Wilson's Algorithm and Random Walks on Graphs

Ayan Chowdhury

Contents

Abstract

Graphs (in the Graph Theoretic sense) are a useful tool for representing data. Tree's in particular are helpful for modeling real a variety of real world phenomena in physics, chemistry, and more. It could be the case that you have data in which you can ascertain meaning by giving it a tree structure. You may then want to analyze how your data behaves over all possible tree structures you can give it. Wilson's algorithm provides a simple way to generate spanning trees over sets of vertices uniformly, enabling such analysis numerically.

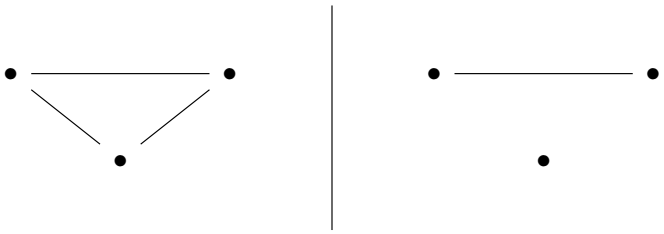
Preliminaries

We define a graph $G = (V, E)$ to be a collection of a set of vertices V and a set of edges E. Edges are pairs of vertices. We can consider vertices to represent "objects" and edges to represent connections between these objects. We can visualize such structures by drawing vertices, and drawing edges as a line between vertices. For example

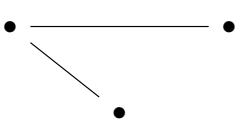
is a representation of a graph with 6 vertices and 4 edges, with the edges corresonding to the segments between our vertices.

We say a graph G is simple if it has no loops or multi-edges. This means there are no edges from any vertex to itself, and there is at most one edge between any two vertices. From now on we consider only simple and finite graphs.

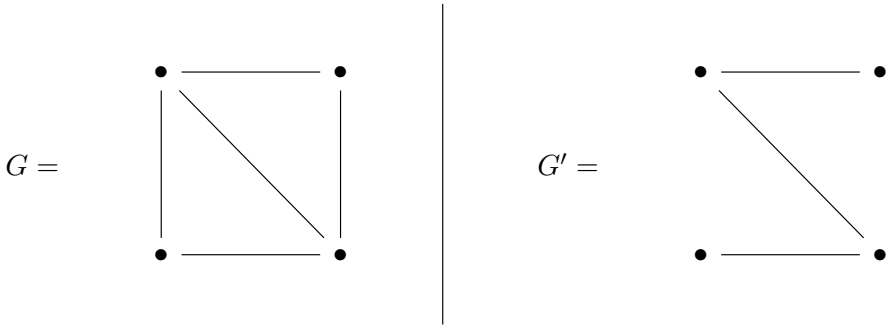
We define a walk on a graph to be a sequence (finite or infinite) of vertices connected by edges. We say a graph is connected if there exists a walk between any two vertices. We define a cycle of a graph to be a walk of length greater than 1 which travels along distinct edges and starts and ends on the same vertex. We define a tree to be a connected graph with no cycles. For example the following graphs:



are not trees, as the left graph has a cycle, and the right graph is not connected, while



is a tree. Given a connected graph $G = (V, E)$, we say a spanning tree is a subgraph $G' = (V, E')$ where $E' \subseteq E$ and G' is a tree. For example



we have G is a connected graph, and G' is a spanning tree of G. We can observe, and prove via induction [2], that every tree (and thus spanning tree) has the property $|V| - 1 = |E|$ and that every connected graph has a spanning tree.

Random Walks and Loop Erasure

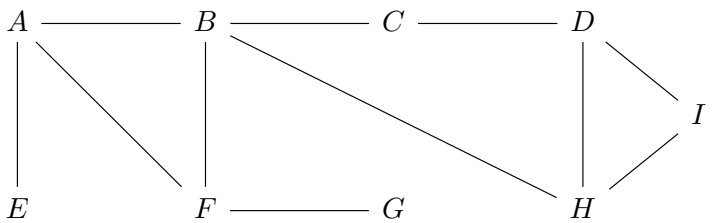
Given a graph G, one can consider a random walk on G. Informally, such a walk will entail choosing some starting vertex, and moving through other vertices based on probabilities associated with edges, and then ending after some condition has been met. One way to model such a random walk is with a finite state Markov chain and some stopping time.

Let us define a simple random walk on a graph $G = (V, E)$. Let V be our finite set of states. We construct a Markov chain of independent identically distributed random variables which map from state u_1 to state u_2 with probability 0 if no edge is between u_1 and u_2, and $\frac{1}{\deg(u_1)}$ if u_1 and u_2 are connected via some edge, where $\deg(u_1)$ is the number of edges incident to u_1. Our simple random walk is the walk generated by this markov chain along with some stopping time. Such a walk is very "uniform", leading to usefulness in a variety of randomized algorithms.

Whether working with a simple random walk, or some other model, one may want to develop tools to maintain some semblance of "randomness", while preserving certain properties of a graph. One such tool is loop erasure.

If studying random walks on graphs in the context of spanning trees, one would want our walks to have the property of not self-intersecting. However, it is difficult to engineer such a model. A simple random walk very clearly does not have this property. Thus, instead of embedding this property into our random model, it is easier to deterministically force this property onto a walk generated randomly.

Let G be a connected graph, and let π be a finite walk in G. We define the loop-erasure of π, denoted $\mathrm{LE}(\pi)$, as the walk generated from π by removing all cycles in order of visitation. That is, if v is the first vertex visited twice, then remove the section of the walk from after v is first reached to when v is reached for the second time. Repeat this procedure until no cycles remain. For example, consider the graph:



Consider the walk

$$\pi = A \to F \to G \to F \to B \to C \to D \to H \to I \to H \to B \to H \to I$$

We observe that the first repeated edge is F, and thus we remove the section of the walk after the first instance of F up to the second instance of F, giving us the walk

$$A \to F \to B \to C \to D \to H \to I \to H \to B \to H \to I$$

The next cycle is from H to itself, and removing this cycle leaves us with the walk

$$A \to F \to B \to C \to D \to H \to B \to H \to I$$

We can then remove the next cycle from B to itself

$$A \to F \to B \to H \to I$$

We are left with no cycles remaining, meaning

$$\text{LE}(\pi) = A \to F \to B \to H \to I$$

It is worth noting that if π is a random-walk generated from some stochastic process, there is no reason for $\text{LE}(\pi)$ to preserve any of the properties of our stochastic process. However, certain models and algorithms can be constructed which preserve useful properties under loop erasure.

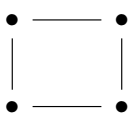
Wilson's Algorithm

Given a connected graph G, one can easily formulate an algorithm to generate a spanning tree of G. For instance, you could start at some vertex, and continue traversing through unvisited vertices until none remain, or you could instead choose to start with the whole graph and remove an edge from a cycle until no cycles remain. Such algorithms could be deterministic, or random. One randomized algorithm for such a task is Wilson's algorithm:

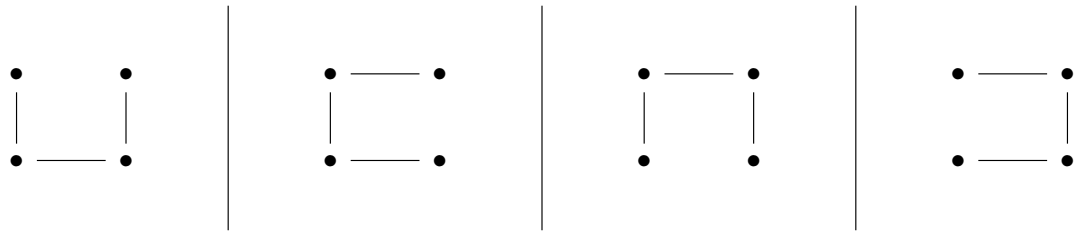
Require: Connected Simple Graph $G = (V, E)$
  Fix $r \in V$ arbitrarily.
  $T = (\{r\}, \varnothing)$
  while T is not a spanning tree of G **do**
    Fix an arbitrary vertex $v \in V$ such that $v \notin T$
    Run a simple random walk π starting from v, and ending whenever it reaches T
    Attach $\text{LE}(\pi)$ to T
  return T

One can show that Wilson's algorithm halts almost surely in finite time and produces a spanning tree of G [1]. One astounding property of this algorithm is that it generates a spanning tree uniformly. That is, the probability of producing any spanning tree of G with this algorithm is 1 divided by the total number of distinct spanning trees of G. It is also worth noting that when r and v are fixed arbitrarily, this can be done with any rule. For instance, one choose an r and run the algorithm with such an r. Similarly, one can choose v uniformly, randomly with certain probabilities of vertices, or even deterministically, using some preset order or deciding based on some notion of distance. Such choices do not impact the probability of producing any given spanning tree of G.

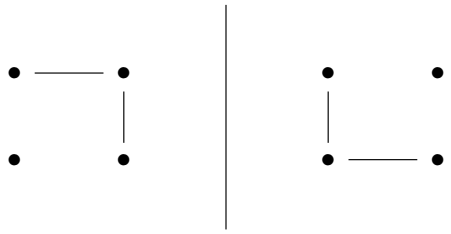
Let us consider the example graph G:



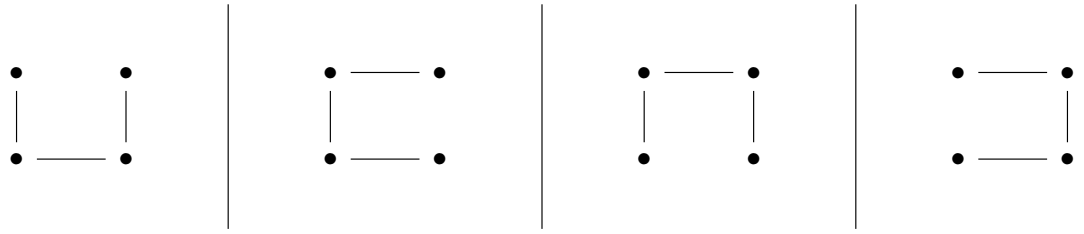
We have that there are 4 spanning trees of such a graph:



Let us consider running Wilson's algorithm on G, with r being the top left vertex, and our first choice of v being the bottom right vertex. From symmetry we have that our next step in the algorithm will have T be equal to either



with probability $\frac{1}{2}$. There is only a single choice for the next vertex. In the case of the left graph, we have the algorithm will choose the bottom left vertex, and then walk either up or right with probability $\frac{1}{2}$, and similarly in the right graph the algorithm will choose the top right vertex and walk left or down with probability $\frac{1}{2}$. Therefore, our algorithm will generate one of

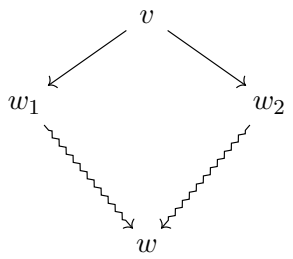


each with probability $\frac{1}{4}$. Thus, the algorithm generates a spanning tree of G uniformly.

The Diamond Lemma

A key algebraic/combinatorial result useful for proving the uniform nature of Wilson's algorithm is the Diamond lemma for single player games. Consider a single player game that consists of game states and moves. Moves are from states to other states, in one direction, and you can not traverse back to a previous state ever. That is, we can view our game as a directed acyclic graph. We define the diamond condition as such: Given an arbitrary state v, consider two arbitrary states w_1, w_2 which can be moved into from v in one game move. The diamond condition asserts that there exists a directed path from w_1 and w_2 which both arrive at some game state w. Visually, this condition is saying for any

v, w_1, w_2 we have the existence of some w as such:



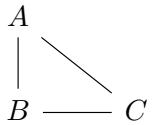
Informally, the condition asserts that irrelevant of any local choices, you can always end up back at some shared state. If this condition is true, the Diamond lemma states that the game has a unique end. That is no matter what choices of moves you make, the game will end up at the same final state. Intuitively, this means if your choices locally always can return to some shared state, then globally, no matter what choices you make, you will always end up at the same state. This intuition can be formalized as a combinatorial argument [5]. This is a useful tool for us, as although our Markov chains are not deterministic, we can gaurantee the final outcomes satisfy some property if each step of the Markov chain satisfies the diamond condition.

Markov Chain with Stacks and Cycle Popping

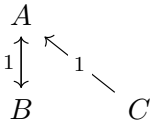
To prove that Wilson's algorithm generates a spanning tree uniformly, we need to decide on what probabilistic structures to use to model the algorithm. A convenient model is using a Markov Chain with Stacks.

Consider a graph G. Associate each vertex with a stack (or ordered list) that is initially empty. For each vertex v, populate its stack randomly with adjacent vertices, with probability $\frac{1}{\deg(v)}$. That is, each stack will contain vertices to move to from v, and such choices will be uniform. Consider starting at some vertex, moving to the vertex at the top of its stack, removing that vertex from the stack, and iterating. Although our stacks are random, once they have been made, doing such a process is deterministic. Such a process also mirrors a simple random walk from v using Markov Chains. Thus, our stochastic process in some way represents a "seeded" simple random walk.

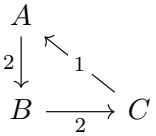
Let us consider the top of each stack (potentially after being depleted some times). We observe that the tops form a directed graph. We can color each directed edge with some color associated to the number of times the outgoing vertex has been visited. For example consider the graph



Let us say that the tops of the stacks on this graph originally form the directed graph



The color of each directed edge would be the color associated with the number 1. We could then pop the cycle from A to B, then revealing the directed graph



Here the of the arrow $C \to A$ will be 1, while the color of the arrows $B \to C$ and $A \to B$ will be 2. This is because the edge from C to A is still associated with the top of the stack of C, while the other arrows are associated with the second elements of the stacks of A and B, as their tops were removed when the directed cycle between them was removed.

We define this removal of directed cycles, and recording of colors as cycle popping. We can model cycle popping as a single player game. Each state represents some state of our stacks. That is each state is associated with some directed graph of G and coloring of edges. Our moves are from states into other states which are reachable via some cycle popping. Observe that any directed cycles must be disjoint in a graph. Therefore, if it is possible to remove two cycles c_1, c_2, we could remove them in any order. Therefore, our game trivially satisfies the diamond condition. The diamond lemma then states that there is a unique final state after performing all possible cycle poppings on our Markov chain with stacks.

Let us apply this model to Wilson's algorithm. We construct a Markov chain with stacks, letting the vertices of T (the vertices in our partial spanning tree), be absorbing states. Runing a random walk on some vertex v and then applying loop erasure is equivalent to cycle popping. We have shown that cycle popping is order independent. Therefore, there is some unique path, along with colorings of edges based on our stacks, from v to some vertex on T defined by our Markov chain on stacks. Equivalently we can say there is some unique set of cycles $\mathcal{C}$ which have been popped, as well as some path p.

After having completed the algorithm, we will be left with some spanning tree, as well our set of popped cycles. In our Markov chain with stacks everything is independent and identically distributed, thus

$$\mathbb{P}(\text{Ending our algorithm with } T \text{ and } \mathcal{C}) =$$

$$\mathbb{P}(\text{Popping all cycles in } \mathcal{C}) \cdot \mathbb{P}(\text{Having our algorithm's unique path } p \text{ define tree } T) =$$

$$(\prod_{c \in \mathcal{C}} \prod_{v \in c} \frac{1}{\deg(c)}) \cdot (\prod_{v \in T \setminus \{r\}} \frac{1}{\deg(v)})$$

We have $\prod_{c \in \mathcal{C}} \prod_{v \in c} \frac{1}{\deg(c)}$ as a cycle will be popped so long as it exists (by the uniquness of cycle popping), so for a cycle to get popped we just need it to have been on our stacks, which means we needed each edge to defined. Similarly we have $\prod_{v \in T \setminus \{r\}} \frac{1}{\deg(v)}$ as we need our stacks to define T after depleting all cycles, which is done by having each stack have the top after all depletions be the vertex towords r in T. We observe that this product is independent of T. That is $(\prod_{c \in \mathcal{C}} \prod_{v \in c} \frac{1}{\deg(c)})$ is independent of T, and $(\prod_{v \in T \setminus \{r\}} \frac{1}{\deg(v)})$ is indpendent of T, as $T \setminus \{r\}$ will contain every vertex except r irrelevant of what T is. Therefore we have

$$\mathbb{P}(\text{Wilson's algorithm generates } T) = \sum_{\mathcal{C}} \mathbb{P}(\text{Ending our algorithm with } T \text{ and } \mathcal{C})$$

which is independent of T. Therefore, the probability of generating T is indepedent of T, meaning Wilson's algorithm generates T uniformly [1].

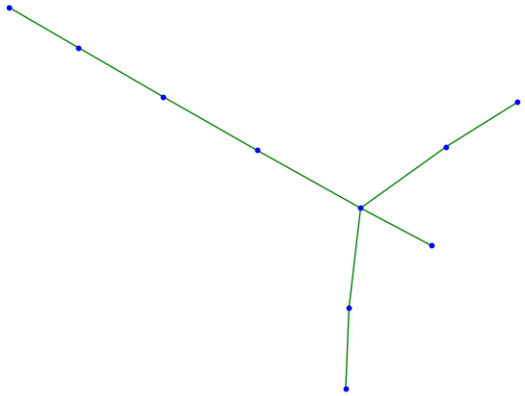
Let us summarize the intuition behind this proof. We notice that our spanning tree corresponds to some directed acyclic graph on our stacks. This directed acyclic graph is then underneath layers of cycles in our stacks. The order in which we pop the cycles does not matter, so the probability of getting any tree is the sum over all possible combinations of cycles of the probability of getting a tree multipled by the probability of having those cycles on top of our tree in our stacks. The tree underneath our cycles does not affect our cycles at all, so the probability of having some set of cycles and then ending at any tree is the same regardless of the tree.

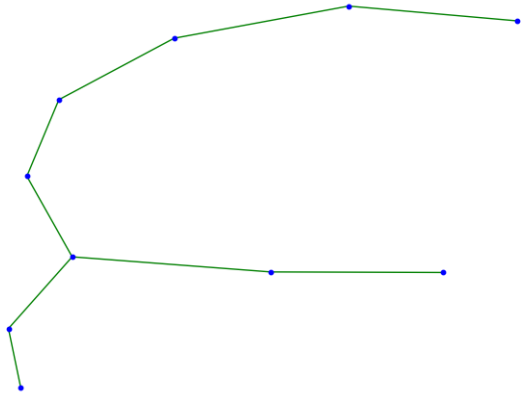
Implementation of Wilson's Algorithm

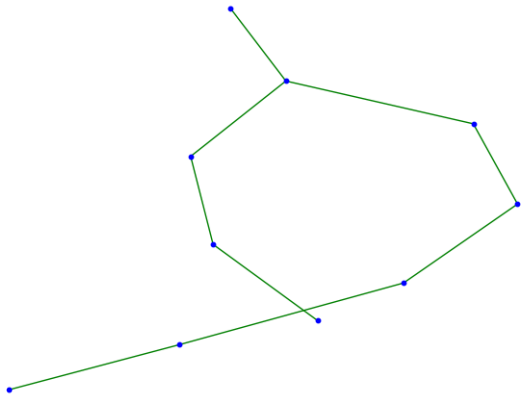
Wilson's algorithm lends it self well to implementation. Here is one such implementation:

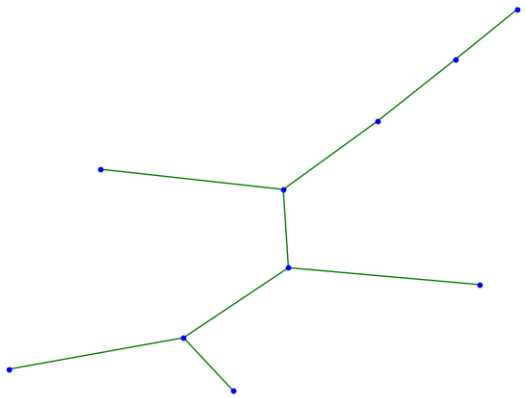
```
from random import randint

# Uniformly choose a vertex adjacent to v on graph G
def rand_vertex(G, v):
    return G[v][randint(0, len(G[v]) - 1)]


# Generates a random spanning tree of a
# connected graph G, taken in as an adjacency list
def gen_tree(G):
    root = 0
    T = [[] for i in range(len(G))]

    for v in range(len(G)):
        if T[v] or v == root:
            continue

        walk = [v]

        while not T[walk[-1]] and walk[-1] != root:
            walk.append(rand_vertex(G, walk[-1]))

        rightmost = {}
        for i in range(len(walk) - 1, -1, -1):
            if walk[i] not in rightmost:
                rightmost[walk[i]] = i

        prev = 0
        curr = max(1, rightmost[walk[prev]] + 1)

        while curr < len(walk):
            if curr != rightmost[walk[curr]]:
                curr = rightmost[walk[curr]] + 1

            T[walk[prev]].append(walk[curr])
            T[walk[curr]].append(walk[prev])

            prev = curr
            curr += 1

    return T
```

This implementation can also be found here. With it, we numerically verify our claim in the case of the square graph. We can also generate spanning tree's of famous graphs, such as the Peterson Graph (albeit drawn unorthodoxly).

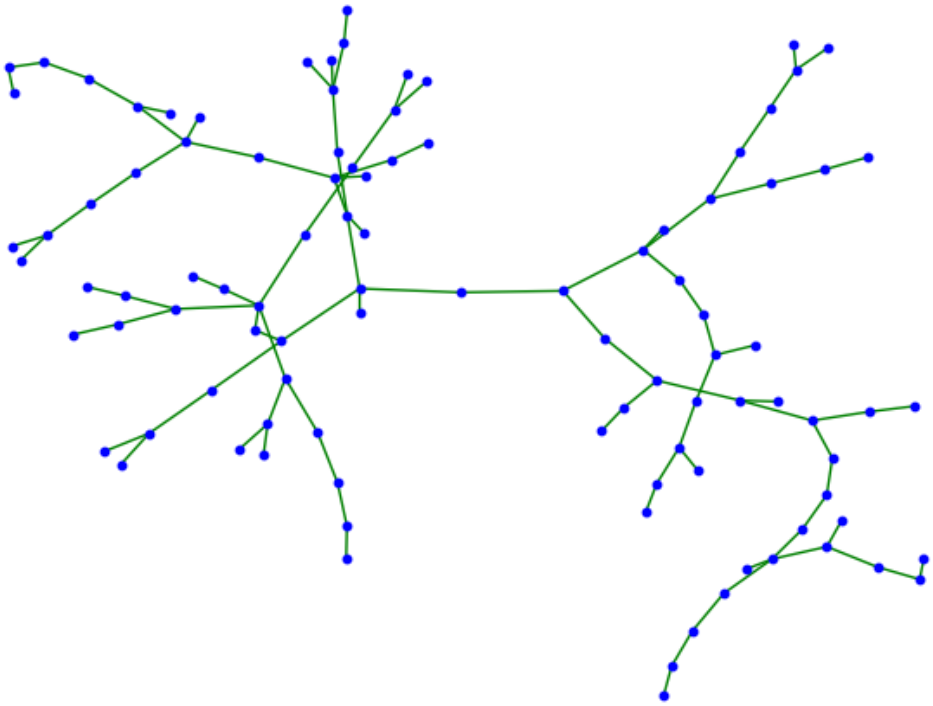








Or one could run the algorithm on a complete graph of 100 vertices 1000 times over, generating something that looks completely chaotic, yet in some way is completely uniform.



References

[1] David Bruce Wilson. *Generating random spanning trees more quickly than the cover tim*. Cambridge, Massachusetts, 1996. DOI: 10.1145/237814.237880.

[2] Richard P. Stanley. *Algebraic combinatorics: Walks, trees, tableaux, and more*. 2nd ed. Basel, Switzerland: Springer International Publishing, 2018. ISBN: 9783030083892.

[3] *Loop-erased random walk*. Mar. 2024. URL: https://en.wikipedia.org/w/index.php?title=Loop-erased_random_walk&oldid=1214260654.

[4] *Implementation of Wilson's Algorithm*. URL: https://github.com/Ca7Ac1/Loop-Erased-Walks.

[5] *The Diamond Lemma and its applications*. URL: https://web.stanford.edu/~niccronc/notes/The%20Diamond%20Lemma%20and%20its%20applications.pdf.