Minimum Spanning Tree: Prim's Algorithm

Minimum Spanning Trees

First let's define a tree, a spanning tree, and a minimum spanning tree:

tree: A connected graph without cycles. (A cycle is a path that starts and ends at the same vertex.)

spanning tree: a subtree of a graph that includes each vertex of the graph. A subtree of a given graph as a subset of the components of that given graph. (Naturally, these components must form a graph as well. Thus, if your subgraph can't just have vertices A and B, but contain an edge connecting vertices B and C.)

Minimum spanning tree: This is only defined for weighted graphs. This is the spanning tree of a given graph whose sum of edge weights is minimum, compared to all other spanning trees.

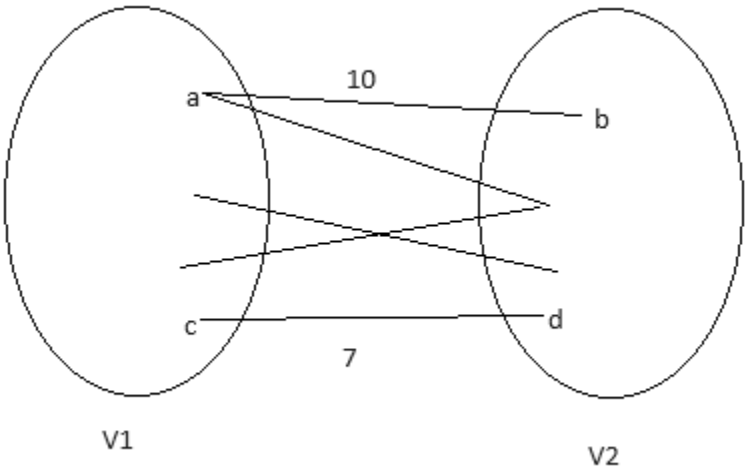
The practical application of finding a minimum spanning tree in a graph is as follows:

Imagine needing to build a railroad network to make sure that set of locations is connected. Namely, that once the tracks are built, goods from any of the locations in the set can be sent to any of the other locations in the set. We don't require that the distance of any pair of locations be minimized, but just that it's possible to ship goods from any one location to any other location. Since building railroad tracks is extremely costly in time and labor, we would like to minimize the distance (or perhaps monetary cost) of the tracks built. Thus, given all possible connections we could create and the cost of creating them, this information builds a weighted graph. The minimum spanning tree of that graph represents the minimum cost of building the tracks so that goods can be shipped between any pair of locations.

Crucial Fact about Minimum Spanning Trees

Let G be a graph with vertices in the set V partitioned into two sets V_1 and V_2. Then the minimum weight edge, e, that connects a vertex from V_1 to V_2 is part of a minimum spanning tree of G.

Proof: Consider a MST T of G that does NOT contain the minimum weight edge e. This MUST have at least one edge in between a vertex from V_1 to V_2. (Otherwise, no vertices between those two sets would be connected.) Let G contain edge f that connects V_1 to V_2. Now, add in edge e to T. This creates a cycle. In particular, there was already one path from every vertex in V_1 to V_2 and with the addition of e, there are two. Thus, we can form a cycle involving both e and f. Now, imagine removing f from this cycle. This new graph, T' is also a spanning tree, but its total weight is less than or equal to T because we replaced e with f, and e was the minimum weight edge.



Thus basically, if someone claims that T is an MST, and that T contains edge ab (weight 10) and not edge cd (weight 7), we know that in T, the two vertices connected by the 7 are already connected indirectly. Thus, if we add the edge cd to T, this will cause a cycle, where we can go from c to a (indirectly potentially), then a to b, then b to d (indirectly potentially). We can remove the cycle by deleting the edge ab, and this will create a new spanning tree. This new spanning tree has the weight of the old one, plus 7, minus 10, which means that our new spanning tree is better! (Thus, in any MST, a minimum edge between the partition must be included.)

Prim's algorithm makes use of this fact by "growing" the set V1. Basically, we start with a single vertex in V1, and then at each iteration of Prim's, we find the smallest edge that connects from the current V1 to the current V2. Based on this proof, it's safe to add that edge into our minimum spanning tree. When we do so, we add that vertex into V1 (our set of connected locations) and delete it from V2.

Prim's Algorithm

We use the crucial fact about minimum spanning trees in this algorithm by starting with one vertex and "growing" a larger tree that ALWAYS stays connected. Thus, we start off with the set V_1 having 1 vertex and V_2 having the rest, and at each step, adding the minimum edge from V_1 to V_2 to our MST, which will then "grab" one new vertex at each step to add to V_1 and remove from V_2. When we are done, V_2 will be empty!

Here is the algorithm:

1. Initialize a set $V_1 = \{v\}$, where v is the starting vertex of your choosing. (By default, many pick vertex 0 since vertices are usually numbered 0 to n – 1.)

2. Create a Priority Queue of edges. Add to this priority queue all edges from v to other vertices.

3. While the set V_1 isn't equal to V, the set of vertices in the graph, do the following:

    a. Remove the next edge in the priority queue, if it exists. (If it doesn't, the input graph isn't connected.)

    b. If the edge connects two vertices already in V_1, skip the edge and go back to step a above.

    c. Otherwise, add this edge into the minimum spanning tree, and take the vertex from the edge that was previously in V_2 and add it to V_1.

4. When we get here, if $V_1 = V$, the edges we've added are a valid minimum spanning tree.

To implement this, use a boolean array to keep track of the set v1. Thus, set used[i] = true once vertex i gets added to the set v1. If necessary, we can keep a set of edges to indicate which edges are in the minimum spanning tree. In many questions though, just the weight of the minimum spanning tree is required, so in these cases, a single accumulator variable is necessary, where you just add in the weight of each minimum spanning tree edge.