Fair Allocation of Indivisible Goods: Experiments and Implementation

Dimitrios Konstantinou

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

With different notions of fairness and corresponding algorithms designed to achieve them, the next phase to continue the research is the implementation and testing of such algorithms.

In this study, we focus on Envy-Freeness (EF1, EFR, EFX) and Max Nash Welfare (MNW) concerning indivisible goods, for which we have identified relevant algorithms and their respective fairness notions guarantees.

These algorithms are implemented in Python with the intention of making them publicly available. Throughout the implementation phase, we encountered instances where certain algorithms lacked critical components in their descriptions, while others provided no information whatsoever. By addressing these limitations and devising efficient solutions, we have produced a collection of algorithms, many of which had not been implemented before. Notably, we encountered significant implementation feasibility challenges with 0.73-EFR, as well as a potential flaw in the current state-of-the-art 1.45-MNW.

Subsequently, our implementations underwent strategic testing utilising both real-life (Spliddit) data and synthetic cases with varying numbers of goods and agents under different types of valuations (Random, Identical, Ordered, Binary, and Bivalued). In our testing suite, we examined the execution time of these algorithms as well as their fairness characteristics. Through our results, we make informed recommendations for future researchers, to assist in appropriate algorithm selection based on fairness requirements while also accounting for execution time. Furthermore, our investigations yielded new fairness observations from some algorithms that if theoretically proven, could lead to the formulation of new theorems as to the fairness guarantees of the algorithms.

As a concluding remark, we also outline key considerations for future algorithm designers and encourage the research community to perform further analysis using targeted test instances for possible further insights.

We summarise our key findings in Tables 5.1 and 5.2.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Dimitrios Konstantinou*)

Acknowledgements

I would like to take some time to thank my supervisor Dr. Aris Filos-Ratsikas for his guidance, feedback and support. It was a true pleasure having the opportunity to work with him. I have learned a lot from his mentorship and I am thankful for his insights and for introducing me to this topic.

A final thank you goes to my (non-tech savvy) family for their immense support, and for having the pleasure of listening to my frustrations about unexpected and hard challenges faced during this research such as decimal precision.

Table of Contents

Chapter 1

Introduction

Fair division (or fair allocation) is defined as allocating resources among several people who have an entitlement to them so that each person receives their due share. As straightforward as it might sound, fair division has been a problem in mathematical economics for a long time and is now receiving attention from computer scientists (Walsh, 2020). It has numerous applications ranging from border settlement in international disputes to reduction of greenhouse gas emissions while in the era of the Internet, it also appears regularly in distributed resource allocation and cost-sharing in communication networks (Caragiannis et al., 2012).

The first attempt for mathematical solutions dates back to 1948 with Steinhaus introducing the concept of what is now being called "fair cake-cutting", a method of ensuring fair allocation of infinitely divisible heterogeneous goods (Steinhaus, 1949; Stromquist, 1980). Although the study of fair *divisible* allocations is still active, it is better understood compared to fair allocation of *indivisible* goods, with indivisible goods becoming a very active area of research in computer science.

What is considered fair could sometimes be ambiguous or impossible to achieve. Therefore many notions of fairness were introduced with the two predominant ones being proportionality (PROP) (Steinhaus, 1949) and Envy-freeness (EF) (Gamow and Stern, 1958; Varian, 1974). Proportionality focuses more on agents receiving goods such that their allocation is valued equally or more to the total valuation of all goods if it were to be divided by the total number of agents. Envy-freeness on the other hand, revolves around the concept of envy, by requiring that an agent does not envy another agent's allocation.

There are other significant and interesting notions of fairness that we will not focus on here, as they fall outside of the scope of this research.

In our study, we will focus on *indivisible* goods paired with the notions of Envy-freeness and Maximum Nash Welfare by diving deeper into their core definitions, achievability, relaxations, and implementations. For interested readers wishing to examine a more thorough overview of the field, we direct them to the most recent survey of the current state of the field of fair allocation of indivisible goods by Amanatidis et al. (2023).

1.1 Project description, motivation and key findings overview

In this study, we will start by developing a common language that we will use throughout this report which is also coherent with most of the already established research. Based on that, we will formally define a set of notions of fairness (revolving around Envy-Freeness and Maximum Nash Welfare) and we will then look at the current state of the field by focusing on a selection of proposed algorithms that achieve those notions of fairness. With our goal of implementing those algorithms and testing them, an overview of similar projects will be provided, showcasing the lack of similar projects, especially for our selection of notions of fairness.

Once we understand the current field of study, we will delve into the algorithms in much greater detail. Despite the broad description of implementing and analysing the algorithms, it is crucial to note that most of them have never been implemented or put into practice before. This presented new challenges as only some provided high-level pseudocode on how to implement them. We found out that others had key components missing from their provided pseudocodes while others did not include any details at all as to how to implement them. This meant that novel solutions had to be created to enable the algorithms' implementations, focusing both on feasibility and efficiency. Through our implementation phase, we identified key challenges and points of focus for future researchers to aid in the development of feasible algorithms. We also uncovered potential flaws in some of them, leading to further questions about the possible validity of some published papers. Our implementations will also be publicly available for anyone to use with the future goal of disseminating them as a Python library.

Motivated by the lack of analysis of the algorithms with real or synthetic data, it remained unknown as to the algorithms' behaviours and resulting guarantees. Following our implementations, we will therefore put them into practice with real-life and synthetic data to examine their running times across different scenarios while also looking into their fairness guarantees. Interesting correlations or combinations of these algorithms could arise. Are their performance guarantees achievable? Do they achieve better results than promised? Maybe only for specific types of data? Which algorithm should I choose to use if I expect a certain performance characteristic? Through our study, we aim to answer these and many other similar questions, contributing both to future research but also to the current need of implementations of these algorithms.

We will then conclude by revisiting our work to identify potential limitations and areas of improvement. A concise summary of our findings has been compiled to guide researchers make decisions as to which algorithm to use depending on their desired characteristics and running time requirements. In addition, through our experiments, we have identified new, unexpected results and interesting behaviours that could lead to new theorems. Our summarised key findings can be found in Table 5.1 and Table 5.2.

Chapter 2

Preliminaries and Previous Work

2.1 Instance model

We define a model formulation for our problem to aid in our understanding of the following concepts, by establishing a common language that is coherent with any relevant papers.

We start by defining a set of agents N and a set of goods M such that each good can be given (allocated) to only one agent. We formally define this notion of allocation by having a set named A, such that A_i is the set of goods allocated to agent i. We sometimes refer to a list of unallocated goods. These are all the goods that have not been allocated to any agent.

Each agent can express their valuation for each good through their valuation array V. This means that an agent i will have a valuation array V_i which contains a numerical valuation for each good. In addition, we also allow to extract the valuation that a specific agent (i) has for a specific good (j) by writing $V_{i,j}$. We then exploit this notation to allow us to showcase the total value of an agent's allocation by writing $V_i(A_i) = \sum_{j \in A_i} V_{i,j}$ meaning that we have the total (sum) worth of agent $i's$ allocation according to agent $i's$ valuations. Throughout this paper we will only be dealing with such valuations, also referred to as *additive* valuations however, it is important to note that some algorithms also work for non-additive valuations. Any such occurrences will be noted.

2.2 Fairness Notions

Definition 1.1. *Envy-Freeness (EF) can be formally expressed as $V_i(A_i) \geq V_i(A_j)$ for all $i, j \in N$. Meaning that according to agent i, i's bundle is worth more the agent j's and hence agent i does not envy agent j.*

Envy-free allocations of divisible goods had remained an open field of study for a long time with a major breakthrough in 1995 where Brams and Taylor (1996) showed an envy-free protocol for any number of agents. Although the protocol is guaranteed to terminate in finite time, the running time, number of queries and number of cuts, are

unbounded even for four agents. Aziz and Mackenzie (2017) showed that an envy-free allocation of divisible resources can always be found in a finite number of steps with high complexity. However, the same cannot be said for cases with indivisible goods.

Our focus, however, falls under the umbrella of indivisible goods for which, achieving Envy-Free allocations is not as simple as one might expect. Consider the case where there are more agents than goods. There is no possible way of allocating the goods without having at least one agent envying another (as there will be at least one agent who receives no goods) assuming that all goods are positively valued by all agents. It is clear that fair allocation of indivisible goods poses to be a greater challenge.

Due to these limitations, Lipton et al. (2004) introduced a relaxation of EF called Envy-Freeness up to one good (EF1) which was then formally defined by Budish (2011).

Definition 1.2. *Envy-Freeness up to one good (EF1) is defined as* $V_i(A_i) \geq \alpha \times V_i(A_j \setminus \{g\})$ *for some* $g \in A_j$ *and for all pairs of agents* $i, j \in N$ *when* $\alpha = 1$. *Meaning that if agent i envies agent j then there is an item in agent j's bundle that if removed, would remove then envy that agent i has towards agent j. Approximations of EF1 (*α*-EF1) exist when* $1 > \alpha > 0$.

As we will see, EF1 is easy to achieve. However, EF1 can be a fairly weak fairness notion. Imagine the case where the good removed from an agent is a very highly-valued good. This new allocation would therefore be considered as fair even though a large portion of its value has been lost.

To overcome such cases, the idea of EFX was introduced, firstly by Gourvès et al. (2014) under the name near envy-freeness, which was later re-introduced in 2016 by Caragiannis et al. (2019) as EFX. EFX is a stricter version of EF1 as it requires that any item removed from agent j's bundle will remove the envy of agent i towards agent j.

Definition 1.3. *EFX is defined as* $V_i(A_i) \geq \alpha \times V_i(A_j \setminus \{g\})$ *for any* $g \in A_j$ *and for all pairs of agents* $i, j \in N$ *when* $\alpha = 1$. *This means that if agent i envies agent j then if any-one good from agent j's bundle was removed, it would eliminate the envy that agent i has towards agent j. Approximations of EFX (*$\alpha - EFX$*) exist when* $1 > \alpha > 0$.

Although EFX is more strict than EF1, the existence of EFX allocations is still an open problem (Amanatidis et al., 2023). Farhadi et al. (2021) therefore introduced EFR, envy-freeness up to a random good, which is weaker than EFX, yet stronger than EF1.

Definition 1.4. *Envy-freeness up to a random good (EFR), is defined as* $V_i(A_i) \geq \alpha \times \mathbb{E}_{g \in D_j} V_i(A_j \setminus \{g\})$ *where* D_j *is a uniform distribution over the items of* A_j *that selects each good with probability* $1/|A_j|$ *for all pairs of agents* $i, j \in N$ *when* $\alpha = 1$. *Approximations of EFR (*$\alpha - EFR$*) exist when* $1 > \alpha > 0$.

Definition 1.5. *Maximum Nash Welfare (MNW) is an allocation that results in the maximum possible product of the agents' allocations (Suksompong and Teh, 2022; Caragiannis et al., 2019). In other words, ensuring that an allocation results in the largest possible value of* $\prod_i V_i(A_i)$. *It is also important to note that every MNW allocation is also EF1 (Caragiannis et al., 2019). Approximations of MNW (*$\alpha - MNW$*) exist when* $\alpha > 1$ *such that* $Approximation\_MNW = \alpha \times True\_MNW$.

2.3 Achieving the Fairness Notions; Implementations, approximations and special cases

We begin by briefly introducing algorithms that can achieve various fairness notions, their approximations or specific special cases. We dive into their details in Chapter 3.

Envy-Freeness up to one good - EF1

RoundRobin.
RoundRobin is the simplest polynomial time algorithm which allocates the goods to the agents in multiple rounds.

Envy-Cycle elimination (ECE).
Introduced by Lipton et al. (2004) is proven to run in polynomial time and does not rely on a prefixed sequence for agents. Instead, it uses envy-graphs to choose an agent to allocate goods to.

Improved ECE.
Introduced by Chan et al. (2019a) (under no specific name, we use the ”Improved ECE” name in this paper to help distinguish between the two) shares a lot of similarities with ECE. Despite those similarities, it offers further fairness guarantees compared to ECE such as 1/2-EFX.

Envy-Freeness up to any good - EFX

- Identical valuations.
  Plaut and Roughgarden (2020) showed that when all the agents have the same valuations, a variation of leximin can be used to create EFX allocations even then the common valuation function is not additive. This modified version of leximin was named **leximin++**. However, both leximin and leximin++ are not polynomial time algorithms.
- Ordered (and Identical) valuations.
  This case occurs when all agents have the same ordering in their evaluations (but can have different numerical values for each good as long as their ordering of most to least desired good is the same). Plaut and Roughgarden (2020) showed that using **Envy-Cycle Elimination** for such cases, results in EFX allocations.
- Two and three agents.
  Plaut and Roughgarden (2020) have shown that an EFX allocation is always possible when dealing with 2 agents. This is achieved by pretending that both agents have the same valuation function and by running the Envy-Cycle Elimination algorithm for additive functions, and leximin++ for non-additive functions. However, Goldberg et al. (2023) showed that when the valuation function used is sub-modular then the problem turns from efficient to PLS-complete.

  Akrami et al. (2023) introduced a procedure to compute an EFX allocation in pseudo-polynomial time when the instance involves three agents.

- Bi-valued valuations.
  Amanatidis et al. (2021) showed that EFX allocations exist and can be efficiently computed for any number of agents in polynomial time using **Match&Freeze**. In addition, if the valuations are binary, they show that MNW achieves an MMS (Maximin Share Fairness, a different fairness notion to EF) allocation. Later, Garg and Murhekar (2021) showed that this is possible even in conjunction with Pareto Optimality.
- Approximation α-EFX.
  - Plaut and Roughgarden (2020) perused this to show that $\frac{1}{2}$-EFX allocations always exist, with Chan et al. (2019b) showing that **Improved-ECE** can be used to compute an 1/2-EFX allocation.
  - Amanatidis et al. (2020) further improved this result by finding an $\alpha \approx 0.618$ by combining Round-Robin and Envy-Cycle Elimination with some appropriate pre-processing (additive). We call this algorithm **0.618-EFX**.

Envy-freeness up to a random good - EFR

0.73-EFR.
Currently, it is still unknown whether there is a polynomial time algorithm that can be used to achieve EFR. However, Farhadi et al. (2021) have introduced a polynomial time algorithm that achieves 0.73-EFR.

Maximum Nash Welfare - MNW

Finding an MNW allocation is NP-Hard (Garg et al., 2022). Therefore a series of methodologies were developed to try and achieve the best approximation possible.

1.45-MNW.
The current state-of-the-art MNW approximation algorithm guarantees 1.45-MNW in polynomial time. This algorithm introduced by Barman et al. (2018) utilises Fisher Markets together with their Hierarchy Structure to continuously adjust allocations based on the market's prices.

2.4 Related Work

While plenty of previous research has been concentrated on the theoretical performance and efficiency of the algorithms, there has been limited attention directed towards their implementation and testing using real-life or synthetic data.

Hosseini et al. (2023) performed an interesting study where participants were presented with EF1 and other variations of EF allocations, to see which were the most fair by measuring the number of swapped goods by the participants. Their results show that allocations under the HEF-k treatment are perceived to be fairer than under the sEF1 and EF1 treatments, even when controlling for the size of instances, balance of allocations, and questions in which it is optimal to swap.

Kurokawa et al. (2018) used automated experiments to try and find a counterexample for the existence of MMS allocations. No counter-example was found from the experiments but they aided in the design of manual counterexamples which required precise construction. Despite this, no further information was provided on how these experiments were performed.

SPLIDDIT (Goldman and Procaccia, 2015), represents a real-life application of such research. Spliddit is a non-profit website that allows its users to employ fair division algorithms for everyday problems. It offers functionality to split rent, share credit, and even divisible or indivisible goods allocation. Focusing on the allocation of goods, the system aims at trying to find the highest feasible level of fairness. If envy-freeness and proportionality are infeasible, the algorithm computes the maximum $\alpha > 0$ such that each player can achieve an α fraction of their MMS guarantee. Secondly, the algorithm maximizes social welfare ($\Sigma_i Vi(Ai)$) subject to the fairness constraint found in the first phase.

(Conitzer et al., 2019) performed experiments using SPLIDDIT and synthetic data to examine in practice the performance of locally Nash-optimal allocations in groups of players. They conclude that the local search algorithm converges quickly, and is likely to output an efficient allocation. In the same year, Farhadi et al. (2019) explored a different notion of fairness, WMMS, showing that the best approximation possible would be 1/n-WMMS using a somewhat round-robin procedure and EBAY bidding data.

Some of the more relevant research for our topic include the paper by Nagi and Elgrabli (2022) where the authors created an open-source Python package comprising of multiple implementations of algorithms for fair division of divisible and indivisible goods as well as courses. The research mainly focuses on the experimentation of 3/4-MMS, Max sum allocation, leximin, and PROPm using experiments with SPLIDDIT data. Their results conclude that 3/4-MMS returns similar results with PROPm. While these are very interesting results, PROPm and MMS fall outside of the scope our research. In an equally important paper by Freeman et al. (2020), the authors experimented with both synthetic data and data from SPLIDDIT for allocating chores. The experiments involved the following four algorithms: (1) The greedy algorithm from Proposition 3 in Freeman et al. (2020) (EQX), (2) the Leximin solution, (3) the market-based algorithm Alg-eq1+po, and (4) an algorithm currently deployed on the SPLIDDIT website for dividing chores. Through their research, they introduced an interesting metric of comparing the percentage of instances produced by each of the algorithms that conformed to a collection of different fairness notions. Through their experiments, they identified that out of the algorithms in test, Leximin emerges as the algorithm of choice in terms of simultaneously achieving approximate fairness and economic efficiency.

Despite these studies, it is evident that there is a lack of implementations for algorithms that guarantee any approximation to Envy-Freeness or Max Nash Welfare. Moreover, no experiments have been conducted to ascertain the extent to which these algorithms fulfil fairness guarantees beyond their established theoretical ones. Such inquiries hold the potential to yield discoveries that could reshape the field by inspiring the development of new algorithms or uncovering additional fairness guarantees for existing algorithms.

Chapter 3

Algorithms and their implementations

There is a diverse array of different algorithms that achieve different fairness criteria. In order to facilitate a more comprehensive examination, we opted to concentrate on algorithms that guarantee different Envy-Freeness fairness criteria (such as EFX, EF1, EFR). To aid us provide a more generalised view, we have also investigated the state-of-the-art MNW approximation algorithm to enable further discussion as to the effect that it might have on EF allocations and vice versa.

3.1 RoundRobin

Guarantees: EF1

Description

All agents are ordered in an arbitrary but fixed order. During each round, one agent (chosen based on the ordering) chooses their most valuable good from the set of unallocated goods. In the next round, the following agent according to the ordering gets to choose a good. The ordering of the agents is maintained throughout the rounds and this process ends when all goods have been allocated. Pseudocode of RoundRobin is available in Algorithm 1.

3.1.1 Implementation details description

The implementation of the algorithm is fairly straightforward and no complex operations were needed.

3.2 Envy-Cycle Elimination ECE (Lipton et al., 2004)

Guarantees: EF1, EFX with only two agents, EFX on Identical and Ordered (All agents have the same ordering in preference in their valuation arrays with possibly different numerical values) instances.

3.2.1 Description

Contrary to RoundRobin, ECE does not rely on a prefixed sequence for agents. It instead, selects an agent that is in a disadvantage compared to other agents (not envied by other agents) and allocates to them their most valuable good from the unallocated ones. The decision to select the most valuable good is a variant of the traditional envy-cycle elimination algorithm (EF1 is guaranteed by allocating any good within an iteration). Envy is represented with a graph where vertices represent each agent and directional edges between vertices indicate envy of one agent towards another (based on each agent's valuation of the goods). For example, if agent i envies agent j it means that agent i values agent j's bundle more than their own. This is represented in the graph with a directed edge from vertex i to vertex j.

As aforementioned, at each iteration an unenvied agent gets to choose their most desired good. If no such unenvied agent exists, then the envy graph must contain a directed cycle which can be eliminated by re-distributing the goods among the agents involved in the cycle such that each agent gets the bundle they envy. Assuming that agents i, j, and z are in a directed cycle and each agent envies the agent ahead of them ($i \to j \to z \to i$, where $x \to y$ represents envy of x towards y) then the bundle of agent j is given to agent i, the bundle of agent z is given to j and the bundle of agent i it given to agent z. Repeating this process leads to at least one agent who is not being envied. The algorithm terminates once all goods have been allocated. Envy-Cycle Elimination pseudocode can be found in Algorithm 2.

3.2.2 Implementation details

We have introduced our own graph representation through our Graph class. Although this was not strictly necessary as NetworkX (Hagberg, 2023) (a Python library for graphs and networks) has all the features we needed, we followed our path to enable future flexibility and modifications. We will use our graph implementation to represent the Envy-graph.

Through the Graph class we have operations such as *nodes_without_incoming_edges()* which returns the list of nodes with no incoming edges (unenvied agents) as well as *identify_cycle()* (a wrapper of Algorithm 3) which performs a DFS traversal of the graph until a cycle is found and returns the nodes in the order they appear in the cycle. Graph also includes other functions such as *topologicalSort()* which returns a topological sort of the graph, which is later used in other algorithms.

During initialisation and after re-allocation of goods given a cycle, we use *check_envy()* to recalculate the entire envy-graph by checking all pairs of agents as any re-allocation of goods could have resulted in different envies between agents (this is a potential point of improvement, as there are some comparisons with small cycles that can be avoided). However, when we simply need to assign a good to an unenvied agent we use *check_envy_towards_agent()* so that we only need to check for envy between all the agents and the agent that we have just allocated a good to. This helps decrease complexity from $O(N^2M)$ to $O(NM)$ and hence aids in the overall efficiency of the implementation.

Re-allocation of goods is handled by our *reallocate_goods()* method that takes as a parameter a cycle that was returned by *identify_cycle()*. In addition to the aforementioned, we have also enabled through the use of a different public function (*run_ece_partial()*), the ability to run ECE, given an already provided allocation and set of unallocated goods. This functionality is again used in a different algorithm.

3.3 Improved ECE (Chan et al., 2019b)

Guarantees: EF1, 1/2-EFX

3.3.1 Description

Improved ECE shares a lot of similarities with ECE. It starts by computing a maximum weight matching between the unenvied agents and the available goods in a bipartite graph. The weights are defined by $V_i(A_i \cup \{j\}) - V_i(A_i)$ for agent i and good j. If all edges have weight 0 then a maximum cardinality matching is computed instead. For every edge in the maximum weight matching, each good in the matching is allocated to its associated agent and it is removed from the list of available goods. We then eliminate all envy cycles from the envy graph before repeating the procedure again until all goods have been allocated. Pseudocode for Improved-ECE can be found in Algorithm 4.

3.3.2 Implementation details

Construction of the bipartite graph is implemented using NetworkX with goods represented in the format ”g_Good-index” to differentiate from agents. Then using NetworkX’s *max_weight_matching(graph, maxcardinality=True)* we compute the max weight matching from the graph.

Using ECE’s *check_envy()* and *identify_cycle()* we are then able to build the envy-graph and identify cycles. Each cycle is resolved by appropriately reallocating goods to the agents involved in the cycle (using ECE’S *reallocate_goods()*).

3.4 0.618-EFX (Amanatidis et al., 2020)

Guarantees: EF1, 0.618-EFX

3.4.1 Description

The algorithm starts with a pre-processing step where it reorders the agents so that the first few agents are quite happy with their pick in the first round of the RoundRobin. For the remaining agents (not the first few), they get allocated a second good again through RoundRobin but this time in reverse order. The resulting partial allocation, where each agent receives one or two goods, achieves 0.618-EFX with respect to the currently allocated goods. The remaining unallocated goods are allocated to the agents through ECE which the authors have proven to maintain the 0.618-EFX characteristic.

3.4.2 Implementation details

A variation of the RoundRobin was used (not the original one aforementioned before) as this algorithm only allocates goods to a certain number of agents and with a specified order. Once the two rounds of RoundRobin have finished, ECE takes over using *run_ece_partial()* method from ECE that we have introduced before. The algorithm although it performs a moderately tricky pre-processing step, does not contain any critical parts of the implementation that deserve any special explanation.

3.5 0.73-EFR (Farhadi et al., 2021)

Guarantees: EF1, 0.73-EFR

3.5.1 Description

The algorithm starts by allocating one good to each agent such that the Nash Welfare is maximised. Based on those allocations an envy-ratio graph (a generalisation of the envy-graph, introduced by Lipton et al. (2004)) is created by forming a fully connected graph of agents (represented as nodes) and setting the weights of each edge to be $w_{i,j} = V_i(A_j)/V_i(A_i)$. We then define the envy-rank of each agent as r_i where $r_i = \max_{j_0,j_1,\ldots,j_k} \prod_{z=1}^{k} w_{j_z,j_{z-1}}$ and $j_0 = i$.

Agents are then split into three categories (G1,G2,G3) based on their envy-ratio. Agent i belongs to G1 if $r_i > \theta$, to G2 if $2 < r_i \leq \theta$, and to G3 if $r_i \leq 2$, where $\theta = \sqrt{3}+1$. The agents in each category are then sorted based on a topological sort of the envy-graph. Each agent in G3 chooses their most valued good, then again agents in G3 choose their most valued good out of the available ones followed by agents in G2. If not all goods have been allocated the Improved ECE algorithm is used to allocate the rest of the goods. For further details and pseudocode, we direct the reader to the original paper as it includes more detailed descriptions and explanations.

3.5.2 Implementation details and challenges

Despite the straightforward description of the algorithm, implementing and experimenting with real data was proven to be a great challenge.

Starting from the first step, the authors have provided (although in only some versions of their paper), a polynomial time algorithm for allocating one good to each agent such that Nash Welfare is maximised (we will call this allocation, NSW). This is achieved by forming a bipartite graph using NetworkX where one set of the nodes is the agents and the other set is the goods, encoded similarly to Improved-ECE as "g_GoodIndex". The weights of each edge going from an agent to a good is the logarithm of the agent's valuation for each good. The use of logarithms allows us to exploit the behaviour of logarithms when they are being added as it is equivalent to multiplying the raw valuations. To find an NSW allocation a maximum weight matching is computed (using NetworkX) which is then translated to be our initial allocation.

A formidable obstacle was lurking in the corner unfortunately for the very next part. Recalling the calculation for the envy-rank for each agent $r_i = \max_{j_0,j_1,\ldots,j_k} \prod_{z=1}^{k} w_{j_z,j_{z-1}}$, it can be translated as the longest possible path leading to node i when the weights are multiplied. Through the authors' Observation 7, we know that the path will be simple. The authors recommend a similar approach to our solution for NSW allocation, to take the logarithm of each weight and make it negative. That way we can use the Bellman-Ford algorithm to find the shortest possible path, which given our negative log transformation directly corresponds to our goal.

Assume that we have two agents with identical valuations. One agent was allocated a good with value 10 and the other agent a good with value 11. We therefore know that the non-transformed weights will be $10/11$ and $11/10$. If we transform them we have, $-\log_{10} 11/10$ and $-\log_{10} 10/11$ which when added result $-\log_{10}(10/11 \times 11/10) = -\log_{10} 1 = 0$ (not a negative cycle). When implementing this however, we have $-\log_{10} 10/11 = 0.0953101798043249$ and $-\log_{10} 11/10 = -0.09531017980432493$. It is evident that the last two decimal places differ and when adding these two together results in $-4.163336342344337e-17$, which is a negative cycle and would cause Bellman-Ford to fail *(this behaviour is not consistent and took an instrumental amount of time to pinpoint)*. Our first attempt at solving this was to use the Decimal Python library (Decimal, 2024) to allow for exact decimal representation (also avoids cases like 1.1 + 2.2 = 3.3000000000000003 which can happen with Python's float).

Using Python's Decimal, however, introduced another problem. It sometimes rounds the very last decimal place resulting again in cases where negative cycles were created even though they should not exist. As a last resort, we decided to increase the Decimal's precision to 50 decimal places and then truncate them to 40 decimal places to hopefully avoid those rounding errors. However, this again would not work. Figure 3.1 showcases two sub-graphs of the envy-ratio graph. The graph on the left uses decimal precision of three decimal places while the one on the right uses a truncated version with two decimal places precision. As can be seen, the graph on the left has no negative cycles while the one on the right has a negative cycle (when adding the weights).

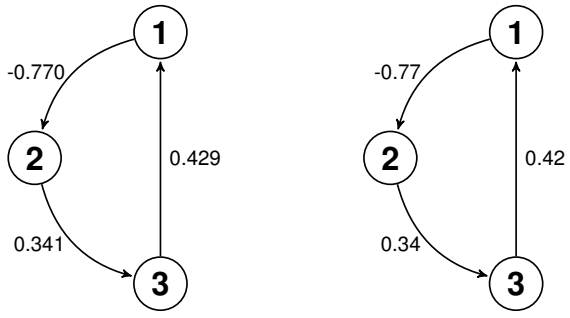

Figure 3.1: Left: part of the envy-ratio graph with weight precision set to three decimal places. Right: part of the envy-ratio graph with weights truncated to two decimal places.

Therefore, we concluded that it would be impossible to handle all such possible cases and therefore this strategy would not work in the implementation. Whilst this shows that there are concerns with regards to the feasibility of implementing this algorithm,

something that will be crucial to anyone who needs to use this algorithm in practice, we decided to make a compromise and take a different route. Our solution was to produce all possible simple paths leading to each node (agent) and then choose the longest one (based on the addition of their transformed, positive logarithmic, weights). This drastically increased running time but, provided a consistent solution that worked. As you will see in our analysis later on, we have used this path enumeration method to analyse the fairness guarantees of the algorithm and the Bellman-Ford algorithm, whenever possible, for the experiments involving running time. We acknowledge that this is far from an ideal solution, and given the time constraints of this project this seemed like the only solution possible at the time.

Despite this setback in terms of running time, this observation of ours merits some focus from the research community. Such precision problems occurred in other algorithms as well, and it is therefore evident that such real-life limitations should be taken into account when designing such algorithms.

We employed NetworkX's graph representation to use ready-made path enumeration functions for the envy-ratio graph but then used our aforementioned Graph representation to take advantage of our ECE functions for running the ECE part of the algorithm.

3.6 Match&Freeze (Amanatidis et al., 2021)

Guarantees: EFX [Binary & Bi-valued]

3.6.1 Description

The algorithm guarantees EFX for binary and bi-valued instances with the compromise that it only works with binary and bi-valued valuations. At the beginning, an ordering of agents is established that remains constant throughout all rounds. In each round of the algorithm, a bipartite graph is created with one category being agents in list L (all agents are originally in L) and the other being unallocated goods. An edge connects an agent and a good only if the agent values the good as α (the highest valuation of the two possible values). A maximum matching is then created from the graph, and based on the matching, we allocate goods to their corresponding agents. For agents which have not been allocated goods in this round, we order them according to our predefined ordering and allocate to them one arbitrary good.

We now construct a set of frozen agents. Included are all the agents that some other agent (named j) believes were allocated a good of value α while agent j was allocated a good that was not valued as α according to j. We then also add any other agents which someone else believes that they were allocated a good of value α.

These agents are then removed from L for the next $\lfloor \alpha/b - 1 \rfloor$ rounds, where b is the second possible value for a valuation (not α). Once these agents are no longer frozen, they are added back at the end of the list L.

This process is repeated for each round until all goods have been allocated. Pseudocode of Match&Freeze can be found in Algorithm 5.

3.6.2 Implementation details and challenges

Implementing Match&Freeze did not pose any major challenges. The bipartite graph was implemented using NetworkX again for its convenience. Although it does provide functionality for finding maximum matching, it does not work when the graph has disconnected components. This meant that an additional step had to do be implemented in order to find all the connected components and then find the maximum matching on each. This was again handled with NetworkX functionality.

Handling the list of frozen agents was something that was not specified in the algorithm and it makes sense as it could be handled in different ways depending on the language being used to implement the solution, if multi-threading is used any many other factors. For our implementation, we keep track of how many rounds we have done and maintain a dictionary where the key is the value that the round counter should have for the agents to be unfrozen and the value is a list of the agents. This simple approach was chosen for its simplicity and low runtime complexity.

3.7 Leximin++ (Plaut and Roughgarden, 2020)

Guarantees: EFX [Ordered]

3.7.1 Description

Whilst this algorithm does not execute in polynomial time, we decided to include it as a showcase that EFX allocations exist in instances where the valuations of all agents have the same order. The complexity of discovering a leximin allocation is known to be NP-hard (Plaut and Roughgarden, 2020). Consequently, it is highly probable that the task of finding a leximin++ allocation also falls within the realm of NP-hardness.

Leximin++ is a variation of leximin that can be used to create EFX allocations even when the common valuation function is not additive. This modified version of leximin was named leximin++ and works as follows:

- Identify the maximum minimum bundle valuation (single agent's allocation) that can occur in all possible allocations. In other words, find the minimum valuation of each allocation's total bundle value (from all allocations find the minimum valued bundle that any agent has received). Then find the maximum value that this minimum can have based on all the allocations and disregard all the allocations that have a bundle with a value less than this identified maximum minimum. Essentially we are maximising the minimum bundle value that an agent can have.
- Instead of repeating the above with the remaining allocations (the steps for the original leximin), we aim to maximise the minimum size of bundles. In other words, maximise the minimum bundle size. Out of those minimums find the maximum and disregard all allocations which contain a bundle with a size less than our identified minimum.
- Repeat the steps above.

3.7.2 Implementation details and challenges

Firstly we needed to generate all possible allocations. The original authors, however, did not provide a systematic method for achieving that. This was solved by utilising a backtracking process to continuously allocate each good to each agent. Once all items have been allocated, we backtrack and start assigning the backtracked goods to the next agent.

To help aid in the optimisation of generating all the possible allocations we have employed a branch-pruning technique. At each step of the recursion, our implementation assigns the current good to each agent and calculates the utility of each of the bundles (agent's allocation) in the current allocation configuration. We also keep track of the minimum utility encountered (min_utility) among all possible allocations encountered so far. Before exploring a potential allocation further, the algorithm checks whether the minimum utility of the current allocation is greater than or equal to the overall minimum utility encountered so far (min_utility). If it is not, we prune that branch of the search tree, by removing the most recently allocated good and allocate it to the next agent. Imagine the case when we have three agents and three goods. We know that allocating the three goods to the first agent will result in the second agent receiving no goods. So there is no need to allocate the third good to the first agent. We can prune that branch and try to allocate that good to the next agent. This solution of ours helps not only decrease the time needed for generating all the possible allocations but also decreases the search space of the first phase of the algorithm (as there will be fewer allocations to try and maximise the minimum bundle value). Pseudocode on how allocations are generated and pruned can be found in Algorithm 6.

Once we have all the allocations generated, we pre-process them so that each bundle contains three key details, the valuation of the bundle, its size and a key signifying which allocation it is part of. This pre-processing allows us to have all the calculations that we will ever need to do, ready from the beginning, eliminating the need to repeat the same calculations for the same allocations at each iteration of the algorithm.

We then process the pre-processed allocations according to Leximin++ by selecting a base allocation and the next allocation and comparing them using the Leximin++Cmp function described in the original paper. If we find that the next allocation is preferred compared to the base allocation, it then becomes the new base allocation.

To help provide context as to the optimisation of our implementation, our optimised implementation tested with 5 agents and 10 goods executes in 36.9 seconds compared to 109 seconds without any branch pruning.

3.8 1.45-MNW (Barman et al., 2018)

Guarantees: 1.45-MNW

3.8.1 Description

1.45-MNW is currently the state-of-the-art approach for the best MNW approximation.

The algorithm utilises Fisher Markets to establish common prices for all goods and adjusts them accordingly. Goods are allocated to agents according to a Hierarchy which is computed based on the current allocations and if an agent can ”purchase” a good by spending their *maximum bang per buck*. Due to the large complexity of the algorithm, we encourage the reader to have a read through the algorithm and supporting explanations in the original paper. In the following sections, we go through some of the parts of the algorithm through an example. However, due to the structure of the algorithm and the example chosen, we do not go through phase 3 of the algorithm.

3.8.2 Implementation details and challenges

Despite the detailed explanations provided in the paper, there were numerous modifications and new methods that needed to be implemented to be able to make the pseudo-code provided, functional.

Starting with the execution flow, it was evident that the pseudo-code uses goto functionality. Since such functionality is not offered directly by Python, we restructured the code to avoid this need. This was achieved by having a general method which itself calls another method that performs phases 2 and 3. In the method that performs phases 2 and 3, if phase 2 needs to be repeated, it returns the key ”2” as well as any other necessary parameters needed to be able to re-run phases 2 and 3 to the main method. If on the other hand, the method needed to terminate, it would return the key ”0” to signify to the main method that the algorithm has finished executing.

Finding Max bang per buck goods
To calculate the *maximum bang per buck* of an agent we define *bang per buck* to be $a_{i,j} = v_{i,j}/p_j$ for each agent i and each good j. We define the *maximum bang per buck* for each agent as α_i. Using this α_i we then identify which goods achieve this maximum. However, as previously mentioned with the decimal precision problems in 0.73-EFR, Python’s float division can lead to unexpected behaviours. We therefore again employed the Decimal library and increased the decimal precision to 50 decimal places. This again would not completely solve the issue as Decimal sometimes rounds up the last decimal place. To combat this, we have introduced a tolerance that if a bang per buck is within 1×10^{-40} of α_i then, we will consider the two to be equal. Although this is not a robust solution as with very small valuations, some cases might incorrectly fall within the tolerance, it was the only solution that would enable us to implement the algorithm as the randomness of when Decimal would round up the final decimal place was unpredictable. Other solutions like choosing the goods with the highest *bang per buck* per agent would again need tolerance as even if the valuation and price were the same between them, the calculated *bang per buck* might differ even by a small amount (there would be no systematic method of verifying if that difference is meant to be there or was caused due to Python, requiring tolerance again). Even Python’s Fractions library (Python, 2024), cannot, be used with floats as numerators or denominators.

Building the Hierarchy
Only in the most recent version of the paper, the authors have included the BuildHierarchy method which executes as expected. However, during phase 2 of the algorithm, we also require the path we followed to reach each agent in the hierarchy. A method for

finding these paths was not included in the paper.

In our solution, we first start by building the augmented graph using NetworkX, with goods being preceded with ”g” so that they differentiate from the agents and then add all the edges with appropriate types (”MBB” and ”allocation”).

Using the augmented graph created, we perform a BFS-like traversal of the graph with some modifications to achieve the requirement for the path to be an alternating path of MBB and allocation edges. In each BFS iteration, we first find all the neighbouring goods we can reach through MBB edges and then from those goods we check from each, which agents we can reach through allocation edges. Once we find those agents, we add them to the BFS queue as the agents to be then explored by BFS and mark them as visited so that we do not add them to the queue again. We then store in a dictionary, using the levels as keys, and as values, a list of tuples, the tuple being the reachable agent and the path followed. Our implementation for creating the hierarchy has been included as Algorithm 8. Plenty of additional helper functions were also necessary totalling over 400 lines of code for implementing 1.45-MNW.

All other parts did not require any specialised implementations that were more challenging or complicated than expected.

3.8.3 Counter-example not achieving 1.45-MNW

Trying to debug such a complicated algorithm with multiple iterations and prices that are floats requiring multiple decimal places to reflect the price increases, was no easy task. This was a major challenge in our project as keeping track of all the parameters, constructing graphs at each iteration and handling all the mathematical precision when verifying by hand, as a last resort, was extremely time-consuming.

From the countless hours spent trying to identify potential issues, we prioritised on finding simple to-execute examples that would result in unexpected results. Through this process, we were able to identify some simple counter-examples that to our surprise did not achieve the 1.45-MNW guarantee. As a disclaimer, we would like to inform the reader that the following analysis is based on our understanding of the algorithm. The following counter-example and analysis will also be sent to the authors of the paper.

In this section, we run through one of the simplest instances we could find, to showcase how the algorithm does not achieve 1.45-MNW and also try to pinpoint possible roots for this abnormal behaviour.

Counter-example.
We begin by setting $\varepsilon = 0.01$. In our example, we will have two agents with three goods. We therefore generate some random valuations that are power-of-$(1+\varepsilon)$ as required:

$$V = [[1.01^{480}, 1.01^{118}, 1.01^{138}], [1.01^{340}, 1.01^{11}, 1.01^{78}]]$$

The algorithm starts by allocating each good to the agent that values it the most (we use 0-index for both agents and goods). In this case, agent 0 gets all the goods as they value all of them the most:

$$A = [[0,1,2], []]$$

We then set the price for each good to be the value that the agent who was allocated the good has for it:

$$P = [1.01^{480}, 1.01^{118}, 1.01^{138}]$$

We now need to check if the current allocation is 3ε-pEF1. 3ε-pEF1 is defined as follows: for every pair of agents $i,k \in N$ there exists a good $g \in A_k$ such that $(1+\varepsilon)P(A_i) \geq P(A_k \setminus \{g\})$ (you can assume that P would behave in the same way V_i would by providing the total value by adding the prices of all the goods in the allocation provided). From this, we know that when $i = 1$ and $k = 0$, the allocation is 3ε-pEF1 as k has an allocation of value 0. When $i = 0$ and $k = 1$ we know that the left-hand side of the equation will always be 0 so, the allocation is not 3ε-pEF1.

Now we are ready to begin Phase 2. Our next step is to identify the agent that is the least spender. This would be agent 1 as their allocation is valued at 0 compared to agent 0's $(1.01^{480} + 1.01^{118} + 1.01^{138})$. Using the least spender we can start to build the Hierarchy, but as a pre-requisite, we need to find the *maximum bang per buck* of each agent and the goods that achieve this *maximum bang per buck*. To follow the paper's, notation we define *maximum bang per buck* ($\alpha_{i,j} = V_{i,j}/P_j$) of agent i as $\alpha_i = \max_j \alpha_{i,j}$ for all $j \in M$. We then define the goods that maximise this ratio for each agent as $MBB\_i = \{j \in M : V_{i,j}/P_j = \alpha_i\}$.

$$\alpha_0 = 1, MBB_0 = \{0,1,2\}$$

$$\alpha_1 = 1.01^{78}/1.01^{138}, MBB_1 = \{2\}$$

Based on these and the current allocation, we can construct the augmented MBB graph:

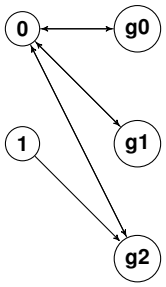

Figure 3.2: The augmented MBB graph of our counter-example during the first iteration. Edges going from agents to goods are MBB edges and edges going from goods to agents are allocation edges (good allocated to agent).

From this graph, we can build the hierarchy. For convenience, the Hierarchy will be formatted in the following format: H={level_of_agents : [{agent, path_to_reach_agent}, {another_agent, path_to_reach_another_agent}]}. The level of an agent is defined as half the length (number of edges) needed to reach that agent from the starting point. The starting point has level 0 and any unreachable agents have level N (although unreachable agents are not added to the hierarchy). In brief terms, an agent is reachable if from the starting point, we can follow a series of alternating MBB and allocation edges such that there is a simple path leading to that agent. It is important to note that this description is a simplification and we direct the reader to the original paper for the full definition. In

our example, we begin our search for reachable agents from node 1 (the least spender). We know we can reach the least spender with level 0 (no path) and agent 0 by visiting g2 (in total 2 edges so agent 0 has level 1):

$$H_1 = \{0 : [\{1, UNDEFINED\}], 1 : [\{0, 1 \to g2 \to 0\}]\}$$

We know that the current allocation and prices are not 3ε-pEF1 as the allocations and prices are the same as before. Starting from level 1, we need to check for each agent at that level if the agents are path violators. An ε-path-violator is an agent with respect to the alternating path $Path = (i, j_1, i_2, j_2, ..., i_{l-1}, j_l, k)$ if $P(A_k \setminus \{j_l\}) > (1+\varepsilon)P(A_i)$. From our path for agent 0, $1 \to g2 \to 0$, we have $(1.01^{480} + 1.01^{118}) > (1+\varepsilon)0$ meaning that agent 0, is an ε-path-violator. We therefore need to adjust the allocations by giving g2 to agent 1. Our updated allocation is as follows:

$$A = [[0, 1], [2]]$$

We now start phase 2 again. Our least spender is still agent 1 as $P(g2) < P(g0 + g1)$. Since the prices have not changed, MBBs remain the same and we can start to construct our new augmented MBB graph based on the updated allocation.

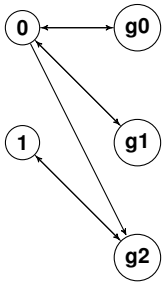

Figure 3.3: Updated augmented MBB graph of our counter-example during the second iteration. Edges going from agents to goods are MBB edges and edges going from goods to agents are allocation edges (good allocated to agent).

As we can see from the graph we cannot reach any other agent other than our starting agent (agent 1, as they are still the least spender). The updated Hierarchy only includes agent 1:

$$H_1 = \{0 : [\{1, UNDEFINED\}]\}$$

Since our Hierarchy is empty we proceed to the next step which is to check if the allocation is 3ε-pEF1. We have the following three checks that we need to perform:

For i=0 and k=1 (by removing g2 from agent 1):

$$(1+\varepsilon)(1.01^{480} + 1.01^{118}) \geq 0$$

For i=1 and k=0 (by removing g1 from agent 0):

$$(1+\varepsilon)1.01^{138} \geq 1.01^{480}$$

Or with i=1 and k=0 (by removing g0 from agent 0)

$$(1+\varepsilon)1.01^{138} \geq 1.01^{118}$$

Since there is at least one good for each agent that if removed satisfies the requirement, we know that the allocation is infarct 3ε-pEF1 and the algorithm terminates.

We can now calculate the Nash Welfare of the current allocation:

$$NW\_algorithm = (1.01^{480} + 1.01^{118}) \times 1.01^{78} = 1.01^{558} + 1.01^{196}$$

We also know that the allocation that maximises the Nash Welfare is the following:

$$A = [[0], [1,2]]$$

$$NW\_optimal = 1.01^{480} \times (1.01^{11} + 1.01^{78}) = 1.01^{491} + 1.01^{558}$$

To find the α-MNW we need to divide the optimal MNW by the MNW of the algorithm's allocation:

$$\frac{NW\_optimal}{NW\_algorithm} = \frac{1.01^{491} + 1.01^{558}}{1.01^{558} + 1.01^{196}} \approx 1.4732$$

As you can see this exceeds the guarantee of 1.45.

Possible source of the problem.
We would like to draw the reader's attention to Lemma 1 of the original paper. We repeat this lemma in this report for the reader's convenience.

Lemma 1. Given a fair division instance with identical and additive valuations, any ε-EF1 allocation provides a $e^{(1+\varepsilon)/e}$-approximation to Nash Social Welfare.

ε-EF1 is defined by the authors as the following: given any $\varepsilon > 0$, an allocation A is said to be ε-approximately envy-free up to one good if for every pair of agents $i, k \in N, \exists$ a good $j \in A_k$, such that $(1+\varepsilon)V_i(A_i) \geq V_i(A_k \setminus \{j\})$. For the reader's convenience, we provide a counter-example for this lemma again with two agents and three goods:

$$V = [[1000, 100, 99], [1000, 100, 99]]$$

We can now construct an ε-EF1 allocation:

$$A = [[0,2], [1]]$$

We can set $\varepsilon = 0.01$ so that $e^{(1+\varepsilon)/e} \approx 1.4499$. Using our decision for the value of ε, we can verify that our allocation is ε-EF1 as when k=1 and i=0, the RHS of the equation has value 0 and is therefore true and when k=0, i=1, we know that if we remove the most expensive good allocated to agent 0, the equation will be $(1+\varepsilon)100 \geq 99$ which is again true.

However, we know that the MNW-optimal allocation is:

$$A = [[0], [1,2]]$$

To calculate the MNW approximation we use the same formula as before:

$$\frac{NW\_optimal}{NW\_allocation} = \frac{1000 \times (100+99)}{(1000+99) \times 100} = \frac{1990}{1099} \approx 1.81$$

We have therefore shown that even if an ε-EF1 allocation with identical and additive valuations is found, it is not guaranteed to be $e^{(1+\varepsilon)/e}$-approximate to Nash Social Welfare. Lemma 1, is a very strong claim especially since there is no correlation between ε and the valuations.

Chapter 4

Experiments

1.45-MNW was removed from our test suite due to the inconsistencies identified before. If our understanding of the algorithm is incorrect then so will all our results. If the algorithm turns out to need some modifications then any results that would have been published here would have been invalid and could cause confusion for future readers.

4.1 Execution time experiments

4.1.1 Methodology

To perform timing experiments and properly evaluate how different properties affect the running time of each algorithm, we split the possible valuations into five different categories that arose from the literature; Random, Ordered, Binary, Bivalued, and Identical valuations.

Valuations generation
Random and Identical valuations were generated with uniform distribution over the range 0-1000 inclusive. Binary and Bivalued instances used normal distribution with Bivalued again ranging from 0 to 1000. Ordered valuations were a bit trickier as we wanted to maintain properties such as when two goods have the same valuation, this should be reflected in all agents' valuations.

To start generating Ordered instances, we first randomly generated with random distribution a valuation array for a single agent and then ordered it in ascending order. To create a new valuation arrays for the following agents we follow the following procedure:

1. Find a valuation for the least valued good as a random number in the range 0 to $smallest\_valuation\_from\_first\_agent + 99$. We added 99 to the first agent's valuation to allow for some variation between the agents' valuations. Using a larger value would have resulted in possibly more unbiased results but could cause the valuations for the following goods to have extreme values.
2. We then check if the following good has the same value as our current good in the first agent's valuations. If that is the case, then from the valuation that we are currently building, we re-use the same value that we have used for the last added

good. This means that if good2 and good3 have the same valuation in agent_1's valuation, then good2 and good3 will also have the same valuation in agent_x's valuation.

3. If they do not need to have the same valuation then we can take the valuation of our previously added good and generate a value for the next one with a range $prev\_val + 1$ to $prev\_val + 99$. This again ensures that our new valuation will be larger than our previous one. In addition, due to the way that the first value (of the very first good) is initialised, it is possible that the current good's valuation might have a lower value than our original agent (helping to ensure randomness). Combining this with the addition of +99 cascades the variation as we move on to the next good allowing for possibly much greater variation.
4. We start again from step 2 for the next good.

Experiment plan
To enable meaningful results, only one variable had to be free in each experiment. That meant splitting the experiments into two sections, one assessing how the number of goods affects the execution time and one on how the number of agents affects the execution time.

For the first case, the number of agents was fixed to 20 and then one instance would be generated per number of goods ranging from 50 to 990 with increments of 20. This ensured that the test set consisted of both, instances where the number of goods was a multiple of the number of agents, and instances where they were not.

For the second case, the number of goods was fixed to 500 (to hopefully still enable challenging scenarios by requiring to allocate possibly more than one good to an agent) and set the number of agents to range from 2 to 50 with increments of 1.

Experiment restrictions
Due to the complexity of large instances and the lack of similar research work, expected running times were unknown. This led to some instances taking very long to execute. Hence, some experiments unfortunately had to be cut short. As a fixed restriction, any experiment taking longer than 400 seconds was terminated (with one exception, as seen in our results).

It is also important to note that Leximin++ is not included in any of the execution time experiments. The algorithm has exponential running time and can therefore only be tested with extremely small instances. Based on these details, the decision was made not to include it.

Experiments test-bench
All execution time experiments were carried out on a DELL XPS 15 9500 with an Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s), 16GB of RAM and an NVIDIA 1650Ti.

4.1.2 Results: fixed number of agents, varying number of goods

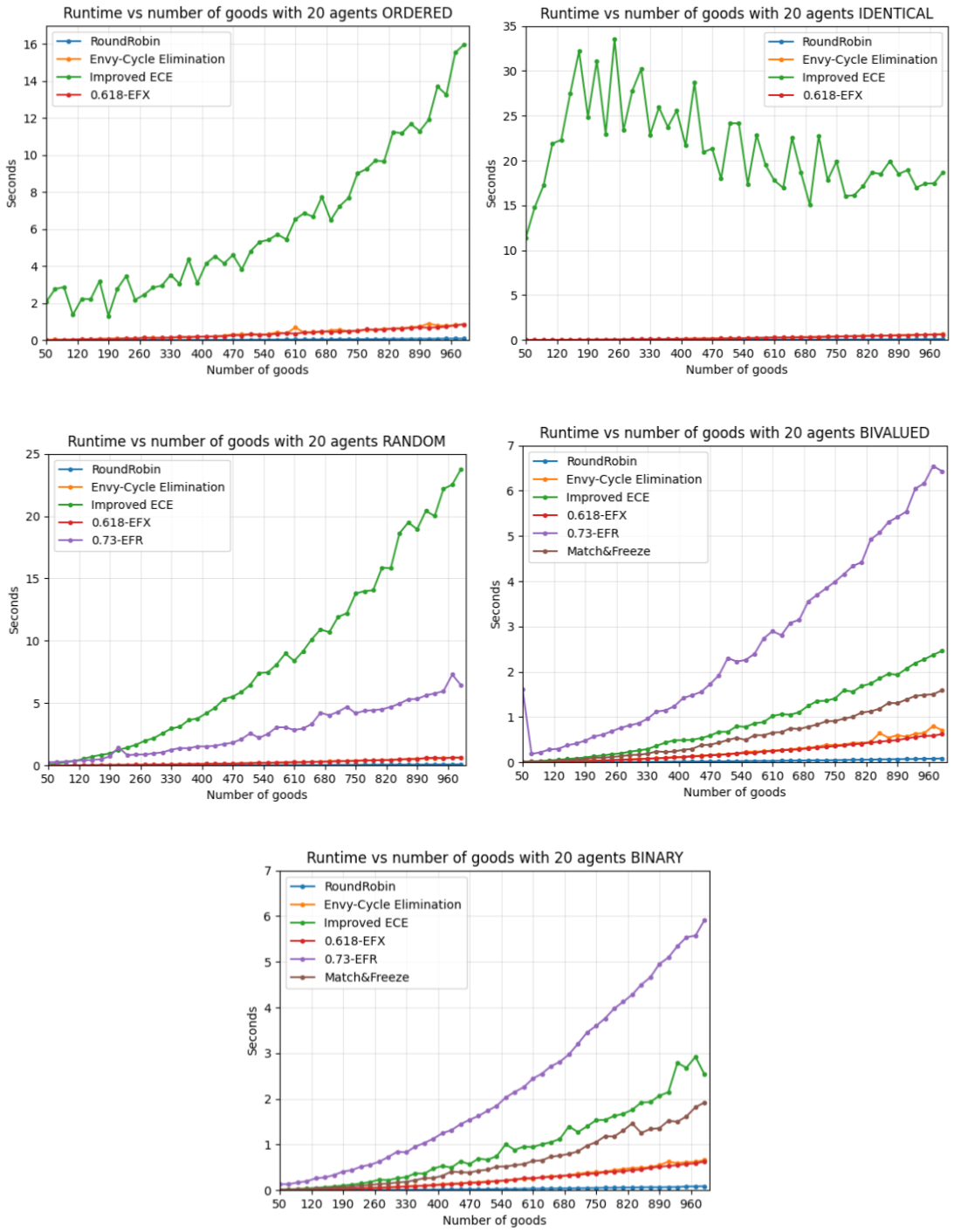

Figure 4.1: Running time experiment results with 20 agents and an increasing number of goods, per valuation type.

As expected, RoundRobin, being the simplest and least complex algorithm, has consistently low runtime and is barely affected both by the number of goods and the types of valuations as seen in Figure 4.1. ECE and 0.618 EFX have shown a small increase

in running time as the number of goods increases but they also remain unaffected by the different valuation types. Given that 0.618-EFX uses both RoundRobin and ECE, a comparatively small increase in running time compared to RoundRobin is expected. Match&Freeze has almost double the running time of ECE and 0.618-EFX consistently, but, interestingly remains one of the fastest algorithms. Especially when the number of goods is less than 190, Match&Freeze is incredibly fast (still slower than 0.618-EFX) meaning that it might be favoured over 0.618-EFX due to the better EFX performance.

0.73-EFR on the other hand, has a much greater running time compared to all the aforementioned and with Random, Bivalued and Binary instances, it remains unaffected given the differences in valuation types. Since EFR is a weaker notion compared to EFX and stronger than EF1, we can assume that having 0.73-EFR could also possibly result in the best case to 0.73-EFX, which is not far from 0.618-EFX especially when the running time of 0.73-EFR is also approximately six-times more than 0.618-EFX. Overall, a large running time of 0.73-EFR is expected, as computing the NSW allocation at the beginning and then finding the shortest paths in a complete graph, have very high time complexities.

Improved-ECE has very high running times and is most always the slowest out of our collection (we cannot make precise comparative conclusions as we do not have results for all valuation types for 0.73-EFR). This can easily be explained as finding a maximum weight matching and resolving all possible cycles in each iteration take a substantial amount of time. Interestingly, the main promise of Improved-ECE is that it guarantees 1/2-EFX but, it consistently takes more time to execute than 0.618-EFX which is also a better approximation. Therefore, we can conclude that if achieving a high EFX approximation is the goal then 0.618-EFX would not only yield a better approximation but it will also execute much faster.

Another interesting trend of Improved-ECE is that the running time differs a lot based on the type of valuations being used. Binary and bivalued, result in comparatively low running times, barely exceeding Match&Freeze whilst Ordered, Random and especially Identical instances take much longer (not even comparable to 0.618-EFX). One other interesting observation is that as the number of goods increases in the Identical case, the running for Improved-ECE also decreases when compared to the average trend of smaller number of goods. This is a very interesting behaviour that requires further investigation.

4.1.3 Results: fixed number of goods, varying number of agents

Based on the results as seen in Figure 4.2 and Figure A.6, RoundRobin remains again unaffected by all varying factors (both the number of agents and valuation types) and can therefore be concluded that it is the fastest EF1 algorithm. ECE still has almost a similar trend as with 0.618-EFX by having an overall very small execution time. However, this trend is not followed with Ordered instances, where there is a noticeable increase in execution time when we use 39 agents or more. We can therefore conclude that ECE is only limited by the number of agents, which is expected as the main component of the algorithm is the envy-graph which is comprised of agents as vertices, with that limit being 43 agents. The same conclusions can be drawn for 0.618-EFX with the only

difference being that in the Ordered case, it exceeds our maximum execution time limit at 39 agents with only a single other spike when the number of agents is 32. Overall, it can be concluded that ECE and 0.618-EFX execute in a very little amount of time when the number of agents is less than 39 with some very rare spikes in between.

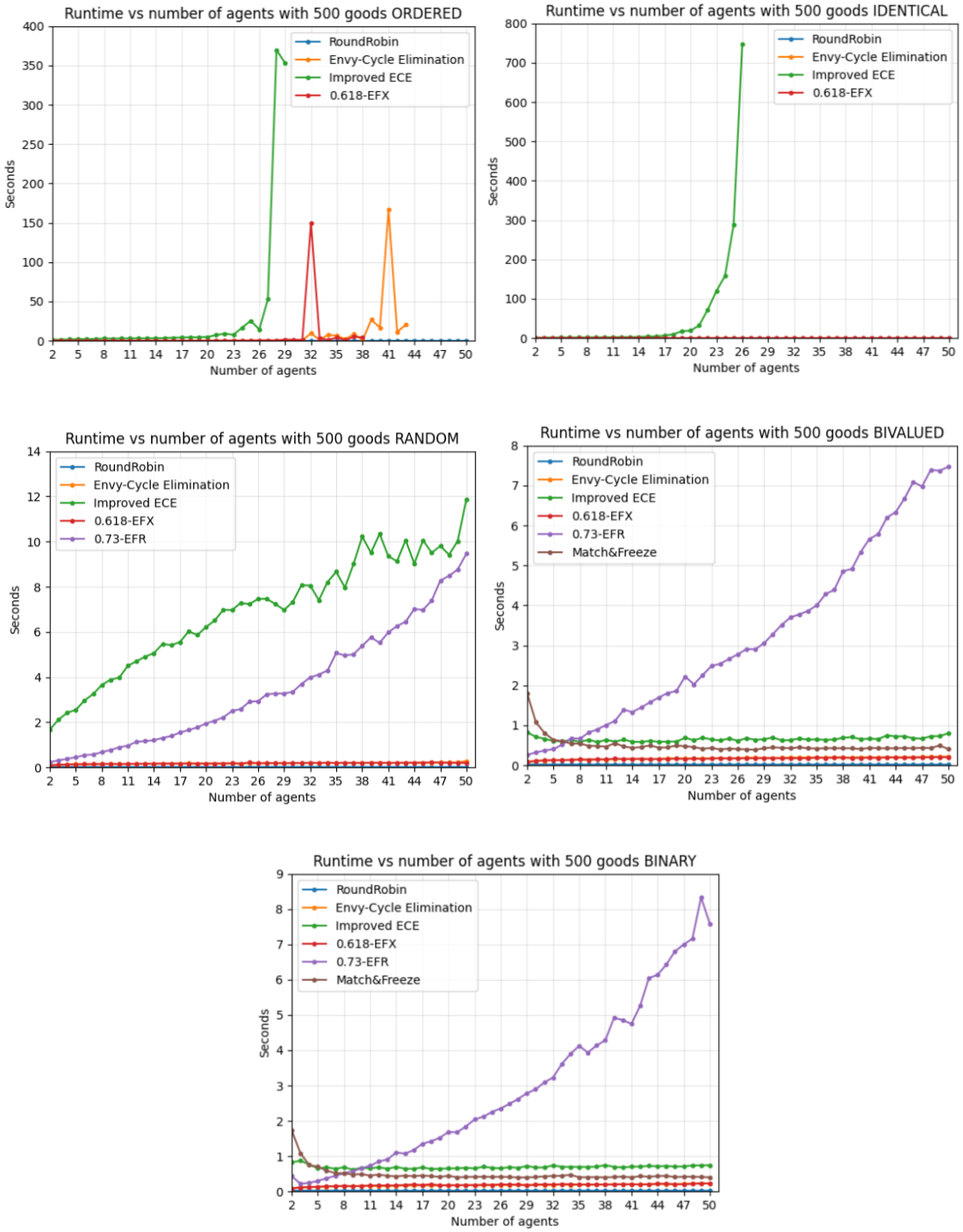

Figure 4.2: Running time experiment results with 500 goods and an increasing number of agents, per valuation type. For readers interested more in the behaviour of Identical instances without Improved-ECE, we have included a zoomed-in version in Figure A.6.

0.73-EFR again has some of the highest running times in all tested cases. Similarly to our last experiments, the rate of increase is consistent throughout all three types of valuations being tested.

Match&Freeze seems to have extremely consistent timings when the number of agents is more than 7. Interestingly, the execution time peaks when we only have 2 agents and slowly decreases as agents decrease with the most significant decrease happening in the agent range 2 to 7. Overall, it performs slightly worse than ECE and 0.618-EFX by always remaining under 1s when there are more than 3 agents. Given our results from both experiments, we can conclude that if EFX with Binary/Bivalued instances is the desired characteristic then Match&Freeze is an overall fast algorithm.

Improved ECE, has very varying performance based on the types of valuations used. Binary and Bivalued instances have the overall best performance as it never exceeds 1s and takes less time than Match&Freeze when the number of agents is less than 5. Once we surpass 5 agents, it performs slightly worse than Match&Freeze meaning that if we wish to prioritise EFX (and hence EF1 and EFR), for bivalued instances, no matter the number of goods or agents, Match&Freeze not only guarantees EFX but also achieves it faster than Improved-ECE (which only guarantees 1/2-EFX). On the other hand, when dealing with Random instances, we see that Improved-ECE is always extremely slower than 0.618-EFX so again, 0.618-EFX is the better choice for any number of agents and goods. When looking at the Ordered or Identical instances, Improved-ECE seems to be performing relatively well when the number of agents is less than 19 with the running time never exceeding 10 seconds. Once we surpass that threshold we notice a very aggressive, almost exponential, increase in execution time exceeding the 300-second range when there are more than 28 agents in Ordered instances, and 26 agents in Identical instances.

4.2 Fairness Criteria conformity experiments

4.2.1 Methodology

Test instances
To enable thorough testing of the algorithms, the testing dataset was comprised of two parts; one containing real-life instances and one containing synthetic data that cover the 5 types of valuations (Random, Ordered, Identical, Binary, Bivalued). Given that no one else had performed similar experiments, the behaviour of these algorithms was unknown. Therefore, splitting our test set into different categories provided a great starting point to help shine some light on the behaviours of each algorithm per specific type.

For real-life data, we were provided with all the instances that were submitted to Spliddit. That collection contains 2309 instances with varying numbers of goods and agents. One important characteristic of Spliddit instances is that each agent was given a ”budget” of 1000 to formulate their valuations. This means that the total sum of each agent’s valuations should equal 1000. Due to the nature of our algorithms (such as 0.73-EFR requiring an MNW allocation with one good per agent) and our goal of comparing

MNW approximations, we filtered out instances where it was not possible to allocate one good to each agent so that they do not have a total value of 0. This requirement decreased the number of applicable instances to 2066 which is still a high number of instances.

Synthetic instances were generated for the 5 aforementioned types but for Binary and Bivalued instances, we generated instances for both using uniform and normal distributions (increasing our types to 7). This decision was taken after we examined the results with the uniform distribution as they portrayed somewhat misleading results, even though the results are correct and valid. In total, with the inclusion of Spliddit data, we have 8 types.

The methodology for generating these instances remains the same as with the execution time experiments. We generated 400 instances for each type, with the number of agents ranging from 2 to 40 (chosen randomly with uniform distribution), the number of goods ranging from num_of_agents to 300 (again with uniform distribution) and the valuation of each good per agent ranged from 0 to 1000 (uniformly chosen). 400 instances were generated to provide a comprehensively large dataset that was also appropriately sized for a reasonable testing duration. As we have seen from the timing experiments, the larger the number of goods, the more the execution time for most of the algorithms, so, a compromise was made hence the 300 goods limitation. Similar reasoning was used for the number of agents leading to limiting the maximum number of agents to 40. However, as we already know some algorithms take an exceptionally long time with a high number of goods and agents (mainly with Ordered and Identical instances) so for some algorithms we had to filter out some of those larger instances. For all the other algorithms, all the instances were used and algorithms that used the restricted instances will be marked. We could have generated smaller instances for all algorithms but on the other hand, we would have intentionally limited our test set to relatively small instances. As with the Spliddit data, it was made sure that with each instance, it was possible to have an MNW allocation where each agent received a single good and no agent had a value of 0.

Assessment criteria
For each algorithm we recorded the following Key Performance Indicators (KPIs) for all 8 types of valuations (whenever possible):

Percentage of instances achieving EF1, EFX, EFR, MNW, 1.45-MNW, Approximation of α-EF1, α-EFX, α-EFR, α-MNW.

Finding a Maximum Nash Welfare allocation (MNW) is an NP-Hard problem so one might question how we calculated all the KPIs that involve MNW. Our approach was to formulate the problem in Integer Linear Programming (ILP) and use a solver, more specifically Gurobi (Gurobi, 2023) to find such optimal allocations. Maximising a product, however, is not formally a feature of ILP, so all agent valuations were converted to logarithms allowing us to exploit the behaviour of logarithms when they are being added as it is equivalent to multiplying the raw valuations. More formally, in our Gurobi environment, we had to define variables for allocations, valuations, constraints to convert the valuations to logarithms, variables to store the logarithmic valuations and one other final constraint to ensure that each good is only allocated to a single agent. In

addition, we have set Gurobi to use that maximum level of MIPFocus (3) and set the MIPGap to 0 to ensure that we only get the optimal solution. In addition, Gurobi's time limit was set to 30 seconds to find the optimal solution. Any instances where Gurobi was not able to find the optimal allocation within the time limit were removed from the KPIs that involve MNW. This restriction was necessary as even with the 30-second limit, Gurobi could take up to 47 hours to find an optimal allocation for all our instances. Understandably, this is not ideal, however, limited by computational resources and time allocation for this project, such a compromise had to be made. The formulation of the problem in ILP together with Gurobi's settings can be found in Algorithm 7.

KPIs that involve approximation (excluding MNW) were also time-consuming and challenging to calculate. To compensate, a binary search approach was implemented with a tolerance of 0.001, therefore the reader can assume that the true α approximation is within 0.001 of the reported approximation.

Availability of raw results and methods
All raw results were recorded, together with an instance ID, the resulting allocation, optimal MNW allocation according to Gurobi, a flag to signify if Gurobi reaches the time limit, execution time[1], and all the KPIs mentioned above. All raw data will be made publicly available for any researcher to use.

In addition, our fairness checkers (functions used to generate the KPIs) as well as our functions that can be used to generate instances based on desired parameters will also be made publicly available.

4.2.2 RoundRobin

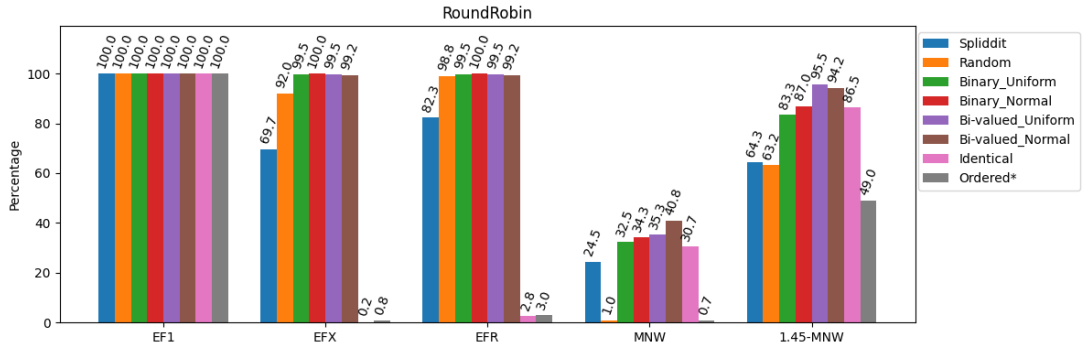

Figure 4.3: RoundRobin percentage of instances achieving specific fairness notions for each type of valuations.

Unsurprisingly, RoundRobin achieves EF1 (Figure 4.3), with almost all binary and bivalued instances also achieving EFX. To validate this observation we tested all possible binary valuations with two agents and six goods and found that 95% of instances achieve EFX. This is still a very promising result revealing a very interesting behaviour of

[1]Experiments were executed on the University of Edinburgh's Informatics Student Compute server. Resources might have been dynamically allocated and therefore, running times might not be consistent. For more information visit https://computing.help.inf.ed.ac.uk/compute-servers.

RoundRobin that we did not know. Unfortunately, no obvious pattern could be found for the cases that do not achieve EFX that could help lead to new theorems.

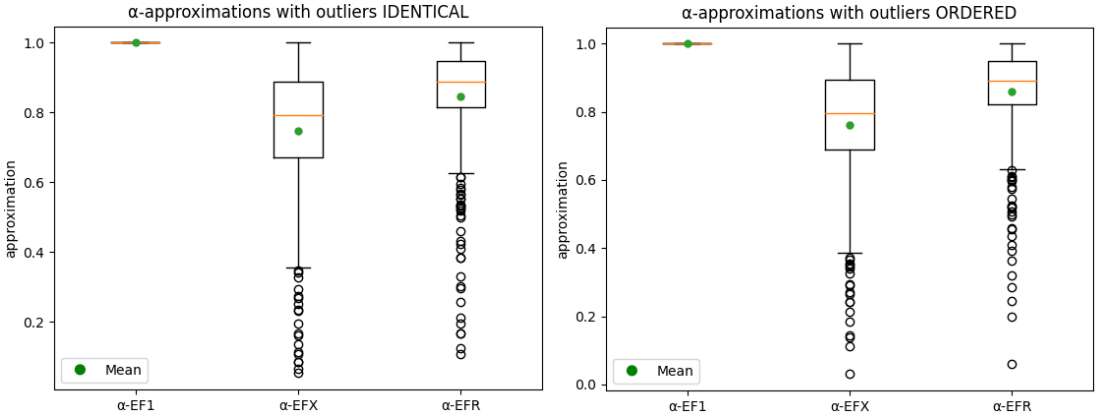

Figure 4.4: Approximations of EF1, EFX and EFR1 for Ordered and Identical instances using RoundRobin.

More surprisingly, it is also evident that with Identical and Ordered valuations RoundRobin struggles a lot to find EFX and even EFR allocations. Looking at the approximations for EFX and EFR for Ordered and Identical (Figure 4.4), it can be seen that there is a large variation and therefore no substantial conclusions can be drawn.

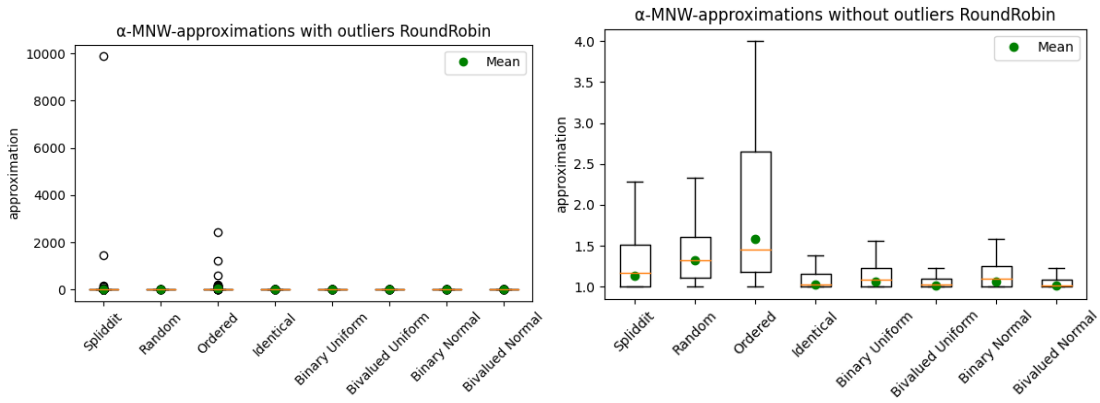

Figure 4.5: MNW approximations using RoundRobin, with and without outliers.

As for MNW approximations, when outliers are removed, Figure 4.5 shows that we have a maximum approximation of 4, however, this is not truly representative. When outliers are included, the maximum approximation increases to 10000. Unfortunately, again, no meaningful conclusions can be drawn as although we might display them as outliers, it is possible that a dataset could be comprised of solely similar cases.

4.2.3 Improved ECE

Ordered instances were limited to up to 20 agents and up to 200 goods (total 126 instances) and Identical instances to up to 25 agents and 300 goods (total 170 instances).

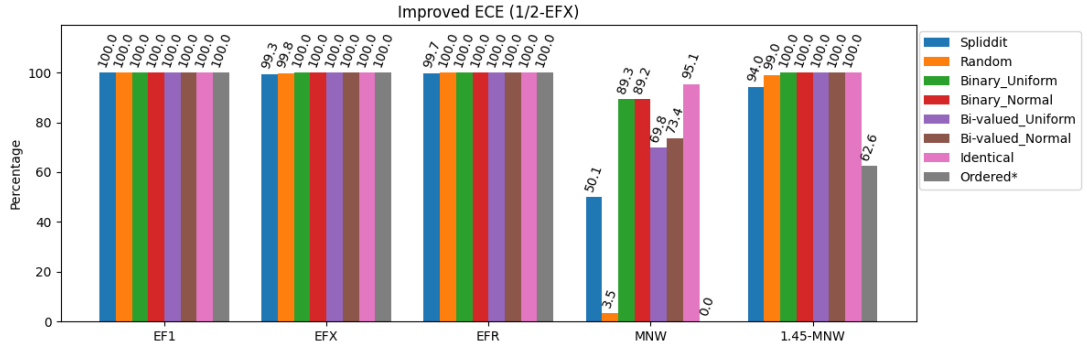

Figure 4.6: Improved-ECE percentage of instances achieving specific fairness notions for each type of valuations.

Improved-ECE overall performs extremely well as almost all instances achieve full EFX (and therefore EFR) and a large number of instances also achieve 1.45-MNW (Figure 4.6). Looking at the EFX approximations of Spliddit and Random instances (Figure 4.7), it is evident that the algorithm does achieve 1/2-EFX and surpasses that boundary for all of our test instances. This finding could mean that overall, one should expect on average better than 1/2-EFX approximation and possibly 1/2 for only some very specific cases.

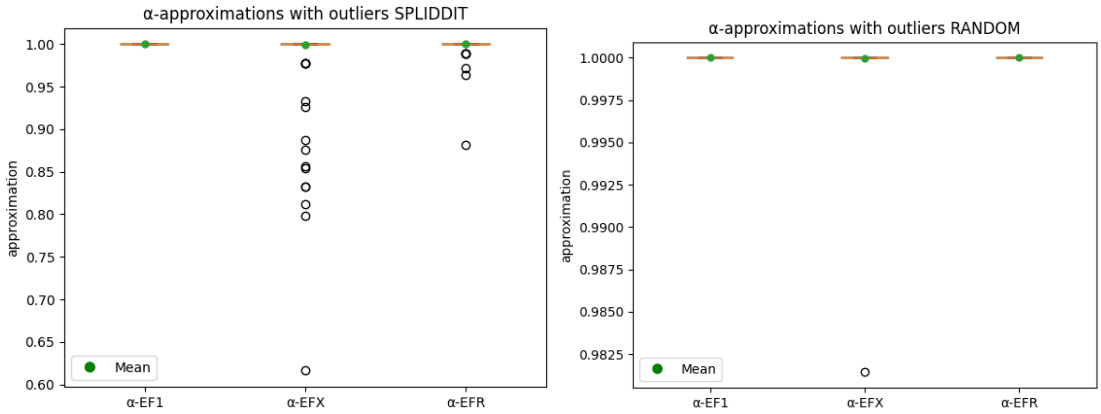

Figure 4.7: Approximations of EF1, EFX and EFR for Spliddit and Random instances using Improved-ECE.

MNW approximations are considerably better than RoundRobin (Figure 4.8), as the highest approximation was 4. However, some Spliddit instances resulted in at least one agent receiving an allocation valued at 0 (we wish to remind the reader that all instances tested can result in allocation where all agents receive a value more than 0). A 0 approximation in this graph indicates a 0 MNW and therefore the reader can treat this value as an infinity approximation. So, Improved-ECE generates overall extremely good MNW approximations with the exception of only a small number of cases where the approximation was infinity.

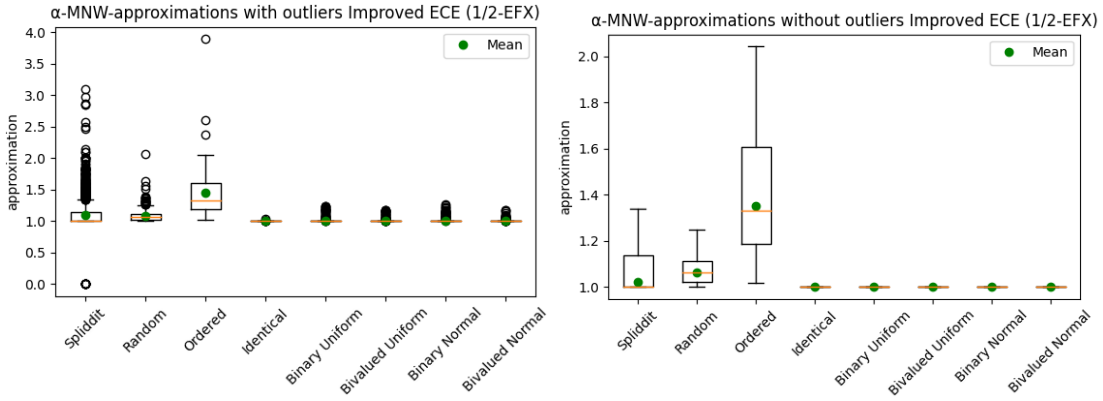

Figure 4.8: MNW approximations using Improved-ECE, with and without outliers. A 0 approximation indicates an infinity approximation.

One very crucial observation is that Improved-ECE has the best MNW approximations for all types of Binary and Bivalued instances compared to all our algorithms.

As a follow-up investigation, a modification of Improved-ECE was made to use ECE's cycle resolution (resolve cycles until an unenvied agent is found, rather than resolving all cycles) in an attempt to investigate if some time could be saved by not having to resolve all cycles. The results can be seen in Figure 4.9, which overall shows a decrease in performance mainly for MNW allocations with Identical instances and EFX allocations with Ordered/Identical instances. In addition, this modification has also lost the 1/2-EFX guarantee.

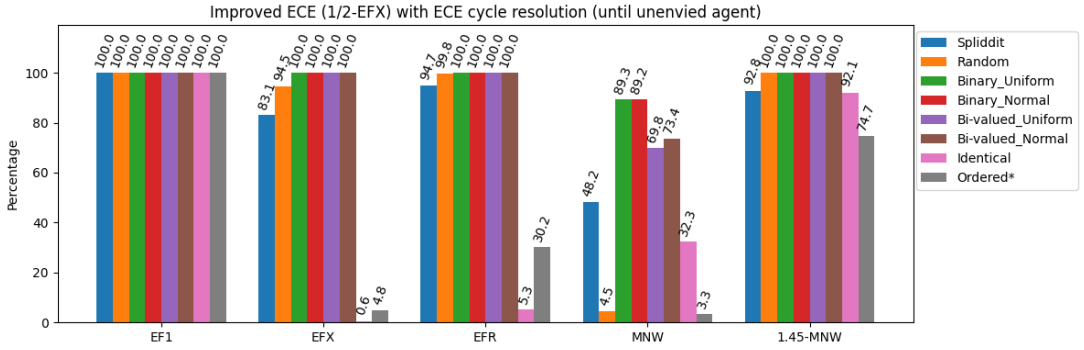

Figure 4.9: Improved-ECE (with ECE cycle resolution) percentage of instances achieving specific fairness notions for each type of valuations.

4.2.4 Envy-Cycle Elimination (ECE)

Ordered instances were limited to up to 20 agents and up to 200 goods (total 186 instances).

As expected, ECE achieves EF1 and EFX for Identical and Ordered instances (Figure 4.10). Based on the results from Figure 4.11, very promising MNW approximations can be seen for Bivalued instances and near-perfect approximations for Identical instances.

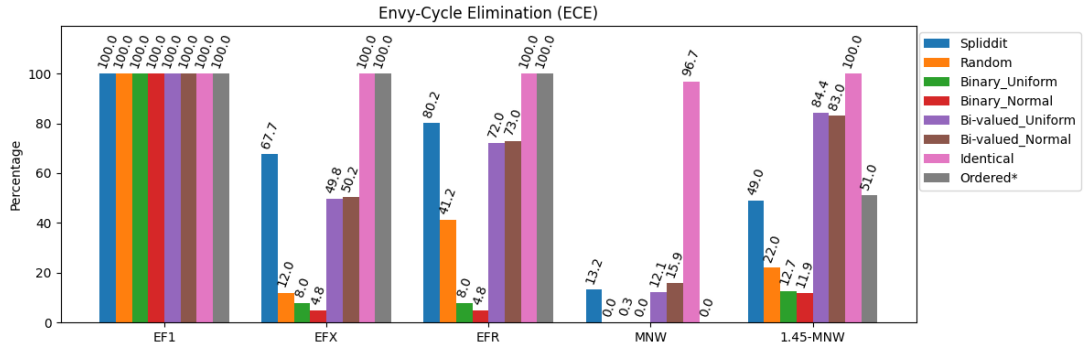

Figure 4.10: ECE percentage of instances achieving specific fairness notions for each type of valuations.

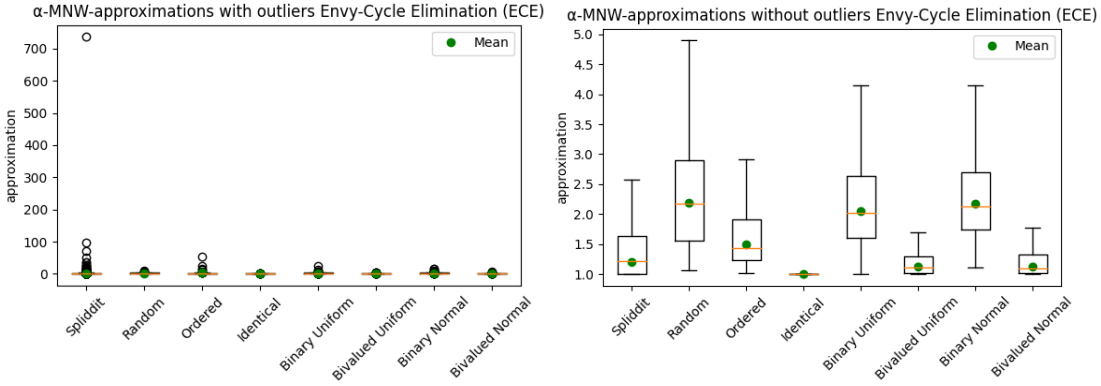

Figure 4.11: MNW approximations using ECE, with and without outliers.

It seems that ECE behaves extremely well with Bivalued instances but not with Binary instances (a specialisation of Bivalued).

EF1, EFR, and EFX approximations have not resulted in any note-worthy results and their graphs are included in Figure A.3. Based on the positive findings from Improved-ECE, it was investigated whether eliminating all cycles at each iteration or at the very end of the algorithm would result in better allocations. These results are again included in Figures A.1 and A.2, without any significant results.

4.2.5 0.618-EFX

Ordered instances were limited to up to 20 agents and up to 200 goods (total 186 instances).

From Figure 4.12, it is obvious that a promised, 0.618-EFX achieves EF1. One other observation is that it also achieves EFR for Identical and ordered instances. Despite knowing that this is an ECE property (which is part of the algorithm), it shows that the pre-processing and the two rounds of RoundRobin do not influence this property. As for MNW, we see slightly better results compared to ECE (Figure 4.13).

From Figure 4.17, it can be seen that 0.618-EFX achieves its promised EFX approxima-

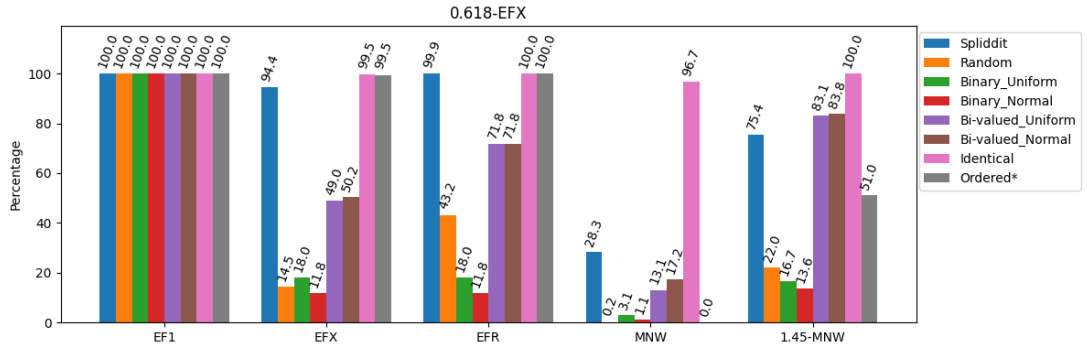

Figure 4.12: 0.618-EFX percentage of instances achieving specific fairness notions for each type of valuations.

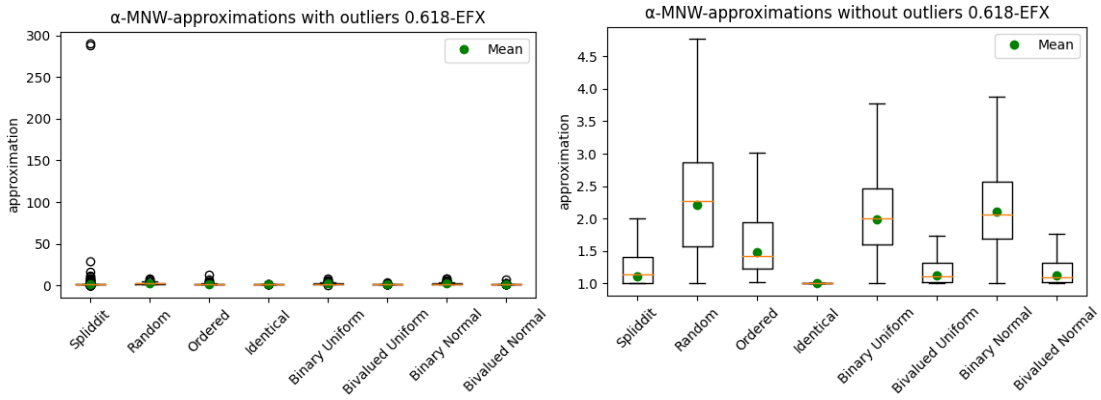

Figure 4.13: MNW approximations using 0.618-EFX, with and without outliers.

tion and possibly ensures a higher EFR approximation (greater than 0.73).

In addition to the aforementioned, a further experiment was carried out where the ECE component of 0.618-EFX was replaced with Improved-ECE. This change caused overall better results than the original 0.618-EFX but, worse results than the plain Improved-ECE (Figures A.4 and A.5).

4.2.6 0.73-EFR

Ordered and Identical instances were limited to up to 20 agents and up to 200 goods (total of 186 instances for each type).

0.73-EFR achieves EF1 as seen in Figure 4.14, but also seems to follow the trend of achieving full EFR for Identical and Ordered valuations as most of the other algorithms. Interestingly, 0.73-EFR has the best overall MNW approximation reaching only a maximum of 8. It might not have the most instances achieving MNW but it has the smallest MNW approximation so far (Figure 4.15).

From Figure 4.18 it can also be verified that it achieves its promised 0.73-EFR approximation.

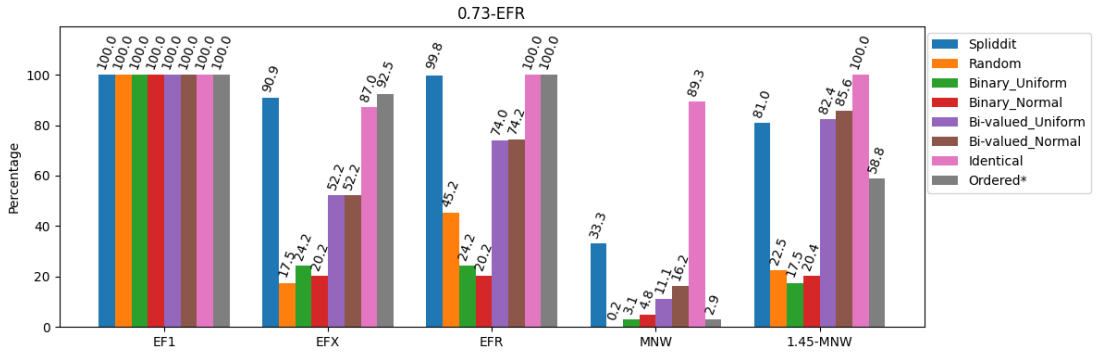

Figure 4.14: 0.73-EFR percentage of instances achieving specific fairness notions for each type of valuations.

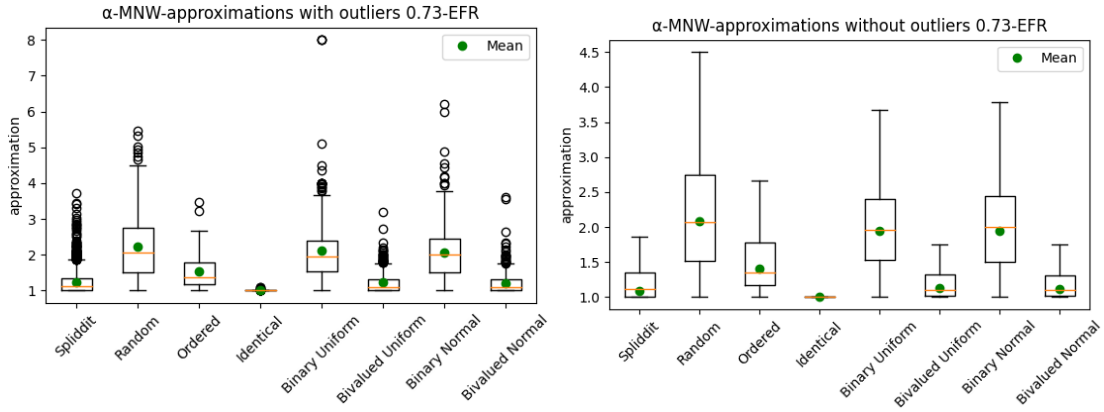

Figure 4.15: MNW approximations using 0.73-EFR, with and without outliers.

4.2.7 Match&Freeze

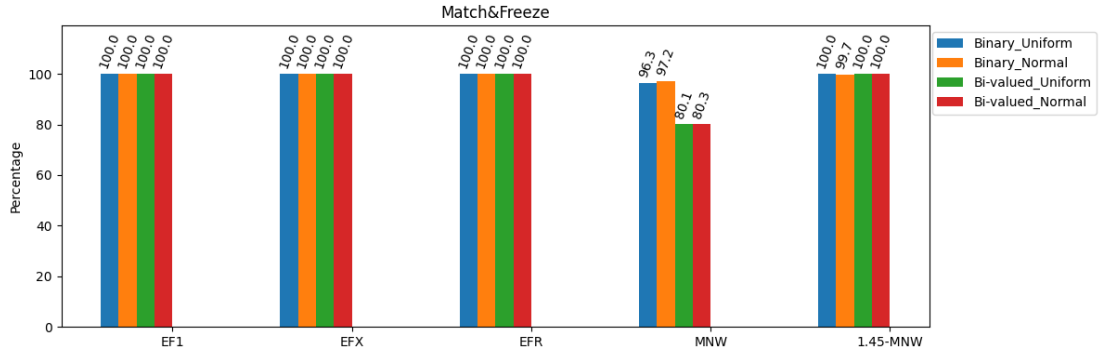

Figure 4.16: Match&Freeze percentage of instances achieving specific fairness notions for each type of valuations.

Match&Freeze performs as expected by achieving EFX (Figure 4.16), and also has the second best MNW approximations for binary and bivalued instances (Figure 4.19).

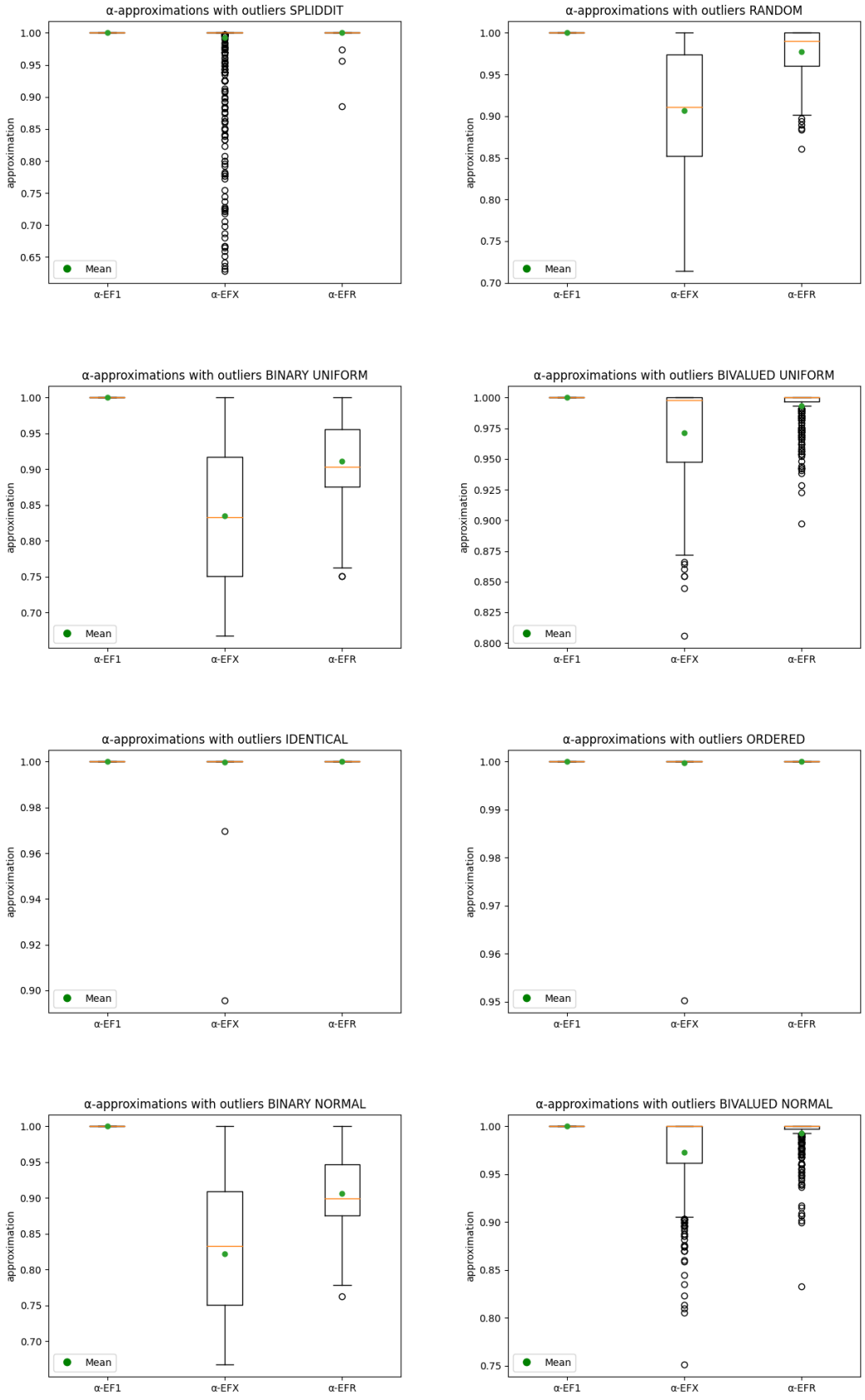

Figure 4.17: 0.618-EFX EF1, EFR and EFX approximations.

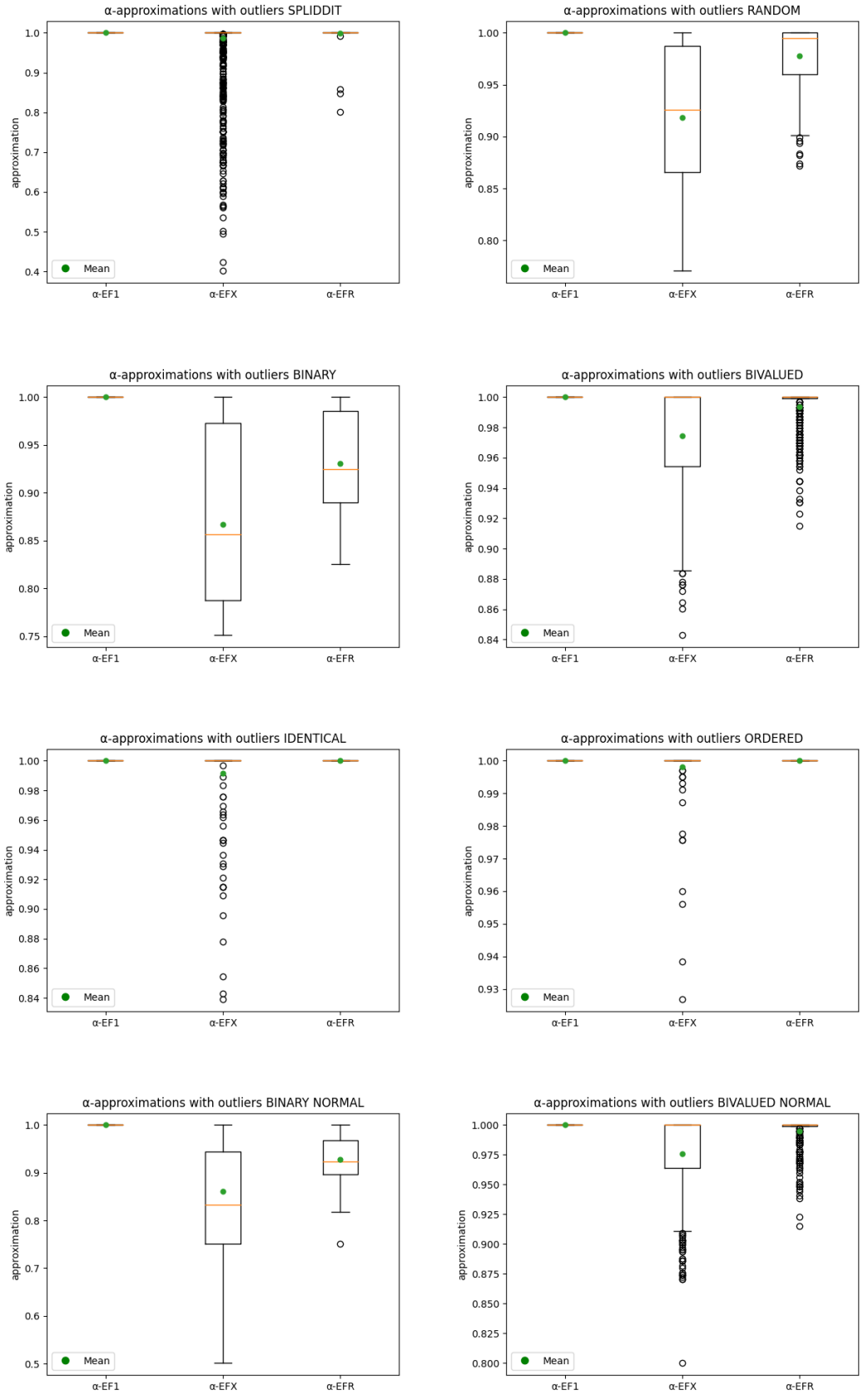

Figure 4.18: 0.73-EFR EF1, EFR and EFX approximations.

4.2.8 Leximin++

Due to the exponential running time of Leximin++, smaller instances were generated only for this algorithm. Therefore, no comparison will be made with the other algorithms to ensure fair treatment.

As expected, Leximin++ achieves EFX for Ordered Instances (Figure 4.20) but, also achieves very good MNW approximations at the same time (Figure 4.21).

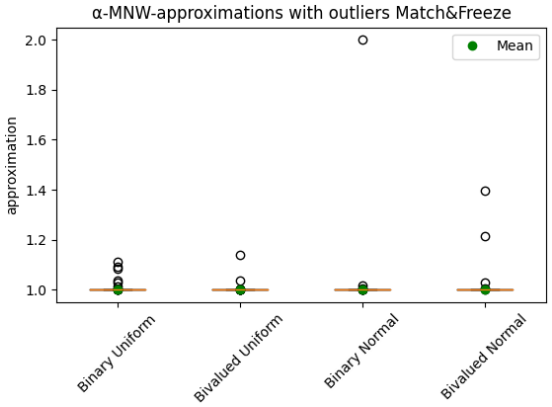

Figure 4.19: MNW approximations using Match&Freeze, with outliers.

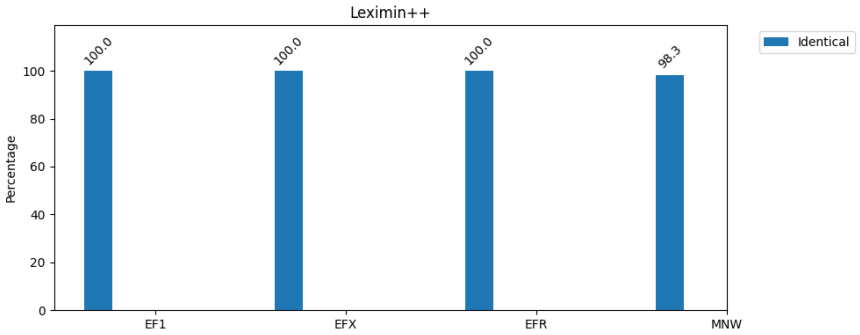

Figure 4.20: Leximin++ percentage of instances achieving specific fairness notions for each type of valuations.

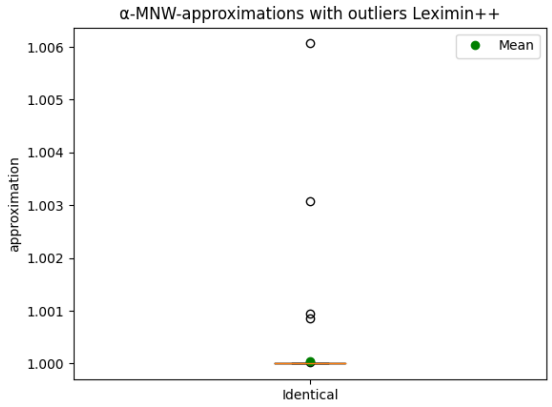

Figure 4.21: MNW approximations using Leximin++, with outliers.

Chapter 5

Conclusions

In the course of this research, an array of algorithms was implemented, encountering many challenges regarding the feasibility of their implementation, key missing components and possible flaws which all required an instrumental amount of time to detect and solve. Through strategic testing, a guide for future researchers has been produced. Furthermore, several key findings suggest the emergence of novel theorems, whose formal validation could lead to new performance guarantees for some algorithms.

5.1 Key findings & Future work

Concentrating on the implementation phase, this project identified a crucial attribute that seems to be overlooked by algorithm designers; the feasibility of implementing the algorithm. Through our description of 0.73-EFR, we have showcased how precision errors, due to the nature of programming languages, can cause significant and unsolvable problems making the algorithms infeasible to use with a computer. While we detail our mitigation strategies in Section 3.5.2, further investigation is warranted to provide a less computationally expensive solution.

One of our most significant findings was the identification of potential problems (as well as algorithmic omissions) with the current state-of-the-art algorithm for achieving an MNW approximation (1.45-MNW). Counter-examples were identified, and an analysis of potential causes of this abnormal behaviour was provided which will be sent to the authors of the paper.

Utilising our efficient implementations, we now have a comprehensive collection of implemented algorithms, a majority of which had not been implemented before, available for anyone to use (in the future as a Python library). It became evident that some algorithms lacked essential implementation details which although we overcame, serves as a reminder for future algorithm designers to not only establish polynomial time guarantees but also, to elucidate practical implementation recommendations. Complimenting our collection of algorithms, we also provide fairness checkers to check if an allocation achieves EF1, EFX, EFR and their approximations as well as MNW (and its approximation) which was implemented through our ILP implementation using Gurobi.

From our experimental results, we have devised a guide (Table 5.2) to assist researchers in selecting the best algorithm tailored to their requirements. The guide is influenced both by the algorithms' fairness observations as well as their execution time. Further findings include valuation restrictions that could influence running time, as well as how the number of goods or agents influences the decision as to which algorithm to choose.

Finally, three key findings from our experiments have unveiled previously unknown behaviour with certain algorithms, which could lead to new theorems as to the algorithms' guarantees. This constitutes a promising avenue for future research. Overall, we have demonstrated the potential impact of conducting experiments on such algorithms and we encourage future researchers to explore targeted instances for further insights.

Our experimental results can be found in Table 5.2, whilst in Table 5.1 we briefly summarise our observations on the key implementation details of 0.73-EFR, 1.45-MNW and Leximin++.

5.2 Limitations

Reflecting upon the work outlined in this project, it is evident that a lot of compromises needed to be made in certain sections.

Starting with algorithm selection, this aspect was constrained secondary to the complexity and challenges identified when implementing some of those algorithms which required more time than initially anticipated. However, we are confident that we have included the most prominent algorithms as identified in the literature, nonetheless, we appreciate that a larger selection could have provided broader insight.

In the implementation phase, we have identified limitations with regard to precision errors, specifically in 0.73-EFR and 1.45-MNW. Although we have attempted to address them, the best solutions we could and have provided, often come with increased running time or limitations on valuation values.

Although our test set covers a broad selection of valuation types and contains numerous instances, including real-life data (totalling 5594 instances), it lacks targeted instances aimed at challenging the algorithms further. Our approach was to split possible valuations into certain types and then generate (at random) a large collection of instances for each. Given that this is the first time that such experiments with these algorithms were conducted, there were no initial indications as to what to expect. Therefore, focusing on targeted instances first, seemed like a step ahead of what is currently known.

It is also understandable that limiting the instances for some valuation types and only for some algorithms could make it more challenging to draw conclusions but, limited by computational resources, this decision, unfortunately, had to be made.

Owing to time constraints, previously cited delays, and extensive testing duration, particularly evident in large instances, further exhaustive testing was precluded. Our focus was directed towards elucidating potential high-level correlations among the algorithms, notably examining distinct methodologies for cycle resolution in ECE and their consequential impact on ECE itself and other algorithms reliant on this component.

A final limitation was the execution time results for 0.73-EFR (for some valuation types) which we identified that it was impossible to implement, as well as, any results for 1.45-MNW which we decided to remove in order to maintain an unbiased report.

New Discovery	**Remarks**
0.73-EFR cannot be implemented with the original algorithm in Python	Due to decimal precision challenges, running Bellman-Ford is not always possible. A solution was identified but at the cost of extra execution time.
A potential flaw with 1.45-MNW	1.45-MNW suffers from decimal precision challenges, lacks key implementation details and, most importantly, seems to miss its MNW approximation guarantee. We have provided a counter-example and identified a possible source of the problem. Our investigation will be sent to the authors of the paper.
Our leximin++ implementation is more efficient in comparison to generating all possible allocations	Given that the authors did not provide any guidance as to how to generate all possible allocations, based on preliminary testing, our branch pruning implementation performs extremely well (compared to naive generation).

Table 5.1: Discoveries through implementations of algorithms and future work.

Requirement/Discovery	**Recommendation/Remark**
High probability EFX Bivalued/Binary	RoundRobin (faster than Match&Freeze which guarantees EFX)
High probability EFX	Improved-ECE
Good on average MNW approximations	Improved-ECE
1.5-MNW for Binary, Bivalued	Match&Freeze (faster) or Improved-ECE (also for Identical)
Near-perfect MNW with Identical	ECE/0.618-EFX (faster) or Leximin++ or 0.73-EFR
Full EFR for Identical or Ordered	0.618-EFX or ECE (both faster) or 0.73-EFR **0.618-EFX and 0.73-EFR require formal proof**
$\geq$0.73-EFR approximation	0.618-EFX (faster, **Requires formal proof**) or 0.73-EFR
High EFX approximation for all types	0.618-EFX (Faster than Improved ECE)
Improved-ECE gets faster as the number of goods increase (Identical val.)	**Needs further investigation**
Match&Freeze unaffected by the number of agents (Contrary to the number of goods) and faster than Improved-ECE	
0.618-EFX and ECE execution time under 400s with ordered instances	Limit number of agents to 39 and 43 respectively
Improved-ECE with Ordered/Identical valuations	Exponential-like running time when more than 26 agents
Binary/Bivalued instances that require approximate-EFX allocations	Use Match&Freeze when the number of goods$<$190 to get full EFX as it has almost the same running time as 0.618-EFX
EF1 allocations of any size and type	Use RoundRobin (always the fastest)

Table 5.2: Key discoveries from experiment observations and future work.

Bibliography

Hannaneh Akrami, Noga Alon, Bhaskar Ray Chaudhury, Jugal Garg, Kurt Mehlhorn, and Ruta Mehta. Efx: A simpler approach and an (almost) optimal guarantee via rainbow cycle number. In *Proceedings of the 24th ACM Conference on Economics and Computation*, EC '23, page 61, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701047. doi: 10.1145/3580507.3597799. URL https://doi.org/10.1145/3580507.3597799.

Georgios Amanatidis, Evangelos Markakis, and Apostolos Ntokos. Multiple birds with one stone: Beating 1/2 for efx and gmms via envy cycle elimination. *Theoretical Computer Science*, 841:94–109, 2020. ISSN 0304-3975. doi: https://doi.org/10.1016/j.tcs.2020.07.006. URL https://www.sciencedirect.com/science/article/pii/S0304397520303790.

Georgios Amanatidis, Georgios Birmpas, Aris Filos-Ratsikas, Alexandros Hollender, and Alexandros A. Voudouris. Maximum nash welfare and other stories about efx. *Theoretical Computer Science*, 863:69–85, April 2021. ISSN 0304-3975. doi: 10.1016/j.tcs.2021.02.020.

Georgios Amanatidis, Haris Aziz, Georgios Birmpas, Aris Filos-Ratsikas, Bo Li, Hervé Moulin, Alexandros A. Voudouris, and Xiaowei Wu. Fair division of indivisible goods: Recent progress and open questions. *Artificial Intelligence*, 322:103965, sep 2023. doi: 10.1016/j.artint.2023.103965. URL https://doi.org/10.1016%2Fj.artint.2023.103965.

Haris Aziz and Simon Mackenzie. A Discrete and Bounded Envy-Free Cake Cutting Protocol for Any Number of Agents, August 2017. URL http://arxiv.org/abs/1604.03655. arXiv:1604.03655 [cs].

Siddharth Barman, Sanath Kumar Krishnamurthy, and Rohit Vaish. Finding fair and efficient allocations. In *Proceedings of the 2018 ACM Conference on Economics and Computation*, EC '18, page 557–574, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358293. doi: 10.1145/3219166.3219176. URL https://doi.org/10.1145/3219166.3219176.

S.J. Brams and A.D. Taylor. *Fair Division: From Cake-Cutting to Dispute Resolution*. Fair Division: From Cake-cutting to Dispute Resolution. Cambridge University Press, 1996. ISBN 9780521556446. URL https://books.google.co.uk/books?id=cLUA-sRhJ5QC.

Eric Budish. The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes. *Journal of Political Economy*, 119(6):1061–1103, 2011. ISSN 00223808, 1537534X. URL http://www.jstor.org/stable/10.1086/664613.

Ioannis Caragiannis, Christos Kaklamanis, Panagiotis Kanellopoulos, and Maria Kyropoulou. The efficiency of fair division. *Theor. Comp. Sys.*, 50(4):589–610, may 2012. ISSN 1432-4350. doi: 10.1007/s00224-011-9359-y. URL https://doi.org/10.1007/s00224-011-9359-y.

Ioannis Caragiannis, David Kurokawa, Hervé Moulin, Ariel D. Procaccia, Nisarg Shah, and Junxing Wang. The unreasonable fairness of maximum nash welfare. *ACM Trans. Econ. Comput.*, 7(3), sep 2019. ISSN 2167-8375. doi: 10.1145/3355902. URL https://doi.org/10.1145/3355902.

Hau Chan, Jing Chen, Bo Li, and Xiaowei Wu. Maximin-aware allocations of indivisible goods. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 137–143. International Joint Conferences on Artificial Intelligence Organization, 7 2019a. doi: 10.24963/ijcai.2019/20. URL https://doi.org/10.24963/ijcai.2019/20.

Hau Chan, Jing Chen, Bo Li, and Xiaowei Wu. Maximin-aware allocations of indivisible goods. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 137–143. International Joint Conferences on Artificial Intelligence Organization, 7 2019b. doi: 10.24963/ijcai.2019/20. URL https://doi.org/10.24963/ijcai.2019/20.

Vincent Conitzer, Rupert Freeman, Nisarg Shah, and Jennifer Wortman Vaughan. Group fairness for the allocation of indivisible goods. *Proceedings of the AAAI Conference on Artificial Intelligence*, 33(01):1853–1860, Jul. 2019. doi: 10.1609/aaai.v33i01.33011853. URL https://ojs.aaai.org/index.php/AAAI/article/view/4010.

Python Decimal. Decimal - decimal fixed point and floating point arithmetic, Mar 2024. URL https://docs.python.org/3/library/decimal.html.

Alireza Farhadi, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, Sébastien Lahaie, David Pennock, Masoud Seddighin, Saeed Seddighin, and Hadi Yami. Fair allocation of indivisible goods to asymmetric agents. *J. Artif. Int. Res.*, 64(1):1–20, jan 2019. ISSN 1076-9757. doi: 10.1613/jair.1.11291. URL https://doi.org/10.1613/jair.1.11291.

Alireza Farhadi, MohammadTaghi Hajiaghayi, Mohamad Latifian, Masoud Seddighin, and Hadi Yami. Almost envy-freeness, envy-rank, and nash social welfare matchings. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(6):5355–5362, May 2021. doi: 10.1609/aaai.v35i6.16675. URL https://ojs.aaai.org/index.php/AAAI/article/view/16675.

Rupert Freeman, Sujoy Sikdar, Rohit Vaish, and Lirong Xia. Equitable allocations of indivisible chores. *CoRR*, abs/2002.11504, 2020. URL https://arxiv.org/abs/2002.11504.

G. Gamow and M. Stern. *Puzzle-math*. Viking Press, 1958. ISBN 9780670583355. URL https://books.google.co.uk/books?id=_vdytgAACAAJ.

Jugal Garg and Aniket Murhekar. Computing fair and efficient allocations with few utility values. In *Algorithmic Game Theory: 14th International Symposium, SAGT 2021, Aarhus, Denmark, September 21–24, 2021, Proceedings*, page 345–359, Berlin, Heidelberg, 2021. Springer-Verlag. ISBN 978-3-030-85946-6. doi: 10.1007/978-3-030-85947-3_23. URL https://doi.org/10.1007/978-3-030-85947-3_23.

Jugal Garg, Edin Husić, Aniket Murhekar, and László Végh. Tractable fragments of the maximum nash welfare problem, 2022.

Paul W. Goldberg, Kasper Høgh, and Alexandros Hollender. The frontier of intractability for efx with two agents, 2023.

Jonathan Goldman and Ariel D. Procaccia. Spliddit: unleashing fair division algorithms. *SIGecom Exch.*, 13(2):41–46, jan 2015. doi: 10.1145/2728732.2728738. URL https://doi.org/10.1145/2728732.2728738.

Laurent Gourvès, Jérôme Monnot, and Lydia Tlilane. Near fairness in matroids. In *Proceedings of the Twenty-First European Conference on Artificial Intelligence*, ECAI'14, page 393–398, NLD, 2014. IOS Press. ISBN 9781614994183.

Gurobi. Gurobi optimizer, Sep 2023. URL https://www.gurobi.com/solutions/gurobi-optimizer/.

Aric Hagberg. Networkx documentation, Oct 2023. URL https://networkx.org/.

Hadi Hosseini, Joshua Kavner, Sujoy Sikdar, Rohit Vaish, and Lirong Xia. Hide, not seek: Perceived fairness in envy-free allocations of indivisible goods, 2023.

David Kurokawa, Ariel D. Procaccia, and Junxing Wang. Fair enough: Guaranteeing approximate maximin shares. *J. ACM*, 65(2), feb 2018. ISSN 0004-5411. doi: 10.1145/3140756. URL https://doi.org/10.1145/3140756.

R. J. Lipton, E. Markakis, E. Mossel, and A. Saberi. On approximately fair allocations of indivisible goods. In *Proceedings of the 5th ACM Conference on Electronic Commerce*, EC '04, page 125–131, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581137710. doi: 10.1145/988772.988792. URL https://doi.org/10.1145/988772.988792.

Liad Nagi and Moriya Elgrabli. Comparative study in fair division algorithms, 2022.

Benjamin Plaut and Tim Roughgarden. Almost envy-freeness with general valuations. *SIAM Journal on Discrete Mathematics*, 34(2):1039–1068, 2020. doi: 10.1137/19M124397X. URL https://doi.org/10.1137/19M124397X.

Fractions Python. Fractions - rational numbers, Mar 2024. URL https://docs.python.org/3/library/fractions.html.

H. Steinhaus. Sur la division pragmatique. *Econometrica*, 17:315–319, 1949. ISSN 00129682, 14680262. URL http://www.jstor.org/stable/1907319.

Walter Stromquist. How to cut a cake fairly. *The American Mathematical Monthly*, 87(8):640–644, 1980. ISSN 00029890, 19300972. URL http://www.jstor.org/stable/2320951.

Warut Suksompong and Nicholas Teh. On Maximum Weighted Nash Welfare for Binary Valuations. Papers 2204.03803, arXiv.org, April 2022. URL https://ideas.repec.org/p/arx/papers/2204.03803.html.

Hal R Varian. Equity, envy, and efficiency. *Journal of Economic Theory*, 9(1): 63–91, 1974. ISSN 0022-0531. doi: https://doi.org/10.1016/0022-0531(74)90075-1. URL https://www.sciencedirect.com/science/article/pii/0022053174900751.

Toby Walsh. Fair division: The computer scientist's perspective. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 4966–4972. International Joint Conferences on Artificial Intelligence Organization, 7 2020. doi: 10.24963/ijcai.2020/691. URL https://doi.org/10.24963/ijcai.2020/691. Survey track.

Appendix A

Additional results

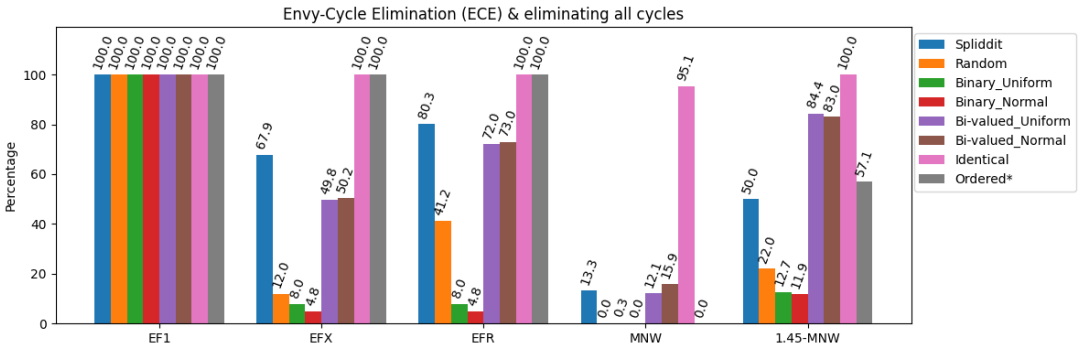

Figure A.1: ECE with all cycles being eliminated at each iteration.

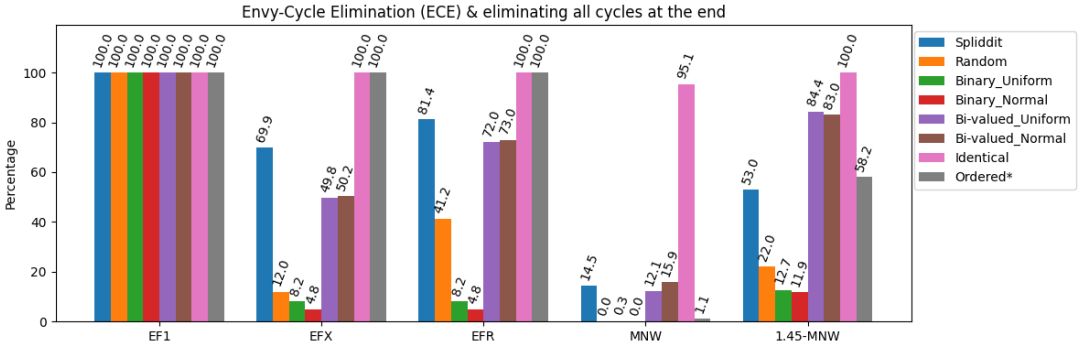

Figure A.2: ECE with all cycles being eliminated at the very end of the algorithm.

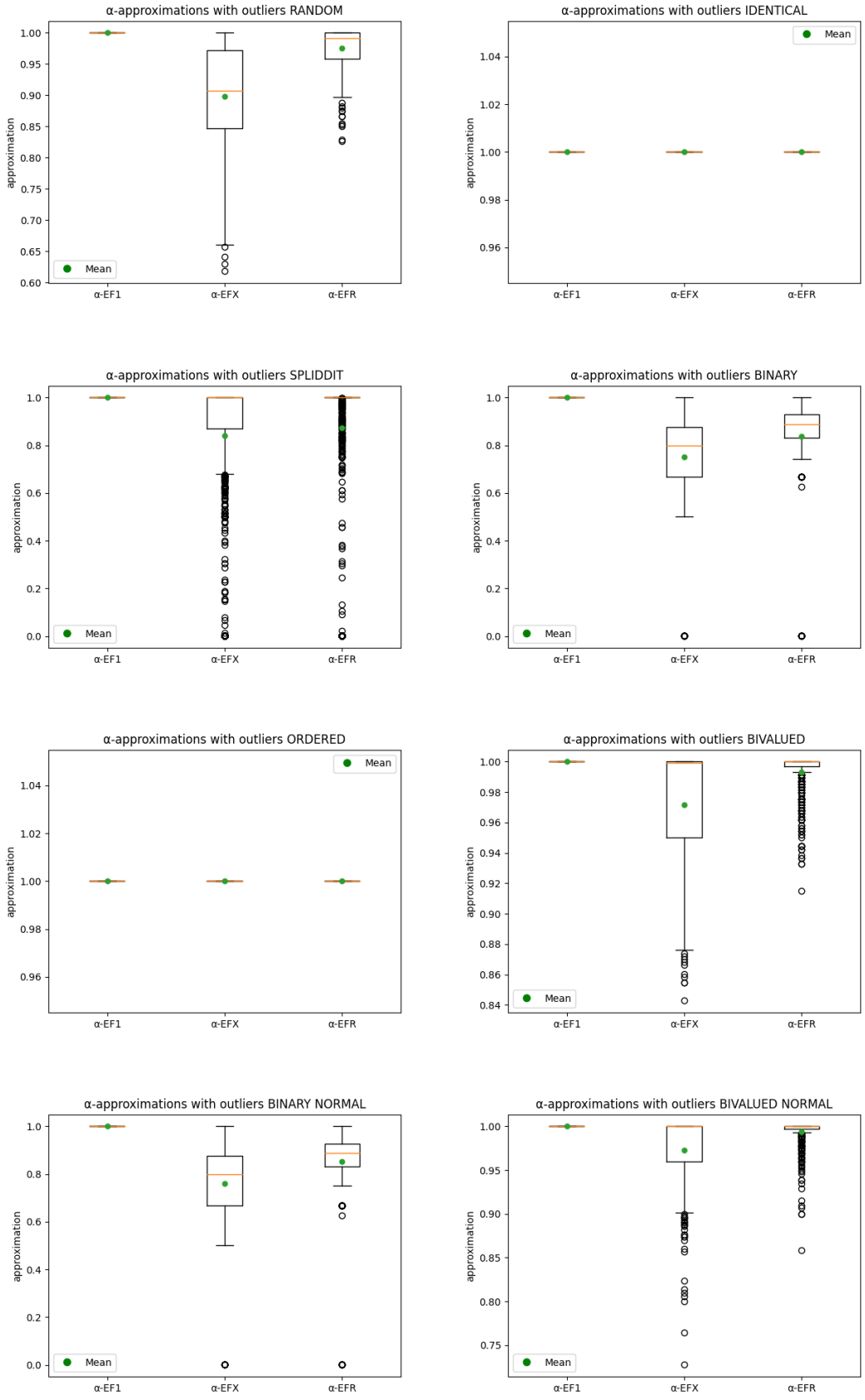

Figure A.3: ECE EF1, EFR and EFX approximations.

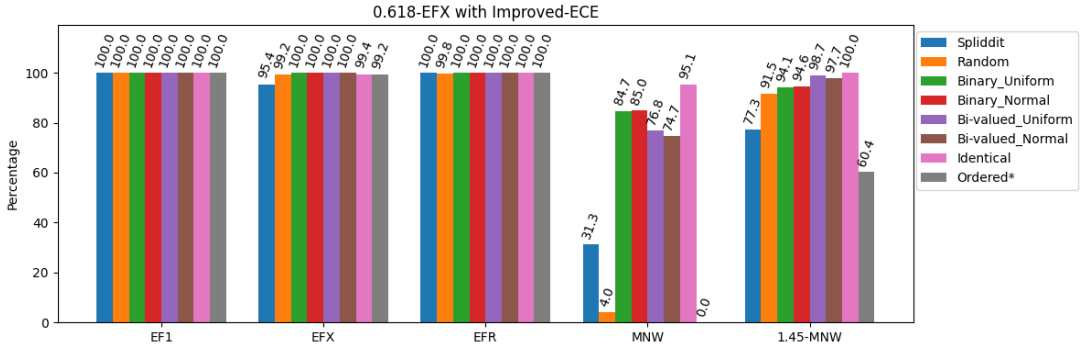

Figure A.4: 0.618-EFX with ECE component replaced with Improved-ECE.

Figure A.5: 0.618-EFX EF1, EFX, AND EFR approximations with ECE component replaced with Improved-ECE

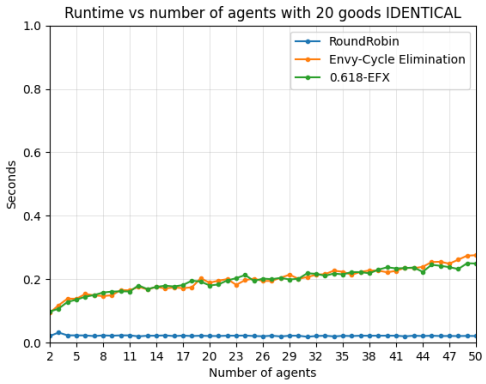

Figure A.6: Running time experiment results with 500 goods and increasing number of agents, per valuation type. Experiment on Identical instances without Improved-ECE.

Appendix B

Pseudocodes

Algorithm 1 Round-Robin

Input: A fair allocation instance $I = (N, M, V)$ with n agents and m goods.
Output: Allocation $A = (A_1, \ldots, A_n)$.
for $i = 1$ to n **do**
    $A_i \leftarrow \emptyset$
end for
for $l = 1$ to m **do**
    Let $i = l \mod n$
    Let $g^* = \arg\max_{g \in M} V_i(g)$
    $A_i \leftarrow A_i \cup \{g^*\}$
    $M \leftarrow M \setminus \{g^*\}$
end for

Algorithm 2 Envy-Cycle Elimination

Input: A fair allocation instance $I = (N, M, V)$ with n agents and m goods.
Output: Allocation $A = (A_1, ..., A_n)$.
for each agent $i \in N$ **do**
    $A_i \leftarrow \emptyset$;
end for
for $l = 1, ..., m$ **do**
    while there does not exist an unenvied agent **do**
        Find an envy-cycle $C = (i_1, ..., i_d)$ and resolve the cycle as follows:

$$A_{i_j}^C = \begin{cases} A_{i_{j+1}}, & \text{for all } 1 \leq j < d-1 \\ A_{i_1}, & \text{for } j = d \end{cases}$$

        $A_i \leftarrow A_i^C$ for all i in C
    end while
    Let i be an unenvied agent
    Let $g^* \in \arg\max_{g \in M} \{V_i(g)\}$
    $A_i \leftarrow A_i \cup \{g^*\}$
    $M \leftarrow M \setminus \{g^*\}$
end for

Algorithm 3 Find Cycles

```
1:  function FINDCYCLES
2:      procedure DFS(node, start, visited, path, max_depth)
3:          if length of path > max_depth then
4:              return None                ▷ Indicate no cycle found within max depth
5:          end if
6:          visited.add(node)
7:          path.append(node)
8:          for neighbor in graph[node] do
9:              if neighbor == start then
10:                 return path + [start] ▷ Return cycle found (including starting point)
11:             else if neighbor not in visited then
12:                 cycle ← DFS(neighbor, start, visited, path, max_depth)
13:                 if cycle ≠ None then
14:                     return cycle       ▷ Cycle found, stop further exploration
15:                 end if
16:             end if
17:         end for
18:         path.pop()                                              ▷ Revert path
19:         visited.remove(node)                                    ▷ Revert visited
20:         return None                                  ▷ Indicate no cycle found
21:     end procedure
22:
23:     max_depth ← length of graph  ▷ Maximum depth for DFS (number of agents)
24:     for node in graph do
25:         visited ← empty set
26:         cycle ← DFS(node, node, visited, [], max_depth)
27:         if cycle then
28:             return cycle
29:         end if
30:     end for
31:     return None                                              ▷ No cycle found
32: end function
```

Algorithm 4 Improved-ECE

Initiate $L = N$ and $R = M$.
Initiate $A_i = \emptyset$ for all $i \in N$.
while $R \neq \emptyset$ **do**
    Compute a maximum weight matching M between L and R, where the weight of edge between $i \in L$ and $j \in R$ is given by $\upsilon_i(A_i, \cup\{j\}) - V_i(A_i)$. If all edges have weight 0, then we compute a maximum cardinality matching M instead.
    For every edge $(i, j) \in M$, allocate j to i: $A_i = A_i, \cup\{j\}$ and exclude j from R: R = R $\setminus\{$j$\}$.
    As long as there is an envy-cycle with respect to A = $(A_i)_{i \in N}$, invoke procedure P (Procedure P is similar to ECE by finding each cycle and resolving it by re-allocating goods as with ECE).
    Update A = $(A_i) i \in N$ to be the allocations after P.
    Update the set of agents not envied by any other agents: L= $\{i \in N : \forall j \in N, v_j(A_j) \geq V_j(A_i)\}$.
end while

Algorithm 5 MATCH&FREEZE$(N, M, (V_i)_{i \in N})$

Input: a 2-value instance using the values $a, b (a > b \geq 0)$
$L \leftarrow N$ ▷ set of active agents
$R \leftarrow M$ ▷ set of unallocated goods
$\ell = (1, 2, ..., n)$ ▷ ordered list of agents
while $R \neq \emptyset$ **do** ▷ every iteration is a round
    Construct the bipartite graph $G = (L \cup R, E)$.
    Compute a maximum matching on G.
    for `each matched pair` (i, g) **do**
        Allocate good g to agent i.
        Remove g from R.
    end for
    for `each unmatched active agent` i `w.r.t` ℓ **do**
        Allocate one arbitrary unallocated good g to i.
        Remove g from R.
    end for
    Construct the set F of agents that need to freeze.
    Remove agents of F from L for the next $\lfloor \alpha / b - 1 \rfloor$.
    Put agent of F to the end of ℓ.
end while
return allocation A.
procedure CONSTRUCT-FROZEN
    Construct $F := \{i \in L | \exists j \in L : V_j(g_i) = a, V_j(g_i) = b\}$. ▷ g_i denotes the good allocated to agent i in the current round.
    while there exists $i \in L \setminus F$ such that $\alpha \in [V_j(g_i) | j \in F]$ **do**
        Add i to F.
    end while
    return the set of agents F that will freeze.

Algorithm 6 GenerateSplits

```
1: function GENERATESPLITS(M, N, V)
2:     min_utility ← 0
3:     splits ← ∅
4:     current_split ← [∅ for N]
5:     procedure BACKTRACK(start, current_split)
6:         nonlocal min_utility
7:         if start > items then                ▷ base case of having considered all items
8:             if len(current_split) == N then      ▷ If current split contains all agents
9:                 splits.append([element[:] for element in current_split])  ▷ Append a copy of current_split to splits
10:            end if
11:            return
12:        end if
13:        for i ∈ N do    ▷ Assign current item to each agent and recursively explore
14:            current_split[i].append(start)
15:            if start < items then
16:                BACKTRACK(start + 1, current_split)
17:            end if
18:            current_split_utilities ← ∅
19:            for bucket in current_split do
20:                current_split_utilities.append(∑_{g∈bucket} V_i[g-1])      ▷ g-1 as goods here are 1-based while V is 0-based
21:            end for
22:            current_split_min_utility ← min(current_split_utilities)
23:            if min_utility ≤ current_split_min_utility then                ▷ Pruning
24:                min_utility ← current_split_min_utility
25:                BACKTRACK(start + 1, current_split)
26:                current_split[i].pop()
27:            else
28:                current_split[i].pop()
29:            end if
30:        end for
31:    end procedure
32:    BACKTRACK(1, current_split)
33:    return splits
34: end function=0
```

Algorithm 7 MNW Using Gurobi

```
def run(num_agents, num_goods, varrays):

    with gp.Env(empty=True) as env:
        env.setParam('OutputFlag', 0)
        env.start()
        with gp.Model(env=env) as m:
            m.setParam('TimeLimit', 30)
            m.setParam('MIPFocus', 3)
            m.setParam('MIPGap', 0)


            # Add binary decision variables indicating whether a good is allocated to an agent
            allocation = m.addVars(num_agents, num_goods, vtype=GRB.BINARY, name="x")

            # Add auxiliary variables for each agent's utility sum
            agent_utilities = m.addVars(num_agents, vtype=GRB.CONTINUOUS, name="agent_utilities")

            # Add constraints to compute each agent's utility sum
            for i in range(num_agents):
                m.addConstr(
                    agent_utilities[i] == gp.quicksum(allocation[i, j] * varrays[i][j] for j
                        in range(num_goods)),
                    f"agent_utility_{i}")

            # Add auxiliary variables for the logarithm of each agent's utility sum
            log_utilities = m.addVars(num_agents, vtype=GRB.CONTINUOUS, name="log_utilities")

            # Add constraints to compute the log of each agent's utility sum
            for i in range(num_agents):
                m.addGenConstrLog(agent_utilities[i], log_utilities[i], name=f"log_constraint_{i}")

            # Add Max Nash Welfare objective function (sum of log_utilities)
            m.setObjective(gp.quicksum(log_utilities[i] for i in range(num_agents)), GRB.MAXIMIZE)

            # ----- MNW Constraints -----

            # Add constraints to ensure each good is allocated to at most one agent
            for j in range(num_goods):
                m.addConstr(gp.quicksum(allocation[i, j] for i in range(num_agents)) <= 1,
                     f"good_{j}_allocation")

            # Optimize the model
            m.optimize()

            # ----- Output -----
            output_alloc = [[] for i in range(num_agents)]
            # Print the optimal solution
            if m.status == GRB.OPTIMAL:

                for i in range(num_agents):
                    for j in range(num_goods):
                        if allocation[i, j].x > 0.5:
                            output_alloc[i].append(j)
            elif m.status == GRB.TIME_LIMIT:
                return None
            return output_alloc
```

Algorithm 8 build_hierarchy

```
function BUILD_HIERARCHY_NEW(agents, goods, x, MBBi, i)
    graph ← build_augmented_graph_new(agents, goods, MBBi, x)
    Hi ← {}                                          ▷ Initialize the hierarchy dictionary
    agents_levels ← [None] * agents
    agents_paths ← [[] for _ in range(agents)]
    visited_agents ← {}                              ▷ Use a set to store visited agents
    queue ← deque([(i, [i])])                        ▷ Initialize the queue with the starting agent and its path
    visited_agents.add(i)
    agents_levels[i] ← 0
    agents_paths[i] ← [i]
    while queue is not empty do
        current_agent, current_path ← queue.pop_left()
        neighbouring_goods ← neighbors_by_edge_type(graph, current_agent, "mbb")
        for neighbour in neighbouring_goods do
            reachable_agents_from_goods ← neighbors_by_edge_type(graph, neighbour, "allocation")
            for reachable_agent in reachable_agents_from_goods do
                if reachable_agent not in visited_agents then
                    visited_agents.add(reachable_agent)
                    new_path ← current_path + [neighbour, reachable_agent]
                    queue.append((reachable_agent, new_path))
                    level ← len(new_path)//2
                    agents_levels[reachable_agent] ← level
                    agents_paths[reachable_agent] ← new_path
                end if
            end for
        end for
    end while
    for agent in range(agents) do
        level ← agents_levels[agent]
        if level is not None and level is not −1 then
            if level not in Hi then
                Hi[level] ← []
            end if
            Hi[level].append((agent, agents_paths[agent]))
        end if
    end for
    Hi[0] ← [(i, agents_paths[i])]                   ▷ Include the initial agent i at level 0
    return Hi
end function
```