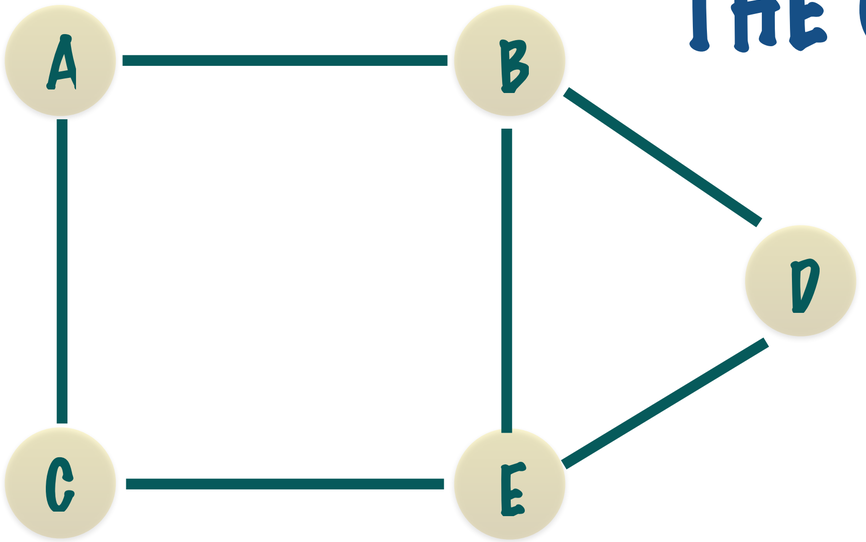
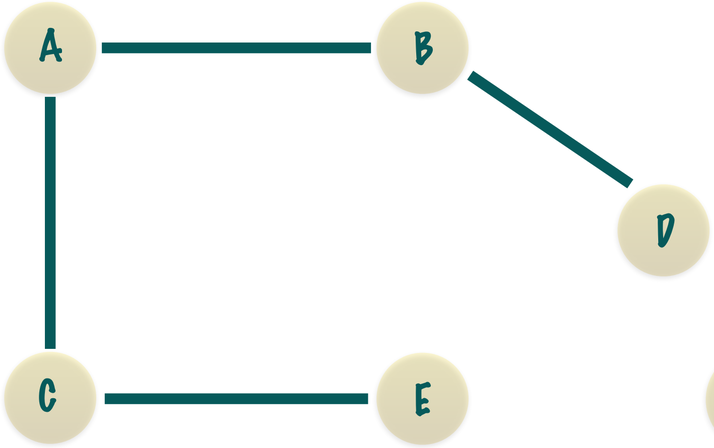
THE GRAPH

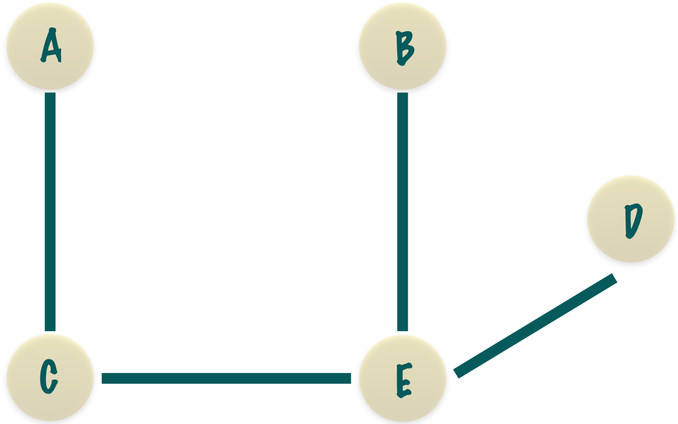
MINIMAL SPANNING TREE

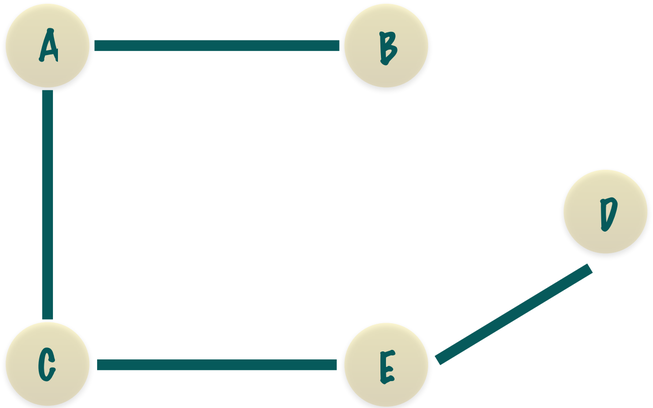
THE GRAPH SPANNING TREE

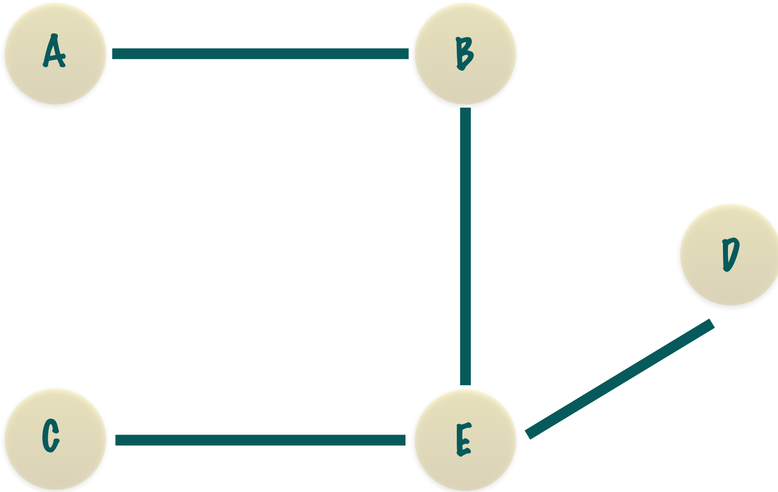


SPANNING TREE IS A SUBGRAPH THAT CONTAINS ALL THE VERTICES AND IS ALSO A TREE

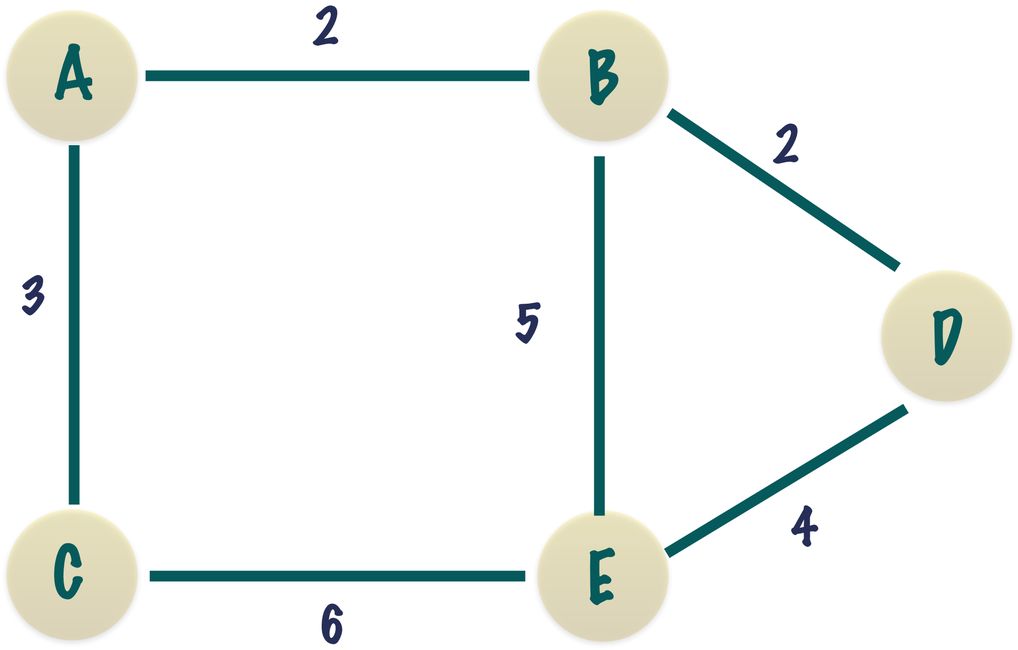






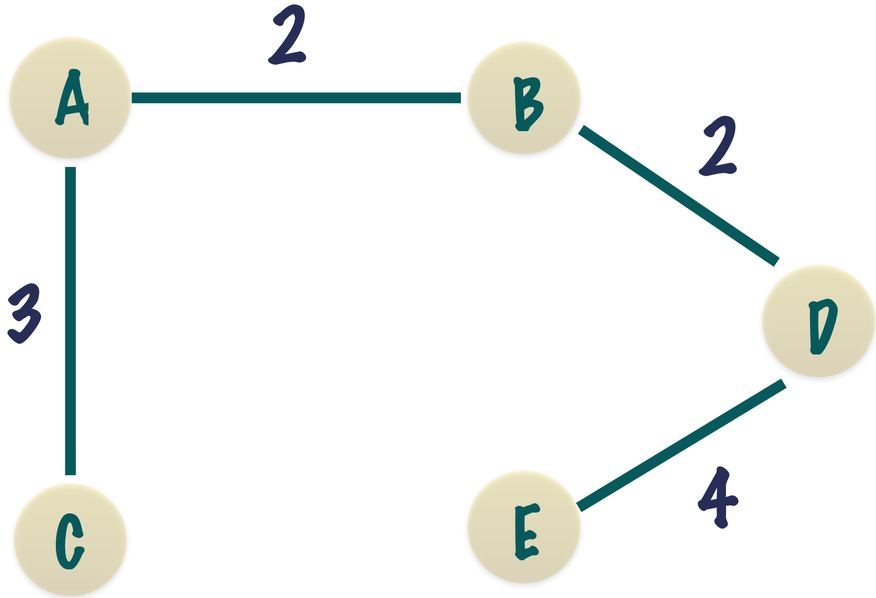


THE GRAPH SPANNING TREE

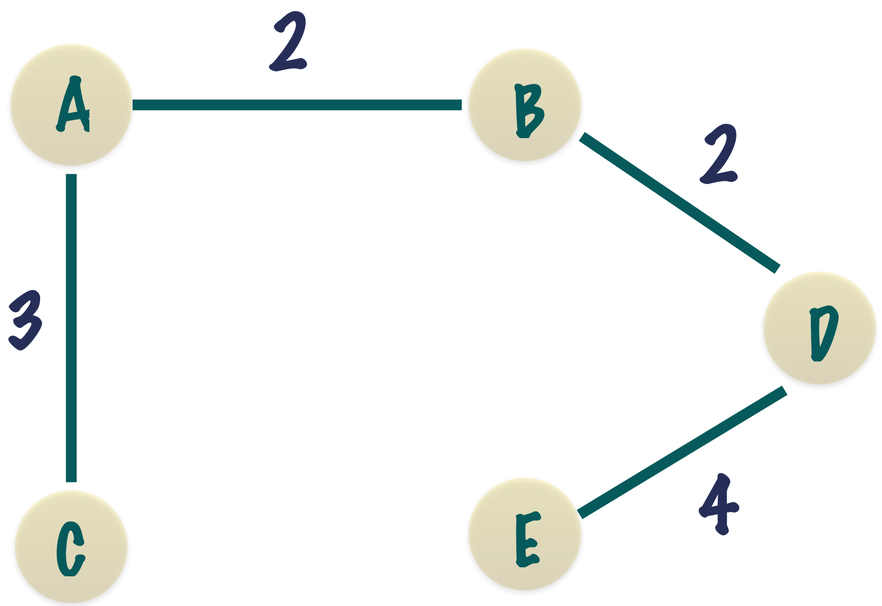


MINIMUM SPANNING TREE IS A SUBGRAPH THAT CONTAINS ALL THE VERTICES AND IS ALSO A TREE

WITH MINIMUM TOTAL COST



THE GRAPH



MINIMAL SPANNING TREE

MINIMUM SPANNING TREE IS A SUBGRAPH THAT CONTAINS ALL THE VERTICES AND IS ALSO A TREE

WITH MINIMUM TOTAL COST

THE ALGORITHM THAT WE WILL STUDY WORKS WELL FOR UNDIRECTED CONNECTED GRAPHS

(SHORTEST WEIGHT PATH FROM A TO B IS SAME AS B TO A)

THE GRAPH MINIMAL SPANNING TREE

THE ALGORITHM THAT WE WILL STUDY WORKS WELL FOR UNDIRECTED CONNECTED GRAPHS

(SHORTEST WEIGHT PATH FROM A TO B IS SAME AS B TO A)

THIS IS PRIM'S ALGORITHM

IT'S VERY SIMILAR TO DIJKSTRA'S SHORTEST PATH FOR WEIGHTED GRAPHS

THE GRAPH MINIMAL SPANNING TREE

THIS IS PRIM'S ALGORITHM

THERE ARE 2 PARTS TO THIS ALGORITHM:

1. GENERATE THE DISTANCE TABLE USING DIJKSTRA'S LIKE ALGORITHM USING ANY VERTEX AS THE SOURCE

2. USE THIS DISTANCE TABLE TO GET PATHS TO ALL OTHER VERTICES FROM THE ARBITRARILY CHOSEN SOURCE

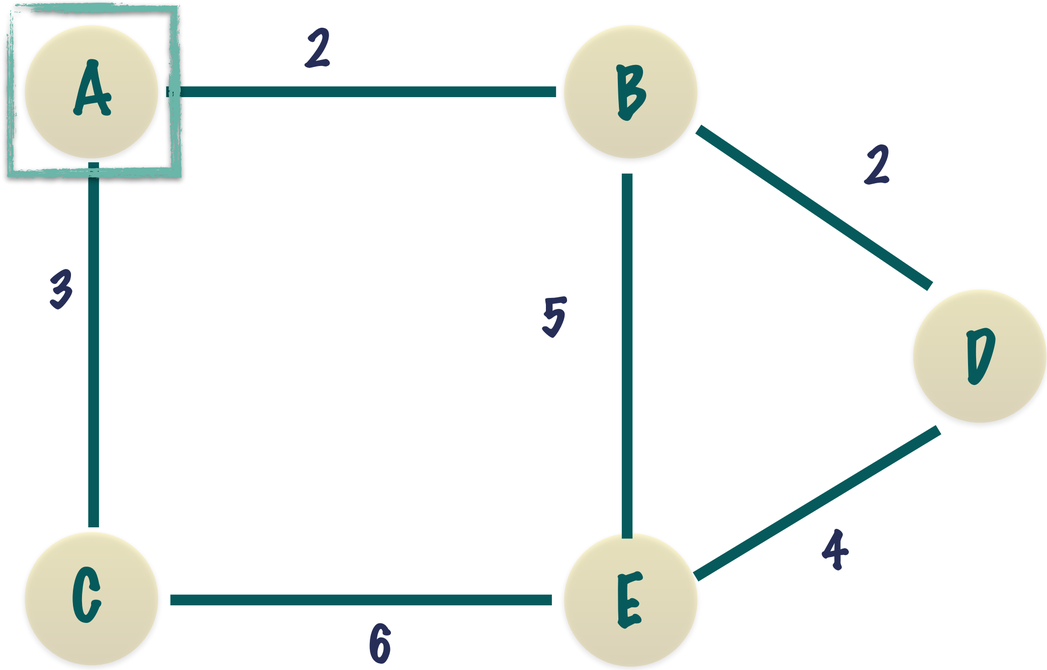
THERE IS ONE IMPORTANT DIFFERENCE THOUGH...

THE GRAPH MINIMAL SPANNING TREE

THERE IS ONE IMPORTANT DIFFERENCE THOUGH...

WE WANT TO MINIMIZE TOTAL DISTANCE AND NOT
JUST DISTANCE FROM ONE SPECIFIC VERTEX!

(IN DJIKTRA'S ALGORITHM, YOU SPECIFY ONE SOURCE VERTEX)



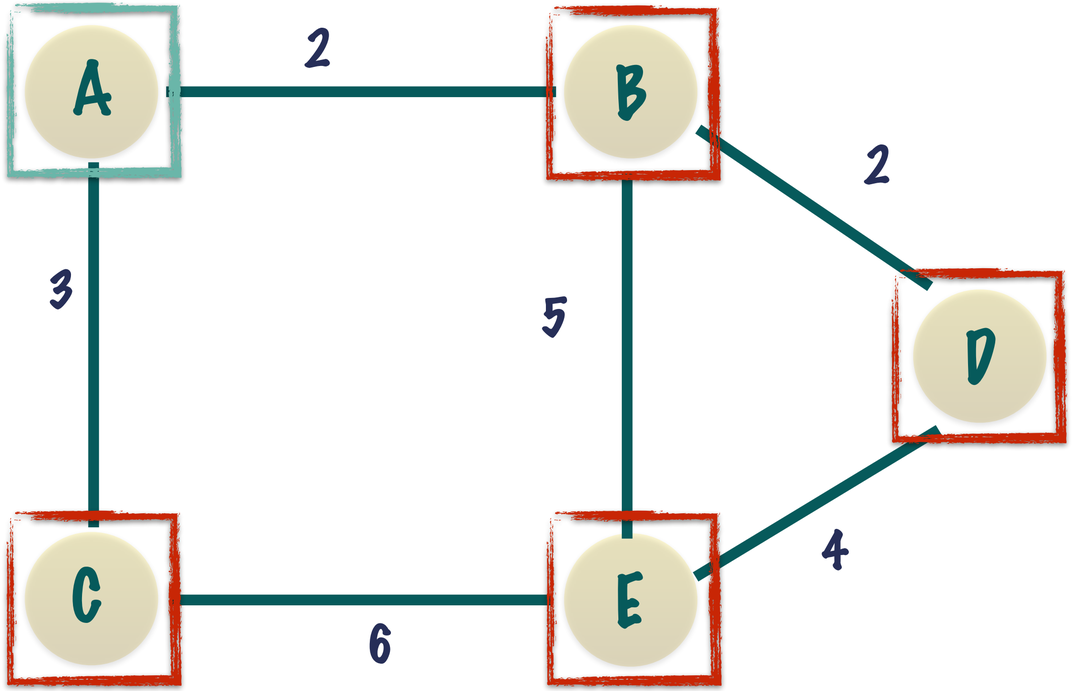
THE GRAPH MINIMAL SPANNING TREE

THERE IS ONE IMPORTANT DIFFERENCE THOUGH...

WE WANT TO MINIMIZE TOTAL DISTANCE AND NOT JUST DISTANCE FROM ONE SPECIFIC VERTEX!

(IN DJIKTRA'S ALGORITHM, YOU SPECIFY ONE SOURCE VERTEX)

IN PRIM'S ALGORITHM START WITH ANY RANDOM VERTEX!



THE GRAPH MINIMAL SPANNING TREE

1. GENERATE THE DISTANCE TABLE USING DIJKSTRA'S LIKE ALGORITHM USING ANY VERTEX AS THE SOURCE

DISTANCE [NEIGHBOUR] = DISTANCE [VERTEX] + WEIGHT OF EDGE [VERTEX, NEIGHBOUR]

DISTANCE [NEIGHBOUR] =
WEIGHT OF EDGE [VERTEX,NEIGHBOUR]

WE ONLY CARE ABOUT THE WEIGHT OF THE EDGE NOT THE CUMULATIVE DISTANCE FROM THE SOURCE

THE GRAPH MINIMAL SPANNING TREE

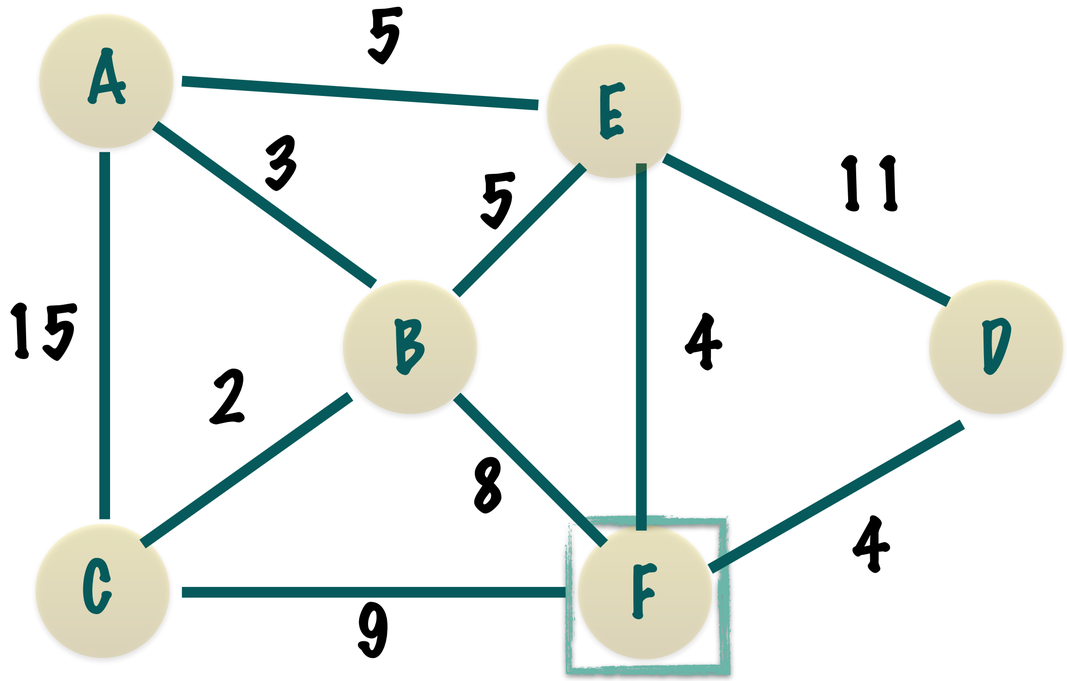
AN EDGE IS CHOSEN TO BE PART OF THE SPANNING TREE IF:

1. THE VERTEX IS THE CLOSEST VERTEX AT THE CURRENT STEP (I.E. CONNECTED BY THE LOWEST WEIGHTED EDGE)

2. THE VERTEX IS NOT PART OF THE SPANNING TREE ALREADY

THE GRAPH MINIMAL SPANNING TREE

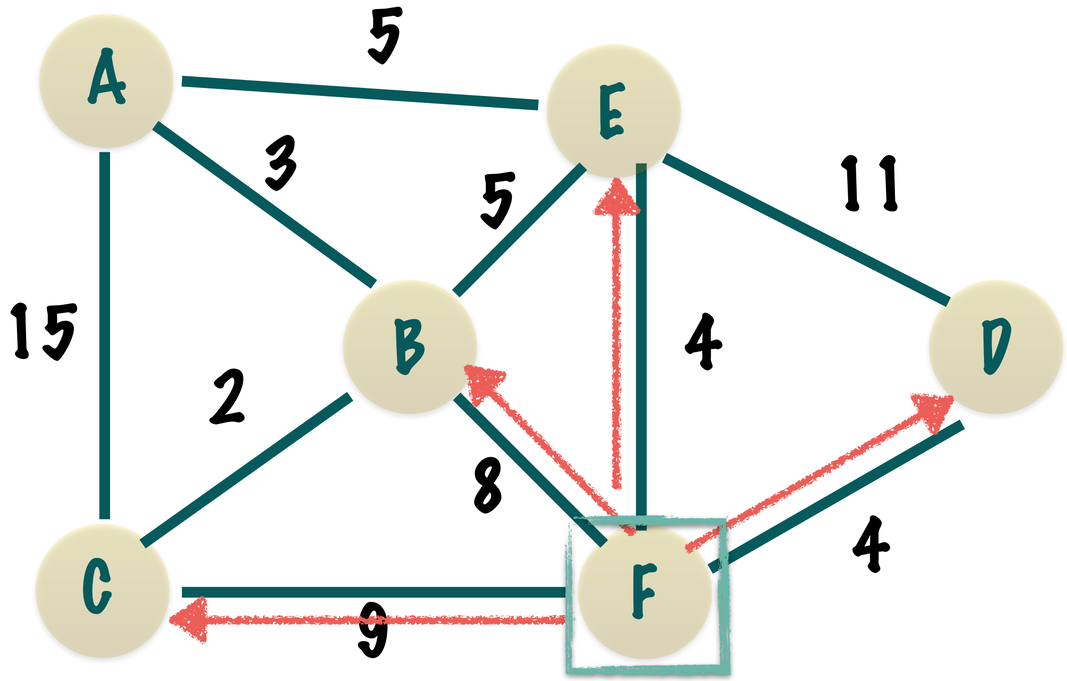
LET'S START WITH ANY VERTEX - CHOOSE F



VERTEX	DISTANCE	LAST VERTEX
A	INF	
B	INF	
C	INF	
D	INF	
E	INF	
F	0	F

THE GRAPH MINIMAL SPANNING TREE

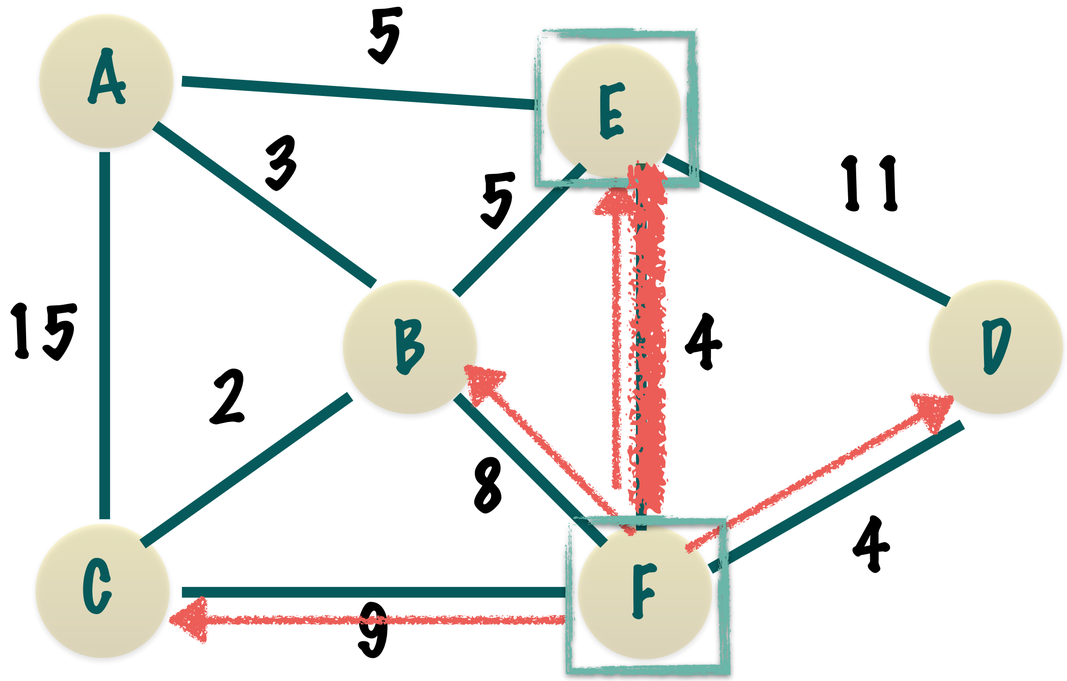
FIND ALL THE VERTICES ADJACENT TO F



VERTEX	DISTANCE	LAST VERTEX
A	INF	
B	8	F
C	9	F
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

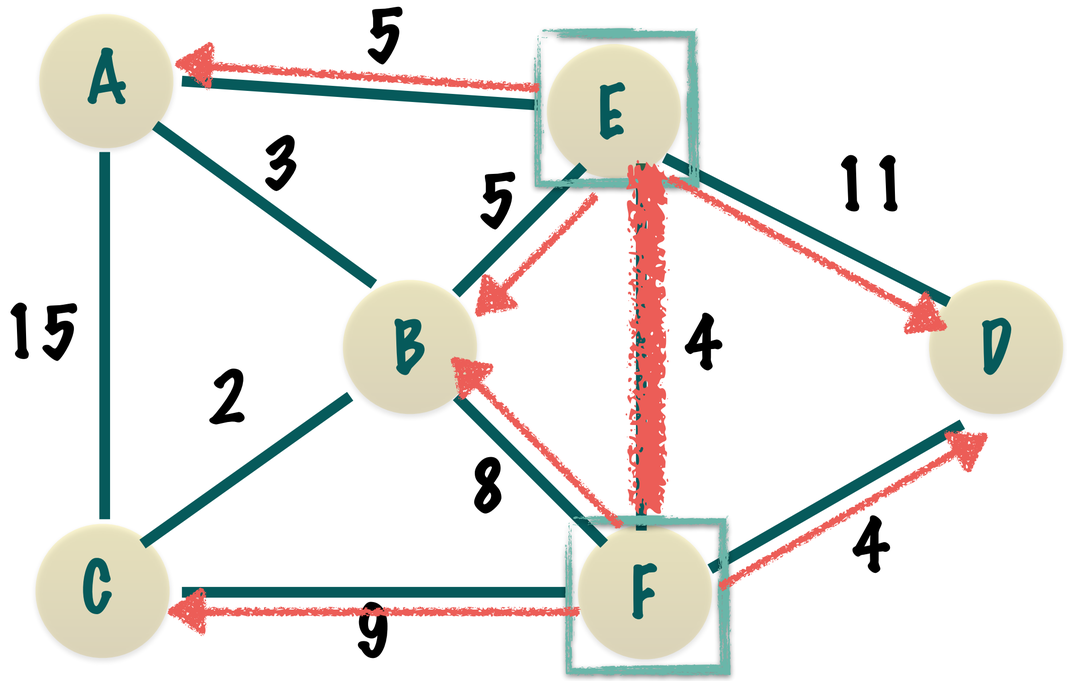
FIND ALL THE VERTICES ADJACENT TO F



VERTEX	DISTANCE	LAST VERTEX
A	INF	
B	8	F
C	9	F
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

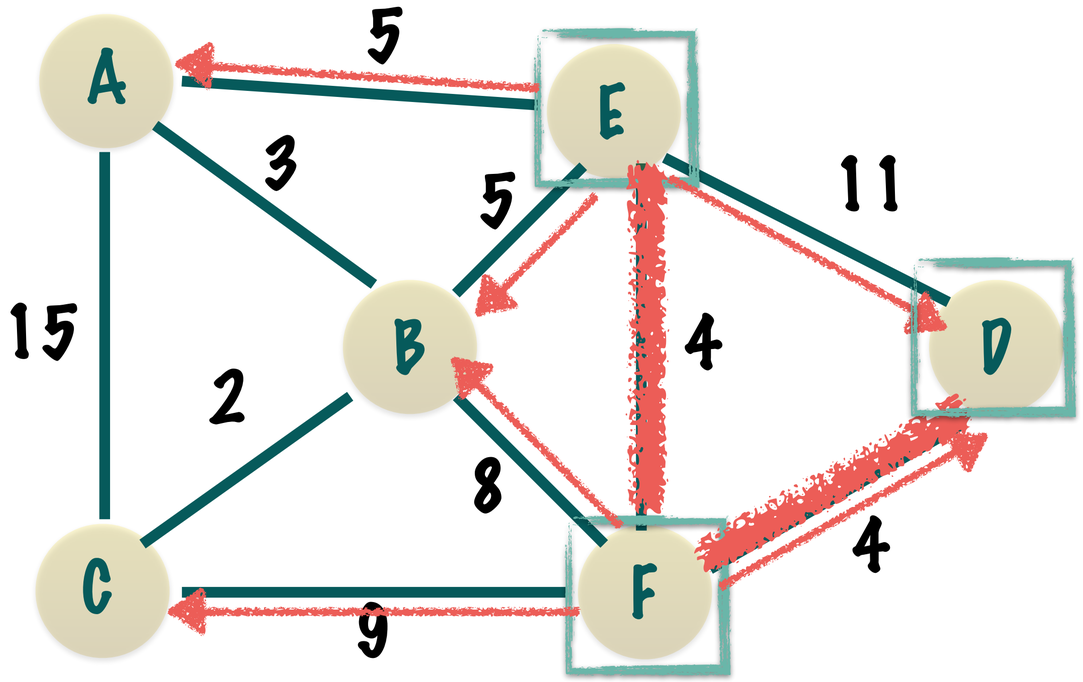
FIND ALL THE VERTICES ADJACENT TO F
AND E



VERTEX	DISTANCE	LAST VERTEX
A	5	E
B	5	E
C	9	E
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

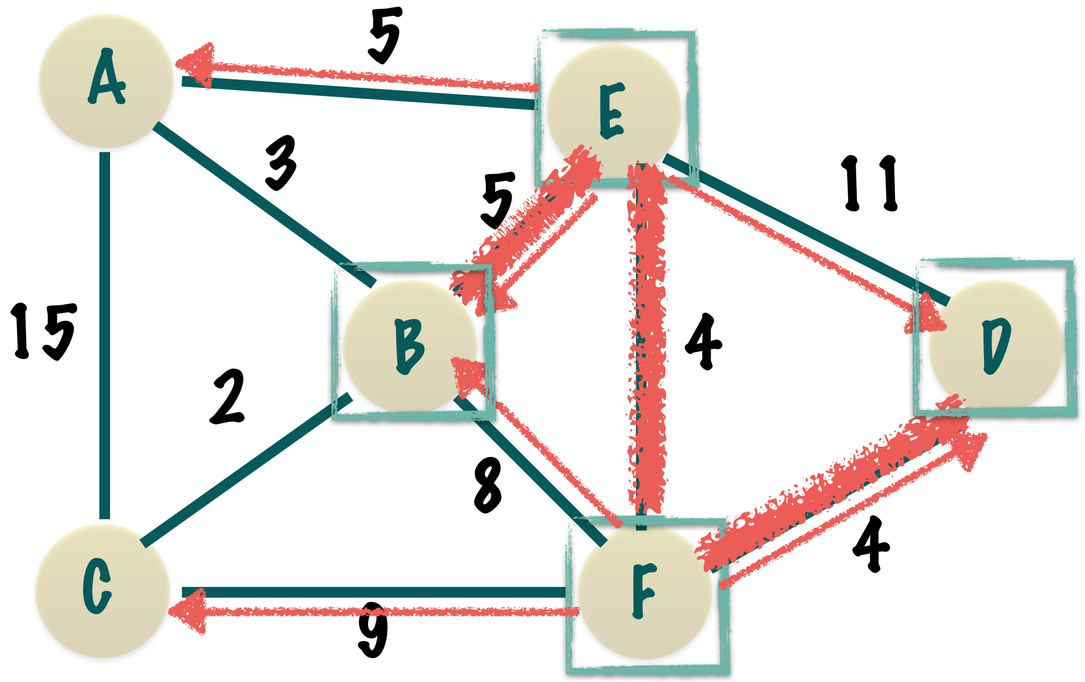
FIND ALL THE VERTICES ADJACENT TO F AND E



VERTEX	DISTANCE	LAST VERTEX
A	5	E
B	5	E
C	9	F
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

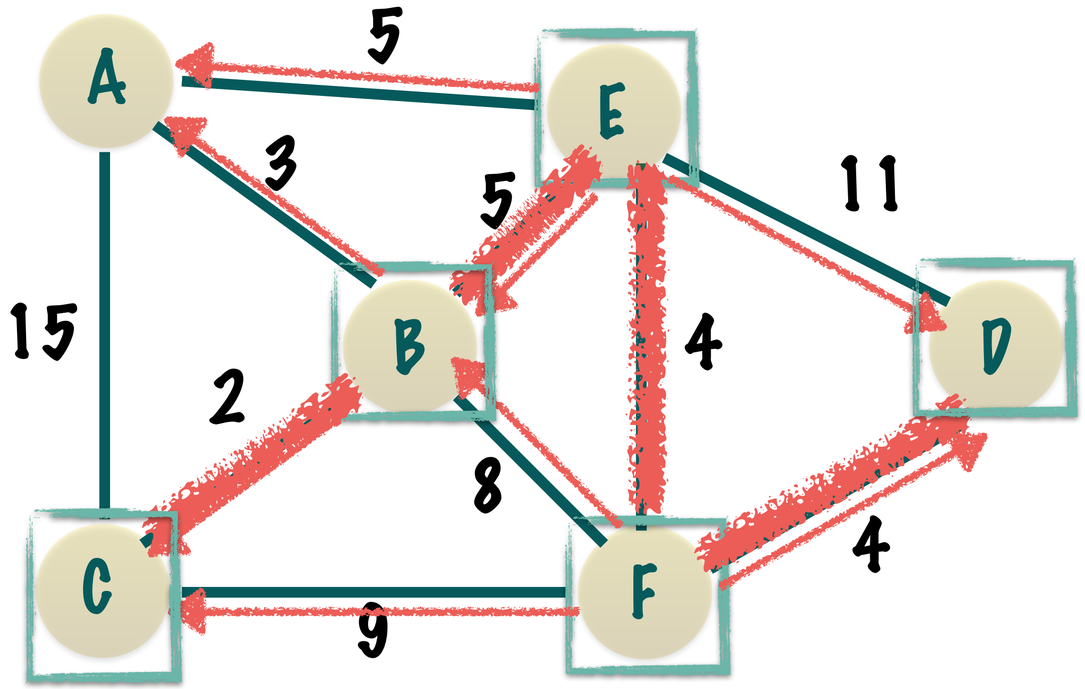
FIND ALL THE VERTICES ADJACENT TO F, E AND D



VERTEX	DISTANCE	LAST VERTEX
A	5	E
B	5	E
C	9	F
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

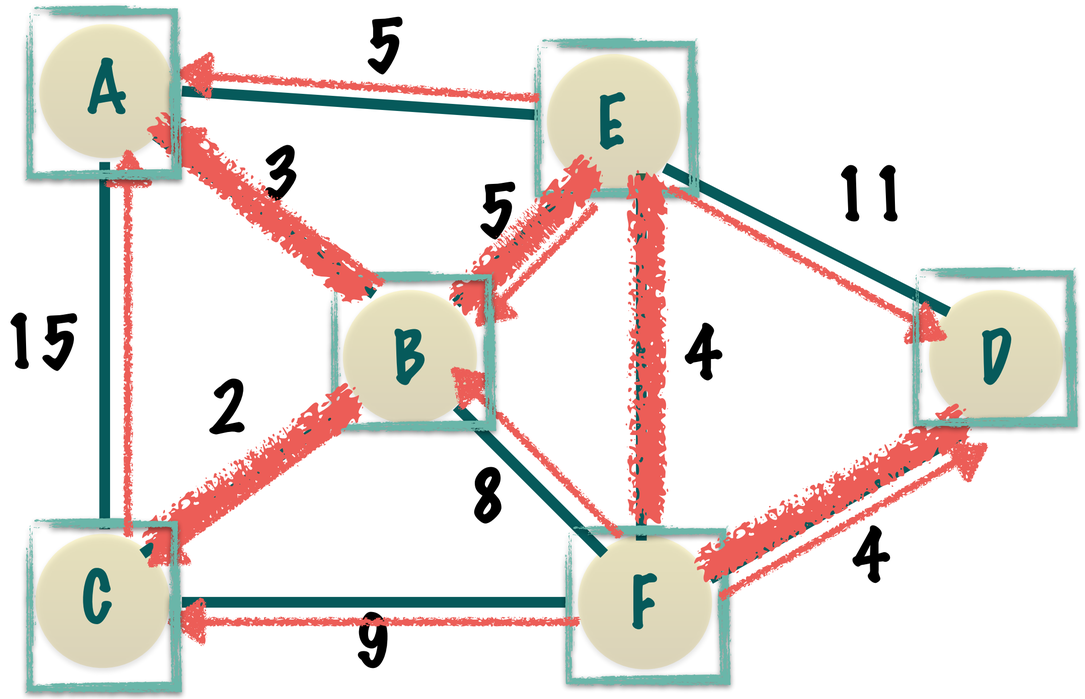
FIND ALL THE VERTICES ADJACENT TO F, E, D AND B



VERTEX	DISTANCE	LAST VERTEX
A	3	B
B	5	E
C	2	B
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

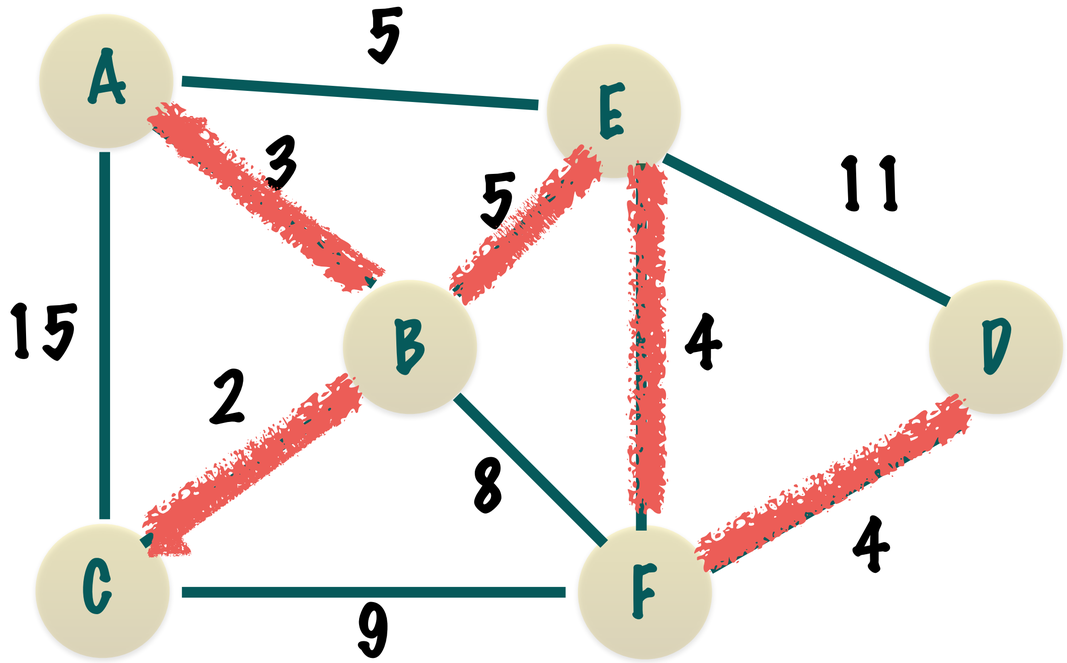
FIND ALL THE VERTICES ADJACENT TO F, E, D, B AND C



VERTEX	DISTANCE	LAST VERTEX
A	3	B
B	5	E
C	2	B
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

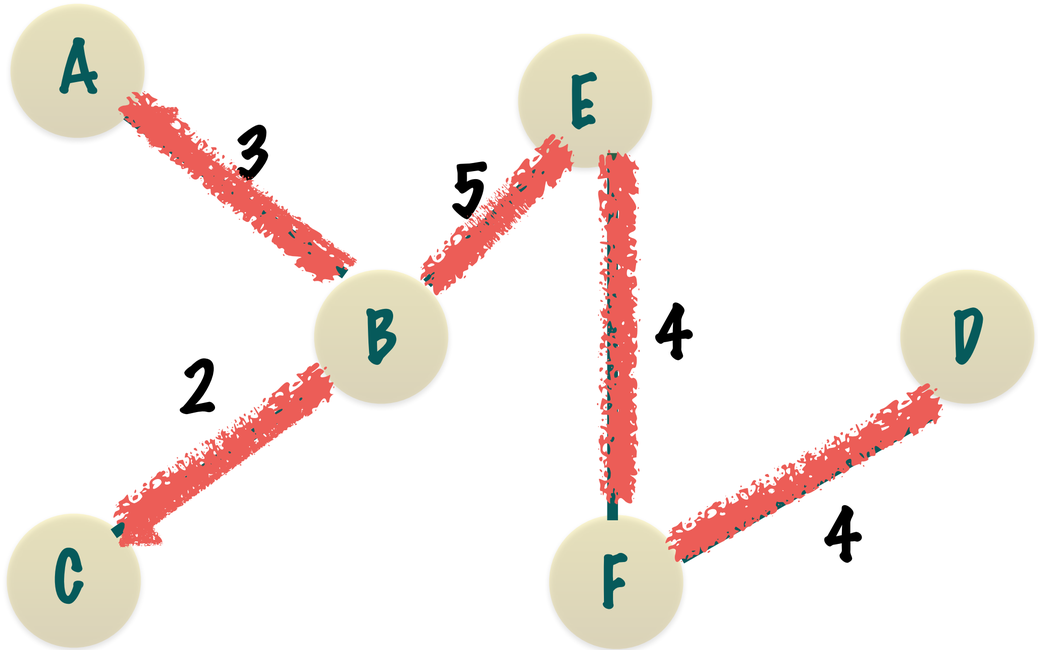
THE MINIMAL SPANNING TREE USING PRIM'S ALGORITHM



VERTEX	DISTANCE	LAST VERTEX
A	3	B
B	5	E
C	2	B
D	4	F
E	4	F
F	0	F

THE GRAPH MINIMAL SPANNING TREE

THE MINIMAL SPANNING TREE USING PRIM'S ALGORITHM



THE GRAPH MINIMAL SPANNING TREE

AN EDGE IS CHOSEN TO BE PART OF THE SPANNING TREE IF:

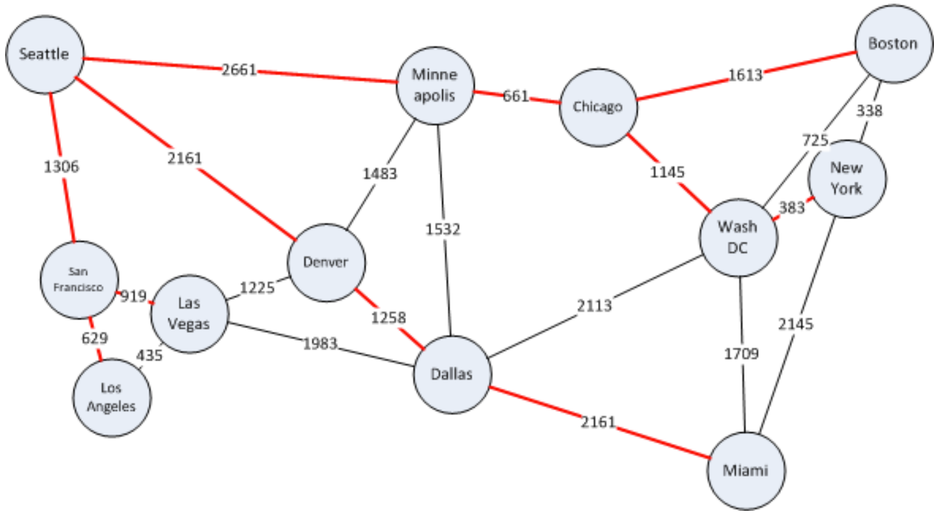
1. THE VERTEX IS THE CLOSEST VERTEX AT THE CURRENT STEP (I.E. CONNECTED BY THE LOWEST WEIGHTED EDGE)

2. THE VERTEX IS NOT PART OF THE SPANNING TREE ALREADY

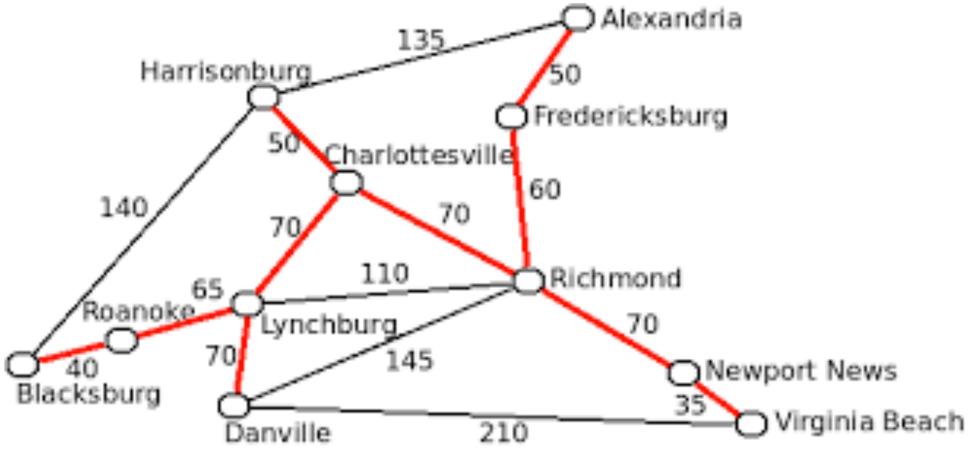
THE GRAPH MINIMAL SPANNING TREE

WHEN WOULD YOU USE THE MINIMAL SPANNING TREE VS THE SHORTEST PATH ALGORITHM?

SHORTEST PATH TO FIND THE SHORTEST ROUTE TO TRAVEL FROM A SOURCE TO GET TO A DESTINATION



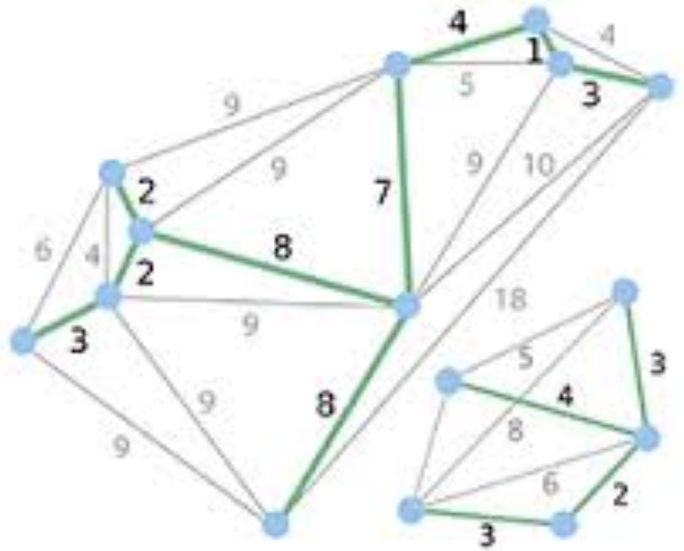
MINIMAL SPANNING TREE TO FIND THE CHEAPEST WAY TO INTERCONNECT CITIES



FOR INSTANCE SEATTLE TO MIAMI

THE GRAPH MINIMAL SPANNING TREE

TO FIND THE MINIMAL SPANNING TREE FOR A FOREST (UNCONNECTED TREES) JUST RUN PRIM'S ALGORITHM ON EACH CONNECTED GRAPH



THE GRAPH MINIMAL SPANNING TREE

THE ALGORITHM'S COMPLEXITY IS SIMILAR TO DJIKTRA'S ALGORITHM

RUNNING TIME IS : O(E LOG V)
[IF BINARY HEAPS ARE USED FOR PRIORITY QUEUE]

RUNNING TIME IS : O(E + V*V)
[IF ARRAY IS USED FOR PRIORITY QUEUE]

DISTANCE TABLE DATA STRUCTURE

```java
/**
 * A class which holds the distance information of any vertex.
 * The distance specified is the distance from the source node
 * and the last vertex is the last vertex just before the current
 * one while traversing from the source node.
 */
public static class DistanceInfo {

    private int distance;
    private int lastVertex;

    public DistanceInfo() {
        // The initial distance to all nodes is assumed
        // infinite.
        distance = Integer.MAX_VALUE;
        lastVertex = -1;
    }

    public int getDistance() {
        return distance;
    }

    public int getLastVertex() {
        return lastVertex;
    }

    public void setDistance(int distance) {
        this.distance = distance;
    }

    public void setLastVertex(int lastVertex) {
        this.lastVertex = lastVertex;
    }
}
```

FOR EVERY VERTEX STORE

1. THE DISTANCE TO THE VERTEX FROM THE SOURCE
2. THE LAST VERTEX IN THE PATH FROM THE SOURCE

NOTE THAT WE SET THE INITIAL DISTANCE OF ALL THE NODES TO INFINITE, AS WE FIND OUR PATH TO THESE NODES WE'LL UPDATE IT WITH THE RIGHT DISTANCE

VERTEX INFO - TRACK THE CURRENT DISTANCE FOR EVERY VERTEX

```java
/**
 * A simple class which holds the vertex id and the weight of
 * the edge that leads to that vertex from its neighbour
 */
public static class VertexInfo {

    private int vertexId;
    private int distance;

    public VertexInfo(int vertexId, int distance) {
        this.vertexId = vertexId;
        this.distance = distance;
    }

    public int getVertexId() {
        return vertexId;
    }

    public int getDistance() {
        return distance;
    }
}
```

TRACK EVERY VERTEX ID AND IT'S DISTANCE FROM THE SOURCE

GETTERS FOR THE INFORMATION IN THE VERTEX INFO

BUILD THE DISTANCE TABLE AND SPANNING TREE- SETUP

```
public static void spanningTree(Graph graph, int source) {
    Map<Integer, DistanceInfo> distanceTable = new HashMap<>();
    PriorityQueue<VertexInfo> queue = new PriorityQueue<>(new Comparator<VertexInfo>() {
        @Override
        public int compare(VertexInfo v1, VertexInfo v2) {
            return ((Integer) v1.getDistance()).compareTo(v2.getDistance());
        }
    });

    for (int j = 0; j < graph.getNumVertices(); j++) {
        distanceTable.put(j, new DistanceInfo());
    }

    distanceTable.get(source).setDistance(0);
    distanceTable.get(source).setLastVertex(source);

    Map<Integer, VertexInfo> vertexInfoMap = new HashMap<>();

    VertexInfo sourceVertexInfo = new VertexInfo(source, 0);
    queue.add(sourceVertexInfo);
    vertexInfoMap.put(source, sourceVertexInfo);

    Set<String> spanningTree = new HashSet<>();
    Set<Integer> visitedVertices = new HashSet<>();
```

SET UP A PRIORITY QUEUE WHICH RETURNS NODES IN THE ORDER OF THE SHORTEST DISTANCE FROM THE SOURCE - "THE GREEDY SOLUTION"

ADD A DISTANCE TABLE ENTRY FOR EACH NODE IN THE GRAPH

ADD THE SOURCE TO THE PRIORITY QUEUE AND SET UP A MAPPING TO THE VERTEX INFO FOR EVERY VERTEX IN THE QUEUE

THE SPANNING TREE IS THE SET OF EDGES CONNECTING ALL THE NODES OF THE GRAPH, AN EDGE IS REPRESENTED BY "01" IF IT CONNECTS VERTICES 0 AND 1

THE VERTICES WHICH WE'VE ALREADY INCLUDED IN THE SPANNING TREE

BUILD THE DISTANCE TABLE AND SPANNING TREE - PROCESS

```
while (!queue.isEmpty()) {
    VertexInfo vertexInfo = queue.poll();
    int currentVertex = vertexInfo.getVertexId();

    // Do not re-visit vertices which are already part of the
    // tree.
    if (visitedVertices.contains(currentVertex)) {
        continue;
    }
    visitedVertices.add(currentVertex);

    // If the current vertex is a source we do not have an edge
    // yet.
    if (currentVertex != source) {
        String edge = String.valueOf(currentVertex)
                + String.valueOf(distanceTable.get(currentVertex).getLastVertex());
        if (!spanningTree.contains(edge)) {
            spanningTree.add(edge);
        }
    }

    for (Integer neighbour : graph.getAdjacentVertices(currentVertex)) {
        int distance = graph.getWeightedEdge(currentVertex, neighbour);

        // If we find a new shortest path to the neighbour update
        // the distance and the last vertex.
        if (distanceTable.get(neighbour).getDistance() > distance) {
            distanceTable.get(neighbour).setDistance(distance);
            distanceTable.get(neighbour).setLastVertex(currentVertex);

            VertexInfo neighbourVertexInfo = vertexInfoMap.get(neighbour);
            if (neighbourVertexInfo != null) {
                queue.remove(neighbourVertexInfo);
            }

            neighbourVertexInfo = new VertexInfo(neighbour, distance);
            vertexInfoMap.put(neighbour,neighbourVertexInfo);
            queue.add(neighbourVertexInfo);
        }
    }
}

for (String edge : spanningTree) {
    System.out.println(edge);
}
```

ACCESS THE PRIORITY QUEUE TO FIND THE CLOSEST VERTEX - FOLLOWING THE GREEDY ALGORITHM

DON'T REVISIT VERTICES WHICH ARE ALREADY PART OF THE SPANNING TREE

ADD THE EDGE TO THE SPANNING TREE IF IT DOES NOT ALREADY EXIST

NOTE THAT WE ONLY CONSIDER THE WEIGHT OF THE EDGE IN ASSIGNING THE DISTANCE TO A NODE, NOT THE CURRENT DISTANCE FROM THE SOURCE TO THAT NODE

THE REST OF THE CODE IS SIMILAR TO THE SHORTEST PATH ALGORITHM