Department of Computer Science

TDT4900 — Master's thesis

Matroids and fair allocation

Author:
Andreas Aaberge Eide

Supervisor:
Magnus Lie Hetland

March 15, 2023

Contents

Chapter 1

Introduction

Chapter 2

Background

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.

Perhaps the most common way to define a matroid is in terms of its *independent sets*. An independence system is a pair $(E, \mathcal{S})$, where E is the ground set of elements, $E \neq \emptyset$, and $\mathcal{S}$ is the set of independent sets, $\mathcal{S} \subseteq 2^E$. A matroid is an independence system with the following properties:

1. The empty set is an independent set, $\emptyset \in \mathcal{S}$.
2. A matroid is closed under inclusion: if $A \subseteq B$ and $B \in \mathcal{S}$, then $A \in \mathcal{S}$.
3. If $A, B \in S$ and $|A| > |B|$, then there exists an $e \in A$ st. $B \cup \{e\} \in S$.

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

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.

We also need to establish the concept of *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 $A \cup \{b\}$

1

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 with a universe size n of about 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 over E where each set in $\mathrm{X}[r]$ is a closed set of rank r. The output is the list $\mathrm{F} = [F_0, \ldots, F_r]$, where r is the final rank of $\mathfrak{M}$ and F_i is the set of closed sets of rank i. In the paper, Knuth shows that $\bigcup_{i=0}^{r} \mathrm{F}[r] = \mathcal{F}$, and so the resulting structure $\mathfrak{M} = (E, \mathcal{F})$ is a matroid.

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 ∪ {a} : A ∈ F[r], a ∈ E \ A}
```

Given no enlargements (X = []), the resulting matroid is the uniform matroid of rank $|E|$. 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 set 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 some $C \in F_r$, 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 ∈ F_{r+1}
    for B ∈ F_{r+1}
        flag ← true
        for C ∈ F_r
            if A ∩ B ⊆ C
                flag ← false

        if flag = true
            F_{r+1} ← F_{r+1} \ {A, B}
            F_{r+1} ← F_{r+1} ∪ {A ∪ 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 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. It should already be clear that this brute force implementation leads to poor performance – for instance, the SUPERPOSE! method uses a triply

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.

$r = 0, \mathrm{F} = [\{\emptyset\}]$
while true
    PUSH!(F, GENERATE-COVERS(F, r, E))
    $\mathrm{F}[r+1] = \mathrm{F}[r+1] \cup \mathrm{X}[r+1]$
    SUPERPOSE!($\mathrm{F}[r+1], \mathrm{F}[r]$)
    if $E \notin F[r+1]$
        $r \leftarrow r + 1$
    else
        return (E, F)

nested for loop! Subpar stuff. 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 the naïve implementation of the algorithm. Table 3.1 shows the performance of this first implementation.

The performance was measured using Julia's `@timed` macro [1], which returns the time it takes to execute a function call, how much of that time was spent in garbage collection and the size of the memory 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

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 $p[r] > 0$
        let A **be a random set in** $\mathrm{F}[r+1]$
        let a **be 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)
        $p[r] -= 1$
    $r += 1$

Representing sets as binary numbers

The first improvement we will attempt is to represent our families as sets of hexadecimal numbers, instead of sets of sets of numbers. Sets are represented using Julia's native `Set` type [2]. 2

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

Table 3.1: Performance of `randomized_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 instance, 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 $n = 16$). Set operations have equivalent binary operations; intersection corresponds to bitwise AND, union to bitwise OR and the set difference between sets A and B to the bitwise OR of A and the complement of B. Subset equality is also simple to implement: $A \subseteq B \iff A \cap B = A$. 3

Table 3.2: Performance of `randomized_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

It is clear that representing closed sets using binary numbers is a substantial improvement – we are looking at performance increases of 100x-1000x across the board.

Sorted superpose

Can we improve the running time of the algorithm further? One idea might be to perform the superpose operation in descending order based on the size of the sets. This should result in fewer calls, as the bigger sets will "eat" the smaller sets that fully overlap with them in the early iterations, however, the repeated sorting of the sets might negate this performance gain. 4

Unfortunately, as Table 3.3 shows, this implementation is a few times slower and more space demanding than the previous implementation. This is likely due to the fact that an ordered list is more space inefficient than the hashmap-based `Set`.

Table 3.3: Performance of `randomized_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

5

Iterative superpose

So far, we are inserting all covers of F_r into F_{r+1} along with all enlargements, and then running the superpose operation on all of them at once. In the worst case, when no enlargements have been made, F_{r+1} is the set of all $r+1$-sized subsets of E, $|F_{r+1}| = \binom{n}{r+1}$. Until this point, the superpose operation was performed with a triply nested `for` loop, comparing each $A, B \in F_{r+1}$ with each $C \in F_r$ to see whether $A \cap B \subseteq C$ or whether A, B should be replaced by $A \cup B$. Thus we were looking at a whopping $\mathcal{O}(\binom{n}{r+1}^2 \binom{n}{r})$ operations to perform the superpose part of step r.

Table 3.4: Performance of `randomized_kmc_v4`.

Rank table

Non-redundant cover generation and iterative superpose

3.1.3 Finding independent sets and circuits

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 a useful matroid library for experimenting with algorithms for fair allocation with matroid constraints, or matroid rank utility functions. 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.

This chapter is a literature study of relevant algorithms, and then a discussion on how the results of this has informed the API design of Matroids.jl.

4.1 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. 6

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. 7

[1] See for instance `BitIntegers.jl`

Chapter 5

Fair allocation with binary submodular valuations

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. Intro til matroider
2. Skrive mer om hvordan Set{Set{Integer}} lagres i minnet og fordelene med å gå over til Set{Integer}.
3. Beskrive KMC v2. Kode? Pseudokode? Putte i appendix? Finn ut.
4. KANSKJE: Skrive bedre om idéen bak sorted superpose.
5. Skrive om variansen mellom tilfeldige matroider! @benchmark osv. Histogram
6. Referer til Spliddit og vanlige størrelser på fordelingsproblemer
7. Beskriv åssen man kan oppgi valgfri Integer-type

Bibliography

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

Appendices

Appendix A

Tables

Table A.1: Observed mean values for Random-Knuth-Matroid.

n	$(p_1, p_2, \ldots)$	Trials	Bases	$\lvert F_2\rvert$	$\lvert F_3\rvert$	$\lvert F_4\rvert$	$\lvert F_5\rvert$	$\lvert F_6\rvert$
10	$(6,0,0)$	44 [a]	100.0	30.3	1.0			
10	$(6,0,0)$	917 [b]	76.6	28.3	25.5	1.0		
10	$(6,0,0)$	39 [c]	51.6	31.0	38.5	27.8	1.0	
10	$(5,1,0)$	26 [a]	107.2	33.3	1.0			
10	$(5,1,0)$	935 [b]	102.6	32.7	33.0	1.0		
10	$(5,1,0)$	39 [c]	53.0	33.0	44.6	48.0	1.0	
10	$(5,2,0)$	791 [a]	108.0	32.5	1.0			
10	$(5,2,0)$	201 [b]	100.0	32.9	32.6	1.0		
10	$(5,2,0)$	8 [c]	24.6	30.1	39.9	66.0	1.0	
10	$(6,1,0)$	862 [a]	99.2	28.4	1.0			
10	$(6,1,0)$	137 [b]	69.8	28.1	29.1	1.0		
10	$(6,1,0)$	1 [c]	48.0	33.0	41.0	33.0	1.0	
10	$(4,2,0)$	12 [a]	111.1	36.3	1.0			
10	$(4,2,0)$	950 [b]	119.2	35.9	42.5	1.0		
10	$(4,2,0)$	38 [c]	73.4	36.4	52.6	39.4	1.0	
10	$(3,3,0)$	4 [a]	115.0	39.0	1.0			
10	$(3,3,0)$	911 [b]	138.0	38.5	53.3	1.0		
10	$(3,3,0)$	85 [c]	90.6	38.7	61.9	36.2	1.0	
10	$(0,6,0)$	767 [b]	171.8	45.0	85.6	1.0		
10	$(0,6,0)$	230 [c]	128.4	45.0	95.8	72.7	1.0	
10	$(0,6,0)$	3 [d]	52.3	45.0	94.7	90.3	32.7	1.0

[a] Averages for experiments when final rank was 3.
[b] Averages for experiments when final rank was 4.
[c] Averages for experiments when final rank was 5.
[d] Averages for experiments when final rank was 6.