Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Karoline Kanestrøm Sæbø

Exploring Z3 for the Fair Allocation Domain

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2023

NTNU
Norwegian University of
Science and Technology

Karoline Kanestrøm Sæbø

Exploring Z3 for the Fair Allocation Domain

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

This master thesis explores how the SMT-solver Z3 can be used, and how well it is suited, for the problem of allocating indivisible goods to agents under the fairness notions envy-freeness up to one good (EF1), envy-freeness up to any good (EFX), and maximin share fairness (MMS), when the valuations are additive. This was explored in the unconstrained case, as well as under conflict and connectivity constraints. Programs for finding allocations of different fairness notions were developed, and their runtimes were compared to existing mixed integer programs (MIPs) programmed to achieve the same task. The experiments showed that MIPs appear to be faster, and thus a better aid for finding allocations than programs made with Z3. Programs for exploring open problems in the field were also developed. The open problems explored were the characterization of conflict graphs that do not guarantee EF1 allocations, the existence of EF1 allocations when the conflict graph is a path, the existence of EF1 allocations under conflict constraints when the valuations are binary, and the existence of EFX allocations in the unconstrained case. The programs found that when there are conflict constraints and the number of items is less than or equal to seven, EF1 allocations always exist when the conflict graph forms a path, and they always exist when the valuations are binary. Moreover, it was found that in the unconstrained case, EFX allocations always exist when the number of items is six or less. Finally, it was demonstrated that programs made with Z3 could work as a tool for exploring ideas for and discovering patterns of problem instances that do not admit EF1 allocations under conflict constraints.

Sammendrag

Denne masteroppgaven utforsker hvordan Z3, en “satisfiability modulo theories solver” (SMT-løser), kan brukes innenfor feltet rettferdig fordeling. Z3 ble testet for tilfeller der varene som skal fordeles er udelelige, og der agentene som skal motta varene bruker additive verdivurderinger på varene. Fokuset var rettet mot rettferdighetskravene “envy-freeness up to one good” (EF1), “envy-freeness up to any good” (EFX), og “maximin share fairness” (MMS). Flere ulike tilfeller ble utforsket: tilfeller uten noen begrensninger, tilfeller der noen varer kan ha konflikter mellom seg og tilfeller der det er krevd at varene som deles ut må henge sammen. Det ble utviklet programmer for å finne fordelinger som oppfyller de ulike rettferdighetskravene, og kjøretidene ble sammenlignet med eksisterende blandet heltalls-programmer (MIP-er) laget for å løse de samme oppgavene. Sammenlikningen viste at MIP-er ser ut til å være raskere på disse problemene, og dermed et bedre hjelpemiddel for å finne rettferdige fordelinger enn programmer laget med Z3. Det ble også utviklet programmer for å utforske åpne problemer i feltet. De åpne problemene som ble utforsket var karakterisering av konflikt-grafer som ikke garanterer EF1-fordelinger, eksistensen av EF1-fordelinger når konflikt-grafen er en sti, eksistensen av EF1-fordelinger når varene har binære verdier og konflikter mellom seg, og eksistensen av EFX-fordelinger når det ikke er noen begrensinger på de mulige fordelingene. Programmene viste at når det er konflikter mellom varene og antall varer er mindre enn eller lik syv, eksisterer alltid EF1-fordelinger i tilfellene der konflikt-grafen danner en sti og i tilfellene der varene har binære verdier. Dessuten ble det oppdaget at EFX-fordelinger alltid eksisterer når det ikke er noen begrensninger på fordelingene og antallet varer er seks eller mindre. Det ble også demonstrert at programmer laget med Z3 kan fungere som verktøy for å utforske ideer for og oppdage mønstre i probleminstanser der EF1-fordelinger ikke er garantert når varene har konflikter mellom seg.

Acknowledgements

I would like to thank Magnus Lie Hetland and Halvard Hummel for their guidance throughout the work of this thesis and for sharing their ideas and knowledge in the fair allocation domain with me. I would also like to thank my family for their love and support.

Contents

Abstract i

Sammendrag ii

Acknowledgements iii

Contents iv

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Related Work . . . 2
- 1.2 Objectives . . . 3
- 1.3 Contributions . . . 4
- 1.4 Outline . . . 4

2 Background 5
- 2.1 Fair Allocation . . . 5
- 2.2 SMT Solvers and Z3 . . . 8
  - 2.2.1 Boolean Satisfiability Problem (SAT) . . . 8
  - 2.2.2 Satisfiablity Modulo Theories (SMT) . . . 9
  - 2.2.3 Z3 . . . 10

3 Implementation 11
- 3.1 Language . . . 11
- 3.2 Helper Functions . . . 12
  - 3.2.1 Find Least Valuable Item In Bundle . . . 12
  - 3.2.2 Find Most Valuable Item In Bundle . . . 13
  - 3.2.3 Ensure That Conflicts Are Respected . . . 13
  - 3.2.4 Ensure That Connectivity Is Respected . . . 13
  - 3.2.5 Ensure That Maximum Degree of the Graph is Lower than the Number of Agents . . . 14
  - 3.2.6 One Item to One Agent . . . 15

3.2.7 Ensure EF1 . . . 15
3.2.8 Ensure Envy-Freeness Up to One Outer Good . . . 16
3.2.9 Ensure EFX . . . 16
3.2.10 Calculate Maximin Share . . . 16
3.3 Programs for Finding Allocations . . . 17
3.3.1 Find an EF1 Allocation . . . 17
3.3.2 Find an EFX Allocation . . . 18
3.3.3 Find an MMS Allocation . . . 18
3.3.4 Conflicting Items: Find EF1, EFX and MMS Allocations 18
3.3.5 Connectivity Constraints Items: Find an EF1 Allocation . 18
3.4 Explore Problem Instances That Do Not Admit EF1 Allocations Under Conflict Constraints . . . 19
3.4.1 Find Valuation Functions . . . 19
3.4.2 Find Valuation Functions and Conflict Graph . . . 20
3.4.3 Find Valuation Functions, Conflict Graph and the Number of Agents . . . 21
3.5 Instances Under Conflict Constraints That Do Not Admit EF1 Allocations When the Conflict Graph's Components Are Paths . 23
3.6 Instances Under Conflict Constraints That Do Not Admit EF1 Allocations When the Valuation Functions Are Binary . . . 25
3.7 Instances With No Constraints That Do Not Admit EFX Allocations . . . 26

4 Experimental Setup **27**
4.1 Runtimes for Finding Unconstrained Allocations . . . 27
4.2 Runtimes for Finding Allocations Subject to Conflict Constraints 28
4.3 Runtimes for Finding Allocations Subject to Connectivity Constraints . . . 28
4.4 Exploring How Programs Can Look for Problem Instances With No EF1 Allocations Under Conflict Constraints . . . 29
4.5 Search for Problem Instances That Do Not Admit EF1 Allocations When the Conflict Graph Is a Path with Non-negative, Additive Valuations . . . 31
4.6 Search For Problem Instances With Binary Valuations and Arbitrary Conflict Graphs That Do Not Admit EF1 Allocations . . . 32
4.7 Search For Unconstrained Problem Instances That Do Not Admit EFX Allocations . . . 32

5 Results **33**
5.1 Programs for Finding Allocations . . . 33
5.1.1 Comparing Runtimes For Finding Allocations: Z3 vs. Gurobi . . . 33
5.1.2 Comparing Runtimes For Finding Allocations: Connectivity Constraints, Conflict Constraints and Unconstrained 33
5.2 Searching for Problem Instances With No EF1 Allocations Under Conflict Constraints: Performance and Findings . . . 34

5.2.1 Finding Valuation Functions That Do Not Admit EF1 Allocations . . . 34
5.2.2 Comparing Runtimes for Finding Parts of Problem Instances . . . 34
5.2.3 Graphs Discovered by Programs . . . 44
5.3 Existence of Problem Instances With No EF1 Allocations Under Conflict Constraints When the Conflict Graph's Components are Paths . . . 44
5.4 Existence of Problem Instances With No EF1 Allocations Under Conflict Constraints With Any Conflict Graph When the Valuations are Binary . . . 47
5.5 Existence of Unconstrained Problem Instances With No EFX Allocations . . . 47

6 Discussion **48**
6.1 Finding Allocations . . . 48
6.2 Z3 as an Aid to Try to Solve Four Open Problems . . . 49
6.2.1 Searching for Problem Instances Under Conflict Constraints That Do Not Admit EF1 Allocations . . . 49
6.2.2 Looking For Counterexamples Of EF1 Being Guaranteed When the Conflict Graph is a Path . . . 50
6.2.3 Looking For Counterexamples Of EF1 Being Guaranteed Under Conflict Constraints When the Valuations are Binary 51
6.2.4 Looking For Counterexamples Of EFX Being Guaranteed With Unrestricted Additive Valuations . . . 51
6.2.5 General Thoughts on Using Z3 to Prove Something . . . 52
6.3 Using Z3 for Fair Allocation in General . . . 52
6.4 Program is Slow for Medium-Size Inputs . . . 52

7 Conclusions **54**
7.1 Future work . . . 55

List of Figures

3.1 How the degrees of nodes are calculated. 15
3.2 Matrix used to keep track of which of the m items are allocated to which of the n agents. 17
3.3 Graphs used to test that the program is able to look for problem instances with no EF1 allocation. 21
3.4 Illustration to show the similarities between a path and an array or a row in a matrix. 24
3.5 Illustration showing how we can encode disjoint paths. 25

4.1 Graphs where problem instances with no EF1-allocation under conflict constraints can be constructed. 30

5.1 Runtimes for Z3 and MIP for finding unconstrained EF1 allocations. 35
5.2 Runtime for finding EFX allocations. 36
5.3 Runtime for finding MMS allocations. 37
5.4 Runtime for finding EF1 allocations subject to conflict constraints. 38
5.5 Runtime for finding EFX allocations subject to conflict constraints. 39
5.6 Runtime for finding MMS allocations subject to conflict constraints. 40
5.7 Runtime for finding EF1 allocations subject to connectivity constraints, compared to runtime for finding unconstrained EF1 allocations and EF1 allocations under conflict constraints. 41
5.8 Runtime for finding valuation functions such that the final problem instance does not admit EF1 allocations. 42
5.9 Runtime comparison of looking for different parts of problem instance in a way such that EF1 allocations do not exist. 44
5.10 Graphs with $n > \Delta(G)$ found by the programs within a time limit of two hours. 45

List of Tables

5.1 Discovered values by the program for the conflict graph $K_{3,n-1}$ such that the problem instance does not admit an EF1 allocation. 43
5.2 Discovered values by the program for the conflict graph $K_{n-1,n-1}$ such that the problem instance does not admit an EF1 allocation. 46
5.3 Runtime and result for trying to find problem instances that do not admit EF1 allocations when the conflict graph is a path. . . 46
5.4 Runtime and result for trying to find problem instances that do not admit EF1 allocations when the values of the items are binary. 47
5.5 Runtime and result for trying to find problem instances that do not admit EFX allocations when the number of agents is less than the number of items. 47

Chapter 1

Introduction

Fair allocation is a research field concerned with assigning items, or goods, to a set of agents so that each agent is happy with the outcome. Practical applications include dividing goods in an inheritance settlement, splitting rent for different apartment rooms and splitting tasks between workers.

While the problem of fair allocation has been extensively studied in the last decade, several theoretical questions in the field still remain unanswered. One of these open problems concerns the existence of envy-free allocations up to any good (EFX). For now, the existence has only been proven for special cases where the potential values for the items are restricted in some way [1, 2] or when there are few agents [1, 3]. Another open problem involves the characterization of instances where envy-free allocations up to one good (EF1) exist when the goods may have conflicts between them. It has been proven that EF1 is guaranteed when the graph describing the conflicts has certain properties and that it is not guaranteed when the graph has other properties [4]. However, a complete description of when EF1 is guaranteed is yet to be made.

In addition, there are practical situations where it is desired to calculate an allocation of a certain fairness measure in a reasonable amount of time. This is a challenging problem, given that for some fairness measures, like giving each agent their maximin share (MMS) or maximizing the Nash welfare (MNW), finding such allocations is known to be NP-hard.

A mixed integer linear program (MIP) has been used to calculate one such type of allocation, namely an MNW allocation, in a matter of seconds, for typical real-life problem instances [5]. MIPs have also been used to formulate other allocations, particularly MMS and EF1 allocations, to observe how common it is to achieve specific fairness measures [4]. Before these findings were published, work was also put into describing how various social welfare functions for maximizing agent satisfaction can be formulated as mixed integer linear programs [6].

While MIPs have been used to calculate such allocations, the calculation may take longer than desired, especially for MMS allocations. If one wishes to perform experiments on an extensive collection of problem instances, having a

faster way of performing these calculations would be advantageous.

In recent years, articles comparing the performance of MIP solvers to Satisfiability Modulo Theories (SMT) solvers have emerged. The findings concluded that the two types of solvers gave comparable results [7], and in some cases, the SMT solver outperformed the MIP solver [8].

Multiple efficient SMT solvers are now available, and it is held an annual competition where different solvers can showcase their strengths [9].[1] Z3 is one of these efficient SMT solvers. In one of the aforementioned competitions, SMT-COMP'07, a prototype of the Z3 solver won four first places and seven second places [10].

When a problem is given to an SMT solver, there are often multiple satisfying assignments to the variables, and the users may want the assignment that maximizes (or minimizes) some objective value. Some SMT solvers have been extended with optimization tools to meet this demand. Z3 is one of these solvers and has, on some benchmarks, proved superior to other solvers with optimization [8].

The aim of this thesis was to examine and provide information on how well-suited the SMT solver Z3 is in the fair allocation domain. Moreover, to see if some of the open problems in the field can be answered during this exploration of Z3 combined with fair allocation.

1.1 Related Work

SMT solvers, and the related Boolean satisfiability problem (SAT) solvers, have already been utilized in the field of fair allocation and closely related fields. For fair allocation of continuous items, known as cake-cutting, SMT solvers have been used to verify cake-cutting protocols automatically [11]. In social choice theory, a field closely related to fair allocation, SAT and SMT solvers have been used to help prove theorems and verify existing results. For example, Tang and Lin used a SAT solver to verify well-known impossibility theorems [12], Brandl et al. proved the incompatibility of efficiency and strategyproofness using an SMT solver (and verified the result with the interactive theorem prover Isabelle/HOL) [13], and Brandt et al. used a SAT solver when studying multi-agent voting [14].

Linear programs (LPs) and MIPs have also been used for fair allocation of indivisible items. Caragiannis et al. made a MIP that could find maximum Nash welfare (MNW) allocations, which scales well on real-world data and is used in the fair allocation app Spliddit [5]. Hummel and Hetland have used MIPs in their work on the fair allocation of conflicting items [4]. Their programs could find allocations of different fairness notions, like EF1, EFX and MMS, both with and without conflicts between the items. They used these programs to examine how fairness is affected by item conflicts. Feige et al. used a MIP to find a negative example of MMS allocations [15]. Xiao et al. used an LP to compute approximate MMS allocations and tight examples of the nonexistence

[1] Competition website: https://smt-comp.github.io/

of MMS allocations on negatively valued items, called chores, under connectivity constraints [16].

We see that problem solvers such as MIPs, SAT solvers and SMT solvers are not completely new in the field of fair allocation and related fields. However, while SMT solvers have been used in the fair allocation of continuous items, the use of SMT solvers for indivisible items still remains to be explored and compared to the use of MIPs.

1.2 Objectives

In order to fulfil the aim of examining and providing information on how well-suited Z3 is in the fair allocation domain, the work with this thesis has been organised with the following objectives:

1. Develop programs to find unconstrained EF1, EFX and MMS allocations using Z3.
2. Develop programs that integrate conflict and connectivity constraints with the ability to find allocations using Z3.
3. Assess how integrating constraints to allocation-finding programs made with Z3 influences runtime.
4. Compare runtimes of allocation-findings programs made with Z3 to existing, comparable MIPs.
5. Explore how Z3 can be used as an interactive tool to aid users in discovering new results by making programs that help look for conflict graphs that do not guarantee EF1 allocations.
6. Develop a program that can give new insight into the existence of EF1 allocations under conflict constraints when the conflict graph's components are paths.
7. Develop a program that can give new insight into the existence of EF1 allocations under conflict constraints when the values for the items are binary.
8. Develop a program that can give new insight into the existence of unconstrained EFX allocations.
9. Assess how useful Z3 is for answering open problems concerned with the existence of valid allocations.

1.3 Contributions

The main contributions of this thesis are:

- Prototypes of programs to find allocations and to aid the search for new results in fair allocation are developed.
- Z3 is shown to be able to find EF1, EFX and MMS allocations.
- Z3 is shown to be able to integrate conflict constraints and connectivity constraints when looking for allocations.
- Comparison of the prototype programs to comparable MIPs, shows that MIPs are faster at finding allocations.
- Experiments with the prototype programs show that using Z3 as a tool is best suited for small problem instances due to rapidly increasing runtimes.
- It is demonstrated that Z3 can be used to show the existence or find counterexamples of allocations with different fairness criteria when the number of items is limited.
- Existence results for fair allocation with additive valuations have been expanded by having Z3 establish that
  - EF1 allocations always exist under conflict constraints when the conflict graph's components are paths and the number of items is less than or equal to seven.
  - EF1 allocations always exist under conflict constraints and binary valuations when the number of items is less than or equal to seven.
  - EFX allocations always exist when the number of items is less than or equal to six.

1.4 Outline

The thesis is organized as follows. Chapter 2 presents the relevant background information in the fair allocation field,[2] as well as giving an introduction to SMT solvers and Z3, which is the SMT solver used in this thesis. Then Chapter 3 describes the programs implemented based on the objectives, and Chapter 4 describes the experimental setup used to test the programs. Chapter 5 presents the results, and the results are discussed in Chapter 6. Finally, conclusions and ideas for future work are presented in Chapter 7.

[2]The section about fair allocation is a modified and extended version of the equivalent section in the specialization project I did in the fall of 2022.

Chapter 2

Background

In this chapter, necessary background information is presented. First, relevant definitions and results in the field of fair allocation are introduced, followed by information on SMT solvers and Z3.

2.1 Fair Allocation

This thesis is centered around the fair allocation of indivisible items in unconstrained and constrained cases. There are several types of constraints in the literature; in this thesis, conflict and connectivity constraints will be the focus. Conflict constraints require that the items that are given to an agent do not have an edge between them in an underlying conflict graph. Connectivity constraints are somewhat opposite. They require that the items allocated to an agent are connected in an underlying item graph. Among multiple fairness measures possible to study, I have chosen EF1, EFX and MMS.

A problem instance for the fair allocation of items in the unconstrained case consists of three parts, $\langle N, M, V \rangle$. Here, N is a set of n agents and M is a set of m indivisible items. V is a family of n valuation functions, where a valuation function $v_i : 2^M \to \mathbb{R}_{\geq 0}$ assigns a value to each subset of items for agent i. To ease notation, $v_i(g)$ will denote $v_i(\{g\})$ in this thesis. A *bundle* denotes such a subset of items. An *allocation*, $A = (A_1, A_2, ..., A_n)$, is a partition of the items into n bundles, assigning bundle A_i to agent i. We say that an allocation is *complete* if the union of the bundles is equal to the set of all items. In this project, it is required that allocations are complete.[1] Moreover, the valuations are assumed to be additive. That means that an agent's value of any subset of items is equal to the sum of the values of individual items in the set. The following definition is based on the one by Bhaskar et al. [17].

[1] If they are not required to be complete, one can trivially find a fair allocation (in the envy-free meaning of the word, see Definition 2.2) by not allocating any items to anyone.

Definition 2.1 (Additive valuations)**.** For a problem instance $\langle N, M, V \rangle$ and agent $i \in N$, the valuation function is *additive* if $v_i(S) = \sum_{j \in S} v_i(j)$ for every set of items $S \subseteq M$, and $v_i(\emptyset) = 0$.

As previously noted, there are multiple ways to look at fairness. In this thesis, *envy-freeness*, or rather its relaxations, is the focus. Envy-freeness means that no agent prefers another agent's bundle over their own. The following definition is also based on Bhaskar et al. [17].

Definition 2.2 (EF)**.** A feasible allocation A is called *envy-free* (EF) if for every pair of agents $i, j \in N$ we have that $v_i(A_i) \geq v_i(A_j)$

Envy-free allocations do not always exist when the items are indivisible. Imagine allocating one item to two agents, where both agents value the item positively. A popular relaxation, introduced by Lipton et al. [18], and formally defined by Budish [19], is *envy-freeness up to one good.* Allocations that fulfil this fairness criterion are guaranteed via simple polynomial-time algorithms [20]. The following definition is based on the one used by Hummel and Hetland [4].

Definition 2.3 (EF1)**.** A feasible allocation A is called *envy-free up to one good* (EF1) if for every pair of agents $i, j \in N$ we have that $v_i(A_i) \geq v_i(A_j) - \max_{g \in A_j} v_i(g)$. That is, no agent prefers another agent's bundle if they are allowed to remove an item from the other agent's bundle.

Caragiannis et al. proposed another, stricter relaxation of envy-freeness, namely *envy-freeness up to any good* [5]. The following definition is based on the formulation by Amanatidis et al. [20].

Definition 2.4 (EFX)**.** A feasible allocation A is called *envy-free up to any good* (EFX) if for every pair of agents $i, j \in N$ we have that $v_i(A_i) \geq v_i(A_j \setminus \{g\})$ for any $g \in A_j$ such that $v_i(g) > 0$. That is, no agent prefers another agent's bundle if any item is removed from the other agent's bundle. Or equivalently, if the envious agent can remove their least favourable item from the other agent's bundle, they will no longer envy the other agent.

While EFX has been shown to always exist in some special cases, the general existence of EFX is an open problem.

In the literature, sometimes the requirement that the inequality the EFX definition must hold only for positively-valued goods is skipped, and we get an even stricter version of EFX. This stricter version is usually called EFX_0. In this thesis, I will use Definition 2.4 when calculating EFX, thus only requiring that the inequality holds for positively valued goods.

A third relaxation of envy-freeness is known as *maximin share fairness.* This fairness criterion was, like EF1, also introduced by Budish [19]. The goal here is that each agent should end up with a bundle worth at least their *maximin share.* The maximin share is the value the agent can guarantee for themselves if they were allowed to divide the set of items into n bundles but had to be the last to choose a bundle. To even just calculate maximin shares of agents is an NP-hard problem [21], and MMS allocations are not guaranteed always to exist [22]. The following definition of MMS is formulated by Amanatidis et al. [20].

Definition 2.5 (MMS)**.** Let $\mathcal{A}_n(M)$ be the collection of possible allocations of the goods in M to n agents. An allocation A is said to be *maximin share fair* (MMS) if for each agent $i \in N$, it holds that $v_i(A_i) \geq \mu_i^n(M) = \max_{\mathcal{B} \in \mathcal{A}_n(M)} \min_{S \in \mathcal{B}} v_i(S)$.

Under conflict and connectivity constraints, the problem instance has an extra part, G. Thus, the problem instance is $\langle N, M, G, V \rangle$ instead. G is an undirected graph in which the nodes represent items, while the meaning of the edges depends on what context the graph is used in. For conflict constraints, the edges represent the conflicts between the items, while for connectivity constraints they indicate that the items are connected in some way. For both constraints, EF1 as defined above, is no longer guaranteed [4, 23].

A graph can have different properties that are relevant in the fair allocation context. One of these is the graph's maximum degree, where a vertex's degree is the number of neighbouring vertices.

Definition 2.6 (Maximum degree)**.** The *maximum degree* of a graph G, denoted $\Delta(G)$, is the degree of the vertex with the largest number of edges incident to it.

Another important property of a graph G is the cardinality of the vertex set of its largest connected component, denoted $\mathcal{C}(G)$.

Hummel and Hetland have discovered several useful properties of problem instances with conflict graphs and additive valuations [4]. The relevant propositions for this project are presented in Propositions 2.7 to 2.10.

Proposition 2.7. For any graph $G = (M, E)$ with $\Delta(G) \geq n$, there is a problem instance $([n], M, V, G)$ that has no EF1 allocation.

Proposition 2.8. For any graph $G = (M, E)$ with $\mathcal{C}(G) \leq n$, all problem instances $([n], M, V, G)$ have EF1 allocations that can be found in polynomial time.

Proposition 2.9. For any $n \geq 4$, there exists a graph G with $n > \Delta(G)$ and a set of valuations V, so that the instance $\langle [n], M, V, G \rangle$ does not admit any EF1 allocations.

Proposition 2.10. If a problem instance $\langle N, M, V, G \rangle$, where $|N| > 2$, has valuation functions $v_i : M \to \{0, 1\}$, and the components of G are paths, then the instance has an EF1 allocation, which may be found in polynomial time.

Propositions 2.7 and 2.8 tell about the non-existence and existence of EF1 allocations, when $n \leq \Delta(G)$ and $n \geq \mathcal{C}(G)$, respecitvly. However, they leave the case $\Delta(G) < n < \mathcal{C}(G)$ open for both possibilities. Proposition 2.9 fills this open gap partly by telling us that when $n \geq 4$, there are some cases where an EF1 allocation does not exist in this interval. Hummel and Hetland remarked that six is the fewest number of items for which there is no EF1 allocation when $n > \Delta(G)$. However, they still leave a complete characterization of the instances where an EF1 allocation does not exist as an open problem.

Proposition 2.10 tells us that EF1 allocations always exist when the values of the items are binary and the components of the conflict graph are paths. Biswas et al. have generalized this result by showing that EF1 allocations always exist when the values of the items are binary, and the conflict graph is an interval graph [24]. It is unknown whether it is possible to generalize these results further by showing that EF1 allocations always exist when the values are binary for any conflict graph. It is also unknown whether Proposition 2.10 can be generalized the other way by showing that EF1 allocations always exist when the conflict graph is a path when the valuations are not restricted to be binary.

In the context of connected bundles, a stronger definition of EF1 than Definition 2.3 has been proposed. It is called *envy-freeness up to one outer good* and is more restrictive because it adds the requirement that after an item is hypothetically removed from an agent, the agent still has to have a connected bundle left. The following definition is based on Bilò et al. [25].

Definition 2.11 (Envy-freeness up to one outer good)**.** A feasible allocation A is called *envy-free up to one outer good* (EF1) if for every pair of agents, $i, j \in N$ we have that either $A_j = \emptyset$ or there is a good $g \in A_j$ such that $A_j \setminus \{g\}$ is connected and $v_i(A_i) \geq v_i(A_j \setminus \{g\})$.

That is, no agent prefers another agent's bundle if they are allowed to remove an item from the other agent's bundle, with the requirement that the other agent's bundle still remains connected.

In the rest of the thesis, *EF1* will refer to the "outer good"-version when the context is connectivity. In all other contexts, *EF1* will refer to the "normal version", that is, Definition 2.3.

2.2 SMT Solvers and Z3

Satisfiability modulo theories (SMT) is the problem of determining whether a logical first-order formula with respect to combinations of different background theories, is satisfiable. It is a generalization of the Boolean satisfiability (SAT) problem, a problem where one is concerned with finding a satisfying assignment for a formula that uses only plain Boolean logic. Several efficient SMT solvers have emerged in recent years, Z3 being one of them [26]. This section gives a brief introduction to the SAT problem and SAT solvers, followed by a description of SMT and SMT solvers. Finally, information on the efficient SMT solver Z3 is presented.

2.2.1 Boolean Satisfiability Problem (SAT)

The Boolean satisfiability problem (SAT) is concerned with whether a Boolean formula is satisfiable or not. In other words, given a formula with Boolean variables, we want to know whether it is possible to assign values to the variables in a way where the formula evaluates to $True$. For example, the formula

$$a \wedge \neg b$$

has the satisfying assignment $a = True$ and $b = False$ and is therefore satisfiable. However, the formula

$$a \wedge \neg a$$

has no possible assignment to make it evaluate to $True$. The formula is, therefore, unsatisfiable.

SAT was the first problem to be proven to be NP-complete with Cook's theorem [27] . SAT being NP-complete implies that all problems in NP are reducible to SAT in polynomial time, meaning that if we were able to solve the SAT problem efficiently, we would also be able to solve many other complex problems efficiently. While it is unknown whether a deterministic polynomial time algorithm exists to solve the problems in NP, the potentially long runtime applies to worst-case instances. Fortunately, with sophisticated techniques, many real-world SAT-instances, such as software verification, can be solved efficiently with SAT solvers [28][29] .

A SAT solver is a computer program made to solve the SAT problem. Most modern SAT solvers are based on the *Davis–Putnam–Logemann–Loveland* (DPLL) *algorithm* [30] [31] [32], a sound and complete algorithm that performs branching search with backtracking. The algorithm being sound means that if it returns an answer, this answer is guaranteed to be correct, while it being complete means that it terminates with a solution when one exists. There also exist SAT solvers that are not based on the DPLL algorithm. For example, GSAT [33] and WalkSAT [34] are solvers based on stochastic local search.

2.2.2 Satisfiablity Modulo Theories (SMT)

This section is largely based on information presented in the article *Lazy Satisfiability Modulo Theories* by Sebastiani [26].

Formulating a problem in pure Boolean logic is too restrictive in some applications. The formula

$$x < y \wedge y < 5$$

is a simple example. This formula is satisfiable, and one possible assignment is $x = 1$ and $y = 4$. However, this is formulated with more than just plain Boolean logic; integer arithmetic is also used.

In other applications, it is possible to formulate a problem in Boolean logic, but this formulation will make the abstraction level less efficient than desired. We see this in the verification of assembly-level code. It *is* possible to encode this in plain Boolean logic. However, there should be more efficient ways to encode a word than as a collection of unrelated Boolean variables. A way to formulate and solve the satisfiability with respect to some background theory is therefore needed.

The need for more expressiveness than Boolean logic has to offer was answered with theory-specific solvers (T-solvers). T-solvers are efficient procedures that can check the consistency of conjunctions of atomic expressions in decidable

first-order theories. Unfortunately, these T-solvers cannot handle the Boolean constraint component of reasoning; they can only deal with conjunctions of atomic expressions.

We see that there are tools to solve propositional satisfiability expressed with plain Boolean logic and tools to solve conjunctions of atomic expressions in first-order theories. SMT solvers are concerned with solving a combination and offer a way to express and combine theory-specific reasoning with Boolean reasoning. SMT applications range from resource planning to formal verification of compiler optimizations and real-time embedded systems. Possible background theories for SMT include integer or linear arithmetic, bit-vectors, arrays and lists.

When creating an SMT solver, one must combine SAT solvers with theory-specific procedures. There are two main approaches to doing this. The first is called eager, while the second is called lazy. The eager approach is based on encoding the SMT formula into an equivalently satisfiable Boolean formula and then giving this formula to a SAT solver. The advantage of this is that the SAT solver can be used "as is". The disadvantage is that the SAT solver may work harder than needed for "obvious" theory-specific facts. On the other hand, the lazy approach is based on separating the Boolean and theory-specific components of the problem and making a SAT solver and a T-solver communicate to find the correct solution. Today, the most efficient SMT tools use the lazy approach.

In recent years, multiple papers suggesting new and efficient techniques for SMT have been published, and several powerful SMT tools are now available. These tools include but are not limited to CVCLite/CVC3 [35], Zapatop [36] and Z3 [10].

2.2.3 Z3

Z3 is an SMT solver from Microsoft Research, with support for various theories, free to use for everyone [10]. It is originally targeted at solving problems in software verification and analysis. As mentioned in the introduction, Z3 has demonstrated its power by winning four first places and seven second places in the SMT competition SMT-COMP'07 [10].

Z3 is written in C++, and is composed of multiple solvers, each with their designated task. It uses a DPLL-based SAT solver for boolean reasoning, a core theory solver for equalities and uninterpreted functions and a satellite solver for arithmetic and arrays. In addition, it uses an E-matching abstract machine for quantifiers. The various parts will not be explained in detail here; the interested reader is referred to [10].

While the source code of Z3 is written in C++, the solvers have bindings to various other languages, including Python, Julia and Java.[2]

[2]See https://github.com/Z3Prover/z3 for a complete list of bindings.

Chapter 3

Implementation

This chapter describes the implementation of programs made to fulfil the objectives presented in Section 1.2. The source code can be found on GitHub.[1] The programs have been implemented in the Python programming language, version 3.7.16. The iGraph package, version 0.9.10, was used to handle graphs.

The choice of programming language will first be explained. Then important helper functions will be described, followed by descriptions of how these helper functions are combined to make the intended programs.

3.1 Language

As mentioned in Section 2.2.3, Z3 offers bindings to multiple programming languages. For this project, the Python API has been used for all new code. The decision to use the Python API is based on Python being a user-friendly language, making the code for the project accessible to people with various types of programming experience. In addition, the Python API is well documented and has a relatively large community on forums like Stack Overflow.

Some parts of the code were initially planned to be written in Julia instead of Python. These parts are the ones where the program's performance is measured against the MIPs made for the *Fair Allocation of Conflicting Items* article [4], as the package with these MIPs is written in Julia. If a difference in performance were found, it would be good to know that this difference did not come because Julia is a faster language than Python.[2]

However, as of the spring of 2023, the Julia API is only a Julia wrapping of the C++ API and has significantly worse documentation than the Python API. [3] In addition, likely because of CxxWrap.jl, which is used to do the wrap-

[1] The source code is available at https://github.com/karoliks/master-testing.

[2] See performance of Julia compared to Python and other programming languages at https://julialang.org/benchmarks/

[3] See https://github.com/ahumenberger/Z3.jl for the Z3 Julia package.

ping, there are reportings of segmentation faults when using the API. [4] I also experienced segmentation faults when attempting to use Z3 with Julia. I never experienced this with the Python API. For a better developer experience and to be able to use more time to produce findings and not debug, all parts of the code were written in Python instead.

3.2 Helper Functions

While the programs made have different objectives, they still have several things in common. For example, the fairness measure EFX is used in several programs, but the requirements for an allocation being EFX stay constant. Consequently, there is a collection of helper methods, which are put together in different combinations to explore different use cases of Z3. These basic helper functions are presented in this section. How they are combined is explained in Sections 3.3 to 3.7.

3.2.1 Find Least Valuable Item In Bundle

From Definition 2.4 we know that to have an EFX allocation, the envy from one agent to another must go away if any item is removed from the better off agent. This translates to checking whether the envy disappears if the lowest valued item is removed, as this implies that the envy would also be removed if any more valuable item was removed.

This helper function's purpose is to create a formula that calculates the lowest-valued item. The function is provided with a list of valuations for one agent a_1 and a list of Boolean variables indicating what items are allocated to another agent a_2.

A variable to store the value of the lowest-valued item is made. This variable is first given the value -10.[5] Then, the contents of the two lists are inspected sequentially for each item to determine the actual lowest-valued item. For each item, the list indicating what items are allocated to a_2 is checked to see if this item is allocated to a_2. If this is the case, and the value for this item is lower than the current value while still being above zero, or the current lowest value is negative, the current lowest value is updated to the current item's value. The reason for checking whether the current lowest value is negative is to be able to move away from the default value we defined in the beginning. If the requirements are not fulfilled, the current lowest value from earlier is kept. The reason for checking whether the value is above zero, is that we use the EFX definition in Definition 2.4, not the stricter EFX_0 definition.

But what if no items are allocated to a_2? Then the procedure described above would return -10 as the lowest valued item. If we then would check if the formula for EFX was fulfilled by calculating the bundle value of a_1, and check

[4] Reports of segmentation faults: https://github.com/ahumenberger/Z3.jl/issues/12 and https://forem.julialang.org/zxzkja/z3jl-optimization-example-5db3

[5] -10 is used in the implementation, but any negative value could be chosen.

if this was more or equal to the bundle value of a_2 minus the value of the least valued item, the bundle of a_1 would have to be worth at least $0 - (-10) = 10$, even when a_1 is not allocated anything! To circumvent this problem, another formula is added: If the answer of OR-ing the variables in the list that indicate what is allocated to a_1 is $False$, meaning that a_1 is not allocated anything, the least valued item is set to 0.

3.2.2 Find Most Valuable Item In Bundle

From Definition 2.3 we know that we need the highest valued item of one agent's (a_2) bundle from another agent's (a_1) perspective when calculating EF1.

This helper function creates a formula to find the highest-valued item in a similar fashion to how the helper function above finds the least-valued item. It is supplied with a list of values for one agent a_1 and a list indicating what items are allocated to another agent a_2. The difference is that we now do not need extra logic to take care of the case where a_2 is not allocated anything.

The maximum value is first set to 0. Then, each item is inspected, and if a_2 is allocated an item, and a_1's value for this item is more than the current maximum value, the current maximum value is updated to be the value of this item. After all the items have been inspected, the formula specifying the maximum value is returned.

3.2.3 Ensure That Conflicts Are Respected

This function is made for cases where the items can have conflicts between them. The formula returned by the function makes sure that no two items allocated to the same agent can have an edge between them in the underlying conflict graph.

Multiple versions of this function have been implemented as the final programs require different ways of specifying graphs. These include getting a graph created by the Python library iGraph, getting a graph given by a known adjacency matrix and a graph given by an unknown adjacency matrix (a matrix filled with boolean Z3 variables). Common for all these versions is the formula for avoiding conflict. For all pairs of items g_j and g_k connected by an edge, we have that

$$\bigwedge_{i=1}^{n} \neg(A_{i,g_j} \wedge A_{i,g_k}). \tag{3.1}$$

More informally, none of the n agents can be allocated both items if a conflict edge connects the items.

3.2.4 Ensure That Connectivity Is Respected

When looking at connectivity constraints, we need to ensure that each agent receives a connected bundle. Therefore, we need a way to check and ensure a

bundle is connected.

How can we do this? One way is to find the transitive closure of the graph, as taking the transitive closure of an undirected graph $G = (V, E)$ results in a new graph $G^* = (V, E^*)$ with edges between each pair of vertices that were connected in G. We can then iterate through each pair of vertices in a bundle and require that they have an edge between them in the transitive closure of the graph.

So how do we do this? Luckily, Z3 has built-in support for transitive closures. To use this, the graph, originally made with iGraph, is modelled as a binary relation with Z3. A transitive closer of this relation is then created, using Z3's `TransitiveClosure()`, and finally, we can do as we planned: Each pair of items for the bundle in question is required to be connected in the transitive closure.

3.2.5 Ensure That Maximum Degree of the Graph is Lower than the Number of Agents

This helper function was needed when looking for conflict graphs where an EF1 allocation is not guaranteed. From Proposition 2.7 we already know that if the maximum degree of the conflict graph is greater than or equal to the number of agents, there is a problem instance with this graph that have no EF1 allocation. It is therefore not interesting to look for graphs with this property, as we already know that we are guaranteed to find "good" results. The programs are made as an aid to discover new graph classes where we are not guaranteed EF1. Therefore we need to find the maximum degree of the graph, such that we can require that the programs only look for "interesting" cases, where the maximum degree is less than the number of agents.

This function is only relevant for the cases where one wants to discover a conflict graph, and thus it is made to receive a graph represented by an adjacency matrix with unknown boolean Z3 variables, not a graph made with iGraph. Other than being supplied with an adjacency matrix, the function is also given the number of agents and the number of items.

The function does not actually calculate the maximum degree of the graph and then check that this is less than the number of agents; it rather calculates the degree of each node and makes sure that all the nodes have a degree less than the number of agents. There is no native support in Z3 to find a maximum value of variables, so rather than making a custom function to calculate the maximum, this seemed like a good approach.

But how does the function find the degree of each node? Because the conflict graphs are undirected, the adjacency matrix will be symmetric. To limit the number of computations, Z3 is instructed to only look at one half of the matrix when looking for graphs. Therefore, This function can only use information from one side of the diagonal and will be summing entries in the matrix horizontally until the diagonal is hit and then vertically to the bottom of the matrix.

Let us take a look at an example to illustrate how it works. Take the adjacency matrix for the complete graph with four nodes, as shown in Figure 3.1. As the graphs we are looking at are undirected, if we look at the whole adjacency

(a) *Using the whole adjacency matrix.* **(b)** *Using only one side of the diagonal.*

Figure 3.1: *How the degrees of nodes are calculated. The lines illustrate what entries are added together for each node in the calculation of the degree. In this example, the addition will show that each node have a degree of three.*

matrix, we can calculate each node's degree by summing each row's entries, illustrated with lines in Figure 3.1a.

But, in order to save Z3 from a few variables, the program is instructed to only look at the lower half. Therefore, the degree of the graph is rather summed up horizontally until the diagonal and then vertically. This is shown in Figure 3.1b.[6]

3.2.6 One Item to One Agent

When finding valid allocations, two things are important. First, a good should not be allocated to multiple agents simultaneously. Second, all the items should be allocated. The problem becomes trivial if all items do not have to be allocated, as no one will envy anyone if no one gets anything.

With these things in mind, a rule that forces each item to be allocated to one, and only one agent, has been created. The function is given an $n \times m$ allocation matrix A as an argument, where n is the number of agents and m is the number of items. For each column, it is required that only one of the boolean variables can be set to $True$. This has been implemented with the Pseudo-Boolean equality constraint that Z3 has: `PbEq()`. Mathematically, for each item g in the allocation matrix, treating boolean variables as zeroes and ones, we get

$$\sum_{i=1}^{n} A_i, g = 1. \tag{3.2}$$

3.2.7 Ensure EF1

Ensuring EF1 is done by going through all pairs of agents and checking that each agent thinks that the value of their own bundle is worth at least as much as the other agent's bundle after subtracting the value of the most valuable item from the other agent's bundle, as described in Definition 2.3. The helper function described in Section 3.2.2 is used to calculate the most valuable item.

[6]In the illustrations, the upper right half of the matrix still has values. This is just for illustration purposes, and the redundant values are not implemented in the Z3 program.

3.2.8 Ensure Envy-Freeness Up to One Outer Good

To ensure envy-freeness up to one outer good (Definition 2.11), we need a way to check whether an item is an outer good. This is done by first removing the potential item from the bundle, and then sending the remaining bundle to the helper function that checks if a bundle is connected, described in Section 3.2.4. If the remaining bundle was connected, we know that we removed an outer good. If the remaining bundle was not connected, it was not an outer good.

Now that we know how to check whether an item is an outer good, we can find a formula that ensures that envy-freeness up to one outer good is fulfilled. To do this, we loop through every pair of agents a_i and a_j, and for each pair of agents, we make an OR-expression that we require to be $True$. This OR-expression is OR-ing together one formula for each item g_k. This formula checks whether the bundle of the first agent, a_i, is worth at least as much as the other agent's, a_j's, bundle (from the first agent's perspective) when the value of g_k is subtracted. However, the value of g_k is only subtracted if g_k is an outer item. The final OR-expression is shown in Equation (3.3). If any of the formulas in the OR-expression evaluates to $True$, there exists an outer good that can be removed such that the first agent is not envious of the other, and the whole OR-expression will evaluate to $True$.

$$\bigvee_{k=1}^{m} v_i(A_i) \geq v_i(A_j) - If(is_outer_item, v_i(g_k), 0) \tag{3.3}$$

3.2.9 Ensure EFX

The construction of the formula that ensures EFX is similar to the construction of the formula that ensures "normal" EF1. The difference lies in what item is hypothetically removed from a bundle. We loop through all pairs of agents, and for each pair, we make sure that each agent thinks that the value of their own bundle is worth at least as much as the other agent's bundle after subtracting the value of the least valuable item from the value of the other agent. The helper function to find the least valuable item is described in Section 3.2.1.

3.2.10 Calculate Maximin Share

When calculating an MMS allocation, the goal is to give each agent a bundle of value at least equal to that agent's maximin share. This is the maximum value that the agent can guarantee for themselves if they could partition the items in any way possible but had to take the worst bundle.

Therefore, to be able to compute an MMS allocation, we need to calculate each agent's maximin share. That is what this helping function does.

This has been implemented by looking at each of the agents one by one. When calculating the maximin share for an agent, a boolean matrix is made to keep track of possible allocations that the agent we look at may choose to maximize the value of the worst bundle. The allocation matrix is set to use the

$$A_{n \times m} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{bmatrix}$$

Figure 3.2: *Matrix used to keep track of which of the m items are allocated to which of the n agents.*

earlier mentioned helper function that ensures that each item is allocated and that an item is not given to more than one person as well as the helper function that the solution adheres to the potential conflicts between the items. The value of each bundle in the proposed partition is calculated with the valuation function of the agent we are looking at. A Z3 variable is made for the maximin share, and this variable is instructed to be less than or equal to the value of each bundle.

Then, the optimization solver of Z3 is instructed to maximize the variable holding the maximin share. When the optimization is complete, the value of the maximin share is returned.

3.3 Programs for Finding Allocations

This section will describe how the programs are set up to find allocations for the fairness measures EF1, EFX and MMS, both with and without conflict constraints, as well as EF1 allocations with connectivity constraints.

Common for all the programs that implement this functionality is that they receive the parameters n (the number of agents), m (the number of items), and V, an $n \times m$ matrix with the values each agent assigns to each item. In the cases of conflict and connectivity constraints, a graph is also provided as an argument. The graph is made with the iGraph library for Python.

3.3.1 Find an EF1 Allocation

To find a valid EF1 allocation for a given problem instance, an $n \times m$ matrix A is set up with Boolean Z3 variables, see Figure 3.2. The fact that the matrix is filled with Z3 boolean variables means that Z3 can reason on its own about what values should be $True$ and what should be $False$ in the matrix, and the values are not known until the program has run.

To ensure that the assignments to the allocation matrix give a valid allocation, we require that formulas from some of the earlier described helper functions evaluate to $True$. The helper function described in Section 3.2.7 is used to require that the final allocation must satisfy the requirements of an EF1 allocation, and the helper function described in Section 3.2.6 is used to require

that each item is given to one, and only one, agent. These helper functions pose restrictions on matrix A.

The function ends with a call to Z3, instructing it to find valid entries in A, or come up with a proof of why it is impossible.

3.3.2 Find an EFX Allocation

The structure used for finding EFX allocations is very similar to the one used for finding EF1 allocations, described above. The only difference is that instead of posing restrictions on the allocation by using the helper function that ensures an EF1 allocation, the helper function that ensures an EFX allocation is used. This helper function is described in Section 3.2.9.

Other than requiring an EFX allocation, the use of an $n \times m$ allocation matrix filled with boolean Z3 variables is the same as for EF1, and the use of the helper function that requires that each item is given to exactly one agent, is the same.

3.3.3 Find an MMS Allocation

The structure of the code that finds MMS allocations is somewhat different from the code that finds EF1 and EFX allocations.

When finding MMS, an array with calculated MMS values for all the agents is first filled by calling the helper function for calculating maximin shares for each agent. The helper function is described in Section 3.2.10.

Then, the program is instructed to find an allocation that ensures that each agent gets a bundle that they consider is worth at least their maximin share. This allocation also has to adhere to the requirement that each item is given to one, and only one, agent.

3.3.4 Conflicting Items: Find EF1, EFX and MMS Allocations

These programs are written exactly like the programs where the unconstrained allocations are found, except here there are posed extra restrictions on the possible allocations. These are posed by using the helper function for conflicting items, described in Section 3.2.3. For the MMS allocations, these extra restrictions are posed both on the allocations when finding individual maximin shares and when finding the final allocation.

3.3.5 Connectivity Constraints Items: Find an EF1 Allocation

As for conflict constraints, the setup for finding EF1 allocations under connectivity constraints was similar to just finding normal EF1 allocations. The only differences were that the helper function for ensuring envy-freeness up to one

outer good was used instead of the helper function for normal EF1 and that the helper function that ensures that each bundle is connected was used.

EF1 was the only fairness notion tested with connectivity constraints. This is because the main goal of making a program to find connectivity constraints was to see if it was possible to do so with Z3. In contrast to the case of conflict constraints, there do not seem to be any existing programs to compare the runtime for finding allocations under connectivity constraints. It was, therefore, not as interesting to test the constraint for more than one fairness measure. Moreover, from the setup described for programs with conflict constraints, changing the fairness measure when one has a formula that describes the constraint is relatively straightforward. Consequently, the focus was on formulating connectivity constraints, not testing connectivity constraints for multiple fairness notions.

3.4 Explore Problem Instances That Do Not Admit EF1 Allocations Under Conflict Constraints

This section will describe programs to explore conflict graphs without EF1 allocations. This is to help to give insight to the open problem of getting a more detailed characterization with problem instances of conflicting items where EF1 does not exist, presented in [4].

3.4.1 Find Valuation Functions

The function that is going to calculate the valuation function where there exists no EF1 allocation is given n (the number of agents), m (the number of items), and G (the conflict graph), but naturally, no valuation functions. The valuation functions are now represented by an $n \times m$ matrix filled with integer Z3 variables. The program, therefore, has to find out more than in the previously described functions when it only had to find a valid allocation (or report that no such allocation existed).

Here, like earlier, we require that no agent can have the same item as any other agent and that all items have to be allocated to someone. We require this by using the helper function that ensures that each item is allocated to exactly one agent, described in Section 3.2.6. Then, like for the programs where conflict constraints were used, we require that no two items with a conflict between them can be given to the same agent. This is done with the helper function for conflicting items described in Section 3.2.3. Like earlier, the graph is made with the iGraph Python package

Compared to the other programs, what is new is that instead of looking for an allocation, we want to find a valuation function for an incomplete problem instance, such that the final problem instance has no possible EF1 allocation. We, therefore, end up with the following formula that we want Z3 to solve, giving us V:

$$\forall A((I(A) \wedge C(A,G)) \rightarrow \neg EF1(A,V)), \tag{3.4}$$

where A is a boolean allocation matrix, as shown in Figure 3.2, $I(x)$ is the predicate "each item in allocation x is given to one and only one agent", $C(x,y)$ is the predicate "allocation x respects the conflicts in the conflict graph y", and $EF1(x,z)$ is the predicate "the allocation x fulfilles the requirments for EF1 given valuation functions z". In less formal terms, we ask Z3 to give us the set of valuation functions V such that when the requirements of allocating all items correctly and respecting conflicts are fulfilled, there will be no possible EF1 allocation.

The reason for using implication in Equation (3.4), and not just AND-ing the three predicates, is that when *all* allocations are considered, there will be some allocations where $I(x)$ or $C(x)$ evaluates to $False$. For example, when all possible allocations are considered, there will be allocations where an agent will get two items that are in conflict. Because this would mean that the formula with the predicates is not true for all allocations, Z3 would say that the problem we are trying to get it to solve is unsatisfiable. However, when using implication, an expression always evaluates to $True$ when the antecedent (the first part in the implication) is $False$. That gives us the desired behaviour, as we want a valuation function that makes sure that there is no possible EF1 allocation when conflicts are respected and when each item are given to one and only one agent.

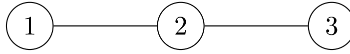
To test the program, examples derived from Hummel and Hetland's article [4] have been used to verify that it works as it should. The chosen examples are based on the results about there always being a problem instance with no EF1 allocation when the degree of the graph is greater or equal to the number of items, see Proposition 2.7, and the result that when $n \geq 4$ there exists a graph with $n > \Delta(G)$ such that the problem instance does not admit any EF1 allocation, see Proposition 2.9, and where this graph can be the complete bipartite graph $K_{n-1,n-1}$. Illustrations of the graphs used for testing is shown in Figures 3.3a and 3.3b, respectivly.

3.4.2 Find Valuation Functions and Conflict Graph

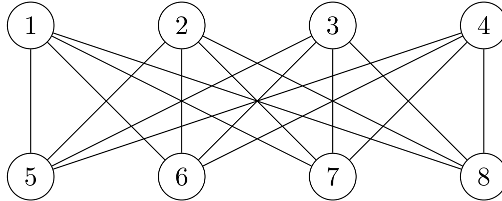
In this case, we want to give the program a number of agents and a number of items, and we want the program to figure out the rest of the problem instances. That is, what graph and valuation functions are needed so that no EF1 allocation exist.

Finding the valuation functions is done in the same way as in Section 3.4.1. In addition, the program is set to look for a graph and the agents' values for the items, such that Equation (3.4) evaluates to $True$. As it would be hard for Z3 to create its own graphs with the Python iGraph library, the graphs will now be represented by adjacency matrices filled with Boolean Z3 variables.

When setting up formulas for the conflicts based on the adjacency matrix, the fact that conflict graphs are undirected was taken advantage of. In an undirected



(a) *When the number of agents is two, there exists a valuation function that makes it impossible to find an EF1 allocation for three items with this conflict graph.*



(b) *For five agents, it has been shown that a valuation function can be found for the items in this conflict graph such that the problem instance has no EF1 allocation.*

Figure 3.3: *Graphs used to test that the program is able to look for problem instances with no EF1 allocation.*

graph, the adjacency matrix will be symmetric. Therefore, the program only has to calculate the values on one side of the diagonal.

In addition, the goal of making this program is to gain more knowledge of when an EF1 allocation is guaranteed and not in the context of conflicting items. Therefore, it would be preferable if the program avoided giving us answers we already knew. Therefore, when the program is looking for a graph, the graph will be restricted to have a maximum degree less than the number of agents. This is done by using the helper function described in Section 3.2.5.

3.4.3 Find Valuation Functions, Conflict Graph and the Number of Agents

When the program is able to find both the conflict graph and valuation function of the problem instance, as described in Section 3.4.2, the user of the program still has to provide the number of agents and the number of items. Thus, the user will have to guess what ratio between the two numbers is likely to produce an instance that does not admit any EF1 allocation. With the version of the program described in this subsection, the user will no longer have to guess this ratio; they will only have to give a guess on how many items should be checked.

The implementation of this version builds on the implementation of the version where both the number of agents and the number of items had to be provided. The biggest difference is that the number of agents is a Z3 integer variable that Z3 has to find a value for, not an argument the user provides.

However, the fact that the number of agents n is a Z3 variable makes it hard to make $n \times m$ matrices to fill with boolean Z3 variables. Matrices like this were

used earlier for the allocation matrix A and the valuation matrix V. Space for these matrices needs to be allocated at the start of the program, but because n is a Z3 variable we do not know the value of n until the program has run.

Fortunately, the number of agents can be bounded by the number of items, based on Proposition 2.8. The proposition tells us that when the number of agents is equal to or greater than the cardinality of the largest component of the conflict graph, all problem instances have EF1 allocations. Therefore, when trying to look for problem instances that do not admit any EF1 allocations, we only have to look for instances where the number of items is larger than the number of agents, as the cardinality of the largest connected component cannot be more than the number of nodes in the whole graph (which in our case is the number of items.) If the number of agents was larger than the number of items, we would be guaranteed that the cardinality of the largest component was smaller than the number of agents, thus guaranteeing that all possible found problem instances have EF1 allocations.

The problem of not knowing how many variables to create for the valuation functions and the allocations can therefore be solved by using matrices of size $m \times m$ instead of $n \times m$ and instructing the program that $n < m$.

However, several of the helper functions described earlier expect an $n \times m$ matrix. We, therefore, have to modify the helper functions to handle a $m \times m$ matrix. This is relevant for the helper functions for getting one item to one and only one agent, respecting the conflicts of the conflict graph, and checking whether the allocation is an EF1 allocation.

One Item to One Agent: Modification After the modification of the helper function that ensures that each item is allocated to one and only one agent, we still count the number of agents an item is allocated to, and we require the sum to be zero. However, we now only consider an entry in the $m \times m$ allocation matrix A to be valid if the agent it represents is one of the n chosen agents. That is, for each column in the matrix, indicating where a good g is allocated, we require

$$\sum_{i=1}^{m}(i \leq n \wedge A_i, g) = 1. \tag{3.5}$$

One may wonder why the upper bound of the summation is not just kept to be n, thus removing the need for the inequality in the formula. The reason is that n is only a symbolic variable now, and its value can be anything within our constraints. The implementation of the summation uses a for loop, and a symbolic variable cannot give the number of loops.

Ensure That Conflicts are Respected: Modification For the conflicts, the modification is simpler. Here, earlier, for all pairs of items g_j and g_k that were in conflict, all the agents would get the constraint that they could not get both items. Instead of going through all agents now, however, we just pose this constraint for all variables in the allocation matrix A, independent of knowing

whether the agent we are posing the constraint for is a valid agent or not, as these possible surplus constraints do not make the rest of the calculations wrong. For all pairs of items, g_j and g_k, that are connected in the conflict graph, we require that

$$\bigwedge_{i=1}^{m} \neg(A_{i,g_j} \wedge A_{i,g_k}). \tag{3.6}$$

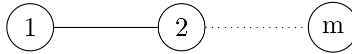
Ensure EF1: Modification The modification to this helper function is also simple. Here the `If()` function that Z3 provides is used to check if we are looking at envy between to valid agents (their index is less than the symbolic variable n). The `If()` function takes three arguments: an expression to be evaluated, an expression that has to be $True$ if the first expression evaluates to $True$ and a third expression that has to be True if the conditional expression evaluates to $False$. If both are valid agents, we require that EF1 is fulfilled. If they are not both valid agents, we just say that the expression should always evaluate to $True$.

3.5 Instances Under Conflict Constraints That Do Not Admit EF1 Allocations When the Conflict Graph's Components Are Paths

As the open problem to be explored is "Is EF1 guaranteed when the conflict graph's components are paths?", the program will be instructed to try and find an example where the conflict graphs components are paths but where there does not exist an EF1 allocation.

The ideal way to do this would be to make Z3 look at all possible problem instances with a conflict graph having path components (and the number of agents is more than two) to try to find an example where there is not an EF1 allocation. However, this is a challenging task. Therefore, this program will search for counterexamples for a specific number of items. While being dependent on a number of items is not ideal, it does still give some opportunity to look for counterexamples. Furthermore, if no counterexamples are found for some specific number of items, we know that EF1 is guaranteed in these cases.

To achieve this, we need a way to model the conflict graph and ensure its components are paths. To encode this, we could instruct the program to use adjacency matrices like earlier and pose restrictions on what kind of graphs it was allowed to find in a way that guaranteed that it would only end up with components that are paths. A naive way to do this would be to require that each node had a degree of either zero, one or two, and that there are an even number of vertices with degree one (terminal vertices). However, this would allow cycles to appear, which is not desired. Now, there are ways to detect



(a) *Path graph for m items.*

$$\begin{bmatrix} a_{b1} & a_{b2} & \cdots & a_{bm} \end{bmatrix}.$$

(b) *Row in allocation matrix A for agent b.*

Figure 3.4: *Illustration to show the similarities between a path and an array or a row in a matrix.*

cycles in adjacency matrices, for example, with depth-first search. However, implementing such an algorithm in Z3 would add unnecessary complexity.

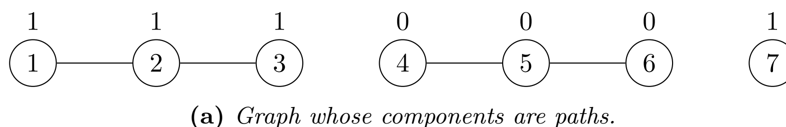
Instead, we can encode the path restrictions directly into the allocation matrix. To explain, let us first forget about how to encode a collection of paths and look at how a single path can be encoded instead.

Recall how our allocation matrix looks in Figure 3.2.[7] Each of the n rows shows which of the m items that are allocated to each of the n agents. These rows bear a striking similarity to paths, as we see in figure Figure 3.4. The path restrictions are implemented by adding the constraint that an agent cannot be given two items with consecutive indices in the row. Thus, for each row, indicating what is allocated to an agent b, we pose the restriction

$$\bigwedge_{i=1}^{m-1} \neg(A_{b,g_i} \wedge A_{b,g_{i+1}}). \tag{3.7}$$

Now that we know how to encode one path, let us look at encoding multiple paths of arbitrary length. In this case, we can imagine all the paths being placed in a line, one after the other, as illustrated in Figure 3.5a. We can give each component a name, and say that the restriction in Equation (3.7) only counts if the two goods has the same component name. If two goods are next to each other, but from different components, no restrictions are applied. As we only pose restrictions for items next to each other, we only need two unique names for the different components. We can therefore model this with Boolean variables; see the numbers above each node in Figure 3.5a. We can add an extra boolean array of length m, where each index in the array corresponds to the item with that number, see Figure 3.5b. If two consecutive items are part of the same graph component, they are given the same Boolean value; if they are part of different components, they are given different values. Whether Equation (3.7) should be enforced for these consecutive items can then be checked with the negation of XOR-ing the component names of the two items, as XOR will evaluate to $True$ when two variables are different and $False$ when they are

[7]Here, the allocation matrix is an $n \times m$ matrix to ease the explanation. In the actual implementation, it will be a $m \times m$ because the number of agents is unknown, as explained earlier in Section 3.4.3



(a) *Graph whose components are paths.*

$$\left[\begin{array}{ccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right].$$

(b) *Array encoding the components in a graph whose components are paths.*

Figure 3.5: *Illustration showing how we can encode disjoint paths.*

equal. Thus the negation will evaluate to $True$ if the items are form the same component, and $False$ if they are not/

However, while this should work, it may overengineer the problem for our purpose. Remember that we want to determine whether EF1 is guaranteed under conflict constraints when the conflict graph's components are paths. A single path of all the items gives a more restricted problem instance than disjoint paths. If we can prove that EF1 always is possible for conflict paths, we will, therefore, automatically prove that EF1 is possible for conflict graphs that consist of disjoint paths. Consequently, we only have to model paths when looking for counterexamples and can use the setup described with Figure 3.4 and Equation (3.7).

Other than posing this path restriction instead of letting the program finding a graph on its own, the code for this problem is almost identical to the code for when the program was looking for graphs, valuation functions, and the number of agents. The only other difference is that the number of agents n is forced to be more than two, as two is the maximum degree of the graph, and from Proposition 2.7, we already know that problem instances that do not admit EF1 allocations can be constructed in this case.

3.6 Instances Under Conflict Constraints That Do Not Admit EF1 Allocations When the Valuation Functions Are Binary

To try to answer the question, "Is EF1 guaranteed when the values are binary under conflict constraints?" a similar approach to when we were interested in finding graphs that do not guarantee EF1 allocations was used. Here, we still let the program look for a graph, valuation functions and the number of agents, but we add the restriction that the values for the items must be binary.

Here, like for the open problem surrounding conflict paths, it would be best to be able to formulate a formula that could incorporate all possible problem instances, which Z3 could then use to find a counter-example or come up with a proof of why EF1 is always guaranteed for problem instances with conflicting

items and binary valuations when the number of agents is greater than two. However, formulating a program for an unbounded number of items is challenging, so our program is instead concerned with finding results when the number of items is below a specific limit.

3.7 Instances With No Constraints That Do Not Admit EFX Allocations

The setup for looking for problem instances with no EFX allocations is similar to the setup for the earlier presented programs that look for problem instances with no EF1 allocations. The only differences are that we now require that no allocation can fulfill the requirements for EFX, as explained in Section 3.2.9, and that there is now no conflict graph that restricts the possible allocations.

Chapter 4

Experimental Setup

The performance and usefulness of the programs were evaluated by running experiments that could simulate potential use cases. All experiments were run on an Intel Core i7-8550U CPU @ 1.80GHz.

4.1 Runtimes for Finding Unconstrained Allocations

To verify the programs' correctness, problem instances where the possibility of finding a valid allocation was known in advance were given to the program, and the answers were compared to the expected results . While it was possible to test both negative and positive results for the MMS program, there were only positive results to test for EF1 and EFX allocations. This is because EF1 allocations are always possible in the unconstrained case, and because it is an open question whether EFX allocations always exist, so there are no known problem instances that do not have EFX allocations.

After the programs were tested on known instances, randomly generated instances were used to gain information on the programs' runtimes. The runtimes of the programs based on Z3 were compared to analogous MIPs from the Allocations package, version 0.1.0, made by Hummel and Hetland, with Gurobi v0.11.5 as the backend for solving the MIPs.[1] Both Z3 and Gurobi were given a timeout of 5 minutes (300 seconds), and 100 randomly generated problem instances were tested for each program.[2] The MIP for finding MMS allocations was specified only to ensure that each agent gets their maximin share, not to maximize the value given to each agent as well.

[1] Source code for the Allocations package is available at https://github.com/mlhetland/Allocations.jl

[2] The MMS allocations are found by first finding each agent's maximin share with optimization and then using these values to find an allocation where each agent gets a bundle worth at least this value. The timeout of 5 minutes will be used when finding each of these maximin shares and when finding the final allocation. The final runtime when finding MMS allocations is therefore allowed to be a maximum of $5 \cdot (n + 1)$ minutes.

The running time was measured on instances where the number of agents n was randomly picked between 2 and 10, and the number of items m was randomly picked between $2 \cdot n$ and $4 \cdot n$. The number of agents is based on the maximum number of agents reported in the Spliddit app, and the number of items is inspired by the average ratio for users of the same app, which is reported to be $m/n \approx 3$ [5]. The values of each of the items are also based on how users can value their items in the Spliddit app: For each agent, a valuation was created by randomly distribution integer valuations such that the sum for each agent is approximately 1000. This was implemented as described by Hummel and Hetland: First, for each agent, each item was assigned a random real value, and then the sum for each agent was scaled to 1000 [4].

4.2 Runtimes for Finding Allocations Subject to Conflict Constraints

The program was confirmed to work as expected by seeing if the programs returned an allocation when one was known to exist and returned that it was impossible to find one when it was known that no such allocation exists, analogous to the program verification in the unconstrained case. However, in the conflict-constrained case, in contrast to the unconstrained case, there are known examples where allocations exist and where they do not exist for EF1 and EFX, as well as MMS allocations. The programs' correctness can therefore be tested more thoroughly.

Almost the same setup as for the unconstrained case was used for the runtime experiments for allocations under conflict constraints. The only difference was the additional generation of conflict graphs. The Erdős–Rényi model was used to generate conflict graphs, with the number of nodes equal to the number of items and the probability randomly selected in the interval $[0, 1)$. Graphs with $\Delta(G) \geq n$ were discarded, as an allocation may not be feasible in this case, as well as graphs with no edges, as the problem then would be an ordinary allocation problem with no constraints. For the programs with Z3, the graphs were generated with the iGraph package. For the MIPs, which were written in Julia, the graphs were generated with the Graphs package, version 1.7.4.

4.3 Runtimes for Finding Allocations Subject to Connectivity Constraints

The correctness of the program for finding EF1 allocations subject to connectivity constraints was tested by making it look for allocations where it should and where it should not be possible to find them and see if the program behaved as expected. To test that the program was able to find allocations when they were proven to exist, problem instances with a path as the underlying graph of length six were tested. This was tested for instances with two agents and

instances with four agents. It has been proven that when items are arranged on a path, connected EF1 allocations exist when there are two, three, or four agents [25]. To test the opposite, a problem instance with three agents, where the graph was a star with five nodes and all the items had the value one, was tested. This problem instance has been shown not to admit an EF1 allocation [23].

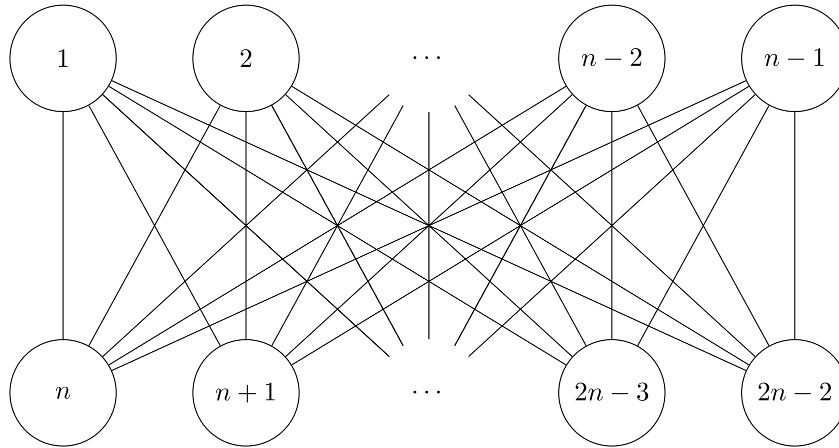
A setup very similar to the experiments for conflict constraints was used to perform the runtime experiments for the program that finds EF1 allocations subject to connectivity constraints. The number of agents and items and the valuation functions were generated in the same way, and 100 problem instances were tested. The graph was also based on the Erdős–Rényi model with the same parameters as earlier. The only difference was that graphs with $\Delta(G) \geq n$ were not discarded, as this discarding was based on results for connectivity constraints, not conflict constraints.

Another difference is that the experiments for connectivity constraints were not repeated for an analogous MIP, as there are no known MIPs to compare to. To get insight into the performance, the runtimes were compared to the runtimes of the programs that find allocations under conflict constraints and in unconstrained cases instead.

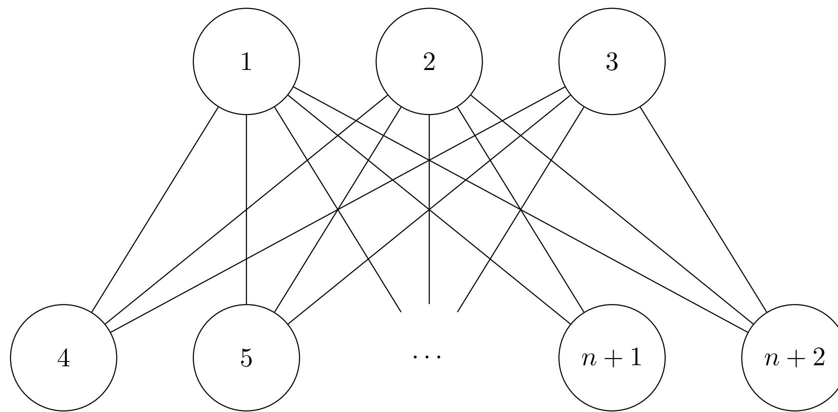
4.4 Exploring How Programs Can Look for Problem Instances With No EF1 Allocations Under Conflict Constraints

Finding Valuation Functions To evaluate how the program for finding valuation functions such that no EF1 allocation exists performs, it was asked to find such valuations for combinations of agents, items and conflict graphs where it is known that EF1 is not guaranteed. The graphs used have maximum degrees lower than the number of agents, and the cardinality of their largest connected component is more than the number of agents. Graphs with these characteristics have been chosen because it is in the interval $\mathcal{C}(G) < n < \Delta(G)$ that a complete characterization of graphs that do not guarantee EF1 allocations are left to be found, so it is in this interval that the aid of the program is relevant.

The first type instances that the program was set to find valuations for were instances with n agents, $(n-1)*2$ items, and the complete bipartite graph $K_{n-1,n-1}$ as the conflict graph, see Figure 4.1a. Problem instances like this have been shown not to guarantee EF1 allocations when $n \geq 4$ [4]. The proof demonstrates this by having the items on one side of the graph have a value of 2 and the items on the other side have a value of 3, for all agents. The second type of instance has $K_{3,n-1}$ as the conflict graph (Figure 4.1b), and thus $3+n-1 = n+2$ items. Hummel proved that when $K_{3,n-1}$ is the conflict graph, and the three items on the one side of the graph are worth 2 for all agents, while the items on the other side are worth 3 for all agents, an EF1 allocation does not exist (H. Hummel, personal communication, April 25, 2023).



(a) *The graph $K_{n-1,n-1}$*



(b) *The graph $K_{3,n-1}$.*

Figure 4.1: *Graphs where problem instances with no EF1-allocation under conflict constraints can be constructed.*

The experiments were run with an increasing number of agents, n, and items and graphs as specified above until the program could no longer produce a result within a timeout of two hours.

Finding Valuation Functions And Conflict Graph The next program to assess the performance of was the program for finding both valuation functions and a graph, such that the final problem instance does not admit any EF1 allocation. The same concept as when looking for valuation functions was used: Problem instances where it is known that no EF1 allocation exists were used as a starting point, and then components from this problem instance were removed in accordance with what components the program is able to look for. These problem instances are based on the ones with the graph $K_{3,n-1}$, which was also used when testing the program that only found valuation functions. Thus, the program will be asked to find valuation functions and graphs for instances with n agents and $3 + n - 1 = n + 2$ items. With this ratio of agents to items, we know that it should be possible for the program to find problem instances with no EF1 allocations, and we do not risk giving the program an impossible task.

The program was run with an increasing number of agents, n, and thus also an increasing number of items, until the program could no longer produce a result within a timeout of two hours.

Finding Valuation Functions, Conflict Graph and the Number of Agents This program only has to be supplied with a number of items. It is, therefore, not necessary to base the inputs on the program on known problem instances that do not admit EF1 allocations. However, it was shown by Hummel and Hetland that six is the smallest number of items for which an EF1 allocation does not exist. The program was, therefore, first given six as the number of items, and then the program was run with an increasing number of items until it could no longer produce a result within a time limit of two hours.

4.5 Search for Problem Instances That Do Not Admit EF1 Allocations When the Conflict Graph Is a Path with Non-negative, Additive Valuations

To explore the existence of EF1 when the conflict graph is a path, the program for investigating this was set to look for non-EF1 instances with a path as the conflict graph for $m \geq 6$ and with $n < m$.[3] The program was asked to find a problem instance with no EF1 allocation for an increasing number of items,

[3]Remember we already know that six is the smallest number of items a problem instance can have to not admit an EF1 allocation when $n > \Delta G$ For cases where $n \geq m$, we know that the cardinality of the largest connected component is less than or equal to the number of agents. Therefore EF1 allocations will always exist in this case.

starting with the number of items being 6. For each iteration, the program was given a timeout of five hours. The program was stopped when it could not produce a result for the specified number of items within the time limit.

4.6 Search For Problem Instances With Binary Valuations and Arbitrary Conflict Graphs That Do Not Admit EF1 Allocations

To explore the existence of EF1 under conflict constraints and binary valuations, the program for investigating this was asked to look for problem instances with no EF1 allocation for an increasing number of items. The first instance the program was asked to look for had six items. The program was asked to look for instances with an increasing number of items until it was not able to produce a result within five hours.

4.7 Search For Unconstrained Problem Instances That Do Not Admit EFX Allocations

When exploring the existence of EFX with the designated program, the strategy was similar to when exploring the existence of EF1 in the aforementioned cases. The program was asked to look for problem instances with no EFX allocation for an increasing number of agents. However, for the EFX case, the smallest number of items tested was five. This number is based on the fact that EFX allocations are proven to always exist when the number of agents is three, but it is unknown whether they exist when the number of agents is four or more. Furthermore, we are only interested in exploring cases where the number of agents is fewer than the number of items. In the case where there are the same amount of agents and items or fewer items than agents, an EFX allocation is trivial by allocating a maximum of one item to each agent. Therefore we are interested in cases where the number of items is five or more, as this can possibly answer the existence of EFX when the number of agents is four or more.

The program was run with an increasing number of items until the program could not produce a result within a time limit of five hours.

Chapter 5

Results

This chapter presents runtimes, valuation functions, graphs and other findings from the experiments described in Chapter 4.

5.1 Programs for Finding Allocations

The programs for finding EF1, EFX, and MMS allocations (described in Sections 3.3.1 to 3.3.3) behaved as expected, finding allocations when it was known that they should exist, on the problem instances used to verify correctness.

5.1.1 Comparing Runtimes For Finding Allocations: Z3 vs. Gurobi

The comparisons of the runtimes of the programs made with Z3 and the MIPs using Gurobi for the unconstrained case (Figures 5.1 to 5.3) show that the MIPs outperform the programs made with Z3.

The same contrast in performance between the program made with Z3 and the MIPs using Gurobi can be observed when the allocations are subject to conflict constraints: The MIP generally outperforms the programs made with Z3 when finding EF1, EFX and MMS allocations (Figures 5.4 to 5.6). The only exception we see is that the MIPs consistently seem to have one run that takes significantly longer to compute than the others for the same amount of items and agents. The slow run was always the first of the 100 iterations; however, the plot does not include the iteration numbers.

5.1.2 Comparing Runtimes For Finding Allocations: Connectivity Constraints, Conflict Constraints and Unconstrained

When comparing the runtimes of the programs that find unconstrained EF1 allocations, EF1 allocations with conflict constraints and EF1 allocations with

connectivity constraints, it is apparent that the program for finding allocations with connectivity constraints is significantly slower than the two other programs (Figure 5.7).

5.2 Searching for Problem Instances With No EF1 Allocations Under Conflict Constraints: Performance and Findings

Z3 was able to find valuation functions that do not admit EF1 allocations when the number of agents, items and graph was given, when the number of agents and items were given, as well as when only the number of items was given.

5.2.1 Finding Valuation Functions That Do Not Admit EF1 Allocations

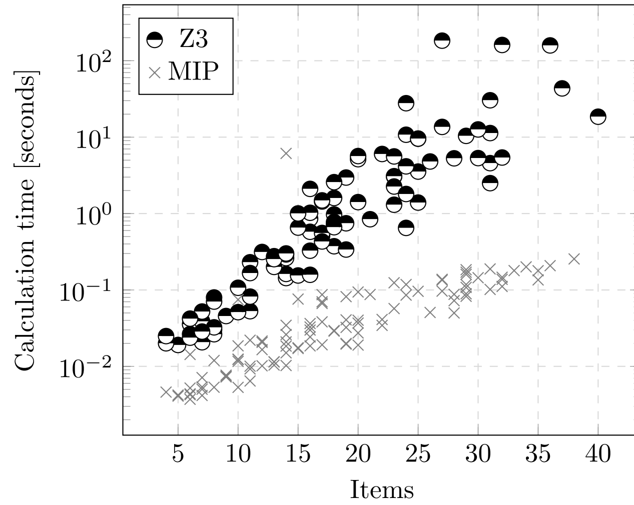
The search for valuation functions for the conflict graphs $K_{3,n-1}$ and $K_{n-1,n-1}$ show that when the number of items surpassed eight, the program was not able to find desired valuations functions for either of the problem instances within the time limit of two hours. When comparing runtimes for finding valuations for the two types of problem instances, we see that when we look at two problem instances with the same amount of items, it takes the longest to find valuations functions for the problem instance with the most agents and vice versa (Figure 5.8).

The values found by the program are not equal to the values used to prove that the problem instance does not admit EF1 allocations, but we see that the values for one side of the bipartite graph are consistently higher than the values on the other side of the graph (Tables 5.1 and 5.2).[1] The program produces the same valuations for $K_{3,3}$ on different runs (Tables 5.1 and 5.2a).

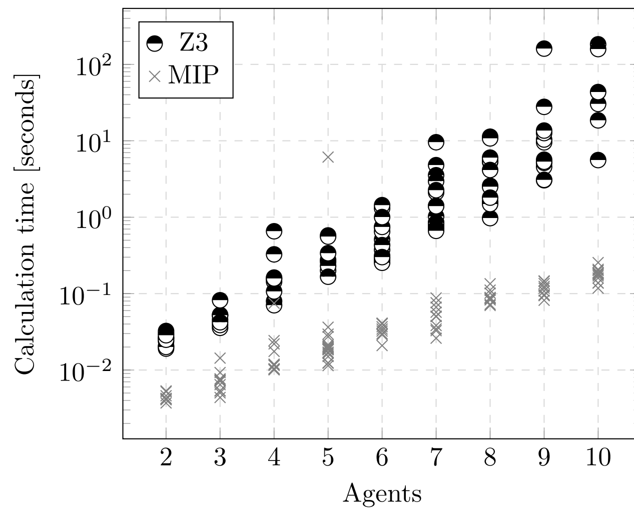
5.2.2 Comparing Runtimes for Finding Parts of Problem Instances

The comparison for finding different parts of a problem instance $\langle N, M, V, G \rangle$, shows that the more parts that have to be found, the longer the runtime (Figure 5.9). Moreover, the comparison reveals that it takes significantly more time to find problem instances when only the number of items m is given, compared to when both n and m are given and when n, m and G are given. Within the two-hour time limit, finding a problem instance with no EF1 allocation for more than six items was not possible when only the number of items, m, was

[1] In the proofs that demonstrate that each of the problem instances does not guarantee EF1 allocations, each of the items on one side of the bipartite graphs have value a value of 2 and each of the items on the other side of the graph have a value of 3. These values were used for all agents.

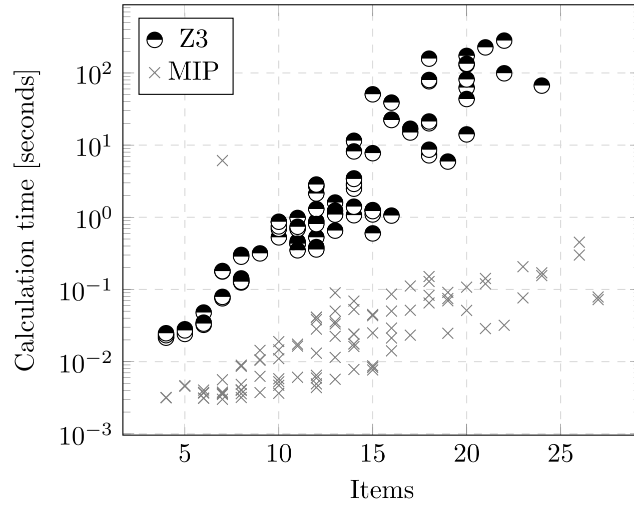


(a) *Calculation time for problem instances with different number of items.*

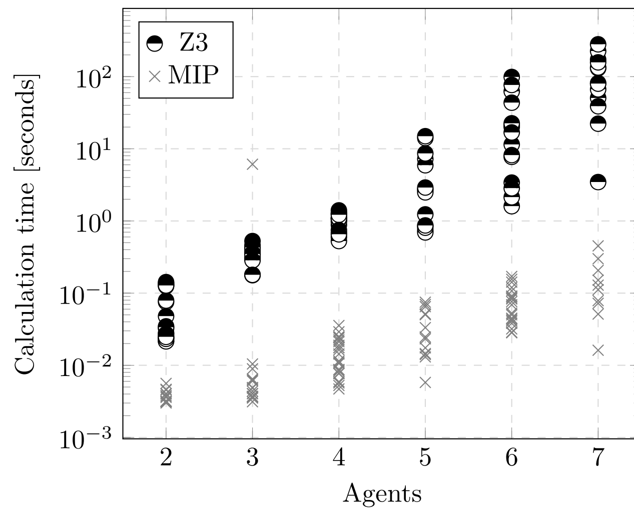


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.1: *Runtimes for Z3 and MIP for finding unconstrained EF1 allocations. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval $[2 \cdot n, 4 \cdot n]$*

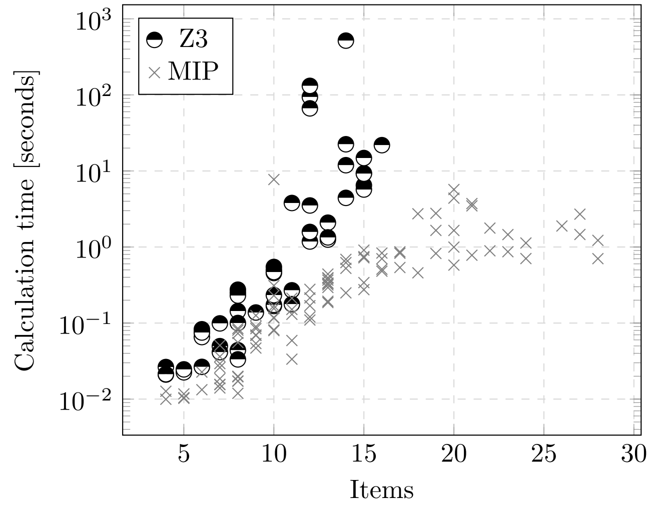


(a) *Calculation time for problem instances with different number of items.*

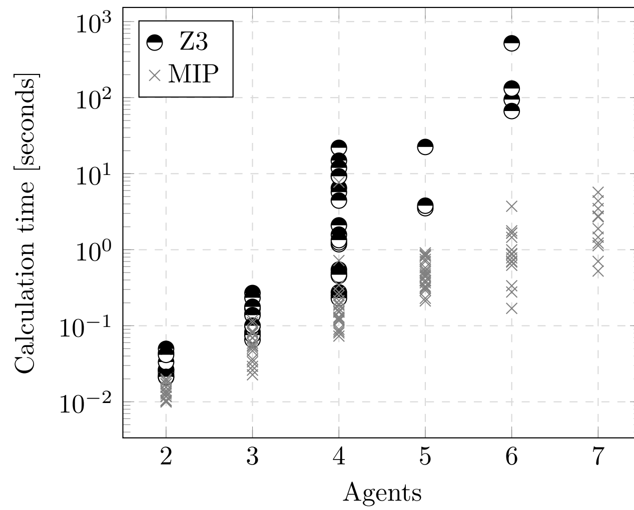


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.2: *Runtime for finding EFX allocations. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval* $[2 \cdot n, 4 \cdot n]$

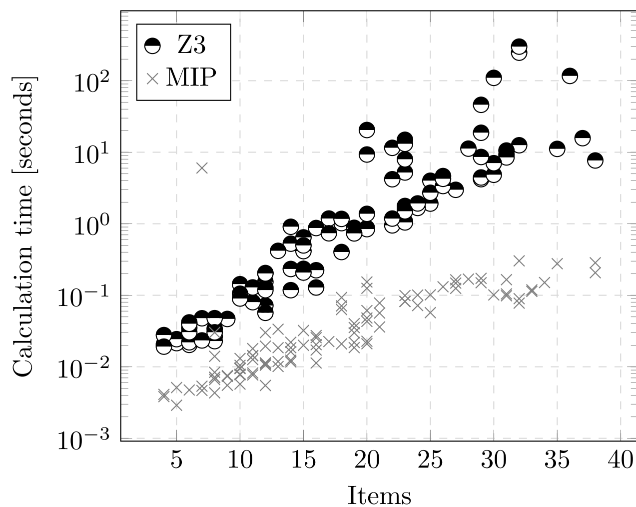


(a) *Calculation time for problem instances with different number of items.*

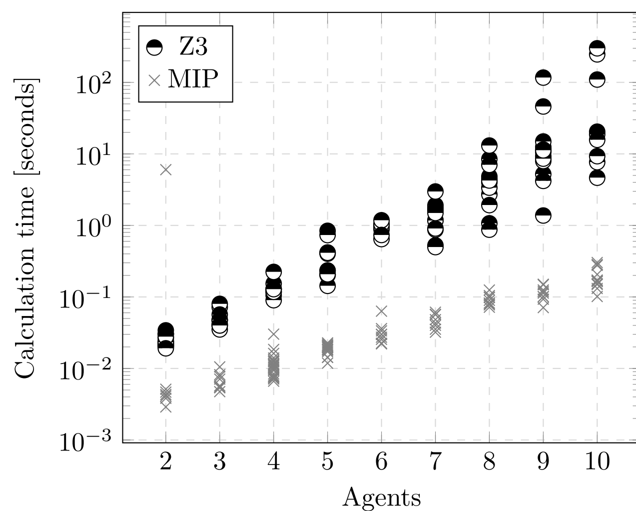


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.3: *Runtime for finding MMS allocations. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval $[2 \cdot n, 4 \cdot n]$*

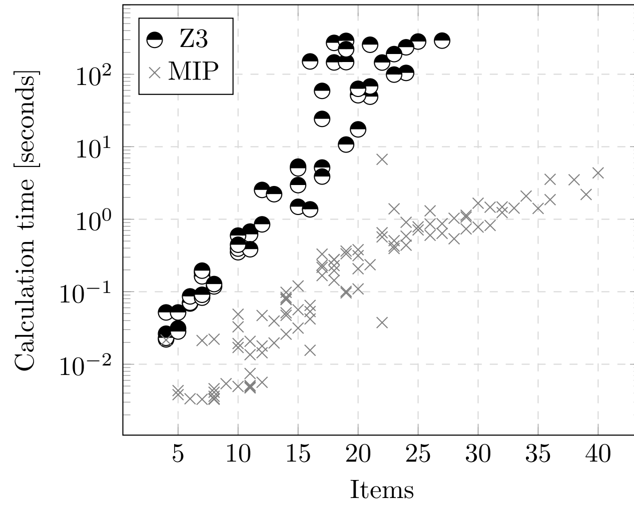


(a) *Calculation time for problem instances with different number of items.*

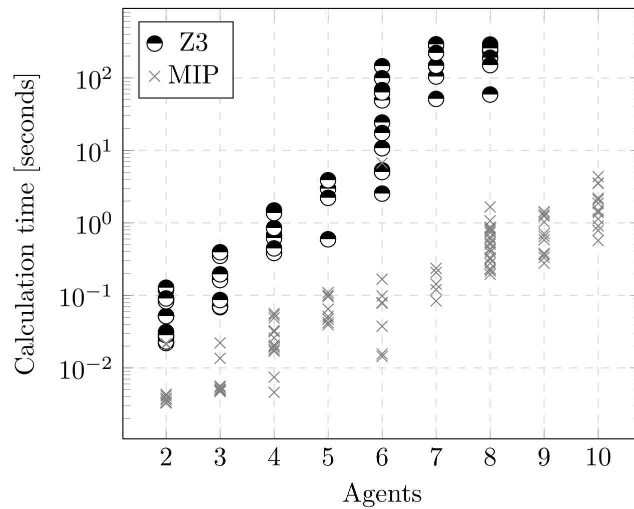


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.4: *Runtime for finding EF1 allocations subject to conflict constraints. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval $[2 \cdot n, 4 \cdot n]$*

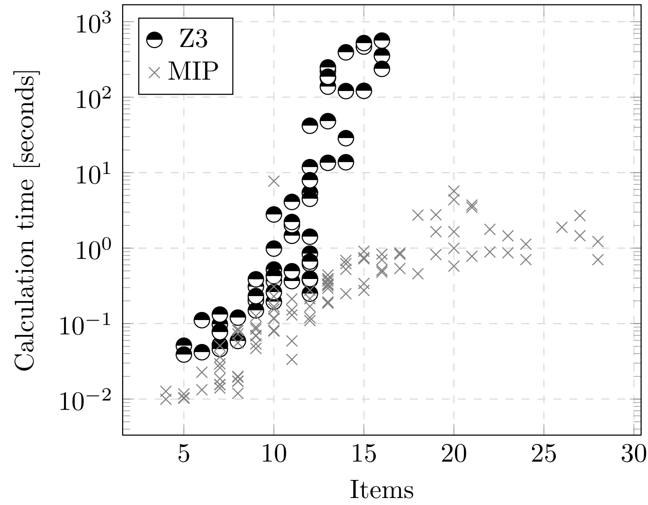


(a) *Calculation time for problem instances with different number of items.*

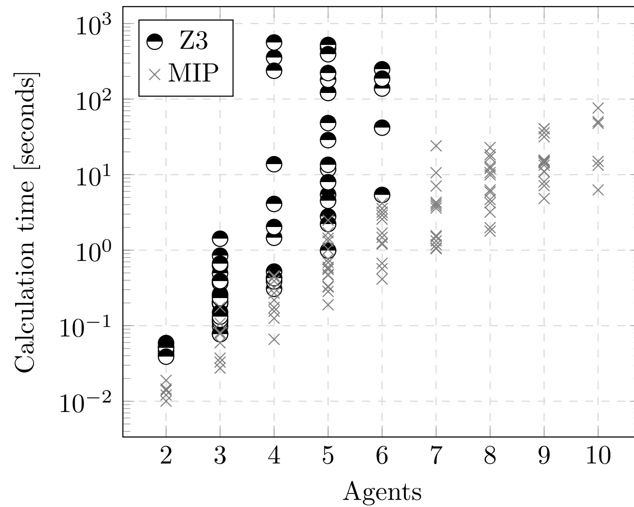


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.5: *Runtime for finding EFX allocations subject to conflict constraints. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval $[2 \cdot n, 4 \cdot n]$*

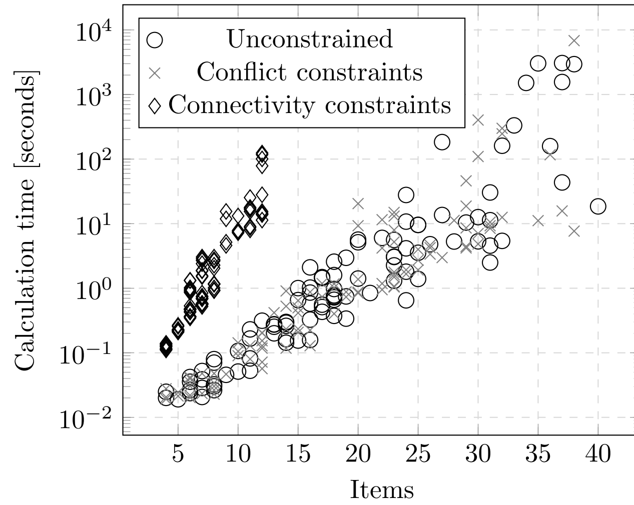


(a) *Calculation time for problem instances with different number of items.*

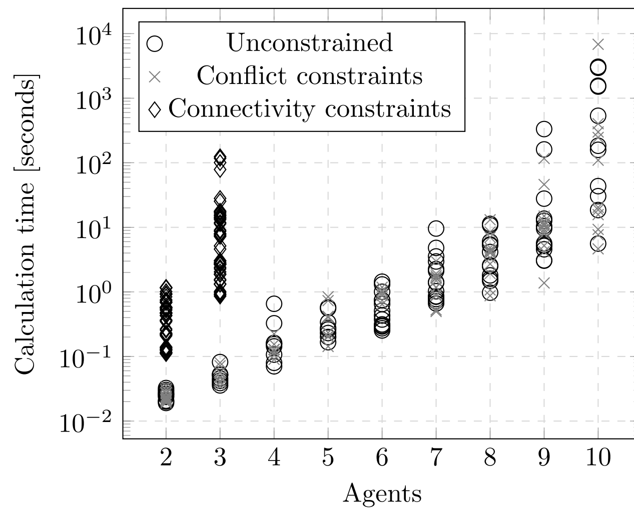


(b) *Calculation time for problem instances with different number of agent.*

Figure 5.6: *Runtime for finding MMS allocations subject to conflict constraints. The number of agents n is randomly picked between 2 and 10, and the number of items is randomly picked from the interval $[2 \cdot n, 4 \cdot n]$*

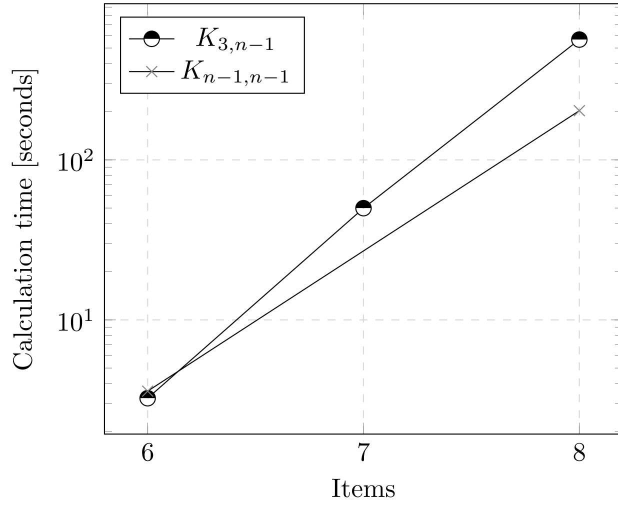


(a) *Calculation time for problem instances with different number of items.*

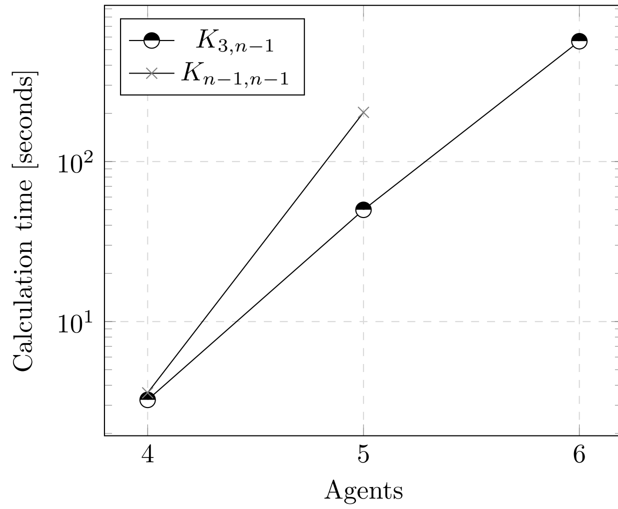


(b) *Calculation time for problem instances with different number of agents.*

Figure 5.7: *Runtime for finding EF1 allocations subject to connectivity constraints, compared to runtime for finding unconstrained EF1 allocations and EF1 allocations under conflict constraints.*



(a) *Calculation time for problem instances with different number of items.*



(b) *Calculation time for problem instances with different number of agents.*

Figure 5.8: *Runtime for finding valuation functions such that the final problem instance does not admit EF1 allocations.*

Table 5.1: *Discovered values by the program for the conflict graph $K_{3,n-1}$ such that the problem instance does not admit an EF1 allocation. The three leftmost values correspond to the three nodes on one side of the bipartite graph, while the rest correspond to values on the other side.*

(a) *Discovered values for conflict graph $K_{3,3}$ when there are 4 agents and 6 items.*

Agent i	$v_i(g_1)$	$v_i(g_2)$	$v_i(g_3)$	$v_i(g_4)$	$v_i(g_5)$	$v_i(g_6)$
1	4	3	5	0	1	2
2	10	12	11	3	8	9
3	7	8	8	4	5	6
4	9	8	10	5	6	7

(b) *Discovered values for conflict graph $K_{3,4}$ when there are 5 agents and 7 items.*

Agent i	$v_i(g_1)$	$v_i(g_2)$	$v_i(g_3)$	$v_i(g_4)$	$v_i(g_5)$	$v_i(g_6)$	$v_i(g_7)$
1	2	3	1	6	4	5	7
2	1	2	3	7	7	8	6
3	1	2	3	5	7	6	7
4	5	6	7	9	10	8	10
5	6	6	5	7	9	8	10

(c) *Discovered values for conflict graph $K_{3,5}$ when there are 6 agents and 8 items.*

Agent i	$v_i(g_1)$	$v_i(g_2)$	$v_i(g_3)$	$v_i(g_4)$	$v_i(g_5)$	$v_i(g_6)$	$v_i(g_7)$	$v_i(g_8)$
1	6	6	6	8	10	11	7	9
2	4	5	5	8	8	6	9	7
3	6	7	5	11	9	8	10	11
4	5	6	7	10	8	10	10	9
5	7	6	8	10	11	12	9	12
6	2	1	1	4	6	7	5	3

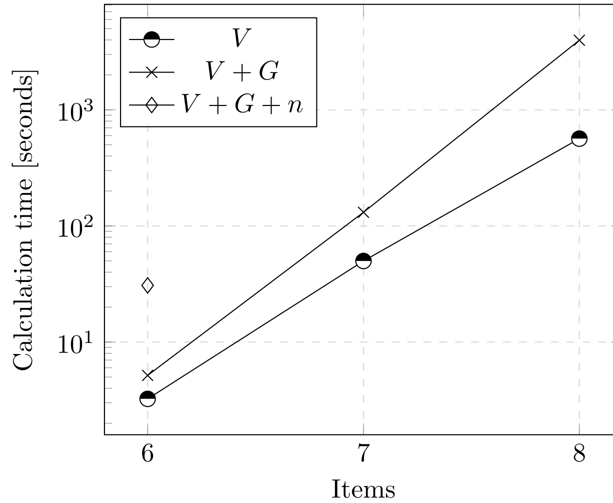


Figure 5.9: *Comparison of looking for different parts of problem instance in a way such that EF1 allocations do not exist. In the plot "V" is used for "valuation functions", "G" is used for "conflict graph", and "n" is used for "number of agents". The letters indicate what part of the problem instance the program is looking for.*

supplied. It was possible to find such problem instances for up to eight items within the time limit for the two other versions of the program.

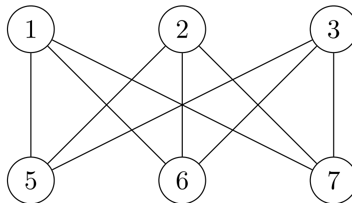
5.2.3 Graphs Discovered by Programs

The graphs that were produced when looking at different parts of the problem instance all have a complete bipartite graph as a subgraph (Figure 5.10). When the number of items was six, the same graph ($K_{3,3}$) was found when specifying both the number of agents and the number of items, as when specifying only the number of items (Figures 5.10a and 5.10b).

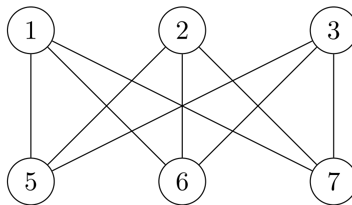
5.3 Existence of Problem Instances With No EF1 Allocations Under Conflict Constraints When the Conflict Graph's Components are Paths

That the program strengthened the hypothesis of EF1 always existing on a path by proving that EF1 allocations always exist on a path when the path has less than eight nodes (Table 5.3).[2] The time to calculate whether there exists an EF1 allocation or not increases rapidly with the number of items.

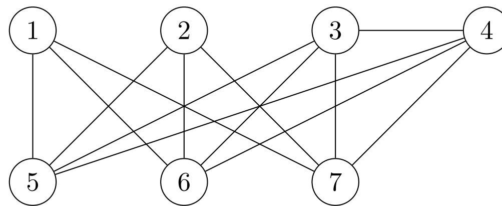
[2]It is already known that EF1 allocations always exist when the number of items is less than six



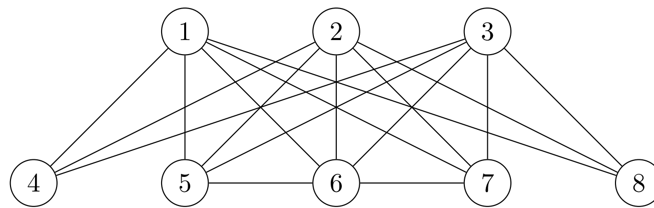
(a) *Graph found when only specifying that there should be six items.*



(b) *Graph found when specifying that there should be four agents and six items.*



(c) *Graph found when specifying that there should be five agents and seven items.*



(d) *Graph found when specifying that there should be six agents and eight items.*

Figure 5.10: *Graphs with $n > \Delta(G)$ found by the programs within a time limit of two hours.*

Table 5.2: *Discovered values by the program for the conflict graph $K_{n-1,n-1}$ such that the problem instance does not admit an EF1 allocation. The $m/2$ leftmost values correspond to nodes on one side of the bipartite graph, while the rest correspond to values on the other side.*

(a) *Discovered values for conflict graph $K_{3,3}$ when there are 4 agents and 6 items.*

Agent i	$v_i(g_1)$	$v_i(g_2)$	$v_i(g_3)$	$v_i(g_4)$	$v_i(g_5)$	$v_i(g_6)$
1	4	3	5	0	1	2
2	10	12	11	3	8	9
3	7	8	8	4	5	6
4	9	8	10	5	6	7

(b) *Discovered values for conflict graph $K_{4,4}$ when there are 5 agents and 8 items.*

Agent i	$v_i(g_1)$	$v_i(g_2)$	$v_i(g_3)$	$v_i(g_4)$	$v_i(g_5)$	$v_i(g_6)$	$v_i(g_7)$	$v_i(g_8)$
1	20	10	9	20	4	4	1	2
2	19	20	18	17	6	7	8	9
3	17	18	8	8	1	1	3	4
4	1	17	16	17	7	5	6	8
5	11	10	9	11	4	1	2	4

Table 5.3: *Runtime and result for trying to find problem instances that do not admit EF1 allocations when the conflict graph is a path. When the result of the program is that the formula is unsatisfiable, "False" is entered in the last column. A timeout of five hours was used, and a "-" indicate that the program timed out.*

Items	Runtime [s]	Non-EF1 instance possible
6	13.04	False
7	5015.64	False
8	-	Unknown

Table 5.4: *Runtime and result for trying to find problem instances that do not admit EF1 allocations when the values of the items are binary. When the result of the program is that the formula is unsatisfiable, "False" is entered in the last column. When the program has timed out, "Unknown" is entered in the last column. A timeout of five hours was used, and a "-" indicate that the program timed out.*

Items	Runtime [s]	Non-EF1 instance possible
6	11.94	False
7	8385.28	False
8	-	Unknown

Table 5.5: *Runtime and result for trying to find problem instances that do not admit EFX allocations when the number of agents is less than the number of items. When the result of the program is that the formula is unsatisfiable, "False" is entered in the last column. When the program has timed out, "Unknown" is entered in the last column. A timeout of five hours was used, and a "-" indicate that the program timed out.*

Items	Runtime [s]	Non-EFX instance possible
5	1.25	False
6	4126.91	False
7	-	Unknown

5.4 Existence of Problem Instances With No EF1 Allocations Under Conflict Constraints With Any Conflict Graph When the Valuations are Binary

The program was able to rule out that there exist problem instances with binary valuations where the number of agents is higher than the maximum degree of the graph that do not admit EF1 allocations when the valuations are binary when the number of items is less than eight (Table 5.4).

5.5 Existence of Unconstrained Problem Instances With No EFX Allocations

It was possible to rule out that there exist instances that do not admit EFX allocations when there are six or less items and the number of agents is less than the number of items (Table 5.5).

Chapter 6

Discussion

6.1 Finding Allocations

The results of Section 5.1 show that it is possible to use Z3 to find allocations of the fairness criteria EF1, EFX and MMS. However, the comparison with equivalent MIPs indicates that Z3 should not be the preferred choice when finding allocations in unconstrained cases and under conflict constraints. The anomalous runtimes observed for the MIPs are probably due to compilation done by Julia at startup.

Still, this does not prove that a program for fair allocation made with Z3 cannot have better performance than a MIP. Further optimisation of the programs may better exploit the power of Z3, possibly making it faster; there has been an account of Z3 performing better on a problem instance than a MIP with Gurobi, though in another context than fair allocation [8].

Out of all the randomly generated problem instances given to the Z3 program that calculate EFX allocations and the equivalent MIP, none were reported to not have EFX allocations within the time limit of five minutes. While this is far from proof of the existence of EFX, it does strengthen the hypothesis of there always being an EFX allocation.

Regarding constraints, the programs made for this thesis have shown that it is possible to model the conflict and connectivity constraints when using Z3. While there exist programs that model allocations under conflict constraints, there are, to my knowledge, no existing programs written to calculate allocations under connectivity constraints for any given graph. Others can hopefully find it useful to see that this is possible with Z3.

When comparing the runtime for finding EF1 allocations subject to connectivity constraints to runtimes for finding EF1 allocations that were unconstrained and under conflict constraints, it was apparent that connectivity constraints increased the runtime significantly. This may be because the transitive closure is found multiple times, both when checking that each agent gets a connected bundle and when checking that envy-freeness up to one outer good is

fulfilled for each pair of agents. Finding the transitive closure is an extra computation that is not necessary when finding allocations in unconstrained cases and under conflict constraints.

The program for connectivity constraints was only tested by looking for EF1 allocations. It would be interesting to see the program extended to have a more experimental character, like the programs developed for this thesis that look for graphs with no EF1 allocations under the conflicting items constraint. Such a program, but under connectivity constraints, could help answer the question of the existence of EF1 on a path under connectivity constraints. Bilò et al. proved the existence of EF1 on a path for up to four agents, but the question remains open for more agents.

6.2 Z3 as an Aid to Try to Solve Four Open Problems

The open questions used as a backdrop for using Z3 as an aid were concerned with the existence of allocations of certain fairness criteria under different conditions. Modifying the allocation programs to look for parts of the problem instance instead, such that no valid allocation exists, was possible.

6.2.1 Searching for Problem Instances Under Conflict Constraints That Do Not Admit EF1 Allocations

The findings indicate that the programs that look for problem instances that do not admit EF1 allocations under conflict constraints may be useful when attempting to find a full characterization of conflict graphs that guarantee EF1 allocations. When looking at the valuations proposed by the program, trends in the values were possible to see. The program may therefore help point the user toward valuation functions that seem promising for further investigation. However, the run times in the experiments also show that this program is best suited for small problem instances, thus limiting how much exploration the program can be used for.

The advantage of specifying different amounts of the problem instance is that the users do not initially need to have a clear idea of what they are looking for. For example, they may only specify the number of items and see what the program produces. After that, inspired by what the program found, they can restrict the problem instances the program can find. They can, for example, generate a specific graph with the iGraph package and see whether there are problem instances with this conflict graph that has no EF1 allocation. If there are, they can use the generated instances as inspiration when looking for more general descriptions of such problem instances. This way, the user can work *with* the program, and one idea can lead to a problem instance that leads to another or refined idea. Additionally, the restrictions posed on the problem instance when exploring do not necessarily have to be limited to the number of agents

or the whole conflict graph. While not tested for this thesis, the possibility of posing restrictions on the graph that are vaguer than the whole graph, seems promising. An example could be that the graph has to be bipartite, as this can be specified by forcing the adjacency matrix to have smaller zero matrices with dimensions equal to the number of nodes in each set.

It was not attempted to specify the number of items combined with the conflict graph and let the program look for the number of agents. It was not prioritised because the goal was not to let the user specify every possible combination of knowns and unknowns but rather explore what seems to be possible and what seems challenging to implement. However, as it was possible to create a program that could look for the number of agents, the valuation functions and the graph, it should be possible to create a program that looks only for the number of agents and the valuation functions.

However, creating a program that is only given the number of agents and has to look for the valuation functions, conflict graph, and the number of items seems more challenging. It seems more challenging because we no longer have a result in the field of fair allocation that can restrict the number of variables. When looking for the number of agents in the context of finding a problem instance with no EF1 allocation, we know that we can limit the number of agents by the number of items, as an EF1 allocation will always exist when there are more agents than items.

This is not to say that it is impossible to create a program that can find a problem instance that does not admit any EF1 allocation under conflict constraints, given only the number of agents. While it may be possible, it may require a different and less straightforward approach than what the programs in this thesis used. Z3 does offer ways to formulate formulas less tied to a specific, bounded number of variables, for example, with uninterpreted functions.

6.2.2 Looking For Counterexamples Of EF1 Being Guaranteed When the Conflict Graph is a Path

It was possible to construct a program to get insight into the existence of EF1 when the conflict graph is a path. However, here, like in the programs mentioned above, the users of the program were restricted to looking for instances where the number of items was manually chosen. Despite this restriction, it was possible to answer the question partly. The program could rule out the non-existence of EF1 allocations when the conflict graph's components are paths in cases where the number of items is less than eight and the number of agents is greater than the maximum degree of the graph.

An improvement of the program would be if it could prove the existence of (or find a counterexample of) EF1 allocations subject to conflicting constraints on a path of any size. While this program generalisation proved difficult when the conflict graph could be any graph, the task may be more achievable when we only look at paths. Z3 does support regular expressions, and the restrictions used for ensuring that the conflicts on paths are respected can be written as a regular expression. The regular expression `^\"1?(01|0)*\"$` accepts the

strings with the same pattern as what we require each row in the allocation matrix to have. That is, no consecutive items can be marked as $True$ (or 1), as that would indicate that the agent gets two items that are in conflict. This regular expression, combined with other necessary, such as restrictions that would ensure that no agents are allocated the same item, could be explored to get a more complete result on the existence of EF1 allocations on a path.

6.2.3 Looking For Counterexamples Of EF1 Being Guaranteed Under Conflict Constraints When the Valuations are Binary

When looking at the existence of EF1 under binary valuations and conflict constraints, the results were similar to those when looking at items on a path. Given that the program is correctly written and that Z3 gives correct results, it was shown that EF1 is guaranteed when the values for the items are binary, the number of items is larger than the maximum degree of the graph and the number of items is less than or equal to seven .

The result generalises, in part, the same result generalised when looking at conflict graphs that were paths: EF1 allocations always exist when the conflict graph consists of disjoint paths, the valuations are binary, and there are more than two agents. However, the generalisation is now in another direction, where we allow more graph classes but keep the requirement that the valuations are binary.

This result has been generalised in this direction before. It has already been shown by Biswas et al. that the existence is guaranteed when the conflict graph is an interval graph, which is more general than disjoint-path graphs. However, while the result found by the program with Z3 is restricted in the number of items, the result is valid for all types of graphs.

6.2.4 Looking For Counterexamples Of EFX Being Guaranteed With Unrestricted Additive Valuations

The program could report that EFX allocations were always possible for problem instances with six or fewer items.

Here, like with the programs for the other open questions, it would be favourable not to be restricted by the number of items. In the literature, it is currently unknown whether EFX allocations exist when the number of agents is four or more. Therefore, even changing the program in a way where one still restricted the number of agents, but not the number of items, could significantly contribute to the field of fair allocation.

To study the question of the existence of EFX for only four agents, one could make a more restricted version of the program used where one specifies both the number of agents and the number of items and see how high the number of items can increase before the runtime exceeds a chosen timeout. Testing this approach was not prioritised in this thesis, as the main focus was to explore

how Z3 could be used in the context of fair allocation, not to find a solution to the existence of EFX. With the program made to explore the existence of EFX, it has been shown that Z3 can be used to examine the existence of EFX for a bounded number of items. Whether it is possible for an unbounded number of items remains an open question.

6.2.5 General Thoughts on Using Z3 to Prove Something

The programs with the SMT-solver Z3 could partly answer open questions using Z3 by bounding the number of items for which the results were valid. How many items the results were valid for was limited by how much time was spent running the programs: when the size of the problem instances tested increases, so does the time to run the program. More time dedicated to running the programs and more powerful computers can presumably expand the domain for which the results are valid.

When writing programs that use Z3 to prove something, the soundness of the proofs relies on the belief that the program that uses Z3 does not have bugs and that Z3 produces correct results. While a good test suit can minimise the chance of the program having bugs, these tests can be hard to write for cases where one is interested in the existence of allocations of specific fairness measures. For example, as it is unknown whether problem instances with no EFX allocation exist, there are no available problem instances to test to verify that the program can discover such instances. However, to ensure that Z3 produces correct results, the proof can be reconstructed and verified in the interactive theorem prover Isabelle/HOL. While not done for this master thesis, Brandl et al. verified their result this way when using SMT-solving to prove the incompatibility of efficiency and strategyproofness in the context of computational social choice [13].

6.3 Using Z3 for Fair Allocation in General

While it was possible to create programs that can aid the exploration or solving of all the problems investigated, this does not necessarily mean that all problems in fair allocation are suitable to explore with Z3 or SMT-solving. We may have been lucky with the choice of problems. Fairness combined with economic efficiency is one such problem that has yet to be investigated with Z3; using valuations that are more general than additive valuations is another.

6.4 Program is Slow for Medium-Size Inputs

We see from the results presented earlier in this chapter, especially when trying to prove the existence of EF1 on conflict paths and when valuations are binary, that the runtime of the program increased significantly when the problem instances increased. More specifically, we see that when the problem instances have more than seven items, significantly larger amounts of time have to be

invested in finding a result. The case was similar when exploring the existence of EFX allocations when there were more than six items. As the programs are not suitable for finding answers for problem instances of any size, it would be nice to find results for instances that have a size that can be expected in typical practical situations. The data from the Spliddit app can give us insight into what may be expected in such a practical situation. To cover a good portion of the Spliddit cases, a result for instances of 30 items would be adequate. This estimate comes from the article *The Unreasonable Fairness of Maximum Nash Welfare* [5], where they reported that the largest instance on the Spliddit app was with 10 players and the average ratio between items and agents is approximately 3. To gain insight into paths of "normal" Spliddit length, would mean cases with up to 30 items (10 players times 3). Given the runtimes reported, this seems unrealistic to achieve with the programs described in this thesis.

Chapter 7

Conclusions

This thesis has examined and explored how the SMT-solver Z3 can be used in the context of fair allocation and how suited Z3 is for this. The following conclusions are based on the results from the experiments and the experience gathered from the work:

- Z3, and probably other SMT solvers, seems suitable to use in the context of fair allocation. The expressiveness of SMT makes it possible to formulate a range of problems related to fair allocation.
- It is possible to use Z3 to find allocations with the fairness criteria EF1, EFX and MMS in the unconstrained case, under conflict constraints and under connectivity constraints. However, in the unconstrained case and under conflict constraints, when comparing the runtimes of the programs based on Z3 to the runtimes of existing programs based on mixed integer programming with the Gurobi solver, the programs based on Z3 performed significantly poorer. Therefore, Z3 will not be recommended to find allocations in cases where an equivalent program can be written using mixed integer programming.
- It was observed that integrating conflict constraints in an allocation-finding program does not notably increase the runtime. However, adding connectivity constraints made the program significantly slower.
- Z3 can be a helpful tool to explore ideas and discover patterns in the context of the existence of EF1 allocations under conflict constraints. However, the runtimes of programs exploring the existence seem to increase exponentially with the problem size. Therefore, such programs seem best suited for exploring ideas concerning small problem instances.
- Z3 can be used to prove the existence of different allocations when the number of items is limited. Moreover, Z3 can be useful for finding counterexamples if one is able to provide an estimate of how many items are used in the counterexample. However, Z3 is best suited for problems with

a bounded number of variables and can be hard to use to prove results for arbitrary-sized problem instances.

- For additive valuations, where the values are integer, Z3 was able to establish that
  - EF1 allocations always exist under the conflicting items constraint when the conflict graph's components are paths, and the number of items is less than or equal to seven.
  - EF1 allocations always exist under the conflicting items constraint when the valuations are binary and the number of items is less than or equal to seven.
  - EFX allocations always exist when the number of items is six or less.

  It remains to verify the soundness of these results with another tool than Z3.

7.1 Future work

This thesis is a preliminary exploration of what potential Z3 and other SMT-solvers has in the field of fair allocation, and more is left to be explored. The focus of the project has been on the fairness criteria EF1, EFX and MMS in conjunction with connectivity and conflict constraints when the agents have additive valuations. Future work could look into the usability of Z3 with other constraints, like cardinality and matroid constraints, and less restrictive valuation functions, for example, monotonic valuations. Looking at chores or mixed resources instead of only positively valued goods could also be a subject of interest.

Moreover, connectivity constraints could be explored further with a program similar to the program made to explore conflict graphs. A program for exploring graphs in the context of connectivity could be a valuable tool when exploring the open question of characterizing the class of graphs that guarantee EF1 when there are more than three or more agents [25]. It would also be valuable to construct a mixed integer program for finding allocations under connectivity constraints to see if Z3 is also outperformed in this case.

While it proved difficult to model fair allocation problems with an unbounded number of variables, it is not proof that this is impossible. Looking into how this could be formulated opens the possibility for Z3 to be more powerful in the context of fair allocation. If it is possible to formulate such a program, less restricted answers could be given to the open problems studied in this thesis.

One of the open problems studied was the existence of EF1 when the items have conflicts that form a path. However, the results found were only valid for a limited number of items. As mentioned in Section 6.2.2, the restrictions posed on the allocations for each agent to ensure that they respect the conflict path can be written as a regular expression. However, the regular expression is valid for an arbitrarily long string and, thus, valid for an arbitrary number of

items. It would be interesting to see if one could circumvent the need to limit the number of items by using regular expressions, which Z3 supports.

Finally, it has only been explored how to formulate problems related to fair allocation in Z3; the soundness of the results produced by Z3 has not been verified. This verification could be carried out by reconstructing the proofs given by Z3 in Isabelle/HOL.

References

[1] B. Plaut and T. Roughgarden, "Almost envy-freeness with general valuations," *SIAM Journal on Discrete Mathematics*, vol. 34, no. 2, pp. 1039–1068, Jan. 2020, Publisher: Society for Industrial and Applied Mathematics, ISSN: 0895-4801. DOI: 10.1137/19M124397X. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/19M124397X (visited on 06/08/2023).

[2] G. Amanatidis, G. Birmpas, A. Filos-Ratsikas, A. Hollender, and A. A. Voudouris, "Maximum nash welfare and other stories about EFX," *Theoretical Computer Science*, vol. 863, pp. 69–85, Apr. 8, 2021, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2021.02.020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397521000931 (visited on 06/08/2023).

[3] B. R. Chaudhury, J. Garg, and K. Mehlhorn, "EFX exists for three agents," in *Proceedings of the 21st ACM Conference on Economics and Computation*, ser. EC '20, New York, NY, USA: Association for Computing Machinery, Jul. 13, 2020, pp. 1–19, ISBN: 978-1-4503-7975-5. DOI: 10.1145/3391403.3399511. [Online]. Available: https://dl.acm.org/doi/10.1145/3391403.3399511 (visited on 06/08/2023).

[4] H. Hummel and M. L. Hetland, "Fair allocation of conflicting items," *Autonomous Agents and Multi-Agent Systems*, vol. 36, no. 1, p. 8, Apr. 2022, ISSN: 1387-2532, 1573-7454. DOI: 10.1007/s10458-021-09537-3. arXiv: 2104.06280[cs]. [Online]. Available: http://arxiv.org/abs/2104.06280 (visited on 03/31/2023).

[5] I. Caragiannis, D. Kurokawa, H. Moulin, A. Procaccia, N. Shah, and J. Wang, "The unreasonable fairness of maximum nash welfare," Jul. 21, 2016, pp. 305–322. DOI: 10.1145/2940716.2940726.

[6] J. Lesca and P. Perny, "LP solvable models for multiagent fair allocation problems," presented at the Frontiers in Artificial Intelligence and Applications, vol. 215, Jan. 1, 2010, pp. 393–398. DOI: 10.3233/978-1-60750-606-5-393.

[7] J. C. Silva *et al.*, "Evaluation of solvers' performance for solving the flexible job-shop scheduling problem," *Procedia Computer Science*, CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN – International Conference on Project MANagement / HCist – International Conference on Health and Social Care Information Systems and Technologies 2022, vol. 219, pp. 1043–1048, Jan. 1, 2023, ISSN: 1877-0509. DOI: 10.1016/j.procs.2023.01.382. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050923003915 (visited on 04/08/2023).

[8] S. F. Roselli, K. Bengtsson, and K. Akesson, "SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, vol. jsp, Munich, Germany: IEEE, Aug. 2018, pp. 547–552, ISBN: 978-1-5386-3593-3. DOI: 10.1109/COASE.2018.8560344. [Online]. Available: https://ieeexplore.ieee.org/document/8560344/ (visited on 03/31/2023).

[9] C. Barrett, L. de Moura, and A. Stump, "SMT-COMP: Satisfiability modulo theories competition," in *Computer Aided Verification*, K. Etessami and S. K. Rajamani, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2005, pp. 20–23, ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_4.

[10] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2008, pp. 337–340, ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[11] N. Bertram, A. Levinson, and J. Hsu, *Cutting the cake: A language for fair division*, Apr. 10, 2023. DOI: 10.1145/3591293. arXiv: 2304.04642[cs]. [Online]. Available: http://arxiv.org/abs/2304.04642 (visited on 05/23/2023).

[12] P. Tang and F. Lin, "Computer-aided proofs of arrow's and other impossibility theorems," *Artificial Intelligence*, vol. 173, no. 11, pp. 1041–1053, Jul. 1, 2009, ISSN: 0004-3702. DOI: 10.1016/j.artint.2009.02.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370209000320 (visited on 05/23/2023).

[13] F. Brandl, F. Brandt, M. Eberl, and C. Geist, "Proving the incompatibility of efficiency and strategyproofness via SMT solving," *Journal of the ACM*, vol. 65, no. 2, 6:1–6:28, Jan. 31, 2018, ISSN: 0004-5411. DOI: 10.1145/3125642. [Online]. Available: https://dl.acm.org/doi/10.1145/3125642 (visited on 05/23/2023).

[14] F. Brandt, C. Saile, and C. Stricker, "Voting with ties: Strong impossibilities via SAT solving," in *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '18, Rich-

land, SC: International Foundation for Autonomous Agents and Multiagent Systems, Jul. 9, 2018, pp. 1285–1293. (visited on 05/23/2023).

[15] U. Feige, A. Sapir, and L. Tauber, *A tight negative example for MMS fair allocations*, Oct. 19, 2021. DOI: 10.48550/arXiv.2104.04977. arXiv: 2104.04977[cs]. [Online]. Available: http://arxiv.org/abs/2104.04977 (visited on 05/23/2023).

[16] M. Xiao, G. Qiu, and S. Huang, *MMS allocations of chores with connectivity constraints: New methods and new results*, Feb. 25, 2023. DOI: 10.48550/arXiv.2302.13224. arXiv: 2302.13224[cs]. [Online]. Available: http://arxiv.org/abs/2302.13224 (visited on 05/05/2023).

[17] U. Bhaskar, A. R. Sricharan, and R. Vaish, "On approximate envy-freeness for indivisible chores and mixed resources," in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2021)*, M. Wootters and L. Sanità, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), ISSN: 1868-8969, vol. 207, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 1:1–1:23, ISBN: 978-3-95977-207-5. DOI: 10.4230/LIPIcs.APPROX/RANDOM.2021.1. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2021/14694 (visited on 05/24/2023).

[18] 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*, New York NY USA: ACM, May 17, 2004, pp. 125–131, ISBN: 978-1-58113-771-2. DOI: 10.1145/988772.988792. [Online]. Available: https://dl.acm.org/doi/10.1145/988772.988792 (visited on 05/24/2023).

[19] E. Budish, "The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes," *Journal of Political Economy*, vol. 119, no. 6, pp. 1061–1103, 2011, Publisher: The University of Chicago Press, ISSN: 0022-3808. DOI: 10.1086/664613. [Online]. Available: https://www.jstor.org/stable/10.1086/664613 (visited on 05/24/2023).

[20] G. Amanatidis, G. Birmpas, A. Filos-Ratsikas, and A. A. Voudouris, *Fair division of indivisible goods: A survey*, Mar. 4, 2022. DOI: 10.48550/arXiv.2202.07551. arXiv: 2202.07551[cs]. [Online]. Available: http://arxiv.org/abs/2202.07551 (visited on 04/12/2023).

[21] B. L. Deuermeyer, D. K. Friesen, and M. A. Langston, "Scheduling to maximize the minimum processor finish time in a multiprocessor system," *SIAM Journal on Algebraic Discrete Methods*, vol. 3, no. 2, pp. 190–196, Jun. 1982, ISSN: 0196-5212, 2168-345X. DOI: 10.1137/0603019. [Online]. Available: https://epubs.siam.org/doi/10.1137/0603019 (visited on 05/25/2023).

[22] D. Kurokawa, A. Procaccia, and J. Wang, "When can the maximin share guarantee be guaranteed?" *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 30, no. 1, Feb. 21, 2016, Number: 1, ISSN: 2374-3468. DOI: 10.1609/aaai.v30i1.10041. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/10041 (visited on 05/25/2023).

[23] X. Bei, A. Igarashi, X. Lu, and W. Suksompong, "The price of connectivity in fair division," *SIAM Journal on Discrete Mathematics*, vol. 36, no. 2, pp. 1156–1186, Jun. 30, 2022, Publisher: Society for Industrial and Applied Mathematics, ISSN: 0895-4801. DOI: 10.1137/20M1388310. [Online]. Available: https://epubs.siam.org/doi/abs/10.1137/20M1388310 (visited on 05/26/2023).

[24] A. Biswas, Y. Ke, S. Khuller, and Q. C. Liu, *An algorithmic approach to address course enrollment challenges*, Apr. 17, 2023. DOI: 10.48550/arXiv.2304.07982. arXiv: 2304.07982[cs]. [Online]. Available: http://arxiv.org/abs/2304.07982 (visited on 05/05/2023).

[25] V. Bilò *et al.*, "Almost envy-free allocations with connected bundles," *Games and Economic Behavior*, vol. 131, pp. 197–221, Jan. 2022, ISSN: 08998256. DOI: 10.1016/j.geb.2021.11.006. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0899825621001524 (visited on 05/26/2023).

[26] R. Sebastiani, "Lazy satisfiability modulo theories," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 3, no. 3, pp. 141–224, Dec. 1, 2007, ISSN: 15740617. DOI: 10.3233/SAT190034. [Online]. Available: https://www.medra.org/servlet/aliasResolver?alias=iospress&doi=10.3233/SAT190034 (visited on 04/02/2023).

[27] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, ser. STOC '71, New York, NY, USA: Association for Computing Machinery, May 3, 1971, pp. 151–158, ISBN: 978-1-4503-7464-4. DOI: 10.1145/800157.805047. [Online]. Available: https://dl.acm.org/doi/10.1145/800157.805047 (visited on 04/24/2023).

[28] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Chapter 33. satisfiability modulo theories," in *Frontiers in Artificial Intelligence and Applications*, A. Biere, M. Heule, H. Van Maaren, and T. Walsh, Eds., IOS Press, Feb. 2, 2021, ISBN: 978-1-64368-160-3 978-1-64368-161-0. DOI: 10.3233/FAIA201017. [Online]. Available: http://ebooks.iospress.nl/doi/10.3233/FAIA201017 (visited on 04/24/2023).

[29] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman, "Chapter 2 satisfiability solvers," in *Foundations of Artificial Intelligence*, ser. Handbook of Knowledge Representation, F. van Harmelen, V. Lifschitz, and B. Porter, Eds., vol. 3, Elsevier, Jan. 1, 2008, pp. 89–134. DOI: 10.1016/S1574-6526(07)03002-7. [Online]. Available: https://www.

sciencedirect.com/science/article/pii/S1574652607030027 (visited on 04/24/2023).

[30] M. R. Prasad, A. Biere, and A. Gupta, "A survey of recent advances in SAT-based formal verification," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 2, pp. 156–173, Apr. 1, 2005, ISSN: 1433-2787. DOI: 10.1007/s10009-004-0183-4. [Online]. Available: https://doi.org/10.1007/s10009-004-0183-4 (visited on 04/24/2023).

[31] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/368273.368557. [Online]. Available: https://dl.acm.org/doi/10.1145/368273.368557 (visited on 04/03/2023).

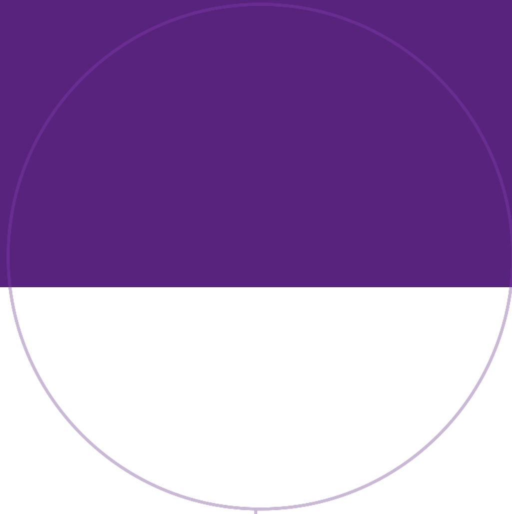
[32] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, vol. 7, no. 3, pp. 201–215, Jul. 1, 1960, ISSN: 0004-5411. DOI: 10.1145/321033.321034. [Online]. Available: https://dl.acm.org/doi/10.1145/321033.321034 (visited on 04/24/2023).

[33] B. Selman, H. Levesque, and D. Mitchell, "A new method for solving hard satisfiability problems," presented at the Proceedings Tenth National Conference on Artificial Intelligence, Jul. 12, 1992.

[34] B. Selman, H. Kautz, and B. Cohen, "Noise strategies for improving local search," *Proceedings of the National Conference on Artificial Intelligence*, vol. 1, Sep. 30, 1999.

[35] C. Barrett and S. Berezin, "CVC lite: A new implementation of the cooperating validity checker," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 515–518, ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_49.

[36] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang, "Zapato: Automatic theorem proving for predicate abstraction refinement," in *Computer Aided Verification*, R. Alur and D. A. Peled, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2004, pp. 457–461, ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_36.



Norwegian University of
Science and Technology