CS 598CSC: Topics in Graph Algorithms	Lecture date: 08/27/2024
Instructor: Chandra Chekuri	Scribe: CC

1 MST

Finding the minimum cost spanning tree (MST) in a connected graph is a basic algorithmic problem that has been long-studied. Standard algorithm that are covered in most undergraduate courses are Kruskal's algorithm, Jarnik-Prim's (JDP) algorithm (which is typically attributed usually to Prim but first described by Jarnik), and (sometimes) Boruvka's algorithm. There are many different algorithms for MST and their correctness relies on two simple rules (structural properties). We will assume that the input is a graph $G = (V, E)$ with edge costs. Note that the edge costs need not be positive but we can make them positive by adding a large number without affecting correctness. Why?

Lemma 1 (Cut rule)**.** *Let $G = (V, E)$ be a connected graph with edge costs $c : E \to \mathbb{R}$. Suppose e is a minimum cost edge in a cut $\delta(S)$ for some $S \subseteq V$. Then there is some MST T of G that contains e. In particular, if e is the unique cheapest cost edge in the cut then e is* every *MST of G.*

We call an edge *safe/light* if there is a cut such that e is the cheapest cost edge crossing the cut.

Lemma 2 (Cycle rule)**.** *Let $G = (V, E)$ be a connected graph with edge costs $c : E \to \mathbb{R}$. Suppose e is a highest cost edge in a cycle C. Then there is some MST T of G that does not contain e. In particular, if e is the unique highest cost edge in C then e cannot be in any MST of G.*

We call an edge *unsafe/heavy* if there is a cycle C such e is the heaviest cost edge in C.

Corollary 3. *Suppose the edge costs are unique and G is connected. Then the MST is unique and consists of the set of all safe/light edges.*

It is relatively easy to ensure that edge costs are unique so we will assume this property. We quickly recall the three standard algorithms, the data structures they use, and the run-times that they yield.

Kruskal's algorithm sorts the edges in increasing cost order and greedily inserts edges in this order while maintaining a maximal forest F at each step. Soring takes $O(m \log n)$ time. When considering the i'th edge e_i, the algorithm needs to decide if $F + e_i$ is a forest or whether adding e creates a cycle. The standard solution for this is to use a union-find data structure. Union-find data structure with path compression yields a total run time, after sorting, of $O(m\alpha(m, n))$ where $\alpha(m, n)$ is inverse Ackerman function which is extremely slowly growing. Thus the bottleneck is sorting and the run-time is $O(m \log n)$.

JP algorithm grows a tree starting at some arbitrary root vertex r while maintaining a tree T rooted at r. In each iteration it adds the cheapest edge leaving T until T becomes spanning. Thus the algorithm takes $n - 1$ iterations. To find the cheapest edge leaving T one typically uses a priority queue data structure where we maintain vertices not yet in the tree with a key for v equal to the cost of the cheapest edge from v to the current tree. When a new vertex u is added to T the algorithm scans the edges in $\delta(u)$ to update the keys of neighbors of v. Thus, one sees that there

are a total of $O(m)$ decrease-key operations, $O(n)$ delete-min operations and initially we set up an empty queue. Standard priority queues implement decrease-key and delete-min in $O(\log n)$ time each so the total time is $O(m \log n)$. However, Fibonacci heaps and related data structures show that one can implement decrease-key in amortized $O(1)$ time which reduces the total run time to $O(m + n \log n)$. Thus the algorithm runs in linear-time for moderately dense graphs!

Boruvka's algorithm seems to be the first MST algorithm. It has very nice properties and essentially uses no data structures. The algorithm works in phases. We describe it recursively to simplify the description. In the first phase the algorithm finds, for each vertex v the cheapest edge in $\delta(v)$. By the cut rule this edge is in every MST. Note that an edge $e = uv$ may be the cheapest edge for both u and v. The algorithm collects all these edges, say F, and adds them to the tree. One can easily implement this in $O(m)$ time by a linear scan of the adjacency lists. It then shrinks the connected components induced by F and recurses on the resulting graph $H = (V', E')$. Computing H can be done in $O(m)$ time. The main observation is that $|V'| \leq |V|/2$ since each vertex v is in a connected component of size at least 2 since we add an edge leaving v to F. Thus the algorithm terminates in $O(\log n)$ phases for a total of $O(m \log n)$ time. Note that this algorithm is easy to parallelize unlike the other two algorithms.

2 Faster Algorithms

A natural question is whether there is a linear-time MST algorithm, that is an algorithm that runs in time $O(m)$. Very early on Yao [cY75], in 1975, obtained an algorithm that ran in $O(m \log \log n)$ time. Note that this is before many developments in data structures — it was inspired by and partly based on the linear-time Selection algorithm that was discovered in 1974. Fredman and Tarjan [FT87] obtained an algorithm, via Fiboannci heaps, that runs in $O(m\beta(m, n))$-time where $\beta(m, n)$ is the minimum value of i such that $\log^{(i} n \leq m/n$ where $\log^{(i)} n$ is the logarithmic function iterated i times. Since $m \leq n^2$, $\beta(m, n) \leq \log^* n$. This was further improved by Gabow et al [GGST86] to $O(m \log \log^* n)$. Karger, Klein and Tarjan [KKT95] obtained a linear time randomized algorithm that will be the main topic of this lecture. Chazelle's algorithm [Cha00] that runs in $O(m\alpha(m, n))$ where $\alpha(m, n)$ is the inverse Ackerman function is the fastest known deterministic algorithm. Pettie and Ramachandran [PR02] gave an optimal deterministic algorithm in the comparison model without knowing what its actual running time is!

A potentially easier question is the following. Given a graph G and a tree T, is T an MST of G? This is called the MST verification problem. Clearly, one can always use an MST algorithm to solve the verification problem, but not necessarily the other way around. Interestingly there is indeed a linear-time MST verification algorithm. It is based on several non-trivial ideas and data structures and was first developed in the RAM model by Dixon, Rauch, and Tarjan [DRT92] based on insights from Komlos [Kom85]. King [Kin97] simplified it. The RAM model allows bit-wise operations on $O(\log n)$ bit words in $O(1)$ time.

Theorem 4. *There is a linear-time MST verification algorithm in the RAM model.*

In fact the algorithm is based on a more general result that we will need.

Theorem 5. *Given a graph $G = (V, E)$ with edge costs and a spanning tree $T = (V, F)$, there is an $O(m)$-time algorithm that outputs all the F-heavy edges of G.*

The original algorithm is quite complicated and it has been simplified over the years. See lecture notes of Gupta and Assadi for accessible explanations (also the MST surveys by Eisner [Eis97] and Mares [Mar08]).

Fredman-Tarjan algorithm: Here we briefly describe Fredman and Tarjan's algorithm via Fibonacci heaps. See [FT87, Mar08] for a precise description. The algorithm is reasonably simple to describe and analyze modulo a few implementation details that we will gloss over for sake of brevity.

First, we develop a simple $O(m \log\log n)$ time algorithm by combining Boruvka and JP algorithms. Recall that JP algorithm takes $O(m + n \log n)$ time via Fibonacci heaps, and this is already very good if m is large. The bottleneck is when $m = o(n \log n)$. Boruvka's algorithm starts with a graph on n nodes and after i phases reduces the number of nodes to $\leq n/2^i$; each phase takes $O(m)$ time. Suppose we run Boruvka's algorithm for k phases and then run JP algorithm once the number of nodes is reduced. We can see that the total run time is $O(mk)$ for the k phases of Boruvka, and $O(m + \frac{n}{2^k} \log \frac{n}{2^k})$ for the JP algorithm on the reduced graph. Thus, if we choose $k = \log\log n$ we obtain a total run-time of $O(m \log\log n)$.

Fredman and Tarjan obtained a more sophisticated scheme based on the JP algorithm but the basic idea is to reducing the number of vertices. The algorithm runs in phases. We describe the first phase. Let us choose an integer parameter t (with $1 < t \leq n$) that we will set later. Pick an arbitrary root r_1 and grow a tree T_1 following the JP algorithm with a Fibonacci heap. We stop the tree growth when the heap size exceeds t for the first time or if we run out of vertices. All the vertices in the tree are marked as visited. Now pick an arbitrary unmarked vertex as root $r_2 \in V - T$ and grow a tree T_2. We stop growing T_2 if touches T_1, in which case it merges with it, or if the heap size exceeds t or if we run out of vertices. Note that the heap, while growing T_2, may contain previously marked vertices. It is only when the algorithm finds one of the marked vertices as the cheapest neighbor of the current tree that we merge the trees and stop. The algorithm proceeds in this fashion by picking new roots and growing them until all nodes are marked. We see that the algorithm correctly adds a set of MST edges F.

We prove that there are at most $2m/t$ vertices after shrinking the connected components induced by F. Let $C_1, C_2, \ldots, C_h$ be the connected components of F. We claim that for each C_i, $\sum_{v \in C_i} \deg(v) \geq t$. Suppose this is true then

$$2m = \sum_u \deg(v) = \sum_{i=1}^{h} \sum_{u \in C_i} \deg(v) \geq ht$$

which implies that $h \leq 2m/t$. To see the claim. Consider the growth of a tree T_j by the algorithm. If we stop T_j because heap size exceeds t then each of the vertex in the heap is a witness to a unique edge incident to T_j which proves the desired claim. If T_j merged with a previous tree then the property holds because the previous tree already had the property and adding vertices can only increase the total degree of the component. The only reason the property may not hold is if the algorithm terminates a tree because all vertices are already included in it but then the phase finishes the algorithm.

What is the running time of the phase? We see that the total time to scan edges and insert vertices into heaps and do decrease keys is $O(m)$ since an edge is only visited twice, once from each end point. Since each heap is not allowed to grow to more than size t, the total time for all

the delete-min operations $O(n \log t)$. We are utilizing the fact that the initialization of each data structure is easy because it starts as an empty one. Thus, we have obtained a new algorithm that, in a single phase, takes $O(m + n \log t)$ time and reduces the number of vertices to $2m/t$. This can be seen as a parameterized version of Boruvka. How should we choose t? If we want linear time in the first phase we want $n \log t$ to be no more than $O(m)$ so it turns out that it make sense to set $t = 2^{2m/n}$. This choice yields $O(m)$ time per phase.

To bound the overall time, we need to bound the number of phases. Let m_i and n_i be the number of edges and vertices at the start of phase i. So we have $m_1 = m$ and $n_1 = n$. We have $n_{i+1} \leq 2m_i/t_i$ by the claim. We set $t_i = 2^{2m/n_i}$; technically we choose $t_i = 2^{\lceil 2m/n_i \rceil}$ but we will be bit sloppy and ignore the ceilings here. Therefore, $n_{i+1} \leq 2m_i/t_i$. We see that

$$t_{i+1} = 2^{2m/n_{i+1}} \geq 2^{2m/(2m_i/t_i)} \geq 2^{t_i}.$$

Thus t_i is a power of twos with $t_1 = 2^{2m/n}$. When do we stop? The algorithm stops for sure if $t_i \geq n$ since it will grow a single tree and finish (the algorithm may stop even if this is not the case due to trees merging). Thus the algorithm needs at most $\beta(m, n)$ phases and each phase, by design, takes $O(m)$ time. Hene the total time is $O(m\beta(m, n))$.

3 Linear-time Randomized Algorithm

Karger, Klein and Tarjan developed a randomized linear-time algorithm.

Theorem 6 ([KKT95]). *There is an algorithm that computes the MST of a graph in $O(m)$ time with high probability (at least $1 - 1/poly(m)$).*

We will only prove an expected running time bound and refer the reader to the paper for details on the high probability guarantee.

3.1 Sampling Lemma

The algorithm uses the MST verification result, captured by Theorem **??**, as a black box. This is the only complicated part of the algorithm.

The algorithm is inspired by Karger's random sampling work for graphs (see his thesis [Kar95]) and the crucial lemma is the following which is independently interesting outside of its algorithmic application.

Lemma 7. *Let $G = (V, E)$ be a graph with edge costs. Suppose $E' \subseteq E$ is a random subset of E obtaining by picking each edge independently with probability $p \in (0, 1)$. Let $F \subseteq$ be a minimum cost spanning forest in the graph induced by E'. Then the expected number of F-light edges in G is at most n/p.*

Note that if we choose $p = 1/2$, it is saying that the number of F-light edges from E is at most $2n$. Thus, we can eliminate most of the edges from $E \setminus E'$ from consideration if we can efficiently compute the F-light edges (equivalently the F-heavy edges) which we know how to do via the MST verification algorithm. We will describe how this leads to the desired randomized algorithm after we prove the lemma. The proof is short and simple but subtle one if one is not used to tricks in probabilistic proofs. In particular it is based on the *principle of deferred decisions* in randomized

analysis. Our own Timothy Chan has an alternate slick proof though we will go with the longer route.

Let A be the set of F-light edges. Note that both A and F are random sets that are generated by the process of sampling E'. To analyze $\mathbf{E}[A]$ we considers Kruskal's algorithm to obtain F from E'. We will sort edges of E as $e_1, e_2, \ldots, e_m$ according to increasing cost and will generate E' on the fly.

1. $A, F, E' \leftarrow \emptyset$
2. For $i = 1$ to m do
   (a) toss a biased coin that is heads with prob p
   (b) If coin is heads then
      - $E' \leftarrow E' + e_i$
      - If $F + e_i$ is a forest then add e_i to F and to A
   (c) Else If e_i is F-light, add e_i to A
3. Output F, A

You should be convinced that the above algorithm is equivalent to the sampling process. We now consider an alternative algorithm which is a small twist.

1. $A, F \leftarrow \emptyset$
2. For $i = 1$ to m do
   (a) If e_i is F-light then
      - $A \leftarrow A + e_i$
      - toss a biased coin that is heads with prob p
      - If coin is heads then $F \leftarrow F + e_i$
3. Output F, A

Note that the second algorithm does not keep track of E' but you should convince yourself that it produces the same A, F as the previous algorithm. In particular, the sorting of edges implies that $F + e_i$ is a forest iff e_i is F light. One can adjust the algorithm to keep track of E' as well. The second algorithm makes the following clear. An edge e_i is added to A implies that it is added to F with probability p. Thus $\mathbf{E}[F] = p\,\mathbf{E}[A]$. However, $|F| \leq n - 1$ deterministically. Thus $\mathbf{E}[A] \leq (n-1)/p$ as desired.

Sometimes it is useful to peel the banana so we add more detail. Let X_i be the indicator random variable for $e_i \in A$ and similarly let Y_i be the indicator for $e_i \in F$. Let R_i denote the random string of the first i choices. When considering e_i we condition on $R_{i=1} = r$ which fixes subset of first $i-1$ edges that were sampled in E' and this also fixes the subset of those edges that are included in F after $i-1$ edges. Conditioned on r, X_i is a deterministic choice — either e_i is F-light or it is not. However, conditioned on r, Y_i is still a random variable. If $X_i = 0$ then $Y_i = 0$ but if $X_i = 1$ then $Y_i = 1$ with probability p and $Y_i = 0$ with probability $(1-p)$. Since X_i, Y_i are indicator random variables this implies that $\mathbf{E}[Y_i \mid R_{i=1} = r] = p\,\mathbf{E}[X_i \mid R_{i-1} = r]$. This implies that $\mathbf{E}[X_i] = p\,\mathbf{E}[Y_i]$. By linearity of expectation, we have $\mathbf{E}[A] = \sum_{i=1}^{m} \mathbf{E}[X_i]$ and $\mathbf{E}[F] = \sum_i \mathbf{E}[Y_i]$.

This finishes the simple proof.

3.2 Algorithm

The sampling lemma naturally suggests a recursive divide and conquer algorithm, reminiscent of the linear-time deterministic algorithm for Selection. However, there is a small extra step that needs to be done to make it work out.

First, the sampling lemma and the natural recursion that it implies means that we need to work with a slightly more general problem than MST, namely the problem of computing the minimum cost spanning forest (MSF). MSF and MST are very closely related and one is reducible to the other in linear time. The cut and cycle properties are easy to generalize to MSF and we won't do it formally.

A natural recursive randomized algorithm based on the sampling lemma for MSF is the following.

1. If $|V| < n_0$ for some constant n_0 use a standard deterministic algorithm and output its result.
2. Sample each edge independently with probability $1/2$ to obtain $E_1 \subseteq E$ and let $G_1 = (V, E_1)$.
3. Recursively compute MSF F_1 in G_1.
4. Use linear time MST verification algorithm to compute all the F_1-light edges in G. Let E_2 be the set of F_1-light edges
5. Recursively compute MSF F_2 in graph $G_2 = (V, E_2)$.
6. Output F_2.

The correctness of the algorithm should be clear from the cut and cycle properties. The only issue is the running time. The expected number of edges in G_1 is $m/2$. The expected number of edges in G_2, via the sampling lemma, is at most $2n$. The algorithm does $O(m+n)$ work outside of these two recursive calls. Let $T(m, n)$ be the expected running time of the algorithm. Informally, one can see that we have the following recurrence:

$$T(m, n) \leq T(n, m/2) + T(n, 2n) + O(m + n).$$

Does this lead to linear running time? If we take the problem size to be $n + m$ then the algorithm is generating two problems of expected size $(n + m/2)$ and $(n + 2n)$ for a total expected size of $4n + m/2$. If $m > 10n$ say, then the total problem size is shrinking by a constant factor. However when m is closer to n, we are not necessarily reducing the overall problem size. Here is where one can use a trick. We run Boruvka's algorithm for a few iterations as a preprocessing step so that the number of vertices goes down.

KKT-MST-Algorithm(G):

1. If $|V| < n_0$ for some sufficiently large constant, use standard algorithms. Assume no connected component of G is small.
2. Run Boruvka's algorithm for two phases to obtain graph $G_1 = (V_1, E_1)$ with $|V_1| \leq |V|/4$. Let F_1 be the set of edges added.

3. Sample each edge in G_1 independently with probability $1/2$ to obtain $E_2 \subseteq E_1$ and let $G_2 = (V, E_2)$.
4. Recursively compute MSF F_2 in G_2.
5. Use linear time MST verification algorithm to compute all the F_2-light edges in G. Let E_3 be the set of F_2-light edges
6. Recursively compute MSF F_3 in graph $G_2 = (V, E_3)$.
7. Output $F_1 \cup F_3$.

The correctness is easy to see as before. What about the run time? Recall that Boruvka's algorithm takes $O(m)$ time for each phase so the total time for step 2 is $O(m)$. We can now write a recurrence for $T(m, n)$ which is

$$T(m, n) \leq T(n/4, m/2) + T(n/4, n/2) + O(m + n)$$

With this recurrence we see that the sum of the expected problem sizes of the two problems is $n + m/2$ which is a constant factor smaller than $n + m$ (we can assume $m \geq n - 1$ since we eliminate small components including singletons). Thus, by a simple inductive proof, one can show that $T(n, m) = O(n + m)$. See notes of Assadi for a rigorous derivation of the recurrence and time bounds. A more refined analysis of the sampling lemma can be used to show that the running time is linear with high probability.

Odds and Ends

Many properties of forests and spanning trees can be understood in the more general context of matroids. In many cases this perspective is insightful and also useful. The sampling lemma applies in this more general context and has various applications. See Karger's work on this [Kar98, Kar95].

Obtaining a deterministic $O(m)$ time algorithm is a major open problem. Obtaining a simpler linear-time MST verification algorithm, even randomized, is also a very interesting open problem.

See [Pet14] for algorithms on a related problem called MST sensitivity analysis where the goal is to find for each edge the minimum amount of perturbation in its cost to make it enter/leave the MST.

References

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

[cY75] Andrew Chi chih Yao. An $O(|E| \log \log |V|)$ algorithm for finding minimum spanning trees. *Information Processing Letters*, 4(1):21–23, 1975.

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

[Eis97] Jason Eisner. State-of-the-art algorithms for minimum spanning trees. Unpublished survey/report, `https://www.cs.jhu.edu/~jason/papers/eisner.mst-tutorial.pdf`, 1997.

[FT87] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GGST86] Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

[Kar95] David Ron Karger. *Random sampling in graph optimization problems.* PhD thesis, stanford university, 1995.

[Kar98] David R Karger. Random sampling and greedy sparsification for matroid optimization problems. *Mathematical Programming*, 82(1):41–81, 1998.

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

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

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

[Mar08] Martin Mareš. The saga of minimum spanning trees. *Computer Science Review*, 2(3):165–221, 2008.

[Pet14] Seth Pettie. Sensitivity analysis of minimum spanning trees in sub-inverse-ackermann time. *arXiv preprint arXiv:1407.1910*, 2014.

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, 49(1):16–34, 2002.