Definitions

Minimum Spanning Tree (MST) For any connected, undirected, edge-weighted graph $G = (V, E)$, an MST of $G$ is a graph $T = (V, E')$, such that $T$ is a tree and the sum of weights in $E'$ is minimized. In other words, an MST of $G$ is a spanning tree whose edges have the minimum sum out of all possible edge sets which create a spanning tree.

Cut property Let $G = (V, E)$ be an connected, undirected graph. Let $S$ be any subset of nodes that is neither empty nor equal to all of $V$. Let $e = (u, v)$ be the minimum cost edge with one end in $S$ and the other in $V - S$. Then some MST (every MST if edge weights are distinct) of $G$ contains $e$.

Cycle Property For any cycle $C$ in any graph $G$, if the weight of an edge $e$ of $C$ is larger than the individual weights of all other edges of $C$, then $e$ cannot belong to an MST of $G$.

Edge-Weighted DAGs (Directed Acyclic Graphs)

Algorithm

The algorithm for shortest path on edge weighted DAGs is simpler and faster than Dijkstra’s algorithm. Instead of considering vertices by priority of their distance estimates, we consider the vertices of the DAG in a topological order. Then we just relax each vertex in the topological ordering. This algorithm for shortest paths on edge weighted DAGs is simpler and faster than Dijkstra’s algorithm – it runs in $O(V + E)$ time.

Intuition

All greedy shortest path algorithms are built on the same fundamental assumption:

For each vertex $v$ and it’s predecessor $u$, the shortest path to $u$ must be calculated correctly by the time we calculate the shortest path to $v$.

Dijkstra’s accomplishes this task by using a priority queue and extracting based on minimum distance. In a DAG, we do not need as complicated of a method. Because all of the edges go the same direction in a DAG, the predecessor of any vertex must be before it in the topological ordering. So, by calculating shortest paths in topological order, we ensure that the shortest path to all predecessors is found before the path to their successors.

Testing your understanding

Problem 1. Does Kruskal’s algorithm work on a graph with negative weights? What about Prim’s?

Solution. Yes. The algorithm will still select the lowest weight edges in order (more negative edges first). Prim’s will also not be affected, as it does not rely on edges being positive.

Problem 2. Say we have some MST, $T$, in a positively weighted graph $G$. Construct a graph $G'$ where for any weight $w(e)$ for edge $e$ in $G$, we have weights $(w(e))^2$ in $G'$. Does $T$ still remain an MST in $G'$? Prove your answer. Now if $G$ also had negative weights, would your answer change from the previous part? Prove your answer.
Solution. If $G$ only has positive weights, then this claim holds. Proof by contradiction: assume the claim does not hold. This means that there is some tree $T'$ in $G'$ with lower total weight than $T$. Because $T \neq T'$, there is some edge in $T$ that is not in $T'$, call this edge $e$. We know that $T$ is a tree, so removing $e$ from $T$ creates a cut, $C$. Our edge $e$ is not in $T'$, but $T'$ is connected, so there must be some edge $e'$ which crosses this cut in $T'$. Trees are defined to be acyclic, so $e'$ cannot also be in $T$.

By the cut property, $w(e) \leq w(e')$, because $e'$ is not in $T$ and $T$ is a valid MST. However, because we know that $T$ is not valid in $G'$ but $T'$ is, we know that $w(e)^2 > w(e')^2$. If the edge weights are positive, this claim is impossible, completing the proof. However, this claim is possible for negative integers. Thus, if $G$ had negative weights, our answer would change to no, $T$ is not necessarily an MST in $G'$.

Problem 3. Suppose we have a function $f : \mathbb{R} \to \mathbb{R}$ such that for any graph $G$ and any MST $T$ of $G$, if we apply $f$ to the weights of $G$ creating $G'$, then $T$ is still an MST of $G'$. Give a property of $f$ such that this claim holds true if and only if the property of $f$ is true. That is, give a property such that iff $f$ has this property, then $T$ must be an MST in $G'$. Prove your answer.

Hint: For two edges, $e_1$ and $e_2$ such that $w(e_1) \leq w(e_2)$ in $G$, what must be true about $f(w(e_1))$ and $f(w(e_2))$ in $G'$?

Solution. In line with the hint, the necessary and sufficient property is that $w(e_1) \leq w(e_2) \implies f(w(e_1)) \leq f(w(e_2))$. The proof for the sufficiency is similar to the proof above. Following the same steps as in the contradiction proof leaves us with this claim: if $w(e) \leq w(e')$, but $f(w(e)) > f(w(e'))$. This contradicts the original definition of our function, so we have shown our definition to be sufficient.

To show this claim is necessary, assume for contradiction there is some function $f'$ that properly maintains $T$ in $G'$ but for at least one pair of values $x, x'$ it is true that $w(x) \leq w(x')$ and $f'(w(x)) > f'(w(x'))$. Because we must prove our claim for any arbitrary graph and MST, to contradict this we must only give a specific $G$ and $T$ for which $f'$ does not maintain $T$. Knowing this, we define the graph below as $R$, using the edge weights $x$ and $x'$ which violate our property:

![Diagram](https://via.placeholder.com/150)

For $R$, it is clear that the MST of $(A, B), (B, C)$ is a valid. This is because $w(x) \leq w(x')$, as assumed in our contradiction claim. Now apply $f'$ to all edge weights, creating $R'$. We know that $f'(w(x)) > f'(w(x'))$. Therefore, by the cycle property, we know that $x$ cannot be part of any valid MST. So, the MST given for $R$ is not valid in $R'$, completing the proof.

For intuition on this claim, we look to Kruskal’s. Note that the definition of $f$ as we have defined it keeps the sorted ordering of the edges. That is, for any set of edges $E$, the sorted ordering of $E$ would still be a valid sorted ordering for the edges in $E$ after applying $f$. Intuitively then, applying $f$ cannot change the validity of our MST, as Kruskal’s only ever looks at the ordering of the edges, not their actual magnitude.

Problem 4. A directed graph $G = (V, E)$ is semiconnected if, for all pairs of vertices $u, v \in V$ we have $u \leadsto v$ or $v \leadsto u$. Give an efficient algorithm to determine whether or not $G$ is semiconnected. Prove that your algorithm is correct, and analyze its running time.

Solution. The algorithm is as follows:
1. Call Kosaraju’s and obtain $G^{SCC}$.

2. Topologically sort $G^{SCC}$, obtaining an ordering of the vertices $v_1, v_2, ..., v_k$.

3. Verify whether for each pair of vertices $v_i$ and $v_{i+1}$, there is an edge $(v_i, v_{i+1})$. That is, check to make sure there is an edge between each adjacent pair in the topological ordering.

Because we know that all vertices in each SCC are mutually reachable from each other, it suffices to show that the component graph is semiconnected if and only if it contains a linear chain. We must also show that if there’s a linear chain in the component graph, it’s the one returned by topological sort.

We’ll first show that if there’s a linear chain in the component graph, then it’s the one returned by topological sort. In fact, this is trivial. A topological sort has to respect every edge in the graph. So if there’s a linear chain, a topological sort must give us the vertices in order.

Now we’ll show that the component graph is semiconnected if and only if it contains a linear chain.

First, suppose that the component graph contains a linear chain. Then for every pair of vertices $u, v$ in the component graph, there is a path between them. If $u$ precedes $v$ in the linear chain, then there’s a path $u \rightsquigarrow v$. Otherwise, $v$ precedes $u$, and there’s a path $v \rightsquigarrow u$.

Conversely, suppose that the component graph does not contain a linear chain. Then in the list returned by topological sort, there are two consecutive vertices, $v_i$ and $v_{i+1}$ such that there is no edge $(v_i, v_{i+1})$. There is no path from $v_{i+1}$ to $v_i$, because edges only point forward with respect to the topological ordering. We know that all edges going out of $v_i$ must be to vertices $v_j$, where $j > i + 1$. Therefore, there is no path from $v_i$ to $v_{i+1}$, because as discussed before there are no paths backward in a topological ordering. So, there is no path from $v_i$ to $v_{i+1}$ or from $v_{i+1}$ to $v_i$ and the component graph is not semiconnected.

Running time of each step:

1. $O(V + E)$

2. $O(V + E)$

3. Also $O(V + E)$. We just check the adjacency list of each vertex $v_i$ in the component graph to verify that there’s an edge $(v_i, v_{i+1})$ We’ll go through each adjacency list once.

Thus, the total running time is $O(V + E)$.

Problem 5. Imagine we have a graph $G$ where all edge weights are equal. Design an algorithm to efficiently find an MST of $G$. Analyze the running time.

Solution. Our optimal algorithm would be to run DFS and only keep track of the tree edges (so we don’t introduce any cycles). Notice at any step we can choose any edge, since the edge weights are all equal. The running time is there $O(E + V)$.