AMS Proceedings Style
Volume **00**, 1997

A software system for matroids

R. J. Kingan and S. R. Kingan

Abstract. We present an open source extensible software system for experimenting with matroids. In designing this system we also designed a framework for systems that work with specific objects and treat algorithms for those objects as data. Algorithms are plug-ins to the system, so new algorithms can be added without recompiling the existing code.

1. Introduction

In this paper we present the design details of Oid - an open source software system for matroids. Matroids are an abstraction of graphs, matrices, codes, designs, finite geometries, linear spaces, and several other concrete (relatively speaking) objects. The name Oid comes from a humorous paper called "Oids and their Ilk" that gently makes fun of a mathematician's tendency to work with the most abstract possible object.

We used the ideas in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides [G] to design Oid. We programmed to interfaces not classes and we designed a framework that can be reused in other systems. A *framework* is a set of cooperating classes that make up a reusable design for a specific class of software [G, p. 26]. In designing Oid we built a framework for systems that work with specific objects and treat algorithms for those objects as data. The framework has an extensible library of combinatorial object classes and utility classes that can be used to build a software system for other combinatorial objects. The library of object classes range from simple ones such as a subset class with subset operations to more complicated ones such as finite field classes.

The system models combinatorial objects using Java interfaces. A matroid may be entered and stored in more than one way. For example, it can be entered as a matrix over a finite field or as a family of bases. Each such representation is modeled by a class that implements the object's interface. Algorithms in the system are also implemented by classes that in turn implement generic algorithm interfaces. At runtime, a simple intelligent system delegates tasks to the appropriate algorithm.

Key words and phrases. algorithm, combinatorial computing, graph, linear space, matroid, software.

©1997 American Mathematical Society

So, algorithms can be regarded as plug-ins to the system and new algorithms can be added without recompiling the existing code.

There are a number of reasons to write a framework that treats algorithms as data. As mathematicians we frequently devise procedures to look for patterns in the objects we are studying. This system allows us to quickly code algorithms without having to think of any other programming details. It is also flexible enough to accomodate a variety of perspectives. Matroids, like topogical spaces, can be defined in numerous different ways. It is useful both from a pedagogical as well as research point of view to have a software system that can move seamlessly between the different matroid perspectives such as geometric, graphic, and lattice theoretic.

The system's design helps us maintain quality control as we add new features. This is frequently not a priority as noted in the following quote from *Dr. Dobb's Journal*: "Scientists consider laboratory results valid only if equipment is calibrated, samples are free of contamination, and all relevant steps are recorded. Software, on the other hand, is rarely required to meet these standards, or any standards at all. Despite everyone's personal experience with buggy code, the correctness of scientific simulations is rarely questioned, and reproducibility is rarely – if ever – demanded [B1]." All the code in our system's core library has been thoroughly unit-tested. Furthermore, its correctness is verified every time it is used in a different context.

While creating the system we made the following fundamental design decisions that reflect our philosophy on computing:

1. *The system mirrors the structure of combinatorial objects.* The object classes and algorithms in the system correspond to combinatorial objects and algorithms. Moreover, the relationships between the classes and algorithms correspond to the relationships between combinatorial objects and algorithms.
2. *The system treats algorithms as data.* Data is not stored in a program's source code and can be manipulated by the program. Similarly, the names of algorithms are not hard-coded in the system. Instead, when the system runs, it reads the list of algorithms it has available from a text file. Each algorithm is a Java class file by itself so new algorithms can be easily added. The system searches among the algorithms it has available to find the best one, in terms of complexity, or the user can specify. New algorithms can be added without recompiling the system.
3. *The system is light with many small classes.* The scope of our first system using the framework is limited to matroids. We feel that a small system with an intuitive design is preferable to a large system without one.
4. *The system is interactive but the classes in its library can be assembled into batch programs for larger computations.* The system's user-friendly GUI makes it suitable for small-scale experiments and pedagogical use. However, it is too cumbersome to work within the constraints enforced by the GUI for larger scale experiments and optimization problems. In this case, the library of classes can be used to build customized batch programs. For example, we used the classes in Oid to put together a batch program that generates isomorph-free matroids over finite fields [K1].

In Section 2, we describe some of the system functions. In Section 3, we discuss the design details and give the steps for adding new algorithms to the system. In Section 4 we describe some next steps that we can pursue.

2. System Functions

We begin this section with the definition of a matroid. Hassler Whitney introduced matroids in his 1935 paper *On the abstract properties of linear dependence* [W]. In defining a matroid Whitney tried to capture the fundamental properties of dependence that are common to graphs and matrices. A *matroid* M is defined as an ordered pair $(E, \mathcal{I})$ consisting of a finite set E and a collection $\mathcal{I}$ of subsets of E called *independendent sets* such that:

1. $\phi \in \mathcal{I}$
2. If $I_1 \in \mathcal{I}$ and $I_2 \subseteq I_1$, then $I_2 \in \mathcal{I}$
3. If $I_1, I_2 \in \mathcal{I}$ and $|I_1| < |I_2|$ then there exists $e \in I_2 - I_1$ such that, $I_1 \cup e \in \mathcal{I}$.

A *dependent set* is a subset of E not in $\mathcal{I}$. A *circuit* is a minimal dependent set. A *basis* is a maximal independent set. The *rank of* M is the size of a basis set. For a subset X of E, the *rank of* X is the size of a maximal independent subset in X. We say X is *spanning* if $r(X) = r(M)$. The *closure* of X is the union of X and all elements e not in X, such that $r(X \cup e) = r(X)$. We say X is *closed* if $r(X \cup e) = r(X) + 1$ for all elements e not in X. A *flat* is a closed set. The flats of a matroid form a geometric lattice under the subset relation. A *hyperplane* is a flat of rank $r(M) - 1$. We can also define a matroid in terms of its circuits, rank, closed sets, flats, hyperplanes, spanning sets etc. See [O] for a detailed introduction to matroids.

If we let E be the set of columns of a matrix over a field and $\mathcal{I}$ be the set of independent sets of columns, then $\mathcal{I}$ satisfies the independent set axioms. If we let E be the set of edges of a graph and $\mathcal{C}$ be the set of cycles, then $\mathcal{C}$ satisfies the circuit axioms. Thus matrices and graphs can be viewed as special types of matroids. A linear space is also a special type of matroid - a simple matroid with rank at most three. Simple matroids are matroids in which all the one and two-element sets are independent. Thus a system that performs matroid computations inherently performs computations for these other combinatorial objects.

A rank r, n-element matroid can be entered in the system in several ways. If the matroid corresponds to a matrix with entries from a finite field $GF(q)$, where q is a power of a prime, then the matrix is the obvious way of entering the matroid. Otherwise, it can, for example, be entered as a family of bases. If the matroid corresponds to a graph, then it can be entered in its incidence matrix format. A linear space can be entered as a matrix or in terms of its non-trivial lines.

The system has all the basic matroid oracles such as the independence, basis, circuit, flat, hyperplane, and closure oracles. An *independence oracle* is a boolean function f defined for all subsets X of E as $f(x) = 1$, if $X \in \mathcal{I}$ and 0 otherwise [0, p. 320]. The other oracles can be similarly defined. The closure oracle determines whether or not an element is in the closure of a subset. For a matroid entered as a matrix over $GF(q)$, the rank of a specific k-element subset can be computed in $O(n^2 r)$ steps since it involves column reduction. The independence, basis, circuit, closure, and hyperplane oracles make a linear number of calls to the rank function as shown in Figure 1. However, the situation changes somewhat, if the matroid is entered in its basis representation (see Figure 2). The independence and basis oracles search the list of bases and are therefore linear only in terms of the size of the list. Moreover, the rank function makes an exponential number of calls to the independence function since in the worst case it has to search all subsets of the given set to find the size of the largest independent subset.

Lists of circuits, basis, etc. can be obtained by calling the appropriate oracles. Determining whether or not one set is, for example, a circuit can be done in polynomial time. Listing all the circuits cannot be done in polynomial time. The best that can be done to speed things up is to use shortcuts based on matroid results. For example, once a circuit is identified none of its proper subsets need to be checked. In practice the algorithms in Oid are quite fast.

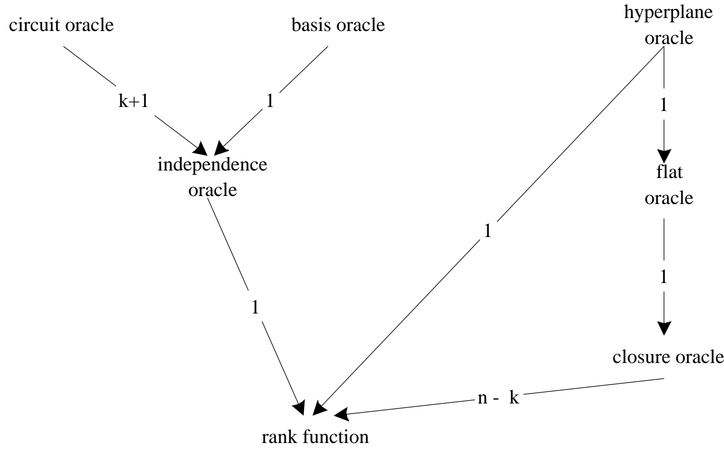

Figure 1: $GF(p)$-representation

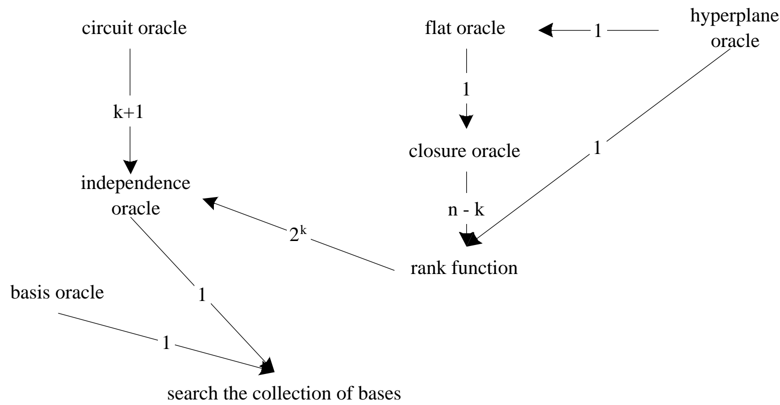

Figure 2: Basis representation

For a specific matroid, Oid can compute all the defining families of subsets, the dual matroid, several matroid invariants, and several matroid polynomials such as the Tutte polynomial, chromatic polynomial, connectivity polynomial etc. Oid also has an abstract isomorphism checker that can determine whether or not two matroids are isomorphic regardless of the format in which they are entered. Additionally, there are several algorithms useful for obtaining structural results such as single-element deletions, contractions, and extensions of a matroid. A detailed list of tasks is given in the User's Manual. The source code for Oid and the User's Manual are posted on the second author's web site (http://math.hbg.psu.edu/ srk1/matroids).

3. Design Details

Java is an object-oriented language with broad cross-platform support and a rich set of libraries for GUI elements. Compilers and other development tools are freely available. It is easy to learn and does not require explicit handling of pointers. Java also provides a set of *reflection classes* which allow a system to refer to object classes as data. This is important to the system's ability to accept new algorithms without changes to the core of the system itself. So, we decided to implement the framework in Java. However, the design could be implemented in any object-oriented language, including C++. We followed the guidelines in Gamma, *et al* [G] for object-oriented design using patterns, and particularly followed these principles:

1. *Program to an interface, not an implementation.* All of the major object classes in the system are accessed through Java interfaces, which allow the classes themselves to be modified or added to without breaking the system.
2. *Favor object composition over class inheritance.* To the extent possible, the object classes have been designed to be small and independent, to make them reusable in new representations and algorithms.
3. *Use delegation to allow composition of behaviors at run-time.* The system operates by accepting user input and passing that input to a "toolkit" class which delegates the command to a "smart" matroid class, which in turn selects a matroid representation and algorithm to perform the task. Therefore, a developer adding an algorithm to the system does not need to be concerned with the inner workings of the system.

3.1. Interfaces.

An *interface* is a specification of a set of methods or member functions. An object class is said to *implement* an interface if it has as member functions all of the functions specified in the interface. When an object class implements an interface, an instance of the object can be referred to using only the interface, instead of the object's intrinsic type. Each combinatorial object in the system has its own Java interface and all representations of the object perform the functions specified in the interface. At present, the system has a `Matroid` interface and two matroid representation classes, `FiniteFieldMatroid` class and `BasisMatroid` class that implement the interface. The `Matroid` interface specifies the matroid oracles that should be in each matroid representation class.

Interfaces are very useful because they allow portions of a system to be modified without affecting the rest of the system. An object class can be modified, but nothing else needs to change as long as it continues to implement the interface that the rest of the system uses to refer to it. Furthermore, since an interface specifies a general set of behaviors for classes that implement it, new matroid representation classes can be added without changing the rest of the system.

Each algorithm class must specifiy which representations it works with and must implement one of the five algorithm interfaces listed below. These five interfaces are not meant to be exhaustive, rather just a reflection of our limited knowledge of what would be useful.

1. `SubsetLister` interface: Algorithms that implement this interface accept a matroid as input, and produce a collection of subsets as output. Examples include algorithms that list circuits, independent sets, and flats.

2. `PropertyCalculator` interface: Algorithms that implement this interface accept a matroid as input and produce a numerical value.
3. `SingleConstructor` interface: Algorithms that implement this interface accept a matroid as input and return another matroid as output. Examples include an algorithm which contracts a specified element from a matroid.
4. `CollectionConstructor` interface: Algorithms that implement this interface accept a matroid as input, and generate a collection of matroids as output. Examples include algorithms to generate all single-element contractions of a single matroid.
5. `Combiner` interface: Algorithms that implement this interface accept a collection of matroids as input, and generate a single matroid as output.

The above interfaces are specified in terms of Java objects, not matroids. The methods which are used to provide input to the algorithms simply reject inappropriate input by throwing an exception. So these same interfaces can be used in other systems developed with this framework.

3.2. Self-Referentiality.

With all of these representations and algorithms that work for some but not all representations, the system needs at least rudimentary intelligence for keeping track of what representations and algorithms are available and optimal for the task at hand. It does this by using self-referentiality. Representations and algorithms are self-referential; that is, they can provide an estimate of the number of steps required to execute their methods.

When a matroid in a particular representation is entered and a task selected, the system can determine not only the fastest algorithm for that particular representation, but the fastest combination of algorithm and representation. For example, given a choice between a `FiniteFieldMatroid` instance and a `BasisMatroid` instance, the system will generally choose the `BasisMatroid` instance if it is generating a list of independent sets. On the other hand, if the number of bases is larger than the number of steps required to determine rank of a subset, the system will choose `FiniteFieldMatroid`.

To provide these estimates, each matroid representation and algorithm implements an interface called `ComplexityProvider`. This interface specifies methods that return estimates of the number of steps required for various matroid methods to execute. `ComplexityProvider` methods can return a flag value of -1 to indicate that a particular method cannot be executed with the indicated input; for example, if an algorithm requires a specific matroid representation, its complexity estimating methods will return a -1 if asked to provide an estimate with a different representation.

3.3. Smart Objects.

Each combinatorial object also has a "smart" class that implements the object's interface and keeps track of the different representations and algorithms. For example, the `SmartMatroid` class stores a collection of different representations of a single matroid, and determines which algorithm and representation to use in response to a user request. It selects the algorithm and matroid representation that, together, provide the lowest complexity estimate for the class, and uses that combination to complete the task. This feature can be extended to include some planning capability. For example, in some cases it may be faster to use one representation

to create an instance of another representation and then execute an algorithm than to simply run the algorithm with the existing representation.

3.4. Steps for adding a new algorithm.

A new algorithm may be added to the system without recompiling any of the rest of the system's code. The steps are listed below:

1. Choose the right algorithm interface from the five mentioned above, and write the algorithm to implement the interface. It is easiest to begin with the skeleton provided by the interface, and then add code particular to the algorithm.
2. Specify which representations the algorithm works for. It should generate an exception if not given the correct representation.
3. Write methods to estimate the complexity of the algorithm. The algorithm should implement the `ComplexityProvider` interface. The algorithm's complexity estimating methods may, in turn, rely on its input representation's complexity estimating methods. This enables the system to determine, for example , whether using a `FiniteFieldMatroid` representation or a `BasisMatroid` representation of a given matroid will result in faster execution.
4. The system reads a text file, `matroidTaskAlgorithmList.txt`, to determine which algorithms are available. When more than one algorithm is available to perform a specific task, each must implement the same algorithm interface. If the algorithm performs a new task, that is not already listed in the Tasks menu, add a line to `matroidTaskAlgorithmList.txt` describing the task as follows:

   `task` *taskName interfaceName taskDescription*

   Here, *interfaceName* refers to the algorithm interface the new algorithm implements, and *taskDescription* refers to the description to be displayed on the Tasks menu.
5. Add another line describing the new algorithm as follows:

   `algorithm` *className taskName representationClass algorithmDescription*

   Here, *className* is the name of the Java class containing the algorithm, *taskName* is the same as in Step 4, *representationClass* is the matroid representation accepted by the algorithm as input, and *algorithmDescription* is the menu description for the algorithm. This description will appear only if there is more than one non-trivial algorithm for a specific task, in which case the system provides the user with a choice of algorithms on the Tasks menu.
6. Compile only the new algorithm class and run the system. The new algorithm should appear on the Tasks menu.

4. Conclusion

Oid is an evolving system for which new parts can be added with varying levels of difficulty. It is easiest to add a new algorithm. The entire framework is geared to this purpose and we reduced the process of adding algorithms to a fill-in-the-blanks template. We can also add new algorithm interfaces and new matroid representations as the need for them arises.

There are three somewhat distinct directions we can take this project. The first of course is to continue using Oid interactively as a discovery tool to get insight after

which either a completely theoretical proof can be written or a computer-generated proof can be given. Structural results in matroids are frequently of two types: those that determine the minimal excluded minors of a class of matroids with a given structure and those that characterize the structure of a class without certain minors. A structural result of the first type would be for example, the Kuratowski-Wagner minimal excluded minor characterization of planar graphs. This famous result states that a graph is planar if and only if it has no minor isomorphic to K_5 or $K_{3,3}$. There are many other results of this kind. We are currently using Oid to find minimal excluded minors for interesting classes of matroids. A well-known conjecture by Rota states that "if q is a prime power, then the complete set of minimal excluded minors for representability over $GF(q)$ is finite" [O, 14.1.1]. The conjecture holds for $q = 2, 3, 4$ and presently remains open for fields of order 5 and higher. Oid's library of code was used to obtain several previously unknown minimal excluded minors for $GF(5)$-representable matroids [B2]. In the second type of problem we examine a class of matroids without certain minors to see what we can say about its structure. For example, the following problem in [O, 14.8.7] "Find all binary matroids such that, for all elements e, $M\backslash e$ or M/e is graphic" was expressed in terms of excluded minor classes and completely settled in 2002 [K2]. Software was used to get an idea on how to tackle the problem. However, the proof was completely theoretical and did not rely on software.

A second direction is to give Oid automated conjecture generating capabilities. We took the first step in this direction by using matroid structural properties to generate large numbers of matroids. In [K1] we present an algorithm for generating isomorph-free $GF(q)$-representable matroids. In the scheme outlined by McKay [M], this algorithm is in the class of algorithms for generation via canonical construction path. Now that we can obtain a large number of matroids for analyses, we need to store and process matroid invariants along with the matroids. We can develop code for matroid "frames," similar to artificial intelligence frames, that will store various representations of a single matroid, along with known invariants and links to related matroids, such as single-element deletions, contractions, extensions, and significant minors. Once these frames are developed and integrated with the rest of the system, we will build a database for storage of matroids and frame information, along with routines for persistence and retrieval of matroid frames. We will then be in a position to begin building conjecture generating tools that can manage large numbers of frames and apply pattern discovery techniques to identify relationships among the various matroid frames.

Finally, a third direction is to use the framework to build systems for other combinatorial objects and their algorithms.

Acknowledgement: The authors thank Thang Bui for many helpful discussions without which this software system would have remained an idea.

References

[B1] P. Beckman and G. V. Wilson, *Open Source Meets Big Iron*, Dr. Dobb's Journal **25 (6)** (2000), 30.

[B2] A. Betten, R. J. Kingan and S. R. Kingan, *Representability of rank 3 matroids* (in preparation).

[G] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, 1995.

[K1] R. J. Kingan and S. R. Kingan and Wendy Myrvold, *On matroid generation*, Congressus Numerantium **164** (2003), 95-109.

[K2] S. R. Kingan and M. Lemos, *Almost-graphic matroids*, Advances in Applied Mathematics **28** (2002), 438 - 477.

[M] B. D. McKay, *Isomorph-free exhaustive generation*, J. Algorithms, **26** (1998), 306–324..

[O] J. G. Oxley, *Matroid Theory*, Oxford University Press, New York, 1992.

[W] H. Whitney, *On the abstract properties of linear dependence*, Amer. J. Math. **57** (1935), 509-533.

Kingan Analytics Inc., Hershey, PA 17033
E-mail address: rkingan@kingananalytics.com

Department of Mathematical and Computer Sciences, Capital College, Pennsylvania State University, Middletown, PA 17057
E-mail address: srkingan@psu.edu