MST

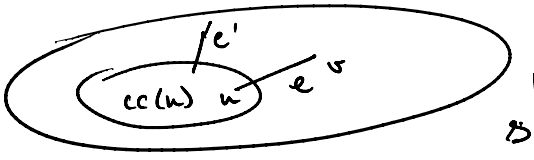
Tuesday, September 22, 2015
7:57 PM

- We ended the last lecture on DP with the Floyd-Warshall algorithm for all pairs shortest path (APSP). These classical problems on graphs occur in a wide variety of applications
  - By the way, for APSP, current best algorithm is $O\left(\frac{n^3}{2^{\sqrt{\log n}}}\right)$ by quite sophisticated techniques!
- In next two lectures, we look at other classical computational problems on graphs.

Minimum Spanning Tree

- Given undirected, edge-weighted graph G, a subgraph F is a spanning tree if it is an acyclic graph with $n-1$ edges (i.e., on all the n vertices of G)
  - Throughout, n = # of vertices of G
    m = # of edges of G
- For simplicity, assume that all edge weights distinct. This implies a unique MST.
  - Question: Why?
- All our MST algorithms use the same "meta-algorithm". This meta-algorithm constructs a spanning tree F incrementally
- Initially, F is the empty graph on n vertices
- As F is updated, it satisfies two invariants:
  1. F is a subgraph of the MST
  2. Each connected component of F is an MST on its vertices

- To understand which edges we can include in T without violating invariants, we make 2 definitions
- Defn: Suppose an edge $e = (u, v)$ is not in F. If both u and v are in the same cc of F, then e is useless. If e is the min wt edge among all edges touching cc(u) or all edges touching cc(v), then it's safe.
- Claim: Adding a useless edge to F violates invariants.
  Pf: It creates a cycle.
- Claim: A safe edge can be added to F without violating invariants.
- Pf: Suppose wlog that e is min wt among all edges touching cc(u), but MST contains some other edge e'. So, wt(e') > wt(e).



But then delete e' from MST and add e to get spanning tree of smaller wt!

- Now, we need to decide which safe edges to put in at every step.

~~Boruvka's algorithm~~

- Discovered in 1926!
- Idea: Add all safe edges to F at every step.
- Claim: After $O(\log n)$ steps, F becomes a spanning tree.
  Pf: At each step, # of cc's reduces by factor of at least 2. (Note: In practice, convergence is even faster!)

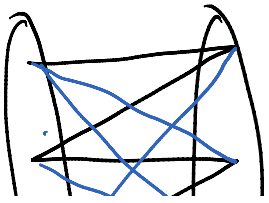
- Claim: Each step can be implemented in $O(m)$ steps, label each vertex with an id of the cc it belongs to.

Pf: First in $O(m)$ steps, label each vertex with an id of the cc it belongs to.
Then, find the min wt edge touching each cc by scanning through all edges (for every edge (u,v), record if it's the min wt among all edges touching $cc(u)$ or $cc(v)$.)

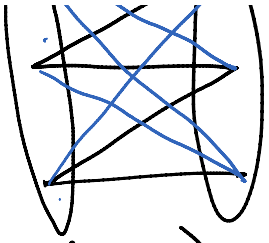
- Total time complexity: $O(m \log n)$

Prim's algorithm

- Instead of adding many safe edges at once, let's add one safe edge at a time by growing one component.
- Idea: Grow a tree by adding one safe edge at a time to F.
- We'll keep a priority queue of edges with one endpoint in the tree, ordered by their wt. At each iteration, we extract the min wt edge e in the priority queue, add e to F, and insert into the priority queue all non-useless edges that have one endpoint in common with e.
- The priority queue can be implemented by a min-heap, with $O(\log n)$ insertions and extraction of min element.
- To test whether an edge is useless or not, we can mark all vertices in the tree (using $O(n)$ space).
- Minimum is extracted $O(n)$ times but there can be $O(m)$ insertions.



Black edges wt: 0
Blue edges wt: 100

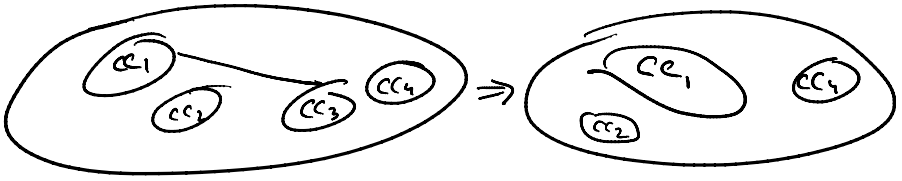


Black edges wt : 0
Blue edges wt : 100

- Time complexity : $O(m \log n)$
- Can be improved using Fibonacci Heaps. Will see later.

Kruskal's algorithm

- Idea: Again add one edge in each step. Sort the edges in increasing order of wt and add the first safe edge to F.
- Note that after sorting, the first edge that has its endpoints in different components of F must be safe.
- So, each newly added edge merges previously separated cc's.



- Need to maintain partition into disjoint sets, while allowing union operations

- KRUSKAL (V, E):
  sort E by increasing wt
  $F \leftarrow (V, \emptyset)$
  for each $v \in V$:
    MakeSet (v)
  for i = 1 to m:
    let $e_i = (u, v)$ be i'th edge in E
    if Find(u) ≠ Find(v):

Union (u, v)
Add e_i to F

return F

- In above code:
  - MakeSet, Find and Union are operations of a ~~Union-Find data structure~~ that maintains a collection of disjoint sets
  - MakeSet (x) creates new set $\{x\}$
  - Find (x) returns an id of the set containing x.
  - Union (x, y) replaces the set containing x and the set containing y by their union.
- Cost of Kruskal is $O(m \log n) + m \cdot (\text{Cost (Find)} + \text{Cost (Union)})$
  ↑ sorting
- We next show a Union-Find data structure where Find and Union take $O(\log n)$ time. So, Total complexity is $O(m \log n)$ with this implementation of Kruskal.

Union-Find

- Idea: Represent a set by a tree of height $O(\log n)$ and let its id be the root.

- Example:

Union (a, b)

Union (c, d)

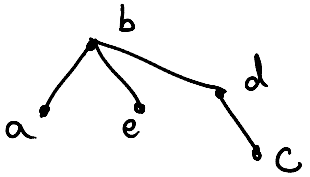
(Put tree of smaller height as child

Union (a, e)

Union (a, c)

(height as child of root of larger height tree)

Union (a, c)



(If trees of same height, then increment the height of the new root by 1)

Formally:

```
MakeSet(u):
    π(u) = u
    height(u) = 0

Find(u):
    while π(u) ≠ u:
        u = π(u)

Union(u, v):
    r1 = Find(u)
    r2 = Find(v)
    if height(r1) < height(r2):
        π(r1) = r2
    else:
        π(r2) = r1
        if height(r1) = height(r2):
            height(r1) += 1
```

Claim: Any root node of height k has $\geq 2^k$ nodes in its tree.

Pf: A root node of height k is formed by taking

To union of two trees of height $k-1$.

Corollary: Height of any root node is $\leq \log_2 n$.
Hence, Find and Union take $O(\log n)$ time.