Minimum Spanning Trees (MSTs)

A minimum spanning tree is a tree within a weighted graph that contains the minimum total weight. Below are two well-known algorithms for finding such a tree.

Kruskal’s Algorithm

The algorithm can be described as follows:

- Start off with all vertices as standalone trees within a forest.
- Consider the next minimum edge from the graph.
- Add the edge to the forest if it connects two disjoint trees, otherwise discard it.
- Continue this until every vertex has been considered, and return the tree formed.

The premise is simple, but there are two complex operations that are taking place: union — the merging of two subsets into one, and find — determining if two subsets are connected. The fastest data structure for these operations is the union-find data structure. The analysis of union-find is complex and out of the scope of this class, but we can take for granted that it can perform union and find in $O(\alpha(V))$. Using the union-find data structure, the run time of Kruskal’s is $O(E\alpha(V))$.

Prim’s Algorithm

The algorithm can be described as follows:

- Start with an arbitrary vertex $v$.
- Find the minimum weight edge involving $v$ and any vertex not in the tree $u$, and add $u$ and the edge $(u,v)$.
- Repeat the previous step for any vertices not in the tree until all vertices have been considered, and return the tree formed.

Kruskal’s vs. Prim’s

- Both are algorithms to find the minimum spanning tree.
- Prim starts with a single vertex, and grows a tree from this vertex.
- Kruskal starts with every vertex as a separate tree, and combines them to form a single tree.
- Complexity of eager Prim (with a binomial heap): $O((E + V) \log V)$, complexity of Kruskal (with union-find with path compression): $O(E\alpha(V))$.
- In Kruskal, checking to see if adding an edge will create a cycle can be slow. Thus, Kruskal’s algorithm works better when there are fewer edges to vertices.
- Prim’s algorithm works better for dense graphs with more edges than vertices.
- Since Prim’s algorithm “grows” the tree by adding vertices, it always has a partial tree. If you only need a partial solution, use Prim.
Strongly connected components

Given a directed graph $G = (V,E)$, a strongly connected component (SCC), is a set $S \subseteq V$ such that for $u,v \in S$, there exists a path from $u$ to $v$ and a path from $v$ to $u$. The strongly connected component graph (SCC graph), is a graph $G' = (V',E')$ where the vertices in $V'$ represent the SCCs and the edges in $E'$ are the edges between them.

One reason the SCC graph is very useful is that it is a directed, acyclic graph (DAG). Why?

The idea of the proof is that cycles can only exist within SCCs; a cycle of SCCs is actually just a single SCC. Because the SCC graph is a DAG, we can topologically order its vertices. This technique (computing SCC graph and topologically ordering its vertices) is crucial to many linear time graph algorithms.

**Kosaraju’s Algorithm:** Run DFS on $G$, noting finish times. Then, in decreasing order of finish time, run DFS on the vertices of $G^T$ ($G$ with its edges reversed). The output is a DFS forest where each tree in the forest is an SCC of $G$.

**Running time analysis:** DFS takes $O(n+m)$ time. We perform it twice, for a total of $O(2(n+m)) = O(n+m)$. Computing $G^T$ requires simply iterating over $G$’s adjacency list once, $O(n+m)$ time. Thus, the total runtime is $O(n+m)$.

**Correctness:** In the DFS of $G$, after we visit a node $x$, we visit its SCC $C$ and some edges out of $C$. We observe that if there is a path from $x \rightarrow u$ in $G$, then $u$ and $x$ are strongly connected only if there is also a path from $x \rightarrow u$ in $G^T$. Because $G$ and $G^T$ have the same SCCs, there will be a path in $G^T$ from $x$ to every vertex in $C$ but the edges out of $C$ will have been reversed and they will not be followed before the algorithm finishes processing $C$. When $C$ is finished, the part of the DFS starting from the vertex with the next highest finish time will, by logic similar to the above, only reach vertices in its SCC. Continuing to apply this logic, we see that the output of the algorithm is a forest of DFS tree, each of which is a strongly connected component of $G$.

### Testing your understanding

**Problem 1.** Imagine we have a graph with negative weights. Do Prim’s and Kruskal’s still work for finding a minimum spanning tree? Prove your hypothesis by constructing a counterexample or making a modification to the graph.

**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.

**Problem 3.** Imagine we have a tree $T$ where all edge weights are equal. Design an algorithm to efficiently find an MST of $T$. Analyze the running time.

**Problem 4.** Let $G = (V,E)$ be a in which each vertex $u \in V$ is labeled with a unique integer $L(u)$ from the set $\{1,2,\ldots,n\}$. For each $u \in V$, let $R(u) = \{ v \in V \mid u \rightarrow v \}$ be the set of vertices reachable from $u$. Define $\min(u)$ to be the vertex in $R(u)$ whose label is minimum. Give an $O(n+m)$ algorithm to compute $\min(u)$ for each $u \in V$.

**Problem 5.** Bob is given some set of tasks $T_1, T_2, \ldots, T_n$ to complete. Some of the tasks can be performed in parallel with particular other tasks. Any such tasks are guaranteed to finish at the same time. He receives a list of pairs of tasks $(T_i, T_j)$, such that $T_i$ can be run in parallel $T_j$. Bob is also given pair list of dependencies, such that for any pair $(T_i, T_j)$ on the list, $T_i$ that need to be completed before $T_j$. Describe an algorithm for Bob to efficiently process the tasks.

An example would be: parallel tasks: $(T_1, T_2), (T_2, T_3), (T_3, T_1), (T_4, T_5)$ and dependencies: $(T_1, T_4)$. Then Bob should run $T_1, T_2, T_3$ in parallel, then run $T_4, T_5$.  

2