UNIT 1 GRAPH ALGORITHMS- II

Structure **Page No**

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.

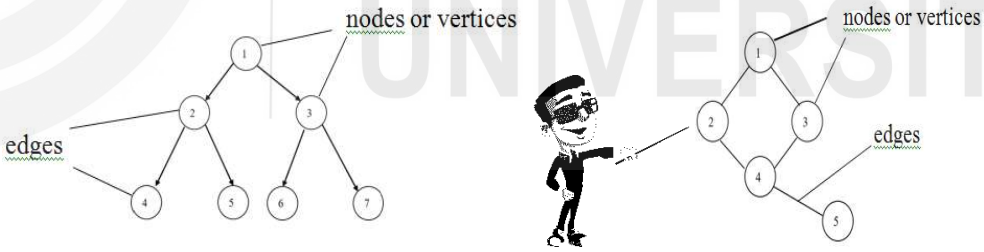
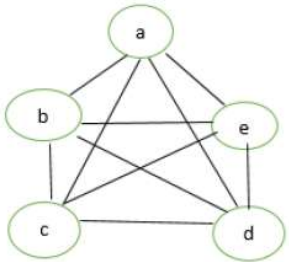


Figure 1:(a)*Tree* ***(b)*** *Graph*

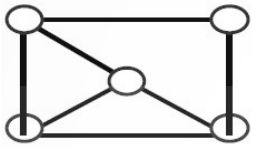
Graph Terminologies:

- **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

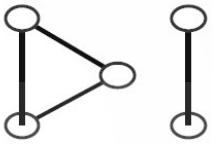
every vertex to all other vertex there must be direct path. Then that graph is a complete graph. Below figure is a complete graph of 5 vertices and each vertex is connected to all other vertex.



- **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**

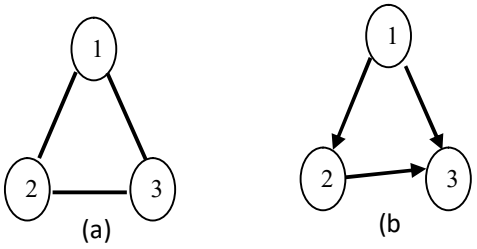


Figure 2 (a) *Represent undirected graph with 3 edges and 3 vertices.* ***(b)*** *Represents directed graph with 3 edges and 3 vertices.*

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:

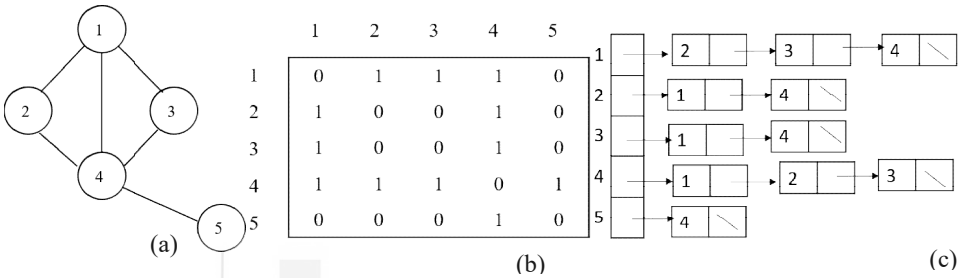


Figure3: *(a) An undirected graph G with 5 vertices and 5 edges. (b) The adjacency matrix representation of graph G. (c) The adjacency list of a graph G.*

- 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.

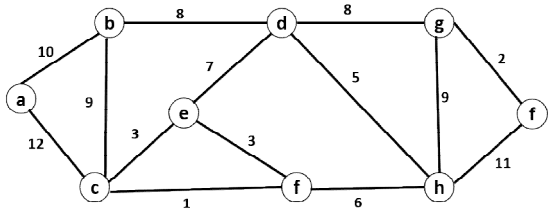
Define an edge (u, v) to be **safe** for A iff $A \cup \{(u, v)\}$ is also a subset of some MST for G. If we only add safe edges to A, once $|V| - 1$ edges have been added we have a MST for G.

Generic-MST(G, w)

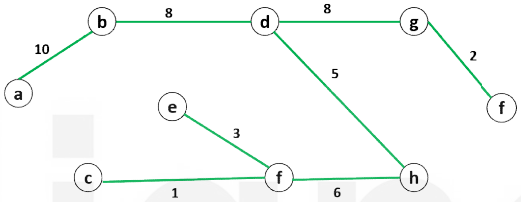
1. A= { }
2. while A is not a spanning tree
3. do find an edge (u, v) that is a safe for set A
4. A = $A \cup (u, v)$
5. return A

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:

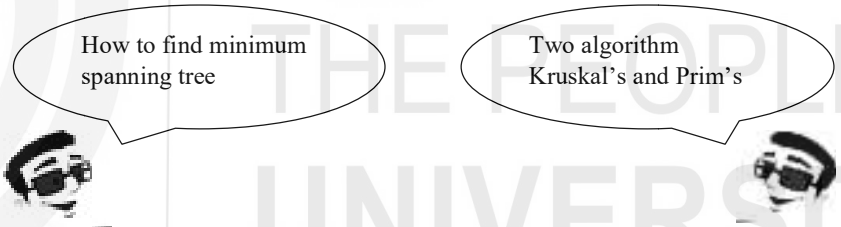


The MST of the above graph will be:



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
```

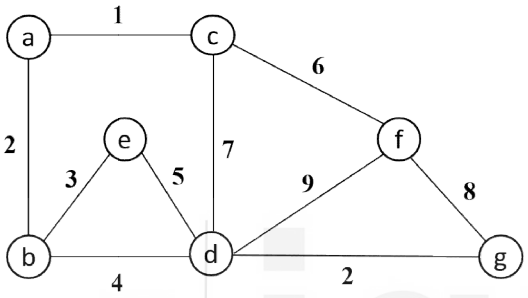
- **K**: Initially set **K** is empty. After executing the above algorithm **K** contains the edges of the MST.

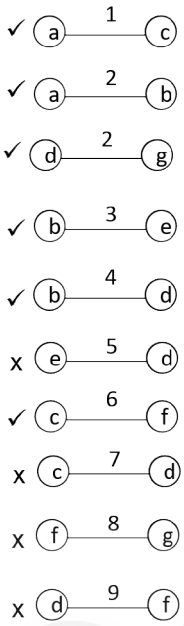
Disjoint-set data structure:

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.

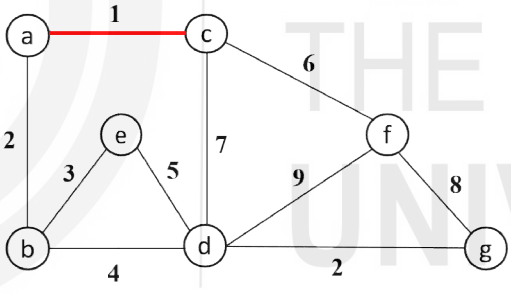
Example: Let's run the Kruskal algorithm on the following graph.





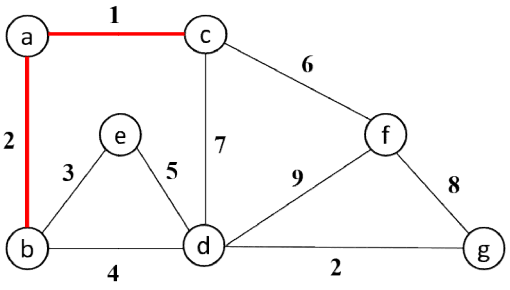
Sort all the edges in non-decreasing order:

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)}

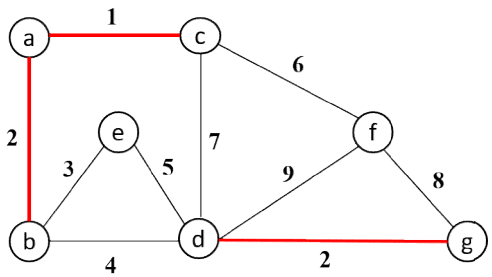
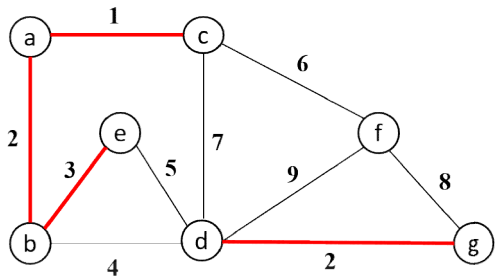
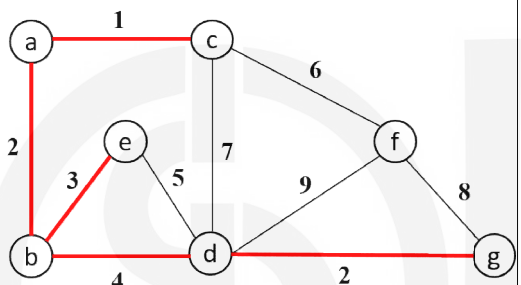
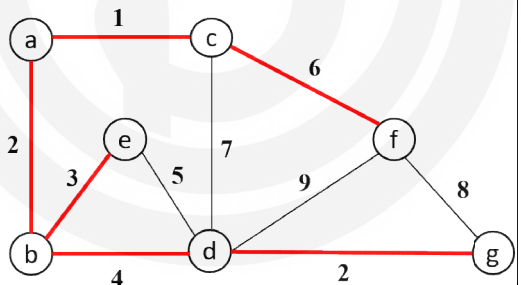
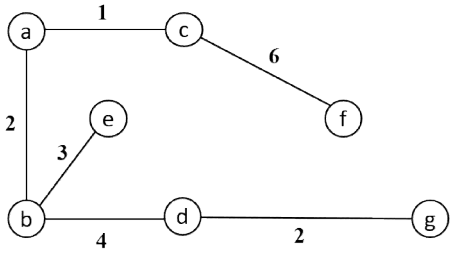
 (c)	Now edge **(d, g)** is the minimum weight edge among all the non-highlighted edge. So we select edge **(d, g)** in the graph **G**. K = {(a, c), (a, b), (d, g)}
 (d)	Now edge **(b, e)** is the minimum weight edge among all the non-highlighted edge. So we select edge **(b, e)** in the graph **G**. 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.

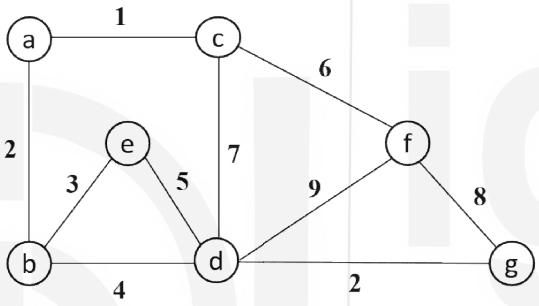
Pseudo code of Prim's Algorithm:

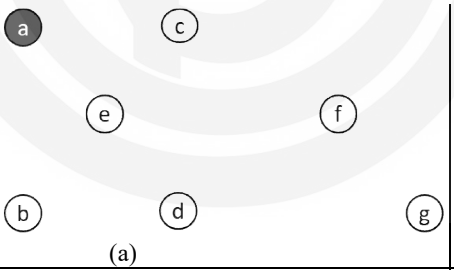
- In the pseudo code$\boldsymbol{r}$is the root of the minimum spanning tree to be grown.
- $\boldsymbol{SE}$ is the set of selected edges.
- $\boldsymbol{SV}$ is the set of vertices already in the spanning tree. Initially $\boldsymbol{SV}$ contains the root vertex **r**.
- $\boldsymbol{V}\ and\ \boldsymbol{E}$are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



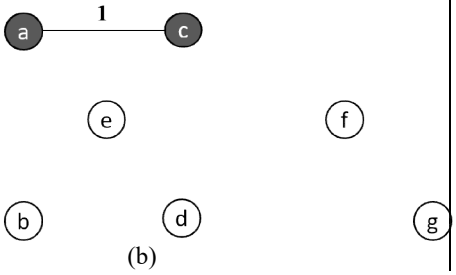


(a)

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

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

$SV = \{a, c\}$

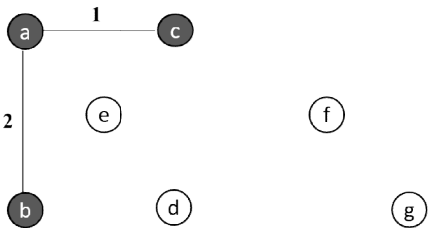
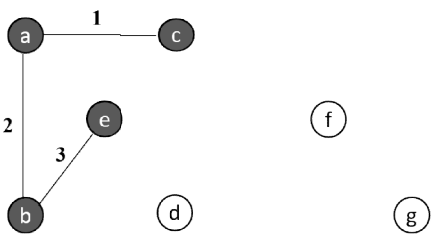
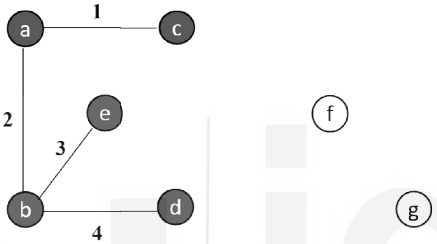
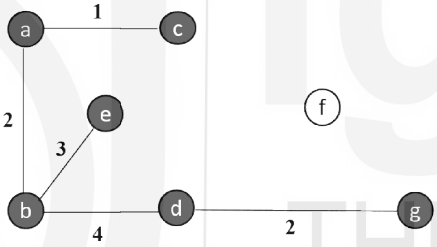
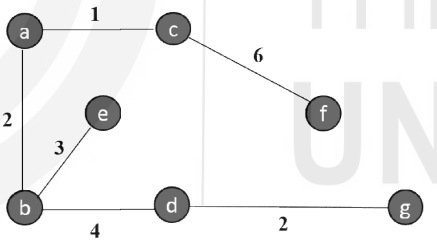
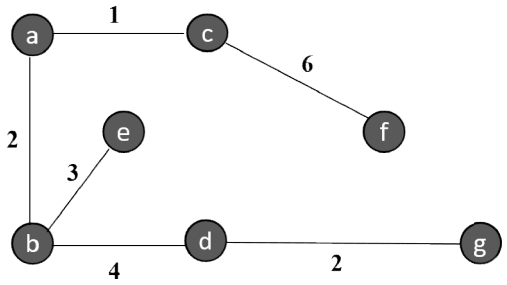
	Now select the minimum weight edge vertex from **G** that start with vertices$\boldsymbol{a}$ and vertex**c.** $(\boldsymbol{a},\boldsymbol{b})$ have minimum weight among all the edge start with vertex $\boldsymbol{a}$ and add this edge to the tree $SE = \{(a,c),(a,b)\}$ $SV = \{a,c,b\}$
	Now select 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) $e-1$

.....

.....

.....

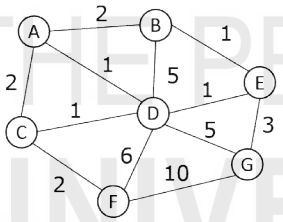
.....

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

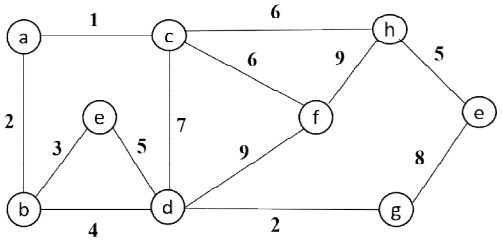
Q5.Assume that Prim's and Kruskal's algorithm runs on a Graph G. The spanning tree created from both the algorithm areS_1,S_2respectively and S_1, S_2 are not equal. Select the true statement about MCST:

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.

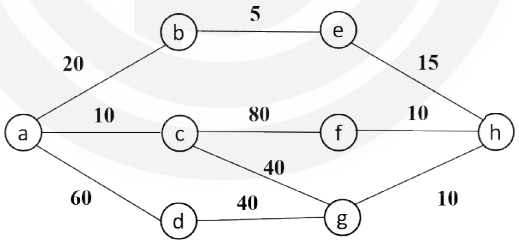


***Figure** 7: Example of greedy approach*

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.

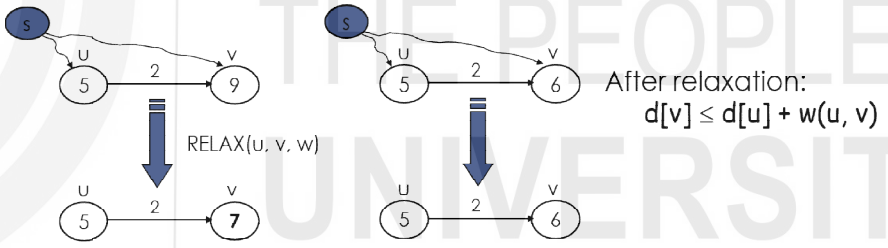
Shortest path $\boldsymbol{a} \rightarrow \boldsymbol{h}$ via $\boldsymbol{a} \rightarrow \boldsymbol{c} \rightarrow \boldsymbol{g} \rightarrow \boldsymbol{h}$and weight is **10+40+10= 50**. In greedy it chooses the optimal solution at that particular time, that's why path $\boldsymbol{a} \rightarrow \boldsymbol{c}$ 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

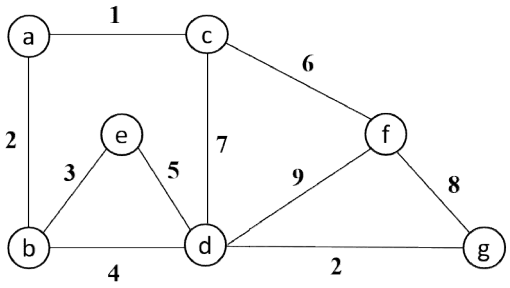
- **dist**$[\boldsymbol{V}]$ is the array that contains ∞ for all the vertices $\boldsymbol{v} \in \boldsymbol{V}$ except the source vertex $\boldsymbol{s}$. Source vertex $\boldsymbol{s}$ have zero distance $\boldsymbol{dist}[\boldsymbol{s}] \leftarrow \boldsymbol{0}$.
- **Priority queue:** stores the 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.

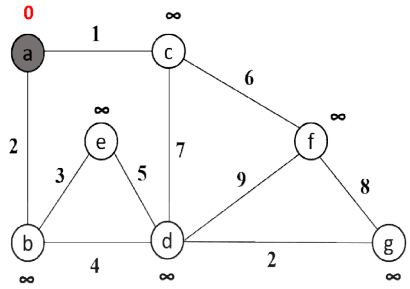


Dijkstra's(G, V, s)

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[]

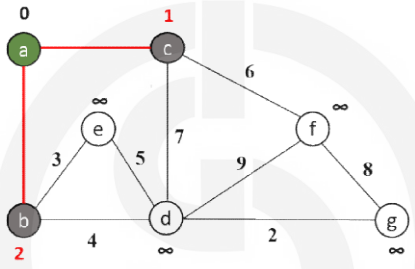
Example 1: Apply Dijakstra's algorithm on the following graph.





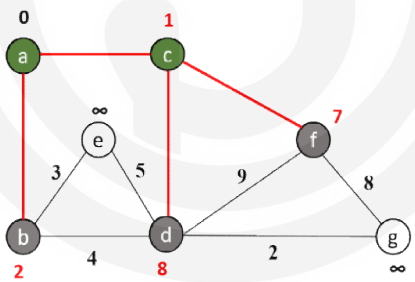
Given a graph $\boldsymbol{G(V, E)}$all the vertices have ∞ distance and only source vertex have **0** distance.
$dist[s] \leftarrow 0$
$dist[V - s] \leftarrow \infty$

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



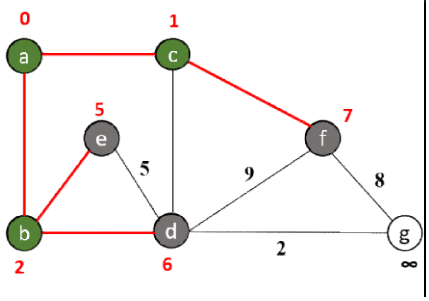
Vertex$\boldsymbol{c}$ and $\boldsymbol{b}$ are reachable from source vertex$\boldsymbol{a}$. Update the distance of $\boldsymbol{c}$ and $\boldsymbol{b}$ from ∞ to 1 and 2 respectively. Red edges shows the relaxed edge and updated weight on the vertex head. Vertex in green color are added in the tree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

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



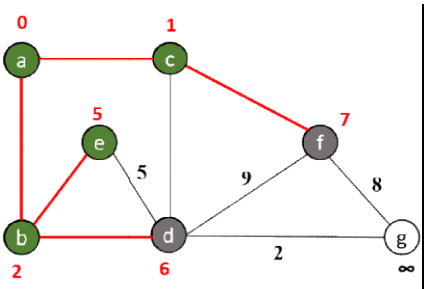
Vertex $\boldsymbol{c}$ having minimum distance is selected and added in the tree **T**. $\boldsymbol{c}$ is also called as a closest vertex to source $\boldsymbol{a}$. Update the distance of all the vertex adjacent to $\boldsymbol{c}$ except the vertex already added in the tree **T**.

$dist[V] =$	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	-



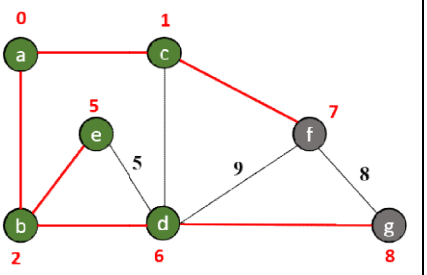
Vertex ***e***having minimum distance among all the remaining vertex is selected and added in the tree **T**.
There is only one vertex ***d*** reachable from ***e*** whose distance is **6** via vertex ***b*** less than the distance **10**via ***e***. Thus distance of***d*** will not be updated.

$dist[V] =$

0	1	2	6	5	7	∞

$prev[V] =$

-	a	a	b	b	c	-



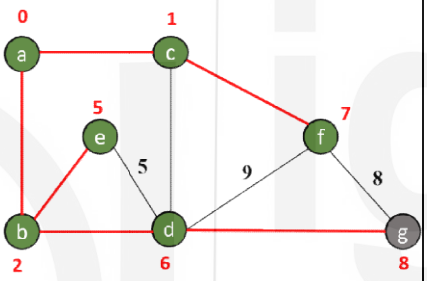
Vertex ***d*** having minimum distance among all the remaining vertex is selected and added in the tree **T**.
Vertex ***f*** and ***g*** are reachable from ***d***. Distance of vertex ***g*** updated from ∞ to **8** and weight of vertex ***f*** will not be updated because it has minimum distance from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

$dist[V] =$

0	1	2	6	5	7	**8**

$prev[V] =$

-	a	a	b	b	c	**d**



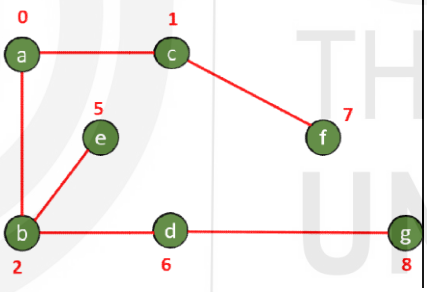
Vertex ***f*** having minimum distance among all the remaining vertex is selected and added in the tree **T**.
Vertex ***g*** is reachable from ***f*.** Distance of vertex ***g*** will not be updated because it has minimum distance from vertex ***d***.

$dist[V] =$

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:

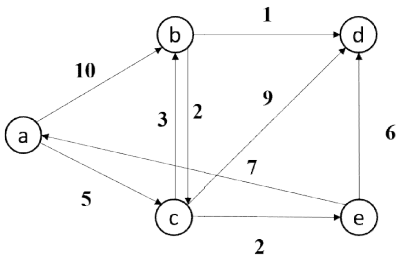
What is the shortest path from vertex ***a*** to vertex ***g***and distance of the path.

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.

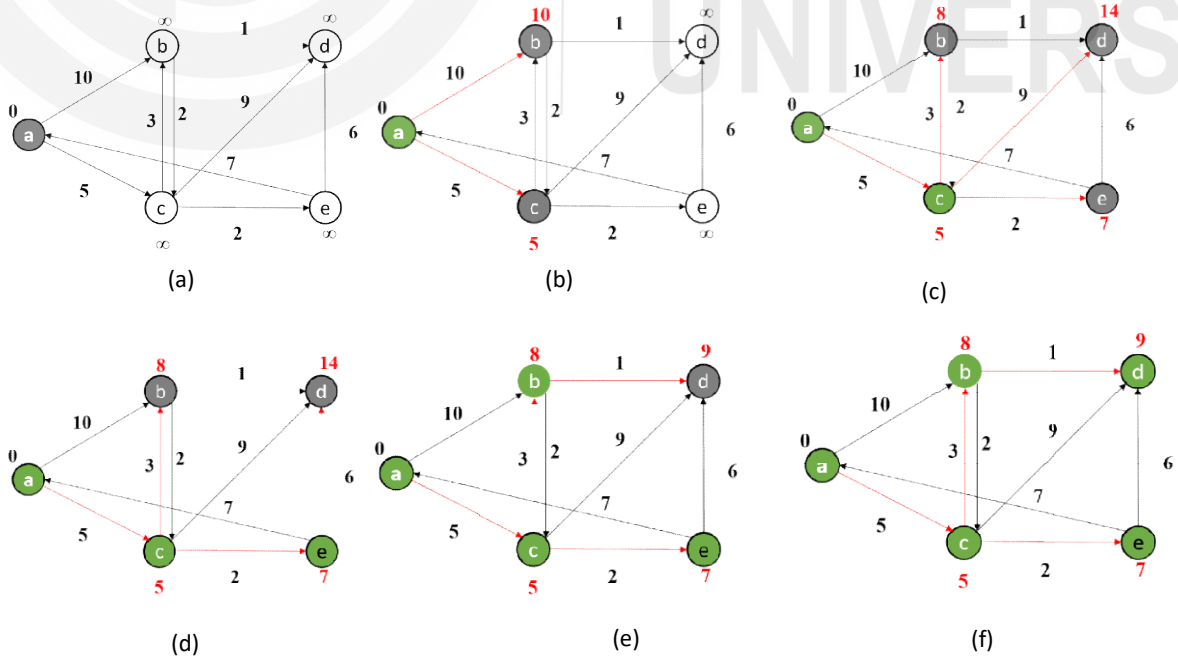


Figure 9*: Shows the step by step execution of Dijakstra's algorithm*

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.

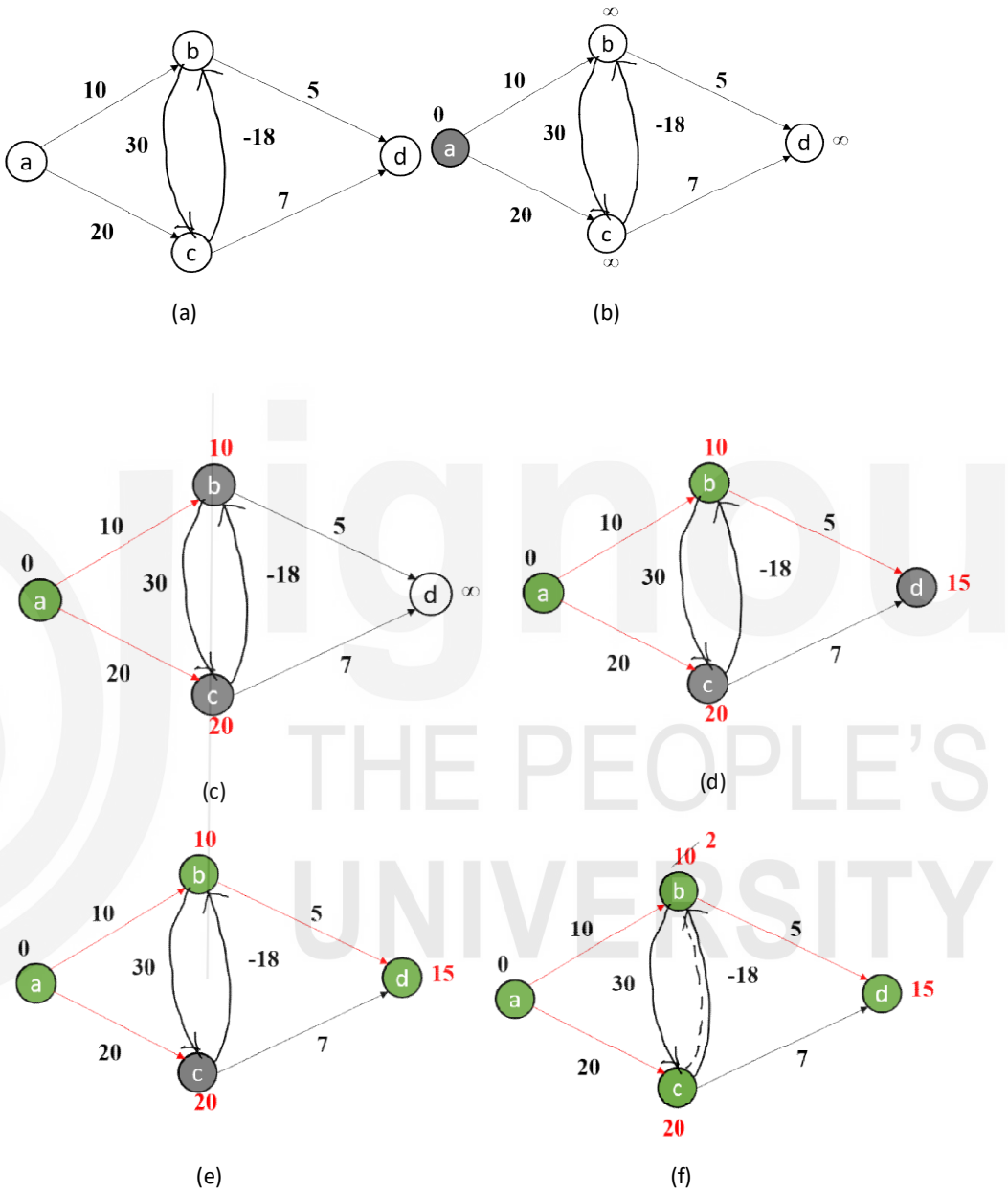


Figure 10: *The execution of dijakstra's algorithm in negative edge weight. (a) The source****a*** *is the leftmost vertex. (b) Assigned* ***0*** *to vertex* ***a*** *and∞ to the remaining vertex. (c)- (e) apply dijakstra's algorithm to update the distance and select the vertex in tree* ***T****. (f) 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:

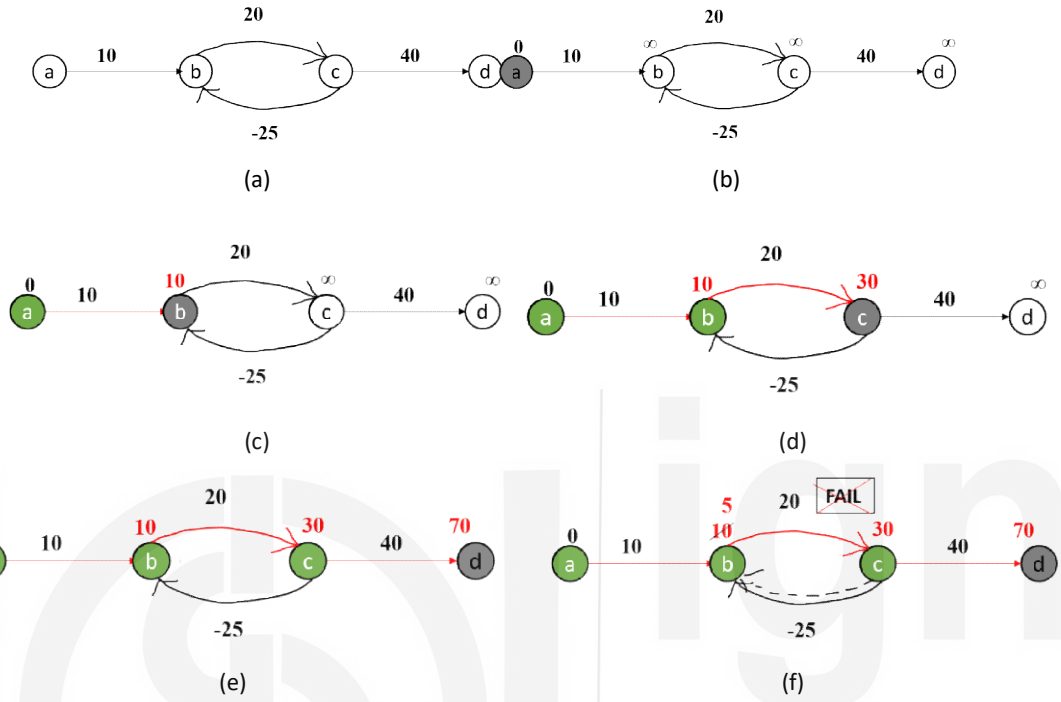


Figure 11: *(a) is a actual graph. (b)* ***0*** *assigned to the source vertex and ∞ assigned to the remaining vertex. (c)- (d) apply dijakstra's algorithm and edges are relaxed and distance are updated. (e)- (f) when distance from vertex* ***c*** *to all the reachable vertex calculated. Here distance 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.

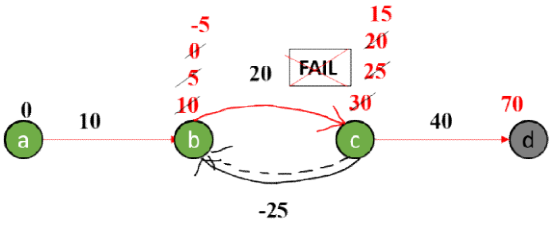


Figure 12: *shows example of negative edge weight cycle and dijakstra's always fail for negative edge weight cycle.*

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

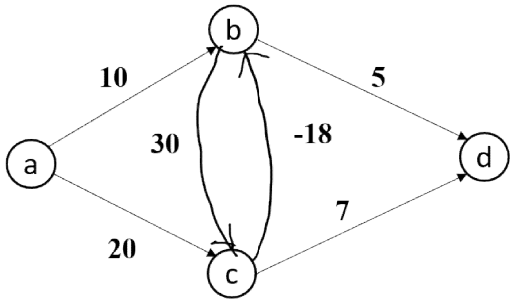
The Bellman-Ford algorithm solves the single-source shortest-paths problem in case where edge weights may be negative or there is a negative edge weight cycle in the graph. This algorithm, like dijakstra's algorithm uses the notation of edge relaxation but does not used greedy method to relax the edges. 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**.

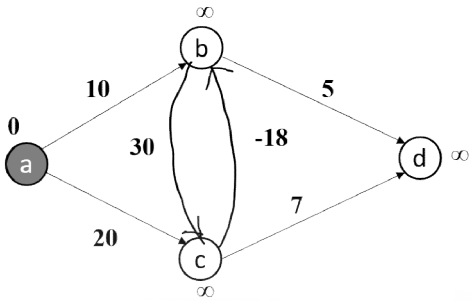
Pseudo-code of Bellman-ford algorithm

- 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.

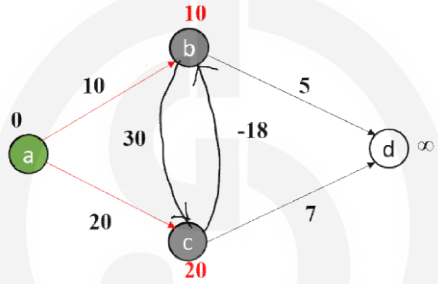
```
BELLMAN − FORD(G, V, E, w, s)
1.  Initialize dist[s]←0
2.  for each vertex v ∈ V-s
3.  dist[V] ← ∞
4.  for each vertex i=1 to V − 1
5.      for each edge (u, v) in E[G]
6.          RELAX(u, v, w)
7.  for each edge edge(u, v) in E[G]
8.      if dist[u] + w(u , v) < dist[v]
9.          return FALSE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

Example1: Apply Bellman-Ford Algorithm on a following graph



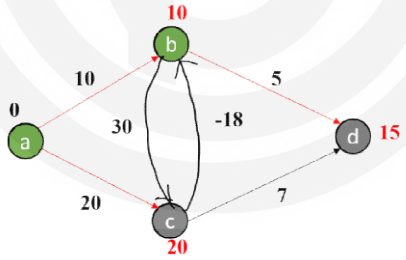


a	b	c	d
0	∞	∞	∞

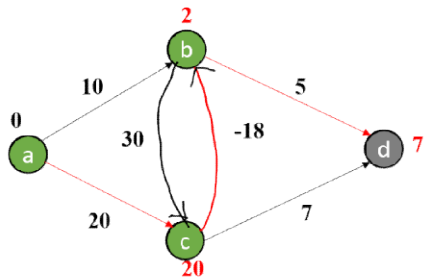


a	b	c	d
0	∞	∞	∞
0	10	∞	∞

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.

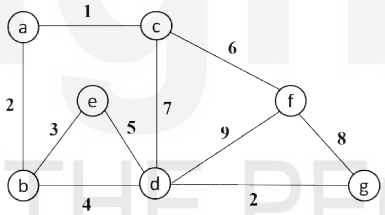
For loop of line 2 takes $O(V)$ time.

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:

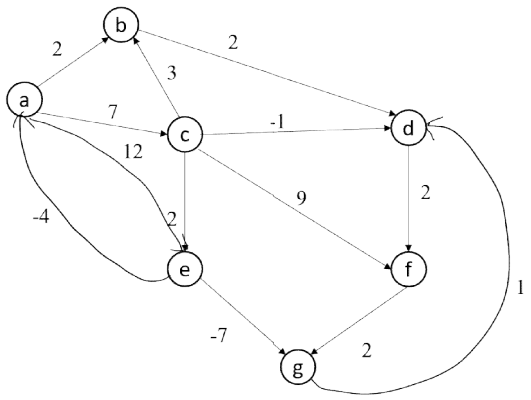
Q1. Apply Dijakstra's algorithm on the following graph with source vertex a.



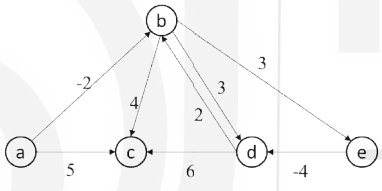
Q2. What are the disadvantages of using Dijakstra's algorithm? List out any three.

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.

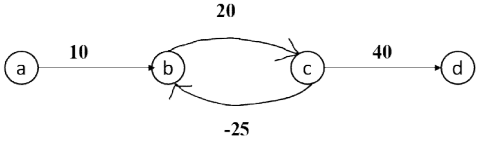


Q5. Apply Bellman-ford on the following graph. List out the order of execution of vertices.



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.

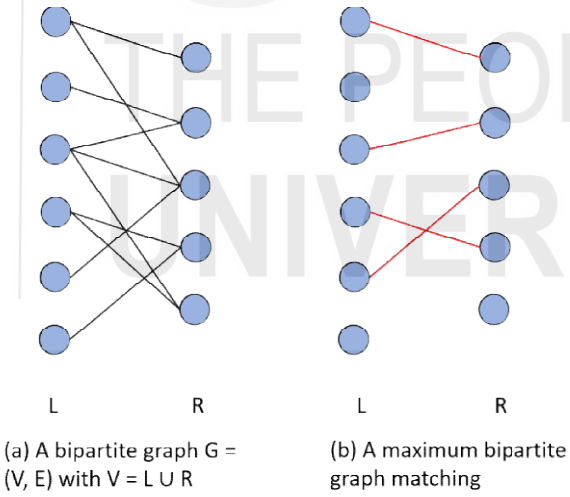


***Figure 13**: Illustration of a bipartite matching problem a maximum matching*

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.

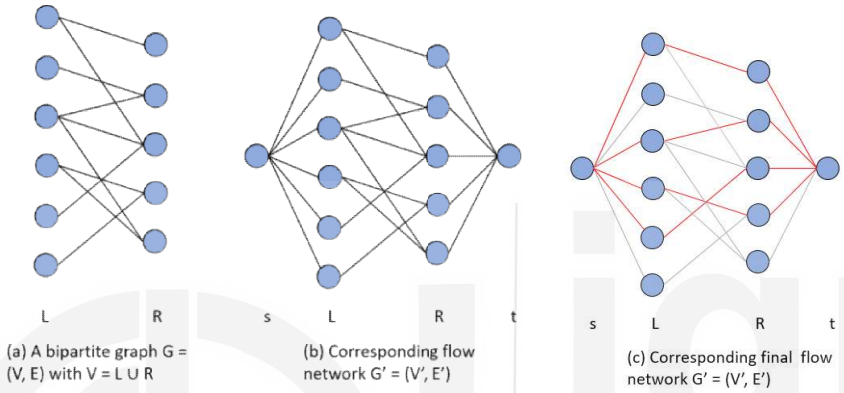


*Figure 14: Flow network corresponding to the bipartite graph. Right most graph shows the solution for a maximum flow problem (the final flow network of (b)). Each edge have unit capacity, and shaded edge from **L** to **R** corresponds to those in the maximum matching from (b).*

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.

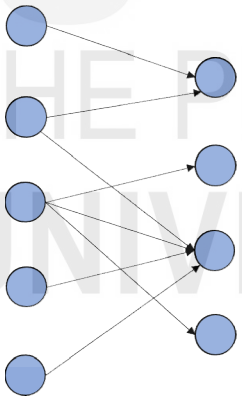
.....

.....

.....

.....

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

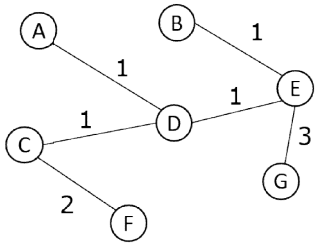
= 2 E log V + 2 E log V

= 4 E log V

= O (E log V)

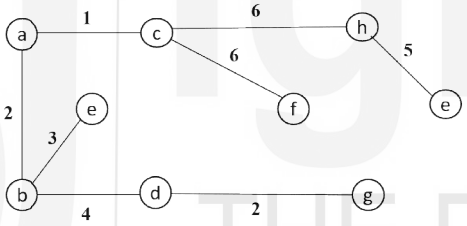
Q5. c)

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:

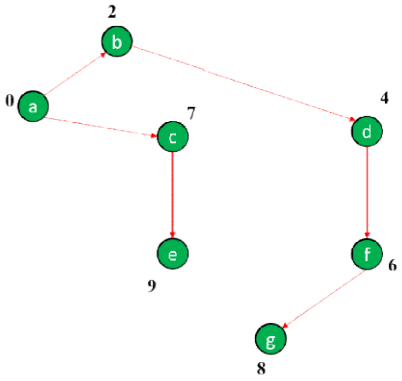
Q2. Dijakstra's algorithms search blindly to find the path then update distance. It waste lot of time while processing.

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

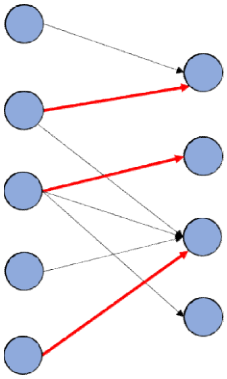
Q8. Bellman-ford algorithm can be used in to route packets of data on network used in distance vector routing protocol.

♣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).

UNIT 2 DYNAMIC PROGRAMMING TECHNIQUE

Structure **Page No**

2.0 INTRODUCTION

Dynamic programming is an optimization technique. Optimization problems are those which required either minimum result or maximum result. Means finding the optimal solution for any given problem. Optimal means either finding the maximum answer or minimum answer. For example if we want to find the profit, will always find out the maximum profit and if we want to find out the cost will always find out the minimum cost. Although both greedy and dynamic are used to solve the optimization problem but there approach to finding the optimal solution is totally different. In greedy method we try to follow defined procedure to get the optimal result. Like Kruskal's for finding minimum cost spanning tree. Always select edge with minimum weight and that gives us best result or Dijkstra's for shortest path, always select the shortest path vertex and continue relaxing the vertices, so you get a shortest path. For any problem there may be a many solutions which are feasible, we try to find out the all feasible solutions and then pick up the best solution. Dynamic programming approach is different and time consuming compare to greedy method. Dynamic programming granted to find the optimal solution of any problem if their solution exist. Mostly dynamic programming problems are solved by using recursive formulas, though we will not use recursion of programming but formulas are recursive. It works on the concept of recursion i.e. dividing a bigger problem into similar type of sub problems and solving them recursively. The only difference is- DP does save the outputs of the smaller sub problem into a table and avoids the task of repetition in calculating the same sub problem. First it searches for the output of the sub problem in the table, if the output is not available then only it calculates the sub problem. This reduces the time complexity to a great extent. Due to its nature, DP is useful in solving the problems which can be divided into similar sub problems. If each sub problem is different in nature, DP does not help in reducing the time complexity.

So how the dynamic programming approach works.

- Breaks down the complex problem into simple sub problem.
- Find optimal solution to these sub problems.
- Store the results of sub problems.
- Re-use that result of sub problems so that same sub problem comes you don't need to calculate again.
- Finally calculates the results of complex problem.
- Storing the results of sub problems is called memorization.

2.1 OBJECTIVES

After going through the unit you will be able to:

- Define the Principle of Optimality
- Define all the stages of Dynamic Programming
- Solve Chained Matrix Multiplication, Optimal Binary Search Tree, All Pair Shortest Path Problem through Dynamic Programming Approach

2.2 PRINCIPAL OF OPTIMALITY

Dynamic programming follows the principal of optimality. If a problem have an optimal structure, then definitely it has principal of optimality. A problem has optimal sub structure if an optimal solution can be constructed efficiently from optimal solution of its sub-problems. Principal of optimality shows that a problem can be solved by taking a sequence of decision to solve the optimization problem. In dynamic programming in every stage we takes a decision. The Principle of Optimality states that components of a globally optimum solution must themselves be optimal.

A problem is said to satisfy the Principle of Optimality if the sub solutions of an optimal solution of the problem are themselves optimal solutions for their sub problems.
Examples:

- The shortest path problem satisfies the Principle of Optimality.
- This is because if $a, x_1, x_2, \dots, x_n$is a shortest path from node a to node b in a graph, then the portion of x_i to x_j on that path is a shortest path from x_i to x_j.
- The longest path problem, on the other hand, does not satisfy the Principle of Optimality. For example the undirected graph of nodes $a, b, c, d,$ and e and edges$(a, b), (b, c), (c, d), (d, e)$and (e, a). That is, G is a ring. The longest (noncyclic) path from a to d to a, b, c, d. The sub-path from b to c on that path is simply the edge b,c. But that is not the longest path from b to c. Rather b, a, e, d, c is the longest path. Thus, the sub path on a longest path is not necessarily a longest path.

Steps of Dynamic Programming:

Dynamic programming has involve four major steps:

1. Develop a mathematical notation that can express any solution and sub solution for the problem at hand.
2. Prove that the Principle of Optimality holds.

3. Develop a recurrence relation that relates a solution to its sub solutions, using the math notation of step 1. Indicate what the initial values are for that recurrence relation, and which term signifies the final solution.
4. Write an algorithm to compute the recurrence relation.

- Steps 1 and 2 need not be in that order. Do what makes sense in each problem.
- Step 3 is the heart of the design process. In high level algorithmic design situations, one can stop at step 3.
- Without the Principle of Optimality, it won't be possible to derive a sensible recurrence relation in step 3.
- When the Principle of Optimality holds, the 4 steps of DP are guaranteed to yield an optimal solution. No proof of optimality is needed.

2.3 MATRIX MULTIPLICATION

- Matrix multiplication a binary operation of multiplying two or more matrices one by one that are conformable for multiplication. For example two matrices A, B having the dimensions of $p \times q$ and $s \times t$ respectively; would be conformable for $A \times B$multiplication only if q==s and for $B \times A$ multiplication only if t==p.
- Matrix multiplication is associative in the sense that if A, B, and C are three matrices of order$m \times n$, $n \times p$ and $p \times q$ then the matrices (AB)C and A(BC) are defined as (AB)C = A (BC) and the product is an $m \times q$ matrix.
- Matrix multiplication is not commutative. For example two matrices A and B having dimensions $m \times n$ and $n \times p$ then the matrix AB = BA can't be defined. Because BA are not conformable for multiplication even if AB are conformable for matrix multiplication.
- For 3 or more matrices, matrix multiplication is associative, yet the number of scalar multiplications may very significantly depending upon how we pair the matrices and their product matrices to get the final product.

Example: Suppose there are three matrices A is 100×10, B is 10×50 and C is 50×5, then number of multiplication required for (AB)C is $AB = 100 \times 10 \times 5 = 50000$, $(AB)C = 100 \times 50 \times 5 = 25000$. Total multiplication for (AB)C= 75000 mulplication.Similarly number of multiplication required for A(BC) is BC = $10 \times 50 \times 5 = 2500$ and A(BC) = $100 \times 10 \times 5 = 5000$. Total multiplication for A(BC) = 7500.

In short the product of matrices (AB)C takes 75000 multiplication when first the product of A and B is computed and then product AB is multiplied with C. On the other hand, if the product BC is calculated first and then product of A with matrix BC is taken then 7500 multiplications are required. In case when large number of matrices are to be multiplied for which the product is defined, proper parenthesizing through pairing of matrices, may cause dramatic saving in number of scalar operations.
This raises the question of how to parenthesize the pairs of matrices within the expression $A_1A_2 \ldots A_n$, a product of n matrices which is defined; so as to optimize the computation of the product $A_1A_2 \ldots A_n$. The product is known as Chained Matrix Multiplication.

2.4 MATRIX CHAIN MULTIPLICATION

Given a sequence of matrices that are conformable for multiplication in that sequence, the problem of matrix-chain-multiplication is the process of selecting the optimal pair of matrices in every step in such a way that the overall cost of multiplication would be minimal. If there are total N matrices in the sequence then the total number of different ways of selecting matrix-pairs for multiplication will be $^{2n}C_n/(n+1)$. We need to find out the optimal one. In directly we can say that total number of different ways to perform the matrix chain multiplication will be equal to the total number of different binary trees that can be generated using N-1 nodes i.e. 2nCn/(n+1). The problem is not actually to perform the multiplications, but merely to decide in which order to perform the multiplications. Since matrix multiplication follows Associative property, rearranging the parentheses also yields the same result of multiplication. That means any pair in the sequence can be multiplied and that will not affect the end result: But the total number of multiplication operations will change accordingly. A product of matrices is fully satisfied if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. For example, if the chain of matrices is $\langle A_1, A_2, A_3, A_4 \rangle$ then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

$(A_1(A_2(A_3A_4)))$,
$(A_1((A_2A_3)A_4)$,
$(A_1(A_2A_3))A_4)$,
$((A_1A_2)(A_3A_4))$,
$(((A_1A_2)A_3)A_4)$.

How we parenthesize a chain of matrices can have a dramatic impact on the cost of evaluating the product. We shall use the dynamic-programming method to determine how to optimally parenthesize a matrix chain.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution.
4. Construct an optimal solution from computed information.

Step 1: The structure of an optimal parenthesization:

- An optimal parenthesization of $A_1 \ldots A_n$ must break the product into two expressions, each of which is parenthesized or is a single array
- Assume the break occurs at position k.
- In the optimal solution, the solution to the product $A_1 \ldots A_k$ must be optimal
  - Otherwise, we could improve $A_1 \ldots A_n$ by improving $A_1 \ldots A_k$.
  - But the solution to $A_1 \ldots A_n$ is known to be optimal
  - This is a contradiction
  - Thus the solution to $A_1 \ldots A_n$ is known to be optimal
- This problem exhibits the Principle of Optimality:
  - The optimal solution to product $A_1 \ldots A_n$ contains the optimal solution to two subproducts
- Thus we can use Dynamic Programming
  - Consider a recursive solution
  - Then improve it's performance with memorization or by rewriting bottom up

Step 2: A recursive solution

For the matrix-chain multiplication problem, we pick as our sub problems the problems of determining the minimum cost of parenthesizing $A_i\, A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$.

- Let $m[i,j]$ be the minimum number of scalar multiplication needed to compute the matrix $A_{i\ldots j}$; for the full problem, the lowest cost way to compute $A_{1\ldots n}$would thus be $m[1,n]$.
- $m[i,i] = 0$, when i=j, problem is trivial. Chain consist of just one matrix $A_{i\ldots i} = A_i$, so that no scalar multiplications are necessary to compute the product.
- The optimal solution of $A_i \times A_j$must break at some point, k, with $\leq k < j$. Each matrix A_i is $p_{i-1} \times p_i$ and computing the matrix multiplication $A_{i\ldots k}\, A_{k+1\ldots j}$ takes $p_{i-1}p_k p_j$ scalar multiplication.

  Thus, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
  Equation 1

3. Computing the optimal costs

It is a simple matter to write a recursive algorithm based on above recurrence to compute the minimum cost $m[1,n]$ for multiplying $A_1 A_2 \ldots A_n$.

MATRIX-CHAIN-ORDER (P)

1. $n \leftarrow length[p] - 1$
2. let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
3. for i←1 to n
4. do $m[i,i] \leftarrow 0$
5. For l← 2 to n
6. do for i←1 to n-1+1
7. do j←i+l-1
8. $m[i,j] \leftarrow \infty$
9. for k←l to j-1
10. do $q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
11. if q<$m[i,j]$
12. then $m[i,j] \leftarrow q$
13. $s[i,j] \leftarrow k$
14. return m and s

The following pseudo code assumes that matrix $\boldsymbol{A_i}$ has dimensions $\boldsymbol{p_{i-1} \times p_i}$ for **i=1,2,…,n**. The input is a sequence$\boldsymbol{p = p_0, p_1, \ldots, p_n}$, where **length[p]=n+1**. The procedure uses an auxiliary table $\boldsymbol{m[1 \ldots n, 1 \ldots n]}$ for storing the $\boldsymbol{m[i,j]}$ costs and the auxiliary table $\boldsymbol{s[1 \ldots n, 1 \ldots n]}$ that records which index of k achieved the optimal cost in computing $m[i,j]$. We will use the table **s** to construct an optimal solution.

- In line 3-4 algorithm first computes $m[i,i] = 0$ for $i = 1, 2, \ldots, n$ (the minimum costs for chains of length 1).
- During the first execution of for loop in line 5-13, it uses Equation (1) to computes$m[i, i+1]$ for $i = 1, 2, \ldots, n-1$ (the minimum costs for chains of length 2).
- The second time through the loop, it computes $m[i, i+2]$ for $i = 1, 2, \ldots, n-2$ (the minimum costs for chains of length 3, and so forth.
- At each step, the $m[i,j]$ cost computed in lines 10-13 depends only on table entries $m[i,k]$ and $m[k+1,j]$ already computed.

4. Constructing an Optimal Solution:

The MATRIX-CHAIN-ORDER determines the optimal number of scalar multiplication needed to compute a matrix-chain product. To obtain the optimal solution of the matrix multiplication, we call **PRINT_OPTIMAL_PARES(s,1,n)** to print an optimal parenthesization of $\boldsymbol{A_1A_2, \ldots, A_n}$**.** Each entry $s[i, j]$ records a value of **k** such that an optimal parenthesization of $\boldsymbol{A_1A_2, \ldots, A_j}$splits the product between A_k and A_{k+1}.

```
PRINT_OPTIMAL_PARENS(s,i,j)
1. If i==j
2. Then print "A"i
3. else print "("
4. PRINT_OPTIMAL_PARENS(s, i, s[i,j])
5. PRINT_OPTIMAL_PARENS(s, s[i,j]+1, j)
6. print ")"
```

Example: Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is as follows:

Matrix	Dimension
A1	30 × 35
A2	35 × 15
A3	15 × 5
A4	5 × 10
A5	10 × 20

Solution:

Table m

j→ 1	2	3	4	5	i↓
0	15750	7875	9375	11875	1
	0	2625	4375	7125	2
		0	750	2500	3
			0	1000	4
				0	5

Table s

j→2	3	4	5	i↓
1	1	3	3	1
	2	3	3	2
		3	3	3
			4	4

Step 1: l=2

m[1 2]= m[1 1]+m[2 2] + $p_0p_1p_2$ = 0 + 0 + 30 ×35 × 15 = 15750 and k = 1 s[1 2]=1

m[2 3]= m[2 2] + m[3 3] + $p_1p_2p_3$ = 0 + 0 + 35 ×15 × 5 = 2625 and k=2 s[2 3]= 2

m[3 4]= m[3 3] + m[4 4] + $p_1p_3p_4$ = 0 + 0 + 15 ×5 × 10 = 750 and k=3 s[3 4]= 3

m[4 5]= m[4 4] + m[5 5]+ $p_3p_4p_5$ = 0 + 0 + 5 ×10 × 20 = 1000 and k =4 s[4 5] = 4

Step 2: l=3

m[1 3] = min (m[1 1] + m[2 3] + $p_0p_1p_3$, m[1 2] + m[3 3] + $p_0p_2p_3$)

= min (0 + 2625+ 30 ×35 × 5 , 15750+0+ 30 ×15 × 5)

= min(7875, 18000) = 7875 and k = 1 gives minimum s[1 3]= 1

m[2 4] = min (m[2 2] + m[3 4] + $p_1p_2p_4$, m[2 3] + m[4 4] + $p_1p_3p_4$)

= min (0 + 750+ 3 ×15 × 10 , 2625+0+ 35 ×5 × 10)

= min(6000, 4375) = 4375 and k = 3 gives minimum s[2 4]= 3

m[3 5] = min (m[3 3] + m[4 5] + $p_2p_3p_5$, m[3 4] + m[5 5] + $p_2p_4p_5$)

= min (0 + 1000+ 15 ×5 × 20 , 750+0+ 15 ×10 × 20)

= min(2500, 3750) = 2500 and k = 3 gives minimum s[3 5]= 3

Step 3: l=4

m[1 4] = min (m[1 1] + m[2 4] + $p_0p_1p_4$, m[1 2] + m[3 4] + $p_0p_2p_4$, $m[1\ 3] + m[4\ 4] + p_0p_3p_4$)

= min(0 + 4375 + 30 ×35 × 10, 15750 + 750 + 30 ×15 × 10, 7875+0+ 30 ×5 × 10)

= min(14875, 21900, 9375) = 9375 and k = 3 gives minimum s[1 4] = 3

m[1 4] = min (m[1 1] + m[2 4] + $p_0p_1p_4$, m[1 2] + m[3 4] + $p_0p_2p_4$, $m[1\ 3] + m[4\ 4] + p_0p_3p_4$)

= min(0 + 4375 + 30 ×35 × 10, 15750 + 750 + 30 ×15 × 10, 7875+0+ 30 ×5 × 10)

= min(14875, 21900, 9375) = 9375 and k = 3 gives minimum s[1 4] = 3

m[2 5] = min (m[2 2] + m[3 5] + $p_1p_2p_5$, m[2 3] + m[4 5] + $p_1p_3p_5$, $m[2\ 4] + m[5\ 5] + p_1p_4p_5$)

= min(0 + 2500 + 35 ×15 × 20, 2625 + 1000 + 35 ×5 × 20, 4375+0 + 35 ×10 × 20)

= min (13000, 7125, 11375) = 7125 and k = 3 gives minimum s[2 5] = 3

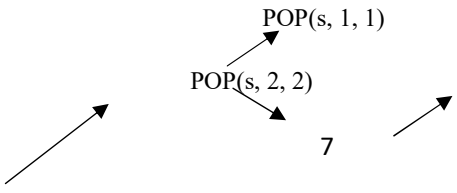
Step 4: l=5

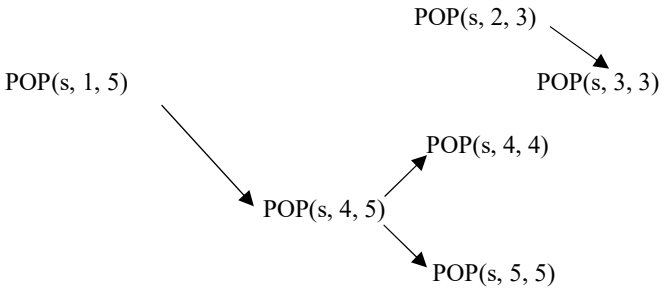
m[1 5] = min (m[1 1] + m[2 5] + $p_0p_1p_5$, m[1 2] + m[3 5] + $p_0p_2p_5$, $m[1\ 3] + m[4\ 5] + p_0p_3p_5$, m[1 4] + m[5 5] + $p_0p_4p_5$)

= min(0 + 7125 + 30 ×35 × 20, 15750+2500 + 30 ×15 × 20, 7875+ 1000 + 30 ×5 × 20, 9375+ 0 +30 ×10 × 20)

= min (28125, 27250, 11875, 15375) = 11875 and k = 3 gives minimum s[1 5] = 3

Now, print optimal parenthesis (POP) algorithm is runs:

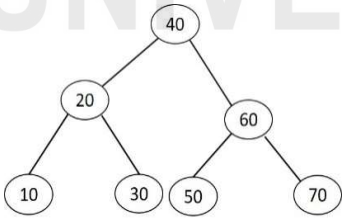




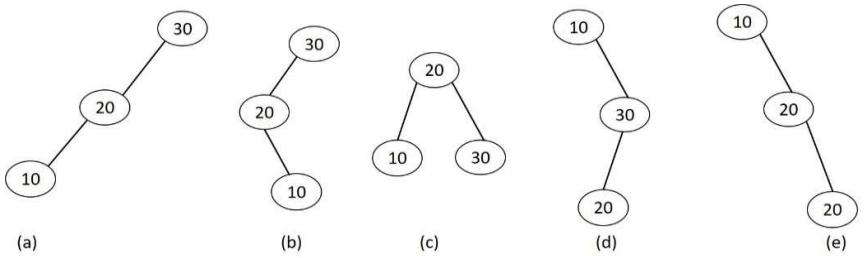
Hence, the final solution is $(A_1(A_2A_3)(A_4A_5))$.

2.5 OPTIMAL BINARY SEARCH TREE

Binary Search Tree: A binary tree is binary search tree such that all the key smaller than root are on left side and all the keys greater than root or on right side. Figure shown below is a binary search tree in which 40 is root node all the node in the left sub tree of 40 is smaller and all the node in the right sub tree of 40 is greater. No questions is why keys are arrange in order. Because it gives and advantage of searching. Suppose I want to search 30, we will start searching from the root node. Check whether root is a 30, no, then 30 is smaller than 40, definitely 30 will be in left sub tree of 40. Go in the left sub tree, check left sub tree root is 30, no, 20 is smaller than 30, definitely 30 will be in the right hand side. Yes now 30 is found. So to search the 30 how many comparison are need. Out of 7 elements to search 30 it, takes 3 comparison. No of comparison required to search any element in a binary search tree is depend upon the number of levels in a binary search tree. We are searching for the key that are already present in the binary search tree, is successful search. Now let's assume that key is 9 and search is an unsuccessful search because 9 is not found in the binary search tree. Total number of comparison it takes is also 3. There are a two possibility to search any element successful search and unsuccessful search. In both the cases we need to know the amount of comparison we are doing. That amount of comparison is nothing but the cost of searching in a binary search tree.



Let's take an example: Keys: 10, 20, 30 How many binary search tree possible. T(n)= $\frac{2n_{C_n}}{n+1}$ So there are 3 keys number of binary search tree possible = 5. Let's draw five possible binary search tree shown in below figure (a)-(e).



Now the cost of searching an element in the above binary tree is the maximum number of comparison. In tree (a), (b), (d), and (e) takes maximum three comparison whereas in tree (c) it takes only two comparison. It means that if you have set of n keys, there is a possibility that any tree gives you less number of comparison compare to other trees in all possible binary search tree of n keys. It means that if a height of binary search tree is less, number of comparison will be less. Height play an important role in searching any key in a binary search tree. So we want a binary search tree that requires less number of comparison for searching any key elements. This is called an optimal binary search tree.

The problem of optimal binary search tree is, given a keys and their probabilities, we have to generate a binary search tree such that the searching cost should be minimum. Formally we are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order, and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i. Some searches may be for the values that is not in K, so we have also n+1 dummy keys $d_0, d_1, d_3, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, 3, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search corresponds to d_i. Below Figure 2 shows the binary search tree for set of 5 keys. Each k_i is a internal node and each d_i is leaf node.

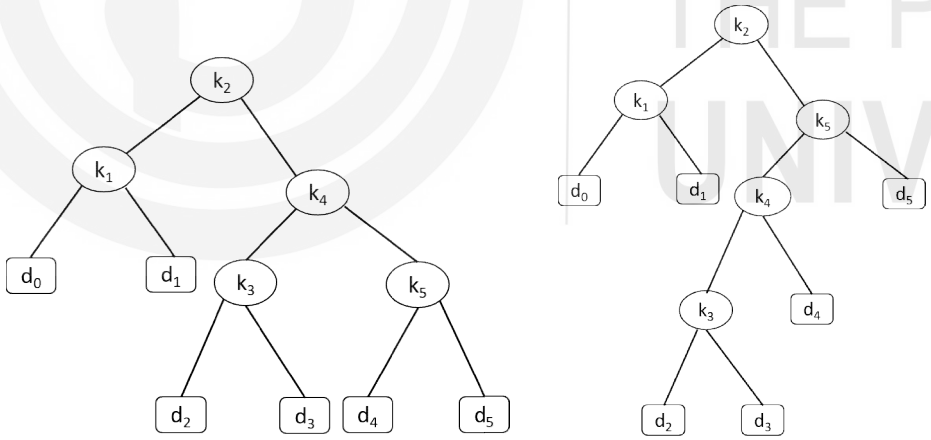


Figure 2: (a) (b)
Two Binary search tree with dummy keys with the following probabilities:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search cost 2.75. This tree is optimal

Every search is either successful or unsuccessful, and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

Because we have probabilities of searches for each key and each key and each dummy key, we can determine the expected cost of a search is a given binary search tree T. Let we assume that actual cost of search equals the number of nodes examined, i.e., depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is

$$E(search\ cost\ in\ T) = \sum_{i=1}^{n} (depth_T(k_i) + 1).p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1).q_i$$

$= 1 + \sum_{i=1}^{n} depth_T(k_i).p_i + \sum_{i=0}^{n} depth_T(d_i).q_i,$

Where $depth_T$ denotes a node depth in the tree T. Expected search cost node by node given in following table:

node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.20	0.80
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such tree an optimal binary search tree. Figure 2(b) shows an optimal binary search tree for the probabilities given in figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Now question is we have n number of keys finding minimum cost means draw all possible trees and picking the tree having minimum cost is an optimal binary search tree. But this approach is not an efficient approach. Using dynamic programming we can find the optimal binary search tree without drawing each tree and calculating the cost of each tree. Thus, dynamic programming gives better, easiest and fastest method for trying out all possible binary search tree and picking up best one without drawing each sub tree.

Dynamic Programming Approach:

- Optimal Substructure: if an optimal binary search tree T has a subtree T` containing keys $k_i, \ldots, k_j$, then this sub tree T' must be optimal as well for the sub problem with keys $k_i, \ldots, k_j$and dummy keys $d_{i-1}, \ldots, d_j$.
- Algorithm for finding optimal tree for sorted, distinct keys $k_i, \ldots, k_j$:
    - For each possible root k_r for $i \leq r \leq j$

- Make optimal subtree for $k_i, \ldots, k_{r-1}$.
- Make optimal subtree for $k_{r+1}, \ldots, k_j$.
- Select root that gives best total tree

- Recursive solution: We pick our sub problem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1, j \leq n$, and $j \geq i - 1$. Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1, n]$.

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j=i-1 \\ \min_{i \leq r \leq j}\{e[i, r-1] + e[r+1, j] + w(i,j)\} & \text{if } i \leq j \end{cases} \quad \text{Equation 2}$$

where, $w(i,j) = \sum_{k=i}^{j} p_i$ is the increase in cost if $k_i, \ldots, k_j$ is a subtree of a node. When j=i-1, there are no actual keys; we have just the dummy key d_{i-1}.

Computing the Optimal Cost:

We store the $e[i, j]$ values in a table $e[1..n+1, 0..n]$. The first index needs to run to n+1 rather than n because in order to have a sub tree containing only the dummy key d_n, we need to compute and store $e[n+1, n]$. Second index needs to start from 0 because in order to have a sub tree containing only the dummy key d_0, we need to compute and store $e[1,0]$. We use only the entries $e[i, j]$ for which $j \geq i - 1$. We also use a table $root[i, j]$, for recording the root of the sub tree containing keys $k_i, \ldots, k_j$. This table uses only entries for which $1 \leq i \leq j \leq n$. We will need one other table for efficiency. Rather than compute the value of $w(i, j)$ from scratch every time we are computing $e[i, j]$ which would take $\theta(j - 1)$ additions- we store these values in a table $w[1..n+1, 0..n]$. For the base case, we compute $w[i, i-1] = q_{i-1}$ for $1 \leq i \leq n + 1$. For $j \geq i$, we compute

$$w[i,j] = w[i,j-1] + p_j + q_j \quad \text{Equation 3}$$

The pseudo code that follows takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and returns the table e and root.

```
OPTIMAL_BST(p,q,n)
1.  let e[1..n+1,0..n], w[1..n+1,0..n] and root[1..n,1..n] be new tables.
2.  for i=1 to n+1
3.  e[i, i − 1] = q_{i−1}
4.  w[i, i − 1] = q_{i−1}
5.  for l=1 to n
6.      for i=1 to n-l+1
7.          J=i+l-1
8.  e[i, j] = ∞
9.  w[i, j] = w[i, j − 1] + p_j + q_j
10.         for r=i to j
11.             t= e[i,r-1]+e[r+1,j]+w[i, j]
12.             if t < e[i, j]
13. e[i, j] = t
14. root[i, j] = r
15. return e and root.
```

In the above algorithm:

- The for loop of lines 2-4 initializes the values of $e[i, i\text{-}1]$ and $w[i, i-1]$.
- The for loop of lines 5-14 then uses the recurrence in equation 2 to compute $e[i, j]$ and $w[i, j]$ for all $1 \leq i \leq j \leq n$.
- In the first iteration, when l=1, the loop computes $e[i, i]$ and $w[i, i]$ for $i = 1,2, \ldots, n$. The second iteration, when l=2, the loop computes $e[i, i+1]$ and $w[i, i+1]$ for $i = 1,2, \ldots, n-1$ and so forth.
- The innermost for loop, in lines 10-14, tries each candidates index r to determine which key $k_r$to use as the root of an optimal binary search tree containing keys $k_i, \ldots k_j$. This for loops saves the current value of the index r in $root[i, j]$whenever it finds a better key to use as the root.

Example of running the Algorithm:

Find the optimal binary search tree for N=4, having successful search and unsuccessful given in p_i and q_i rows.

Keys		10	20	30	40
$\mathbf{p_i}$		3	3	1	1
$\mathbf{q_i}$	2	3	1	1	1

Row 1

Let's fill up the values of **w**.

w[0 0]=q_0 w[1 1]=q_1 w[2 2]=q_2 w[3 3]=q_3 w[4 4]=q_4

Let's fill the cost **e**

Initial cost $\mathbf{e_{00}, e_{11}, e_{22}, e_{33}, e_{44}}$ will be 0. Similarly root **r** will be 0.

Row 2

Using equation 3**w** can be filled as:

$w[0\ 1]= w[0\ 0] + p_1 + q_1 \quad = \quad 2+3+3 = 8$

$w[1\ 2]= w[1\ 1] + p_2 + q_2 \quad = \quad 3+ 3 +1 = 7$

$w[2\ 3]= w[2\ 2] + p_3 + q_3 \quad =1+ \ 1 + 1=3$

$w[3\ 4]= w[3\ 3] + p_4 + q_4 \quad = \quad 1 + 1 +1 =3$

$e[0\ 1] = \min(e[0\ 0] + e[\ 1\ 1]) + w[0\ 1] = \min(0 + 0) + 8 = 8$

$e[1\ 2] = \min(e[1\ 1] + e[\ 2\ 2]) + w[1\ 2] = \min(\ 0 + 0) + 7 = 7$

$e[2\ 3] = \min(e[2\ 2] + e[\ 3\ 3]) + w[2\ 3] = \min(\ 0 + 0) + 3 = 3$

$e[3\ 4] = \min(e[3\ 3] + e[4\ \ 4]) + w[3\ 4] = \min(\ 0 + 0) + 3 = 3$

$r[0\ 1] = 1, \quad r[1\ 2] = 2, \quad r[2\ 3]= 3, \quad r[\ 3\ 4]=4$

j	**0**	**1**	**2**	**3**	**4**
j-i=0	$w_{00}= 2$ $e_{00}=0$ $r_{00}=0$	$w_{11}= 3$ $e_{11}=0$ $r_{11}=0$	$w_{22}= 1$ $e_{22}=0$ $r_{22}=0$	$w_{33}= 1$ $e_{33}=0$ $r_{33}=0$	$w_{44}= 1$ $e_{44}=0$ $r_{44}=0$
j-i=1	$w_{01}=8$ $e_{01}= 8$ $r_{01}= 1$	$w_{12}= 7$ $e_{12}= 3$ $r_{12}=2$	$w_{23}= 3$ $e_{23}= 3$ $r_{23}= 3$	$w_{24}= 3$ $e_{34}= 3$ $r_{34}= 4$	
j-i=2	$w_{02}= 12$ $e_{02}= 19$ $r_{02}=1$	$w_{13}= 9$ $e_{13}=12$ $r_{13}=2$	$w_{24}= 5$ $e_{24}=8$ $r_{24}=3$		
j-i=3	$w_{03}= 14$ $e_{03}=25$ $r_{03}=2$	$w_{14}= 11$ $e_{14}= 19$ $r_{14}=2$			
j-i=4	$w_{04}= 16$ $e_{04}=32$ $r_{04}=2$				

Table 1 shows the calculation of cost of the optimal binary search tree

Row 3

$w[0\ 2]= w[0\ 1] + p_2 + q_2 = 8 + 3 +1 = 12$

$w[1\ 3]= w[1\ 2] + p_3 + q_3 = 7 + 1 +1 =9$

w[2 4]= w[2 3] + p_4 + q_4 = 3 + 1 +1 =5

e[0 2] = k= 1, 2 = min(e[0 0] + e[1 2], e[0 1]+ e[2 2])+w[0 2] = min(0+7,8+0)+12 = 7+12=19 here k=1 has given minimum value thus, r[0 2]= 1

e[1 3]= k = 2, 3 = min(e[1 1]+ e[2 3], e[1 2]+e[3 3]) + w[1 3] = min(0+3, 7+0)+9 = 3+9= 12, here k=2 has given minimum value thus, r[1 3]=2

e[2 4]= k = 3, 4 = min(e[2 2]+e[3 4], e[2 3]+e[3 4])+w[2 4]= min(0+3, 3+3)+5 = 8, here k=3 has given minimum value thus, r[2 4]=3

Row 4

w[0 3]= w[0 2] + p_3 + q_3 = 12 + 1 + 1 = 14

w[1 4]= w[1 3] + p_4 + q_4 = 9 + 1 + 1 = 11

e[0 3] = k= 1, 2, 3 = min(e[0 0] + e[1 3], e[0 1]+ e[2 3], e[0 2]+ e[3 3])+w[0 3] = min(0+12,8+3, 19+0)+14 = 11+14=25 here k=2 has given minimum value thus, r[0 3]= 2

e[1 4]= k = 2, 3, 4 = min(e[1 1]+e[2 4], e[1 2]+e[3 4], e[1 3]+e[4 4])+w[1 4]= min(0+8, 7+3, 12+0)+11 = 19, here k=2 has given minimum value thus, r[1 4]= 2

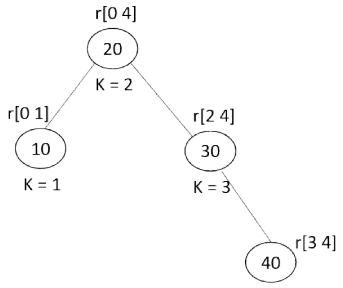
Row 5

w[0 4]= w[0 3] + p_4 + q_4 = 14 + 1 + 1 = 16

e[0 4]= k = 1, 2, 3, 4= min(e[0 0]+e[1 4], e[0 1]+ e[2 4], e[0 2]+e[3 4], e[0 3]+e[4 4] + w[0 4]=

min(0+19, 8+8, 19+3, 25+0]+16= 16+16=32 here k =2 has given minimum value thus, r[0 4]=2

Table 1 shows the calculation of cost matrix, weight matrix and root. While calculating the matrix, we have tried all possible binary search tree and select the minimum cost each time, thus it gives the minimum cost of the optimal binary search tree. Now we can draw the optimal binary search tree from the table 1. Below is a optimal binary search tree whose cost is minimum that is 32/16 = 2.



2.6 BINOMIAL COEFFICIENT COMPUTATION

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming. **Binomial Coefficient** is the coefficient in the Binomial Theorem which

is an arithmetic expansion. It is denoted as c(n, k) which is equal to $\frac{n!}{k! \times (n-k)!}$where ! denotes factorial.This follows a recursive relation using which we will calculate the n binomial coefficient in linear time $O(n \times k)$using Dynamic Programming.

What is Binomial Theorem?

Binomial is also called as Binomial Expansion describe the powers in algebraic equations. Binomial Theorem helps us to find the expanded polynomial without multiplying the bunch of binomials at a time. The expanded polynomial will always contain one more than the power you are expanding.
Following formula shows the General formula to expand the algebraic equations by using Binomial Theorem:

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

Where, n= positive integer power of algebraic equation and$\binom{n}{k}$= read as "n choose k".
According to theorem, expansion goes as following for any of the algebraic equation containing any positive power,

$$(a+b)^n = \binom{n}{0} a^n b^0 + \binom{n}{1} a^{n-1} b^1 + \binom{n}{2} a^{n-2} b^2 + \ldots + \binom{n}{n-1} a^1 b^{n-1} + \binom{n}{n} a^n b^n$$

Where, $\binom{n}{k}$ are binomial coefficients and$\binom{n}{k} = \boldsymbol{n_{c_k}}$gives combinations to choose k elements from n-element set. The expansion of binomial coefficient can be calculated as:$\frac{n!}{k! \times (n-k)!}$.

We can compute $\boldsymbol{n_{c_k}}$ for any n and k using the recurrence relation as follows:

$$\boldsymbol{n}_{C_k} = \begin{cases} 1, & \textit{if } k=0 \textit{ or } n=k \\ n-1 C_{k-1} + n-1 C_k, & \textit{for } n>k>0 \end{cases}$$

Computing C(n, k): Pseudo code:

BINOMIAL(n, k)

Input: A pair of nonnegative integers $n \geq k \geq 0$

Output: The value of $c(n, k)$

```
1.  for i←0 to n  do
2.    for j← 0 to min(i, k) do
3.      if  j=0  or  j=i
4.          c[i, j]←1
5.      else
6.          c[i, j]← c[i-1, j-1] + c[i-1, j]
7.  return c[n, k]
```

Let's consider an example for calculating binomial coefficient:

Compute binomial coefficient for n=5 and k=5

n/k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Table 3: gives the binomial coefficient

According to the formula when k=0 corresponding value in the table will be 1. All the values in column1 will be 1 when k=0. Similarly when n=k, value in the diagonal index will be 1.

c[2, 1] = c[1, 0]+c[1,1] = 1+1= 2 c[3, 1]= c[2,0] + c[2,1] = 1+2 = 3

c[4, 1] = c[3,0]+ c[3,1]= 1+3 = 4c[5, 1] = c[4, 0] + c[4, 1] = 1 + 4 = 5

c[3, 2] = c[2,1] + c[2, 2] = 2+1=3 c[4, 2] = c[3, 1] + c[3, 2] = 3 + 3 = 6

c[5, 2] = c[4, 1]+ c[4, 2] = 4 + 6 =10 c[4, 3] = c[3,2] + c[3, 3] = 3 + 1= 4

c[5, 3] = c[4, 2] + c[4, 3] = 6 + 4 = 10 c[5, 4] = c[4, 3] + c[4, 4]= 4 + 1= 5

Table 3 represent the binomial coefficient of each term.

Time Complexity:

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row. Time complexity of the binomial coefficient is the size of the table i.e, O(n*k)

2.7 ALL PAIRSHORTEST PATH

Till now we have seen single source shortest path algorithms to find the shortest path from a given source S to all other vertices of graph **G**. But what if we want to know the shortest distance among all pair of vertices? Suppose we require to design a railway network where any two stations are connected with an optimum route. The simple answer to this problem would be to apply single source shortest path algorithms i.e. Dijkstra's or Bellman Ford algorithm for each vertex of the graph. The other approach may be to apply all pair shortest path algorithm which gives the shortest path between any two vertices of a graph at one go. We would see which approach will give the best result for different types of graph later on. But first we will discuss one such All Pair Shortest Path Algorithm i.e. Floyd Warshall Algorithm.

2.7.1 Floyd Warshall Algorithm

Floyd Warshall Algorithm uses the Dynamic Programming (DP) methodology. Unlike Greedy algorithms which always looks for local optimization, DP strives for global optimization that means DP does not relies on immediate best result. It works on the concept of recursion i.e. dividing a bigger problem into similar type of sub problems and solving them recursively. The only difference is- DP does save the outputs of the smaller subproblem and avoids the task of repetition in calculating the same subproblem. First it searches for the output of the sub problem in the table, if the output is not available then only it calculates the sub problem. This reduces the time complexity to a great extent. Due to its nature, DP is useful in solving the problems which can be divided into similar subproblems If each subproblem is different in nature, DP does not help in reducing the time complexity.

Now when we have the basic understanding of how the Dynamic Programming works and we have found out that the main crux of using DP in any problem is to identify the recursive function which breaks the problem into smaller subproblems. Let us see how can we create the recursive function for Floyd Warshall Algorithm (thereafter refers as FWA) which manages to find all pair shortest path in a graph.

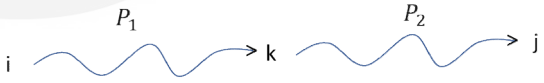
Note- Floyd Warshall Algorithm works on directed weighted graph including negative edge weight. Although it does not work on graph having negative edge weight cycle.

FWA uses the concept of intermediate vertex in the shortest path. Intermediate vertex in any simple path $\mathbf{P} = \{v_1, v_2 \ldots v_l\}$ is any vertex in **P** except source vertex v_1 and destination vertex v_l. Let us take a graph **G (V, E)** having vertex set **V= {1,2,3.....n}**. Take a subset of **V** as V_{s1}**={1,2...k}** for some value of **k**. For any pair of vertices $i, j \in V$ lists all paths from **i** to **j** such that any intermediate vertex in those paths belongs to V_{s1}. Find the shortest path P_s among them. Now there is one possibility between the two cases listed below-

1. The vertex **k** does not belong to the shortest path P_s.
2. The vertex **k** belongs to the shortest path P_s.

Case 1- P_s will also be the shortest path from **i** to **j** for another subset of **V** i.e. V_{s2}**={1,2...k-1}**

Case 2- We can divide path P_s into two paths P_1 and P_2 as below-



Where P_1 is the shortest path between vertices **i**and **k** and P_2 is the shortest path between vertices **k** and **j**. P_1 and P_2 both have the intermediate vertices from the set V_{s2}**= {1,2…k-1}**.

Based on the above understanding the recursive function for FWA can be derived as below-

$$d_{i,j}^{(k)} = \begin{cases} w(i,j) & \text{if } k = 0 \\ \min \{d_{i,j}^{(k-1)},\ d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & \text{if } k > 0 \end{cases}$$

Where,

$d_{i,j}^{(k)}$ - Weight of the shortest path from **i** to **j** for which all intermediate vertices ∈ **{1, 2...k}**

w(i,j) - Edge weight from vertex **i**to **j**

The recursive function implies that if there is no intermediate vertex from **i** to **j** then the shortest path shall be the weight of the direct edge. Whereas, if there are multiple intermediate vertices involved, the shortest path shall be the minimum of the two paths, one including a vertex k and another without including the vertex **k**.

2.7.2 Working Strategy for FWA

- Initial Distance matrix $\boldsymbol{D^0}$of order $\boldsymbol{n \times n}$ consisting direct edge weight is taken as the base distance matrix.
- Another distance matrix $\boldsymbol{D^1}$of shortest path $\boldsymbol{d_{i,j}^{(1)}}$including one (1) intermediate vertex is calculated where **i**, **j** ∈ **V**.
- The process of calculating distance matrices $\boldsymbol{D^2, D^3 \ldots D^n}$by including other intermediate vertices continues until all vertices of the graph is taken.
- The last distance matrix $\boldsymbol{D^n}$ which includes all vertices of a matrix as intermediate vertices gives the final result.

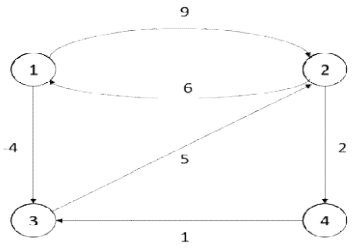
2.7.3 Pseudo-code for FWA

- **n** is the number of vertices in graph **G (V, E).**
- $\boldsymbol{D^k}$is the distance matrix of order $\boldsymbol{n \times n}$ consisting matrix element $\boldsymbol{d_{i,j}^{(k)}}$as the weight of shortest path having intermediate vertices from **1,2…k** ∈ **V**
- **M** is the initial matrix of direct edge weight. If there is no direct edge between two vertices, it will be considered as ∞.

```
FWA(M)
1. D^0 = M
2. For (k=1 to n)
3. For (i=1 to n)
4.                 For (j=1 to n)
5. d_{i,j}^{(k)} = min {d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}}
6. Return D^n
```

Since there are three nested for loops and inside which there is one statement which takes constant time for comparison and addition, the time complexity of FWA turns out to be $\boldsymbol{O(n^3)}$**.**

Example- Apply Floyd-Warshall Algorithm on the following graph-



We will create the initial matrix $\boldsymbol{D^0}$ from the given edge weights of the graph-

$$D^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & \infty & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

For creating distance matrices$\boldsymbol{D^1}$,$\boldsymbol{D^2}$ etc. We use two simple tricks for avoiding the calculations involved in each step-

1. Weight of shortest path from one vertex to itself will always be zero since we are dealing with no negative weight cycle. So we will put diagonal elements as zero in every iteration.
2. For the intermediate vertex **j**, we will put row and column elements of $\boldsymbol{D^j}$as it is for $\boldsymbol{D^{j-1}}$.

If we don't use the above tricks, still the result will be same.

Now for creating $\boldsymbol{D^1}$, we have to find distance between every two vertices via vertex 1. We will consider $\boldsymbol{D^0}$as base matrix as below-

$\boldsymbol{d^1[2,3]}$ = min $\{\boldsymbol{d^0[2,3]}, \boldsymbol{d^0[2,1] + d^0[1,3]} = \text{min } \{\infty, 6+(-4)\} = \text{min } \{\infty, 2\} = 2$

Note - *We have used* $\boldsymbol{d^k[i,j]}$ *instead of* $\boldsymbol{d_{i,j}^{(k)}}$ *for simplicity here.*

$\boldsymbol{d^1[2,4]}$ = min $\{\boldsymbol{d^0[2,4]}, \boldsymbol{d^0[2,1] + d^0[1,4]} = \text{min } \{2, 6+\infty\} = \text{min } \{2, \infty\} = 2$

$\boldsymbol{d^1[3,2]}$ = min $\{\boldsymbol{d^0[3,2]}, \boldsymbol{d^0[3,1] + d^0[1,2]} = \text{min } \{5, \infty+9\} = \text{min } \{5, \infty\} = 5$

Similarly, we will calculate for other vertices.

$$D^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & \infty \\ 6 & 0 & 2 & 2 \\ \infty & 5 & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

For creating $\boldsymbol{D^1}$, we have to find distance between every two vertices via vertex 2. We will consider $\boldsymbol{D^1}$ as base matrix as below-

$\boldsymbol{d^2[1,3]}$ = min $\{\boldsymbol{d^1[1,3]}, \boldsymbol{d^1[1,2]} + \boldsymbol{d^1[2,3]}$ = min $\{-4, 9+2\}$ = min $\{-4, 11\} = -4$

$\boldsymbol{d^2[1,4]}$ = min $\{\boldsymbol{d^1[1,4]}, \boldsymbol{d^1[1,2]} + \boldsymbol{d^1[2,4]}$ = min $\{\infty, 9+2\}$ = min $\{\infty, 11\} = 11$

$\boldsymbol{d^2[3,1]}$ = min $\{\boldsymbol{d^1[3,1]}, \boldsymbol{d^1[3,2] + d^1[2,1]}$ = min $\{\infty, 5+6\}$ = min $\{\infty, 11\} = 11$

Similarly, we will calculate for other vertices.

$$D^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 9 & -4 & 11 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ \infty & \infty & 1 & 0 \end{bmatrix} \end{array}$$

Create distance matrix $\boldsymbol{D^3}$ by taking vertex 3 as intermediate vertex and $\boldsymbol{D^2}$ as base matrix as below-

$$\boldsymbol{d^3[1,2]} = \min\{\boldsymbol{d^2[1,2]}, \boldsymbol{d^2[1,3]} + \boldsymbol{d^2[3,2]} = \min\{\mathbf{9}, -\mathbf{4} + \mathbf{5}\} = \min\{9,1\} = 1$$

$$\boldsymbol{d^3[1,4]} = \min\{\boldsymbol{d^2[1,4]}, \boldsymbol{d^2[1,3]} + \boldsymbol{d^2[3,4]} = \min\{\mathbf{11}, -\mathbf{4} + \mathbf{7}\} = \min\{11,3\} = 3$$

Similarly calculate for other pairs.

$$D^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

Create distance matrix $\boldsymbol{D^4}$ by taking vertex 4 as intermediate vertex and $\boldsymbol{D^3}$ as base matrix as below-

$$\boldsymbol{d^4[1,2]} = \min\{\boldsymbol{d^3[1,2]}, \boldsymbol{d^3[1,4]} + \boldsymbol{d^3[4,2]} = \min\{\mathbf{1}, \mathbf{3} + \mathbf{6}\} = \min\{\mathbf{1}, \mathbf{9}\} = \mathbf{1}$$

Similarly calculate for other pairs-

$$D^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & -4 & 3 \\ 6 & 0 & 2 & 2 \\ 11 & 5 & 0 & 7 \\ 12 & 6 & 1 & 0 \end{bmatrix} \end{array}$$

$\boldsymbol{D^4}$ gives the shortest path between any two vertices of given graph. e.g. weight of shortest path from vertex **1** to vertex **2** is **1**.

2.7.4 When FWA gives the best result

Do you remember the starting point of this topic where we stated that one can use Floyd Warshall Algorithm for finding all pair shortest path at one go or one can also use Dijkstra's or Bellman Ford algorithm for each vertex? We have analyzed the time complexity of FWA. Now we will find out the time complexities of running Dijkstra's and Bellman Ford algorithm for finding all pair shortest path to see which gives the best result-

Using Dijkstra's Algorithm - The time complexity of running Dijkstra's Algorithm for single source shortest path problem on graph **G(V,E)** is **O(E+VlogV)** using Fibonacci heap. For **G** being complete graph, running Dijkstra's Algorithm on each vertex will result in time complexity of **O($\boldsymbol{V^3}$).** For graph other than complete, it shall be **O(n*n log n),** less than FWA but since Dijkstra's Algorithm does not work with negative edge weight for single source shortest path problem it will also not work for all pair shortest path problem given negative edge weight.

Using Bellman Ford Algorithm – The time complexity of running Bellman Ford algorithm for single source shortest path problem on Graph **G (V,E)** is **O(VE).** If **G** is complete graph then this complexity turns out to be **O(V*$\boldsymbol{V^2}$)** i.e. **O($\boldsymbol{V^3}$).** Therefore,

the time complexity of running Bellman Ford Algorithm on each vertex of graph **G** shall be **O(V^4).**

From the above two points it can be concluded that FWA is the best choice for all pair shortest path problem when the graph is dense whereas Dijkstra's Algorithm is suitable when the graph is sparse and no negative edge weight exist. For graph having negative edge weight cycle the only choice among the three is Bellman Ford Algorithm.

2.8 SUMMARY

- The Dynamic Programming is a technique for solving optimization Problems, using bottom-up approach. The underlying idea of dynamic programming is to avoid calculating the same thing twice, usually by keeping a table of known results that fills up as substances of the problem under consideration are solved.
- In order that Dynamic Programming technique is applicable in solving an optimization problem, it is necessary that the principle of optimality is applicable to the problem domain.
- The principle of optimality states that for an optimal sequence of decisions or choices, each subsequence of decisions/choices must also be optimal.
- **The Chain Matrix Multiplication Problem:** The problem of how to parenthesize the pairs of matrices within the expression A1A2 … An, a product of n matrices which is defined; so as to minimize the number of scalar multiplications in the computation of the product A1A2 … An. The product is known as Chained Matrix Multiplication.

♣Check Your Progress:

Q1. Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is <5, 10, 3, 12, 5, 50, 6>.

Q2. Show that a full parenthesization of an n-element has exactly n-1 pairs of paranthesis.

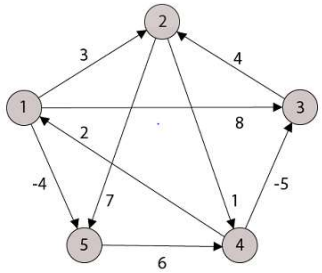
Q3. Determine the cost and structure of an optimal binary search tree for a set of n=7 keys with the following properties:

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Q4.Determine the cost and structure of an optimal binary search tree for a set of n=7 keys with the following properties:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Q5. Apply Floyd-Warshall algorithm on the following graph. Show the matrix D^5 of the graph.



Q6. When graph is dense and have negative edge weight cycle. Which of the following is a best choice to find the shortest path

a) Bellman-ford algorithm
b) Dijakstra's algorithm
c) Floyd-Warshall algorithm
d) Kruskal's Algorithm

Q7. What procedure is being followed in Floyd Warshall Algorithm?
a) Top down
b) Bottom up
c) Big bang
d) Sandwich

Q8. What happens when the value of k is 0 in the Floyd Warshall Algorithm?
a) 1 intermediate vertex
b) 0 intermediate vertex
c) N intermediate vertices
d) N-1 intermediate vertices

2.9 SOLUTION / ANSWERS

♣Check Your Progress Answer:

Q1. Multiplication sequence is (A1A2)(A3A4)(A5A6)

m	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

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

Q5
.

0	3	8	3	-4
3	0	11	1	-1
-3	0	0	-5	-7
2	5	10	0	-2
8	11	16	6	0

Q6.Floyd-warshall algorithm

Q7. Bottom up

Q8. When k=0, a path from vertex i to vertex j has no intermediate vertices at all. Such a path has at most one edge and hence $d_{ij}^{0} = w_{ij}$

UNIT 3 STRING MATCHING TESHNIQUES

Structure **Page No**

3.0 INTRODUCTION

String matching is an important problem in computer science in which a sub-string (also called a pattern) is searched in a larger string or a text (e.g., a sentence, a paragraph of a book) and returns the index of a starting character of a substring. When we search for a word in a text file/data base /browser, string matching algorithm is used to find the result. Some of its interesting applications are found in designing of text editors, plagiarism checking software, Spell Checkers, Search Engines, Bioinformatics, Digital Forensics and Information Retrieval Systems , searching for a pattern in DNA sequences, etc.,. In this unit we present three string matching algorithms: Brute Force or the Naïve algorithm, the Rabin-Karp algorithm and the Knuth-Morris-Pratt (KMP)algorithm. The Naïve algorithm is the simplest algorithm of all. The other two algorithms require some kinds of preprocessing.

3.1 OBJECTIVES

After going through this unit , you should be able to:

- Identify applications of string matching algorithms
- Write pseudo-codes of string matching algorithms
- Explain the working of string matching algorithms
- Calculate the running time efficiency of string matching algorithms

3.2 THE NAÏVE OR BRUTE FORCE ALGORITHM

The purpose of the string matching algorithm is to search for a location of a smaller text pattern in a paragraph or a book or any other sources.

The Naïve search algorithm compares the pattern string to the text, one character at a time .This process continues until there is mismatch of characters between the pattern

string and the text. Depending upon the requirement, the algorithm can be designed to find a single occurrence or multiple occurrences of a pattern string in the text. In the latter case, the entire text is required to be searched.

Pseudo-Code of Naïve String Matching Algorithm

do

{

if (pattern string character == text character)

compare next character of the pattern string to next character of text character;

else

shift the pattern string to the next character of the text;

while(entire pattern string matched or end of the text)

}

The following examples illustrates the concepts:

(i)Example 1(shift =0)

Text = a b c d e f g I

Pattern String = a b c

(ii) Example 2(shift =4)

Text = a b c d e f g h I

Pattern =e f g

(iii) Example 3(maximum valid shift in the text=(length of the text-length of pattern)=6)

Text = a b c d e f g h i

Pattern= g hi

Time Complexity of Naïve string matching algorithm

The algorithm finds matching of the pattern in a text using all valid shifts. Let us rewrite the algorithm:

1. n = length of a text T
2. m = length of a pattern string P
3. n-m = maximum valid shifts of a pattern in a text (refer to ex.3)
4. s – shift index
5. for (s = 0 , s<= n-m, s++)
6. if P[1..m] == T[s +1..s + m]
7. Display " Occurrence of a pattern string with shift" s

Best case- It happens if the pattern matches in the first m positions of the text. Total number of comparisons = m(size of the pattern string)
Therefore Best case time Complexity= O(m)
In the worst case scenario, the total number of comparisons: m(m-n+1)
Therefore worst case time complexity= O(mn)

☞ Check Your Progress-1

Q1 What is a string matching problem?

.....

.....

.....

.....

Q2 What are different types of string matching algorithms?

.....

.....

.....

.....

Q3What are the applications of string matching algorithms?

.....

.....

.....

.....

3.3 THE RABIN KARP ALGORITHM

The central idea in Rabin Karp algorithm is computation of hash function to speed up the pattern matching. The algorithm calculates hash values for (i) pattern string of m-characters (ii) m-character substring of a text . If the hash values of (i) and (ii) are equal, the algorithm will perform a brute force comparison between the pattern string and m-character text substring. Advantage of this approach is that there is only one comparison per text substring and brute force method is required only when hash values are equal.

If hash values do not match, the algorithm will pick up the next m-character text substring for calculation of its hash values.

Pseudo-code of Robin –Karp Algorithm

Input

m- length of a pattern substring

P_ hash – Hash value of a pattern string

T_ hash – Hash value of a first m character of a text substring

do

if(P_ hash == T_ hash)

brute force comparison of the pattern string and the first m-character text substring ;

else

T_ hash = hash value of the next m-character of the text substring after one character shift;

while(match of the pattern string **or** end of the text)

But how to design a hash function? In simple terms, a hash function maps a big n string to a small value. A good hash function should have properties such as: efficiently computable and should uniformly distribute the keys and do not generate spurious hits. The hash function considered here has been suggested in the Rabin Karp algorithm. It is also called **rolling hash function**

To compute the hash value , let us number the alphabet as a=1, b=2, c= 3, d=4,e=5,f= 6,g=7,h=8,i=9 and j=10 to simplify computation instead of assigning the ASCII value to each alphabet. Also consider m-character sequence as m-digit number having base 10, where 10 is the number of alphabets used in our pattern string . Now let us convert m-character text which can be written in form of a polynomial expression:

hash(m-character sequence)= P[1]* 10^{m-1} +P[2] *10^{m-2}+P[3]* $10^{m-3}$where P[1], P[2],P[3] are first, second and third equivalent digital values of 3-character pattern. If the first character is 'a' then P[1] =1. Furthermore, given hash(m- character string), we can compute the hash value of the next m-character substring skipping one character by subtracting the leftmost digit and adding new rightmost digit in constant time. Using this approach, we simply adjust the existing value but never explicitly compute the new value. This is demonstrated through an example below:

Now let us take one example to understand :

Text T = "baecddabcdef"

Pattern P= "ecd".

Step 1: Calculate the hash value of a pattern string P

Hash(P) =5*10^2 + 3*10^1 + 4* 10^0 = 534

Step 2: Compute hash value of a text substring of the size equal to P

Hash(T=bae) = $2 *10^2+ 1* 10^1+ 5* 10^0= 215$

Step 3 : Compare the hash values of P and T(a text substring). Since both are not equal, pick up the next text substring sliding over one character, compute its hash value and compare it with P. Using rolling hash function, the hash value of the next text substring can be calculated easily from the previous expression by subtracting the first digit , multiplying the second and the third digits by 10 and finally adding the new third digit as :

- Hash (aec) = $[2 *10^2+ 1* 10^1+ 5* 10^0]- 2 *10^2] *10+3*10^0$
  If you write it directly, you get the same expression:

  $1* 10^2+ 5* 10^1+ 3* 10^0= 153$

Step 3(contd..)Again there is mismatch, so we will take the third text substring "ecd", compute its hash value and compare it P

Hash (ecd)= $[1* 10^2+ 5* 10^1+ 3* 10^0]- 1* 10^2] * 10] + 4*10^0$

$= 53* 10 +4 = 534$

The hash value of both are equal. Then we compare each character of a pattern with each character of a text substring. All characters are equal. Therefore the pattern string was found.

Rabin-Karp Complexity

Best Case –O(n) where n is a length of a text. If sufficiently large base number or a large prime number is used for computing hash value , there will be no spurious hits and the hashed values would be distinct for both a pattern string and a text substring. In such a case , the searching would take O(n) time

Worst Case = O (*mn*) where m is a length of a pattern string and n is a length of a text string.. This may happen if there are spurious hits because of use a small base number/prime number in hash calculation of a pattern and a text string

☞ Check Your Progress-2

Q1 How does the Rabin Karp algorithm work?

Q2 What is the worst case time complexity of Rabin Karp algorithm?

3.4 KNUTH MORRIS PRATT ALGORITHM

This is linear time string matching algorithm. The complexity is **O(*m*+*n*)** where m and n are the length of a pattern string and a text string respectively. This happens because the KPS algorithm avoids frequent backtracking in the text string as it is done in the naïve algorithm. The key idea in KMP algorithm is to build a LPS(largest prefix as suffix) array to determine from which point in the pattern string to restart comparing for pattern matching in a text in case there is a mismatch of a character **without moving the text pointer backward**. In such a case first we go back one character backward from the position where mismatch occurred, read its value in the LPS array which defines the length of a prefix also as a suffix if any ,i.e., we check whether there is any occurrence of the largest prefix as a suffix in the pattern to decide how many characters in the pattern need to be skipped to start searching for the string matching hat the next stage. If there is a mismatch at the i^{th} character of a pattern string, we move to $(i-1)^{th}$ character) in the pattern string and find out the LPS array value of this character. Suppose the LPS array value of $(i-1)^{th}$ character is 2 . This number defines the length of the largest prefix which is also a suffix in a pattern string. It also indicates the first two characters in the pattern string need to be skipped for the next comparison of pattern string with a text. If the length of a pattern string is m then only (m-2) characters will be compared with a text(not from the beginning of the text but from the position where there was a mismatch).

In the following example we first do the pattern matching exercise without building the LPS array to get the idea quickly. But efficient implementation of KPS would be done through LPS array only.

Example :

Text :abcfa b c x a b f a b a b c x a b c z

Pattern = abcxa b c z

Examining the text and the pattern string we notice that there is mismatch of characters at the fourth position. In the text string it is ‘f’ where as in the pattern string it is ’x’. To decide from which position in the pattern string the search should restart , we go back and examine the substring in the pattern just before the position where there was a mismatch, i.e., we examine “a b c” substring of pattern (so far we have not built up the LPS array). Since all the characters are unique, there is no prefix also as a suffix in this substring, therefore we can not skip any character in the pattern string, the comparison will start from the beginning character of the pattern, i.e., ‘a’ with ‘f’(at this position , there was a mismatch). Again there is a mismatch, so we will move to the next character in the text, i.e., ‘a’ and start comparing with the first

character in the pattern string ,i.e., 'a'. There is a match of a character, we will compare the next character of a pattern with the next character of a text and continue till there is a mismatch. Please notice that there is a mismatch at the 7th character in the pattern, i.e., 'c' with 'f' in the text. Now we examine the substring of a pattern just before the position of the mismatch, i.e., "a b c x a b " and decide how many characters to be skipped in the pattern. We try to find out the length of prefix also as a suffix in this substring. It is "ab" which is suffix as well as prefix and the length is 2. Therefore we will skip two characters in the pattern from the beginning, which is now, 'c' and start comparing with the same position of character in the text where there was a mismatch, i.e., 'f'. It makes a sense because " ab " is existing in the text just before 'f'. It need not be compared again. Now there is a mismatch, so we go back and examine the substring in pattern to find out if there is any prefix which is also as a suffix. The substring is "ab". There is no prefix and suffix information in the substring, because both are unique characters. Therefore the comparison starts with the first character of the pattern ('a') and the next character in the text('a'). The next character is 'b' in the pattern as well as in the text. Finally we find that the pattern is found in the text .

It is time now to build a LPS array for a pattern P.

P = d e f g d e f

I j

d	e	f	g	D	e	f	d
0	1	2	3	4	5	6	7
0	0	0	0	1	2	3	1

Figure : LPS Array (last row)

In the above table , the last row builds information about prefix and suffix of the pattern. The first and the second row indicate the pattern and its index values. We consider two pointers i and j. Initially i is pointing to the first character of the pattern and j is pointing to the second character of the pattern. The first entry in LPS array(last row of the table) is zero. For the second entry, we compare i with j which are not equal. i is pointing to d and j is pointing to e. If i and j are not equal, the related entry in the LPS array will be zero. Then j will be incremented by one. Again i and j are not equal. i is pointing to a and j is pointing to 'f'. The LPS entry for this index will be zero. Similarly the next entry in the LPS array is zero again. What will happen when j is pointing to d and i is also pointing to d. In this case the LPS entry of the character pointed to by j will be equal to the current index value of i and + 1. Therefore the entry will be 1 at this column. The next two entries will be 2 and 3. Each time there is a match, i will get incremented. Now i is pointing to g and j is pointing to d. There is mismatch. In this case i will be decremented by 1. i is pointing to 'f' now. Its LPS entry value is zero. Accordingly i will shift to the beginning of the array. Both i and j are pointing to the same character. The LPS entry for the last character will be the index value of i, which is zero plus 1,i.e., 1

The question is now how to use LPS array for pattern matching? Please refer to the last row of the array. There is entry 1 at the 4th column. We will ignore the remaining entries in the LPS array after this entry. It indicates that the length of a prefix which is also a suffix is one. Therefore skip one character from the beginning in the pattern for the next comparison. Let us take one example:

Text = d e f g d x . . . .

Pattern = d e f g d e

There is a mismatch between 'x' and 'e' characters. After mismatch we go back to the previous character, i.e., 'd'. The entry for d in the array is 1 , which says that the next comparison will start from 'e' in the pattern and 'x' in the text.

Time Complexity of KMP Algorithm

The worst case time complexity of KMP is O(m+n) where O(m) is time taken to build LPS array and O(n) is the time taken to search for entire text.

☞ Check your Progress-3

Q1 What is the basic principle in KMP algorithm?

.....

.....

.....

.....

Q2. How do you build LPS array in KMP algorithm?

.....

.....

.....

.....

3.5 SUMMARY

String matching is a problem in which we look for a pattern string into a larger text and return the location where the pattern has occurred in the text. In this unit we examined three string matching algorithms: Naïve algorithm, Rabin Karp Algorithm and Knuth, Morris and Pratt algorithm. The naïve algorithm is the simplest algorithm of all. The Rabin Karp algorithm is based on computing the hash values of an m-character pattern string and an m-character text substring. If the hash values are equal then only we make comparison of the pattern string and a text substring character by character, otherwise we compute the hash value of the next text substring and compare with the pattern string unless there is an end of the text or the pattern was found at some step. The KMP is linear time algorithm. The algorithm forbids moving a text pointer backward.

3.6 SOLUTION / ANSWERS

☞ Check Your Progress-1

Q1 What is a string matching problem?

Ans- The string matching is a problem of finding occurrence(s) of a **pattern string** within a larger text. If there is a match, the algorithm returns the location of the text where the pattern has occurred.

Q2 What are different types of string matching algorithms?

1. The Naive String Matching Algorithm
2. The Rabin-Karp-Algorithm
3. Finite Automata (not discussed in this unit)
4. The Knuth-Morris-Pratt Algorithm
5. The Boyer-Moore Algorithm(not discussed in this unit)

Q3 What are the applications of string matching algorithms?

Ans. String matching algorithms have found applications in various real world problems .A few applications are spell checkers, information retrieval systems, spam filters, intrusion detection system, search engines and plagiarism detection

☞ Check Your Progress-2

Q1 How does the Rabin Karp algorithm work?

AnsThe Rabin-Karp string matching algorithm performs calculation of a hash value for the pattern string , as well as for each m-character substring(size of a pattern string) of text to be compared. If the hash values are equal, the algorithm will compare the pattern string and the m-character text substring . In this way, there is only one comparison per text substring, and character by character pattern matching is only needed when the hash values match. In case the hash values do not match, the algorithm will determine the hash value for the next m-character substring.

Q2 What is the worst case time complexity of Rabin Karp algorithm?

Ans The worst **case complexity** is O(mn) . The worst-**case complexity** occurs when spurious hits occur very frequently because of use a small base number/prime number in hash calculation of a pattern and a text string

☞ Check your Progress-3

Q1 What is the basic principle in KMP algorithm?

The Knuth–Morris–Pratt string-searching **algorithm** (or **KMP algorithm**) searches for occurrences of a pattern string within a text by employing the observation that when a mismatch occurs, the pattern itself contain sufficient information related to prefix and suffix which determine where the next match could begin.

Q2 How do you build LPS array in KMP algorithm?

Steps for Creating LPS array

Step 1 - Define a one dimensional array with the size equal to the length of the Pattern string P

Step 2 - Define two variables i& j. Let i point to the first character and j points to the second character of P

Step 3. LPS[0] = 0

Step 4 - Compare the characters at P[i] and P[j].

Step 5 - If both are matched then set **LPS**[j] = i+1 and increment both i& j values by one.