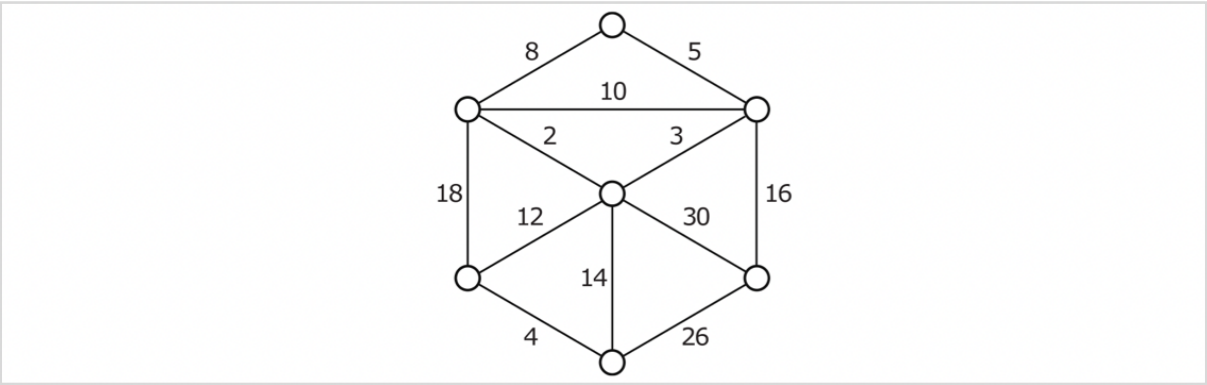
CS 6363.003.21S Lecture 14—March 25, 2021

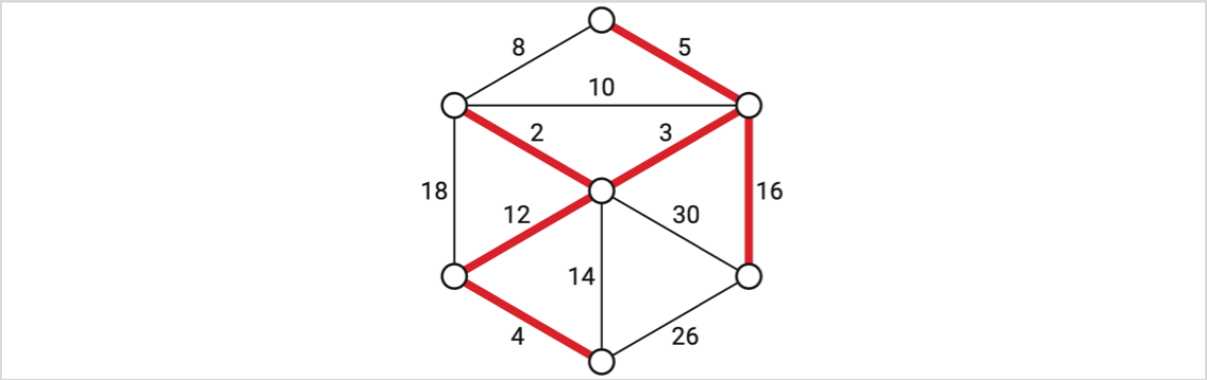
Main topics are #minimum_spanning_trees.

Minimum Spanning Trees

- Time to move on from graph searches and instead look for interesting graph structures.
- 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.
- 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. Our options are shown as black line segments.



- Our goal here is to connect the stations together in the cheapest way possible. The best way for this network is shown with the thick red line segments.



- 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. Any real weight is fine, even negative ones.
- We want to find a *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,

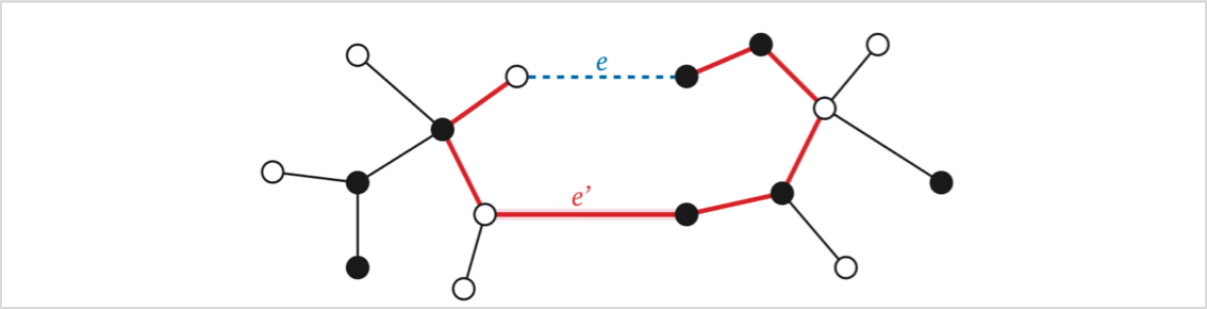
all spanning trees have V - 1 edges, so if the weights are all 1, then *every* spanning tree is also a *minimum* spanning tree.

- Erickson describes a way to break ties so that if the procedure claims w(a) < w(b) and w(b) < w(c), it will also claim w(a) < w(c).
- Because there is a way to compare edge weights that guarantees distinct edge weights, I'll just keep things simple by assuming distinct edge weights for the rest of the lecture.

The Only Minimum Spanning Tree Algorithm

- There is really only one minimum spanning tree algorithm (outside of modern research at least), 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.
- Let T be the minimum spanning tree we're trying to find. The idea is we're going to be adding edges bit-by-bit to build T.
- Suppose we pause partway through the process. We have an acyclic subgraph F ⊆ T we'll call the *intermediate spanning forest*.
- We'll maintain the invariant (always true statement) that F ⊆ T at all times.
- 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.
- Let's rephrase what this algorithm does recursively: We have a subset of edges F that belong to the minimum spanning tree. We need to pick one or more edges to add to F, creating F', and then recursively compute a minimum weight spanning tree T ⊇ F'.
- But which edges are we going to add to F to create 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.
- 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 that is the minimum weight edge with exactly one endpoint in that component.
- So, that's one safe edge per component. Note however that some edges uv may be the safe edge for the components of both u and v.
- Some edges are neither useless nor safe for this particular forest F. But as long as F ≠ T, we'll have at least one safe edge available. Eventually, we'll have F = T, and all edges outside of T become useless.
- Claim (Prim '57): The minimum spanning tree T contains every safe edge. In fact, for *any* subset of vertices S, tree T contains the minimum-weight edge e with one endpoint in S.
  - We're claiming something about a greedy choice, so let's use an exchange argument.

- Suppose to the contrary that 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.

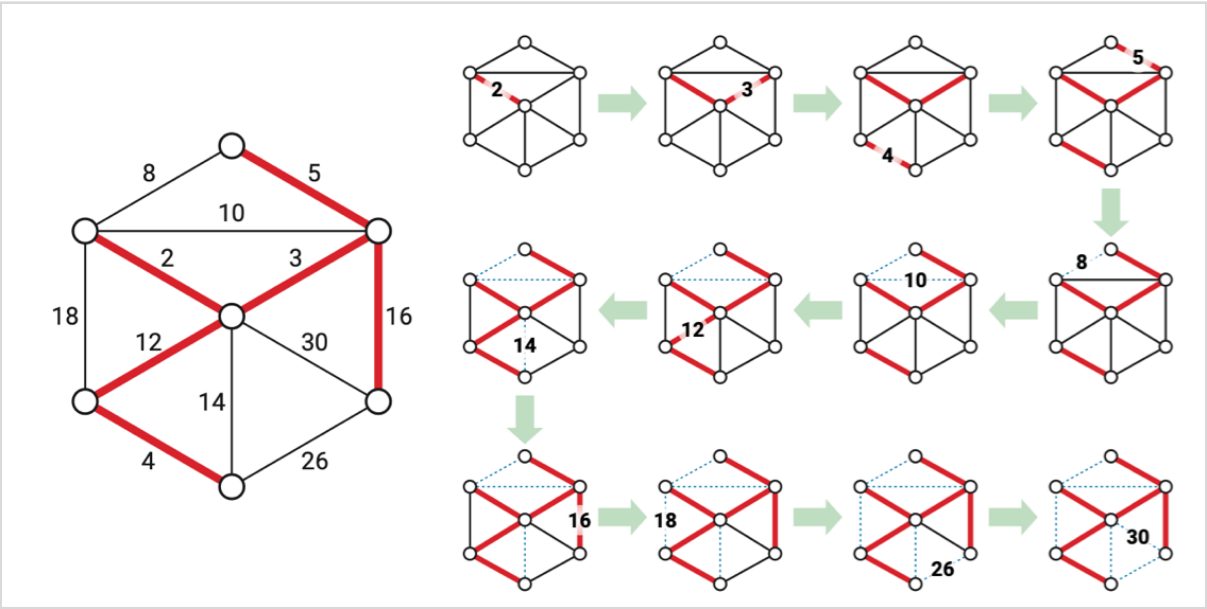


- T is acyclic, so if we remove e', we create a spanning forest with two components.
- Each component contains an endpoint of e. Otherwise, the path we were talking about would not cross to the other component, meaning it wouldn't have contained e'.
- So that means we can add e in to get a new spanning tree T' = T - e' + e.
- But w(e) < w(e'), so w(T') < w(T). Tree T was not minimal after all!

- Note how the lemma kind of forces our hand for what edges to pick if edge weights are unique. That's most of an argument for the minimum spanning tree being unique in this case. Erickson has a more detailed argument for the claim.
- So, we must avoid useless edges, and we must include every safe edge.
- That implies the following greedy algorithm: add one or more safe edges to the evolving forest F, and recurse.
- And each time we add new edges to F, some undecided edges become safe or useless.
- We just need to figure out which safe edges to add in each iteration, and how to identify new safe and new useless edges.

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.



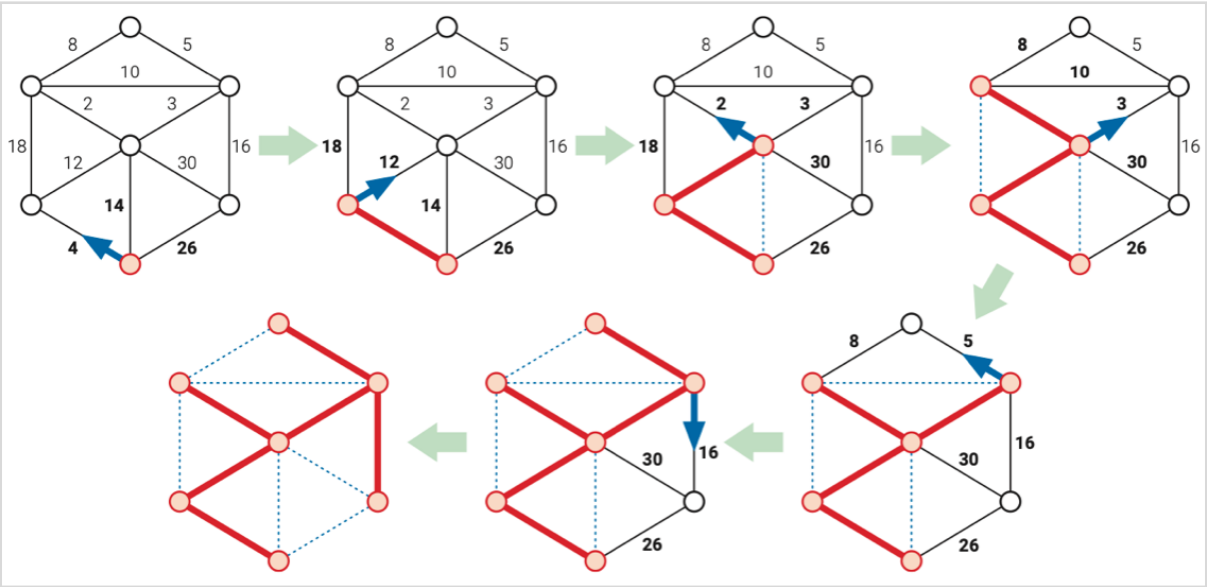
- Claim: Immediately after scanning any edge e and maybe putting it in the tree, every edge of weight ≤ w(e) is either in F or is useless.
  - Assume inductively the claim has remained true up to the moment we scan an edge e. All all edges of weight ≤ w(e) outside F will remain useless whether or not we add e to F. Now scan e.
  - If e is useless, it's just another edge outside F that is useless. Good.
  - Suppose e isn't useless. It must be the lightest non-useless edge outside the tree. Meaning it's the lightest edge spanning two components, so it's safe for both and we add e to F.
- To implement the algorithm, we use something called a disjoint sets data structure that supports the following operations:
  - MakeSet(v) creates a set containing only the vertex v.
  - Find(v) returns a unique identifier for the set containing v. If u and v are in the same set, then Find(u) = Find(v). Otherwise, Find(u) ≠ Find(v).
  - Union(u, v) replaces the sets containing u and v with the union of the two sets.
- For each component, we'll record the set of vertices within the component. So initially, we call MakeSet on each vertex to represent all the single-vertex components.
- For each edge in increasing order of weight, we'll check if its two endpoints lie in the same component. If they do, we discard the edge. Otherwise, we add the edge to F and Union the two component sets to represent the new bigger component.

```
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^2) = 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.
- One way to implement this data structure is to explicitly maintain labels on each individual vertex stating which component they belong to. Find takes O(1) time since we can just lookup the label.
- When we call Union(u, v), we traverse the *smaller* of u and v's components, relabeling all the vertices with the label of the larger component in time proportional to the number of vertices in the smaller component.
- Each time we change a vertex's label, it's component at least doubles in size. So a vertex has its label changed O(log V) times. The total time for all Union operations is O(V log V). The O(E log E) = O(E log V) time needed to sort the edges is larger.
- A more traditional description of the algorithm uses a black-box disjoint set data structure with better performance.
- The best disjoint set data structure spends O(alpha(V)) time per Find or Union. alpha is something called the inverse Ackermann function. It grows incredibly slowly. Like, for any graph you might possibly work with, alpha(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(V)= o(E log V), so the time to sort still dominates. The total running time is O(E log V) just like before.

Prim-Jarník Algorithm

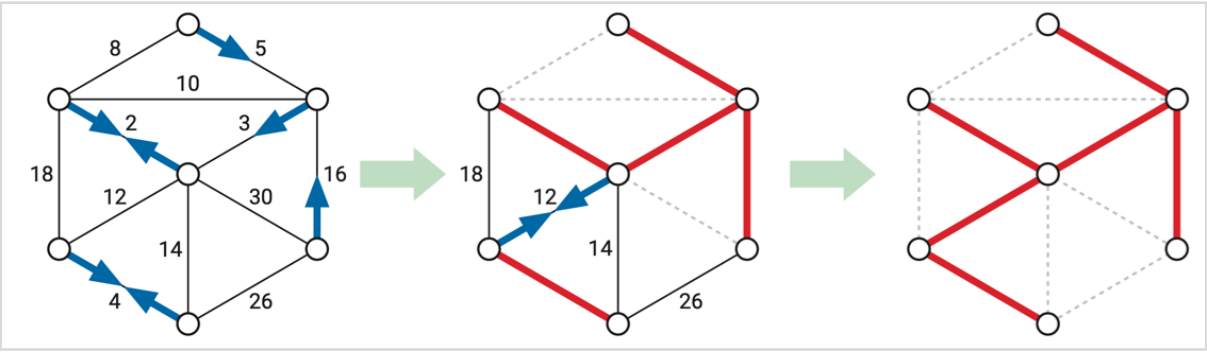
- Found by Jarník in 1929. Prim found it 1957, and somehow he won the naming game.
- In this algorithm, F always has one component T that is allowed to have edges, and the rest are isolated vertices. Initially T is just an arbitrary vertex.
- Jarník: Repeatedly add T's safe edge to T.



- One way to implement this algorithm is to keep a priority queue of edges incident to T. Each time we add a vertex to T, we put its incident edges in the queue. When we pull an edge out of the queue, we check if one endpoint is outside T (say, by marking vertices added to T) and add the edge to T if so.
- So it's a whatever first search using a priority queue as the data structure of edges we want to traverse.
- 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.
- We'll discuss a faster implementation later after go over Dijkstra's algorithm for shortest paths.

Bor˚ovka's Algorithm

- Time permitting, let's discuss one more strategy for choosing safe edges today. The other two are classic strategies that good to know if you ever want to prove things about MSTs. This next one is the one you *should* use if you have to implement an MST algorithm in practice.
- It was found by Bor˚uvka in 1926. As often happens, many others rediscovered it including George Sollin in the 1961. 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.



- So in our example, we only go through two iterations/levels of recursion.
- How do we implement it? One can modify any basic graph search algorithm like DFSAll to count components and label vertices by component number in time linear in the graph size.
- Specifically, we can count and label components of F so all vertices in the first component have label 1, all in the second have label 2, and so on.
- If F has one component, we're done.
- Otherwise, we'll compute an array S[1 .. count] of safe edges where S[i] is the minimum weight edge with one endpoint in component i.
- To compute S we'll loop through all the edges, comparing the edge to the one stored for the label of its endpoints. In the pseudocode below, comp(u) is the component label assigned to u during the last call to CountAndLabel.

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      safe[i] ← NULL
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = NULL or w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = NULL or w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

- CountAndLabel takes O(V) time, because F has at most V - 1 edges.
- AddAllSafeEdges takes O(E) time and $V \leq E + 1$, so the while loop takes O(E) time per iteration.

- 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. Again, this is the MST algorithm you want to use in practice.
    - That O(log V) is a *worst-case* upper bound on number of iterations, and the algorithm may use fewer iterations in practice.
    - In fact, you can implement this algorithm so if you happen to be given certain nice types of graphs as input, like those that can be drawn in the plane without edge crossings, it ends up running in O(E) time. The O(E log V) bound continues to hold even if you're given worst-case inputs.
    - This algorithm is really easy to parallelize since you can search for the safe edge of each component in a separate thread.
    - There are even faster MST algorithms than the ones we went over in class. They're all based on Bor˚uvka's algorithm.
- So if you need to implement minimum spanning trees, use Bor˚uvka's.