MST & Shortest Path

Now that we all know what a graph is, its basic properties, and how to traverse them. Now let's talk about some more specific features of graphs.

Minimum Spanning Tree (MST)

A tree is a **undirected**, **connected**, **acyclic** graph where there is **exactly one** path between any two vertices. A spanning tree in a graph is sub-graph that is a tree that connects all vertices within a given graph. A **Minimum Spanning Tree (MST)** is a spanning tree with the minimum total edge weight.

In English, a MST is a subset of the edges that connects the graph with the minimum total weight. There are various real life applications of minimum spanning tree, such as choosing how to lay optic cable to connect a bunch of data centers or roads connecting cities. But in contest, a lot of times you will just be asked to "find the MST."

There are two algorithms used, both are pretty intuitive, and have similar runtime but it's important to understand them as somtimes you'll need to construct the MST without constructing the entire graph (due to memory/time restrictions)

Prim's Algorithm

Given $G(V, E)$, Prim's algorithm starts with an arbitrary vertex, v_0, and divide up the graph into two sets of vertices: S and T. Initially $S = \{v_0\}$ and $T = V \setminus S$ (V without S). While T is non-empty / $S \neq V$, we pick the edge (v, u) with $v \in S$ and $u \in T$ with the minimum edge weight. Add the edge to the MST, and move u form T to S. In psudocode, it is:

```
Prim(Graph G):
    boolean visited[V] initialized to false
    visited[v_0] = true                 // for arbitrary v_0
    MST = {}

    while not all visited:
        e = (v, u) with minimum edge weight
            where visited[v] = true and visited[u] = false
        MST.add(e)
        visited[u] = true

    return MST
```

The runtime of this dependent on our implementation. The important operation is finding the edge with minimum weight between the visited and unvisited set. This is usually done by storing the vertices in a priority heap ordered by their shortest distance to the visited set, initialized to infinity. At each step, we take the vertex with the minimum distance to the partially constructed tree and add that to the MST. Then we update its neighbor's distances if necessary. This calls for two operations: remove min and decrease priority, both of which can be done in $\log(V)$ time. So the total runtime of the algorithm if implemented with a heap as priority queue is $O(|V|\log(|V|) + |E|\log(|V|))$, as we need to remove min $|V|$ times and decrease priority at most $|E|$ times. But since in a connected graph $|E| \geq |V|$, the runtime is often written as $O(|E|\log(|V|))$

Kruskal's Algorithm

Kruskal's algorithm is the opposite of Prim's. Instead of building a tree spawning from one vertex. It builds a forest that eventually becomes one single spanning tree. This algorithm is greedy in nature as well. We began by picking the shortest edge, adding that to the MST, then we combine the two trees at the endpoints of the edge. In psudocode it is:

```
Kruskal(Graph G):
    MST = {}
    set trees[V] initialized to {v} for each v in V

    for e = (u, v) in G.E ordered by weight increasingly:
        if trees[u] =/= trees[v]:
            MST.add(e)
            trees[u] = trees[v] = UNION(trees[u], trees[v])

    return MST
```

Note that the runtime of this is exactly like the runtime of Prim $O(E\log(V))$ because we literally go through each edge and the unions of sets as well as checking if two elements are in the same set can be done in $O(\log(N))$ time where N is the size of the larger set.

Shortest Path – Dijkstra

Similar to BFS and DFS, Dijkstra's algorithm is also an algorithm for searching for a path between any two nodes. The advantage of this over the other two is that this algorithm is targeted in finding the shortest path, and have significantly better run time. The general idea behind Dijkstra's algorithm is known as **best first search**, which means at any stage of the search we prioritize the best candidate.

We assign a distance to each node, 0 at the initial node and infinity at every other node. Then at each node, we calculate the tentative distance to each of its neighbors by adding the weight of the edge to the distance at the current node. If the tentative distance is smaller than the distance currently associated with the neighbor, we will update the distance to the smaller value. We will then mark the current node as visited and move to the neighbor with the lowest associated distance.

This algorithm is a bit more complex than the others when written in psudocode but here is what it should look like:

```
Dijkstra(Graph G, Vertex src, Vertex dest):
    boolean visited[V] initialized to false
    int dist[V] initialized to infinity
    priorityQueue frontier sorted by dist, with min on top

    dist[src] = 0
    frontiner.push(src)

    while (!frontier.empty()):
        v = frontier.pop()

        if visited[v]:
            continue; // already been here, skip!

        if v == dest:
            return dist[dest]

        for (v, u) in G.E with weight w:
            if dist[u] > dist[v] + w:
                dist[u] = dist[v] + w
                frontier.push(u)

        visited[v] = true

    return dist[dest]
```

The algorithm terminates when the node we are looking for has been visited, or when the entire graph has been traversed. The algorithm has the worst-case runtime of $O((|V| + |E|)\log(|V|))$ if we use a priority queue to store distance.

Floyd-Warshall

Dijkstra's algorithm is one that calculates the shortest distance from one point to another (and potentially one point to every other point in the graph). However, sometimes we need to find all pairwise distances between all pairs of nodes. This can be done with repeated Dijkstra's with time complexity of $O(|E||V| + |V|^2\log(|V|))$. However, this is doing quite a bit of overcounting as we are traversing the graph more times than necessary. In truth we only need to traverse all the edges one per vertex. This is known as the Floyd-Warshall algorithm. It repeated gets estimates of all pairwise distances and stop at the optimal one.

In psudocode it is:

```
let dist be a 2D array of distances between vertices initialized to infinity
for v from 1...V:
    dist[v][v] = 0
for (u, v) in E with weight w:
    dist[u][v] = w

for k from 1...V:
    for j from 1...V:
        for i from 1...V:
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] = dist[i][k] + dist[k][j]
```

Basically the idea behind this is that if we can go from point i to j faster through k, we'll do it. Note that this algorithm does not give the exact path, a simple modification will return the path. [every time we do the substitution w/ k, we know it is a part of the path]