CMPUT 498/501: Advanced Algorithms | Fall 2025

Lecture 4 (Sep 15, 2025): Minimum Spanning Trees

Lecturer: Mohammad R. Salavatipour | *Scribe: Keven Qiu*

4.1 Brief History of Minimum Spanning Trees

Finding a minimum spanning tree (MST) is a classic graph theory problem. Given an edge weighted, simple graph $G = (V, E)$, $w : E \to \mathbb{R}^+$, we want to find a minimum weight spanning tree T where the weight of T is

$$w(T) = \sum_{e \in T} w(e)$$

1. Boruvka first came up with MST algorithm in 1926. A few others rediscovered Borukva's later on.
2. Jarnik gave his own algorithm in 1930, which was rediscovered by Prim and Dijkstra notably, which it is now name Prim's algorithm.
3. Kruskal gave his own algorithm in 1956.
4. Freman and Tarjan gave an $O(m \log^* n)$ algorithm in 1984.
5. Karger, Klein, and Tarjan gave an $O(m)$ linear time randomized algorithm in 1995.

In this lecture, we go through Prim's, Kruskal's, and Boruvka's algorithms, then Fredman and Tarjan's $O(m \log^* n)$ algorithm.

4.2 Minimum Spanning Tree Rules

Without loss of generality, we can assume the edge weights are all distinct. Otherwise, we can put in a tie breaking rule.

Proposition 1 *Given a graph with distinct edge weights, the MST is unique.*

Proof. Suppose there are two distinct MSTs T_1 and T_2 for a contradiction. Let e be the minimum weight edge among the two trees and without loss of generality, let $e \in T_1$.

$T_2 \cup \{e\}$ contains a cycle C. Let $e' \in E(C) - \{e\}$, where $e' \notin T_1$. Since e is the minimum weight edge and edge weights are distinct, i.e. $w(e) < w(e')$, then $T_2 \cup \{e\} - \{e'\}$ is a spanning tree of smaller weight than T_2 (adding e creates a cycle and removing a different e' from the cycle leaves the graph connected and acyclic that still spans all vertices). This means that T_2 was an MST, but we achieved a tree of smaller weight, a contradiction. ■

Definition 1 *For a graph G, any non-trivial set $S \subset V$ defines a cut*

$$\delta(S) = \{e \in E : e \textit{ has one endpoint in } S \textit{ and the other in } V - S\}$$

We define the Cut Rule, a property of cut edges of any MST:

Proposition 2 *(Cut Rule) In any graph G, for any cut S, the minimum edge $e \in \delta(S)$ belongs to the MST T.*

Proof. Suppose not, and say the minimum weight edge $e_{\min} \notin T$. Now consider adding $e_{\min}$ to T. $T \cup \{e_{\min}\}$ will create a cycle C. There exists an edge $e' \in C$ across the cut with $w(e') > w(e_{\min})$. If we remove e' from T and add $e_{\min}$ into T, we get a spanning tree with less weight than before. This contradicts T being an MST. ■

Proposition 3 *(Cycle Rule) For any cycle C, the heaviest edge on C cannot be in the MST.*

Proof. Assume for a contradiction that the heaviest edge $e = (u, v)$ of a cycle C is in the MST T. Delete e from T. Since T is a tree, $T - e$ will create two components, say C_1 and C_2. Without loss of generality, let $u \in C_1$ and $v \in C_2$.

Note that the path $C - e$ is a uv-path and u and v are in separate components, so there is an edge $e' \in C - \{e\}$ on the uv-path $C - e$ that has one endpoint in C_1 and the other in C_2 that connects the two components. So $T' = T \cup \{e'\} - \{e\}$ is a spanning tree since $|E(T')| = |V| - 1$. By choice of e, $w(e') < w(e)$, which implies $w(T') < w(T)$. ■

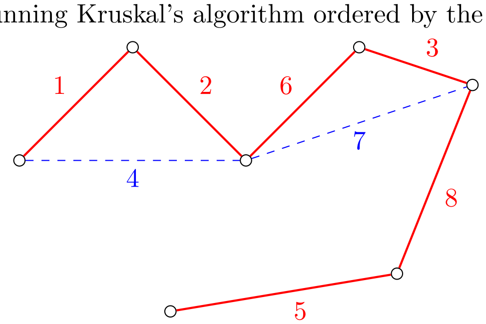
4.3 Classic MST Algorithms

4.3.1 Kruskal's Algorithm

Algorithm 1 Kruskal's Algorithm

Sort edges E in non-decreasing order of weight, i.e. $w(e_1) \leq w(e_2) \leq \cdots \leq w(e_m)$.
$T = (V, \emptyset)$
Each vertex of G will be its own component
for $e \in E$ **do**
    if e connects two different components **then**
        Add e to T.
        Merge components into one connected component.
return T

Figure 4.1: Running Kruskal's algorithm ordered by the labelled edges.



Proof of Correctness. Each ignored edge by the algorithm creates a cycle. Since we consider edges in non-decreasing order, this edge must be the heaviest of the cycle, which cannot be in a MST by the Cycle Rule. ■

Definition 2 *A union-find is a data structure that supports three operations on an input set* $[n] := \{1, \ldots, n\}$*:*

- *Initialize: starts data structure by creating* n *disjoint sets* $S_1, \ldots, S_n$ *where* $S_i := \{1\}$*.*
- *Union*(i, j)*: given index of two sets* i, j*, replace* S_i *and* S_j *with* $S_i \cup S_j$ *and the index of the new set with* $\min\{i, j\}$*.*
- *Find*(e)*: given an element* $e \in [n]$*, return the index of the set.*

Easy implementations of union-find can be done with $O(n)$ time for initialization and $O(\log n)$ update time. We can use Union-Find to implement Kruskal's algorithm in $O(m \log n)$ runtime.

Using a more efficient implementation of Union-Find, we can find/union using m operations in amortized time $O(m \cdot \alpha(m))$, where α is the inverse Ackermann function (grows slower than $\log^*(\cdot)$ where $\log^* n$ is the number of iterated log one needs to take to get to 1).

The total running time for Kruskal's algorithm is $O(m \log n + m\alpha(m))$.

4.3.2 Prim's Algorithm

Unlike Kruskal's algorithm that grows trees from individual nodes and merges them, Prim's algorithm grows one tree until it is spanning.

Algorithm 2 Prim's Algorithm

```
Start with arbitrary vertex s.
Let T be tree with zero edges on s
for i = 1 to n − 1 do
    Pick the cheapest edge e = (u, v) between T and V − T, where u ∈ T, v ∈ V − T.
    Add e to T
return T
```

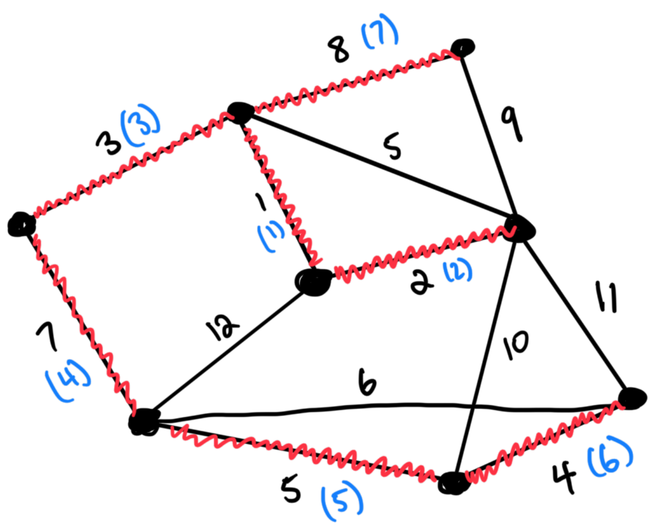


Figure 4.2: Prim's algorithm example

We can use a priority queue to pick the cheapest edges in each iteration.

Algorithm 3 Prim's Algorithm with PQ

Start with arbitrary vertex s.
$T = (\{s\}, \emptyset)$
Let Q be a priority queue with keys being vertices and key-values being edge(v) and its weights
For each $v \in N(s)$, edge(s) $= (s, v)$ and Q.insert($v, w(\text{edge}(v))$)
for $i = 1$ to $n - 1$ **do**
    Pick v with smallest value in Q (Q.extract_min
    for each $u \in N(v)$ **do**
        if $u \notin T$ **then**
            Update Q.decrease_key($u, w(uv)$) (if u is not in Q, we instead run Q.insert($u, w(uv)$)).
return T

Proof of correctness: First note that the algorithms adds $n-1$ edges and it always adds an edge from a tree to another vertex outside of the tree; so never creats a cycle. Thus it finds a spanning tree. To prove it returns a MST we use the cut-rule: anytime we add an edge e to the tree T, the set of vertices in T are connected and all the edges between T and $V - T$ are added to Q, so Q contains all $\delta(T)$ and e is the smallest edge across the cut. So it belongs to a MST.

The runtime analysis:

- $O(n)$ insert operations from the iterations of running the loop.
- $O(m)$ decrease_key operations, which we do twice for each edge.
- $O(n)$ extract_min operations to add the next edge.
- Using min-heap implementation for PQ the total time will be $O(m \log n)$.

4.3.3 Boruvka's Algorithm

One of the oldest MST algorithms and its implementation is between Kruskal's and Prim's.

Algorithm 4 Boruvka's Algorithm

Start from $S(v) = \{v\}$ for each $v \in V$.
$T = \emptyset$.
while there are more than one sets **do**
    For each set S, find the cheapest edge $e \in \delta(S)$, call it e_S.
    Add all edges e_S to T and merge all sets that these edges run between.
return T

Correctness: It is a spanning tree because no edge that gets added creates a cycle and we repeat until there is one single component. A connected, acyclic graph that contains all vertices of G is a spanning tree. To show it is minimum, we use the Cut Rule. Any edge selected by the algorithm is a min-cost edge going out of a component and hence is in the MST.

Runtime: There are $O(\log n)$ rounds as the number of component goes down by a factor of 2 each time. In each round we spend $O(m)$ time for a total of $O(m \log n)$.

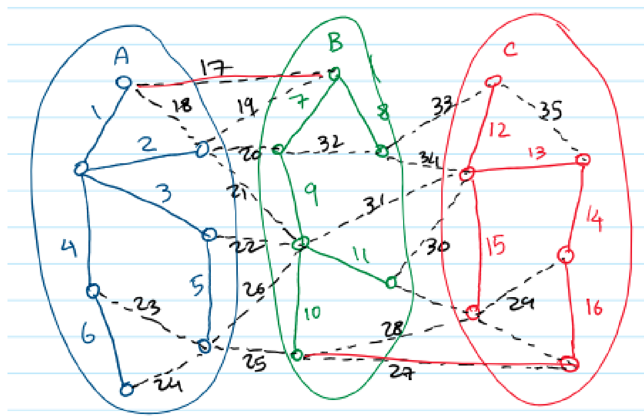


Figure 4.3: The 3 components/trees of Fredman-Tarjan and the edges chosen after shrinking each component.

4.4 Fredman-Tarjan MST

Definition 3 *A Fibonacci heap is an advanced implementation of the priority queue using heaps. It supports*

- $O(1)$ *amortized runtime for insert and decrease_key.*
- $O(\log n)$ *amortized time for extract_min where n is the maximum size of the heap at any time.*

If we use Fibonacci heaps on Prim's algorithm, we have

- $O(n)$ insert operations.
- $O(m)$ decrease_key operations.
- $O(n)$ iterations and extract_min operations, for a total of $O(m + n \log n)$ time.

We can improve on Prim's idea to get a better runtime. Each iteration, we have a priority queue of edges going out of the current tree. We can ensure the current tree has a small boundary and use a Fibonacci heap.

Run Prim's algorithm as long as the boundary is bounded by a constant. Once it becomes too big, start growing tree from another vertex. Once every vertex belongs to one of these trees, $C_1, C_2, \ldots,$ contract each C_i into a single vertex and recurse on this new contracted graph.

This is the main idea of Fredman-Tarjan. The algorithm runs in rounds, where in round i we have graph G_i with n_i vertices and m_i edges, obtained by contracting some trees in the previous rounds.

Let $G_1 = G$. In each round,

1. Pick an unmarked vertex and run Prim's algorithm to grow a tree T. Keep track of lightest edges going out of T to vertices in $N(T) = \{u \in V - T : \exists v \in T, uv \in E\}$.
2. If $|N(T)| \geq k_i$ or if added an edge to a marked vertex, stop. Mark all vertices in T and go to Step 1.
3. If no unmarked vertex left, contract each tree into a single vertex and go to next round.

Runtime: The runtime for each round is $O(m + n \log k_i)$ since

- each edge is considered twice
- each time we grow a tree, the number of components decrease, so $O(n)$ times
- the priority queue operations take $O(\log k_i)$.

Observation 1: For every component C, $\sum_{v \in C} \deg(v) \geq k_i$.

Proof of Observation 1. When v gets added to a component C, this is because the number of edges going out of the current tree is $\geq k_i$. So clearly, $\sum_{v \in C} \deg(v) \geq k_i$. ■

So if $C_1, \ldots, C_\ell$ are the components at the end of the current round, then

$$\sum_{i=1}^{\ell} \sum_{v \in C_i} \deg(v) \geq \ell k_i$$

Assuming we have m_i edges, by handshaking lemma

$$2m_i = \sum_{v} \deg(v) \geq \ell k_i \implies \ell \leq \frac{2m_i}{k_i} \leq \frac{2m}{k_i}$$

We let $k_i = 2^{2m/n_i}$. Also note that the number of vertices in each round satisfies

$$n_{i+1} \leq \frac{2m_i}{k_i} \implies k_i \leq \frac{2m_i}{n_{i+1}} = \log k_{i+1}$$

So the threshold k_i exponentiates in each round ($k_{i+1} \geq 2^{k_i}$), implying the number of rounds is bounded by $\log^* n$. Thus, total runtime of the algorithm is $O(m \log^* n)$.

4.5 Linear Time MST

There is a randomized $O(m+n)$ time MST algorithm by Karger-Klein-Tarjan (KKT).

Definition 4 *Suppose $F \subseteq G$ is a forest. An edge $e \in E$ is F-heavy if e creates a cycle in $F \cup \{e\}$ and it is the heaviest edge in that cycle. Otherwise, e is F-light.*

Observation:

- e is F-light if and only if $e \in \mathrm{MST}(F \cup \{e\})$.
- If T is an MST, then e is T-light if and only if $e \in T$.
- For any forest F, the F-light edges contain MST of G, i.e. for any F-heavy edge e, $\mathrm{MST}(G-e) = \mathrm{MST}(G)$.

If F is a forest, we can discard F-heavy edges from G. So our goal is to find a forest with as many F-heavy edges as possible. F is close to an MST in this case. We can recurse on the remaining edges.

The problem arises on how to find a good forest F and how to classify edges as F-heavy fast.

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

Idea for KKT: Randomly choose half of the edges and find a minimum spanning forest F over the edges. Find the F-heavy edges and discard them, then recurse on the rest of the graph.