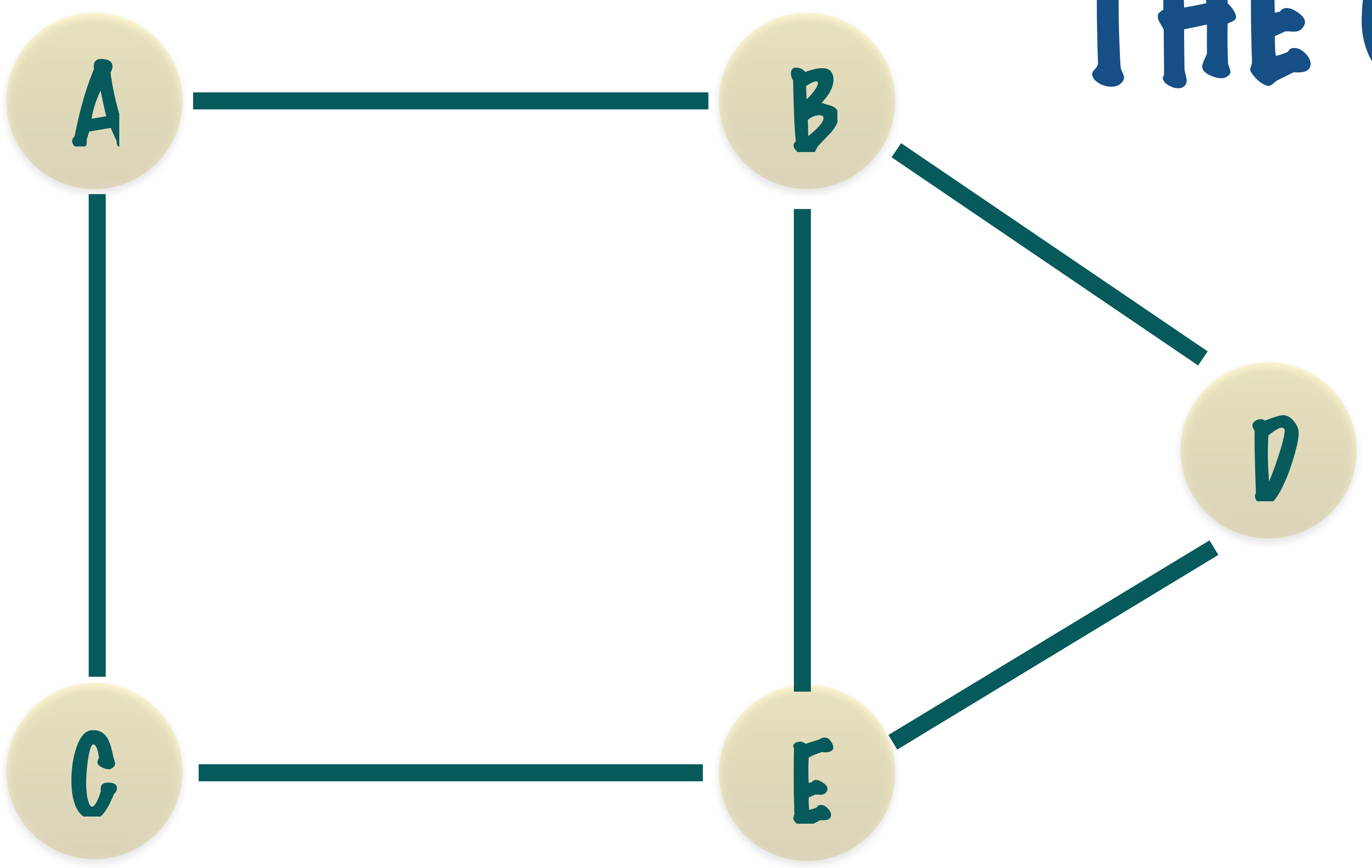
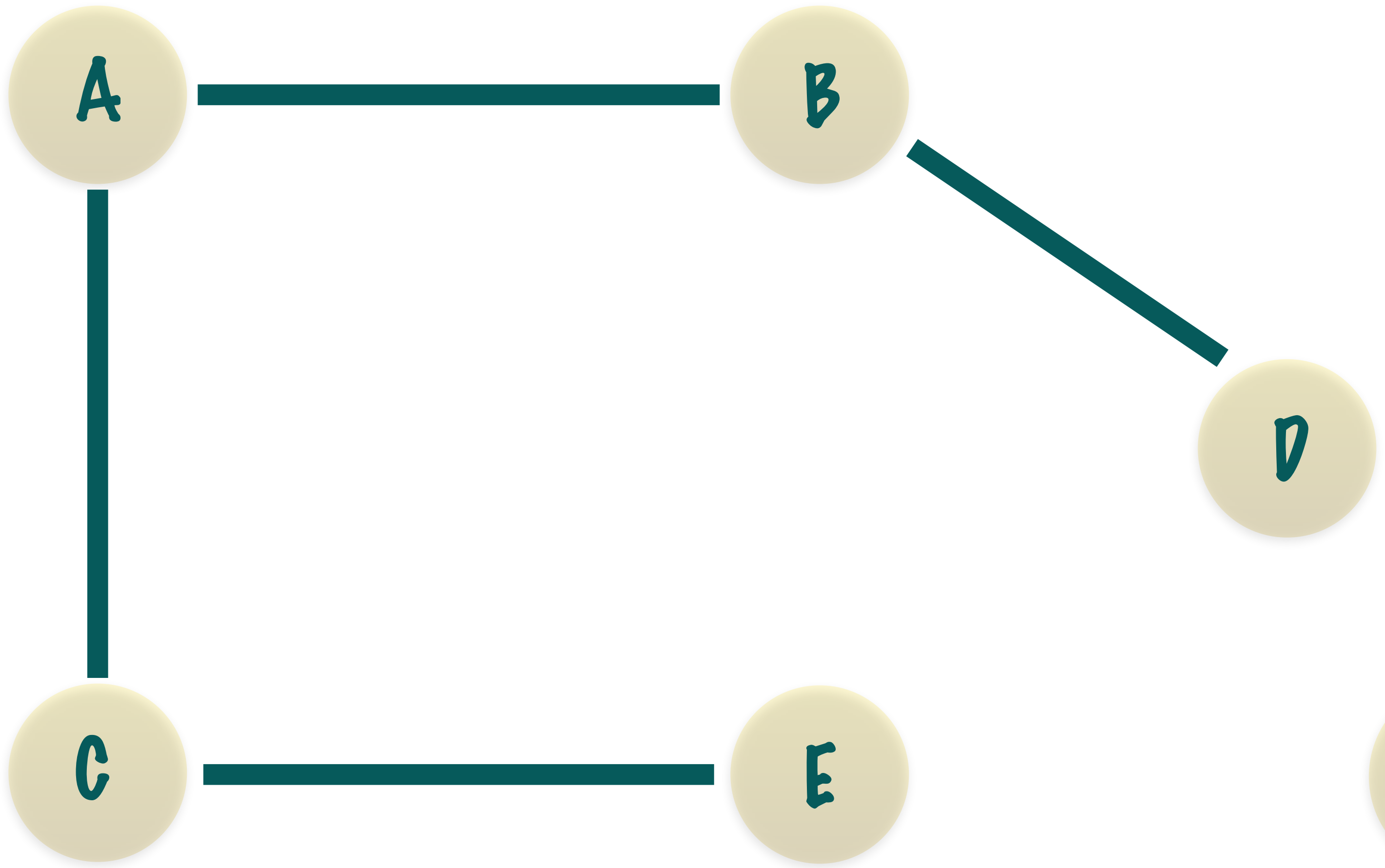
THE GRAPH

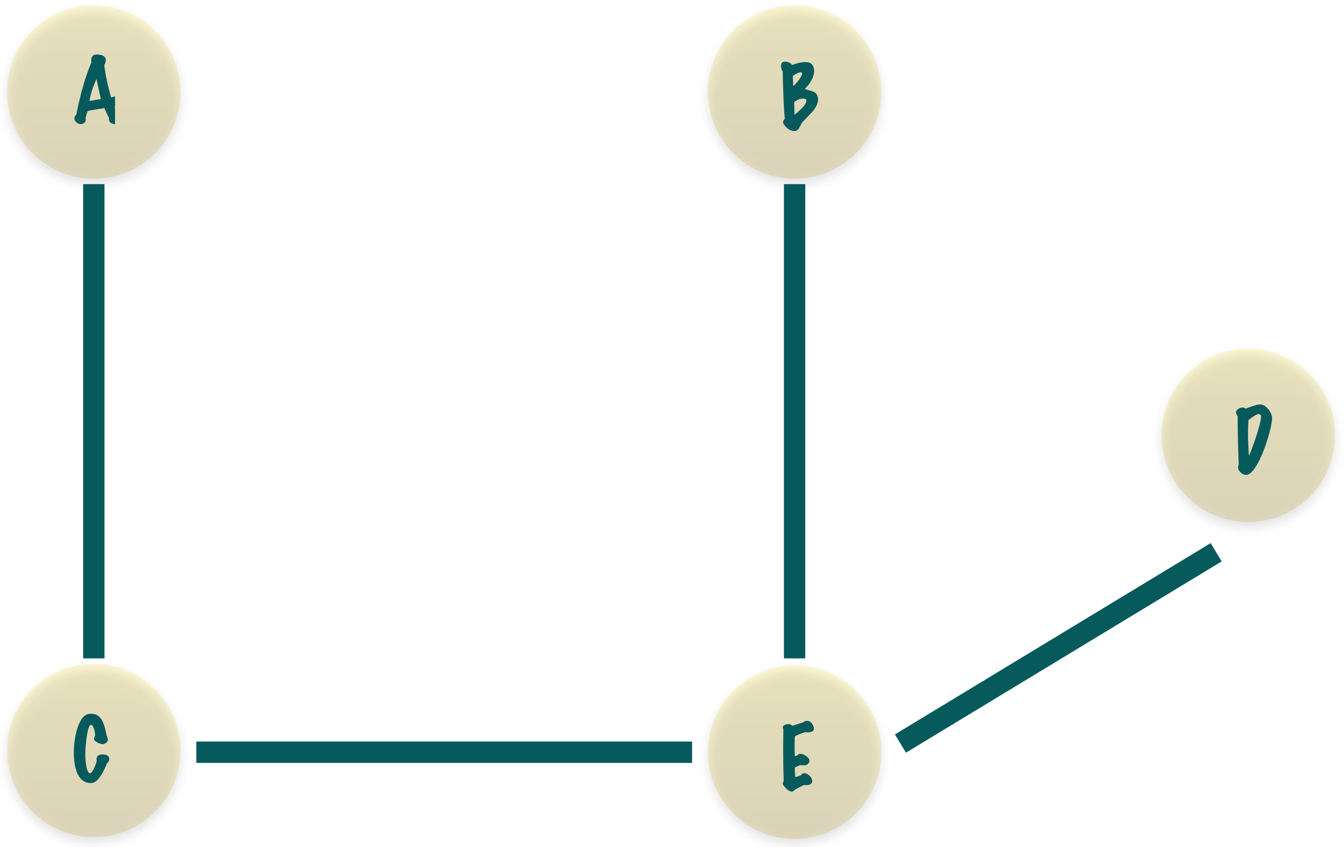
MINIMAL SPANNING TREE FOR FORESTS

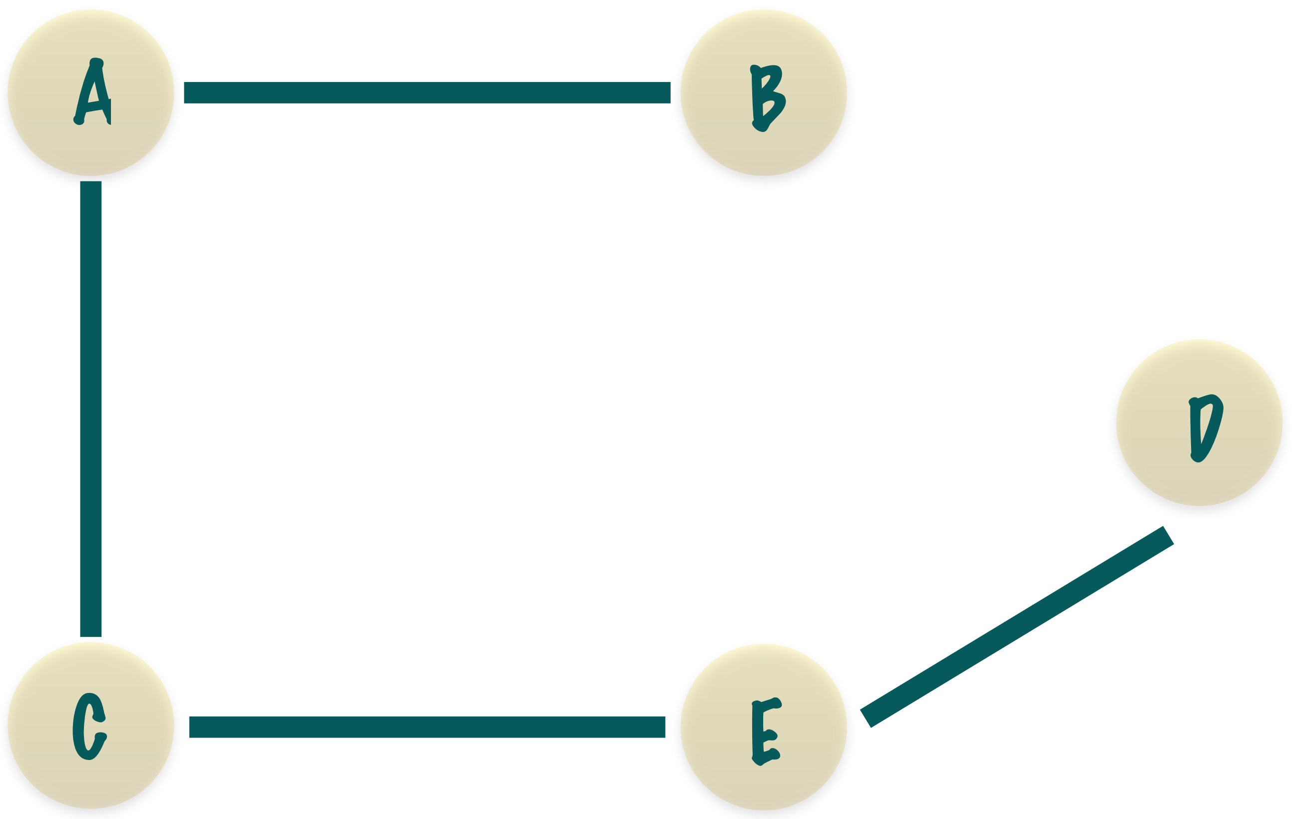
THE GRAPH SPANNING TREE

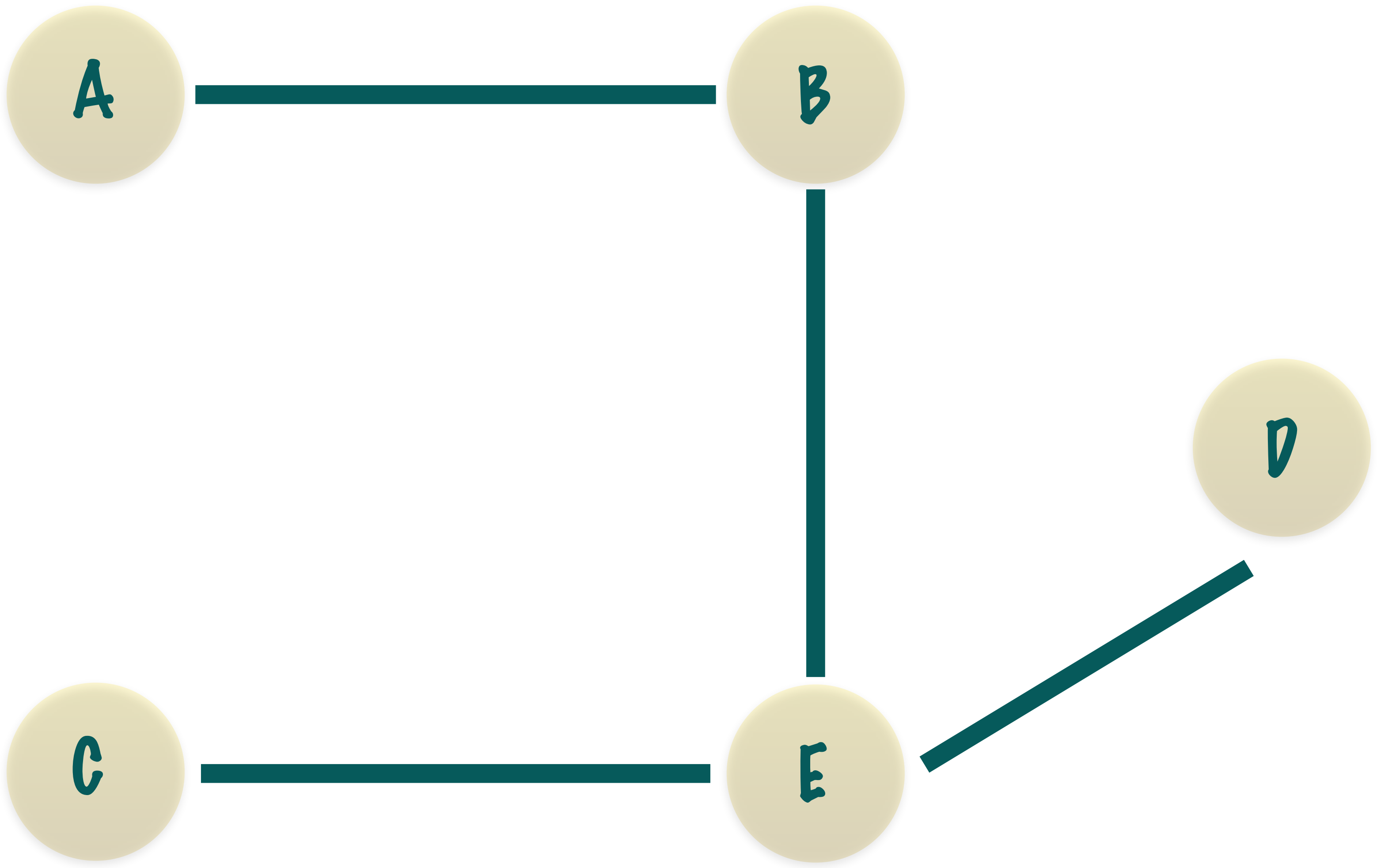
SPANNING TREE IS A SUBGRAPH THAT CONTAINS ALL THE VERTICES AND IS ALSO A TREE











THE GRAPH
MINIMAL SPANNING TREE FOR FORESTS

A FOREST IS AN UNCONNECTED GRAPH

KRUSKAL'S ALGORITHM

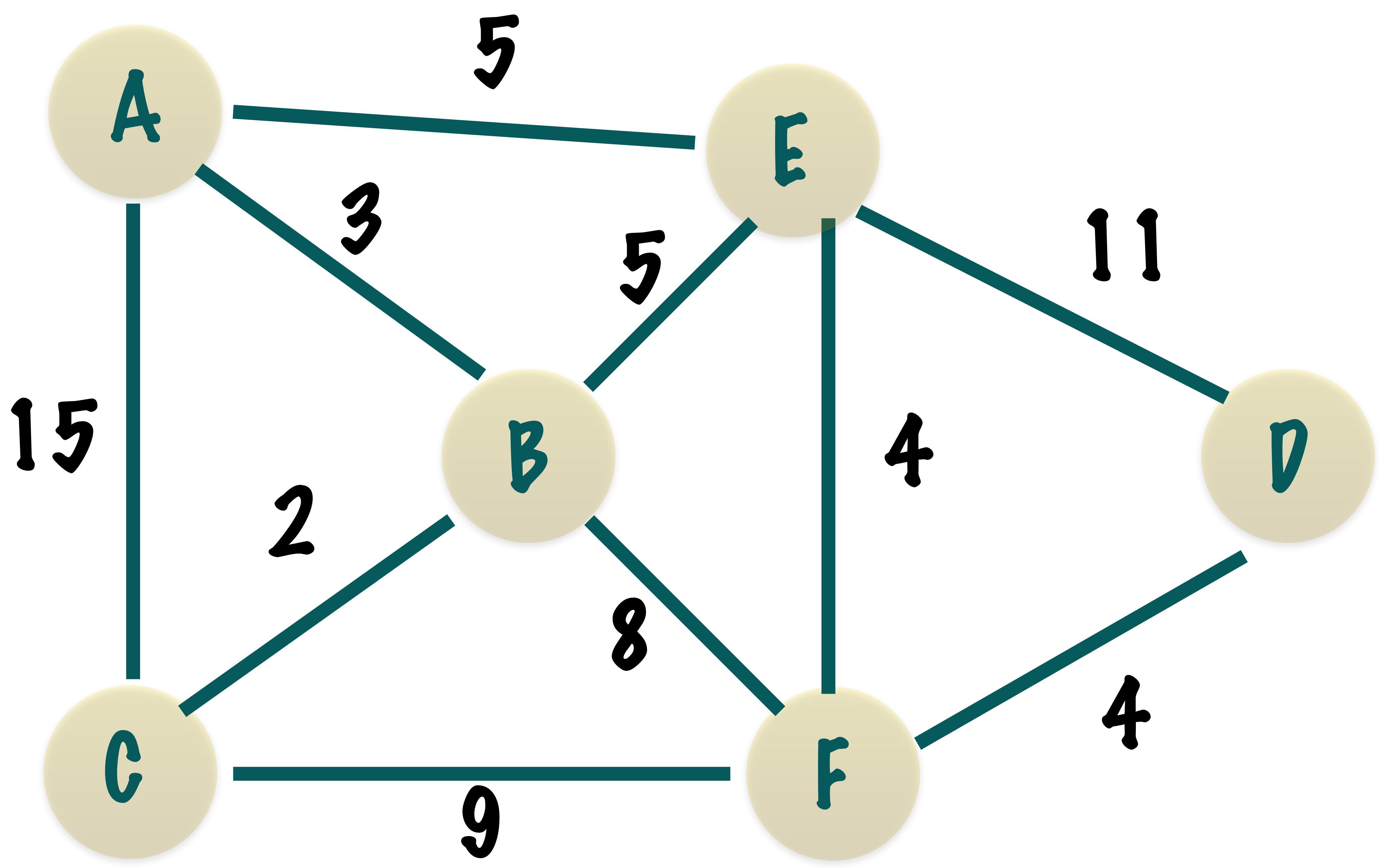
THIS IS A 'GREEDY ALGORITHM' IT TRIES TO FIND THE OPTIMAL NEXT STEP AT EVERY STEP - A LOCAL OPTIMUM NOT THE GLOBAL OPTIMUM

AT EVERY STEP WE CHOOSE THE SMALLEST WEIGHTED EDGE FROM THE ENTIRE GRAPH

2 THINGS TO KEEP IN MIND WHILE IMPLEMENTING KRUSKAL'S ALGORITHM

THE GRAPH MINIMAL SPANNING TREE FOR FORESTS

1 USE A PRIORITY QUEUE OF EDGES WHERE THE WEIGHTS OF THE EDGES DETERMINE THE PRIORITY OF THE EDGE



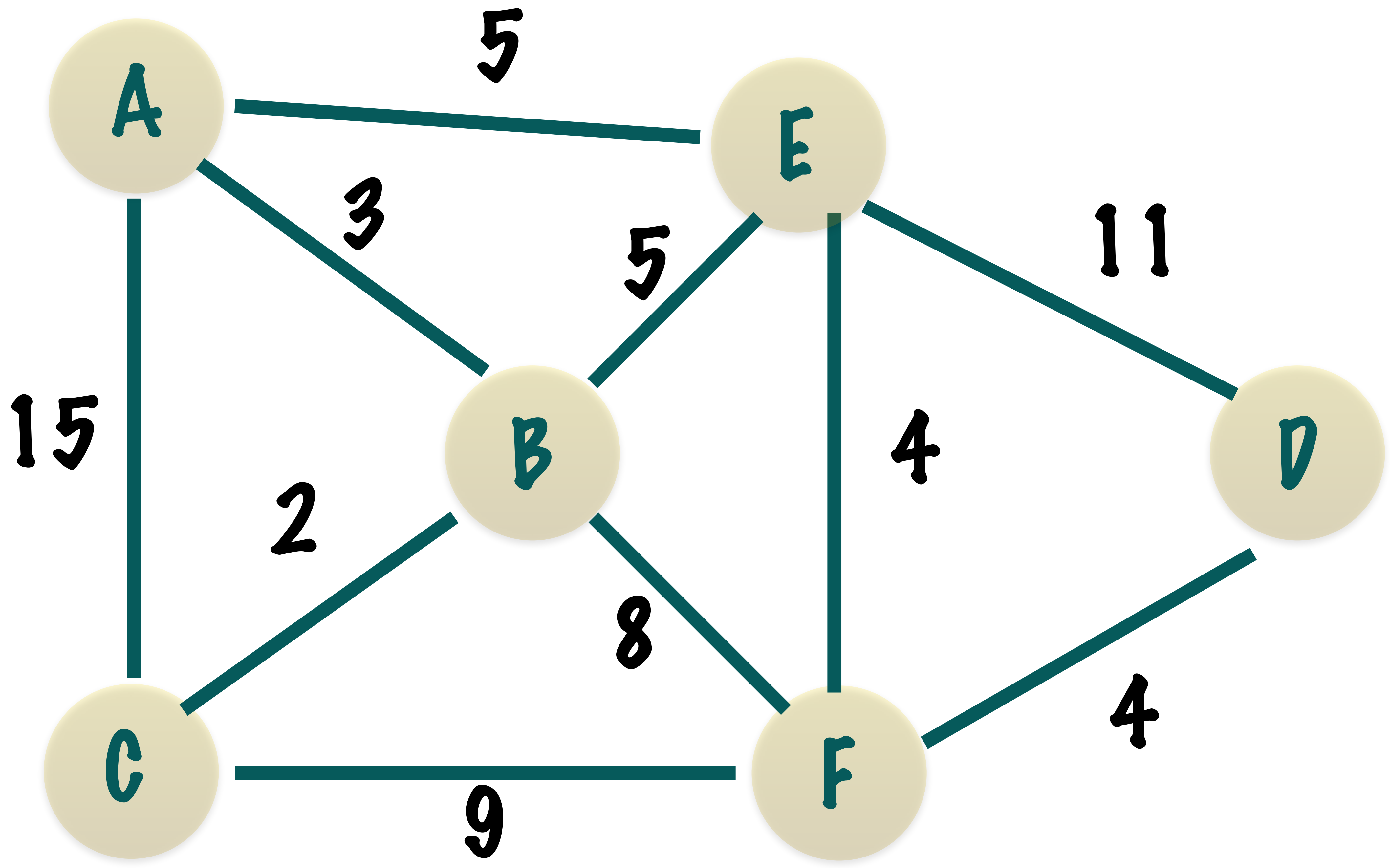
THE GRAPH

MINIMAL SPANNING TREE FOR FORESTS

2

WHILE ADDING A NEW EDGE, ALWAYS MAKE SURE THAT THE NEW EDGE DOES NOT CREATE A CYCLE IN THE SPANNING TREE

CONTINUE ADDING EDGES TILL WE GET V -1 EDGES SO THE GRAPH IS CONNECTED I.E. IT'S A TREE

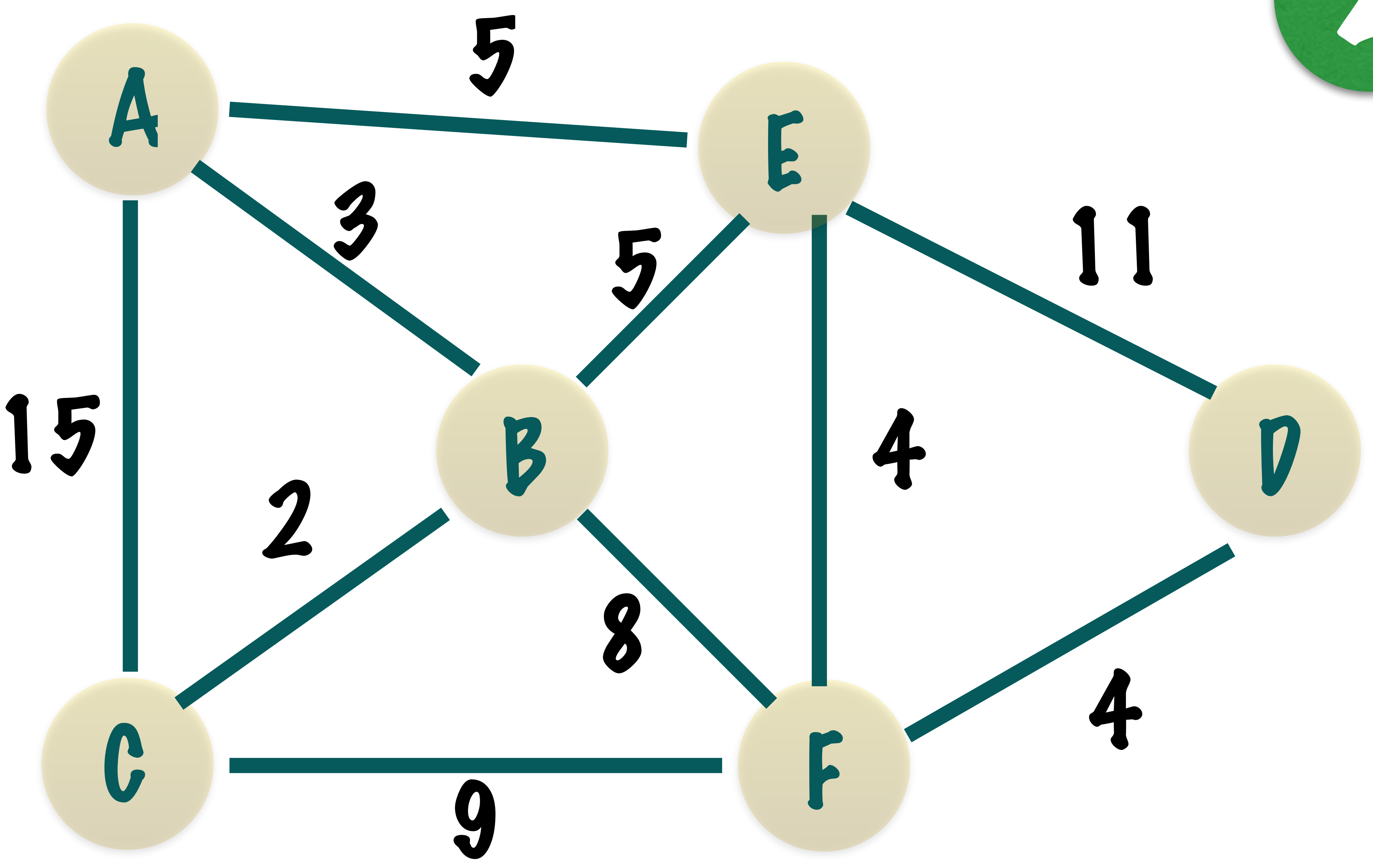


THE GRAPH

MINIMAL SPANNING TREE FOR FORESTS

1. USE A PRIORITY QUEUE OF EDGES WHERE THE WEIGHTS OF THE EDGES DETERMINE THE PRIORITY OF THE EDGE

2. WHILE ADDING A NEW EDGE, ALWAYS MAKE SURE THAT THE NEW EDGE DOES NOT CREATE A CYCLE IN THE SPANNING TREE

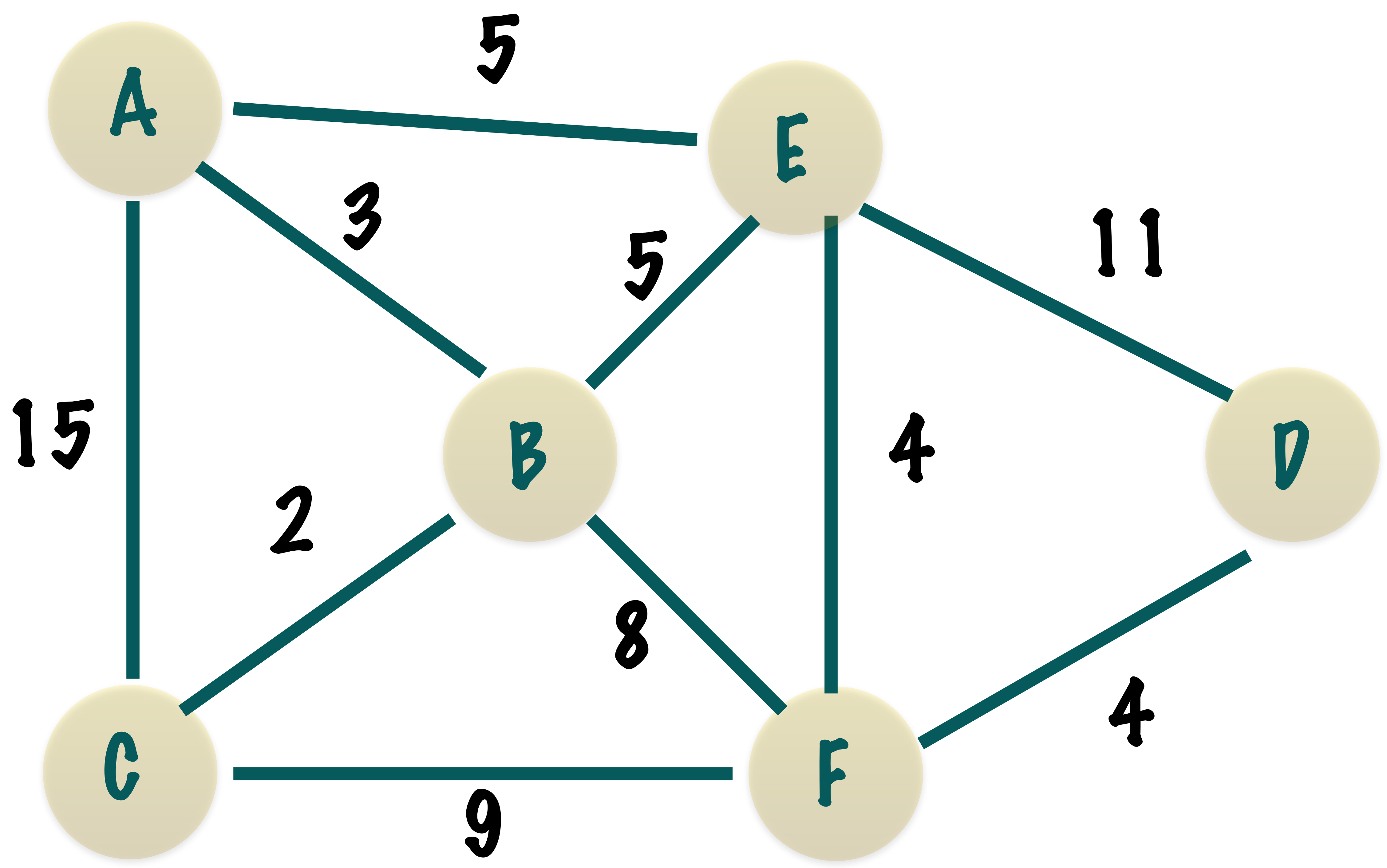


THE GRAPH

MINIMAL SPANNING TREE FOR FORESTS

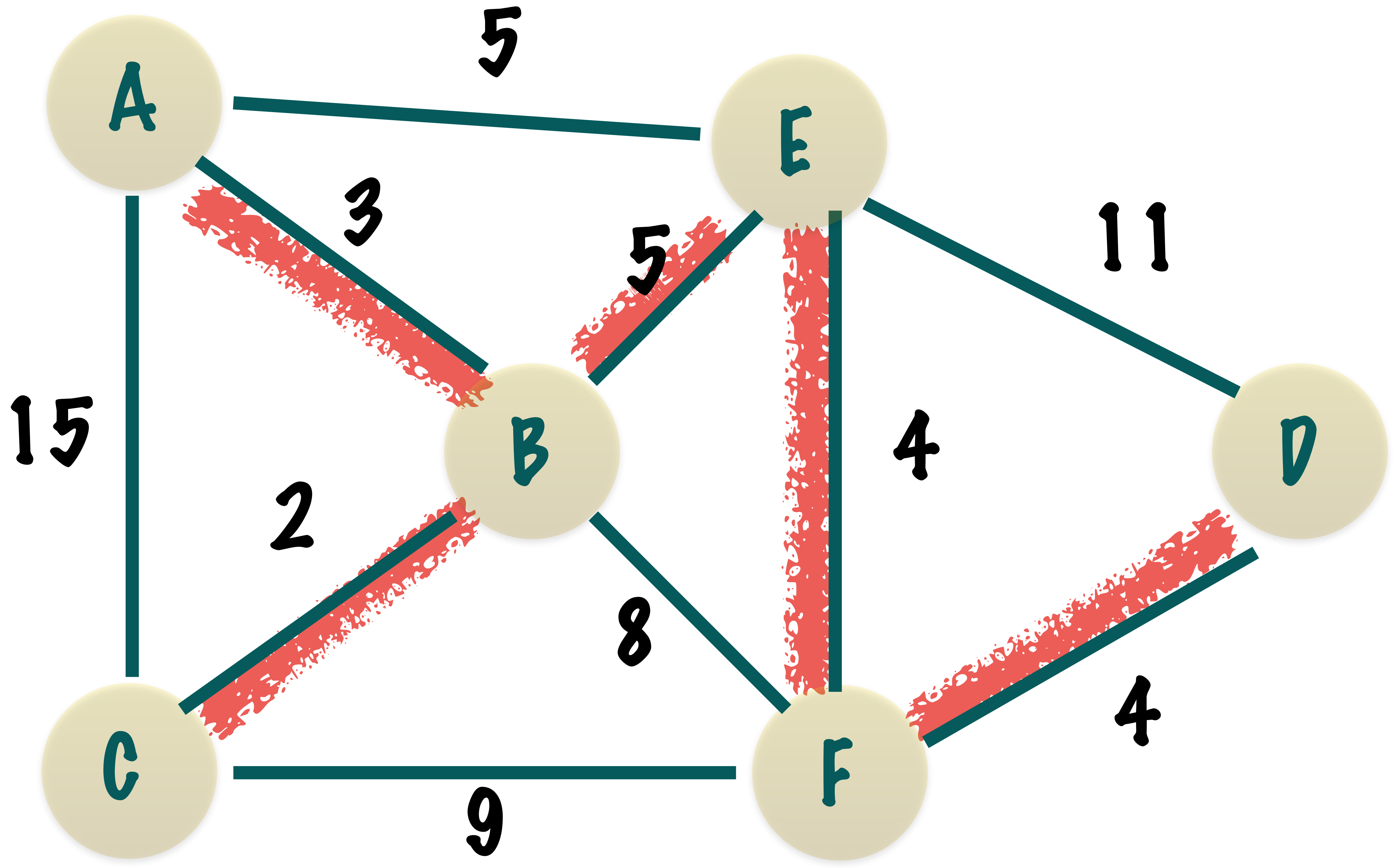
PRIORITY QUEUE OF EDGES

PRIORITY = WEIGHT OF EDGE



- BC 2
- AB 3
- DF 4
- EF 4
- BE 5
- AE 5
- BF 8
- CF 9
- ED 11
- AC 15

THE GRAPH MINIMAL SPANNING TREE FOR FORESTS



WE HAVE COVERED ALL THE VERTICES!

BC 2

AB 3

DF 4

EF 4

BE 5

AE 5

BF 8

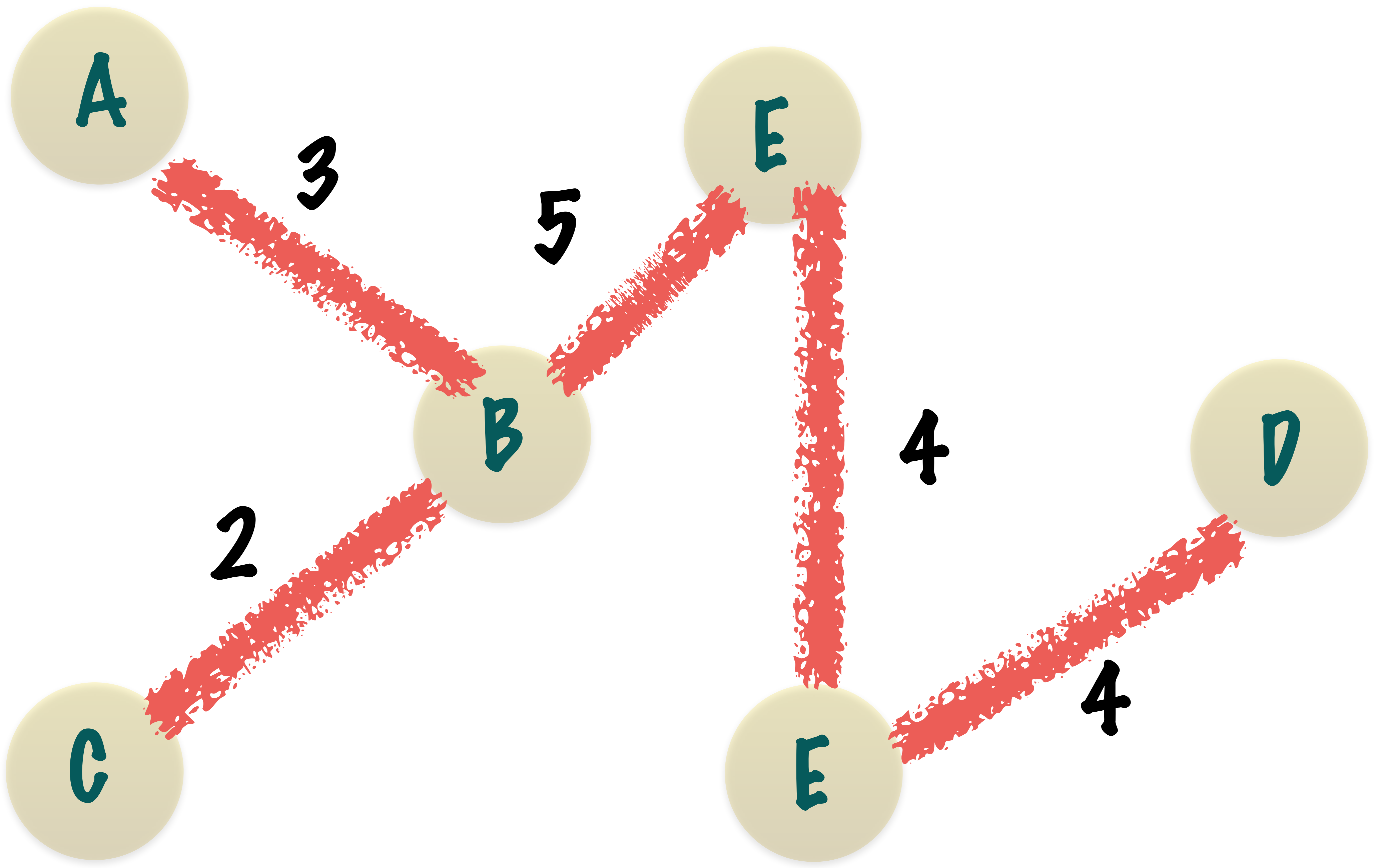
CF 9

ED 11

AC 15

THE GRAPH

MINIMAL SPANNING TREE FOR FORESTS



THIS ALGORITHM WORKS FOR BOTH CONNECTED AND UNCONNECTED GRAPHS I.E FORESTS

THE ALGORITHM'S RUNNING TIME IS E (LG E)!

THE MAIN PROCESSING TIME INVOLVES SORTING THE EDGES BY WEIGHT AND THIS IS THE RUNNING TIME OF THE BEST SORTING ALGORITHMS

EDGE INFO DATA STRUCTURE

```java
/**
 * A class which represents an edge in an undirected weighted graph.
 */
public static class EdgeInfo {

    private Integer vertex1;
    private Integer vertex2;
    private Integer weight;

    public EdgeInfo(Integer vertex1,Integer vertex2, Integer weight) {
        this.vertex1 = vertex1;
        this.vertex2 = vertex2;
        this.weight = weight;
    }

    public Integer getVertex1() {
        return vertex1;
    }

    public Integer getVertex2() {
        return vertex2;
    }

    public Integer getWeight() {
        return weight;
    }

    @Override
    public String toString() {
        return String.valueOf(vertex1) + String.valueOf(vertex2);
    }
}
```

REPRESENTS AN EDGE USING THE TWO VERTICES AND THE EDGE WEIGHT

STRING REPRESENTATION OF AN EDGE WHICH IS "02" FOR AN EDGE WHICH CONNECTS VERTEX 0 WITH VERTEX 2

BUILD THE EDGE MAP AND SPANNING TREE- SETUP

```java
static void spanningTree(Graph graph) {
    // A priority queue to store and retrieve the edges on the basis of their
    // weights.
    PriorityQueue<EdgeInfo> queue = new PriorityQueue<>(new Comparator <EdgeInfo> () {
        @Override
        public int compare(EdgeInfo o1, EdgeInfo o2) {
            return o1.getWeight().compareTo(o2.getWeight());
        }
    });

    // Add all edges to the priority queue.
    for (int i= 0; i < graph.getNumVertices(); i++) {
        for (int neighbour : graph.getAdjacentVertices(i)) {
            queue.add(new EdgeInfo(i, neighbour, graph.getWeightedEdge(i, neighbour)));
        }
    }

    Set<Integer> visitedVertices = new HashSet<>();
    Set<EdgeInfo> spanningTree = new HashSet<>();
    Map<Integer, Set<Integer>> edgeMap = new HashMap<>();
    for (int v = 0; v < graph.getNumVertices(); v++) {
        edgeMap.put(v, new HashSet<>());
    }
```

SET UP A PRIORITY QUEUE WHICH RETURNS EDGES WITH THE SMALLEST WEIGHT - "THE GREEDY SOLUTION"

ADD EVERY EDGE TO THE PRIORITY QUEUE

KEEP TRACK OF THE VERTICES ALREADY VISITED, EACH EDGE SHOULD ADD A NEW VERTEX TO THE SET TILL WE GET NUMBER OF VERTICES - 1 EDGES

THE EDGE MAP TRACKS THE EDGES ADDED TO THE SPANNING TREE TO SEE IF IT FORMS A CYCLE

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

BUILD THE EDGE MAP AND SPANNING TREE - PROCESS

```java
while(!queue.isEmpty() && spanningTree.size() < graph.getNumVertices() - 1) {
    EdgeInfo currentEdge = queue.poll();

    // Add the new edge to the edge map and see if it ends up with a cycle.
    // If yes then discard this edge and get the next edge from the priority
    // queue.
    edgeMap.get(currentEdge.getVertex1()).add(currentEdge.getVertex2());
    if (hasCycle(edgeMap)) {
        edgeMap.get(currentEdge.getVertex1()).remove(currentEdge.getVertex2());
        continue;
    }

    spanningTree.add(currentEdge);

    // Add both vertices to the visited list, the set will ensure
    // that only one copy of the vertex exists.
    visitedVertices.add(currentEdge.getVertex1());
    visitedVertices.add(currentEdge.getVertex2());
}

// Check whether all vertices have been covered with the spanning tree.
if (visitedVertices.size() != graph.getNumVertices()) {
    System.out.println("Minimum Spanning Tree is not possible");
} else {
    System.out.println("Minimum Spanning Tree using Kruskal's Algorithm");
    for(EdgeInfo edgeInfo : spanningTree ) {
        System.out.println(edgeInfo);
    }
}
```

THE SPANNING TREE SHOULD HAVE NUMBER OF VERTICES - 1 EDGES

RETRIEVE THE EDGES WITH THE SMALLEST WEIGHT FIRST - THE GREEDY SOLUTION

ADD THE EDGE TO THE EDGE MAP AND SEE IF IT CAUSES A CYCLE - IF YES THEN DO NOT USE THE EDGE IN THE SPANNING TREE

ADD THE EDGE TO THE SPANNING TREE

ADD BOTH VERTICES TO THE VISITED VERTEX LIST

IF ALL VERTICES HAVE BEEN COVERED THE SPANNING TREE EXISTS!

CHECK FOR CYCLES IN THE SPANNING TREE

```java
private static boolean hasCycle(Map<Integer, Set<Integer>> edgeMap) {
    for (Integer sourceVertex : edgeMap.keySet()) {
        LinkedList<Integer> queue = new LinkedList<>();
        queue.add(sourceVertex);
        Set<Integer> visitedVertices = new HashSet<>();
        while (!queue.isEmpty()) {
            int currentVertex = queue.pollFirst();
            if (visitedVertices.contains(currentVertex)) {
                return true;
            }

            visitedVertices.add(currentVertex);
            queue.addAll(edgeMap.get(currentVertex));
        }
    }

    return false;
}
```

START FROM EVERY VERTEX IN THE EDGE MAP AND EXPLORE ALL VERTICES PRESENT IN THE SPANNING TREE

IF WE EVER RE-VISIT A VERTEX WE'VE ALREADY SEEN IN THE SPANNING TREE IT MEANS THERE IS A CYCLE IN THE SPANNING TREE

ADD TO QUEUE ALL THE ADJACENT VERTICES OF THE CURRENT VERTEX WHICH ARE PART OF THE SPANNING TREE