Minimum Spanning Tree

- Given a weighted graph $G$, we want to find the least-cost tree that spans the graph.
MST vs SPT

Shortest path tree from A
Total Cost: 8
Total Cost of Paths from A:
3 + 3 + 2 = 8

Minimum Spanning tree
Total Cost: 6
Total of Paths from A:
2 + 4 + 4 = 10
Kruskal's Algorithm

• One way to find a MST is via Kruskal's algorithm:
• Take the smallest edge that does not induce a cycle, and insert it into our subgraph.
• Do this until all nodes are connected
Kruskal's Algorithm

- One way to find a MST is via Kruskal's algorithm:
- Take the smallest edge that does not induce a cycle, and insert it into our subgraph.
- Do this until all nodes are connected
- A naive way to make sure an edge does not induce a cycle is by using DFS or BFS from one of the edge's vertices, and seeing if we reach the other. If we do, adding that edge would create a cycle.
Remember the Cut Property:
If we split all the vertices into 2 groups,
The least edge between them is part of some MST.
If we have a MST with e, we are done.

Otherwise, we have to look closer.
Since we had a MST, adding $e$ creates a cycle.

So there must be some other edge, $e'$ that goes between $V_1$ and $V_2$ in that cycle.

(Since it's a cycle, we have to go from $V_1$ to $V_2$, and then back again)
However, since $e$ is the least edge between $V_1$ and $V_2$, we can replace $e'$ with $e$ without increasing the cost of the MST.

The graph is also connected because removing one edge from a cycle never disconnects the graph.
In Kruskal's algorithm, we add the least edge $e$ that does not form a cycle.

In other words, $e$ is the least edge of some cut where $V_1$ is a connected component, and $V_2$ is the rest of the edges.
Priority Queue

• Then, we can just use a Priority Queue to store the edges, since we only want the current cheapest one.

• However, we may poll an edge that is cheapest, but forms a cycle
Cycles

- The least cost edge is an edge between two connected components.
- So we want to ignore and edge if it is incident to two vertices in the same component.
Connected Components

• So all we have to do is keep track of the connected components we have formed.

• The best way to do this is with a Union-Find data structure
  • These are in your book
Simple Union-Join

• There are more efficient ways, but for our purposes we will use an array
• What we can do is have an array that has an entry for every vertex.
• The entry corresponds to which component the vertex belongs to
Simple Union-Find

- Initially, each entry is just the index of the array (each vertex is its own component)
- When we connect two components together, with numbers x and y.
- We then iterate through the array, replacing each y with x.
init:
    for (each node k) do
        groupID[k] = k;  // groupID[k] = id of the group that node k belongs

Edges = queue of edges ordered by the cost of the edge

Kruskal's Algorithm:
    while (not all nodes included) {
        e = next edge in Edges;  // least cost unprocessed edge
        if (e connects 2 vertices of the same group)
            discard edge;
        else  // e connect 2 different groups of nodes together
            Add e to MST;
            G1 = group ID of one of the groups connected by e;
            G2 = group ID of the other group connected by e;

            for (each node k with groupID == G2)
                groupID[k] = G1;  // Put node in group G2 into group G1
    }

Edges: \{(BC,1),(AC,2),(DE,2),(AB,3),(AE,4),(CE,5),(BD,7),(CD,7)\}

Vertex Groups: \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}\}
Edges: \{(AC,2),(DE,2),(AB,3),(AE,4),(CE,5),(BD,7),(CD,7)\}

Vertex Groups: \{{A}, \{B,C\}, \{D\}, \{E\}\}
Edges: \{\text{(DE,2), (AB,3), (AE,4), (CE,5), (BD,7), (CD,7)}\}

Vertex Groups: \{\{A,B,C\}, \{D\}, \{E\}\}
Edges: {(AB,3), (AE,4), (CE,5), (BD,7), (CD,7)}

Vertex Groups: {{A,B,C}, {D,E}}
Edges: \{(AE, 4), (CE, 5), (BD, 7), (CD, 7)\}

Vertex Groups: \{\{A, B, C\}, \{D, E\}\}

Edge AB ignored because A and B are part of the same connected component.
Edges: \{(CE,5),(CD,7)\}

Vertex Groups: \{\{A,B,C,D,E\}\}
Run Time

• Run time of Kruskal's: $n$ vertices, $m$ edges. Assuming heap for priority queue

• Priority Queue operations $O(m \log(m))$ for insertions, but there is a linear way to do it as well.

• At worse, we need to remove all edges from the PQ, which is $O(m \log(m))$
Run Time

• Since the graph is simple, the number of edges is at most $n^2/2$ which we'll simplify to $n^2$.
• So our removal is
  $O(m \log(n^2)) = O(2m \log(n)) = O(m \log(n))$

• Using a union-join data structure, we can form clusters and query clusters in $m \log(n)$ time.
• So the total run time is $O(m \log(n))$
Prim's Algorithm

- Mark a vertex.
- while we still don't have a spanning tree
- Take the least edge that is between a marked and unmarked vertex
- mark the unmarked vertex
Simple Prim's
Implementation Notes

- To implement Prim's algorithm, we take some ideas from Dijkstra's
- We give each vertex a label corresponding to the weight of the least edge connected to a marked vertex.
- Since we always want the least, we can use a priority queue.
  - But as we mark vertices, the label can change.
  - So we want to use an adaptable priority queue.
Vertex Label Updates

• When we mark a vertex, we iterate through all of its edges and update each unmarked vertex.
Init:
For each vertex v:
    Label v infinity
Vertices = all vertices of the graph ordered by the label

Prims Algorithm:
while(Vertices is not empty):
    V = next vertex in Vertices //Least cost vertex
    Add V to the subgraph;
    if(V has an edge) :
        Add the edge to the subgraph; //The first added vertex will not have a corresponding edge

    for(each edge e that contains V) :
        V2 = vertex in e that is not V;
        if(e.cost < label of V2) :
            V2's label = e.cost;
            V2's edge = e;
PQ: [(C,2), (B,3), (E,4), (D,infinity)]
PQ: [(B, 1), (E, 4), (D, 7)]
PQ: [(E,4), (D,7)]
PQ: [(D,2)]
Prim's Run Time

- We always need to remove all vertices from the PQ, which is $O(n \log(n))$
- We may have to update a vertex on every edge it has, which is $O(m \log(n))$.
  - M priority queue updates, each which is $\log(n)$
- So the total running time is $O((m+n)\log(n))$.
- Since $m$ is either close to $n$ (since the graph must be connected) or much greater than $n$ (up to $O(n^2)$), we can write this as $O(m \log(n))$