1 Negative Cycles

A special case with finding shortest paths is the existence of negative cycles: this is a cycle of edges whose total weight is negative.

Because we’re looking for the shortest path, it becomes advantageous to keep on going around such cycles, and take the distances to infinite.

Such a case can be characterized by the Bellman-Ford algorithm always making updates.

We can prove this by contradiction. Suppose no updates are made, but we have a negative cycle with vertices $u_1, u_2 \ldots u_k$. First observe that the algorithm only no longer makes updates when

$$d[u] + l_{u \rightarrow v} \geq d[v],$$

for all edges $uv$. Adding these inequalities over all vertices of a cycle $u_1, u_2 \ldots u_k$ gives:

$$\sum_i d[u_i] + l_{u_i \rightarrow u_{i+1}} \geq d[u_{i+1}],$$

and cancelling the $\sum_i d[u_i]$ terms from both sides then gives

$$\sum_i l_{u_i \rightarrow u_{i+1}} \geq 0,$$

a contradiction.

On the other hand, if $G$ has no negative cycles, then any path can involve at most $n$ vertices: or it has a cycle that we can remove without increasing the length of the path. This plus the claims proven at the end of the previous part gives that the total number of iterations is bounded by $O(n)$, and hence a total of $O(nm)$. 
2 Spanning Trees

Shortest paths naturally lead to trees. If the graph has no negative cycles, the union of all shortest paths starting from $s$ (to all vertices $v$) is a tree.

Formally, a tree is a subgraph meeting any two out of the following three conditions:

1. is connected,
2. $n - 1$ edges,
3. has no cycles

To see that the union of shortest paths form a tree, note that if we take the ‘from’ edge from each vertex (aka. the last edge on their shortest path), we never form a cycle. That is, each node other than $s$ has a unique ‘parent’, for a total of $n - 1$ edges. This plus the fact that the set of all shortest paths connects the graph gives that their union is a tree.

This tree ensures that each vertex has a shortest path to the source. However, it does not ensure that the total cost needed to connect everyone together is minimum. This is because of reuse of edges.

3 Minimum Spanning Trees

We now move onto the problem of finding a subset of edges of minimum total weight to make things connected. This is known as the minimum spanning tree (MST) problem. Note that maximum spanning tree has the same abbreviation, we will use MST to denote minimum spanning tree unless specified otherwise.

For simplicity we assume that all edge weights are distinct. The goal of this lecture is to show that the following greedy algorithm due to Kruskal works.

1. Initialize $H = \emptyset$.
2. Sort edges in increasing order of weights.
3. Loop through edges in increasing order of weights.
   (a) If endpoints of $e$ are not connected in $H$.
      i. Add $e$ to $H$.

The correctness of this algorithm crucially relies on the following fact, known as the cut property. To define this property, we need the definition of a cut. It refers to a set of edges, but is defined by a subset of vertices $S$.

Definition 3.1. A cut given by $S \subseteq V$, $E(S, V \setminus S)$, is the set of edges between $S$ and $V \setminus S$. 
**Theorem 3.2.** For any cut $S$, the minimum weight edge on the cut is in the MST.

The proof relies on the following operation that transforms between trees.

1. Add an edge $e = uv$.
2. Identify the path in $T$ between $u$ and $v$.
3. Remove any edge from this path.

This still gives a tree because the number of edges remains the same, and connectivity is preserved.

*Proof.* The proof is by contradiction.

Suppose $T$ is a minimum spanning tree that does not use $e = uv$. Without loss of generality by symmetry, assume $u \in S$ and $v \notin S$.

Then consider the path in $T$ between $u$ and $v$. This path goes from $S$ to $V \setminus S$, so must contain an edge $e'$ in the cut $E(S, V \setminus S)$.

Since $e$ is the minimum weight edge in $E(S, V \setminus S)$, we have

$$w_{e'} > w_e,$$

so adding $e$ to $T$ and removing $e'$ from the cycle gives a tree with smaller weight, a contradiction.

This gives the correctness of the above algorithm. Implemented directly, we spend $O(m \log n)$ sorting the edges, then $O(n)$ time to check connectivity between vertices in $H$ at each step. As we do this $m$ times, the total cost is $O(nm)$.

The text book also has another property, known as the **cycle property** for certifying when an edge is not in the minimum spanning tree. This rule is fairly simple: edge of (unique) maximum weight in a cycle is not in any minimum spanning tree and is also fairly easy to prove. However, it is sufficiently disjoint from what we’re covering for the rest of this class that it’s not required.