The Expected Complexity of Prim's Minimum Spanning Tree Algorithm

Chip Martel

Dept. of Computer Science
University of California
Davis, CA 95616

martel@cs.ucdavis.edu

May 15, 2001

Abstract

We study the expected performance of Prim's minimum spanning tree (MST) algorithm implemented using ordinary heaps. We show that this implementation runs in linear or almost linear expected time on a wide range of graphs. This helps to explain why Prim's algorithm often beats MST algorithms which have better worst-case run times.

Specifically, we show that if we start with any n node m edge graph and randomly permute its edge weights, then Prim's algorithm run's in expected $O(m + n \log n \log(2m/n))$ time. Note that $O(m + n \log n \log(2m/n)) = O(m)$ when $m = \Omega(n \log n \log \log n)$.

We extend this result to show that the same expected run times apply even when an adversary can select the weights of $m/\log n$ edges and the possible weights of the remaining edges (which are then randomly assigned).

1 Introduction

Finding a minimum spanning tree (MST) is a classic problem which has extensive applications to a variety of settings. There is a vast literature extending back to the papers of Kruskal and Prim in the 1950's [Pri57, Kru56] and continuing with many improved algorithms including Fredman and Tarjan's Fibonacci heap implementation of Prim's algorithm [FT87], Gabow, Galil, Spencer and Trajan's nearly linear time algorithm [GGST86], and the recent expected linear time algorithm of Karger, Klein and Tarjan [KKT95] (an extensive discussion of MST algorithms is given in [MS94]). However, the range of worst case run times of these algorithms is actually fairly small: for a graph with m edges and n nodes any algorithm must take $\Omega(m)$ time and even the oldest algorithms take $O(m \log m)$ time if implemented using standard data structures (we assume throughout that our graph is connected so $m = \Omega(n)$; otherwise we can always start by finding the connected components in linear time and then find the MST in each component).

Since the gap between the best and worst algorithms is only a log factor, practical performance may not be predicted well by worst case asymptotic run times. In particular, constant factors and performance on typical problems is likely to be important. The experimental study of MST algorithms by Moret and Shapiro [MS94] supports this view. Their experiments suggest that Prim's algorithm implemented using ordinary heaps is the best algorithm for dense random graphs and is competitive with other algorithms in most settings. In this paper we analyze the expected performance of Prim's algorithm. We show that if we start with an arbitrary graph and then randomly permute the edge weights, then Prim's algorithm using ordinary heaps runs in expected $O(m + n \log n \log(1 + m/n))$ time. We also show that this expected run time holds even if an adversary gets to select the graph topology, the set of possible edge weights, and the actual weights of $m/\log n$ edges, as long as the remaining edge weights are assigned randomly.

Note that $O(m + n \log n \log(1 + m/n)) = O(m)$ as long as $m = \Omega(n \log n \log\log n)$. Thus this implementation runs in expected linear time except on sparse graphs.

This behavior was suggested by Moret and Shapiro and by Noshita [Nos85]. Noshita proved an almost identical result for Dijkstra's algorithm except that he assumed the edge weights were independent, identically distributed random variables (which is almost the same as a random permutation of the edges for the MST setting). Our proof uses the same general approach as Noshita's, and our main technical lemmas 2.1 and 2.2 are analogous to results he proved for Dijkstra's algorithm. However, our details are somewhat different due to the differences between Prim's and Dijkstra's algorithms, and due to the differences in our models of random edge weights. Our paper also gives a clearer and more general characterization of the graphs for which both the MST and shortest path results apply.

2 Expected new minimums for Prims

Our analysis is for the following implementation of Prim's algorithm. For each vertex v not yet in the tree we keep a value *near(v)*, the cheapest edge weight connecting v to a vertex in the tree, and store the *near* values in a min-heap.

Algorithm Prim-Heap

We start by initializing our tree T to contain an arbitrary vertex s.
For each neighbor u of s set *near(u)* to $w(u, s)$, the weight of the edge (u, s).
All other vertices have their *near* value set to infinity.

Now the algorithm adds the other $n - 1$ vertices as follows:

(1) Find the vertex v not in T with minimum *near* value;

(2) **For** each neighbor u of v,
if $(w(u, v) < near(u)$ and u not in $T)$ **then** $near(u) \leftarrow w(u, v)$;

(3) Add v to T;

The running time of Prim-Heap is dominated by steps (1) and (2). Step (1) is done using a delete min operation in $O(\log n)$ time, (2) requires looking at each neighbor. Whenever *near* is updated a *decrease-key* operation is used which takes $O(\log n)$ time. Thus the total worst case time is $O(n \log n)$ for step (1) and $O(m \log n)$ for step (2), making step (2) the dominant step for connected graphs. Despite this superlinear worst case behavior, this algorithm exhibits linear behavior for dense random graphs [MS94].

An explanation of this behavior is given in [MS94, Nos85]. If a vertex u has degree d and the neighbors of u are added to T in a random order, then the first neighbor added to T will always cause a decrease key, the second will cause a decrease key half the time (when its weight is smaller than the weight of the first one added), the third one a third of the time, ... Thus the expected number of decrease keys is:

$$E[decrease - keys] \leq 1 + 1/2 + \ldots + 1/d \;=\; H_d \tag{1}$$

Since H_d, the dth harmonic number, is well approximated by $\log_e(d)$, they argue that one should expect $O(\log_e(d))$ decrease-key operations for a vertex with degree d over the course of the algorithm. The above discussion assumes the neighbors of u are added so that their weights form a random permutation. However, in analyzing Prim's algorithm we have to be a little careful since the order in which the neighbors of a vertex are added does depend on both the graph topology and their weights. Below we give a formal justification for this run time analysis and show this holds for any graph topology (random or not) as long as the edge weights are randomly assigned (in fact our analysis will hold even if an adversary can choose the weights of up to $m/\log n$ edges.)

For our analysis let G be an arbitrary undirected graph with a designated vertex s which will be the first one added to the tree. We start by proving a key technical lemma which shows that until u is added to T, the order $u's$ neighbors are added to T is independent of the weights of the edges incident to u.

We define G_u to be the graph we obtain if we take G and modify it by changing the weights of all edges incident to u so they are larger than any other edge in the graph.

Lemma 2.1 *Let G be an undirected graph, s an arbitrary start node and u an arbitrary vertex in G. When we run Prim-Heap on G_u using start node s, let $x_1, x_2, \ldots, ..., x_k$ be the*

neighbors of u which are added to T before u is addd to T. The indices indicate the order they are added to T.

Let H be any graph obtained by taking G and modifying the weights of the arcs incident to u. When we run Prim-Heap on H using start node s, if r neighbors of u are added to T before u is added, those r neighbors added must be $x_1, x_2, \ldots, x_r$ (with $r \leq k$) and they are added in that order.

Proof First note that if removing u doesn't disconnect G, u will be the last vertex added to T in G_u and k is the degree of u. If removing u does disconnect G, then u will be added after all other vertices in the connected component which contains s in the graph resulting from removing u.

Now consider Prim-Heap running on H and on G_u. Note that the *near* values depend only on the edges incident to vertices which are already in T. Thus until u is added to T, the *near* values of every vertex except u are identical for both H and G_u. Thus until u is added to T the algorithm will make the same choices of which vertices to add next in step (1) regardless of the weights of $u's$ incident edges. □

Lemma 2.2 *Let G be an arbitrary undirected graph with a start vertex s. If we take any vertex u of degree d and randomly permute the weights of the edges incident to u to create a new graph G^p, then when we run Prim-Heap on G^p using start vertex s, the expected number of decrease keys to update* near*(u) is at most H_d.*

Proof

We only update *near*(u) before u is added to T. By lemma 2.1, until u is added to T the neighbors of u are added to T in a fixed order which is always a prefix of $x_1, x_2, \ldots, x_k$, (independent of the weights of $u's$ incident edges).

In G^p, each relative ordering of the weights assigned to the edges $(x_1, u), (x_2, u), \ldots (x_k, u)$ is equally likely, and a random permutation will make the jth value smaller than the first $j-1$ values with probability $1/j$. Thus the expected number of updates to *near(u)* is at most:

$$1 + 1/2 + \ldots + 1/k \;=\; H_k \leq H_d \tag{2}$$

The above result about random permutations is well known and is proved in [Nos85]. Note that this actually overestimates the total number of updates since we assume all k neighbors are added prior to u being added and we ignore possible duplicate weights among $u's$ neighboring edges which would also reduce the number of updates.

□

Theorem 2.1 *Let G be an arbitrary undirected connected graph with n vertices and m edges. We pick the start vertex s and then the edge weights in G are randomly permuted to form G^p. When we run Prim-Heap on G^p, starting with vertex s, the expected number of decrease-keys is at most $nH_d \leq n \log(2m/n))$ where $d = 2m/n$.*

Proof Let $d_1, d_2, \ldots, d_n$ be the degree of the vertices in G. Each vertex satisfies the properties of Lemma 2.2, so the expected number of decrease-keys for vertex i is $\leq H_{d_i}$. The

total number of expected decrease-keys is the sum of the expected number for each node, so $\sum_{i=1}^{n} H_{d_i} \leq nH_d \leq n\log(2m/n)$. The first inequality uses the fact that $\sum_{i=1}^{n} d_i = 2m$ and that the sum of the logs is maximized when each element in the sum (here the d_i values) is equal to the average.

□

To analyze the entire algorithm's running time, recall that the total time is $O(m+n\log n+D)$ where D is the time for decrease-keys. Each decrease-key takes $O(\log n)$ time. Thus we get a total expected running time of $O(n\log(m/n)\log n)$ for the decrease-keys. When $m = n\log n$ this is $O(n\log\log n\log n)$ and for $m = n\log n\log\log n$ it is $O(n(\log\log n + \log\log\log n)\log n) = O(m)$. In general if $m = \Omega(n\log n\log\log n)$ the expected work for decrease-keys will be $O(m)$ and thus the overall algorithm runs in expected $O(m)$ time which is optimal.

As Moret and Shapiro observed [MS94], this linear expected performance on dense random graphs helps to explain why normal heaps tend to outperform Fibonacci heaps in these settings. Though Prim's with Fibinacci heaps has better worst case performance than ordinary heaps on dense graphs ($O(m)$ versus $O(m\log n)$), our analysis shows both take expected $O(m)$ time on dense graphs with random edge weights. Since ordinary heaps have smaller constants for the extract-min operation it is not surprising that experiments usually show ordinary heap implementations as faster for this setting. A more extensive analysis of this is given in [GT96] (though their focus is on Dijkstra's algorithm, the analysis is still relevant to Prim's algorithm).

3 Other Settings

The prior analysis assumed all the edge weights were distinct. We now consider the case where there can be duplicate edge weights. Suppose that the edge weights are independent, identically distributed (iid) random variables. The operation of Prim's algorithm only depends on the relative size of the weights not their values. Thus we get results analogous to lemma 2.2 and Theorem 2.1: if the edge weights incident to a vertex u of degree d are idd random variables, then the expected number of updates to *near(u)* is $O(\log d)$ and if all edge weights in the graph are iid random variables, then the total number of expected updates is $O(n\log(2m/n))$.

Of course in the case of iid random variables there may be duplicate edge weights which will reduce the number of updates to near.

The prior results show that Prim-Heap has good expected performance when the edge weights are independent of the topology. However, it is easy to imagine settings where the edge weights would depend on the topology (e.g. all arcs connected to a given node are expensive, there are few arcs between nodes which are far apart and they are all expensive, ...). It is worth noting however, that the results of Theorem 2.1 will still apply as long as only a few arcs are correlated with the topology.

In fact, we can prove the following more general theorem. We show that as long as the final step of graph construction is a random assignment of most of the edge weights, our results apply even when an adversary can specify a significant portion of the graph.

Theorem 3.1 *For any vertex size n, let G be an n-vertex graph constructed as follows: an adversary begins by choosing,*
(i) m, the number of edges in G;
(ii) which edges are in G;
(iii) the weights of $m/\log n$ edges in G;
(iv) a multiset W containing $m - m/\log n$ values;
(v) the start vertex s;

Now, (vi) the remaining edge weights are set by selecting values uniformly at random from W (this selection can be either with or without replacement).

The expected running time of Prim-Heap on G using start vertex s is $O(m+n\log n\log(2m/n))$.

Proof

If each of the $m/\log n$ edges set in step (iii) causes a decrease-key operation, the total time for them is $m/\log n \times O(\log n) = O(m)$. For the remaining edges we now show that the expected work associated with them matches the analysis in theorem 2.1.

Lemma 2.1 still applies to any vertex in G (since it applies to any weight assignment). Thus for any node u, until u is added to T its incident arcs with weights set in step (vi) are added in a fixed order, so the analysis of lemma 2.2 applies to them, and the bound of theorem 2.1 applies to the number of updates due to edges set in step (vi). Thus the total expected time is no worse than $\mathrm{O}(m + n\log n\log(2m/n))$. □

This theorem suggests that we can expect good expected performance by Prim-Heap except in fairly pathological settings.

4 Dijkstra's Algorithm

Dijkstra's single source shortest path algorithm is structurally quite similar to Prim's algorithm. As noted earlier, Noshita [Nos85] proved expected run time results for Dijkstra's which are quite similar to lemma 2.2 and theorem 2.1 (and his proof uses a result very similar to lemma 2.1). His analysis is for the case where the weight of each edge is an iid random variable, but as in our analysis, this can be extended to settings where the edge weights are randomly permuted. Because lemmas 2.1 and 2.2 both apply to Dijkstra's algorithm, it is also easy to prove an analogue of theorem 3.1 which shows that even if an adversary constructs a graph as in that theorem, Dijkstra's algorithm will still run in expected $\mathrm{O}(m + n\log n\log(2m/n))$ time. In fact, for directed graphs, we can even allow the adversary to choose a multi-set of values for the unset edges entering each vertex (thus for a vertex u, the adversary selects a set of values for the remaining edges entering u, and these values are now randomly assigned).

5 Acknowledgements

This work was partly supported by NSF grant 0085961. I'd like to thank Eric Key for some useful results on repeats. I'd also like to thank Matt Levine and Andrew Goldberg for some helpful pointers.

References

[FT87] M. Fredman and R.E. Tarjan. Fibonaci heaps and their use in improved network optimization algorithms. *Journal of the A.C.M.*, 34:596–615, 1987.

[GGST86] H. Gabow, Z. Galil, T. Spencer, and R. Tarjan. Efficient algorithms for finding minimum spanning trees in directed and undirected graphs. *Combinatorica*, 6:109–122, 1986.

[GT96] A. Goldberg and R.E. Tarjan. Expected performance of dijkstra's shortest path algorithm. Technical Report 96-062, NEC Research Institute, June, 1996.

[KKT95] D. Karger, P. Klein, and R. Tarjan. A randomized linear-time algorithm for finding a minimum spanning tree. *Journal of the A.C.M.*, 42:321–328, 1995.

[Kru56] J.B. Kruskal. On the shortest spanning tree of a graph and the traveling salesman problem. *Proceedings of the American Math society*, 7:48–50, 1956.

[MS94] B. Moret and H. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Series in Discrete Math and Theoretical CS*, 15:99–117, 1994.

[Nos85] K. Noshita. A theorem on the expected complexity of dijkstra's shortest path algorithm. *Journal of Algorithms*, 6:400–408, 1985.

[Pri57] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.