Minimum Spanning Trees – Contd

A weight of the tree's weight is defined by $w(T) = \sum_{e \in T} w(e)$, but may can define some variation over the set of weights of the edges of the tree, for instance $w(T) = \prod_{e \in T} w(e)$.

In that case, nothing would really change, because for the original graph G we can take the **log** of the graph, say G_{log}. Then: $\min_{T^*} \sum_{w \in T^*} \log w = \min_{T^*} \lg \prod_{w \in T^*} w$.

Union sets:

Is a data structure used for our algorithm to find the MST.

- $make - set(u)$: create a singleton, set of node u that points to itself.
- $Find - set(u)$: returns the node u points to.
- $union(u, v)$: redirect all pointers of the smaller set to the sentinel of the larger set.

For union we saw the amortized running time is $\lg n$ per set.

Kruskal's Algorithm:

Runs in $O\big((m + n)\lg n\big) - m \lg n$ for the edges weight sort, and $n \lg n$ for the n union sets in construction the MST.

Safe cut: A division of V into two sets of nodes $(S, V - S)$ such that no edge of the currently iteration of the MST crosses the cut. The edge that would be safe for the MST would be the lightest edge that crosses the cut.

Say we have a cut $(S, V - S)$, and we have e that is the lightest edge that crosses the cut. Say we take $\hat{e}$ instead of e and we construct $\hat{T}$. We will show that $\hat{T}$ is not a MST.

But, $\hat{T} \cup \{e\}$ will have a cycle. So we can take $\hat{T} \cup \{e\} - \{\hat{e}\}$ and we know $\hat{e}$ is heavier than e, and now we got a tree with a smaller weight, therefore $\hat{T}$ cannot be a MST.

Prim's MST algorithm

As opposed to Kruskal's algorithm, here we construct a tree instead of taking light edges (which are not necessarily connected when we take them).

The algorithm works as follows:

- Start with a root node r.
- At each iteration all nodes that are not in the tree yet will hold the minimum weight to a node that is already in the tree. To know at each iteration which of the outside-nodes is the one with the minimum weight of them all, we hold them in a **min-heap**.
- Each time a new node is inserted to the tree, the min-heap keys should be updated (decrease key).

The total cost of the 2^{nd} phase is $n \lg n - n$ extract-min that each would cost $\lg n$.

The cost of the 3^{rd} phase is $m \lg n$.

The algorithm:

We initialize a priority queue Q, insert r with $key = 0$ to Q, and everybody else with $key = \infty$.

We update the parent of r to be $p[r] \coloneqq nil$.

Then we do the following until the queue is empty:

- Extract-min into u
- For all u's neighbors $v \in adj[u]$ we need to update their minimum weight to get inside the tree, and decrease key accordingly:
  - If $v \in Q$ & $w(u,v) < key[v]$ – this is the condition to update v's key.
  - Then update $p(v) \coloneqq u$ and $Q.decrease - key(v, w(u,v))$

So over all $u \in V$ the decrease-key total cost is: $\sum_{u\in V} \deg(u) \cdot \lg n = \lg n \cdot \underbrace{\sum_{u\in V} \deg(u)}_{=2m} = O(m \lg n)$.

In addition, over all $u \in V$ that we extract from Q, the total cost of extract-min is: $\sum_{u\in V} \lg n = \lg n \sum_{u\in V} 1 = O(n \lg n)$.

⇒ The total cost is like Kruskal's algorithm: $\boxed{O((n+m)\lg n)}$.

<u>Theorem 23.1</u>:

- At each step the cut is $(Q, V - Q)$
- Extract min returns the light edge of the cut.
- $key[v] \coloneqq w(u,v)$ ensures that at the next iteration Q is up to date.
- The algorithm stops when Q is empty – after n iterations.

Prim's advantages:

- Prim's algorithm has a better constant than Kruskal's.
- Prim's algorithm works with negative weights, and Kruskal's can't.

Graph traversal algorithms

Breadth First Search (BFS)

In a given graph $G = (V, E)$ and a starting node s, the algorithm discovers all nodes that are accessible from s, and the shortest distances to those nodes.

The algorithm works in layers (from the source node the all other nodes). The **active vertices**, also denoted the frontier of the search, is the front layer that is tested at the current iteration (starts out with only s, then all nodes connected to s and so on).

<u>Colors</u>:

- **White**: pre visited node.
- **Gray**: visited node, still haven't finished exploring it.
- **Black**: done exploring this node and edges connected to it.

For each node we maintain $p(u), d(u)$ which are the parent node in the path from s and distance from s, respectively.

The predecessor-pointers create an inverted tree, and reversed they construct the **BFS** tree.

The frontier is consisted at each iteration from nodes that are at most 1 level apart from each other.

All edges are:

- Tree edges: when they are part of the BFS tree.

- Back edges: other edges in that connect between two nodes in the tree. Existence of back edge = cycle.

If after Q is empty we still have disconnected nodes, the graph is not connected.

What's the difference between odd and even cycle: if the back edge that closes the cycle is between two different d values, that an odd cycle. Why is that interesting? Graph is bipartite $\Leftrightarrow$ it doesn't have an odd cycle.

Running time:

Every edge is looked once from each end. Coloring would happen 3 times to each node (white, gray, black). The total is: $\boxed{\Theta(m+n)}$.

One use for BFS: calculate the diameter of a tree – run BFS on some starting node, and get the farthest node from it, say u. Then run BFS on u and get the maximum distant v from u - u, v are the two diameter edges.

Depth First Search (DFS)

Assume the graph is directed.

- First time you see a node: color white to grey.
- Second time: color grey to black.

We store $d[v], f[v]$: $d[v]$ is the discovery time of the node, $f[v]$ is the finish time – the time when we visit the node at the second time. There's also a timer that goes along the algorithm and increases by 1 each step.

Types of edges:

- Tree edge
- Back edge
- Forward edge
- Cross edge

Running time: $\Theta(m+n)$.