CMPUT 498/501: Advanced Algorithms **Fall 2025**

Lecture 5 (Sep 18, 2025): Linear-Time MST and Arborescences

Lecturer: Mohammad R. Salavatipour *Scribe: Churong Yu*

5.1 Linear-Time Minimum Spanning Tree (MST)

In this lecture we will discuss the algorithm by Karger-Klein-Tarjan to compute a minimum spanning tree (MST) in *expected linear time*. First we need to define few terminologies.

5.1.1 Heavy and Light Edges

First we will introduce the concept of *heavy* and *light* edges in a graph. These notions also lead to a linear-time algorithm for *verifying* MST.

Definition 1 *Let F be a forest in a weighted graph $G = (V, E)$. We can assume the weights are distinct. An edge $e \in E$ is:*

- ***F-heavy** if adding e to F creates a cycle and e is the heaviest edge in that cycle.*
- ***F-light** otherwise.*

Observation 1 *If T is an MST:*

1. *Every edge $e \in T$ is light.*
2. *Every edge $e \notin T$ is heavy (with respect to T).*

Proof. Take T an MST. If $e \in T$ were heavy, then there would exist a strictly lighter edge on its fundamental cycle, contradicting minimality. Conversely, if $e \notin T$ were not heavy, then we could add e and remove a heavier edge on the cycle to obtain a cheaper spanning tree, again a contradiction. ■

Observation 2 *For any forest F, the set of F-light edges always contains an MST of G. Equivalently, no F-heavy edge belongs to any MST.*

Proof. Suppose some MST T contains an F-heavy edge e. Then e is the unique heaviest edge on the cycle formed with F, so T could be improved by replacing e with a lighter edge. Thus e cannot appear in any MST. ■

These properties immediately give rise to the *verification problem*: given a spanning tree T of G, can we check if T is in fact an MST?

- For each non-tree edge $e \in E \setminus T$, check whether it is T-heavy.
- If all such edges are T-heavy, then T is an MST.

Why is this interesting? This illustrates a central theme in complexity theory: sometimes it is much easier to *verify* a proposed solution than to *compute* one from scratch. In fact, verifying an MST can be done in linear time $O(m)$ and there are several algorithms for this. This distinction is reminiscent of the famous P vs NP question: is verification always easier than finding the solution?

5.1.2 Karger–Klein–Tarjan Algorithm

We now describe the randomized algorithm of Karger, Klein, and Tarjan (1995) which computes an MST in *expected linear time*. The following theorem is the main tool to achieve linear running time.

Theorem 1 *Given a forest $F \subseteq G$, there is an algorithm that outputs all F-heavy (or F-light) edges in $O(m+n)$ time.*

Theorem 1 is crucial for fast MST verification: by identifying F-heavy edges efficiently, we can verify whether a given spanning tree is an MST in linear time. Here, we will not prove this theorem and assume this theorem as a black box. This theorem follows from the work of Kolmos'85, Dixon/Rauch/Tarjan'92, and King'97.

The idea of KKT algorithm is to choose (randomly) half of the edges and then find a minimum spanning forest F over them. Then find F-heavy edges and discard them; now recurse on the remaining graph.

Algorithm 1 Karger-Klein-Tarjan (KKT) MST Algorithm

1: Run 3 rounds of Borůvka's algorithm on G. Contract the edges to form $G' = (V', E')$ with $|V'| = n' \leq n/8$ and $|E'| = m' \leq m$.
2: Let E_1 be a random sample of E' where each edge is included independently with probability $1/2$. Let $F_1 = \text{KKT}(G_1 = (V', E_1))$ (i.e., compute a MSF of G_1).
3: Let E_2 be all F_1-light edges of E' using Theorem 1.
4: Let $F_2 = \text{KKT}(G_2 = (V', E_2))$ (i.e., delete the F_1-heavy edges from G' and compute the MSF of the remainder).
5: Return F_2 together with the edges found in Step 1.

Proof of Correctness.

- By the proof of Borůvka's algorithm, the edges chosen in Step 1 (cheapest outgoing edges) must belong to every MST.
- By Observation 2, the F-heavy edges discarded in Step 4 cannot belong to any MST.
- Therefore the recursion is always on a smaller graph that still contains some MST of the original. By induction, the final output is an MST.

5.1.3 Runtime Analysis

We use the following two claims on the contracted graph $G' = (V', E')$ obtained after 3 rounds of Borůvka:

Claim 1 $\mathbb{E}[\,|E_1|\,] = \dfrac{m'}{2}$, *where* $m' = |E'|$.

Proof. The proof of claim 1 is trivial. Each edge of E' is sampled independently with probability $1/2$. ■

Claim 2 $\mathbb{E}[\,|E_2|\,] \le 2\,(n'-1)$, *where* $n' = |V'|$.

We show by using claim 1 and 2, KKT terminates in expected time $O(m+n)$. We will prove claim 2 later in this section.

Recurrence. Assume the total time for Steps 1 & 3 on a graph of size (m,n) is $c(m+n)$ for some constant c. Let T_G denote the expected running time of KKT(G), and define

$$T_{m,n} \;=\; \max_{G:|V|=n,|E|=m} \{T_G\}.$$

From the algorithm:

$$T_G \;\le\; c(m+n) + \mathbb{E}[T_{G_1} + T_{G_2}] \;\le\; c(m+n) + T_{m_1,n'} + T_{m_2,n'},$$

where $m_1 = \mathbb{E}[\,|E_1|\,] \le \frac{m'}{2}$, $m_2 = \mathbb{E}[\,|E_2|\,] \le 2(n'-1)$.

We prove by induction that $T_{m,n} \le 2c(m+n)$.

$$\begin{aligned}
T_G &\le c(m+n) + \mathbb{E}[\,2c(m_1+n')\,] + \mathbb{E}[\,2c(m_2+n')\,] \\
&\le c(m+n) + 2c\Big(\tfrac{m'}{2} + n'\Big) + 2c(2(n'-1)+n') \\
&\le c(m+n) + c(m'+2n') + 2c(2n'+n') \\
&= c(m+n+m'+8n') \\
&\le c(m+n+m+8\tfrac{n}{8}) \qquad (\text{since } m' \le m,\ n' \le n/8) \\
&\le 2c(m+n).
\end{aligned}$$

Thus, by induction, $T_{m,n} \le 2c(m+n)$ for all m, n.

Proof of Claim 2. We must show that the expected number of F_1-light edges is at most $2(n'-1)$.

For the sake of this proof, assume F_1 (the MSF of G_1) is obtained by Kruskal's algorithm. We also assume the process of randomly selecting edges for E_1 is *mixed* with computing F_1 as follows:

1. Sort all edges of G' as $e_1, \ldots, e_{m'}$.
2. For each e_i: if it creates a cycle, *ignore* it (then it is F_1-heavy); otherwise flip a fair coin and add it to F_1 with probability $1/2$.

Observation 3 *This mixed process generates the same distribution for F_1 as first choosing G_1 and then running Kruskal's.*

The number of F_1-light edges is bounded by the number of edges we did not ignore, i.e., the edges we tossed a coin for. Since F_1 has at most $(n'-1)$ edges and the probability of coin toss is $1/2$, the expected number of such coin tosses (i.e. F_1-light edges) is at most $2(n'-1)$.

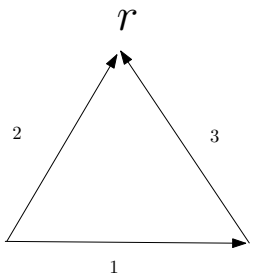

Figure 5.1: A bad example for Prim's on a directed graph.

Discussion on Randomization

Why must the sampling be random? Why not simply take, say, the first half of the edges? The answer is that the analysis depends crucially on independence: only under random sampling do we get the sparsification guarantee. A fixed deterministic half could adversarially retain all "bad" edges.

This illustrates a broader open question in algorithms and complexity theory: can all randomized algorithms be derandomized? For MST, we do not yet know a deterministic linear-time algorithm.

Open Problem. Find a deterministic linear time MST algorithm.

5.2 Directed MST (Arborescence)

So far we have studied MST in undirected graphs. We will see how a refined greedy algorithm we saw for MST will work for directed MST. Given a digraph $G = (V, A)$ where A is the set of arcs (directed edges), $w : A \to \mathbb{R}^+$, a root $r \in V$.

Definition 2 *An* r-arborescence *is a subgraph* $T = (V, A')$ *with* $A' \subseteq A$ *that is a directed tree rooted at* r *with each* $v \in V$ *having a directed path to* r*. So ignoring the directions,* T *is a spanning tree.*

How to check if G has an r-arborescence? We can run (a backward DFS) to see if each vertex has a path to the root. The question we are interested is to design an efficient algorithm to find a minimum cost arborescence. One may try to use the same greedy algorithms we have for MST for the arborescence. For example, will Prim's algorithm work? There is a fairly simple counter example that shows this will faile (see Figure 5.3). What about Kruskal's algorithm? One can come up with a counter example for this algorithm as well.

5.2.1 Chu–Liu / Edmonds / Bock Algorithm (1967)

Let us define the following: for a single vertex $v \in V$, $\delta^+(v)$ denotes the set of arcs going *out* of v, and $\delta^-(v)$ denotes the set of arcs going *into* v. More generally, for a subset $S \subseteq V$, we write $\delta^+(S)$ for arcs leaving S and $\delta^-(S)$ for arcs entering S.

We reduce weights on outgoing arcs s.t. for each v, at least one of them is 0.

$$M_G(v) := \min\{w(a) : a \in \delta^+(v)\}.$$

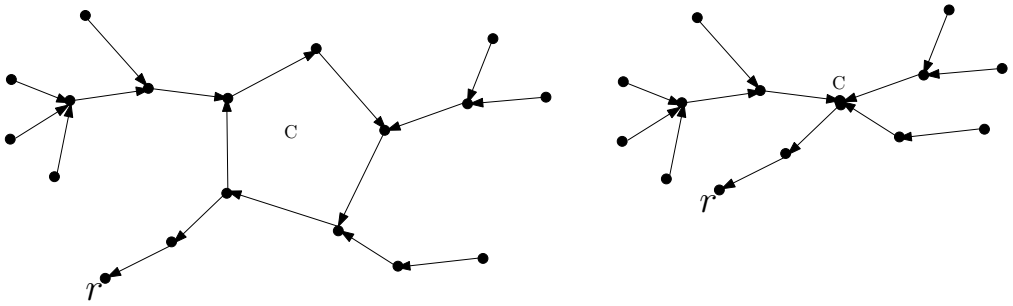

Figure 5.2: a) A component with a cycle C; b) cycle C contracted into a node.

For all $v \in V$ and outgoing arc $a \in \delta^+(v)$, set

$$w'(a) \leftarrow w(a) - M_G(v).$$

Call this new graph G'.

Lemma 1 *T is a min-cost arborescence in $G \iff T$ is a min-cost arborescence in G'.*

Proof. In each arborescence each vertex has one outgoing edge. So decreasing all outgoing edges of each node by $M_G(v)$ does not change optimal. ■

Idea of the algorithm. After the per-vertex weight reduction, every vertex has at least one 0-weight outgoing arc. Select one such arc for each $v \in V$; the selected arcs form a directed graph H with $\text{outdeg}_H(v) = 1$ for all v except possibly the root.

If H contains no directed cycle and every vertex reaches r, then H is an r-arborescence. Since all chosen arcs have weight 0 (and all weights are nonnegative), this arborescence is minimum.

Otherwise, each component of H contains exactly one directed 0-cycle C; the remaining vertices in the component form in-arborescences whose roots lie on C.

Contract every such 0-cycle C into a single supernode; if parallel arcs appear between supernodes, keep the cheapest under the reduced weights. Denote the contracted graph by $G'' = G'/C$.

Recurse on G'' to obtain a minimum arborescence T''. Finally, expand the contractions: for each contracted cycle C, add back all its 0-weight arcs and delete exactly one arc on C to break the cycle, the vertex we choose is the vertex that have 2 outgoing edges; together with T'' this yields a minimum arborescence in G' (and hence in G).

Lemma 2 $\text{opt}(G'') = \text{opt}(G')$.

Proof. First we show $\text{opt}(G') \leq \text{opt}(G'')$. Let T'' be a min arborescence for G''. Consider T' obtained from T'' by expanding the contracted node v_C back to C and removing one edge of C (this has to be chosen carefully so that all the nodes have a path to r). Since C is a 0-weight cycle, $w(T') = w(T'')$.

To show $\text{opt}(G'') \leq \text{opt}(G')$, suppose T' is a solution for G'. Identify the nodes of C into a single node. We still have a directed path from each node to r; we can remove some edges to find an arborescence T'' for G'', and cost can only go down. ■

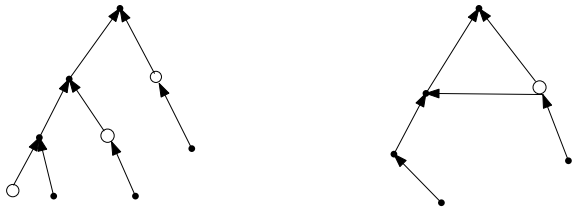

Figure 5.3: a) A tree $T' \subseteq G'$, hollow nodes are nodes of a cycle C. b) nodes of C are contracted into a single node

Algorithm 2 Minimum Arborescence algorithm

1: Obtain G' by reducing outgoing weights for each v by $M_G(v)$.
2: Take an arbitrary 0-weight outgoing edge of each v.
3: Contract 0-weight cycles to get G''..
4: Recurse on G''; let T'' be the solution.
5: Add back edges of the cycles (except one per cycle) to expand T'' to a solution T' for G'.
6: Return T'.

Time Complexity

In each round of the algorithm, the number of vertices strictly decreases (since every contraction merges at least one directed cycle). Hence there can be at most $O(n)$ iterations. The main operations in one iteration are:

1. *Weight reduction*: subtract $M_G(v)$ from all outgoing arcs of v.
2. *Contraction*: shrink every 0-weight cycle into a supernode.
3. *Lifting/expansion*: when recursion returns, expand contracted cycles back.

Each of these steps can be carried out in $O(m)$ time per iteration. Therefore, the overall running time is $O(mn)$,

Improvements. With more sophisticated data structures, one can improve the time to $O(m \log \log n)$.

Open question: Is there a linear time algorithm for this problem?

Later in the course we will also see how linear programming can be used to characterize and compute minimum arborescences.