CS 6363.005.19S Lecture 14—March 7, 2019

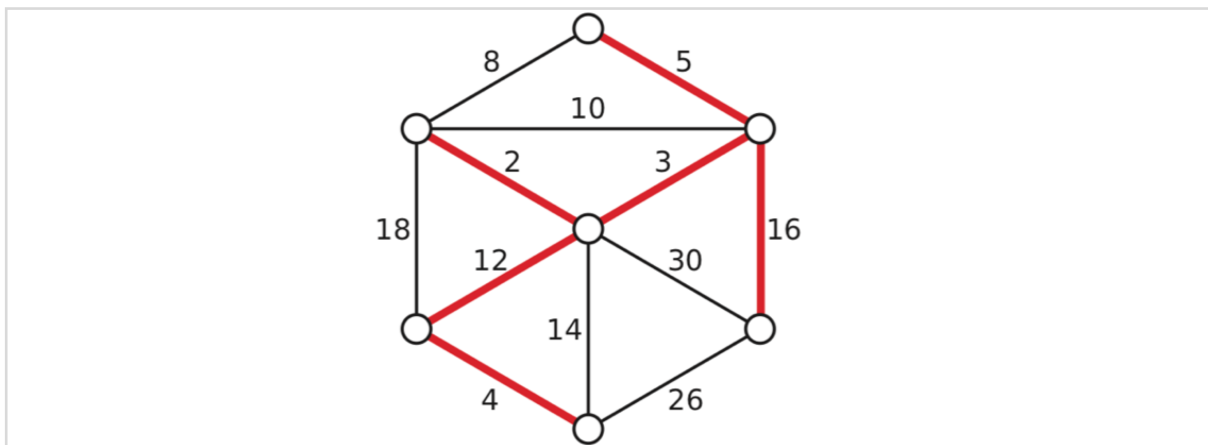
Main topics are #minimum_spanning_trees.

Prelude

- Homework 3 is due **Thursday** March 14.

Minimum Spanning Trees

- Let's say we have a bunch of electrical stations that we want to connect in a grid.
- We can represent the stations as circles on the board.

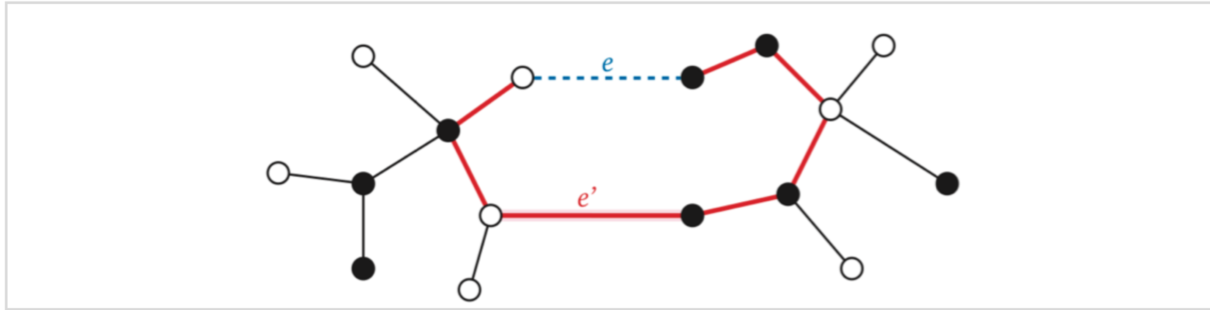


- We can connect some pairs of stations together using a single electrical line, but each choice of line has a different cost; maybe the distance between the stations.
- And our goal here is to connect the stations together in the cheapest way possible.

- Formally, let's say we're given a connected, undirected, *weighted* graph G = (V, E). The weights are a function w : E → R that assigns weight w(e) to each edge e. I'm even allowing negative weight edges.
- We want to find the *minimum spanning tree*, the spanning tree T that minimizes w(T) = sum_{e in T} w(e).
- For simplicity, we'll assume edge weights are distinct: w(e) ≠ w(e') for any pair of edges e and e'. One neat consequence is that the minimum spanning tree is unique if you do this. I'll argue why when I start describing algorithms for this problem.
- If you do have ties, then you might have multiple minimum spanning trees. For example, all spanning trees have V - 1 edges, so if the weights are all 1, then *every* spanning tree is also a *minimum* spanning tree.
- You can avoid the assumption if you have a consistent way to break ties, but that's all I'll say about that today.

The Only Minimum Spanning Tree Algorithm

- There is really only one minimum spanning tree algorithm (at least for this class), and all you need to do to get your name attached to it is to figure out a fast way to implement it.
- And despite all my warnings, this one is a greedy algorithm that actually works.
- So the idea is we're going to be adding edges one-by-one to build the minimum spanning tree. Let's say this is a snapshot of what the world looks like halfway through. **draw like three edges of the MST**
- We have this acyclic subgraph F we'll call the *intermediate spanning forest*.
- We're only adding edges, so F needs to be a subgraph of the minimum spanning tree.
- F consists of n one-node trees before we've added any edges.
- As we add edges to F, we'll merge these trees together. The algorithm halts when F consists of a single n-node tree, the minimum spanning tree.
- But which edges are we going to add to F?
- We'll define two types of edges based on the current intermediate spanning forest F.
- *Useless* edges are outside of F, but both endpoints are in the same component of F. **draw arrow to a useless edge**
- Claim: The minimum spanning tree contains no useless edge.
  - If we added a useless edge to F, it would create a cycle!
- For each component of F, we'll associate a *safe* edge as the minimum weight edge with exactly one endpoint in that component.
- So, that's one safe edge per component, although a pair of components may share a safe edge.
- Some edges are neither useless nor safe for this particular forest F. By the time we've computed the minimum spanning tree, all edges will be in the tree or be designated as useless.
- Claim: The minimum spanning tree contains every safe edge. In fact, for every subset of vertices S, the minimum spanning tree contains the minimum-weight edge e with one endpoint in S.
  - We'll use an exchange argument like we saw with greedy algorithms.
  - Suppose to the contrary that the minimum spanning tree T *does not* contain e.
  - T contains a path between the endpoints of e, but that path starts in S and ends not-in S. There must be some edge e' on the path with one endpoint in S.
  - Here's the situation: The black vertices are S, the rest are V - S.

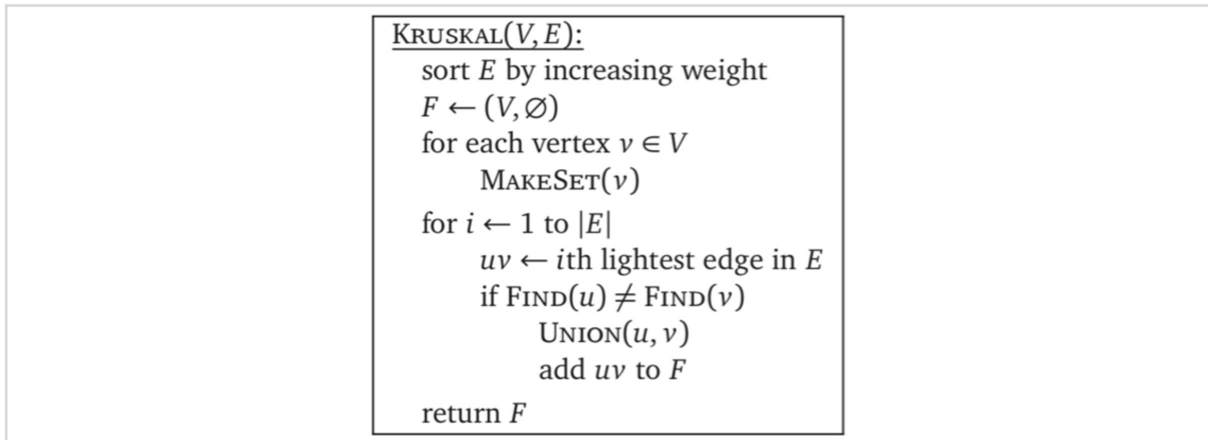


  - T is acyclic, so if we remove e', we create a spanning forest with two components.
  - Each component contains an endpoint of e. Otherwise, the path we were talking about would not cross to the other component or contain e'.
  - So that means we can add e in to get a new spanning tree T' = T - e' + e.
  - But w(e) < w(e'), so this new spanning tree has smaller total weight. T must not have been the minimum spanning tree.
- So, we can throw away useless edges, and we can safely include every safe edge.
- That implies the following greedy algorithm: add one or more safe edges to the evolving forest F, and recurse.
- As we add new edges to F, some undecided edges become safe or useless.
- We need to figure out which safe edges to add in each iteration, and how to identify new safe and new useless edges.
- There's two algorithms that most people know and are particularly useful for teaching. We'll go over those today.

Kruskal's Algorithm

- Found by Kruskal in 1956.
- Kruskal: Scan all edges in increasing weight order; if an edge is safe, add it to F.
- Claim: Immediately after scanning edge e and maybe putting it in the tree, all edges of weight ≤ w(e) are correctly in F or are useless.
  - Assume inductively the claim is true for lighter edges than e.
  - Maybe e is useless. The claim follows immediately.
  - Or, e isn't useless. But then it's the lightest non-useless edge outside the tree. Meaning it's the lightest edge spanning two components, so it's safe to add e.
- To implement the algorithm, we use something called a disjoint sets data structure that supports two operations, Union and Find.
- In our setting, every component of F has a "leader" node.
- If you call Find(v), then it returns the leader of v's component. So Find(u) = Find(v) if and only if u and v are in the same component.
- If you call Union(u, v), then you are declaring to the data structure that you are combining u and v's components, and you select a new leader for the one combined component.
- MakeSet(v) makes v the leader of its own one-vertex component.

- So, for each edge in increasing order of weight, we'll check if the two leaders differ using Find and if they do, we'll add the edge to F and Union its two endpoints.

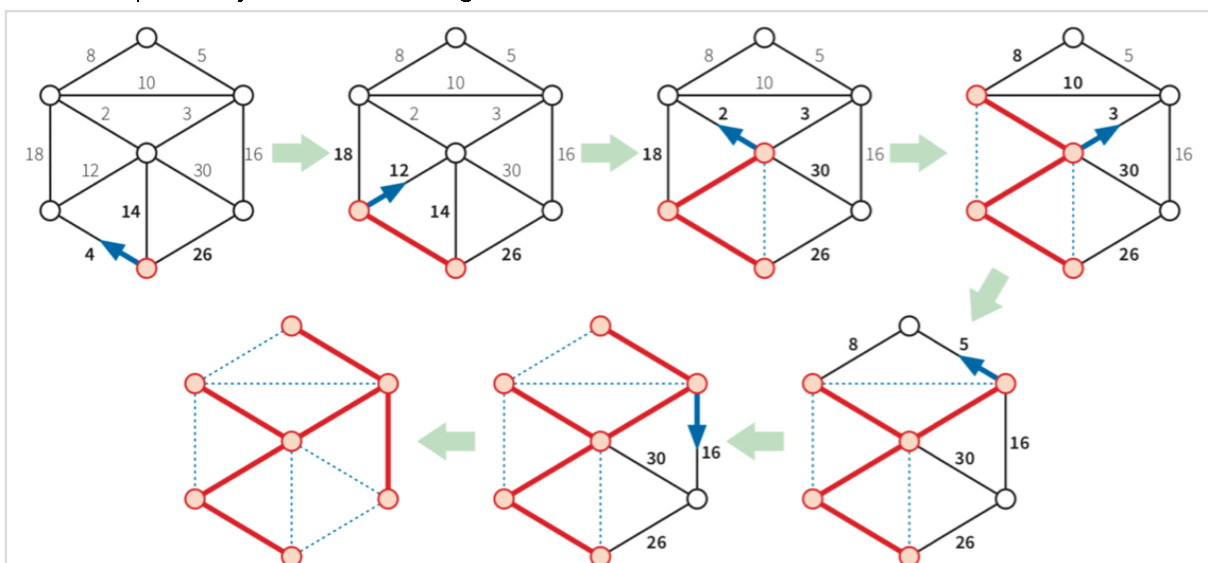


```
KRUSKAL(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv to F
  return F
```

- We'll take O(E log E) = O(E log V) time to sort the edges.
- We do O(E) find operations, one per edge.
- We do O(V) union operations, one per edge of the minimum spanning tree.
- Using the best Union-Find data structure these operations take O(E alpha(E, V)) time total where alpha(E, V) is something called the inverse Ackermann function. It grows incredibly slowly. Like, for any graph you might possibly work with, alpha(E, V) will be no more than 4. You'd need more edges than there are stars in the universe to make it bigger.
- E alpha(E, V) = o(E log V), so the time to sort dominates. The total running time is O(E log V) just like before.

Prim-Jarník Algorithm

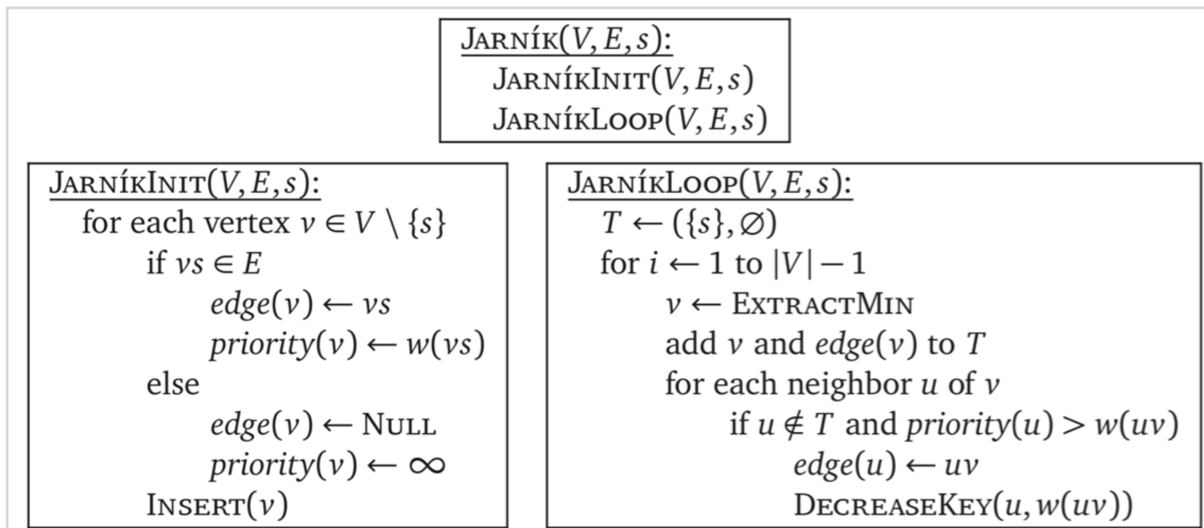
- Found by Jarník in 1929. Prim found it 1957, and somehow he won the naming game.
- In this algorithm, F always has one component T that is allowed to have edges, and the rest are isolated vertices. Initially T is just an arbitrary vertex.
- Jarník: Repeatedly add T's safe edge to T.



- One way to implement this algorithm is to keep a priority queue of edges incident to T. We

pull the edge out of the queue, and if it goes to a vertex outside of T, we add it to T.

- So it basically looks like BFS, except we use a priority queue instead of a normal first-in-first-out queue.
- We'd do one min-heap operation for each edge in O(log E) = O(log V) time per operation, so the algorithm runs in O(E log V) time.
- But a faster way of writing the algorithm is to use a priory queue of *vertices*.
- For each vertex v outside T, we keep the weight of the lightest edge from T to v or infinity if we haven't found such an edge yet. We also remember which edge to v is lightest.
- When we extract the minimum vertex from the priority queue, its edge to T must be the safe edge.
- **[time permitting, write your own version that sets everything to infinity except priority(s) = 0 and edge(s) = emptyset. then it's more clear that we're using an adjacency list]**

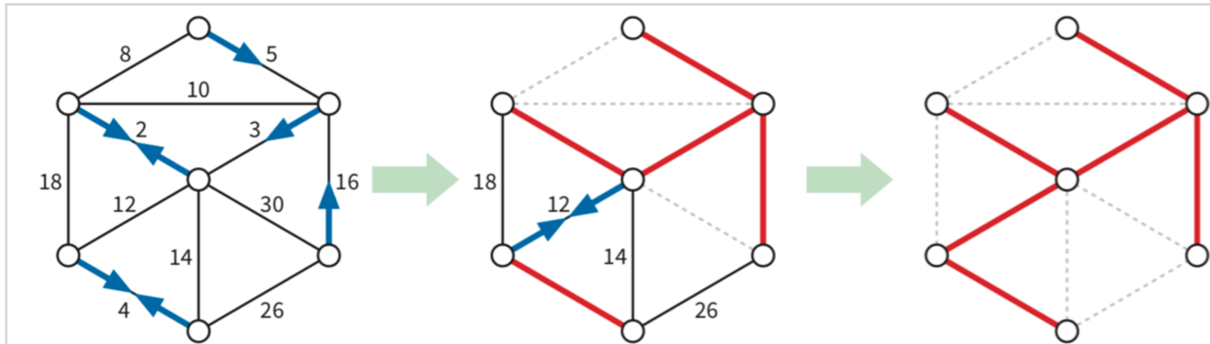


- There are O(E) DecreaseKey operations, O(V) Inserts, and O(V) ExtractMins.
- Using a normal binary heap for the priority queue, operations takes O(log V) time each, so the algorithm still takes O(E log V) time total.
- But, we can get fancy and use something called a Fibonacci heap instead. These handle DecreaseKeys in O(E) time on average across all the operations, so the whole thing takes O(E + V log V) time instead.
- But this is probably slower in practice unless you have very large dense graphs.

Borůvka's Algorithm

- Not that we'll get to this, but here's the algorithm nobody teaches even if it's actually better than the other ones.
- Found by Borůvka in 1926. As often happens, many others rediscovered it including George Sollin in the 1960s. It was credited to him in a textbook, and now some people call it Sollin's algorithm.

- Bor˚uvka: Add ALL the safe edges and recurse.



- Erickson describes a method for finding and adding all the safe edges during a single iteration in O(E) time. It involves running several WhateverFirstSearch's to label which component each vertex belongs to.
- In the worst case, each iteration merely combines pairs of components, reducing the total number of components by a factor of 2. So there are O(log V) iterations.
- The total running time is O(E log V).
- The original descriptions of this algorithm were too complicated, so nobody really took it seriously and it rarely appears in textbooks. But it has a lot of nice features.
    - That O(log V) is a worst-case upper bound, and the algorithm may run much faster in practice. For some classes of graphs, like those that can be drawn in the plane without edge crossings, you can implement this algorithm so it runs in O(E) time.
    - This algorithm is really easy to parallelize since you can search for the safe edge of each component in a separate thread.
    - So if you need to implement minimum spanning trees, use this algorithm.
    - I teach the others mostly so we can practice thinking about and proving things about minimum spanning trees.
    - Also, because you'll get strange looks if you claim to know about minimum spanning trees but not the other two algorithms.