Matroids.jl

Ricky Shapley Logan Grout
School of Operations Research and Information Engineering
Cornell University

May 20, 2021

1 Introduction

In this project, we create a package for working with matroids in Julia. Matroids are a combinatorial object that, in some ways, generalizes graphs and the idea of linear independence. Matroids can be defined in many different, but equivalent, ways. Here we give one of the more common definitions.

Definition 1. *Let E be a finite set and $\mathcal{I} \subseteq 2^E$. We call $M = (E, \mathcal{I})$ a matroid if it satisfies the following axioms:*

(i) $\mathcal{I} \neq \emptyset$

(ii) *If* $I \subseteq J$ *and* $J \in \mathcal{I}$, *then* $I \in \mathcal{I}$

(iii) *If* $I, J \in \mathcal{I}$ *and* $|I| < |J|$, *there exists* $e \in J \setminus I$ *such that* $I \cup \{e\} \in \mathcal{I}$

2 Motivation

Matroids are an intriguing area of combinatorial mathematics with multiple applications. While Julia has many packages for different areas of mathematics (including graphs, differential equations, etc.), it currently has no package for matroids. As far as we have discovered, there are very few packages implementing matroids in any programming language; the one we are most familiar with is included as part of SageMath. There is a lot of potential for computational investigations to benefit matroid research. In fact, some current results are limited by computational barriers, and by providing operations on matroids in a fast language like Julia may allow us to discover new and exciting results!

3 Implementation

We implement matroids as an abstract type in Julia. As there are many commonly used cryptomorphic definitions for matroids (that is, separate sets of equivalent axioms that describe them), we implement these as subtypes of the abstract matroid type. These subtypes

include graphic matroids (which are created from graphs), linear matroids (from matroids), and basis matroids (that can be created from independent sets, among other similar inputs). To work, each subtype must implement a groundset function that returns the groundset of the matroid, and a rank function that gives the maximum number of independent elements of any subset of a matroid. Beyond simple bookkeeping operations, these two functions are sufficient to implement all other functions without worrying about how they are stored. Although not necessary for correctness, it is possible to derive significant speedups by overriding functions for specific implementations of matroids, especially complex functions like checking validity.

3.1 Interesting Details

Here are some details of the implementation that may be of interest.

- Matroids constructed from independent sets, bases, and circuits are all stored as the bases of the matroid.
- Linear matroids are constructed from matrices over arbitrary fields. In fact, many of the most common matroids are defined on matrices over finite fields like $GF(3)$ and $GF(4)$. To accommodate these fields, we used the `AbstractAlgebra.jl` and the `Nemo.jl` packages, which allow for matrix row operations on fields.
- Linear matroids are stored in a mercurial format. Many matrices encode identical matrices (which we can navigate between using certain row operations). Our rank function defined on linear matroids may change the underlying matrix at each evaluations, so almost any function on a linear matroid will change it. This makes equality checks not very useful.
- Although there are many graph packages in Julia, none provided all the functionality we wished for well (namely accurate iteration over edges in a multigraph). Since we don't do that much with our graphs, we instead just accept a list of edges as the representation of a graph and implement all necessary graph algorithms ourselves.
- Determining matroid isomorphism is at least as hard as graph isomorphism (and hence very hard). However, we implement an algorithm (taken from the SageMath implementation) that provides some significant benefit over a pure brute-force check. This algorithm calculates a characteristic value for every element, then uses this information to recursively partition the elements of the matroid into equivalence classes. In practice, this algorithm seems to perform well.

4 Future Directions

There are some features of, and operations on, matroids that are important to their study that we did not have enough time to implement. The first such feature we would like to implement is the ability to take the dual of matroid. Duality in matroids is one of the fundamental motivations for their invention. Further, we would like to implement operations

that are vital to understanding the structural properties of matroids. In particular, we would like to implement contractions, deletions, k-sums, unions, and intersections. For questions in the realm of extremal combinatorics, it is important to restrict our attention to matroids without loops or circuits of length 2, called *simple* matroids. A method that simplifies a given matroid would be useful for answering such questions. Finally, we could always add more families of matroids to the catalogue.