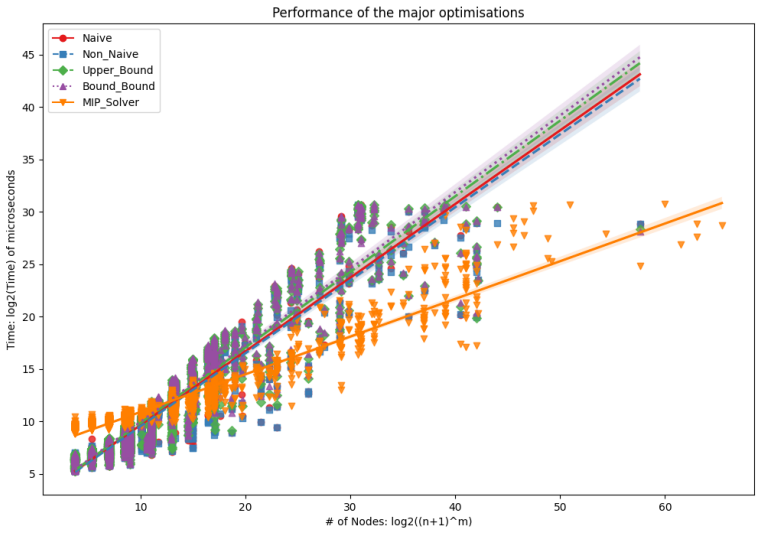


Figure 6.3: All major optimisations plotted together over the entire dataset.

Figure 6.4 shows the same plot as in Figure 6.3 but excludes the MIP-solver to easier see the difference between the remaining major optimisations. They look to be very close in performance. The confidence intervals overlap for all of them, so none are statistically better than the others. At the far right, the difference between the bottom line (non-naive) and the top line (bound and bound) is about two. With the log_2 scale, it means a difference of four times the runtime. So, there is quite a bit of variation in the runtimes. Also, addressing the fact that the naive approach is seemingly just as good as the optimised approach might be a bit surprising. However, as discussed in the implementation of the algorithm, it is not implemented to be hyper-optimised code-wise since the interesting results are in correlations between optimisations and attributes, which will be evident regardless. The naive keep up by not having any overhead, while the optimisations have a lot of overhead from implementation details that could be removed.

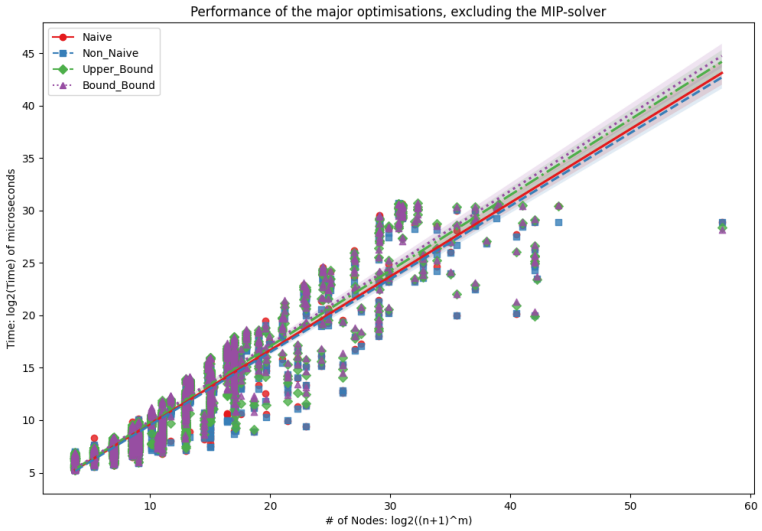
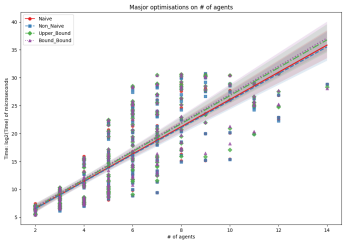
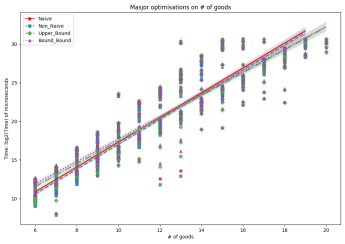


Figure 6.4: All major optimisations plotted together over the entire dataset, excluding the MIP-solver.

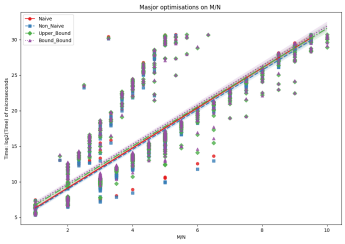
The plots so far have used the entire dataset; plotting for the sub-datasets for the respective attributes shows how the major optimisations behave differently. Now, the plots will have the attribute's value on the x-axis. Figure 6.5 shows the major optimisations plotted for the attributes. In all, except one case, we see that the major optimisations follow each other and overlap, meaning they are not significantly different. Only in the plot for the number of goods do we see a significant difference. For a small number of goods, the upper bound and bound-and-bound optimisations seem to have more overhead than the others, and at the most number of goods, they are almost significantly faster. This must be tested further to draw any conclusions, but from this data, it is projected that the non-naive and naive approaches will be outperformed. Notice that the plot for the average value distance shows that the generation of instances yielding values greater than 1 proved improbable with the generation method used.



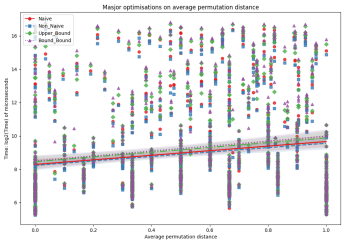
(a) Number of agents.



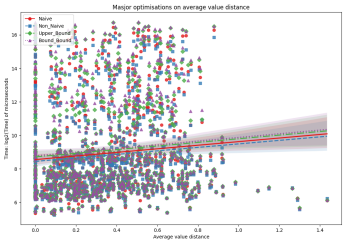
(b) Number of goods.



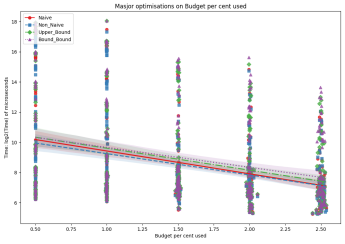
(c) Ratio of goods to agents.



(d) Average permutation distance.



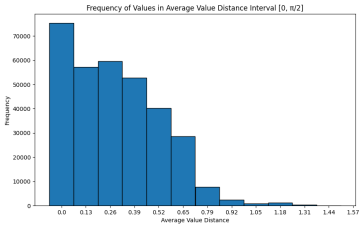
(e) Average value distance.



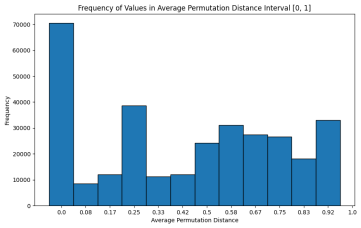
(f) Budget per cent used.

Figure 6.5: The performance of all major optimisations over the attributes on the attributes' respective sub-dataset.

To get a clearer picture of the distribution of the values for the average value distance sub-dataset, the frequency of the values was plotted, and the same was done for the average permutation distance to compare. The plots can be seen in Figure 6.6. We can see that the probability of generating an average value distance greater than 1 is almost zero. The probability of generating any given value in the average permutation distance is equal; there is a bias for the zero distance, but that value is guaranteed from the generation method; any other value will be generated with a probability, giving some wiggle room. The same is true for the zero value for the average value distance. Plotting performance of the major optimisations for the average value distance only in the interval from 0 to 1 showed no difference in the performance.



(a) Frequency of average value distance.



(b) Frequency of average permutation distance.

Figure 6.6: The frequency of the generated average value/permutation distances, plotted as histograms with 12 bars.

6.2 Correlation Matrices

We can look at the correlation matrix to see if there is any correlation between the attributes and the runtime. The correlation matrix shows the linear relationships between all the factors: attributes, runtime, and the number of nodes. By analysing the correlation matrices for different parts of the dataset, we aim to build an understanding of how the problem's predictive model behaves. It is important to look at these relationships to discover potential biases or unexpected behaviour.

In Table 6.1, we see the correlation matrix for the entire dataset. There are a few somewhat interesting numbers. Firstly, notice a stronger correlation between the number of goods and time, 0.94, than the number of nodes and time, 0.84. A slight negative correlation exists between the budget per cent used and time, -0.13. There is a big difference in the correlation factor for n's and m's effect on the time, 0.67 and 0.94, respectively.

	n	m	$\frac{m}{n}$	perm	val	budget %	time	# of nodes
n	1.00	0.68	-0.16	0.38	0.13	-0.05	0.67	0.91
m		1.00	0.58	0.33	0.24	-0.06	0.94	0.91
$\frac{m}{n}$			1.00	0.01	0.20	-0.03	0.53	0.20
avg perm dist				1.00	0.49	0.02	0.37	0.37
avg val dist					1.00	0.01	0.26	0.21
budget % used						1.00	-0.13	-0.06
time							1.00	0.84
# of nodes								1.00

Table 6.1: Correlation matrix for all runs of the algorithm.

The attributes for the average permutation distance and average value distances do not correlate much with the runtime. They have a small correlation with the runtime and a similar correction with the number of nodes. Due to the strong correlation between runtime and the number of nodes and the independence of the distance attributes to the number of nodes, they are not the causality for

the correlation to the runtime. The ratio of goods to agents has a medium-strong correlation to the runtime of 0.53 and only 0.20 with the number of nodes. Lastly, the correlation between the average permutation distance and the average value distance is 0.49.

The correlation matrix was the same, only differing by at most ± 0.01 on some entries when constructed for each major optimisation's runs individually. Let's look at some subsets of the dataset where the attributes are isolated to be changed in an interval while the others remain in small, tighter intervals, still excluding the MIP-solver.

	n	m	$\frac{m}{n}$	perm	val	budget %	time	# of nodes
n	1.00	0.60	0.11	0.23	0.01	-0.07	0.64	0.75
m		1.00	0.85	0.15	0.06	-0.04	0.94	0.98
$\frac{m}{n}$			1.00	0.04	0.09	0.00	0.71	0.71
avg perm dist				1.00	0.44	-0.02	0.16	0.19
avg val dist					1.00	-0.01	0.03	0.04
budget % used						1.00	-0.14	-0.05
time							1.00	0.96
# of nodes								1.00

Table 6.2: Correlation matrix for intervals of permutation distance value.

Table 6.2 shows no linear correlation for the average permutation distance to time or number of nodes, confirming our earlier claim. This means there is no linear correlation between the average permutation distance and the runtime.

	N	M	$\frac{M}{N}$	perm	val	budget %	time	# of nodes
n	1.00	0.55	0.05	0.36	0.12	0.04	0.58	0.73
m		1.00	0.84	0.24	0.09	0.00	0.94	0.97
$\frac{m}{n}$			1.00	0.08	0.05	-0.03	0.71	0.69
avg perm dist				1.00	0.61	0.02	0.24	0.29
avg val dist					1.00	0.02	0.09	0.10
budget % used						1.00	-0.08	0.02
time							1.00	0.95
# of nodes								1.00

Table 6.3: Correlation matrix for intervals of values function distance value.

Table 6.3 looks at the same thing for the average value distance. Notice that there is some correlation between the average value distance, the runtime, and the number of nodes. It is a very weak correlation, though. We also see that the average permutation distance in this matrix has an equally strong correlation. Since we are looking at two isolated datasets where the respective attribute is the changing factor, it makes sense that we see this. Note that due to the implementation of the data generation tool, the average permutation/value distance is not kept constant while the other is changed; see Section 4.3. There is, therefore, no evidence of any linear correlation between the average value distance

and the runtime.

	N	M	$\frac{M}{N}$	perm	val	budget %	time	# of nodes
N	1.00	0.57	0.08	0.45	0.09	-0.05	0.58	0.73
M		1.00	0.85	0.27	0.24	-0.02	0.89	0.97
$\frac{M}{N}$			1.00	0.05	0.25	0.01	0.67	0.71
avg perm dist				1.00	0.30	0.04	0.28	0.35
avg val dist					1.00	-0.06	0.23	0.22
budget % used						1.00	-0.31	-0.03
time							1.00	0.90
# of nodes								1.00

Table 6.4: Correlation matrix for intervals of budget per cent used value.

Lastly, we will look at the correlation matrix for the interval changes of the budget per cent used attribute, seen in Table 6.4. Here, we can confirm the relationship we expected. There is a decent negative correlation with runtime and no correlation with the number of nodes. There is a linear correlation between the number of abundant goods, in the form of total weight beyond the budgets' capacities, and the percentage of possible solutions being infeasible, yielding a lower runtime.

6.3 The Decision Tree

One way to classify which optimisation should be used based on the attributes is a decision tree classifier. They are binary trees and work by splitting a dataset into two subsets at each branch in the tree. The splits can be decided by different measures, such as maximising the entropy in the division of the data; there are also constraints on the size of the dataset subset, the depth of the tree and the minimum criteria for allowing a split. The dataset given to the classifier is two lists of data points, one with features and one with results. For this problem, the features are the attribute values for the problem instance solved, and the result is which optimisation was the fastest. The decision trees presented in the section are simplified; the complete ones can be found in Appendix 9.

The decision tree was generated with Scikit-learn [33]. The parameters for the tree are based on the values used by Mehrani et al. for their application of classifying the best algorithm for instances of their *Bin-Packing Problem with Minimum Colour Fragmentation* (BPPMCF) [28]. The parameters used in our model are:

- **Criterion**: Entropy (to measure the quality of splits)
- **Max Depth**: Equal to the number of attributes (6)
- **Minimum Samples per Leaf**: 5% of total samples
- **Minimum Impurity Decrease**: 0.1
- **Random Seed**: 65

Figure 6.8 shows the decision tree with the MIP-solver included. It suggests using the MIP-solver for all instances where $m > 6.5$. The average m in the dataset where $5 \leq m \leq 7$ gives us a log problem size equal to 12.8. This corresponds to the intersection between the regression line for the MIP-solver and the other major optimisations in Figure 6.3. The average runtime (excluding the MIP-solver) for the instances with $m \leq 6.5$ is $0.709ms$, the median is $0.194ms$, and the fastest and slowest is $0.036ms$ and $49.500ms$.

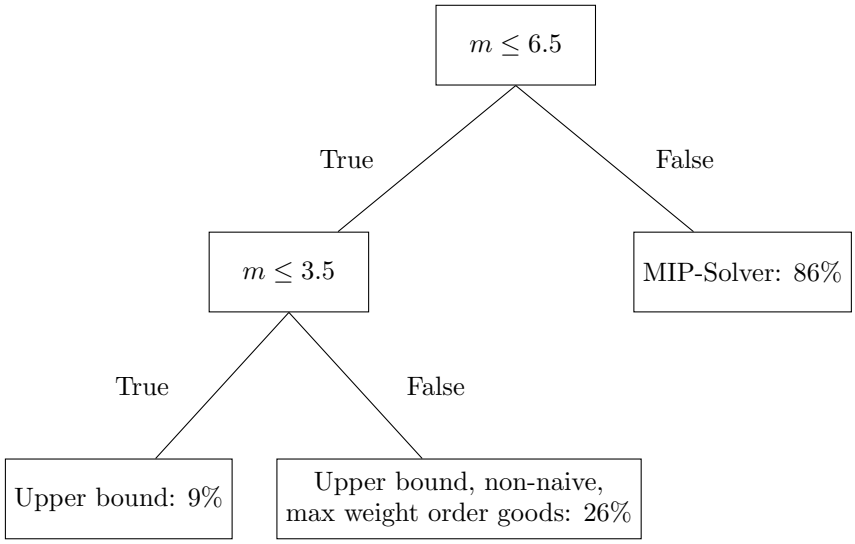


Figure 6.7: The decision tree included the MIP-solver, classification accuracy of 35%. The leaf nodes list which optimisations were dominant and what percentage of the fastest runs it has.

By ignoring the MIP-solver, we see how the rest of the optimisations fare against each other. Figure **??** shows the decision tree without the MIP-solver. This decision tree looks more interesting and has some more branching and diversity. However, the attribute for the number of goods, m, has a dominant entropy factor. This makes sense from the strong correlation it had with the runtime, as discussed earlier in Section 6.2. The non-naive optimisation by itself is the most dominant; it inherited about half the instances from the MIP-solver when we removed it from the tree, and it was fastest about 14% of the time.

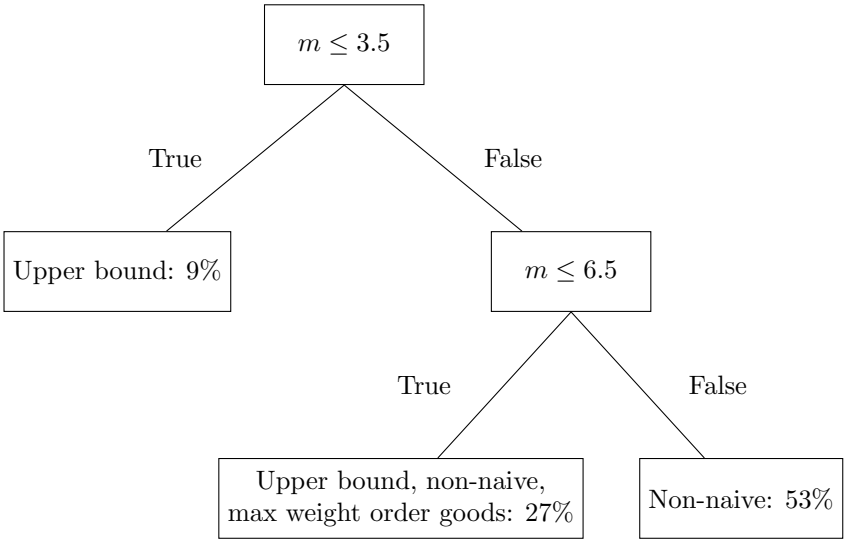


Figure 6.8: The decision tree excluding the MIP-solver, classification accuracy of 26%. The leaf nodes list which optimisations were dominant and what percentage of the fastest runs it has.

We have now looked at the decision trees generated from the entire dataset and will generate the decision trees for each sub-dataset. There are six of these, one for each attribute, and in the respective sub-dataset, that is the only attribute that changes value; see Section 4.3 for details. All but one sub-dataset showed the same decision tree with different values for m as the splitting criteria. The sub-dataset for the attribute of budget per cent used produced the decision tree in Figure 6.9. It has a split based on the budget per cent used attribute is less than or equal to 1.491.

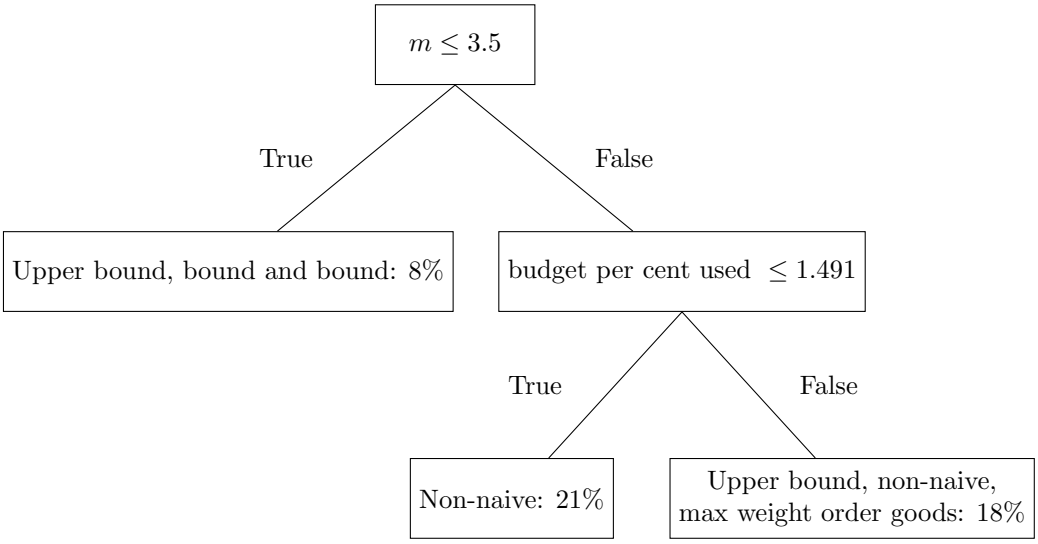
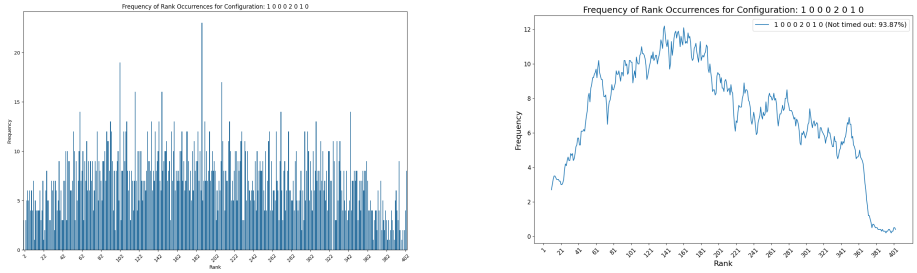


Figure 6.9: The decision tree excluding the MIP-solver on the sub-dataset for intervals of the budget per cent used attribute, the classification accuracy of 8%. The leaf nodes list which optimisations were dominant and what percentage of the fastest runs it has.

The decision tree approach has limitations. It only counts the number of times an optimisation has been the fastest, meaning that even if an optimisation was the second fastest in all runs, it would be ignored by the decision tree. Another approach is needed to see how the optimisations performed in general.

6.4 Frequency of Rankings

To see how well each optimisation performed compared to the others, a frequency plot of its rankings for each problem instance could be used. Figure 6.10a shows such a plot; however, it is a bit hard to read, as there are many spikes, and it is hard to see how many runs are in a given area, say the top 20. Since the value of plotting this is not from seeing exactly how many nth places the optimisation got but rather how concentrated it was in the given area of the rankings, we do a running average on the plot to smooth it out and make it easier to interpret, while still retaining the information. Figure 6.10b shows the result of smoothing the frequency plot with a running average, with a sliding window of 5. It is easier to see that the middle part is split in two, with the left part being a bit higher than the right, and that there is a drop-off at both ends but a steeper one on the right.

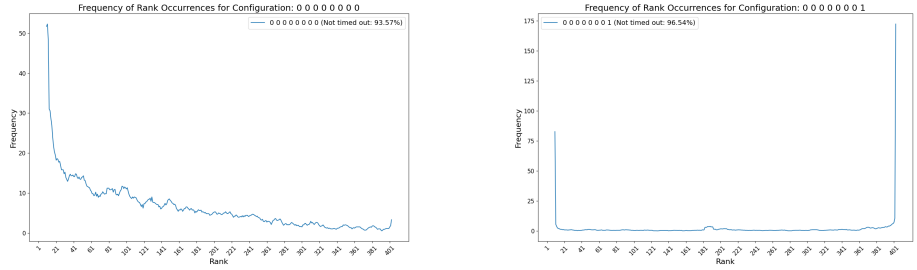


(a) Frequency plot

(b) Smoothed frequency plot

Figure 6.10: Examples of frequency plots showing the optimisation with upper bound and reverse Nash ordering for the agents.

There are 403 optimisations, but not all plots of them will be shown; many of them are less interesting, such as the one above; it does not tell us much when it is close to uniformly distributed. Figure 6.11 shows the plots for the naive approach and the MIP-solver. These showcase two types of interesting plots. The naive shows that even though it only was the fastest optimisation 27 times, it often was in the top 30 and decreasingly had fewer occurrences further back in the ranking. The MIP-solver has a very interesting case where it is either very good or awful, with a few occurrences in between in the transition area. We saw this in Figure 6.3 earlier. If any other optimisation has this type of split distribution at the extremes, e.g. for the sub-dataset for permutation distance, it could indicate a relationship between the attribute and the optimisation. This gives about 2800 plots given all sub-datasets for all optimisations. Here we will try to present the most interesting findings.

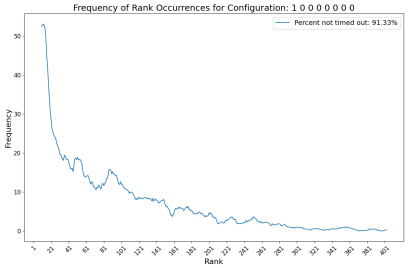


(a) Naive, no optimisations

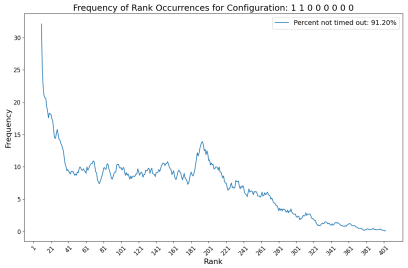
(b) MIP-solver

Figure 6.11: Smooth frequency plots for the naive approach and the MIP-solver.

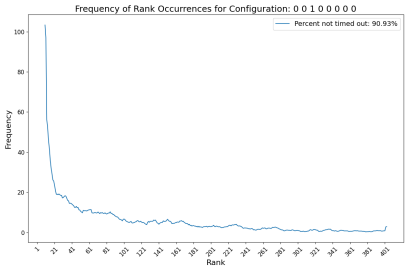
The introduction of the major optimisations shows how they fare against the naive approach. We can see them in Figure 6.12. The upper bound performs very similarly to the naive. The bound-and-bound has a major drop-off in performance. The non-naive optimisation has an even better performance. The case for all three being active performs somewhere in between the bound-and-bound and the upper bound performance.



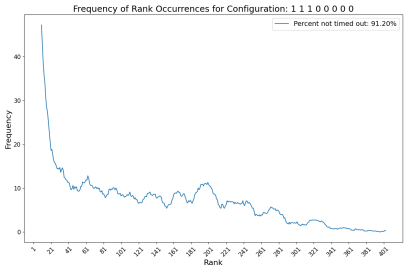
(a) Upper bound optimisation.



(b) Bound-and-bound optimisation.



(c) Non-naive optimisation.



(d) All major together.

Figure 6.12: Smooth frequency plots for the major optimisations.

All the results thus far have shown similar results, and not much separates them. However, the nature of the frequency plot tells us that if we add all of them together, we get equally many occurrences in each column. The plots we have seen have had many more occurrences in the top half with the best performances, meaning that the occurrences for the worst performances must be in some other plots. Many of the worst runtimes are found where the picking order for random agents is used. Every configuration of optimisations that uses the random agents picking order is similar; they all look like the sample ones shown in Figure 6.13.

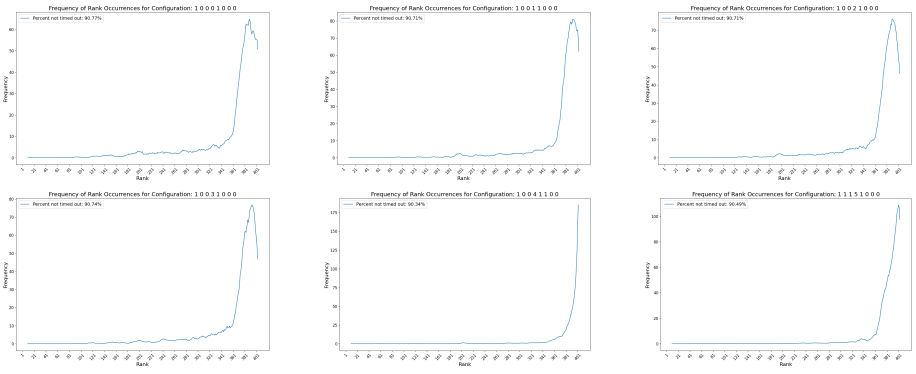
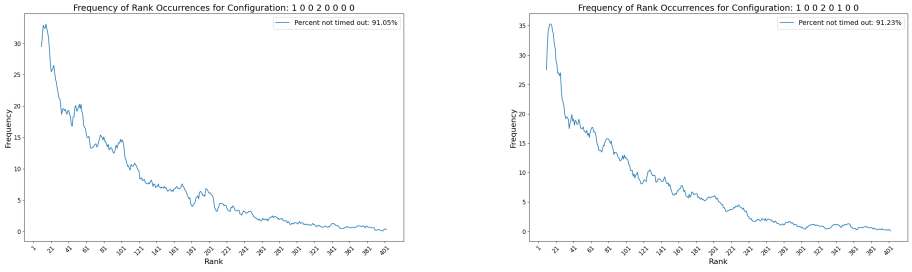


Figure 6.13: Six examples of how the frequency plots for the optimisations with the random agents picking order look.

The difference in the runtime comes from the picking orders chosen. Just like

the picking order of random agents was a bad one, other orders show interesting properties. The novel picking order of Nash ordering and value ordering for goods had no interesting difference in performance based on the pairings with other picking orders. However, they did show an interesting property: they both have very similar performance when reversed; an example is shown in Figure 6.14.

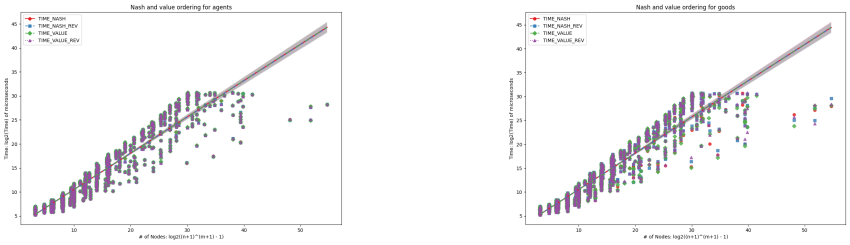


(a) Nash ordering.

(b) Reverse Nash ordering.

Figure 6.14: Nash orderings for goods with upper bound optimisation.

Since they behave the same way, seeing how they compare in performance is interesting. We have plotted this in Figure 6.15. As we can see, they are very similar in both cases, showing little to no difference in performance.

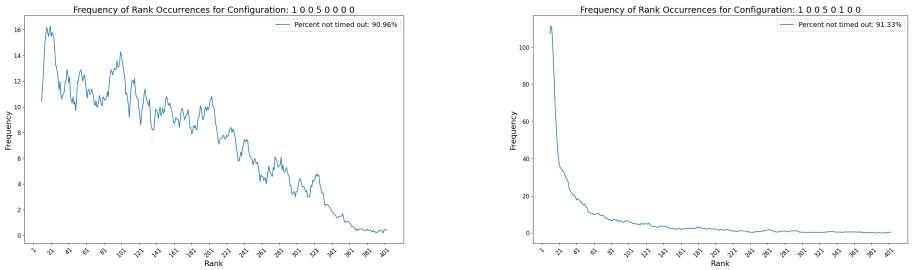


(a) Nash vs. value ordering for agents.

(b) Nash vs. value ordering for goods.

Figure 6.15: Performance of Nash and value ordering and their reverse for both goods and agents.

The goods picking order for profit shows an interesting property between the order and the reverse order. The reverse order is better for the profit, meaning that we explore the goods with the least profit first. Figure 6.16 shows that it still tends to perform on the better half, but it is a fairly even spread when we look at the difference in most occurrences to least, 16 to 1. The reverse ordering here looks very good, having a solid spike at the far left and then dropping off rapidly from about 20th place. This behaviour is consistent with all pairings when picking orders for the agents. The reverse order is always very good, and the non-reversed order is uniformly distributed; in some instances, it is pushed either towards better or worse performance.



(a) Profit ordering.

(b) Reverse profit ordering.

Figure 6.16: Profit orderings for goods with upper bound and non-naive optimisations.

Figure 6.17 shows the performance of the profit picking order and its reverse. They are significantly different for problem sizes up to about 2^{20} nodes. This is the same as in the frequency plots; the reverse order is better, but the normal order is not bad, so the difference is not too great.

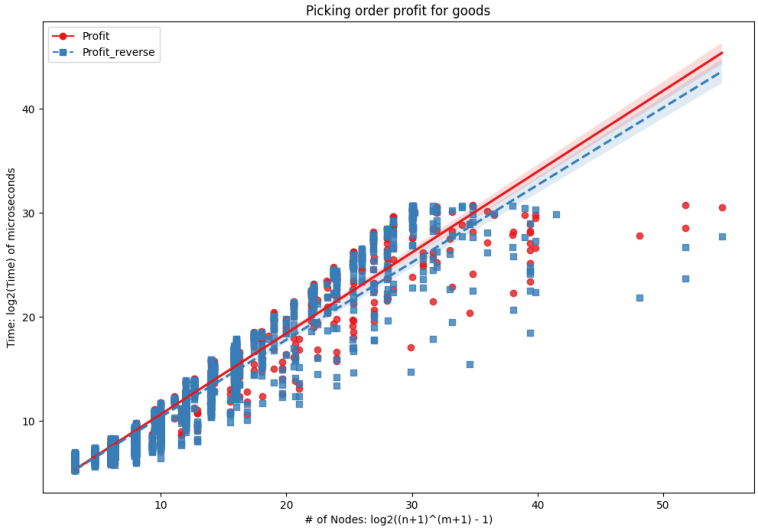
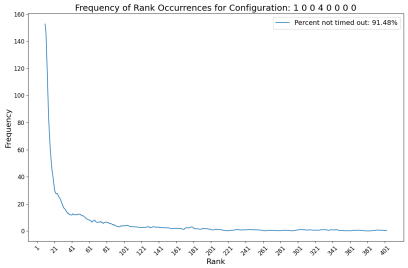
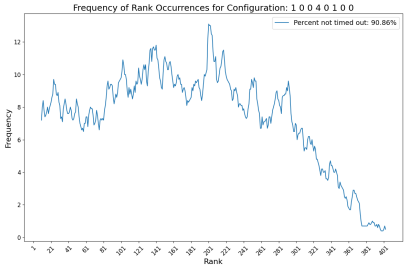


Figure 6.17: Difference between performance for profit as picking order for goods and its reverse.

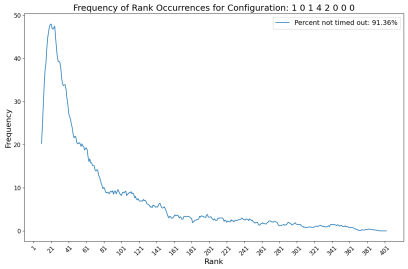
The reverse ordering for the weight picking order for agents also had an interesting property. It behaved by the reverse order yielding the opposite results. Searching the goods by max weight first is the most dominating optimisation, seen in Figure 6.20, it has the most consistently high values. In most cases, there is a falloff at the top rankings, implying that the picking order by weight is rarely the fastest but is very often the best of the rest.



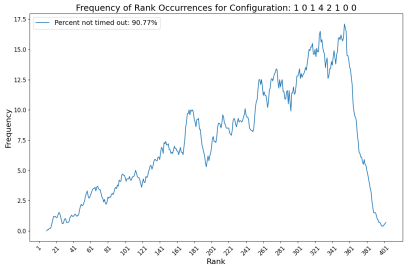
(a) Weight ordering.



(b) Reverse weight ordering.



(c) Weight ordering and Nash.



(d) Reverse weight ordering and Nash.

Figure 6.18: Weight orderings for goods combined with no order and Nash order for agents.

Figure 6.19 shows the weight order for goods and its reverse having a significant difference between their performance. This thus yields the same result as we saw in the frequency plots; since the reverse order is the opposite of the normal order, it would be expected that they differ significantly.

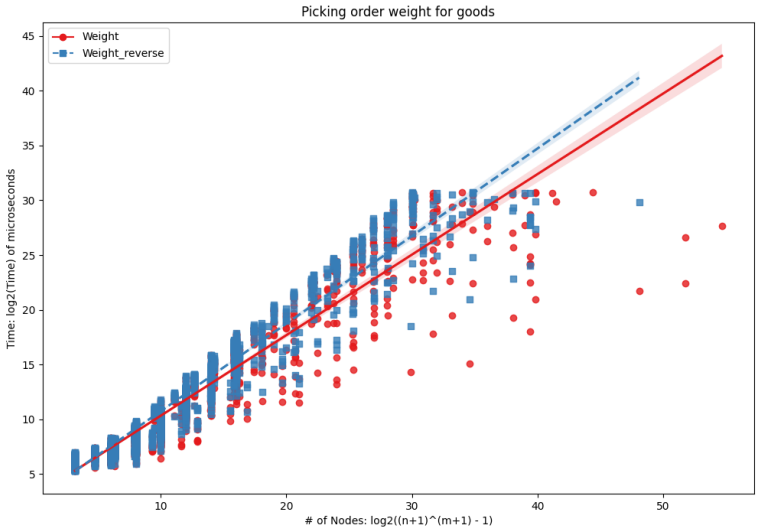
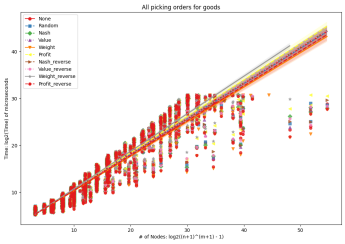
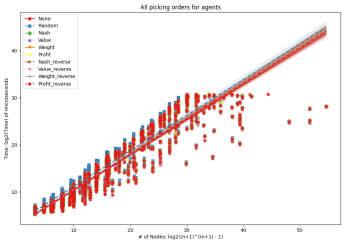


Figure 6.19: Performance of weight ordering for goods and its reverse.

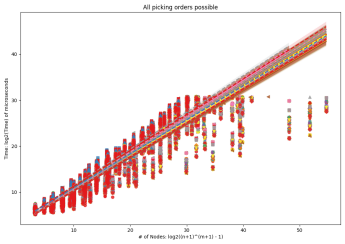
Figure 6.20a shows all the picking orders for goods in one plot. The reverse weight order is significantly worse than all others except for the profit order. All the rest are not significantly different from each other. Figure 6.20b shows all the picking orders for agents combined in one plot. None of the orders are significantly different, except that the random order is worse than all the others up until about 2^25 nodes. To ensure we have not overlooked any special combination of picking orders for goods and agents, we plotted all 100 possible combinations in Figure 6.20c. As we can see, no orders look too far off from the ones we already have shown, and most seem not to have a significant difference between them.



(a) Performance for all picking orders for goods.



(b) Performance for all picking orders for agents.



(c) All 100 possible combinations of picking orders.

Figure 6.20: Weight orderings for goods combined with no order and Nash order for agents.

We have now analysed the optimisations over the entire dataset. The dataset is composed of six different subsets containing intervals where only one attribute is changed. Plotting the same frequency plots for these subsets lets us see if there is a change in behaviour between the entire dataset and the isolated case where one attribute changes. There was no noticeable difference in the optimisations' rankings from the general dataset and the subsets, suggesting that the optimisations behave the same compared to each other independently of the values of the attributes.

Chapter 7

Discussion

In this thesis, we have studied MMS under budget constraints with the definition of BSIMMS. We developed a data generation tool to generate problem instances of BSIMMS with given values for its attributes. Then, we developed a branch-and-bound algorithm with optimisations to solve BSIMMS. The branch-and-bound algorithm with all its configurations of optimisations was run on the entire dataset. The main focus of the thesis was to study the connections between attributes of the problem instances of BSIMMS and the optimisations used in the branch-and-bound algorithm. We will review and assess the results presented in Chapter 6 and discuss their connection to the research questions presented in Chapter 1, looking at their practical implications, limitations, and the behaviour of the optimisations.

7.1 Linear Correlation

Multiple correlation matrices were presented in the results. This gives us insight into the linear relationship between the attributes and the runtime. Establishing if there are any correlations is important so that when looking at the relation to the optimisations, it can be taken into account. As expected from theory, a strong correlation between the runtime and the number of nodes was observed, supporting the validity of our predictive model. Again, the number of nodes had a strong correlation to n and m, which is expected as it is derived from those attributes. There is a slightly weaker correlation between runtime and the attributes n and m compared to the correlation between the number of nodes and the attributes n and m. This suggests that the budget constraints have affected the runtime. The budget constraints cause more infeasible nodes, so the number of feasible nodes decreases, lowering the runtime. It shows that the more excess goods in the form of weight, the less time it will take to find a solution. This is further supported by the slightly negative correlation between runtime and the budget per cent used attribute. If a better predictive model were to be created, using the attribute for budget per cent used in the model would be suggested.

The number of goods is a much better indicator of runtime than the number of agents. This makes sense from the structure of the dataset; for each value

of n, there can be multiple values of m, so while the runtime increases when m is increased, n remains the same and thus yields a wider range of runtimes for the same value of n. This comes from the fact that the average interval for n is about three times smaller than for m when generating most problem instances in the dataset.

The ratio of goods to agents shows a stronger correlation of 0.53 with runtime compared to only 0.20 with the number of nodes, illustrating an important relationship. The ratio gives useful information about the expected runtime; if the ratio is high, the problem is harder, and if the ratio is low, the problem is easier. This is intuitive since the runtime is based on the number of feasible nodes. When searching for a solution, it will probably not give all or even half the goods to one agent; it would not leave enough value for the rest. So, giving too many goods to one agent is quickly identified as infeasible. The ratio of goods to agents can be considered an attribute explaining the number of goods expected to be given to each agent, thus creating a proxy for the number of feasible nodes. A problem instance with 100 goods and 10 agents is really hard and has a ratio of 10, while a problem instance with 100 goods and 100 agents has a greater number of possible nodes but a much higher percentage of infeasible nodes in the search space tree. The ratio does not explain the number of nodes very well since when n increases, the ratio decreases, not increases, as it does for the expression for the number of nodes.

The average permutation and value distance attributes have some correlation to runtime and a similar correlation to the number of nodes. However, due to the strong correlation between runtime and the number of nodes, it is not likely that the average permutation or value distance attributes cause their correlation to the runtime.

7.2 Decision Tree Classifier

To find connections between attributes and optimisations, we created a decision tree. It made processing all the data into a compact conclusion easy. The work involved tweaking the parameters that guide and control the tree's structure to generate a useful decision tree. The parameters were tweaked and tested until a decision tree that was balanced and made good sense was generated. To obtain a balanced tree, we ensured that it was not over-optimised. Over-optimisation happens when the tree is allowed to create branches without a reasonable basis for the splits. A split should not be taken such that one leaf node states that optimisation X is the best if it only has one instance to support that claim. Increasing the minimum samples per leaf node will not happen. The same principle applies to the maximum depth of the tree; too many consecutive decisions are narrowed into a niche case of the attributes, giving an over-optimised and not a general solution. The tree reached was fairly stable, changing little to nothing when tweaking the parameters for smaller or larger amounts individually.

The decision tree in Figure 6.8 shows that the MIP-solver is the best in most cases. To see if there are any connections between the attributes and the optim-

isations in the branch-and-bound algorithm, we removed the MIP-solver and got the decision tree in Figure **??**. Unfortunately, only the attribute for the number of goods was used to split the tree. We saw the budget per cent attribute used as a criterion in the decision tree in Figure 6.9. This showed the possibility of other attributes than the number of goods containing information that could split the choice of optimisations. It was too small of a sub-dataset to confidently prove that this value applies in a general setting.

7.3 Attributes Effect on Optimisations' Effectiveness

The main aim of this thesis was to study if any attributes in a problem instance of BSIMMS could indicate which optimisations would work well for the given problem instance. This is under the assumption that the attributes could be found in polynomial time in a preprocessing step to the branch-and-bound algorithm. Through the correlation matrices, we saw no linear relationship between the attributes and the runtime that the model did not expect or predict. This means that the problem is generally equally hard regardless of the attributes' value. Only the attribute for the ratio of goods to agents had some interesting correlation to runtime independent of the number of nodes. This means that the ratio attribute is a general indicator that the runtime will increase with the ratio, which applies to all optimisations.

When analysing the frequency plots for the different optimisations, little to no difference was observed between the general case and the specific sub-dataset for each attribute. Since the optimisations behaved similarly with regard to each other, no change in any of the attributes caused a shift where one optimisation became better than another. There were, in general, very few cases of optimisations being statistically significantly better than others. It might be that the experiments must be made even bigger to push the optimisations' scalability further to see significant differences in the methods used.

7.4 Picking orders Behaviour

Here, we will discuss the different picking orders, how they performed relative to each other, and their effects and relation to the attributes. We have grouped them by behaviour.

7.4.1 Dynamic and Static Random Ordering Behaviour

The random picking order, both dynamic and static, turned out to have some challenges. In Section 3.4, they were presented to be expected as decent orderings. The static random, or naive random, picks the order at the start of the algorithm and does not change it. For ordering goods, this was a decent order since all the orders had to be picked once and did not have the benefit of adapting to the state of the search. This also meant that the dynamic random order was the same as the static one for the goods order. However, for the agents, it was a different result. The static was one of the better orders due

to its non-existent overhead. It was, however, not significantly better than any other picking order for the agents. The dynamic random, which we presented in Section 3.4 as an improvement on the static one, was the worst-performing order. It was significantly worse than all orders until the problem grew to about 2^{22}. From the frequency plots, we see it was quite clearly the worst-performing ordering in most cases, without any exceptions for different configurations of the other optimisations it was paired with. The fact that the dynamic random picking order is so decisively worse leads us to two useful insights. First, the picking order does matter; even though the other results have shown very even performance, that performance is still evidently better than just picking at random. Second, it illustrates that BSIMMS is a hard problem with statistically very few good solutions.

7.4.2 Nash and Value Ordering Behaviour

The picking order by value is intuitive and simple: give value to where there is the least value. The novel Nash picking order is, on the other hand, more intricate. Nash ordering ensures that the Nash score is increased gradually and evenly. These two orderings don't seem related at first glance, but their behaviour is almost identical. They are constructed almost identically when used as a picking order for the goods. The difference is that Nash takes the product of the agents' valuations for each good, while the value ordering takes the sum of the agents' valuations. They are both *utilitarian* approaches to finding a distribution [2, 29]. Thus, it makes sense that they similarly traverse the search space. When the orders are used for agents, the difference is that each agent evaluates their bundle and scores it for the value order. The Nash order aggregates all agents' evaluations for a given bundle together. Regardless, these orders yield almost identical performances, as seen in Figure 6.15.

The value picking order is an established and well-known order; however, the Nash order is novel and presented here based on the idea from the precursor project [13]. The fact that the Nash ordering is not significantly worse than any of the other picking orders shows that using it as a picking order might be useful. Although it did not outperform anything, it shows that it is competitive as an approach. This is, in particular, a positive result when considering the amount of overhead involved in finding the Nash order. It has a lot of potential improvement in the implementation of it. Studying how efficient it can be could be the focus of another project.

Neither the value nor Nash ordering showed any change in behaviour between the different sub-datasets for isolated attribute changes. This indicates no relationship between these orderings and any of the attributes.

7.4.3 Weight and Profit Ordering Behaviour

The two best-performing picking orders were the weight and profit orderings. Specifically, the reverse profit order, or least profit order, and the maximum weight order performed best. The least profit order seems unintuitive at first. As discussed in Section 3.4.6, we expected the maximum profit order to work well since it took both value and weight into account, allowing for a decision

to be taken on the value increase and the budget decrease. If there must be a charity, it would make more sense to distribute the most value first, leading to a higher maximum value since some goods must be donated to, regardless. As is evident from the experiments, the branch-and-bound algorithm does not behave like this. It turns out that the maximum weight ordering is the best. It is not statistically significantly better than all the other, only the minimum weight order. Still, from the frequency plots and analysis, it tends to be one of the better choices consistently. This shows us that the budget constraints are more relevant than the actual value of the goods. Trying to fit the most goods into the budgets works better than finding the most value. The budgets restrict the allocations of goods such that it, in general, is the allocation that packs the most goods into the budgets that also has the highest α-BSIMMS value. Since maximum weight is better than minimum weight, it makes sense that the least profit is better than the maximum profit. The maximum profit is inversely proportional to the maximum weight, meaning it is positively correlated to the minimum weight. It could be that the logic for the profit order still works but is less significant than the weight. This could explain why the maximum profit and least profit are not significantly different, but the maximum and minimum weight are. Further, maximum weight ordering was also shown to be the best order in the similar problem MOKP by Jørgen Steig [38]. The same behaviour was also found for the least profit. The orders used in that experiment were weight, profit and cardinality. So, it is not the same set of picking orders and a different problem, but both deal with budget constraints for distributing goods to a set of agents.

Neither of these orders showed any different behaviour between the different sub-datasets for the attributes. So, no relationship between these orders and the attributes was found.

7.5 Behaviour of Budgets

The introduction of budget constraints on MMS is a key part of the problem. The budgets are an aspect of the problem instance that is expected to hold a lot of information about the runtime. Since the tighter the budgets get, the fewer feasible nodes will exist, directly reducing the runtime. We developed the budget per cent used attribute to try and extract some information before even running the problem instance to see if we could predict how much of a reduction we could get. There is a weak correction between this attribute and the runtime as -0.13 over the entire dataset; see Table 6.1. This indicated, as expected, that the runtime decreased as the budgets were expected to be filled up more. This correlation was even stronger in the sub-dataset with intervals of the budget per cent used attribute changing, with -0.31, see Table 6.4. It also did not correlate with the problem size, which is strongly correlated with the runtime. This further supports that the attribute is independent of the problem size and indicates that the runtime is, in some regard, dependent on this attribute.

The relationship between the budget per cent used and the runtime is, in theory, not linear, so even the weaker correlation observed is a stronger indication than first thought. There is a lower part of the budget attribute where all the goods

will still fit until we reach a certain threshold when the need for a charity occurs. The change from this threshold and further is the more interesting part, as well as finding out where the threshold is. A more in-depth analysis to study this relationship would use a *interrupted time series* (ITS) analysis [4]. The decision tree in Figure 6.9 shows that a split has been made with the value of 1.491 for the budget per cent used attribute. This supports the theory of the threshold value since with a value greater than 1, there is a guaranteed charity [13]. In this given case, it was more efficient to use one optimisation than another to solve the instances separated by this value. This sub-dataset used to generate the decision tree is not too big since it is a subset of the entire dataset. To say that the threshold value we see here is general is a stretch; more importantly, it shows that this attribute holds information that could be useful when choosing optimisations.

The optimisations used in the branch-and-bound algorithm did not seem to be affected by the value for the attribute of budget per cent used. They behaved no differently than they did in other cases.

7.6 Behaviour of Average Permutation and Value Distance

We introduced two novel attributes to describe the problem instance of BSIMMS: the average permutation and value distance. The hypothesis was that these could describe something about the problem instance that might indicate which optimisation would work well. If the average value distance were zero, the agents' value functions would be the same, meaning the problem would be reduced to a simpler problem with objective value functions [13]. The problem is also expected to be easier if the average value distance is the maximum. The agents have completely different evaluations, meaning it should be obvious which good to give to which agent. This would at least yield an easy, greedy picking order by picking by maximum profit or value. From the experiments, we can see no such results in the data. The performance of each optimisation is almost identical, and all are non-significantly different. The same results can be seen for the average permutation distance as well.

The method for generating values for the average value distance attribute had a problem. It did not statistically generate average value distances greater than 1 for problem instances with more than two agents. When testing this generation method, many instances were created to see that all the numbers in the defined interval showed up. The method can produce all the numbers in the defined interval of 0 to $\frac{\pi}{2}$. The observed interval was from 0 to 1.44. A difference from the theoretical maximum of 1.57 was expected since the agents would most likely not have zero values for all except one good; they always valued the other goods a little bit. Therefore, this difference was accepted and expected. The average and median values of the attribute were both 0.46. It was expected that the probability of a given average value distance would shrink as it approached the theoretical maximum and as the problem size grew. Since there are more ways to obtain an average value distance of half of the max-

imum than the actual maximum, remember the hypersphere in Figure 4.5 and how each value function was a point on its surface. A cluster of points close together could easily be moved around and rotated while preserving the average value distance. While the theoretical maximum only exists while each point is in a different corner of the hypersphere restricted to the positive real domain, $\mathbb{R}^m_{\geq 0}$.

Observing the average and median value for distances lower than the midpoint, 0.79 was also accepted and expected. In retrospect, it is clear that the values should have been plotted, as in Figure 6.6, but the abrupt cutoff at around an average value distance of 1 was a bit surprising. It was expected that it would at least look more like an S-shaped curve, starting high, then dropping fast, then stabling out for the most part in the middle and then going low at the end. From the histogram in Figure 6.6, we see that there is quite a quick fall-off in probability at around 0.8 average value distance. The performance of the optimisations did not show any sign of differing results when looking at this subset of the interval. It is hard to say anything about the last part of the interval; this part of the experiment must be redone to draw any conclusions. However, from our evidence, it does not show any sign of affecting the optimisations.

7.7 MIP-Solver

We have utilised a MIP-solver for the upper bound optimisations and to solve the problem. The MIP-solver outperformed the branch-and-bound algorithm in almost all aspects of the experiments. It was only slower for very small problem instances that would take less than a second to solve with any method; for bigger problems, it was faster. From the range of experiments we tested, limiting the runtime to half an hour, the MIP-solver scaled better with the problems. The branch-and-bound algorithms code was not optimised, but this only affects the time per node, not the scalability of the method. In the performance plots in Figure 6.3, we see that the slope of the MIP-solver is far shallower than the branch-and-bound methods.

It was neither a goal nor expected that the branch-and-bound algorithm developed would compete with or beat the MIP-solver. The MIP-solver is a well-developed open-source tool with years of development behind it [30]. We did not see any relationship between the attributes and the MIP-solver's runtimes; it seems independent of the attributes. If minimising runtime is the goal, the MIP-solver is still the best choice.

7.8 UNs Sustainable Development Goals

The United Nations has 17 sustainable development goals [34]. Two of these goals are relevant for the work done in this thesis: goals 10 and 13. The first goal, 10, is *reduced inequalities*; our effort to make algorithms for fair allocation feasible for practical use in real-world scenarios is a direct contribution to a tool that could help solve real-world allocation problems. The second goal, 13, is *climate action*. Our efforts to make a more efficient algorithm so it becomes practical to use also reduces the runtime and, thus, the power usage of the

algorithm. There is a counter-argument to be made using Jevon's paradox [15], which states that the more efficiently a resource is used, the more the resource is used.

Chapter 8

Conclusion

In this thesis, we developed a tool for generating problem instances of BSIMMS with values for a set of attributes and a branch-and-bound algorithm to solve these instances. We also studied the relationship between these attributes and the optimisations used in the branch-and-bound algorithm and compared the algorithm's performance to an open-source MIP-solver.

Key findings from the experiments include the decisive outperformance of the MIP-solver for practical applications and the minimal impact the attributes of the problem instances had on the algorithm's optimisations. We introduced the Nash ordering for picking orders within the branch-and-bound algorithm, which proved competitive with established orderings. The maximum weight ordering was the most effective, underlining the significance the budgets effect have in the BSIMMS problem.

The developed tool for generating problem instances of BSIMMS directly addresses the lack of public datasets. Apart from the method for generating the average value distance attribute, the tool performs well. The two novel attributes of average value distance and average permutation distance did not show any relation to the optimisations here. The attributes are well-defined to be independent of the problem size, and there is a potential for future optimisations to exploit them.

Although no attributes were found to yield any information useful for choosing optimisations in a preprocessing step to the algorithm, groundwork for this approach has been made. The next step in research would be designing attributes and optimisations to work together.

Chapter 9

Further Work

Throughout the thesis, we have needed to make choices that limit the scope of the research; here, we will present some of the more interesting directions that could have and should be researched in future work.

Picking orders is a very central idea in the branch-and-bound algorithm. In this thesis, we chose to have the orders for the goods be static; exploring the more complex approach of having these be dynamic would open up lots of interesting research on the balance and relationship between the order for goods and agents. Possibly to maximise flexibility, a branch-and-bound algorithm that at each step chooses which picking order for both goods and agents to use based on the state of the search would give the most freedom to take advantage of the information available in the current state.

In this thesis, we introduced a couple of new attributes and a new picking order. Further designing attributes or optimisations that could work together would be interesting. For example, a picking order ensures that the traversal of the search tree benefits a specific optimisation. Ideas on this front that would work could allow the algorithm to be applied in practical settings. In Section 3.2.2, we presented an alternative to the upper bound function used in this thesis that there was no room to test; we also described how the lower bound would work with it.

The biggest design choice was to have the branch-and-bound algorithm search good-by-good and not bundle-by-bundle. Exploring the other option opens up a different set of optimisations and challenges that could change how the algorithm performs.

References

[1] Abraham Adrian Albert. *Solid analytic geometry*. Courier Dover Publications, 2016.

[2] Georgios Amanatidis et al. 'Fair division of indivisible goods: A survey'. 2022.

[3] Haris Aziz et al. 'Algorithmic fair allocation of indivisible items: A survey and new questions'. In: *ACM SIGecom Exchanges* 20.1 (2022), pp. 24–40.

[4] James Lopez Bernal, Steven Cummins and Antonio Gasparrini. 'Interrupted time series regression for the evaluation of public health interventions: a tutorial'. In: *International journal of epidemiology* 46.1 (2017), pp. 348–355.

[5] Niclas Boehmer et al. 'Guide to Numerical Experiments on Elections in Computational Social Choice'. In: *arXiv preprint arXiv:2402.11765* (2024).

[6] Sylvain Bouveret and Michel Lemaıtre. 'Characterizing conflicts in fair division of indivisible goods using a scale of criteria'. In: *Autonomous Agents and Multi-Agent Systems* 30.2 (2016), pp. 259–290.

[7] Róbert Busa-Fekete, Eyke Hüllermeier and Balázs Szörényi. 'Preference-based rank elicitation using statistical models: The case of mallows'. In: *International conference on machine learning*. PMLR. 2014, pp. 1071–1079.

[8] Ioannis Caragiannis et al. 'The unreasonable fairness of maximum Nash welfare'. In: *ACM Transactions on Economics and Computation (TEAC)* 7.3 (2019), pp. 1–32.

[9] Richard Cole and Vasilis Gkatzelis. 'Approximating the Nash social welfare with indivisible items'. In: *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*. 2015, pp. 371–380.

[10] Elena Deza et al. *Encyclopedia of distances*. Springer, 2009.

[11] Uriel Feige, Ariel Sapir and Laliv Tauber. 'A tight negative example for MMS fair allocations'. In: *International Conference on Web and Internet Economics*. Springer. 2021, pp. 355–372. ISBN: 978-3-030-94676-0.

[12] Jugal Garg, Peter McGlaughlin and Setareh Taki. 'Approximating maximin share allocations'. In: *Open access series in informatics* 69 (2019), pp. 3–4.

[13] Ola Kristoffer Hoff. 'Defining MMS Under Budget Constraints, and Adapting a Branch-and-Bound-Algorithm To Solve It'. 2023. URL: `unarchived`.

[14] Halvard Hummel. 'Maximin Shares in Hereditary Set Systems'. In: *arXiv preprint arXiv:2404.11582* (2024).

[15] H Winefrid Jevons and H Stanley Jevons. 'William Stanley Jevons'. In: *Econometrica, Journal of the Econometric Society* (1934), pp. 225–237.

[16] Nocedal Jorge and J Wright Stephen. *Numerical optimization*. Spinger, 2006. ISBN: 978-0387-30303-1.

[17] Hans Kellerer, Ulrich Pferschy and David Pisinger. 'Multidimensional Knapsack Problems'. In: *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 235–283. ISBN: 978-3-540-24777-7. DOI: 10.1007/978-3-540-24777-7_9. URL: https://doi.org/10.1007/978-3-540-24777-7_9.

[18] Maurice G Kendall. 'A new measure of rank correlation'. In: *Biometrika* 30.1-2 (1938), pp. 81–93.

[19] Rucha Kulkarni. 'Fair and Efficient Division of Indivisibles'. In: *Economics* (2022).

[20] Ailsa H. Land and Alison G. Doig. 'An Automatic Method for Solving Discrete Programming Problems'. In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Ed. by Michael Jünger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 105–132. ISBN: 978-3-540-68279-0. DOI: 10.1007/978-3-540-68279-0_5. URL: https://doi.org/10.1007/978-3-540-68279-0_5.

[21] Weidong Li and Bin Deng. 'The budgeted maximin share allocation problem'. Sept. 2023. URL: https://doi.org/10.21203/rs.3.rs-3333303/v1.

[22] Gunnar G. Løvskog. *Statistikk for universiteter og høgskoler*. 4th ed. Universitetsforlaget, 2018. ISBN: 978-82-15-03104-0.

[23] Tyler Lu and Craig Boutilier. 'Effective sampling and learning for mallows models with pairwise-preference data.' In: *J. Mach. Learn. Res.* 15.1 (2014), pp. 3783–3829.

[24] Colin L Mallows. 'Non-null ranking models. I'. In: *Biometrika* 44.1/2 (1957), pp. 114–130.

[25] Silvano Martello and Paolo Toth. 'A bound and bound algorithm for the zero-one multiple knapsack problem'. In: *Discrete Applied Mathematics* 3.4 (1981), pp. 275–288.

[26] Nicholas Mattei and Simon Rey. *PrefLib: A Library for Preferences*. 2024. URL: https://preflib.simonrey.fr (visited on 29/06/2024).

[27] Peter McCullagh. 'Models on spheres and models for permutations'. In: *Probability models and statistical analyses for ranking data*. Springer, 1993, pp. 278–283.

[28] Saharnaz Mehrani, Carlos Cardonha and David Bergman. 'Models and algorithms for the bin-packing problem with minimum color fragmentation'. In: *INFORMS Journal on Computing* 34.2 (2022), pp. 1070–1085.

[29] Roger B Myerson. 'Utilitarianism, egalitarianism, and the timing effect in social choice problems'. In: *Econometrica: Journal of the Econometric Society* (1981), pp. 883–897.

[30] Peter Notebaert and Kjell Eikland. *Introduction to lp_solve 5.5.2.11*. 2020. URL: https://lpsolve.sourceforge.net/5.5/ (visited on 25/06/2024).

[31] NTNU. *Idun*. 2024. URL: https://www.hpc.ntnu.no/idun/ (visited on 21/05/2024).

[32] Mícheál O'Searcoid. *Metric spaces*. Springer Science & Business Media, 2006.

[33] F. Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[34] United Nations Development Programme. *Sustainable Development Goals Integration*. 2024. URL: https://www.undp.org/sustainable-development-goals (visited on 29/06/2024).

[35] Timothy Sauer. *Numerical Analysis Second Edition*. Person Education Limited, 2014. ISBN: 978-1-292-02358-8.

[36] Masaaki Sibuya. 'A method for generating uniformly distributed points on N-dimensional spheres'. In: *Annals of the Institute of Statistical Mathematics* 14.1 (1962), pp. 81–85.

[37] Magnus Själander et al. *EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure*. 2019. arXiv: 1912.05848 [cs.DC].

[38] Jørgen Steig. 'Multiple Opinionated Knapsacks'. MA thesis. NTNU, 2023. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/3093957.

[39] Yoshihiro Tashiro. 'On methods for generating uniform random points on the surface of a sphere'. In: *Annals of the Institute of Statistical Mathematics* 29 (1977), pp. 295–300.

[40] Peter Van der Linden. *Expert C programming: deep C secrets*. Prentice Hall Professional, 1994.

[41] Vignesh Viswanathan and Yair Zick. 'Yankee Swap: A Fast and Simple Fair Allocation Mechanism for Matroid Rank Valuations'. In: *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*. AAMAS '23. London, United Kingdom: International Foundation for Autonomous Agents and Multiagent Systems, 2023, pp. 179–187. ISBN: 9781450394321.

[42] Eric W Weisstein. 'Great circle'. In: *https://mathworld. wolfram. com/* (2002).

Appendix

Hardware on the IDUN Cluster

Table below shows the hardware on the IDUN Cluster [31, 37].

Compute nodes

Node	Amount	Type	#CPUs	Processor	#Cores	RAM[GB]	#GPUs	GPU type
idun-02-[01-48]	48	Dell C6520	2	Intel Xeon Gold 6348	56	256		
idun-03-[01-36]	36	Dell C6520	2	Intel Xeon Gold 6348	56	256		
idun-04-[01-03]	3	Dell DSS8440	2	Intel Xeon Gold 6148	40	754	8	8xV100 32GB
idun-04-[04-07]	4	Dell DSS8440	2	Intel Xeon Gold 6248R	48	1509	10	NVIDIA A100 40GB
idun-04-[08-10]	3	Dell DSS8440	2	Intel Xeon Gold 6248R	48	1509	10	NVIDIA A100 80GB
idun-05-[01-06]	6	Dell XE8545	2	AMD EPYC 75F3	64	1007	4	NVIDIA A100 80GB
idun-05-07	1	Dell XE8545	2	AMD EPYC 7543	64	2015	4	NVIDIA A100 80GB
idun-05-[08-12]	5	Dell R740	2	Intel Xeon Gold 6132	28	754	2	NVIDIA V100 16GB
idun-06-[01-11]	11	Dell R730	2	Intel Xeon E5-2650 v4	24	128	2	NVIDIA P100 16GB
idun-06-[12-14]	3	Dell R730	2	Intel Xeon E5-2650 v4	24	128	2	NVIDIA V100 16GB
idun-06-[15-18]	4	Dell R730	2	Intel Xeon E5-2695 v4	36	128	2	NVIDIA A100 40GB
idun-07-[01-12]	12	Dell C6420	2	Intel Xeon Gold 6132	28	192		
idun-07-[13-20]	2	Dell C6420	2	Intel Xeon Gold 6242	32	768		
idun-07-[21-22]	2	Dell C6420	2	Intel Xeon Gold 6242	32	768		
idun-07-[23-32]	10	Dell C6420	2	Intel Xeon Gold 6252	48	192		
idun-08-[01-04]	4	Dell XE9680	2	Intel Xeon Platinum 8470	52	2014	8	NVIDIA H100 80GB HBM3
idun-10-[01-19]	19	Dell R630	2	Intel Xeon E5-2630 v4	20	128		
idun-10-[21-22]	2	Dell R740	2	Gold 6226 CPU @ 2.70GHz	24	376	2	FPGA (Xilinx)

Complete decision trees

This section contains the complete decision trees generated, the ones in Chapter 6 are simplified from these.

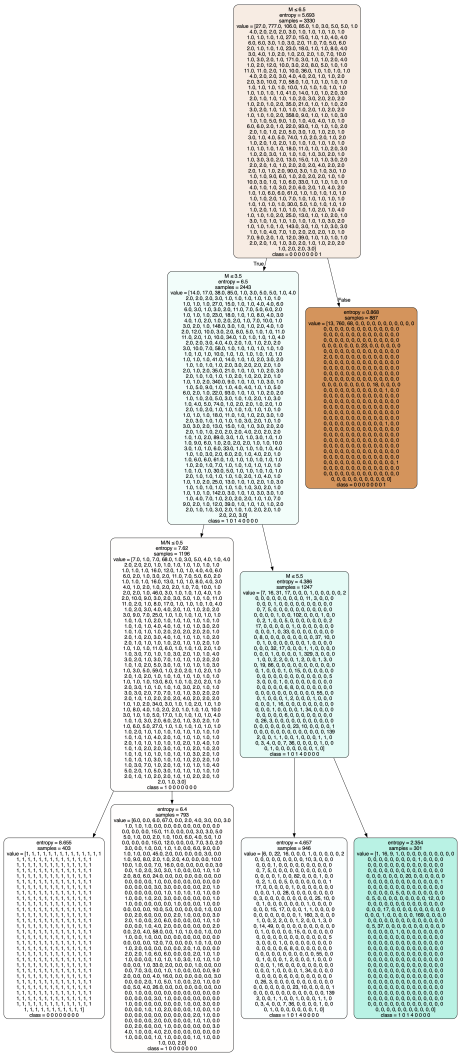


Figure 1: The decision tree included the MIP-solver, classification accuracy of 35%.

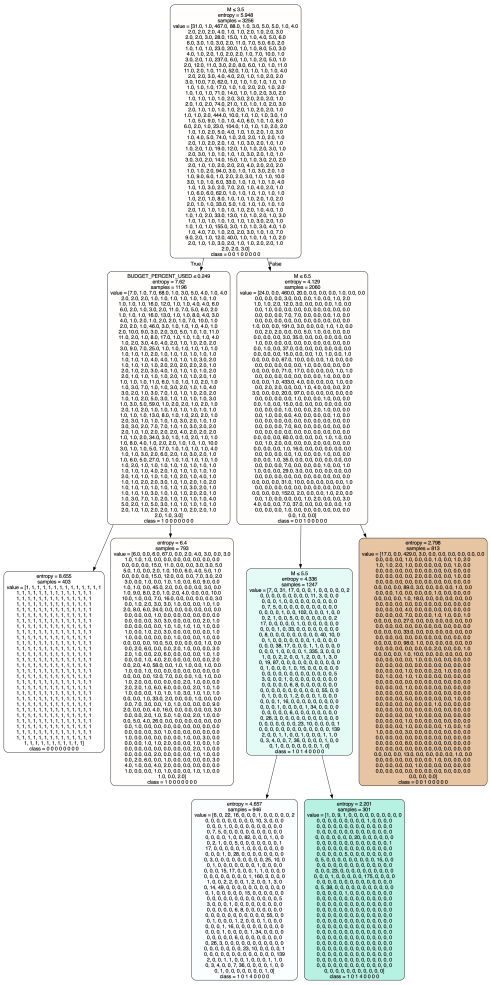


Figure 2: The decision tree included the MIP-solver, classification accuracy of 26%.

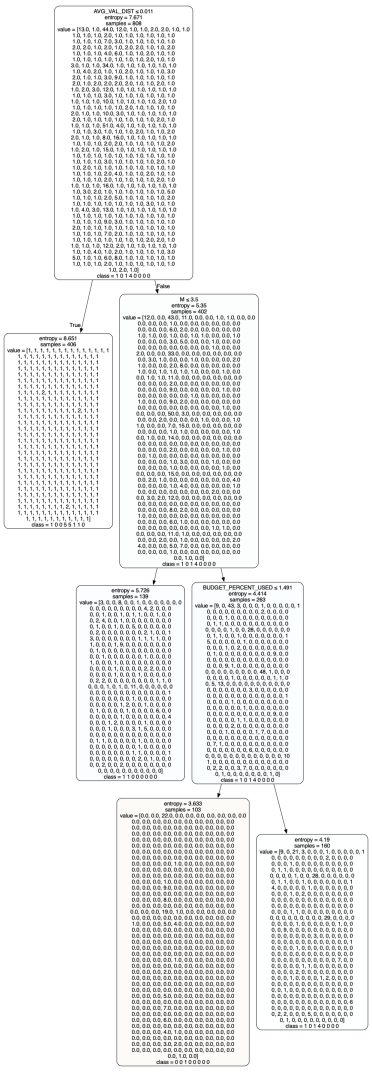


Figure 3: The decision tree for the budget per cent used sub-dataset, classification accuracy of 8%.