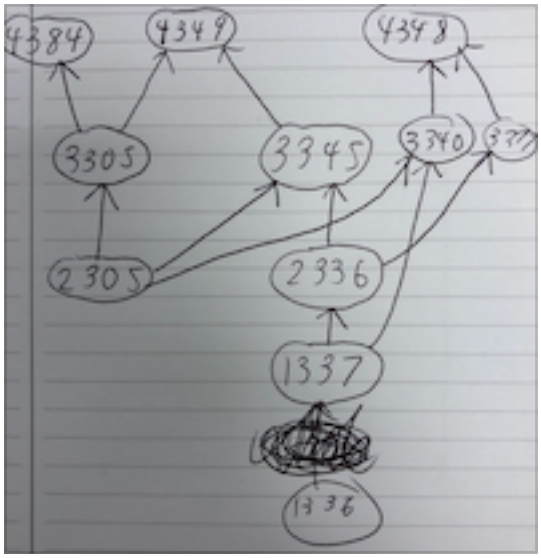
CS 4349.003.19F Lecture 15—October 16, 2019

Main topics for #lecture include #topological_sort and #minimum_spanning_trees.

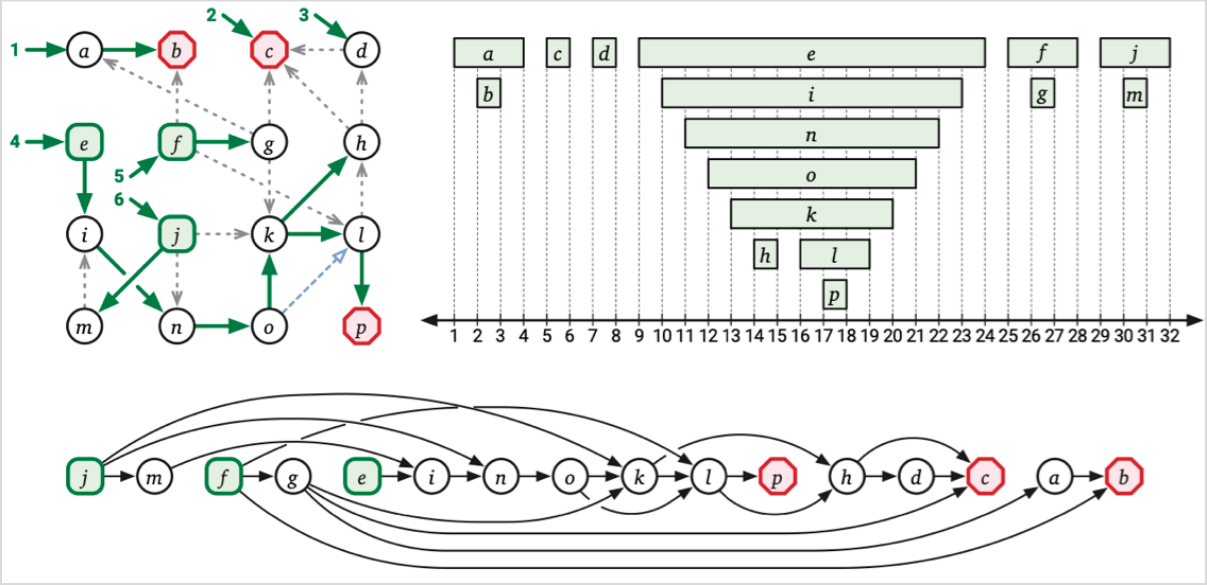
Topological Sort



- Does anybody recognize this graph? It's a subset of the required CS classes. There's an edge from class u to class v if u is a prerequisite for v.
- All prerequisite graphs must be directed acyclic graphs or DAGs. In other words, they cannot contain directed cycles.
- After all, to take a course on a directed cycle, we'd have to first take all the courses before it, and the courses before them, and so on. We'd eventually see that we need to take our first course before we can take our first course. That doesn't work too well.
- On Monday, we discussed a method for detecting directed cycles in a given a directed graph G.
- We proved G has a cycle if and only if DFSAll(G) yields a back edge.
- Edge u ➜ v is a back edge if and only if u.post < v.post, so here's an algorithm for detecting cycles:
  - Call DFSAll(G) to compute a post ordering in O(V + E) time.
  - For each edge u ➜ v
    - If u.post < v.post
      - Return "Cycle!"
  - Return "No cycle."
- But now suppose we want to compute what order should we take the classes in.
- What we want is a *topological ordering* of the vertices. Formally, it's a total order where u < v if there is an edge u ➜ v. Less formally, we want to draw the vertices on a line going left to right so there are no edges directed from right to left.
- Again, topological orderings don't exist if there are directed cycles: in any ordering the

rightmost vertex of a cycle would have an edge going back to the left.

- But what if we have no directed cycles, meaning we have a *directed acyclic graph* (DAG)?
- G being a DAG implies there are no back edges after a call to DFSAll, meaning u.post > v.post for *every* edge u ➔ v.
- So, going by *decreasing* u.post, or reverse post ordering, you get a topological ordering!



- In particular, *every* directed acyclic graph has a topological ordering.
- If we want to put the vertices in a separate data structure in order, we can add them in reverse postorder by having a clock tick *down* from V to 1.

```
TopologicalSort(G):
  for all vertices v
      v.status ← New
  clock ← V
  for all vertices v
      if v.status = New
          clock ← TopSortDFS(v, clock)
  return S[1..V]
```

```
TopSortDFS(v, clock):
  v.status ← Active
  for each edge v→w
      if w.status = New
          clock ← TopSortDFS(v, clock)
      else if w.status = Active
          fail gracefully
  v.status ← Finished
  S[clock] ← v
  clock ← clock − 1
  return clock
```

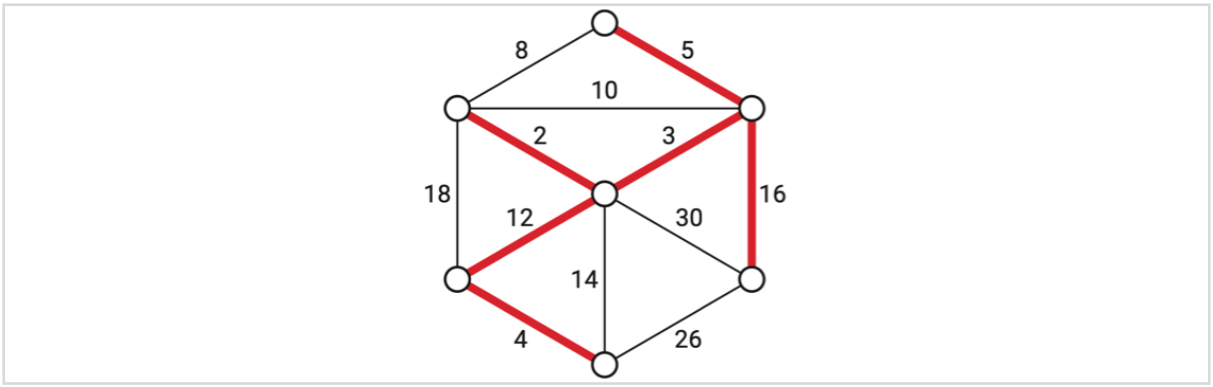
- Again, it's just DFSAll with different constant time pre and postprocessing routines, so O(V + E) time total.
- There are many more applications of depth-first search including using it as a different way to think about or even implement dynamic programming algorithms. Unfortunately, we don't have time this semester to get into it. See Erickson 6 or CLRS Chapter 22 if you're interested.

Minimum Spanning Trees

- But for the rest of today and next Monday, let's move away from graph searching and

discuss a different problem.

- 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 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).
- Since there is a way to compare edge weights that guarantees uniqueness, I'll just assume unique edge weights for the rest of the lecture and that there is a single minimum spanning tree.

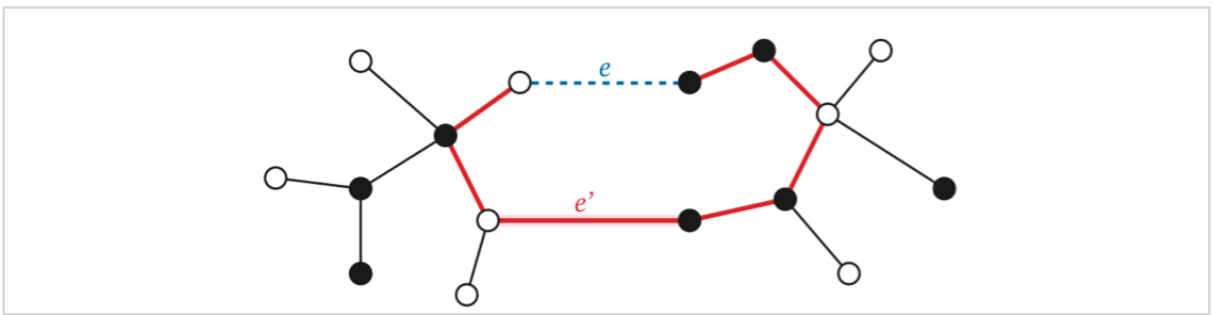
The Only Minimum Spanning Tree Algorithm

- 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'll maintain the invariant (always true statement) that F is 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.
- 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 for which F' is a subgraph.
- 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.

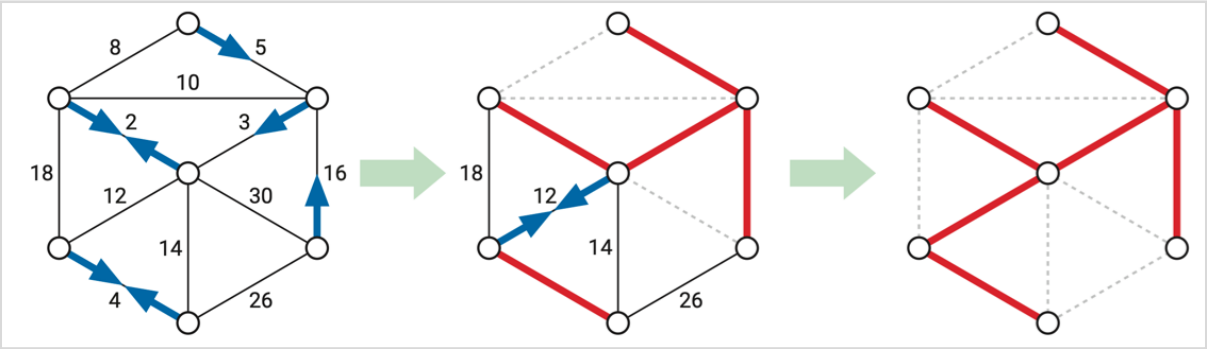
  [draw arrow to a useless edge]
- Claim: The minimum spanning tree contains no useless edge.
  - If we added a useless edge to F, it would create a cycle!
- For each component of F, we'll associate a *safe* edge as the minimum weight edge with exactly one endpoint in that component.
- So, that's one safe 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 (Prim '57): The minimum spanning tree contains every safe edge. In fact, for any subset of vertices S, the minimum spanning tree 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 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, 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 this new spanning tree has smaller total weight. T must not have been the minimum spanning tree.
- 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 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.
- 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, and we have an algorithm.

Bor˚ovka's Algorithm

- We'll discuss just one strategy for choosing safe edges today. This 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? Well, the time we spent last week counting and labeling components is finally about to pay off.
- Count and label components of F so all vertices in the first component have label 1, all in the second have label 2, …
- 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 (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. 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 ≤ 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. This is the MST algorithm you want to use in practice.
    - That O(log V) is a *worst-case* upper bound, and the algorithm may run much faster in practice. Sometimes you'll get lucky and you'll only need to do a couple iterations like in the example.
    - 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 even in the worst case.
    - 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'll go over in class. They're all

based on Bor˚uvka's algorithm.

- So if you need to implement minimum spanning trees, use this algorithm.
- On Monday, I'll go over two more algorithms for this problem, because those two algorithms are useful for teaching you to think about and prove things about minimum spanning trees.
- Also, because you'll get strange looks if you tell somebody you know about minimum spanning trees but not those two algorithms.