CS 466/666: Algorithm Design and Analysis **University of Waterloo: Fall 2024**

Lecture 7

September 26, 2024

Instructor: Sepehr Assadi

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

Topics of this Lecture

1 Karger-Klein-Tarjan Algorithm for MST

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

In this lecture, we will go over an algorithm for this problem due to Karger, Klein, and Tarjan [KKT95], which runs in $O(m)$ time but it is randomized, i.e., its $O(m)$ runtime is in expectation.

Theorem 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.*

In this lecture, we only prove the runtime of the algorithm in expectation (although extending it to a with high probability bound is quite simple also). Before moving on, we recall the following two key rules in the design of MST algorithms:

Cut Rule: *In any graph $G = (V, E)$, the minimum weight edge e in* any *cut S* always *belongs to the MST of G. This rule allows us to determine which edges to include in the MST.*

Cycle Rule: *In any graph $G = (V, E)$, the maximum weight edge e in* any *cycle C* never *belongs to the MST of G. This rule allows us to determine which edges to exclude from the MST.*

The general idea behind the Karger-Klein-Tarjan algorithm (henceforth, *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.

1.1 Preliminaries

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

Definition 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.

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

Observation 3. *For any graph G, any forest F that is a subgraph of G, and any F-heavy edge e, the MST of $G - e$ is the same as the MST of G.*

How do we use Observation 3 in the algorithm? Suppose first that F is the MST of G; then *every* edge $e \in G \setminus F$ 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 3 tells us that we can still neglect *every* F-heavy edges 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 *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* each 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 the we can find *all* F-heavy edges in linear time as well!

Theorem 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) with the goal of finding simpler algorithms, but it seems we still have not reached this goal.

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 not-necessarily connected graph, we use the term *Minimum Spanning Forest (MSF)* throughout which refers to a collection of MSTs on each connected components of the graph.

Algorithm 1 (**Karger-Klein-Tarjan Algorithm**).

(*i*) Run 3 rounds of the Boruvka's algorithm and let G' be the contracted graph obtained from G. [a]

(*ii*) 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.

(*iii*) Use the algorithm of Theorem 4 to find all F-heavy edges of G' and let G_2 be the graph obtained from G' after removing them.

(*iv*) **Recursively** find the MSF of G_2 and return it as the answer.

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

Proof of Correctness. The correctness of this algorithm is actually quite easy to proof. 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 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, 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.

Lemma 5. *The expected number of F-light edges in Algorithm 1 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 our assumption on distinct weights from the last lecture), 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 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 belongs to G_1 even 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 seeming changes, we actually have not changed the distribution of F in anyway 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 in any way).

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 are the heaviest weight edge of that cycle. Thus, the number of F-light edges is at most equal to the number of edges that we did not ignored, 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 *all* the random bits we use is r (note that $A(G, r)$ is deterministically fixed after we fixed 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 needed for Boruvka's algorithm in the first step, the use of Theorem 4, and general bookkeeping throughout the algorithm ignoring the recursive calls. Thus, the expected runtime of the algorithm on a graph G is

$$\mathop{\mathbb{E}}_{r}[A(G, r)] \leqslant c \cdot (m+n) + \mathop{\mathbb{E}}_{r}[A(G_1, r)] + \mathop{\mathbb{E}}_{r}[A(G_2, r)]. \tag{1}$$

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 the number of edges and vertices in H, respectively. Also, let r_1 be the randomness used out of the recursive calls and r_2 be the randomness of the recursive calls. Given Eq (1), we have,

$$\mathop{\mathbb{E}}_{r}[T(m, n, r)] \leqslant c \cdot (m+n) + \mathop{\mathbb{E}}_{r}[T(m(G_1), n(G_1), r)] + \mathop{\mathbb{E}}_{r}[T(m(G_2), n(G_2), r)]$$

$$\leqslant c \cdot (m+n) + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_1), n(G_1), r_2)]\right] + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_2), n(G_2), r_2)]\right]$$

(the recursive calls themselves only depend on r_2 (after fixing their input based on r_1))

$$\leqslant c \cdot (m+n) + \mathop{\mathbb{E}}_{r_1}[2c \cdot (m(G_1) + n/8)] + \mathop{\mathbb{E}}_{r_1}[2c \cdot (m(G_2) + n/8)]$$

(by induction hypothesis and as 3 rounds of Boruvka's algorithm reduces vertices by a factor of 8)

$$\leqslant c \cdot (m+n) + 2c \cdot (m/2 + n/8) + 2c \cdot (2n/8 + n/8)$$

(as $\mathbb{E}[m(G_1)] = m'/2 \leqslant m/2$ trivially and $\mathbb{E}[m(G_2)] \leqslant 2n' \leqslant 2n/8$ by Lemma 5)

$$= 2c \cdot m + c \cdot (n + n/4 + 3n/4) = 2c \cdot (m+n),$$

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

This concludes the proof of Theorem 1 and our study of MST algorithms in this course.

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 Chazelle [Cha00] with runtime $O(m \cdot \alpha(n))$ where $\alpha(n)$ is a certain *Inverse Ackerman* function (this is an extremely slowly growing algorithm and for any reasonable number—say, number of atoms in the universe—is bounded by 5; however, it is not constant still). There is also the algorithm of Pettie and Ramachandran [PR02] that is *provably optimal (in a very strong sense)* but its runtime is not known. .

A **longstanding open question** in the area of graph algorithms is to obtain a deterministic algorithm for MSTs that also runs in $O(m)$ time.

2 A Detour: Optimal Algorithms with Unknown Runtime (Optional Topic)

How can we have an algorithm that we can provably is optimal without even knowing its runtime (e.g., like the one in [PR02] for MSTs)? Here, we show a simple solution to this problem from the computational complexity literature due to Jones [Jon97].

Let P be *any* problem in mind. Suppose we have an algorithm 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 $Ver(n)$ denote the worst-case runtime of this algorithm. Moreover, let $Opt(n)$ 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:

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} (so, a diagonalization-type approach).
3. If a program C_i terminates, run the verification algorithm to check its solution and terminate A if this is a valid solution; however, we "wait" for C_i to accumulate at least $Ver(n)$ steps (in Line 2 above) 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 $Opt(n)$ 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 steps. Thus, the entire time spent on the programs $C_1, \ldots, C_{o-1}$ is also at most $2^o \cdot Opt(n)$. The programs $C_{o+1}, \ldots,$ also run $Opt(n)/2$ steps, $Opt(n)/4$ steps, and so on, hence the total time spent over all programs is $O(2^o \cdot Opt(n))$.

In addition, at most $o + \log Opt(n)$ programs have run any steps and thus the total number of times we could have run the verification algorithm is $O(o+\log Opt(n))$ times. However, we also forced the algorithm to only run the verification on programs that have already spent $Ver(n)$ steps themselves. If $Opt(n) \leqslant Ver(n)$ (which is quite unlikely in general but not impossible 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 $O(2^o \cdot Ver(n))$. If $Opt(n) \geqslant Ver(n)$, let v be such that $Opt(n) \approx Ver(n)/2^v$. Then, only programs $C_{o+1}, \ldots, C_v$ will be running the verification algorithm (after C_o) and thus their total verification time is $O(v \cdot Ver(n)) = O(v \cdot Opt(n)/2^v) = O(Opt(n))$.

All in all, the algorithm A takes $O(2^o \cdot (Opt(n) + Ver(n)))$ time. But now 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 $O(Opt(n) + Ver(n))$, which is *asymptotically optimal* (modulo the extra verification time, which as we said earlier, is very rarely more than the optimal algorithm time anyway).

It is worth examining this result a bit more: to some extent, it says that we already know how to design an algorithm for *every possible problem* that is *asymptotically* as good as it gets. Of course, in reality, this algorithm is never going to work for solving almost any problem given the astronomical hidden constants that it creates[1]. So, in my humble opinion, this "algorithm", more than anything, points to an inherent flaw of *asymptotic analysis* and is a good reminder to not loose 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 any problem!

Before concluding this detour, we note that the result of [PR02] is quite stronger than the above generic algorithm and does not suffer from astronomical constants; in fact, the runtime of their algorithm is proportional to *optimal* number of comparisons, with a reasonable hidden constant, and not only optimal runtime.

References

[Cha00] Bernard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. 4

[DRT92] Brandon Dixon, Monika Rauch, and Robert Endre Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992. 2

[Jon97] Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997. 4

[Kin97] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997. 2

[KKT95] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. 1

[Kom85] János Komlós. Linear verification for spanning trees. *Comb.*, 5(1):57–65, 1985. 2

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. 4, 5

[1]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 is...