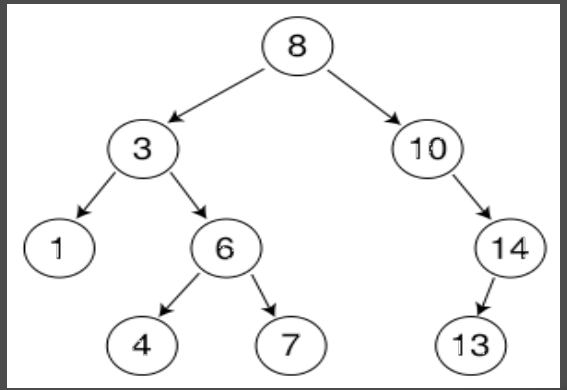
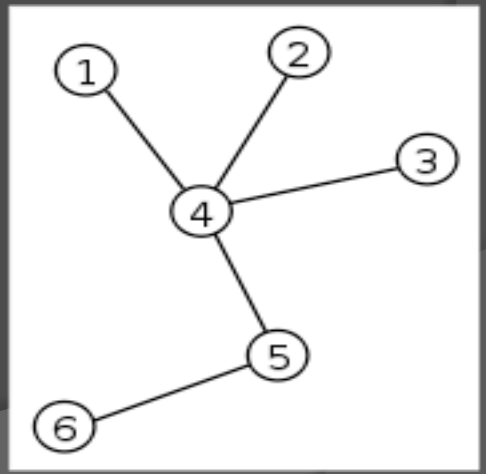
Advanced Computer Contest Preparation
Lecture 8

MINIMUM SPANNING TREE
PRIM'S ALGORITHM

Recall: What is a Tree?

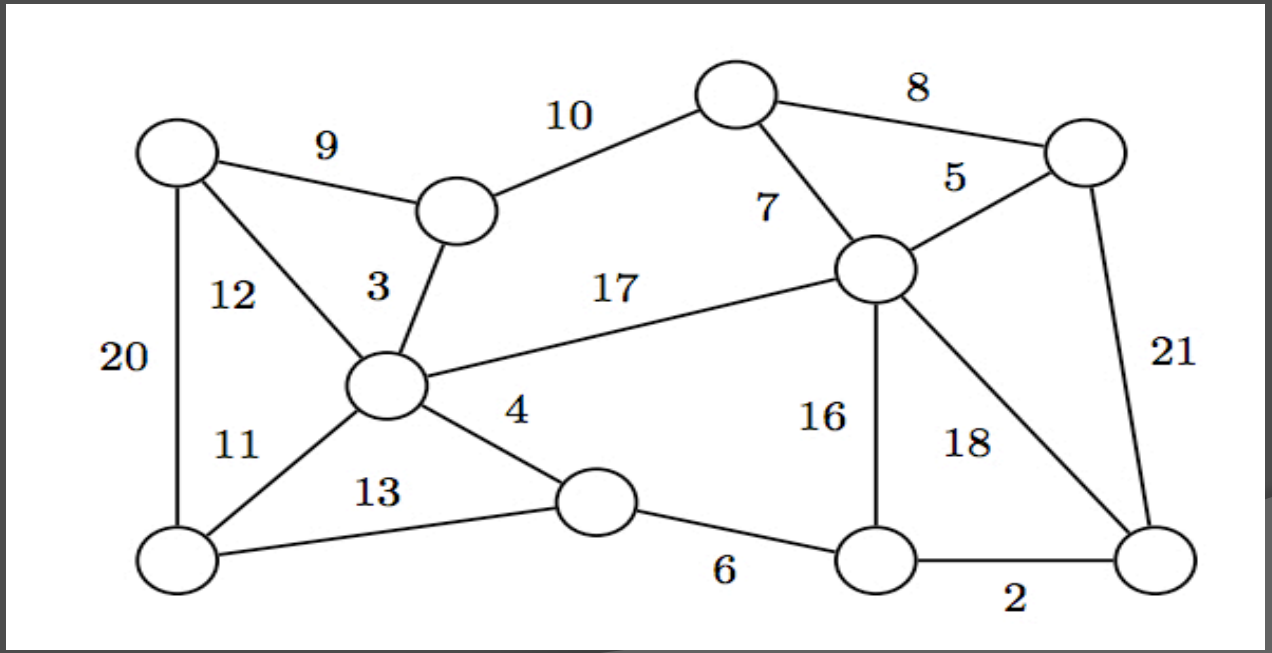
- A graph
- Connected (with respect to root)
- Contains no cycles
- Has $V-1$ edges





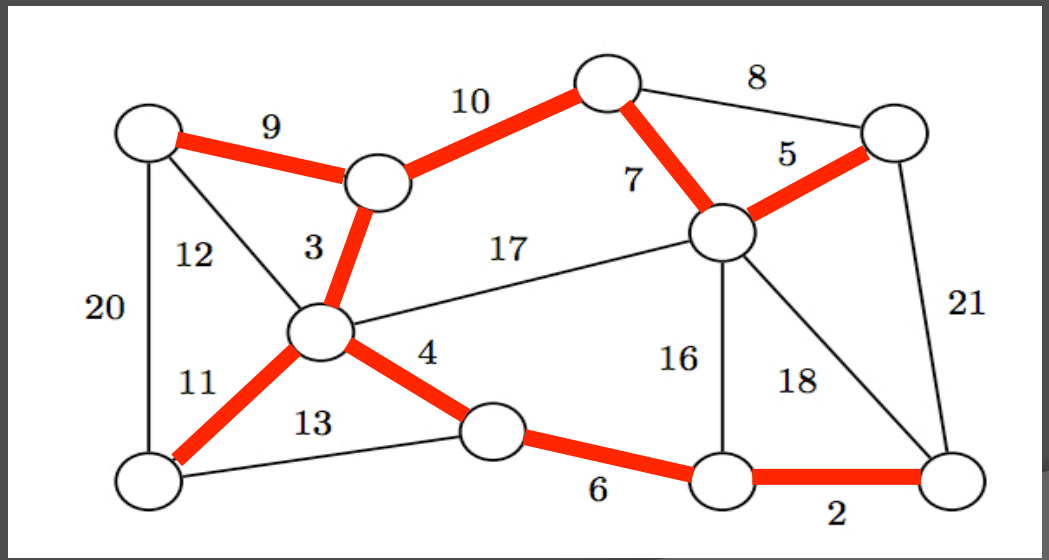
Minimum Spanning Tree

- What is a minimum spanning tree?



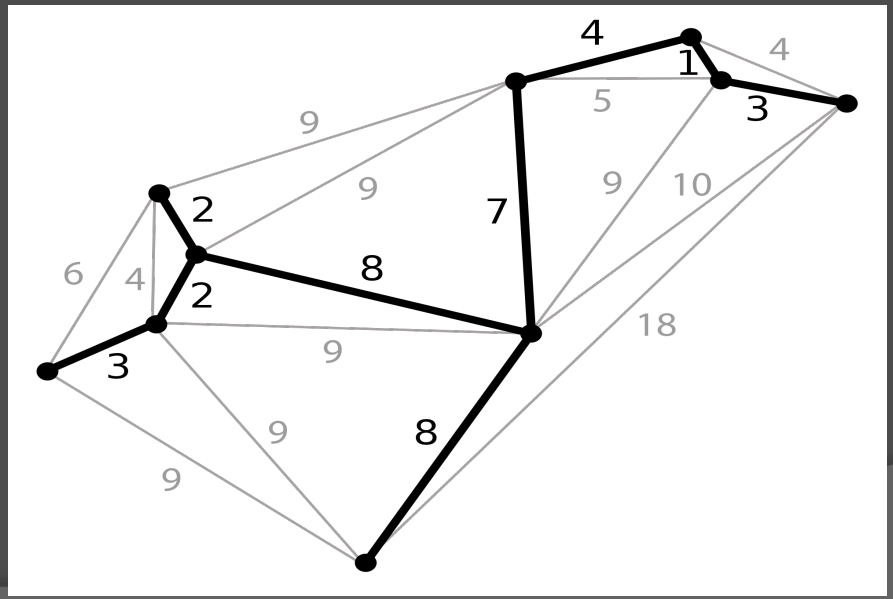
Minimum Spanning Tree

- Given a graph, the minimum spanning tree (MST) is a set of edges where all nodes are connected, and the total cost is minimum.



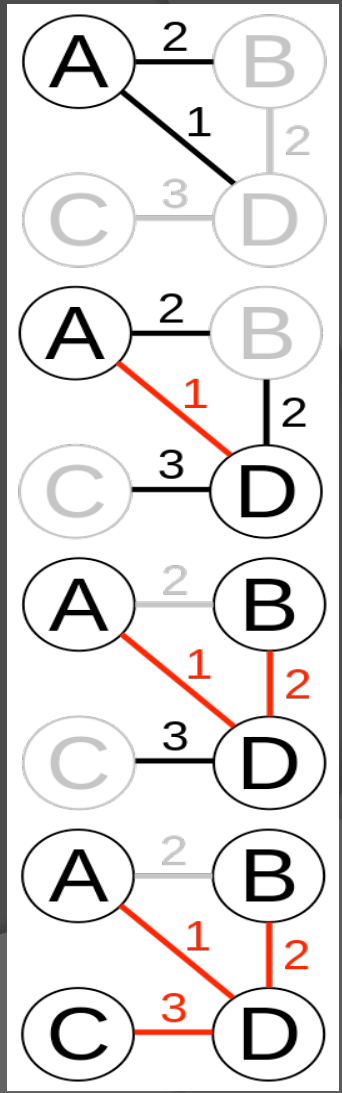
Minimum Spanning Tree

- How to find the minimum spanning tree?
- Can this problem be solved using an algorithm we already know?



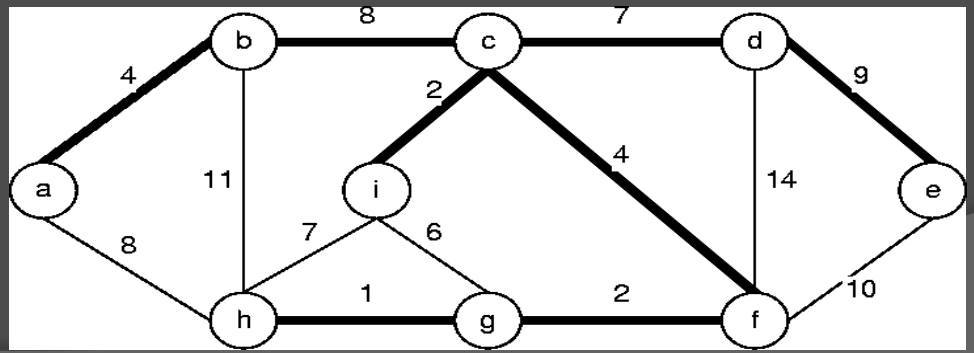
Prim's Algorithm

- Very similar to Dijkstra's algorithm
- **Difference:**
  - **We are finding smallest possible edges to add to the MST**
  - In Dijkstra's algorithm, we find the node with the smallest cumulative distance to visit next instead



Prim's Algorithm

- Finding an edge to add to the MST is the same as finding the unvisited node with the smallest edge cost (not cumulative).
- **In Prim's, the `dist[]` value of each node is the cost of the smallest cost edge connected to that node, not distance from start**



Prim's Algorithm

- You might want to store MST edges into a new graph (edge list, adjacency matrix/list)
- Not always necessary
  - Examples of when it is NOT necessary:
    - Getting maximum cost of all edges in the MST
    - Summing cost of all edges in MST

```
[[a, [b],
 [b, [a, c, g],
 [c, [b, d],
 [d, [c, e],
 [e, [d, f],
 [f, [e, g],
 [g, [b, f]]
```

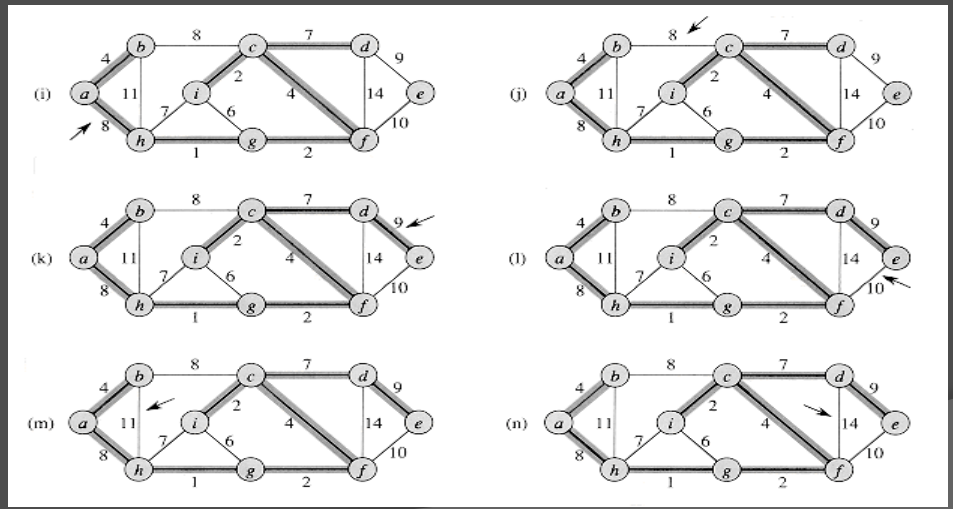
Adjacency List

```
   a  b  c  d  e  f  g
a [ , x,  ,  ,  ,  ,  ]
b [x,  , x,  ,  ,  , x]
c [ , x,  , x,  ,  ,  ]
d [ ,  , x,  , x,  ,  ]
e [ ,  ,  , x,  , x,  ]
f [ ,  ,  ,  , x,  , x]
g [ , x,  ,  ,  , x,  ]
```

Adjacency Matrix

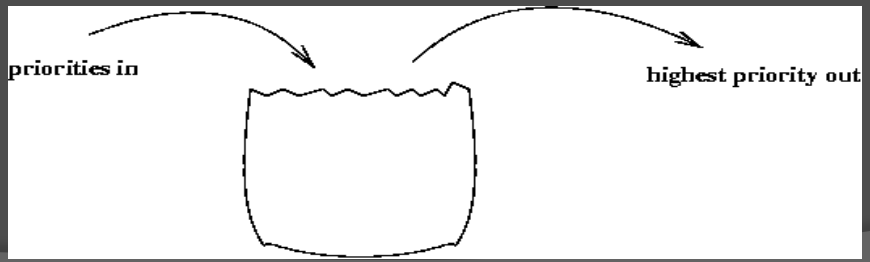
Prim's Algorithm

- To record edges in a graph (the two nodes connected), use an additional `parent[]` array to keep track of the neighboring node that last updated edge distance.



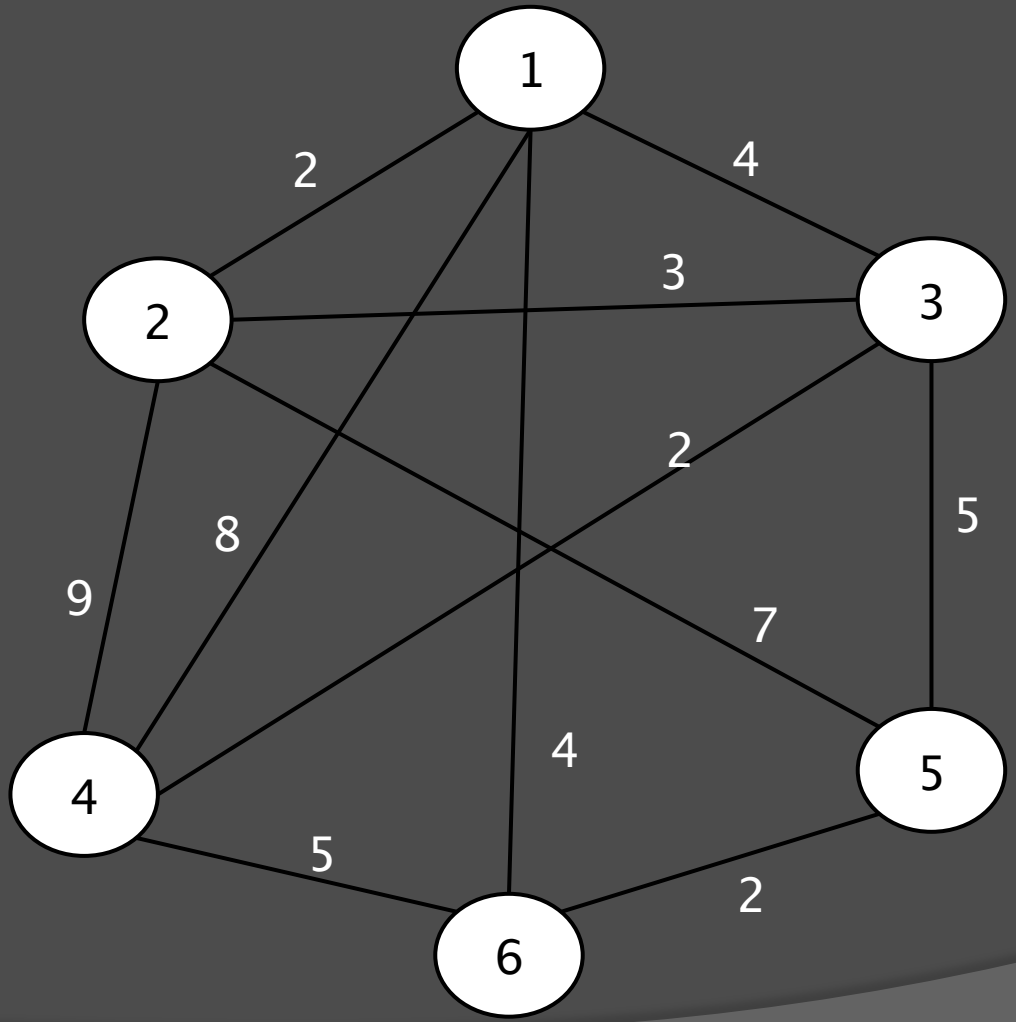
Prim's Algorithm

- Just like Dijkstra's algorithm, a priority queue can be used to speed up Prim's algorithm on sparse graphs
- You **need** a visited array for the priority queue version
- Runtime is same as Dijkstra's



Prim's Algorithm - Summary

- Initialize everything (`vis` to 0, `dist` to ∞)
- Start at any node (`dist` to 0)
- While there are nodes left to be processed:
  - Find the current node (unvisited node with lowest `dist`)
    - You are actually finding smallest edge not in the MST
  - If current node is not start, add edge with previous node and current node to MST (optional)
    - Use the `parent` array
  - Update distance to neighbors
    - Update if cost of edge to neighbor is less than `dist` value of neighbor
    - Update parent value of neighbor if distance was updated



Initialize Everything:
Set **`vis`** array to false
Set **`dist`** array to infinity

Vis

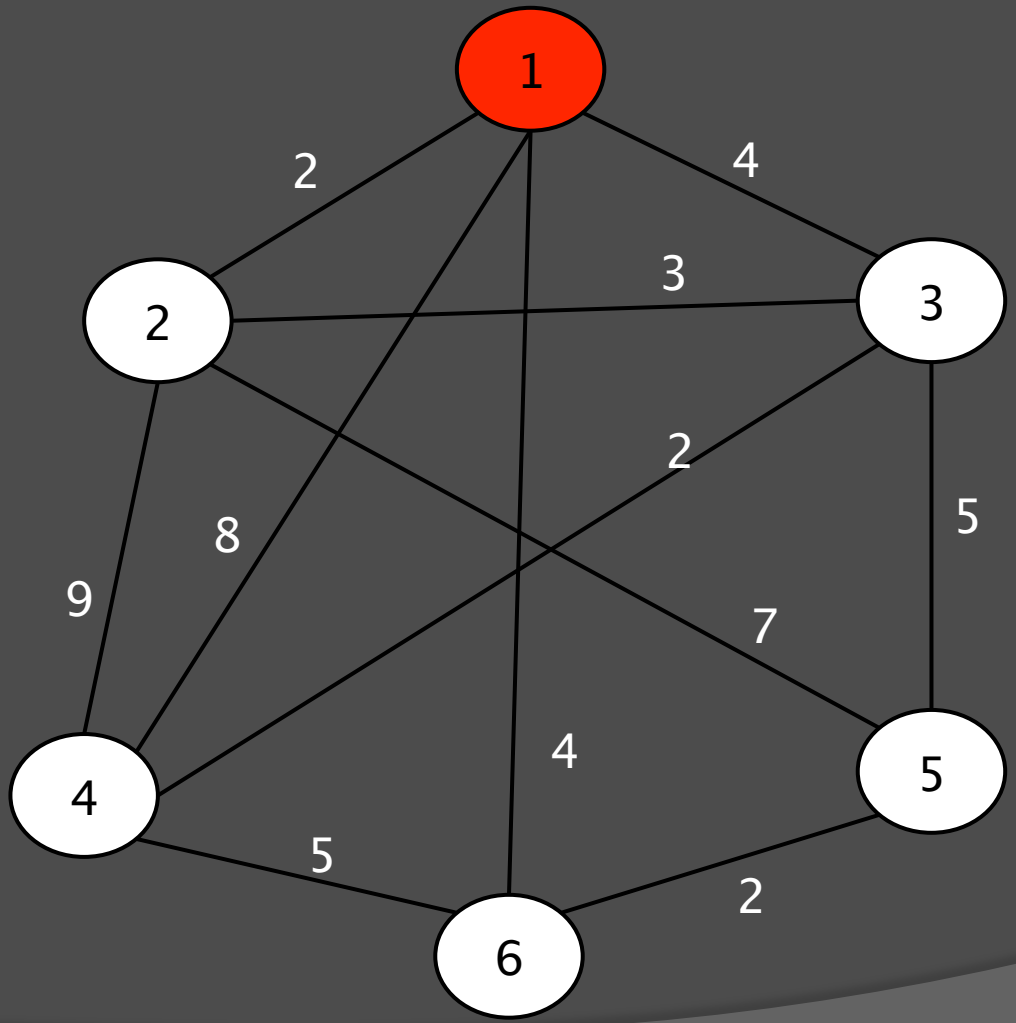
1	2	3	4	5	6
0	0	0	0	0	0

Parent

1	2	3	4	5	6
0	0	0	0	0	0

Dist

1	2	3	4	5	6
∞	∞	∞	∞	∞	∞



Start from node 1, set distance to 0, mark as visited, update distance to neighbors

Vis

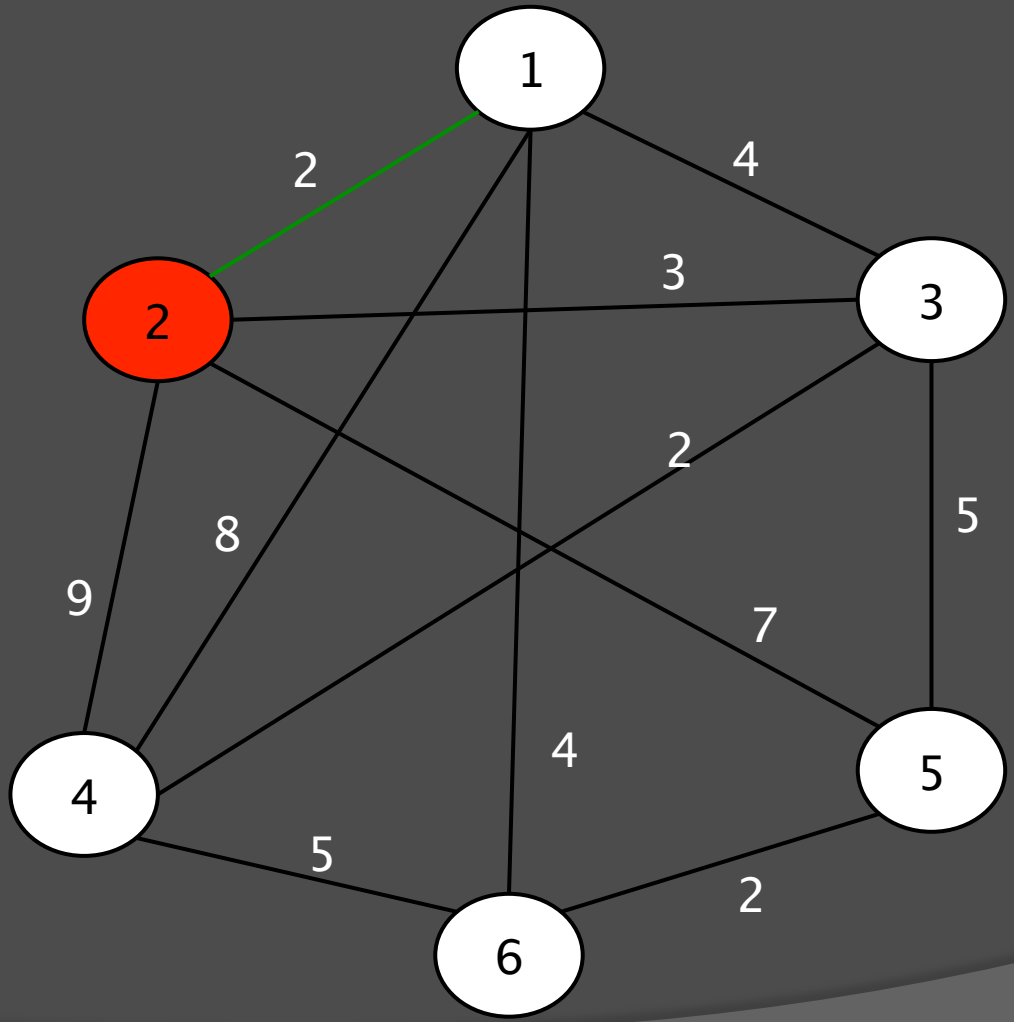
1	2	3	4	5	6
1	0	0	0	0	0

Parent

1	2	3	4	5	6
0	1	1	1	0	1

Dist

1	2	3	4	5	6
0	2	4	8	∞	4



Choose unvisited node with lowest distance (node 2)

Vis

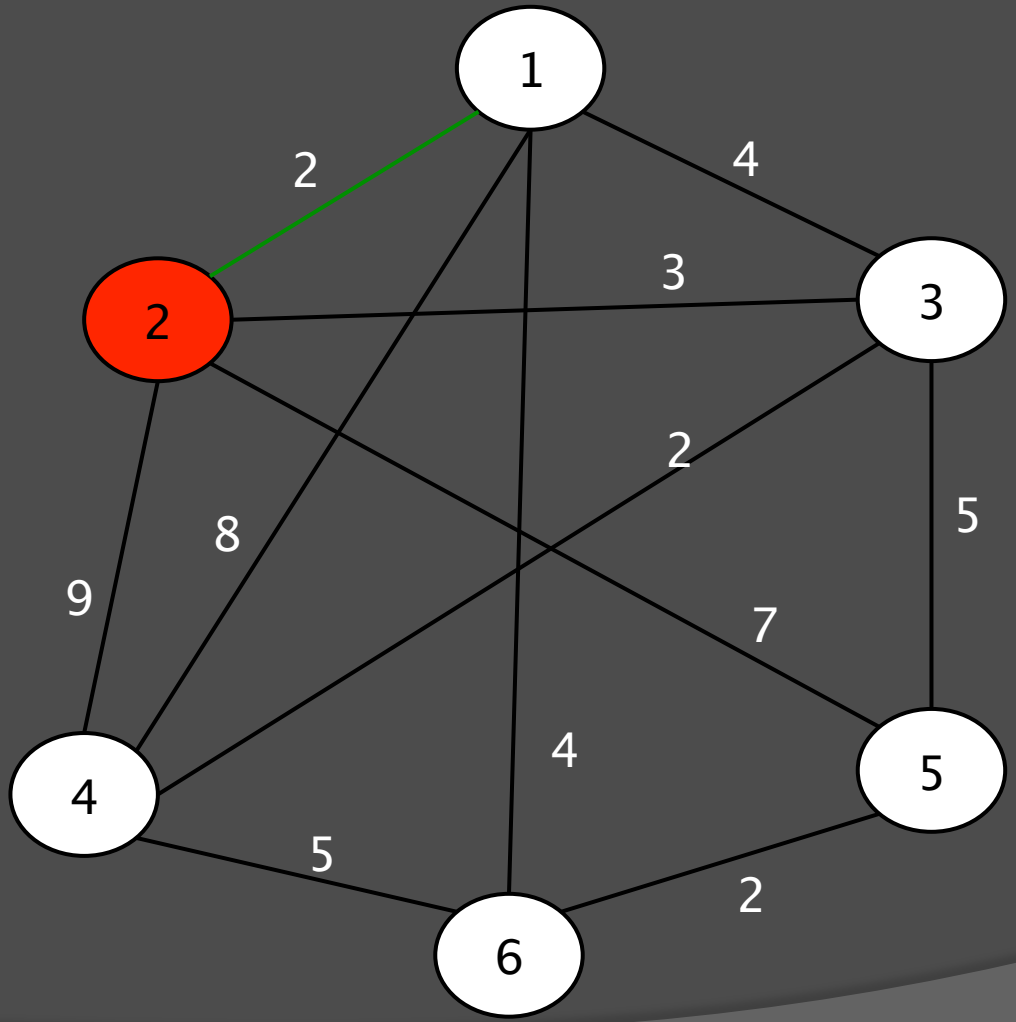
1	2	3	4	5	6
1	0	0	0	0	0

Parent

1	2	3	4	5	6
0	1	1	1	0	1

Dist

1	2	3	4	5	6
0	2	4	8	∞	4



Mark node 2 as visited, update distance to neighbors (ONLY edge cost). Change distance if it is **less**.

Vis

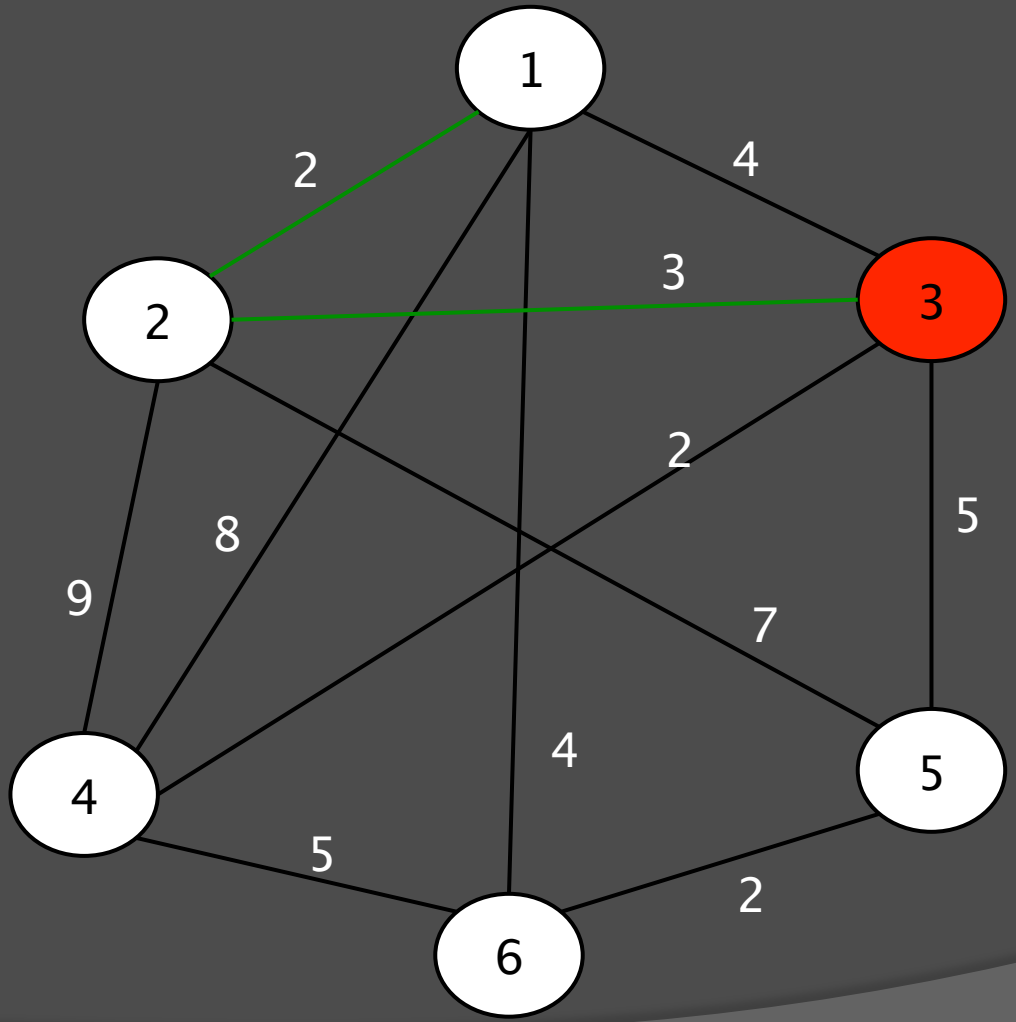
1	2	3	4	5	6
1	1	0	0	0	0

Parent

1	2	3	4	5	6
0	1	2	1	2	1

Dist

1	2	3	4	5	6
0	2	3	8	7	4



Choose unvisited node with lowest distance (node 3)

Vis

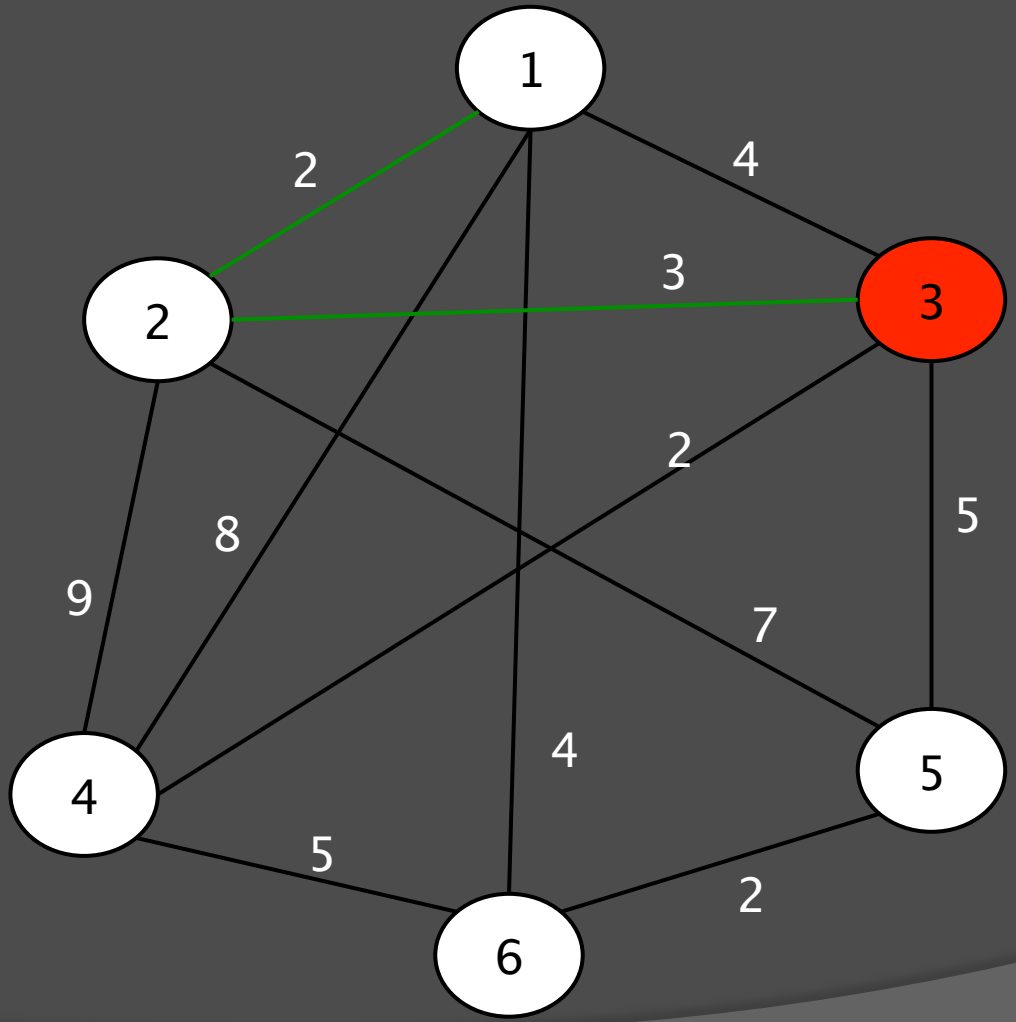
1	2	3	4	5	6
1	1	0	0	0	0

Parent

1	2	3	4	5	6
0	1	2	1	2	1

Dist

1	2	3	4	5	6
0	2	3	8	7	4



Mark node 3 as visited, update distance to neighbors

Vis

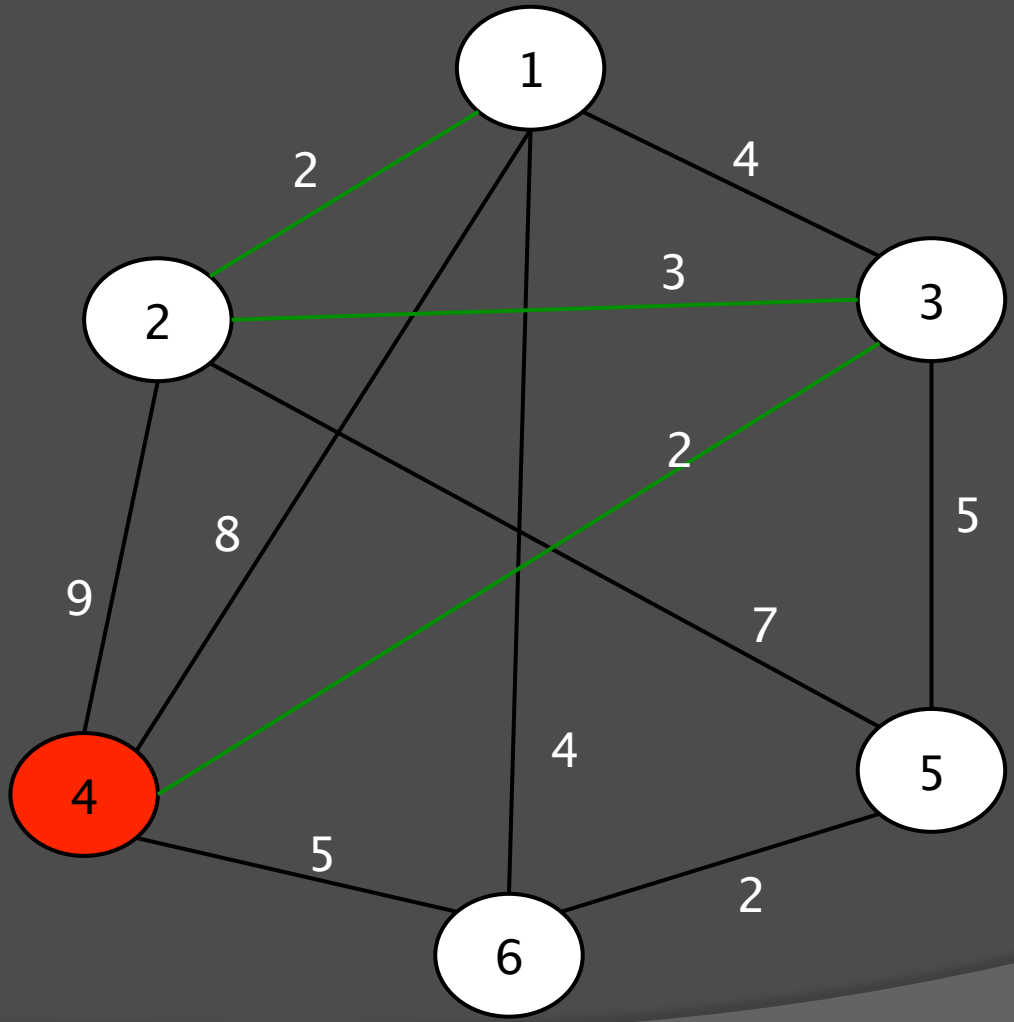
1	2	3	4	5	6
1	1	1	0	0	0

Parent

1	2	3	4	5	6
0	1	2	3	3	1

Dist

1	2	3	4	5	6
0	2	3	2	5	4



Choose unvisited node with lowest distance (node 4)

Vis

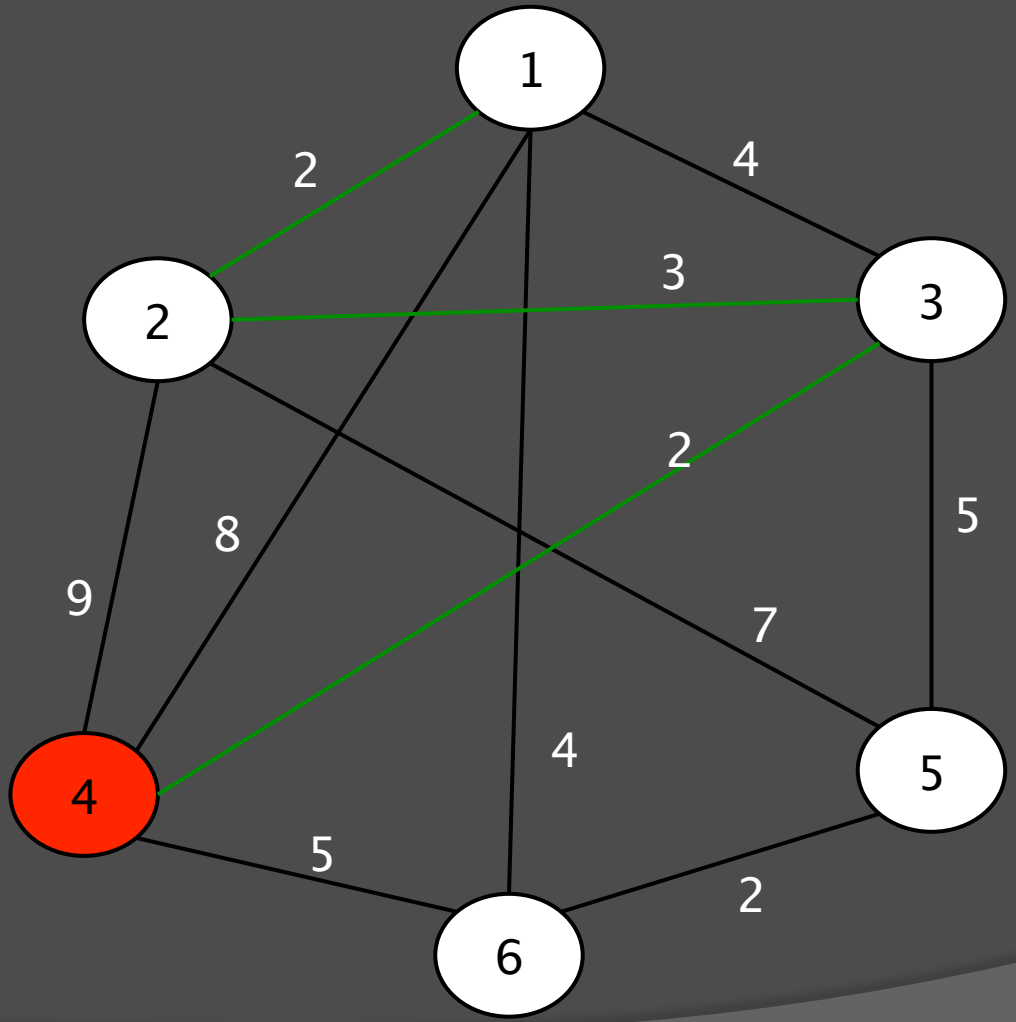
1	2	3	4	5	6
1	1	1	0	0	0

Parent

1	2	3	4	5	6
0	1	2	3	3	1

Dist

1	2	3	4	5	6
0	2	3	2	5	4



Mark node 4 as visited, update distance to neighbors

Vis

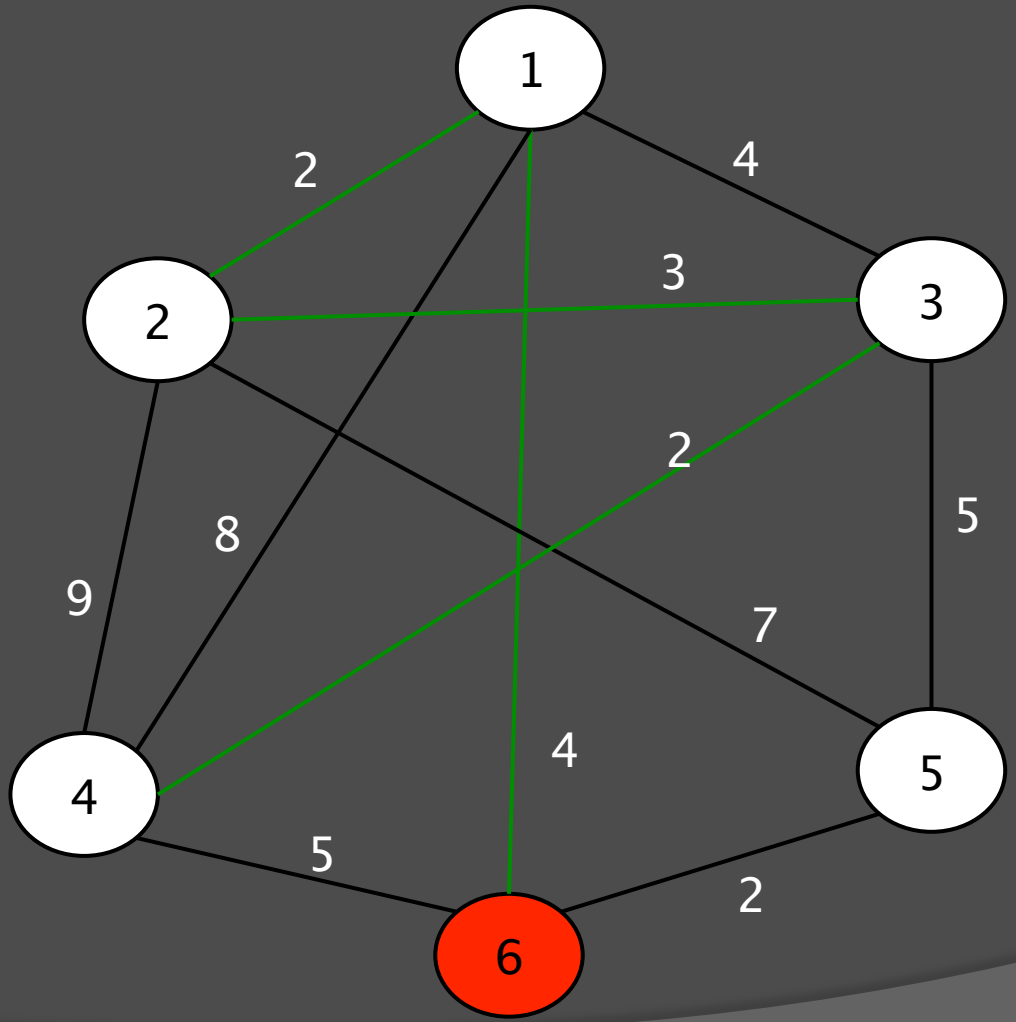
1	2	3	4	5	6
1	1	1	1	0	0

Parent

1	2	3	4	5	6
0	1	2	3	3	1

Dist

1	2	3	4	5	6
0	2	3	2	5	4



Choose unvisited node with lowest distance (node 6)

Vis

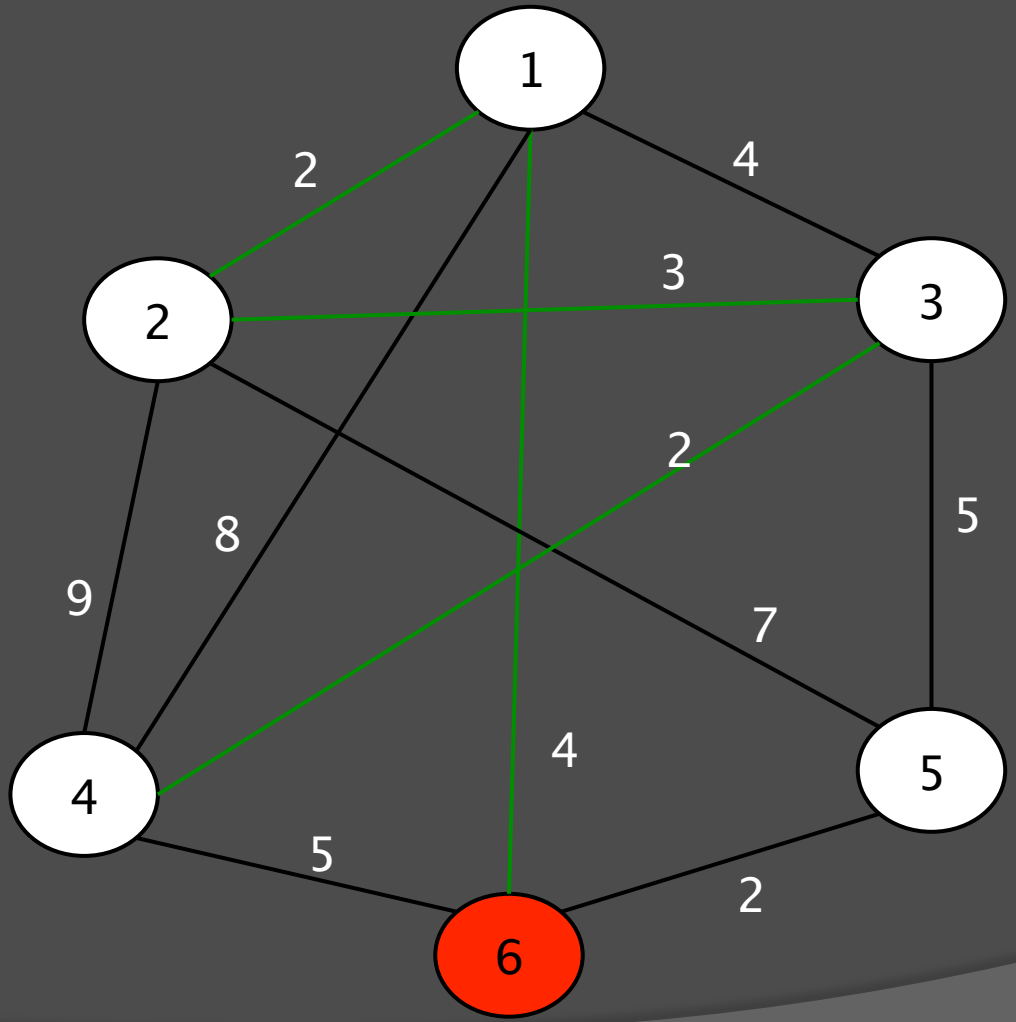
1	2	3	4	5	6
1	1	1	1	0	0

Parent

1	2	3	4	5	6
0	1	2	3	3	1

Dist

1	2	3	4	5	6
0	2	3	2	5	4



Mark node 6 as visited, update distance to neighbors

Vis

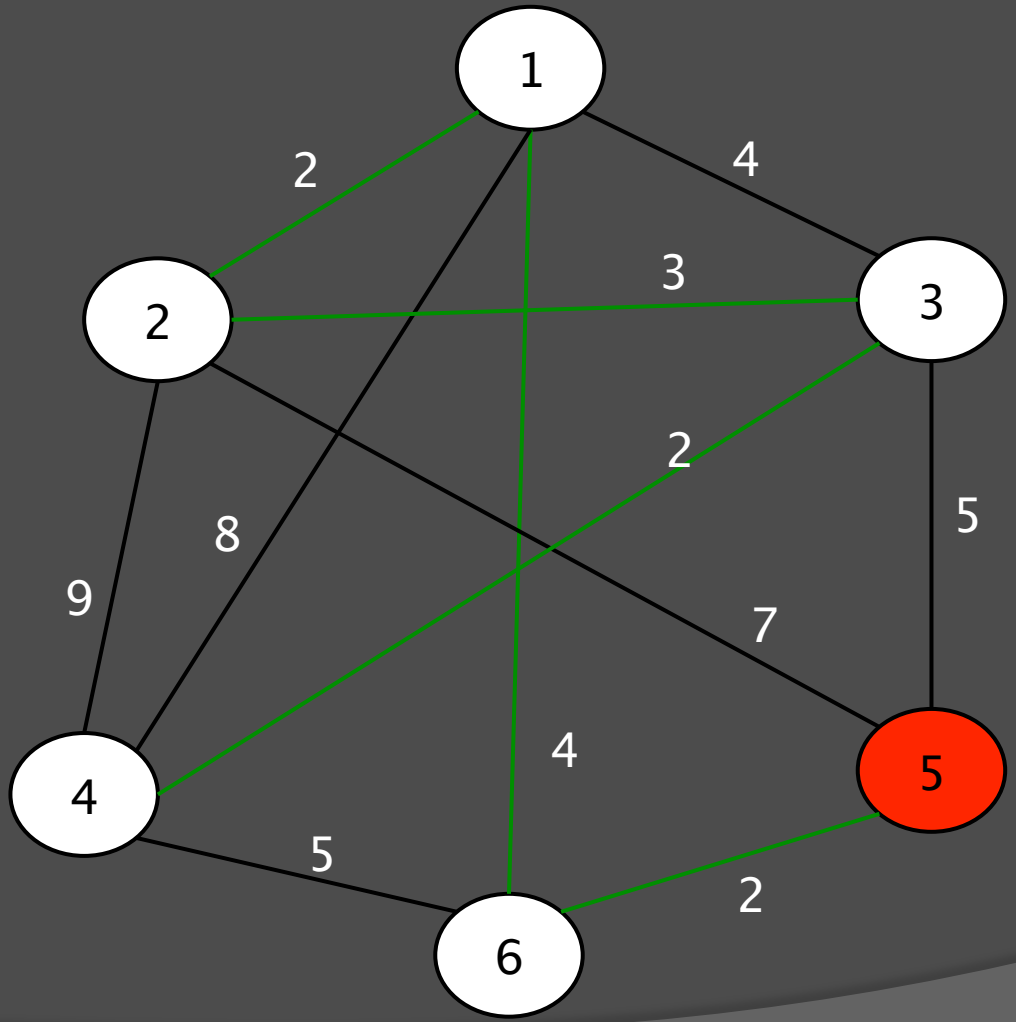
1	2	3	4	5	6
1	1	1	1	0	1

Parent

1	2	3	4	5	6
0	1	2	3	6	1

Dist

1	2	3	4	5	6
0	2	3	2	2	4



Choose unvisited node with lowest distance (node 5)

Vis

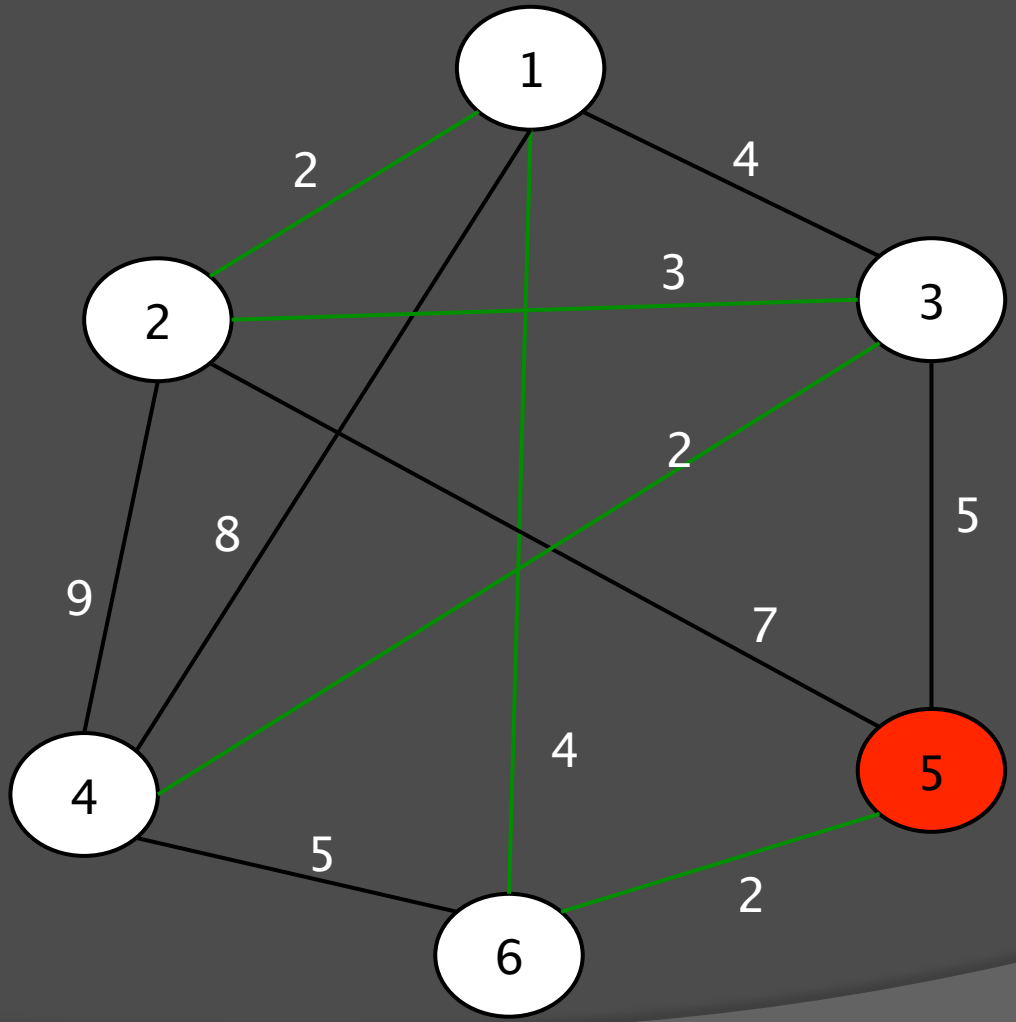
1	2	3	4	5	6
1	1	1	1	0	1

Parent

1	2	3	4	5	6
0	1	2	3	6	1

Dist

1	2	3	4	5	6
0	2	3	2	2	4



Mark node 5 as visited, update distance to neighbors

Vis

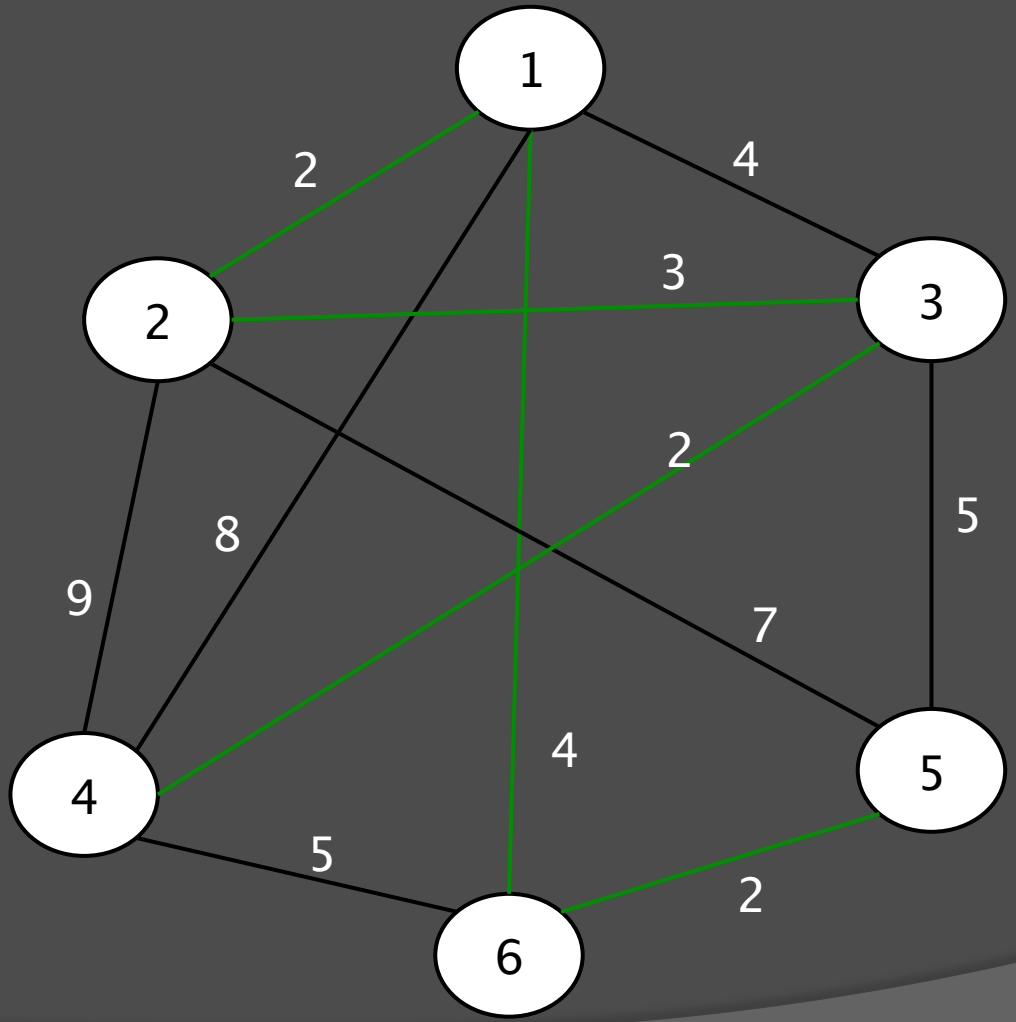
1	2	3	4	5	6
1	1	1	1	1	1

Parent

1	2	3	4	5	6
0	1	2	3	6	1

Dist

1	2	3	4	5	6
0	2	3	2	2	4



Done!
The green edges are part of the MST.
You can keep track of exact edges using the parent array

Vis

1	2	3	4	5	6
1	1	1	1	1	1

Parent

1	2	3	4	5	6
0	1	2	3	6	1

Dist

1	2	3	4	5	6
0	2	3	2	2	4

Prim's Algorithm – Pseudocode

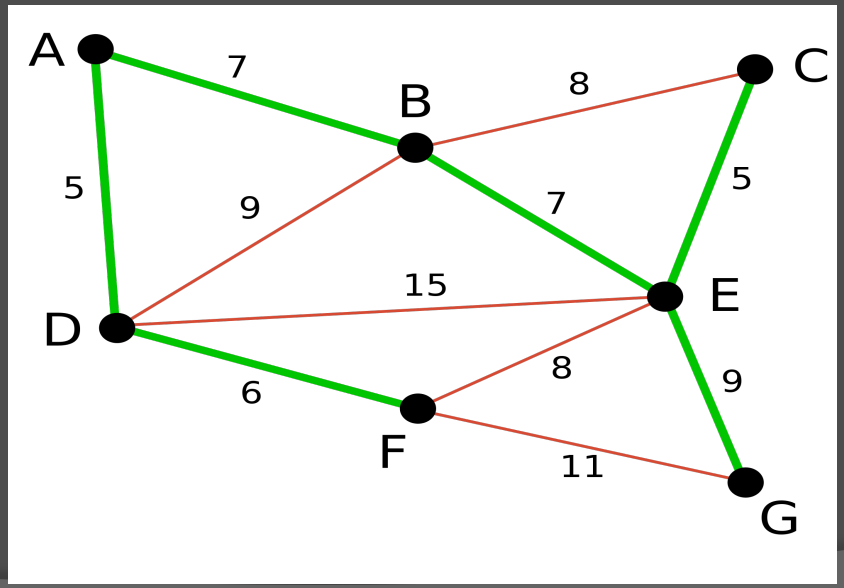
```
bool vis[]; int dist[]; node parent[];
void prim(node root){
        set all dist to infinity;
        dist[root] = 0;
        while (there are nodes left){
                node cur = unvisited node with lowest distance;
                mark cur as visited;
                //do what you want with edge (use parent[])
                for (every node u adjacent to cur){
                        if (u is not visited){
                                if (cost[cur][u] < dist[u]){
                                        dist[u] = cost[cur][u];
                                        parent[u] = cur;
                                }
                        }
                }
        }
}
```

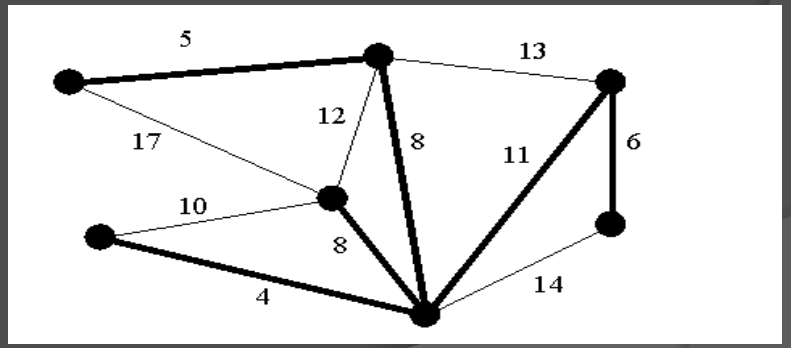
Prim's Algorithm – Priority Queue

```
int dist[], parent[]; bool vis[]
priority_queue<pair<int,node> > q;
void prim(node root){
            //init vis and dist here
            q.push(make_pair(0,root));
            while (q is not empty){
                        pair cur_pair = q.top(); q.pop();
                        int cur_dist = cur_pair.first; node cur = cur_pair.second;
                        if (cur is visited) continue;
                        mark cur as visited;
                        //do what you want with edge (use parent[])
                        for (every node u adjacent to cur){
                                    if (u is not visited){
                                                if (edges[cur][u] < dist[u]){
                                                            dist[u] = edges[cur][u];
                                                            parent[u] = cur;
                                                            q.push(make_pair(dist[u],u));
                                                }
                                    }
                        }
            }
}
```

Try it Out!

- Given a connected, weighted graph, print all of the edges in its minimum spanning tree in any order.





THANK YOU!