UNIT 1 GRAPH ALGORITHMS- II

Structure **Page No**

1.1 Introduction
  1.1.1 Objectives
  1.1.2 Graph Basics
  1.1.3 Representation of Graph
1.2 Minimum Cost Spanning Tree (MCST)
  1.2.1 Generic MST algorithm
  1.2.2 Kruskal's Algorithm
  1.2.3 Prim's Algorithm
1.3 Single Source Shortest Path
  1.3.1 Dijkstra's Algorithm
  1.3.2 Bellman-Ford Algorithm
1.4 Maximum Bipartite Matching
1.5 Solution/ Answers
1.6 Further Readings

1.1 INTRODUCTION

Graph is a non-linear data structure just like tree that consist of set of vertices and edges and it has lot of applications in computer science and in real world scenarios. In a real-world, we can see graph problem in road networks, computer networks, and in social networks. What if you want to find the cheapest and shortest path to reach from one place to other places? What will be the cheapest way to connect among computer networks? What efficient algorithm will you use to find out communities (friends or friend of friend) in Facebook? The answer to all these problems is one and only one Graph Algorithm. Using graph algorithm you can find the shortest path, cheapest path and predicted outputs. Graph is used to model and represents a variety of systems and it is useful in both computer-science and real world.

Real Life-example of Graph:

- Maps: You can think of map as a graph where intersections of roads are vertices and connecting roads are edges.
- Social Networks: It is another example of graph structure where peoples are connected based on the friendships, or some relationships.
- Internet: You can think of internet as graph structures, where there are certain webpages and each webpages is connected through some link

1.1.1 Objectives:

After completion of this unit, you will be able to:

- Understand the basics of graph, graph representation.
- Understand greedy method to solve the graph optimization problem such as minimum cost spanning tree and single source shortest path.
- Apply minimum cost spanning tree algorithm on the graph and apply single source shortest path to find the shortest path from source vertex.

1.1.2Graph Basics

A graph is just like tree is a collection of vertices and edges where each edge is connected with a pair of vertices. But in a tree there are certain constraints such as tree should be acyclic, always connected and directed graph and there must be a root element. While in graph there is no-root element, it can be connected or disconnected graph, can be directed or undirected, there may or not be cycle in the graph.

- In a tree if there are n nodes, it can have maximum n-1 edges.
- Each edges shows parent-child relationship and each node except the root node have a parent.
- In a tree all nodes must be reachable from root and there is exactly one path from root to child.

While in a Graph:

- In a graph there are no rules to dictating the connections.
- There is no root node in a graph. Graph can be cyclic.
- A graph contains set of edges and vertices and a node can connected using edge in any possible way.

In mathematical term a graph is an ordered pair of vertices and edges **G = (V, E)** where **V** is a finite number of vertices in the graph and **E** is an edge that connects pair of vertices. In Figure 1 (a) is an example of tree and (b) is an example of graph.

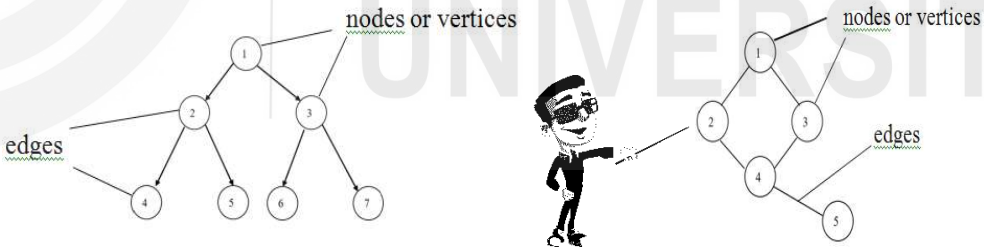
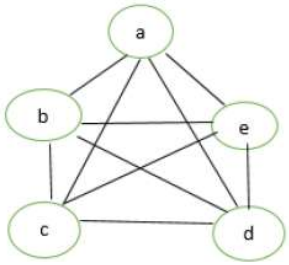


Figure 1:(a)*Tree* ***(b)*** *Graph*

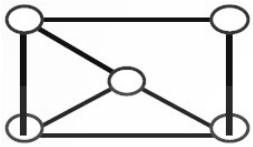
Graph Terminologies:

- **Edges:** Edge is a line that connects two vertices in a graph. A graph can have zero edges. Edges are represented as a two endpoints in an ordered pair manner. In the above graph edges are **E = {(1,2) (1,3) (2,4) (3,4) (4,5)}**.
- **Vertices:** Vertices are called as nodes. Vertices represents the connecting points in the graph. A graph can have minimum one vertex. In the above graph (b) vertices are **V = {1, 2, 3, 4, 5}**.
- **Null Graph:** A graph that single vertex and no edge is called a null graph.
- **Complete Graph:** A graph in which all vertices are connected to each other is called as complete graph. In other term in a graph from

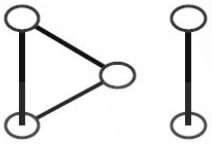
every vertex to all other vertex there must be direct path. Then that graph is a complete graph. Below figure is a complete graph of 5 vertices and each vertex is connected to all other vertex.



- **Connected Graph:** A graph is connected when there is at least one path from one vertex to other vertex. Every complete graph is connected graph but not vice-versa. Below figures shows the connected and disconnected graph of 5 vertices.



(a) Connected Graph



(b) Disconnected Graph

- **Weighted Graph:** A graph is a weighted graph in which every edge have some weights. These weights are called as cost of travelling from vertex **u** to vertex **v**.

Types of Graph:

There are two types of graph.

1. **Directed Graph:** In the directed graph there is direction given on the edge. In mathematic an edge represented as an ordered pair $(\boldsymbol{u}, \boldsymbol{v})$,where edge $\boldsymbol{u}$ is incident on vertex $\boldsymbol{v}$.Directed graph also called as digraph.
2. **Undirected Graph:** In the undirected graph there is no direction given on the edges.

In the following figure graph (a) is undirected graph and graph (b) is directed graph. In figure (b) there is a direction from vertex **1** to vertex**2, 3**. Similarly from vertex **2**to vertex **3** and vertex **1** to vertex **3**

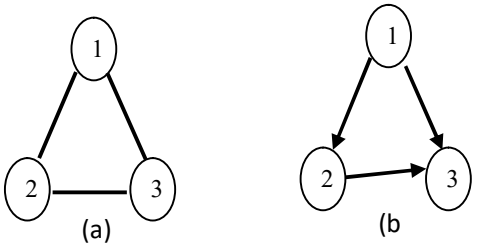


Figure 2 (a) *Represent undirected graph with 3 edges and 3 vertices.* ***(b)*** *Represents directed graph with 3 edges and 3 vertices.*

1.1.3 Representation of Graph:

Graph can be represented in two ways:

- Adjacency List
- Adjacency Matrix

Adjacency Matrix:

An adjacency matrix of a Graph $\boldsymbol{G}\ (\boldsymbol{V}, \boldsymbol{E})$ is a 2-D array of size

$\mathbf{V} * \mathbf{V}$ represented as:

$$\boldsymbol{Adj_{matrix}(aij)} = \begin{cases} \mathbf{1} & if(\boldsymbol{v_i}, \boldsymbol{v_j}) \in \boldsymbol{E}\ i.e.\ there\ is\ an\ edge\ between\ \boldsymbol{v_i}\ and\ \boldsymbol{v_j} \\ \mathbf{0} & otherwise \end{cases}$$

Example 1:

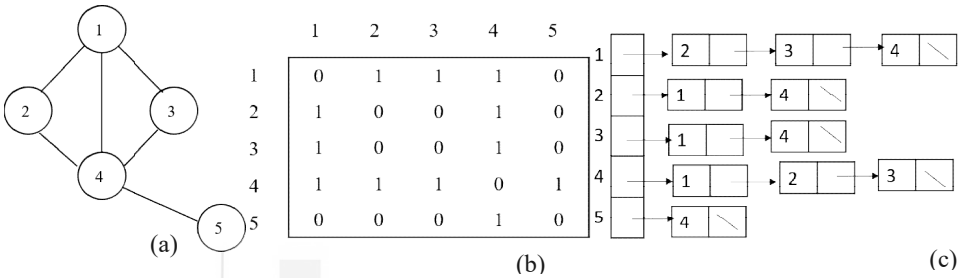


Figure3: *(a) An undirected graph G with 5 vertices and 5 edges. (b) The adjacency matrix representation of graph G. (c) The adjacency list of a graph G.*

- Adjacency matrix of the graph needs $\boldsymbol{O}(\boldsymbol{V})^{\mathbf{2}}$memory to store the data.
- Figure 3 (c) shows the adjacency matrix of the corresponding graph.

Adjacency List:

An adjacency list of graph $\boldsymbol{G}(\boldsymbol{V}, \boldsymbol{E})$is an array of list. Size of the array is the no of vertex in the graph and elements of the array are vertices. Element of $\boldsymbol{Adj}[\boldsymbol{V}]$represents the index for each vertex and corresponding list represent the number of vertices connected to that vertex. Alist nodes corresponding to vertex$\boldsymbol{v_i}$in array $\mathbf{A}[\boldsymbol{v_i}]$ represents the vertices adjacent to $\boldsymbol{v}_\mathrm{i}$. Figure 3(c) represents the adjacency list of the graph **G**. For each$\boldsymbol{u} \in \boldsymbol{V}$, the adjacency list $\boldsymbol{Adj}[\boldsymbol{u}]$contains all the vertices $\boldsymbol{v}$such that there is an edge$(\boldsymbol{u}, \boldsymbol{v}) \in \boldsymbol{E}$. If E is a number of edges in a graph. For each vertex we need to create a list that contains number of edges adjacent to that vertex. Thus space complexity will be $\boldsymbol{O}(\boldsymbol{V} + \boldsymbol{E})$**.** If the graph is a complete graph, then in worst case space complexity will be$\boldsymbol{O}(\boldsymbol{V^2})$.

1.2 MINIMUM COST SPANNING TREE (MCST)

A connected sub graph **S** of Graph$\mathbf{G}(\mathbf{V}, \mathbf{E})$is said to be spanning tree if and only ifit contains all the vertices of the graph **G** and have minimum total weight of the edges of **G**. Spanning tree must be acyclic. Spanning tree are used to solve real-world problems such as finding the shortest route, optimizing airline routes, finding least costly path in a network etc.

Spanning Tree Problem

- Consider a residential area that contains multiple roads and apartments. Two apartment u and $v$are connected by single/multiple roads has a cost $\boldsymbol{w(u,v)}$. Now you need to find the minimum path from one apartment to all other part such that:
  - All apartments are connected to each other.
  - Total cost is minimum.
- To optimize the airline route by finding the minimum miles path with no cycles. The vertices of the graph will be cities and edges will be the path from one city to other city. Miles could be the weight of those edges.

In mathematical term given an undirected graph $\boldsymbol{G(V,E)}$ and weight $\boldsymbol{w(u,v)}$on each edge $\boldsymbol{(u,v)} \in \boldsymbol{E}$. The problem is to find $S \in E$ such that:

1. S connects all vertices (S is a spanning tree), and It contains all the vertices of a Graph $\boldsymbol{G}$
2. $\boldsymbol{w(S)} = \sum_{\boldsymbol{(u,v)\in S}} \boldsymbol{w(u,v)}$ is minimized.

A Spanning tree whose weight is minimum over all spanning trees is called a minimum spanning tree or MST. Some important properties of MST are as follows:

- A MST has |V|-1 edges.
- MST has no cycles.
- It might not be unique.

1.2.1 Generic MST Algorithm

The generic algorithm for finding MSTs maintains a subset A of the set of edges E. At each step, an edge (u, v)∈E is added to A if it is not already in A and its addition to the set does not violate the condition that there may not be cycles in A. The algorithm also considers edges according to their weights in non-decreasing order. Generic MST algorithm is:

- We build a set of edges A.
- Initially **A** has no edges.
- As we add edges to A, we maintain a loop invariant: A is a subset of *some* MST for G.

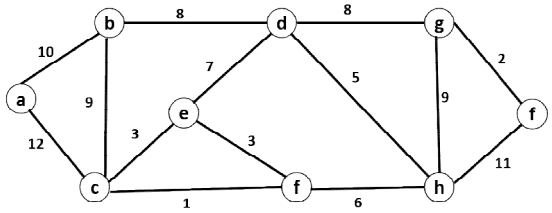
Define an edge (u, v) to be **safe** for A iff $A \cup \{(u, v)\}$ is also a subset of some MST for G. If we only add safe edges to A, once $|V| - 1$ edges have been added we have a MST for G.

Generic-MST(G, w)

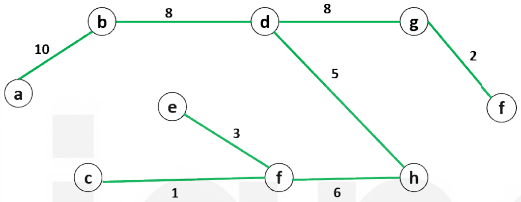
1. A= { }
2. while A is not a spanning tree
3. do find an edge (u, v) that is a safe for set A
4. A = $A \cup (u, v)$
5. return A

Safe-edge: Let G = (V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, S-V) be any cut of G that respects A, and let (u, v) be a light edge crossing the cut (S, S-V). Then, edge (u, v) is safe for A.

Example1: Find the MST of the following graph:

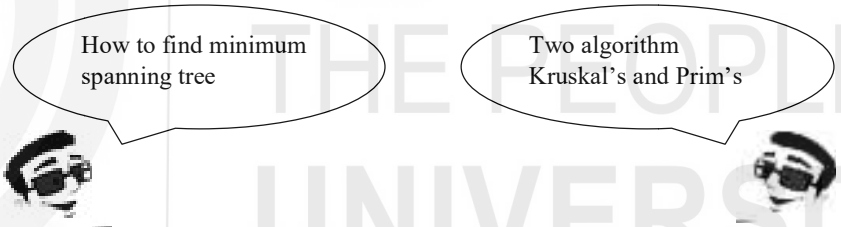


The MST of the above graph will be:



In the above example, there is more than one MST. Replace edge (e, f) by (c, e). We get a different spanning tree with same weights.

Now question arises how to find a minimum spanning tree.



Two find the minimum spanning tree of a graph G, two algorithm has been proposed:

3. Kruskal's Algorithm
4. Prim's Algorithm

1.2.2Kruskal's Algorithm

The minimum spanning tree algorithm first given by Kruskal's in 1956. Kruskal's algorithm finds a minimum weighted edge (safe edge) from a graph G and add to the new sub-graph S. Then again find a next minimum weight edge and add it to the sub-graph. Keep doing it until you get the minimum spanning tree. Intermediate steps of kruskal's algorithm may generate a forest.

Working strategy of Kruskal's algorithm:

1. Sort all the edges of a graph in a non-decreasing order of their weight.

2. Now select the minimum weight edge and check if it forms a cycle in a sub-graph. If not select that edge from a graph **G** and add it to the sub-graph. Otherwise leave it.
3. Repeat step 2 until there are $|V| - 1$ vertices in the sub graph.
4. This graph is called a minimum spanning tree.

Pseudo code of Kruskal's Algorithm:

```
MCST_Kruskal(V, E, w)

1.  K←{ }
2.  for each vertex v ∈ V
3.  MAKE − SET(v)
4.  Sort the edge E of G in a non-decreasing order by weight w
5.  for the edge (u, v) ∈ E, taken from the sorted-list
6.      if FIND − SET(u) ≠ FIND − SET(v)
7.      then K ← K ∪ {(u, v)}
8.  UNION(u, v)
9.  return K
```

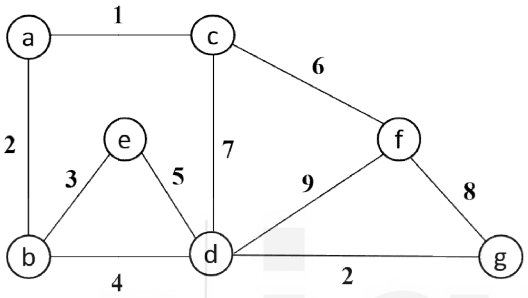
- **K**: Initially set **K** is empty. After executing the above algorithm **K** contains the edges of the MST.

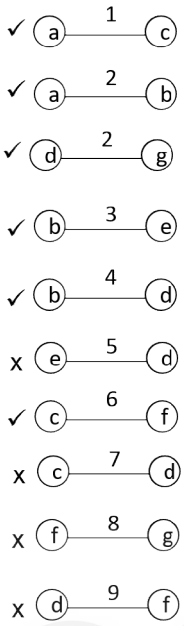
Disjoint-set data structure:

Disjoint set is a data structure that stores a collection of disjoint (non-overlapping) sets. Equivalently, it stores a partition of a set into disjoint subsets. It provides operations for adding new sets, merging sets (replacing them by their union), and finding a representative member of a set. These are the following operations on the disjoint-set:

- $MAKE - SET(v)$: Create a new set whose only member is pointed by $\boldsymbol{v}$. For this operation $\boldsymbol{v}$ must already be in a set.
- $FIND - SET(u)$: Returns an element from a set that contains $\boldsymbol{u}$.Using the $FIND - SET$ we can determine whether two vertices $\boldsymbol{u}$ and $\boldsymbol{v}$ belongs to the same tree.
- $UNION(u, v)$: It combines the dynamic sets that contains $\boldsymbol{u}$ and $\boldsymbol{v}$ into a new set. Basically $UNION$ combines the trees.

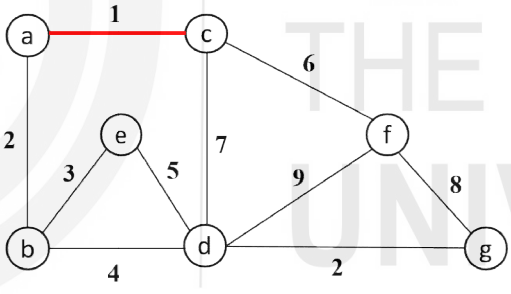
Example: Let's run the Kruskal algorithm on the following graph.





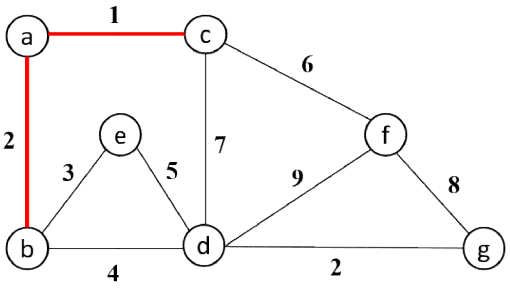
Sort all the edges in non-decreasing order:

Figure 4: *Sorted edges of a graph G*



(a)

In the graph, the edge**(a, c)** is the smallest weight edge. So we select the edge **(a, c)** in the graph **G** as highlighted.
K = {(a, c)}



(b)

In the graph, the edge **(a, b)**and edge **(d, g)**is the next smallest weight edge. Both have similar weight, you can choose any one edge. So we select the edge **(a, b)** in the graph **G**.
K = {(a, c), (a, b)}

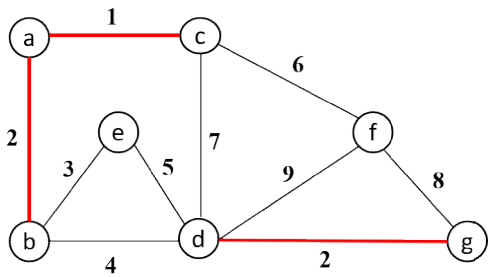
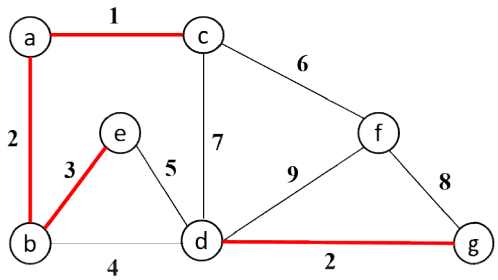
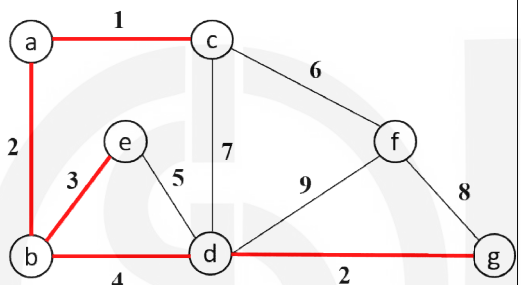
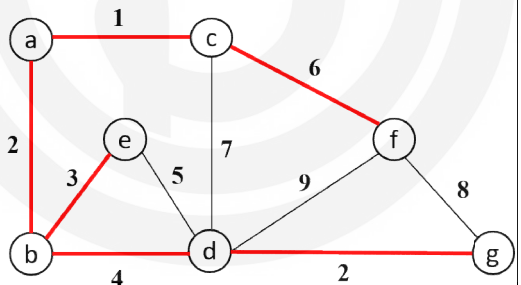
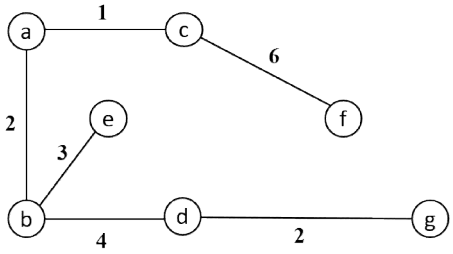
 (c)	Now edge **(d, g)** is the minimum weight edge among all the non-highlighted edge. So we select edge **(d, g)** in the graph **G**. K = {(a, c), (a, b), (d, g)}
 (d)	Now edge **(b, e)** is the minimum weight edge among all the non-highlighted edge. So we select edge **(b, e)** in the graph **G**. K = {(a, c), (a, b), (d, g), (b, e)}
 (e)	Now edge **(b, d)** is the minimum weight edge among all the non-highlighted edge. So we select edge **(b, d)** in the graph **G**. K = {(a, c), (a, b), (d, g), (b, e), (b, d)}
 (f)	Now edge **(e, d)** have the minimum weight among all the non-selected edge. But when we select that edge it will create a cycle **(e→b→d)** in the graph. So we discard the edge (e, d) and select next minimum weight edge **(c, f)** among all the non-highlighted edge. So we select edge **(c, f)** in the graph **G**. It completes the spanning tree that have exactly **|V|-1** edges and having minimum cost. K = {(a, c), (a, b), (d, g), (b, e), (b, d), (c, f)}

Figure 5: *Shows the step by step execution of kruskal's algorithm*

Final MCST of graph will be



Cost of the spanning tree= 1+2+2+3+4+6 = 18

Time analysis of Kruskal's Algorithm:

The running time of kruskal's depends on execution time of each statement in the pseudo code given above.

Line 1 Initialize the set K takes $O(1)$.

Line 2 and 3 MAKE-SET for loop take $O(|V|)$ time. It is basically the number of times loop runs.

Line 4 to sort Etakes $O(E\ log\ E)$.

Line 5-8 for the second for loop perform in $O(E)$ to $FIND-SET$ and $UNION$ on the disjoint-set forest. To check whether a safe edge or not it takes O($log\ E$) time. Thus, the total time will be $O(E\ log\ E)$

Adding all the times:
T(n) = O(1) + O(V) + O(E log E) + O(E log E)
= O(E log E) + O(E log E)(O(1) + O(V) can be neglected.)
In worst case E = V*V= V^2
T (n) = E log V^2 + E log V^2
= 2 E log V + 2 E log V
= 4 E log V
= **O (E log V)**

1.2.3PRIM's Algorithm

Prim's algorithm discovered by Prim's in 1957. Like kruskal's algorithm prim's algorithm also based on the greedy algorithm to find the minimum cost spanning tree. Here also two disjoint-sets are defined. One set contains all the vertices included in the spanning tree and other set included all the vertices that are not included in the tree. The basic idea of the prim's algorithm is very simple, it finds safe edge and keep it in the set$\boldsymbol{K}$.

Working strategy of Prims's algorithm

- We begin with some vertex v in a given graph $\boldsymbol{G(V, E)}$ defines the initial set of vertex in $\boldsymbol{K}$.
- Next choose a minimum weight edge $\boldsymbol{(u, v) \in E}$ in the graph that have one end vertex $\boldsymbol{u}$ in the set $\boldsymbol{K}$ and vertex $\boldsymbol{v}$ outside of the set $\boldsymbol{K}$.
- Then vertex$\boldsymbol{v}$ added in the set$\boldsymbol{K}$.
- Repeat this process until you get the$\boldsymbol{|V| - 1}$ edges in the spanning tree.

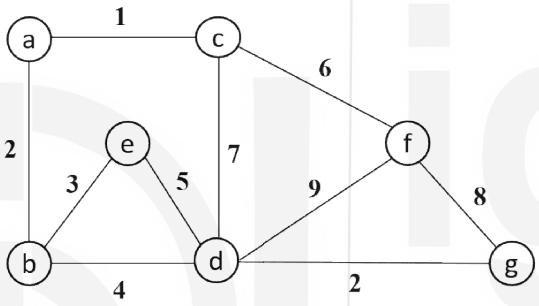
Pseudo code of Prim's Algorithm:

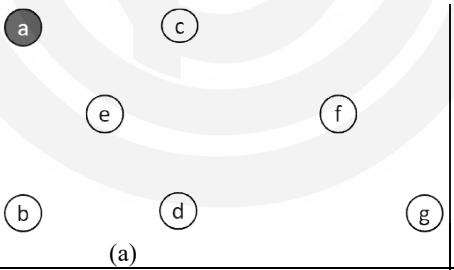
- In the pseudo code$\boldsymbol{r}$is the root of the minimum spanning tree to be grown.
- $\boldsymbol{SE}$ is the set of selected edges.
- $\boldsymbol{SV}$ is the set of vertices already in the spanning tree. Initially $\boldsymbol{SV}$ contains the root vertex **r**.
- $\boldsymbol{V}\ and\ \boldsymbol{E}$are vertices and edges of the Graph **G**.

```
MCST_Prim's( V, E, w, r )

1.  SE ← {}
2.  SV ← {r}
3.  while E ≠ ∅&& size |SE| ≠ |V| − 1
4.        Select minimum weight edge (u, v) from G
5.        If ~( u ∈ SV and v∉SV)
6.              break;
7.  E = E − {(u, v)}
8.  SE = SE ∪ {(u, v)}
9.  SV = SV ∪ {u}
10.   if (|SE| = = |V| − 1)
11. SE is a minimum spanning tree that contains |V| − 1 vertices
12.    else
13.    Graph is disconnected
```

Example: Let's run the Prim's algorithm on the following graph. Assume that root vertex $\boldsymbol{r = a}$



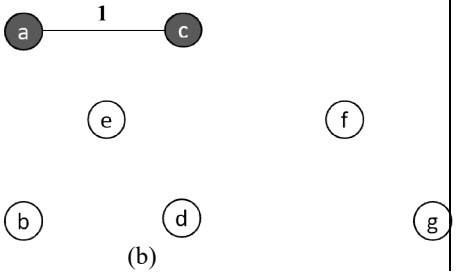


(a)

In the tree highlighted vertex are the root vertex. There is no edge selected in the tree. Initially

$SE = \{\}$

$SV = \{a\}$



(b)

Now select the minimum weight edge vertex from **G** that start with source vertex $\boldsymbol{a}$.$(\boldsymbol{a}, \boldsymbol{c})$ have minimum weight among all the edge start with vertex $\boldsymbol{a}$ and add this edge to the tree

$SE = \{(a, c)\}$

$SV = \{a, c\}$

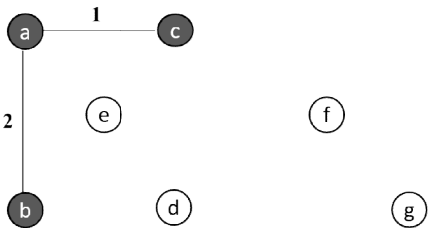
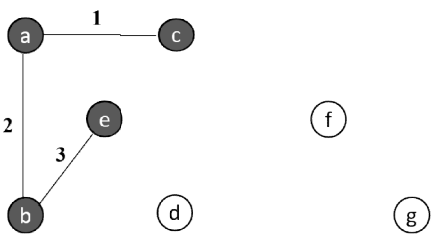
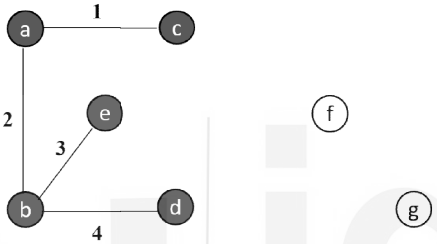
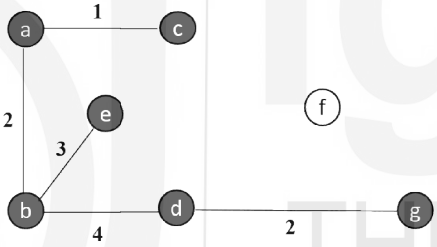
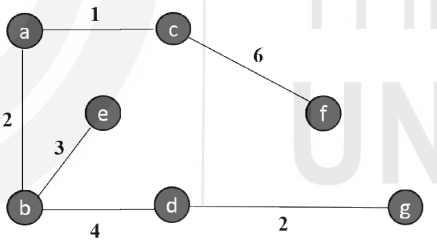
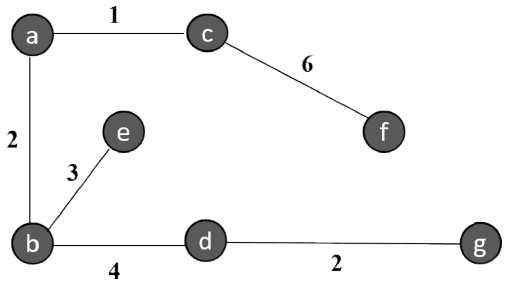
	Now select the minimum weight edge vertex from **G** that start with vertices$\boldsymbol{a}$ and vertex**c.** $(\boldsymbol{a},\boldsymbol{b})$ have minimum weight among all the edge start with vertex $\boldsymbol{a}$ and add this edge to the tree $SE = \{(a,c),(a,b)\}$ $SV = \{a,c,b\}$
	Now select the minimum weight edge vertex from **G** that start with vertices$\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}$. $(\boldsymbol{b},\boldsymbol{e})$have minimum weight among all the edge start with vertex $\boldsymbol{b}$ and add this edge to the tree. $SE = \{(a,c),(a,b),(b,e)\}$ $SV = \{a,c,b,e\}$
	Now select the minimum weight edge vertex from **G** that start with vertices $,\boldsymbol{b},\boldsymbol{c},\boldsymbol{e}$. $(\boldsymbol{b},\boldsymbol{d})$have minimum weight among all the edge start with vertex $\boldsymbol{b}$ and add this edge to the tree. $SE = \{(a,c),(a,b),(b,e),(b,d)\}$ $SV = \{a,c,b,e,d\}$
	Now select the minimum weight edge vertex from **G** that start with vertices $,\boldsymbol{b},\boldsymbol{c},\boldsymbol{e},\boldsymbol{d}$. $(\boldsymbol{d},\boldsymbol{g})$have minimum weight among all the edge start with vertex$\boldsymbol{d}$ and add this edge to the tree. $SE = \{(a,c),(a,b),(b,e),(b,d),(d,g)\}$ $SV = \{a,c,b,e,d,g\}$
	Now select the minimum weight edge vertex from **G** that start with vertices $\boldsymbol{a},\boldsymbol{b},\boldsymbol{c},\boldsymbol{e},\boldsymbol{d}$**,g.** $(\boldsymbol{e},\boldsymbol{d})$have minimum weight among all the edge start with vertex $\boldsymbol{e}$and end with **d**. Both vertex$\boldsymbol{e},\boldsymbol{d}$ in the **SV** that means adding this edge creates a cycle in the tree. Therefore, we will not add this edge and find the next minimum weight edge is $(\boldsymbol{c},\boldsymbol{f})$add this edge to the tree. $SE = \{(a,c),(a,b),(b,e),(b,d),(d,g),(c,f)\}$ $SV = \{a,c,b,e,d,g,f\}$ Now we will stop execution because all the vertices traversed once and we got $\boldsymbol{|V|-1}$ edge in the spanning tree.

Figure 6: *shows the step by step execution of the Prim's algorithm*

Final MCST of graph will be



Total cost of the spanning tree= 1+2+2+3+4+6 = 18

Time analysis of Prim's Algorithm:

Line 1 and Line 2 takes a constant time$O(1)$

In Line 3 while loop executes until all edges traversed or $|V|-1$ edges have been selected. In worst case while loops executes $E = V^2$ times$O(V^2)$

Line 4- 9 executes as many times while loop executes.

Line 10-14 take constant time.$O(1)$

Adding all the time= $O(1)$+ $O(V^2)$+ $O(1)$ = $\boldsymbol{O(V^2)}$

♣ Check your progress-1:

Q1. What is the difference between Kruskal's and Prim's algorithm.

.....

.....

.....

.....

Q2. Choose the appropriate option for greedy algorithm:

a) It finds the optimal solution.
b) It choose the best you can get right now
c) It does consider the consequences of future.
d) It used optimization to find best solution.

.....

.....

.....

.....

Q3. How many edges are in minimum cost spanning tree with n vertices and e edges.

a) $n-1$ b) $n+1$ c) n^2 d) $e-1$

.....

.....

.....

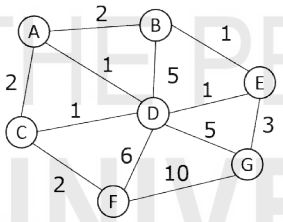
.....

Q4. Derive the complexity of kruskal's algorithm and prim's algorithm for worst and best case.

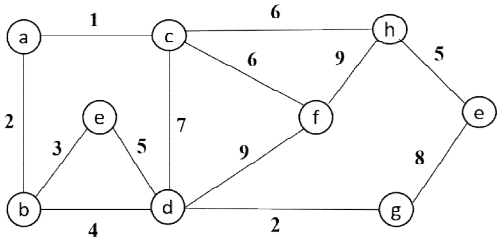
Q5.Assume that Prim's and Kruskal's algorithm runs on a Graph G. The spanning tree created from both the algorithm areS_1,S_2respectively and S_1, S_2 are not equal. Select the true statement about MCST:

a) There is an edge that have minimum weight and included in both the spanning tree.
b) There is an edge that have maximum weight and excluded in both the spanning tree.
c) Some pair of edges that have similar weight in the Graph G.
d) All edges have same weight in the Graph.

Q6. Apply the kruskal's and prim's algorithm on the following graph.



Q7. Apply kruskal's and prim's algorithm on the following graph and find the total minimum weight in the spanning tree.



.....

.....

.....

.....

1.3 SINGLE SOURCE SHORTEST PATH

In real world life graph can be used to represent the cities and their connections between each city. Vertices represents the cities and edges representing roads that connects these vertices. The edges can have weights which may be the miles from one city to other city. Suppose a person wants to drive from a city **P** to city **Q**. He may be interested to know the following queries:

- Is there any path exist from **P** to **Q**?
- If there are multiple paths from **P** to **Q**, then which is the shortest path?

The above discussed problem is considered in finding the shortest path problem. In the given weighted graph **G (V, E),** we want to find a shortest path from given vertex to each other vertex on **G**. The shortest path weight from a vertex $\boldsymbol{u} \in \boldsymbol{V}$ to a vertex $\boldsymbol{v} \in \boldsymbol{V}$ in the weighted graph is the minimum cost of all paths from $\boldsymbol{u}$to $\boldsymbol{v}$.

There are two algorithm to solve the single-source-shortest path problem.

1. Dijkstra's Algorithm
2. Bellman Ford Algorithm

1.3.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest path problem when all edges have non-negative weights. It is a greedy algorithm's similar to Prim's algorithm and always choose the path that are optimal right now not for future consequences.

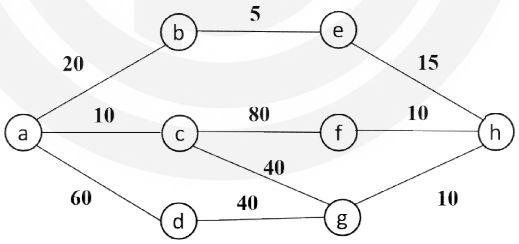


***Figure** 7: Example of greedy approach*

In the Figure 7 graph find a shortest path from vertex$\boldsymbol{a} \rightarrow \boldsymbol{h}$. In the greedy approach at first step it will chose the path who have minimum weight. Here vertex $\boldsymbol{a} \rightarrow \boldsymbol{c}$ have minimum weight **10**. So path from vertex $\boldsymbol{a} \rightarrow \boldsymbol{c}$ is selected. When we reach at vertex $\boldsymbol{c}$ we have only two option $\boldsymbol{c} \rightarrow \boldsymbol{f}$ **and** $\boldsymbol{c} \rightarrow \boldsymbol{g}, \boldsymbol{c} \rightarrow \boldsymbol{g}$ have minimum weight **40** is selected. As we can see $\boldsymbol{b} \rightarrow \boldsymbol{e}$ have minimum weight **5** but currently we are at vertex $\boldsymbol{c}$ and can't go back to choose path from $\boldsymbol{a} \rightarrow \boldsymbol{b}$. Now from $\boldsymbol{g} \rightarrow \boldsymbol{h}$ only one path having weight **10** is selected.

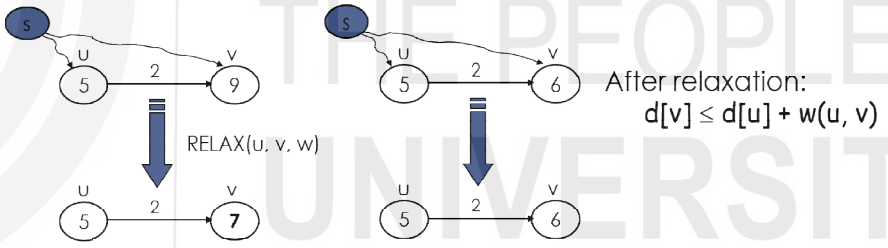
Shortest path $\boldsymbol{a} \rightarrow \boldsymbol{h}$ via $\boldsymbol{a} \rightarrow \boldsymbol{c} \rightarrow \boldsymbol{g} \rightarrow \boldsymbol{h}$and weight is **10+40+10= 50**. In greedy it chooses the optimal solution at that particular time, that's why path $\boldsymbol{a} \rightarrow \boldsymbol{c}$ selected rather than thinking of future optimal solution that is $\boldsymbol{a} \rightarrow \boldsymbol{b}$. If we choose path $\boldsymbol{a} \rightarrow \boldsymbol{b} \rightarrow \boldsymbol{e} \rightarrow \boldsymbol{h}$ then weight is **20+5+15 = 40**. This problem can be solve using dynamic programming that we will study in next module.

Working strategy for Dijkstra's algorithm:

1. Algorithm starts from a source vertex$\boldsymbol{s}$, it grows the tree **T** that spans all the vertices reachable from source vertex $\boldsymbol{s}$.
2. In the first step **0** is assigned to source vertex $\boldsymbol{s}$ and∞to all other vertex.
3. Now vertices are added in the tree **T** in order of distance i.e., first vertex $\boldsymbol{s}$, then the next vertex closest, and so on.
4. The distance from source vertex $\boldsymbol{s}$ to the reachable vertex from source is updated: $\boldsymbol{if}(\boldsymbol{dist}(\boldsymbol{s}) + \boldsymbol{w}(\boldsymbol{s},\boldsymbol{v}) < dist(\boldsymbol{v}), \boldsymbol{w}(\boldsymbol{s},\ \boldsymbol{v})$ is a weight from $\boldsymbol{s}$to $\boldsymbol{v}$. $\boldsymbol{dist}(\boldsymbol{v})$will be updated from ∞ to the $\boldsymbol{dist}(\boldsymbol{s}) + \boldsymbol{w}(\boldsymbol{s},\boldsymbol{v})$**.**
5. After updating the distance of all vertex reachable from $\boldsymbol{s}$, vertex who have minimum distance will be selected and calculate the distance of all other vertex reachable from newly selected vertex.
6. Repeat the above **3-5** steps until we traversed all the vertices of the graph.

Pseudo Code for Dijkstra's Algorithm

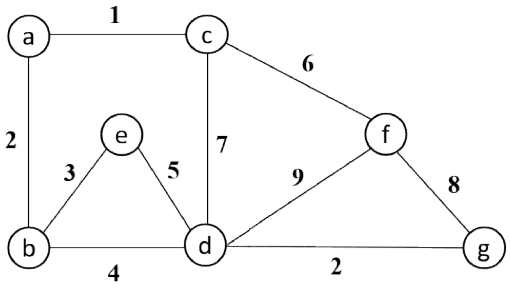
- **dist**$[\boldsymbol{V}]$ is the array that contains ∞ for all the vertices $\boldsymbol{v} \in \boldsymbol{V}$ except the source vertex $\boldsymbol{s}$. Source vertex $\boldsymbol{s}$ have zero distance $\boldsymbol{dist}[\boldsymbol{s}] \leftarrow \boldsymbol{0}$.
- **Priority queue:** stores the vertex in a queue based on the priority. In dijkstra's vertices are stored based on the vertex weight(lowest weight vertex having high priority).Initially only source vertex has 0 weight and all other vertices have ∞ weight.
- $\boldsymbol{prev}[\boldsymbol{V}]$is a predecessor array that contains initially **NULL**.
- $\boldsymbol{EXTRACT_MIN}$extract the minimum weight vertex from the priority queue.
- In the pseudo code line **9-12** are relaxation steps$\boldsymbol{RELAX}(\boldsymbol{u},\boldsymbol{v},\boldsymbol{w})$. Process of relaxing an edge $(\boldsymbol{u},\boldsymbol{v})$ consists of testing whether we can improve the shortest path to$\boldsymbol{v}$ found so far by going through $\boldsymbol{u}$If dist[v] >dist[u] + w(u, v).updating the $\boldsymbol{dist}[\boldsymbol{V}]and\ \boldsymbol{prev}[\boldsymbol{V}]$ at line 11 and 12.

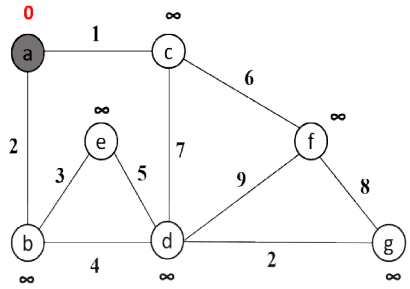


Dijkstra's(G, V, s)

1. for each vertex $v \in V$
2. dist$[V] \leftarrow \infty$
3. prev$[V] \leftarrow NULL$
4. If $v \in V! = s$ add v to the priority queue Q
5. dist$[s] \leftarrow 0$
6. while $Q \neq \emptyset$
7. $u \leftarrow EXTRACT\_MIN(Q)$
8. for each vertex v in Adj$[u]$
9. $Tdist \leftarrow dist[u] + w(u, v)$
10. if $Tdist < dist[v]$
11. $dist[v] \leftarrow Tdist$
12. $prev[v] \leftarrow u$
13. return dist[], prev[]

Example 1: Apply Dijakstra's algorithm on the following graph.



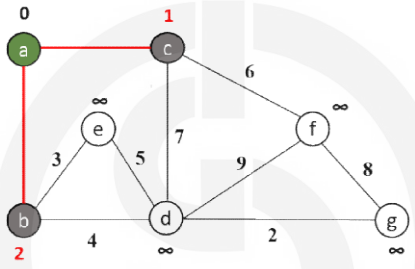


Given a graph $\boldsymbol{G(V, E)}$all the vertices have ∞ distance and only source vertex have **0** distance.

$dist[s] \leftarrow 0$

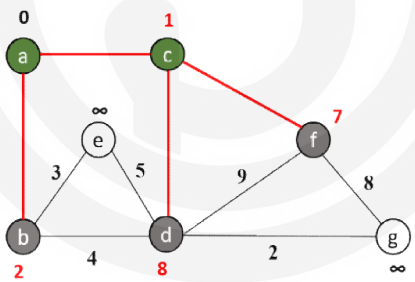
$dist[V - s] \leftarrow \infty$

	a	b	c	d	e	f	g
$dist[V] =$	0	∞	∞	∞	∞	∞	∞
$prev[V] =$	-	-	-	-	-	-	-



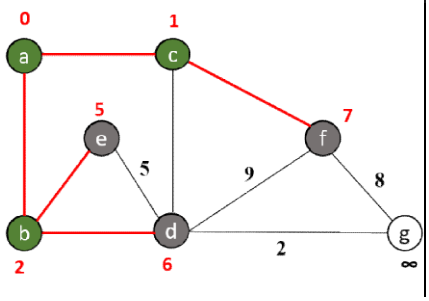
Vertex$\boldsymbol{c}$ and $\boldsymbol{b}$ are reachable from source vertex$\boldsymbol{a}$. Update the distance of $\boldsymbol{c}$ and $\boldsymbol{b}$ from ∞ to 1 and 2 respectively. Red edges shows the relaxed edge and updated weight on the vertex head. Vertex in green color are added in the tree T.

$dist[V] =$	0	**1**	**2**	∞	∞	∞	∞
$prev[V] =$	-	**a**	**a**	-	-	-	-



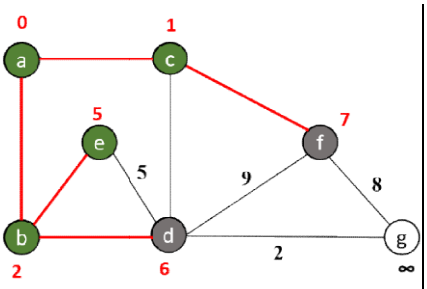
Vertex $\boldsymbol{c}$ having minimum distance is selected and added in the tree **T**. $\boldsymbol{c}$ is also called as a closest vertex to source $\boldsymbol{a}$. Update the distance of all the vertex adjacent to $\boldsymbol{c}$ except the vertex already added in the tree **T**.

$dist[V] =$	0	1	2	**8**	∞	**7**	∞
$prev[V] =$	-	a	a	**c**	-	**c**	-



Vertex $\boldsymbol{b}$ having minimum distance is selected and added in the tree **T**. Update the distance of all the vertex adjacent to **b** except the vertex already added in the tree **T**. Here $\boldsymbol{dist[d]}$ is updated from 8 to 6 because $\boldsymbol{dist[d]}$ via $\boldsymbol{c}$is higher than $\boldsymbol{dist[d]}$via **b.**

$dist[V] =$	0	1	2	**6**	**5**	7	∞
$prev[V] =$	-	a	a	**b**	**b**	c	-



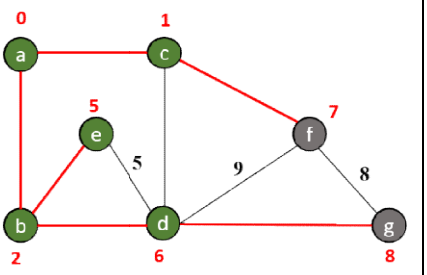
Vertex ***e***having minimum distance among all the remaining vertex is selected and added in the tree **T**.
There is only one vertex ***d*** reachable from ***e*** whose distance is **6** via vertex ***b*** less than the distance **10**via ***e***. Thus distance of***d*** will not be updated.

$dist[V] =$

0	1	2	6	5	7	∞

$prev[V] =$

-	a	a	b	b	c	-



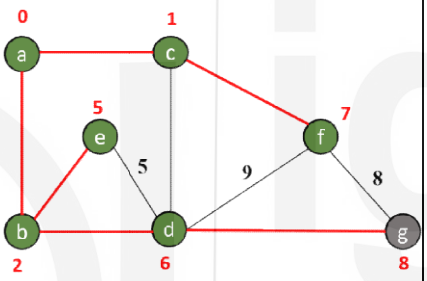
Vertex ***d*** having minimum distance among all the remaining vertex is selected and added in the tree **T**.
Vertex ***f*** and ***g*** are reachable from ***d***. Distance of vertex ***g*** updated from ∞ to **8** and weight of vertex ***f*** will not be updated because it has minimum distance from vertex ***c***.

$dist[V] =$

0	1	2	6	5	7	**8**

$prev[V] =$

-	a	a	b	b	c	**d**



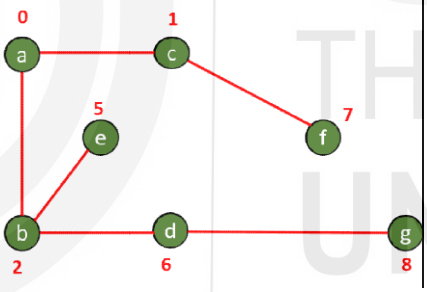
Vertex ***f*** having minimum distance among all the remaining vertex is selected and added in the tree **T**.
Vertex ***g*** is reachable from ***f*.** Distance of vertex ***g*** will not be updated because it has minimum distance from vertex ***d***.

$dist[V] =$

0	1	2	6	5	7	8

$prev[V] =$

-	a	a	b	b	c	d



gis only vertex left. ***g***is selected and added in tree **T**. Now all the vertex are selected and path from source vertex to all other vertex has been found as shown in graph highlighted in red color.

$dist[V] =$

0	1	2	6	5	7	8

$prev[V] =$

-	a	a	b	b	c	d

Figure 8:*Shows the step by step execution of Dijakstra's algorithm*

From the above graph answer the following questions:

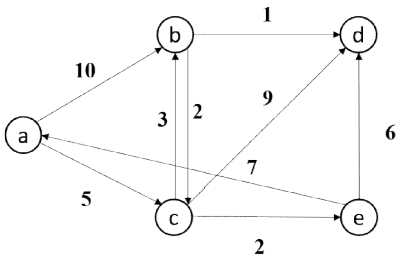
What is the shortest path from vertex ***a*** to vertex ***g***and distance of the path.

Answer: $\boldsymbol{a \rightarrow b \rightarrow d \rightarrow g}$ distance= 8 from the $prev[V]$you can find the path checking in backward direction

$prev[V] =$

-	a	a	b	b	c	d

Example 2: Apply Dijakstra's algorithm on directed graph from source vertex *a*



Method 1:

Selected vertex	a	b	c	d	e
a	**0**	∞	∞	∞	∞
c		**10**	**5**	∞	∞
e		**8**		**14**	**7**
b		**8**		**13**	
d				**9**	

You can apply the dijakstra's using the above table. In each row distance array is created and updated according to the minimum distance. Left column represents the selected vertex in the tree or order of the vertex. To find the path from vertex ***a*** to ***d***. In first row only source has **0** weight and all other vertices have ∞ weight. Now, vertex **a** is chosen and weight of all adjacent vertex to **a** is relaxed if needed. Thus, weight of **b** and **c** is relaxed. Now next minimum vertex is **c** is chosen and weights of all adjacent vertices is relaxed if needed and so forth. To find the path from vertex ***a*** to ***d***.You need to check in backward direction. First check for distance of ***d*** updated due to which vertex that is ***b*** mark **d** by arrow in table. Now check for distance of **b** updated due to which vertex that is ***c***. Now value of ***c*** updated due to ***a***. Thus path from ***a*** to ***d*** is: $a \rightarrow c \rightarrow b \rightarrow d$.

Method 2: Apply dijakstra's as explain in Example 1.

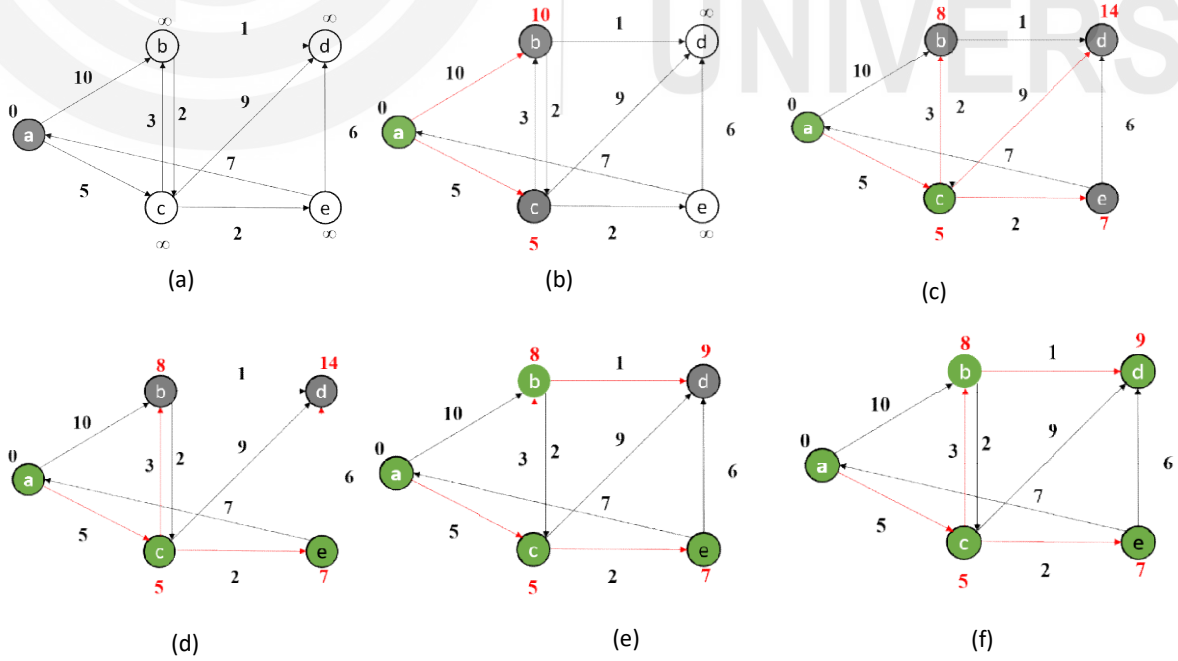


Figure 9*: Shows the step by step execution of Dijakstra's algorithm*

Figure 9 (f) is final shortest path graph. From graph we can clearly see that path vertex ***a*** to ***d***is***a*** → ***c*** → ***b*** → ***d*** highlighted edges in red color.

Dijakstra's failure for negative edge weight graph:

In the below graph apply the Dijakstra's algorithm on the negative edge weight.

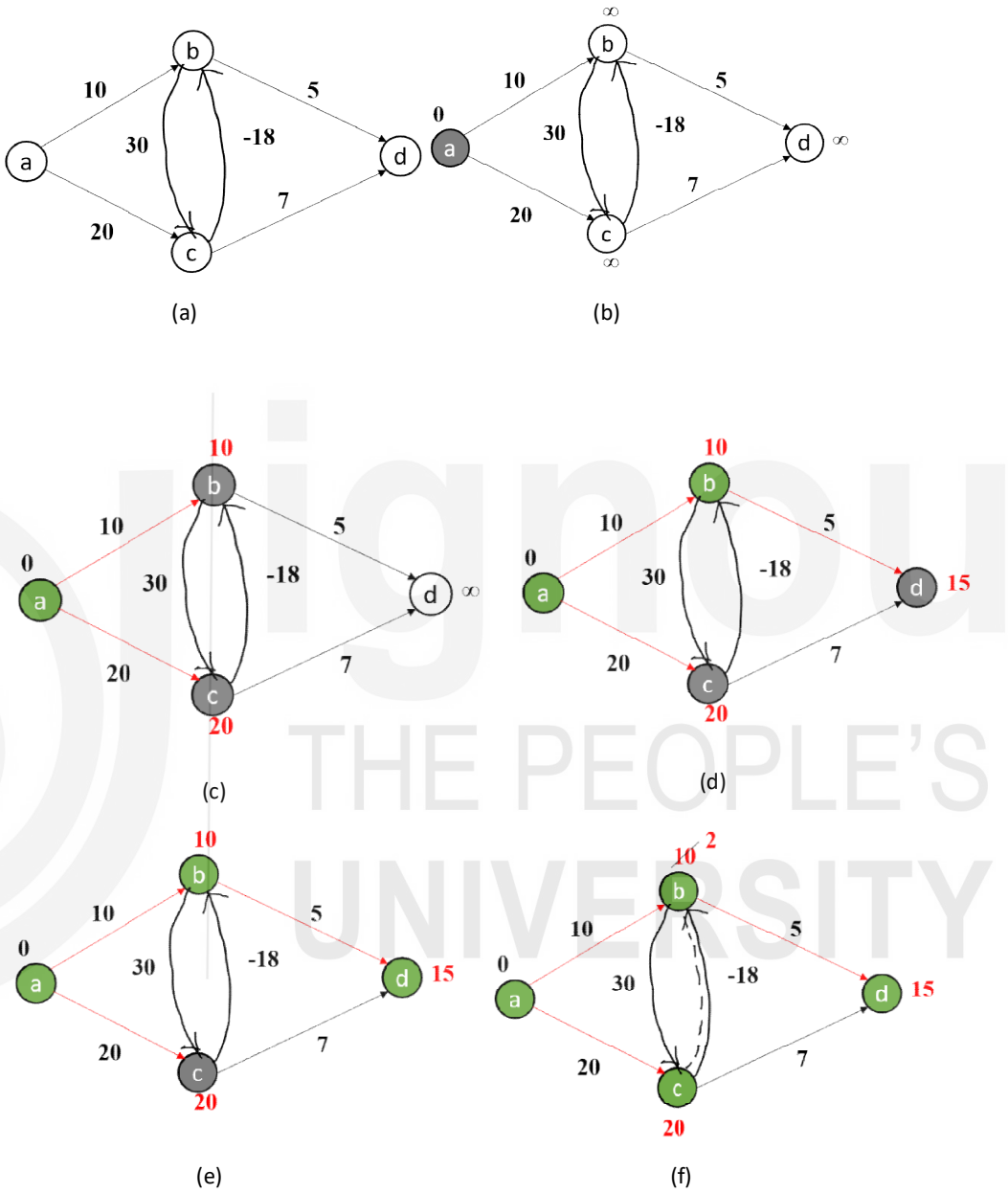


Figure 10: *The execution of dijakstra's algorithm in negative edge weight. (a) The source****a*** *is the leftmost vertex. (b) Assigned* ***0*** *to vertex* ***a*** *and∞ to the remaining vertex. (c)- (e) apply dijakstra's algorithm to update the distance and select the vertex in tree* ***T****. (f) vertex* ***c*** *is selected and distance from* ***c*** → ***b*** *is* **2** *that can't be updated because vertex****b*** *is already added in the tree.*

In Figure 10 (f) we can see that here dijakstra's algorithm fails. Because ***b*** is already traversed and added in the tree T. Distance from ***a*** → ***b*** is **10** while distance ***a*** → ***c*** → ***b*** is **2** that can't be updated. When there is a negative edge weight, dijakstra's may not work because relaxation can be done only one time. In the graph there is negative edge but no negative weight cycle. Because ***c*** → ***b***there is a minimum distance but

b → c distance is 32 that is not less than **20**. Hence there is no cycle no negative weight cycle. If weight on edge **(c, b)** is **-5** then dijakstra's algorithm works perfectly fine. Relaxation many time on any edge is called Bellman Ford algorithm.

Dijakstra's failure for negative edge weight cycle graph:

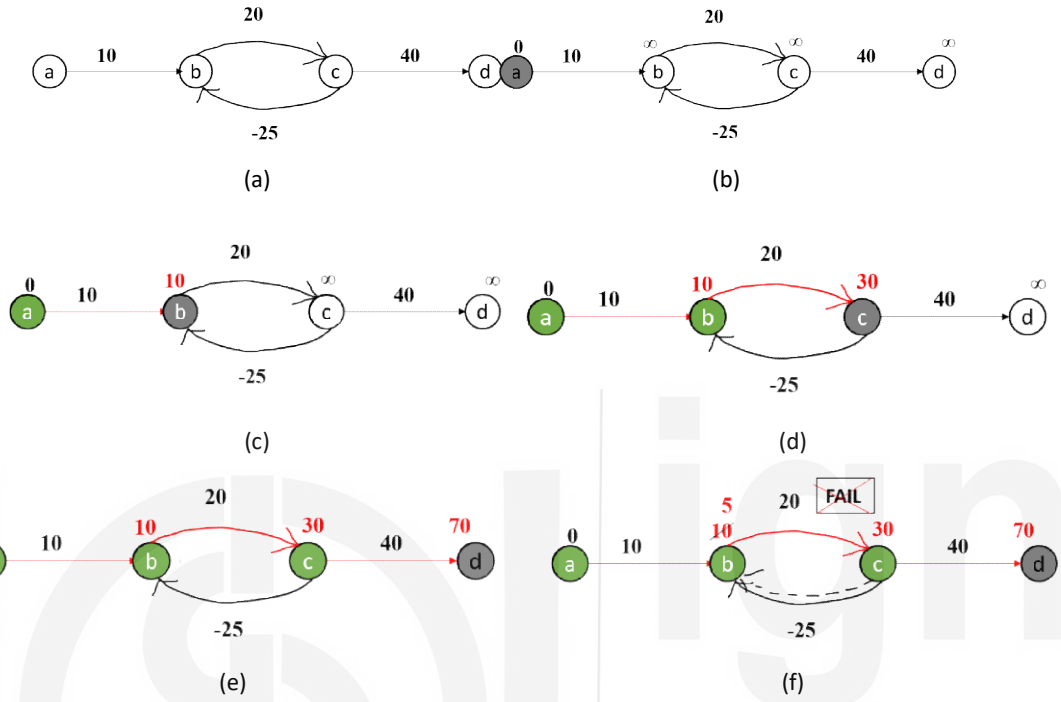


Figure 11: *(a) is a actual graph. (b)* ***0*** *assigned to the source vertex and ∞ assigned to the remaining vertex. (c)- (d) apply dijakstra's algorithm and edges are relaxed and distance are updated. (e)- (f) when distance from vertex* ***c*** *to all the reachable vertex calculated. Here distance from* ***c → b****is* ***5*** *less than actual distance* ***10****. - - arrow shows that there is minimum distance but vertex* ***b*** *already relaxed and selected it can't be updated now. Here dijakstra's fails.*

In the above graph there is negative edge weight cycle. As we can see in the Figure 12that distance from ***b → c*** and ***c → b***always reducing and keeps on reducing. This is called a negative edge weight cycle and dijakstra's always fails for negative weight cycle.

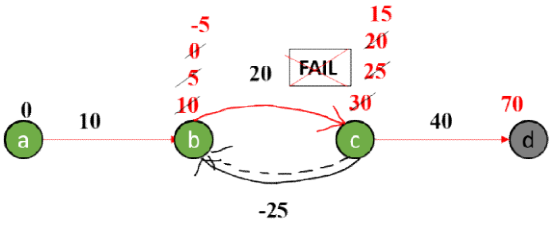


Figure 12: *shows example of negative edge weight cycle and dijakstra's always fail for negative edge weight cycle.*

Time Complexity of Dijakstra's Algorithm:

The running time of dijakstra's algorithm depends on how we implements the min-priority queue.

Line 1-4 takes $O(V)$ times.

Line 5 constant time $O(1)$

Line 6-7: While loop iterates V times. In priority queue Q total V-1 vertex are there. $EXTRACT\_MIN$ takes $O(1)$ time to extract the minimum and for V vertex time will be O(V).

Line 8-12 is a relaxation steps. For loop iterates for E times.

So total time = $O(V) + O(V^2 + E) = \boldsymbol{O(V^2)}$

If binary min-heap is used to create the priority queue then $EXTRACT\_MIN$ take $\log(V)$ times and relaxation steps also takes $\log(V)$ times.

The total running time is therefore $O\big((V+E)logV\big) = \boldsymbol{O(ElogV)}$

1.3.2 Bellman-Ford Algorithm

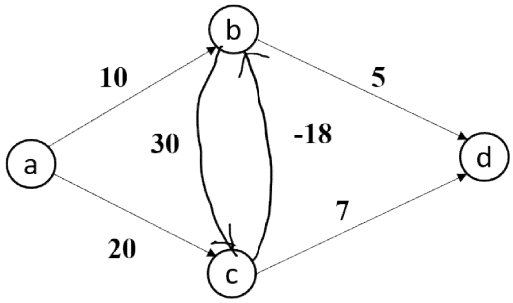
The Bellman-Ford algorithm solves the single-source shortest-paths problem in case where edge weights may be negative or there is a negative edge weight cycle in the graph. This algorithm, like dijakstra's algorithm uses the notation of edge relaxation but does not used greedy method to relax the edges. The algorithm progressively decreases the distance on vertex on the weight of the shortest path from source vertex $\boldsymbol{s}$ to each vertex $\boldsymbol{v}$in **V** until it achieve the actual shortest path. The algorithm returns a boolean value **TRUE** if the given directed graph contains no negative cycle that are reachable from the source vertex$\boldsymbol{s}$, otherwise it returns **FALSE**.

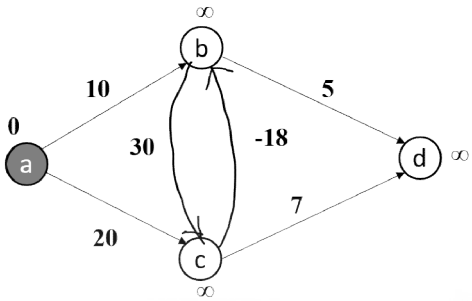
Pseudo-code of Bellman-ford algorithm

- Initialize source vertex with **0** and all other vertex ∞.
- Traverse each edge and update the distance of each edge if inaccurate.
  If **dist[v] >dist[u]** + weight of edge **(u, v),** then update **dist[v]**
  dist[v] = dist[u] + weight of edge **(u, v)**
- Do this $|\boldsymbol{V}|$ times because in a path need to be adjusted $|\boldsymbol{V}|$times.
- After all the vertices have their path check for the negative edge weight cycle in the graph.

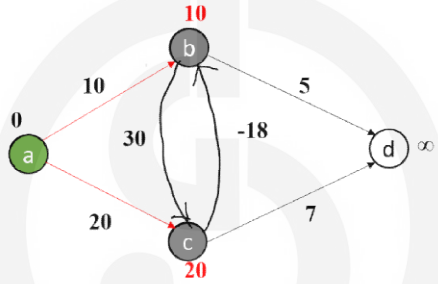
```
BELLMAN − FORD(G, V, E, w, s)
1.  Initialize dist[s]←0
2.  for each vertex v ∈ V-s
3.  dist[V] ← ∞
4.  for each vertex i=1 to V − 1
5.      for each edge (u, v) in E[G]
6.          RELAX(u, v, w)
7.  for each edge edge(u, v) in E[G]
8.      if dist[u] + w(u , v) < dist[v]
9.          return FALSE
10. return TRUE
```

Example1: Apply Bellman-Ford Algorithm on a following graph



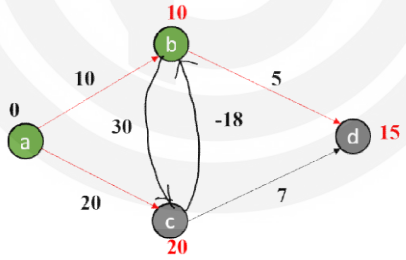


a	b	c	d
0	∞	∞	∞

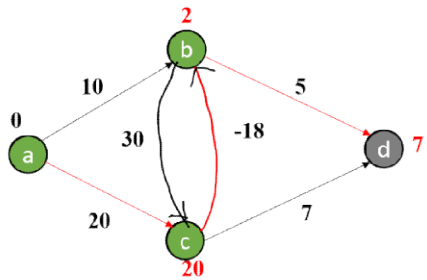


a	b	c	d
0	∞	∞	∞
0	10	∞	∞

a	b	c	d
0	∞	∞	∞
0	10	∞	∞
0	10	20	∞



a	b	c	d
0	∞	∞	∞
0	10	20	∞
0	10	20	15



a	b	c	d
0	∞	∞	∞
0	10	20	∞
0	10	20	15
0	2	20	15

a	b	c	d
0	∞	∞	∞
0	10	20	∞
0	10	20	15
0	2	20	15
0	2	20	7

.

Figure: 12*The execution of Bellman-Ford algorithm. The source vertex is* ***a****.*
Highlighted vertex in green color is a selected vertex in the corresponding step and

all edge connected to that vertex will be relax. Above figure shows the step by steps relaxation of edges and final distance written in red color on the top of each vertex.

Figure 12 shows the execution of the Bellman-ford algorithm on a graph with **4** vertices. Bellman-ford algorithm runs exactly **V-1** passes. After **V-1** passes it will check for a negative-weight cycle and return the appropriate boolean value.

- Bellman-ford gives correct answer for all vertices even though graph contain –ve edge weight
- Bellman-ford gives correct answer for all vertices even though graph contain –ve edge weight cycle. It will find out all –ve edge weight cycles which are reachable from source.

Time Complexity of Bellman-Ford Algorithm:

The initialization in line 1 takes $O(V)$ time.

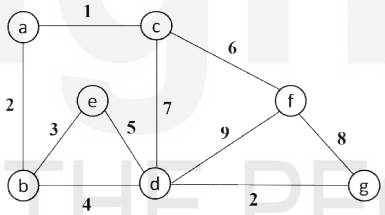
For loop of line 2 takes $O(V)$ time.

For loop of lines 3-4, it takes $O(E)$ time and for-loop of line 5-7, it takes $O(E)$

Therefore, Bellman-ford runs in $\boldsymbol{O(VE)}$**.**

♣Check you progress-2:

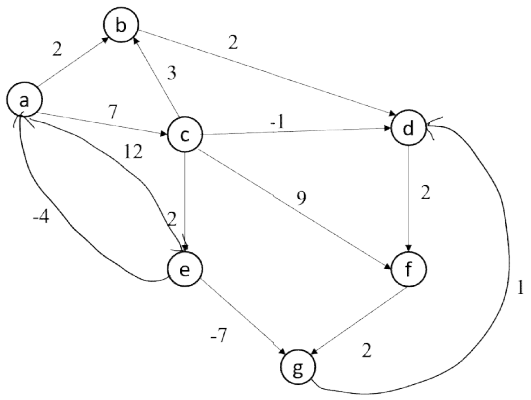
Q1. Apply Dijakstra's algorithm on the following graph with source vertex a.



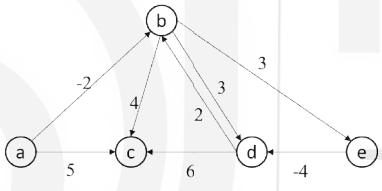
Q2. What are the disadvantages of using Dijakstra's algorithm? List out any three.

Q3. Suppose we have a graph that have no weight on the edges. Each vertex have weight/cost. Can dijakstra's algorithm applicable on such graph. If so, give proper justification and if not then why?

Q4. Apply Dijakstra's algorithm on the following graph.

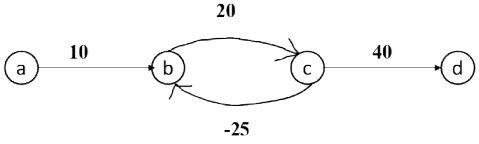


Q5. Apply Bellman-ford on the following graph. List out the order of execution of vertices.



Q6. Why bellman-ford algorithm works for negative weight cycle.

Q7. Apply bellman-ford on the following graph.



.....

.....

Q8. What are the application of bellman-for algorithms?

.....

.....

.....

.....

1.4 MAXIMUM BIPARTITE MATCHING

Maximum bipartite matching problem is a subset of matching problem specific for bipartite graphs (graphs where the vertex set **V** can be partitioned into two disjoint sets **L** and **R** i.e. **V** = **L**∪**R** and the edges **E** is present only between **L** and **R**). A matching in a bipartite graph consists of a set of edges such that no two edges **e** ∈ **E** share a vertex **v** ∈ **V**. A matching of maximum size (i.e. maximum number of edges) is known to be maximum matching if the addition of any edge makes it no longer a matching.

Many real-world problems can be represented as bipartite matching. For example, there are **L** applicants and **R** jobs with each applicant has a subset of job interest but can be hired for only one job. Figure 1 illustrates the matching problem and a solution.

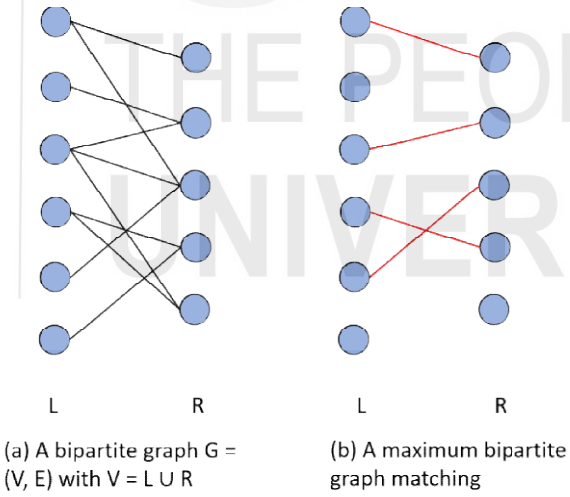


***Figure 13**: Illustration of a bipartite matching problem a maximum matching*

Solution to find a maximum bipartite matching

The maximum bipartite matching problem can be solved by transforming the problem into a flow network. Next, Ford-Fulkerson algorithm can be used to find a maximum matching. Following are the steps:

A. Construct a flow network:
   A flow network **G' = (V', E')** is defined by adding a source node **s** and sink node to the bipartite graph **G = (V, E)** such that **V' = V ∪ {s, t}** and **E' = {(s, u): u ∈ L} ∪ {(u, v): (u, v) ∈ E} ∪ {(v, t): v ∈ R}**. Here **L** and **R** are the

partitions of vertex **V** as shown in Figure 1 (a). The capacity of every new edge is marked as **1**. The flow network constructed for bipartite graph given in Figure 1 is shown in Figure 2.

B. Find the maximum flow:

Ford-Fulkerson algorithm is the most efficient method to find a solution for the flow network. It relies on the Breadth First Search (BFS) to pick a path with minimum number of edges. Ford-Fulkerson, computes a maximum flow in a network by applying the greedy method to the augmenting path approach used to prove max-flow. The intuition behind it that keep augmenting flow among an augmenting path until there is no augmenting path.

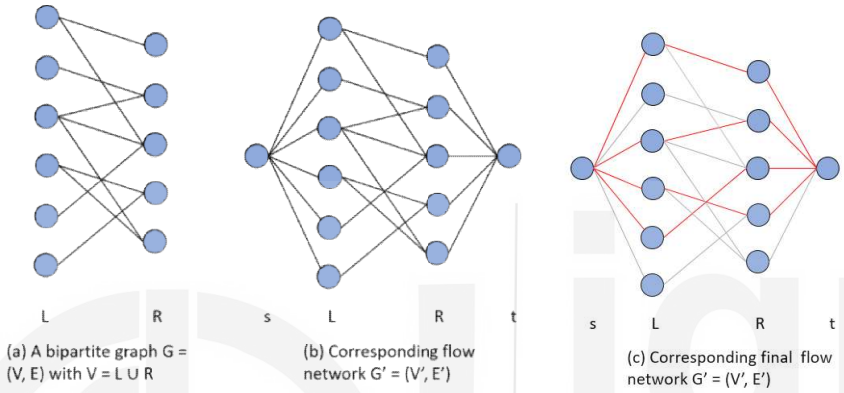


*Figure 14: Flow network corresponding to the bipartite graph. Right most graph shows the solution for a maximum flow problem (the final flow network of (b)). Each edge have unit capacity, and shaded edge from **L** to **R** corresponds to those in the maximum matching from (b).*

Pseudo code for Ford-Fulkerson algorithm:

- The main idea of algorithm is to incrementally increase the value of flow in stages, where at each stage some amount of flow is pushed along an augmenting path from source to the sink.
- Initially the flow of each edge is equal to **0**.
- In line 3, at each stage path **p** is calculated and amount of flow equal to the residual capacity c_f of **p** is pushed along **p**.
- The algorithm terminates when current flow does not accept an augmenting path.
- An augmenting path is a path p from the start vertex **(s)** to the end vertex **(t)** that can receive additional flow without going over capacity.
- Line 5 is adding flow and line 6 is reducing flow in the edge belongs to the path **p**.

Ford − Fulkerson(G, s, t)

Inputs Given a Network $G' = (V', E')$ with flow capacity c, a source node s, and a sink node t

Output Compute a flow f from s to t of maximum value

1. $f(u, v) \leftarrow 0$ for all edges (u, v)
2. *While* there is a path p from s to t in G_f, such that $c_f(u, v) > 0$ for all edges $(u, v) \in$ p
3. Find $c_f(p) = \min \{c_f(u, v) : (u, v) \in \text{p}\}$
4. For each edge $(u, v) \in$ p

5. $f(u,v) \leftarrow f(u,v) + c_f(p)$(*Send flow along the path*)
6. $f(u,v) \leftarrow f(u,v) - c_f(p)$ (*The flow might be "returned" later*)

Since in the current implementation of Ford Fulkerson algorithm, it is using BFS, the algorithm is also known as Edmonds-Karp algorithm.

♣Check you progress-3:

Q1. What is the worst-case running time of the Ford-Fulerson algorithm if all edge capacities are bounded by a constant.

.....

.....

.....

.....

Q2. Let K be a flow network with v vertices and e edges. Show how to compute an augmenting path with the largest residual capacity in $O(v(v+e)log\ v)$ time.

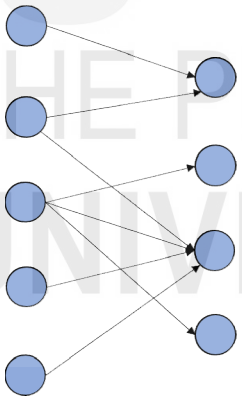
.....

.....

.....

.....

Q3. Find maximum matching of below graph.



.....

.....

.....

.....

Q4.InFord-Fulerson algorithm which graph traversal algorithm is used to pick a path with minimum number of edges.

.....

.....

.....

.....

1.5 SOLUTION / ANSWERS

♣Check you progress-1

Q1.

	Kruskal's Algorithm	Prim's Algorithm
1	Kruskal's algorithm creates a forest (more than a connected components) in the intermediate step of spanning tree creation.	In prim's algorithm there will be only one connected components.
2	Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST.	Prim's algorithm always selects a vertex (say, v) to find MCST
3	Time Complexity: $O(E \log V)$	Time Complexity: $O(V^2)$

Q2. b)

Q3. a)

Q4. Complexity of Prim's Algorithm:

Line 1 and Line 2 takes a constant time $O(1)$

In Line 3 while loop executes until all edges traversed or $|V| - 1$ edges have been selected. While loop executes $O(E)$times.

Line 10-14 take constant time. $O(1)$

Adding all the time= $O(1)$+ $O(E)$ + $O(1)$ = $O(E)$
Now in worst case when graph is a complete graph, you need to traverse all the edges. The no of edges

$$E = \frac{|V|(|V|-1)}{2} = O(|V|^2) = O(V^2)$$

Now in the average or best case to find spanning tree graph must be connected. Means there must be at least $|V| - 1$ edges in the graph. Thus in best or average case complexity will be = $O(E)$

Complexity of Kruskal's Algorithm:

The running time of kruskal's depends on execution time of each statement in the pseudo code given above.

Line 1 Initialize the set K takes $O(1)$.

Line 2 and 3 MAKE-SET for loop take $O(|V|)$ time. It is basically the number of times loop runs.

Line 4 to sort Etakes $O(E \log E)$.

Line 5-8 for the second for loop perform $O(E)$ $FIND-SET$ and $UNION$ on the disjoint-set forest. To check whether a safe edge or not it takes O(log E) time. So time will be $O(E \log E)$

Adding all the times:
T(n) = O(1) + O(V) + O(E log E) + O(E log E)
= O(E log E) + O(E log E)(O(1) + O(V) can be neglected.)

In worst case E = V*V= V^2

T (n) = E log V^2 + E log V^2

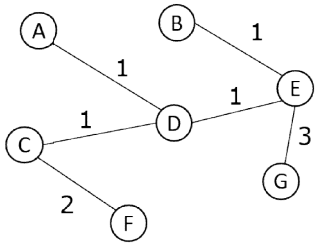
= 2 E log V + 2 E log V

= 4 E log V

= O (E log V)

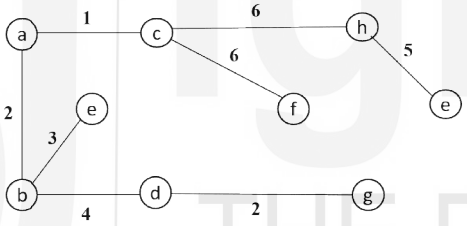
Q5. c)

Q6.



Total Weight = 1+1+1+1+2+3 = 9

Q7.



Total Weight= 1+2+2+3+4+5+6+6 = 29

♣Check you progress-2 answers:

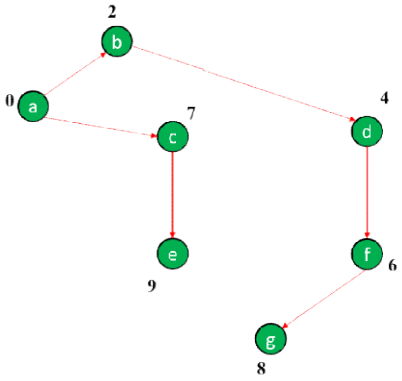
Q2. Dijakstra's algorithms search blindly to find the path then update distance. It waste lot of time while processing.

Dijakstra's algorithms fail for negative edge weight cyclic graph.

In dijakstra's algorithms need to keep track of already visited vertices. It take storage space to store visited vertices.

Q3. Yes on such graph dijakstra's is applicable. In dijakstra's algorithm we find the distance from one vertex to other vertex and given a source vertex find the lowest-cost path from source to every other vertex. The cost of a path is the sum of the weights of the vertices on the path.

Q4. Order of execution of vertices: a, b, d, f, c, g, e



Q5.

a	b	c	d	e
0	∞	∞	∞	∞
0	-2	∞	∞	∞
0	-2	5	∞	∞
0	-2	2	∞	∞
0	-2	2	∞	1
0	-2	2	1	1
0	-2	2	-3	1
0	-2	2	-3	1

Q6. Bellman-Ford calculates the shortest distance to *all* vertices in the graph from the source vertex. Dijkstra's algorithm is a greedy algorithm that selects the nearest vertex that has not been processed. Bellman-Ford, on the other hand, relaxes *all* of the edges.

Q7.

a	b	c	d
0	∞	∞	∞
0	10	∞	∞
0	10	30	∞
0	5	30	∞
0	5	25	70
0	0	20	65

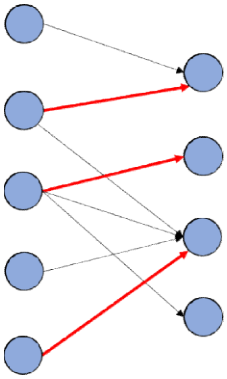
Q8. Bellman-ford algorithm can be used in to route packets of data on network used in distance vector routing protocol.

♣Check you progress-3 answers:

Q1. If every edge capacity is bounded by a constant m, then the maximum flow must be bounded by O(m) and the algorithm would run in $O(m^2)$in worst-case.

Q2. To compute an augmenting path with the largest residual capacity, we use maximum spanning tree algorithm, which is just like a minimum spanning tree algorithm with all the weights multiplied by -1. Thus to compute augmenting path it takes $\boldsymbol{O(v(v+e)log\ v)}$time.

Q3. Maximum flow must use the maximum number of unitary capacity edges across the cut (L, R). Consider unitary capacity because no maximum flow is given on the edges.



Q4. Breadth first search

1.6 FURTHER READING

1. Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson (PHI)

2. Foundations of Algorithms, R. Neapolitan & K. Naimipour: (D.C. Health & Company, 1996).

3. Algoritmics: The Spirit of Computing, D. Harel: (Addison-Wesley Publishing Company, 1987).

4. Fundamentals of Algorithmics, G. Brassard & P. Brately: (Prentice-Hall International, 1996).

5. Fundamental Algorithms (Second Edition), D.E. Knuth: (Narosa Publishing House).

6. Fundamentals of Computer Algorithms, E. Horowitz & S. Sahni: (Galgotia Publications).

7. The Design and Analysis of Algorithms, AnanyLevitin: (Pearson Education, 2003).

8. Programming Languages (Second Edition) — Concepts and Constructs, Ravi Sethi: (Pearson Education, Asia, 1996).