NTNU

Department of Computer Science

TDT4900 — Master's thesis

Exploring Matroids in Fair Allocation: Building the Matroids.jl Library

Author:
Andreas Aaberge Eide

Supervisor:
Magnus Lie Hetland

May 30, 2023

In matroid theory, we explore
A structure with properties galore
From bases to circuits
It never discourages
Mathematicians always wanting more

The axioms it holds are so grand
And its applications vast and unplanned
From optimization to graphs
It can solve so many tasks
Matroid theory, truly a wonderland

ChatGPT

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 problem with a colleague from the computer science department who immediately lessens your fears by informing you that your problem 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 [6].

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 [3]), and remains so today. The mathematical study of fair allocation started with a seminal work by Steinhaus in 1948 [27], 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 [2]. 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 model a student's willingness (value of 1) or unwillingness (value of 0) to enroll in a course. The diminishing returns property allow us to model 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 [4, 5, 6, 7, 29] 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 fill this gap, this thesis proposes Matroids.jl, a library for the Julia programming language [8], which extends the existing Allocations.jl library [20] with the functionality required to enable the empirical study of matroidal fair allocation algorithms.

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 4, I show how this API can be used to implement Viswanathan and Zick's Yankee Swap algorithm [29] and some other algorithms for matroid-rank-valued fair allocation. In Chapter 5, 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 [22], the successful implementation of which was a significant sub-goal of the project. In Chapter 6, I provide some experimental results for the algorithms over different matroid types. Finally, in Chapter 7, 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 [13]

Matroids were first introduced by Hassler Whitney in 1935 [31], 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; for instance, matroid rank functions are a natural way of formally describing course allocation for students [7].

2.1 Fair allocation

In this thesis, I use $[k]$ to denote the set $\{1, 2, \ldots, k\}$. To ease readability, I abuse notation a bit and replace $A \cup \{g\}$ and $A \setminus \{g\}$ with $A + g$ and $A - g$, respectively.

An instance of a fair allocation problem consists of a set of agents $\mathcal{N} = [n]$ and a set of m goods $\mathcal{G} = \{g_1, g_2, \ldots, g_m\}$. Each agent has a valuation function $v_i : 2^{\mathcal{G}} \to \mathbb{R}^+$; $v_i(A)$ is the value agent i ascribes to the bundle of goods A. The marginal value of agent i for the good g, given that she already owns the bundle A, is given by $\Delta_i(A, g) := v_i(A+g) - v_i(A)$. 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 $A \subset \mathcal{G}$ and $g \in \mathcal{G}$

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

Any function v_i adhering to these properties precisely determine a matroid [26]. There are many other ways to characterize matroids, some of which are given in Section 2.

2.1.1 Envy-freeness

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. 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. An allocation A is EF iff, for all agents $i, j \in \mathcal{N}$,

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

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) [24], which allows for the envy of up to the value of one (highest-valued) good. A is an EF1 allocation iff, for all agents $i, j \in \mathcal{N}$, 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. [10] requires that this least valued good be positively valued. We call this fairness objective EFX_+. A is an EFX_+ allocation iff, for all agents $i, j \in \mathcal{N}$,

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

Plaut and Roughgarden [25], 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 iff, for all agents $i, j \in \mathcal{N}$,

$$v_i(A_i) \geq v_i(A_j - g), \ \forall g \in A_j. \tag{EFX_0}$$

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) iff each agent $i \in \mathcal{N}$ 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, ie.

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

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 [2]. An allocation is PROP1 iff there for each agent $i \in \mathcal{N}$ exists some good $g \in \mathcal{G} \setminus A_i$ that, if given to i, would ensure that agent i received her proportional share; that is,

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

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 iff 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.

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

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

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

Maximin share fairness

Budish introduces a relaxation of proportionality known as *maximin share fairness* (MMS) [9], in which the threshold for each agent i is her *maximin share*, defined as the maximum value she could receive if she partitioned $\mathcal{G}$ among all agents and then picked the worst bundle. Let $\mathcal{A}_n(\mathcal{G})$ be the family of all possible allocations of the goods in $\mathcal{G}$ to the agents in $\mathcal{N}$. The maximin share for agent i, denoted by MMS_i, is given by

$$\text{MMS}_i := \max_{B \in \mathcal{A}_n(\mathcal{G})} \min_{A \in B} v_i(A).$$

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

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

2.1.3 Efficiency

As mentioned in the introduction, 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. I enumerate the three most commonly discussed in the literature:

1. **Egalitarian social welfare (ESW):** The ESW of an allocation A is given by the minimum value of an agent. $\text{ESW}(A) = \min_{i \in \mathcal{N}} v_i(A_i)$.

2. **Utilitarian social welfare (USW):** The USW of an allocation is given by the sum of agent values. $\text{USW}(A) = \sum_{i \in \mathcal{N}} v_i(A_i)$.

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. $\mathrm{NW}(A) = \prod_{i \in \mathcal{N}} v_i(A_i)$.

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 rule is that if an allocation with agent values $(0, 0, 100)$ is MAX-ESW, then so is the allocation with agent values $(0, 0, 0)$. A stricter version of MAX-ESW 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.

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 of this, the terms used in matroid theory are borrowed from analogous concepts in both graph theory and linear algebra. In this section, I will describe a few ways to characterize a matroid, using the axiom systems given by Whitney in his original paper.

2.2.1 Characterization via independent sets

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 $2^E \setminus \mathcal{I}$.

In practice, the ground set E represents the universe of elements in play, and the independent sets of typically represent the legal combinations of these items. In the context of fair allocation, the independent sets represent the legal (in the case of matroid constraints) or desired (in the case of matroid utilities) bundles of items.

A matroid is an independence system with the following properties [31]:

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

(2) If $A, B \in \mathcal{I}$ and $|A| > |B|$, then there exists $e \in A - B$ such that $B + e \in S$.

(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.

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,

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

2.2.2 Characterization via bases

When $S = E$, (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*. 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) above). A theorem by Whitney [31] 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 $e \in B$, then for some $e' \in B'$, $B - e + e' \in \mathcal{B}$.

2.2.3 Characterization via circuits

A *circuit* is a minimal dependent set of a matroid – it is an independent set plus one "redundant" element. Equivalently, the collection of circuits of a matroid is given by

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

A set is independent if and only if it contains no circuit [26], and so a matroid is uniquely determined by the collection of its circuits. The following conditions characterize $\mathcal{C}$ [31]:

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.

2.2.4 Characterization via closed sets

We also need to establish the concept of the *closed sets* 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 [22]:

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

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

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 $|S| - r(S)$, ie. the number of elements that must be removed from S to obtain an independent set.

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

Babaioff 2021 – PE mechanism – non-redundant Lorenz dominating allocation always exists and is MNW, MAX-USW, leximin, EFX (EF1) and .5-MMS. With a randomized prioritization it is ex ante EF and ex ante proportional as well. Yankee Swap computes prioritized non-redundant Lorenz dominating allocations.

In a fair allocation instance with matroid-rank valuations, each agent i has a corresponding matroid $\mathfrak{M}_i = (\mathcal{M}, \mathcal{I}_i)$, where $\mathcal{M}$ (the set of goods) is the ground set of elements common to all agents' matroids.

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. Suksompong [28] gives the 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 can be modeled with a partition matroid, and 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 5 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 [20], 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 (which, incidentally, are partition matroid constraints [4]). Our goal is to 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.

This chapter is structured as follows:

1. I first extend Allocation.jl's existing functionality to support representing and evaluating fair allocation instances with matroid-rank valuations and matroidal constraints

2. I then present the requirements of the algorithms bla bla bla

3. I implement some stuff (matroid partition??)

In the next chapter, I describe an implementation, using Matroids.jl, of the Yankee Swap algorithm for fair allocation with matroidal valuations [29] and present some experimental results.

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.

A matroid-rank-valued allocation problem instance is represented as a struct containing the matroids for each of the agents, along with the number of goods. Agent i's value for the set of goods S, is the rank of S in agent i's matroid.

```
"""
    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 [m].
"""
struct MatroidRank <: Profile
    matroids::Vector{Matroid}
    m::Int
end

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

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, an agent i and a bundle S, and return the agent i's value for S, up to some item.

To check efficiently if an allocation is EF1, we make use of the fact that a matroid rank function has binary marginals; in other words, the highest valued good in a bundle will always have value 1, unless the bundle value is 0. This gives us a simple way of checking for EF1. Similarly, since the least positively-valued good also has value 1, EFX_+ is the same as EF1.

```
value_1(V::MatroidRank, i, S) = max(value(V, i, S)-1, 0)
value_x(V::MatroidRank, i, S) = value_1(V, i, S)
```

The value of the least valued good overall (including 0-values) depend on whether the bundle is independent. An independent set contains by definition no redundant elements, so if the bundle is independent, the least-valued good has value 1. If the bundle is dependent, it contains at least one 0-valued good, or, equivalently, a good whose removal does not affect the bundle value. This gives us EFX_0.

```
value_x0(V::MatroidRank, i, A) =
    is_indep(V.matroids[i], A) ? value_1(V, i, A) : value(V, i, A)
```

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

```
prop(V::MatroidRank, i, _) = rank(V.matroids[i])/na(V)
```

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 have two possible cases, the first being that $v_i(A_i) = v_i(\mathcal{M})$. This is trivially PROP1. In the other case, $v_i(A_i) < v_i(\mathcal{M})$. We know, due to the hereditary property (as given in Section 2.2.1) that there exists $g \in \mathcal{M} \setminus A_i$ such that $\Delta_i(A_i, g) = 1$ (this is equivalent to saying that A_i does not a contain a basis of $\mathfrak{M}_i$). To figure out if A is PROP1, then, we need to check whether $v_i(A_i) + 1 \geq \frac{1}{n} v_i(\mathcal{M}) \iff v_i(A_i) \geq \frac{1}{n} v_i(\mathcal{M}) - 1$.

```
prop_1(V::MatroidRank, i, A) = value(V, i, A) >= prop(V, i, A) - 1
```

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. If A_i is a closed set, then any additional good will increase the rank by 1, otherwise there is an element

Chapter 4

Implementing Yankee Swap

[redacted]

4.1 The matroid union algorithm

[redacted]

4.2 Some experimental results

Chapter 5

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 various types of matroids, including uniform, linear (vector), graphic, and partition matroids. The final part of the chapter is a significant portion of the thesis as a whole, and describes the implementation of Knuth's interesting algorithm [22] for the erection of arbitrary rank-r matroids.

1. `rank`
2. `closure`
3. `bases`
4. `is_indep`
5. `is_circuit`
6. `is_basis`

5.1 Uniform matroids and partition matroids

A 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. In Matroids.jl, we represent uniform matroids with a simple struct.

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

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

5.2 Linear matroids

5.3 Graphic matroids

We begin with defining 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*. 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. 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*).

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 $\mathfrak{M} = (E, \mathcal{I})$ is a matroid [26, p. 657]. To understand how, we will show that it adhers to axioms (1) and (2'), as given in Section 2. (1) holds trivially, as all subsets of a forest are forests. To see that (2') holds, consider the bases $\mathcal{B} \subseteq \mathcal{I}$. 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, 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.*

5.3.1 Random graphs

Since generating random graphic matroids will require us 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 [16]) 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 [14, 18]. 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, the so-called "giant" component of size $\mathcal{O}(n)$ emerges, and starts to absorb the smaller components [21]. 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 [16]. 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 [1].

Another approach is the Watts-Strogatz model, 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 [30].

5.3.2 Properties of random graphic matroids

We will use the Graphs.jl library [15] for handling graphs in Matroids.jl. This library has built-in methods for the random graph models described in the previous chapter[1].

When constructing matroids, we want to be able to specify the size of the ground set, and perhaps also the rank of the matroid. Let us see how we can achieve this with the random graph models we have discussed. The method `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 $k|m$ and solve for n. 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 method `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. Erdős-Rényi is the simplest model for our purposes, as the method `erdos_renyi(nv, ne)` simply takes in the desired number of nodes and edges. However, since the resulting graph has a large number of components for $|E| < \frac{n}{2}$, we have less fine-grained control over the final rank of the induced graphic matroid.

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 [11]. 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.

```julia
using Graphs
```

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

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

The rank function returns the size of a spanning forest of the subgraph induced by some subset of the edges. This is the rank of that subset. Thus, the rank of a subset $S \subseteq E$ can be found in $\mathcal{O}(|S| \lg |S|)$ time (when the MST is found using Kruskal's algorithm).

```
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
```

The indepence oracle returns whether the subgraph induced by a supplied subset of edges is acyclic. While independence can also be determined with the rank function, by checking whether the cardinality of a set equals its rank, this uses a DFS behind the scenes[2], which runs in linear time [11].

```
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
```

The circuit oracle

```
function is_circuit(m::GraphicMatroid, S)
  #TODO
end
```

The closure function 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

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

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 $G[S]$. The method `closure` below returns the set of all edges whose endpoints are both located in the subgraph induced by S.

```
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
```

5.4 Matroid erection

Before we delve into Knuth's general matroid construction, we need to establish a few concepts. The *rank-k truncation* of a matroid $\mathfrak{M} = (E, \mathcal{I})$, is the matroid $\mathfrak{M}^{(k)} = (E, \mathcal{I}^{(k)})$, where

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

When $\mathfrak{M}$ is of rank r, its *truncation* is given as $T(\mathfrak{M}) = \mathfrak{M}^{(r-1)}$. As a simple example, we have that the uniform matroid $U_n^{n-1} = T(\mathfrak{F}_n)$, where $\mathfrak{F}_n$ is the free matroid with n elements. The *erection* of the matroid $\mathfrak{M}$ is the matroid $\mathfrak{N}$ such that $\mathfrak{M} = T(\mathfrak{N})$ [12]. A matroid can have many erections. For example, it is easy to see that U_n^4 is an erection of U_n^3, since U_n^4 by definition has all the same independent sets of the rank-3 matroid U_n^3, along with all size-4 subsets of E. However, let $\mathfrak{U}$ be the rank-4 matroid over E where every subset of E of rank ≤ 3 is independent, and which has one size-4 rank-3 dependent set. The rank-3 independent sets of $\mathfrak{U}$ are the same as those of U_n^3, so we have $U_n^3 = T(\mathfrak{U})$, however the introduction of a size-4 dependent set of rank 3 has erected a different rank-4 matroid.

5.5 Knuth's matroid construction

In the preparatory project delivered fall of 2022, I implemented Knuth's 1974 algorithm for the random generation of arbitrary matroids via the erection of closed sets [22]. 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 1) 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.

Algorithm 1 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.

```
r = 0, F = [{∅}]
while TRUE
    Push!(F, Generate-Covers(F, r, E))
    F[r + 1] = F[r + 1] ∪ X[r + 1]
    Superpose!(F[r + 1], F[r])
    if E ∉ F[r + 1]
        r ← r + 1
    else
        return (E, F)
```

To understand the procedure, let us investigate what Algorithm 1 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).

```
GENERATE-COVERS($\mathrm{F}, r, E$)
1  return $\{A \cup \{a\} : A \in \mathrm{F}[r], a \in E \setminus A\}$
```

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 (ie. in accordance with the closed set axioms given in Section 2). If F_{r+1} contains two sets A, B whose intersection $A \cap B \nsubseteq C$ 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 $C \in F_r$.

```
SUPERPOSE!($F_{r+1}, F_r$)
 1  for $A \in F_{r+1}$
 2      for $B \in F_{r+1}$
 3          flag ← TRUE
 4          for $C \in F_r$
 5              if $A \cap B \subseteq C$
 6                  flag ← FALSE
 7
 8          if flag = TRUE
 9              $F_{r+1} \leftarrow F_{r+1} \setminus \{A, B\}$
10              $F_{r+1} \leftarrow F_{r+1} \cup \{A \cup B\}$
```

5.5.1 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 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 given up to this point corresponds closely to the initial Julia implementation, which can be found in Appendix **??**. It should already be clear that this brute force implementation leads to poor performance – for instance, the SUPERPOSE! method uses a triply nested for loop, which should is a candidate for significant improvement. Section 5.5.2 describes the engineering work done to create a more performant implementation.

Algorithm 2 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, $\mathrm{F} = [F_0, \dots, F_r]$, where F_i is the set of closed sets of rank i.

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

5.5.2 Improving performance

When recreating Knuth's table of observed mean values for the randomly generated matroids, some of the latter configurations of n and $(p_1, p_2, \dots)$ was unworkably slow, presumably due to my naïve implementation of the algorithm. Table 5.1 shows the performance of this first implementation.

Table 5.1: Performance of `random_kmc_v1`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	(0, 6, 0)	100	0.0689663	0.0106786	147.237 MiB
10	(0, 5, 1)	100	0.1197194	0.0170734	251.144 MiB
10	(0, 5, 2)	100	0.0931822	0.0144022	203.831 MiB
10	(0, 6, 1)	100	0.0597314	0.0094902	132.460 MiB
10	(0, 4, 2)	100	0.1924601	0.0284532	406.131 MiB
10	(0, 3, 3)	100	0.3196838	0.0463972	678.206 MiB
10	(0, 0, 6)	100	1.1420602	0.1671325	2.356 GiB
10	(0, 1, 1, 1)	100	2.9283978	0.3569357	5.250 GiB
13	(0, 6, 0)	10	104.0171128	9.9214449	161.523 GiB
13	(0, 6, 2)	10	11.4881308	1.3777947	20.888 GiB
16	(6, 0, 0)	1	-	-	-

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. As is evident from the data, larger matroids are computationally quite demanding to compute with the current approach, and the time and space requirements scales exponentially with n. Can we do better? As it turns out, we can; after the improvements outlined in this section, we will be able to generate matroids over universes as large as $n = 128$ in a manner of seconds and megabytes.

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. Appendix **??** contains all the code referenced in this chapter; the Julia implementation at this point can be found in **??**.

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

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

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 $B = A$

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 performance of `random_kmc_v2` is shown in Table 5.2. It is clear that representing closed sets using binary numbers represents a substantial improvement – we are looking at performance increases of 100x-1000x across the board. Great stuff!

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

Table 5.2: Performance of `random_kmc_v2`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	[0, 6, 0]	100	0.0010723	0.0001252	1.998 MiB
10	[0, 5, 1]	100	0.0017543	0.0001431	3.074 MiB
10	[0, 5, 2]	100	0.0008836	0.0001075	2.072 MiB
10	[0, 6, 1]	100	0.0007294	6.73e-5	1.700 MiB
10	[0, 4, 2]	100	0.0020909	0.0001558	3.889 MiB
10	[0, 3, 3]	100	0.0024636	0.0002139	4.530 MiB
10	[0, 0, 6]	100	0.007082	0.0004801	9.314 MiB
10	[0, 1, 1, 1]	100	0.0132477	0.0008307	17.806 MiB
13	[0, 6, 0]	10	0.042543	0.0014988	31.964 MiB
13	[0, 6, 2]	10	0.0183313	0.0012176	21.062 MiB
16	[0, 6, 0]	10	1.2102877	0.0146129	450.052 MiB

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 $\mathcal{O}(\binom{n}{r+1}^2\binom{n}{r})$ operations. In the worst case, no enlargements are made at all, and we build the free matroid in $\mathcal{O}(2^{3n})$ time (considering only the superpose step).

After larger closed sets have been added to 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 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.

```
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
```

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 `sorted_bitwise_superpose!`, which was used in `random_kmc_v3`. The full code can be found in Appendix **??**.

Unfortunately, as Table 5.3 shows, this implementation is a few times slower and more space demanding than the previous implementation. This is might be due to the fact that an ordered list is more space inefficient than the hashmap-based `Set`.

Iterative superpose

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

```
  # 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)
```

Table 5.3: Performance of `random_kmc_v3`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	[0, 6, 0]	100	0.0023382	0.0001494	4.042 MiB
10	[0, 5, 1]	100	0.001853	0.0001433	4.383 MiB
10	[0, 5, 2]	100	0.0017845	0.0001341	4.043 MiB
10	[0, 6, 1]	100	0.0015145	0.0001117	3.397 MiB
10	[0, 4, 2]	100	0.0030704	0.0002125	6.385 MiB
10	[0, 3, 3]	100	0.0037838	0.0002514	7.018 MiB
10	[0, 0, 6]	100	0.008903	0.000557	14.159 MiB
10	[0, 1, 1, 1]	100	0.0142828	0.0008823	21.838 MiB
13	[0, 6, 0]	10	0.0627633	0.002094	51.492 MiB
13	[0, 6, 2]	10	0.0106478	0.0007704	20.774 MiB
16	[0, 6, 0]	10	0.6070136	0.0095656	310.183 MiB

```
            setdiff!(F[r+1], [A, B])
            push!(F[r+1], A | B)
        end
    end
end
```

In `random_kmc_v4`, the full code of which can be found in Appendix **??**, 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 5.4 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`.

Table 5.4: Performance of `random_kmc_v4`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	[0, 6, 0]	100	0.0014585	3.94e-5	724.635 KiB
10	[0, 5, 1]	100	0.0007192	9.39e-5	659.729 KiB
10	[0, 5, 2]	100	0.0005943	3.53e-5	617.668 KiB
10	[0, 6, 1]	100	0.0003502	2.88e-5	408.666 KiB
10	[0, 4, 2]	100	0.001013	5.36e-5	887.618 KiB
10	[0, 3, 3]	100	0.0011847	5.03e-5	1.003 MiB
10	[0, 0, 6]	100	0.0015756	9.7e-5	1.066 MiB
10	[0, 1, 1, 1]	100	0.0046692	0.0001385	2.455 MiB
13	[0, 6, 0]	10	0.0118201	0.0005486	6.289 MiB
13	[0, 6, 2]	10	0.0075668	0.0002458	4.666 MiB
16	[0, 6, 0]	10	0.2819294	0.0040792	81.317 MiB
16	[0, 6, 1]	10	0.8268207	0.0070206	154.451 MiB
16	[0, 0, 6]	10	95.1959596	0.0290183	553.597 MiB

Rank table

While Superpose! is getting more efficient, it is still performing the same comparisons over and over again. Let's consider what we are really trying to achieve with this function, to see if we can't find a smarter way to go about it.

After adding the closed sets for a rank, Superpose! is run to maintain the closed set properties of the matroid (given in Section 2). These are 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 $A \cap B \subseteq C$. Up to this point, this has been done 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 next improvement: 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.

```
# The rank table maps from the representation of a set to its assigned rank.
rank = Dict{T, UInt8}(0=>0)

[...]

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


  for B in setdiff(F[r+1], A)
    if !haskey(rank, A&B) || rank[A&B] >= r
      # Update insert queue.
      push!(to_insert, A | B)

      # Update F[r+1].
      setdiff!(F[r+1], [A, B])
      push!(F[r+1], A | B)

      # Update rank table.
      rank[A|B] = r
      break
    end
  end
end
```

The full code for `random_kmc_v5` can be found in Appendix ??. Table 5.5 shows that implementing a rank table was an extremely significant improvement. For smaller matroids, it is around 5-10x faster, however it is for larger matroids that it truly outshines its predecessors – `random_kmc_v5` is a whopping 13 000 times faster than `random_kmc_v4` with $n = 16, p = [0, 0, 6]$ as input.

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

Table 5.5: Performance of `random_kmc_v5`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	[0, 6, 0]	100	0.0001335	0.0	138.966 KiB
10	[0, 5, 1]	100	0.0001436	0.0	158.691 KiB
10	[0, 5, 2]	100	0.0001928	0.0	167.487 KiB
10	[0, 6, 1]	100	0.0002204	0.0	148.812 KiB
10	[0, 4, 2]	100	0.0001578	0.0	173.455 KiB
10	[0, 3, 3]	100	0.0001743	0.0	202.566 KiB
10	[0, 0, 6]	100	0.0003433	0.0	431.089 KiB
10	[0, 1, 1, 1]	100	0.0004987	0.0	439.511 KiB
13	[0, 6, 0]	100	0.0004776	0.0	422.431 KiB
13	[0, 6, 2]	100	0.0003469	0.0	441.621 KiB
16	[0, 6, 0]	100	0.0009073	0.0	1010.452 KiB
16	[0, 6, 1]	100	0.0007939	0.0	997.022 KiB
16	[0, 0, 6]	100	0.0066951	0.0	8.564 MiB
20	[0, 6, 0]	100	0.0030797	0.0	4.042 MiB
20	[0, 6, 2]	10	0.0022849	0.0	4.547 MiB
32	[0, 6, 2, 1]	10	0.0269912	0.0	63.082 MiB

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

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

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

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

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 $B \subseteq C$. This is implemented in the following manner:

```
  # Generate minimal closed sets for rank r+1 (random_kmc_v6)
  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
      if t == 0 break end
    end
    # Insert y ∪ a for all a ∈ t.
    while t > 0
      x = y|(t&-t)
      add_set!(x, F, r, rank)
      t &= ~x
    end
  end
```

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

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

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

  push!(F[r+1], x)
  rank[x] = r
end
```

As such, F$[r+1]$ is empty when the first cover $y \in$ F is generated, and all covers $\{y \cup \{a\} : a \in E \setminus y\}$ are added. For later sets y, we are comparing with the previously added covers, and dropping any element to be found in a cover x that fully includes y. This way, we avoid re-generating the cover x.

The full code for `random_kmc_v6` can be found in Appendix **??**.

[redacted]

5.5.3 What do the generated matroids look like?

Observations

1. The average cardinality of the closed sets of a given rank is usually not very much higher than the rank. If the average cardinality were to stray much, all the sets merge instead and the sole closed set of that rank would become E. This might be a useful heuristic when finding the rank of a set.

[redacted] 2

Table 5.6: Performance of `random_kmc_v6`.

n	$(p_1, p_2, \ldots)$	Trials	Time	GC Time	Bytes allocated
10	[0, 6, 0]	100	0.000157	0.0	11.306 KiB
10	[0, 5, 1]	100	0.0001427	0.0	12.257 KiB
10	[0, 5, 2]	100	0.000121	0.0	11.568 KiB
10	[0, 6, 1]	100	8.61e-5	0.0	10.447 KiB
10	[0, 4, 2]	100	0.0001237	0.0	13.597 KiB
10	[0, 3, 3]	100	0.0001233	0.0	14.029 KiB
10	[0, 0, 6]	100	0.0002856	0.0	15.414 KiB
10	[0, 1, 1, 1]	100	0.0001942	0.0	14.446 KiB
13	[0, 6, 0]	100	0.0004483	0.0	19.117 KiB
13	[0, 6, 2]	100	0.0004541	0.0	18.957 KiB
16	[0, 6, 0]	10	0.0014919	0.0	34.531 KiB
16	[0, 6, 1]	10	0.0014731	0.0	36.016 KiB
16	[0, 0, 6]	10	0.0168858	0.0	127.652 KiB
20	[0, 6, 0]	10	0.0061574	0.0	81.573 KiB
20	[0, 6, 2]	10	0.0059717	0.0	82.323 KiB
32	[0, 6, 2, 1]	10	0.1599507	0.0	279.531 KiB
63	[0, 6, 4, 2, 1]	1	11.138914	0.0	4.912 MiB
64	[0, 6, 4, 4, 2, 1]	1	12.508729	0.0	4.912 MiB
128	[0, 6, 6, 4, 4, 2, 1]	1	1232.8570	0.0114583	102.159 MiB

[redacted]

5.5.4 Finding the properties of arbitrary matroids

To build a general matroid library, we want to be able to access all properties of a generated matroid $\mathfrak{M}$, viz. its bases $\mathcal{B}$, independent sets $\mathcal{I}$, circuits $\mathcal{C}$, and rank function $rank : 2^E \rightarrow \mathbb{Z}^*$.

In this section, I will first describe an extension of Knuth-Matroid that is also fully enumerates $\mathcal{I}$ and $\mathcal{C}$ for $\mathfrak{M}$ when n is small enough. However, this approach does not scale well for larger values of n. For values of n up to 128, we will therefore restrict our attention to independent sets and the rank function, as these are the matroid properties that are relevant to our usecase of fair allocation.

Up-front enumeration of circuits and independent sets for smaller matroids

In his 1974 paper [22], Knuth includes an ALGOL W [32] implementation that also enumerates all circuits and independent sets for the generated matroid. A later implementation in C called ERECTION.W can be found at Knuth's home page [23]. `random_erect` is 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` can be found in Appendix XXX.

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

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 (meaning F[$r+1$] contains the closed sets of rank $r+1$), a new operation, `mark_independent_subsets!` is called on each closed set.

Determining matroid properties after erection

In his 1989 paper, Greene introduces the concept of *descriptive sufficiency*. 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 [19].

```julia
"""
  function is_indep(M::ClosedSetsMatroid, S::Integer)

Determines whether a given set S is independent in the matroid M, given by the
     closed sets of M grouped by rank. Uses (I.1) in Greene (1989).
"""
function is_indep(M::ClosedSetsMatroid, S::Integer)
  t = Base.count_ones(S)

  for F in M.F[t]
    if Base.count_ones(S&F) > t-1 return false end
  end

  return true
end
```

Indepence oracle.

```julia
"""
    function is_circuit(M::ClosedSetsMatroid, S::Integer)

Determines whether a given set S is a circuit in the matroid M, given by the
     closed sets of M. Uses (C.1) and (C.2) in Greene (1989).
"""
function is_circuit(M::ClosedSetsMatroid, S::Integer)
```

```
  t = Base.count_ones(S)

  for F in M.F[t] # (C.1) S ⊆ F for some F ∈ F_{t-1}.
    if S&F==S @goto C2 end
  end
  return false

  @label C2
  for F in M.F[t-1] # (C.2) |S ∩ F| ≤ r(F) for all F ∈ F_{t-2}.
    if Base.count_ones(S&F) > t-2 return false end
  end

  return true
end
```

Circuit oracle.

```
"""
    function minimal_spanning_subsets(M::ClosedSetsMatroid, A::Integer)

A modification of Algorithm 3.1 from Greene (1989) that finds all minimal
     spanning subsets of A ⊆ E, given a matroid M = (E, F). If A = E, this finds
      the bases of M.
"""
minimal_spanning_subsets(M::ClosedSetsMatroid, A::Integer) = _mss_all(M, 0, A)
```

```
function _mss_all(M::ClosedSetsMatroid, j::Integer, Ā::Integer)
  B = [Ā&F for F in M.F[j+1] if Base.count_ones(Ā&F) > j]

  while length(B) == 0
    if j >= Base.count_ones(Ā)-1 return Ā end

    j += 1
    B = [Ā&F for F in M.F[j+1] if Base.count_ones(Ā&F) > j]
  end

  bases = Set()
  t = reduce(|, B)
  while t > 0
    x = t&-t
    bases = bases ∪ _mss_all(M, j, Ā&~x)
    t &= ~x
  end
  return bases
end
```

Minimal spanning subsets and bases.

Chapter 6

Results

Chapter 7

Conclusions

7.1 Limitations

What types of user preferences are we unable to represent using MRFs? Ref. Halpern's introduction; MRFs can model substitutes, but not complementaries (left shoe worthless without right shoe).

[redacted]

7.2 Future work

1. Barman and Verma, in their 2020 paper on MMS-fair allocations under matroid-rank valuations, give a

[redacted]

[redacted]

Notes

1. Skrive mer om hvordan Set{Set{Integer}} lagres i minnet og fordelene med å gå over til Set{Integer}.
2. @Benchmarking. Histogrammer. Beskrive variansen i matroide-størrelse ifht input.

Bibliography

[1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.

[2] Georgios Amanatidis, Haris Aziz, Georgios Birmpas, Aris Filos-Ratsikas, Bo Li, Hervé Moulin, Alexandros A. Voudouris, and Xiaowei Wu. Fair division of indivisible goods: A survey, 2022.

[3] Robert J Aumann and Michael Maschler. Game theoretic analysis of a bankruptcy problem from the Talmud. *Journal of Economic Theory*, 36(2):195–213, 1985.

[4] Siddharth Barman and Arpita Biswas. Fair division under cardinality constraints, 2020.

[5] Siddharth Barman and Paritosh Verma. Existence and computation of maximin fair allocations under matroid-rank valuations, 2021.

[6] Nawal Benabbou, Mithun Chakraborty, Ayumi Igarashi, and Yair Zick. Finding fair and efficient allocations when valuations don't add up. In *Algorithmic Game Theory*, pages 32–46. Springer International Publishing, 2020.

[7] Nawal Benabbou, Mithun Chakraborty, Ayumi Igarashi, and Yair Zick. Finding fair and efficient allocations for matroid rank valuations. *ACM Trans. Econ. Comput.*, 9(4), oct 2021.

[8] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[9] Eric Budish. The combinatorial assignment problem: Approximate competitive equilibrium from equal incomes. *Journal of Political Economy*, 119(6):1061–1103, December 2011.

[10] Ioannis Caragiannis, David Kurokawa, Hervé Moulin, Ariel D. Procaccia, Nisarg Shah, and Junxing Wang. The unreasonable fairness of maximum nash welfare. *ACM Trans. Econ. Comput.*, 7(3), sep 2019.

[11] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press. MIT Press, London, England, 3 edition, July 2009.

[12] Henry H. Crapo. Erecting geometries. *Annals of the New York Academy of Sciences*, 175(1):89–92, July 1970.

[13] Henry H. Crapo and Gian-Carlo Rota. *On the foundations of combinatorial theory: Combinatorial geometries.* M.I.T. Press, 1970.

[14] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6(3-4):290–297, July 1959.

[15] James Fairbanks, Mathieu Besançon, Schölly Simon, Júlio Hoffiman, Nick Eubank, and Stefan Karpinski. JuliaGraphs/Graphs.jl: an optimized graphs package for the julia programming language, 2021.

[16] Stephen E. Fienberg. A brief history of statistical models for network analysis and open challenges. *Journal of Computational and Graphical Statistics*, 21(4):825–839, 2012.

[17] Rafael Fourquet. BitIntegers.jl. https://github.com/rfourquet/BitIntegers.jl. Accessed: 2023-05-20.

[18] E. N. Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30(4):1141 – 1144, 1959.

[19] Tom Greene. Descriptively sufficient subcollections of flats in matroids. *Discrete Mathematics*, 87(2):149–161, 1991.

[20] Magnus Lie Hetland and Halvard Hummel. Allocations.jl, November 2022.

[21] Svante Janson, Donald E. Knuth, Tomasz Łuczak, and Boris Pittel. The birth of the giant component, 1993.

[22] Donald E. Knuth. Random matroids. *Discrete Mathematics*, 12:341–358, 1975.

[23] Donald E. Knuth. ERECTION.W. `https://www-cs-faculty.stanford.edu/~knuth/programs/erection.w`, Mar 2003.

[24] R. J. Lipton, E. Markakis, E. Mossel, and A. Saberi. On approximately fair allocations of indivisible goods. In *Proceedings of the 5th ACM Conference on Electronic Commerce*, EC '04, page 125–131, New York, NY, USA, 2004. Association for Computing Machinery.

[25] Benjamin Plaut and Tim Roughgarden. Almost envy-freeness with general valuations, 2017.

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

[27] Hugo Steinhaus. The problem of fair division. *Econometrica*, (16):101–104, 1948.

[28] Warut Suksompong. Constraints in fair division. *SIGecom Exch.*, 19(2):46–61, dec 2021.

[29] Vignesh Viswanathan and Yair Zick. Yankee swap: a fast and simple fair allocation mechanism for matroid rank valuations, 2023.

[30] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[31] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, July 1935.

[32] Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, jun 1966.

Appendices

Appendix A

Sets as numbers – some useful tricks

Section* 5.5.2 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 [17]). We reiterate the bitwise equivalents of the basic set operations in Table A.1. These bitwise

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

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

equivalents allow us to perform the set operations in constant time (right???), resulting in significant performance increases. In the code snippets included throughout Section 5.5.2 and Appendix **??**, 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
```