NTNU

Department of Computer Science

TDT4900 — Master's thesis

Matroids and fair allocation

Author:
Andreas Aaberge Eide

Supervisor:
Magnus Lie Hetland

March 27, 2023

Contents

Chapter 1

Introduction

Chapter 2

Background

2.1 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.

First introduced by Hassler Whitney in 1935 [Whi35], 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". 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 been widely studied in game theory and economics, 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 [Ben21].

Independent sets

Perhaps the most common way to define a matroid is in terms of its *independent sets*. 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, $\mathcal{I} \subseteq 2^E$. A matroid is an independence system with the following properties:

1. The empty set is independent: $\emptyset \in \mathcal{I}$.
2. The hereditary property: if $A \subseteq B$ and $B \in \mathcal{I}$, then $A \in \mathcal{I}$.
3. The augmentation property: If $A, B \in \mathcal{I}$ and $|A| > |B|$, then there exists $e \in A$ such that $B \cup \{e\} \in S$.

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.

Rank

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

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

Matroid rank functions are *binary submodular*. Binary because they have binary marginals, that is, $rank(A \cup \{e\}) - rank(A) \in \{0, 1\}$, 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

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

This diminishing returns property makes the rank function useful for modeling user preferences, and is a reason why matroids show up so often in economics and game theory (???).

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:

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 Examples of matroids

The free matroid

The uniform matroid

The vector matroid

The graphic matroid

Chapter 3

Random matroid generation

One goal for this project is to create the Julia library `Matroids.jl`, which will supply functionality for generating and interacting with random matroids. 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 [Knu75]. 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 chapter, 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 random generation of other specific types of matroids is discussed as well.

3.1 Knuth's matroid construction (KMC)

Knuth-Matroid (given in Algorithm 1) accepts the ground set E and a list of enlargements X, and produces the matroid $\mathfrak{M} = (E, \mathcal{F})$, such that for each set $X \in \mathrm{X}[r]$, we have $X \in \mathcal{F}$, $\operatorname{rank}(X) = r$. 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 $\bigcup_{i=0}^{r} \mathrm{F}[r] = \mathcal{F}$, and so the algorithm produces a valid matroid represented by its closed sets.

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 of a closed set A of rank r is simply all sets obtained by adding one more element from E to A. The covers are generated with the helper method Generate-Covers(F, r, E).

Generate-Covers(F, r, E)
return $\{A \cup \{a\} : A \in \mathrm{F}[r], a \in E \setminus A\}$

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

Superpose!$(\mathrm{F}[r+1], F[r])$ 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 $A \cap B \nsubseteq C$ 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!(F_{r+1}, F_r)
for $A \in F_{r+1}$
    for $B \in F_{r+1}$
        flag $\leftarrow$ TRUE
        for $C \in F_r$
            if $A \cap B \subseteq C$
                flag $\leftarrow$ FALSE

        if *flag* $=$ TRUE
            $F_{r+1} \leftarrow F_{r+1} \setminus \{A, B\}$
            $F_{r+1} \leftarrow F_{r+1} \cup \{A \cup B\}$

3.1.1 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 A.1. It should already

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)
```

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 3.1.2 describes the engineering work done to create a more performant implementation.

3.1.2 Improving performance

When recreating Knuth's table of observed mean values for the randomly generated matroids, some of the latter configurations of n and $(p_1, p_2, \ldots)$ was unworkably slow, presumably due to my naïve implementation of the algorithm. Table 3.1 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

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

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, $\mathrm{F} = [F_0, \ldots, F_r]$, 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!$(\mathrm{F}[r+1], \mathrm{F}[r])$
  if $E \in \mathrm{F}[r+1]$ **return** (E, F)
  while $p[r] > 0$
    $A \leftarrow$ a random set in $\mathrm{F}[r+1]$
    $a \leftarrow$ a random element in $E \setminus A$
    replace A with $A \cup \{a\}$ in $\mathrm{F}[r+1]$
    SUPERPOSE!$(\mathrm{F}[r+1], \mathrm{F}[r])$
    if $E \in \mathrm{F}[r+1]$ **return** (E, F)
    $p[r] = p[r] - 1$
  $r = r + 1$

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`[2] of elements of E. Appendix A contains all the code referenced in this chapter; the Julia implementation at this point can be found in A.2.

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

Table 3.1: 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 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.

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.

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

Table 3.2: 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

The performance of `random_kmc_v2` is shown in Table 3.2. 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 $\mathrm{F}[r+1]$, SUPERPOSE! will cause sets to merge, so that only maximal dependent sets remain. Some sets will even simply disappear. In the case where $X = \{1, 2\}$ was added by GENERATE-COVERS, and the $Y = \{1, 2, 3\}$ was added manually as an enlargement, the 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 $\mathrm{F}[r]$) 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 – notice how much more space is demanded by the 10-element matroid with $p = [0, 0, 6]$ than the one with $p = [0, 6, 0]$ in any of the performance tables in this section. Making enlargements at earlier ranks will result in smaller matroids as more sets will 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 A.2.

Unfortunately, as Table 3.3 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 3.3: 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

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
```

In `random_kmc_v4`, the full code of which can be found in Appendix A.3, 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 3.4 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`.

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 $A, B \in \mathrm{F}[r+1]$, there exists $C \in \mathrm{F}[r]$ 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

Table 3.4: 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

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
```

Table 3.5: 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

The full code for `random_kmc_v5` can be found in Appendix A.4. Table 3.5 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

Observing the step-by-step calls to SUPERPOSE!, we see that there are a lot of duplicate and unnecessary sets being processed. The duplicate sets stem from GENERATE-COVERS, whose implementation up to this point does not take 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.

`random_kmc_v6` improves upon the brute force cover generation described at the beginning of the chapter, 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 $B \subseteq C$.

```
# 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
```

The full code for `random_kmc_v6` can be found in Appendix A.5.

Table 3.6: 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

3.1.3 What do the generated matroids look like?

2

3.1.4 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 $cl : 2^E \to \mathcal{F}$, and
5. the rank function $rank : 2^E \to \mathbb{Z}^*$ of $\mathfrak{M}$.

In this section, I will first describe an extension of KNUTH-MATROID that is also finds $\mathcal{I}$ 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

[Knu75] includes an ALGOL W [WH66] 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 [Knu03]. ERECTION is an extension of KNUTH-MATROID that finds $\mathcal{I}$ and $\mathcal{C}$ by pre-populating the rank table with all subsets of E.

```
# 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
```

3.2 Other kinds of matroids

3.2.1 Uniform matroids

3.2.2 Graphic matroids

3.2.3 Vector matroids

Chapter 4

A library for fair allocation with matroids

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 this library generates random matroids, however this is not very useful 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.

4.1 The matroid union algorithm

4.2 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. 3

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 types[1]. `Matroids.jl` supports arbitrary integer types. 4

[1]See for instance `BitIntegers.jl`

Chapter 5

Fair allocation with matroid rank utilities

We wish to study fair allocation in which the utility function of each agent is a matroid rank function. That is, the utility function for each agent $i \in [n]$ is the rank function for a matroid $\mathfrak{M}_i = ([m], \mathcal{I}_i)$.

$S_i \in \mathcal{I}_i \iff v_i(S_i) = |S_i|$.

5.1 Yankee Swap

5.2 Barman and Verma's MMS algorithm

Chapter 6

Fair allocation under matroid constraints

6.1 Gourvès and Monnot's MMS approximation algorithm

6.2 Biswas and Barman's algorithm for EF1 under cardinality constraints

6.3 Biswas and Barman's algorithm for EF1 under matroid constraints

Chapter 7

Results

Chapter 8

Conclusions and future work

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.
3. Referer til Spliddit og vanlige størrelser på fordelingsproblemer
4. Beskriv åssen man kan oppgi valgfri Integer-type

Bibliography

[Ben21] et al. Benabbou, Nawal. Finding fair and efficient allocations for matroid rank valuations. *ACM Transactions on Economics and Computation*, 9:1–41, December 2021.

[Knu75] Donald E. Knuth. Random matroids. *Discrete Mathematics*, 12:341–358, 1975.

[Knu03] Donald E. Knuth. Erection.w, Mar 2003.

[WH66] Niklaus Wirth and C. A. R. Hoare. A contribution to the development of ALGOL. *Commun. ACM*, 9(6):413–432, jun 1966.

[Whi35] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, July 1935.

Appendices

Appendix A

Code snippets

A.1 random_kmc_v1

```
"""
Generate the set F_{r+1} of all "covers" of the sets in F_r, given the ground
     set of elements E. Set-based.
"""
function generate_covers_v1(Fr, E)
  Set([A ∪ a for A ∈ Fr for a ∈ setdiff(E, A)])
end


"""
  If F_{r+1} contains any two sets A, B whose intersection A ∩ B is not
       contained in C for any C ∈ F_r, replace A, B ∈ F_{r+1} by the single set
       A ∪ B. Repeat this operation until A ∩ B ⊆ C for some C ∈ F_r whenever A
       and B are distinct members of F_{r+1}.

  F and F_old should be Family: A Set of Sets of some type.

  This is the first, Set-based implementation of this method.
"""
function superpose_v1!(F, F_old)
  for A ∈ F
    for B ∈ F
      should_merge = true
      for C ∈ F_old
        if A ∩ B ⊆ C
          should_merge = false
        end
      end
```

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

  return F
end

"""
First implementation of Knuth's random matroid construction through random "
     coarsening".

n is the size of the universe.
p is a list (p_1, p_2, ...), where p_r is the number of coarsening steps to
     apply at rank r in the construction. The first entry of p should usually be
      0, since adding closed sets of size > 1 at rank 1 is equivalent to
     shrinking E.

This uses the Set-based KMC methods.
"""
function random_kmc_v1(n, p, T)::KnuthMatroid{Set{Integer}}
  E = Set([i for i in range(0,n-1)])

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

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

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

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

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

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

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

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

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

      pr -= 1
    end


    r += 1
  end
end
```

A.2 `random_kmc_v2` and `random_kmc_v3`

```
"""
Generate the set F_{r+1} of all "covers" of the sets in F_r, given the size of
     the universe. Bit-based.
"""
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

"""
Returns whether the intersection of A and B is contained within
some C in F_prev.
"""
function should_merge(A, B, F_prev)
  for C in F_prev
    if subseteq(A & B, C)
      return false
    end
  end
  return true
end

"""
If F contains any two sets A, B whose intersection A ∩ B is not contained in C
     for any C ∈ F_prev, replace A, B ∈ F with the single set A ∪ B. Repeat this
      operation until A ∩ B ⊆ C for some C ∈ F_prev whenever A and B are
     distinct members of F.

This implementation represents the sets using bits.
"""
function bitwise_superpose!(F, F_prev)
  As = copy(F)
  while length(As) !== 0
    A = pop!(As)

    for B in setdiff(F, A)
```

```
      if should_merge(A, B, F_prev)
        push!(As, A | B)
        setdiff!(F, [A, B])
        push!(F, A | B)
        break
      end
    end
  end

  return F
end

function 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

"""
Bitwise implementation of Knuth's approach to random matroid generation through
     a number of random "coarsening" steps. Supply the generate_covers and
     superpose methods to study the effects of different implementations of
     these.

n is the size of the universe.
p is a list (p_1, p_2, ...), where p_r is the number of coarsening steps to
     apply at rank r in the construction. The first entry of p should usually be
      0, since adding closed sets of size > 1 at rank 1 is equivalent to
     shrinking E.
"""
function random_bitwise_kmc(generate_covers, superpose, n, p)::KnuthMatroid{Any}
  # Initialize.
  r = 1
  pr = 0
  F = [Set(0)]
  E = 2^n - 1 # The set of all elements in E.

  while true
    # Generate covers.
    push!(F, generate_covers(F[r], n))

    # Superpose.
    superpose(F[r+1], F[r])

    # Test for completion.
    if E ∈ F[r+1]
```

```
      return KnuthMatroid{Any}(n, F, [], Set(), Dict())
    end

    # Apply coarsening.
    if r <= length(p)
      pr = p[r]
    end

    while pr > 0
      # Get random closed set A in F_{r+1} and element a in E - A.
      A = rand(F[r+1])
      a = random_element(E&~A)

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

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

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

      pr -= 1
    end

    r += 1
  end
end

"""
Second implementation of random-KMC. This uses the bit-based KMC methods.
"""
function random_kmc_v2(n, p, T=UInt16)
  return random_bitwise_kmc(generate_covers_v2, bitwise_superpose!, n, p)
end

"""
Third implementation of random-KMC. This sorts the sets by size before
     superposing.
"""
function random_kmc_v3(n, p, T=UInt16)
  return random_bitwise_kmc(generate_covers_v2, sorted_bitwise_superpose!, n, p)
end
```

A.3 random_kmc_v4

```
"""
This is an attempt at a smarter implementation than directly following the setup
      from Knuth's 1974 article. The superpose step is replaced by an insert
```

```
    operation that inserts new closed sets into the family of current rank one
    at a time, superposing on the fly.
"""
function randomized_knuth_matroid_construction_v4(n, p, T=UInt16)::KnuthMatroid{
    T}
  r = 1
  pr = 0
  F = [Set(0)]
  E = 2^n - 1 # The set of all elements in E.

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

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

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

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

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

    r += 1
  end
end
```

A.4 random_kmc_v5

```
"""
```

```julia
Fifth implementation of random-KMC. This one uses a dictionary to keep track of
     previously seen sets.
"""
function random_kmc_v5(n, p, T=UInt16)::KnuthMatroid{T}
  r = 1
  pr = 0
  F::Vector{Set{T}} = [Set(T(0))]
  E = 2^n-1
  rank = Dict{T, UInt8}(0=>0) # The rank table maps from the representation of a
      set to its assigned rank.

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

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

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

    if E ∈ F[r+1]
      return KnuthMatroid{T}(n, F, [], Set(), rank)
    end

    r += 1
  end
end
```

A.5 random_kmc_v6

```
"""
Sixth implementation of random-KMC, in which a rank table is used to keep track
     of set ranks, and the covers and enlargements are added one at a time,
     ensuring the matroid properties at all times.
"""
function random_kmc_v6(n, p, T=UInt16)::KnuthMatroid{T}
  r = 1
  pr = 0
  F::Vector{Set{T}} = [Set(T(0))]
  E::T = BigInt(2)^n-1
  rank = Dict{T, UInt8}(0=>0)

  while E ∉ F[r]
    # Create empty set.
    push!(F, Set())

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

    if E ∈ F[r+1]
      break
    end

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

```
    end

    r += 1
  end

  return KnuthMatroid{T}(n, F, [], Set(), rank)
end

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
```