Master's thesis

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

Jørgen Steig

# Multiple Opinionated Knapsacks

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
Co-supervisor: Halvard Hummel
June 2023

NTNU
Norwegian University of
Science and Technology

Jørgen Steig

# Multiple Opinionated Knapsacks

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
Co-supervisor: Halvard Hummel
June 2023

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

NTNU
Norwegian University of
Science and Technology

# Multiple Opinionated Knapsacks

Jørgen Steig

June 12, 2023

**Abstract**

Allocating resources, whether fairly or optimally, is a fundamental challenge in many parts of society. We investigate the allocation of a set of indivisible items amongst agents which each have a budget. Each item has two properties: a profit and a cost. The profit depends on which agent it is allocated to, while the cost is the same for every agent. The goal is to maximize the total amount of profit, while ensuring that the total cost of the items allocated to an agent does not exceed that agent's budget. We call this problem the *multiple opinionated knapsack problem*, and present an exhaustive algorithm for finding optimal solutions to it, which intelligently prevents investigating suboptimal solutions. The *multiple knapsack problem* is identical, except that the profit of an item is the same for all agents, i.e., the agents are not opinionated. We adapt established optimizations for the multiple knapsack problem to the opinionated variant, and review their efficacy. Additionally, we consider which of these optimizations could remain useful if one wished to adopt the algorithm to find fair allocations of items as opposed to optimal ones.

We find that most of the optimizations survive the transition to subjective profits rather well, although the increased complexity of the new problem does have its consequences on how effective each optimization is. There are also certain optimizations that rely on arguing that the profit of an item is independent of which agent receives it, which naturally have to be excluded. Unfortunately our implementation is not efficient enough to outcompete current Integer Linear Program (ILP)-solvers, but some of the optimizations may prove useful when adapting the algorithm to find fair allocations, as ILP-solvers are generally ill-suited for such tasks.

**Sammendrag**

Fordeling av ressurser, enten med mål om rettferdighet eller optimalitet, er en grunnleggende problemstilling i flere deler av ethvert samfunn. Vi utforsker fordeling av udelelige gjenstander mellom agenter som alle har et eget budsjett. En gjenstand har to attributter: profitt og kostnad. Profitten til en gjenstand varierer basert på hvilken agent som får den, mens kostnaden er lik for alle agentene. Målet er å maksimere den totale summen av profittene til hver agent, under kravet om at summen av kostnadene til en agent ikke overskider budsjettet. Vi kaller dette problemet "the multiple opinionated knapsack problem", og presenterer en algoritme som vurderer alle mulige fordelinger, untatt de den kan bevise at ikke er optimale. "The multiple knapsack problem" er identisk, unntatt at profitten til en gjenstand ikke endres basert på hvilken agent som får den. Vi endrer eksisterende optimaliseringer for the multiple knapsack problem slik at de passer til varianten med subjektive profitter, og vurderer bidragene deres etter endringene. I tillegg gjøres en vurdering av hvilke av optimaliseringene som kan ha verdi dersom vi ønsker å endre algoritmen til å finne rettferdige fordelinger i stedet for optimale.

Det viser seg at de fleste av optimaliseringene kan endres til å passe subjektive profitter, og fortsatt redusere kjøretiden betydelig, til tross for at alle optimaliseringene blir svakere grunnet den økte kompleksiteten av problemet. Enkelte av optimaliseringene baserer seg og på å argumentere for at profitten en gjenstand genererer er uavhengig av hvilken agent de blir allokert til, som naturligvis gjør at de ikke kan anvendes med subjektive profitter. Vi konkluderer med at vår implementasjon av algoritmen ikke klarer å utkonkurrere tilgjengelige algoritmer for å løse "Integer Linear Program"(ILP) problemer, men at enkelte av optimaliseringene kan anvendes for å løse rettferdig fordelingsproblemer, som de generiske løsningene for ILP'er sliter med.

# Acknowledgements

I would like to sincerely thank my supervisor Magnus Lie Hetland for all his help throughout this past year, from helping sculpt a project we both found interesting, to the consistently great input in our weekly talks. Additionally, a big thank you to my co-supervisor Halvard Hummel, specifically for providing valuable feedback on every page of a 60-page document in roughly the time it takes Usain Bolt to run 100 meters.

My friends and deskmates also deserve some credit for the innumerable unproductive breaks, as well as the consistently great banter which helped this year be as entertaining as it turned out to be. I'm genuinely not sure whether I learned more about knapsack problems or Mario Kart..

Massive thanks go out to my parents for an amazing upbringing and constant support throughout. Although helping with my homework isn't as easy at it used to be, I greatly appreciate your commitment to supporting me in any way you can. My mom also deserves a shoutout for buying the best festival tickets ever, which allowed me to meet Guro. You've been an exclusively amazing addition to my life, thanks for the great year!

Sist, men ikke minst, tusen takk til Ola. Du er den viktigste personen i livet mitt, og du har en fantastisk evne til å gjøre enhver situasjon til en komedie. Sorry for at jeg har vært borte i 5 år nå, gleder meg veldig til å komme hjem og ta noen skulkedager hvor vi raider Metallco. Både verdens beste **lille**bror og klækost!

Vroom Vroom!

# Table of Contents

# List of Figures

# List of Tables



# List of Algorithms

# Chapter 1

# Introduction

The *0-1 knapsack problem* (KP) is an optimization problem which involves filling a single container, also called a knapsack, with items. A knapsack has a capacity which represents how much it can contain, while each item has both a profit and a cost. The goal is to maximize the total profit in the knapsack, while ensuring that the items' combined costs do not exceed the capacity of the knapsack. It has several real world applications, from literally filling containers that are limited in volume or weight capacity with items that have a volume or weight, to scheduling problems, where the capacity of the knapsack represents some time slot, and the weight of an item is how much of that time slot it takes up.

The *multiple* knapsack problem (MKP) is a natural extension of the knapsack problem, where we have multiple knapsacks with differing capacities that we can distribute the items into. The optimal solution to the MKP is obviously different from the optimal solution to a single knapsack problem in which we create one knapsack with the combined capacities of every knapsack, as an item bigger than any single knapsack in the MKP may be possible to allocate to the combined knapsack. The introduction of several knapsacks introduces more flexibility to the problem, and can be useful in modeling many real-world scenarios. The multiple knapsack problem may for instance be used to figure out how to optimally distribute your belongings between the suitcase and the backpack you plan to bring on vacation. In this scenario, the profit of an item would be some measure of how badly you want to bring it. This allows us to prioritize which items to exclude, given that there isn't enough space to bring everything you would like.

In the spirit of continuing to generalize this problem, there also exists a version called the *generalized assignment problem* (GAP). It works much like the MKP, where the multiple knapsacks can have differing capacities, but it allows the costs and profits of the items to vary as well. Specifically, each knapsack has their own opinion on both how much an item costs and how much profit it generates. This opens up for even more exact representations of real-world scenarios.

Continuing with the earlier vacation example, we may for example find it more valuable to bring our headphones with us in the backpack as opposed to having them stashed in the suitcase, since it allows us to listen to music on the flight. We would represent this by giving the headphones a larger profit if they are placed in the backpack than in the suitcase. Likewise, we may wish to adjust the weight of an item based on the container. Since a suitcase has to abide by a maximum weight at the airport while you can essentially get away with as heavy of a backpack as you'd like, a small but heavy item can have a large cost for the suitcase, and a small cost for the backpack. What this essentially allows us to do is to change what the capacity of the container represents, with the capacity of the suitcase in this scenario being a maximum weight, while the capacity of the backpack represents its total volume.

The downside with treating the meaning of the cost of an item this way is that we may lose information about an important aspect of the item. When changing the cost of the small and heavy item to represent its weight instead of its volume for the suitcase, we are essentially committed to changing the suitcase's capacity to be weight, not volume. While this allows us to ensure the items we place in our suitcase won't make it too heavy for the weigh-in, there is no longer a restriction on how much volume fits in the suitcase, and so there is no guarantee that the items assigned to the suitcase will physically fit within it anymore. To properly model such realistic scenarios one would need to implement several capacities for each knapsack, e.g., one for volume and one for weight, which is another possible generalization.

Seeing as there are many real-world scenarios where the cost of some item is objective, e.g., weight, volume, time spent, money etc., an appropriate middle ground between the MKP and the GAP might be for the profits of an item to be subjective, while the cost remains objective. Having not discovered a name for this problem formulation in the literature, we will call this the *multiple opinionated knapsack problem* (MOKP).

## 1.1 Our Work

The goal of this project is to explore whether optimization concepts for the MKP can be translated to more dynamic settings, chiefly subjective valuations, and to assess their efficacy after the transition. The long-term motivation being to lay a foundation for utilizing branch-and-bound to find optimal solutions to non-linear problems, which often arise in the field of fair allocation.

We present a branch-and-bound algorithm based on bin-completion for exactly solving multiple opinionated knapsack instances. After presenting some relevant theory in section 2, we thoroughly review existing optimizations for branch-and-bound in the context of the MKP in section 3, and discuss which of them can be modified to fit the MOKP, and which of them are incompatible. Section 4 describes the algorithm in its entirety, before experiments are conducted to assess the efficacy of each optimization compared to their implementation in the MKP in sections 5 and 7. At the end we share some thoughts on how the algorithm may be improved in the future, and how MOKP may be altered to solve problems in fair allocation.

## 1.2 Related Work

Pisinger has made several contributions to both the KP and the MKP, and has written a book with Kellerer and Pferschy [13] which covers more than one could ever wish to know about the knapsack problem and the family of problems surrounding it. Fukuanga and Korf [10, 11] improved upon previous efforts to solve the MKP, and came up with the branch-and-bound algorithm *bin-completion*, which has inspired much of the structure of our own solution. Martello and Toth, in addition to making several important contributions to efficiently solving the MKP [17], also have some work covering algorithms for solving GAPs [18].

In addition, Martello contributed to an article covering a new algorithm for solving MKPs by Dell'Amico et al. [7], which relies on a hybrid methodology. In essence, it first searches for optimal solutions using branch-and-bound for a set time limit. If the optimal solution is not found within this time limit, the problem is modeled as a graph, in which the edges represent items and the nodes represent partial knapsack fillings. One path then represents the complete allocation to a knapsack, and the goal is to find the set of paths which maximize

total profit. This approach produced successful results compared to previous efforts by utilizing the strengths of different solution techniques on differing instances.

The initial motivation for this project was to contribute to the field of *fair allocation*, where we aim to allocate indivisible items to agents in a way which satisfies every agent as much as possible. The measure we use to say something about the fairness of a allocation is called a *fairness criterion*. EF-1 is one such fairness criterion, but what actually constitutes a fair allocation is obviously more of a philosophical inquiry than an algorithmic one. We specifically wished to expand on Wu et al.'s work on budget-constrained fairness scenarios [25]. In this setting each agent has a budget, essentially the capacity constraint on a knapsack, and each item has an objective weight and a subjective value. Instead of attempting to maximize the *Utilitarian Social Welfare* (the sum of profits) as in the MKP and MOKP, they investigate the maximum *Nash Social Welfare* (NSW) solution, which aims to maximize the product of the profits of each agent. The setting for the MOKP and the budget-constrained fair allocation problem are identical, but they optimize for different welfare measures. They show that the maximum NSW-solution cannot guarantee an *EF1*-allocation, but that it can guarantee $\frac{1}{4}$-approximate EF1 and *Pareto-optimality*. We will describe these terms in more detail in section 2.7.

Shortly before the submittal of this thesis, Barman et al. published an article expanding on the work of Wu et al. [1]. They proposed a greedy polynomial algorithm for generating fair allocations under budget constraints with subjective valuations. This algorithm guarantees an EF-2 allocation, implying an EF-2 allocation always exists under budget constraints with subjective valuations. The result also applies for the case where we have subjective weights instead of valuations, but not for the case where both are subjective.

# Chapter 2

# Theory

To ensure a consistent understanding of the terms used and to introduce some central topics, we will briefly go over some definitions and concepts related to both multiple knapsack and fair allocation problems. Note that sections 2.5 and 2.6.2, as well as 3.3.7 are largely inspired by the precursor project to this thesis [23].

## 2.1 Multiple Knapsack

Throughout this thesis we may also refer to a knapsack as a container, bin or agent. Given $m$ knapsacks and $n$ items, and letting:

- $1 \leq i \leq m$
- $1 \leq j \leq n$
- $c_i$ be the capacity of knapsack $i$
- $w_j$ be the weight, volume or cost of item $j$
- $p_j$ be the profit or value of item $j$
- $x_{ij}$ be a binary variable, where $x_{ij} = 1$ means item $j$ was placed in knapsack $i$

The MKP can be formulated as an *Integer Linear Program* (ILP), where the variables are restricted to be integers, and both the constraints (limits on what values $x_{ij}$ can take) and the objective function are linear. The ILP-formulation of the MKP is the following:

$$\text{Maximize:} \qquad \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \tag{2.1.1}$$

$$\text{Constrained by:} \qquad \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \qquad \text{i} = 1, \ldots, \text{m} \tag{2.1.2}$$

$$\sum_{i=1}^{m} x_{ij} = 0 \parallel 1, \qquad \text{j} = 1, \ldots, \text{n} \quad (2.1.3)$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \quad (2.1.4)$$

Where constraint (2.1.2) ensures the assignment to each knapsack is *feasible* (the items fit within the knapsack), constraint (2.1.3) ensures no item is allocated to more than 1 knapsack, and constraint (2.1.4) defines $x_{ij}$ as a binary variable. This means that we do not consider divisible items, which can be useful for modelling real-world scenarios where two halves of an item have significantly less value than the original, e.g., a car.

In addition, we will assume $c_i$, $w_j$ and $p_j$ to all be positive integers. Note that when the inputs are rational numbers, one may simply multiply them by a common multiple of their denominators to transform them into integers. This assumption will extend to the other problems mentioned below.

## 2.2 Generalized Assignment Problem

The general assignment problem is a general formulation of a large family of optimization problems related to filling containers. MKP exists in this family, and is a specific, simplified version of GAP. Using the same variables as we did with the MKP, but changing them slightly such that

- $w_{ij}$ is the weight of item $j$ when placed in knapsack $i$.
- $p_{ij}$ is the profit of item $j$ when placed in knapsack $i$

we can also formulate the GAP as an ILP:

$$\text{Maximize:} \qquad \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \quad (2.2.1)$$

$$\text{Constrained by:} \qquad \sum_{j=1}^{n} w_{ij} x_{ij} \leq c_i, \qquad \text{i} = 1, \ldots, \text{m} \quad (2.2.2)$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{j} = 1, \ldots, \text{n} \quad (2.2.3)$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \quad (2.2.4)$$

Note that constraint (2.2.3), in contrast to constraint (2.1.3) for the MKP, requires that an item be allocated to exactly one knapsack. This means that technically, any instance where there are more items than can possibly fit within the knapsacks have no feasible solution since excluding items is not an option. This can be solved by borrowing the concept of a charity from fair allocation [5], essentially an additional knapsack with infinite capacity which considers every item to be worth nothing. By donating the leftover items to the charity, we can obtain a feasible solution.

## 2.3 Multiple Opinionated Knapsack Problem

We are interested in solving a version of the GAP where the weights of the items are objective, i.e., the same for every container, such that $w_{ij} \to w_j$ in (2.3.2), and where solutions are considered feasible even if not every item is assigned, i.e. changing (2.2.3) such that $\sum_{i=i}^{m} x_{ij}$ can also equal 0.
The ILP formulation of the multiple opinionated knapsack problem is therefore:

$$\text{Maximize:} \qquad \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \quad (2.3.1)$$

$$\text{Constrained by:} \qquad \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \qquad \text{i} = 1, \ldots, \text{m} \quad (2.3.2)$$

$$\sum_{j=1}^{n} w_j x_{ij} \leq c_i, \qquad \text{i} = 1, \ldots, \text{m} \quad (2.3.3)$$

$$\sum_{i=1}^{m} x_{ij} = 0 \parallel 1, \qquad \text{j} = 1, \ldots, \text{n} \quad (2.3.4)$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j \qquad (2.3.5)$$

Note that when we refer to the "MOKP" we always mean the multiple opinionated knapsack problem, and not the *multiobjective knapsack problem* with the same acronym.

## 2.4 Related Container Problems

There are several container-filling optimization problems, often referred to as *multicontainer packing problems*, which can be used to solve tangential problems. A closely related problem to the MKP is the *bin packing problem*, in which the items are devoid of profit, and the objective is to fit all the items in as few bins with identical capacity as possible. Similar to the MKP, bin packing is also strongly NP-hard. It has applications in several fields, such as transportation and memory allocation in computers. It can be solved by the same bin-completion algorithm that inspired our own solution [10, 11] given adjustments to domination criteria, upper bounds etc. The bin packing application of bin-completion was later improved upon by Schreiber and Korf [22].

The dual problem of bin packing is *bin covering*, where the items are still devoid of profit, but each bin has an identical quota instead of a capacity. Each bin is assigned a set of items, the weights of which must sum to at least the quota. The goal is then to maximize the number of bins that can be filled to at least their quota. Reformulating the problem such that the weight of an item instead represents its value, and the quota of a knapsack is some demand from the agents for how much value they ought to receive, we can quickly see the relevance of bin covering to fair allocation. Whilst perhaps not being best suited for allocating items fairly unless we are fine with distributing the items amongst arbitrarily many agents, the bin covering problem is well-suited for answering questions such as: "How many agents can we grant their proportional share of the total objective value of the items?", where the quota for each agent is set to $\frac{1}{m}$ of the total objective value.

## 2.5 Complexity

While the single knapsack problem is possible to solve in pseudo-polynomial time with a dynamic programming approach, the multiple knapsack problem does not grant the same luxuries. The MKP has been shown to be strongly $\mathcal{NP}$-complete [26], and does not admit a *fully polynomial-time approximation scheme* (FPTAS) [13]. A FPTAS would allow for an approximate solution that is polynomial in the size of the problem and in $\frac{1}{\epsilon}$, where $\epsilon$ is the fraction of error you allow for. Therefore, the possible approaches are narrowed to *polynomial-time approximation schemes* (PTAS), which still require the function to be polynomial in the size of the problem, but allows for it to be exponential in the $\frac{1}{\epsilon}$-term.

For instance, a FPTAS may have a runtime of $n \times \frac{1}{\epsilon}$, while a PTAS allows for runtimes like $n^{\frac{1}{\epsilon}}$. In this example, if we were to decrease $\epsilon$ from 0.1 to 0.01 (10x the accuracy), the PTAS would go from $n^{10}$ to $n^{100}$, while the FPTAS would go from $10n$ to $100n$. This a considerable difference, which is why a FPTAS would be preferable. Nonetheless, one does not exist for the MKP unless $\mathcal{P} = \mathcal{NP}$, so branching strategies remain the preferred option.

## 2.6 Branch-and-Bound

*Branch-and-bound* (B&B) is a well known strategy for solving optimization problems. Whilst any optimization problem can, given enough time, be solved with brute force by enumerating through every solution and finding the maximum objective value among the feasible ones, the number of possible solutions grows exponentially with the problem size. Recall the ILP-formulation of MOKP in section 2.3. A solution is given by assigning a binary value to every $x_{ij}$, which there are $k = m \cdot n$ of. The number of possible configurations of $k$ binary variables is given by $2^k$, such that for an instance with 10 knapsacks and 28 items, there are more solutions than atoms in the universe. Naturally, the constraints severely limit how many of these solutions are feasible, and any remotely viable algorithm for solving moderately large instances must therefore limit its search to the set of feasible solutions.

Instead of exploring the entire set of feasible solutions, branch-and-bound aims to intelligently eliminate (or only implicitly explore) feasible solutions which cannot be the optimal solution. We explore these solutions by generating a tree. The root node represents the entire problem, and we create nodes called

children of the root for each decision we have to make. For instance, the layer of nodes under the root node may represent every knapsack we can put item 1 into. Each of those children get their own children, which represent in which knapsack we place item 2. Every child node represents a subproblem, in which we have already committed to making the choices given by the path from the root down to that node. Eventually this will generate every feasible combination of items in knapsacks. When none of the remaining items can be placed in any of the knapsacks, or we are out of items, we have a complete solution, and the nodes are called leaf nodes.

The strategy can be explained in terms of the words that compose it. We *branch* to build the tree by exploring feasible subsets of the search space, which alone would constitute a brute force algorithm. The *bounding* principle is what constitutes the intelligent elimination of non-optimal solutions, and is based on computing two values, a lower and upper bound.

Assuming we are aiming to maximize some function $f(x)$, the upper bound is some optimistic value, which is guaranteed to be larger than or equal to the optimal value of $f(x)$. The lower bound is a pessimistic estimate, which is less than or equal to the optimal value. Whilst exploring the tree we can set the lower bound to be the value of the best solution thus far, and calculate an upper bound for the current subproblem.

Considering a node in the tree where we have already allocated some items and removed them from the problem, the upper bound for that node will be at least the value of the optimal solution to the subproblem without the already allocated items, and with the reduced capacities. If we then see that the sum of the value gained from the already allocated items and the upper bound is less than or equal to the lower bound, the entire subtree under that node can be excluded, or *pruned*, since it cannot possibly surpass the value of our best solution so far. The efficacy of the branch-and-bound strategy is highly reliant upon the tightness of the upper bound (how close it is to the actual optimal solution), since a tighter upper bound will more often be able to eliminate subtrees from consideration.

### 2.6.1 Item-based Branch-and-Bound

The perhaps most natural approach to solving the multiple knapsack problem relies upon an item-based approach. In this strategy, we start out with a node representing an item, and construct the tree by making branches from that node, which each represent a possible bin that the item could be put in. In other words, at depth $d$ in our tree we determine the placement of the $d$-th item. The number of branches springing from this node is 1 + the number of feasible options we had for placing the $d$-th item. The extra branch represents not allocating the item to any knapsack, which me are forced to do if all the items can't fit in the knapsacks. In figure 2.1 going left means allocating the

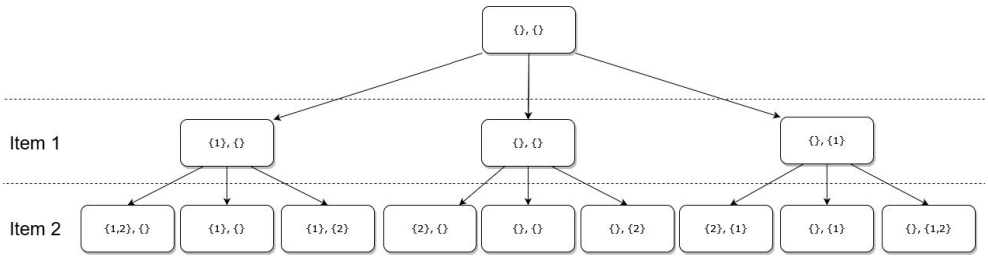



Figure 2.1: Example of item-based B&B with 2 knapsacks and 2 items

item to knapsack 1, middle means not allocated and right means allocated to knapsack 2. Feasible bins are those whose capacity would not be exceeded by receiving the item.

### 2.6.2 Bin-based Branch-and-Bound

An alternate approach to item-based branch-and-bound, presented by Fukunaga and Korf [10], is bin-based branch-and-bound. At depth $d$, instead of determining the placement of item $d$, we instead determine the complete filling of bin $d$. The set of all items $j$ in a bin is called a *bin assignment*, $A_i = \{j_1, \ldots, j_k\}$. This allows us to represent the solution to a multiple bin-packing problem as $S = \{A_1, \ldots, A_m\}$. In this model, the branches from a node represent the possible feasible assignments for the current bin.

A rationale for why this is a more efficient approach can be explained in terms of the depth and branching factors of the resulting trees. In the item-based model, the depth of the tree is the number of items, while the branching factor at each depth is the number of feasible bins. In contrast, the bin-based

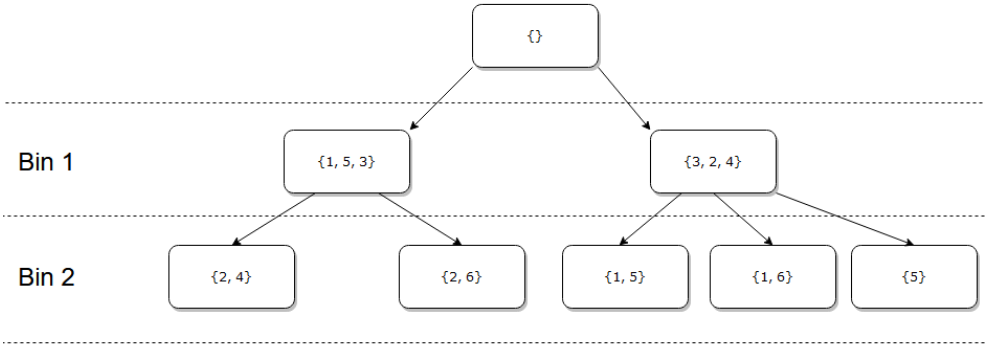



Figure 2.2: Example of bin-based B&B

model makes the depth of the tree be the number of bins, while the branching factor at each depth is proportional to the amount of remaining items. Since the number of leaves in a full tree is determined by $b^d$, where $b$ is the branching factor and $d$ is the depth, having a higher depth is worse for performance than having a higher branching factor.

## 2.7 Fairness Notions

To form a basis for discussing MOKPs relationship with fair allocation, we will briefly cover some relevant theory.

We will assume a set $M$ of $m$ agents and a set $N$ of $n$ indivisible goods, where each agent $i \in M$ has an additive *valuation function* $v_i(j)$ that returns the non-negative value agent $i$ has for one or more goods $j \in N$. An additive function is a function such that $f(S) = \sum_{s \in S} f(s)$. Whenever we refer to an agent's or bin's valuations, we mean the vector $[v_i(1), v_i(2), \ \dots \ , v_i(n)]$. Likewise, an item's valuations are represented by the vector $[v_1(j), v_2(j), \ \dots \ , v_m(j)]$.

An allocation $A = (A_1, \dots, A_m)$ is a collection of subsets of $N$ such that $A_i \cap A_k = \varnothing$ for every pair of assignments $i$ and $k$. In the case where $\bigcup_{A_i \in A} A_i \neq N$ the allocation is called *partial*, if not it is called *complete*.

### 2.7.1 Envy-freeness

An allocation is considered *envy-free* (EF) if for every pair of agents $i, k \in M$, $v_i(A_i) \geq v_i(A_k)$. That is, no agent prefers another's assignment over their own.

An allocation is considered *envy-free up to any good* (EFX) if for every pair of agents $i, k \in M$, $v_i(A_i) \geq v_i(A_k \setminus g)$, for all $g \in A_k$. That is, removing any item from $A_k$ would eliminate agent $i$'s envy towards agent $k$.

An allocation is considered *envy-free up to one good* (EF1) if for every pair of agents $i, k \in M$, $v_i(A_i) \geq v_i(A_k \setminus g)$, where $g = \max_{j \in A_k}(v_i(j))$. That is, no agent prefers another's bundle without the most valuable item in that bundle from the envious agent's perspective over their own.

EFX is a relaxation of the envy-freeness criterion, and EF1 is a relaxation of EFX. While EF is an attractive notion of fairness, cases in which it is achievable are relatively rare. For indivisible items, which we assume in this thesis, an EF allocation is not guaranteed to exist. The simplest example of this is an instance with two agents who positively value a single item which must be allocated between them. The relaxations therefore function as weaker, but still valuable notions of fairness, which are attainable in more circumstances.

Wu et al. redefine envy-freeness and its relaxations slightly to fit the context of budget-constrained settings [25]. The need for this redefinition comes from agents with smaller budgets being able to envy agents with bundles that don't fit in their own budgets under the definition above. Therefore, with a budget $B$ and cost $c(S)$, an agent $j$ is envious of an agent $k$ if and only if there exists a $S_k \subset A_k$ such that $v_j(S_k) > v_j(A_j)$ and $B_j \geq c(S_k)$. The relaxations are changed similarly, such that EF1 is broken if $j$ still values $S_k$ more than $A_j$ if $j$'s most valued item from $S_k$ is removed.

Envy-freeness is usually coupled with a requirement of completeness, to prevent the trivial EF-solution of not allocating any items. With budget constrains, we may not feasibly be able to allocate every item within the budgets. This can be dealt with by introducing some requirement of efficiency, which *Pareto optimality* (PO) is a good candidate for. An allocation of items is *Pareto-optimal* if there is no feasible way of giving one agent more value without making some other agent lose value. If we reintroduce the concept of a charity, an additional agent with unlimited budget whose valuation function is $v_c(j) = 0$ for every $j$

as we mentioned for the GAP, the complete allocation $A = (\emptyset, \emptyset, \ \ldots \ , M)$ is not PO as long as any agent can feasibly attain an item.

### 2.7.2 Proportionality

A *proportional* allocation is an allocation in which every agent $i \in M$ receives their $\frac{1}{m}$ share of their valuation of the total pot. $v_i(A_i) \geq \frac{v_i(N)}{m}$

The *maximin share* (MMS) fairness criterion is relaxation of proportionality. It requires that each agent's bundle be worth at least their MMS. Each agent calculates their MMS by considering every feasible allocation of the items in $N$ to the agents in $M$, then considering the worst bundle (according to their own valuation function $v_i$) in each allocation, and taking the maximum value among the worst bundles. We essentially ask the agents to delegate the items themselves, and force them to only expect the value of the worst bundle in their own allocation.

A compelling feature of MMS as a fairness notion for constrained settings is that it is directly based on *feasible* allocations. Since constraints reduce the set of feasible allocations, they cannot make it easier to guarantee the other fairness notions, and usually make it harder, as evidenced by the findings of Bouveret et al. in 2017 [2] and Hummel and Hetland in 2021 [12]. Unlike the other fairness notions, MMS adapts to the reduced set of feasible allocations, which makes it more achievable in constrained settings. Therefore, one can argue that it is a more realistic expectation to have of a fair allocator.

## 2.8 Integer Linear Programming

*Integer linear programming* problems are linear optimization problems in which the decision variables are required to be integers. The set of feasible solutions may also be restricted by linear constraints. If some of the decision variables are allowed to take continuous values, the problem is called a *mixed-integer linear programming problem.* As an example, a possible ILP is to maximize $z(x, y) = 3y + x$ by finding the optimal value to the decision variables $x$, $y$ under the constraints:

$$y - x \leq 0,$$

$$y + 3x \leq 14,$$

$$x, y \geq 0,$$

$$x, y \in \mathbb{Z}_+$$

This ILP can be represented visually as shown in figure 2.3.

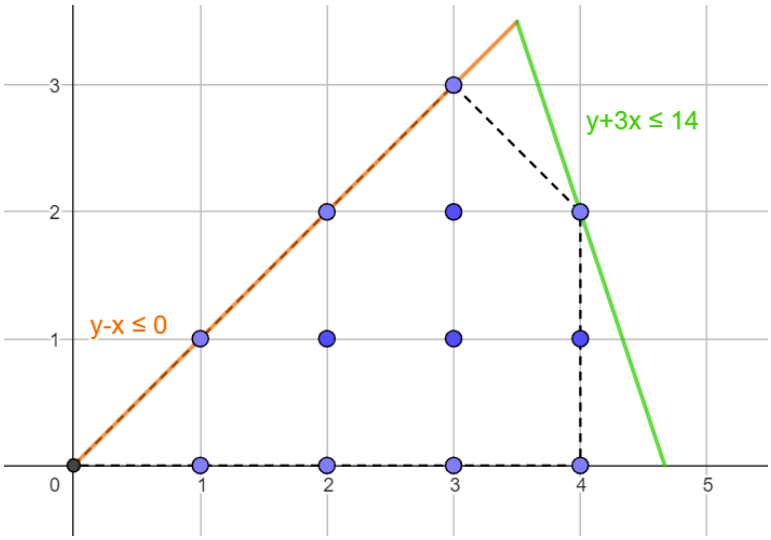



Figure 2.3: Visual example of an ILP

Constraints 1, 2 and 3 combine to form a feasible region which our decision variables can exist within, and is defined by the orange and green lines, as well as the x-axis. Constraint 4 further limits the possibilities to the integer points within that feasible region, marked by blue dots. The dashed black line indicates the *convex hull*, the smallest convex polyhedron containing all the possible solutions within the feasible region. We are left with a set of feasible values for $x$ and $y$, and can for example utilize branch-and-bound to explore combinations of $x$ and $y$ to find the maximum value of $z$.

The optimal solution if we allowed all rational values of $x$ and $y$ would be $z(3.5, 3.5) = 14$, whilst the optimal integer solution is $z(3, 3) = 12$. Applied to the MOKP, $z$ would be a function of every binary variable $x_{ij}$, with a feasible region marked by a polyhedron in $(i \cdot j)$-dimensional space. We expand on how problems like these are solved in section 3.4.

# Chapter 3

# Optimizing Branching

The naive solution to the multiple knapsack problem is to simply explore every possible configuration of items in bins by traversing through a tree. This tree could as discussed earlier either have a depth equal to the number of bins or the number of items, depending on whether we go for a bin-based or an item-based solution. Here we will discuss how to improve upon the naive solution by going through each component that contributes to the slowness of the tree exploration. We can split the work of the algorithm into the following categories: preprocessing, total number of nodes, and work done at each node. Preprocessing happens before any search, and is usually done to either reduce the problem size or to arrange the input in a favorable order for the search. At each node some work will have to be done, e.g., generating feasible assignments. The total amount of work done by the algorithm can be represented by: $preprocessing + n\_nodes \cdot work\_at\_node$. We will consider a number of optimizations to the MKP, and discuss their viability when subjective valuations are introduced.

## 3.1 Preprocessing

There are instances when the problem size can be reduced before we even start exploring the tree. Specifically, there are three trivial cases where we can remove knapsacks or items before starting the search. Considering $m$ knapsacks with capacity $c_i$ and $n$ items with weight $w_j$, these cases are:

$$w_j > \max_{1 \leq i \leq m}(c_i), \tag{3.1.1}$$

$$c_i < \min_{1 \leq j \leq n}(w_j), \tag{3.1.2}$$

$$\sum_{j=1}^{n} w_j \leq \max_{1 \leq i \leq m}(c_i), \tag{3.1.3}$$

If (3.1.1) is true, we can remove item $j$ as it is too large to fit into any of the knapsacks. (3.1.2) allows us to remove any knapsacks which have a capacity

smaller than the smallest item, since they cannot possibly fit any items. Whilst the first two remain viable with subjective valuations, the third one only applies to objective valuations. If true, it allows us to immediately return the trivial solution of packing the largest bin with every item. This naturally doesn't work with subjective valuations as the optimal solution may only be achievable by handing the items to different agents.

All of these tests can be done in linear time, and may potentially reduce the number of items and/or knapsacks in the problem instance, which can have a significant impact on the number of nodes in the tree. However, these cases are unlikely to be present in realistic problems, as the user submitting the problem ought to understand that asking an algorithm to allocate an item that is bigger than all the potential containers is futile. Nonetheless, seeing as the potential gain is so much higher than the initial cost, it seems a worthwhile implementation detail. These apply nicely to budget-constrained fairness scenarios as well, as they are independent of the optimization goal.

Certain optimizations require a specific ordering of the items. Given that the required ordering does not change throughout, one can save considerable time by sorting the items in the preprocessing stage and maintaining that ordering throughout the search, instead of sorting them at each node.

## 3.2 Work Done at Each Node

### 3.2.1 Generating Assignments

At each internal node we need to generate the feasible combinations of items with respect to the current bin. The naive solution to this problem is to generate every possible combination of items and test each of them for feasibility. This will require us to create and loop through $\sum_{r=1}^{n} \frac{n!}{(n-r)!r!}$ combinations of $n$ items.

A well known optimization of this is to only calculate the combinations of length up to an integer $k \leq n$. If we first sort the items by ascending cost, we can sum the lowest cost items until they exceed the capacity of the bin, and call the number of items that fit within the capacity $k$. Since we only included the least costly items, there can exist no feasible combination of items of size greater than $k$. For instances with many more items than can fit in a bin this can severly cut down on the number of combinations we need to test. If we have 20 items

with weights $[1, 2, 3, \ldots, 20]$ and a capacity of 16, we get $k = 5$ and only need to evaluate $\sum_{r=1}^{k=5} \frac{20!}{(20-r)!r!} = 21\,699$ as opposed to $\sum_{r=1}^{n=20} \frac{20!}{(20-r)!r!} = 1\,048\,575$ combinations. Whilst up-to-k will return the same number of feasible assignments, and therefore generates the same number of nodes for us to explore, it was able to avoid testing over a million additional combinations for feasibility. Luckily there are ways to reduce the number of feasible assignments we have to explore as well, which we will consider in section 3.3.

### 3.2.2 Incrementally Generated Assignments

An alternative to generating every assignment at once is to incrementally generate them, by first generating $h$ assignments, and then exhaustively exploring all their subtress before generating the next $h$ assignments. This concept has shown promise in other applications [10, 22], but has not been implemented in our solution.

Since bin-completion is a depth-first-search algorithm, we always completely explore all $k$ nodes before considering node $k + 1$. Incrementally generating the assignments is therefore not with the intent of getting a tighter lower bound for later nodes. The intent is to reduce the amount of time spent generating assignments, which can become very large in situations where many items fit within the knapsack. The hope being that we find an optimal solution early and suspend the search. In order to know that a solution is optimal without exploring the rest of the tree we rely on the *bound-and-bound* technique. We describe this in more depth later, but the concept is essentially: Assume we at node $n$ calculate an upper bound and explore $h$ assignments. If we manage to find an assignment to each knapsack in a subtree under one of the nodes in $h$ whose profit is the same as the upper bound in $n$, we have found an optimal solution to $n$, and can confidently omit exploring the assignments not in $h$.

The trade-off we incur with this strategy is the efficacy of our *value-ordering heuristic*, which is the heuristic we use to choose how to sort the assignments for a bin. Assuming that the order generated by our value-ordering heuristic is actually the optimal way to explore the assignments, we lose some of the benefits of that when only considering $h$ items at a time, since the $h$ most optimal assignments to explore first are very unlikely to be present in the $h$ assignments we generate first.

According to Fukunaga and Korf [10] the optimal value of $h$ varies significantly based on the problem instance, and no clear best value was discovered for different classes of problems, e.g., few knapsacks many items. For their experiments they chose to set $h = 100$.

## 3.3 Number of Explored Nodes

The maximum number of leaf nodes in a tree is $b^d$, where $b$ is the branching factor (how many direct child nodes spawn from a parent node), and $d$ is the depth of the tree. The total amount of nodes in the tree is then $n = \sum_{i=1}^{d} b^i$. For bin-based trees, the branching factor of a node is the amount of feasible combinations of items in the bin at that node. This number varies depending on both the weight of the remaining items and the capacity of the bin. Much of the slowness of tree-based algorithms comes from the exponential relationship between the problem size and the number of nodes to explore. Strategies for reducing the number of explored nodes are therefore integral to speeding up these algorithms.

### 3.3.1 Upper Bounds

As previously mentioned, an essential pruning technique for branching problems relies on comparing an upper bound for the current subproblem to the lower bound. Once we have derived an upper bound at a node, we can add it to our profit from the bins we have already filled, and check if their sum is greater than our best solution so far. If their sum is less or equal, the path we have taken in the tree has no hope of superseding our best solution so far, so we can prune the subtree under the node.

Calculating an upper bound requires a function $u(s)$ such that $u(s) \geq f(s)$ for any problem instance $s$, where $f(s)$ returns the value of the actual optimal solution to $s$. The function $f(s)$ is itself therefore a valid candidate for $u(s)$, but would be no more efficient since we are accurately solving the entire subproblem. For $u(s)$ to be an advantageous inclusion it must therefore also be able to solve the subproblem considerably more efficiently than $f(s)$. A natural candidate for $u(s)$ is the solution to a version of $f(s)$ with relaxed (weaker) constraints. Since the set of feasible solutions to a problem is merely a subset of the set of feasible solutions to a relaxed version of the same problem, the optimal solution to the relaxed version must result in a value at least as large as the optimal

solution for the original problem. Relaxations therefore lend themselves nicely as candidates for $u(s)$. Reconsidering the ILP-formulation of the MKP from section 2.1, there are many ways one could relax the constraints. Two of the most common are the *linear relaxation* and the *surrogate relaxation* [13].

The linear relaxation is based on replacing $x_j \in \{0, 1\}$ with $0 \leq x_j \leq 1$ for $j = 1, \ldots, n$. We call this new problem, where we consider including fractions of the items, the LPMKP (LP-relaxed MKP). The equivalent problem for subjective valuations will be referred to as LPMOKP. In the 0-1 knapsack problem we can solve the LP-relaxed problem by greedily assigning items to the bin in order of decreasing efficiency, and guarantee a valid bound [13]. This is done by sorting the items in order of decreasing efficiencies (value-to-weight ratio), assigning the most efficient to the bin until we get a *break item*, which is the item responsible for hitting or exceeding the capacity. We then fill whatever space is left in the bin with an appropriate portion of the break item's profit. The LPMKP can be solved in essentially the same way, by assigning the most efficient items to bin 1, and splitting up the break item such that the first portion goes in bin 1, while the remaining portion is introduced as its own item. We then fill bin 2 in the same way with the most efficient remaining items, and do this for every bin until they are all full.

Another possible constraint to relax is allowing a merger of the knapsacks into a single aggregate knapsack. We call this the surrogate relaxation, and the resulting problem the surrogate MKP (SMKP, or SMOKP for the MOKP). More formally, we relax constraint (2.1.2) such that it instead becomes:

$$\sum_{i=1}^{m} \mu_i \sum_{j=1}^{n} w_j x_{ij} \leq \sum_{i=1}^{m} \mu_i c_i \tag{3.3.1}$$

by utilizing a set of multipliers for each knapsack $(\mu_1, \ldots, \mu_m)$. The surrogate dual problem for an instance $I$ is then:

$$\text{Minimize}: \quad z(S(I, \mu)), \qquad \mu \geq 0 \tag{3.3.2}$$

In essence, find the set of multipliers $\mu$ such that the value $z$ of the SMKP (S) is minimized. Martello and Toth [17] proved that setting all these multipliers to be the same positive constant always leads to the smallest possible surrogate upper bound for the MKP. The choice of which positive constant does not matter. Moreover, Nemhauser and Wolsey [19] show that the surrogate relaxed

problem with optimal multipliers yields a bound no larger than the LP-relaxed problem for the MKP.

The proof for the optimality of constant positive multipliers for the SMKP relies on arguing that an item's value is independent of which knapsack it is placed in, which is not the case for the MOKP. This result does therefore not extend to the MOKP, but whether the SMOKP with optimal multipliers dominates the LPMOKP is unclear to us.

Upper bound-based pruning is still very much relevant when considering subjective valuations, but the bound's usefulness is directly associated with how close it can get to the optimal value. With the introduction of subjective valuations this becomes harder, since we consider the value of the item to be its maximum valuation when solving the SMOKP in order to ensure the validity of the bound. This is especially costly in instances where a single agent values many items highly, but only has capacity for a few of them. Naturally, the bounds will become tighter the deeper in the tree we are, as we can then omit the valuations of the knapsacks that have already been filled. In instances that lend themselves nicely to normalizing the valuations, this may be a worthwhile endeavor, since it would eliminate potentially skewed valuations generate too optimistic upper bounds.

For budget-constrained fairness scenarios the upper bound would need to be adjusted to fit the new objective function. With maximum NSW for example, where we optimize the product of the profits of each agent, our upper bound would need to be some optimistic estimate of the product. The surrogate relaxation is seemingly ill-suited for this, as the objective function relies on multiplying the profit in each knapsack, which is hard to do when we have combined all the knapsacks. In this scenario the LP-relaxation might therefore be a more suitable choice.

### 3.3.2 Bound-and-Bound

Bound-and-bound is a procedure which calculates a lower bound distinct from the best solution so far at each node in the search tree. It was first introduced by Martello and Toth [17], as part of their MTM algorithm, which featured an item-based search tree. After calculating an upper bound by solving the SMKP, the MTM algorithm attempts to validate the upper bound by finding some feasible solution who's value is the lower bound. If the lower bound has the same

value as the upper bound, the solution which generated the lower bound is an optimal solution to the subproblem under the node, and we can backtrack. In the scenario where the lower bound is less than the upper bound we are forced to explore the subtree under the node.

The MTM algorithm uses a greedy solution which, for an instance with $m$ unfilled knapsacks, involves solving $m$ 0-1 knapsack problems in order to produce the lower bound. It iterates through each knapsack, first optimally filling knapsack $i = 1$, then removing the items assigned to it, and optimally filling $i = 2$ with the remaining items. This repeats until every knapsack has been filled, or we are out of items. We are then left with a feasible solution, with the sum of the profit in each knapsack being our lower bound. These $m$ problems can be solved with the same dynamic programming approach used for deriving the SMKP-based upper bound.

Pisinger [20] proposed an alternative approach to validating the upper bound in his Mulknap algorithm. Instead of solving $m$ 0-1 knapsack problems, he noted that solving a series of subset-sum problems can also be used to validate the upper bound. The relevant version of the subset sum problem is an optimization problem, where one attempts to make a sum from a set of integers that is as close to, but not above, some integer T. In this context the set of integers are the weights of the items included in the upper bound, and the value T is the capacity of the current knapsack. We essentially attempt to fill each knapsack as much as possible with the items. If we are able to assign all the items included in the upper bound solution to the individual knapsacks, we have a feasible solution with a lower bound that matches the upper bound, and we can backtrack as in bound-and-bound.

Fukunaga and Korf [10] and Pisinger [20] report very promising results for the application of the Mulknap bound-and-bound technique in problem instances with large ratios of items to bins. Unfortunately, the Mulknap approach is ill-suited to subjective valuations. Simply showing that every item allocated in the SMKP can be packed into the bins is not enough to give us the optimal solution, as the value of the solution is dependent upon which items go in which bins. The MTM approach lends itself nicely to subjective valuations, however. Since it is based on solving each bin optimally, we can pass in a parameter indicating which valuation to consider for each bin, and receive a solution which respects the valuations of each bin. Additionally, given that the upper bound has been adjusted, similarly adjusting the lower bound allows bound-and-bound

to remain a viable optimization for budget-constrained fairness scenarios.

### 3.3.3 Maximal Assignments

A maximal assignment is an assignment to a bin such that no remaining item not already in the assignment can be added to the assignment without exceeding the capacity of the bin. In the standard MKP setting with objective valuations, one can leverage the fact that the bag in which an item is placed is irrelevant to the value added by that item to remove any non-maximal assignments as possible branching candidates.

When adapting the problem to budget constrained fairness scenarios, it is immediately evident that forcing a bin to be completely filled is unlikely to be conducive to generating fair allocations. However, it turns out that maximal assignments are also not guaranteed to be the optimal solution when individual valuations are introduced. To see this we will consider two agents, each with a capacity of 10, and the items in table 3.1.

| Item | Value for $A_1$ | Value for $A_2$ | Weight |
|---|---|---|---|
| 1 | 6 | 1 | 5 |
| 2 | 1 | 6 | 5 |
| 3 | 2 | 2 | 9 |

Table 3.1: Items with subjective valuations

The only maximal assignments are $\{1,\ 2\}, \{3\}$, which would produce a total value of 9 when handed to the agents. This misses the optimal solution, which is assigning $\{1\} > A_1$ and $\{2\} > A_2$ for a total value of 12. Therefore, we are forced to explore a larger number of assignments than in the MKP setting with identical valuations.

### 3.3.4 Dominance

Another way of reducing the amount of assignments considered is to remove any assignment which we can prove is dominated. Given two feasible candidate assignments A and B, A dominates B if the optimal solution acquired by choosing A has a value no smaller than the value of the optimal solution acquired by choosing B. In the MKP, we can utilize this to eliminate candidate assignments for a bin, thereby considerably reducing the branching factor since we only have to consider a small subset of possible assignments. Fukukaga and Korf base their dominance criterion for the MKP on the work of Martello and Toth [18], and define it as such:

**Proposition 1** (The MKP Dominance Criterion)**.** *Let $A$ and $B$ be feasible assignments with respect to a capacity $c$. $A$ dominates $B$ if $B$ can be partitioned into $i$ subsets $B_1, \ldots, B_i$ such that each subset $B_k$ is mapped one-to-one, but not necessarily onto an element of $A$: $a_k$. In addition, for all $k \leq i$, both the weight and the profit of $a_k$ must be greater than or equal to the sum of the weights and profits in $B_k$ respectively.*

If each element $a_k$ fulfills this criteria, choosing $a_k$ and discarding $B_k$ will lead to at worst an equally good solution, and we can therefore avoid considering $B$. The MKP dominance criteria applies for identical valuations, and is therefore only used on maximal assignments. The criteria requires that the item $a_k$ is heavier than or equally as heavy as the subset $B_k$, which may sound counter-intuitive at first. However, this guarantees that the item we swapped for the subset wouldn't have been better placed in a later bin, as the total value is at worst the same, whilst the later bins have more flexibility in terms of capacity, since the item(s) in the discarded subset weigh less combined, and can potentially be split into different assignments.

For subjective valuations however, although a subset may be dominated by an item with respect to a certain bin, we must consider that the optimal solution may require us to relegate the profit of that bin in favor of another bin with more potential gain. In addition, subjective valuations require us to explore the empty set as a possible assignment to every bin. A trivial example of why is having two items which bin 2 value more than bin 1, such that both items fit in bin 2, and the optimal value is obtained by granting bin 2 both items. Without exploring the empty set, we would be unable to achieve the optimal solution if we filled bin 1 first.

Although this domination criteria doesn't translate directly to subjective valuations, being able to prune more nodes from the branching tree is an attractive property. In order to retain this property we need a new dominance criterion which is able to eliminate assignments that cannot lead to a better solution than another assignment. This criteria must necessarily consider the potential value of all items in every remaining knapsack to confidently exclude the item from an assignment.

**Proposition 2** (The MOKP Dominance Criterion)**.** *Let* $\boldsymbol{B}$ *be a feasible assignment with respect to a bin* $\boldsymbol{b}$ *with capacity* $\boldsymbol{c}$*.* $\boldsymbol{B}$ *is dominated by another assignment* $\boldsymbol{A}$ *if there exists a subset* $\boldsymbol{s} \subseteq \boldsymbol{B}$ *and an excluded item* $\boldsymbol{x} \notin \boldsymbol{B}$ *such that:*
*(1)* $\boldsymbol{B}$ *remains feasible after replacing* $\boldsymbol{s}$ *with* $\boldsymbol{x}$*,*
*(2)* $\boldsymbol{x}$ *weighs at least as much as* $\boldsymbol{s}$*,*
*(3)* $\boldsymbol{x}$ *is worth at least as much as* $\boldsymbol{s}$ *to* $\boldsymbol{b}$*,*
*(4) The total value of scenario 1, where* $\boldsymbol{b}$ *gets* $\boldsymbol{x}$ *and the bin which values an item in* $\boldsymbol{s}$ *the least is assigned that item, is greater than the total value of scenario 2, where* $\boldsymbol{b}$ *gets* $\boldsymbol{s}$ *and the bin which values* $\boldsymbol{x}$ *the most is assigned* $\boldsymbol{x}$*.*

*The assignment B is then dominated by an assignment* $\boldsymbol{A = (B/s) \cup x}$

Requirement (1) is necessary for us to even consider the swap, whilst (2) is derived from the same reasoning as in proposition 1. (3) may seem somewhat out of place, as one may think (4) alone would be enough to ensure we don't throw away an optimal solution. However, it turns out that (3) is also required for us to be sure. Consider two bins $B_1$ with $c_1 = 43$ and $B_2$ with $c_2 = 108$, and the items in table 3.2.

| Item | Value for $B_1$ | Value for $B_2$ | Weight |
|---|---|---|---|
| 1 | 16 | 9 | 9 |
| 2 | 13 | 13 | 11 |
| 3 | 15 | 4 | 18 |
| 4 | 15 | 7 | 25 |
| 5 | 18 | 14 | 20 |
| 6 | 10 | 14 | 29 |
| 7 | 20 | 19 | 57 |

Table 3.2: Items for the MOKP domination criteria example

When considering the assignment {1, 2, 3} for $B_1$, we can see that it is feasible to swap the subset {1, 2} with item 4, and that $w(\{4\}) \geq w(\{1, 2\})$. In scenario 1, where we give $B_1$ item 4 instead of {1, 2}, we calculate a value of:

$$p_1(4) + \min_{2 \leq i \leq m} (p_i(1)) + \min_{2 \leq i \leq m} (p_i(2)) = 15 + 9 + 13 = 37$$

In scenario 2, where we let $B_1$ keep {1, 2} and take the most optimistic remaining value for item 4, we get:

$$p_1(\{1, 2\}) + \max_{2 \leq i \leq m} (p_i(4)) = 29 + 7 = 36$$

According to requirement 4, this would indicate that the assignment {1, 2, 3} is dominated by {4, 3}, which is not the case. Whilst the optimal solution would be better if we had to assign item 4 to $B_1$ or $B_2$, we don't know whether item 4 is included at all in the optimal solution. In fact, the optimal assignment for $B_2$ is {5, 6, 7}, and it is therefore disinterested in item 4 altogether. This means we would have given up value for $B_1$ in order to facilitate a better value for $B_2$ in a case where $B_2$ had better options than the excluded item we were testing against the subset. Therefore, due to the agnosticism toward optimal assignments for later bins, we are also forced to ensure that the excluded item grants the current bin more value than the candidate subset.

Whilst this domination criteria is less powerful than the one for objective valuations, this is a natural consequence of the increased complexity of the problem. An important implementation detail here is to only take the maximum and

minimum of the valuations of the remaining bins, and not across all bins in the problem. We are essentially computing an upper bound for the case where we swap the subset with the item and comparing it to a lower bound for the case where we keep the subset, and the tightness of these bounds are essential to how much of the tree we are able to prune.

Requirement (4) in proposition 2 does not extend to other social welfare functions like NSW. Instead, one would need some requirement which guarantees that the total product is maximized by taking $x$ and leaving $s$ to the remaining bins. Given a sum of profits, say 10, to be divided amongst 2 agents, the assignment which maximizes the product of the profits of the two agents is the one which distributes them profit the most equally, i.e. $5 \cdot 5$. Say we could gain 10 total profit by assigning items such that the agents got 2 and 8 profit respectively. This would not be better than an assignment which gained a total of 9 profit, but distributed them as 4 and 5, since $4 \cdot 5 = 20 > 2 \cdot 8 = 16$. Essentially, depending on the difference in valuations for an item, giving up total profit to boost the profit of the least profitable agent may grant a higher product of profits. Knowing whether the current bin or one of the remaining bins will be the least profitable is hard to do before exploring the options for the remaining bins, so we don't see a direct way to adjust this domination criteria to fit the NSW.

### Generating Undominated Assignments

An advantage of both the domination criteria for objective and subjective valuations is that they are able to assess whether an assignment is dominated without making comparisons to other generated assignments. The only relevant information is the assignment and the remaining items that are not part of the assignment. This means that we can generate the assignments one by one, and only store them if they turn out to be undominated. This is advantageous both in terms of memory and complexity, which is essential for bins which can fit many possible assignments.

We utilized a binary tree to generate the undominated assignments. In this tree two branches spring from the root, representing whether we include item 1 or not. The branch which included item 1 reduces the remaining capacity by the cost of item 1. We then generate two children for each of those, representing whether to include item 2. If item 2 is not feasible to include, we just generate one child. By checking for dominance at both every internal node and leaf node

in this tree, we are left with exclusively all the assignments which are feasible and undominated.

### 3.3.5 Pisinger R2 Reduction

In *An exact algorithm for large multiple knapsack problems* [20] Pisinger describes a reduction procedure for removing items from consideration. This is done by deriving an upper bound for a solution which includes the item, and comparing it to a lower bound representing the current best solution. If the upper bound for the partial solution which includes the item is lower than the lower bound, the inclusion of the item is suboptimal in the rest of that subproblem. This procedure can be performed at each node, and will eliminate items for consideration in the subtree under that node. It works by first sorting the remaining $n$ items such that we have non-increasing profit-to-weight ratios, $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$. We then combine the capacities of the remaining $m$ bins into a bin with $c = \sum_{i=1}^{m} c_i$ , and attempt to fill this combined bin with the most efficient remaining item until we exceed the capacity.

We call the item responsible for (over)filling the bin the *break item* $b$, such that $\sum_{j=1}^{b-1} w_j < c \leq \sum_{j=1}^{b} w_j$. At this point we know the first $b-1$ items are the most efficient use of space in the aggregate bin. In order to derive an upper bound $u(k)$ for the items $(b+1 \leq k \leq n)$, we add $k$ to the partially filled aggregate bin:

$$p_k + \sum_{j=1}^{b-1} p_j \tag{3.3.3}$$

We then find ourselves in one of three possible scenarios:

- Item $k$ fit within the aggregate knapsack, and we have remaining capacity
- Item $k$ filled the aggregate knapsack perfectly, and we have no leftover space
- Item $k$ did not fit in the aggregate knapsack, and it is overfilled

If $k$ filled the knapsack perfectly we are left with a valid and relatively accurate bound. In the case where we have remaining capacity to work with, (3.3.3) is not a valid upper bound, as although the initially included items are the most efficient in terms of profit/weight, there may be an excluded item which fills the knapsack more optimally and generates more value. To correct this without

introducing much computational complexity, we resort to filling the remaining capacity of the knapsack with a fraction of the break item $b$.

Since $b$ is the most efficient excluded item, it is the most optimistic candidate to fill the remaining capacity with, thereby ensuring the validity of the bound. We can therefore add the product of the profit-to-weight ratio and the remaining capacity to fractionally fill the rest:

$$(c - w_k - \sum_{i=1}^{b-1} w_i) \cdot \frac{p_b}{w_b} \tag{3.3.4}$$

In fact, this deals with the last scenario as well, since the remaining capacity will be negative, and we subtract the fractional value of the break item with respect to how much the bin was overfilled. Combining these two, and flooring the value of the fractional part, since we only deal with integer values, gives us a complete upper bound for item $k$:

$$u(k) = p_k + \sum_{j=1}^{b-1} p_j + \left\lfloor (c - w_k - \sum_{i=1}^{b-1} w_i) \cdot \frac{p_b}{w_b} \right\rfloor \tag{3.3.5}$$

The rationale for extending this notion is the exact same as for normal upper bounds. Using the maximum valuation of the item among the remaining bins and calculating the upper bound based on that value yields a valid, although somewhat exaggerated upper bound. Note that there exists a trade-off here. In order to calculate as accurate a bound as possible, we ought only take the maximum of the relevant valuations. What the already filled knapsacks think of the item is not relevant when the knapsacks can't fit the item anyway. However, sorting the items at each node, such that they are sorted with respect to the value of the remaining knapsacks, is computationally expensive and may outweigh the benefit of the improved bound.

It turns out that sorting the items at each node is actually not necessary. Simply initially sorting the items once for each combination of remaining knapsacks and querying the result at each node is enough. Since we can decide which knapsacks will be filled in which order beforehand, we can cut down the number of ways we need to sort the items significantly. We will expand on this optimization and the trade-off in the results- and discussion-sections.

Since the R2-reduction is based on combining a surrogate and linear relaxation to derive an upper bound, its application to the maximum NSW solution suffers from the same issue we described for surrogate upper bounds. It is therefore not viable in a non-linear context.

### 3.3.6 Ordering of Bins and Items

The order in which we choose to fill the bins and which assignments we choose to explore first can have large consequences for how many nodes have to explore. Martello and Toth's item-based MTM algortihm [17] considers items in order of non-decreasing efficiency $e_j = \frac{p_j}{w_j}$ and which knapsack to then assign the item to in order of non-decreasing remaining capacity. Fukunaga's bin-completion algorithm [11] fills the bins in order of non-decreasing capacity, such that the smallest bins are filled first, but no explanation is provided for this choice. The article's precursor [10] does however discuss the importance of the ordering of the assignments we consider for each bin, and finds that among 11 different heuristics for sorting assignments, one ought to sort them according to increasing cardinality (how many items the assignment contains), breaking ties by non-increasing profit.

Considering we found little convincing rationale for considering the knapsacks in the order that was presented, and could therefore not reason about whether that order would remain optimal (if it even is for bin-completion) when considering subjective valuations, we decided to experiment with some other orderings.

An interesting candidate was considering the knapsacks in order of decreasing maximum number of top valuations, such that the knapsack which has the top valuation for the most items is considered first. As previously mentioned, a large issue with solving MOKPs as opposed to MKPs is the inflation of surrogate upper bounds due to considering the max valuation of each item. By removing the knapsacks that contribute the most to the inflation of these upper bounds first, we ought to be able to calculate more precise upper bounds deeper in the tree, thereby increasing the utility of the upper bound pruning and bound-and-bound.

To illustrate the potential impact of the ordering, we will take a sneak peek as some experimental results comparing smallest capacity first, largest capacity first and number of maximum valuations first in figure 3.1.
Whilst the orderings track each other decently for many instances, certain in-

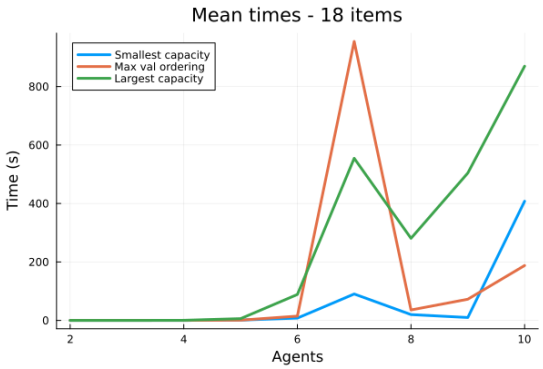



Figure 3.1: Comparison of bin ordering

stances seem to be extremely sensitive to the order in which we consider the bins, resulting in large differences in runtime.

The order in which one considers assignments can have a similarly large impact. If we are able to accurately predict which assignments will generate the most optimal solutions, and explore these first, large parts of the tree may be pruned. Other branching algorithms such as *minimax*, which amongst other things is used to find the best moves in chess, have very strong ordering heuristics for their branches. Since it is completely infeasible to explore every possible game of chess, minimax only explores some $d$ moves ahead. Each leaf node is then given some value based on a heuristic. The optimal move is found by comparing and propagating the values of these leaf nodes up to the root. If we wished to explore $d$+1 moves deep, the previous tree would be seemingly useless, since the entire tree is based on the value of the leaf nodes. However, due to the pruning criteria *alpha-beta pruning* being much more effective when the best moves are explored first, the ranking of moves from the previous tree can be used as a value-ordering heuristic in the new tree. This ordering is so strong in fact, that first exploring a tree of depth $d$, then using its result as a value-ordering heuristic and exploring a new tree of depth $d$+1, is generally more efficient than exploring just the $d$+1 tree [21].

The optimal ordering of assignments in the MKP and MOKP is not as obvious as they are for minimax, but the potential impact they can have is hard to understate.

### 3.3.7 Path-symmetry

Fukunaga introduced the concept of *path-symmetry* and a *nogood* in a follow-up article [11] to the original bin-completion paper [10]. Considering a node $N$ and its siblings $S_1 \dots S_b$, we say that every sibling $S_i$ that has been expanded before N is a nogood with respect to each child, direct or indirect, of N. In Figure 3.2, $S_1$ is a nogood with respect to $C_1$ and $C_2$, since $S_1$ is a sibling of $C_1$ and $C_2$'s parent N, and has already been expanded. Since bin-completion is a depth first search (DFS)-algorithm, every child of $S_1$ has been exhaustively searched.

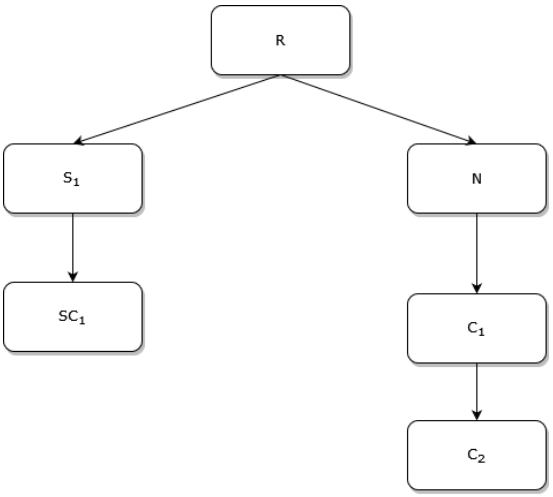



Figure 3.2: Example of a nogood

We define a *path* from some depth $d_1$ to $d_m$ as the union of the bins at each depth between $d_1$ and $d_m$. For instance, the path from N ($d_1$) to $C_2$ ($d_m$) is {N, $C_1$, $C_2$}. The *path items* are the union of the items in each of these bins.
We say that there is path symmetry with respect to a nogood $S^{d_1}$ for some possible bin assignment $C^{d_m}$ and the path items $P$ from $d_1$ to $d_m$ given that:

1. Every item in $S^{d_1}$ is in $P$.
2. We can:

a) Feasibly assign the items that are in $S^{d_1}$ to $N$

b) Assign the remaining items in $(P \setminus N_{items})$ to the rest of the bins down to $d_m$ with all the assignments being feasible.

If a path symmetry exists we can prune the node $C^{d_m}$, since it will achieve the same total value as the already expanded nogood. Requirement b) essentially involves solving an instance of the bin-packing problem, another NP-complete problem, but which can be approximately solved with a heuristic like first-fit-decreasing (FFD).

With subjective valuations one can no longer argue that the assignments are symmetric given the above requirements, since the placement of an item impacts the value added by that item. In fact, the only times symmetries like this would appear under subjective valuations is if two or more agents have identical valuations of a set of items. This can of course be the case, but path-symmetry would be unable to prune nodes at low depths unless many agents shared the same valuations for the many remaining items. With the utility of this optimization being restricted to such niche cases, we have chosen not to implement it in our algorithm.

## 3.4 ILP-solvers

We will later compare our algorithm to an ILP-solver, of which there are several. The purpose of this section is to briefly introduce the basic concepts behind how they work, in order to have an understanding of what differences in runtime may stem from. To be clear, this is not how our own algorithm in implemented.

ILP-solvers generally rely on some sort of branch-and-bound to find the optimal solution. They work by dropping the *integrality* constraint (which requires that the decision variables be integers), and solving the associated linear program to obtain an upper bound. The fundamental theorem of linear programming states that the maximum and minimum value of a feasible linear function which is bounded below is found at the corners of the feasible polyhedron defined by its constraints[24].

Recalling the visual example from section 2.8, this means that regardless of what function $z(x, y)$ is, so long as it is linear, the optimal solution will be in one of the three corners. A candidate for finding the optimal solution amongst the corners is the simplex algorithm. Describing the procedure in detail is beyond the scope of this thesis, but the important point is that the there exist polynomial-time simplex algorithms [14]. This is important, because the number of corners grows very large when we have many variables, as in the MOKP.

Once we have found the optimal solution to the relaxed non-integer linear program in one of the corners, we have a valid upper bound for the ILP. If it so happens that every decision variable in the optimal solution is an integer, we have found the optimal solution to the original ILP. As with the previously described upper bounds, we would like for these upper bounds to be as close to the actual optimal solution as possible. In the scenario where the optimal solution to the non-integer linear program was not an integer, we can introduce *cuts*. These cuts are additional constraints which we add to the linear program in hopes of reducing the feasible region. They are called cuts since the new constraint can be thought of as introducing a plane to our polyhedron, creating a smaller one.

One method for generating cuts is the knapsack cover. Considering the constraint:

$$3x_{2,1} + x_{2,2} + 4x_{2,3} \leq 7$$

Where we have items whose costs are the coefficients of each $x_{i,j}$ (3, 1 and 4), and knapsack 2, which has a capacity of 7. Clearly, every item can not fit in the knapsack, since $3 + 1 + 4 = 8 > 7$. This can be represented by adding another constraint to the problem:

$$x_{2,1} + x_{2,2} + x_{2,3} \leq 2$$

Since at most two of the items can be included in knapsack 2. We then draw a plane representing the new constraint in our polyhedron, and thereby shrink it. By iteratively adding such cuts, we can exclude more and more of the feasible region, thereby finding optimal values to the non-integer LP which are closer to the optimal value of the ILP. ILP-solvers utilize many different methods of generating such cuts, as can be seen in figure 3.3. The eventual goal being to get the feasible region of the LP as close to the convex hull as possible, since the optimal solution then is guaranteed to be an integer solution by way of the fundamental theorem of linear programming.

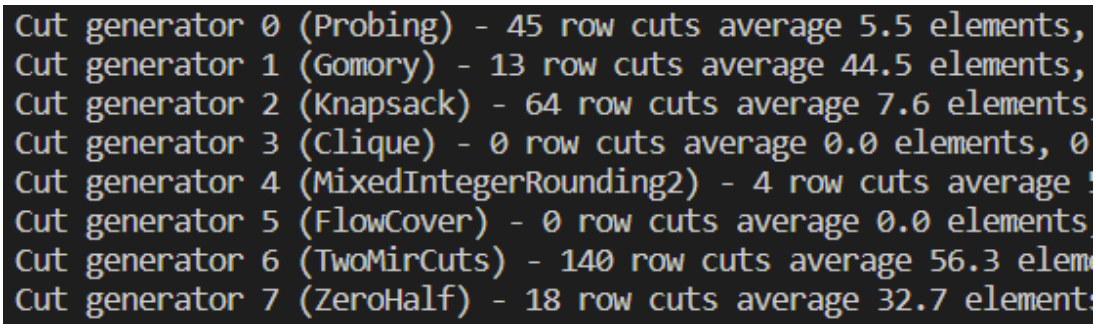



Figure 3.3: Cut generators in the ILP-solver Cbc

# Chapter 4

# Implementation

## 4.1 The Algorithm

The algorithm is implemented in Julia, and is available in its entirety at: https://github.com/jorgstei/FairMulKnap. We describe it with pseudocode in algorithm 1.

Lines 1-4 check whether we are at a leaf node in the tree, and whether the solution we acquired is more optimal than the our current best.
Lines 6-8 are the Pisinger R2 reduction, described in 3.3.5. It returns the items which it could remove from the instance, and if there were any we recurse with the reduced set of items.
In lines 10-12 we calculate the surrogate-relaxed upper bound, which combines the capacities of the bins into a single large bin, and solves a 0-1 knapsack problem where the items have the value of the highest valuation among the remaining bins.

The bound-and-bound procedure described in 3.3.2 happens in lines 13-17. By first filling bin 1 optimally, then bin 2 with the remaining items and etc, we get a set of feasible assignments to the bins. If it so happens that these assignments have the same total value as our upper bound, the assignments are the optimal solution, and we can avoid searching deeper.
Line 19 is responsible for picking the next bin to generate assignments for, which is the remaining bin with the least capacity.
Line 20 generates the undominated assignments (section 3.3.4) for the bin, and sorts them according to a value-ordering heuristic.

We loop through each assignment in 22-26, and solve a subproblem for each without the assignment and the bin. The assignment which generated the best total profit is then returned as the optimal choice for the bin. This propogates up until we reach the root of the tree, and have the optimal solution to the entire problem.

**Algorithm 1** MOKP-completion

```
PREPROCESSING(bins, items)
1  global best_profit = 0
2  remove_infeasible_bins_and_items(bins, items)
3  sort_items_by_max_efficiency(items) /* If using R2 nosort */
4  search_MOKP(bins, items, 0)

SEARCH_MOKP(bins, items, sum_profit)
 1
 2  if bins == ∅ or items == ∅
 3       /* We are at a root node */
 4       if sum_profit > best_profit
 5            best_profit = sum_profit
 6       return (sum_profit, bins)
 7
 8  /* Run R2-reduction to get items which we can omit */
 9  ri = r2_reduce(bins, items)
10  if ri ≠ ∅
11       return SEARCH_MOKP(bins, items \ ri, sum_profit)
12
13  upper_bound = SMKP_upper_bound(bins, items)
14  if sum_profit + upper_bound ≤ best_profit
15       /* Upper bound pruning */
16       return (−1, {})
17
18  /* MTM-version of bound-and-bound*/
19  lower_bound = greedily_fill_bins(bins, items)
20  if lower_bound.value == upper_bound
21       if sum_profit + lower_bound.value > best_profit
22            best_profit = sum_profit + lower_bound.value
23       return (sum_profit + lower_bound.value, lower_bound.assignments)
24
25  /* Get bin with least capacity */
26  B = SELECT_BIN(bins)
27  /* Sort depends on value-ordering heuristic */
28  assignments = SORT(generate_undominated_assignments(B, bins, items))
29  best_assignment = {}
30  for A ∈ assignments
31       subproblem = SEARCH_MOKP(bins \ B, items \ A, sum_profit + B.value(A))
32       if subproblem.best_value > best_assignment.best_value
33            best_assignment.assignment = A
34            best_assignment.best_value = subproblem.best_value
35  return best_assignment
```

Note that since none of the optimizations rely on symmetry between branches, the algorithm is well-suited for multi-threading at essentially no cost.

## 4.2 JuMP algorithm

*JuMP* [8, 16] is a Julia framework for modeling mathematical optimization problems, and solving them through external solvers. Cbc [4] is one such open-source solver, implemented in C++, which can be used to solve mixed-integer linear problems. The problems are formulated by defining the decision variables, the optimization objective, and the constraints as shown in our implementation in algorithm 2.

**Algorithm 2** JuMP-program for MOKP in Julia

m = Model(Cbc.Optimizer) ▷ Init model

@variable(m, $x[1:n_{agents}, 1:n_{items}]$, binary = true) ▷ Init binary variable
@objective(m, Max, dot(values, x)) ▷ Define objective

**for** `i in 1:`$n_{agents}$ **do** ▷ Ensure no bin exceeds its capacity
    @constraint(m, dot($weights[i,:], x[i,:]$) $\leq$ capacities[i])
**end for**

**for** `i in 1:`$n_{agents}$ **do** ▷ Ensure no item is picked twice
    @constraint(m, sum(x[:, i]) $\leq$ 1)
**end for**
optimize!(m) ▷ Solve

# Chapter 5

# Experiments

After having implemented working versions of both our MOKP-solver and the JuMP-program we wrote code to pseudo-randomly generate problem instances. These functions take as input the number of knapsacks/items, in addition to a range for values, weights and capacities. The experiments conducted and presented in this article vary in number of agents and knapsacks, but have constant ranges for values, weights and capacities.
These are:

1. Values: 1–30
2. Weights: 5–50
3. Capacity: 30–120

The rationale for these values was simply that they seemed somewhat reasonable, and we wished to keep them constant to ensure some consistency between the results. Note that it has been shown for the MKP that the difficulty of solving instances scales with the number of significant digits in these values [20].

We ran experiments on instances with differing amounts of knapsacks and items to get a wholistic understanding of the performance of the optimizations, but the values chosen for the number of knapsacks and items were handpicked, due to how rapidly the solving times scaled. Our understanding of performance therefore generally came from running several experiments on different problem types, e.g. different ratios of items to knapsacks, and then analyzing the results.

The implementations of the optimizations were made flexible in order to facilitate easy testing, so the optimizations were passed as arguments to the search_MOKP-function. A test consisted of a list of different versions of the algorithm, in the form of a struct with the optimization options.
Each version of the algorithm was asked to solve the same instance for each combination of number of items and knapsacks. The graphs presented in the results section plot time against the number of knapsacks for a fixed number

**Algorithm 3** MOKP_Options

```
struct MOKP_Options()
 1    individual_vals::Bool
 2    preprocess::Bool
 3    reductions::Vector
 4    compute_upper_bound::Function
 5    validate_upper_bound::Bool
 6    choose_bin::Function
 7    value_ordering_heuristic::Function
 8    reverse_value_ordering_heuristic::Bool
 9    generate_assignments::Function
10    is_undominated::Function
```

of items. The items were also regenerated whenever the number of knapsacks changed, such that although the number of items is the same in a graph, the actual items differ in each datapoint.

To ensure accurate measurements of performance, even on small instances, we used the "BenchmarkTools.jl" [6] package to solve the instances several times when necessary, and return the mean time spent. This library does not return the result from the function itself, leading to us both benchmarking the function and calling the function directly for each instance in order to ensure parity in the results for the different optimization we were testing. This way we could be relatively sure the implementation of the optimizations were not wrong.

The experiments were run on an 11th Gen Intel i7-1165G7 @ 2.8GHz, through Windows Subsystem for Linux version 2 with Ubuntu 22.04.2. We used version 1.8.1 of Julia, and ran our algorithm on a single thread.

# Chapter 6

# Results

To assess the efficacy of each optimization we will present some experimental results for each, mentioning briefly in which types of instances the performance gain seems to be the greatest. Unless otherwise stated, the different versions of the algorithm preprocess the items and knapsacks, use SMKP upper bounds, bound-and-bound, r2 reduction and undominated assignments. The default value-ordering heuristic is min-cardinality, while the default choice for which bin to fill is smallest capacity.

## 6.1 Assignments

### 6.1.1 The Loss of Maximal Assignments

To see the impact of having to consider non-maximal assignments, we solved some MKP instances and compared the nodes explored with and without non-maximal assignments.

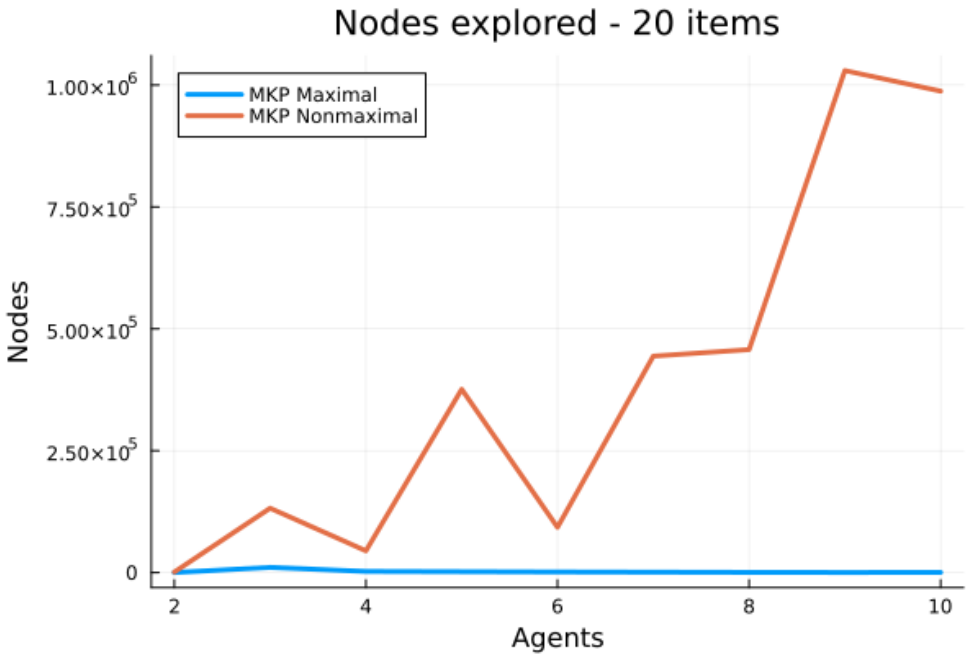



Figure 6.1: MKP instances, 20 items

As expected, the difference in the number of possible assignments to each bin compounds to a massive difference in the number of nodes we are forced to explore. Since the difference is so large, the graph is somewhat hard to read. We have therefore included table 6.1, which shows the number of nodes explored in each instance in the graph above.

| Knapsacks | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| Maximal | 65 | 1e4 | 2404 | 1838 | 1261 | 847 | 431 | 253 | 386 |
| All | 1120 | 1e5 | 4e4 | 3e5 | 9e4 | 4e5 | 4e5 | 1e6 | 1e6 |

Table 6.1: Maximal vs non-maximal, number of nodes

### 6.1.2 Up-to-k vs All combinations

Whilst the up-to-k-optimization doesn't allow us to explore fewer nodes, we assumed the time saved at each node would produce a noticeable impact.

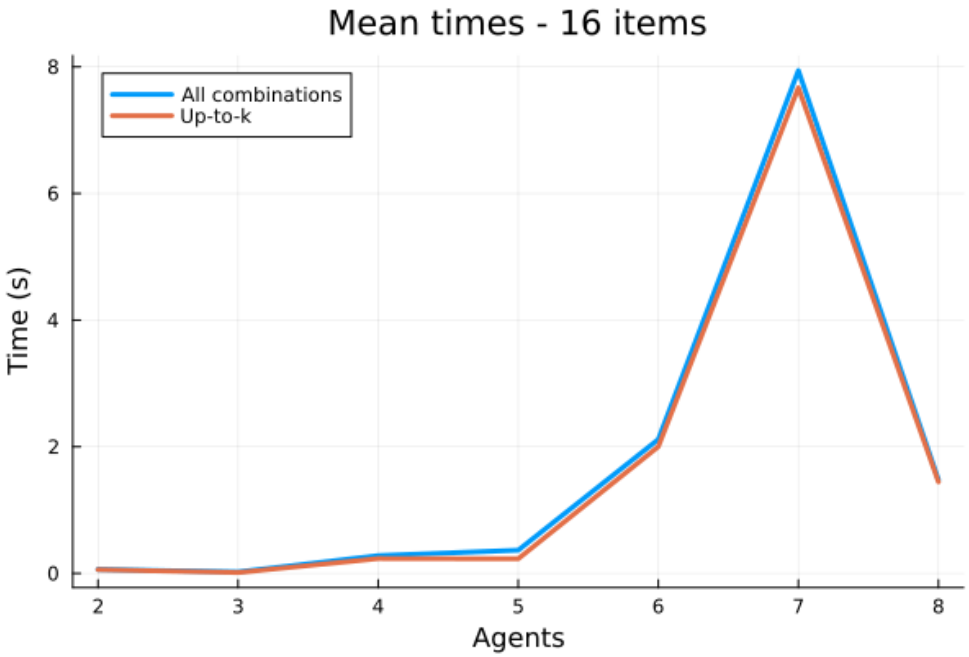



Figure 6.2: All combinations vs up-to-k, 16 items

For some instances however, the difference was barely noticeable. Although the time saved by considering fewer combinations seems to consistently outweigh the extra cost of sorting the items, it barely seems worthwhile for smaller instances.

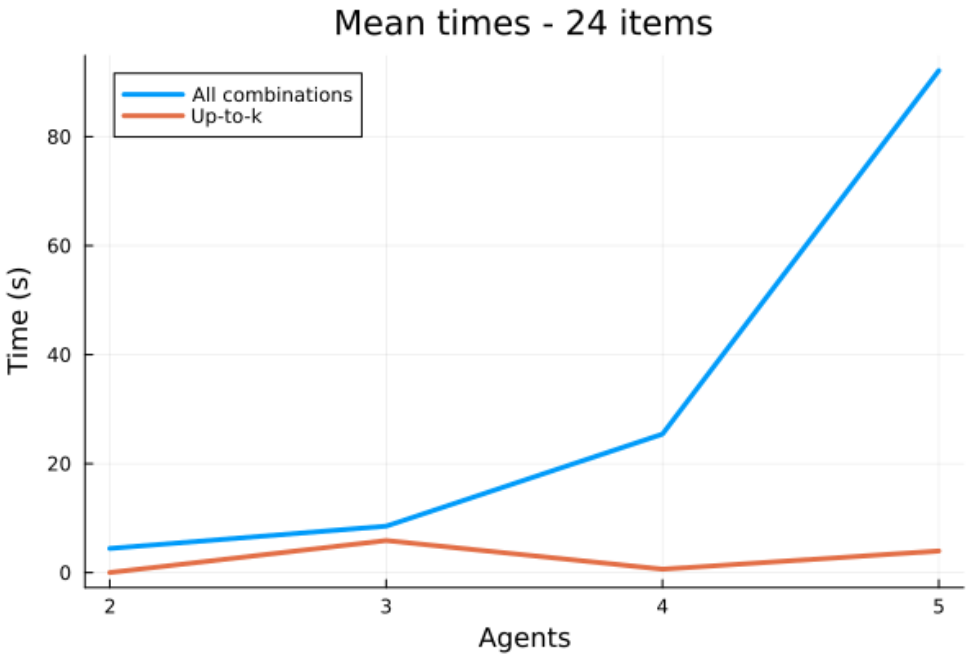



Figure 6.3: All combinations vs up-to-k, 24 items

The larger instances showed very different results. The efficacy of the up-to-k approach scales significantly with the number of items in the instance, again consistently outperforming checking all combinations, and even taking as little as 1/20th of the time in certain instances.

Especially for instances with many items and where the sizes of the assignments are small, e.g. due to large items or small bins, a massive number of combinations must be considered at each level in the tree, exacerbating the differences in runtime.

### 6.1.3 Up-to-k vs Undominated

We performed the same tests comparing up-to-k to our method for generating undominated assignments with a binary tree. In addition, we tested the undominated approach with no domination criteria ("all undominated" in the following graphs), in order to see how much of the performance difference can be attributed to the alternate method for generating assignments, and how effective the domination criteria is.

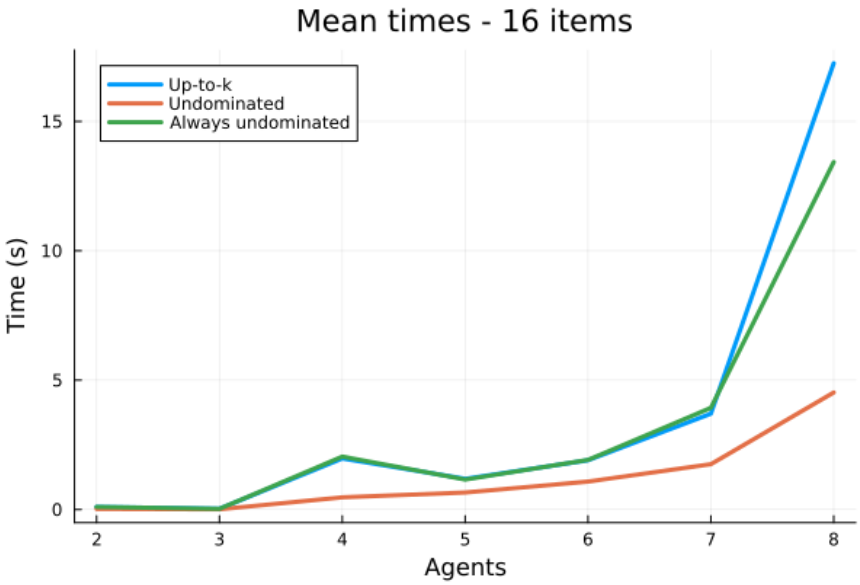



Figure 6.4: Up-to-k vs Undominated, 16 items

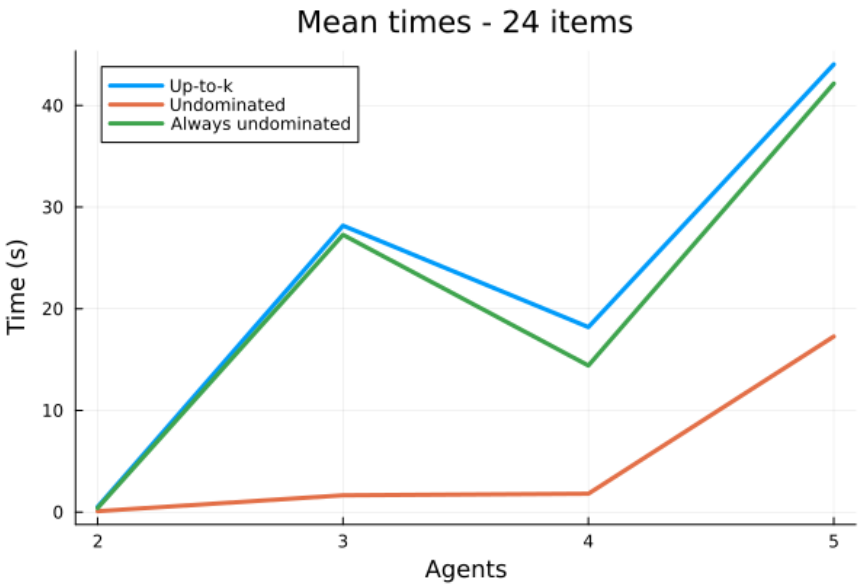



Figure 6.5: Up-to-k vs Undominated, 24 items

Clearly the main improvement is the implementation of the domination criteria.

Although the binary tree approach saves some time by itself, it is not nearly as influential as the reduced amount of nodes we have to explore with help from the domination criteria.

## 6.2 Upper Bounds

In order to be able to say something about the accuracy of the upper bounds in the MKP in comparison to the MOKP, we ran some tests comparing the surrogate upper bound computed at the root node to the optimal value of the problem. We generated 100 instances for each combination of number of items and knapsacks, and averaged the ratio between the root upper bound and the optimal value. Both models solved the same instance each time, with the MKP-version considering the valuations of the first bin to be the objective valuations for each item. In figure 6.6 the ratio between the root upper bound and the optimal solution is presented as the number of 1/100's over 1. So 5 means 1.005, while 65 means 1.065 etc. MKP is on the left, while MOKP is on the right.

| Items \ Bins | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| **10** | 5 / 65 | 0.7 / 48 | 0.5 / 34 | 0.5 / 30 | 0 / 19 |
| **14** | 6 / 90 | 4 / 82 | 0.7 / 58 | 0 / 33 | 0.1 / 34 |
| **18** | 5 / 99 | 3 / 84 | 3 / 80 | 2 / 60 | 0.2 / 47 |
| **22** | 3 / 80 | 2 / 78 | 2 / 77 | 1 / 83 | 0.6 / 61 |
| **26** | 1 / 68 | 2 / 72 | 2 / 74 | 1 / 74 | 1 / 73 |

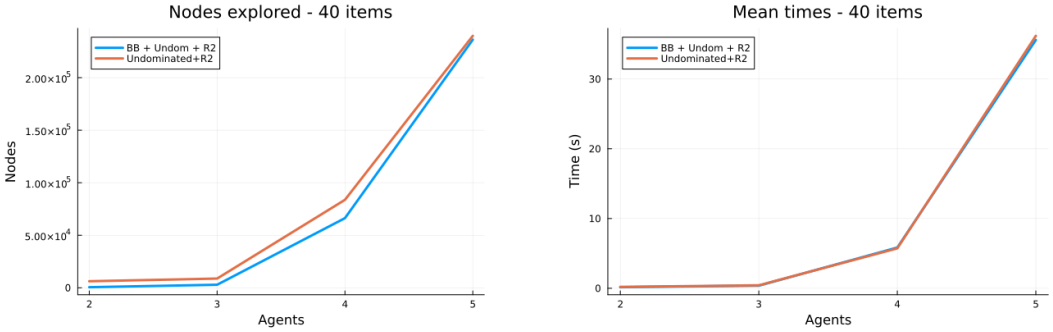
Figure 6.6: Objective valuations / Subjective valuations

The MKP upper bound was consistently an order of magnitude more accurate than the MOKP upper bound. The worst average accuracy for the MKP (14 items, 6 bins) was an overestimation of 0.6%. Meanwhile, the MOKP upper bound overestimated the same combination by 9%.

On average across all the tests, the ratio between the root upper bound and the actual optimal solution was 40 times higher for the SMOKP than it was for the SMKP.

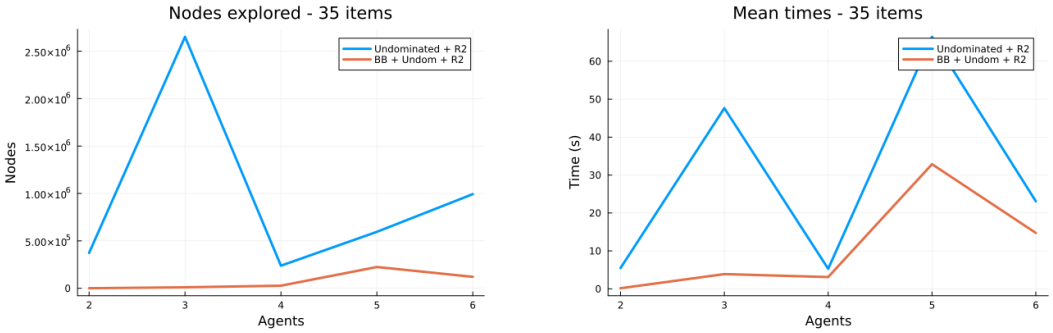
## 6.3 Bound-and-Bound

Previous articles [10, 11, 20] report great efficacy of bound-and-bound, specifically the Mulknap subset-sum approach, in cases with large ratios of items to knapsacks $n/m$. Although we are utilizing the greedy approach of MTM instead of subset-sum in order to conform with subjective valuations, we were interested in seeing how much bound-and-bound would contribute considering our less tight upper bounds.




(a) Nodes (b) Mean times

Figure 6.7: Bound-and-bound 2-5 knapsacks, 40 items




(a) Nodes (b) Mean times

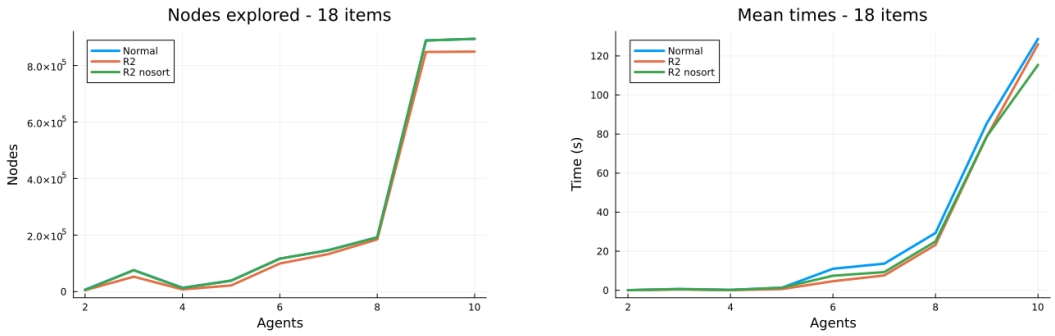
Figure 6.8: Bound-and-bound 2-6 knapsacks, 35 items

In certain instances with large number of item-to-knapsack ratios we see little improvement over omitting bound-and-bound altogether. Considering the looseness of the upper bounds we derive, this is not surprising, as the upper

bounds are often quite a bit higher than the best possible solution. It does however seem like bound-and-bound has something going for it, even with the looser bounds. Bound-and-bound only serves to slow down the algorithm if the upper bounds are infeasible to attain, but the upper bounds we derive seem to be tight enough in certain subproblems in the tree to still enable bound-and-bound to prune a respectable amount of nodes.

## 6.4 R2 reduction

As mentioned in the chapter on the R2 reduction procedure, an interesting trade-off arises when adapting it to subjective valuations. We can derive tighter upper bounds by only considering the valuations of the knapsacks which have not yet been filled, but this requires us to sort the items again each time we fill a knapsack. We ran these experiments on 3 different versions, all with undominated assignments. "Normal" has no implementation of R2 reduction, "R2" is the R2 procedure where we sort the items based on the set of relevant valuations at each node, while "R2 nosort" sorts the items once according to efficiency based on the highest valuation for each item at the root, and keeps that ordering throughout the rest of the search.

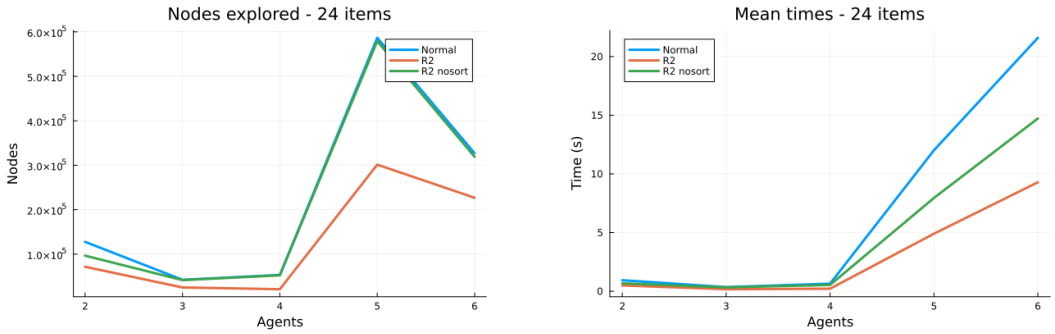



(a) Nodes (b) Mean times

Figure 6.9: R2 comparison 2-10 knapsacks

As we can see, the R2 nosort prunes a marginal amount of nodes compared to R2 (R2 nosort overlaps with normal in the node-graph), but makes up for it in instances with small ratios of items-to-knapsacks due to the reduced overhead at each node. However, for larger ratios of items-to-knapsacks, R2 seems to begin outperforming R2 nosort, due to exploring as few as half the amount of

nodes as R2 nosort.




(a) Nodes

(b) Mean times

Figure 6.10: R2 comparison 2-6 knapsacks, 24 items

## 6.5 Value-ordering heuristic

We tested 6 different value-ordering heuristics to deduce which order we ought to explore the assignments in. These were based on 3 different heuristics: cardinality, weight and profit. Both a version which maximized the heuristic, and a version which minimized it, was included in the experiments.

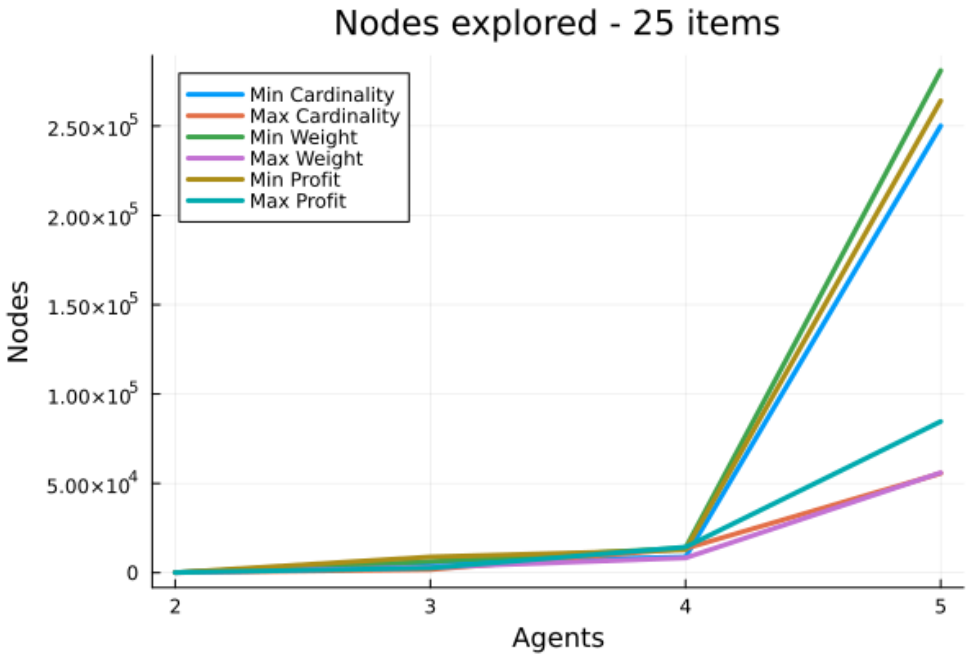



Figure 6.11: Value ordering heuristics, 25 items

We found that among these, exploring the heaviest assignments first was the most consistently well-performing heuristic. Considering Fukunaga and Korf found min-cardinality to be the best value-ordering heuristic, we were surprised to see max-cardinality to perform as well as it did in many scenarios. In many of our tests max-cardinality actually performed better than min-cardinality, but in some instances it turned out to be by far the worst heuristic, such as in figure 6.12.

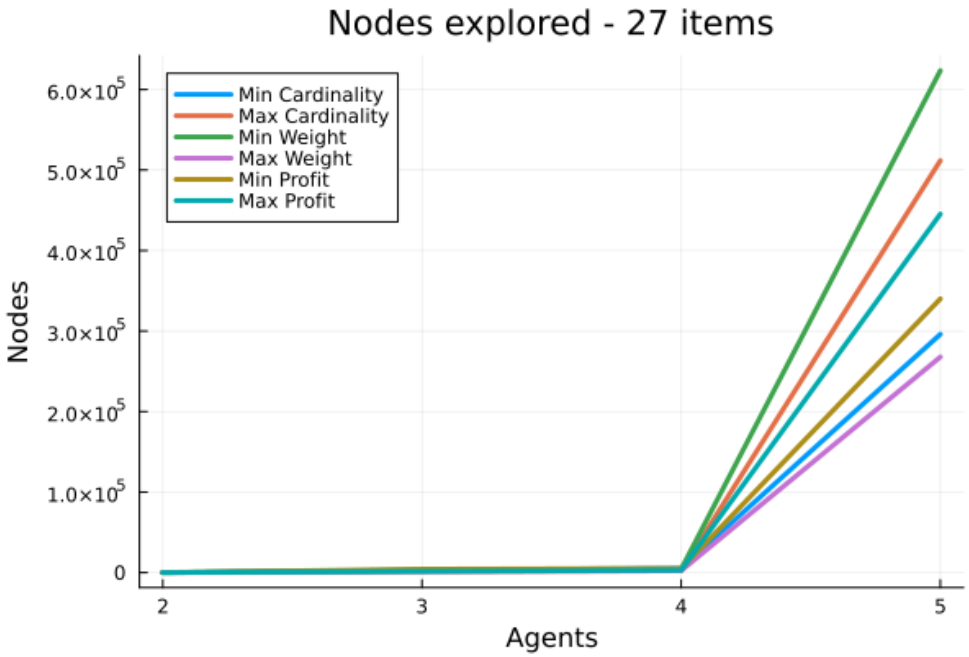



Figure 6.12: Value-ordering heuristics, 27 items

## 6.6 Ordering of bins

The ordering criteria we chose to test for the bins were smallest/largest capacity, largest number of maximum valuations, and smallest capacity divided by n maximum valuations. To refresh, by number of maximum valuations, we mean the number of items for which the knapsack $k$ had the highest valuation out of all the remaining knapsacks.

$$nmax_k = \sum_{j=1}^{n} x = \begin{cases} 1, & \text{if } p_k(j) = \max_{1 \leq i \leq m}(p_i(j)) \\ 0, & \text{otherwise} \end{cases}$$

By smallest capacity divided by number of maximum valuations we mean:

$$\min_{1 \leq i \leq m} \left(\frac{c_i}{nmax_i}\right)$$

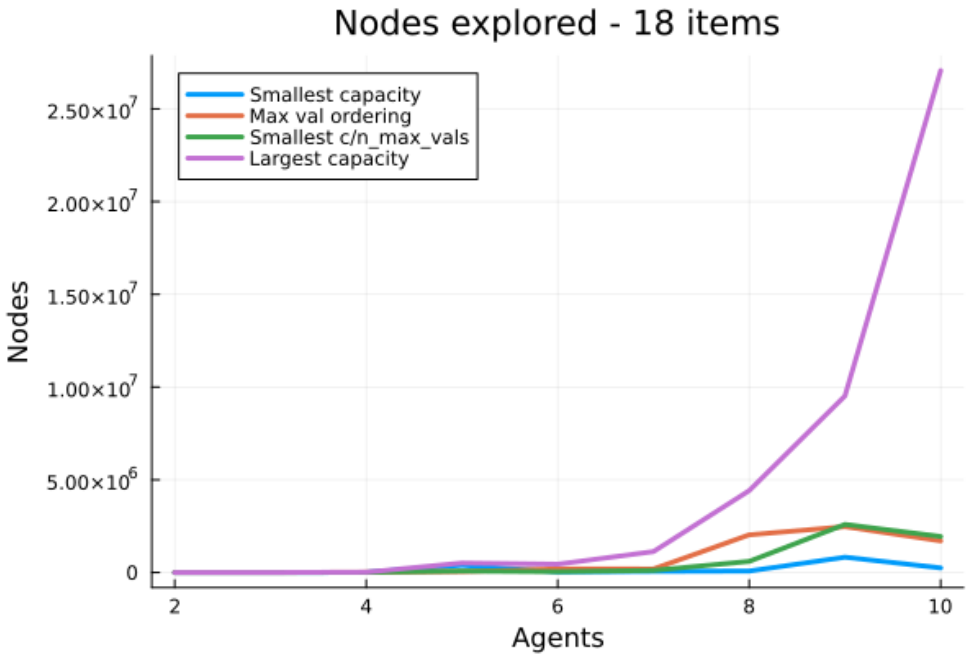
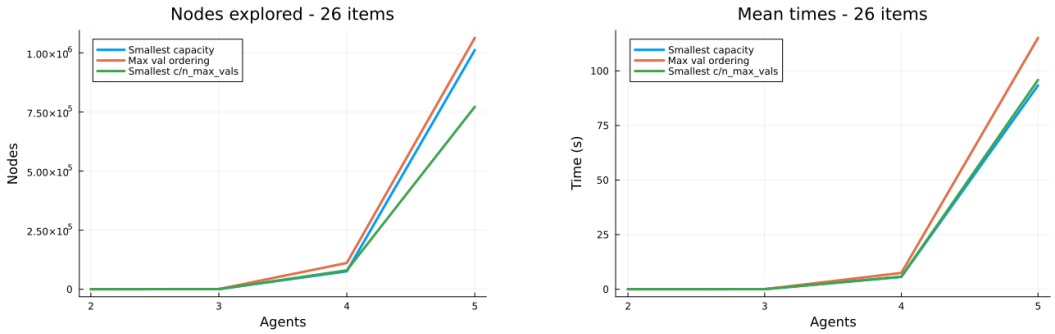



Figure 6.13: Bin ordering, 2-10 agents, 18 items

Filling the largest capacity bin first consistently performed the worst of the orderings. In certain instances like the one in figure 6.13 with 10 agents, it explored an order of magnitude more nodes than its competitors. We therefore chose to exclude it from further experiments, as it severely slowed down the data-gathering process.




(a) Nodes

(b) Mean times

Figure 6.14: Bin ordering, 2-5 agents, 26 items

The reasoning behind the inclusion of the smallest capacity divided by n maximum valuations was to strike a balance between the apparent efficacy of smallest capacity first, and the improved bounds gained by filling knapsacks with inflated valuations early. While this ordering often did end up exploring fewer nodes than smallest capacity first, the overhead from calculating the number of max valuations for each remaining bin at each node seems to cancel out the advantage gained.

## 6.7 Compared to ILP-solvers

Unfortunately the algorithm in its current implementation does not hold up to the standards of current ILP-solvers.

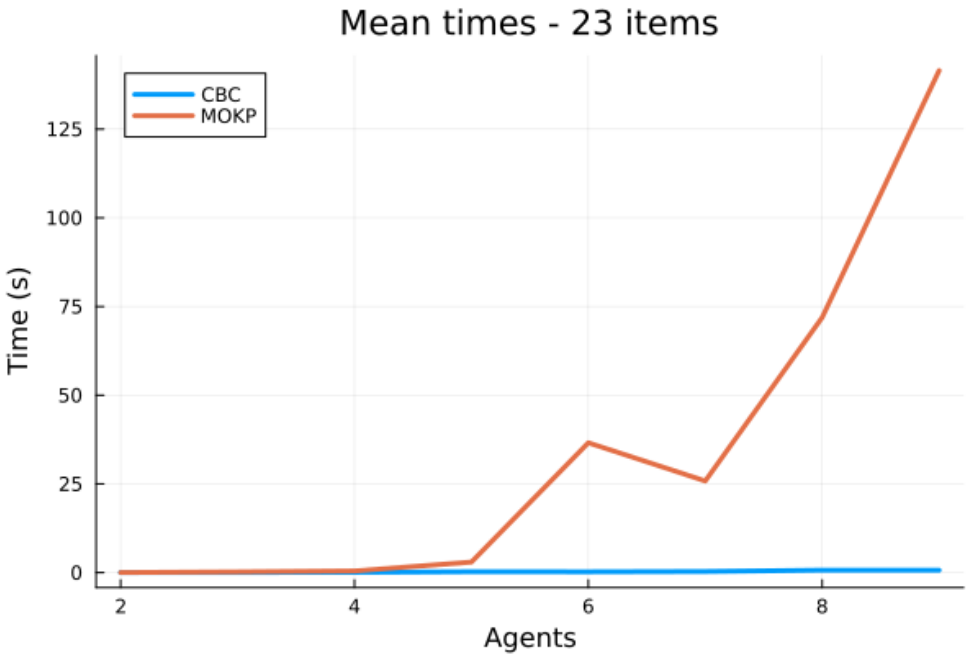



Figure 6.15: Cbc vs MOKP algorithm

# Chapter 7

# Discussion

We aimed to adapt the bin-completion algorithm such that it could solve what we chose to call multiple opinionated knapsack problems. Having developed theory on how existing optimizations for the MKP could be adapted to fit the MOKP, we implemented the theory and did experiments to assess their efficacy.

## 7.1 MOKP vs MKP

The multiple opinionated knapsack problem is more complex and harder to solve than the multiple knapsack problem simply by virtue of containing more relevant variables. The MKP is a specific case of the MOKP, where the valuations are identical amongst all the agents.

The branch-and-bound approach, whether bin- or item-based, utilized to find the optimal solution to a multiple knapsack problem can be transformed to also work for the MOKP. With no optimizations implemented, this process just requires adjusting any calculation which involves the value of an item such that it respects the valuation of the knapsack we are considering putting the item into. To ensure the validity of a surrogate upper bound, we made the surrogate knapsack value each item the same as the knapsack which values that item the most.

A valuable insight in the branch-and-bound tree one generates for the MKP is that since the placing of the items don't affect the item's value added, many of the generated solutions produce the same total value. Exploiting this symmetry and pruning unnecessary duplicates is one of the biggest improvements to exactly solving multiple knapsack problems with branch-and-bound in the last 20 years [3]. Unfortunately, these symmetries are not present in the MOKP except in the very few cases where the valuations of the relevant items coincide for two or more knapsacks. Although not studied in this thesis, the overhead incurred by checking for such instances likely outweighs the benefits, unless one plans to solve instances where knapsacks often have identical valuations.

The rest of the traditional optimizations described in the literature are applicable to the MOKP, but are consistently weakened in the transition.

## 7.2 Upper Bounds and Their Consequences

The upper bound is the most essential pruning tool available for solving the MKP with branch-and-bound, as it in addition to pruning many nodes by comparing it to the lower bound, also fuels other optimization techniques. The usefulness of the upper bound scales rapidly with its tightness, mainly due to the implementation of bound-and-bound. The bound-and-bound implementation of Mulknap has been shown to often be able to solve instances where $n/m \geq 5$ at the root node, not requiring any branch-and-bound search [11]. Unfortunately, the Mulknap approach of solving a series of subset sum problems is not applicable to the MOKP, and we therefore had to implement the original bound-and-bound strategy proposed by Martello and Toth [17].

Our current best implementation of upper bounds is based on a surrogate relaxation where the knapsack's capacities are combined, and the items take on the value of their highest valuation. As mentioned in 3.3.1, the proof for the optimality of constant multipliers by Martello and Toth [17] does not extend to the MOKP, so there may exist better surrogate upper bounds by finding more optimal multipliers. In section 6.2 we showed that the tightness of our SMKP upper bound is significantly reduced in the MOKP compared to the MKP, especially at the root node. In addition, the SMOKP upper bound more rarely matches the optimal solution's value, making the upper bound infeasible to attain, which is detrimental to bound-and-bound as a concept. We see this reflected somewhat in section 6.3, which reports minimal improvement with the implementation of bound-and-bound in certain instances with large $n/m$. However, other instances with slightly smaller $n/m$ benefitted much more from bound-and-bound.

Another optimization which relies on deriving upper bounds is Pisinger's R2 reduction [20]. It combines a surrogate relaxation of the knapsacks with a linear relaxation of the break item to derive an upper bound for a partial solution which includes some item $k$. This upper bound is used to decide whether $k$ can be present in any later assignment, given the assignments we have already chosen at that node, and still have a chance to beat our current best solution. Experimentally we found some decent improvement with the implementation of

R2, especially for instances with large ratios of $n/m$.

## 7.3 Assignments

With the help of a counterexample in section 3.3.3 we showed that the optimal assignments in the MOKP are not guaranteed to be maximal. As shown in 6.1.1, where we compared the nodes explored when including non-maximal assignments to their exclusion, this negative result has significant impact on the number of assignments generated at each node. This is the source of much of the increase in difficulty we observe when transitioning from the MKP to the MOKP.

While the domination criteria for the MKP is not suitable for the MOKP, we proposed a new domination criteria for the MOKP, and proved the necessity of each sub-criteria. Although it is weaker than the domination criteria for the MKP, section 6.1.3 showed very promising results, especially on larger instances. The performance gain can almost exclusively be attributed to the domination criteria. While the binary tree approach for generating feasible assignments is slightly faster than generating every assignment of length up-to-k, the version in which we omitted the domination criteria did not compare to the one including it.

Although briefly discussed, we did not implement any form of incrementally generated assignments. The reason for this is twofold. Firstly, the issue of the algorithm spending too much time at each node generating possible assignments is not as severe for us as it was for Fukunaga and Korf [10]. Due to both the increased complexity of the MOKP, and likely suboptimal implementation of aspects of our algorithm, we are unable to solve instances large enough to have an obscene amount of undominated and feasible assignments. Secondly, as discussed in the previous section the efficacy of bound-and-bound is severely limited due to our less tight upper bounds. The motivation behind incrementally generated assignments is to more quickly reach leaf nodes, and hopefully use bound-and-bound to find an optimal solution which allows us to backtrack and never consider the assignments we haven't generated yet. With our version of bound-and-bound being so ineffective for the MOKP, this optimization is unlikely to produce great results.

## 7.4 Ordering

As shown in sections 6.6 and 3.3.6 both the order in which we explore assignments, and the order in which we fill the bins, can have large consequences for the amount of nodes we need to explore.

Our intuition that filling knapsacks which value many items highly first to achieve more accurate upper bounds did not seem to outperform the previously established smallest capacity first ordering. The attempt at merging the two measures was more competitive, exploring less nodes than just smallest capacity first, but lost much of its efficacy to the increased overhead at each node.

A key insight which we failed to implement in time is that the overhead of calculating which bin to assign items to at each node can be moved to the preprocessing stage, given that the heuristic does not rely on the items, e.g. smallest capacity. This is an advantage of bin-based trees as opposed to item-based ones. Since a knapsack is never partially filled, we never need to recalculate its remaining capacity. By precalculating the order of the bins and instead passing it as an argument in the recursive function, we can presumably save a respectable amount of processing time at each node.

This insight also allows for less overhead for the R2 reduction, since we can sort the items by efficiency at the preprocessing stage, once for each combination of remaining bins we will encounter. Assuming we will fill the bins in the order [1, 3, 2], we would only need to sort the items 3 times:

$$\text{items}_{\{2\}} = \text{sort}(\frac{p_{2j}}{w_j})$$

$$\text{items}_{\{3,2\}} = \text{sort}(\max_{i \in \{3,2\}} (\frac{p_{ij}}{w_j})),$$

$$\text{items}_{\{1,3,2\}} = \text{sort}(\max_{i \in \{1,3,2\}} (\frac{p_{ij}}{w_j})),$$

We could pass an array of all the sorted arrays to the algorithm, index it by how many bins we have to left, and create a new array with the same ordering which only contains the remaining items.

The value-ordering heuristic for the assignments was one of the less clear-cut results we obtained. We observed max weight to be a solid candidate, which

makes sense considering that we would expect assignments which utilize as much space as possible within a bin to generally be decent solutions. However, the heuristics seemed to be very sensitive to changes in the instances, making it hard to confidently assert an optimal choice.

This remains an area for others to explore, perhaps with heuristics more suited to the MOKP specifically. The notion of comparing the value gained from an item in the current knapsack to its maximum valuation among the remaining knapsacks turned out to be an effective part of the domination criteria discussed earlier. The same notion may for example be an applicable heuristic for the assignments, providing a more nuanced concept of which assignments are likely to be optimal.

## 7.5 ILP-solvers

We found that our current implementation of the algorithm does not come close to competing with open-source ILP-solvers. There could be multitudes of reasons for this. Cbc has been developed and maintained by a much larger group of people over a longer period of time, and is therefore much more optimized. Our Julia implementation is far from as optimized as possible, which could contribute to some of the difference in runtime.

All the different techniques Cbc uses to solve ILP's is beyond the scope of this thesis, but are naturally an important factor. Comparing the nodes explored by Cbc to our own solution proved difficult, as it often reports having enumerated 0 nodes after explicitly reporting finding a new best solution by branch-and-bound. As it stands, it seems the multiple cutting techniques it deploys to reduce the set of feasible solutions to the linear relaxation of the problem are very effective in limiting the number of nodes one has to explore in B&B. Considering our version of the surrogate relaxation is not proveably better than the linear relaxation, the more sophisticated techniques that have been developed to tighten the linear relaxation may be a determining factor.

# Chapter 8

# Conclusion

We set out to adapt known optimization strategies for the MKP to the MOKP, where items are valued differently by each knapsack. A bin-based branch-and-bound algorithm was presented and implemented in Julia, which managed to incorporate versions of each optimization we considered, excluding the ones that intrinsically relied on objective valuations. Specifically, these were path-symmetry and the subset sum version of bound-and-bound.

The increased complexity of the MOKP weakened the contribution of each optimization to differing degrees. Due to less tight upper bounds, the bound-and-bound technique, which has significant impact in the MKP, was deemed essentially worthless. Both R2 reduction and the normal upper bound pruning, which also rely on upper bounds, suffered as well, but still remained valueable strategies. Ways of deriving tigther upper bounds are the main insufficiency of the proposed algorithm, and would go a long way towards optimizing it.

We proved that optimal assignments in the MOKP are not necessarily maximal assignments, and demonstrated the severe effect this has on the number of assignments one must consider. To combat this a new domination criteria was presented for the MOKP, which proved to be a very valueable inclusion. Even with all of these optimizations we were unable to compete with the highly optimized ILP-solver Cbc, which solved the problem instances in significantly less time.

Regarding the algortihm's relevance to fair allocation, we found that some of the optimization concepts, such as bound-and-bound, can be adapted to fit non-linear optimization goals such as maximum NSW, which ILP-solvers are not designed for. Adjustments to the derivation of upper bounds are required, but branch-and-bound remains a viable method for solving such scenarios as well.

# Chapter 9

# Further Work

As previously mentioned, the main way to improve the algorithm is by looking for better ways to calculate upper bounds. A missing component of our algorithm that has shown promise in other applications [10, 22] is incrementally generated completions. Whilst we currently doubt its contribution somewhat considering the reduced effectiveness of bound-and-bound, testing whether it grants any improvement in this context may be a worthwhile endeavor after improving the upper bounds.

The initial inspiration for this project was for it to be a contribution to the field of fair allocation. Whilst we didn't get as far with this as we would have liked, the expansion of optimization techniques for the MKP to general assignment problems with individual valuations and the discussion of their further expansion to budget-constrained fairness scenarios hopefully leaves a clear path for further work.

In order to relate the algorithm to fair allocation, one could for instance experiment with changing the maximization goal to something like egalitarian or utilitarian social welfare, or include a constraint which requires the solution to satisfy some $\alpha$-MMS for each agent. Perhaps the easiest to implement would be the maximum Nash social welfare, where one attempts to maximize the product of values instead of the sum, although this has already been explored by Wu et al. [25].

There are also several intriguing open questions remaining in budget-constrained fair allocation, including the existence and computation of EF-1 guarantees under identical valuations, and what MMS and proportionality guarantees can be provided by a Max-NSW allocation. Note that it has been proved that for individual valuations and $m \geq 3$ agents, there is always a possible set of items which result in no allocation satisfying the MMS-requirements of every agent [15].

## 9.1 Definition of MMS under budget constraints

Under budget constraints the definition of MMS is not immediately obvious, and we are yet to find one proposed in the literature. Due to the variance in agents' budgets, agents may become envious of a bundle which they cannot afford themselves. There are several possibilities for such a definition, but which one is best boils down to what one expects from an MMS-share. One such quality, and why it is often useful especially in constrained scenarios, is its ability to adapt to the shrinking set of feasible solutions, i.e. it is a more realistic expectation than proportionality. At the same time, if one relaxes the fairness notion too much, it can hardly be said to be a useful measure of how fair an allocation is. Additionally, one could always simply scale the fairness notion some with some $0 \leq \alpha \leq 1$ to relax the requirement of each agent.

We will present some ideas for a definition of MMS under budget constraints, but leave the choice of which is better suited to future studies:

1. Assume that every other agent has the same budget as the agent we are currently calculating the MMS-value of.

2. Calculate the MMS-value for an agent based a feasible allocation that respects the actual budgets of the other agents.

3. Respect all the budgets, and scale the value of an assignment to an agent $j$ with the ratio between the budgets of the agent we are calculating the MMS-value for $i$, and $j$.

Number 3. is essentially the notion of WMMS introduced by Farhadi et al. in [9] for instances where agents have differing entitlements to the pot of items, which is a possible way of thinking about budget constraints. It may therefore be the most natural candidate, as to align with the related field.

# References


[1] Siddharth Barman, Arindam Khan, Sudarshan Shyam, and K. V. N. Sreenivas. *Finding Fair Allocations under Budget Constraints*. Mar. 17, 2023. arXiv: 2208.08168[cs]. URL: http://arxiv.org/abs/2208.08168 (visited on 05/24/2023).

[2] Sylvain Bouveret, Katarína Cechlárová, Edith Elkind, Ayumi Igarashi, and Dominik Peters. "Fair Division of a Graph". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. Place: Melbourne, Australia. AAAI Press, 2017, pp. 135–141. ISBN: 978-0-9992411-0-3. URL: http://arxiv.org/abs/1705.10239.

[3] Valentina Cacchiani, Manuel Iori, Alberto Locatelli, and Silvano Martello. "Knapsack problems — An overview of recent advances. Part II: Multiple, multidimensional, and quadratic knapsack problems". In: *Computers & Operations Research* 143 (2022), p. 105693. ISSN: 0305-0548. DOI: https://doi.org/10.1016/j.cor.2021.105693. URL: https://www.sciencedirect.com/science/article/pii/S0305054821003889.

[4] *Cbc (Coin-or branch and cut)*. Version 2.10.10. DOI: 10.5281/zenodo.7843975. URL: https://zenodo.org/record/7843975.

[5] Bhaskar Ray Chaudhury, Telikepalli Kavitha, Kurt Mehlhorn, and Alkmini Sgouritsa. "A Little Charity Guarantees Almost Envy-Freeness". In: *SIAM Journal on Computing* 50.4 (2021), pp. 1336–1358. DOI: 10.1137/20M1359134. eprint: https://doi.org/10.1137/20M1359134. URL: https://doi.org/10.1137/20M1359134.

[6] Jiahao Chen and Jarrett Revels. "Robust benchmarking in noisy environments". In: *arXiv e-prints* (Aug. 2016). _eprint: 1608.04295.

[7] Mauro Dell'Amico, Maxence Delorme, Manuel Iori, and Silvano Martello. "Mathematical models and decomposition methods for the multiple knapsack problem". In: *European Journal of Operational Research* 274.3 (May 1, 2019), pp. 886–899. ISSN: 0377-2217. DOI: 10.1016/j.ejor.2018.10.043. URL: https://www.sciencedirect.com/science/article/pii/S0377221718309111 (visited on 10/20/2022).

[8] Iain Dunning, Joey Huchette, and Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". In: *SIAM Review* 59.2 (Jan. 2017), pp. 295–320. ISSN: 0036-1445, 1095-7200. DOI: 10.1137/15M1020575. URL: https://epubs.siam.org/doi/10.1137/15M1020575 (visited on 12/09/2022).

[9] Alireza Farhadi, Mohammad Ghodsi, MohammadTaghi Hajiaghayi, Sebastien Lahaie, David Pennock, Masoud Seddighin, Saeed Seddighin, and Hadi Yami. "Fair Allocation of Indivisible Goods to Asymmetric Agents". In: *Journal of Artificial Intelligence Research* 64 (Mar. 2017). DOI: 10.1613/jair.1.11291. URL: http://arxiv.org/abs/1703.01649.

[10] A. S. Fukunaga and R. E. Korf. "Bin Completion Algorithms for Multicontainer Packing, Knapsack, and Covering Problems". In: *Journal of Artificial Intelligence Research* 28 (Mar. 30, 2007), pp. 393–429. ISSN: 1076-9757. DOI: 10.1613/jair.2106. URL: https://jair.org/index.php/jair/article/view/10492 (visited on 11/03/2022).

[11] Alex S. Fukunaga. "A branch-and-bound algorithm for hard multiple knapsack problems". In: *Annals of Operations Research* 184.1 (Apr. 2011), pp. 97–119. ISSN: 0254-5330, 1572-9338. DOI: 10.1007/s10479-009-0660-y. URL: http://link.springer.com/10.1007/s10479-009-0660-y (visited on 10/10/2022).

[12] Halvard Hummel and Magnus Lie Hetland. "Fair allocation of conflicting items". In: *Autonomous Agents and Multi-Agent Systems* 36.1 (Dec. 23, 2021), p. 8. ISSN: 1573-7454. DOI: 10.1007/s10458-021-09537-3. URL: https://doi.org/10.1007/s10458-021-09537-3 (visited on 12/06/2022).

[13] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN: 978-3-642-07311-3 978-3-540-24777-7. DOI: 10.1007/978-3-540-24777-7. URL: http://link.springer.com/10.1007/978-3-540-24777-7 (visited on 10/10/2022).

[14] Jonathan A. Kelner and Daniel A. Spielman. "A randomized polynomial-time simplex algorithm for linear programming". In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing*. STOC06: Symposium on Theory of Computing. Seattle WA USA: ACM, May 21, 2006, pp. 51–60. ISBN: 978-1-59593-134-4. DOI: 10.1145/1132516.1132524. URL: https://dl.acm.org/doi/10.1145/1132516.1132524 (visited on 06/10/2023).

[15] David Kurokawa, Ariel D. Procaccia, and Junxing Wang. "Fair Enough: Guaranteeing Approximate Maximin Shares". In: *Journal of the ACM* 65.2 (Apr. 30, 2018), pp. 1–27. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/3140756. URL: https://dl.acm.org/doi/10.1145/3140756 (visited on 06/01/2023).

[16] Miles Lubin, Oscar Dowson, Joaquim Dias Garcia, Joey Huchette, Benoît Legat, and Juan Pablo Vielma. "JuMP 1.0: Recent improvements to a modeling language for mathematical optimization". In: *Mathematical Programming Computation* (2023). DOI: 10.1007/s12532-023-00239-3.

[17] Silvano Martello and Paolo Toth. "A Bound and Bound algorithm for the zero-one multiple knapsack problem". In: *Discrete Applied Mathematics*. Special Copy 3.4 (Nov. 1, 1981), pp. 275–288. ISSN: 0166-218X. DOI: 10.1016/0166-218X(81)90005-6. URL: https://www.sciencedirect.com/science/article/pii/0166218X81900056 (visited on 12/10/2022).

[18] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. USA: John Wiley & Sons, Inc., 1990. ISBN: 0-471-92420-2.

[19] George L. Nemhauser and Laurence A Wolsey. *Integer and Combinatorial Optimization — Wiley*. 1st ed. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley-Interscience, 1999. 763 pp. ISBN: 0-471-82819-2. URL: https://www.wiley.com/en-kr/Integer+and+Combinatorial+Optimization-p-9780471359432 (visited on 06/01/2023).

[20] David Pisinger. "An exact algorithm for large multiple knapsack problems". In: *European Journal of Operational Research* 114.3 (May 1999), pp. 528–541. DOI: 10.1016/S0377-2217(98)00120-9. URL: https://ideas.repec.org/a/eee/ejores/v114y1999i3p528-541.html.

[21] Stuart Russel and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Pearson Series in Artificial Intelligence. 2022. 1166 pp. ISBN: 1-292-40113-3. URL: https://aima.cs.berkeley.edu/.

[22] Ethan L Schreiber and Richard E Korf. "Improved Bin Completion for Optimal Bin Packing and Number Partitioning". In: *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence* (Aug. 2013). URL: https://www.ijcai.org/Proceedings/13/Papers/103.pdf.

[23] Jørgen Steig. "Solving budget constrained fair allocation problems with multiple knapsack". Precursor project in TDT4501. 2022.

[24] Favio Tardella. "The fundamental theorem of linear programming: extensions and applications". In: *Optimization - Taylor & Francis* 60.1-2 (2011), pp. 283–301.

[25] Xiaowei Wu, Bo Li, and Jiarui Gan. "Budget-feasible maximum nash social welfare is almost envy-free". In: International Joint Conferences on Artificial Intelligence Organization, 2021. DOI: 10.48550/arXiv.2012.03766. URL: http://arxiv.org/abs/2012.03766.

[26] Li'ang Zhang and Suyun Geng. "The complexity of the 0/1 multi-knapsack problem". In: *Journal of Computer Science and Technology* 1.1 (Mar. 1986), pp. 46–50. ISSN: 1000-9000, 1860-4749. DOI: 10.1007/BF02943300. URL: http://link.springer.com/10.1007/BF02943300 (visited on 12/09/2022).

# Appendix

This appendix contains a wider selection of results than we were able to fit in the results section. They are grouped by which optimization they relate to, and contain both mean time spent and nodes explored for each. The code for our implementation is not included here, but is retrieveable at https://github.com/jorgstei/FairMulKnap.

## Up-to-k vs Undominated Assignments

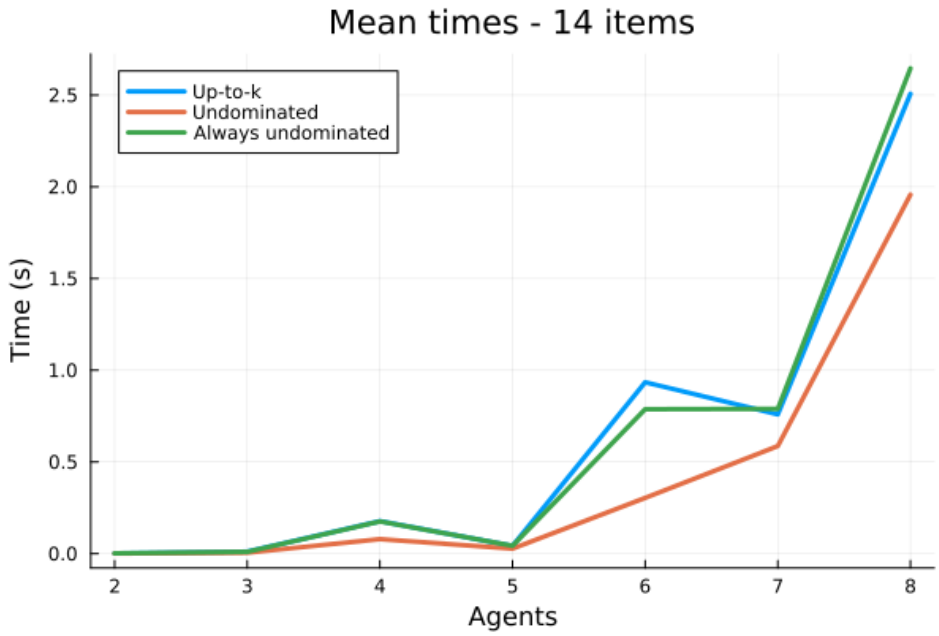



Figure 1

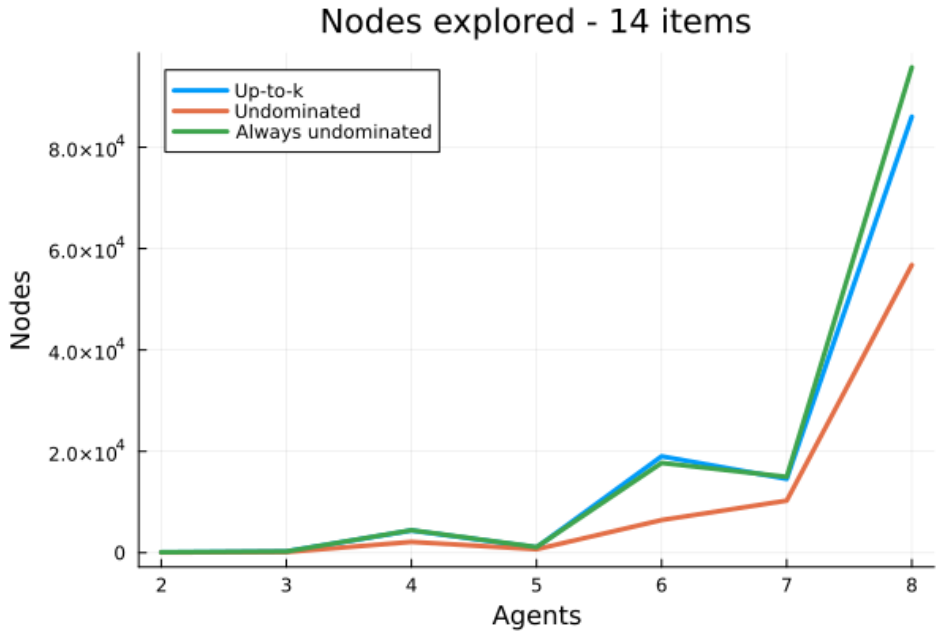



Figure 2

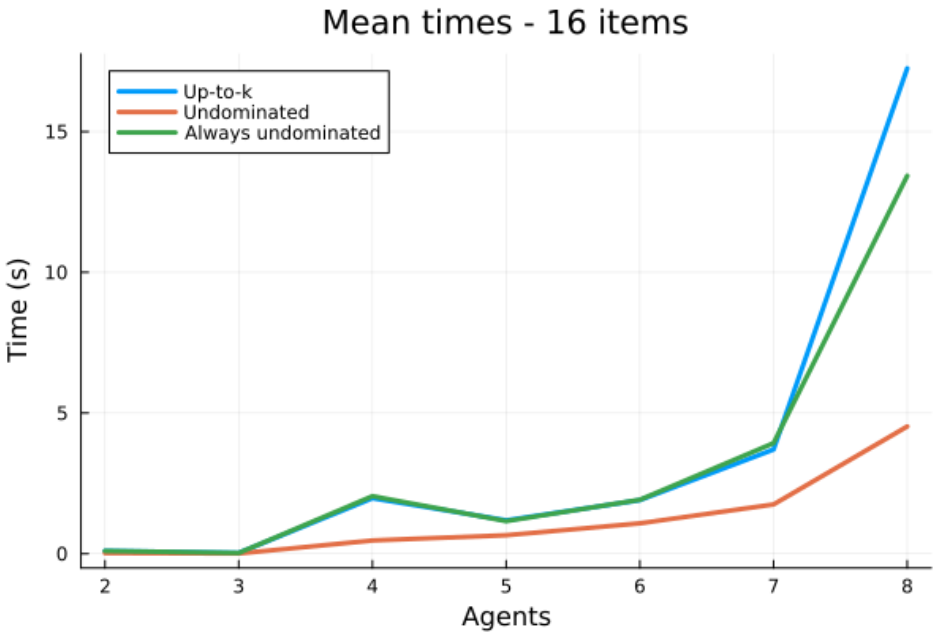



Figure 3

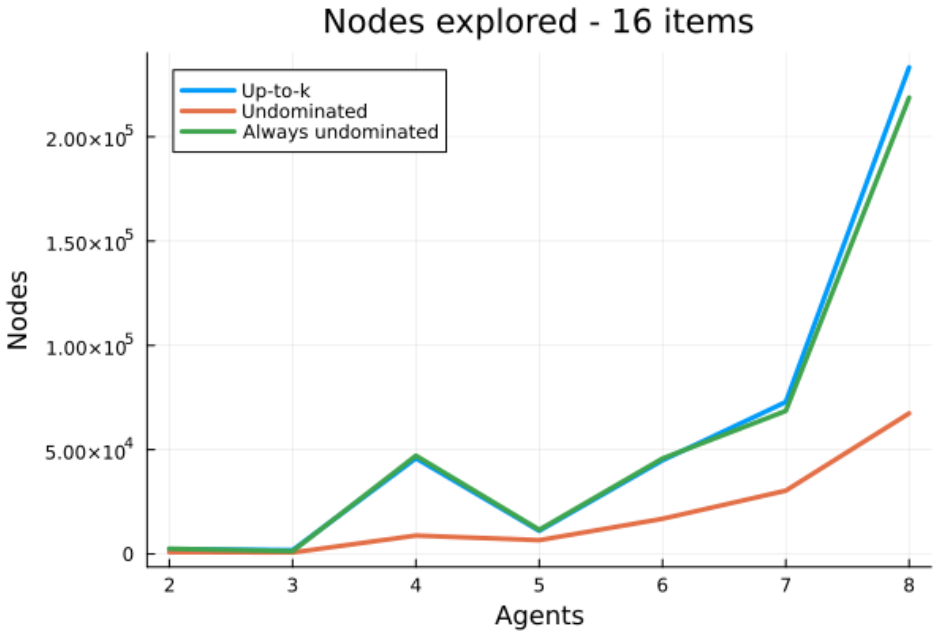



Figure 4

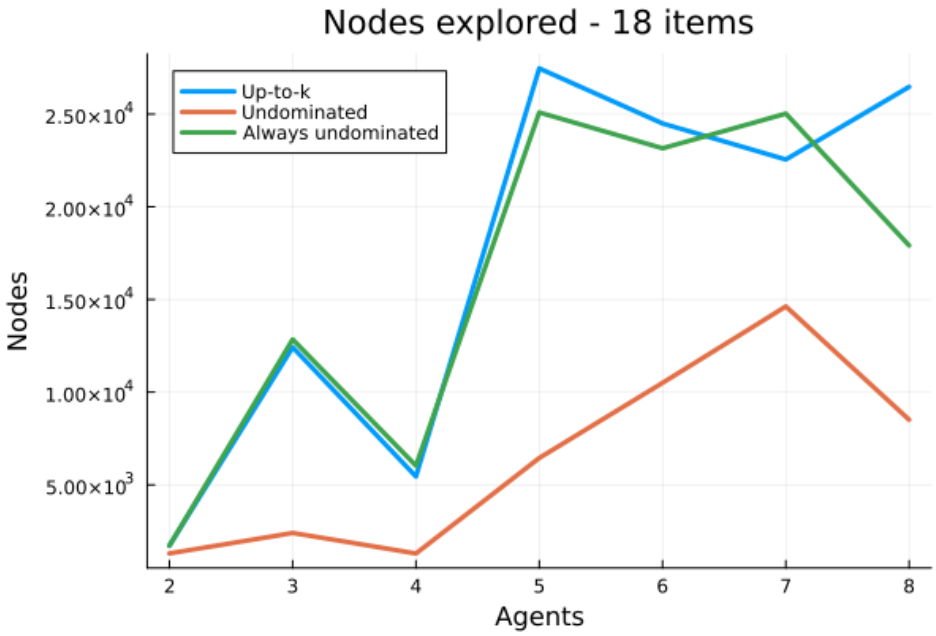



Figure 5

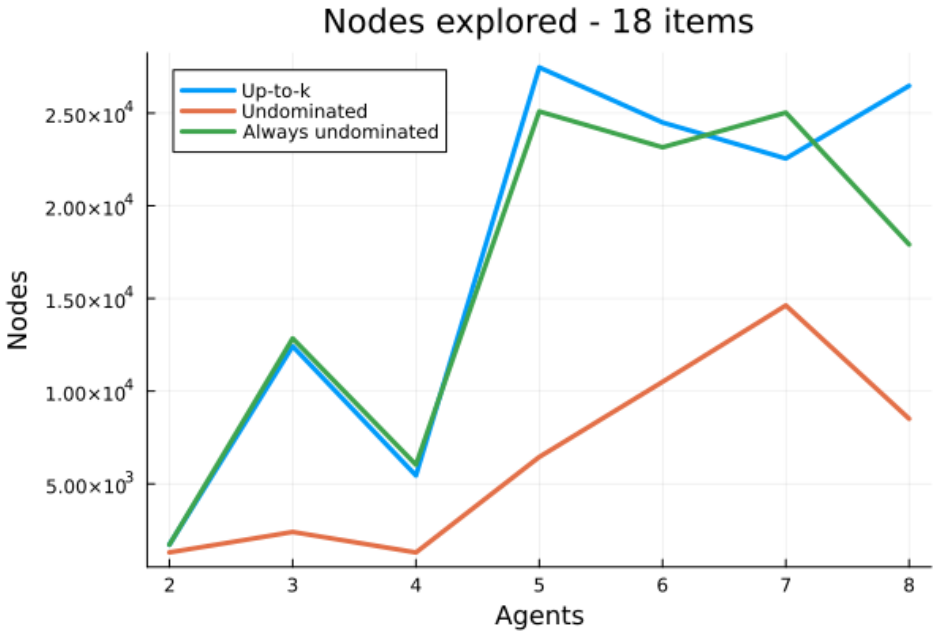



Figure 6

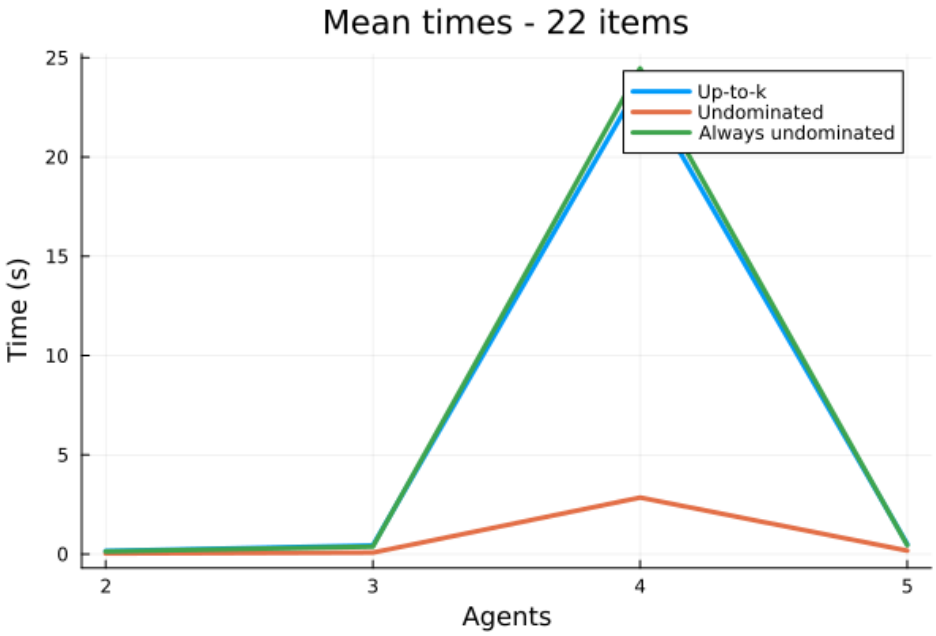



Figure 7

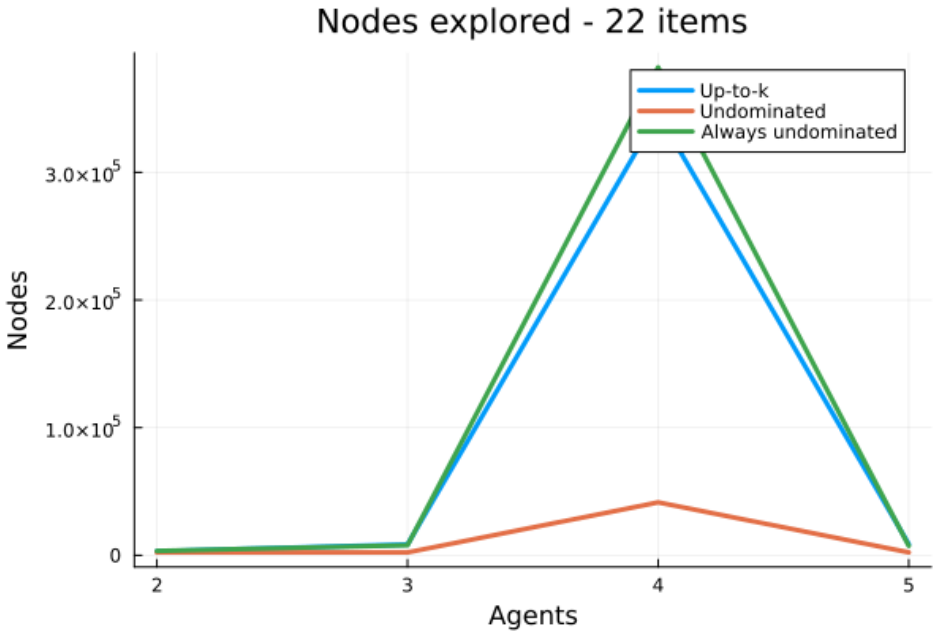



Figure 8

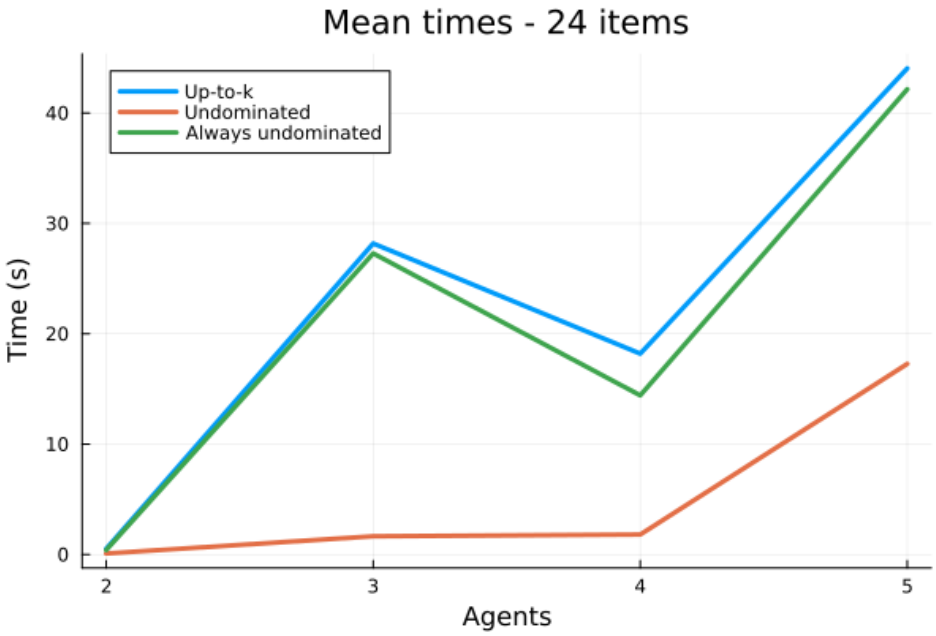



Figure 9

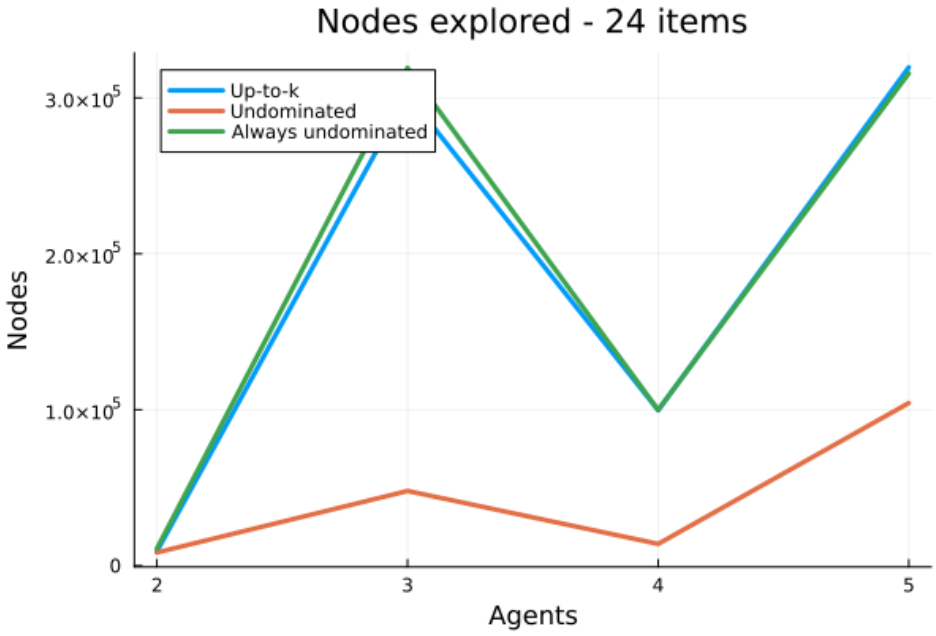



Figure 10

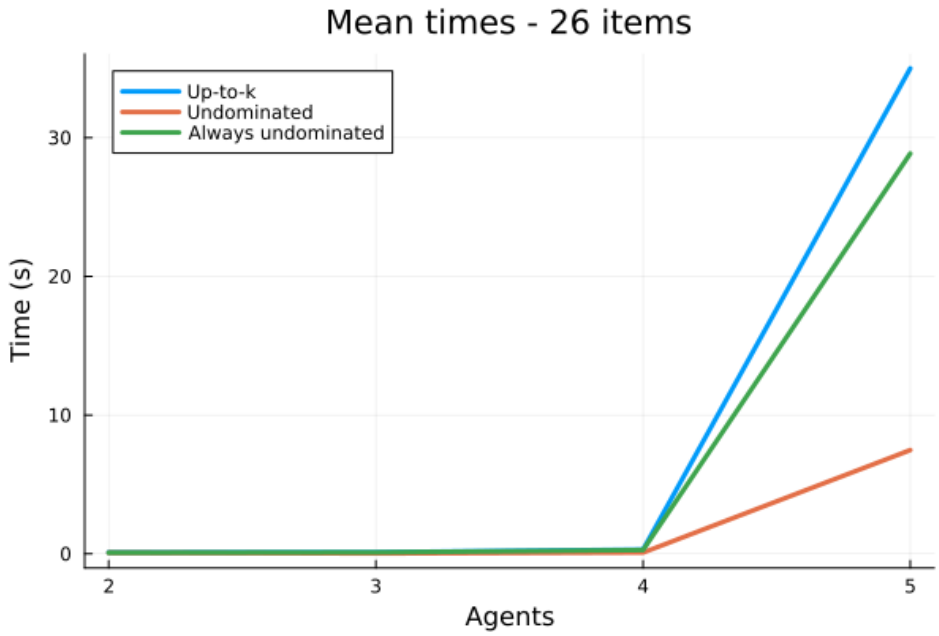



Figure 11

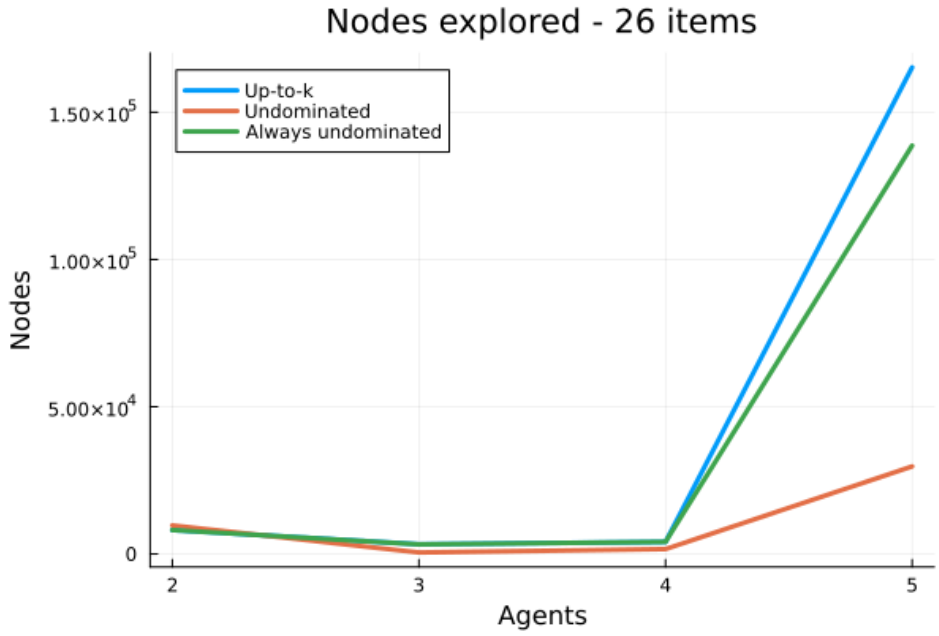



Figure 12

## Upper bounds

| Items \ Bins | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 2 | 1.00 / 1.02 | 1.00 / 1.01 | 1.00 / 1.02 | 1.00 / 1.00 | 1.00 / 1.00 | 1.00 / 1.00 |
| 4 | 1.12 / 1.48 | 1.00 / 1.16 | 1.00 / 1.03 | 1.00 / 1.05 | 1.00 / 1.04 | 1.00 / 1.05 |
| 6 | 1.00 / 1.07 | 1.00 / 1.06 | 1.03 / 1.16 | 1.01 / 1.10 | 1.00 / 1.08 | 1.00 / 1.07 |
| 8 | 1.05 / 1.10 | 1.00 / 1.09 | 1.02 / 1.16 | 1.00 / 1.06 | 1.00 / 1.03 | 1.00 / 1.04 |
| 10 | 1.04 / 1.18 | 1.01 / 1.12 | 1.01 / 1.12 | 1.01 / 1.13 | 1.02 / 1.13 | 1.00 / 1.07 |
| 12 | 1.00 / 1.07 | 1.00 / 1.09 | 1.01 / 1.10 | 1.00 / 1.10 | 1.00 / 1.06 | 1.00 / 1.08 |



Figure 13

| Items \ Bins | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|
| **10** | 5 / 65 | 0.7 / 48 | 0.5 / 34 | 0.5 / 30 | 0 / 19 |
| **14** | 6 / 90 | 4 / 82 | 0.7 / 58 | 0 / 33 | 0.1 / 34 |
| **18** | 5 / 99 | 3 / 84 | 3 / 80 | 2 / 60 | 0.2 / 47 |
| **22** | 3 / 80 | 2 / 78 | 2 / 77 | 1 / 83 | 0.6 / 61 |
| **26** | 1 / 68 | 2 / 72 | 2 / 74 | 1 / 74 | 1 / 73 |

Figure 14

## With and without bound-and-bound

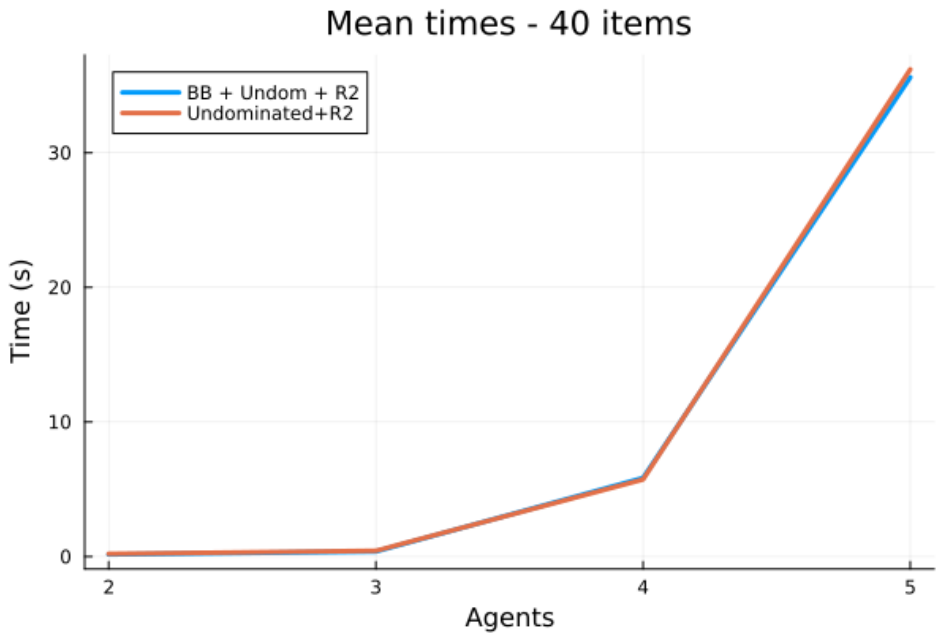



Figure 15

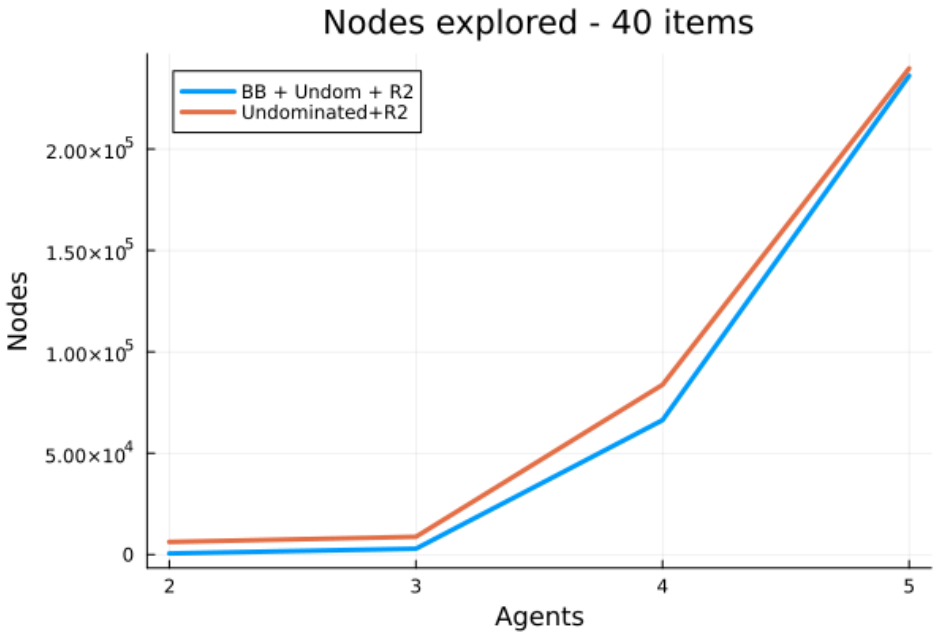



Figure 16

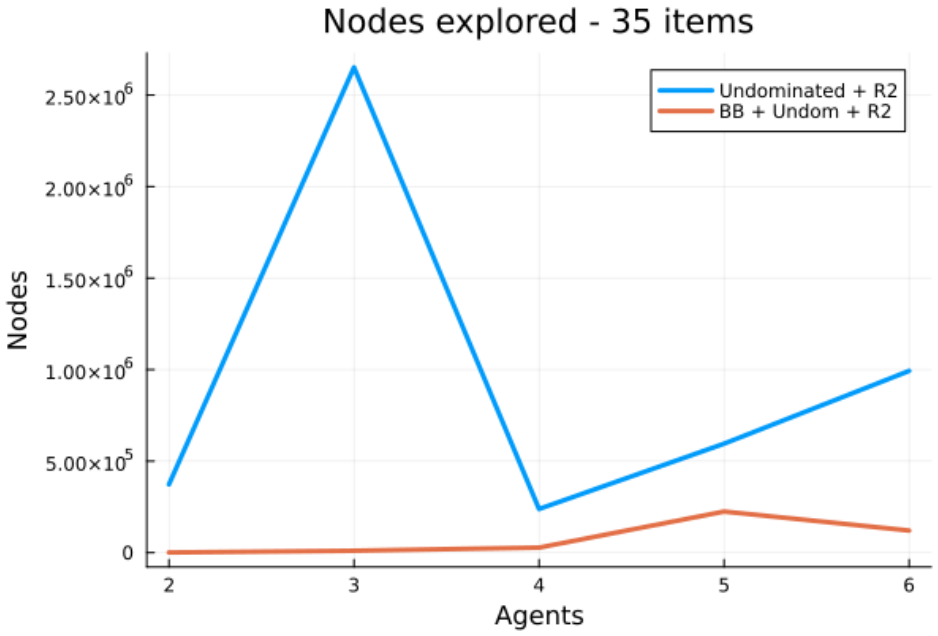



Figure 17

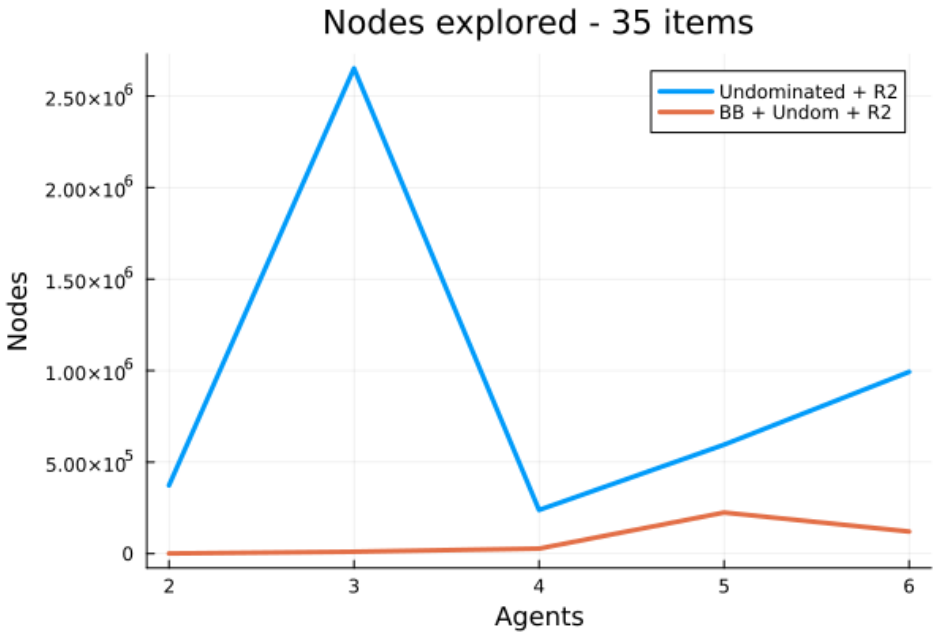

Nodes explored - 35 items
Undominated + R2
BB + Undom + R2
Nodes
2.50×10^6
2.00×10^6
1.50×10^6
1.00×10^6
5.00×10^5
0
2
3
4
5
6
Agents

Figure 18

## R2 reduction

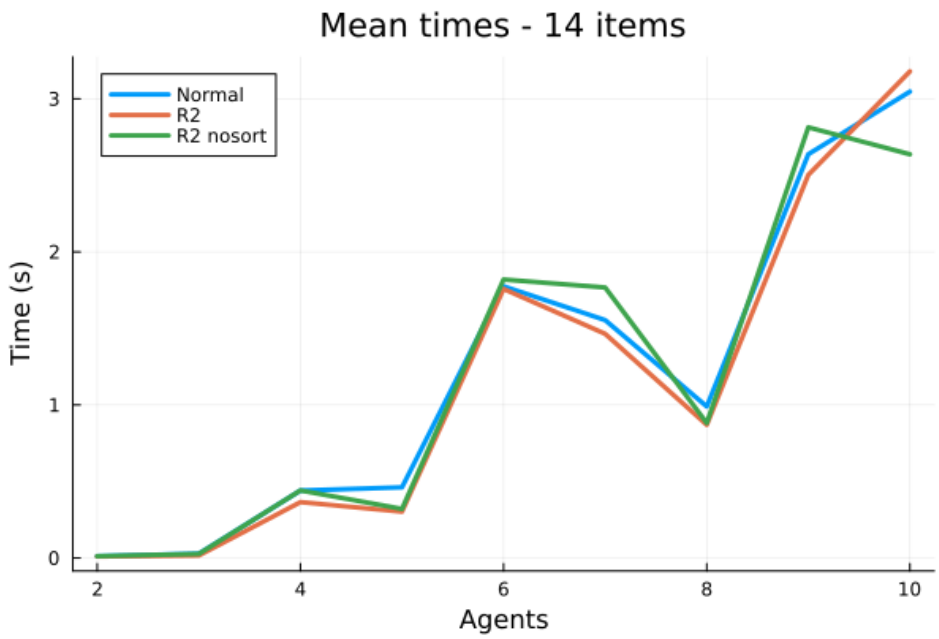



Figure 19

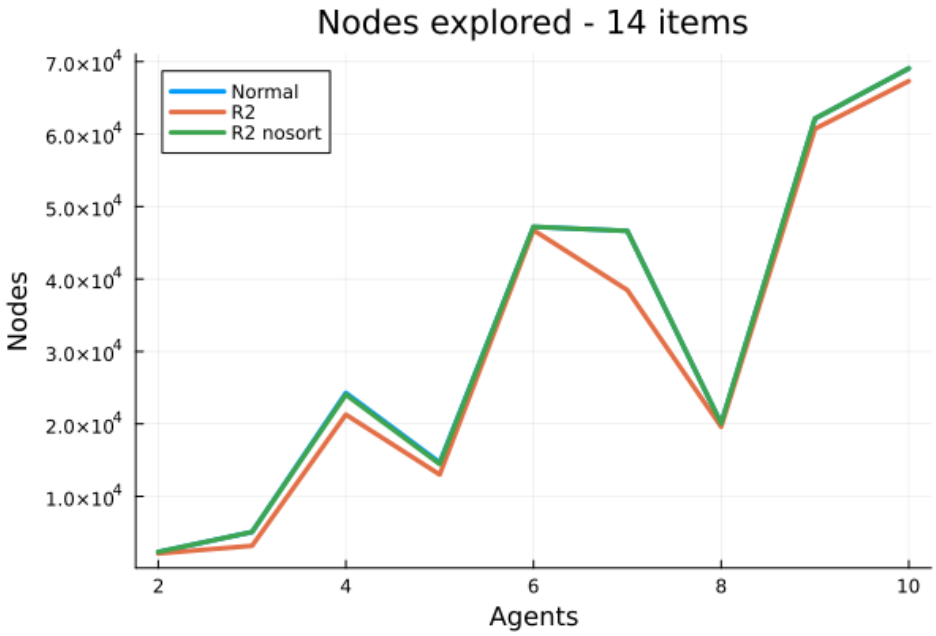



Figure 20

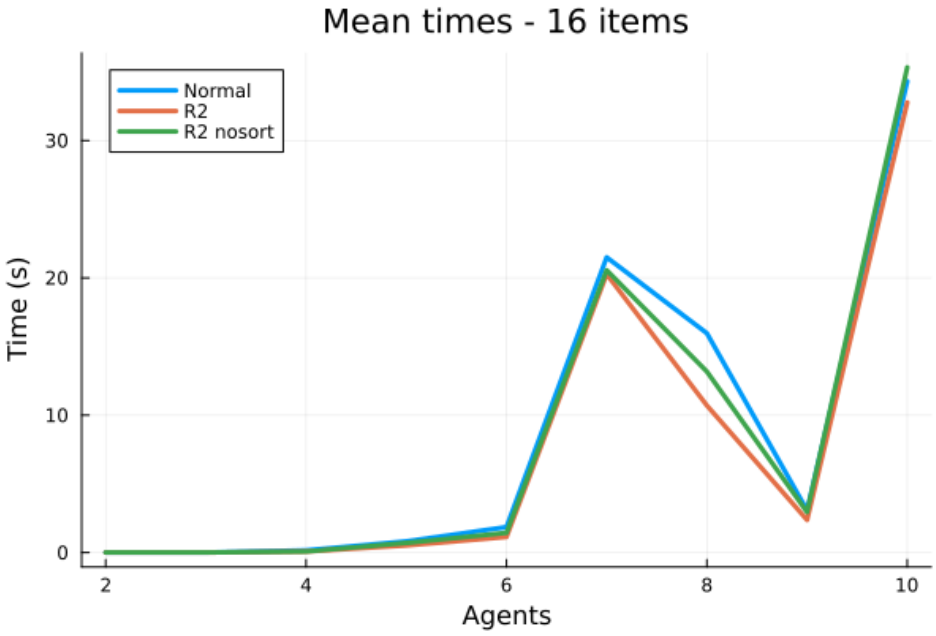



Figure 21

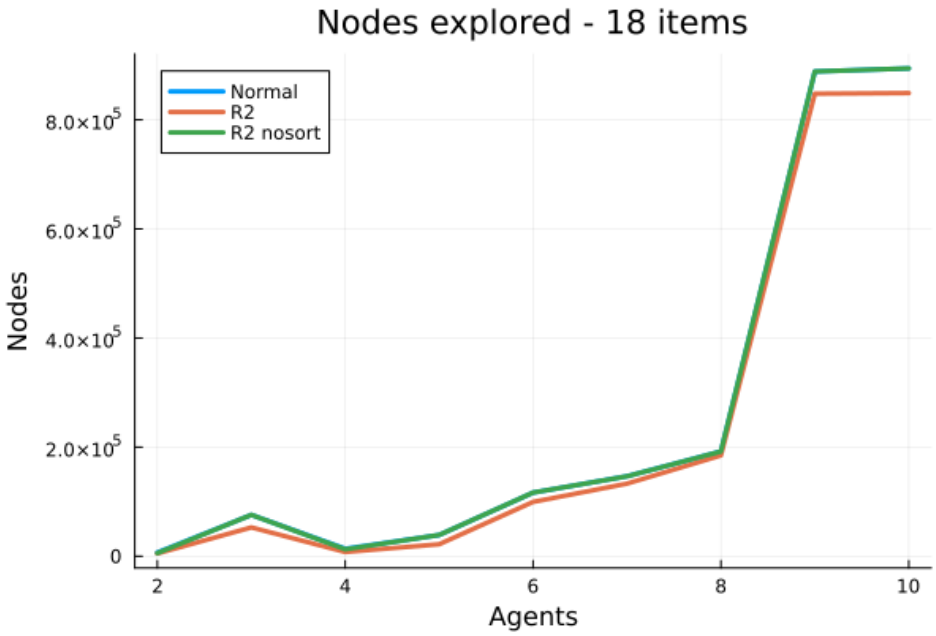



Figure 22

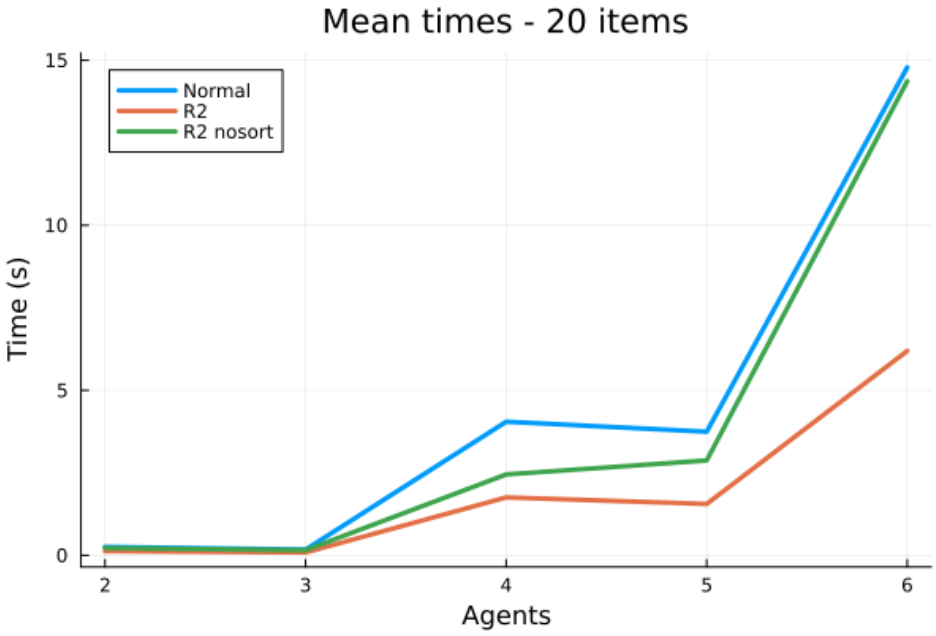



Figure 23

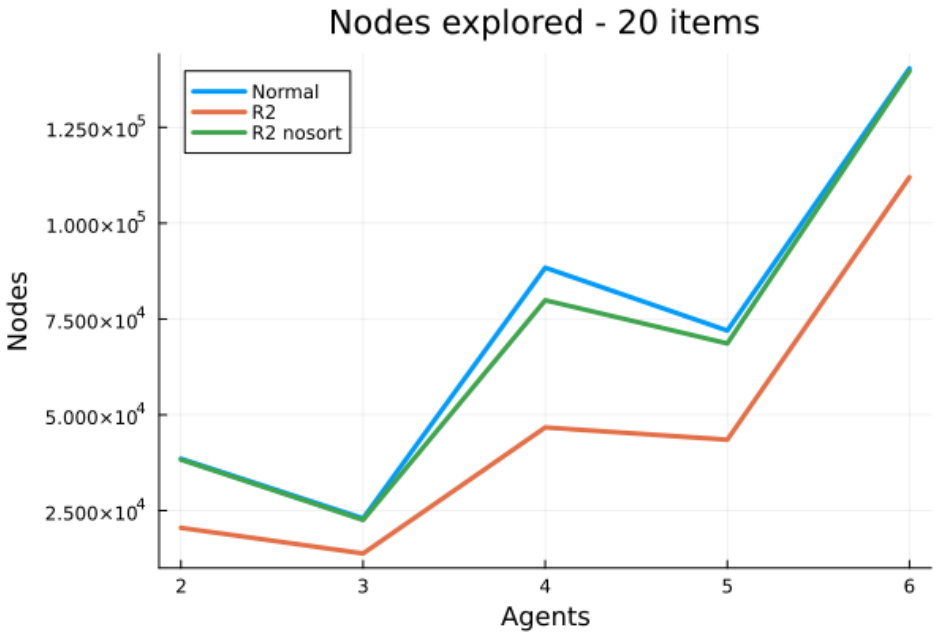



Figure 24

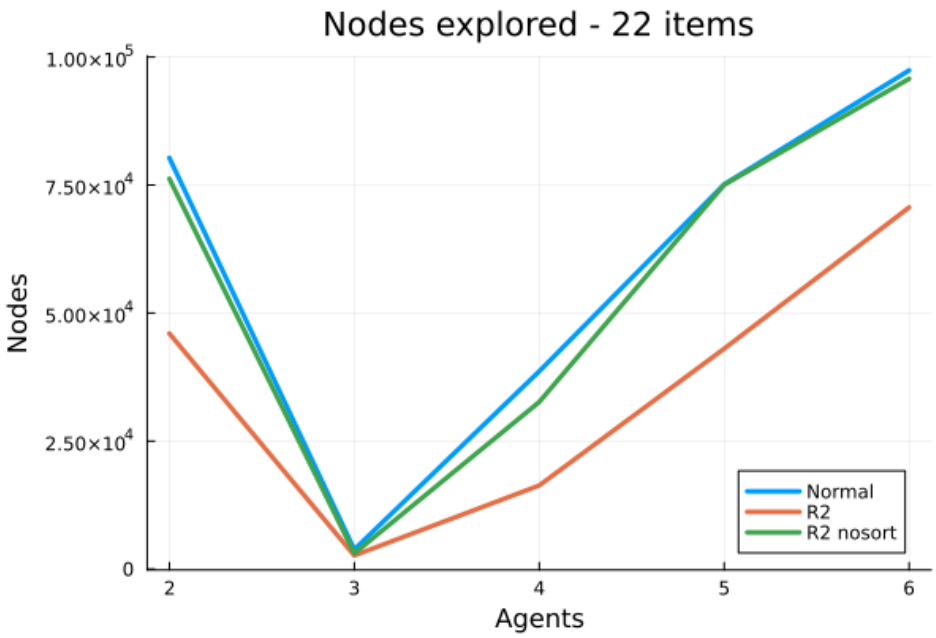



Figure 25

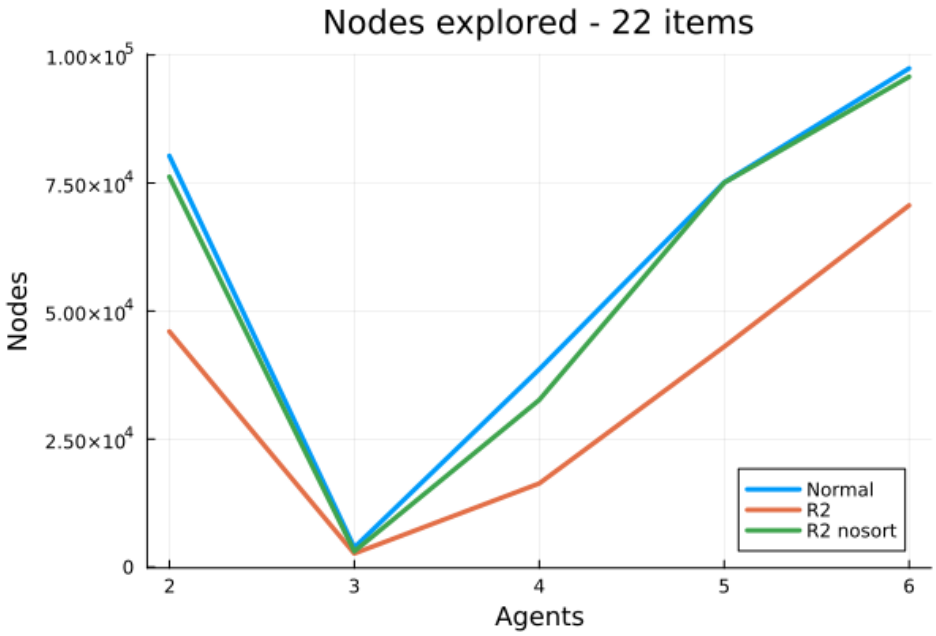



Figure 26

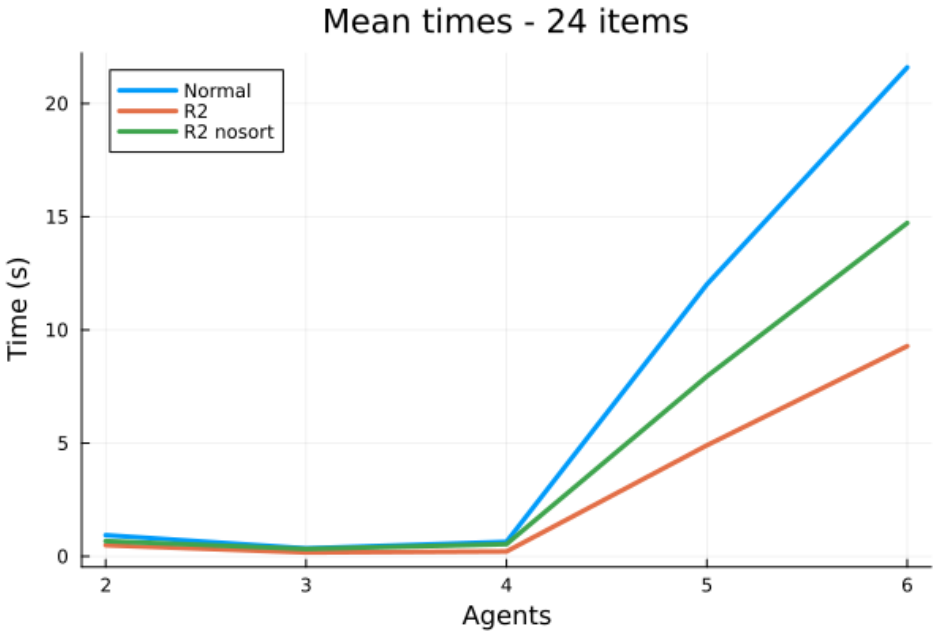



Figure 27

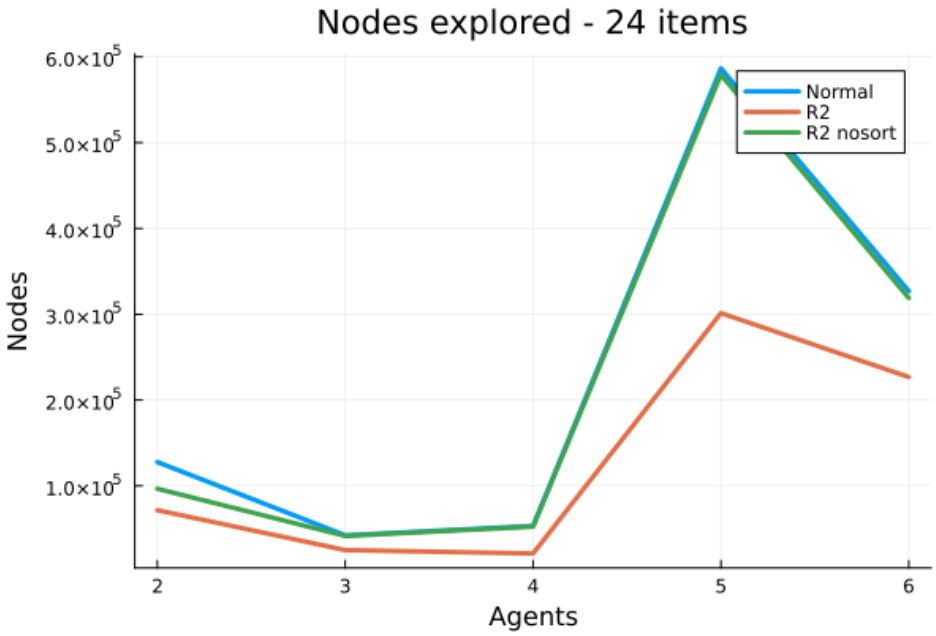



Figure 28

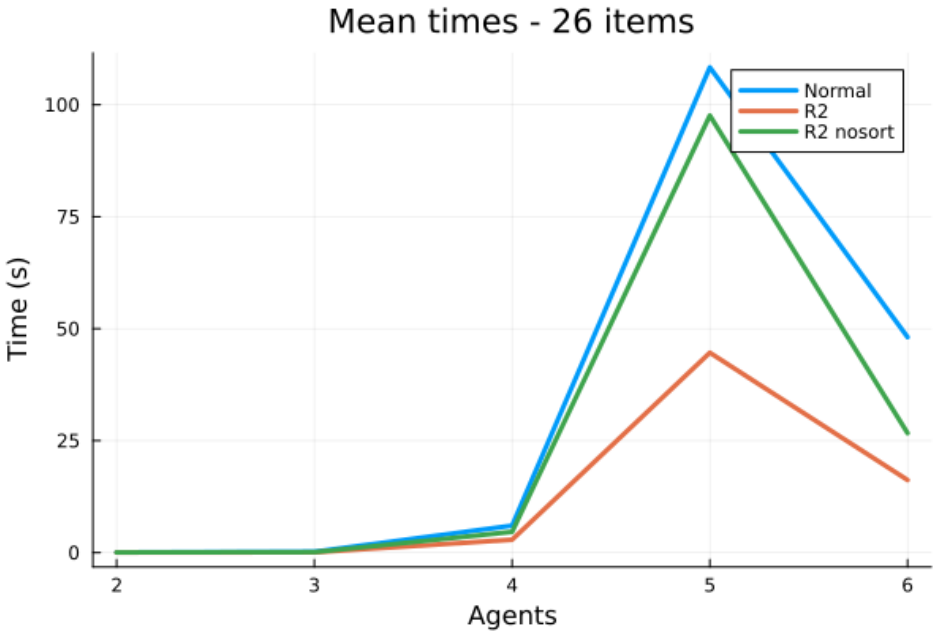



Figure 29

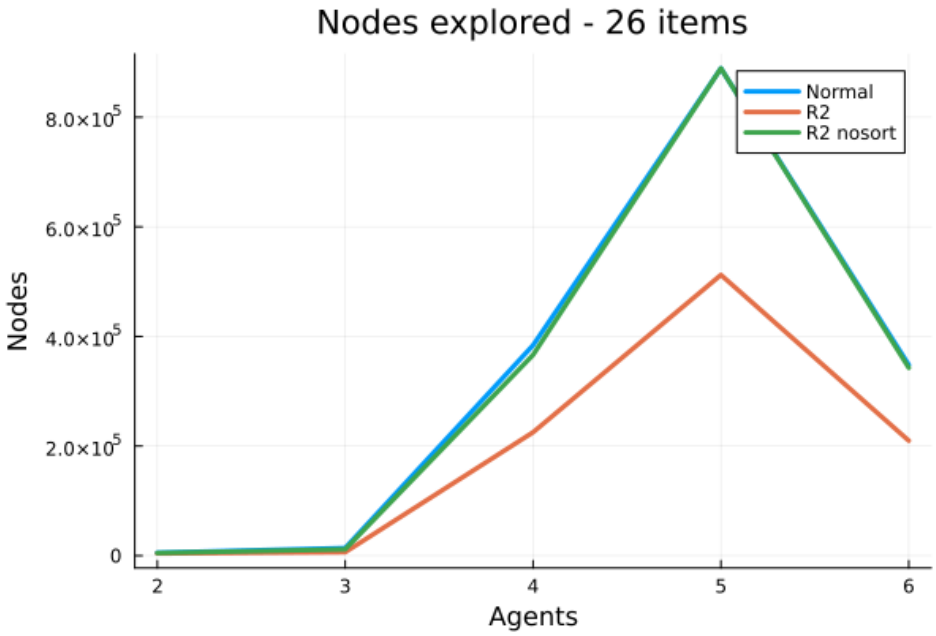

Nodes explored - 26 items
Normal
R2
R2 nosort
Nodes
8.0×10^5
6.0×10^5
4.0×10^5
2.0×10^5
0
2
3
4
5
6
Agents

Figure 30

## Bin ordering

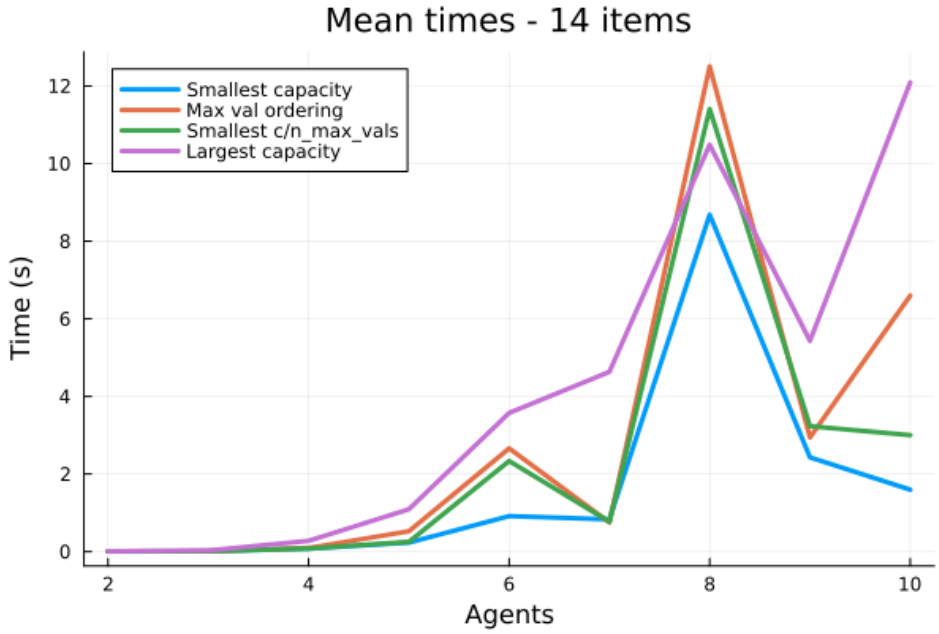



Figure 31

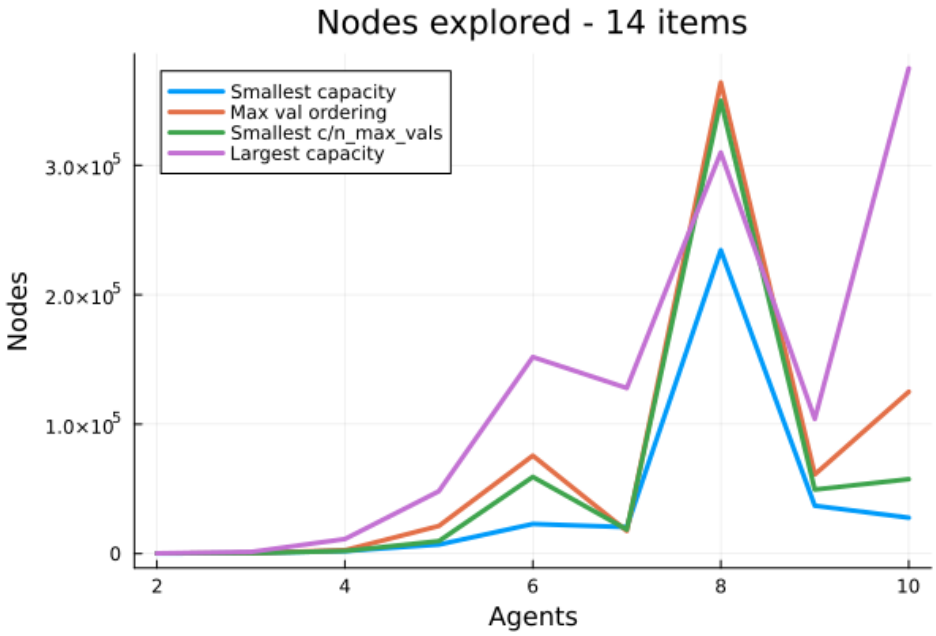



Figure 32

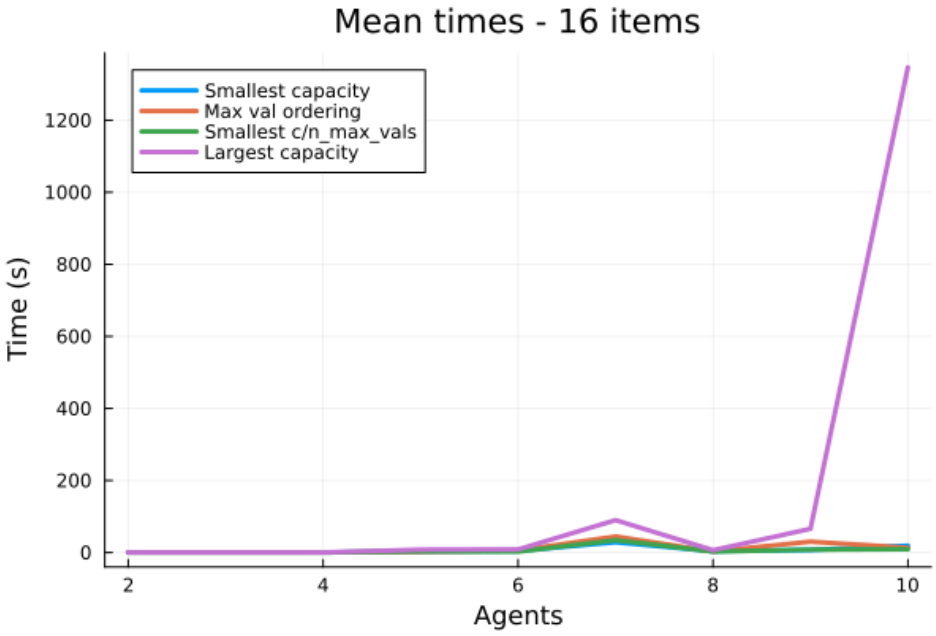



Figure 33

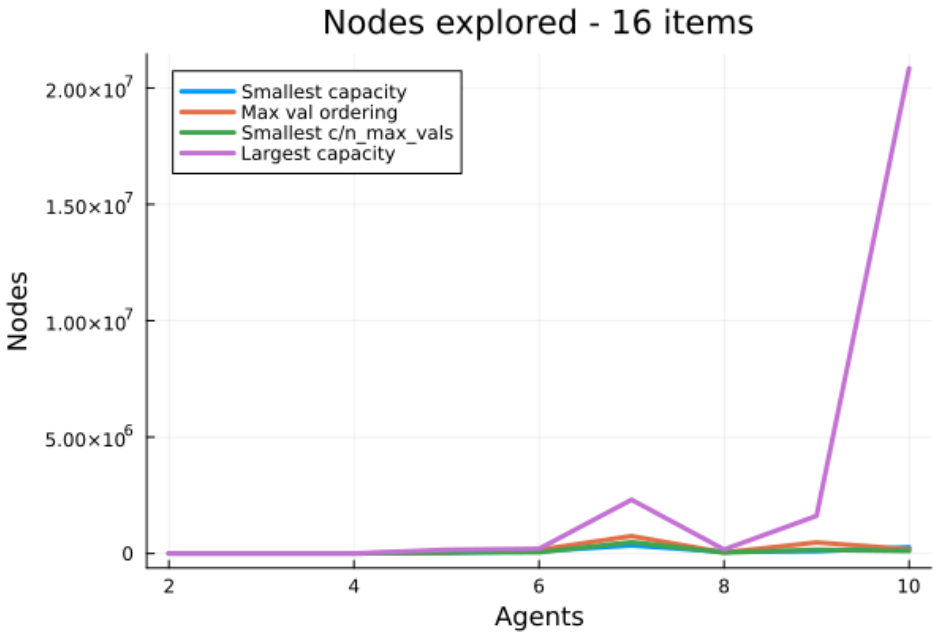



Figure 34

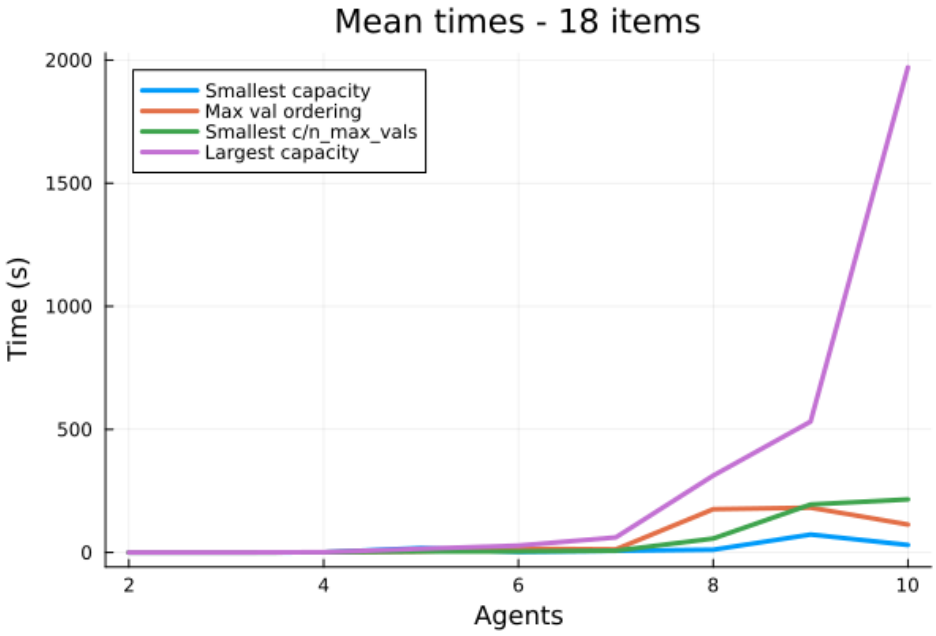



Figure 35

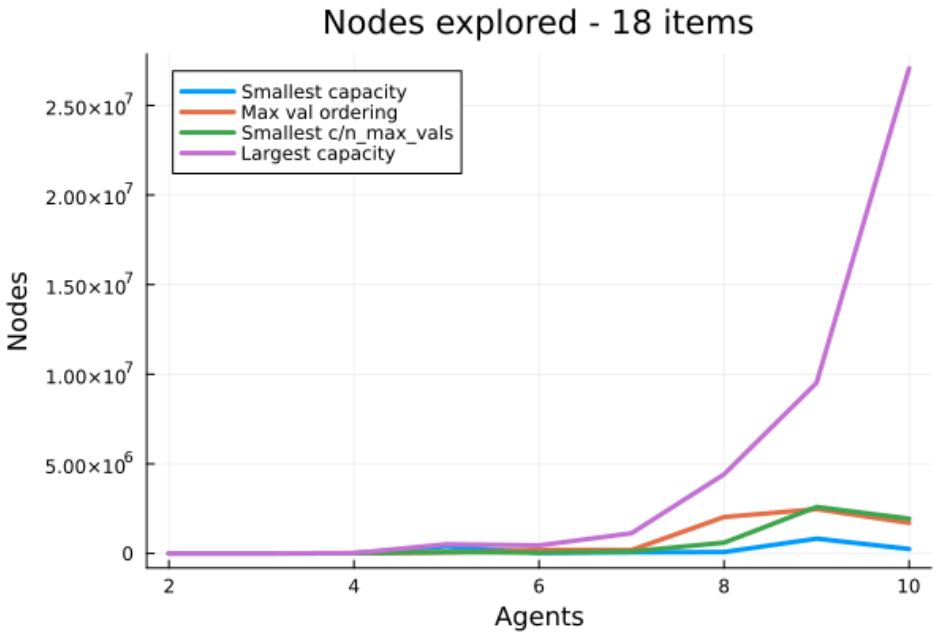



Figure 36

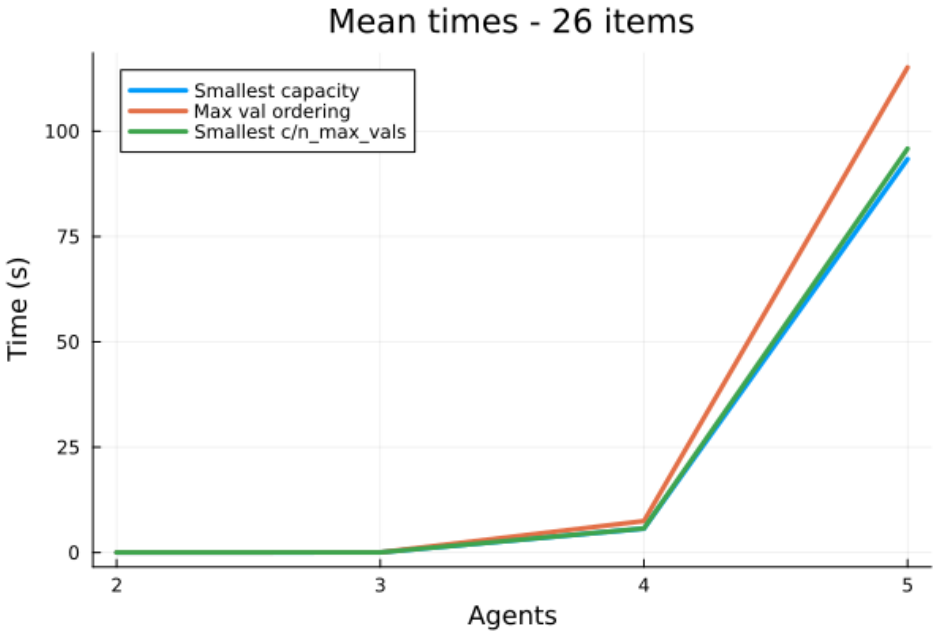



Figure 37

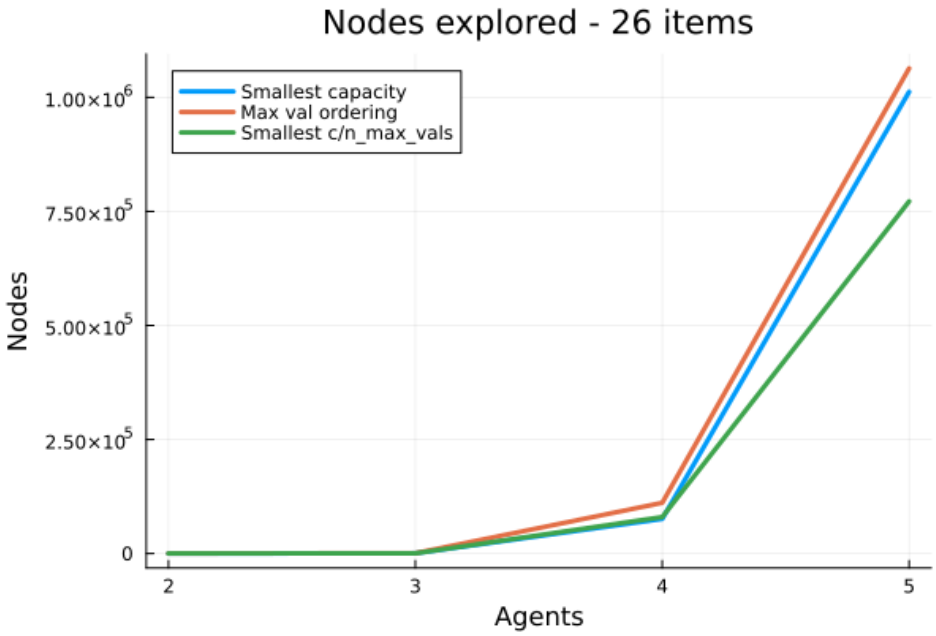

Nodes explored - 26 items
Smallest capacity
Max val ordering
Smallest c/n_max_vals
1.00×10^6
7.50×10^5
5.00×10^5
2.50×10^5
0
Nodes
2
3
4
5
Agents

Figure 38

## Value-ordering heuristics


Figure 39

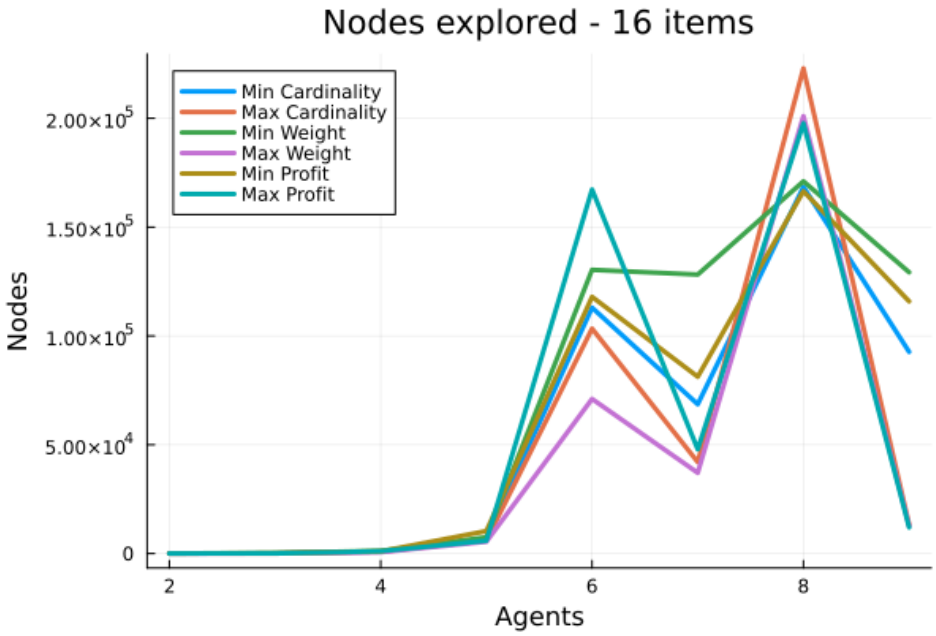



Figure 40

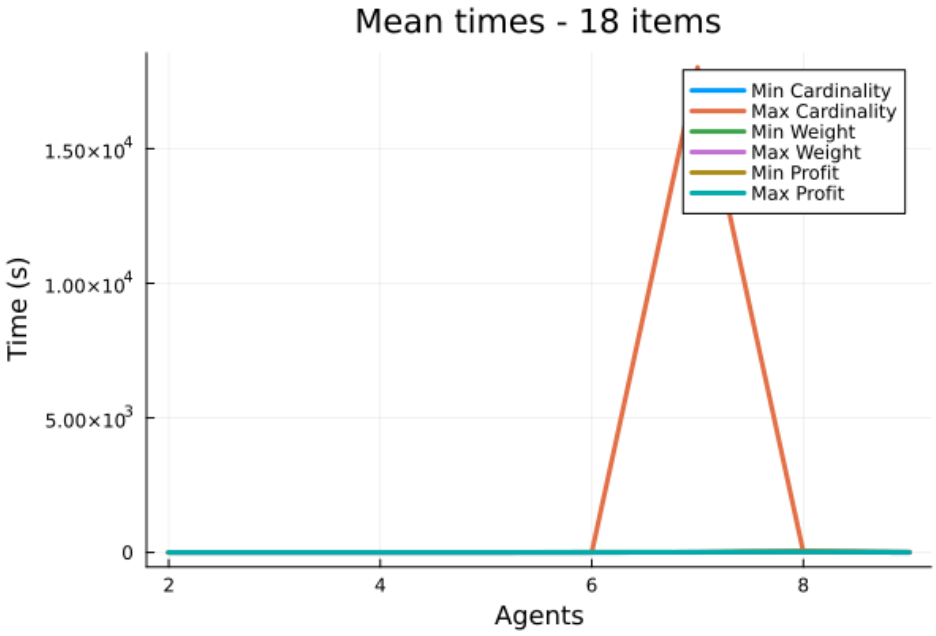



Figure 41

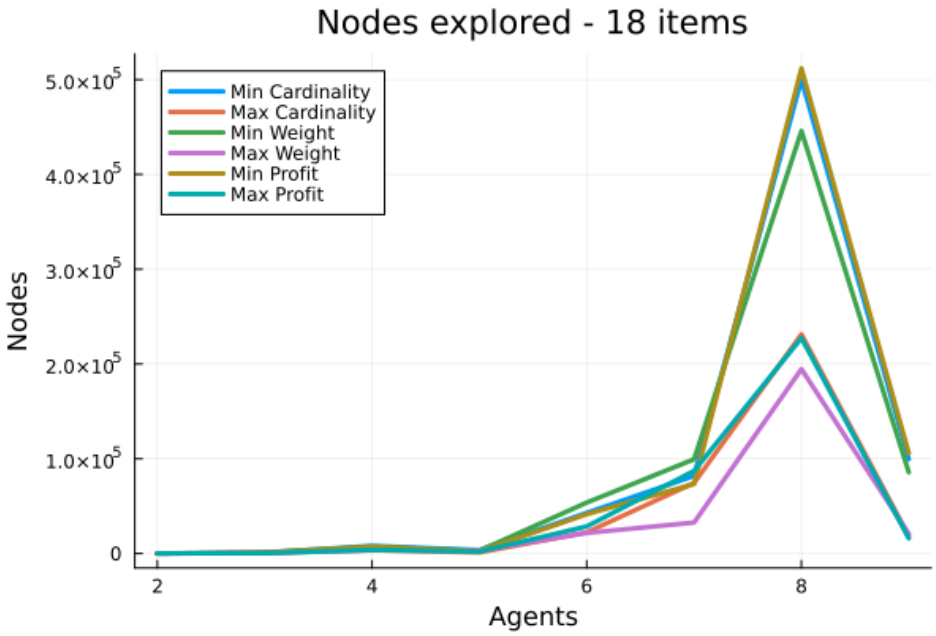



Figure 42

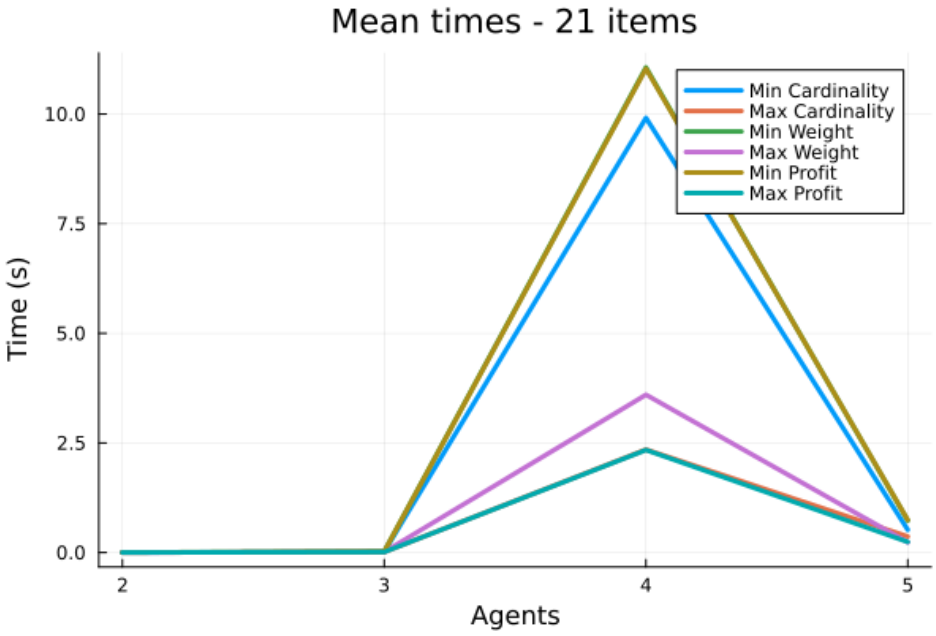



Figure 43

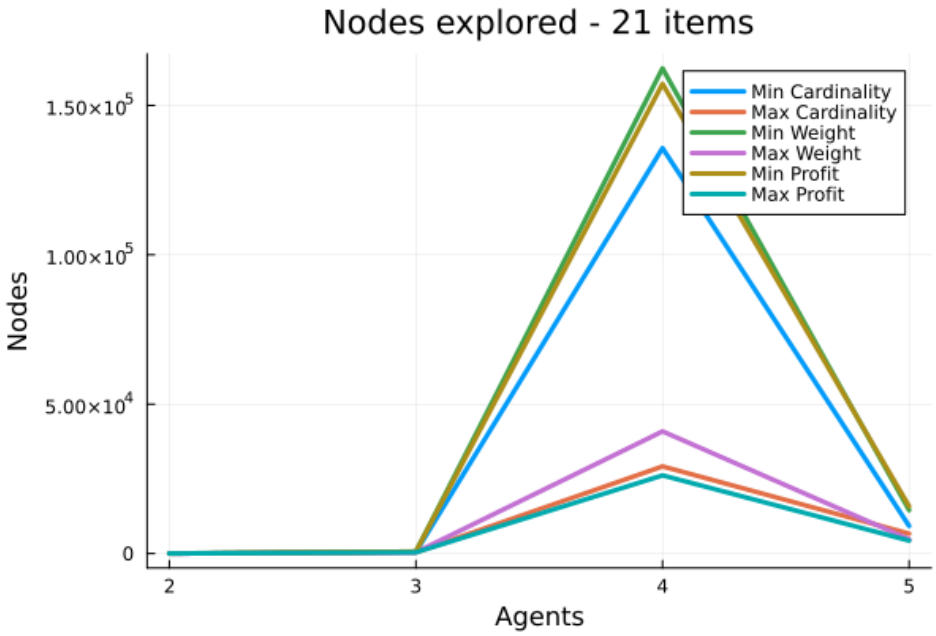



Figure 44

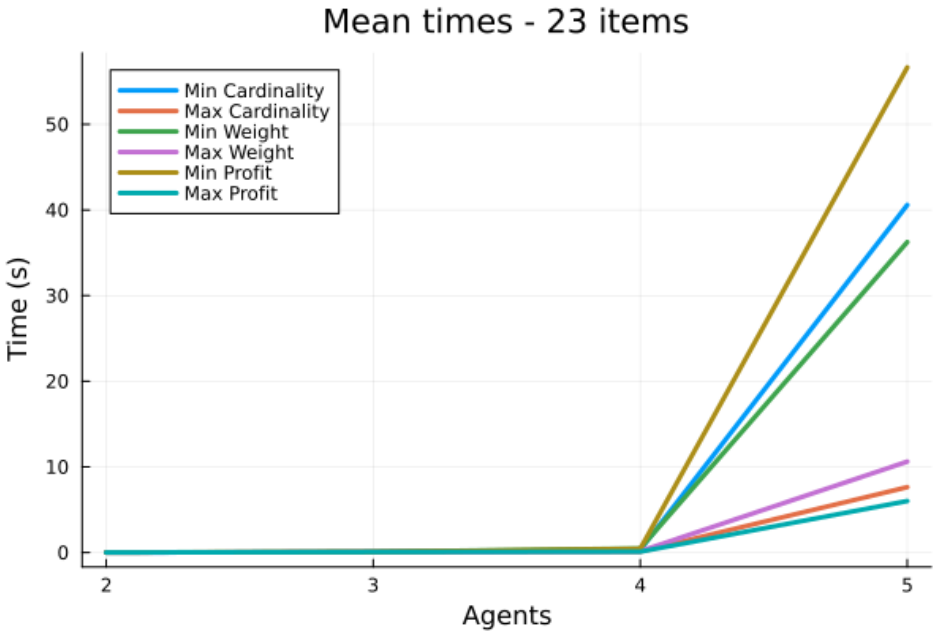



Figure 45

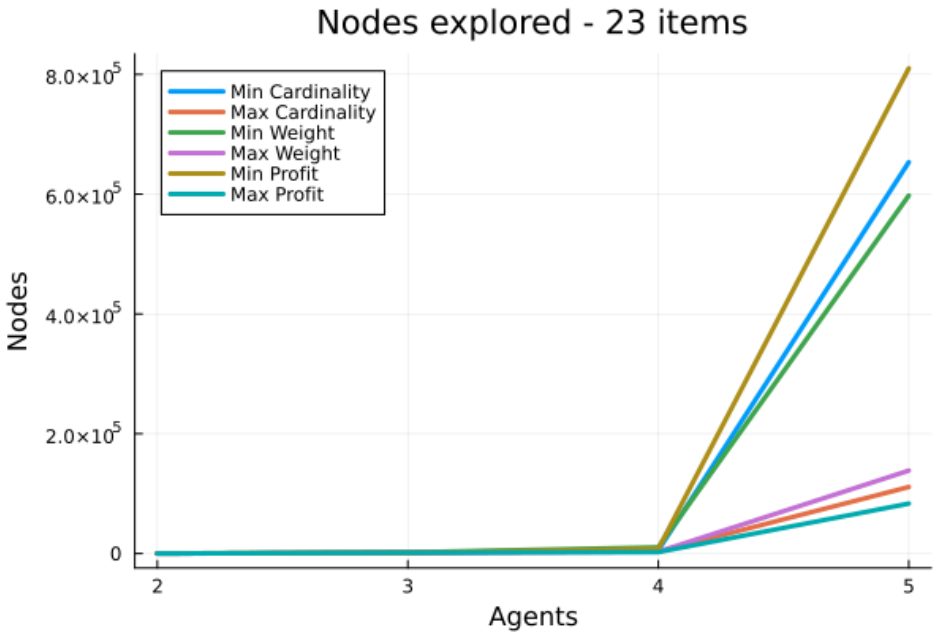



Figure 46

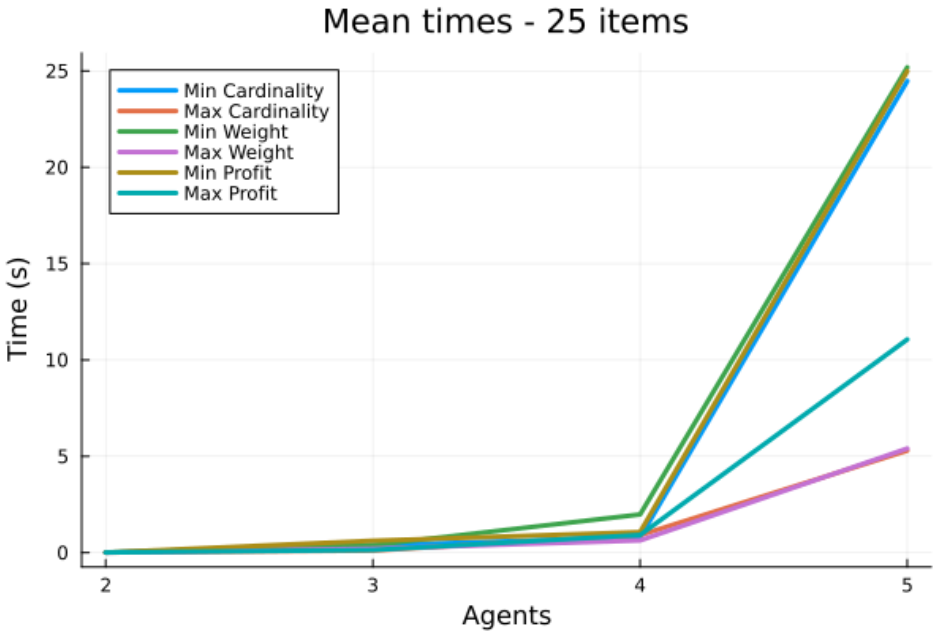



Figure 47

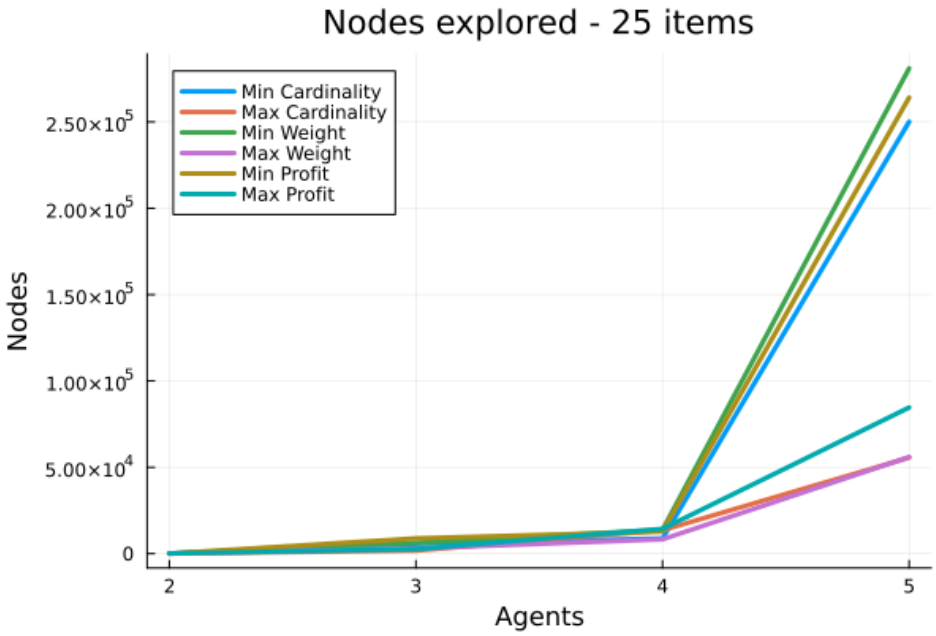



Figure 48

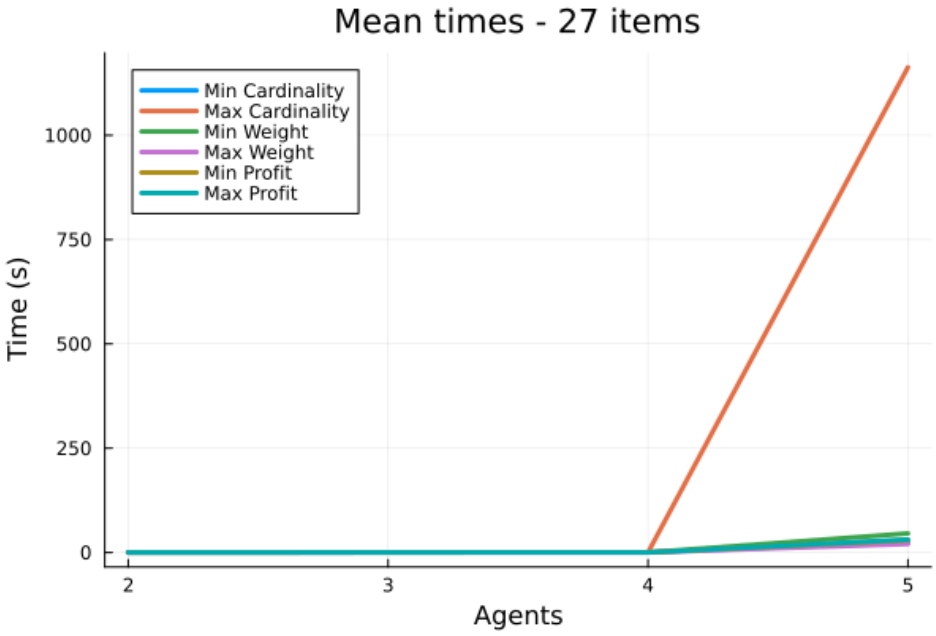



Figure 49

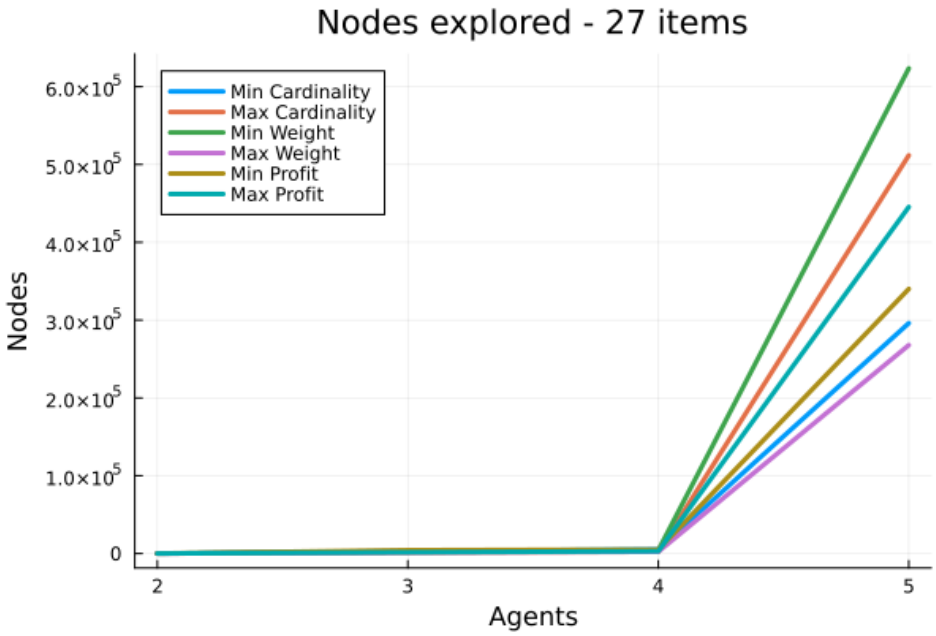
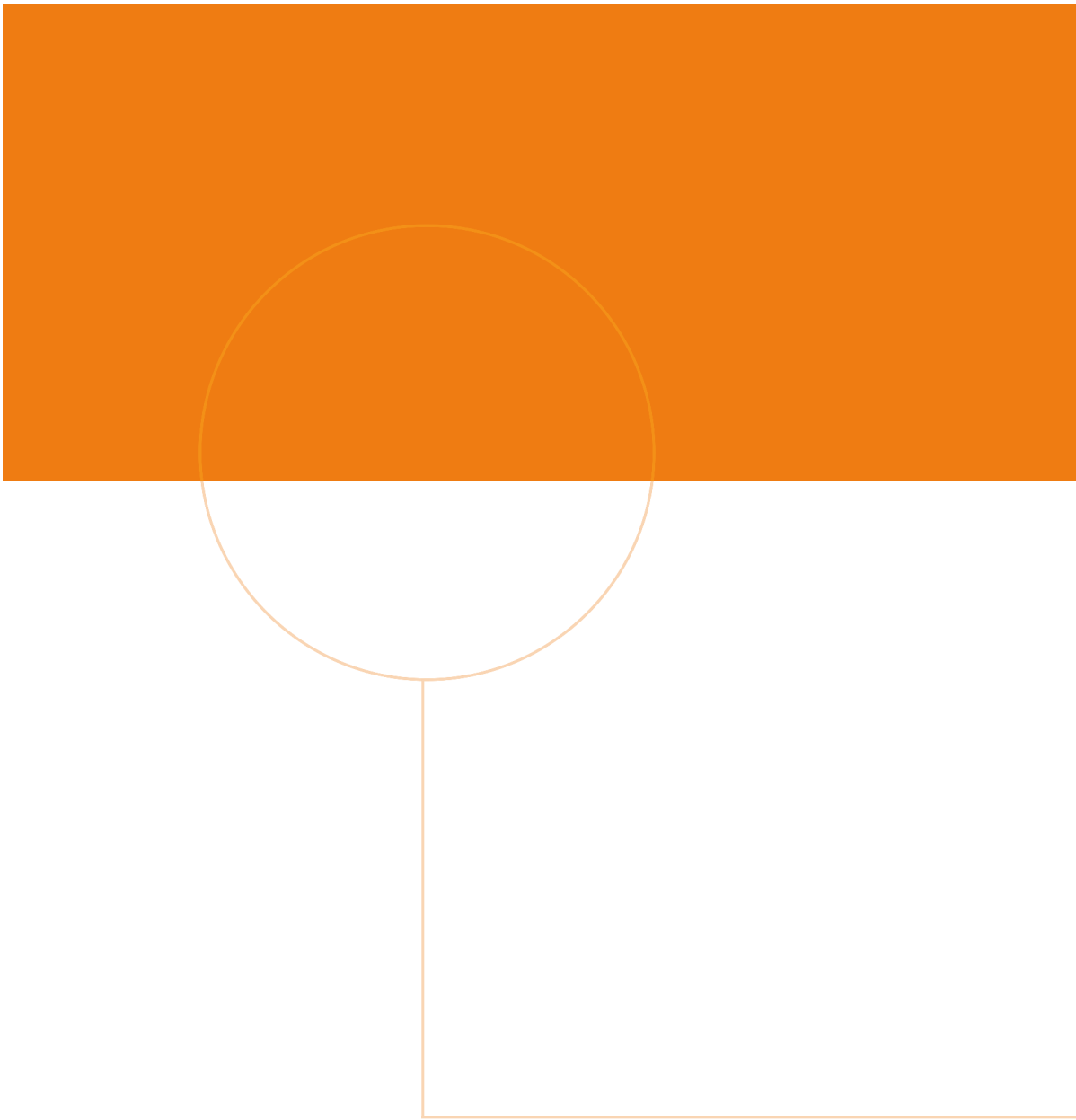

Nodes explored - 27 items
Min Cardinality
Max Cardinality
Min Weight
Max Weight
Min Profit
Max Profit
Nodes
6.0×10^5
5.0×10^5
4.0×10^5
3.0×10^5
2.0×10^5
1.0×10^5
0
2
3
4
5
Agents

Figure 50



NTNU

Norwegian University of
Science and Technology