NTNU

DEPARTMENT OF COMPUTER SCIENCE

TDT4900 — MASTER'S THESIS

Matroids and fair allocation

Author:
Andreas Aaberge Eide

Supervisor:
Magnus Lie Hetland

March 10, 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.

One 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. $B \cup \{e\} \in S$.

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 fair allocation instances

1

Chapter 3

Random matroid generation

One goal for this project was to create a Julia library for generating and interacting with random matroids. In the preparatory project fall of 2022, I implemented Knuth's 1974 algorithm for random matroid generation 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 infeasibly slow.

3.1 Knuth's matroid construction

3.1.1 Randomized KMC

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 $(p_1, p_2, \ldots)$ was unworkably slow, presumably due to the naïve implementation of the algorithm. Table 3.1 shows the performance of this first implementation.

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 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

3.2.1 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, using Julia's native `Set` type [2]. 2

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

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.

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

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

3.2.2 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

3.2.3 Iterative superpose

Chapter 4

A library for fair allocation with matroids

Chapter 5

Fair allocation with binary submodular valuations

5.1 Yankee Swap

5.2 Some other MRF alloc algorithm as well maybe?

Chapter 6

Fair allocation with matroid constraints

6.1 A very cool algorithm will be discussed here

6.2 This algorithm is okay as well

Chapter 7

Results

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

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.