Matroid Theory Implementation

William Andrews

August 8, 2025

https://github.com/William-Thomas-Andrews/Matroid_Algorithms

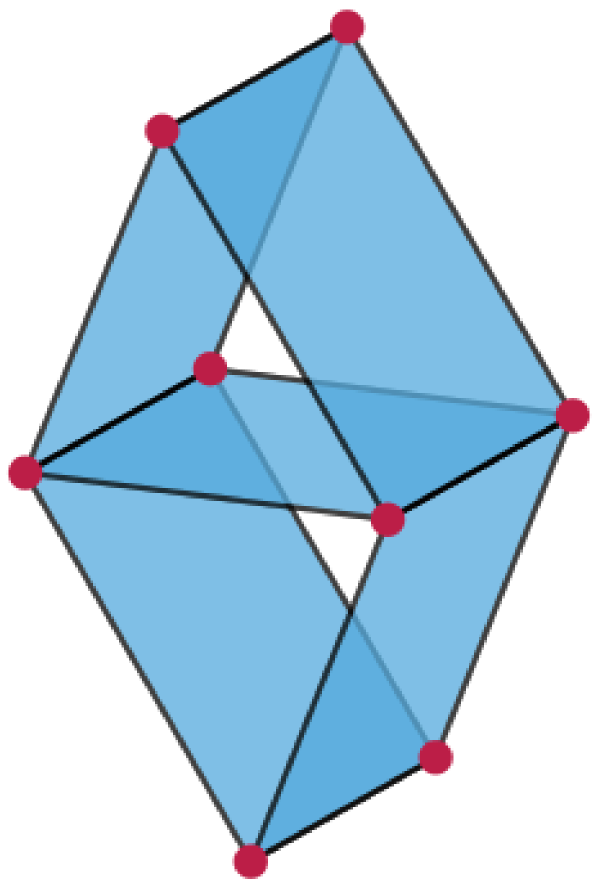


Figure 1: A Vámos matroid

"Matroids take 'It's useful to have multiple perspectives on this thing' to a ridiculous extent."

- anonymous

1 Introduction

Algebraic structures that can be 'solved' by greedy algorithms can be abstracted into one algebraic structure: the matroid. 'Solved' in this context is referring to a set being linearly independent, and 'unsolved' is when the set is linearly dependent. The independence and dependence are different for each algebraic structure. Some of the more popular and easy to comprehend algebraic structures that are used in matroids are graphs, vector spaces, bipartite graphs, and partition sets. All of these we will be reviewing in this write-up.

1.1 Definition of a Matroid

A *matroid* is defined as an ordered pair $M(E, \mathcal{I})$ where E is a finite set, referred to as the *ground set*, and $\mathcal{I}$ is a collection of *independent* subsets of E which satisfy the following properties:

- **Property 1:** $\emptyset \in \mathcal{I}$.
- **Property 2:** If $\mathcal{I}_1 \in \mathcal{I}$ and $\mathcal{I}_2 \subseteq \mathcal{I}_1$, then $\mathcal{I}_2 \in \mathcal{I}$.
- **Property 3:** If $\mathcal{I}_1$ and $\mathcal{I}_2$ are in $\mathcal{I}$ and $|\mathcal{I}_1| < |\mathcal{I}_2|$, then there is an element e of $\mathcal{I}_2 \setminus \mathcal{I}_1$ such that $\mathcal{I}_1 \cup \{e\} \in \mathcal{I}$.

Property 1 states that the empty set is an independent subset of E. **Property 2** states that if the set $\mathcal{I}_1$ is an independent subset of E and the set $\mathcal{I}_2$ is a subset of $\mathcal{I}_1$, then I_2 is also an independent subset of E. **Property 3** states that if the sets $\mathcal{I}_1$ and $\mathcal{I}_2$ are independent subsets of E and the cardinality (the dimension) of $\mathcal{I}_1$ is less than $\mathcal{I}_2$, then there is an element e which is in $\mathcal{I}_2$ but not in $\mathcal{I}_1$ such that $\mathcal{I}_1 \cup \{e\} \in \mathcal{I}$.

- **Weight:** Each element of a ground set can have a weight, and the weight of each set is the weight of each of the elements added up. This concept of weight is integral to our algorithms.
- **Span:** The span of a set of elements is the set formed by the elements that can be written as combinations of the elements that belong to the given set. And, in terms of linear algebra, the span of a set of vectors, also called linear span, is the linear space formed by all the vectors that can be written as linear combinations of the vectors belonging to the given set.
- **Basis:** An independent set of maximal size.
- **Dimension:** Dimension refers to the number of independent parameters required to specify an element in a space or a system, or in other words the number of elements of the independent set of maximal size.
- **Oracle:** The Oracle Model is a black-box model used to represent a matroid, providing a way to access information about the matroid's structure and properties.

Theorem: All bases of a matroid have the same cardinality.

Proof. Let us assume that there are two bases of a matroid M, B_1 and B_2, with different cardinalities, and without loss of generality, assume that $|B_1| < |B_2|$. Since B_1 and B_2 are bases, then by the definition of a basis, they are independent and cannot get any larger. However, by **Property 3**, there exists $e \in B_2 \backslash B_1$ such that $B_1 \cup e \in \mathcal{I}$, which contradicts that B_1 cannot be a base which has a maximal size. Therefore either B_1 is not in fact a base, or our assumption is wrong and the two bases have the same cardinality.

□

1.2 The Greedy Algorithm

Here is an overview of the algorithm and the format for the code. We have separate class files for each algebraic structure which we input to the `Matroid` class. The `Oracle` class serves one purpose: to tell us whether the inputted algebraic structure is independent or not, given that specific structure's conditions for independence. The `Matroid.hpp` class is listed below:

```cpp
// The SET being the type of input set, like Graph, or a Matrix
// The ELEMENT being the corresponding element for each set, like Edge for graphs, and Vector for matrices
template <class SET, typename ELEMENT>
class Matroid {
    private:
        SET ground_set;
        SET solution_set;
        Oracle<SET, ELEMENT> oracle;
    public:
        Matroid() : ground_set(SET()), solution_set(SET()) {}
        Matroid(SET& input_set) : ground_set(SET(input_set)), solution_set(SET()) {}
        Matroid(SET& input_set, SET& other_set) : ground_set(SET(input_set)), solution_set(SET(other_set)) {
            while (!(solution_set.get_vertices().empty())) {
                solution_set.remove_element();
            }
        }

        // Minimum Greedy Algorithm
        SET min_optimize_matroid() {
            ground_set.min_sort();                  // For minimum basis
            while (ground_set.not_empty()) {
                ELEMENT e = ground_set.top();
                if (oracle.independent(solution_set, e)) solution_set.add_element(e);
                ground_set.pop();
            }
            return solution_set;
        }

        // Maximum Greedy Algorithm
        SET max_optimize_matroid() {
            ground_set.max_sort();                  // For maximum basis
            while (ground_set.not_empty()) {
                ELEMENT e = ground_set.top();
                if (oracle.independent(solution_set, e)) solution_set.add_element(e);
                ground_set.pop();
            }
            return solution_set;
        }
};
```

The mathematical version of this algorithm can be summarized with this figure in pseudo-code below. It accepts the matroid $M = (E, \mathcal{I})$, and the weight function $w(\cdot)$ which is essential for the sorting algorithm `max_sort`. It returns the solution set of the maximum weighted spanning independent set, given the input matroid.

Algorithm 1 The Matroid Greedy Algorithm (Maximization)

1: **Input:** Matroid $M = (E, \mathcal{I})$, $w(\cdot)$.
2: **Output:** Maximum element in A
3: MAX_SORT(E)
4: $S \leftarrow \emptyset$
5: **while** $E \neq \emptyset$ **do**
6:     $e \leftarrow top(E)$ ▷ This gets the top (maximum) value of the ground set and assigns it to e.
7:     **if** $e \cup S \in \mathcal{I}$ **then** ▷ This line is equivalent to asking the oracle if e added to S would still be independent.
8:         add e to S
9:     **end if**
10:     $pop(E)$ ▷ This pops the top (maximum) value in the ground set
11: **end while**
12: **return** S

To begin, we sort the ground set E, then we assign S to be the empty set (because it is the solution set we will append to).

Next, we begin the while loop which runs on the condition that E is not empty. Then we assign an element e to be the top (the maximal value in this case) of the ground set E. Since E is already sorted from `line_3`, this is a constant time operation.

Then we enter an if condition that asks if $e \cup S \in \mathcal{I}$, or in other words, it asks if top element e appended to solution set S is an independent set, since $\mathcal{I}$ is a collection of independent subsets of E which satisfy **Property 1**, **Property 2**, and **Property 3**. If true (if the addition of e to S would result in a still independent solution set), then we add e to S. If not, then we continue. Regardless of the if condition, we pop the top value from E (which was the value that was assigned to e) and continue to the beginning of the while loop again.

One interesting thing to note is that the set $\mathcal{I}$ has a size so large, that it is not feasible to generate it in computation, and hence only works theoretically.

For example, let us look at a graphic matroid. In this sense, an independent graph has no cycles and a dependent graph has cycles (dependency is based on cyclicity), and the span of a graph is the amount of different nodes the tree reaches, so the set $\mathcal{I}$ essentially represents all the different paths of the graph that contain no cycles (which obey the basic three matroid properties by default). For reference, there is a 'cycle' if and only if there exists a non-empty 'path' in which the first and last vertices are equal. A 'path' is a finite or infinite sequence of edges which joins a sequence of vertices which, by most definitions, are all distinct. The 'weight' of each edge is given by $weight = w(u, v)$, where u and v are nodes.

If we were to try to find all the paths that would make up $\mathcal{I}$, we would be operating with $O(N!)$ time. This is just not feasible. Then how can we proceed? The answer is instead of asking if $e \cup S \in \mathcal{I}$ in `line_7`, we use an oracle, and ask it if the given set $(e \cup S)$ is independent, which is presumably much faster than creating all of $\mathcal{I}$, then asking that question. The check for independence that the oracle uses is different for each matroid (i.e. linear independence check for matrices, the cycle check using union find for graphs, etc.) which we will dive more deeply into when we disucss Matroid Variations.

Now we will look at the algorithm in the code I implemented to see how this mathematical algorithm can actually be adapted to be used in real life. In the matroid class, there are two functions for the algorithm: `min_optimize_matroid()` and `max_optimize_matroid()`, which represent the greedy algorithm that minimizes the weight of a ground set, and the greedy algorithm that maximizes the weight of a ground set. They are almost the same function, with the difference being one minimizes and the other maximizes. Let's take a look at the `max_optimize_matroid()` function:

```
// Maximum Greedy Algorithm
SET max_optimize_matroid() {
    ground_set.max_sort();                    // For maximum basis
    while (ground_set.not_empty()) {
        ELEMENT e = ground_set.top();
        if (oracle.independent(solution_set, e)) solution_set.add_element(e);
        ground_set.pop();
    }
    return solution_set;
}
```

The algorithm setup is quite simple. As seen from the `Matroid` class code given above, in the problem setup, we initially create a ground set `ground_set` of the same type as the input set (graph, matrix, bipartite graph, etc.) called the typename `SET`, and we also create a solution set `solution_set` also of type `SET`. Next, we create an oracle called `oracle` which is an instance of the `Oracle` class with template inputs `SET` and `ELEMENT`, where `SET` is the same as above and `ELEMENT` is the corresponding element type of the given `SET`.

The reason why we need `ELEMENT` to be specified in addition to `SET` is because the all these setup operations are compile time, and to deduce the element type of a given `SET` is runtime behavior, that happens after the compile time procedure. That erroneous sequence of events is akin to, for example, taking a math test and not studying for a certain section of it, expecting to use the answer sheet you will receive after taking the test to answer it during the test (this would only be possible if you could time travel), so to avoid this compile time error, the instantiation must be specified with type

`Oracle<SET, ELEMENT>`.

To reiterate, mathematically we have now a ground set (G, or `ground_set`), a solution set (S, or `solution_set`), and an oracle. All we do next is sort the `ground_set` and iterate through its top values, asking the `oracle` whether this new addition yields a dependent or independent `solution_set`. If the new maximal (top) value `e` makes the `solution_set` now dependent when appended to it, we discard that `e`. If the new maximal `e` makes `solution_set` still independent when appended to it, we go ahead and append that `e` to `solution_set`. Regardless of the `if` statement, we still perform `ground_set.pop()` to pop off that used `e` and begin the while loop again.

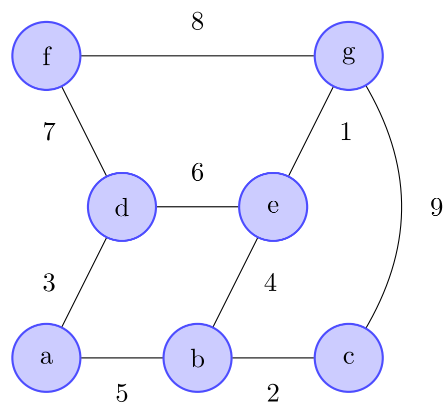
This same algorithm can maximize the independent basis for graphs, matrices, and many more structures, so long as the structure is a matroid! Since the concept of a greedy algorithm is universal for matroids, we can generalize it to take in as input different templated type inputs to perform the same algorithm on different structures. Isn't that remarkable? That little piece of code solves problems in graph theory, linear algebra, basic set theory, and many more fields. Now let's take a look at the specific uses of this algorithm and respective code implementations.

1.3 Matroid Variations

1.3.1 Graphic Matroid:

A *graphic matroid* $M = (E, \mathcal{I})$ is a matroid that uses a graph as its algebraic structure. Let our graph in this case be denoted at G. The ground set E consists of the set of edges of the graph G, and $\mathcal{I}$ is the set of all linearly independent subsets of E. We consider a set of edges to be linearly dependent if there is a 'cycle' in the set. There is a 'cycle' if and only if there exists a non-empty path in which the first and last vertices are equal. A path is a finite or infinite sequence of edges which joins a sequence of vertices which, by most definitions, are all distinct. The weight of each edge is given by $weight = w(u, v)$, where u and v are nodes.

Example: Let graph G be



with the ground set defined as $E = \{(f,g),(d,f),(d,e),(e,g),(c,g),(b,e),(a,d),(a,b),(b,c),(a,c)\}$. Although the whole ground set of G spans G, it is also cyclic, so E is not an independent base.

Algorithm: This algorithm is essentially a modified version of Kruskal's algorithm which has a time complexity of $O(E_g \log E_g)$, where E_g is the number of edges in the graph. In this example, we will be maximizing the resulting matroid, or in other words, creating a maximum spanning tree. We initially set the solution set $S = \emptyset$. We proceed by sorting the edges in E to have the maximum value at the top, using the weight function $w(u, v)$. This now results in

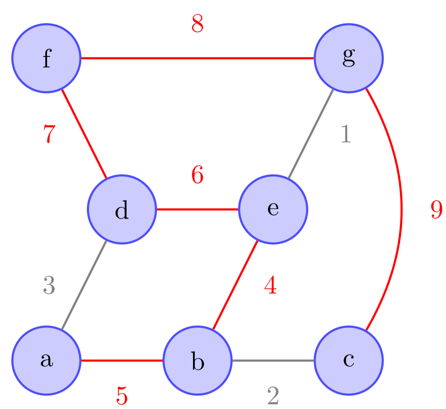
$$max\_sort(E) = \{(c,g),(f,g),(d,f),(d,e),(a,b),(b,e),(a,d),(b,c),(e,g)\}.$$

$$\text{and for simplicity, } E \leftarrow max\_sort(E)$$

Since E is clearly not empty, we take $top(E) = (c, g)$, and check if (c, g) added to S (S is currently an empty graph) is independent. $(c, g) \cup S = (c, g)$, and (c, g) does not create a cycle, so we add (c, g) to S, so $S \leftarrow (c, g) \cup S$. To finish up this loop, we pop edge (c, g) out of E by calling $pop(E)$, so now we have $E = \{(f,g),(d,f),(d,e),(a,b),(b,e),(a,d),(b,c),(e,g)\}$, and $S = \{(c,g)\}$

Now we can begin the next step in the while loop, since E is not empty. We take $top(E) = (f, g)$ and since $(f, g) \cup S$ also does not result in a cycle, we perform $S \leftarrow (f, g) \cup S$, and $pop(E)$. Now we have, $E = \{(d,f),(d,e),(a,b),(b,e),(a,d),(b,c),(e,g)\}$, and $S = \{(c,g),(f,g)\}$.

Let us continue until we reach the maximum spanning tree shown below (the edges in S are highlighted in red):



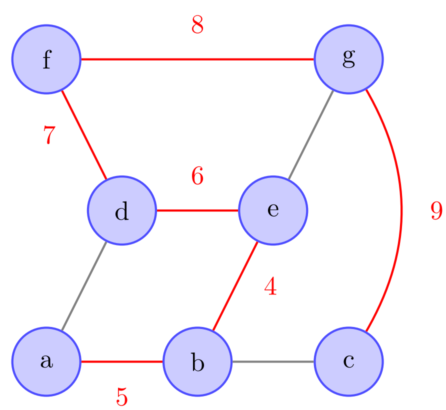
with $E = \{(a,d),(b,c),(e,g)\}$ and $S = \{(c,g),(d,f),(d,e),(a,b),(b,e)\}$. If we want to optimize this algorithm we can set a marker value the number of nodes of the ground set E, and if the number of nodes in the solution set S reaches the number of nodes in E, we stop because any new addition will cause a cycle (union find will find that we would be trying to connect the same partition together).

However, let us continue to show how this algorithm works fully. Next we get the top value of E to be (a,d), and since $(a,d) \cup S$ results in a cycle by union find, we discard it and pop the top value of E. Then we do the same for (b,c) and (e,g) which both result in cycles, so our final result is:

$$E = \emptyset \text{ and } S = \{(c,g),(d,f),(d,e),(a,b),(b,e)\}$$

and we have just found the maximum spanning tree of a graph. The minimizing version of this matroid algorithm finds the minimum spanning tree of a graph.

The code for this graph class and its special matroid functions are listed on the next page below.



`Graph.hpp` is listed below, with some parts omitted for simplicity.

```cpp
// The input set for a Graphic Matroid
class Graph {

    private:
        std::vector<Edge> edges;
        UnionFind union_set;

    public:
        Graph(std::vector<std::tuple<Vertex, Vertex, Weight>> input_data) : union_set(UnionFind(input_data.size())) {
            for (auto x : input_data) {
                Edge e = Edge(std::get<0>(x), std::get<1>(x), std::get<2>(x));
                this->add_element(e);
                union_set.union_operation(e.get_left(), e.get_right());
            }
        }

        // Matroid functions begin ---------------------------------------------------
        void min_sort() {
            std::sort(edges.begin(), edges.end(), MinCompare<Edge>{});
        }

        void max_sort() {
            std::sort(edges.begin(), edges.end(), MaxCompare<Edge>{});
        }

        bool not_empty() {
            return (!edges.empty());
        }

        Edge top() {
            if (edges.empty()) { throw std::runtime_error("Cannot get first element of an empty graph"); }
            else {
                return edges[edges.size()-1];
            }
        }

        // If adding Edge e does not create a cycle then it will return true
        bool is_independent(Edge& e) {
            // If both sides of the edge are in the same partition, then it creates a cycle and we return false because adding 'e' is not valid if we want to keep the graph acyclic.
            // Otherwise return true because both paritiitions are disjoint
            return (!(union_set.find_operation(e.get_left()) == union_set.find_operation(e.get_right())));
        }

        void add_element(Edge e) {
            edges.push_back(e);
            union_set.union_operation(e.get_left(), e.get_right());
        }

        void pop() {
            edges.pop_back();
        }
        // Matroid functions end ---------------------------------------------------
};
```

1.3.2 Linear Matroid:

A *linear matroid*

The head of the Matrix.hpp file is listed below:

```cpp
// The input set for a Linear Matroid
class Matrix {
    private:
        int rows;
        int columns;
        std::vector<Vector> data; // columns entries of row vectors
```

and the head of the Vector.hpp file is listed below:

```cpp
class Vector {
private:
    std::vector<double> data;
    double weight = 0;
```

The Matrix class consists of a std::vector of objects of the Vector class, and each Vector contains a std::vector of doubles, and a weight that gets adjusted to be the sum of all the values of the elements of the data from Vector.

Example: Here is an example of the maximizing matroid algorithm used on a linear matroid:

Let the matrix M be over the integers:

$$M = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 4 & 0 & 0 \\ 0 & 0 & 0 & 2 & 3 \end{pmatrix}$$

Each column corresponds to an element e_i in the ground set $E = \{e_1, e_2, e_3, e_4, e_5\}$. Let the weight of each element be:

Element	Column Vector	Weight
e_1	$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$	1
e_2	$\begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$	3
e_3	$\begin{bmatrix} 1 \\ 4 \\ 0 \end{bmatrix}$	5
e_4	$\begin{bmatrix} 0 \\ 0 \\ 2 \end{bmatrix}$	2
e_5	$\begin{bmatrix} 1 \\ 0 \\ 3 \end{bmatrix}$	4

Algorithm: We first create the solution set $S \leftarrow \emptyset$ and sort M by weight. Now we have

$$max\_sort(M) = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 \\ 4 & 0 & 3 & 0 & 0 \\ 0 & 3 & 0 & 2 & 0 \end{pmatrix}$$

and for simplicity, $M \leftarrow max\_sort(M)$.

we then append the top column vector e_3 to the empty matrix S because

$$e_3 \cup S = \begin{pmatrix} 1 \\ 4 \\ 0 \end{pmatrix}$$

is linearly independent by basic linear algebra rules. So now $S \leftarrow e_3 \cup S$. We continue to the next step and get

$$e_5 \cup S = \begin{pmatrix} 1 & 1 \\ 4 & 0 \\ 0 & 3 \end{pmatrix}$$

which is also linearly independent, so $S \leftarrow e_5 \cup S$. In the next step we try to add e_2, but that would result in the matrix

$$e_2 \cup S = \begin{pmatrix} 1 & 0 & 1 \\ 4 & 3 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

which is linearly dependent, so we do not change S and move on to the next column vector.

We eventually arrive at the maximum basis (the maximum spanning linearly independent matrix):

$$S = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 4 & 0 \\ 0 & 0 & 3 \end{pmatrix}.$$

In code, this is accomplished by the functions in `Matroid.hpp`. It is fascinating that the only difference between this algorithm and the graph algorithm before is the input set type to the `Matroid` class. The same procedure and functions are used because they are essentially the same problem in vastly different contexts.

The independence function for the `Matrix.hpp` class is as follows:

```
bool is_independent(Vector& v) {
    // First to check if it is the zero vector
    if (v.is_zero()) return false; // If yes, then it returns false because adding the zero
        vector makes the matrix linearly dependent
    Matrix A = *this;
    A.add_element(v);
    row_reduce(A);
    int rank_A = rank(A);
    int rank_this = rank(*this);
    if (rank_A == rank_this) {
        return false;
    }
    return true;
}
```

Surprisingly, coding precise `row_reduce()`, `rank()`, `dim()`, and `add_element()` functions that worked without fail was one of the hardest parts of this project.

1.3.3 Partition Matroid:

A *partition matroid* is a matroid that is so abstract that it is just plain simple. In the way I implemented it, its data is made up of a `std::vector` of my custom `ParitionPair` class which is just a modified `std::tuple`, with an extra attribute of an `int` of the partition of the pair.

The set's elements are partition pairs. The set's independence check is based on the partitions of its elements: if all of the set's elements have different partitions, then the set is independent, and if at least one of the set's elements have the same partition as another element, then the set is dependent.

The `PartitionMatroid` class is listed (with some functions omitted for simplicity) below.

```cpp
class PartitionMatroid {
    private:
        std::vector<PartitionPair> set;
    public:
        PartitionMatroid() {}
        PartitionMatroid(std::vector<PartitionPair>& input) : set(input) {}

        // Matroid functions begin ------------------------------------------------
        void min_sort() {
            std::sort(set.begin(), set.end(), MinCompare<PartitionPair>{});
        }

        void max_sort() {
            std::sort(set.begin(), set.end(), MaxCompare<PartitionPair>{});
        }

        bool not_empty() {
            return (!set.empty());
        }

        PartitionPair top() {
            if (set.empty()) { throw std::runtime_error("Cannot get first element of an empty graph"); }
            else {
                return set[set.size()-1];
            }
        }

        // If element e does not share the same partition with another element already in the set then it will return true
        bool is_independent(PartitionPair& e) {
            for (int i = 0; i < set.size(); i++) {
                if (e.get_partition() == set[i].get_partition()) { // if we are about to add an element with the same partition as a previous element
                    return false;
                }
            }
            return true;

        }

        void add_element(PartitionPair e) {
            set.push_back(e);
        }

        void pop() {
            set.pop_back();
        }
        // Matroid functions end ----------------------------------------------------
};
```

The algorithm is simply the same greedy algorithm listed multiple times above, that sorts the ground set E, then loops through the top values and chooses whether or not independence is maintained. If independence is maintained with an addition of a `PartitionPair`, then we add it. If not then we move on.

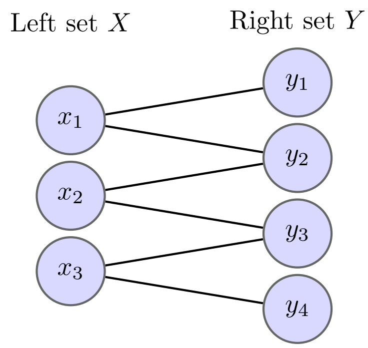
1.3.4 Bipartite Matroid:

A *bipartite matroid*

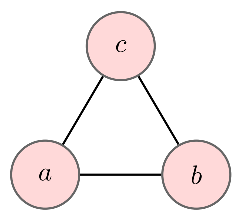
There is a `BipartiteGraph.hpp` file that holds the `BipartiteGraph` class which has the object

```
std::vector<std::vector<BipartiteEdge>> edges;
```

which consists of `BipartiteEdge` objects. This class is similar to our `Graph` class but has extra checks to make sure that the bipartite property is maintained. A bipartite graph is a graph where the vertices can be divided into two disjoint sets such that all edges connect a vertex in one set to a vertex in another set. There are no edges between vertices in the disjoint sets. Below is an example of a bipartite graph:



If any of the elements from X were connected to another element from X, then the graph would not be bipartite (with the same case for Y). Below is an example of a graph that is not bipartite:



Algorithm: The algorithm is the same greedy matroid algorithm we have been using but with a different application. It is very similar to the standard graph matroid but the dependency conditions are slightly different. The standard graph matroid maintains independency by maintaining acyclicity, while the bipartite graph matroid maintains independency by making sure that in a graph with two partitions, no vertices from one set are connected to the same set (this inherantly maintians acyclicity too).

First we get the inputs: the ground set based off the bipartite graph data E, and the weight function $w(u, v)$. Then we set the solution set $S = \emptyset$ and sort the ground set.

To begin the body of the algorithm, we again start the for loop by checking if E is empty, and if not, we proceed. Then we check if the top value from E added to the solution bipartite graph yields an acyclic bipartite graph (maintains independency), and if so, we add it to the solution set S and pop the top element from E and repeat just like the past algorithms.

1.4 UML

For reference, here is the rough structure of my files in this project.