BHARATI VIDYAPEETH PUNE

Data and File Structures

(MCA-102)

Unit – 3

[Graph]

by

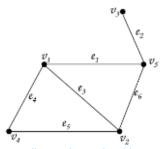
Dr. Sunil Pratap Singh
(Assistant Professor, BVICAM, New Delhi)

2021

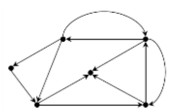
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.1

Graph

- A graph G = (V,E) consists of a finite set of vertices V = {v_1, v_2, . . . , v_n} and a finite set E of edges E = {e_1, e_2, . . . , e_m}.
- To each edge *e,* there corresponds a pair of vertices (*u, v*) where *e* is said to be *incident* on.
- A graph is said to be a directed graph (or digraph for short) if the vertex pair (u, v) associated with each edge e (also called arc) is an ordered pair.



Undirected graph with 5 vertices and 6 edges



Digraph with 6 vertices and 11 edges

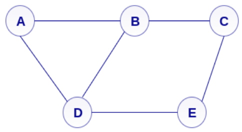
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.2

Representation of Graphs in Computer

- Array-based Representation
  - Using Adjacency Matrix
- Linked Representation
  - Using Adjacency List

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.3

Array-based (2-D Array) Representation



Undirected Graph

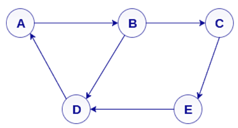
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

Adjacency Matrix

Matrix Representation of an Undirected Graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Array-based (2-D Array) Representation



Directed Graph

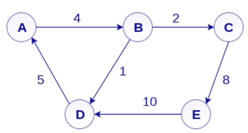
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

Matrix Representation of a Direct Graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Array-based (2-D Array) Representation



Weighted Directed Graph

	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

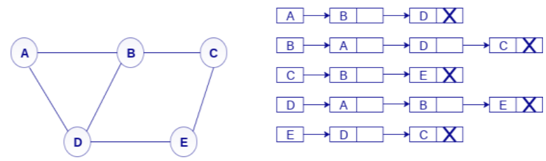
Adjacency Matrix

Matrix Representation of a Weighted Graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Adjacency List Representation



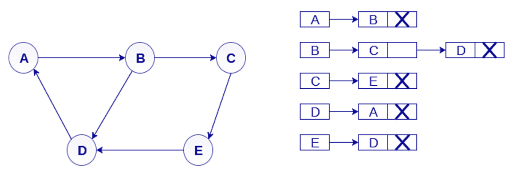
Undirected Graph Adjacency List

Adjacency List Representation of an Undirected Graph

Note: Adjacency List implementation needs Array and Linked List

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Adjacency List Representation

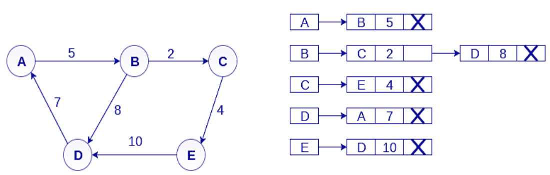


Directed Graph Adjacency List

Adjacency List Representation of a Directed Graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Adjacency List Representation



Weighted Directed Graph Adjacency List

Adjacency List Representation of a Weighted Graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Operations of Graph

- Insertion
  - There are two major components of a graph – Vertex and Edge. Therefore, a node or an edge or both can be inserted into an existing graph.
- Deletion
  - Similarly, a node or an edge or both can be deleted from an existing graph.
- Traversal
  - A graph may be traversed for many purposes – to search a path, to search a shortest path b/w two given nodes, etc.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Insertion Operation

- Insertion of a vertex and its associated edge with other vertices in an adjacency matrix involves:
  - Add a row for the new vertex
  - Add a column for the new vertex
  - Make appropriate entries into the rows and columns of the matrix.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Algorithm: addVertex() – Undirected Graph

```
Algorithm addVertex()
{
    lastRow = lastRow + 1;
    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = 0;
    Set all elements of last col = 0;
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Algorithm: addEdge() – Undirected Graph

```
// A new edge (v1, v2) is added to matrix with entry 1.
Algorithm addEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 1;
    adjMat[colV2][rowV1] = 1;
}
```


Algorithm: addVertex() – Directed Graph

```
Algorithm addVertex()
{
    lastRow = lastRow + 1;
    lastCol = lastCol + 1;
    adjMat[lastRow][0] = verTex;
    adjMat[0][lastCol] = verTex;
    Set all elements of last row = 0;
    Set all elements of last col = 0;
}
```


Algorithm: addEdge() – Directed Graph

```
// A new edge (v1, v2) is added to matrix with entry 1.
Algorithm addEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find col corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 1;
}
```

Delete Operation

- Deletion of a vertex and its associated edge involves:
  - Delete the row corresponding to the vertex
  - Delete the col corresponding to the vertex
  - **Mark 0 in relation with other vertices (if other vertices are adjacent to the deleted vertex).**

Algorithm: delVertex() – Undirected Graph

```
Algorithm delVertex()
{
  Find row corresponding to verTex and set all ist elements = 0;
  Find col corresponding to verTex and set all ist elements = 0;
}
```


Algorithm: delEdge() – Undirected Graph

```
Algorithm delEdge()
{
    Find row corresponding to v1, i.e., rowV1;
    Find row corresponding to v2, i.e., colV2;
    adjMat[rowV1][colV2] = 0;
    adjMat[colV2][rowV1] = 0;
}
```

Graph Traversal

- Depth First Search
- Breadth First Search
- Spanning Tree
  - Minimum Cost Spanning Tree
    - Prim's Algorithm
    - Kruskal's Algorithm

Graph Traversal: BFS

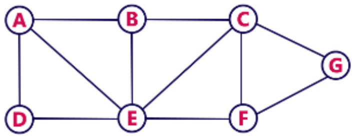
- Breadth First Search (BFS) is an algorithm for traversing or searching graph data structures.
  - It starts at the root (selecting some arbitrary node as the root) and explores the neighbor nodes first, before moving to the next level neighbors.

Breadth First Search (BFS)

- **Step 1:** Define a Queue of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3:** Visit all the **adjacent** vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
- **Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
- **Step 5:** Repeat step 3 and 4 until queue becomes empty.
- **Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

BFS with Queue: Example

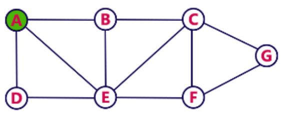
Consider the following example graph to perform BFS traversal



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

BFS with Queue: Example

Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.



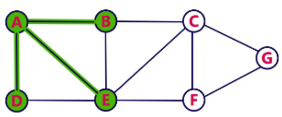
Queue

A						

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

BFS with Queue: Example

Step 2:
- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..



Queue

	D	E	B			

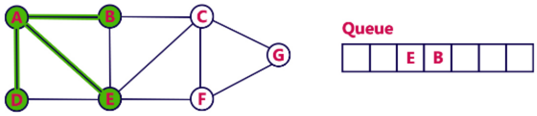
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

BFS with Queue: Example

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

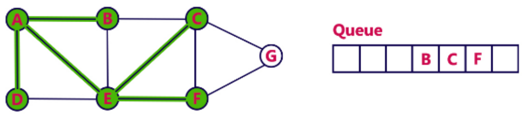


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

BFS with Queue: Example

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

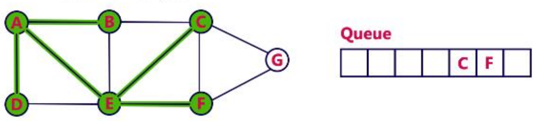


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

BFS with Queue: Example

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

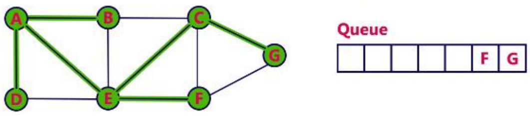


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

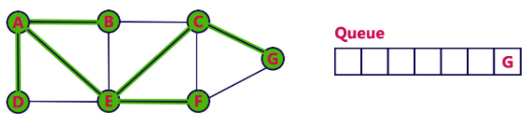
BFS with Queue: Example

Step 6:
- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.



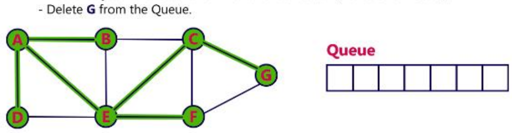
BFS with Queue: Example

Step 7:
- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.



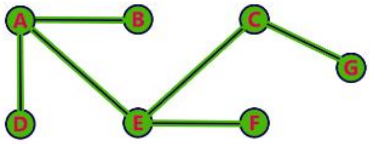
BFS with Queue: Example

Step 8:
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



BFS with Queue: Example

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.31

Graph Traversal: DFS

- Depth First Search (DFS) is an algorithm for traversing or searching graph data structures.
  - It starts at the root (selecting some arbitrary node as the root) and explores as far as possible along each branch before backtracking.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.32

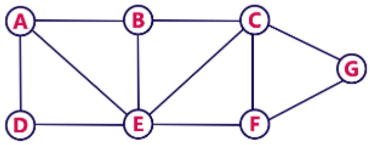
Depth First Search using Stack

- **Step 1:** Define a Stack of size total number of vertices in the graph.
- **Step 2:** Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3:** Visit any one of the **adjacent** vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.
- **Step 4:** Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.
- **Step 5:** When there is no new vertex to be visit then use **backtracking** and pop one vertex from the stack.
- **Step 6:** Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7:** When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.33

Example of DFS

Consider the following example graph to perform DFS traversal

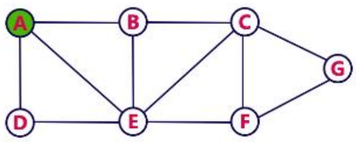


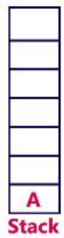
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.34

DFS with Stack: Example

Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



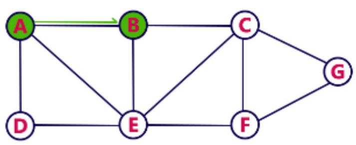


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.35

DFS with Stack: Example

Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

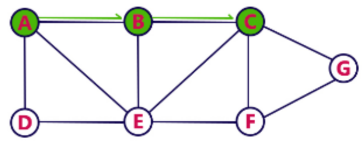


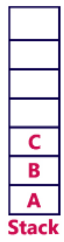
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.36

DFS with Stack: Example

Step 3:

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.



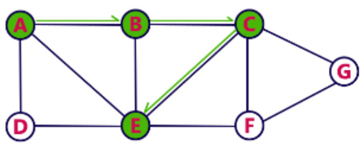


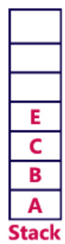
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.37

DFS with Stack: Example

Step 4:

- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack



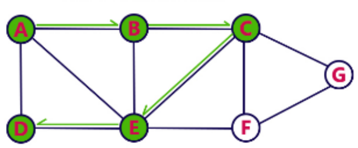


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.38

DFS with Stack: Example

Step 5:

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack

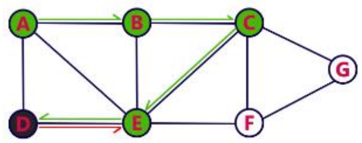


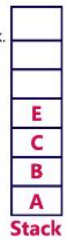
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.39

DFS with Stack: Example

Step 6:

- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



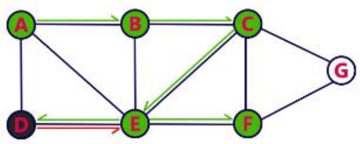


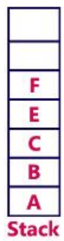
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.40

DFS with Stack: Example

Step 7:

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



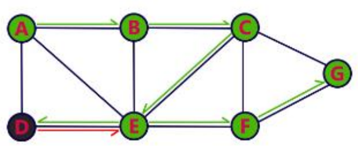


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.41

DFS with Stack: Example

Step 8:

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

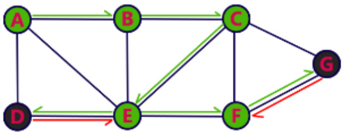


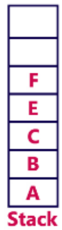
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.42

DFS with Stack: Example

Step 9:

- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.



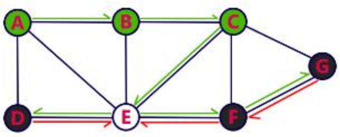


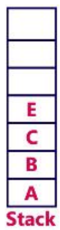
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

Step 10:

- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.



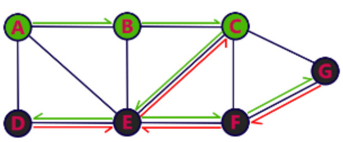


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

Step 11:

- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

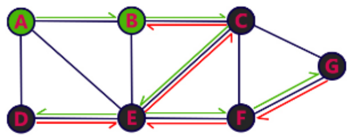


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

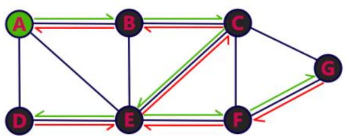
Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.

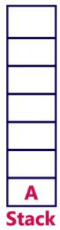


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.

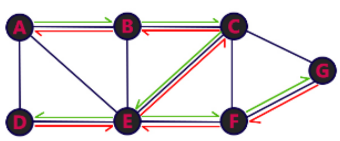


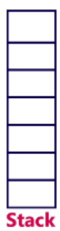


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



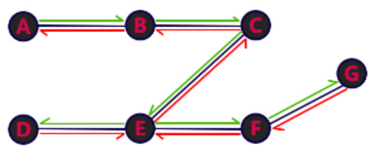


© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

DFS with Stack: Example

- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.



BFS and DFS in Directed Graphs

- Similar to undirected graphs, the same processes work for directed graphs.
- The only difference is that when exploring a vertex v, we only want to look at edges (v,w) going out of v; we ignore the other edges coming into v.
- BFS finds shortest (link-distance) paths from a single source vertex to all other vertices.

Spanning Tree

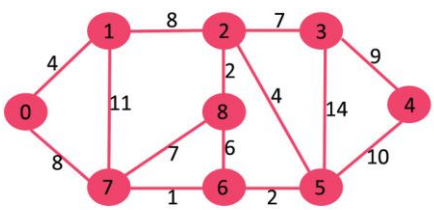
- Given a connected and undirected graph, a **spanning tree** of that graph is a sub-graph that is a tree and connects all the vertices together.
- A single graph can have many different spanning trees.
- A **minimum spanning tree** (MST) for a weighted, connected and undirected graph is a spanning tree with weight less than or equal to the weight of every other spanning tree.
  - Prim's Algorithm
  - Kruskal's Algorithm

Steps for finding MST using Kruskal's Algo.

Let G be a graph with V vertices:

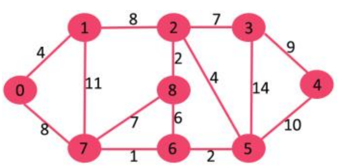
1) Sort all the edges in increasing order of their weight.
2) Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.
   a) If cycle is not formed, include this edge.
   b) Else, discard it.
3) Repeat step 2 until there are (V-1) edges in the spanning tree.

MST using Kruskal's Algo.: Example



The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

MST using Kruskal's Algo.: Example 1 (contd...)



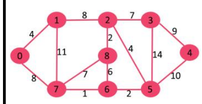
After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

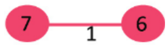
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



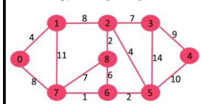
1. Pick edge 7-6: No cycle is formed, include it..



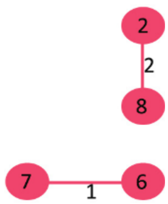
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



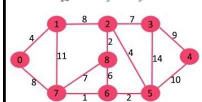
2. Pick edge 8-2: No cycle is formed, include it.



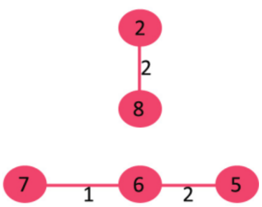
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



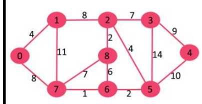
3. Pick edge 6-5: No cycle is formed, include it.



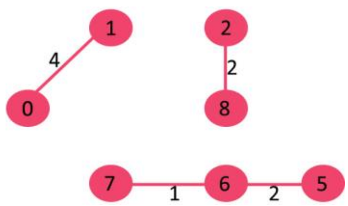
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



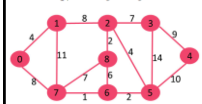
4. Pick edge 0-1: No cycle is formed, include it.



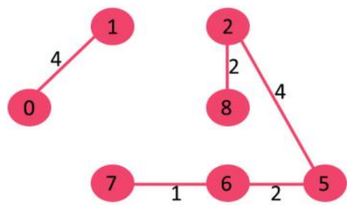
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



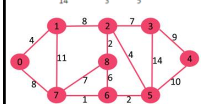
5. Pick edge 2-5: No cycle is formed, include it.



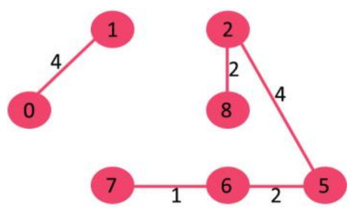
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



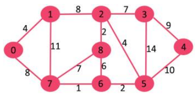
6. Pick edge 8-6: Since including this edge results in cycle, discard it.



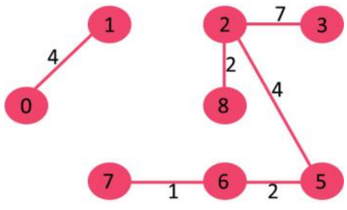
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



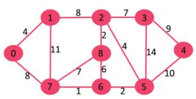
7. Pick edge 2-3: No cycle is formed, include it



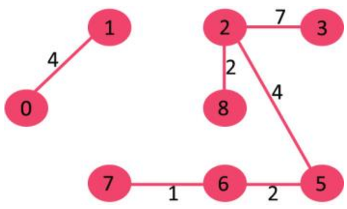
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



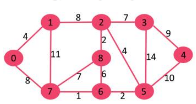
8. Pick edge 7-8: Since including this edge results in cycle, discard it.



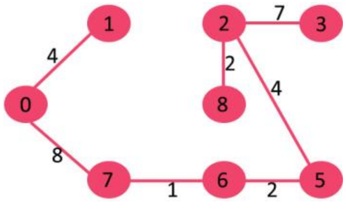
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



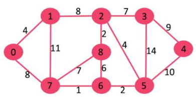
9. Pick edge 0-7: No cycle is formed, include it.



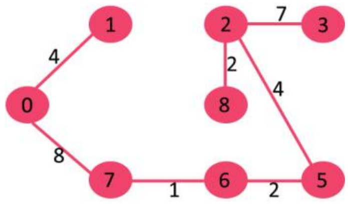
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5



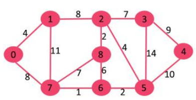
10. Pick edge 1-2: Since including this edge results in cycle, discard it.



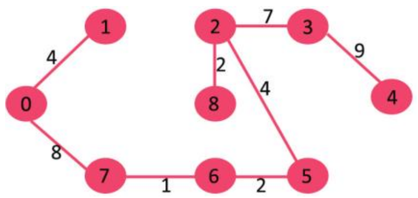
MST using Kruskal's Algo.: Example 1 (contd...)

After sorting:

Weight	Src	Dest
1	7	6
2	8	2
2	6	5
4	0	1
4	2	5
6	8	6
7	2	3
7	7	8
8	0	7
8	1	2
9	3	4
10	5	4
11	1	7
14	3	5

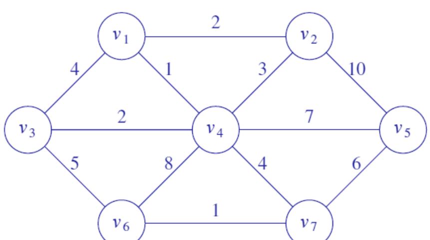


11. Pick edge 3-4: No cycle is formed, include it.



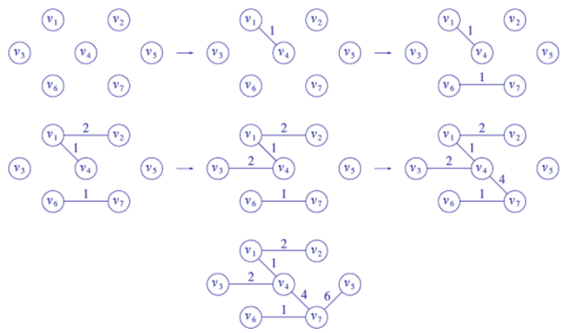
Since the number of edges included equals (V – 1), the algorithm stops here.

MST using Kruskal's Algo.: Example 2



Graph

MST using Kruskal's Algo.: Example 2 (contd...)



Minimum Cost Spanning Tree

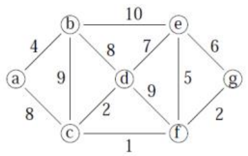
Steps for finding MST using Prim's Algo.

Step 0: Choose any element r; set $S = \{r\}$ and $A = \emptyset$. (Take r as the root of our spanning tree.)

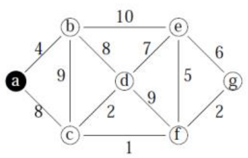
Step 1: Find a lightest edge such that one endpoint is in S and the other is in $V \setminus S$. Add this edge to A and its (other) endpoint to S.

Step 2: If $V \setminus S = \emptyset$, then stop & output (minimum) spanning tree (S, A). Otherwise go to Step 1.

MST using Prim's Algo.: Example 1



Connected graph



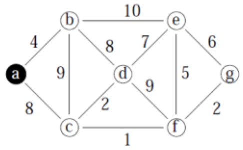
Step 0

S={a}

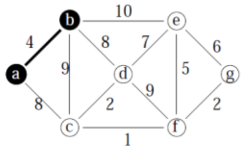
V \ S = {b,c,d,e,f,g}

lightest edge = {a,b}

MST using Prim's Algo.: Example 1 (contd...)



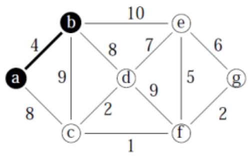
Step 1.1 before
S={a}
V \ S = {b,c,d,e,f,g}
A={}
lightest edge = {a,b}



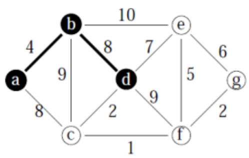
Step 1.1 after
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.70

MST using Prim's Algo.: Example 1 (contd...)



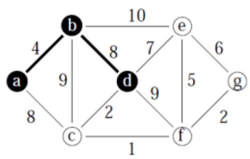
Step 1.2 before
S={a,b}
V \ S = {c,d,e,f,g}
A={{a,b}}
lightest edge = {b,d}, {a,c}



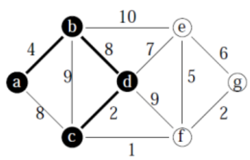
Step 1.2 after
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.71

MST using Prim's Algo.: Example 1 (contd...)



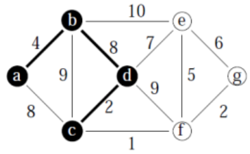
Step 1.3 before
S={a,b,d}
V \ S = {c,e,f,g}
A={{a,b},{b,d}}
lightest edge = {d,c}



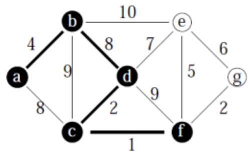
Step 1.3 after
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.72

MST using Prim's Algo.: Example 1 (contd...)

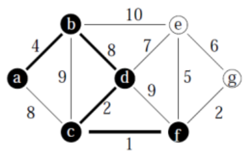


Step 1.4 before
S={a,b,c,d}
V \ S = {e,f,g}
A={{a,b},{b,d},{c,d}}
lightest edge = {c,f}

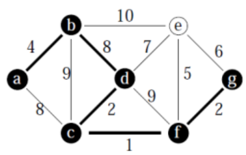


Step 1.4 after
S={a,b,c,d,f}
V \ S = {e,g}
A={{a,b},{b,d},{c,d},{c,f}}
lightest edge = {f,g}

MST using Prim's Algo.: Example 1 (contd...)

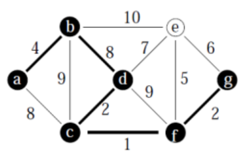


Step 1.5 before
S={a,b,c,d,f}
V \ S = {e,g}
A={{a,b},{b,d},{c,d},{c,f}}
lightest edge = {f,g}

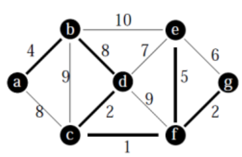


Step 1.5 after
S={a,b,c,d,f,g}
V \ S = {e}
A={{a,b},{b,d},{c,d},{c,f},
{f,g}}
lightest edge = {f,e}

MST using Prim's Algo.: Example 1 (contd...)

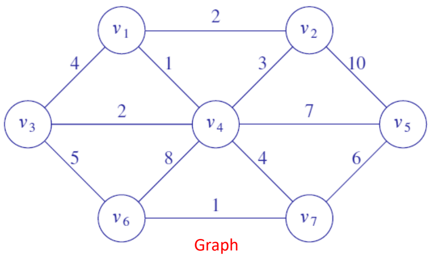


Step 1.6 before
S={a,b,c,d,f,g}
V \ S = {e}
A={{a,b},{b,d},{c,d},{c,f},
{f,g}}
lightest edge = {f,e}



Step 1.6 after
S={a,b,c,d,e,f,g}
V \ S = {}
A={{a,b},{b,d},{c,d},{c,f},
{f,g},{f,e}}
MST completed

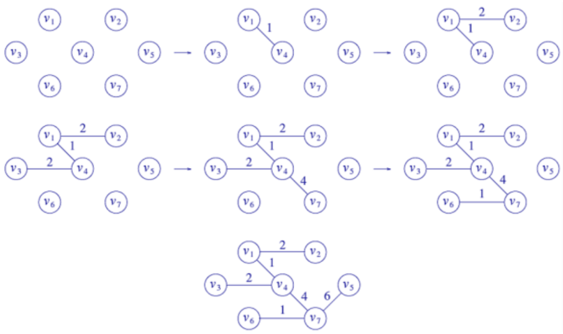
MST using Prim’s Algo.: Example 2



Graph

© Bharati Vidyapeeth’s Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

MST using Prim’s Algo.: Example 2 (contd...)



Minimum Cost Spanning Tree

© Bharati Vidyapeeth’s Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Prim’s Algorithm (Programming) for MST

Let G be a graph with **V** vertices:

1) Create an array **Parent[]** of size V and initialize it with **NIL**.
2) Create a Min Heap of size V. Let the Min Heap be **H**.
3) Insert all vertices to **H** such that the key value of starting vertex is 0 and key value of other vertices is infinite.
4) While **H** is not empty
   a) u = extractMin(H).
   b) For every adjcent **v** of **u**,

      if v is in H

      (i) Update key value of **v** in **H** if weight of edge **u - v** is smaller than current key value of **v**.

      (ii) Parent[v] = u

© Bharati Vidyapeeth’s Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth’s Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

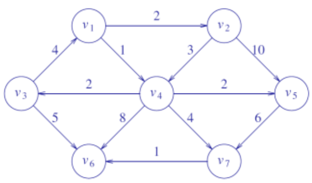
Shortest-Path Problems for Graphs

- **Single-Source Shortest-Path Problem:**
  - Given a (di)graph and a distinguished source vertex, s ϵ V, **determine the shortest path from the source vertex s to every other vertex in the graph.**
- **All-Pairs Shortest-Path Problem**
  - Given a directed graph, **determine the shortest path between all pairs of vertices in the weighted digraph.**

Single-Source Shortest-Path Problem

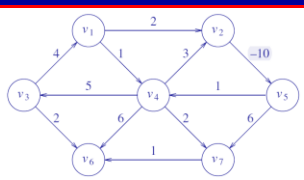
Given a (di)graph G = (V, E) and a distinguished **source vertex**, s ϵ V, **determine the shortest path from the source vertex s to every other vertex in the graph.**

- The shortest weighted path from v1 to v6 has a cost of 6 and goes from **v1** to **v4** to **v7** to **v6**.
- The shortest unweighted path from v1 to v6 has a cost of 2 and goes from **v1** to **v4** to **v6**.
- There is no path from **v6** to **v1**.



Single-Source Shortest-Path Problem (contd…)

- The path from v5 to v4 has cost **1**, but a shorter path exists by following the loop **v5**, **v4**, **v2**, **v5**, **v4**, which has cost **–5**.
- This path is still not the shortest, because we could stay in the loop arbitrarily long.

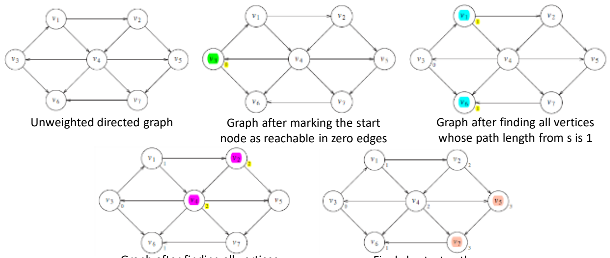


Graph with a negative-cost cycle

- Thus, the shortest path between these two points (v5 to v4) is undefined due to the loop.
- This loop is known as a **negative-cost cycle**; when one is present in the graph, the shortest paths are not defined.

Unweighted Shortest Paths

- In an unweighted (di)graph, the shortest paths from a single source vertex to all other vertices can be determined by following the procedure of **BFS**, which processes the vertices in increasing order of their distance from the source vertex.



Unweighted directed graph

Graph after marking the start node as reachable in zero edges

Graph after finding all vertices whose path length from s is 1

Graph after finding all vertices whose shortest path is 2

Final shortest paths

Dijkstra's Algorithm

- **Purpose and Use Cases**
  - Find the shortest path from a node (called the "source node") to all other nodes in the graph.
  - This algorithm is used in GPS devices to find the shortest path between the current location and the destination.
  - It has broad applications in industry, specially in domains that require modeling networks.
- **History**
  - In 1959, the algorithm was published by Dr. Edsger W. Dijkstra, a brilliant Dutch Computer Scientist and Software Engineer.

Dijkstra's Algorithm (contd…)

- **Basics of the Algorithm**
  - The algorithm basically starts at the node that we choose (the source node) and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
  - The algorithm keeps track of the currently known shortest distance from each node to the source node and it updates these values if it finds a shorter path.
  - Once the algorithm has found the shortest path between the source node and another node, that node is marked as "visited" and added to the path.
  - The process continues until all the nodes in the graph have been added to the path.

NOTE: This algorithm works for both directed and undirected graphs. It works only for connected graphs. The graph should not contain negative edge weights.

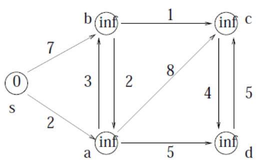
Dijkstra's Algorithm (Pseudocode)

```
Dijkstra(G,w,s)
{                                              % Initialize
    for (each u ∈ V)
    {
        d[u] = ∞;
        color[u] =white;
    }
    d[s] = 0;
    pred[s] = NIL;
    Q = (queue with all vertices);

    while (Non-Empty(Q))                       % Process all vertices
    {
        u = Extract-Min(Q);                    % Find new vertex
        for (each v ∈ Adj[u])
            if (d[u] + w(u,v) < d[v])          % If estimate improves
            {
                d[v] = d[u] + w(u,v);              relax
                Decrease-Key(Q, v, d[v]);
                pred[v] = u;
            }
        color[u] =black;
    }
}
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Shortest-Path using Dijkstra's Algo: Example



Step 0: Initialization.

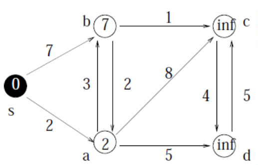
v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Shortest-Path using Dijkstra's Algo: Example



Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

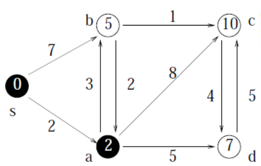
Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Shortest-Path using Dijkstra's Algo: Example



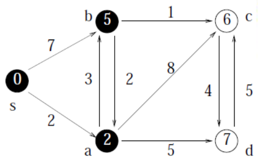
Step 2: After Step 1, a has the minimum key in the priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

Shortest-Path using Dijkstra's Algo: Example



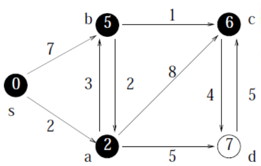
Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

Shortest-Path using Dijkstra's Algo: Example



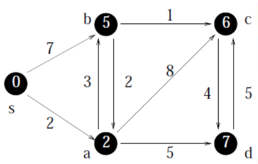
Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

v	d
$d[v]$	7

Shortest-Path using Dijkstra's Algo: Example



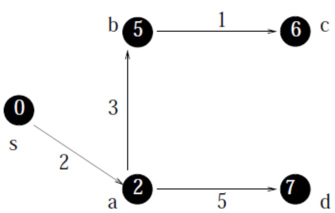
Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.

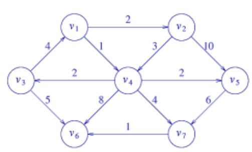
The array pred[v] is used to build the shortest-path tree.

Shortest-Path using Dijkstra's Algo: Example

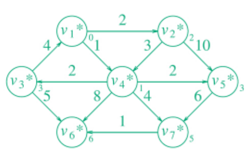


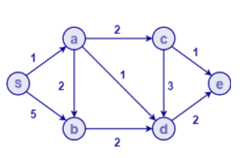
v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a

More Problems related to Dijkstra's Algo.

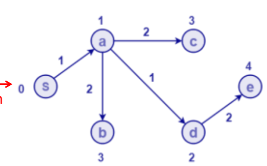


Solution

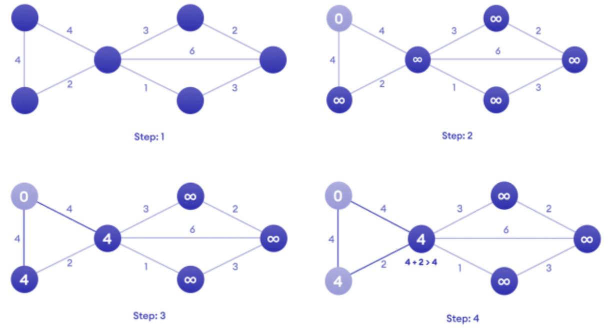




Solution

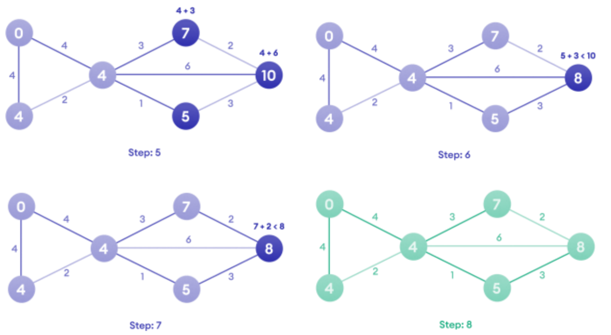


Ex. of Dijkstra's Algo. for Undirected Graph



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.94

Ex. of Dijkstra's Algo. for Undirected Graph



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.95

All-Pairs Shortest-Path Problem

- Given a weighted digraph G = (V, E) with a weight function w: E → R, where R is the set of real numbers, determine the length of the shortest path (i.e., distance) between all pairs of vertices in G.
- **Solution 1:** Assume no negative edges. Run Dijkstra's algorithm, n times, once with each vertex as source. What's the time complexity?
- **Solution 2:** Floyd-Warshall algorithm (dynamic programming) with time complexity $O(n^3)$, where n is the number of vertices (|V|) in G.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.96

Floyd-Warshall's Algorithm: Background

- Floyd-Warshall's algorithm is a graph analysis algorithm for finding shortest paths in a weighted, directed graph.
- A single execution of the algorithm will find the shortest paths between all pairs of vertices.
- This algorithm compares all possible paths through the graph between each pair of vertices.

Floyd-Warshall's Algorithm: Background

- Let $d_{ij}^{(k)}$ be the length of the shortest path from i to j such that *all* intermediate vertices on the path (if any) are in set $\{1, 2, \ldots, k\}$.
- $d_{ij}^{(0)}$ is set to be w_{ij}, i.e., no intermediate vertex.
- Let $D^{(k)}$ be the n by n matrix $[d_{ij}^{(k)}]$.
- Claim: $d_{ij}^{(n)}$ is the shortest distance from i to j, with the intermediate vertices set $\{1, 2, \ldots, n\}$. So our aim is to compute $D^{(n)}$.

Observation: For a shortest path from i to j such that any intermediate vertices on the path are chosen from the set {1, 2, ..., k}, there are two possibilities:

1. k is not a vertex on the path, the shortest such path has length $d_{ij}^{(k-1)}$.
2. k is a vertex on the path, the shortest such path has length $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

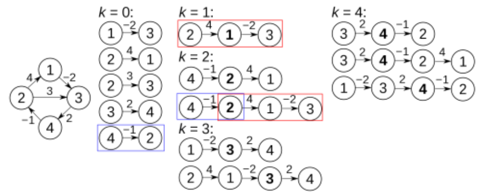
Combining the above two cases we get:

$$d_{ij}^{(k)} = \min \{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

Dijkstra's Algorithm (Pseudocode)

```
Floyd-Warshall(w, n) // w: weights, n: number of vertices
{
  for i = 1 to n do      // initialize, D^(0) = [w_ij]
    for j = 1 to n do
    {
      d[i, j] = w[i, j];
    }
  for k = 1 to n do      // Compute D^(k) from D^(k-1)
    for i = 1 to n do
      for j = 1 to n do
        if (d[i, k] + d[k, j] < d[i, j])
        {
          d[i, j] = d[i, k] + d[k, j];
        }
  return d[1..n, 1..n];
}
```

Shortest-Path using Floyd–Warshall: Example



k = 0	j = 1	2	3	4
i = 1	0	∞	−2	∞
2	4	0	3	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 1	j = 1	2	3	4
i = 1	0	∞	−2	∞
2	4	0	**2**	∞
3	∞	∞	0	2
4	∞	−1	∞	0

k = 2	j = 1	2	3	4
i = 1	0	∞	−2	∞
2	4	0	2	∞
3	∞	∞	0	2
4	**3**	−1	**1**	0

k = 3	j = 1	2	3	4
i = 1	0	∞	−2	**0**
2	4	0	2	**4**
3	∞	∞	0	2
4	3	−1	1	0

k = 4	j = 1	2	3	4
i = 1	0	**−1**	−2	0
2	4	0	2	4
3	**5**	**1**	0	2
4	3	−1	1	0

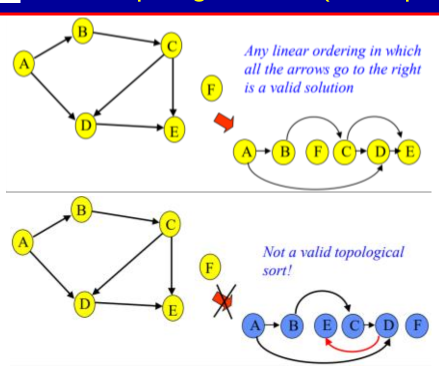
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.100

Topological Sort

- Directed Acyclic Graph **(DAG)**
  - A directed acyclic graph is a directed graph with no cycles.
  - They are often used to represent dependence constraints of some type.
- Topological Sort of a DAG
  - The topological sort a DAG (V, E) is a total ordering, $v_1 < v_2 \ldots < v_n$ of the vertices in V such that for any edge $(v_i, v_j) \in E$, if $j > i$.
  - Topological Sort is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.
  - There may exist multiple different topological orderings for a given DAG.

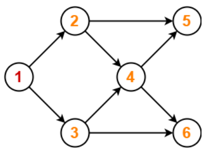
© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.101

Topological Sort (Examples)



© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.102

Topological Sort (contd…)



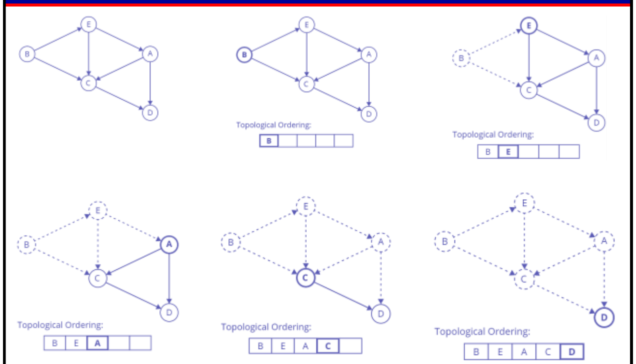
Topo Sorts

1 2 3 4 5 6
1 2 3 4 6 5
1 3 2 4 5 6
1 3 2 4 6 5

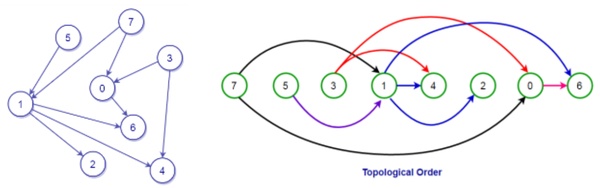
Steps to Find Topological Sort from DAG

1. Identify vertices that have no incoming edge, and select one such vertex.
   - In-degrees of these vertices is zero.
   - If no such edges, graph has cycles (cyclic graph).
2. Delete this vertex of in-degree zero and all its outgoing edges from the graph.
   - Place the deleted vertex in the output.
3. Repeat Steps 1 and Step 2 until graph is empty.

Example: Topological Sort from a DAG



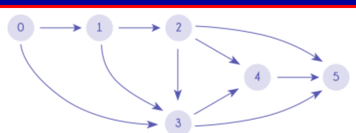
Example: Topological Sort from a DAG



Implementation of Topo Sort using Queue

1. Initialize a queue with each vertex's in-degree.
2. While there are vertices remaining in the queue:
   a) Dequeue and output a vertex.
   b) Reduce in-degree of all vertices adjacent to it by 1.

Ex.: Topo Sort Implementation using Queue



In-degree (Queue)

0	1	2	3	4	5
0	1	2	2	2	3

Output:

Dequeue(0) → **In-degree (Queue)**

0	1	2	3	4	5
0	0	1	1	2	3

Output: 0

Dequeue(1) → **In-degree (Queue)**

0	1	2	3	4	5
0	0	0	1	1	3

Output: 0, 1

Dequeue(2) → **In-degree (Queue)**

0	1	2	3	4	5
0	0	0	0	1	2

Output: 0, 1, 2

Dequeue(3) → **In-degree (Queue)**

0	1	2	3	4	5
0	0	0	0	0	1

Output: 0, 1, 2, 3

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh

Ex.: Topo Sort Implementation (contd…)

Dequeue(4) →

In-degree (Queue)

0	1	2	3	4	5
0	0	0	0	0	0

Output: 0, 1, 2, 3, 4

Dequeue(5) →

In-degree (Queue)

0	1	2	3	4	5
0	0	0	0	0	0

Output: 0, 1, 2, 3, 4, 5

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.109

Bibliography

- E. Horowitz and S. Sahani, "Fundamentals of Data Structures in C"
- Mark Allen Weiss, "Data Structures and Algorithm Analysis in C"
- R. S. Salaria, "Data Structure & Algorithms Using C"
- Schaum's Outline Series, "Data Structure"

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Sunil Pratap Singh U3.110