Name

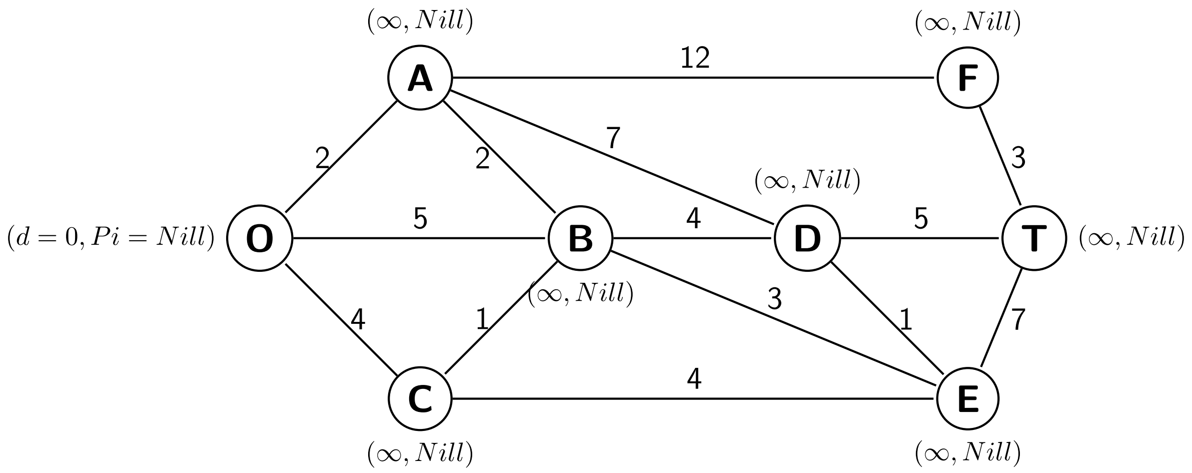
Roll no

## Problem 1

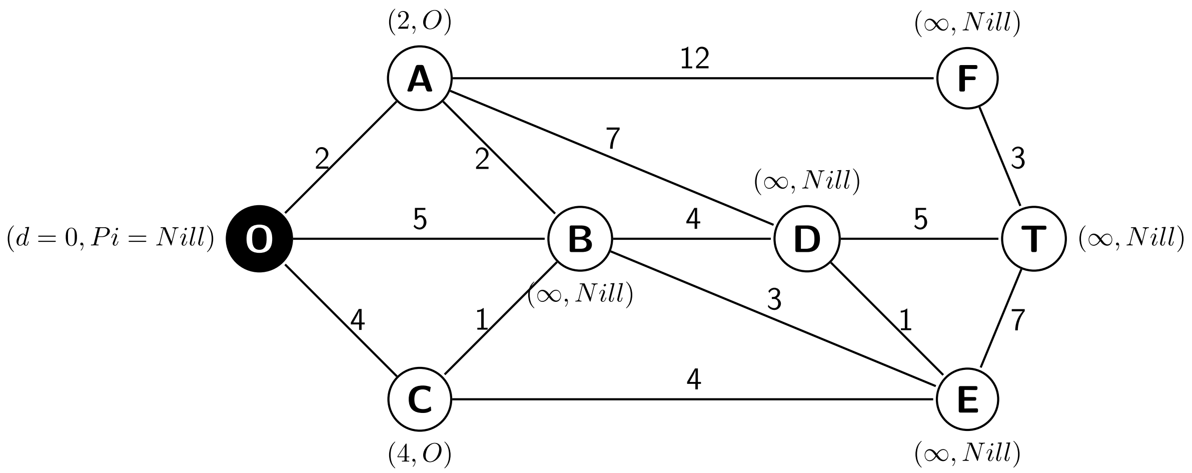
Consider the graph on the right. Step through Dijkstra's algorithm to find the Shortest Path from origin O to destination T. Show all steps.

## Solution

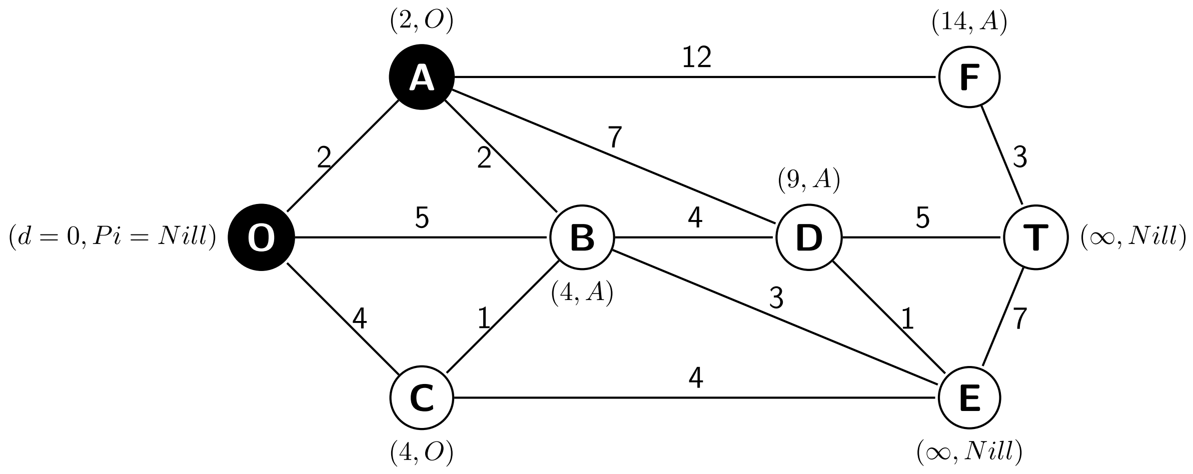
**Black = Visited**
**white = Unvisited**



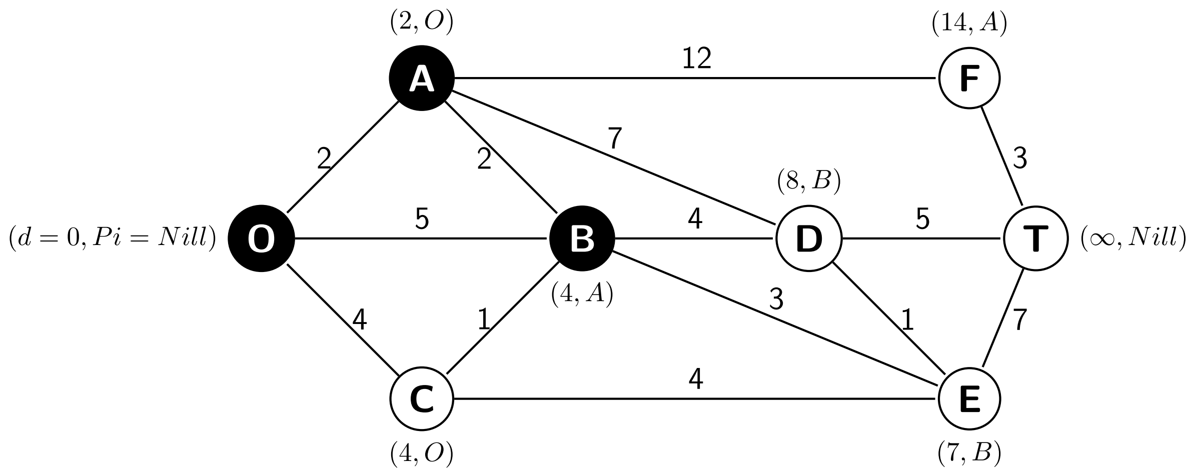
**Step 1:**



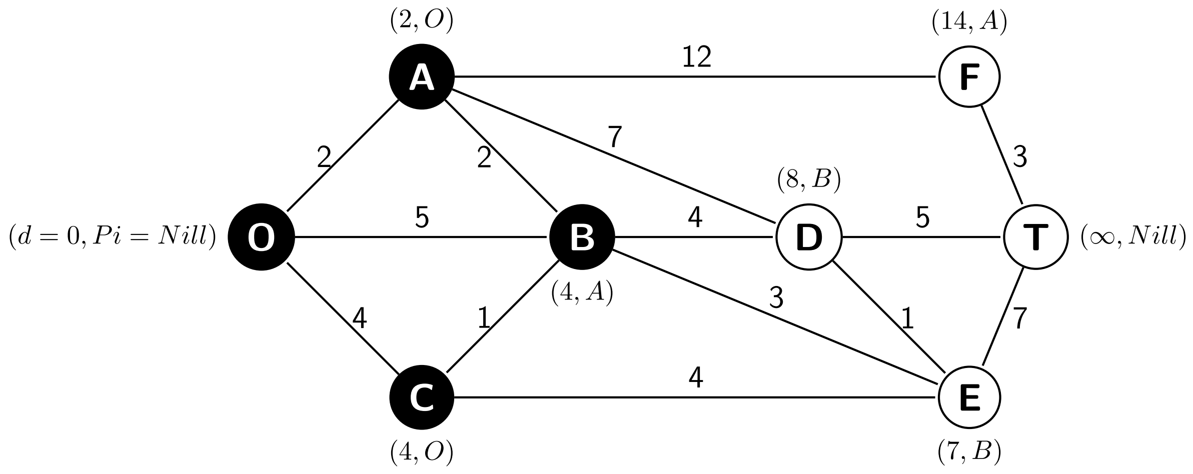
**Step 2:**



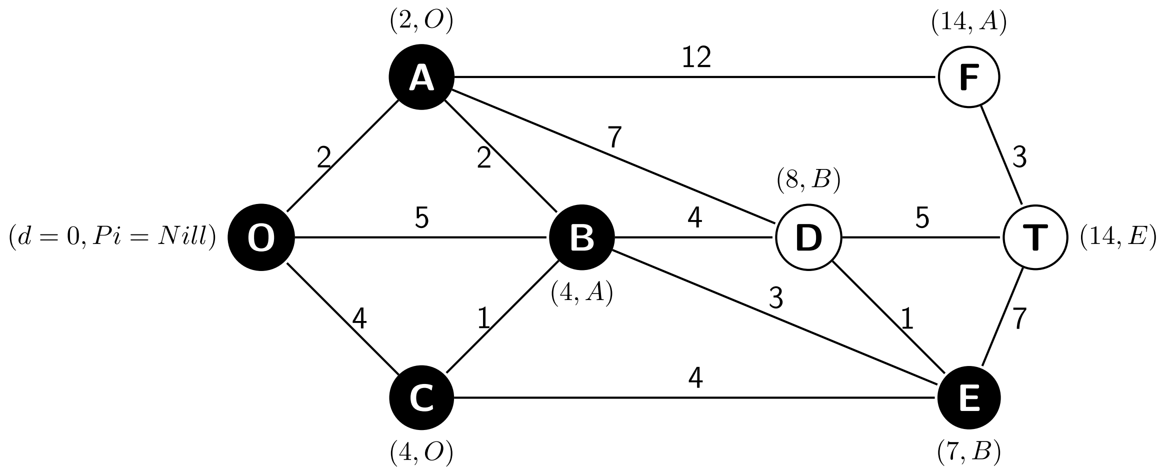
**Step 3:**



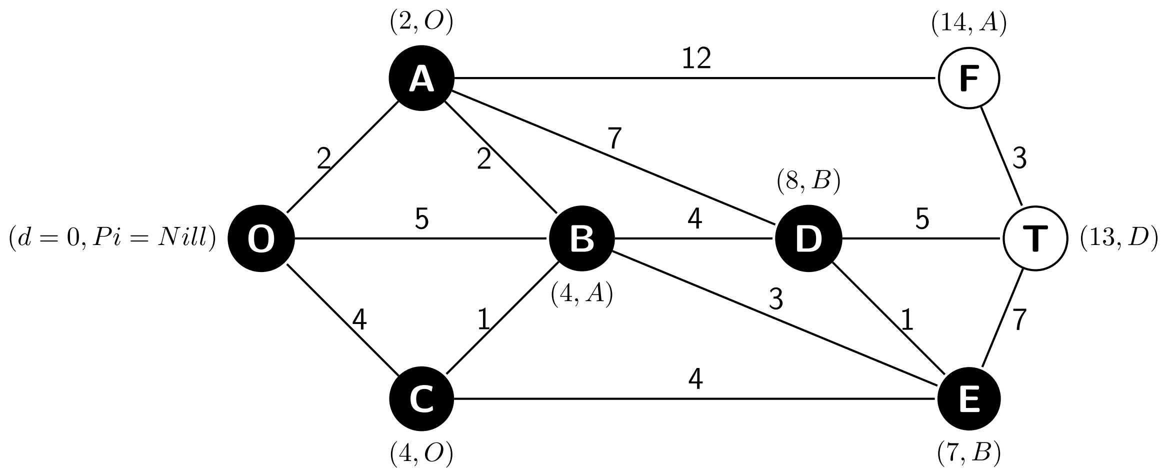
**Step 4:**



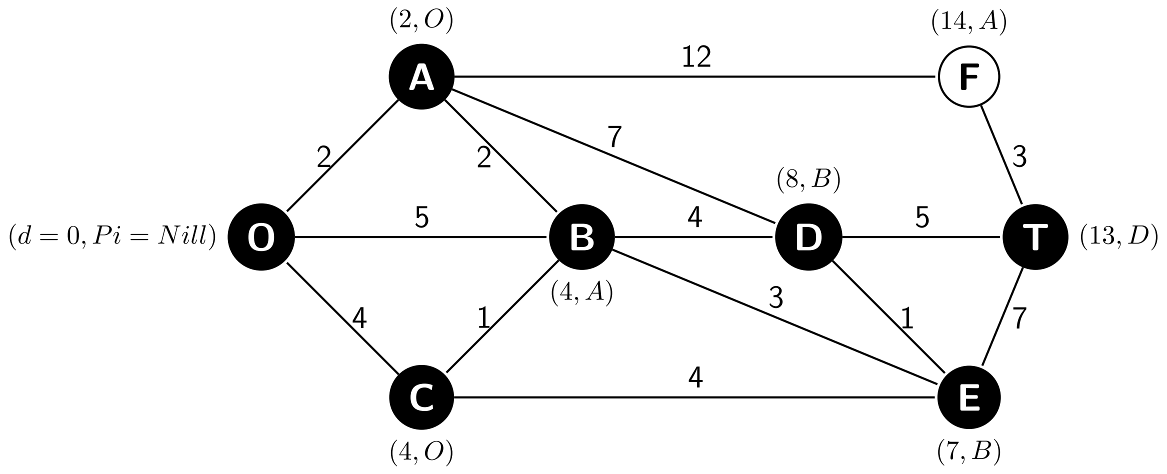
**Step 5:**



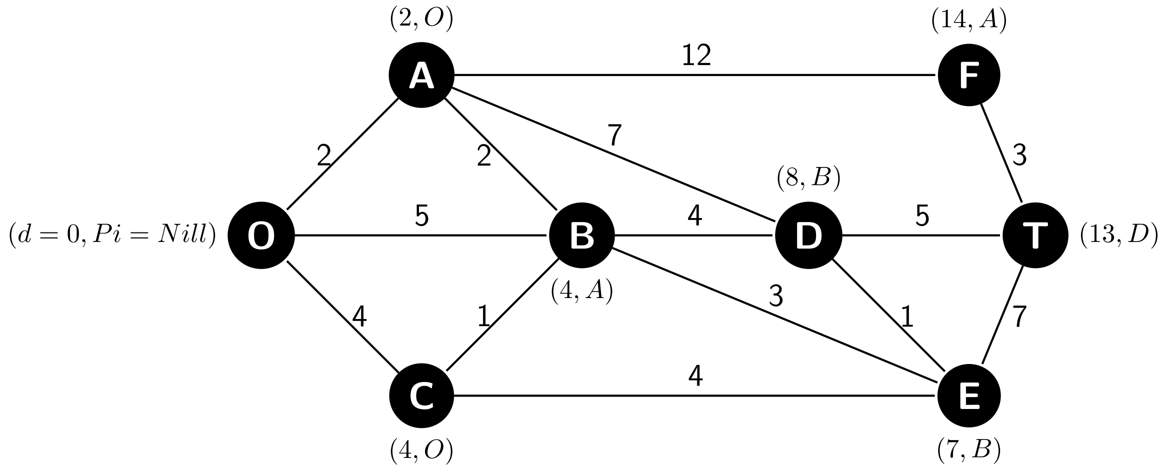
**Step 6:**



**Step 7:**



**Step 8:**



## - Shortest Path from origin O to destination T

Shortest Distance from origin O to destination T is 13.

Path:
$O \rightarrow A \rightarrow B \rightarrow D \rightarrow T$

# Problem 2

Let G = ( V, E, W ) be a connected graph, in which each edge weight we $\epsilon$ W, $\forall$ e $\epsilon$ E is distinct (i.e., not two edges have the same weight). Prove that G has a unique minimum spanning tree

## 1 Solution

To prove that Graph G has a unique minimum spanning tree, we will use the cut property, which states that for any cut of the graph (i.e., a partition of the vertices into two disjoint sets), the minimum weight edge crossing the cut must be in the minimum spanning tree. Suppose for the sake of contradiction that G has two distinct minimum spanning trees, T1 and T2. Since T1 and T2 are both minimum-spanning trees, they must have the same weight which is the minimum of all. Now, let S be the set of vertices that are connected to vertex v by edges in T1 but not in T2, where v is any vertex in G. By the cut property, there must be a minimum weight edge e crossing the cut between S and the rest of the vertices in G. Since T1 is a minimum spanning tree, the weight of e must be less than or equal to the weight of any edge in T1 that crosses the same cut. However, since e is not in T2, another edge f in T2 must cross the same cut with the same weight. But we are given that no two edges have the same weight, so e and f cannot have the same weight. So we have to suppose that w(e) < w(f). Then, we can replace f with e in T2 to obtain a new spanning tree T3 with a weight less than or equal to that of T2. Now, there are two cases:

**Case 1:**

If T3 is a minimum spanning tree, then T1 and T3 have different weights, contradicting the assumption that T1 and T2 have the same weight.

**Case 2:**

If T3 is not a minimum spanning tree, then there exists a spanning tree with a weight less than that of T3. But this contradicts the assumption that T1 and T2 are both minimum-spanning trees.

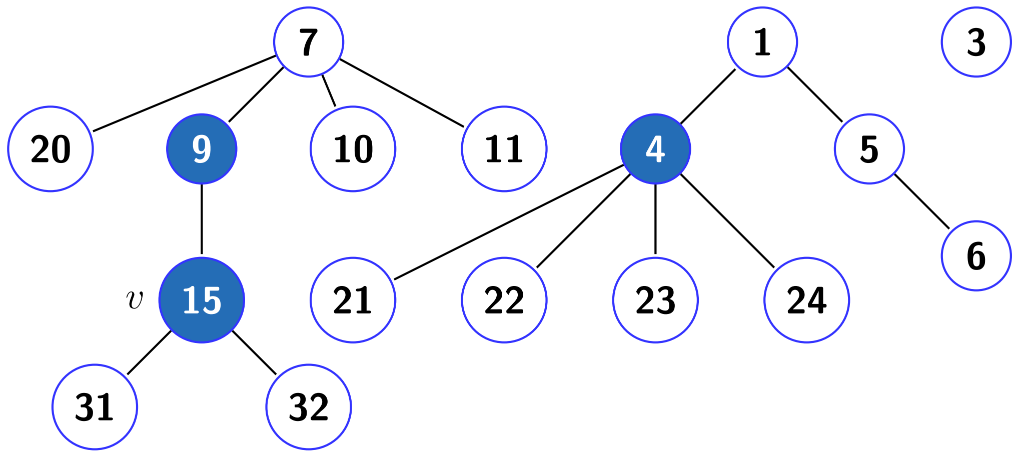
**Conclusion:**

Thus, we have reached a contradiction in both cases, and so our assumption that G has two distinct minimum spanning trees must be false. Therefore, G has a unique minimum spanning tree.

## Problem 3

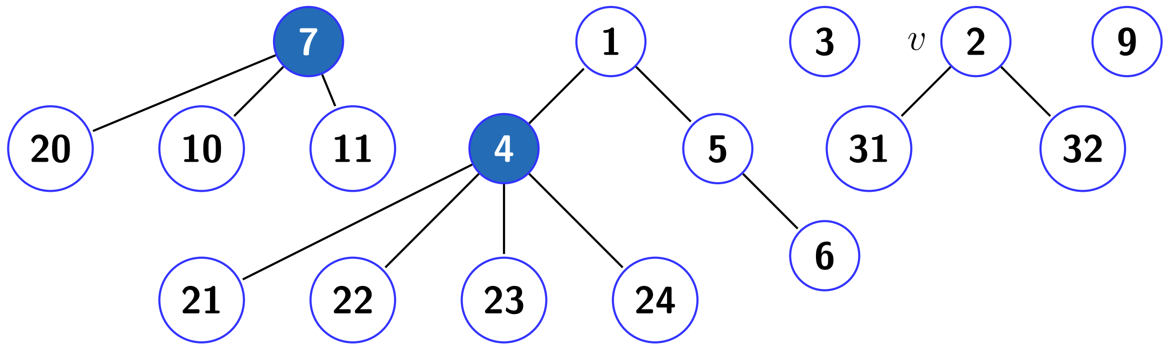
Consider the Fibonacci heap on the right. Marked nodes are highlighted in blue.

- Show the resulting heap after decrease-key(v,2).
- Show the status of the heap after a subsequent extract-min operation.

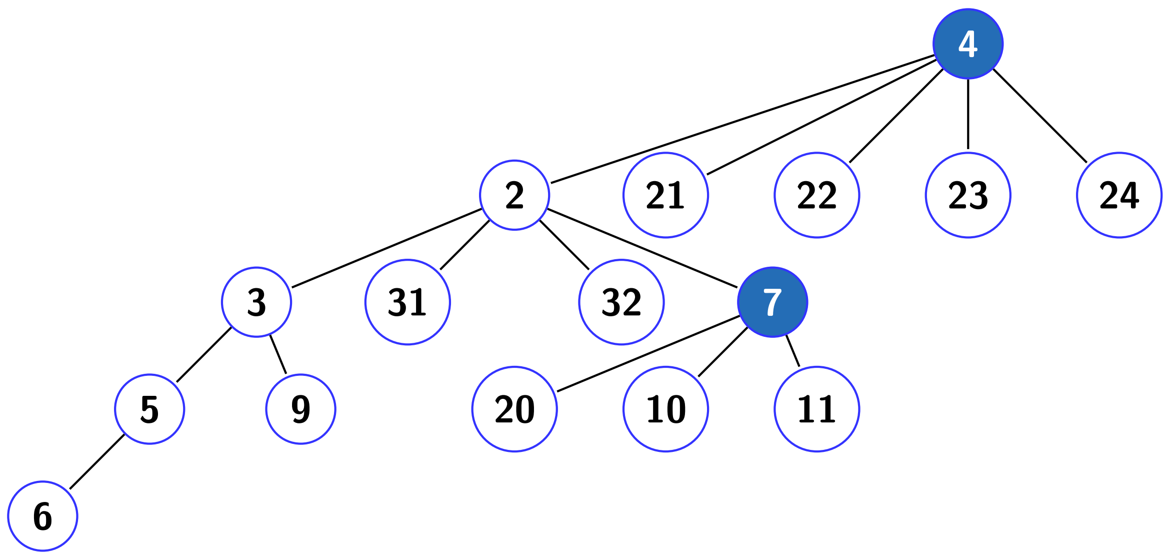


## Solution:

- The resulting heap after decrease-key(v, 2)



- The status of the heap after a subsequent extract-min operation.


4
2
21
22
23
24
3
31
32
7
5
9
20
10
11
6

## Problem 4

Suppose a minimum spanning tree(MST) has already been computed for a graph G using Kruskal's algorithm. Let update-mst denote a procedure that updates the MST when a new node (and its incident edges)is added to G. Describe update-mst and derive its complexity

## Solution

When a new node and its incident edges are added to a graph G for which a minimum spanning tree (MST) has already been computed using Kruskal's algorithm, we can update the MST using a procedure called Update-MST.

To perform Update-MST, we first add the new node and its incident edges to G.

We then initialize a set S containing the new node and all of its neighbors in G.

For each node v in S, we find the minimum-weight edge e=(v, w) connecting v to a node w in G-S, and add e to a priority queue Q. We then initialize an empty set A to store the new MST. We then iterate through the priority queue Q while set A does not form a spanning tree. In each iteration, we remove the minimum-weight edge e=(v, w) from Q. If adding e to A creates a cycle, we discard e. Otherwise, we add e to A and add w to S. The time complexity of Update-MST depends on the size of the graph and the number of edges added. Initializing Set takes O(d) time, where d is the degree of the new node. Making Priority Queue takes O(d log V) time, where V is the number of vertices in the graph. Iterating through Queue takes O(log V) time per iteration, and adding e to A takes O(d log V) time since we need to update the priority queue for each newly added node.

In total, the time complexity of the algorithm is O(E log V + d log V), where E is the number of edges in the graph. Note that if the number of edges added is much smaller than the total number of edges in the graph, then the time complexity can be reduced accordingly.

# Problem 5

Consider the following algorithm to compute a topological ordering of a DAG. Recall that a topological ordering of a graph is an ordering of its nodes as v1,v2,...,vn so that ∀ edge (vi,vj) we have i ¿ j.

# Algorithm

1. Find a node v with no incoming edges and order it first.
2. Delete v from graph G.
3. Recursively compute a topological ordering of G - v and append this order after v.

### 1.1

Prove that this algorithm does indeed result in a topological ordering of G.

## Proof:

We can prove the correctness of the given algorithm that it results in a topological ordering of G, by induction on the number of nodes in the DAG.

#### Base case:

If the DAG has only one node, then it has no incoming edges and the algorithm will correctly output the ordering

#### Inductive step:

Let G be a DAG with n ¿ 1 node, and assume that the algorithm correctly computes a topological ordering for any DAG with k nodes that is smaller than n.

Let v be a node in G with no incoming edges. Since there are no incoming edges to v, it can be ordered first in any topological ordering of G.

Now, let G' be the subgraph of G obtained by deleting v and all its outgoing edges. Since v has no outgoing edges, G' has n - 1 nodes. By the inductive hypothesis, the algorithm correctly computes a topological ordering of G', which we denote as v2, v3, ..., vn.

We can append v to the beginning of this ordering to obtain a topological ordering of G as v, v2, v3, ..., vn.

To see why this is a valid topological ordering, note that since v has no outgoing edges, it cannot be a successor of any other node in G, so there can be no edge (vi, v) for any i ¿ 1. Furthermore, since v has no incoming edges, it cannot be a predecessor of any other node in G', so there can be no edge (v, vj) for any j ¡ i. Therefore, the ordering v, v2, v3, ..., vn satisfies the condition that i ¡ j whenever (vi, vj) is an edge in G.

Therefore, the algorithm correctly computes a topological ordering of G, completing the inductive step and the proof.

### 1.2

Extend the given algorithm for an arbitrary directed graph that may or may not be a DAG. Note: Your solution should take the form of a short essay. Specifically, a topic paragraph should summarize the problem you are solving and what your results are. The body of the essay should provide the following:
- A description of the algorithm in English and, if helpful, pseudo–code.
- At least one worked example or diagram to show more precisely how your algorithm works.
- A proof(or indication) of the correctness of the algorithm.

## Proof:

The problem we are solving is to extend the given algorithm for computing a topological ordering of a directed acyclic graph (DAG) to an arbitrary directed graph that may or may not be a DAG.
In order to extend the algorithm, we need to handle the case where the graph contains cycles. If the graph contains a cycle, it cannot have a topological ordering since the ordering would need to violate the ordering constraint for at least one edge in the cycle. It means that the ordering will contain at least 1 back edge which is not permitted in the topological sortings. Therefore, our algorithm should detect cycles and report that the graph does not have a topological ordering if it contains any cycles.

### Extended Algorithm

The extended algorithm can be described as follows:

1. Initialize a set S of vertices with no incoming edges.

2. Initialize a list L to store the topological ordering.

While S is not empty:

3. Remove a vertex v from S.

4. Append v to L.

5. For each node u such that there is an edge (v,u), remove the edge from the graph.

6. If u has no incoming edges after removing (v,u), add u to S.

7. If there are still edges remaining in the graph, report that the graph contains a cycle and has no topological ordering.

8. Otherwise, return the list L as the topological ordering.

## Working Example

The algorithm works as follows: we start by initializing S to contain all nodes with no incoming edges. We then repeatedly remove a node v from S, append it to the topological ordering, and remove all outgoing edges from v. If any of these removals result in a node u having no incoming edges, we add u to S. If S is ever empty and there are still edges remaining in the graph, then the graph contains a cycle and we cannot have a topological ordering. Otherwise, we return the topological ordering as a list. In pseudo-code, the algorithm looks like this:

## Pseudo-code

```
topological-sort(G):

S = set of nodes with no incoming edges

L = empty list for topological ordering

while S is not empty:

    remove a node v from S

    append v to L

    for each node u such that (v,u) is an edge in G:

        remove the edge (v,u) from G

        if u has no incoming edges:

            add u to S

if G still has edges:

    report that G contains a cycle and has no topological ordering

else:
    return L
```

Overall, this extended algorithm can compute a topological ordering of an arbitrary directed graph if it exists and correctly reports if the graph contains a cycle and does not have a topological ordering

# Proof by Induction:

To prove the correctness of the extended algorithm using induction, we need to show that the algorithm produces a valid topological ordering if one exists, and reports correctly if the graph contains a cycle and does not have a topological ordering.

### Base case:

If the graph has only one vertex, then it has no incoming edges and the algorithm will correctly output the ordering consisting of only that vertex.

### Inductive step:

Let G be a graph with n ¿ 1 vertices, and assume that the algorithm correctly computes a topological ordering for any graph with fewer than n nodes. If G has no cycles, then it is a DAG, and the algorithm will work as described for DAGs, producing a valid topological ordering.

Now, assume that G contains at least one cycle. If the algorithm does not detect the cycle and reports a valid topological ordering, this must mean that the algorithm has incorrectly removed an edge in the cycle during one of the iterations, making it appear that the graph has no cycle. However, since the graph does contain a cycle, this is a contradiction, and therefore the algorithm must correctly detect the cycle and report that the graph does not have a topological ordering.

Therefore, the algorithm correctly computes a topological ordering of G if one exists, and reports correctly if the graph contains a cycle and does not have a topological ordering. Hence, this is proved by induction that this algorithm is correct.

#### 1.3

Provide an analysis of the running time of your algorithm.

### Time Complexity

The execution time of the extended algorithm for computing the topological order of any directed graph is O(V + E).

The main loop of the algorithm runs once for each node in the graph, removing a node from S, adding it to L, and removing all outgoing edges from that node at each iteration.

The sum of degrees of all vertices in the graph is E, so the total time spent removing edges is O(E).

For each removed edge, the algorithm checks the destination node for incoming edges.

So the total time spent adding nodes to S is also O(E). Checking the remaining edges takes O(E) time.

So, as mentioned above, the total running time of the algorithm is O(V + E).

## Problem 6

Prove that for any directed graph G, the transpose of the component graph of GT is the same as the component graph of G

## Solution

## Constructive Proof:

To prove that the component graph transpose GT is the same as the component graph of any directed graph G, we must show that these two graphs have the same vertices and edges. The component graph of a directed graph G is such that the nodes represent the strongly connected components (SCCs) of G and the edges from any one node of the first SCC to any one node of the second SCC.

When considering the transpose of the component graph of GT, it is important to note that this graph has the same vertices as the component graph of G since the vertices of GT are equivalent to the vertices of G. However, it must also be shown that these graphs have the same edges.

If an edge from SCC 1 to SCC 2 exists in the component graph of G, this implies that there is at least one edge in G from anyone node in SCC 1 to a node in SCC 2. Since GT is the transpose of G, there must be at least one edge from a node in SCC 2 of GT to a node in SCC 2. Thus, in the transpose of the component graph of G, there exists an edge from SCC 2 to SCC 1.

Conversely, if there is an edge from SCC 1 to SCC 2 in the transpose of GT's component graph, this means that there is at least one edge from GT's node of his SCC 2 to the node of SCC 1. means Since GT is the transpose of G, this means that there is at least one edge in G from a node in SCC 1 to a node in SCC 2. Therefore, the component graph of G has an edge from SCC 1 to SCC 2. Having shown that two graphs share the same vertices and edges, we can conclude that the component graph transpose of GT is the same as the component graph transpose of any directed graph G.

To prove that the component graph transpose GT is the same as the component graph of any directed graph G, we must show that these two graphs have the same vertices and edges. The component graph of a directed graph G is such that the nodes represent the strongly connected components (SCCs) of G and the edges from any one node of the first SCC to any one node of the second SCC.

When considering the transpose of the component graph of GT, it is important to note that this graph has the same vertices as the component graph of G since the vertices of GT are equivalent to the vertices of G. However, it must also be shown that these graphs have the same edges.

If an edge from SCC 1 to SCC 2 exists in the component graph of G, this implies that there is at least one edge in G from anyone node in SCC 1 to a node in SCC 2. Since GT is the transpose of G, there must be at least one edge from a node in SCC 2 of GT to a node in SCC 2. Thus, in the transpose of the component graph of G, there exists an edge from SCC 2 to SCC 1.

Conversely, if there is an edge from SCC 1 to SCC 2 in the transpose of GT's component graph, this means that there is at least one edge from GT's node of his SCC 2 to the node of SCC 1. means Since GT is the transpose of G, this means that there is at least one edge in G from a node in SCC 1 to a node in SCC 2. Therefore, the component graph of G has an edge from SCC 1 to SCC 2. Having shown that two graphs share the same vertices and edges, we can conclude that the component graph transpose of GT is the same as the component graph transpose of any directed graph G.

## Proof by Contradiction:

Assume that for a directed graph G, the transpose of the component graph of GT is not the same as the component graph of G.

This means that there exists at least one node or edge in one graph that is not present in the other graph. Let's consider the case where there exists a node in the component graph of G that is not present in the transpose of the component graph of GT.

Since the nodes in the component graph of G represent strongly connected components of G, this means that there is at least one SCC in G that is not strongly connected in GT. However, by definition of the transpose of a graph, if there exists an edge from node u to node v in G, then there exists an edge from node v to node u in GT.

This means that if there is an SCC in G that is not strongly connected in GT, then there must be at least one edge in G that is not present in GT.

But this contradicts the assumption that the transpose of the component graph of GT is not the same as the component graph of G.

Therefore, our initial assumption must be false, and we can conclude that for any directed graph G, the transpose of the component graph of GT is the same as the component graph of G.

Assume that for a directed graph G, the transpose of the component graph of GT is not the same as the component graph of G.

This means that there exists at least one node or edge in one graph that is not present in the other graph.

Let's consider the case where there exists a node in the component graph of G that is not present in the transpose of the component graph of GT.

Since the nodes in the component graph of G represent strongly connected components of G, this means that there is at least one SCC in G that is not strongly connected in GT.

However, by definition of the transpose of a graph, if there exists an edge from node u to node v in G, then there exists an edge from node v to node u in GT.

This means that if there is an SCC in G that is not strongly connected in GT, then there must be at least one edge in G that is not present in GT.

But this contradicts the assumption that the transpose of the component graph of GT is not the same as the component graph of G.

Therefore, our initial assumption must be false, and we can conclude that for any directed graph G, the transpose of the component graph of GT is the same as the component graph of G.