Master's thesis

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

Andreas Aaberge Eide

Exploring Matroids in Fair Allocation

Building the Matroids.jl Library

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

NTNU
Norwegian University of
Science and Technology

Andreas Aaberge Eide

Exploring Matroids in Fair Allocation

Building the Matroids.jl Library

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

Abstract

This thesis explores the role of matroids in the context of fair allocation, for the purpose of building Matroids.jl, a proof-of-concept Julia library enabling the empirical study of matroidal fair allocation algorithms. To this end, functionality is given for the representation and random generation of graphic matroids. In addition, Knuth's algorithm for the erection of arbitrary matroids is implemented. Three recent fair allocation algorithms are studied and implemented. Also implemented is the functionality for evaluating an allocation against a variety of common fairness notions. The library is finally used to construct an experimental setup and the results of running the implemented algorithms on diverse matroids are described.

Sammendrag

Denne oppgaven utforsker rollen matroider spiller innen rettferdig fordeling, med mål for øyet å utvikle Matroids.jl, et Julia-bibliotek som muliggjør empirisk analyse av algoritmer for rettferdig fordeling som bruker matroider. Matroids.jl implementerer funksjonalitet for å representere og tilfeldig generere grafiske matroider. I tillegg implementeres Knuths algoritme for å tilfeldig sette opp vilkårlige matroider. Tre nylige algoritmer for rettferdig fordeling med matroierang-verdifunksjoner beskrives og deres behov til Matroids.jl analyseres. Matroids.jl implementerer også funksjonalitet for å evaluere en fordeling mot de fleste vanlige rettferdighetsmål. Til slutt benyttes biblioteket til å sette opp en serie eksperimenter som kjøres på de tre utvalgte algoritmene, og resultatene presenteres.

Acknowledgements

A big thanks is due to my stellar advisors, Magnus Hetland and Halvard Hummel, whose insightful feedback, patience, and passion for the subject has been truly above and beyond, and has made this past year enjoyable as well as highly educational. Kristina, Petter, and Vemund deserve special thanks for their invaluable moral support, and for listening to my incoherent ranting about matroids for the past few months. I would also like to applaud my flatmates Synne and Even for their general cleanliness in the kitchen area. Finally, a heartfelt thanks goes out to my good colleagues at Radio Revolt, without whom I would have completed this degree years earlier, with far better grades and in significantly better health.

Contents

Chapter 1

Introduction

Imagine you are a dean at a large university, responsible for allocating seats in courses to students for the coming semester. Each student has applied for some subset of the courses available, and each course has a limit on the number of students it can accept. In addition, there are scheduling conflicts between courses, and the students have hard limits on the number of courses they can take in a semester[1]. As a good dean, you wish to allocate fairly, ensuring that the students are happy with their courses and do not feel disfavored compared to the other students. At the same time, you strive for efficiency, so that every student gets enough courses to make the expected progress on their grade. How would you go about solving this problem? Over lunch, you discuss your quandary with a colleague from the computer science department who immediately assuages your fears by informing you that your situation is in fact an instance of indivisible fair allocation with matroid-rank valuations, a well-studied problem for which several algorithms exist! Eager to learn more, you ask your colleague to explain her cryptic remark.

What is fair allocation?

Fair allocation is the problem of fairly partitioning a set of resources (in this case, the courses) among agents (the students) with different preferences or valuations

[1] This example is due to Benabbou et al [1].

over these resources. This has been a hot topic of interest since antiquity (a 2000-year old allocation strategy can be found in the Talmud [2]), and remains so today. The mathematical study of fair allocation started with a seminal work by Steinhaus in 1948 [3], and for decades the focus was largely on the *divisible* case, in which the resources can be divided into arbitrary small pieces. In the divisible case, fair allocations always exist, and they can be computed efficiently [4]. In the dean's scenario, however, the course seats are *indivisible goods*. A fair allocation of indivisible goods is, depending on the measure of fairness, not always achievable; consider for example allocating a course with one seat between two students who both applied for it – there is no way of allocating the seat without one agent being unhappy.

Generally speaking, an allocation is measured against two justice criteria: *fairness* and *efficiency*. Fairness relates to the degree to which agents preceive the allocation as favoring other agents over themselves. One common way to describe the *fairness* of an allocation is with the concept of *envy-freeness*. Envy is defined as the degree to which an agent values another agent's received bundle of resources higher than their own. An allocation is envy-free if no agent envies another agent. In the trivial example above, the only envy-free allocation is the one in which no student receives the seat; while this is technically speaking fair, it is highly inefficient. Efficiency deals with maximizing some notion of resource utilization, or, equivalently, reducing waste. The perfectly fair allocation in which no one receives anything is rarely desirable for reasons of efficiency. Conversely, while the allocation in which one agent receives everything might be highly efficient in terms of the total sum of bundle values, it is obviously unfair. The task of the fair allocation algorithm, then, is to find some balance between these criteria.

How do matroids enter into this?

What the colleague from the computer science department noticed about the dean's problem, was that it was well-structured, in fact it is a textbook example of *matroid-rank valuations* in practice. Matroid rank functions (MRFs) are are a class of functions with properties that make them both easy to reason about and practically applicable in a setting such as fair allocation, and can equivalently be referred to as *binary submodular functions*. A submodular function is a set function that obeys the law of *diminishing returns* – as the size of the input set increases, the marginal value of a single additional good decreases. MRFs are the class of submodular functions with *binary marginals*, meaning that the

value of any single good is either 0 or 1.

In practice, these properties make MRFs a compelling framework for modelling user preferences in a setting such as the dean's allocation scenario. The binary marginals reflect a student's willingness (value of 1) or unwillingness (value of 0) to enroll in a course. The diminishing returns property allow us to implement what are known in economics terms as *supplementary goods* and *fixed demand*. A student might be interested in two similar courses, but not wish to enroll in both, so given one, the marginal value of the other drops to 0 (the courses are supplementary goods). In addition, a student needs only at most one seat per course, so the many available seats for the same course are also supplementary goods. A student has limited time and energy, and so for each course seat received, the marginal value of the other courses can only decrease – after some threshold is reached in the number of enrolled courses, all remaining courses have value 0 (there is a fixed demand for courses).

Matroids are extensively studied mathematical structures that generalize concepts from a variety of different fields. A number of interesting algorithms have been developed for fair allocation with matroid-rank valuations [1, 5–8] that make use of deep results from matroid theory in their analysis, and deliver well on a range of justice criteria which might be computationally intractable to achieve under general valuations.

What does this thesis contribute?

Perhaps because matroids are so well-understood and pleasant to work with theoretically, there is a dearth of tooling available for generating and working with them programmatically. In an effort to complement the abundant theoretical toolkit provided by matroid theory, this thesis proposes Matroids.jl, a library for the Julia programming language [9], which extends the existing Allocations.jl library [10] with the functionality required to enable the empirical study of matroidal fair allocation algorithms.

In this work, my primary contribution is the design and implementation of Matroids.jl as a practical tool that introduces new capabilities for handling matroids in the context of fair allocation. This thesis details the practical considerations that went into the development of the library, with code excerpts scattered throughout[2]. In addition, I present some experimental results obtained by implementing three select algorithms. While these results might present somewhat novel insights, I feel the need to underscore that they are given mainly as

[2]The full source code can be found at https://github.com/aaaeide/Matroids.jl

illustrative examples of Matroids.jl's utility and are not the central focus of this thesis.

This thesis describes the work that has been done to design and build a working, proof-of-concept version of Matroids.jl. It is structured as follows. In the next chapter, I establish the concepts from matroid theory and fair allocation necessary to understand the rest of the thesis. In Chapter 3, I describe the design and implementation of the Matroids.jl API, which includes various classic matroid algorithms that have found use in fair allocation algorithms. In Chapter 5, I show how this API can be used to implement Viswanathan and Zick's Yankee Swap algorithm [8] and some other algorithms for matroid-rank-valued fair allocation. In Chapter 4, I show how Matroids.jl implements the random generation of a range of matroid types. Of particular interest here is Knuth's classic method for generating arbitrary matroids [11], the successful implementation of which was a significant sub-goal of the project. In Chapter 5.4, I provide some experimental results for the algorithms over different matroid types. Finally, in Chapter 6, I give a summary discussion on the limitations of Matroids.jl and suggests a few possible avenues of future work.

Chapter 2

Preliminaries

> For simplicity, we also assume that every point in a geometry is a closed set. Without this additional assumption, the resulting structure is often described by the ineffably cacaphonic term "matroid", which we prefer to avoid in favor of the term "pregeometry".
>
> Gian-Carlo Rota [12]

Matroids were first introduced by Hassler Whitney in 1935 [13], in a seminal paper where he described two axioms for independence in the columns of a matrix, and defined any system obeying these axioms to be a “matroid” (which unfortunately for Rota is the term that has stuck). Whitney’s key insight was that this abstraction of “independence” is applicable to both matrices and graphs. Matroids have also received attention from researchers in fair allocation, as their properties make them useful for modeling user preferences. We have already seen this with the course allocation problem described in the previous chapter; other use cases include the assignment of kindergarten slots or public housing estates among people of different ethnicities [7].

2.1 Fair allocation

To ease readability, I abuse notation a bit and use $S + g$ and $S - g$ to refer to $S \cup \{g\}$ and $S \setminus \{g\}$, respectively.

An instance of a fair allocation problem consists of a set of agents $N = \{1, 2, \ldots, n\}$ and a set of m goods $E = \{g_1, g_2, \ldots, g_m\}$. Each agent has a valuation function $v_i : 2^E \to \mathbb{R}^+$; $v_i(S)$ is the value agent i ascribes to the bundle of goods S. The marginal value of agent i for the good g, given that she already owns the bundle S, is given by $\Delta_i(S, g) := v_i(S+g) - v_i(S)$. Throughout most of this thesis, we assume that v_i is a matroid rank function, or, equivalently, a binary submodular function. To formalize the description given in Chapter 1, this means that

(a) $v_i(\emptyset) = 0$,

(b) v_i has binary marginals: $\Delta_i(A, g) \in \{0, 1\}$ for every $A \subset E$ and $g \in E$,

(c) v_i is submodular: for every $A \subseteq B \subseteq E$ and $g \in E \setminus B$, we have that $\Delta_i(A, g) \geq \Delta_i(B, g)$.

Any function v_i adhering to these properties is a valid characterization of exactly one matroid in terms of its rank function [14]. There are many other ways to characterize matroids, some of which are given in Section 2.2.

Throughout this thesis, I will use the terms *allocation* and *partition* somewhat interchangeably. In set theory, an n-partition of a set is a grouping of its elements into n subsets, such that each element occurs in exactly one subset. It is often required that the subsets be non-gmpty—when working with matroids this requirement is usually omitted since the empty set is independent (see Section 2.2). In a fair allocation instance with n agents, an allocation is an n-partition of E.

The output of an algorithm for fair allocation is an allocation of the goods to the agents. An allocation A is an n-partition of E, $A = (A_1, A_2, \ldots, A_n)$, where each A_i is the bundle of goods allocated to agent i. Sharing is not allowed, so we require that $A_i \cap A_j = \emptyset$ for all $i \neq j$. We say that an allocation is *clean* (also known as *non-redundant* in the literature) if no agent has received any good they value at 0. An allocation is *complete* if all goods are allocated, if not it is *partially*. It might not be possible to guarantee both cleanness and completeness; for instance in a case where a good is 0-valued by all agents.

2.1.1 Envy-freeness

We are interested in producing *fair* allocations. One of the most popular notions of fairness in the literature is envy-freeness (EF), which states that no agent should prefer another agent's bundle over her own (in fair allocation, an agent is *envious* of another agent if she prefers that agent's bundle). An allocation A is EF if for all agents $i, j \in N$,

$$v_i(A_i) \geq v_i(A_j). \tag{EF}$$

Because, as mentioned in the introduction, EF is not always achievable when the goods are indivisible, the literature has focused on relaxations thereof. The most prominent such relaxation, which can be guaranteed, is *envy-freeness up to one good* (EF1) [15], which allows for the envy of up to the value of one (highest-valued) good. This is equivalent to saying that any envy can be eliminated by dropping one good from the envied bundle. A is an EF1 allocation if for all agents $i, j \in N$ where $|A_j| > 0$, there exists a $g \in A_j$ such that

$$v_i(A_i) \geq v_i(A_j - g). \tag{EF1}$$

Envy-freeness up to any good (EFX) is an even stronger version of EF. While EF1 allows that agent i envies agent j up to their highest valued good, EFX requires that the envy can be removed by dropping agent j's least valued good. There are two slightly different definitions of EFX in use in the literature. I follow the naming scheme used by Benabbou et al. [7] and refer to these as EFX_+ and EFX_0. Caragiannis et al. [16] requires that this least valued good be positively valued. We call this fairness objective EFX_+. A is an EFX_+ allocation if for all agents $i, j \in N$,

$$v_i(A_i) \geq v_i(A_j - g),\ \forall g \in A_j \text{ st. } v_i(A_j - g) < v_i(A_j). \tag{EFX$_+$}$$

Plaut and Roughgarden [17], on the other hand, allow for 0-valued goods in the envy check – we call this version EFX_0. It is stronger requirement than EFX_+. A is an EFX_0 allocation if for all agents $i, j \in N$,

$$v_i(A_i) \geq v_i(A_j - g),\ \forall g \in A_j. \tag{EFX$_0$}$$

In the general, additive case, the existence of EFX_0 allocations is an open question for instances with $n \geq 4$ [4].

2.1.2 Proportionality

Proportionality is a fairness objective that is fundamentally different from envy-freeness, in that it checks each bundle value against some threshold, instead of comparing bundle values against each other. An allocation A is proportional (PROP) if each agent $i \in N$ receives at least her proportional share PROP_i, which is the $\frac{1}{n}$ fraction of the value she puts on the whole set of goods, i.e.,

$$v_i(A_i) \geq \text{PROP}_i := \frac{v_i(E)}{n}. \tag{PROP}$$

Proportionality might not be achievable in the indivisible case (again, consider two agents and one positively valued good), and so relaxations in the same vein as EF1 and EFX have been introduced – these are called PROP1 and PROPX [4]. An allocation is PROP1 if there for each agent $i \in N$ exists some good $g \in E \setminus A_i$ that, if given to i, would ensure that agent i received her proportional share; that is,

$$\exists g \in E \setminus A_i \text{ st. } v_i(A_i + g) \geq \text{PROP}_i \tag{PROP1}$$

PROPX is a stronger fairness objective than PROP1, and has, as in the case of EFX, two slightly different definitions in the literature. I follow the naming scheme established for EFX above, and refer to these as PROPX_+ and PROPX_0. The logic is similar to that of EFX. An allocation is PROPX_0 if each agent can achieve her proportional share by receiving one additional, least-valued good from the goods not allocated to her. This good might be zero-valued.

$$\min_{g \in E \setminus A_i} v_i(A_i + g) \geq \text{PROP}_i \tag{PROPX$_0$}$$

If we disallow zero-valued items, we arrive at the slightly weaker criteria PROPX_+, given by:

$$\min_{g \in E \setminus A_i, \Delta_i(A_i, g) > 0} v_i(A_i + g) \geq \text{PROP}_i \tag{PROPX$_+$}$$

Maximin share fairness

Budish introduces a relaxation of proportionality known as *maximin share fairness* [18], in which the threshold for each agent i is her *maximin share* (MMS). The MMS of agent i, denoted by μ_i, is defined as the maximum value she could receive if she partitioned E among all agents and then picked the worst bundle.

Let $\Pi_n(E)$ be the family of all possible allocations of the goods in E to the agents in N. Then,

$$\mu_i := \max_{A \in \Pi_n(E)} \min_{A_j \in A} v_i(A_j).$$

An allocation is MMS-fair if all agents receive at least as much as their maximin share:

$$v_i(A_i) \geq \mu_i, \ \forall i \in N. \tag{MMS}$$

In the general, additive case, MMS-fair allocations do not always exist, and even computing the MMS of an agent is an NP-hard problem [4]. In a setting with matroid-rank valuations, however, Barman and Verma showed that MMS-fair allocations always exist, and can be computed in polynomial time [6].

2.1.3 Efficiency

Fairness is usually coupled with some efficiency criterion, to prevent the perfectly fair solution in which the whole set of goods is thrown away. The efficiency of an allocation can be measured with some *welfare function* on the values of the agents. There are three welfare functions commonly used in the literature:

1. **Egalitarian social welfare (ESW):** The ESW of an allocation A is given by the minimum value of an agent. $\text{ESW}(A) = \min_{i \in N} v_i(A_i)$.
2. **Utilitarian social welfare (USW):** The USW of an allocation is the total value received by all agents. $\text{USW}(A) = \sum_{i \in N} v_i(A_i)$.
3. **Nash welfare (NW):** The Nash welfare of an allocation is a compromise between the utilitarian and egalitarian approaches, given by the product of agent utilities. $\text{NW}(A) = \prod_{i \in N} v_i(A_i)$.

Allocations that maximize one of these welfare functions are referred to as MAX-ESW, MAX-USW and MNW, respectively.

Leximin

An obvious drawback of the egalitarian approach is that, in the situation where there are multiple possible allocations that maximize the minimum bundle value, it is indifferent to which of these bundles it prefers. Consider for instance two possible allocations of goods to three agents, where their bundle values are given as (A,B,C):

- Allocation 1: $(5, 7, 10)$

- Allocation 2: $(5, 8, 9)$

Both of these allocations are MAX-ESW. A stricter version of the egalitarian rule is *leximin*: an allocation is leximin if it maximizes the smallest value; subject to that, it maximizes the second-smallest value; subject to that, it maximizes the next-smallest value, and so on. The leximin rule prefers Allocation 2 over Allocation 1.

Pareto Optimality

Another widespread notion of allocation efficiency is that of *Pareto optimality*. An allocation A is said to *Pareto dominate* another allocation B if (1) $\forall i \in N,\ v_i(A_i) \geq v_i(B_i)$ (every agent is at least as happy with A as with B), and (2) $\exists j \in N,\ v_i(A_j) > v_j(B_j)$ (some agent j is strictly happier with A than with B). An allocation is Pareto optimal (PO) if it is not Pareto dominated by any other allocation.

Pareto optimality is a weaker efficiency criterion than MAX-USW. To see why, assume A is a MAX-USW allocation that is not PO. Since it is not PO, there exists another allocation A' with $v_i(A'_i) \geq v_(A_i)$ for all agents i, and $v_j(A'_j) > v_j(A_j)$ for some agent j. Then, $\text{USW}(A') = \sum_{i \in N} v_i(A'_i) > \sum_{i \in N} v_i(A_i) = \text{USW}(A)$, which contradicts that A is MAX-USW. In Section 2.2.4, I show how the matroid union operation allows us to efficiently find MAX-USW, and therefore PO, allocations when the valuations are matroid rank functions.

2.2 Matroid theory

If a mathematical structure can be defined or axiomatized in multiple different, but not obviously equivalent, ways, the different definitions or axiomatizations of that structure make up a cryptomorphism. The many obtusely equivalent definitions of a matroid are a classic example of cryptomorphism, and belie the fact that the matroid is a generalization of concepts in many, seemingly disparate areas of mathematics. As a result, the terms used in matroid theory are borrowed from analogous concepts in both graph theory and linear algebra.

The most common way to characterize a matroid is as an *independence system*. An independence system is a pair $(E, \mathcal{I})$, where E is the ground set of elements, $E \neq \emptyset$, and $\mathcal{I}$ is the set of independent sets, $\mathcal{I} \subseteq 2^E$. The *dependent sets* of a matroid are $2^E \setminus \mathcal{I}$. A matroid is an independence system with the following properties [13]:

(1) If $S \subseteq T$ and $T \in \mathcal{I}$, then $S \in \mathcal{I}$.

(2) If $S, T \in \mathcal{I}$ and $|S| > |T|$, then there exists $g \in S \setminus T$ such that $S + g \in \mathcal{I}$.

(2') If $S \subseteq E$, then the maximal independent subsets of S are equal in size.

Property (1) is called the *hereditary property* and (2) the *exchange property*. Properties (2) and (2') are equivalent. To see that (2) $\implies$ (2'), consider two maximal subsets of S. If they differ in size, (2) tells us that there are elements we can add from one to the other until they have equal cardinality. We get (2') $\implies$ (2) by considering $S = A \cup B$. Since $|A| > |B|$, they cannot both be maximal, and some $e \in A \setminus B$ can be added to B to obtain another independent set.

When $S = E$ (i.e., the entire ground set), (2') gives us that the maximal independent sets of a matroid are all of the same size. A maximal independent subset of E is known as a *basis*. The size of the bases is the *rank* of the matroid as a whole. A matroid can be exactly determined by $\mathcal{B}$, its collection of bases, since a set is independent if and only if it is contained in a basis (this follows from (1)). A theorem by Whitney [13] gives the axiom system characterizing a collection of bases of a matroid:

1. No proper subset of a basis is a basis.
2. If $B, B' \in \mathcal{B}$ and $g \in B$, then for some $g' \in B'$, $B - g + g' \in \mathcal{B}$.

The rank function of a matroid is a function $v : 2^E \to \mathbb{Z}^+$ which, given a subset $B \subseteq E$, returns the size of the largest independent set contained in B. That is,

$$v(B) = \max_{A \subseteq B, A \in \mathcal{I}} |A|.$$

The properties of the matroid rank function are given in Section 2.1. Any function adhering to these properties specify exactly one matroid.

2.2.1 The uniform matroid

The uniform matroid U_n^r is the matroid over n elements where the independent sets are exactly the sets of cardinality at most r. The free matroid $U_n^n = (E, 2^E)$ is a special case of the uniform matroid and is the simplest, biggest and least interesting type of matroid, being the trivial case in which every subset of E is an independent set. While not very exciting matroids in and of themselves, they are the easiest matroid to reason about and will show up from time to time in the examples throughout this thesis.

2.2.2 The graphic matroid

Different types of matroids exist, arising from different sources of "independence"; one well-known subclass of matroids, arising from notions of independence in graphs, is the class of *graphic matroids*.

A *tree* is a connected acyclic graph, and a *forest* is a disconnected graph consisting of some number of trees. A *spanning tree* of G is a subgraph with a unique simple path between all pairs of vertices of G. A *spanning forest* of G is a collection of spanning trees, one for each component. A graph will have some number of different spanning trees. Figure 2.1 shows two spanning trees of the same graphs (or alternatively, a spanning forest over one graph with two components).

Given a graph $G = (V, E)$, let $\mathcal{I} \subseteq 2^E$ be the family of subsets of the edges E such that, for each $I \in \mathcal{I}$, (V, I) is a forest. It is a classic result of matroid theory that $\mathfrak{M} = (E, \mathcal{I})$ (the ground set of the matroid being the edges of the graph) is a matroid [14, p. 657]. To understand how, we will show that it adhers to axioms (1) and (2'), as given in above. By investigating the highlighted spanning trees in Figure 2.1, it is easy to convince oneself that all subsets of a spanning tree are trees, since no subset of an acyclic set of edges will contain a cycle. Thus axiom (1) – the hereditary property – holds.

To see that (2') holds, consider the set of bases of the matroid, $\mathcal{B} \subseteq \mathcal{I}$. Figure 2.1 shows two bases of the matroid described by the graph. By definition, each basis $B \in \mathcal{B}$ is a maximal forest over G. Since a spanning tree of a graph with n nodes must needs have $n - 1$ edges (I recommend drawing trees and counting their edges until one is convinced that this must be the case), we have $|B| = |V| - k$, where k is the number of components of G. This is the same for every $B \in \mathcal{B}$, which proves property (2'). Any matroid given by a graph G, denoted by $\mathfrak{M}(G)$, is called a graphic matroid.

2.2.3 Other characterizations of a matroid

We have already seen how to characterize a matroid using its rank function, its independent sets, or implicitly in terms of a graph, but the other properties of a matroid have their own axiom systems that can equivalently be used to characterize a matroid.

Characterization via circuits. A *circuit* is a minimal dependent set of a matroid – it is an independent set plus one redundant element. Equivalently,

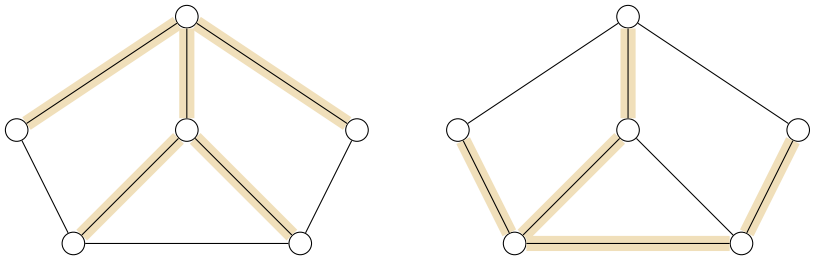


Figure 2.1: Two spanning trees of a graph with 8 edges.

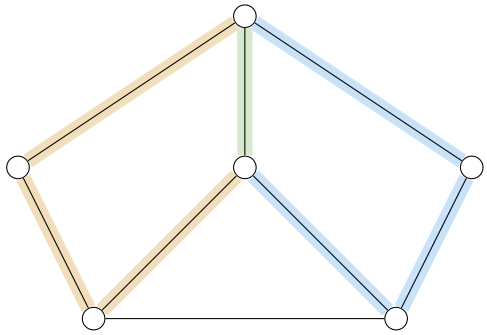


Figure 2.2: Two overlapping circuits on a graph.

the collection of circuits of a matroid is given by

$$\mathcal{C} = \big\{C : |C| = v(C) + 1, C \subseteq E\big\}.$$

A set is independent if and only if it contains no circuit [14], and so a matroid is uniquely determined by the collection of its circuits. The following conditions characterize $\mathcal{C}$ [13]:

(1) No proper subset of a circuit is a circuit.

(2) If $C, C' \in \mathcal{C}$, $x \in C \cap C'$ and $y \in C \setminus C'$, then $C \cup C'$ contains a circuit containing y but not x.

These properties are easily grasped by studying an example. In Figure 2.2, we see two circuits of a graph, highlighted in yellow and blue, overlapping at the edge highlighted in green. Obviously, no circuit in this figure contains a circuit.

We can also verify that the union of the yellow edge set and the blue edge set, minus the green edge, is in fact a third, bigger circuit, as promised by property (2).

Characterization via closed sets. We also need to establish the concept of the *closed sets* (sometimes referred to as *flats* [14]) of a matroid. A closed set is a set whose cardinality is maximal for its rank. Equivalently to the definition given above, we can define a matroid as $\mathfrak{M} = (E, \mathcal{F})$, where $\mathcal{F}$ is the set of closed sets of $\mathfrak{M}$, satisfying the following properties [11]:

1. The set of all elements is closed: $E \in \mathcal{F}$
2. The intersection of two closed sets is a closed set: If $A, B \in \mathcal{F}$, then $A \cap B \in \mathcal{F}$
3. If $A \in \mathcal{F}$ and $a, b \in E \setminus A$, then b is a member of all sets in $\mathcal{F}$ containing $A \cup \{a\}$ if and only if a is a member of all sets in $\mathcal{F}$ containing $A \cup \{b\}$

The *closure function* is the function $cl : 2^E \to 2^E$, such that

$$cl(S) = \big\{x \in E : v(S) = v(S \cup \{x\})\big\}.$$

That is to say, the closure function, when given a set $S \subseteq E$, returns the set of elements in $x \in E$ such that x can be added to S with no increase in rank. It returns the closed set of the same rank as S, that contains S. The *nullity* of a subset S is the difference $|S| - v(S)$, i.e., the number of elements that must be removed from S to obtain an independent set.

Figure 2.3 shows an independent (acyclic) set S of edges in highlighted yellow, along with its closure $cl(S)$ in blue. The closure is the set of edges e such that $v(S + g) = v(S)$, or, equivalently, such that the spanning tree of $S + g$ has the same size as that of S.

2.2.4 Matroid union

The matroid union is an operation that allows us to produce a new matroid by combining the independent sets of a collection of existing matroids. This is a powerful tool, as it allows us to reason about independence across multiple matroids, and has found several practical applications within fair allocation [6–8].

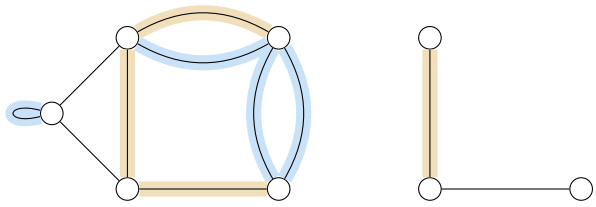


Figure 2.3: An independent subset of edges in yellow, with its closure in blue.

Given n matroids $\mathfrak{M}_i = (E, \mathcal{I}_i)$, their union is given, somewhat obtusely, by

$$\widehat{\mathfrak{M}} = (E, \widehat{\mathcal{I}}) = (E, \{I_1 \cup \ldots \cup I_n : I_i \in \mathcal{I}_i,\ \forall i \in N\}).$$

$\widehat{\mathfrak{M}}$ is in fact a matroid [14, Ch. 42], whose independent sets are the the subsets $S \subseteq E$ that allow an n-partition $S_1, \ldots, S_n$ such that $S_i \in \mathcal{I}_i$, for all $i \in N$. The following statements are equivalent for all n-partitions of the elements of E:

1. Each $S_i \in S$ is independent in $\mathfrak{M}_i$
2. $S = (S_1, \ldots, S_n) \in \mathcal{I}_i \times \cdots \times \mathcal{I}_n$
3. $\bigcup_{i=1}^{n} S_i \in \widehat{\mathcal{I}}$ is independent in $\widehat{\mathfrak{M}}$

In fair allocation jargon, if each $\mathfrak{M}_i$ is the matroid described by agent i's valuation function v_i, a set of goods S is independent in $\widehat{\mathfrak{M}}$ if and only if we can allocate it among the agents and produce utilitarian social welfare (i.e., total value) equal to $|S|$. This follows from the fact that if $\mathcal{S}$ is an n-partition of S such that $\mathcal{S} = (S_1, \ldots, S_n) \in \mathcal{I}_1 \times \cdots \times \mathcal{I}_n$, then $S_i \in \mathcal{I}_i$ for each i. When this is the case, we have $v_i(S_i) = |S_i|$, and so $\mathrm{SW}(S) = \sum_{i \in N} v_i(S_i) = \sum_{i \in N} |S_i| = |S|$.

Hence, each basis in $\widehat{\mathfrak{M}}$ corresponds to a clean (but not necessarily complete), MAX-USW allocation of the goods in E [6]. It is a classic result of Edmonds [19] that a basis in $\widehat{\mathfrak{M}}$ can be computed in polynomial time, using the *matroid union algorithm*. This is achieved by making use of two additional concepts, the *exchange graph* and *transfer paths*.

The exchange graph

Given n matroids $\mathfrak{M}_1 = (E, \mathcal{I}_1), \ldots, \mathfrak{M}_n = (E, \mathcal{I}_n)$, let A be a collection of n sets $A_1, \ldots, A_n$ such that (a) $A_i \cap A_j = \emptyset$ when $i \neq j$, and (b) for each i, $A_i \subseteq E$

and $A_i \in \mathcal{I}_i$. In the context of a matroid-rank-valued fair allocation problem, A is a clean allocation of the goods in E to the agents in N.

We follow the example of Schrijver [14] and define the exchange graph of A as the directed graph $D(A) = (E, x(A))$, where each node corresponds to a good in E, and the edges are given by

$$x(A) = \{(p, q) : p \in A_i, q \in E \setminus A_i, v_i(A_i - p + q) = v_i(A_i)\},$$

where v_i is the rank function of the matroid $\mathfrak{M}_i$. In other words, an edge exists between goods $p \in A_i$ and $q \in E \setminus A_i$ if we can replace p with q for no decrease in the rank of the set containing p. Intuitively, we can understand D as representing for each good p, which other good q the current owner of p can replace p with and be just as happy. This intuitive explanation lets us begin to see why the matroid union algorithm and the concept of the exchange graph has found widespread use in fair allocation with matroid-rank valuations; they allow us to model equitable transfers of goods between agents.

Transfer paths and path augmentation

Let $P = (g_1, \ldots, g_t)$ be a path in the exchange graph $D(A)$. The *transfer* of goods along P is the operation in which g_t is given to the agent who owns g_{t-1}, g_{t-1} is given to the agent who owns g_{t-2}, and so on until g_1 is discarded. This transfer is called *path augmentation*; we use the notation established by Viswanathan and Zick [8] and denote the bundle A_i after augmentation with the path P by $A_i \Lambda P$.

For some $i \in N$, we define $F_i = \{e \in E \setminus A_i : A_i + e \in \mathcal{I}_i\}$ as the set of elements whose addition to A_i yields another, larger independent set (remember that A is clean, so $A_i \in \mathcal{I}_i$ for each i). In a paper on the matroid union algorithm, Knuth [20] shows that, by augmenting along a shortest path $P = (g_1, \ldots, g_t)$ from F_i to A_j for some $j \in N - i$, we get

(a) $v_i(A_i \Lambda P + g_1) = v_i(A_i) + 1$,

(b) $v_k(A_k \Lambda P) = v_k(A_k),\ \forall k \in N - i - j$, and

(c) $v_j(A_j \Lambda P) = v_j(A_j) - 1$.

By greedily growing clean bundles in this manner, we can find a maximal independent set over $\widehat{\mathfrak{M}}$ in polynomial time [14]. Chapter 3 discusses Matroids.jl implementation of Knuth's matroid union algorithm. Viswanathan and Zick's Yankee Swap algorithm [8], discussed in Chapter 5, uses these concepts as well.

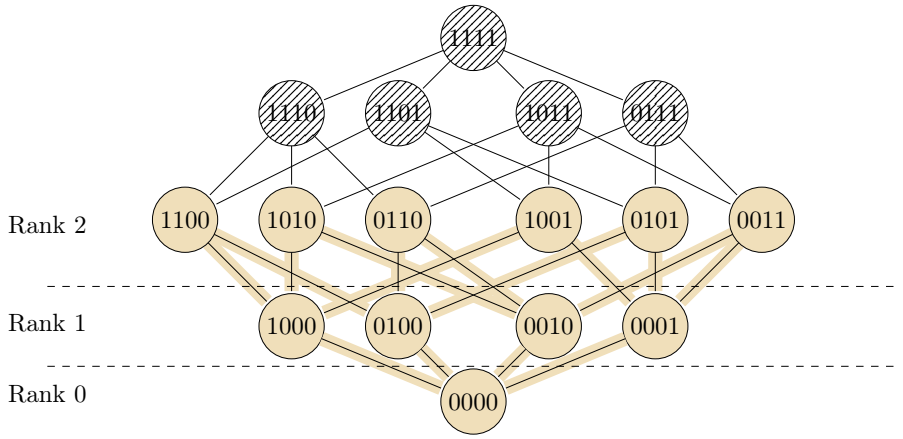


Figure 2.4: The uniform matroid U_4^2.

2.2.5 Matroid erection

Knuth's general matroid construction, the implementation of which is discussed in Section 4.3, requires us to establish a few more matroid concepts. The *rank-k truncation* of a rank-r matroid $\mathfrak{M} = (E, \mathcal{I})$, is the rank-$k$ matroid $\mathfrak{M}^{(k)} = (E, \mathcal{I}^{(k)})$, where

$$\mathcal{I}^{(k)} = \{I \in \mathcal{I} : |I| \leq k\}.$$

The *truncation* of $\mathfrak{M}$ is given as $T(\mathfrak{M}) = \mathfrak{M}^{(r-1)}$. As a simple example, we have that the uniform matroid $U_n^{n-1} = T(F_n)$, where F_n is the free matroid with n elements. The *erection* (I defer to Crapo for the somewhat esoteric choice of term [21]) of the matroid $\mathfrak{M}$ is the matroid $\mathfrak{N}$ of rank $\leq r+1$ such that $\mathfrak{M} = T(\mathfrak{N})$. In other words, when taking the truncation of a matroid, we produce a new matroid by disregarding all independent sets of cardinality higher than some k. When taking the erection of a rank-k matroid, we produce a new matroid by declaring some number of subsets $S \subseteq E$ such that $|S| = r+1$ to be independent, or equivalently, declaring some number of them to be closed sets of rank r. By declaring all of them to be dependent, we get the *trivial erection* $\mathfrak{N} = \mathfrak{M}$. By declaring none of them dependent (i.e., they are all independent sets $r(S) = |S| = r+1$), $\mathfrak{N}$ is the *free erection* of $\mathfrak{M}$ [22].

A matroid can have many erections. Figure 2.4 shows the Hasse diagram for the matroid of 4 elements, where every subset of two or fewer elements is independent (denoted in yellow). Each subset is encoded as a binary string,

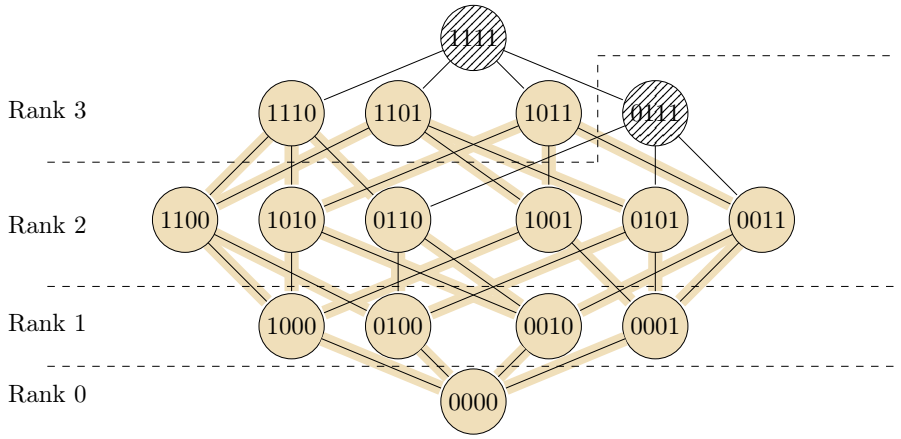


Figure 2.5: An erection of U_4^2.

where a 1 designates membership in the set (a syntax we will become very familiar with in Chapter 4). This matroid is known as the uniform matroid U_4^2—it is the matroid over a ground set of 4 elements, where every subset of size 2 or less is independent. The uniform matroid U_4^3 is the free erection of U_4^2, where we have simply designated all sets of rank 3 as independent. Figure 2.5 shows another erection of U_4^2, one in which only three of the subsets with three elements are designated as independent. The final set, 0111, remains a dependent set, and has therefore in fact been designated as a closed set of rank 2. By starting from the rank-0 matroid $\mathfrak{M}^{(0)}$, one can iteratively erect any matroid $\mathfrak{M}$, at each iteration i designating the sets to be closed sets of rank i (all the while ensuring that the axioms for the closed sets of a matroid are obeyed). This is the approach taken by Knuth in his matroid erection algorithm, which is discussed in Chapter 4.3.

The *essential closed sets* (often referred to as essential flats) of a matroid are the closed sets whose existence cannot be inferred from the closed sets of lower rank. The rank-2 set 0111 in the example above is an essential closed set. The essential closed sets of a matroid, together with their ranks, fully determine the matroid [23]. A rank-r matroid can also be described inductively as a sequence of erections $(X_0, X_1, \ldots, X_{r-1})$, where X_k is a set of $(k+1)$-subsets of E that roughly encode the closed sets of rank k [24]. This insight is the basis for Knuth's matroid construction algorithm.

2.3 Matroids in fair allocation

It should be clear at this point that matroids are compelling structures to work with in the context of fair allocations. There are two main use cases for matroids in fair allocation: matroid-rank valuation functions (as in the example scenario from the introduction) and matroid constraints. Matroids.jl will be developed with the empirical study of algorithms for these two scenarios in mind.

2.3.1 Matroid-rank valuations

In a fair allocation instance with matroid-rank valuations, each agent i has a corresponding matroid $\mathfrak{M}_i = (E, \mathcal{I}_i)$, where E (the set of goods) is the ground set of elements common to all agents' matroids. The example I give in Chapter 1 illustrates the fact that using matroids in this manner is a natural way of modeling many real-world behaviors of user preferences; matroid rank functions model diminishing returns, supplementary goods and fixed demand. This is one reason for the interest in matroid-rank valuations in fair allocation.

Another good reason for the interest in matroid-rank valuations is that the rich field of matroid theory offers many deep results that are useful for reasoning about what fairness guarantees can be made when agents have matroid-rank valuations. For instance, Babaioff gives an allocation mechanism called the *randomized prioritized egalitarian* (RPE) mechanism, that produces allocations that are clean, MAX-USW, EFX, leximin and $\frac{1}{2}$-MMS in polynomial time [25]. This is an appealing set of fairness guarantees, most of which are computationally intractable in the general, additive case

Note that submodular valuations are not fully representative of all combinatorial preferences. One example of a situation that can not be modeled with matroid rank functions is *complementary goods*. Consider, as an example, a fair allocation instance where two goods are the left and right shoe in a pair of shoes. An agent considers each individual shoe to be worthless on its own, but together they have value for the agent. This is a case of *supermodularity* (the opposite of submodularity; i.e., "increasing returns")—the second shoe has marginal value of 1 for an agent if and only if the first shoe is present in the agent's bundle—and is not representable using submodular valuations.

2.3.2 Matroid constraints

Another usage found for matroids in fair allocation is that of *matroid constraints*. The majority of work on fair division assumes that any allocation is feasible,

and the sole concern is finding an allocation that aligns well with the agents' valuation profiles. In many practical applications, however, there will be allocations that are not legal or desirable. Biswas and Barman [5] give an example of a museum with multiple branches distributing exhibits of different categories (sculpture, paintings, et cetera) among the branches. For each category, it wants to create a balanced distribution among the branches, so that the difference in the number of exhibits of a given category differ by at most one between any branch. This is an example of a *cardinality constraint*, which is a subset of the broader class of matroid constraints.

When matroid constraints are enforced on a fair allocation instance, we require that all bundles be independent sets on some supplied matroid, common to all agents.

Chapter 3

The Matroids.jl API

Matroids.jl exists to enable the empirical study of matroidal fair allocation. In this chapter, I consider what fair allocation-specific methods the Matroids.jl API should expose to achieve this goal, and how they might be implemented. While doing so, I keep track of which properties are required from the matroids being used. Chapter 4 describes how Matroids.jl generates a number of different matroid types, and how the getter functions for the properties we need are implemented.

The implementation will draw inspiration from, and be designed to integrate with, Hummel and Hetland's well-organized Allocations.jl library [10], which provides a range of algorithms for fair allocation of indivisible items. Allocations.jl currently supports additive and submodular valuations and a number of constraint types, including conflict constraints and cardinality constraints. Matroids.jl should extend Allocations.jl with support for matroid-rank valuations and matroidal constraints. As such, Matroids.jl should be structured in such a manner as to be familiar to those acquainted with Allocations.jl.

In this chapter, I describe how the Matroids.jl API is designed to enable experimentation with fair allocation algorithms for matroid-rank valuations. First, I describe how Matroids.jl extends the fairness and efficiency measures of Allocations.jl to handle matroid-rank valuations. With that in hand, I enumerate a few recent, interesting algorithms for this use case, and discuss which requirements their implementation would pose to Matroids.jl. Finally, I show how Matroids.jl supports the matroid union operation, using a classic matroid pro-

cedure attributable to Knuth [20] and Edmonds [19] that have found widespread use in fair allocation with matroid-rank valuations.

Implementing support for matroidal constraints is out of scope for this thesis. A discussion on how one might go about doing this in the future is included in Chapter 6.

3.1 Fairness under matroid-rank valuations

If Matroids.jl is to be of use in the empirical study of matroidal fair allocation algorithms, we need to be able to evaluate the fairness of an allocation. In this chapter, I show how Matroids.jl implements the fairness criteria given in Chapter 2. The valuation profile of matroid-rank-valued allocation problem instance gives the valuation function of each agent. This is represented as a struct containing the matroid $\mathfrak{M}_i$ for each agent i. Agent i's value for the set of goods S, $v_i(S)$, is the rank of S in $\mathfrak{M}_i$.

```
"""
    struct MatroidRank <: Profile

A matroid rank valuation profile, representing how each agent values all
     possible bundles. The profile is constructed from `n` matroids, one for
     each agents, each matroid over the set of goods {1, ..., m}.
"""
struct MatroidRank <: Profile
    Ms::Vector{Matroid}
    m::Int
end

value(V::MatroidRank, i, S) = rank(V.Ms[i], S)
value(V::MatroidRank, i, g::Int) = value(V, i, Set(g))
```

Figure 3.1: `MatroidRank` represents a fair allocation instance with matroid-rank valuations.

Envy-freeness

Checking if an allocation is EF is the same for matroid-rank valuations as for additive valuations – simply compare each agent's own bundle value with that

agent's subjective valuation of each other agent's bundle. This is already implemented in Allocations.jl. In this section, I give the functions `value_1`, `value_x` and `value_x0`, which are used for computing EF1, EFX_+ and EFX_0, respectively. These functions take in a valuation profile V, an agent i and a bundle S, and return the agent i's value for S, up to some item. If each agent i's received value exceeds the result of calling `value_1(V, i, S)` on every other agent's bundle S, then the allocation is EF1; equivalently so for EFX_+ and EFX_0.

If S is independent in $\mathfrak{M}_i$, then the removal of any good in S will reduce agent i's valuation of that bundle by 1. Then, agent i's value for S up to a highest-valued good is the same as that up to a least-valued (positively or no) good. In other words, `value_1(V, i, S)` = `value_x(V, i, S)` = `value_x0(V, i, S)` $= v_i(S) - 1$ when S is independent in $\mathfrak{M}_i$.

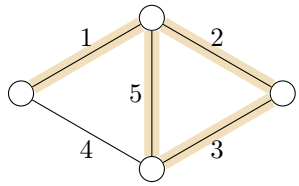


Figure 3.2: A graph, with a rank-3, size-4 bundle of edges highlighted in yellow

If that is not the case, however, the functions behave differently. It might be tempting to think that if S is not independent (dependent) in $\mathfrak{M}_i$, then no matter which good is removed from S, agent i would value the bundle the same. This is certainly the case sometimes; let $\mathfrak{M}_i = U_n^k$, and let S be some set of goods such that $|S| = k + 1$. By the definition of the uniform matroid, we have that $v_i(S) = k$. Yet remove any good g from S, and the resulting set is of size $|S - g| = k$, and the value remains unchanged. We might however also contrive a scenario in which the removal of some goods from a set S reduces i's valuation of that set, while some other goods while not affect it. This is the case when $\mathfrak{M}_i$ is the graphic matroid obtained from the graph in Figure 3.2, and S is the bundle highlighted in yellow. In this situation, we have $v_i(S) = 3$, and $v_i(S-1) = 2 \neq v_i(S-3) = 3$. Thus, agent i's value of S up to a highest-valued item is in this case $v_i(S) - 1 = 2$. Similarly, since the least positively-valued good also has value 1, EFX_+ is the same as EF1 (this is always the case with matroid-rank valuations). Finally, the least-valued good overall can be removed without a reduction in value when S is dependent. This line of reasoning gives us the implementation of `value_1`, `value_x` and `value_x0` given in Figure 3.3.

```
function value_1(V::MatroidRank, i, S)
  if is_indep(V.Ms[i], S)
    return max(length(S)-1, 0)
  end

  return minimum(value(V, i, setdiff(S, g)) for g in items(V))
end

value_x(V::MatroidRank, i, A) = value_1(V, i, A)

value_x0(V::MatroidRank, i, S) =
    is_indep(V.Ms[i], S) ? max(length(S)-1, 0) : value(V, i, S)
```

Figure 3.3: Methods for computing EF1 and EFX_+ and EFX_0.

Proportionality

To check whether an allocation A is PROP or some relaxation thereof, we compare $v_i(A_i)$ against some threshold for every agent i. In this section, we give the functions for computing the threshold for PROP and its relaxations.

```
prop(V::MatroidRank, i, _) = rank(V.Ms[i])/na(V)
prop_1(V::MatroidRank, i, A) = prop(V, i, A) - 1
prop_x(V::MatroidRank, i, A) = prop_1(V, i, A)
prop_x0(V::MatroidRank, i, A) =
    is_closed(V.Ms[i], A) ? prop_1(V, i, A) : prop(V, i, A)
```

Figure 3.4: Methods for computing PROP, PROP1, PROPX_+ and PROPX_0.

PROP_i is simply the rank of $\mathfrak{M}_i$, $v_i(\mathcal{M})$, as this is the maximum value achievable for agent i, divided by the number of agents in the problem instance.

To check for PROP1, we need to figure out if there exists some $g \in \mathcal{M}$ such that $v_i(A_i + g) \geq \frac{1}{n} v_i(\mathcal{M})$. We know, due to the hereditary property (as given in Section 2.2.3) that unless $v_i(A_i) = v_i(\mathcal{M})$ already (in which case we have trivial PROP1), there exists $g \in \mathcal{M} \setminus A_i$ such that $\Delta_i(A_i, g) = 1$. To figure out if A is PROP1, then, we need to check whether $v_i(A_i) + 1 \geq \frac{1}{n} v_i(\mathcal{M})$, or equivalently, whether $v_i(A_i) \geq \frac{1}{n} v_i(\mathcal{M}) - 1 = \text{PROP}_i - 1$. This is our PROP1 threshold. Since the least positively-valued element will also have a marginal value of 1, PROPX_+ is the same as PROP1.

When checking for PROPX_0, we want the $g \in E \setminus A_i$ whose addition would increase the value of A_i the least. The question, then, is whether there exists an element $g \in E \setminus A_i$ such that $\Delta_i(A_i, g) = 0$. If A_i is a closed set (i.e., maximal for its rank), then any additional good will increase the rank by 1, otherwise there exists some such g.

Maximin share

Matroids.jl implements Barman and Verma's [6, Appendix A] method for computing agent i's maximin share in polynomial time. Recall that the maximin share for agent i, μ_i, is the best bundle value she can achieve by allocating the goods to the n agents and choosing the worst bundle for herself. Barman and Verma show that even if we require each bundle considered to be clean (i.e., independent in $\mathfrak{M}_i$), we still find μ_i. The task, therefore, is to find the partition of E into n sets independent in $\mathfrak{M}_i$ maximizing the minimum bundle value.

This is equivalent to finding a maximum-size independent set in the n-fold union of $\mathfrak{M}_i$ with itself, i.e.,

$$\widehat{\mathfrak{M}}_{i\times n} = (E, \widehat{\mathcal{I}}_{i\times n}) = (E, \{I_1 \cup \cdots \cup I_n : I_t \in \mathcal{I}_i,\ \forall t \in N\}),$$

which can be produced in polynomial time using the matroid union algorithm [14, Ch. 42]. Let $\widehat{A} \in \widehat{\mathcal{I}}_{i\times n}$ be such a set. As shown in Section 2.2.4, $\widehat{A}$ allows an n-partition $A = (A_1, \ldots, A_n)$ such that, in this case, $A_t \in \mathcal{I}_i$ for all $t \in N$.

Matroids.jl implements Knuth's 1973 matroid union algorithm [20]. The implementation is given in Appendix B, for now let it suffice to say that we have a function called `matroid_partition_knuth73`, which, when given k matroids $(\mathfrak{M}_1, \ldots, \mathfrak{M}_k)$ over the same ground set E, returns a partition of E into k sets $S = (S_1, \ldots, S_k)$ such that each set S_t is independent in $\mathfrak{M}_t$. By passing n copies of $\mathfrak{M}_i$, we get the n-partition A as above.

With A in hand, Barman and Verma's procedure iteratively update the sets of A as long as there exist $j, k \in N$ such that $v_i(A_j) - v_i(A_k) \geq 2$. This is equivalent to $|A_j| - |A_k| \geq 2$, since the sets are all independent in $\mathfrak{M}_i$. When this is the case, there exists (due to the exchange property) a good $g' \in A_j$ such that $A_k + g' \in \mathcal{I}_i$. The sets are updated $A_j \leftarrow A_j - g'$ and $A_k \leftarrow A_k + g'$ until no two sets differ in cardinality by more than one. Now, we have a partition of E into n evenly sized subsets that are independent in $\mathfrak{M}_i$. The value of worst of these is agent i's maximin share. Matroids.jl's implementation of this procedure is given in Figure 3.5.

```
"""
  function mms_i(V, i)

Finds the maximin share of agent i in the instance V.
"""
function mms_i(V::MatroidRank, i)
  M_i = V.Ms[i]; n = na(V)

  # An initial partition into independent subsets (subjectively so for i).
  (A, _) = matroid_partition_knuth73([M_i for _ in 1:n])

  # Setup matrix D st D[j,k] v_i(A_j) - v_i(A_k) ∀ j,k ∈ [n].
  D = zeros(Int8, n, n)
  for j in 1:n, k in 1:n
      # v_i(A_p) = |A_p| since all sets in A are independent wrt M_i.
      D[j,k] = length(A[j]) - length(A[k])
  end

  jk = argmax(D)
  while D[jk] > 1
      j,k = Tuple(jk)

      # By the augmentation property, ∃g ∈ A_j st A_k + g ∈ I_i.
      g = nothing
      for h ∈ setdiff(A[j], A[k])
          if is_indep(M_i, A[k] ∪ h)
              g = h; break
          end
      end

      # Update A.
      setdiff!(A[j], g); union!(A[k], g)

      # Update D.
      for l in 1:n
          D[j, l] -= 1; D[l, j] += 1 # A_j is one smaller.
          D[k, l] += 1; D[l, k] -= 1 # A_k is one larger.
      end

      jk = argmax(D)
  end

  return minimum(length, A)
end
```

Figure 3.5: Maximin share computation.

3.2 Three selected algorithms

The purpose of Matroids.jl being to enable the empirical study of matroidal fair allocation, we should investigate what requirements algorithms in this space pose of a library that aims to enable their implementation. In order to maintain a manageable scope, I restrict my attention to three recent algorithms for fair allocation with matroid-rank valuations, that, while relatively short and sweet, make use of some deep results from matroid theory to deliver well on a range of fairness criteria.

The Envy-Induced Transfers algorithm. This algorithm is due to Benabbou, Chakraborty, Igarashi and Zick [7]. Named Algorithm 1 in the paper, it relies on a subroutine the authors name *Envy-Induced Transfers* (EIT)—hence the name. Benabbou et al. show that, for matroid-rank valuations, a Pareto Optimal, MAX-USW and EF1 allocation always exist and can be computed efficiently, using the simple greedy algorithm given in Algorithm 1.

The algorithm should look familiar; it is very similar to Barman and Verma's procedure for computing an agent's maximin share detailed in the previous section. The crux of both approaches is the concept of the matroid union: a maximum-size independent set in the union of the matroids in play is a clean MAX-USW allocation. We find such a clean allocation using the MATROID-PARTITION subroutine, which accepts the matroids and the set of elements.

With that in hand, we can use the exchange property of independent sets to greedily choose goods to transfer until the allocation has the desired properties. In this case, the algorithm continues transfering as long some agent envies another agent for more than one good; when it terminates, the allocation is thus EF1.

Algorithm 1 ENVY-INDUCED-TRANSFERS [7]

Input: A matroid-rank-valued fair allocation instance $(N, E, \{\mathfrak{M}_i\}_{i \in N})$.
Output: A clean, MAX-USW, EF1 allocation $A = (A_1, \ldots, A_n)$.

Let $A =$ MATROID-PARTITION$(\{\mathfrak{M}_i\}_{i \in N}, E)$ (*clean and MAX-USW*)
while there are two agents $i, j \in N$ st. i envies j more than 1 good, **do**
    Find good $g \in A_j$ with $\Delta_i(A_i, g) = 1$
    Update $A_j \leftarrow A_j - g$
    Update $A_i \leftarrow A_i + g$
end
return A

AlgMMS. This algorithm is given in Algorithm 2, and is due to Barman and Verma [6]. It is similar to ENVY-INDUCED-TRANSFERS in that it first generates an initial clean, MAX-USW allocation using MARTOID-PARTITION, before massaging this allocation until the desired properties are met. In this case, the desired property is that of MMS-fairness; each agent should receive at least their share μ_i, computed using the MMS subroutine, an implementation of which is described in the previous section.

ALGMMS achieves this by keeping track of which agents have received more than their MMS (these make up the set $S_>$), and which have received less ($S_<$). While there are agents i such that $v_i(A_i) < \mu_i$, the algorithm constructs an exchange graph D with the BUILD-EXCHANGE-GRAPH subroutine. It then finds a shortest path P in D from a good for which i has positive marginal value (the set of goods F_i), to a good currently located in A_j, for some $j \in S_>$. Finally it augments, with TRANSFER subroutine, the allocation along a transfer path Barman and Verma show that MMS-fair MAX-USW allocations always exist for instances with matroid-rank valuations, and that ALGMMS finds them in polynomial time. To build an intuitive understanding of how the algorithm works, let us take a look at an instructive example.

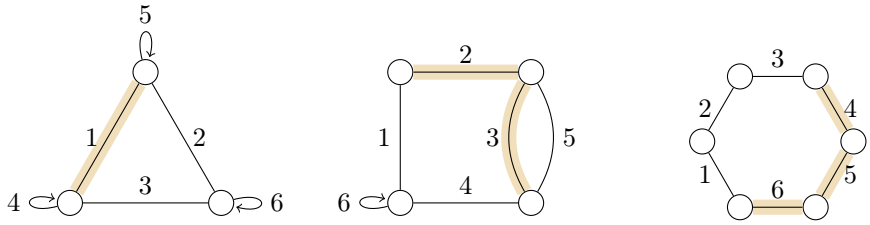
Algorithm 2 AlgMMS [6]

Input: A matroid-rank-valued fair allocation instance $(N, E, \{\mathfrak{M}_i\}_{i\in N})$, and $\mu_i = \text{MMS}(N, E, \mathfrak{M}_i)$ for every $i \in N$.
Output: A clean, MAX-USW, MMS-fair allocation $A = (A_1, \ldots, A_n)$.

Let $A = \text{Matroid-Partition}(\{\mathfrak{M}_i\}_{i\in N}, E)$ (*clean and MAX-USW*)
Initialize $S_< = \{i \in N : v_i(A_i) < \mu_i\}$
Initialize $S_> = \{i \in N : v_i(A_i) > \mu_i\}$
while $S_< \neq \emptyset$, **do**
    Select any agent $i \in S_<$
    Let $F_i = \{g \in E : \Delta_i(A_i, g) = 1\}$
    Let $D = \text{Build-Exchange-Graph}(A)$
    Let $P = \text{Shortest-Path}(D, F_i, \bigcup_{j\in S_>} A_j)$
    Update $A \leftarrow \text{Transfer}(A, P)$
    Reset $S_< = \{i \in N : v_i(A_i) < \mu_i\}$
    Reset $S_> = \{i \in N : v_i(A_i) > \mu_i\}$
end
Let $junk = E \setminus \bigcup_{i=1}^{n} A_i$ be the set of unallocated goods
return $(A_1 \cup junk, A_2, \ldots, A_n)$

Figure 3.6 illustrates a situation in which we have three agents a_1, a_2 and a_3, whose valuation functions are the rank functions over three different graphic matroids, and six goods $E = \{1, \ldots, 6\}$. The allocation A is highlighted in yellow: $A_{a_1} = \{1\}$, $A_{a_2} = \{2, 3\}$ and $A_{a_3} = \{4, 5, 6\}$. This might be the initial allocation output from Matroid-Partition, as it is MAX-USW ($\text{SW}(A) = 6 = |E|$). It should be clear that the maximin share of each agent is 2; in the situation depicted, a_3 has received a bundle of value 3, at the expense of a_1, who only received a bundle of value 1. Yet, a_1 is not envious of a_3's bundle, since the goods in A_{a_3} are all worthless to a_1. So A is EF1, but not MMS-fair.

We begin to see that Envy-Induced-Transfers has it easy; it can eliminate the envy one good at a time by moving a good directly from the envied to the envious agent, stopping when no agent directly envies another. This is equivalent to augmenting along a length-1 transfer path on the exchange graph. AlgMMS, on the other hand, might encounter a situation such as this one, in which some agent has more than their MMS, whilst another has less, yet no good in the fortunate agent's bundle will improve the situation of the unfortunate one—in that case, there must be a transfer path of length > 1 between the two agents.



(a) Agent a_1 (b) Agent a_2 (c) Agent a_3

Figure 3.6: Three agents represented by their valuation matroids, the allocation A highlighted in yellow.

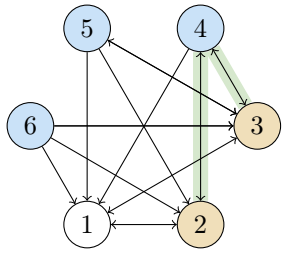
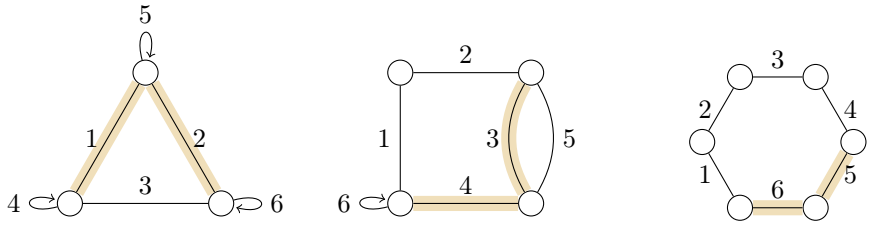


Figure 3.7: The exchange graph of the allocation during an intermediate step of AlgMMS, with F_{a_1} highlighted in yellow, goods belonging to agents in $S_>$ in blue and the transfer paths in green.



(a) Agent a_1 (b) Agent a_2 (c) Agent a_3

Figure 3.8: The resulting MMS-fair allocation after augmenting along $(2, 4)$.

Figure 3.7 shows the exchange graph $D(A)$ (i.e., the directed graph with a node per good, and an edge (u, v) iff good u can be exchanged with good v for no loss in value for the current holder of u). Highlighted in yellow are the goods in F_{a_1}, the set of goods g such that $\Delta_{a_1}(A_{a_1}, g) = 1$. The blue nodes are the goods belonging to an agent in $S_>$, the set of agents who have received more than their MMS. The green edges show the paths between these two sets of goods. As we can see, the available transfer paths are $(2, 4)$ and $(3, 4)$, representing a transfer of good 4 from A_{a_3} to A_{a_2}, and good 2 or 3 from A_{a_2} to A_{a_1}, respectively.

We augment along the transfer path $(2, 4)$ and end up with the allocation shown in Figure 3.8, in which every agent has their maximin share of value. Success! After all agents have received their MMS, any remaining unallocated goods (none in the example case) are simply allocated to agent 1. This ensures that the allocation is complete, though not necessarily clean. These goods, denoted *junk* in Algorithm 2, are the goods for which no agent has any additional value, either because they were always 0-valued or because every agent has achieved a basis in their matroid.

AlgMMS highlights how simple algorithms can deliver strong fairness guarantees when working with matroid-rank valuations, compared to the general, additive case, where even computing the MMS of a single agent is NP-hard. With this short, highly grokkable algorithm, we can produce MMS-fair allocations in polynomial time.

Yankee Swap. The most recent of the three algorithms discussed in this chapter is due to Viswanathan and Zick [8], and is named Yankee Swap[1]. This algorithm delivers very well on a range of fairness and efficiency notions; it finds, in polynomial time, a clean allocation that is MNW, MAX-USW, EFX, leximin and $\frac{1}{2}$-MMS-fair (every agent receiving at least half of their maximin share). In addition to this, the authors argue that a major selling point of the algorithm is that it is easy to reason about, as it does not use complex matroid optimization operations (i.e., Matroid-Partition) as subroutines.

In the paper, the algorithm requires that a priority order π of the agents is passed along with the fair allocation instance, where π represents some permutation of the agents in N, denoting the prioritization of the agents. By choosing π uniformly at random, the algorithm produces allocations that are also, in expectation, EF and PROP (known as *ex ante envy-freeness* and *ex ante pro-*

[1]Named after the gift exchange game also known as White Elephant and Dirty Santa, in which participants, upon receiving a gift, can choose to keep it or steal another player's gift. See https://en.wikipedia.org/wiki/White_elephant_gift_exchange for details.

portionality). When discussing Yankee Swap in this thesis, I will without loss of generality assume that the ordering of the agents in N is selected in such a manner elsewhere beforehand; that is, it has been randomly chosen which agent is to be agent 1 before the algorithm is run.

The pseudocode is given in Algorithm 3. Similarly to Envy-Induced-Transfers and AlgMMS, it starts out with an initial allocation; in this case, however, this initial allocation is the one in which every good is allocated to the new agent 0, whose bundle A_0 represents the unallocated goods. After this, the procedure is reminiscent of the other algorithms described in this section. At each iteration, i is the highest priority (i.e., first) agent with the least value whose bundle can still improve. We find F_i as above, and build the exchange graph D for the allocation A. If there exists a transfer path P from F_i to the set of unallocated good A_0, we augment A along P. The transfer operation can be understood as a sequence of thefts, wherein agent i improves her lot by stealing a good from another agent's bundle, whereupon the robbed agent compensate for this by stealing from another agent, and so on. The last agent in the path will lose a good, and so we are only interested in transfer paths ending at A_0, as these are the paths representing the allocation of one additional good, thereby increasing the social welfare of the allocation by one. If no such path exists, we know that A_i cannot improve and we disregard i in future iterations. The algorithm terminates when no bundles can be further improved, and the resulting allocation has the properties above.

Having understood the three algorithms we will try to implement, we can now enumerate the functional requirements they pose to Matroids.jl. Common logic has been extracted into separate subroutines, viz. Matroid-Partition, Build-Exchange-Graph, Shortest-Path and Transfer. The marginal value function Δ_i is also required—depending on the circumstance, this can be implemented using either a rank function Rank or an independence oracle Indep. AlgMMS also requires the ability to compute an agent's maximin share with the MMS subroutine. With all of this in place, we should be able to implement these algorithms. The requirements are listed in Table 3.1.

The implementation of Rank and Indep will depend on the specific type of matroid in question, and is discussed in the next chapter. We have already seen how Matroids.jl implements MMS. To finish off this chapter, then, let us examine how Matroids.jl supports the matroid partitioning procedure, and related operations on the exchange graph.

Algorithm 3 YANKEE-SWAP [8]

Input: A matroid-rank-valued fair allocation instance $(N, E, \{\mathfrak{M}_i\}_{i \in N})$.
Output: A clean, MAX-USW, EFX, leximin $\frac{1}{2}$-MMS-fair allocation $A = (A_1, \ldots, A_n)$.

$A = (A_0, A_1, \ldots, A_n) = (E, \emptyset, \ldots, \emptyset)$
$\mathit{flag}_j =$ FALSE for all $j \in N$
while $\mathit{flag}_j = \mathit{false}$ for some $j \in N$, **do**
    Let T be the agents $j \in N$ with $\mathit{flag}_j =$ FALSE
    Let T' be the agents in T with least value in A
    Let i be the first agent in T' (*The highest priority agent in* T')
    Let $F_i = \{g \in E : \Delta_i(A_i, g) = 1\}$
    Let $D =$ BUILD-EXCHANGE-GRAPH(A)
    if there exists a shortest path $P =$ SHORTEST-PATH(D, F_i, A_0), **do**
        Update $A \leftarrow$ TRANSFER(A, P)
    else
        $\mathit{flag}_i \leftarrow$ TRUE
    end
end
return A

3.3 Exchange graphs and transfer paths

As shown in Section 3.2, exchange graphs and transfer paths on these are useful tools for fair allocation with matroid-rank valuations. Therefore, Matroids.jl exposes functions for computing the exchange graph $D(A)$ of an allocation A, finding a shortest (transfer) path P between two sets of nodes in $D(A)$, and producing a new allocation $A \Lambda P$ by augmenting A along P.

Building the exchange graph. The implementation of BUILD-EXCHANGE-GRAPH is given in Figure 3.9. Matroids.jl uses the Graphs.jl library [26] to handle graphs, and the Allocations.jl library [10] provide several handy types and functions for representing and working with allocations. Armed with tools from these libraries, building the exchange graph is a breeze. After initializing the empty directed graph $D = (E, \emptyset)$ with a node per good and no edges, `exchange_graph` iterates over all pairs of goods g_i, g_j, and adds an edge (g_i, g_j)

Algorithm	**Requirements**
Envy-Induced Transfers	• Matroid-Partition • Rank
AlgMMS	• Matroid-Partition • MMS • Rank • Indep • Build-Exchange-Graph • Shortest-Path • Transfer
Yankee-Swap	• Build-Exchange-Graph • Shortest-Path • Transfer • Rank • Indep

Table 3.1: Functional requirements for Matroids.jl posed by three recent fair allocation algorithms

```
function exchange_graph(Ms, A; all_indep=true) where T <: Matroid
  m = ni(A)
  D = SimpleDiGraph(m)

  # Checking for each i,j whether element ei can be replaced with ej.
  for gi in 1:m, gj in setdiff(1:m, gi)
    if !owned(A, gi) continue end
    i = owner(A, gi)

    if all_indep
      # Check if A_i - ei + ej is independent in Mi.
      if is_indep(Ms[i], setdiff(bundle(A, i), gi) ∪ gj)
        add_edge!(D, gi, gj)
      end
    else
      if rank(Ms[i], bundle(A, i)) == rank(Ms[i], setdiff(bundle(A,i), gi) ∪ gj)
        add_edge!(D, gi, gj)
      end
    end
  end

  return D
end
```

Figure 3.9: `exchange_graph(Ms, A)` finds the exchange graph of the allocation `A`, given the array of matroids `Ms`

if the owner of g_i would be just as happy with the bundle where g_i is replaced with g_j.

If we know that all bundles A_i are independent in $\mathfrak{M}_i$, then we do not need to actually compute the rank of the bundle before and after some g_i, g_j replacement—instead, we can simply check if the new bundle $A_i - g_i + g_j$ is also independent, replacing a couple of calls to RANK with one call to the cheaper INDEP. For the use cases outlined in Section 3.2, this is always the case, and so this behavior is default.

With this implementation, Matroids.jl constructs the exchange graph for an allocation in $\mathcal{O}(m^2)$ calls to the independence oracle (or rank function).

Finding a transfer path. The purpose of the exchange graph $D(A)$ is to find a transfer path P on it, and produce the augmented allocation $A \Lambda P$. A transfer path is by definition a shortest path from some set of nodes (in the case of YANKEE-SWAP, F_i) to some other set of nodes (A_0) in $D(A)$. Matroids.jl

```
function find_shortest_path(D, from, to)
  X = intersect(from, to)
  if length(X) > 0
    return [X[1]]
  end

  ds = dijkstra_shortest_paths(D, from)
  paths = []

  for g in to
    path = [ds.parents[g], g]

    while path[1] ∉ from
      if path[1] == 0 @goto skip end
      pushfirst!(path, ds.parents[path[1]])
    end

    push!(paths, path)
    @label skip
  end

  if length(paths) == 0 return nothing end
  return argmin(length, paths)
end
```

Figure 3.10: `find_shortest_path` finds a shortest path between `from` and `to` using Dijkstra's algorithm

achieves this using Dijkstra's algorithm [27] for finding the shortest paths between a set of nodes and all other nodes.

Transfer path augmentation. With a transfer path in hand, all that is left is to implement a function to create a new allocation by passing the goods along the path. Figure 3.11 shows the source code for `transfer!`. For each good g in the path, there is a winning agent x (initialized to i) who receives g, and a losing agent y, who loses g. At the end of each iteration, $y \leftarrow x$, so that only the owner of the last good in the path actually experiences a reduction in bundle value. After each transfer, the set of edges out of g in the exchange graph is updated.

```
  function transfer!(Ms, D, A, i, path; all_indep=true)
  # At every iteration, x receives the next good in the path.
  x = i
  for g in path
    # y is the current owner, who loses g.
    y = owner(A, g)
    deny!(A, y, g)
    give!(A, x, g)

    # Recalculate neighbors of g in D.
    for g_ in vertices(D)
      # Check if A_x - g + g_ is independent in Mi.
      if all_indep
        if is_indep(Ms[x], setdiff(bundle(A, x), g) ∪ g_)
          add_edge!(D, g, g_)
        else
          rem_edge!(D, g, g_)
        end

      else
        if rank(Ms[x], bundle(A, x)) == rank(Ms[x], setdiff(bundle(A, x), g) ∪
            g_)
          add_edge!(D, g, g_)
        else
          rem_edge!(D, g, g_)
        end
      end
    end

    x = y
  end
end
```

Figure 3.11: `transfer!` augments `A` along `path`, updating `D` as it goes along

Chapter 4

Generating matroids

The overarching goal for this project is to make Matroids.jl, a proof-of-concept library for working programmatically with matroids, specifically in the context of fair allocation. This chapter covers how Matroids.jl enables the creation of specific matroids and the generation of random ones, as well as how to access important properties such as independent sets, closed sets, circuits, bases, the rank function and the closure function. The first part of the chapter focuses on implementing these features for uniform and graphic matroids. The latter part of this chapter describes the implementation of Knuth's algorithm [11] for the erection of arbitrary rank-r matroids. Getting this algorithm to work represents a significant portion of the project as a whole, and its successful implementation is one of the main achievements of this thesis.

This chapter consists of one section per matroid type this version of Matroids.jl supports. For each type, I describe how the matroid is represented, how to acquire the properties of the matroid, and how it might be randomly generated. Which matroid properties Matroids.jl will have dedicated functions for getting is based on the discussion in the previous chapter, wherein I described how the more high-level functions exposed in the Matroids.jl API are implemented. Naturally, access to a matroid's rank function and independence oracle is of paramount importance—one of these are required by all higher-level functionality. The threshold function for PROPX_0 gave a use case for the closure function. Hence, the properties for which Matroids.jl implements getter functions for every matroid:

1. `rank(M, S)` returns the rank of the set `S` in the matroid `M`.

2. `is_indep(M, S)` returns whether the set `S` is independent in the matroid `M`.

3. `closure(M, S)` returns the closure of the set `S` in the matroid `M`, i.e., the closed set of least rank that contains `S`

While the independence of a set S could trivially be determined by checking whether $v(S) = |S|$, there is often a faster way of determining purely the independence without actually calculating the rank. With these functions in hand, one can procure the other basic properties of a matroid with relative ease—for instance, a circuit oracle checking whether a given set is a circuit could be implemented like this:

```
is_circuit(M, S) = rank(M, S) == length(S) - 1
```

4.1 Uniform matroids

We start off lightly with the most basic type of matroid—the uniform matroid. Recall that the uniform matroid U_n^r is the matroid over n elements where the independent sets are exactly the sets of cardinality at most r. Two useful special cases of the uniform matroid on n elements are the rank-n free matroid, in which every subset of E is independent, and the rank-0 zero matroid, in which only the empty set is independent.

```
struct UniformMatroid
  n::Integer
  r::Integer
end

FreeMatroid(n) = UniformMatroid(n, n)
ZeroMatroid(n) = UniformMatroid(n, 0)
```

Extracting the properties of U_n^r is a simple matter. The rank of a subset $S \subseteq E$ is given by $\max\{|S|, r\}$. S is independent iff $|S| \leq r$. The closure of S is S if $|S| \leq r$ (since every larger set has higher rank), elsewise it is E.

```
rank(M::UniformMatroid, S) = min(length(S), M.r)
is_indep(M::UniformMatroid, S) = length(S) <= M.r
closure(M::UniformMatroid, S) = length(S) < M.r ? S : ground_set(M)
```

4.2 Graphic matroids

Graphic matroids were introduced back in Chapter 2, but in this chapter, I describe how Matroids.jl represents and randomly generates them.

First, some definitions for the graph theory terms used in this section. An undirected graph $G = (V, E)$ is said to be *connected* if there exists at least one path between each pair of nodes in the graph; otherwise it is *disconnected*. A disconnected graph consists of at least two connected subsets of nodes. These connected subgraphs are called *components*.The *degree* of a node v is the number of edges for which v is an endpoint. A *regular graph* is a graph in which all nodes have the same degree. An *induced subgraph* $G[S]$, where S is either a subset of the nodes of G (in which case $G[S]$ is a *node-induced subgraph*) or of the edges of G (*edge-induced*).

4.2.1 Random graphs

In order to generate random graphic matroid, we will need to generate random graphs. Let us take a look at some of the options available to us for this. Luckily for us, random graphs has been an area of extensive study for more than sixty years, and several models with different properties exist.

The Erdős-Rényi (ER) model (also known as Erdős-Rényi-Gilbert [28]) picks uniformly at random a graph from among the $\binom{\binom{n}{2}}{M}$ possible graphs with n nodes and M edges, or, alternatively, constructs a graph with n nodes where each edge is present with some probability p [29, 30]. This model produces mostly disconnected graphs, and the size distribution of its components with respect to the number of edges has been studied extensively. With n nodes and fewer than $\frac{n}{2}$ edges, the resulting graph will almost always consist of components that are small trees or contain at most one cycle. As the number of edges exceeds $\frac{n}{2}$, however, a so-called "giant" component of size $\mathcal{O}(n)$ emerges, and starts to absorb the smaller components [31]. The ER model is the oldest and most basic random graph model, and is often referred to simply as the random graph, denoted by $G(n, p)$.

Variations of the ER model have been developed by physicists and network scientists to produce phenomena commonly seen in real-world networks [28]. These variations include the Barabási-Albert model, which grows an initial connected graph using preferential attachment (a mechanism colloquially known as "the rich get richer"), in which more connected nodes are more likely to receive new connections. This results in graphs in which a small number of nodes ("hubs") have a significantly higher degree than the rest, creating a power-law distribution of node degrees. This property is known as scale-freeness and is thought to be a characteristic of the Internet [32].

Another approach is the Watts-Strogatz model [33], which starts with a ring lattice, a regular graph with n nodes, each with degree k, and then rewires each edge with some probability p. By changing p, one is able to 'tune' the graph between regularity (p=0) and disorder (p=1). For intermediate values of p, Watts-Strogatz produces so-called "small-world" graphs, which exhibit both a high degree of clustering (how likely two nodes with a common neighbor are to be adjacent), and short average distance between nodes. This phenomenon is found in many real-world networks, such as social systems or power grids [28].

4.2.2 Generating random graphic matroids

We will use the Graphs.jl library [26] for handling graphs in Matroids.jl. This library has built-in functions for the random graph models described in the previous chapter[1].

```
function random_ba_graph(m)
  k = rand([x for x in 1:ceil(Integer, sqrt(m)) if m\%x == 0])
  n = m ÷ k + k

  return barabasi_albert(n, k)
end
```

Figure 4.1: Compute a Barabási-Albert model random graph with m edges from random parameters

When generating random matroids, we want to be able to specify the size of the ground set, and perhaps also have some say in the rank of the matroid.

[1] https://docs.juliahub.com/Graphs/VJ6vx/1.4.1/generators/

Let us see how we can achieve this with the random graph models we have discussed. The function `barabasi_albert(n,k)` generates a Barabási-Albert model random graph with n nodes. It starts with an initial graph of k nodes, and adds the remaining $n - k$ nodes one at a time, each new node receiving k edges via preferential attachment. Thus, the final graph has $|E| = (n - k)k$ edges. To specify a matroid with m edges, we pick some k that divides m and solve for n. Figure 4.1 shows a snippet of Julia code that generates a Barabási-Albert model random graph with random parameters such that the number of edges is m.

```
function random_ws_graph(m)
  n = rand([x for x in 2:ceil(Integer, sqrt(2m)) if 2m\%x == 0 && iseven(x)])
  k = 2m ÷ n
  (k, n) = sort([n,k])

  return watts_strogatz(n, k, rand())
end
```

Figure 4.2: Compute a Watts-Strogatz model random graph with m edges from random parameters

Remember that the rank of a graphic matroid is the size of a spanning tree over the graph, which is $n - 1$ when the graph is connected. If we select a smaller k from among the factors of $|E|$, we get a larger final rank, and vice versa. We can generate a Watts-Strogatz model random graph with the function `watts_strogatz(n, k, β)`, where n is the number of nodes, k the node degree and β the probability of rewiring. The number of edges of a regular graph with n nodes and degree k (and thus the size of the ground set of the induced graphic matroid) is given by $\frac{nk}{2}$, so nk must be even. Figure 4.2 shows a snippet of Julia code that randomly generates a Watts-Strogatz model random graph with m edges.

Figure 4.3 shows a snippet of Julia code that generates a random Erdős-Rényi model random graph with m edges. ER is the simplest model for our purposes, as the function `erdos_renyi(nv, ne)` simply takes in the desired number of nodes and edges. The resulting graph consists of mostly small trees and single-cycle components for $m < \frac{n}{2}$ [31]. The code in Figure 4.3 somewhat arbitrarily picks the number of vertices randomly in the range from $\frac{m}{2}$ to $3m$.

```
function random_er_graph(m)
  n = rand(trunc(Integer, m/2):3m)
  return erdos_renyi(n, m)
end
```

Figure 4.3: Compute a Erdős-Rényi model random graph with m edges and a randomly chosen number of edges

```
using Graphs

struct GraphicMatroid
  g::Graph
  n::Integer
  r::Integer
  GraphicMatroid(g::Graph) = new(g, ne(g), length(kruskal_mst(g)))
end
```

Figure 4.4: Matroids.jl representation of a graphic matroid

4.2.3 Properties of random graphic matroids

In Matroids.jl, we "generate" a graphic matroid by simply accepting some graph, and figure out the rank of the matroid using Kruskal's algorithm for maximal spanning forests, which runs in $\mathcal{O}(|E| \lg |E|)$ time [34]. This is shown in Figure 4.4. Implementing the methods for finding the properties of our graphic matroids is simple, as they reduce to well-known algorithms (implemented by Graphs.jl) for finding the properties of the graphs they are derived from.

The rank function. The rank of a set $S \subseteq E$ is the size of a spanning forest of the subgraph induced by S, and can be found in $\mathcal{O}(|S| \lg |S|)$ time using Kruskal's algorithm. Figure 4.5 gives the source code for the rank function on a graphic matroid.

The indepence oracle. A set S is independent if the subgraph induced by S is acyclic. The `is_cyclic` check provided by Graphs.jl uses a DFS behind

```
  function rank(m::GraphicMatroid, S)
  edgelist = [e for (i, e) in enumerate(edges(g)) if i in S]
  subgraph, _vmap = induced_subgraph(m.g, edgelist)
  return length(kruskal_mst(subgraph))
  end
```

Figure 4.5: `rank(m::GraphicMatroid, S)`

the scenes[2], which runs in linear time [34]. Figure 4.6 gives the source code for the independence oracle on a graphic matroid.

```
function is_indep(m::GraphicMatroid, S)
  edgelist = [e for (i, e) in enumerate(edges(g)) if i in S]
  subgraph, _vmap = induced_subgraph(m.g, edgelist)
  return !is_cyclic(subgraph)
end
```

Figure 4.6: `is_indep(m::GraphicMatroid, S)`

The closure function. This operation accepts a set of elements S, and returns the largest set of elements $cl(S)$ such that $S \subseteq cl(S) \subseteq E, r(S) = r(cl(S))$. In a graph context, given a graph $G = (V, E)$ and an edge-induced subgraph $G[S] = (V', S), S \subseteq E$, this is the same as finding the largest edge-induced subgraph $G[T], S \subseteq T \subseteq E$, in which a spanning tree has the same number of edges as one in $G[S]$. Since the size of a spanning tree in $G[S]$ is given by $|V'| - 1$, $G[T]$ cannot contain any edges to nodes not in V', as this would increase the rank of $G[T]$. Therefore, we get that the closure of S is the largest set T of edges between nodes that are present in the edge-induced subgraph $G[S]$. Figure 4.7 gives the source code for the closure function on a graphic matroid.

[2]https://docs.juliahub.com/Graphs/VJ6vx/1.4.1/pathing/#Graphs.is_cyclic

```
function closure(m::GraphicMatroid, S)
  edgelist = [e for (i, e) in enumerate(edges(m.g)) if i in S]
  _sg, vmap = induced_subgraph(m.g, edgelist)
  return [e for e in edges(m.g) if [e.src, e.dst] ⊆ vmap]
end
```

Figure 4.7: `closure(m::GraphicMatroid, S)`

4.3 Knuth's matroid construction

In the preparatory project to this thesis, delivered to my advisor in the fall of 2022, I implemented Knuth's 1974 algorithm for the random generation of arbitrary matroids via the erection of closed sets [11]. With this, I was able to randomly generate matroids of universe sizes $n \leq 12$, but for larger values of n my implementation was unbearably slow. In this section, Knuth's method for random matroid construction will be described, along with the steps I have taken to speed up my initial, naïve implementation.

Knuth-Matroid (given in Algorithm 4) accepts the ground set E and a list X such that $\mathrm{X}[i] \subseteq 2^E$, and produces the rank-$r$ matroid $\mathfrak{M}$ such that $\mathrm{rank}(X) = k$ for each $X \in \mathrm{X}[k]$. This is done in a bottom-up manner through r sequential erections starting from the empty rank-0 matroid, $\mathfrak{M}^{(0)}$, each iteration i producing the erection $\mathfrak{M}^{(i+1)}$ from $\mathfrak{M}^{(i)}$ and $\mathrm{X}[i]$. The algorithm outputs the tuple (E, F), where $\mathrm{F} = [F_0, \ldots, F_r]$, r being the final rank of $\mathfrak{M}$ and F_i the family of closed sets of rank i. In the paper, Knuth shows that $\bigcup_{i=0}^{r} \mathrm{F}[r] = \mathcal{F}$, where $\mathcal{F}$ is the set of closed sets of a matroid, and so the algorithm produces a valid matroid represented by its closed sets.

To understand the procedure, let us investigate what Algorithm 4 does at iteration $1 < i < r$, where r is the final rank $\mathfrak{M}$, the matroid under construction. At iteration i, we produce a rank-$(i+1)$ erection $\mathfrak{M}^{(i+1)}$ of $\mathfrak{M}^{(i)}$, which is represented by its closed sets $\mathrm{F} = [F_0, F_1, ..., F_i]$, where F_i is the set of closed sets of rank i. We want to produce the set F_{i+1} of rank-r closed sets of an erection of $\mathfrak{M}^{(i)}$ such that each $X \in \mathrm{X}[i]$ is contained in some rank-r closed set. First we find the "covers" of each closed set in F_i. The covers of a closed set A of rank r are the sets obtained by adding one more element from E to A. The covers are generated with Generate-Covers(F, r, E).

Algorithm 4 KNUTH-MATROID(E, X)

Input: The ground set of elements E, and a list of enlargements X.

Output: The list of closed sets of the resulting matroid grouped by rank, $\mathrm{F} = [F_0, \ldots, F_r]$, where F_i is the set of closed sets of rank i.

```
1  r = 0, F = [{∅}]
2  while TRUE
3      PUSH!(F, GENERATE-COVERS(F, r, E))
4      F[r + 1] = F[r + 1] ∪ X[r + 1]
5      SUPERPOSE!(F[r + 1], F[r])
6      if E ∉ F[r + 1]
7          r ← r + 1
8      else
9          return (E, F)
```

GENERATE-COVERS(F, r, E)

```
1  return {A ∪ {a} : A ∈ F[r], a ∈ E \ A}
```

SUPERPOSE!(F_{r+1}, F_r)

```
 1  for A ∈ F_{r+1}
 2      for B ∈ F_{r+1}
 3          flag ← TRUE
 4          for C ∈ F_r
 5              if A ∩ B ⊆ C
 6                  flag ← FALSE
 7
 8          if flag = TRUE
 9              F_{r+1} ← F_{r+1} \ {A, B}
10              F_{r+1} ← F_{r+1} ∪ {A ∪ B}
```

Given no enlargements ($\mathrm{X}[i] = \emptyset$), the resulting matroid $\mathfrak{M}^{(i+1)}$ is the *free erection* of $\mathfrak{M}^{(i)}$, and there are no essential closed sets in F_{i+1}. Arbitrary matroids can be generated by supplying different lists X. When enlarging, the sets in $\mathrm{X}[r+1]$ are simply added to $\mathrm{F}[r+1]$, before SUPERPOSE! is run to ensure that the newly enlarged family of closed sets of rank $r+1$ is valid (i.e., in accordance with the closed set axioms given in Section 2.2). If F_{r+1} contains two sets A, B whose intersection $A \cap B \not\subseteq C$ for any $C \in F_r$ (in other words, their intersection is not a closed set), replace A, B with $A \cup B$. Repeat until no two sets exist in F_{r+1} whose intersection is not contained within some set $C \in F_r$.

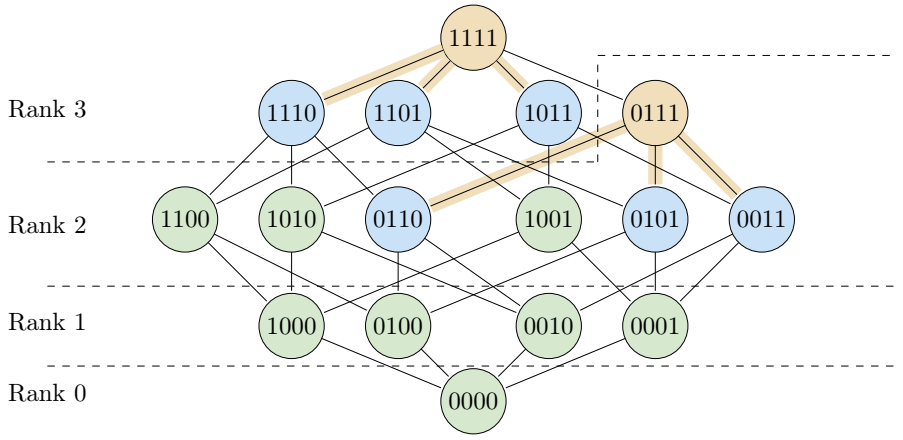


Figure 4.8: A matroid-under-construction. The set 0111 has just been assigned as a closed set of rank 2.

To cement our intuitive understanding of Knuth's matroid erection algorithm, we revisit the example from Chapter 2. Figure 4.8 shows the Hasse diagram of a matroid, the sets represented as binary strings where a 1-digit at position i signifies that element i is present in the set. A blue set is independent and a yellow set is dependent; a green set is both (i.e. both maximal and minimal for its rank). The edges marked in yellow signify the closure of a set. A set with no yellow edges going up and out of it is maximal for its rank (i.e., closed). The situation illustrated is in the second iteration of KMC. The set 0111 has just been designated as a closed set of rank 2.

Let the next element in X[2] be the set 1011—this is the next set we are designating as a rank-2 closed set. Figure 4.9 shows the situation after adding this. This situation is problematic, and we can think about why in several ways.

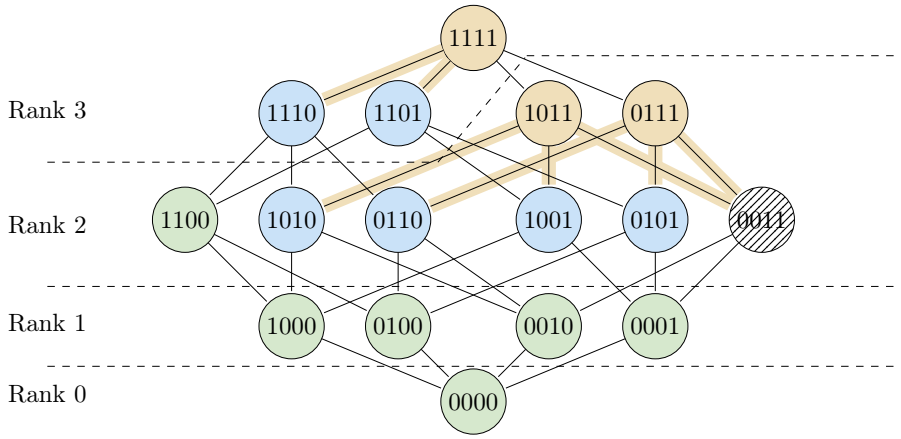


Figure 4.9: The set 1011 has been added as a rank-2 closed set in the matroid-under-construction from Figure 4.8. This is not a valid matroid.

Firstly, the intersection of these two closed sets, $1011 \cap 0111 = 0011$ (marked in grey) is not itself a closed set, breaking an axiom for the closed sets of a matroid. Secondly, the rank-2 independent set 0011 has two outgoing yellow edges—which represents the closure of 0011? Finally, consider the exchange property of independent sets: if S and T are independent sets with $|S| > |T|$, then there exists an element $g \in S \setminus T$ such that $T + g$ is independent (and of increased rank). The independent set 1101 is bigger than 0011, yet no element from the first can be added to the second to produce a bigger independent set. In the situation as it stands at this point, the independent set 0011 is a "blind alley" that cannot grow in rank, even though other independent sets of larger rank exist.

It is this situation that the SUPERPOSE! operation detects, and the two errant closed sets 1011 and 0111 are merged, replaced with $1011 \cup 0111 = 1111$ as a closed set of rank 2. When this happens, the algorithm terminates, as we have found the rank of the ground set, and all sets of size larger than 2 are dependent. The final matroid is given in Figure 4.10.

4.3.1 Randomized KMC

In the randomized version of KNUTH-MATROID, we generate matroids by applying a supplied number of random coarsening steps, instead of enlarging with

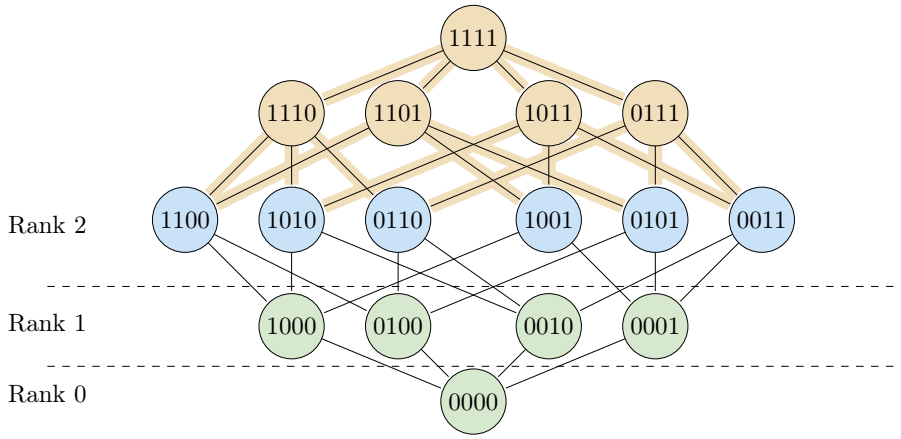


Figure 4.10: The uniform matroid U_4^2.

supplied sets. This is done by applying SUPERPOSE! immediately after adding the covers, then choosing a random member A of $\mathrm{F}[r+1]$ and a random element $a \in E \setminus A$, replacing A with $A \cup \{a\}$ and finally reapplying SUPERPOSE!. The parameter $p = (p_1, p_2, \ldots)$ gives the number of such coarsening steps to be applied at each iteration of the algorithm.

The pseudocode descriptions of Knuth's matroid construction hews closely to the initial Julia implementation. It should already be clear that this brute force approach leads to poor performance—for instance, the SUPERPOSE! method uses a triply nested for loop, which seems like a candidate for significant improvement. Section 4.3.2 describes the engineering work done to create a more performant implementation.

4.3.2 Improving performance

In the preparatory project to this thesis, I was able to recreate Knuth's table of observed mean values for the randomly generated matroids, but I was dismayed to find that my implementation was unable to handle matroids whose ground sets were even just a few elements larger. Considering that Knuth was able to run his experiments on the hardware available to him in the 1970s, I concluded that my implementation had room for improvement. Table 4.1 shows the performance of my first implementation. Even for rank-5 matroids of only 12 elements, this implementation is intolerably slow. For readability's sake, I have

Algorithm 5 RANDOMIZED-KNUTH-MATROID(E, p)

Input: The ground set of elements E, and a list $p = [p_1, p_2, ...]$, where p_r is the number of coarsening steps to apply at rank r in the construction.

Output: The list of closed sets of the resulting matroid grouped by rank, $\mathrm{F} = [F_0, \ldots, F_r]$, where F_i is the set of closed sets of rank i.

$r = 0, \mathrm{F} = [\{\emptyset\}]$
while TRUE
    PUSH!$(\mathrm{F}, $GENERATE-COVERS$(\mathrm{F}, r, E))$
    SUPERPOSE!$(\mathrm{F}[r+1], \mathrm{F}[r])$
    if $E \in \mathrm{F}[r+1]$ **return** (E, F)
    while $p[r] > 0$
        $A \leftarrow$ a random set in $\mathrm{F}[r+1]$
        $a \leftarrow$ a random element in $E \setminus A$
        replace A with $A \cup \{a\}$ in $\mathrm{F}[r+1]$
        SUPERPOSE!$(\mathrm{F}[r+1], \mathrm{F}[r])$
        if $E \in \mathrm{F}[r+1]$ **return** (E, F)
        $p[r] = p[r] - 1$
    $r = r + 1$

moved all lengty code snippets to Appendix D; the initial Julia implementation can be found in Figures D.1 and D.2. In this section, I describe some of the implementation decisions that were made to improve this performance.

The performance was measured using Julia's `@timed`[3] macro, which returns the time it takes to execute a function call, how much of that time was spent in garbage collection and the number of bytes allocated. The experiments was run on a 2021 MacBook Pro with the Apple M1 chip and 16GB RAM. As is evident from the data, larger matroids are computationally quite demanding to compute with this current approach, and the time and space requirements scales exponentially with n.

The tables in this section give the median time and memory performance, as well as average resulting matroid rank, of subsequent versions of my imple-

[3] https://docs.julialang.org/en/v1/base/base/#Base.@timed

n	$(p_1, p_2, \ldots)$	Trials	r	Time	Bytes allocated
10	(0, 6)	380	4.0	158.1ms	149.713 MiB
10	(0, 0, 6)	22	4.4	2.8s	2.481 GiB
12	(0, 7)	21	5.1	3.4s	2.773 GiB
12	(0, 0, 7)	2	5.0	43.9s	36.753 GiB

Table 4.1: Performance of `random_kmc_v1`.

mentation of Random-Knuth-Matroid. The experiments are set up with increasing values for n and p, in a manner to produce matroids of larger rank over larger ground sets. Each experiment is repeated for as many trials as can fit within one minute. The runtime of the random matroid generation functions varies quite a bit depending on which sets are chosen for coarsening, and so for situations where an experiment is run only a few times, the average time might be a bit misleading. The data is presented in order to show that the improvements described in this section are in fact meaningful optimizations.

Representing sets as binary numbers

The first improvement we will attempt is to represent our closed sets using one of Julia's `Integer` types of bit width at least n, instead of as a `Set`[4] of elements of E. The idea is to define a family of closed sets of the same rank as `Set{UInt16}`. Using `UInt16` we can support ground sets of size up to 16. Each 16-bit number represents a set in the family. For example, the set $\{2, 5, 7\}$ is represented by

$$164 = \text{0x00a4} = \text{0b0000000010100100} = 2^7 + 2^5 + 2^2.$$

At either end we have $\emptyset \equiv$ 0x0000 and $E \equiv$ 0xffff (if $n = 16$). The elementary set operations we will need have simple implementations using bitwise operations:

We can now describe the bitwise versions of the required methods. The bitwise implementation of Generate-Covers finds all elements in $E \setminus A$ by finding each value $0 \leq i < n$ for which `A & 1 << i === 0`, meaning that the set represented by `1 << i` is not a subset of A. The bitwise implementation of Superpose! is unchanged apart from using the bitwise set operations described above. The source code for these is given in Figure D.3.

[4] https://docs.julialang.org/en/v1/base/collections/#Base.Set

Set operation	Bitwise operation
$A \cap B$	A AND B
$A \cup B$	A OR B
$A \setminus B$	A AND NOT B
$A \subseteq B$	A AND $B = A$

Table 4.2: Set operations and their corresponding bitwise operations

n	$(p_1, p_2, \ldots)$	Trials	r	Time	Bytes allocated
10	(0, 6)	39975	3.85	1.5ms	1.646 MiB
10	(0, 0, 6)	6277	4.56	9.6ms	7.510 MiB
12	(0, 7)	7337	3.98	8.2ms	5.406 MiB
12	(0, 0, 7)	765	4.52	78.5ms	38.274 MiB
16	(0, 8)	89	5.31	674.3ms	103.595 MiB

Table 4.3: Performance of `random_kmc_v2`.

The performance of `random_kmc_v2` is shown in Table 4.3. It is clear that representing closed sets using binary numbers represents a substantial improvement—we are looking at performance increases on the order of 100x across the board.

Sorted superpose

Can we improve the running time of our implementation further? It is clear that SUPERPOSE! takes up a large portion of the compute time. In the worst case, when no enlargements have been made, F_{r+1} is the set of all $r+1$-sized subsets of E, $|F_{r+1}| = \binom{n}{r+1}$. Comparing each $A, B \in F_{r+1}$ with each $C \in F_r$ in a triply nested for loop requires $O(\binom{n}{r+1}^2 \binom{n}{r})$ operations. In the worst case, no enlargements are made at all, and we build the free matroid in $O(2^{3n})$ time (considering only the superpose step).

After larger closed sets have been added to $\mathrm{F}[r+1]$, SUPERPOSE! will cause sets to merge, so that only maximal dependent sets remain. Some sets will even simply disappear. In the case where $X = \{1, 2\}$ was added by GENERATE-COVERS, and the $Y = \{1, 2, 3\}$ was added manually as an enlargement, the

n	$(p_1, p_2, \ldots)$	Trials	r	Time	Bytes allocated
10	(0, 6)	19183	3.85	3.1ms	3.881 MiB
10	(0, 0, 6)	4009	4.53	15.0ms	14.562 MiB
12	(0, 7)	4722	3.97	12.7ms	10.781 MiB
12	(0, 0, 7)	613	4.52	97.9ms	60.336 MiB
16	(0, 8)	189	5.41	364.8ms	119.229 MiB
16	(0, 0, 8)	18	5.44	3.4s	717.923 MiB
20	(0, 9)	2	9.0	35.0	4.597 GiB

Table 4.4: Performance of `random_kmc_v3`.

smaller set will be fully subsumed in the bigger set, as $\{1,2\} \cap \{1,2,3\} = \{1,2\}$ (which is not a subset of any set in F[r]) and $\{1,2\} \cup \{1,2,3\} = \{1,2,3\}$. In this situation, Y would "eat" the covers $\{1,3\}$ and $\{2,3\}$ as well. This fact is reflected in the performance data – compare the memory allocation differences between the 10-element matroid with $p = [0,0,6]$ and the one with $p = [0,6,0]$ in any of the performance tables in this section. Making enlargements at earlier ranks result in smaller matroids as more sets get absorbed.

Since the larger sets will absorb so many of the smaller sets (around $\binom{p}{r+1}$, where p is the size of the larger set and $r+1$ is the size of the smallest sets allowed to be added in a given iteration), might it be an idea to perform the superpose operation in descending order based on the size of the sets? This should result in fewer calls to SUPERPOSE!, as the bigger sets will remove the smaller sets that fully overlap with them in the early iterations, however, the repeated sorting of the sets might negate this performance gain. This is the idea behind `random_kmc_v3`. The source code for the sorted superpose function is given in Figure D.4.

Unfortunately, as Table 4.4 shows, this implementation is not an especially significant improvement, though it performs somewhat better than the previous version on the later tests. What if we rethink the superpose operation more fundamentally?

Iterative superpose

The worst-case $O(\binom{n}{r+1}^2 \binom{n}{r})$ runtime of SUPERPOSE! at step r is due to the fact that it takes in F after all covers and enlargements have been indiscriminately

```
# Superpose (random_kmc_v4)
push!(F, Set()) # Add F[r+1].
while length(to_insert) > 0
  A = pop!(to_insert)
  push!(F[r+1], A)

  for B in setdiff(F[r+1], A)
    if should_merge(A, B, F[r])
      push!(to_insert, A | B)
      setdiff!(F[r+1], [A, B])
      push!(F[r+1], A | B)
    end
  end
end
```

Figure 4.11: `random_kmc_v4`: On-the-fly superposition.

added to $\mathrm{F}[r+1]$ and then loops through to perform the superposition. Might there be something to gain by inserting new closed sets into the current family one at a time, and superposing on the fly?

In `random_kmc_v4`, the full code of which can be found in Figure D.5, the covers and enlargements are not added directly to $\mathrm{F}[r+1]$, but to a temporary array `to_insert`. Each set A is then popped from `to_insert` one at a time, added to $\mathrm{F}[r+1]$ and compared with the other sets $B \in \mathrm{F}[r+1] \setminus \{A\}$ and $C \in \mathrm{F}[r]$ in the usual SUPERPOSE! manner. This results in fewer comparisons, as each set is only compared with the sets added before it; the first set is compared with no other sets, the second set with one other and the sets in $\mathrm{F}[r]$, and so on. The number of such comparisons is therefore given by the triangular number $T_{\binom{n}{r+1}}$, and so we should have roughly halved the runtime at step r. It is worth noting that this implementation of SUPERPOSE! uses a subroutine `should_merge` that returns early when it finds one set $C \in \mathrm{F}[r]$ such that $C \supseteq A \cap B$, so in practice it usually does not require $\binom{n}{r}$ comparisons in the innermost loop.

Table 4.5 shows that the iterative superpose was a meaningful improvement. For most input configurations, it is a few times faster and a few times less space demanding than `random_kmc_v2`.

n	$(p_1, p_2, \ldots)$	Trials	r	Time	Bytes allocated
10	(0, 6)	95004	4.54	631.6µs	438.354 KiB
10	(0, 0, 6)	26947	4.94	2.2ms	1.103 MiB
12	(0, 7)	18780	4.99	3.2ms	1.553 MiB
12	(0, 0, 7)	3438	5.14	17.5ms	5.210 MiB
16	(0, 8)	450	6.71	134.0ms	30.372 MiB
16	(0, 0, 8)	75	6.12	814.9ms	101.836 MiB
20	(0, 9)	9	8.78	10.0s	847.919 MiB
20	(0, 0, 9)	2	8.5	389.6s	9.446 GiB
24	(0, 10)	1	11.0	189.6s	9.709 GiB

Table 4.5: Performance of `random_kmc_v4`.

Rank table and non-redundant cover generation

Up to this point, our cover generation routine has not taken into account that any two sets of rank r will have at least one cover in common. To see this, consider a matroid-under-construction with $n = 10$ where $A = \{1, 2\}$ and $B = \{1, 3\}$ are closed sets of rank 2. Currently, GENERATE-COVERS will happily generate the cover $C = \{1, 2, 3\}$ twice, once as the cover of A and subsequently as the cover of B. Throughout this analysis, we will assume the worst case scenario of no enlargements, as any enlargements will strictly lower the number of sets in play at a given rank. In this case, $|\mathrm{F}[r]| = \binom{n}{r}$, and for each closed set A of rank r we are generating $|E \setminus A| = (n - r)$ covers, giving us a total of $\binom{n}{r}(n - r)$ covers generated at each rank r, including the duplicates. With no enlargements, we know that there are $\binom{n}{r+1}$ covers, and

$$\begin{aligned}(n-r)\binom{n}{r} &= \frac{n!(n-r)}{r!(n-r)!} \\ &= \frac{n!}{r!(n-r-1)!} \\ &= (r+1)\frac{n!}{(r+1)!(n-r-1)!} \\ &= (r+1)\binom{n}{r+1}.\end{aligned}$$

```
function generate_covers!(F, r, E, insert_fn)
  for y in F[r]
    t = E - y
    # Find all sets in F[r+1] that already contain y and remove excess elements
        from t.
    for x in F[r+1]
      if (x & y == y) t &= ~x end
      if t == 0 break end
    end
    # Insert y ∪ a for all a ∈ t.
    while t > 0
      x = y|(t&-t)
      insert_fn(x)
      t &= ~x
    end
  end
end
```

Figure 4.12: Non-redundant cover generation.

For each step r, we are generating $r + 1$ times as many covers as we need to. Over the course of all steps $0 \leq r \leq n$, we are generating

$$\sum_{r=0}^{n}(r+1) = \sum_{r=1}^{n+1} r = T_{n+1}$$

times the actual number of covers, where $T_{n+1} = \frac{(n+1)(n+2)}{2}$ is the triangular number. In other words, if we find a way to generate each cover only once, we will have shaved off an n^2 factor from the asymptotic complexity of our implementation.

When generating covers, `random_kmc_v6` improves upon the brute force cover generation described above by only adding the covers

$$\Big\{A \cup \{a\} : A \in \mathrm{F}[r], a \in E \setminus A, a \notin \bigcup\big\{B : B \in \mathrm{F}[r+1], A \subseteq B\big\}\Big\}.$$

In other words, we find the covers of A, that is, the sets obtained by adding one more element a from E to A, but we do not include any a that is to be found in another, already added, cover B that contains A. This solves the problem described above; the cover $\{1, 2, 3\} = B \cup \{2\}$ will not be generated, as $2 \in C$ and $B \subseteq C$. Figure 4.12 shows an implementation of this.

We have extracted the iterative superpose logic described above into its own function `add_set!` to allow it to be performed on a cover-per-cover basis. This

function is given in Figure 4.13. This code is very different from the previous version, primarily in its use of a *rank table*. The rank table is a dictionary mapping a closed set in F to its assigned rank. This solves the problem that, while SUPERPOSE! was getting more and more efficient, it was still performing the same comparisons over and over again.

In the previous versions, after adding the closed sets for a rank, SUPERPOSE! was run to maintain the closed set properties of the matroid (given in Section 2.2). These were maintained by ensuring that, for any two newly added sets $A, B \in \mathrm{F}[r+1]$, there exists $C \in \mathrm{F}[r]$ such that $A \cap B \subseteq C$. This was ensured by checking if the intersection of each such A, B is contained in a set C of rank r. We remember that one of the properties of the closed sets of a matroid is that the intersection of two closed sets is itself a closed set. Therefore, we do not need to find a closed set C of rank r that *contains* $A \cap B$, since if A and B are indeed closed sets, their intersection will be *equal* to some closed set C of any rank $\leq r$. This insight leads us to the idea of the rank table: if we keep track of all added closed sets in a rank table, then we can memoize SUPERPOSE! and replace the innermost loop with a constant time dictionary lookup.

`add_set!` accepts a closed set X we are interested in assigning to rank r. It then loops through all existing rank-r closed sets Y, ensuring that X is either added as a closed set or merged with some existing closed set. When comparing a tentatively-closed set X with an existing closed set Y, the function will encounter one of three possible situations:

1. $X \cap Y$ is a closed set (it has an entry in the rank table) of rank less than r. Move on to the next closed set Y.

2. $X \cap Y$ is a closed set (it has an entry in the rank table) of rank equal to r. This will for instance happen when $Y \subseteq X$. Remove Y as a closed set and call `add_set!` on $X \cup Y$.

3. $X \cap Y$ has not been observed until this point. This happens when closed sets have been added of lower rank but similar cardinality, thus obscuring the rank of their subsets. Some additional checks are required:

   (a) If $|X \cap Y| < r$, we know that the rank of $X \cap Y$ is less than r. Move on to the next closed set Y.

   (b) Otherwise, we need to see if $X \cap Y \subseteq Z$ for some Z of lower rank, so we do the manual subset equality check with the lower-ranked closed set, as familiar from earlier implementations of SUPERPOSE!.

```
function add_set!(x, F, r, rank, callback)
  for y in F[r+1]
    if haskey(rank, x&y) && rank[x&y]<r
      continue
    end

    if !haskey(rank, x&y)
      if Base.count_ones(x&y) < r
        continue
      else
        r_ = check_rank(x&y, r, F)

        if r_ !== false
          rank[x&y] = r_
          continue
        end
      end
    end

    # x ∩ y has rank > r, replace with x ∪ y.
    setdiff!(F[r+1], y)
    add_set!(x|y, F, r, rank, callback)
    return
  end

  push!(F[r+1], x)
  callback(x) # Sets rank[x] = r.
end

function check_rank(v, r, F)
  for (i, Fi) in enumerate(F[1:r]), z ∈ Fi
    if v&z == v
      return i-1
    end
  end

  return false
end
```

Figure 4.13: `add_set!` adds a new closed set and runs the superpose logic.

n	$(p_1, p_2, \ldots)$	Trials	r	Time	Bytes allocated
10	(0, 6)	230040	4.03	260.9µs	27.092 KiB
10	(0, 0, 6)	59597	4.23	1.0ms	38.474 KiB
12	(0, 7)	39753	5.03	1.5ms	68.219 KiB
12	(0, 0, 7)	7390	5.15	8.1ms	139.560 KiB
16	(0, 8)	511	8.01	117.5ms	741.442 KiB
16	(0, 0, 8)	61	8.03	998.8ms	1.981 MiB
20	(0, 9)	5	11.0	15.0s	14.217 MiB
20	(0, 0, 9)	1	11.0	108.8s	17.949 MiB
24	(0, 10)	1	14.0	690.4s	75.663 MiB

Table 4.6: Performance of `random_knuth_matroid` (version 5).

Table 4.6 shows the performance of `random_knuth_matroid`, which is the fifth and final version discussed in this chapter. Comparing this with the initial performance depicted in Table 4.1, we can see that we are now able to generate somewhat larger matroids, on the order of a thousand times faster.

4.3.3 Finding the properties of erected matroids

The fact that Knuth-Matroid fully enumerates all closed sets of the matroid as it erects it rank by rank begs the question: can we build the other families of sets for the matroids alongside the closed sets? In Appendix C, I describe an extension of Knuth-Matroid that also fully enumerates $\mathcal{I}$ and $\mathcal{C}$ for $\mathfrak{M}$ when n is small enough. Sadly, this approach does not scale well for larger values of n, as the size of these sets undergoes a combinatorial explosion as n increases.

Determining matroid properties post-erection

In a 1989 paper, Greene introduces the concept of *descriptive sufficiency* [22]. A subcollection of closed sets of a matroid is descriptively sufficient if it can be used to identify the fundamental properties of the matroid using certain easily applied conditions. The collection of all closed sets of a matroid is one descriptively sufficient such collection.

Rank function. With every closed set of a matroid in hand, finding the rank of a set S is simply a matter of finding the closed set F of least rank such that $S \subseteq F$.

```
function rank(M::ClosedSetsMatroid, S::Integer)
  for (r, Fr) in enumerate(M.F), B ∈ Fr
      if S&B == S return r-1 end
  end
end
```

Indepence oracle. To check if a set S is independent, we compare it with the closed sets of rank $|S| - 1$. If S is indeed independent, it cannot be a subset of a closed set of lower rank, so if we find one such set we return false. Otherwise, S is independent.

```
function is_indep(M::ClosedSetsMatroid, S::Integer)
  t = Base.count_ones(S)

  if t > length(M.F) return false end

  for F in M.F[t]
    if S&F==S return false end
  end

  return true
end
```

Closure function. Determining the closure of a set S in this case is the exact same procedure as finding the rank: the closed set F of least rank such that $S \subseteq F$ is the closure of S.

```
function closure(M::ClosedSetsMatroid, S::Integer)
  for Fr in M.F, B ∈ Fr
      if S&B == S return B end
  end
end
```

Chapter 5

Using the library

At this point in the report, I have described how Matroids.jl implements the functionality required for the empirical study of matroidal fair allocation algorithms. The previous chapters detailed three such algorithms—Envy-Induced-Transfers, AlgMMS and Yankee-Swap—and described how Matroids.jl exposes the functions needed to implement these. Subsequently, I showed how Matroids.jl represents and randomly generates matroids. With random matroids and the functional requirements listed in Table 3.1 in hand, it is time to put the library to the test, and investigate whether it in fact des enable the implementation and empirical study of matroidal fair allocation algorithms.

This chapter serves a proof of concept for Matroids.jl, demonstrating the library's ability to facilitate the implementation and evaluation of matroidal fair allocation algorithms. As such, I will, in addition to describing the implementation of the algorithms, provide some experimental results regarding the fairness of the allocations produced. These algorithms are well-understood, so while the experimental results in this chapter may not represent novel findings, they serve as verification of the library's intended functionality, from random matroid generation, through fair allocation, to fairness evaluation. They should be considered proof that Matroids.jl is a tool that enables a new workflow for working programmatically with matroids in fair allocation, successfully extending Allocations.jl with capabilities to run previously inaccessible algorithms. The successful development of this tool is the main contribution of this thesis.

5.1 Implementing Envy-induced transfers

The pseudocode and high-level description of Envy-Induced Transfers can be found in Section 3.2. In this section, I will give a step-by-step explanation of my implementation of the algorithm.

The function accepts an instance of a fair allocation problem with matroid-rank valuations, represented with the `MatroidRank` struct defined in Section 3.1. Initially, a clean, MAX-USW allocation A is found using the matroid partitioning algorithm whose implementation, `matroid_partition_knuth73`, is described in Section B. This function returns a tuple $(A, junk)$, where $junk$ is the set of goods that did not fit into any independent bundle and is disregarded (Envy-Induced-Transfers preferring cleanness over completeness). The partition is converted into an instance of `Allocation`, which is provided by Allocations.jl. Envy-Induced-Transfers continues until no agent envies another for more than one good; the envy between each pair of agents i, j is represented with an $i \times j$-matrix `envy` such that $\texttt{envy}[i,j] = v_i(A_j) - v_i(A_i)$ holds agent i's envy towards agent j. During each iteration a pair of agents i, j is found such that $\texttt{envy}[i,j] > 1$. Then, a good $g \in A_j$ with $\Delta_i(A_i, g)$ is transferred from A_j to A_i. This is the envy-induced transfer from which the algorithm derives its name. After the transfer, the envy table is updated to reflect the new allocation.

The full implementation of Envy-Induced-Transfers is given in Figure 5.1. This implementation highlights an important optimization when working programmatically with matroids, which is to not use the rank function on a set that is known to be independent. As seen in the implementations given in Chapter 4, the rank function is expensive; `rank(m::GraphicMatroid, S)`, for instance, runs Kruskal's algorithm as a subroutine, giving a time complexity of $O(|S| \lg |S|)$ for finding the rank of S. When S is independent, the rank is $|S|$—the size of a set can be found in constant time using `length(S)`[1].

The concept of a *loop invariant* is useful for reasoning about the correctness of an algorithm, and can in this case be used to rigorously show that each bundle A_i is independent in $\mathfrak{M}_i$ throughout the procedure. A loop invariant is a property that is true before the loop starts (initialization), remains true at the start of each iteration (maintenance) and is true upon termination [34]. Envy-Induced-Transfers has a loop invariant stating that the allocation A is clean, or, equivalently, that each A_i is independent in $\mathfrak{M}_i$. This is true on

[1] From the Julia source code (base/set.jl): `length(s::Set) = length(s.dict)` [9]. A set, though itself unindexable, is represented behind the scenes in Julia as a dictionary, which is indexable and hence has a lastindex field, thus allowing the constant time length computation.

```
function alloc_eit_bciz21(V::MatroidRank; partition=nothing)
  n = na(V); m = ni(V)

  if partition === nothing
    # Compute a clean, MAX-USW allocation.
    (partition, _junk) = matroid_partition_knuth73(V.Ms)
  end

  A = Allocation(n, m)
  for (i, bundle) in enumerate(partition)
    give!(A, i, bundle)
  end

  # Envy table envy[i,j] holds i's envy towards j, v_i(A_j) - v_i(A_i).
  envy = zeros(Int, n, n)
  for i in 1:n, j in 1:n
    # We use length when we know the bundles are independent.
    envy[i,j] = value(V, i, bundle(A, j)) - length(bundle(A, i))
  end

  # While there are agents i, j st i envies j more than 1...
  i,j = argmax(envy) |> Tuple
  while envy[i,j] > 1
    # Find item in A_j with marginal gain for i.
    for g in bundle(A,j)
      if Δ(V, A, i, g) == 1
        # Envy-induced transfer:
        deny!(A, j, g)
        give!(A, i, g)

        # Update D.
        for k in 1:n
          envy[i, k] = value(V, i, bundle(A, k)) - length(bundle(A, i))
          envy[k, i] = value(V, k, bundle(A, i)) - length(bundle(A, k))
          envy[j, k] = value(V, j, bundle(A, k)) - length(bundle(A, j))
          envy[k, j] = value(V, k, bundle(A, j)) - length(bundle(A, k))
        end

        break
      end
    end

    i,j = argmax(envy) |> Tuple
  end

  return A
end
```

Figure 5.1: The Matroids.jl implementation of Envy-Induced-Transfers

initialization: MATROID-PARTITION produces a clean, MAX-USW allocation. Each iteration, the algorithm a good g such that $\Delta_i(A_i, g)$ is transferred from A_j to A_i for some agents i, j. Since A_j is independent, $A_j - g$ is independent due to the hereditary property. Similarly, due to the exchange property, $A_i + g$ is also independent, and the loop invariant is maintained. The algorithm terminates when it has reached EF1, and A is returned as-is, a clean allocation. This proves that it is a valid optimization to use `length` instead of `rank` when finding $v_i(A_i)$ in the implementation of this algorithm. Notice that `value` (which in turn uses `rank`—refer to Section 3.1 for details) is still used when checking $v_i(A_j)$ for $i \neq j$.

A bundle will not necessarily be independent in another agent's matroid, hence `rank` calls are still required when checking $v_i(A_j)$. To understand the effect of replacing half the `rank` calls (the ones on sets known to be independent) with calls to `length`, I ran the following simple experiment:

1. Generate six random graphic matroids with 256 goods:
   `m1 = GraphicMatroid(erdos_renyi(rand(128:512), 256))`
2. Precompute the initial partition:
   `(p, _) = matroid_partition_knuth73([m1,m2,m3,m4,m5,m6])`
3. Run `@btime alloc_eit_bciz21(V; partition=p)` with calls to `length` where applicable
4. Run `@btime alloc_eit_bciz21(V; partition=p)` with only calls to `value`

On average, the version that always called `value` took 181.5ms to compute, whereas the optimized version needed only 97.625ms. It is clear that the calls to `value` takes up a significant portion of the runtime of the function, and replacing half of them with a constant-time call to `length` is therefore a substantial improvement, shaving off roughly half the runtime.

5.2 Implementing AlgMMS

The next algorithm I implement in order to demonstrate the capabilities of Matroids.jl is ALGMMS. Refer to Section 3.2 for the pseudocode and high-level description of the algorithm. The full source code of my implementation is given in Figure 5.2.

```
function alloc_algmms_bv21(V::MatroidRank)
  n = na(V); m = ni(V)

  # Compute a clean, (partial) MAX-USW allocation.
  (partition, junk) = matroid_partition_knuth73(V.Ms)
  A = Allocation(n, m)
  for (i, bundle) in enumerate(partition)
    give!(A, i, bundle)
  end

  # Compute MMS of each agent.
  mmss = [mms_i(V, i) for i in 1:n]

  S_less = Set([i for i in 1:n if value(V, i, A) < mmss[i]])
  S_more = Set([i for i in 1:n if value(V, i, A) > mmss[i]])

  D = exchange_graph(V.Ms, A)

  while length(S_less) > 0
    # i is an agent with less than their maximin share.
    i = popfirst!(collect(S_less))

    # The goods for which i has positive marginal value.
    F_i = [g for g in 1:m if is_indep(V.Ms[i], bundle(A, i) ∪ g)]
    A_more = reduce(∪, [bundle(A, j) for j in S_more])

    transfer_path = find_shortest_path(D, F_i, A_more)
    @assert transfer_path !== nothing

    j = owner(A, transfer_path[end]) # The losing agent.

    transfer!(V.Ms, D, A, i, transfer_path)

    # Only i and j have received a new value.
    for k in [i,j]
      if value(V,k,A) < mmss[k] push!(S_less, k) else setdiff!(S_less, k) end
      if value(V,k,A) > mmss[k] push!(S_more, k) else setdiff!(S_more, k) end
    end
  end

  # Give agent 1 any unallocated items (these are 0-valued by everyone).
  give!(A, 1, junk)
  return A
end
```

Figure 5.2: The Matroids.jl implementation of AlgMMS

The function accepts an instance of a fair allocation problem with matroid-rank valuations, and outputs a clean, MAX-USW, MMS-fair allocation. If not passed in the optional parameter `partition`, an initial clean and MAX-USW

allocation A is computed with `matroid_partition_knuth73`, identically to how `alloc_eit_bciz21` starts off. The maximin share μ_i is computed for each agent i, using the function `mms_i`, whose implementation is given in Figure 3.5. Next, the setup phase, the algorithm finds `S_less`, the set of agents whose bundle value in A is less than their maximin share, and `S_more`, the set of agents whose bundle value is higher. Finishing off the setup phase, the exchange graph of the allocation is produced with the function `exchange_graph` from Figure 3.9.

Each iteration, an agent i such that $A_i < \mu_i$ is popped from `S_less`. The set of goods g such that $\Delta_i(A_i, g) = 1$ is computed, this is the set `F_i`. A shortest path is found from the goods in `F_i` to the goods belonging to agents in `S_more` using `find_shortest_path`, which either returns a transfer path or `nothing` if none could be found. In the paper describing AlgMMS, Barman and Verma show that such a path always exist as long as someone has less than their maximin share. A new allocation is acquired by calling `transfer!`, passing it the list of matroids, the exchange graph, the allocation, the agent whose bundle value is about to improve and the transfer path. `transfer!` updates the exchange graph and allocation in place (hence the exclamation mark—a Julia convention). `S_less` and `S_more` are updated for the agents whose bundle value has changed. When `S_less` is empty, the while loop terminates, the first agent is granted the goods that were not allocated by `matroid_partition_knuth73` and the resulting allocation is returned.

`alloc_algmms_bv21` is another example showing that it is often unnecessary to compute the actual value (entailing an expensive call to `rank`) of a bundle. Notice that, when finding the list of positively marginal-valued good F_i for agent i, the Δ_i function is not chosen for the task. The algorithm has the same loop invariant as Envy-Induced-Transfers regarding the cleanness of A at every step of the algorithm; thus, F_i is simply the list of goods such that $A_i + g$ is independent.

5.3 Implementing Yankee Swap

Yankee-Swap is the third and final algorithm whose implementation will serve as a proof of the capabilities of Matroids.jl. Yankee-Swap differs from the previous two algorithms in that it does not start out with a clean, MAX-USW allocation procured with Matroid-Partition, instead adding a new "agent zero", whose bundle is the pot of unallocated items. Agent zero is represented with a `ZeroMatroid`, the special case of the uniform matroid in which only the empty set is independent. This ensures that the goods allocated to agent zero

are sinks on the graph, with no out-edges.

The source code for my implementation of YANKEE-SWAP is given in Figure 5.3. The function proceeds similarly to ALGMMS. Each iteration, a least-fortunate, highest-priority agent i is chosen from among the agents whose bundle value can still improve (denoted with the `flag` array). The set F_i of goods for which i has positive marginal value is found, and a transfer path is produced between F_i and the set of goods currently belonging to agent zero (the unallocated goods). The next allocation is produced by augmenting along the transfer path and the next iteration starts if some agent can still improve their bundle; otherwise it terminates.

5.4 Running some experiments

At this point, it seems like Matroids.jl has achieved its goal of making it easy to implement matroidal fair allocation algorithms. The arcane matroid logic is safely hidden away behind semantically named functions in the library's, allowing the developer to focus on the business logic of the algorithm at hand. Of course, the real purpose of Matroids.jl is to enable the empirical study of matroidal fair allocation algorithms. Let us now turn our attention to how an experimental setup might be implemented. In this section, we will use the functionality available to us in the current, proof-of-concept version of Matroids.jl to investigate how the three algorithms we have implemented perform on a range of fairness criteria. These results, though probably not terribly interesting in and of themselves, should serve to illustrate that Matroids.jl does in fact live up to its purpose in life; namely, to be a practical, empirical tool to be used alongside the abundant kit of theoretical tools afforded by matroid theory.

The experimental plan is as follows:

1. Generate a matroid-rank-valued fair allocation instance (according to some matroid generation scheme) with n matroids over m elements.
2. Find an allocation using some allocation algorithm.
3. Check the resulting allocations against some set of approximate fairness notions.
4. Repeat steps 1-3 k times and present the average results.

```
function alloc_yankee_swap_vz22(V::MatroidRank)
  n = na(V); m = ni(V)

  # Randomly prioritize the agents.
  agents = shuffle(1:n)

  # Agent "0" (n+1) has a corresponding zero matroid.
  Ms_ = [V.Ms..., ZeroMatroid(m)]

  A = Allocation(n+1, m)
  give!(A, n+1, 1:m) # The bundle of unallocated items.
  flag = falses(n)

  D = exchange_graph(Ms_, A)

  while false in flag
    # The agents whose bundle can still improve.
    T = [i for i in agents if flag[i] == false]

    # Find the agents in T with minimim value.
    T_vals = [(i, length(bundle(A, i))) for i in T]
    min_val = minimum(last, T_vals)
    T_ = [i for (i, v) in T_vals if v == min_val]

    # The highest priority agent with minimum value.
    i = T_[1]

    # The goods for which i has positive marginal value.
    F_i = [g for g in 1:m if is_indep(V.Ms[i], bundle(A, i) ∪ g)]

    #  a shortest path from F_i to an unallocated good.
    A_0 = [g for g in 1:m if owner(A, g) == n+1]
    transfer_path = find_shortest_path(D, F_i, A_0)

    # Transfer if path exists.
    if transfer_path !== nothing
      transfer!(Ms_, D, A, i, transfer_path)
    else
      flag[i] = true
    end
  end

  return A
end
```

Figure 5.3: The Matroids.jl implementation of Yankee-Swap

```
function gen_matroidrank_profile(n, gen_matroid, T)
  function gen()
    ms = Array{T}(undef, n)

    Threads.@threads for i in 1:n
      ms[i] = gen_matroid()
    end

    return MatroidRank(ms, ms[1].n)
  end

  return gen
end
```

Figure 5.4: Function to initialize a random matroid-rank valuation profile, given a random matroid generator

Figure 5.4 shows a Julia function that accepts a function that generates a matroid, and returns a function that constructs a matroid-rank valuation profile with n thusly generated matroids. Due to the time-consuming nature of matroid generation, the matroids are generated in parallel[2]. By plugging in a function for generating random graphs with m edges, as given in Figures 4.1, 4.2 and 4.3, we can generate a valuation profile consisting of graphic matroids. Alternatively, we can pass in `random_knuth_matroid` to generate a profile of matroids given via their closed sets representations.

ID	n	m	Matroid type	Avg. rank	Time	Bytes allocated
1	4	24	Graphic (ER)	21.5	63.0µs	33.467 KiB
2	4	24	Graphic (WS)	13.4	64.7µs	20.869 KiB
3	4	24	Graphic (BA)	14.3	57.2µs	24.183 KiB
4	4	24	Knuth $((3, 8, 5, 3))$	5.5	16.8s	15.428 MiB
5	4	24	Knuth $((0, 15, 6))$	4.2	4.2s	5.019 MiB
6	4	24	Knuth $((0, 12, 6))$	6.3	38.2s	21.501 MiB

Table 5.1: Six instance generation schemes with 4 agents and 24 goods.

[2]Depending on how many threads are available. Julia starts single-threaded by default, but supports multi-threading [9].

Table 5.1 shows six schemes for initializing a matroid-rank valued fair allocation instance. Each scheme generates 4 matroids, one per agent, over 24 goods. The first three set up graphic matroids, using Erdős-Rényi, Barabási-Albert and Watts-Strogatz model random graphs, respectively. The latter three use RANDOM-KNUTH-MATROID, with three different coarsening configurations. Each scheme was followed to generate 100 fair allocation problem instances. Then, each of the three algorithms we are studying were used to find an allocation for each instance. Finally, the fairness measures implemented in Chapter 3 were used to evaluate the fairness of each allocation. The output is Tables 5.2-5.4, giving the average approximate ($\alpha-$) fairness of the allocations output by the algorithms per scheme.

This setup illustrates how one might use Matroids.jl to investigate the fairness guarantees of an allocation algorithm. Of course, one would probably want to put a bit more thought behind exactly what schemes to use for initializing instances. The instances used in this example do not seem to pose significant challenges for the algorithms, as indicated by the fact that all allocations meet, on average, almost all the fairness criteria they are evaluated against ($\alpha \geq 1$).

One somewhat interesting observation from these experimental results is that Yankee Swap consistently delivers MMS-fair allocations on all tested schemes, despite its theoretical worst-case guarantee of only $\frac{1}{2}$-MMS fairness. This raises the question: to what extent does Yankee Swap typically exceed this worst-case guarantee in practical applications? This needs to be more rigorously tested than has been done in this thesis, but a positive answer of this kind would be useful for reasoning about the real-world applicability of Yankee Swap.

ID	α-EF1	α-EFX$_0$	α-PROP	α-PROP1	α-PROPX$_0$	α-MMS
1	1.188	1.178	1.004	1.206	1.004	0.99
2	1.1	1.033	1.236	1.661	1.236	0.916
3	1.116	1.049	1.311	1.743	1.311	0.93
4	1.093	1.064	3.945	12.0	3.945	1.013
5	1.021	1.008	3.903	Inf	3.903	1.01
6	1.114	1.02	3.357	8.999	3.357	1.072

Table 5.2: Fairness results; `alloc_eit_bciz21` on 100 random problem instances according to the schemes given in Table 5.1.

ID	α-EF1	α-EFX$_0$	α-PROP	α-PROP1	α-PROPX$_0$	α-MMS
1	1.2	1.2	1.003	1.205	1.003	1.0
2	1.2	1.2	1.3	1.794	1.3	1.0
3	1.2	1.2	1.32	1.778	1.32	1.0
4	1.078	1.036	3.907	11.786	3.907	1.035
5	1.03	1.006	3.921	Inf	3.921	1.029
6	1.117	1.042	3.358	8.831	3.358	1.074

Table 5.3: Fairness results; `alloc_algmms_bv21` on 100 random problem instances according to the schemes given in Table 5.1.

ID	α-EF1	α-EFX$_0$	α-PROP	α-PROP1	α-PROPX$_0$	α-MMS
1	1.2	1.2	1.004	1.206	1.004	1.0
2	1.2	1.2	1.273	1.699	1.273	1.0
3	1.2	1.2	1.32	1.778	1.32	1.0
4	1.197	1.116	3.977	12.68	3.977	1.137
5	1.263	1.2	4.338	Inf	4.338	1.248
6	1.199	1.161	3.407	8.342	3.407	1.091

Table 5.4: Fairness results; `alloc_yankee_swap_vz22` on 100 random problem instances according to the schemes given in Table 5.1.

Chapter 6

Discussion

This thesis has sought to answer the research question: how might one design and implement a Julia library to support programmatic experimentation with matroidal fair allocation algorithms? In an effort to answer this, I pursued several lines of inquiry. I implemented classic matroid operations, such as the matroid union and the exchange graph, and found practical ways to measure the fairness of a matroid-rank-valued allocation. I discussed the requirements of three recent algorithms in this space, later implementing these as a proof-of-concept. Lastly, I described how to represent and randomly generate matroids. With all this in hand, I gave an example of an experimental setup and provided some cursory results.

The primary goal of this thesis was to build Matroids.jl as an answer to that question—a practical tool to complement the theoretical toolkit provided by matroid theory. An important sub-goal was to figure out how to make Matroids.jl performant; as we have seen, matroids permit many powerful polynomial-time operations, such as the matroid union, that papers on matroidal fair allocation use in their analyses to show that allocations can be found efficiently. This obscures many implementation-level optimization decisions that can drastically improve the practical runtime of the implemented algorithms. One example of such an optimization is to use the rank function as sparingly as possible, in favor of cheaper independence or cardinality checks, as I discuss when giving some algorithm implementations in Chapter 5.

6.1 Limitations and future work

The algorithms that were chosen for study (Envy-Induced-Transfers, AlgMMS and Yankee-Swap), were chosen due their being relatively short and sweet, simple to reason about, and presenting a manageable amount of requirements to Matroids.jl. As we have seen, they are similar in spirit; all three produce clean, MAX-USW allocations, the first two starting with a matroid union call, the latter two using exchange graphs and transfer paths to modify the allocation. Consequently, Matroids.jl extracts this common logic into separate functions, to be optimized and reasoned about independently. A current limitation of Matroids.jl is that it is geared heavily towards supporting algorithms that follow this general structure—other hypothetical algorithms that take a completely different approach would likely require the implementation of other matroid operations.

Back in Chapter 2, I briefly mentioned matroidal constraints as the second major use case for matroids in the context of fair allocation. In order to maintain a manageable scope for this thesis, I elected to focus on matroid-rank valuations instead. For completeness' sake, a next step in the development of Matroids.jl should be to hook matroids into the constraint framework provided by Allocations.jl.

This thesis describes two ways of generating matroids: as graphic matroids, using different random graph models, or via the erection of arbitrary matroids. Table 5.1 shows the difference in performance when generating graphic matroids versus arbitrarily erected matroids of a similar size ($m = 24$). Using graphic matroids, we could generate matroids of rank 13,14, and 21, hundreds of thousands of times faster than we were able to erect matroids of only rank 4,5, and 6. This might lead one to assume that graphic matroids are always preferable. To some extent this might be so, but a limitation of graphic matroids is that of representability. While every simple graph describes a matroid in terms of its acyclic subsets of edges, not every matroid can be expressed as a simple graph. Supplementary goods, for instance, would require a multigraph with two parallel edges. Little support exists in Julia for working with multigraphs at the moment. Further, matroids might be hypergraphs (graphs in which edges can connect more than two nodes) or might not lend themselves to a graphic representation at all. These are matroid qualities that are readily representable with our arbitrary erected matroids. Thus, we can see that there is a use for other matroid types as well, in addition to graphic matroids. In addition, different matroid types might have different properties that allow one to make different guarantees regarding fairness in a matroidal setting. As such, an important piece of future work on

Matroids.jl is to add support for more types of matroids. This thesis describes two ways of generating matroids: as graphic matroids, using different random graph models, or via the erection of arbitrary matroids. Table 5.1 shows the difference in performance when generating graphic matroids versus arbitrarily erected matroids of a similar size ($m = 24$). Using graphic matroids, we could generate matroids of rank 13,14, and 21, hundreds of thousands of times faster than we were able to erect matroids of only rank 4,5, and 6. This might lead one to assume that graphic matroids are always preferable. To some extent this might be so, but a limitation of graphic matroids is that of representability. While every simple graph describes a matroid in terms of its acyclic subsets of edges, not every matroid can be expressed as a simple graph. Supplementary goods, for instance, would require a multigraph with two parallel edges. Little support exists in Julia for working with multigraphs at the moment. Further, matroids might be hypergraphs (graphs in which edges can connect more than two nodes) or might not lend themselves to a graphic representation at all. These are matroid qualities that are readily representable with our arbitrary erected matroids. Thus, we can see that there is a use for other matroid types as well, in addition to graphic matroids. Further, different matroid types might have different properties that allow one to make different guarantees regarding fairness in a matroidal setting. As such, an important piece of future work on Matroids.jl is to add support for more types of matroids.

There are numerous avenues available for improving the performance of Matroids.jl. One example is the exchange graph and related functionality. During a Yankee Swap run, the exchange graph is rebuilt, and the shortest paths recalculated from scratch, each iteration. This is likely a big performance drain, as most of the graph will not change between iterations, only the edges out of nodes corresponding to goods on the transfer path. One suggested performance improvement is to store the shortest path from every node to every other node on the exchange graph, though this might be a speed-memory usage tradeoff and should be considered carefully. Another interesting consideration is that of parallelization. I use multithreading in the matroid-rank-valued instance generator, given in Figure 5.4, but it stands to reason that there might be room for parallelization within the matroid erection code, for instance. This could be a substantial source of performance improvement.

The matroid erection functionality, though significantly faster after the improvements outlined in Section 4.3.2, should preferably be made even faster. The latest implementation, whose performance is shown in Table 4.6, is a lot more memory efficient than earlier versions, but we can see that the runtime still explodes as n grows. While it should be an interesting project to further

optimize this implementation of Knuth's matroid construction, we might also consider whether this approach really scales any further with n. At the end of the day, the procedure considers in the worst case all subsets of E—as $n = |E|$ grows, the number of subsets of E undergoes a combinatorial explosion, and quickly becomes intractably large. Any further improvements should begin to restrict which subsets the function looks at. We know, due to Greene [22], that there exist families of closed sets, smaller than the full family of all closed sets of a matroid, that are descriptively sufficient (meaning they can be used to determine the properties of a matroid with simple and efficient algorithms). One such is the family of essential closed sets, i.e., the closed sets whose "closedness" do not follow from the closed sets of lower rank. These are roughly encoded by the enlargements passed to KNUTH-MATROID. Might it be possible to implement Knuth's matroid construction algorithm in such a manner as to only keep track of these essential closed sets? Would such an implementation be able to generate larger matroids, and faster, than the current implementation? These are interesting lines of questioning for future work into the matroid generation capabilities of Matroids.jl.

Another direction that one might take this work going forward, is to investigate the applicability of this product to real-world problems. Matroids are compelling structures to work with in the context of fair allocation, as they are pleasant to reason about and permit strong theoretical fairness guarantees. Unfortunately, people rarely think of their preferences in terms of matroids. The example given in Chapter 1 highlights that matroids do have real-world applicability in modelling user preferences, but design and engineering ingenuity would be needed to build a robust system that can reliably map between the two.

6.2 Concluding remarks

Initially, all I knew about this thesis was that it was going to have something to do with fair allocation. Looking around for recent allocation algorithms, the study of which might form part of a thesis, Viswanathan and Zick's Yankee Swap algorithm caught my attention. By restricting the valuations to the class of matroid rank functions, a seemingly simple algorithm could deliver extraordinarily well on a range of fairness objectives intractable in the general, additive case. My interest piqued, I wanted to understand how it worked, and set about implementing the algorithm using Hummel and Hetland's Allocations.jl library. Almost immediately I was flummoxed by how to represent the matroid rank valuations. I had assumed that there would exist some library to facilitate working

programmatically with matroids, similar to how Graphs.jl enables working with graphs without needing to reinvent the wheel graph. At the time I was unable to find any such library, and so the research question for this thesis came to be: how might one design and implement a Julia library to support the implementation of and experimentation with matroidal fair allocation algorithms?

Late in the project, I realized that there does in fact exist matroid libraries in Julia, in all likelihood vastly more performant and feature-rich than Matroids.jl would ever be[1]. The primary target demographic for Matroids.jl had always been fair allocation researchers, but upon witnessing the capabilities of my more advanced competitors, a secondary target demographic came to the fore, namely students, computer programmers and non-mathematicians such as myself. All along, I realized, I had been building the library for myself, the library that I had needed when I wanted to figure out how Yankee Swap worked, which was a simple-to-use matroid library that only concerned itself with the most basic aspects of matroids as they related to fair allocation.

While matroid theory might seem an extremely abstract and niche subfield of mathematics, it has found applicability in the field of fair allocation, which in the end deals with problems of a highly practical and everyday nature. The aim of fair allocation, to deliver provably fair mechanisms for the distribution of resources, is a noble goal, and if a problem permits a matroidal representation, it can utilize algorithms that deliver very well indeed. If Matroids.jl is able to serve as a soft introduction to matroid theory for a computer programmer interested in understanding fair allocation algorithms, and if that computer programmer goes on to build a real-world solution for fair allocation, then Matroids.jl has achieved its goals as far as I am concerned.

[1] See for instance https://docs.oscar-system.org/stable/Combinatorics/matroids/.

Bibliography

[1] N. Benabbou, M. Chakraborty, A. Igarashi, and Y. Zick, "Finding fair and efficient allocations when valuations don't add up," in *Algorithmic Game Theory*, Springer International Publishing, 2020, pp. 32–46. DOI: 10.1007/978-3-030-57980-7_3. [Online]. Available: https://doi.org/10.1007%2F978-3-030-57980-7_3.

[2] R. J. Aumann and M. Maschler, "Game theoretic analysis of a bankruptcy problem from the Talmud," *Journal of Economic Theory*, vol. 36, no. 2, pp. 195–213, 1985, ISSN: 0022-0531. DOI: https://doi.org/10.1016/0022-0531(85)90102-4. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0022053185901024.

[3] H. Steinhaus, "The problem of fair division," *Econometrica*, no. 16, pp. 101–104, 1948.

[4] G. Amanatidis *et al.*, *Fair division of indivisible goods: A survey*, 2022. arXiv: 2208.08782 [cs.GT].

[5] A. Biswas and S. Barman, "Fair division under cardinality constraints," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18, Stockholm, Sweden: AAAI Press, 2018, pp. 91–97, ISBN: 9780999241127.

[6] S. Barman and P. Verma, "Existence and computation of maximin fair allocations under matroid-rank valuations," in *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*, ser. AAMAS '21, Virtual Event, United Kingdom: International Founda-

tion for Autonomous Agents and Multiagent Systems, 2021, pp. 169–177, ISBN: 9781450383073.

[7] N. Benabbou, M. Chakraborty, A. Igarashi, and Y. Zick, "Finding fair and efficient allocations for matroid rank valuations," *ACM Trans. Econ. Comput.*, vol. 9, no. 4, Oct. 2021, ISSN: 2167-8375. DOI: 10.1145/3485006. [Online]. Available: https://doi.org/10.1145/3485006.

[8] V. Viswanathan and Y. Zick, *Yankee swap: A fast and simple fair allocation mechanism for matroid rank valuations*, 2023. arXiv: 2206.08495 [cs.DS].

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. [Online]. Available: https://doi.org/10.1137/141000671.

[10] M. L. Hetland and H. Hummel, *Allocations.jl*, version 0.1, Nov. 2022. [Online]. Available: https://github.com/mlhetland/Allocations.jl.

[11] D. E. Knuth, "Random matroids," *Discrete Mathematics*, vol. 12, pp. 341–358, 4 1975.

[12] H. H. Crapo and G.-C. Rota, *On the foundations of combinatorial theory: Combinatorial geometries*. M.I.T. Press, 1970.

[13] H. Whitney, "On the abstract properties of linear dependence," *American Journal of Mathematics*, vol. 57, pp. 509–533, 3 Jul. 1935.

[14] A. Schrijver, *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.

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

[16] I. Caragiannis, D. Kurokawa, H. Moulin, A. D. Procaccia, N. Shah, and J. Wang, "The unreasonable fairness of maximum nash welfare," *ACM Trans. Econ. Comput.*, vol. 7, no. 3, Sep. 2019, ISSN: 2167-8375. DOI: 10.1145/3355902. [Online]. Available: https://doi.org/10.1145/3355902.

[17] B. Plaut and T. Roughgarden, "Almost envy-freeness with general valuations," *SIAM Journal on Discrete Mathematics*, vol. 34, no. 2, pp. 1039–1068, 2020. DOI: 10.1137/19M124397X. eprint: https://doi.org/10.1137/19M124397X. [Online]. Available: https://doi.org/10.1137/19M124397X.

[18] E. Budish, "The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes," *Journal of Political Economy*, vol. 119, no. 6, pp. 1061–1103, Dec. 2011. DOI: 10.1086/664613. [Online]. Available: https://doi.org/10.1086/664613.

[19] J. Edmonds, "Matroid partition," in *50 Years of Integer Programming 1958-2008*, Springer Berlin Heidelberg, Nov. 2009, pp. 199–217. DOI: 10.1007/978-3-540-68279-0_7. [Online]. Available: https://doi.org/10.1007/978-3-540-68279-0_7.

[20] D. Knuth, *Matroid partitioning* (Report (Stanford University. Computer Science Department)). Computer Science Department, Stanford University, 1973.

[21] H. H. Crapo, "Erecting geometries," *Annals of the New York Academy of Sciences*, vol. 175, no. 1, pp. 89–92, Jul. 1970. DOI: 10.1111/j.1749-6632.1970.tb56458.x. [Online]. Available: https://doi.org/10.1111/j.1749-6632.1970.tb56458.x.

[22] T. Greene, "Descriptively sufficient subcollections of flats in matroids," *Discrete Mathematics*, vol. 87, no. 2, pp. 149–161, 1991, ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(91)90044-3. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0012365X91900443.

[23] D. G. Kelly and D. Kennedy, "The higgs factorization of a geometric strong map," *Discrete Mathematics*, vol. 22, no. 2, pp. 139–146, 1978, ISSN: 0012-365X. DOI: https://doi.org/10.1016/0012-365X(78)90121-8. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0012365X78901218.

[24] R. Pendavingh and J. van der Pol, *Enumerating matroids of fixed rank*, 2016. arXiv: 1512.06655 [math.CO].

[25] M. Babaioff, T. Ezra, and U. Feige, "Fair and truthful mechanisms for dichotomous valuations," in *AAAI Conference on Artificial Intelligence*, 2020.

[26] J. Fairbanks, M. Besançon, S. Simon, J. Hoffiman, N. Eubank, and S. Karpinski, *JuliaGraphs/Graphs.jl: An optimized graphs package for the julia programming language*, 2021. [Online]. Available: https://github.com/JuliaGraphs/Graphs.jl/.

[27] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959. DOI: 10.1007/bf01386390. [Online]. Available: https://doi.org/10.1007/bf01386390.

[28] S. E. Fienberg, "A brief history of statistical models for network analysis and open challenges," *Journal of Computational and Graphical Statistics*, vol. 21, no. 4, pp. 825–839, 2012. DOI: 10.1080/10618600.2012.738106. eprint: https://doi.org/10.1080/10618600.2012.738106. [Online]. Available: https://doi.org/10.1080/10618600.2012.738106.

[29] P. Erdős and A. Rényi, "On random graphs I.," *Publicationes Mathematicae Debrecen*, vol. 6, no. 3-4, pp. 290–297, Jul. 1959. DOI: 10.5486/pmd.1959.6.3-4.12. [Online]. Available: https://doi.org/10.5486/pmd.1959.6.3-4.12.

[30] E. N. Gilbert, "Random Graphs," *The Annals of Mathematical Statistics*, vol. 30, no. 4, pp. 1141–1144, 1959. DOI: 10.1214/aoms/1177706098. [Online]. Available: https://doi.org/10.1214/aoms/1177706098.

[31] S. Janson, D. E. Knuth, T. Luczak, and B. G. Pittel, "The birth of the giant component," *Random Struct. Algorithms*, vol. 4, pp. 233–359, 1993.

[32] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Rev. Mod. Phys.*, vol. 74, pp. 47–97, 1 Jan. 2002. DOI: 10.1103/RevModPhys.74.47. [Online]. Available: https://link.aps.org/doi/10.1103/RevModPhys.74.47.

[33] D. J. Watts and S. H. Strogatz, "Collective dynamics of 'small-world' networks," *Nature*, vol. 393, no. 6684, pp. 440–442, Jun. 1998. DOI: 10.1038/30918. [Online]. Available: https://doi.org/10.1038/30918.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (The MIT Press), 3rd ed. London, England: MIT Press, Jul. 2009.

[35] R. Fourquet, *BitIntegers.jl*, https://github.com/rfourquet/BitIntegers.jl, Accessed: 2023-05-20.

[36] N. Wirth and C. A. R. Hoare, "A contribution to the development of ALGOL," *Commun. ACM*, vol. 9, no. 6, pp. 413–432, Jun. 1966, ISSN: 0001-0782. DOI: 10.1145/365696.365702. [Online]. Available: https://doi.org/10.1145/365696.365702.

Appendices

Appendix A

Sets as numbers – some useful tricks

Section 4.3.2 details a number of steps taken in order to build a performant Julia implementation of Knuth-Matroid. Perhaps chief among these steps in terms of sheer performance gain compared to the initial, naïve implementation, was the transition from representing subsets of E as a `Set` of integers (or whatever type the elements of E might have), to representing them as a single integer, whose 1-bits denote which elements are in the set. This is possible as long as n is less than the widest available integer type (in off-the-shelf Julia, 128 bits, though one can go wider with the help of libraries [35]). We reiterate the bitwise equivalents of the basic set operations in Table A.1. These bitwise equivalents

Table A.1: Set operations and their equivalent bitwise operations

Set operation	Bitwise equivalent
$A \cap B$	$A \wedge B$
$A \cup B$	$A \vee B$
$A \setminus B$	$A \wedge \neg B$
$A \subseteq B$	$A \wedge B = A$

allow us to perform the set operations in constant time, resulting in significant

performance increases. In the code snippets included throughout Section 4.3.2 and Appendix D, a number of "tricks" are performed with bitwise operations whose workings and purpose might be a bit obtuse. This appendix came to be as I got to grips with working with sets in this manner.

How do I...

...create a singleton set?

The left-shift operator (<<) can be used to set the ith bit to 1 and the others to 0. In general, $\{a\} = 1 << a$. This is used in an early version of GENERATE-COVERS:

```
function generate_covers_v2(F_r, n)
  Set([A | 1 << i for A ∈ F_r for i in 0:n-1 if A & 1 << i === 0])
end
```

...find the smallest element of a set?

Using the two's complement of a set T, denoted by $-T = \neg T + 1$, we can find the smallest element with the operation $T \wedge -T$. This is used in the next trick.

...enumerate all elements of a set one by one?

Using the previous trick, we can repeatedly pop the smallest element in the following manner:

```
  t = 0b11111111
  while t > 0
    x = t&-t  # x is the singleton set consisting of the smallest element of t
    output(x)
    t &= ~x   # t = t setminus x
  end
```

This outputs all numbers from 1 to 0xff with a Hamming weight of 1.

...get a random element from a set?

We find all the positions at which the reversed bitstring of the set has a '1' character, and choose a random one.

```
function rand_el(S::Integer)
  x = rand([2^(i-1) for (i,c) in enumerate(reverse(bitstring(S))) if c == '1'])
  return convert(typeof(S), x)
end
```

...convert a set to its bitwise representation?

Assuming that the sets start counting at 1 (useful for interacting with Graphs.jl, for instance), this can be achieved with the following function:

```
function set_to_bits(set, T=UInt64)
  if length(set) == 0 return T(0) end
  T(reduce(+, (2^(x-1) for x in set), init=0))
end
```

...convert back?

```
function bits_to_set(bits)
  Set(i for (i, c) in enumerate(reverse(bitstring(bits))) if c == '1')
end
```

Appendix B

Matroid partitioning

A common pattern for matroid-rank-valued fair allocation algorithms is, as we have seen, to initialize an allocation that corresponds to a maximum-sized independent set in the union of the matroids in play. This allocation must needs be MAX-USW and clean, but any other fairness notions are not guaranteed. The algorithms then exploit the exchange property of independent sets to massage the allocation into one that has the desired properties, in polynomial time. This being such a widespread approach, Matroids.jl should include the functionality for initializing such an allocation.

This procedure is referred to as both the matroid union algorithm and the matroid partitioning algorithm in the literature, since the task of finding a maximum-size independent set in a union of n matroids over E is equivalent to finding an n-partition of E, such that each part i is independent in the ith matroid $\mathfrak{M}_i$. In this thesis, I refer to the procedure as MATROID-PARTITION, following the example of Knuth, who in a 1973 paper describe the algorithm that will be implemented in this section. In the paper, Knuth expresses the problem in the following manner:

> If $\mathfrak{M}_1, \ldots, \mathfrak{M}_k$ are matroids defined on a finite set E, [find] whether or not the elements-of-E can be colored with k colors such that (i) all elements of color j are independent in $\mathfrak{M}_j$, and (ii) the number of elements of color j lies between given limits, $n_j \leq |E_j| \leq n'_j$. [20]

If such a coloring exists, the algorithm produces it, otherwise it finds a proof to the contrary. While the terminology differs, we can see that this is in fact a

```
function matroid_partition_knuth73(Ms, floors=nothing)
  n = Ms[1].n; k = length(Ms)
  S0 = Set(1:n) # The unallocated items.
  S = [Set() for _ in 1:k] # The partition-to-be.
  color = Dict(x=>0 for x in 1:n) # color[x] = j iff x ∈ S[j].
  for y in 1:k color[-y] = y end # -y is the 'standard' element of color y.
  succ = [0 for _ in 1:n]

  floors = floors === nothing ? [0 for _ in 1:k] : floors

  # Ensure every part gets at least its lower limit.
  for j in 1:k, i in 1:floors[j]
    augment!(j, n, k, Ms, S, S0, succ, color)
  end

  # Allocate the rest.
  while S0 != Set()
    X = augment!(0, n, k, Ms, S, S0, succ, color)

    if length(X) != 0
      return (S, X)
    end
  end

  return (S, Set())
end
```

Figure B.1: `matroid_partition_knuth73`

colorful way of asking a question about a fair allocation instance with matroid-rank valuations. The ground set of goods E we are already familiar with. There are k agents, each of whom has her own color. A good is "colored" j if it is allocated to agent j—we can picture each agent equipped with a can of spray paint they use to denote their goods. The question we are asking then, is whether we can find a clean allocation of the goods, such that each bundle is between certain size limits. Knuth shows that we can find the answer to this question in $O(n^3 + n^2k)$ calls to the independence oracle.

The Matroids.jl implementation of Matroid-Partition is given in Figure B.1. The main action happens in the `augment!` subroutine, whose implementation is given in Figure B.2.

```
function augment!(r, n, k, Ms, S, S0, succ, color)
  for x in 1:n succ[x] = 0 end

  A = Set(1:n)
  B = r > 0 ? Set(-r) : Set(-j for j in 1:k)

  while B != Set()
    C = Set()
    for y ∈ B, x ∈ A
      j = color[y]

      if x ∉ S[j] && is_indep(Ms[j], x ∪ setdiff(S[j], y))
        succ[x] = y
        A = setdiff(A, x)
        C = C ∪ x
        if color[x] == 0

          # x is uncolored - transfer:
          while x ∈ 1:n
            y = succ[x]
            j = color[x]

            if j == 0 setdiff!(S0, x) else setdiff!(S[j], x) end

            j = color[y]
            S[j] = S[j] ∪ x
            color[x] = j
            x = y
          end

          return Set()
        end
      end
    end
    B = C
  end

  # We did not find a transfer path to 0.
  return setdiff(A, reduce(∪, S))
end
```

Figure B.2: The `augment!` subroutine in `matroid_partition_knuth73`

Appendix C

Enumerating circuits and independent sets during erection

In his 1974 paper [11], Knuth includes an ALGOL W [36] implementation that also enumerates all circuits and independent sets for the generated matroid[1].

Matroids.jl includes an implementation of this, called `random_erect`—an extension of `random_kmc_v6` that finds $\mathcal{I}$ and $\mathcal{C}$ by pre-populating the rank table with all subsets of E. The full source code for `random_erect` is given in Figure C.2. Covers are generated and sets inserted in the same manner as in `random_kmc_v6`. After all covers and enlargements have been inserted and superposed (so F$[r+1]$ contains the closed sets of rank $r+1$), a new function, `mark!` is called on each closed set. This function recursively assigns the cardinality (i.e., the Hamming weight, the number of 1s in the binary digit representing that set) to the entry for each subset of the closed set in the rank table. When a subset whose cardinality equals the current rank is found, we have encountered an independent set, and it is added to the family of independent sets. A final loop through all subsets of E finds each circuit, using the new `unmark!` function to ensure only the necessary functions are checked. The function returns the `FullMatroid` struct, which is a `ClosedSetsMatroid` that also holds $\mathcal{I}$ and $\mathcal{C}$.

Fully enumerating matroids in this manner allows really efficient implementations of `is_indep` and `is_circuit`, as we know all the independent sets and

[1] A later implementation in C called ERECTION.W can be found at his home page: https://www-cs-faculty.stanford.edu/~knuth/programs/erection.w

```
function mark!(m, I, r, rank)
  if haskey(rank, m) && rank[m] <= r
    return
  end
  if rank[m] == 100+r push!(I[r+1], m) end
  rank[m] = r
  t = m
  while t != 0
    v = t&(t-1)
    mark!(m-t+v, I, r, rank)
    t = v
  end
end

function unmark!(m, card, rank, mask)
  if rank[m] < 100
    rank[m] = card
    t = mask-m
    while t != 0
      v = t&(t-1)
      unmark!(m+t-v, card+1, rank, mask)
      t=v
    end
  end
end
```

Figure C.1: `mark!` and `unmark!`

circuits ahead of time. However, this approach sadly scales poorly for larger values of n, as it has to allocate bytes for every subset of E. The number of subsets it has to find the Hamming weight of undergoes a combinatorial explosion and quickly becomes intractably large, even on modern hardware.

```
function random_erect(n, p, T=UInt16)
  # Initialize.
  r = 1
  pr = 0
  F::Vector{Set{T}} = [Set(T(0))]
  E::T = big"2"^n-1
  rank = Dict{T, UInt8}()

  # Populate rank table with 100+cardinality for all subsets of E.
  k=1; rank[0]=100;
  while (k<=E)
    for i in 0:k-1 rank[k+i] = rank[i]+1 end
    k=k+k;
  end

  F = [Set(0)] # F[r] is the family of closed sets of rank r-1.
  I = [Set(0)] # I[r] is the family of independent sets of rank r-1.
  rank[0] = 0

  while E ∉ F[r]
    push!(F, Set())
    push!(I, Set())

    # Generate minimal closed sets for rank r+1.
    for y in F[r] # y is a closed set of rank r.
      t = E - y # The set of elements not in y.
      # Find all sets in F[r+1] that already contain y and remove excess
          elements from t.
      for x in F[r+1]
        if (x & y == y) t &= ~x end
      end
      # Insert y ∪ a for all a ∈ t.
      while t > 0
        x = y|(t&-t)
        insert_set!(x, F, r, rank)
        t &= ~x
      end
    end
```

Figure C.2: `random_erect` fully enumerates the independent sets and circuits for a random matroid during erection (continued on the next page).

```
    if r <= length(p)
      # Apply coarsening.
      pr = p[r]
      while pr > 0 && E ∉ F[r+1]
        A = rand(F[r+1])
        t = E-A
        one_element_added::Vector{T} = []
        while t > 0
          x = A|(t&-t)
          push!(one_element_added, x)
          t &= ~x
        end
        Acupa = rand(one_element_added)
        setdiff!(F[r+1], A)
        insert_set!(Acupa, F, r, rank)
        pr -= 1
      end
    end

    # Assign rank to sets and add independent ones to I.
    for m in F[r+1]
      mark!(m, I, r, rank)
    end

    # Next rank.
    r += 1
  end

  C = Set()
  k = 1
  while k <= E
    for i in 0:k-1 if rank[k+i] == rank[i]
      push!(C, T(k+i))
      unmark!(k+i, rank[i], rank, E)
    end end
    k += k
  end

  return FullMatroid{T}(n, r-1, F, I, C, rank, T)
end
```

Figure C.2 continued.

Appendix D

The development of `random_kmc`

This final appendix is where I have stowed away the lengthier bits of code referred to in Section 4.3.2, detailing the development of a somewhat performant implementation of Random-Knuth-Matroid (Algorithm 5).

```
function generate_covers_v1(Fr, E)
  Set([A ∪ a for A ∈ Fr for a ∈ setdiff(E, A)])
end

function superpose_v1!(F, F_old)
  for A ∈ F, B ∈ F
    should_merge = true
    for C ∈ F_old if A ∩ B ⊆ C
      should_merge = false
    end end

    if should_merge
      setdiff!(F, [A, B])
      push!(F, A ∪ B)
    end
  end

  return F
end
```

Figure D.1: Initial implementation `generate_covers` and `superpose`.

```
function random_kmc_v1(n, p, T)
  E = Set([i for i in range(0,n-1)])

  # Step 1: Initialize.
  r = 1
  F = [family([])]
  pr = 0

  while true
    # Step 2: Generate covers.
    push!(F, generate_covers_v1(F[r], E))

    # Step 4: Superpose.
    superpose_v1!(F[r+1], F[r])

    # Step 5: Test for completion.
    if E ∈ F[r+1]
      return KnuthMatroid{Set{Integer}}(n, F, [], Set(), Dict())
    end

    if r <= length(p)
      pr = p[r]
    end

    while pr > 0
      # Random closed set in F_{r+1} and element in E \ A.
      A = rand(F[r+1])
      a = rand(setdiff(E, A))

      # Replace A with A ∪ {a}.
      F[r+1] = setdiff(F[r+1], A) ∪ Set([A ∪ a])

      # Superpose again to account for coarsening step.
      superpose_v1!(F[r+1], F[r])


      # Step 5: Test for completion.
      if E ∈ F[r+1]
        return (E, F)
      end

      pr -= 1
    end


    r += 1
  end
end
```

Figure D.2: `random_kmc_v1`

```
function bitwise_superpose!(F, F_prev)
  i = 0
  As = copy(F)
  while length(As) !== 0
    A = pop!(As)

    for B in setdiff(F, A)
      i += 1
      if should_merge(A, B, F_prev)
        push!(As, A | B)
        setdiff!(F, [A, B])
        push!(F, A | B)
        break
      end
    end
  end

  return F
end

function generate_covers_v2(F_r, n)
  Set([A | 1 << i for A ∈ F_r for i in 0:n-1 if A & 1 << i === 0])
end
```

Figure D.3: The bitset-implementations of GENERATE-COVERS and SUPERPOSE!, first used in `random_kmc_v2`.

```
function sorted_bitwise_superpose!(F, F_prev)
  As = sort!(collect(F), by = s -> length(bits_to_set(s)))
  while length(As) !== 0
    A = popfirst!(As)

    for B in setdiff(F, A)
      if should_merge(A, B, F_prev)
        insert!(As, 1, A | B)
        setdiff!(F, [A, B])
        push!(F, A | B)
        break
      end
    end
  end

  return F
end
```

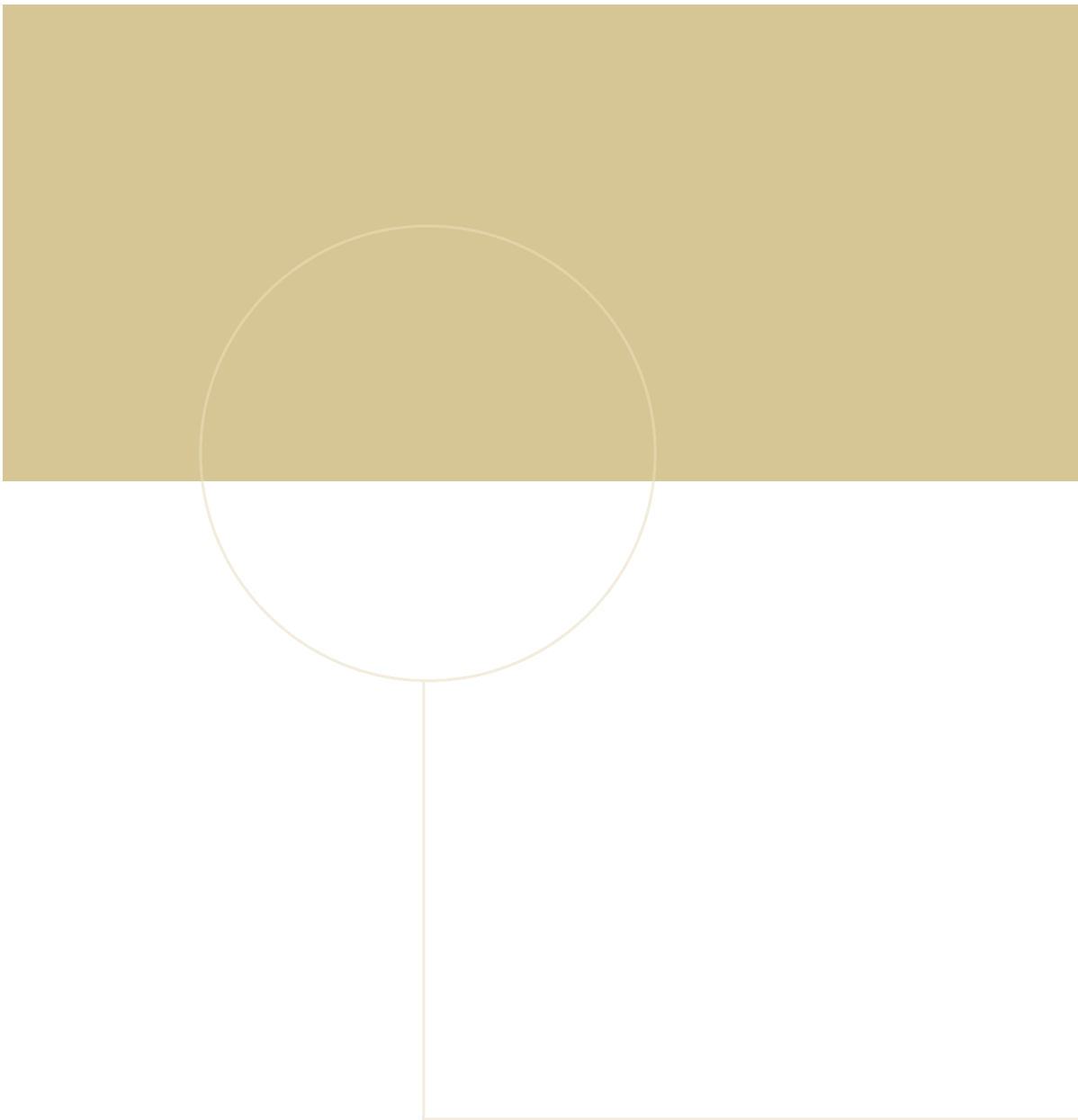
Figure D.4: This implementation of SUPERPOSE! sorts the sets by length.

```
function random_kmc_v4(n, p, T=UInt16)::ClosedSetsMatroid{T}
  r = 1
  pr = 0
  F = [Set(0)]
  E = 2^n - 1 # The set of all elements in E.

  while true
    to_insert = generate_covers_v2(F[r], n)

    # Apply coarsening to covers.
    if r <= length(p) && E ∉ to_insert # No need to coarsen if E is added.
      pr = p[r]
      while pr > 0
        A = rand(to_insert)
        a = random_element(E - A)
        to_insert = setdiff(to_insert, A) ∪ [A | a]
        pr -= 1
      end
    end

    # Superpose.
    push!(F, Set()) # Add F[r+1].
    while length(to_insert) > 0
      A = pop!(to_insert)
      push!(F[r+1], A)

      for B in setdiff(F[r+1], A)
        if should_merge(A, B, F[r])
          push!(to_insert, A | B)
          setdiff!(F[r+1], [A, B])
          push!(F[r+1], A | B)
        end
      end
    end

    if E ∈ F[r+1]
      return ClosedSetsMatroid{T}(n, r, F, Dict(), T)
    end

    r += 1
  end
end
```

Figure D.5: `random_kmc_v4` inserts the sets one at a time, superposing on the fly.

```
function random_knuth_matroid(n, p, T=UInt16)::ClosedSetsMatroid{T}
  r = 1
  F::Vector{Set{T}} = [Set(T(0))]
  E::T = BigInt(2)^n-1
  rank = Dict{T, Integer}(0=>0)

  while E ∉ F[r]
    # Initialize F[r+1].
    push!(F, Set())

    # Setup add_set.
    add_callback = x -> rank[x] = r
    add_function = x -> add_set!(x, F, r, rank, add_callback)

    generate_covers!(F, r, E, add_function)

    # Perform coarsening.
    if r <= length(p) coarsen!(F, r, E, p[r], add_function) end

    r += 1
  end

  return ClosedSetsMatroid{T}(n, r-1, F, rank, T)
end
```

Figure D.6: The final implementation of Random-Knuth-Matroid.



NTNU

Norwegian University of
Science and Technology