CS 161: Design and Analysis of Algorithms
Greedy Algorithms 2: Minimum Spanning Trees

- Definition
- The cut property
- Prim’s Algorithm
- Kruskal’s Algorithm
- Disjoint Sets
Tree

- A tree is a connected graph with no cycles
Minimum Spanning Tree

- Given a weighted undirected graph $G=(V,E)$ with non-negative weights, find a set of edges that connects the graph with the least total weight.
Minimum Spanning Tree

• Given a weighted undirected graph $G=(V,E)$ with non-negative weights, find a set of edges that connects the graph with the least total weight.
Minimum Spanning Tree

• Uses
  – Networks: If we want to connect a collection of computers by directly connecting pairs, and each connection has a cost, what is the least-cost way of achieving this goal?
  – Approximation: Used in approximation algorithm for Traveling Salesman Problem
Properties

• Why a tree?

• Property 1: Removing an edge in a cycle will not disconnect a graph
  – If two nodes were connected through the removed edge, still connected by going other direction around cycle.
Properties

- Property 2: A tree on $|V|$ nodes has $|E| = |V| - 1$
  - Build tree one edge at a time. Start with one node, no edges, and repeatedly add one edge, one node until tree is constructed.
Properties

• Property 3: Any connected undirected graph with $|E| = |V|-1$ is a tree
  – If not a tree, there is a cycle. Remove any edge on the cycle. By property 1, still connected.
  – Continue until no cycles. The result is a tree, so $|E'| = |V|-1 = |E|$. Thus, no edges were actually removed
Properties

• Property 4: An undirected graph is a tree if and only if there is a unique path between any pair of nodes
  – Trees are connected, so at least 1 path. If two paths, union would contain cycle
  – If not a tree, either disconnected or contains cycle. If disconnected, there are two nodes with no paths. If cycle, any two nodes on cycle have two paths
First Attempt: Unweighted Graphs

• If all weights = 1 (equivalent to unweighted graph), any tree is an MST
• Do DFS, marking tree edges
• $O(|V|+|E|)$
Second Attempt: Weighted Graphs

- Replace each edge with weight $w$ with $w$ edges and $w-1$ nodes
Second Attempt: Weighted Graphs

- Replace each edge with weight $w$ with $w$ edges and $w-1$ nodes
Second Attempt: Weighted Graphs

- Let W be total edge weight
- $|E'| = |E| + W$
- $|V'| = |V| + W - |E|$
- $O(|V'| + |E'|) = O(|V| + W)$
- Inefficient if W large
Properties

• A cut of a graph $G=(V,E)$ is a pair $(S,V-S)$

• Property 5: The Cut Property
  – Let $X$ be a set of edges that are part of some MST of $G=(V,E)$, and $(S,V-S)$ be a cut that $X$ does not cross. Let $e$ be the lightest edge across this cut. Then $X \cup \{e\}$ is part of some MST
Proof of Cut Property

• X is part of some MST T
• If e is part of T, then done
• Otherwise, add e to T
• Since T was a tree, adding e creates a cycle.
• Since e crosses the cut, there must be another edge e’ on the cycle that crosses the cut
Proof of Cut Property

$S$ $V-S$
Proof of Cut Property

• Let $T'$ be the graph obtained by adding $e$ to $T$ and removing $e'$

• $T'$ is a tree:
  – Since $T$ was connected, and $e'$ was a cycle edge, $T'$ still connected
  – $T'$ has $|E'| = |V| - 1$, so by Property 3, must be a tree
Proof of Cut Property

• \( \text{weight}(T') = \text{weight}(T) + \text{weight}(e) - \text{weight}(e') \)
• Since \( T \) was an MST, must have \( \text{weight}(T') \geq \text{weight}(T) \)
• This means \( \text{weight}(e) \geq \text{weight}(e') \)
• But \( e \) was chosen to be lightest edge across cut
  – \( \text{weight}(e) \leq \text{weight}(e') \)
• Thus \( \text{weight}(e) = \text{weight}(e') \), and \( \text{weight}(T') = \text{weight}(T) \)
• \( T' \) is an MST
Idea

• From any partial solution $X$ to MST problem with $k$ edges, can get solution $X'$ with $k+1$ edges as follows:
  – Pick cut that $X$ does not cross
  – Let $e$ be lightest edge across cut
  – Add $e$ to $X$
Third Attempt: Prim’s Algorithm

- X is always a tree (never disconnected)
- Cut (S,V-S): S are nodes connected by X
- For each u not in S, must keep track of
  - cost(u): weight of lightest edge from u into S
  - prev(u): the corresponding node in S
- Repeatedly find u with lowest cost(u), and add edge (prev(u),u)
Third Attempt: Prim’s Algorithm

• How to keep track of cost and prev?
• If set correctly before adding edge (prev(u),u):
  – For every neighbor v not in S, cost(v) only changes if weight(u,v) < cost(v), so update accordingly
  – If v is not a neighbor, cost(v) does not change
Third Attempt: Prim’s Algorithm

• How to find lightest node across edge?
• Heap with cost values as keys
Third Attempt: Prim’s Algorithm

• Pick some initial node v
• Set cost(v) = 0, cost(u) = ∞ for u ≠ v
• Create heap q with all nodes, ordered by cost
• While q is not empty
  – u = q.delete_min()
  – cost(u) = 0
  – For each (u,w) in E, if cost(w) > weight(u,w):
    • prev(w) = u
    • cost(w) = weight(u,w)
• Output edges (prev(u),u) for all u ≠ v
Running Time

• Same as Dijkstra’s algorithm!
• $O((|E|+|V|) \log |V|)$ for heaps
• $O(|V| \log |V| + |E|)$ for Fibonacci Heaps
• Is it possible to do any better?
Fourth Attempt: Kruskal’s Algorithm

• Repeatedly add lightest edge that does not create cycle

• Same as adding lightest edge across any cut that partial solution does not cross:
  – If adding $e = (u,v)$ does not create cycle, then $u$ and $v$ cannot be connected
    • Let $S$ be component containing $u$, $V-S$ everything else
  – If adding $(u,v)$ does create a cycle, $u$ and $v$ must have already been connected
    • Any cut separating $u$ and $v$ must be crossed by partial solution
Fourth Attempt: Kruskal’s Algorithm

• How to find lightest edge that doesn’t create cycle:
  – First sort them by weight
  – Have different sets of nodes that represent different components
  – Go through edges, checking if endpoints are in different sets
  – Combine the two sets into one
Disjoint Sets

• Want the following operations:
  – makeset(x): makes a set containing x
  – union(x,y): unions the two sets
  – find(x): returns the set containing x
Idea 1: Linked Lists

- Each set is represented by a linked list. Additionally, we store, for each value x, a pointer set(x) to the list containing that value
- makeset(x) = create new list L containing x, set(x) = L: O(1)
- find(x) = set(x): O(1)
- union(x,y) = link lists together and change set pointers: O(k) where k is the size of the smaller set
Idea 1: Linked Lists

- Claim: Any sequence of k union operations takes $O(k \log k)$ time
  - Proof: Running time constrained by number of updates to set pointers
  - At most 2k nodes affected, so largest set is at most 2k
  - Every time set(x) is updated, the size of the set containing x at least doubles
  - Therefore, number of updates to set(x) is at most $\log 2k$
  - Total number of updates at most $2k \log 2k = O(k \log k)$
Idea 1: Linked Lists

- makeset: $O(1)$
- find: $O(1)$
- union: $O(\log n)$ amortized
Idea 2: Trees

- Each set represented as directed tree
- Set identified by root of tree
Idea 2: Trees

- Each value x has a parent pointer p(x)
- Root of a tree has p(x) = null
- makeset(x) = make new tree with x as root (i.e. set p(x) = null): O(1)
- find(x) = find root of tree: O(h)
  - Let r = x
  - While p(r) ≠ null, let r = p(r)
  - Output r
Idea 2: Trees

• union(x, y) = set root of one tree to be child of the other: $O(h_1 + h_2)$
  – Let $x' = \text{find}(x)$, $y' = \text{find}(y)$
  – $p(x') = y'$ or $p(y') = x'$
Idea 2: Trees
Idea 2: Trees

- makeset(P)
Idea 2: Trees

- makeset(P)
Idea 2: Trees

- find(K)
Idea 2: Trees

- $\text{find}(K)$
Idea 2: Trees

- \text{find}(K)
Idea 2: Trees

- **find(K)**

```
          G
         / \
        F   A
       /   / \
      I   B   \
     /   /   \
    C   M     D
   /   /     / \
  N   E     G   I
 /   /     /   / \
O   E     J   B   \
 /   /     /   /   \
H   B     K   N   P
```
Idea 2: Trees

- Union(K,N)
Idea 2: Trees

- Union(K,N)
Idea 2: Trees

- Union(K,N)
Idea 2: Trees

• Operation times depend on height of tree, so need to keep trees shallow
• When unioning two sets, make the shorter tree the subtree
• Also keep track of rank(x), the height of the subtree at x - 1
Idea 2: Trees

- **makeset(x) = {**
  - Set p(x) = null
  - Set rank(x) = 0

- **find(x) = {**
  - Let r = x
  - While p(r) ≠ null, r = p(r)
  - Return r
Idea 2: Trees

- \( \text{union}(x,y) = \{ O(h_1 + h_2) \) 
  
  - Let \( x' = p(x), y' = p(y) \)
  
  - If \( \text{rank}(x') > \text{rank}(y') \):
    - \( p(y') = x' \)
  
  - Else
    - \( p(x') = y' \)
    - If \( \text{rank}(x') = \text{rank}(y') \): \( \text{rank}(y') = \text{rank}(y') + 1 \)
Idea 2: Trees

• Height of tree?

• rank(x) < rank(p(x))
  – Whenever we set p(x) in union, make sure that rank(x) < rank(p(x))
  – We only change the rank of root nodes, so rank(x) never changes.
  – Ranks can only increase.
Idea 2: Trees

• Height of tree?
• \( \text{rank}(x) < \text{rank}(p(x)) \)
• Any root of rank \( k \) has at least \( 2^k \) descendants
  – True before any unions
  – Assume true before union\((x,y)\). Let \( k_1 \) and \( k_2 \) be the ranks of the root nodes in the two trees
    – Total number of nodes \( \geq 2^{k_1} + 2^{k_2} \)
  – If new node has rank \( k = k_b \) for some \( b \), property holds
  – Otherwise, \( k_1 = k_2, k = k_1 + 1 \), and property holds
Idea 2: Trees

• Height of tree?
• $\text{rank}(x) < \text{rank}(p(x))$
• Any root of rank $k$ has at least $2^k$ descendants
• There are at most $n/2^k$ nodes of rank $k$
  – Any such node has at least $2^k$ descendants
  – No two nodes of rank $k$ can share descendants
Idea 2: Trees

• Height of tree?
• rank(x) < rank(p(x))
• Any root of rank k has at least $2^k$ descendants
• There are at most $n/2^k$ nodes of rank k
• Maximum rank is at most $\log n$
Idea 2: Trees

• Since maximum height of tree is $\log n$, find and union take $O(\log n)$
• Worse than linked lists implementation
• Next time: We will see how to improve this to almost constant time