



NTNU

DEPARTMENT OF COMPUTER SCIENCE

TDT4900 — MASTER'S THESIS

Matroids and fair allocation

Author:

Andreas Aaberge Eide

Supervisor:

Magnus Lie Hetland

May 13, 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

1	Introduction	3
2	Generating matroids	4
2.1	Matroid preliminaries	5
2.2	Basic types of matroids	6
2.2.1	Uniform matroids	7
2.2.2	Linear matroids	7
2.2.3	Graphic matroids	7
2.3	Knuth's matroid construction	8
2.3.1	Randomized KMC	10
2.3.2	Improving performance	11
2.3.3	What do the generated matroids look like?	22
2.3.4	Producing the rank function and independent sets	23
3	The Matroids.jl API	26
3.1	The independence oracle	26
3.2	The matroid union algorithm	26
3.3	Supporting universe sizes of $n > 128$	27
4	Conclusions	28
4.1	Limitations	28
4.2	Future work	28

1 | Introduction

One goal for this project is to create a library for the Julia programming language [3], supplying functionality for generating and interacting with (random) matroids. Throughout the text I will refer to this library as Matroids.jl. 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 [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.



2 | Generating matroids

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 [4]

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.

Matroids were first introduced by Hassler Whitney in 1935 [17], 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. As a result of this, the terms used in matroid theory are borrowed from analogous concepts in both graph theory and linear algebra. 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 [2].

This chapter describes how `Matroids.jl` generates and represents various

types of matroids, from the very basic uniform matroid to the erection of arbitrary matroids using Knuth's 1974 algorithm [11].

2.1 Matroid preliminaries

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 [17]:

- (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 $e \in A \setminus B$ such that $B \cup \{e\} \in \mathcal{I}$.
- (2') If $S \subseteq E$, then the maximal independent subsets of S are equal in size.

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 $S = E$, (3) gives us a fundamental property of the *bases*, or maximal independent sets of a matroid, namely that all bases are of the same size. This is the rank of the matroid.

Rank

Given a matroid $\mathfrak{M} = (E, \mathcal{I})$, the *matroid rank function* (MRF) is a function $r : 2^E \rightarrow \mathbb{Z}^+$ that gives the rank of a set $A \subseteq E$, defined to be the size of the largest independent subset of A . Formally,

$$r(A) = \max\{|X| : X \subseteq A \text{ and } X \in \mathcal{I}\}.$$

Matroid rank functions are *binary submodular*. Binary because they have binary marginals, that is, $r(A \cup \{e\}) - r(A) \in \{0, 1\}$, for all $A \subseteq 2^E$ and $e \in E$. Submodularity refers to rank functions' natural diminishing returns property, namely that for any two sets $X, Y \subseteq E$, we have

$$r(X \cup Y) + r(X \cap Y) \leq r(X) + r(Y).$$

Every binary submodular function is the rank function of some matroid [16]. The diminishing returns property makes the rank function useful for modeling user preferences, as we will see in fair allocation with matroidal valuations.

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 [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 $A \cup \{b\}$

2.2 Basic types of matroids



2.2.1 Uniform 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 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)
```

2.2.2 Linear matroids

2.2.3 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.

To generate random graphic matroids, we need to generate random graphs. 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 [7]) picks uniformly at random a graph from among the $\binom{n}{M}$ 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 [5, 8]. 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 [9]. 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 [7]. 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 and k edges per node, 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 [15].

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 [14, p. 657]. To understand how, we will show that it adheres to axioms (1) and (2’), as given in Section 2.1. (1) holds trivially, as all subsets of a forest are forests. To see that (2’) holds, consider the bases (maximal independent sets) $\mathcal{B} \subseteq \mathcal{I}$. 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*.

We will use the Graphs.jl library [6] for handling graphs in Matroids.jl.

2.3 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 [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.

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 $\mathfrak{M} = T(\mathfrak{N})$ [13]. 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.

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 size-4, rank-3 dependent set of \mathfrak{U} is an essential closed set. The essential closed sets of a matroid, together with their ranks, fully determine the matroid [10].

KNUTH-MATROID (given in Algorithm 1) accepts the ground set E and a list X such that $X[i] \subseteq 2^E$, and produces the rank- r matroid \mathfrak{M} such that $\text{rank}(X) = k$ for each $X \in X[k]$. 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 $X[i]$. The algorithm outputs the tuple (E, F) , where $F = [F_0, \dots, F_r]$, 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 $\bigcup_{i=0}^r F[i] = \mathcal{F}$, and so the algorithm produces a valid matroid represented by its closed sets.

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

The *covers* of a closed set A of rank r are the sets obtained by adding one more element from E to A . The algorithm proceeds in a bottom-up manner, starting with the single closed set of rank 0 (the empty set) and for each rank

$r + 1$ adds the covers of the closed sets of rank r . The covers are generated with the helper method `GENERATE-COVERS(F, r, E)`.

Given no enlargements ($X = []$), the resulting matroid is the free matroid over E . Arbitrary matroids can be generated by supplying different lists X . When enlarging, the sets in $X[r + 1]$ are simply added to $F[r + 1]$.

`SUPERPOSE!(F[r + 1], F[r])` ensures that the newly enlarged family of closed sets of rank $r + 1$ is valid. If F_{r+1} contains two sets A, B whose intersection $A \cap B \not\subseteq C$ for any $C \in F_r$, 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!(Fr+1, Fr)
1  for A ∈ Fr+1
2    for B ∈ Fr+1
3      flag ← TRUE
4      for C ∈ Fr
5        if A ∩ B ⊆ C
6          flag ← FALSE
7
8      if flag = TRUE
9        Fr+1 ← Fr+1 \ {A, B}
10       Fr+1 ← Fr+1 ∪ {A ∪ B}

```

2.3.1 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 supplied sets. This is done by applying `SUPERPOSE!` immediately after adding the covers, then choosing a random member A of $F[r + 1]$ and a random element $a \in E \setminus A$, replacing A with $A \cup \{a\}$ and finally reapplying `SUPERPOSE!`. The parameter $p = (p_1, p_2, \dots)$ 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 be a candidate for significant improvement if possible. Section 2.3.2 describes the engineering work done to create a more performant implementation.

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

```
1  $r = 0, F = [\{\emptyset\}]$ 
2 while TRUE
3   PUSH!(F, GENERATE-COVERS(F,  $r, E$ ))
4    $F[r + 1] = F[r + 1] \cup X[r + 1]$ 
5   SUPERPOSE!(F[r + 1], F[r])
6   if  $E \notin F[r + 1]$ 
7      $r \leftarrow r + 1$ 
8   else
9     return ( $E, F$ )
```

2.3.2 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 2.1 shows the performance of this first implementation.

The performance was measured using Julia’s `@timed`¹ 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.

¹<https://docs.julialang.org/en/v1/base/base/#Base.@timed>

Table 2.1: Performance of `random_kmc_v1`.

n	(p_1, p_2, \dots)	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 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

$$164 = 0x00a4 = 0b0000000010100100 = 2^7 + 2^5 + 2^2.$$

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.

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

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 $A \& 1 \ll i == 0$, meaning that the set represented by $1 \ll i$ is not a subset of A . The bitwise implementation of SUPERPOSE! is unchanged apart from using the bitwise set operations described above.

Table 2.2: Performance of random_kmc_v2.

n	(p_1, p_2, \dots)	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

The performance of random_kmc_v2 is shown in Table 2.2. 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 the algorithm 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 $\mathcal{O}(\binom{n}{r+1}^2 \binom{n}{r})$ 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 2.3 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`.

Table 2.3: Performance of random_kmc_v3.

n	(p_1, p_2, \dots)	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

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)
            setdiff!(F[r+1], [A, B])
            push!(F[r+1], A | B)
        end
    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 $F[r+1]$, but to a temporary array `to_insert`. Each set A is then popped from `to_insert` one at a time, added to $F[r+1]$ and compared with the other sets $B \in F[r+1] \setminus \{A\}$ and $C \in F[r]$ 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 $F[r]$, 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 $C \in F[r]$ such that $C \supseteq A \cap B$, so in practice it usually does not require $\binom{n}{r}$ comparisons in the innermost loop.

Table 2.4 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 2.4: Performance of `random_kmc_v4`.

n	(p_1, p_2, \dots)	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 2.1). These are maintained by ensuring that, for any two newly added sets $A, B \in F[r+1]$, there exists $C \in F[r]$ such that $A \cap B \subseteq C$. Until 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 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 lesser rank. 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
end
```

Table 2.5: Performance of `random_kmc_v5`.

n	(p_1, p_2, \dots)	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

The full code for `random_kmc_v5` can be found in Appendix ???. Table 2.5 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 of sets in play at a given rank. In this case, $|\mathbf{F}[r]| = \binom{n}{r}$, 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

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

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 ??.



Table 2.6: Performance of random_kmc_v6.

n	(p_1, p_2, \dots)	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



2.3.3 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.

2

2.3.4 Producing the rank function and independent sets

To build a general matroid library, we want to be able to access all properties of a generated matroid \mathfrak{M} . This would include:

1. the bases \mathcal{B} of \mathfrak{M} ,
2. the independent sets \mathcal{I} of \mathfrak{M} ,
3. the circuits \mathcal{C} of \mathfrak{M} ,
4. the closure function $cl : 2^E \rightarrow \mathcal{F}$, and
5. the rank function $rank : 2^E \rightarrow \mathbb{Z}^*$ of \mathfrak{M} .

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.

Finding circuits and independent sets for smaller matroids

[11] includes an ALGOL W [18] implementation that also generates the circuits and independent sets for the generated matroid. A later implementation in C called ERECTION.W can be found at [12]. `random_erection` 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_erection` 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.

[REDACTED]

3 | The Matroids.jl API

A goal for this project is to introduce Matroids.jl as a useful library for experimenting with fair allocation that require matroids. In the previous chapter, we explored how such a library might generate and represent matroids, but this is not especially worthwhile until we also have in place an API layer to allow fair allocation algorithms to interface with our matroids in a practical and efficient manner.

3.1 The independence oracle

Whenever matroids show up in the context of fair allocation, the existence is assumed of an *independence oracle*, which can in polynomial time (with respect to the number of elements) decide whether a set is independent.

3.2 The matroid union algorithm



3.3 Supporting universe sizes of $n > 128$

The larger the ground set, the closer we are to an instance of The cake-cutting problem. Typical fair allocation problems with indivisible items deal with less than 100 items.

In other words, the Integer cap of 128 bits is a reasonable upper limit on universe size for fair allocation problems. However, one could look into using packages that add larger fixed-width integer types¹. `Matroids.jl` supports arbitrary integer types.

¹See for instance `BitIntegers.jl`

4 | Conclusions

4.1 Limitations

[Redacted text block for 4.1 Limitations]

4.2 Future work

[Redacted text block for 4.2 Future work]

Notes

1. Skrive mer om hvordan $\text{Set}\{\text{Set}\{\text{Integer}\}\}$ lagres i minnet og fordelene med å gå over til $\text{Set}\{\text{Integer}\}$.
2. @Benchmarking. Histogrammer. Beskrive variansen i matroide-størrelse ifht input.
3. Referer til Spliddit og vanlige størrelser på fordelingsproblemer
4. Beskriv åssen man kan oppgi valgfri Integer-type

Bibliography

- [1] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Rev. Mod. Phys.*, 74:47–97, Jan 2002.
- [2] et al. Benabbou, Nawal. Finding fair and efficient allocations for matroid rank valuations. *ACM Transactions on Economics and Computation*, 9:1–41, December 2021.
- [3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.
- [4] Henry H. Crapo and Gian-Carlo Rota. *On the foundations of combinatorial theory: Combinatorial geometries*. M.I.T. Press, 1970.
- [5] P. Erdős and A. Rényi. On random graphs I. *Publicationes Mathematicae Debrecen*, 6(3-4):290–297, July 1959.
- [6] James Fairbanks, Mathieu Besançon, Schölly Simon, Júlio Hoffman, Nick Eubank, and Stefan Karpinski. Juliagraphs/graphs.jl: an optimized graphs package for the julia programming language, 2021.
- [7] 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.
- [8] E. N. Gilbert. Random Graphs. *The Annals of Mathematical Statistics*, 30(4):1141 – 1144, 1959.
- [9] Svante Janson, Donald E. Knuth, Tomasz Łuczak, and Boris Pittel. The birth of the giant component, 1993.

- [10] Douglas G. Kelly and Daniel Kennedy. The higgs factorization of a geometric strong map. *Discrete Mathematics*, 22(2):139–146, 1978.
- [11] Donald E. Knuth. Random matroids. *Discrete Mathematics*, 12:341–358, 1975.
- [12] Donald E. Knuth. ERECTION.W, Mar 2003.
- [13] Rudi Pendavingh and Jorn van der Pol. Enumerating matroids of fixed rank, 2016.
- [14] A. Schrijver. *Combinatorial Optimization: Polyhedra and Efficiency*. Springer, 2003.
- [15] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, June 1998.
- [16] D J A Welsh. *Matroid Theory*. Monographs / London Mathematical Society. Academic Press, San Diego, CA, October 1976.
- [17] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, July 1935.
- [18] Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, jun 1966.