Chapter 19

Minimum Spanning Trees II: A Linear-Time Randomized Algorithm

In this chapter, we will see a randomized algorithm for MSTs in expected *linear* time.

19.1 A Linear Time Randomized Algorithm for MST

In the previous chapter, we went over the basics of MSTs, the classical algorithms for it, and then saw the Fredman-Tarjan algorithm that solves this problem deterministically in $O(m \log^*(n))$ time. Given the extremely slow growth rate of the $\log^*(n)$ function, for all "practical" purposes, this is as good as a linear time algorithm (which is the optimal time for the problem). But, mathematically speaking, $\log^*(n)$ still goes to $+\infty$ with n, no matter how slowly, and thus, this runtime is still $\omega(m)$ truly.

In this chapter, we go over an algorithm for MSTs due to [KKT95]—termed the *KKT* algorithm—that runs in $O(m)$ time, which is the optimal runtime, but it is randomized and its $O(m)$ runtime is in expectation.

Theorem 19.1 ([KKT95])**.** *There is a randomized algorithm for the minimum spanning tree problem that runs in $O(m)$ time in expectation and with high probability.*

We will only prove the runtime of the algorithm in expectation (and leave the high probability bound to exercises). Recall the *cut rule* and *cycle rule* from Section 18.1.1.

The general idea behind the KKT algorithm is to use the **cycle rule** to get rid of most edges of the graph quickly, namely, *sparsify* the graph, and then solve the problem recursively on this sparser graph. To present this algorithm, we need some preliminaries first.

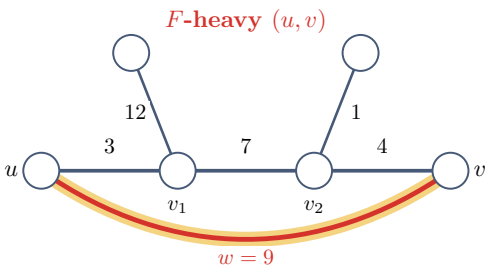
19.1.1 Preliminaries

The following definition is key to the design of *KKT algorithm*.

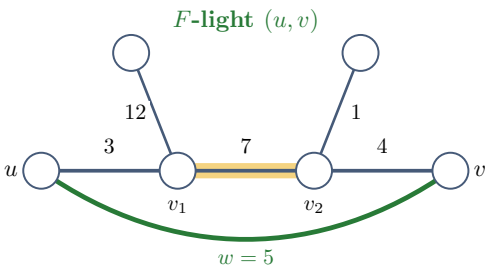
Definition 19.2. Fix any graph $G = (V, E)$ and any forest F that is a *subgraph* of G. We say that an edge $e \in E \setminus F$ is **F-heavy** if adding e to F results in a cycle and e is the maximum weight edge of that cycle. We refer to any other edge as an **F-light** edge.

Figure 19.1 illustrates this definition on a small example.

The following is a direct corollary of the **cycle rule** and the definition of F-heavy edges.



(a) The non-tree edge (u, v) has weight 9, larger than all weights on the u–v path in F: (u, v) is the maximum-weight edge of the cycle it closes and is F-heavy.



(b) Here (u, v) has weight 5 and the maximum-weight edge of the cycle is instead the path edge (v_1, v_2) of weight 7; thus (u, v) is F-light.

Figure 19.1: Here F is all edges except for (u, v) and adding (u, v) to F closes a cycle with the u–v path in F. The edge (u, v) is F-heavy iff it is the maximum-weight edge of that cycle (marked by the yellow halo). By the cycle rule, an F-heavy edge is never in the MST.

Observation 19.3. *For any graph G, any forest F that is a subgraph of G, and any F-heavy edge e, the MST of $G \setminus \{e\}$ is the same as the MST of G.*

How do we use Observation 19.3 in the algorithm? Suppose first that F is the MST of G; then *every* edge $e \in E \setminus F$ is F-heavy and thus can be neglected when computing the MST of G. Obviously however, this is not helpful as we need to first compute the MST of G. But, what if we have some other forest F which is easier to compute than the MST? Then, Observation 19.3 tells us that we can still neglect *every* F-heavy edge without any worry. Thus, in order to find the MST of G, we can first try to quickly find "some approximate" forest F, with the key property that *most* edges of the graph are F-heavy, and then recursively solve the problem on the remaining few F-light edges. This is precisely what the KKT algorithm does.

There is one more missing ingredient in the above approach. Given a graph G and a forest F, how quickly can we find the set of F-heavy edges? It is easy to check if a *single* edge is F-heavy or not in linear time. But, doing this for every edge this way separately leads to a quadratic time algorithm which is way above our budget. Nevertheless, a surprising fact is that we can find *all* F-heavy edges in linear time as well!

Theorem 19.4 ([Kom85, DRT92, Kin97])**.** *There is an algorithm that given any graph G and any forest F which is a subgraph of G, outputs the set of all F-heavy edges in $O(m + n)$ time.*

We will not cover this algorithm and just take it for granted. We only mention that there is a great deal of algorithmic work on this problem, typically under the name of *MST verification* algorithms (since having such an algorithm allows us to detect if a given tree F is an MST or not; the MST of the graph is the only one that makes all other edges F-heavy).

19.1.2 The KKT Algorithm

We are now ready to present the *KKT algorithm*.

We note that given the recursive nature of the algorithm and since it can be called on a not-necessarily-connected graph, we use the term *Minimum Spanning Forest (MSF)* throughout which refers to a collection of MSTs on each connected component of the graph.

Algorithm 19.5.

1. Run 3 rounds of Boruvka's algorithm and let G' be the resulting contracted graph.[a]
2. Sample each edge of G' independently with probability $1/2$ to obtain a graph G_1. **Recursively** find the MSF of G_1 and call it F.
3. Use the algorithm of Theorem 19.4 to find all F-heavy edges of G' and let G_2 be the graph obtained from G' after removing them.
4. **Recursively** find the MSF of G_2, uncontract the vertices of the MSF and G' and add Boruvka's algorithm edges back to it, and return as the answer.

[a] This is a simple preprocessing step just to reduce the number of vertices slightly.

Proof of Correctness. The correctness of this algorithm is actually quite easy to prove. The first step is correct due to the correctness of Boruvka's algorithm established earlier. Regardless of the choice of G_1 and the resulting MSF F, we have by Observation 19.3 that none of the edges removed from G' to obtain G_2 can be part of the MSF of G. Thus, finding the MSF of G_2 is the same as the MSF of G to begin with (after uncontracting the vertices of G' and adding back Boruvka's algorithm edges), and thus the algorithm returns the correct answer.

Runtime Analysis. The key to the analysis of the algorithm is to show that the graph G_2 actually has few edges. In other words, after picking the MSF F on (almost) half the edges, the set of F-heavy edges more or less contains all but $O(n)$ edges of the graph. In the following lemma, recall that edges of F themselves are also considered F-light edges.

Lemma 19.6. *The expected number of F-light edges in Algorithm 19.5 is at most $2 \cdot (n' - 1)$ where n' is the number of vertices in G'.*

Proof. Notice that even though we are computing MSF of G_1 using a recursive call to the KKT algorithm, given that MSF is unique (recall Assumption 18.2 and Proposition 18.3), for the purpose of the analysis, we can assume F is instead computed using Kruskal's algorithm. This is because the distribution of F is identical in both cases.

Now, let us examine how Kruskal's algorithm works. Suppose we sort all edges of G' (and not only G_1) in increasing order of weight and call them $e_1, \ldots, e_{m'}$. Consider the following process. We go over these edges one by one and call F_i the subgraph of F maintained so far when visiting the edge e_i. We check if adding e_i to F_i creates a cycle or not. If it does, then whether or not e_i is sampled in G_1 we are not going to pick this edge in the MSF F so we just ignore it. But, if it does not, it is only now that we check whether e_i even belongs to G_1 or not. This means that only now we toss the coin to decide if e_i joins G_1 or not. Notice that, despite all these changes, we actually have not changed the distribution of F in any way in this process (we can toss a coin for neglected edges and include them in G_1 if we want just to make sure the distribution of G_1 remains identical, although this does not change the distribution of F).

Finally, note that all the edges ignored in this process are certainly F-heavy because they created a cycle even with a subgraph of F and each is the heaviest-weight edge of its cycle. Thus, the number of F-light edges is at most equal to the number of edges that we did not ignore, in other words, the edges that we tossed a coin for. At the same time, whenever we toss a coin, with probability half, we add the edge to the forest F. Moreover, the forest F cannot have more than $n' - 1$ edges. So, the expected number of coin tosses we can have before collecting $n' - 1$ edges in F is $2 \cdot (n' - 1)$, proving the lemma. □

We are now ready to conclude the proof. Firstly, let $A(G, r)$ denote the runtime of the algorithm on a graph G when the string of *all* random bits we use is r (note that $A(G, r)$ is deterministically fixed after we fix the randomness). We have,

$$A(G, r) \leqslant c \cdot (m + n) + A(G_1, r) + A(G_2, r),$$

for some absolute constant $c > 0$ which is the hidden constant in $O(m + n)$ time in Boruvka's algorithm in the first step, the use of Theorem 19.4, and general bookkeeping throughout the algorithm ignoring the recursive calls. Thus, the expected runtime of the algorithm on a graph G is

$$\mathop{\mathbb{E}}_{r}[A(G, r)] \leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r}[A(G_1, r)] + \mathop{\mathbb{E}}_{r}[A(G_2, r)]. \tag{19.1}$$

Now, define $T(m, n, r)$ as the worst-case runtime of the algorithm on a graph with m edges, n vertices, and for the randomness r. We prove inductively that

$$\mathop{\mathbb{E}}_{r}[T(m, n, r)] \leqslant 2c \cdot (m + n).$$

For any graph H, let $m(H)$ and $n(H)$, denote, respectively, the number of edges and vertices in H. Also, let r_1 be the randomness used outside the recursive calls and r_2 be the randomness of the recursive calls. Given Eq (19.1), we have,

$$\begin{aligned}
\mathop{\mathbb{E}}_{r}[T(m, n, r)] &\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r}[T(m(G_1), n(G_1), r)] + \mathop{\mathbb{E}}_{r}[T(m(G_2), n(G_2), r)] \\
&\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_1), n(G_1), r_2)]\right] + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_2), n(G_2), r_2)]\right] \\
&\quad \text{(the recursive calls themselves only depend on } r_2 \text{ (after fixing their input based on } r_1)) \\
&\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r_1}\left[2c \cdot (m(G_1) + n/8)\right] + \mathop{\mathbb{E}}_{r_1}\left[2c \cdot (m(G_2) + n/8)\right] \\
&\quad \text{(by the induction hypothesis and as Line (1) of Algorithm 19.5 reduces vertices by a factor of 8)} \\
&\leqslant c \cdot (m + n) + 2c \cdot (m/2 + n/8) + 2c \cdot (2n/8 + n/8) \\
&\quad \text{(as } \mathbb{E}[m(G_1)] = m'/2 \leqslant m/2 \text{ trivially and } \mathbb{E}[m(G_2)] \leqslant 2n' \leqslant 2n/8 \text{ by Lemma 19.6)} \\
&= 2c \cdot m + c \cdot (n + n/4 + 3n/4) = 2c \cdot (m + n),
\end{aligned}$$

proving the induction step. Thus, the runtime of the algorithm is $O(m + n)$ in expectation.

This concludes the proof of Theorem 19.1 and our study of MST algorithms in this book.

Remark. The *KKT algorithm* provided the first linear time algorithm for MSTs but at the "cost" of randomization. Hence, the search for a deterministic algorithm for this problem still continues and to date we do not know such an algorithm. The current best deterministic algorithm is due to [Cha00] with runtime $O(m \cdot \alpha(n))$ where $\alpha(n)$ is a certain *Inverse Ackermann* function (this is an extremely slowly growing function and for any reasonable number—say, number of atoms in the universe—is bounded by 5; however, it is still not constant). There is also the algorithm of [PR02] that is *provably optimal (in a very strong sense)* but its runtime is not known.

A longstanding open question at this point is to obtain a deterministic algorithm for MSTs that also runs in $O(m)$ time.

19.2 Detour: Optimal Algorithms with Unknown Runtime?

We mentioned that [PR02] provided an algorithm for MSTs whose runtime is provably asymptotically optimal, without us even knowing its runtime. But how can we have such an algorithm to begin with? We show a simple solution to this problem from the computational complexity literature [Jon97]. We emphasize that this algorithm is much weaker than the result of [PR02] both quantitatively and qualitatively—we will get back to this at the end of this section.

Let P be *any* problem in mind. Suppose we have an algorithm V that given an input x and a (supposed) solution y, can *verify* if y is indeed a solution to x for the problem P. For an input of length n, let $\text{Ver}(n)$ denote the worst-case runtime of this algorithm (which is known to us). Moreover, let $\text{Opt}(n)$ denote the runtime of the *optimal* algorithm for P on n-length inputs (which is unknown to us). Now, consider this algorithm A on input x:

Algorithm 19.7.

1. List *all* computer programs (or Turing Machines) in some order $C_1, C_2, \ldots$.
2. Run x on each of these programs in the following order: for every two steps of running x on C_i, run one step on C_{i+1}.
3. If a program C_i terminates, run the verification algorithm V to check its solution and terminate A if this is a valid solution; however, we "wait" for C_i to accumulate at least $\text{Ver}(n)$ steps (in Line (2)) before we run the verification algorithm (by letting the program have "idle" steps after it is finished).

Let C_o be the optimal program for the problem P. We thus know that A terminates for sure after it has run C_o for at most $\text{Opt}(n)$ steps. During each of these steps, C_{o-1} has run two steps, C_{o-2} has run four steps, all the way to C_1 that has run 2^{o-1} steps. Thus, the entire time spent on the programs $C_1, \ldots, C_{o-1}$ is at most $2^o \cdot \text{Opt}(n)$. The programs $C_{o+1}, \ldots,$ also run $\text{Opt}(n)/2$ steps, $\text{Opt}(n)/4$ steps, and so on, hence the total time spent over all programs is $O(2^o \cdot \text{Opt}(n))$ time.

In addition, at most $o + \log(\text{Opt}(n))$ programs have run any steps and thus the total number of times we could have run the verification algorithm is $O(o + \log(\text{Opt}(n)))$. However, we also forced the algorithm to only run the verification on programs that have already spent $\text{Ver}(n)$ steps themselves. If $\text{Opt}(n) \leqslant \text{Ver}(n)$ (which seems unlikely in general but not impossible especially if the answer is not unique), then it means that C_o will be the last program that runs the verification algorithm also and thus the total runtime of verification steps also, by the same argument as above, is $O(2^o \cdot \text{Ver}(n))$. If $\text{Opt}(n) \geqslant \text{Ver}(n)$, let v be such that

$$2^{v-1} \cdot \text{Ver}(n) \leqslant \text{Opt}(n) \leqslant 2^v \cdot \text{Ver}(n).$$

Then, only programs $C_{o+1}, \ldots, C_{o+v}$ will be running the verification algorithm (after C_o) and thus their total verification time is $O(v \cdot \text{Ver}(n)) = O(v \cdot \text{Opt}(n)/2^v) = O(\text{Opt}(n))$.

All in all, the algorithm A takes $O(2^o \cdot (\text{Opt}(n) + \text{Ver}(n)))$ time. But notice that no matter how gigantic o can be, it is still only a *constant* with respect to the input size! Thus, the runtime of algorithm A is $O(\text{Opt}(n) + \text{Ver}(n))$, which is *asymptotically optimal* (modulo the extra verification time, which as we said earlier, is rarely more than the optimal algorithm time).

Let us again emphasize that the result of [PR02] is much stronger than the above generic algorithm and does not suffer from astronomical constants; in fact, the runtime of their algorithm is proportional to the optimal number of comparisons, with a reasonable hidden constant, and not only optimal runtime.

Remark. It is worth examining Algorithm 19.7 a bit more: to some extent, it says that we already know how to design an algorithm for *every problem* with *asymptotically* optimal runtime. Of course, in reality, this algorithm is never going to work for solving almost any problem given the astronomical hidden constants that it creates[a]. So, this "algorithm" points to an inherent flaw of *asymptotic analysis* and is a good reminder to not lose sight when designing algorithms by focusing *only* on *asymptotics* of the algorithms, without other constraints such as simplicity, "elegance", and most importantly, a deeper understanding of the underlying problem—if we only care about asymptotic optimality, we already know how to achieve that for almost any problem (as long as we can design the verification algorithm).

[a]Suppose, very generously, that the optimal algorithm for a problem needs only 100 bits to write down. This means its program is going to appear roughly in a position 2^{100} in the list of all programs. This in turn means that the hidden constant in the above approach is something like $2^{2^{100}}$. Compare this number with the (crude) estimate of 10^{100} on the number of atoms in the universe to see how astronomical the hidden constants of O-notation are here.

Exercises

Exercise 19.1. Prove the high probability bound in Theorem 19.1.

Exercise 19.2 (♦). Recall Theorem 19.4. Design a deterministic algorithm for solving the MST verification problem in this theorem in $O(m \log \log n)$ time (or even faster). This is (much) easier than existing proofs of Theorem 19.4.