SAMPLING SPANNING TREES
THEORY AND ALGORITHMS

Nathan Benedetto Proença
supervised by Marcel K. de Carli Silva
Instituto de Matemática e Estatística — USP

THE PROBLEM

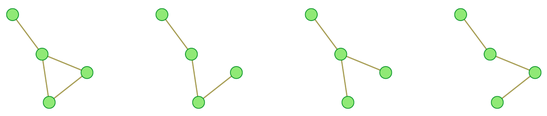

Figure 1: A graph and its spanning trees

A graph usually has many spanning trees. Moreover, The amount of different spanning trees can be exponential on the number of vertices. This work explores the idea of specifying a probability distribution on this set — the set of spanning trees of a given graph — and to sample one of them accordingly.

A convenient way of specifying a probability distributions on the set of spanning trees is to specify weights for the edges, and then define the probability of a tree to be proportional to the product of those weights. This is the general problem explored in the monograph.

However, for a cleaner exposition, here we will focus on the special case when every tree has the same probability. In other words, our problem is to design a polynomial time algorithm that, given a graph, returns any of its spanning trees with equal probability.

WHY CARE?

There are two interesting aspects around this problem: The algorithms that rely on a polynomial time sampling of a spanning tree, and the theory used in solving the problem.

The first remarkable application is an algorithm developed in [Fri+14], which uses random spanning trees to build expander graphs. A complete survey about expander graphs can be found in [HLW06], which begins by highlighting its applications in error correcting codes, pseudo-number generators and complexity theory results.

Another noteworthy algorithm is described in [Asa+10]. It develops a $O(\log n / \log\log n)$-approximation to a variant of the Travelling Salesman Problem, using spanning tree sampling as a subroutine.

As regards the theory itself, it is extremely interesting to note how different areas of mathematics come into play. Linear algebra is a central topic in the monograph. It is not, however, with the usual primal-dual results; determinants and matrix inversion form the foundation on which the first two polynomial time algorithms are developed.

Moreover, Aldous in [Ald90] and Broder in [Bro89] solved the problem with an extremely elegant algorithm, based on random walks on graphs. Therefore, to proper understand the correctness and running time of these algorithms, it is necessary to use results from probability theory and Markov chains.

EDGE CONTRACTION

The first idea needed for the naive algorithm is that to count spanning trees that contain a certain edge is equivalent to count spanning trees on the graph obtained by contracting that edge.

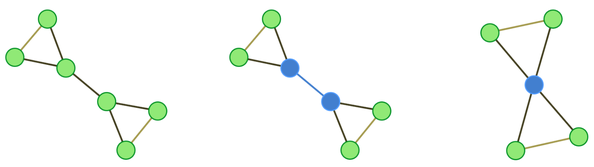

Figure 2: Edge contraction

Therefore, we can count how many trees contain an edge if we can count how many trees there are in a smaller graph. There is, however, a price. To work with edge contraction is to give up on simple graphs.

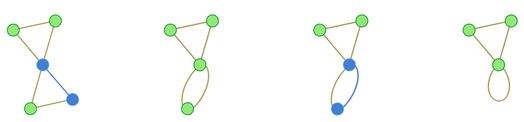

Figure 3: Edge contraction can create loops and parallel edges from simple graphs

This reduces the work of sampling a spanning tree into the work of calculating the probability of an edge to be in the output tree.

With such a probability in hand, we can decide if a given edge will be in the output of the algorithm.

If it is on the answer, the new problem is about sampling on the graph obtained by contracting that edge. If it is not on the answer, we can simply remove the edge from the graph and then sample a spanning tree in it.

NAIVE ALGORITHM

It is interesting to look at the algorithm itself, leaving the question about calculating the probabilities aside. Given an order in which the edges will be processed, the possible choices for the algorithm can be expressed as a tree. In Figure 4, the edges in blue are the ones being considered.

The choices are color coded. The darker arrows correspond to adding the edge to the final tree, and the lighter ones correspond to dropping them.

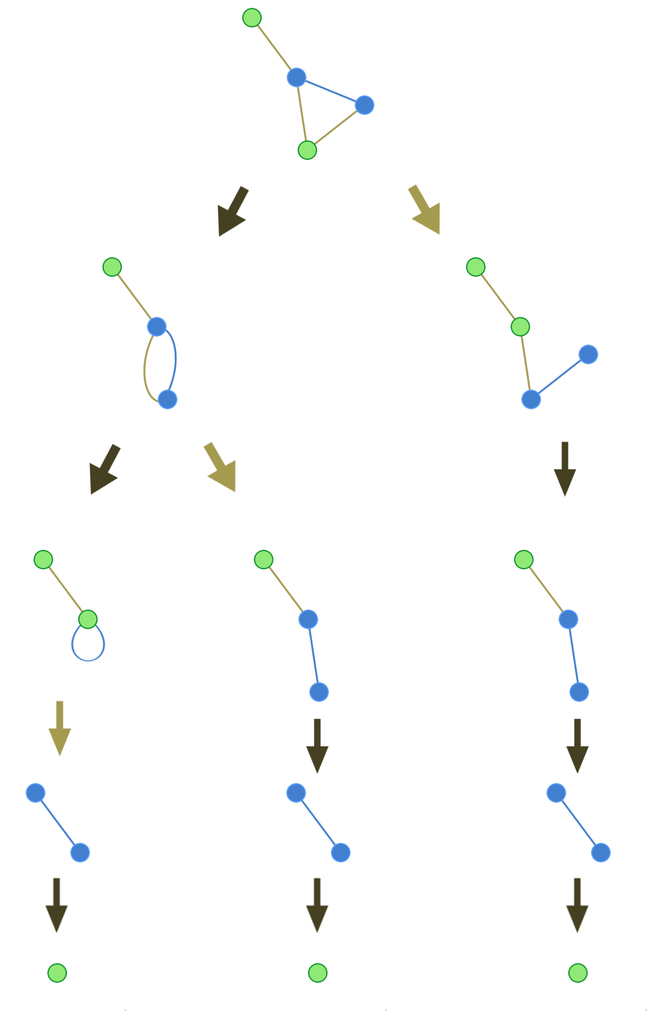

Figure 4: Possible executions of algorithm

Note that every leaf represents a different spanning tree.

Moreover, Figure 4 has nothing random about it. It is completely determined by the graph and the sequence in which the edges are considered. To actually generate a random tree, we must introduce randomness.

A sampling is a path from the root to a leaf. Whenever there is a choice, pick a uniformly distributed random number x in the interval $[0, 1]$. If x is less then the probability of the edge being in the spanning tree, go left. Otherwise, go right.

The algorithm is not producing randomness, it is merely transforming randomness. This is a simple consequence of the fact that every algorithm is, by definition, deterministic.

HOW TO CALCULATE MARGINAL PROBABILITIES?

A procedure similar to the described above can actually be used for calculating the amount of spanning trees in a graph, and, therefore, the probability of an edge belonging in a tree. Unfortunately, this is exponential on the amount of edges in the graph. Kirchhoff presented a better approach, with a result which is known as Kirchhoff's Matrix Tree Theorem.

Given a graph G, let A be its adjacency matrix, and D be a matrix with the vertices' degree on the main diagonal. Then it is possible to define the Laplacian L of the graph as

$$L = D - A.$$

The Laplacian arises naturally when studying electric circuits. If the edges are considered as unit resistors, then it calculates the current accumulated in every vertex, given the voltages in every vertex.

For our purposes, the almost magical result is that the number of spanning trees of a graph is given by

$$\det(L_{ii}),$$

where L_{ii} is the matrix obtained from L by deleting the i-th row and the i-th column.

This determinant can be calculated in polynomial time, so that our naive algorithm has running time $O(n^\omega m)$, where ω is the best-known exponent for matrix multiplication.

SPEEDING UP THE ALGORITHM

When working with electric circuits, it is interesting to measure how much does the circuit as a whole poses as a resistance between two nodes in it. This number is called the effective resistance.

To calculate it, it is necessary to solve a system of linear equations involving the Laplacian of a graph. An algebraic way to work with this equation solving is to do calculations with the Moore-Penrose pseudoinverse of the Laplacian.

As the names suggests, the pseudoinverse generalizes the idea of the inverse. It does so by encoding the process of finding the linear least square solution of a system of equations in a matrix. For a given matrix A, its pseudoinvserse is denoted $A^\dagger$. With such a powerful tool, it is possible to write the effective resistance between two nodes i and j as

$$(e_i - e_j)^\mathsf{T} L^\dagger (e_i - e_j).$$

How can this help speed up the algorithm? Given an edge ij in the graph, the effective resistance between i and j is equal to the probability that ij belongs to a spanning tree sampled uniformly.

This is helpful since it reduces the work of calculating a determinant into finding the value of 4 positions in the pseudoinverse matrix. Moreover, it is possible to update the pseudoinverse quickly whenever a edge deletion or contraction happens. [HX16] explores this idea, and describes an algorithm that runs in $O(n^\omega)$ time.

AS SIMPLE AS A WALK

With the previous algorithms in context, it becomes even more remarkable that, to sample a spanning tree, suffices to do a random walk on the graph, picking any adjacent vertex with equal probability, and to output the edge used to first visit each vertex as the spanning tree.

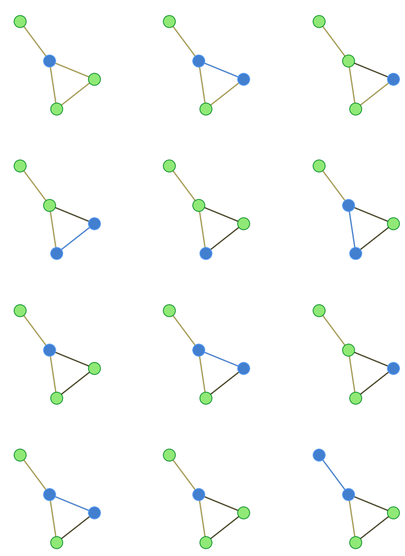

Figure 5: Possible trace of random walk based algorithm. It can do some unnecessary work.

It is an interesting problem, tackled by the monograph, to show that this approach is indeed correct, and to analyze its running time. The algorithm terminates as soon as it has visited every vertex in the graph, so that suffices to bound the expected value of this number. This is what is called the cover time.

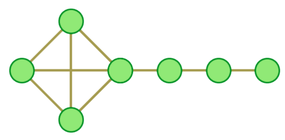

Figure 6: Worst case for random walk based algorithm

Unfortunately, the cover time can be $\Omega(n^3)$ in graphs like the one in Figure 6. The problem is that if the walk starts in the clique, it will take too much time to go into the path, and there is no guarantee that it will not fall back into the clique again. Therefore, the random walk based algorithm is $O(n^3)$.

REFERENCES

[Bro89] Andrei Broder. "Generating random spanning trees". In: *Proceedings of the 30th Annual Symposium on Foundations of Computer Science, FOCS 1989*. 1989, pp. 442–447.

[Ald90] David J. Aldous. "The random walk construction of uniform spanning trees and uniform labelled trees". In: *SIAM J. Discrete Math.* 3.4 (1990), pp. 450–465. ISSN: 0895-4801. DOI: 10.1137/0403039. URL: http://dx.doi.org/10.1137/0403039.

[HLW06] Shlomo Hoory, Nathan Linial, and Avi Wigderson. "Expander graphs and their applications". In: *Bull. Amer. Math. Soc. (N.S.)* 43.4 (2006), 439–561 (electronic). ISSN: 0273-0979. DOI: 10.1090/S0273-0979-06-01126-8. URL: http://dx.doi.org/10.1090/S0273-0979-06-01126-8.

[Asa+10] Arash Asadpour et al. "An $O(\log n / \log\log n)$-approximation algorithm for the asymmetric traveling salesman problem". In: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, Philadelphia, PA, 2010, pp. 379–389.

[Fri+14] Alan Frieze et al. "Expanders via random spanning trees". In: *SIAM J. Comput.* 43.2 (2014), pp. 497–513. ISSN: 0097-5397. DOI: 10.1137/120890971. URL: http://dx.doi.org/10.1137/120890971.

[HX16] Nicholas J. A. Harvey and Keyulu Xu. "Generating Random Spanning Trees via Fast Matrix Multiplication". In: *LATIN 2016: Theoretical Informatics: 12th Latin American Symposium, Ensenada, Mexico, April 11-15, 2016, Proceedings*. Ed. by Evangelos Kranakis, Gonzalo Navarro, and Edgar Chávez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 522–535. ISBN: 978-3-662-49529-2. DOI: 10.1007/978-3-662-49529-2_39. URL: http://dx.doi.org/10.1007/978-3-662-49529-2_39.