Minimum Spanning Tree: (for Directed Graphs)

Kruskal's Algorithm: Consider edges in ascending order of cost. Add the next edge to T unless doing so would create a cycle.

```
Algorithm: Kruskal-MST (G)
Input: Graph G=(V,E) with edge-weights.

Initialize MST to be empty;
Place each vertex in its own set;
Sort edges of G in increasing-order;
for each edge e = (u,v) in order
    if u and v are not in the same set
        Add e to MST;
        Compute the union of the two sets;
    endif
endfor
return MST

Output: A minimum spanning tree for the graph G.
```

Analysis: (adjacency matrix)

- Placing edges in list: requires scan of adjacency matrix $\Rightarrow O(V^2)$.
- Sorting edges: $O(E \log(E))$.
- One union-find operation for each edge: $O(E \log(V))$.
- Total: $O(V^2) + O(E \log(E)) + O(E \log(V))$ $\Rightarrow O(V^2)$, for sparse graphs $\Rightarrow O(V^2) \log V$, for dense graphs

Analysis: (adjacency list)

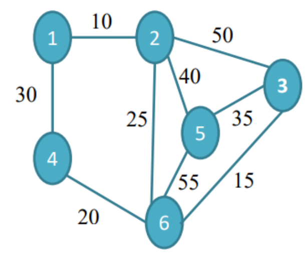
- To place edges in list: simply scan each vertex list once. $\Rightarrow O(E)$ time.
- Total: $O(E) + O(E \log(E)) + O(E \log(V))$ $\Rightarrow O(E \log(E))$ $\Rightarrow O(E \log(V))$ (same for sparse or dense)

Kruskal's algorithm is a greedy algorithm used to obtain MST using a direct generic method. Here, all the edges are considered in non-decreasing (increasing) order and the edges to be included are based on the order.

Procedure for Kruskal's approach to MST:

- Step 01: Construct an edge list E in the increasing order of their costs.
- Step 02: Pick one edge at a time from the list starting from the least cost edge.
- Step 03: Check if the inclusion of the edge causes a cycle or a loop in the output.
- Step 04: If it causes a cycle, discard it. If it does not generate a cycle, include it into the tree.
- Step 05: Repeat the steps 2 – 4 until a tree is generated with all nodes of the graph.

```
Algorithm: Kruskal-MST (adjMatrix)
Input: Adjacency matrix: adjMatrix[i][j] = weight of edge (i,j) (if nonzero)

Initialize MST to be empty;

// Initialize union-find algorithm and
// place each vertex in its own set.
for i=0 to numVertices-1
    makeSingleElementSet (i)
endfor

sortedEdgeList = sort edges in increasing-order;

Initialize treeMatrix to store MST;

// Process edges in order.
while sortedEdgeList.notEmpty()
    edge (i,j) = extract edge from sortedEdgeList.
    // If edge does not cause a cycle, add it.
    if find(i) != find(j)
        union (i, j)
        treeMatrix[i][j] = treeMatrix[j][i] = adjMatrix[i][j]
    endif
endwhile

return treeMatrix

Output: adjacency matrix represention of tree.
```

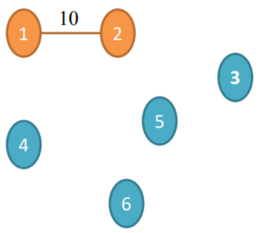


Edge List E in Increasing Order

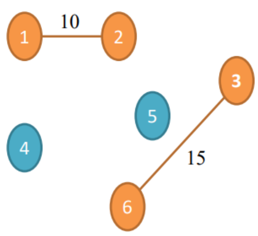
Edge	Cost
1 – 2	10
3 – 6	15
4 – 6	20
2 – 6	25
1 – 4	30
3 – 5	35
2 – 5	40
2 – 3	50
5 – 6	55

✓ 1
✓ 2
✓ 3
✓ 4
✗ 5 cycle
✓ 6
✗ stopping since all nodes are covered

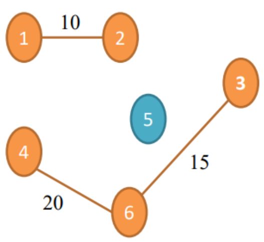
1



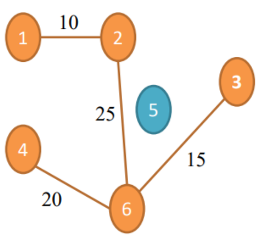
2



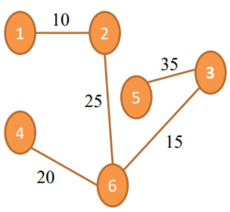
3



4



6



Prim's Algorithm:

Prim's algorithm is a greedy strategy used to obtain the MST from a graph edge by edge. The edge to be included is chosen according to the optimization criteria i.e., to chose an edge that results in a minimum increases in the sum of costs of the edges included so far.

If A is the set of edges selected so far, then A forms a tree, the next edge (u, v) to be included is a minimum cost edge not in A, with the property that $A \cup \{(u, v)\}$ is also tree.

Steps in Prim's Algorithm:

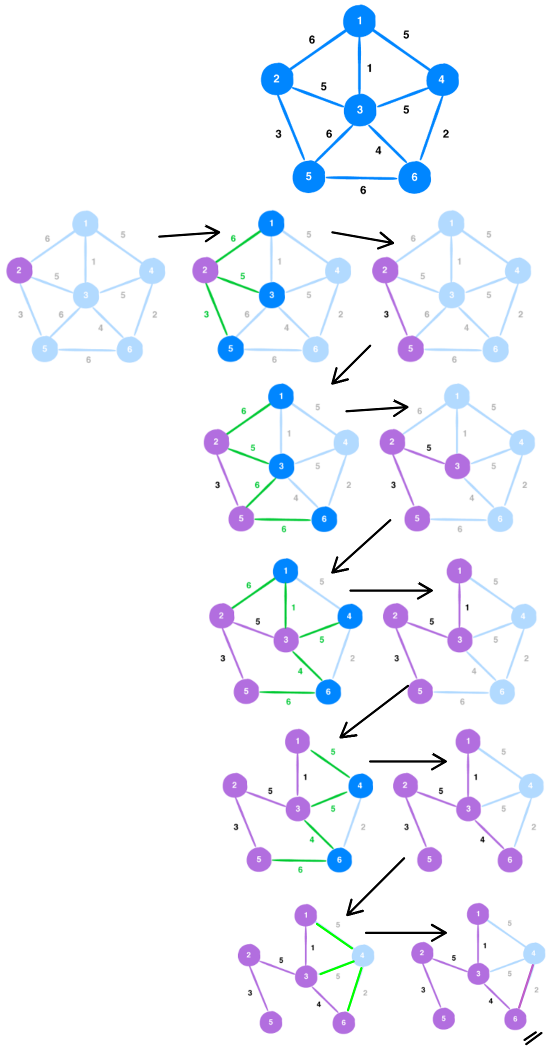
- Step 01: Select any vertex (or alphabetically or given vertex) as the source.
- Step 02: Compute the distances from the source to each other vertices of the graph.
- Step 03: Choose the vertex which leads to a minimum increase in the sum of costs of the edges included. Add the vertex to the tree iff the addition does not lead to a cycle.
- Step 04: Repeat the steps 2 and 3 until a tree is generated with all the vertices of the graph.

3

```
Algorithm: Prim-MST (adjMatrix)
Input: Adjacency matrix: adjMatrix[i][j] = weight of edge (i,j) (if nonzero)

// inMST[i] = true once vertex i is in the MST.
Initialize inMST[i] = false for all i;
Initialize priority[i] = infinity for all i;
priority[0] = 0
numVerticesAdded = 0

// Process vertices one by one. Note: priorities change as we proceed.
while numVerticesAdded < numVertices
  // Extract best vertex.
    v = vertex with lowest priority that is not in MST;
    // Place in MST.
    inMST[v] = true
    numVerticesAdded = numVerticesAdded + 1
    // Explore edges going out from v.
    for i=0 to numVertices-1
        // If there's an edge and it's not a self-loop.
        if i != v and adjMatrix[v][i] > 0
            if priority[i] > adjMatrix[v][i]
                // New priority.
                priority[i] = adjMatrix[v][i]
                predecessor[i] = v
            endif
        endif
    endfor
endwhile

treeMatrix = build adjacency matrix representation of tree using predecessor array;
return treeMatrix

Output: Adjacency matrix representation of MST
```

```
Algorithm: Prim-MST (G)
Input: Graph G=(V,E) with edge-weights.

Initialize MST to vertex 0.
priority[0] = 0
For all other vertices, set priority[i] = infinity
Initialize prioritySet to all vertices;
while prioritySet.notEmpty()
    v = remove minimal-priority vertex from prioritySet;
    for each neighbor u of v
        // See if the priority of u changes because of v.
        w = weight of edge (v, u)
        if w < priority[u]
            priority[u] = w
        endif
    endfor
endwhile

Output: A minimum spanning tree of the graph G.
```

Example:



Kruskal:

https://iq.opengenus.org/time-and-space-complexity-of-kruskal-algorithm/

using union & find:

explaination: https://www.youtube.com/watch?v=JZBQLXgSGfs

code: https://gist.github.com/DanilAndreev/e519d77eff91f03f09616c9170db7941

Prim's:

https://iq.opengenus.org/time-and-space-complexity-of-prims-algorithm/

Links:

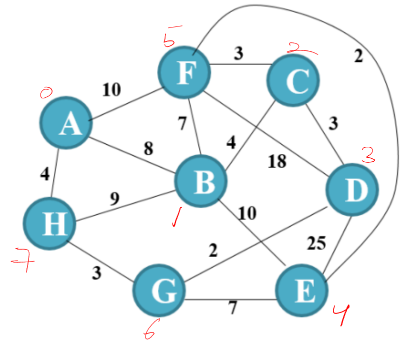
https://stackoverflow.com/questions/53389594/given-an-edge-find-a-minimum-spanning-tree-if-exist (gist: when there is a node with no edges in graph then MST doesn't exist)

https://www2.seas.gwu.edu/~simhaweb/alg/modules/module8/module8.html (pseudo code and run time analysis)

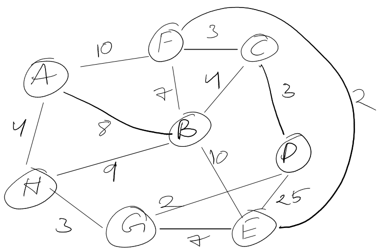
https://users.cs.northwestern.edu/~agupta/_projects/algorithms/Fibonacci%20Heaps/Doc/Design%20Document%20%231.pdf

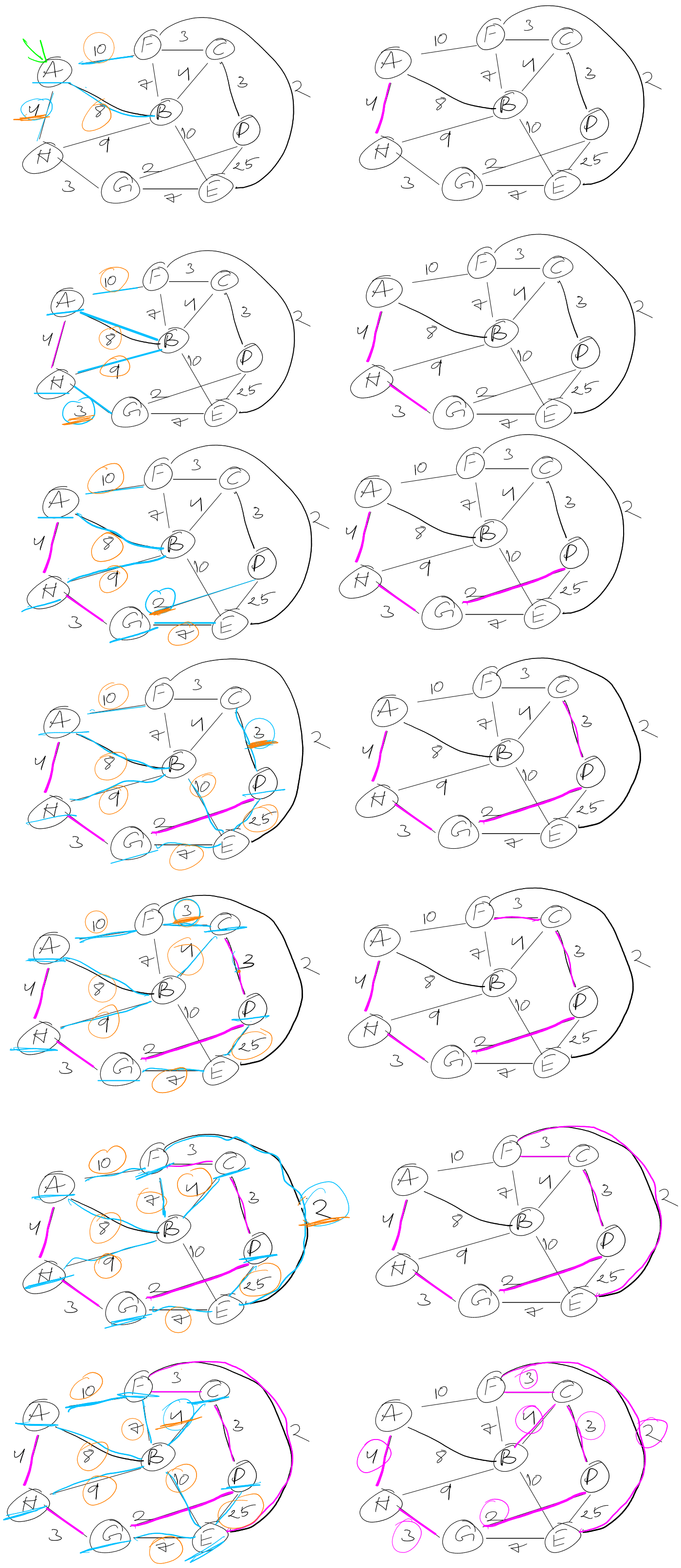
Kruskal:

	Sorted edge list
A B 8	D G 2
A F 10	E F 2
A H 4	C D 3
B C 4	C F 3
B E 10	G H 3
B F 7	A H 4
B H 9	B C 4
C D 3	B F 7
C F 3	E G 7
D E 25	A B 8
D F 18	B H 9
D G 2	A F 10
E G 7	B E 10
E F 2	D F 18
G H 3	D E 25

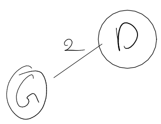


Prim's

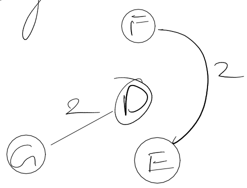




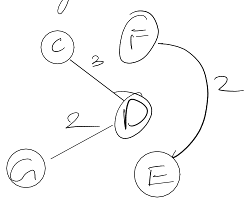
① add edge D-G



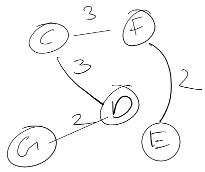
② add edge E-F



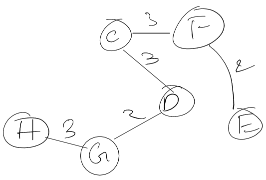
③ add edge C-D



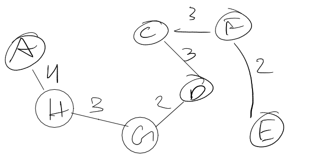
④ add edge C-F



⑤ add edge G-H



⑥ add edge A-H



⑦ add edge B-C