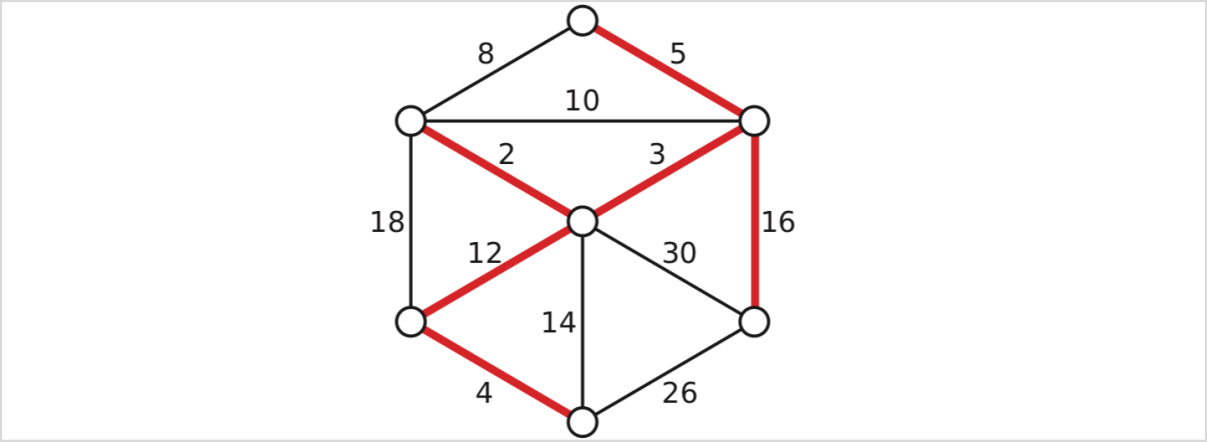
CS 4349 Lecture—October 23rd, 2017

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

Prelude

- Homework 7 due Wednesday, October 25th. Don't forget about the extra credit.

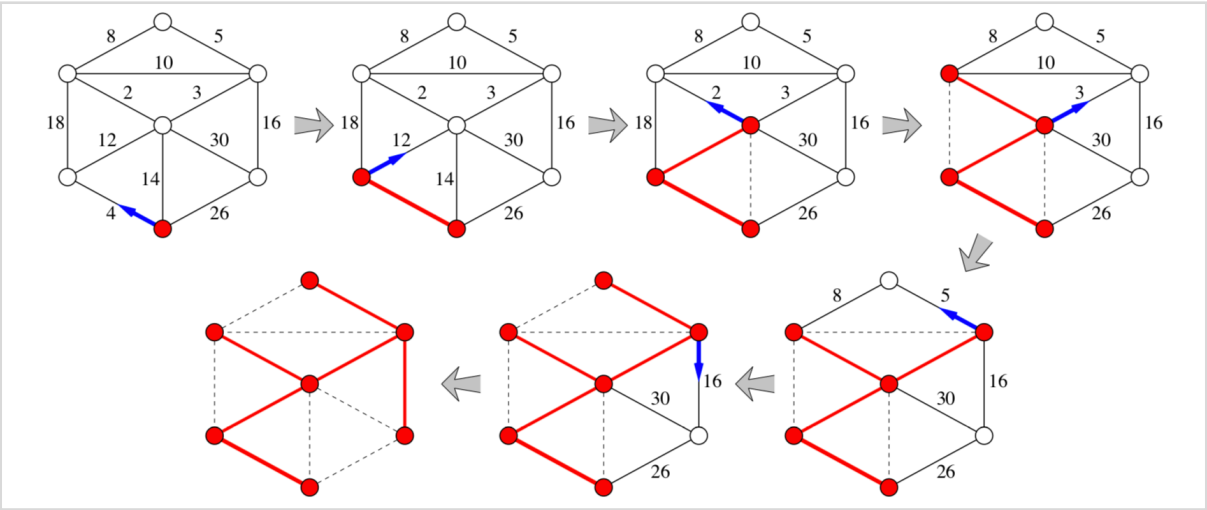
Minimum Spanning Tree Continued



- Wednesday we were discussing minimum spanning trees; given a connected, undirected graph G = (V, E) with distinct edge weights, find the spanning tree T that minimizes w(T) = sum_{e in T} w(e).
- There was only one real algorithm for this problem. Iteratively add edges to the intermediate spanning forest F.
- *Useless* edges are outside of F, but both endpoints are in the same component of F. These don't belong to the minimum spanning tree.
- Each component is associated with one *safe* edge, the minimum weight edge with one endpoint in that component. Safe edges do belong to the minimum spanning tree.
- So, we repeatedly add one or more safe edges to the evolving forest F.
- I discussed two algorithms based on this approach.
- In Kruskal's algorithm, we add the lightest safe edge and recurse.
- In Borvka's algorithm, we add every safe edge we can find all at once and recurse.
- There's one more algorithm I'd like to discuss.

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.
- Another (more common) 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 that is.

```
JarníkInit(V, E, s):
  for each vertex v ∈ V \ {s}
      if (v, s) ∈ E
          edge(v) ← (v, s)
          key(v) ← w(v, s)
      else
          edge(v) ← Null
          key(v) ← ∞
      Insert(v)
```

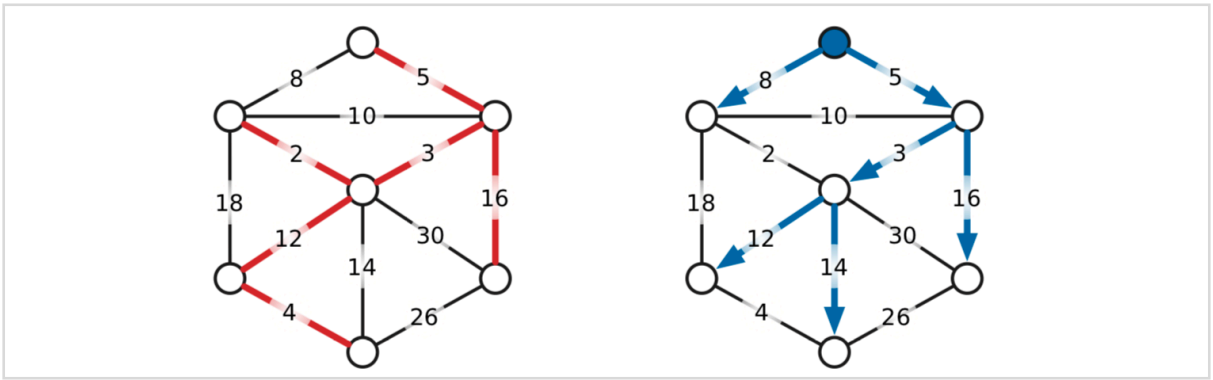
```
JarníkLoop(V, E, s):
  T ← ({s}, ∅)
  for i ← 1 to |V| − 1
      v ← ExtractMin
      add v and edge(v) to T
      for each neighbor u of v
          if u ∉ T and key(u) > w(uv)
              edge(u) ← uv
              DecreaseKey(u, w(uv))
```

- There are O(E) DescreaseKey operations, O(V) Inserts, and O(V) ExtractMins taking O(log E) time each if we use a binary heap, but E = O(V^2) so the algorithm runs in O(E log V) time.
- We can get fancy and use something called a Fibonacci heap instead. These handle DecreaseKeys in O(E) time on average, so the whole thing takes O(E + V log V) instead.
- But this is probably slower in practice unless you have very large dense graphs.

Shortest Paths

- For the rest of today, let's consider a different problem.
- Let's say we're given a map of the country, and you want to figure out how to drive from your home to Austin. How do you figure out the quickest route?
- In graph language, we're given a weighted *directed* graph G = (V, E, w) with two special vertices.
- We want to compute the shortest path from *source* vertex s to *target* vertex t. In other words, we want to find a directed path p from s to t minimizing w(p) := sum_{u → v in p} w(u → v).
- The algorithms we'll talk about today and Wednesday really are used in mapping software from Google or Microsoft.
- I'm going to add one twist to the problem you may not have seen before. We'll allow some of the weights on edges to be *negative*. In terms of routing from point a to point b, a negative edge might represent some benefit you get from taking that edge. Maybe there's some really pretty scenery along that part of the route.
- Negative edges are an issue if there is a directed cycle with negative weight. Because if such a cycle existed, the cheapest way to get from s to t would be to walk from s to the cycle and then go around it over and over and over again, decreasing the total weight with each pass. So, our algorithms will only be correct if there are no negative cycles.
- We can solve shortest paths in undirected graphs by replacing every undirected edge with two directed edges of the same weight going opposite directions. BUT! this won't work with negative weight edges since those two directed edges would form a tiny negative cycle.

- OK, so let's say you want to solve this problem. It turns out almost every algorithm for computing a shortest path from s to t actually computes the shortest path from s to *every other vertex* in the graph.
- This is called the single source shortest path or SSSP problem, and you usually end up computing a shortest path tree rooted at s that contains all the shortest paths from s.

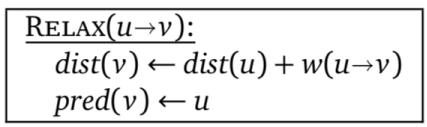


- I want to emphasize that this is a very different problem from minimum spanning tree.
- Shortest path trees are rooted and directed. Minimum spanning trees are unrooted and undirected.

- Shortest path trees are most naturally defined in directed graphs. Minimum spanning trees are only defined for undirected graphs.
- And if edge weights are distinct, the minimum spanning tree is unique. There is a different shortest path tree rooted at every vertex in the graph, and they may *all* be different from the minimum spanning tree.

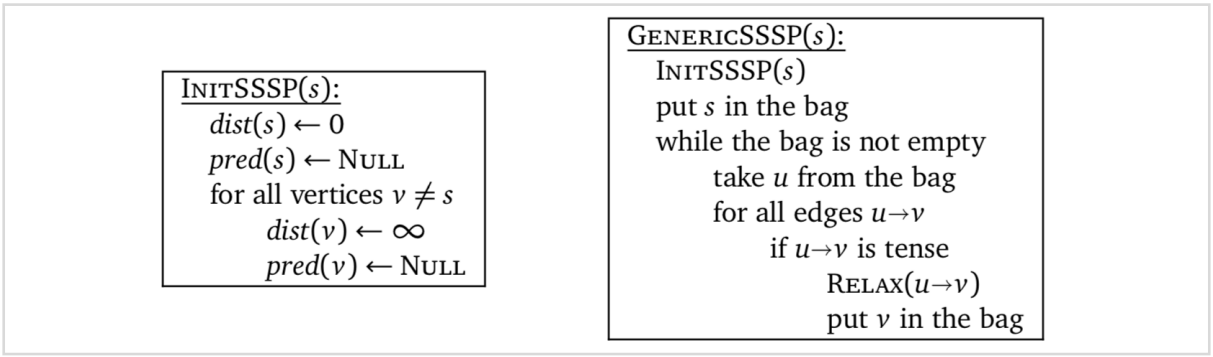
The Only SSSP Algorithm

- Just like graph traversal and minimum spanning tree, there's really only one shortest path tree algorithm independently discovered by Lester Ford and George Dantzig around the same time. You just need to find a good implementation to get your name attached to it.
- The idea is that we'll keep an educated guess on the distance and shortest path to each vertex.
- dist(v) is the length of a tentative shortest s to v path, or infinity if we haven't found one yet.
- pred(v) is the predecessor of v in the tentative shortest s to v path, or Null if we haven't found one yet.
- At the beginning of the algorithm, we know dist(s) = 0 and pred(s) = Null. For every other vertex v ≠ s, we initially set dist(v) = infinity and pred(v) = Null, because we haven't found *any* paths to those vertices yet!
- Call an edge u ➔ v *tense* if dist(u) + w(u ➔ v) < dist(v).
- If an edge is tense, then the path s - - > v is clearly incorrect, because s - - > u ➔ v is shorter.
- We want to *relax* tense edges to represent our newly found shorter path.



- The only SSSP algorithm repeatedly finds a tense edge and relaxes it.
- So why does it work? Let's freeze the algorithm and some point halfway through it execution and take a peak.
  1. For every vertex v, dist(v) is either infinity or the length of some walk from s to v.
     - Can be proved using induction on the number of relaxations.
  2. If there are no negative cycles, then dist(v) is either infinity or the length of some *simple path* from s to v.
     - If the walk from s to v has a cycle, then the cycle must have negative weight. The reason being that the last edge added to the cycle somehow made the distance to its head even smaller than when we first started following the cycle.
     - **So** if there are no negative cycles, then the algorithm eventually halts because there are only a finite number of simple paths in G.
  3. If no edge is tense, then for every vertex v, dist(v) is the length of the path s ➔ … ➔ pred(pred(v) ➔ pred(v) ➔ v.

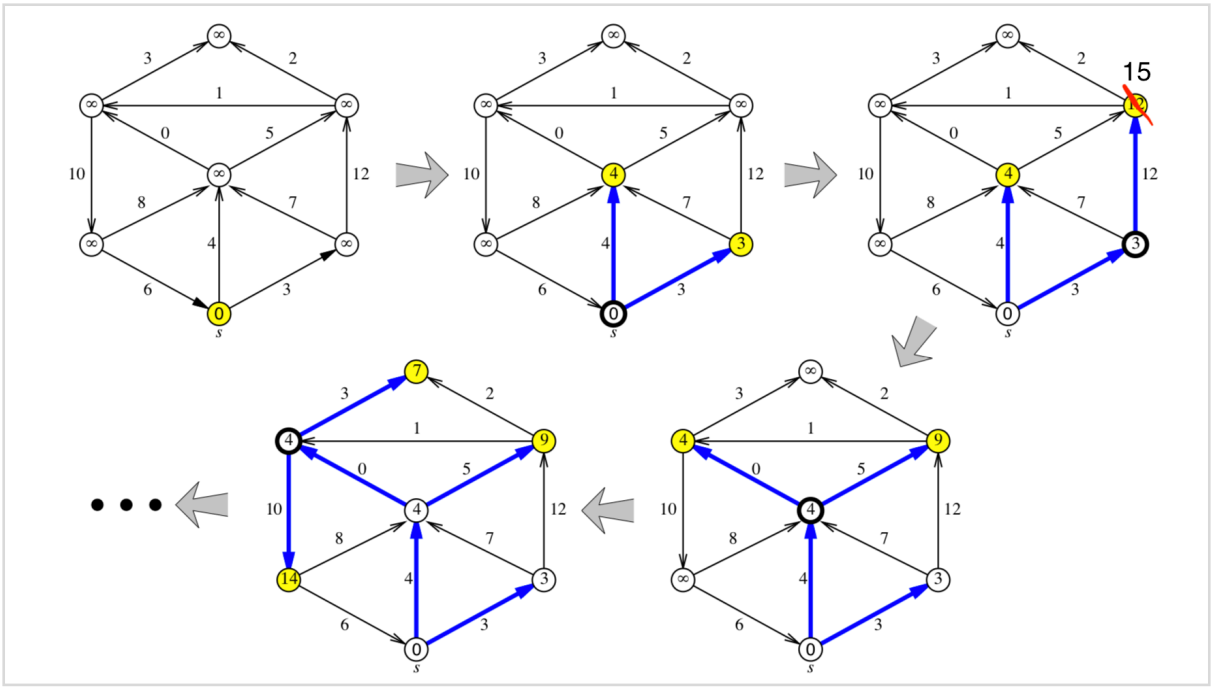
    - If v violates the condition but pred(v) does not, then pred(v) ➔ v is tense and we need to update the distance.
4. If no edge is tense, then for every vertex v, s ➔ … pred(pred(v)) ➔ pred(v) ➔ v is a shortest path from s to v.
    - If v violates the condition but u from a *shortest path* containing u ➔ v does not, then u ➔ v is tense.
    - **So** if G has a negative weight cycle, then some edge is always tense and the algorithm never halts.
- Once again, it's up to us to decide what order to relax tense edges.
- But one thing to note. If an edge u ➔ v is not tense, then the only way to make it tense it to reduce dist(u).
- So we'll do something similar to traversals. We'll keep a bag of vertices initially holding s.
- When we pull vertex u out of the bag, we'll look for tense edges u ➔ v, relaxing each and putting v into the bag.
- We *do not* mark vertices. The same vertex could be visited many times, and the same edge could be relaxed many times.



- The data structure we use for the bag determines the algorithm we're running.
- The obvious choices are stacks, queues, and priority queues, but it turns out stacks can cause Theta(2^V) relaxation steps, so we won't even talk about them today.

Dijkstra's Algorithm

- Let's start with a priority queue. This algorithm was privately discovered by a team of researchers in 1957, but never publicly published. George Dantzig published the algorithm in 1958, and Edsger Dijkstra did the same in 1959. Dantzig will get his chance to shine in a later lecture, and I don't want you to get strange looks for the rest of your careers, so let's call this Dijkstra's algorithm.
- Here's the first few steps of Dijkstra's algorithm:



DijkstraSSSP(V, E, s):

- InitSSSP(s)
- Insert(s, 0)
- while the priority queue is not empty
  - u ← ExtractMin
  - for all edges u → v
    - if u → v is tense
      - Relax(u → v)
      - if v is in the priority queue
        - DecreaseKey(v, dist(v))
      - else
        - Insert(v, dist(v))

- If the edges have *non-negative weights*, then dist(u) is accurate when u is removed from the bag. Also, vertices are removed in increasing order of distance from s.
  - This can be proved by induction. When it comes times to remove vertex u, vertices of lessor distance were already removed and we knew their distance. So then there's no more relaxations that put those vertices back in the bag or reduce u's distance, so u must have the correct distance.
- This means each edge is relaxed at most once.
- There are O(E) DecreaseKeys, O(V) Inserts, and O(V) ExtractMins, so the total running time is O(E log V) if we use a regular binary heap.
- If we use a Fibonacci heap, then DecreaseKeys in O(E) time on average, so the whole thing takes O(E + V log V) instead.
- This analysis assumed non-negative edge weights. If some edges have negative weight, then the algorithm could take exponential time before it terminates.

- But, Dijkstra's algorithm is usually fast even with some negative weight edges.