Matroids and the Greedy Algorithm

Rahul Chandra

June 5, 2020

Contents

1 Introduction **1**

2 Definition(s) of A Matroid **2**
2.1 Definition in terms of Independent Sets 2
2.2 Definition in terms of Circuits 3

3 Matroids of Graphs **3**
3.1 Cycle Matroid of a Graph 3
3.2 Vector Matroid of a Graph 5
3.3 Uniform Matroid 6
3.4 Dual Matroid of a Graph 6

4 Proving Kruskal's Algorithm Using Matroids **7**
4.1 Proof of Correctness without Matroids 9
4.2 Proof of Correctness with Matroids 10

5 Extensions **11**
5.1 Wilson's Algorithm 11
5.2 (Optional) Proof of Wilson's Algorithm 13
5.3 Recent Progress With Matroids 16

1 Introduction

Matroids capture the notion of independence. Introduced by Hassler Whitney in 1935, and discovered independently by Takeo Nakasawa, matroids

were first introduced as any objects adhering to two axioms of independence. These two axioms were constructed as Whitney noticed that these axioms could be applied to both matrices and graphs. The construction of matroids has been extremely useful in graph theory, but more recently, they have been used in combinatorial optimization. The aim of this paper is to introduce the reader to matroids, show a well defined application to a graph algorithm, and to give the necessary background to understand more recent work in the field. We will first go over two of the numerous definitions of matroids. Afterwards, we will show how they interact with graphs as done in the paper by James Oxley, "On the interplay between graphs and matroids."[2]. The highlight of the paper will be when we find how to find the minimum weight spanning tree using Kruskal's algorithm, how to prove the correctness without matroids, and see how easy it is to prove it when the algorithm is translated into matroid language, as done in [3]. Finally, we provide some background for more recent work that uses matroids, specifically that of finding a uniform spanning tree of a graph. We will discuss what a uniform spanning tree is, and look at Wilson's algorithm. This should provide enough background for a reader to understand the algorithm in [1].

The only prequisite knowledge required is knowledge of what a graph is.

2 Definition(s) of A Matroid

There are many ways to describe a matroid. The first definition we will encounter is writing a matroid in terms of it's independence sets.

2.1 Definition in terms of Independent Sets

A matroid can be represented as a (E, I), where E is a finite set that satisfies some "independence" properties, and I is a collection of subsets of E(the independent sets). For example, if E is a collection of vectors, I is the subsets which are independent. Let's formally write the axioms of a matroid.

A **Matroid** $M = (E, I)$ where E is a finite set, and I is a collection of subsets of E, satisfies these axioms:

1. $\emptyset \in I$
2. If $A \in I$ and $B \subset A$, then $B \in I$.

3. If $A, B \in I$ and if $|A| > |B|$, $\exists\, x \in A$ such that $B \cup x \in I$.

The more common way of defining a matroid is through it's circuits.

2.2 Definition in terms of Circuits

A matroid can also be written as $M(E, C)$, where E is the groundset, and C are subsets of E called *circuits*. There is one axiom of circuits: If C_1 and C_2 are distinct circuits, and $e \in C_1 \cap C_2$, then $(C_1 \cup C_2) - \{e\}$ contains a circuit.

3 Matroids of Graphs

Matroids are very closely related to graphs, and can aid in understanding graphs. In this section, G is an undirected graph, and E is the edges of the graph, sometimes called the ground set. With this, we can get many matroids from any given graph.

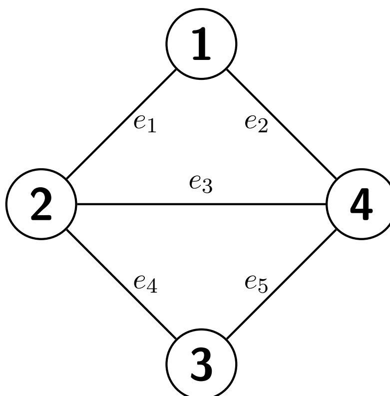
3.1 Cycle Matroid of a Graph

Definition. *A **cycle** in a graph is a non-empty trail in which the only repeated vertices are the first and last vertices.*

The cycle matroid of a graph G, denoted $M(G)$ has the groundset of E and C(the circuits) are the collection of edge sets of cycles.

Let us look at an example.

Example 3.1. Consider the graph G :



In this graph, we have that the ground set is $\{e_1, e_2, e_3, e_4, e_5\}$. There are three cycles, so the circuits would be $\{e_1, e_2, e_3\}, \{e_3, e_5, e_4\}, \{e_1, e_2, e_5, e_4\}$. Note that this follows the axioms of cycles: if you remove an edge common to two cycles, and consider the union of the edges of the two cycles, we will have another set with a cycle in it. For example, e_3 is common to both $\{e_1, e_2, e_3\}$ and $\{e_3, e_5, e_4\}$, so unioning this, and removing the edge e_3, we have the set $\{e_1, e_5, e_4, e_2\}$ which is another circuit!

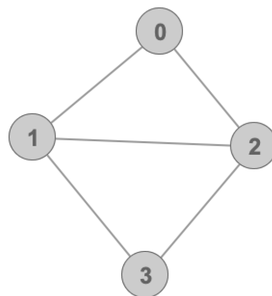
Note we may also describe $M(G)$ by the first definition of matroids. In this case, the independent sets would be the opposite of cycles...trees!(Trees are a graph in which between two vertices, there is only one path between them) That is, the edges that don't cause cycles in the graph! Thus, for $M(G)$, $E = \{e_1, e_2, e_3, e_4, e_5\}$ and $I = \{\{e_1\}, \{e_2\}, ..., \{e_1, e_2, e_5\}...\}$. We can do even better. We can describe the matroid by the maximal independent sets, which are subsets of E such that if you add one more element to the subset, then it is no longer independent. These are the **bases** of the matroid.

Definition. *Maximal independent sets are called* ***Bases****.*

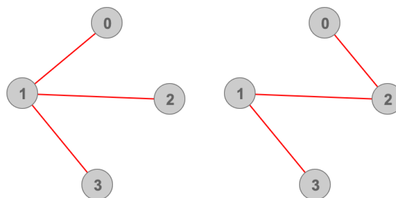
In our example, with $M(G)$, we have that the bases are the **spanning trees** of the graph G.

Definition. ***spanning trees*** *of the graph G are a subset of the edges of the graph G, such that all the vertices are covered, but there are no cycles.*

Example 3.2. Consider this graph:



Then there are many spanning trees of this graph, but here are two:



It is obvious that the spanning trees are "maximally independent," as if you add another edge, there will be a cycle. Let us prove some things about these bases.

Theorem. *The cardinality of the bases of a connected graph is precisely* $|V(G)| - 1$.

Proof. Note that the number of edges on a spanning tree of a connected graph is exactly $|V(G)| - 1$. To prove this, note that every tree which has n verticies has $n - 1$ edges. This is because if there were n edges, then every vertex will have a degree of 2(since by the definition of a tree, there is only one way to get to every vertex). Then starting at a random point, and taking a walk, with the rule that you don't repeat edges, you will eventually come to the same vertex again, meaning there is a cycle. This violates the definition of a tree, so there can only be $n - 1$ edges. Since the bases are the maximal independent sets, and every spanning tree has $|V(G)| - 1$ edges, then every base must have a cardinality of $|V(G)| - 1$. □

In general, a matroid M's bases all have the same cardinality, and this is called the rank $r(M)$ of M.

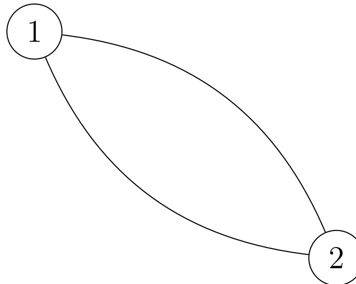
Talking about the independent sets of G is nice because we may represent the matroid as a set of vectors as well!

3.2 Vector Matroid of a Graph

Definition. *The* ***incidence matrix*** A *of a graph is a matrix where once we fix a labelling(number the vertices and edges) of the graph, it has a row for each vertex, and a column for each edge. If there is an edge that is connected to the vertex, there is a* 1, *otherwise there is a* 0.

The vectors of A is the groundset, and the circuits are the minimal linearly *dependent* subsets of the column vectors of A. Note that this is almost entirely analogous to $M(G)$. We will call this new matroid $M[A]$. I are the bases of this matrix with the maximal independence property.

Example 3.3. Consider this graph:



Then $A = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ Every edge is connected to every vertex, so every position in the matrix has a 1. Both these vectors are needed for linear dependence(as there are no loops), so I is both vectors of A.

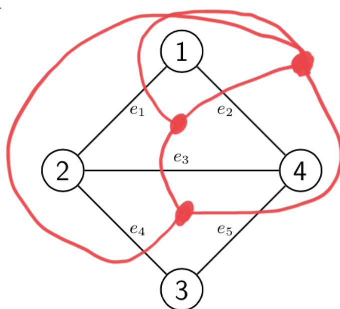
3.3 Uniform Matroid

Another matroid of a graph is the *uniform* matroid of a graph. This is denoted $U_{r,n}$, and $E = e_1, e_2, ..., e_n$ and C is all the $r + 1$ element subsets of E. Note that $r, n \geq 0$ and $r \leq n$.

3.4 Dual Matroid of a Graph

Before we can understand the dual matroid of a graph, let us first discuss the dual of a graph. A dual of a graph G is a graph that has a vertex for each "face" of G, and the edges are constructed so that whenever two faces are separated by an edge, there will be an edge between the corresponding vertices of the faces, and there will be a self-loop if the face is on both sides of an edge.

Below is an example:



Note that the red is the dual graph, and the black is our original graph G. Let us call the dual of the graph G G^*. Then note that the cycle-matroid $M(G^*)$ actually has the same E as the G. The circuits of $M(G^*)$ correspond to the minimal edge-cuts or *bonds* of G. A minimal edge-cut or bond are the minimal sets of edges of G whose removal causes the number of connected edges to go up. We will call these C^*. The dual matroid of a graph G is denoted $M^*(G)$.
Also, the edges of $G*$ are labelled after what edge we "cut." For example, the edge cutting e_1 will be called e_1, etc.

Properties of Dual Matroids

1. The bases of $M^*(G)$ are the complements of the bases of $M(G)$.
2. The rank of $M*$, which is called the *corank* of M, denoted $r^*(M)$ is $E - r(M)$.
3. $(M^*)^* = M$
4. The dual of every uniform matroid is also uniform.

4 Proving Kruskal's Algorithm Using Matroids

Now we have seen a few examples of matroids, let us prove a graph algorithm with them, to show their usefulness. One of the most interesting things about matroids is that they can be used to prove greedy algorithms.

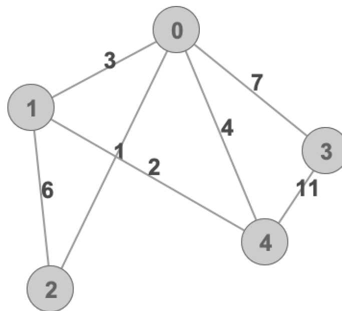
Definition. *A **Greedy Algorithm** is an algorithm in which we make the optimal step at each stage in order to find the global optimum.*

Let us look at Kruskal's Algorithm to demonstrate this. Suppose we have a weighted connected graph, and we would like to find the minimum spanning tree. That is, a spanning tree such that the sum of the weights of the edges is minimum. Consider Kruskal's Algorithm:

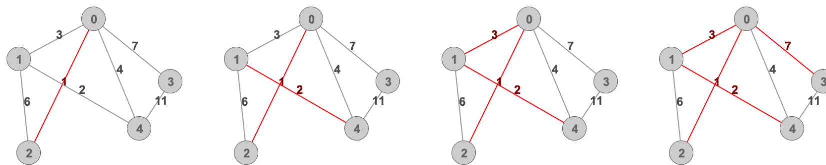
Kruskal's Algorithm:

1. Let T be the tree we are creating. Sort the edges in order of weight, lowest to highest.
2. Pick the top element – if adding this edge to T makes a cycle, don't add it(and eliminate it from consideration), but otherwise, add it to T.
3. Repeat step 2 until there are $V - 1$ edges in our tree, or equivalently, you cannot add another edge without creating a cycle.

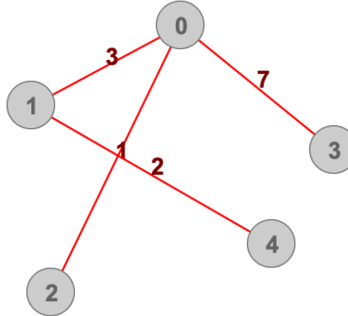
Example 4.1. Let us perform Kruskal's Algorithm on the below graph.



The steps are below. The red edges are the edges added to our tree.



Then our spanning tree looks like this:



Note that this is indeed a "greedy" algorithm. At each iteration, we are picking the lightest edge that does not cause a cycle and and adding it, and we claim that this is the minimum weight spanning tree. Let us first prove this without matroids.

4.1 Proof of Correctness without Matroids

Proof. Note that T is clearly a spanning tree. This is due to 3 reasons. The first is that we clearly do not create any cycles, as we don't add any edges that would create any cycles. The second is that every vertex in G is in T. To prove this, let us assume the contrary: let v be the vertex that is not in T but is in G. Since G is connected, there must be some edges that are connecting to v in G. Note that we could have added an edge from the rest of T to v using any one of those edges without a cycle, so this is impossible. The third is that T must be connected. Proving this is the same as the proof of the last one, as if T is not connected, but G is connected, then we could have used an edge in G and added it to T without causing a cycle, giving us a contradiction.
Now that we proved that T is a spanning tree, we will now prove that it is the minimum weight spanning tree. We will prove this by induction.

Let's say T isn't the minimum spanning tree, but T^* is. Then there is an edge $e \in T^*, e \notin T$. Consider $T \cup e$. Then there is a cycle C in T. In this cycle, with the exception of e itself, every edge will have a smaller weight than e(as otherwise, we would have chosen e to be in T instead). Since T^* doesn't have the cycle C(it's a tree), then there is an edge f that is in T but not in T^*.

Thus, consider the tree T_2 by eliminating f from T and adding e. Now, note that $\textbf{weight}(T_2) \geq \textbf{weight}(T)$, and T_2 has more edges in common with

T^* than T. If we keep doing this(exchanging edges) until we have T^*, we have that

$$\textbf{weight}(T) \leq \textbf{weight}(T_2) \leq ... \leq \textbf{weight}(T^*)$$

But since $\textbf{weight}(T^*)$ is the minimum weight, these inequalities must be equalities, and so T must be the same weight as T^*, and so it is also a minimum weight spanning tree. □

4.2 Proof of Correctness with Matroids

Proof. Consider the cycle matroid of the graph G. This means that the groundset E is the edges of G, and the bases are the spanning trees of G. Let us say there is a function that gives each element in the ground set a weigth(the weight of the edges) or in other words, there is a function w such that $w : E \rightarrow R_+$. Kruskal's algorithm is equivalent to finding the base with the lowest weight. Thus, the matroid equivalent of Kruskal's algorithm is sorting the groundset by weight, and adding the lightest weight elements to I, only if the independence property is satisfied(since it's a cycle matroid, the independence property is that there are not going to be any cycles). Keep doing this until we get a basis(and since we proved that every basis has the same rank, this is until we have $|E| - 1$ elements in I).

To prove this is the correct algorithm, assume that our algorithm gives us the minimal base as $\{x_1, x_2, ..., x_{|E|-1}\}$ with $x_1 \leq x_2 \leq ...x_{|E|-1}$ but the correct minimal base is $\{y_1, y_2, ..., y_{|E|-1}\}$ with $y_1 \leq y_2 \leq ...y_{|E|-1}$. Then there is a k such that $x_k > y_k$. Consider the set $\{x_1, ..., x_{k-1}\}$ and the set $\{y_1, ..., y_k\}$. Note that these are both independent sets, and that $|\{x_1, ..., x_{k-1}\}| < |\{y_1, ..., y_k\}|$. Then from the definition in 2.1, we know that there is some element y_j in $\{y_1, ..., y_k\}$ that we can add to $\{x_1, ..., x_{k-1}\}$ while still maintaining the independence of the latter set. Note that

$$w(y_j) \leq w(y_k) < w(x_k)$$

but if this is the case, then we would have chosen y_j before x_k in our algorithm! This is because y_j weighs less, and maintains the independence when added! This is a contradiction, so it must be that the matroid analog of Kruskal's algorithm gives us the optimal basis, as desired. □

Note that it was considerably easier to prove the correctness of Kruskal's algorithm using matroids, and in general, although not every Greedy Al-

gorithm can be expressed as a matroid, matroids help us generate greedy algorithms and prove them.

5 Extensions

It is clear from the above discussion that matroids are very powerful objects. Many famous conjectures about graphs can be rephrased in the language of matroids, and recent algorithms to find uniform spanning trees also call upon matroids for the proofs.

A **uniform spanning tree** is a spanning tree that was chosen randomly from all the spanning trees with equal probability.

There are many algorithms to find spanning trees of graphs, but ensuring that they are uniform is a much harder task. There do exist some algorithms, such as the Aldous-Broder algorihthm or Wilson's Algorithm, but recently, there have been algorithms found by looking at matroids[1]. To give background to this paper, I will show Wilson's algorithm, and optionally, a reader may go through the proof written in here as well.

5.1 Wilson's Algorithm

Before we can get started on the algorithm and prove it, we must define what a loop erasure is.

Like above, let X_i represent the node(I use vertex and node interchangeably) we are in at time i, so our walk would look like $\{X_1, X_2, ...\}$

Definition. *A **Loop Erasure** of a walk is deleting all the cycles of a graph in the order they appear. More precisely, if $X_i = X_j$, and $i < j$, then we delete the vertices $X_i, X_{i+1}, ..., X_{j-1}$ from the walk, so we just have $\{X_1, X_2, ..., X_i\}$ for our walk.*

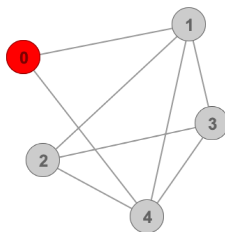
With this defined, we may define Wilson's Algorithm. Assume that the input graph is G, and the initial vertex given is r. Let T be the tree we are constructing, T_V be the set of edges of T(note that we are done when $T_V = V$), and T_E be the edges of the tree.

The algorithm:

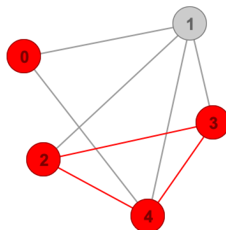
1. Let r be the initial tree. Thus, $T_V = r$, and $T_E = \emptyset$.
2. If T is a spanning tree, we are done. Otherwise, choose a vertex v that is not in T, and do a random walk starting at v(essentially go to a random vertex that you have an edge connecting to), erasing loops as we go, until we reconnect to a vertex in T. Then T is the union of the walk and itself.

Let us look at an example.

Example 5.1. Let the graph be G as shown below, and $r = 0$(it's colored red).

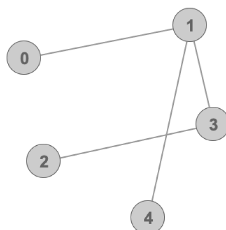


Then the vertices of the tree is just $\{0\}$. Let us choose the random vertex $v = 3$. Then we do a random walk on 3. Then let us walk to 2, and then walk to 4. If we go from 4 to 3, we have a cycle! The graph looks like so:



Thus, we must do the loop erasure method. Thus, we delete the vertices 2 and 4 from our walk, so we are just left with 3, and since we still have not

reconnected with T_0, we continue onwards. This time, let's say that we walk to 1, and from 1 we walk to 0. Then we have reconnected with the graph, and $T_V = \{0, 1, 3\}$, and thus, we can randomly pick 2 or 4, and start walking from there. Let's say we walk from 2 to 3 – then we are already reconnected to the tree, so we only have one vertex to choose from: 4. Then no matter where we walk to, since 4 is connected to only vertices in the tree, it will reconnect to T. For the sake of completeness, let's say 4 walked to 1. Then $T_V = V$, so we are done. This is what the graph described would look like:



The amazing thing is that this algorithm choses a *uniform* random spanning tree, so the spanning tree above is guaranteed to have had the same probability of being chosen as any other spanning tree!

5.2 (Optional) Proof of Wilson's Algorithm

This proof, while having nothing to do with matroids, is quite illuminating on why Wilson's algorithm works, and could serve well for giving intuition for further study related to matroids. We will now prove that the spanning tree chosen is indeed uniform, as done in [4].

Proof. For each vertex in the graph G, with the exception of the root vertex r we are given, let s_v be stack of an infinite amount of random variables associated with vertex v. These random variables will assign a vertex to walk to, that is a neighbor to v. For example, for the vertex 4 in the above example, we could have that the stack associated to it would be $[2, 1, 3, 2, 0, ...]$, and so we would say $s_4^{(1)} = 2$, $s_4^{(2)} = 1$ and so on. Note that this is the same as making a choice of where to walk when we needed it, since what happens externally doesn't really matter here. Thus, the probabilities have not changed.

Consider this new process:

1. "Pop" the first element from every single stack associated with $v \in V \backslash r$, and draw a directed edge from v to $s_v^{(1)}$, labelled 1. This means that in the stack associated with every vertex, the top element is eliminated, and the next random variable is moved up.

2. If there are no directed cycles, we are done but, if there are any directed cycles, we choose an arbitrary cycle, and remove the edges from the vertices in the cycle that causes the cycle. We then again, pop the stacks associated with the vertices, and draw the edges associated to the random variables. Label these new edges 2.

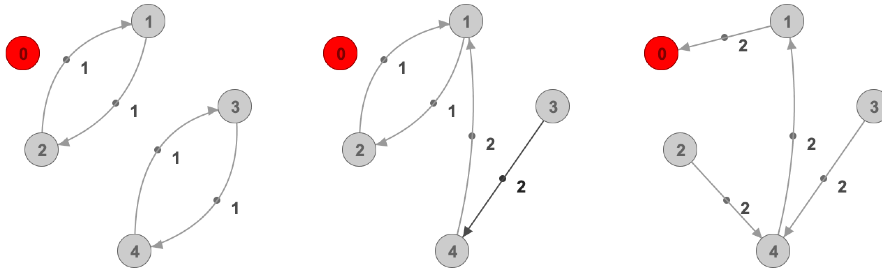
3. Keep repeating this process until there are no more cycles.

Note that we have all the edges $(v, s_v^{(i)})$ as i.

Example 5.2. Let us again use the same graph as the graph in example 4.1, with $r = 0$ again being the starting vertex. Below is the table of the random variables and the labels of the graph.

i	S_1^i	S_2^i	S_3^i	S_4^i
1	2	1	4	3
2	0	4	4	1
3	4	3	2	0

Let us first pop the stacks and draw the directed edges. Note that there are two cycles! Then we can choose to "pop" any of the cycles – let's arbitrarily choose to pop the cycle between 4 and 3 first, and then let us pop the cycle between 2 and 1. We repeat the algorithm until there are no cycles left. The steps are shown below.



Lemma: *The order of the cycles being popped does not matter. Either any ordering of popping will never terminate(we can never get a tree), or in any order of popping, the same set of colored cycles will be popped.*

Proof. Let C be a cycle that can be removed("popped") in some order. Then let $C_1, ...C_k = C$ be the order of poppings to get to C. We want to prove that the order of the cycle being popped does not matter, so let $C' \neq C_1$ be another cycle that can be popped in the beginning. We would like to prove that C will still get popped no matter what, so then we will show that $C' = C$ or C can still get popped by a sequence that begins with the popping of C'.

There are two cases. The first is that C' is vertex-disjoint from $C_1 \cup ... \cup C$. This is easy to deal with, as even if we pop C' first, then we can just pop $C_1, ..., C_k$ without an issue. Thus, C will still be popped.

Let us now consider the second case. Let $1 \leq m \leq k$ be the first index for which C_m shares a non-zero amount of vertices with C'. Then let x be the vertex shared by C' and C_m. Since C' can be popped at the first cycle, then C' must have all edges labelled 1. Since x was not in the sequence $C_1, ..., C_{m-1}$, and C_m can be popped after all of them, we have that the edge leaving x in C_m is also labelled 1. Let us call the vertex the edge of x in C_m points to y_m, and the vertex the edge of x in C' points to y'. Since both the edges are labelled 1, and its the same vertex, we have that $y_m = y'$. Note that we have that y' is not in the sequence $C_1, ..., C_{m-1}$ as well, because if it was, m would have been lower, since C' also has y'. Then we may apply the same argument, over and over again, to get that $C' = C_m$. Then we have that C_m is vertex disjoint from the sequence $C_1, ..., C_{m-1}$, so then we can pop the cycles in the order $C_m, C_1, ...C_{m-1}, C_{m+1}, ..., C_k$. Thus, we are done. □

Note that this process shows that it does not matter in which way we pop the cycles, it will give us the same spanning tree distribution, so the way we pop the cycles in Wilson's Algorithm(loop-erasing the random walks) is completely fine.

Since Wilson's Theorem terminates in finite time(the chance of never hitting a tree converges to 0), we know we will only need to pop finitely many cycles. Then we can view the stacks as a union of a finite number of cycles, and our tree T. However, note that another tree T' is just as likely,

as each random variable is just $\frac{1}{deg(v)}$ of being another vertex. Then the cycles being popped is independent of T, and each T is equally likely, so our spanning tree is indeed uniform. □

5.3 Recent Progress With Matroids

Although Wilson's algorithm is the most famous algorithm to find uniform-spanning trees, there have been many other algorithms to come out since then.

Recently, in [1], an algorithm uses matroids to describe another algorithm to find uniform spanning trees. In this paper, the authors describe a new type of random walk, called the down-up random walk. In this random walk, we remove an edge from a spanning tree(giving us two connected components, AKA two connected subgraphs that are not connected to each other), and we add an edge to connect the components. The paper uses dual-matroids to prove that adding an edge will not significantly slow the process down. The interested reader has most of the background necessary to read the paper.

References

[1] Nima Anari et al. "Log-Concave Polynomials IV: Exchange Properties, Tight Mixing Times, and Faster Sampling of Spanning Trees". In: *arXiv e-prints*, arXiv:2004.07220 (Apr. 2020), arXiv:2004.07220. arXiv: `2004.07220 [cs.DS]`.

[2] James G. Oxley. "On the interplay between graphs and matroids". In: 2005.

[3] Santosh Vempala. "Greedy Algorithms". 6505 Lecture. Jan. 2010. URL: `https://www.cc.gatech.edu/classes/AY2010/cs6505_spring/lectures/lecture3.pdf`.

[4] Yuval Wigderson. "Uniform Spanning Trees and Determinantal Point Processes". Stanford Probability Seminar. Mar. 2018. URL: `http://web.stanford.edu/~yuvalwig/math/teaching/UniformSpanningTrees.pdf`.