CS 4349.003.19F Lecture 16—October 16, 2019

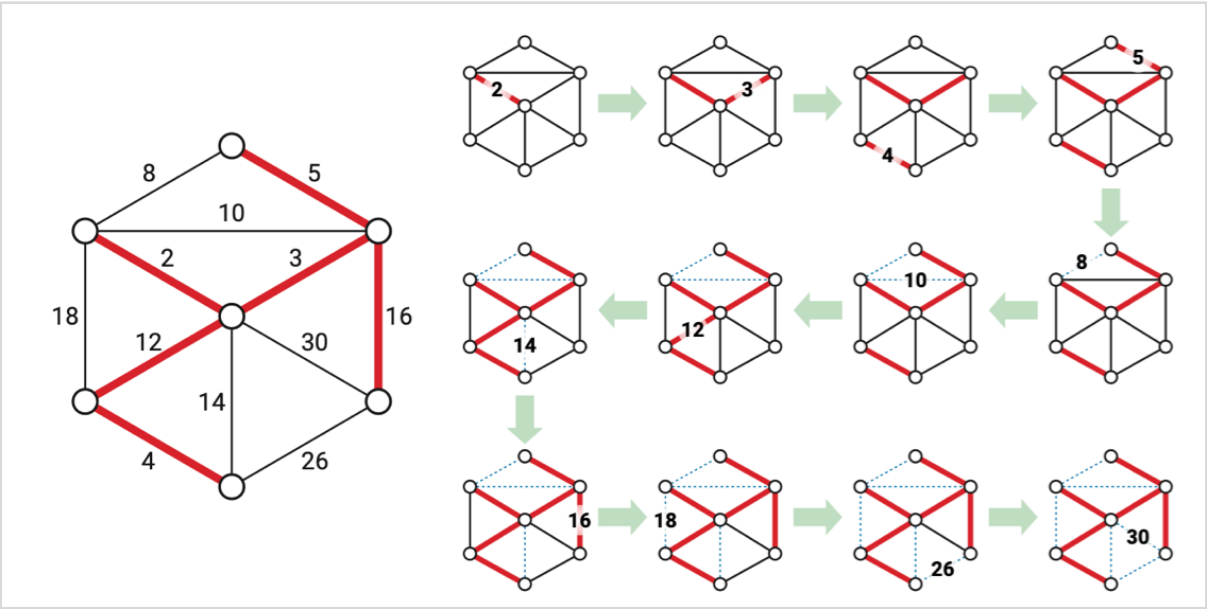
Main topics for #lecture include #minimum_spanning_trees.

Minimum Spanning Trees Redux

- Last time, we were discussing minimum spanning trees.
- 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).
- We discussed a single greedy algorithm for this problem based on iteratively adding edges to an intermediate spanning forest F.
- F is a subgraph of the minimum spanning tree.
- F consists of n one-node trees before we've added any edges.
- *Useless* edges are outside of F, but both endpoints are in the same component of F. No useless edge appears in the minimum spanning tree.
- For each component of F, we'll associate a *safe* edge as the minimum weight edge with exactly one endpoint in that component. *Every* safe edge belongs to the minimum spanning tree.
- That implies the following greedy algorithm: add one or more safe edges to the evolving forest F, and recurse.
- But which ones do we add?

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.



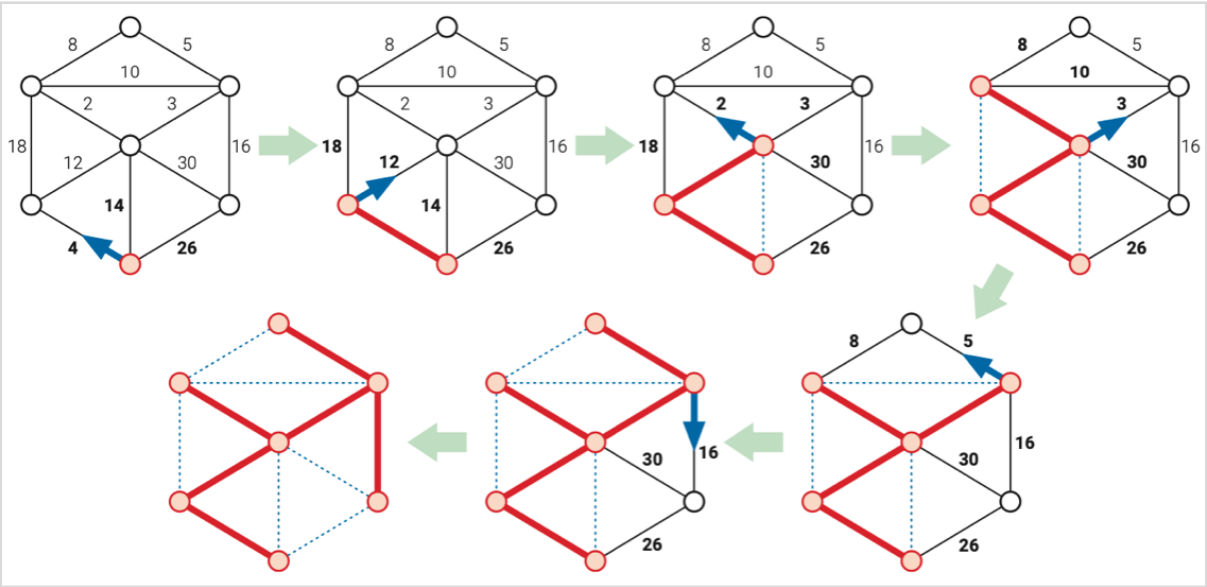
- Claim: Immediately after scanning edge e and maybe putting it in the tree, every edge of weight ≤ w(e) is either in F or is useless.
  - Assume inductively the claim is true for lighter edges than e.
  - Now consider when we scan edge e.
  - If e is useless, then the lemma follow immediately.
  - Or, 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 we correctly 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) 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. 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's a whatever first search using a priority queue for the bag of edges.
- 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.

```
JARNÍK(V, E, s):
  JARNÍKINIT(V, E, s)
  JARNÍKLOOP(V, E, s)
```

```
JARNÍKINIT(V, E, s):
  for each vertex v ∈ V \ {s}
      if vs ∈ E
          edge(v) ← vs
          priority(v) ← w(vs)
      else
          edge(v) ← NULL
          priority(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 priority(u) > w(uv)
              edge(u) ← uv
              DECREASEKEY(u, w(uv))
```

- (This pseudocode appears to use an adjacency matrix, but we can use an adjacency list by precomputing which vertices are adjacent to s in O(V) time.)
- 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 a different priority queue called a Fibonacci heap instead.

- If we do I Inserts, D DecreaseKeys and X ExtractMins, a Fibonacci heap over n items uses O(I + D + X log n) time total.
- So the whole thing here takes O(E + V log V) time.
- But this is probably slower in practice unless you have very large dense graphs. The hidden constant in the big-Oh for Fibonacci heaps is pretty big. The hidden constant in the big-Oh for Bor˚ovka's algorithm is 1. Again, you probably want to use Bor˚ovka's algorithm.
- If you need to compute a minimum spanning tree in a homework or on an exam, using an O(E log V) time algorithm is fine for full credit.