CS 860: Modern Topics in Graph Algorithms **University of Waterloo: Winter 2024**

Lecture 1: Minimum Spanning Trees

January 15, 2023

Instructor: Sepehr Assadi

Topics of this Lecture

1 Minimum Spanning Trees

We start our course with one of the most basic and standard problems in graph algorithms: finding a **minimum spanning tree (MST)** of *weighted* graphs. You have most likely seen this course in your previous algorithms courses. We will provide a quick review in the following.

Let $G = (V, E)$ be an undirected *connected* graph with *positive* weights $w : E \to \mathbb{R}^+$ on its edges. Recall that a spanning tree of a connected subgraph G is any subgraph of G which is a tree – a tree itself is a connected subgraph which has no cycle. With a slight abuse of notation, for any tree T, we write $w(T) := \sum_{e \in T} w(e)$ to denote the sum of the weights of edges in T.

Problem 1 (**Minimum Spanning Tree (MST)**)**.** The minimum spanning tree (MST) problem is defined as follows: Given a graph $G = (V, E)$ and positive edge-weights $w : E \to \mathbb{R}^+$, find a spanning tree T of G with minimum weight $w(T)$.

Distinct-weights assumption: We can assume without loss of generality that the edge weights $w(e)$ for $e \in E$ are *distinct*. This can be achieved for instance by using a consistent tie-breaking rule using the IDs of the edges. A standard consequence of this assumption is that the MST of any given graph G with distinct weights is *unique*.

Proposition 1. *In any graph $G = (V, E)$ with weights $w : E \to \mathbb{R}^+$, if the weights are distinct, then the MST of G is unique.*

Proof. Proof by contradiction: suppose there are two different MSTs T_1 and T_2 for G. Let e_1 be any edge in T_1 not in T_2. Suppose we add e_1 to T_2. This will create a cycle C in $T_2 \cup \{e_1\}$. We consider two cases:

- *Case 1:* There is at least one edge $e_2 \neq e_1$ in C with $w(e_2) > w(e_1)$: Remove e_2 from T_2 and instead add e_1. Let $T := T_2 \cup \{e_1\} \setminus \{e_2\}$. This is also a spanning tree of G but now $w(T) < w(T_2)$ as we removed a "heavy" edge and brought in a "lighter" one. This is a contradiction with T_2 being a MST so this case cannot happen.
- *Case 2:* Every edge in the cycle C has weight higher than e_1. Remove e_1 from T_1 to create two connected components S and $V \setminus S$. Because C was a cycle with e_1, there is still a path in C from the endpoints of e_1 without using this edge and that path has at least one edge, say, f, between S and $V \setminus S$. Let $T := T_1 \setminus \{e_1\} \cup \{f\}$. This is also a spanning tree of G but now $w(T) < w(T_1)$, a contradiction with T_1 being a MST, so this cannot happen either.

Given the edge-weights are distinct, the only possible cases are the above and we showed neither can happen, hence proving the statement by contradiction. □

Consequently, throughout this lecture, we assume the edge-weights are distinct and thus, by Proposition 1, **MST of the input graph is unique**. In particular, we can refer to it as *the* MST of G (instead of a/some MST of G) which makes the analysis (mostly its exposition) easier.

1.1 The Basics Rules of MSTs

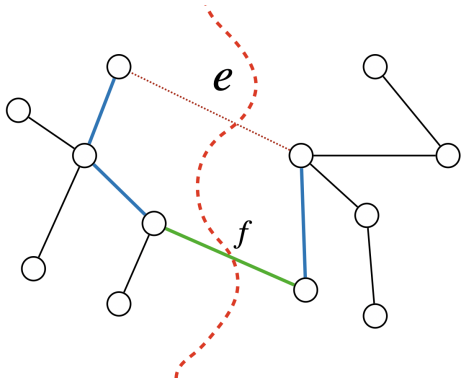
There are two general *rules* that are used quite frequently in the design of essentially every MST algorithm. You have most likely seen these rules before but perhaps not in this exact formulation. You may want to refer to Figure 1 for a simple illustration of the proofs of these rules.

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.*

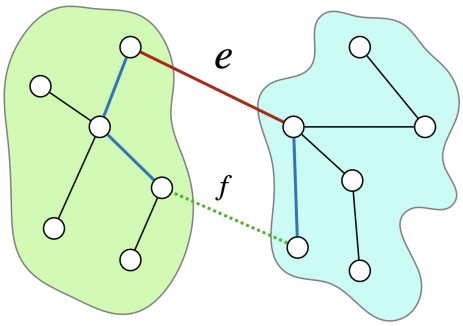
Proof. Suppose, by contradiction, that $S \subseteq V$ is a cut in G such that its minimum weight edge e is not part of the MST T. Add e to T which creates a cycle C and this cycle should have another edge f crossing the cut S. Consider $T' := T \cup \{e\} \setminus \{f\}$, which is still a spanning tree of G but has $w(T') < w(T)$ as $w(e) < w(f)$ since e is the minimum weight edge of the cut S. This is a contradiction with T being the MST of G. □

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.*

Proof. Suppose, by contradiction, that C is a cycle in G such that its maximum weight edge e is part of the MST T. Remove e from T to get two connected components, and find an edge $f \neq e$ in C that connects back these components. Consider $T' := T \setminus \{e\} \cup \{f\}$, which is still a spanning tree of G. We have $w(T') < w(T)$ as $w(f) < w(e)$ since e is the maximum weight edge of cycle C. This contradicts T being the MST. □



(a) Here, e is the minimum weight edge of the cut shown by red dashed line and thus should replace f in the MST. The cycle C is the thick (blue, green) edges and the dashed line for e.



(b) Here, e is the maximum weight edge of the cycle shown by the thick (blue, green, red) edges. Removing e from the MST creates two connected components that should be connected by the edge f instead.

Figure 1: An illustration of the proofs of **cut rule** and **cycle rule**.

2 A Classical Algorithm for MSTs: Boruvka's Algorithm

The three main classical algorithms for MSTs are *Kruskal's, Prim's,* and *Boruvka's* algorithms. You most likely have seen one of these algorithms in your prior courses (most likely the first two[1]). We will review *Boruvka's* algorithm in the following given that it will also play a rule as an important building block in subsequent improved algorithms. You can safely skip this section and continue to the next one if you are already familiar with this algorithm.

As far as I know, Boruvka's algorithm is the oldest algorithm for MSTs (it was first described in 1926 by Otakar Borůvka) and in my opinion is also simplest as it does not require even any particular data structure. It works by start having each vertex its own connected component and then proceeds in *rounds*. In each round, the algorithm picks the minimum weight edge going out of each component, add them to the tree, and merge those components to create larger connected components. Then, it goes to the next round and continues this until the whole graph is a single connected component.

Algorithm 1 (**Boruvka's Algorithm**).

(*i*) Start by creating a set $S(v) = \{v\}$ for each vertex $v \in V$ and an empty subgraph $T = \emptyset$ initially.

(*ii*) In each **round**:

(a) For each set S, find the minimum weight edge going out of S, i.e., $\min \{w(u,v) \mid u \in S, v \notin S\}$. Let e_S be the edge computed for the set S.

(b) Add every edge $e_S = (u,v)$ to the subgraph T and merge the sets of $S(u)$ and $S(v)$ into a single set S.

(*iii*) Repeat the rounds until all vertices are merged into a single set and return T.

Proof of Correctness: We first argue that T is a tree. Given that the maintained sets in the algorithm are connected components of T, the subgraph T is connected. Any edge added by the algorithm in a round also connects two different connected components. The only concern is that what if the edges added in the same round form a cycle? But, this cannot happen because the weight of the picked edges in any path should

[1]Sadly, Burovka's algorithm seems to be missing from many standard textbooks on algorithms even though I think it is actually simpler than both the other two alternatives!

be strictly decreasing (as in, if u picks an edge to v, but v picks an edge to w instead, then $w(u,v) > w(v,w)$) and thus the edges cannot form a cycle. As such, T is indeed a tree.

To argue T is the MST of G, we apply the **cut rule**: any edge inserted by the algorithm is the minimum weight edge going out of its connected component (hence the cut) and thus by the **cut rule** is in the MST.

Runtime Analysis: Each round of the algorithm takes $O(m)$ time as it simply involves going over all edges of the graph twice (once from each endpoint). The number of rounds is also $O(\log n)$ as we argue next. In each round, each connected components gets merged with at least another one, and thus the number of components decreases by, at least, a factor of two. Hence, after at most $\log n$ rounds, only one component remains and the algorithm terminates. Hence, the total runtime is $O(m \log n)$.

3 Karger-Klein-Tarjan Algorithm for MST

We now present one of the most "modern" algorithms for MSTs due to Karger, Klein, and Tarjan [KKT95], which runs in $O(m)$ time[2]—which is asymptotically optimal—but it is *randomized*, i.e., its $O(m)$ runtime is in expectation (and with high probability).

Theorem 2 ([KKT95])**.** *There is a randomized algorithm for the minimum spanning tree problem that runs in $O(m)$ time in expectation and with high probability[3].*

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).

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. This algorithm also benefits from the use of several (in some cases highly complicated) data structures to be able to implement this strategy quite fast. To present this algorithm, we need some preliminaries first.

3.1 Preliminaries

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

Definition 3. Fix any graph $G = (V, E)$ and any subgraph **forest** F 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 (including edges in F) as an **F-light** edge.

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

Observation 4. *For any graph G, any subgraph forest F 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 4 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 4 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.

[2]Throughout this course, we always use n to denote the number of vertices and m for the number of edges.

[3]Throughout this course, we say an event happens *with high probability* to mean its probability is at least $1 - 1/\text{poly}(m)$ for some arbitrarily (large) polynomial in the size of the underlying problem.

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 result is the we can find *all* F-heavy edges in linear time as well!

Theorem 5 ([Kom85, DRT92, Kin97])**.** *There is an algorithm that given any graph G and any subgraph forest F of G, outputs the set of all F-heavy edges in $O(m+n)$ time.*

We will cover this algorithm after showing how it can be used in the KKT algorithm.

3.2 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 2 (**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 5 to find all F-heavy edges of G' and let G_2 be the graph obtained from G' after removing the F-heavy edges.

(*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; we shall see its necessity in the analysis.

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 4 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. This is a so-called "sparsification" result.

Lemma 6. *The expected number of F-light edges in Algorithm 2 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. Let us the subgraph of F maintained so far when visiting the edge e_i by F_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 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. 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 choosing r). 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 5, 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. Let r_1 be the randomness used *out* of the recursive calls and r_2 be the one for recursive calls. Given Eq (1), we have,

$$\begin{aligned}
\mathop{\mathbb{E}}_{r}[T(m, n, r)] &\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r}[T(m(G_1), n(G_1), r)] + \mathop{\mathbb{E}}_{r}[T(m(G_2), n(G_2), r)] \\
&\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_1), n(G_1), r_2)]\right] + \mathop{\mathbb{E}}_{r_1}\left[\mathop{\mathbb{E}}_{r_2}[T(m(G_2), n(G_2), r_2)]\right] \\
&\quad \text{(the recursive calls themselves only depend on } r_2 \text{ (after fixing their input based on } r_1)) \\
&\leqslant c \cdot (m + n) + \mathop{\mathbb{E}}_{r_1}\left[2c \cdot (m(G_1) + n/8)\right] + \mathop{\mathbb{E}}_{r_1}\left[2c \cdot (m(G_2) + n/8)\right] \\
&\quad \text{(by induction hypothesis and as 3 rounds of Boruvka's algorithm reduces vertices by a factor of at least 8)} \\
&\leqslant c \cdot (m + n) + 2c \cdot (m/2 + n/8) + 2c \cdot (2n/8 + n/8) \\
&\quad \text{(as } \mathbb{E}[m(G_1)] = m'/2 \leqslant m/2 \text{ trivially and } \mathbb{E}[m(G_2)] \leqslant 2n' \leqslant 2n/8 \text{ by Lemma 6)} \\
&= 2c \cdot m + c \cdot (n + n/4 + 3n/4) = 2c \cdot (m + n),
\end{aligned}$$

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

This concludes the proof of Theorem 2 modulo the proof of Theorem 5 as one of its key subroutines.

Remark. The *KKT algorithm* provided the first linear time algorithm for MSTs but at the "cost" of randomization. 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 **major open question** in the area of graph algorithms is to obtain a deterministic algorithm for MSTs that also runs in $O(m)$ time.

3.3 Detour: Optimal Algorithms with Unknown Runtime

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[4]. 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.

[4]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...

4 MST Verification and Proof (Overview) of Theorem 5

We now give an overview of the proof of Theorem 5 (but will not cover all the details as they will become quite technical and for lack of a better word somewhat "messy").

We note that Theorem 5 is usually stated in the following equivalent way and is named the **MST verification** problem (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).

Theorem 7 (A reformulation of Theorem 5)**.** *Let T be a tree on n vertices V and $Q := \{(x_i, y_i) \mid x_i, y_i \in V\}$ be a given set of N **query pairs**. There is an algorithm that in $O(n+N)$ time returns a list of size N such that for every query pair $(x_i, y_i) \in Q$, the i-the value in the list, namely, the answer to the query pair, is the weight of the maximum weight edge in the unique x_i-y_i path in T (denoted by $w_T(x_i, y_i)$).*

To see why this theorem is equivalent with Theorem 5, we can set Q to be the set of edges in G and for each edge where the answer to the query pair has a lower weight than the weight of the edge, we know this edge must be T-heavy and otherwise T-light. The runtime of $O(n+m)$ in Theorem 5 also follows because we will be applying Theorem 7 with $N = m$ this way.

The proof of Theorem 7, requires multiple ingredients (due to several different researchers that have simplified this algorithm or its key subroutines over the years):

- **Boruvka trees** due to [Kin97]: reducing the general problem to a special case of the choice of T (being a *fully branching tree* to be defined later) and Q (only queries between the leaf-nodes);
- **Lowest Common Ancestor (LCA) queries** due to [BF00]: further reducing to a special case with more structure on queries in Q (only queries from leaf-nodes to one of their ancestors);
- **Komlos's Comparison-based algorithm** due to [Kom85]: solving the special types of queries above with a small number of *edge-weight comparisons* but not necessarily a fast algorithm still;
- **Fast implementation of Komlos's algorithm via bit-packing and table-lookups** due to [DRT92, Kin97]: implementing Komlos's algorithm in linear time and not just linear number of comparisons.

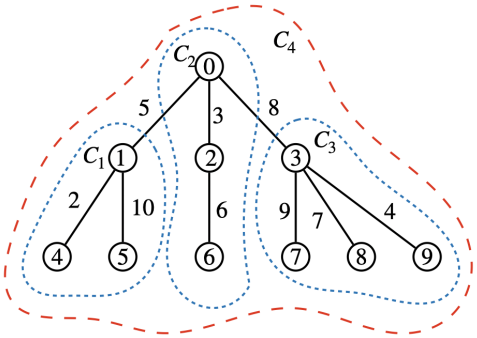
We now overview each of these ingredients in the following.

4.1 Step 1: Boruvka Trees

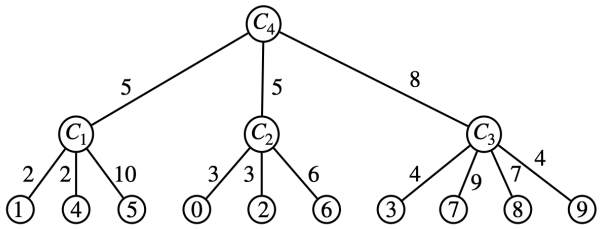
Consider the original tree T in Theorem 7 and create the following tree T' from it (see Figure 2 for an illustration). At first, we let leaf-nodes of T' be all the vertices in V. Then, we run one round of Boruvka's algorithm and for each connected component C created in this round, we create a new node C in T'; the child-nodes of C are the vertices in C and the weight of the edge connecting C to its child-node is equal to the weight of the edge used by the corresponding vertex when connecting to the component C. We then run the second round of Boruvka's algorithm and connect the connected components the same exact way as above, and continue like this until we end up with the root that corresponds to final connected component found by Boruvka's algorithm. We refer to the tree T' obtained this way from T as the Boruvka tree of T.

Lemma 8. *For any tree T on vertices V and its Boruvka tree T':*

1. *T' can be obtained from T in $O(n)$ time;*
2. *T' is a **fully branching tree** meaning that all its leaf-nodes are at the same height and every internal node has at least two child-nodes;*
3. *For every $x, y \in V$, $w_T(x, y) = w_{T'}(x, y)$ (i.e., the answer to a query pair (x, y) is the same over both T and T' for the leaf-nodes of T').*



(a) The connected components in the first round of the algorithm are marked with dashed (blue) edges, and the connected component of the second round is marked with long-dashed (red) edges.



(b) The corresponding Boruvka tree.

Figure 2: An illustration of running Boruvka's algorithm on a tree (in part (a)) and the corresponding Boruvka tree (part (b)).

Proof. We prove each part separately.

1. Recall that running Boruvka's algorithm in an n-edge graph in general takes $O(n \log n)$ time, so we cannot use the previous analysis directly. However, notice that on a tree, after running one step of Boruvka's algorithm, the graph obtained after contractions also is a tree. As such, since its number of vertices is dropped by half, its number of edges should also drop by half. This means that the runtime of Boruvka's algorithm on a tree is actually

$$O(1) \cdot \left(n + \frac{n}{2} + \frac{n}{4} + \ldots\right) = O(n).$$

   We note that this improved runtime is not limited to trees and also holds for planar graphs also (or any other minor-closed family of graphs).

2. This follows from the fact that each round of Boruvka's algorithm creates components of size two at least, and that the height of every leaf-node in T' is equal to the number of rounds before Boruvka's algorithm finishes (which is the same for all leaf-nodes).

3. Firstly, we claim that $w_T(x, y) \geqslant w_{T'}(x, y)$. Consider any node C on the path from x to y in T', which is not a common ancestor of x and y. The edge connecting this node to its parent is taken into account in the computation of $w_{T'}(x, y)$. Since C is not a common ancestor of x and y, it means that x and y do not simultaneously belong to the connected component corresponding to C. As such, at least one edge f of the x-y path in T is leaving this component; thus, when taking minimum weight edge leaving C in Boruvka's algorithm, the weight of the edge connecting C to its parent in T' will be at most equal to $w(f)$. Thus, any edge in the x-y path in T' has another edge in the x-y path in T with the same or higher weight, proving the claim.

   Secondly, we also claim that $w_T(x, y) \leqslant w_{T'}(x, y)$. Let f be the heaviest edge in the x-y path in T. Consider the round in Boruvka's algorithm wherein a connected component C picks the edge f as its minimum weight outgoing edge (such a round should necessarily exist before x and y get connected to each other in T'). We claim that C should contain at least one of x or y: any other component C' that contains a vertex from the x-y path but neither x nor y, has at least two edges leaving it and thus will never pick f which is the heavier of the two. But this implies that C should be on the x-y path in T' and since it is connected to its parent-node with weight $w(f)$, we have $w_{T'}(x, y) \geqslant w(f) = w_T(x, y)$, concluding the proof.

□

4.2 Step 2: Lowest Common Ancestor (LCA) Queries

Lemma 8 allows us to without loss of generality assume all queries Q in Theorem 7 are between leaf-nodes of T. At the same time, a single query (x, y) of this type can be answered using two queries $(x, LCA(x, y))$ and $(LCA(x, y), y)$ where $LCA(x, y)$ is the lowest common ancestor of x and y in T:

$$w_T(x, y) = \max\{w_T(x, LCA(x, y)), w_T(LCA(x, y), y)\}.$$

Thus, we can instead change Q to Q' which replaces each query (x, y) with the two corresponding LCA queries, and still recover the answer to the original problem as well (we will see in the next subsection why this will be helpful). The only catch is that we should also be able to find $LCA(x, y)$ for any given x and y quickly. This is handled by the following lemma.

Lemma 9. *There is a data structure that given a tree T, spends $O(n)$ time preprocessing T and after that for any given pair of vertices x and y, returns $LCA(x, y)$ in $O(1)$ time.*

To prove Lemma 9, we reduce it to another data structural question called the **range minimum** problem. Suppose we run DFS on T and write down both the name of each vertex in an array A and its distance from the root in an array D whenever we are at a vertex (not just the first time; in other words, write down an Euler tour of T starting from the root). See Figure 3 for an illustration.

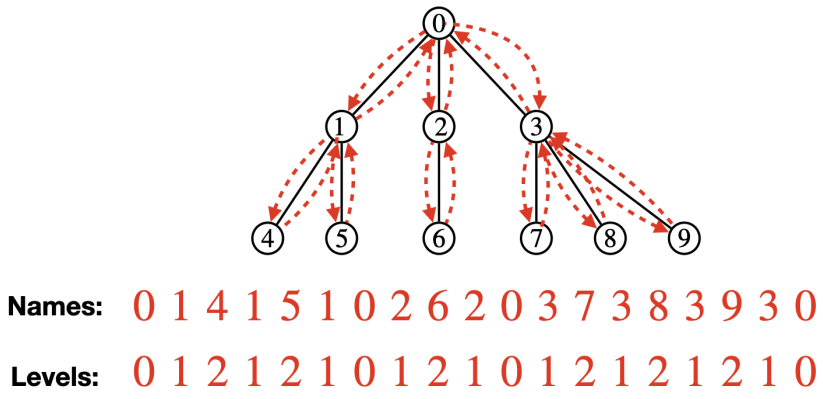


Figure 3: An illustration of the reduction from LCA to the range minimum problem.

We have the following important observation regarding these two arrays.

Observation 10. *For any pairs of vertices x and y in the tree T, let index i and j be the indices of two positions of x and y in the array A (and assume by symmetry $i \leqslant j$). Then, the node z with the minimum value in $D[i : j]$ is $LCA(x, y)$.*

The proof of this observation is simply by the correctness of DFS algorithm and we omit it here.

Since we can run the above DFS and compute the arrays A and D (with size $2n - 1$), our task at this point is reduced to the following: Given an array $D[1 : n']$ (for $n' = 2n - 1$), preprocess D such that given any two $i < j$, we can find the minimum value in $D[i : j]$ in $O(1)$ time. This is the **range minimum** problem, which we solve next.

A naive solution. The most obvious algorithm for the range minimum problem is to spend $O(n^2)$ time and create an $M[n][n]$ where $M[i][j]$ contains the answer to the query (i, j), namely,

$$M[i][j] = \min_{i \leqslant k \leqslant j} D[k].$$

Of course, this algorithm is too slow to be of any use for us.

The sparse-table solution. A far better solution for this problem is called *sparse-table*. Suppose we maintain the following array $M[n][\log n]$: for every $i \in [n]$ and $k \in [\log n]$, we have

$$M[i][k] = \min_{i \leqslant j \leqslant i+2^k} D[i];$$

in other words, $M[i][k]$ stores the minimum value in an interval of length 2^k starting from i.

To answer a query (i, j) we find the largest k such that $2^k \leqslant j - i < 2^{k+1}$ and return

$$\min\left(M[i][k], M[j - 2^k][k]\right);$$

Notice that both intervals $[i : i + 2^k]$ considered in $M[i][k]$ and $[j - 2^k : j]$ considered in $M[j - 2^k][k]$ are inside the range $[i : j]$ (since $j - 2^k \geqslant i$). Moreover, we have $i + 2^k \geqslant j - 2^k$ because $j - i \leqslant 2^{k+1}$ and thus these intervals cover the entire range $[i : j]$. This implies that the returned answer is correct.

Finally, how long does it take to create the array M in the first place? Notice that

$$M[i][k] = \min\left(M[i][k-1], M[i + 2^{k-1}][k-1]\right);$$

This allows us to use a dynamic programming approach (in increasing value of k for $M[i][k]$) and compute each entry in $O(1)$ time, which in turn leads to an $O(n \log n)$ time algorithm for computing all of M.

Thus, overall, the sparse-table solution gives us an algorithm with $O(n \log n)$ preprocessing time and $O(1)$ query time. While quite efficient, even this algorithm is not good enough for our purpose.

The optimal solution of Bender and Farach-Colton [BF00]. To improve sparse-tables, we can make an important observation: the array D created in the context of LCAs has a special property: each entry of D is ± 1 of the previous entry! We are now going to use this extra information using two (closely related) techniques: bit-packing and look-up tables. The solution is as follows:

- We first partition the array D into $t = 2n/\log n$ consecutive blocks of $(\log n)/2$ denoted by $B_1, \ldots, B_t$.

  This step takes $O(n)$ time.

- Inside each block B_i for $i \in [t]$, we maintain *prefix* minimums and *suffix* minimums, i.e., for each $j \in B_i$, we store

  $$P[j] = \min_{k \leqslant j \in B_i} D[k] \qquad \text{and} \qquad S[j] = \min_{k \geqslant j \in B_i} D[k]$$

  This step also takes $O(n)$ time.

- We also create an array D' of size t where $D'[i] = \min_{j \in B_i} D[j]$ and compute a sparse-table over D'.

  This step takes $O(n/\log n \cdot \log n) = O(n)$ time.

- Given any query i and j where i and j do not belong to the block B, we can answer the query using the above information as follows. Let B_1 denote the block of i and B_2 denote the block of j. We compute the minimum between B_1 and B_2 using the sparse-table for D' and use suffix-minimum array of i in B_1 and prefix-minimum array of j in B_2 to find the total minimum.

  This takes $O(1)$ time to answer the query.

- Thus, at this point, the only queries remain to answer are i and j inside the same block B. Notice that in this case, the absolute value of the first entry of B does not matter and only the ± 1 relative values are important. Thus we can treat B as a single number of length $(\log n)/2$ bits. There are in total $\sqrt{n}$ such numbers and for each number there are at most $(\log n)^2/4$ choices for i and j. Thus, we can create a *lookup table* of size $O(\sqrt{n} \cdot \log^2 n) = o(n)$ which is indexed by the choices of these numbers for B and i and j; the value there also points to a single index which is the answer to this query.

  We can create each entry of this table in $O(1)$ time since we can operate on a single number in the RAM-model in $O(1)$ time. Thus, creating the look-up table takes $o(n)$ time and we can answer each query in $O(1)$ time subsequently.

Overall, this solution gives an optimal $O(n)$ preprocessing and $O(1)$ query time algorithm.

Lemma 9 now follows from this algorithm and the reduction to range minimum problem (with ± 1 entries) described earlier.

4.3 Step 3: Komlos' Comparison-Based Algorithm

Recall that by the previous two steps, the problem we need to solve is the following: given a fully branching tree T and a set of leaf-to-ancestor queries Q, find the answer to each query over the tree T. We now design an algorithm for this problem due to Komlos [Kom85] that is not necessarily fast enough, but only requires a linear number of edge-comparisons.

Consider the tree T and for each node $u \in T$, define T_u as the sub-tree of u in T. For every node $u \in T$, we define the following two arrays (see Figure 4 for an illustration):

- **Query array** $q[u] = (w_1, w_2, \ldots, w_k)$ be the name of ancestors of u (including u itself) such that for each w_i, there is some vertex $v \in T_u$ with a query (v, w_i) in Q (w_i's may have repetition). Moreover, these vertices are ordered from the root toward u, meaning that w_1 is closer to the root than w_2, etc.
- **Answer array** $a[u] = (a_1, a_2, \ldots, a_k)$ be the following values: for each $i \in [k]$, the maximum weight of an edge in T on the path from u to w_i, i.e., $a_i = w_T(u, w_i)$. Notice that we have $a_1 \geqslant a_2 \geqslant \ldots \geqslant a_k$.

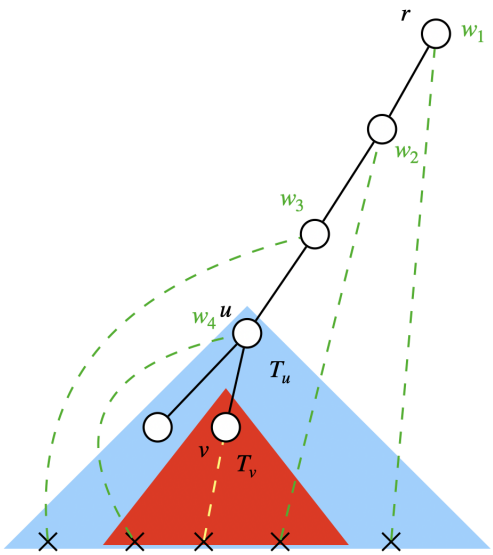


Figure 4: An illustration of the query array $q[u]$ for a vertex $u \in T$. When updating v from u, there is a new query to be added to the query array (for leaf-nodes of T_v that connect to v) and the queries that originate from $T_u \setminus T_v$ needs to be ignored.

By definition, if we know the answer array for every leaf-node, we will be done (as $a[v]$ for a leaf-node is the maximum weight edge from v to its ancestor that v has a query to). We thus start from the root and calculate these values recursively as follows for a child-node v of a node u we already handled:

- $q[v]$ is going to be a subset of $(w_1, \ldots, w_k)$, say, $(w_{i_1}, \ldots, w_{i_{k'}})$ plus possibly vertex v appended to its end (if a leaf-node in T_v queries v; this query was ignored in u as v is a descent of u not ancestor).
- $a[v]$ is going to be
$$a[v] = (\max(a_{i_1}, \theta), \max(a_{i_2}, \theta), \ldots, \max(a_{i_{k'}}, \theta), \theta)$$
where $\theta = w(u, v)$, namely, weight of the edge (u, v) (the last entry of $a[v]$ is θ only if v is appended to the end of of $q[v]$; otherwise, similar to $q[v]$, length of $a[v]$ will also be k').

The important observation is that since $a[u]$ is sorted, to create $a[v]$, we can simply do a binary search on $a[u]$ to figure out the entries larger than θ and use this information to create $a[v]$ as well. This way, the total number of comparisons needed to create $a[v]$ from $a[u]$ is only $(\log(1 + |a[u]|))$.

In the following, we use the fact that T is a fully branching tree to bound the total number of comparisons.

Lemma 11. *The total number of comparisons to answer N leaf-to-ancestor queries on a fully branching tree T with n vertices using Komlos' algorithm is $O(N + n)$.*

Proof. Let h be the height of the tree T, T_i be the set of vertices at height i of the tree T (where leaf-nodes are at height 1) and $n_i = |T_i|$. Also, let Q_u denote the size of $q[u]$ and $a[u]$ for a vertex $u \in T$. We have,

$$\begin{aligned}
\#\text{ of comparisons} &= \sum_{u \in T} (\log(1 + Q_u)) \\
&= \sum_{i=1}^{h} \sum_{u \in T_i} (\log(1 + Q_u)) \qquad \text{(by partitioning the sum across different levels of the tree)} \\
&= \sum_{i=1}^{h} n_i \cdot \mathop{\mathbb{E}}_{u \sim T_i} [\log(1 + Q_u)] \\
&\qquad\qquad \text{(by writing the sum as the expectation for } u \text{ chosen uniformly over } T_i) \\
&\leqslant \sum_{i=1}^{h} n_i \cdot \log(1 + \mathop{\mathbb{E}}_{u \sim T_i} [Q_u]) \\
&\qquad \text{(by Jensen's inequality } E[f(X)] \leqslant f(E[X]) \text{ for a concave function } f \text{ and } f = \log(1 + x) \text{ is concave)} \\
&\leqslant \sum_{i=1}^{h} n_i \cdot \log(1 + \frac{N}{n_i}) \\
&\text{(as each of } N \text{ queries appears in at most one } q[u] \text{ of a single vertex } u \in T_i, \text{ since they originate from sub-tree } T_u) \\
&\leqslant \sum_{i=1}^{h} n_i \cdot \log(\frac{N + n}{n_i}) \\
&= \sum_{i=1}^{h} n_i \cdot \log \frac{N + n}{4n} + \sum_{i=1}^{h} n_i \cdot \log \frac{4n}{n_i}. \\
&\qquad\qquad \text{(as } \log \tfrac{N+n}{n_i} = \log(\tfrac{N+n}{4n} \cdot \tfrac{4n}{n_i}) = \log(\tfrac{N+n}{4n}) + \log(\tfrac{4n}{n_i}))
\end{aligned}$$

We now bound each of these terms separately.

First term. Using $1 + x \leqslant e^x$, we have,

$$\sum_{i=1}^{h} n_i \cdot \log \frac{N + n}{4n} \leqslant \sum_{i=1}^{h} n_i \cdot \frac{N}{n} \cdot \log e = N \cdot \log e = O(N),$$

since sum of n_i's is equal to n, the number of vertices in the tree.

Second term. Using the fact that $n_i \leqslant n/2^{i-1}$, since T is a fully branching tree, we have,

$$\sum_{i=1}^{h} n_i \cdot \log \frac{4n}{n_i} \leqslant \sum_{i=1}^{h} \frac{n}{2^{i-1}} \cdot \log(2^{i-1}) \leqslant n \cdot \sum_{i \geqslant 0} \frac{i}{2^i} = O(n),$$

where in the inequality, we use the fact that the function $f(x) = x \cdot \log(1/x)$ is monotone for $x \geqslant 4$.

Combining the two terms above concludes the proof. □

4.4 Step 4: A Linear-Time Algorithm for MST Verification

The final step is to implement Komlos' comparison-based algorithm in a way that each step can also be implemented in $O(1)$ time. We will not get into the entire details of this step as it involves delicate implementation details (which are beyond the scope of this course) and instead give a high-level overview.

The idea is again to use a bit-packing plus a table-lookup argument quite similar to the algorithm for LCAs we covered earlier. Recall that the entire height of the tree is at most $\log n$ bits. This means that a single query array $q[u]$ (if we ignore the repetitions in the queries) can be represented by a $\log n$-bit number: the i-th bit is 1 iff the i-th vertex on the path from root to node u has a query and is otherwise 0. Similarly, although quite a lot trickier, we can also represent the answer array $a[u]$ with a single $\log n$-bit number: the idea is to store the *indices* of the edges that correspond to this particular answer and use the fact that this edges are monotically moving closer to u; we are going to omit this step entirely but you can read about it in [DRT92] or [Kin97].

With such a *bit-packing* approach, implementing one step of Komlos' algorithm to go from $q[u], a[u]$ of a parent-node u to $q[v], a[v]$ of one of its child-nodes v, we can "load" the arrays q and a in $O(1)$ time into the memory, and spend $O(1)$ time to make any necessary manipulations (here, one can either assume *any* arbitrary operation on $\log n$-bit numbers can be done in $O(1)$ time, or perhaps more realistically, first create a look-up table for the types of operations needed and answer based on that; notice such an approach requires one to use $(\log n)/2$ or small array sizes which in turn needs splitting the arrays q and a into $O(1)$ chunks; this can also be made to work). There is of course also the binary search step of Komlos' algorithm but each step of that can also be done in $O(1)$ time using these representations.

All in all, with all these tricks, one can implement Komlos' comparison-based algorithm in $O(N + n)$ time also. This finalizes the proof (overview) of Theorem 7.

We conclude these notes with another important **open problem**: is there a "simple" algorithm for MST verification that runs in linear time, say, even with randomization? In particular, can we achieve such a runtime without any "bit-tricks" (more formally, in the *pointer machine* model instead of RAM model that does not allow for these types of arbitrary arithmetic computations)?

References

[BF00] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN 2000: Theoretical Informatics, 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000. 8, 11

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

[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. 5, 8, 14

[Jon97] Neil D. Jones. *Computability and complexity - from a programming perspective*. Foundations of computing series. MIT Press, 1997. 7

[Kin97] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997. 5, 8, 14

[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. 4

[Kom85] János Komlós. Linear verification for spanning trees. *Comb.*, 5(1):57–65, 1985. 5, 8, 12

[PR02] Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. 6, 7