CS 4349 Lecture—October 18th, 2017

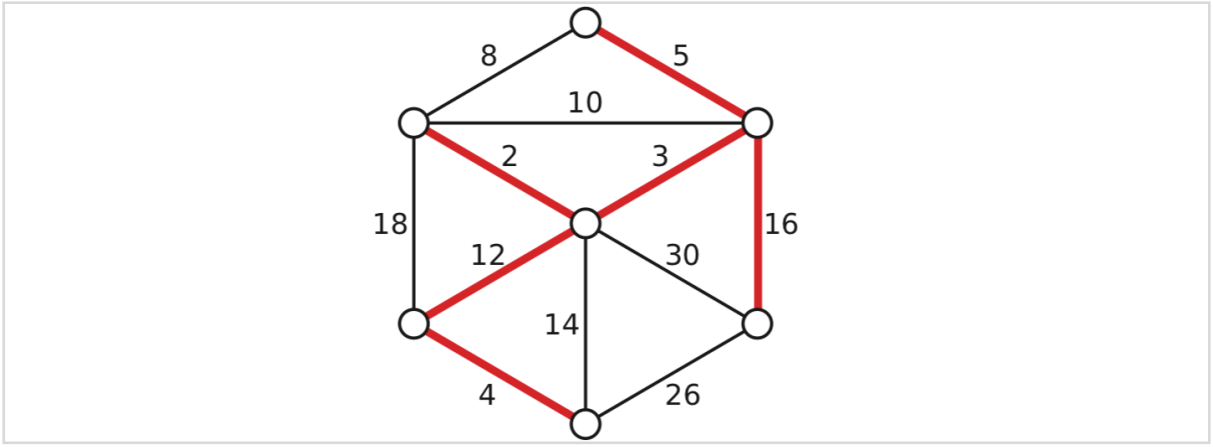
Main topics for #lecture include #minimum_spanning_trees.

Prelude

- Homework 6 due today.
- Homework 7 due Wednesday, October 25th.
- Homework 7 has one normal homework problem. It also has two extra credit problems. If you get 10s on both of them, that's worth 1/4 of the midterm or 7.5% of your final grade. It may not sound like much, but it should be more than enough to boost you at least one third of a letter grade.
- They're extra credit, so they don't effect the curve. If you're satisfied with your grade, you can just pretend they aren't there.
- They're the same difficulty as an exam problem, but you get the whole week. Because you have the time and as a check that you understand the solution you write, I'm going to grade those extra credit problems the same as I do any other extra credit problem. Part of the score will go toward a proof a correctness, although these proofs should be relatively light. See the exam solutions to understand the level of detail I'm expecting. I expect you *not* to look toward outside sources for help.
- I read your feedback. Some people suggested more interactive or entertaining examples like the Towers of Hanoi. My son doesn't have toys appropriate for every lecture, but I'll try my best to help you stay focused. We're getting to more visually interesting graph algorithms now, so that might help.
- Another comment I saw was that the extra credit wasn't worth enough points to bother with. I have to disagree. Two students actually had their midterm grade go up a third of a letter, because they did the two extra credit problems from the homework, and I'm trying to balance it so a good amount of extra credit across the semester will be worth about that much for final grades.

Minimum Spanning Tree

- Alright, we're going to continue graph algorithms this week, but hopefully the lecture will be a little less dry.
- So consider this problem: Let's say we have a bunch of computers that we want to connect in a network.
- We can represent the computers as circles on the board.



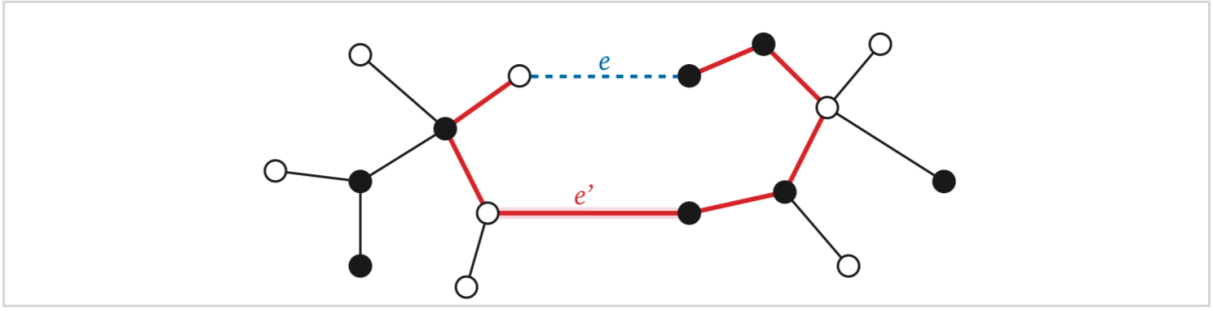
- We can connect some pairs of computers together using a single link, but each choice of link has a different cost.
- And our goal here is to connect the computers together in the cheapest way possible so you can get from any one computer to another just by following the links we choose.
- So I claim this is the best way. I'll explain how I know that as today's lecture.
- Computers are still pretty dry, so maybe we could talk about a different problem instead. Suppose we want to build a sewer system.
- The sewer has all these junction points and you can build lines between some pairs of them, but each line has a different cost.
- What's the cheapest set of lines to guarantee the sewer is connected?
- OK, so these are the same problem.

- 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.
- 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 probably won't prove that fact today.
- 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 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

- Last week, we talked about the one and only graph traversal algorithm. Depending on how we implemented it, we got all the variants you normally think of like depth-first search and breadth-first search.

- The same idea applies to minimum spanning trees. There is really only one minimum spanning tree algorithm (at least for this class), and all need to do to get your name attached to something is to figure out a fast way to implement it.
- 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.
- We have this acyclic subgraph F we'll call the *intermediate spanning forest*.
- We're only adding edges, so F is a subgraph of the minimum spanning tree.
- And that means every component of F is a minimum spanning tree of its vertices.
- In particular, 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**
- Lemma: The minimum spanning tree contains no useless edge.
  - If we added a useless edge to F, it would create a cycle!
- Each component is associated with one *safe* edge, the minimum weight edge with one endpoint in that component. Different components may or may not have different safe edges.
- Some edges are neither useless nor safe, at least for this forest F.
- Lemma: The minimum spanning tree contains every safe edge.
- Proof (of the day): We'll prove something stronger. For *any* subset of vertices S, the minimum spanning tree contains the minimum-weight edge e with exactly one endpoint in S.
  - So a couple weeks ago, we talked about greedy algorithms and exchange arguments. We'll use one of those here.
  - 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.

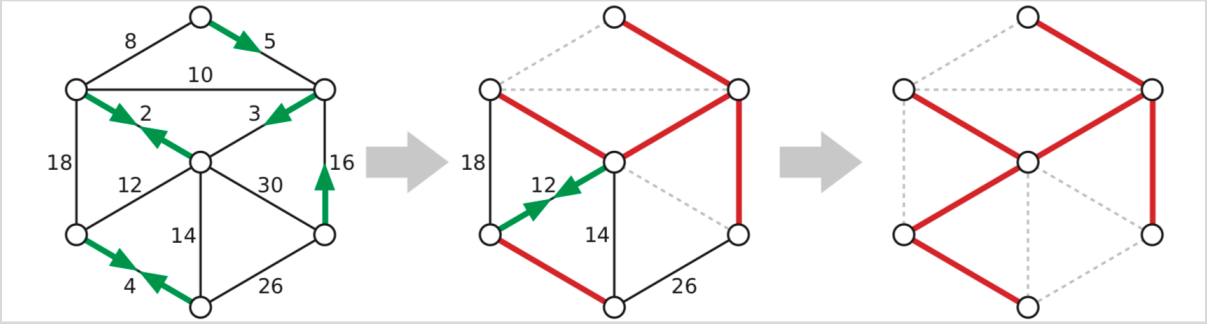


  - T is acyclic, so if we remove e', we create a spanning forest with two components.

    - 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 want to include every safe edge.
- Here's our algorithm: repeatedly add one or more safe edges to the evolving forest F.
- As we add new edges to F, some undecided edges become safe, and some edges become useless.
- We need to figure out which safe edges to add in each iteration, and how to identify new safe and new useless edges.
- Alright, so now it's your turn. I'll pick a couple of you and ask for a safe edge. Don't worry about what the other person is thinking about.
- Right, it didn't matter which safe edge you chose or even if you chose multiple safe edges at once.
- But we should probably write an algorithm for all graphs. I have some examples in mind, but we can go in any order. If you had to write an algorithm, what kind of safe edges would you choose?

Borvka's Algorithm

- Found by Borvka in 1926. As often happens, many others rediscovered it including Sollin in the 1960s, and now some people call it Sollin's algorithm.
- Borvka: Add ALL the safe edges and recurse.



- How do we implement it?
- Count and label components of F so all vertices in the first component have label 1, all in the second have label 2, …. I wrote an O(E) time algorithm for this last Wednesday.
- If F has one component, we're done.
- Otherwise, we'll compute an array S[1 .. V] of safe edges where S[i] is the minimum weight edge with one endpoint in component i (or NULL if component i does not exist).
- To compute S we'll loop through all the edges, comparing the edge to the one stored for the label of its endpoints.

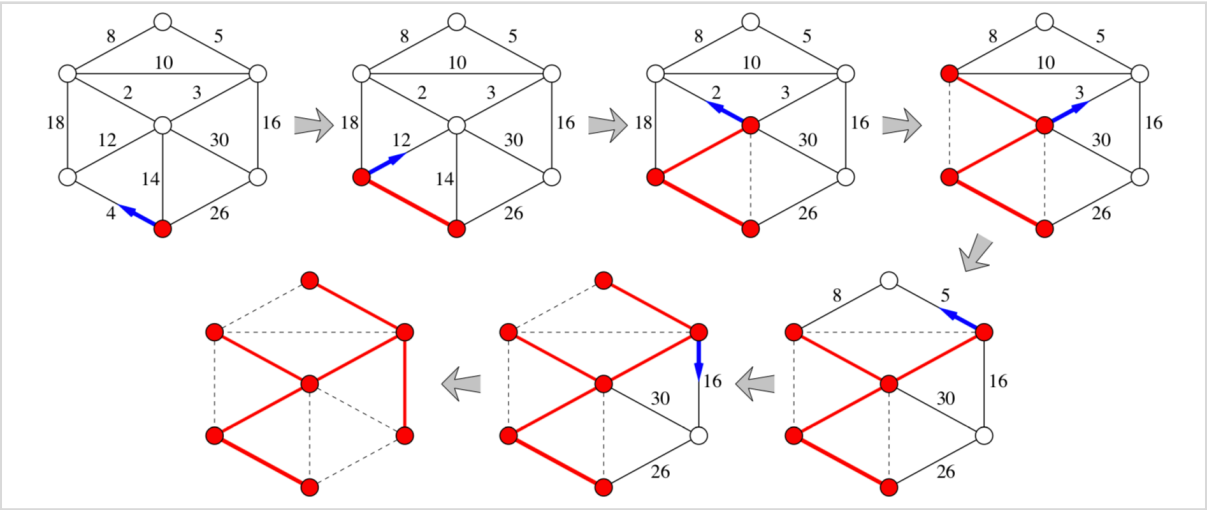
```
BORVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      S[i] ← NULL        ⟨⟨sentinel: w(NULL) := ∞⟩⟩
  for each edge uv ∈ E
      if label(u) ≠ label(v)
          if w(uv) < w(S[label(u)])
              S[label(u)] ← uv
          if w(uv) < w(S[label(v)])
              S[label(v)] ← uv
  for i ← 1 to count
      if S[i] ≠ NULL
          add S[i] to F
```

- CountAndLabel takes O(V) time, because F has at most V - 1 edges.
- AddAllSafeEdges takes O(E) time and V ≤ E + 1, so the while loop takes O(E) time per iteration.
- In the worst case, each iteration 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.

Jarník's ("Prim's") Algorithm

- Found by Jarník in 1929. Prim found it 1957, and somehow he won the naming game.
- In this algorithm, F always has one non-trivial component T, and the rest are isolated vertices.
- Jarník: Repeatedly add T's safe edge to T.



- We keep all edges adjacent to T in a priority queue.
- When we pull out an edge, we check if both endpoints are in T or not.
- If both endpoints are in T, we throw the edge away, because it is useless.
- Otherwise, the edge is safe for T and we add it.
- This is really just that third example from last week of using a priority queue in the generic graph traversal algorithm.
- There are O(E) priority queue operations taking O(log E) time each, but E = O(V^2) so the algorithm runs in O(E log V) time.
- You've seen the pseudocode already, so I'll spare you!

Kruskal's Algorithm

- Found by Kruskal in 1956. Oh, this one got the right name!
- Kruskal: Scan all edges in increasing weight order; if an edge is safe, add it to F.
- Since we're examining edges in increasing order of weight, an edge is safe if and only if its endpoints are in different components of F.
- If they were in the same component, well by definition that edge is useless.
- And if we find an edge between different components, it must be the cheapest such edge. Otherwise, the real safe edge for one of those components would have been found earlier and declared safe or useless.
- To implement the algorithm, we use something called a Union-Find data structure.
- 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 a function that grows incredibly slowly. Like, for any graph you might possibly work with, alpha(E, V) will be no more than 5. But you can pick a number of vertices large enough to make that value 6 or even **gasp** 7.
- 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.