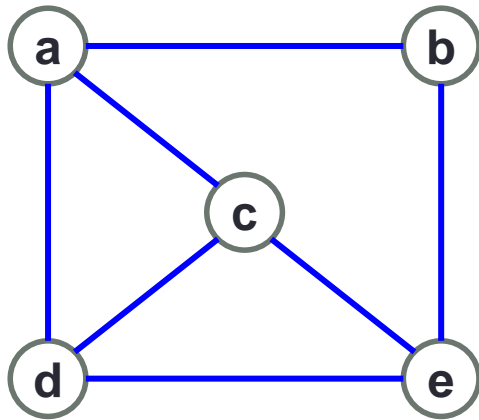


CSN 102: DATA STRUCTURES

Introduction to Course, Classification of Data Structure

What is Graph?

- Graph $G=(V,E)$ is composed of
 - V = set of vertices
 - E = set of edges between vertices
- A vertex (v_i) is a node in the graph
- An edge ($e = (v_i, v_j)$) is a pair of vertices



$V = \{a,b,c,d,e\}$

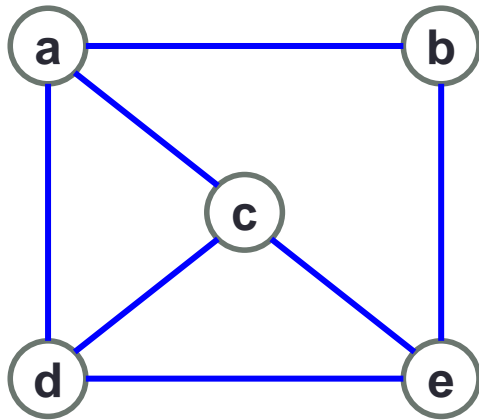
$E = \{(a,b), (b,e), (e,d), (d,a), (a,c), (c,d), (c,e), (c,b)\}$

Applications of Graph

- To represent electric circuits
- To represent networks (cities, flights, communications)
- Once a network of circuit is modelled as graph where node showing the entities and edges showing the relation between entities, known properties and algorithms can be applied
- Eg. Modelling flights between cities as graph, one can find shortest path between any two cities using known shortest path algorithms

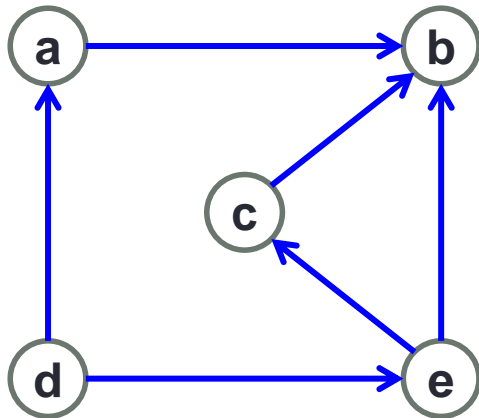
Undirected and Directed Graph

- Undirected Graph: when an edge between vertices is bidirectional. Edge (u,v) is also (v,u)
- Eg. Network of roads between cities



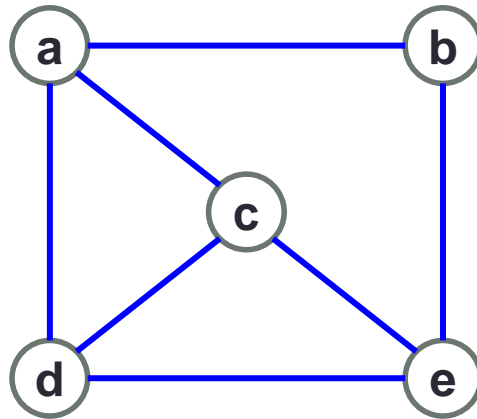
Undirected and Directed Graph(2)

- Directed Graph: Edge also shows the direction between vertices. Edges are **ordered** pair. Edge (u,v) means “u” is source and “v” is destination
- Eg. Network of water supply



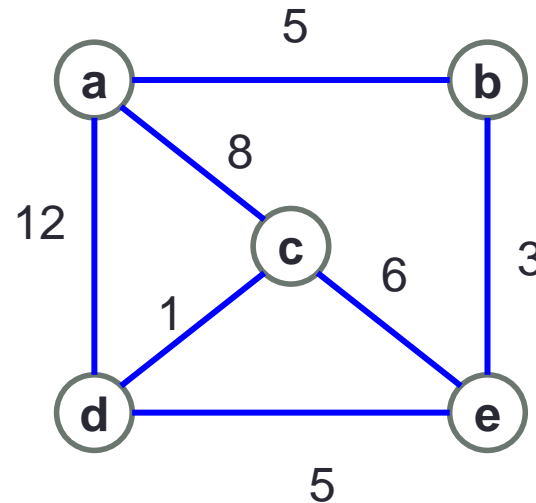
Un-Weighted and Weighted Graph

- Un-weighted Graph: When weight/cost of edges is not specified. Default weight is considered to be 1.



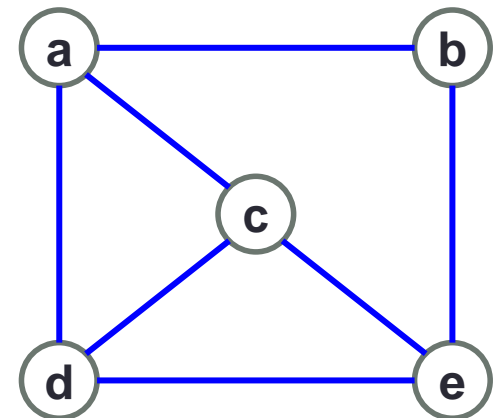
Un-Weighted and Weighted Graph(2)

- Weighted Graph: When each edge has a associated cost/weight. Travelling an edge results in it's cost added to total cost.



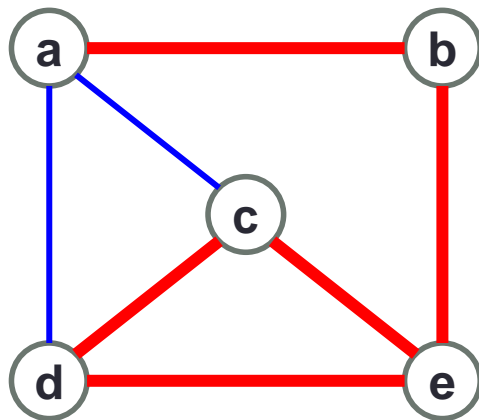
Graph Terminology

- Adjacent vertices: vertices connected by an edge
- Eg. $\text{Adjacent}(a) = \{b, c, d\}$
- Degree of a vertex: No. of adjacent vertices
- Eg. $\text{Degree}(a) = 3$
- Sum of degrees of all vertices
= $2 \times \text{number of edges}$
= $2 \times e$; where $e = |E|$

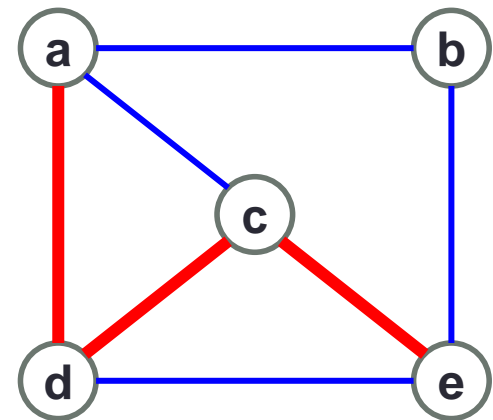


Graph Terminology(cont'd)

- Path: Sequence of vertices v_1, v_2, \dots, v_k such that there is an edge between two consecutive vertices



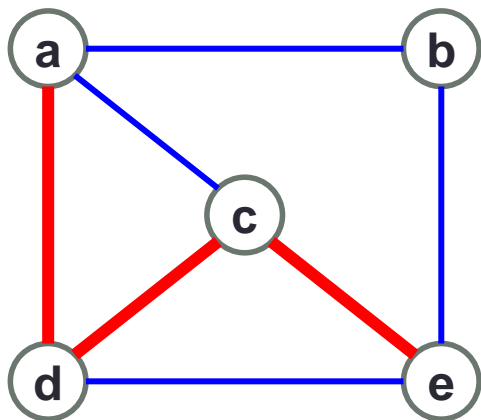
a b e c d e



a d c e

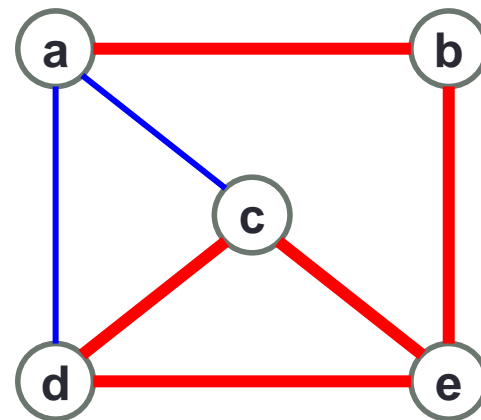
Graph Terminology(cont'd)

- Simple Path: a path with no repeated vertices



a d c e

YES

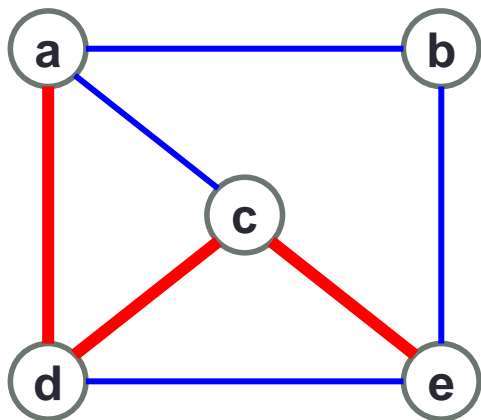


a b e c d e

NO

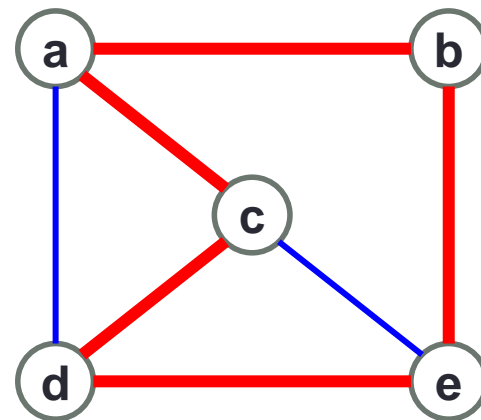
Graph Terminology(cont'd)

- Cycle: simple path with same start and last vertex



a d c e

NO

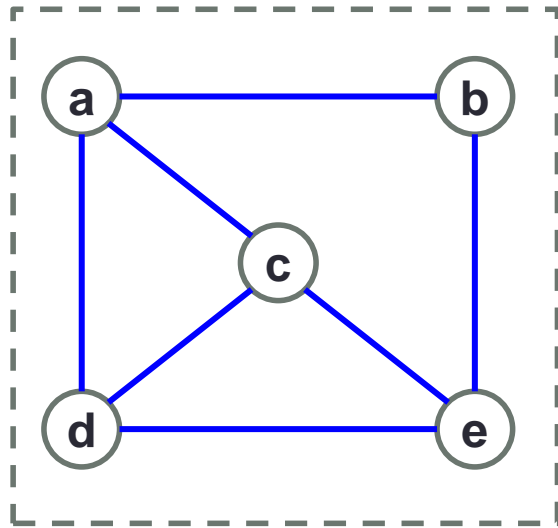


a b e d c a

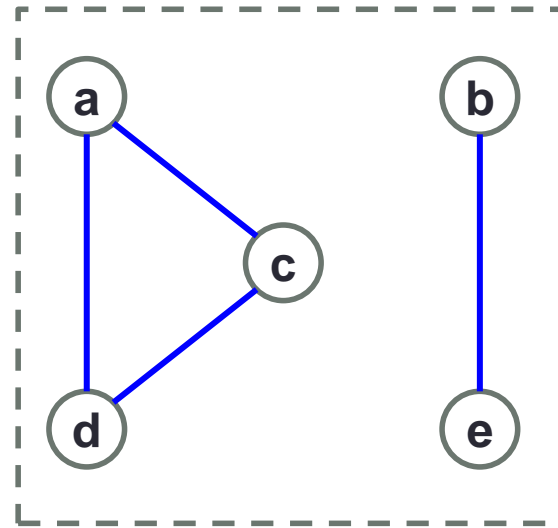
Yes

Graph Terminology(cont'd)

- Connected Graph: any two vertices are connected through some path



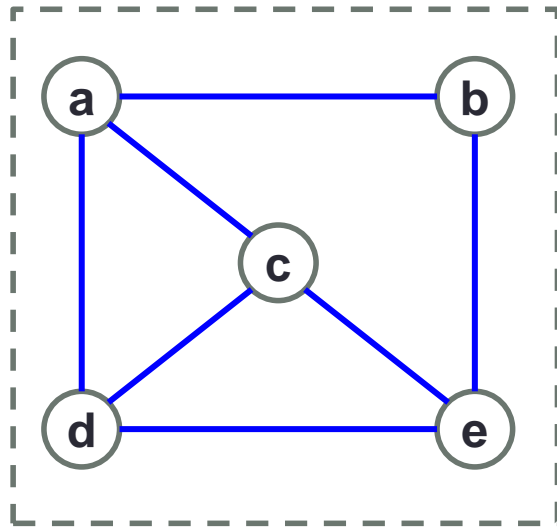
Connected



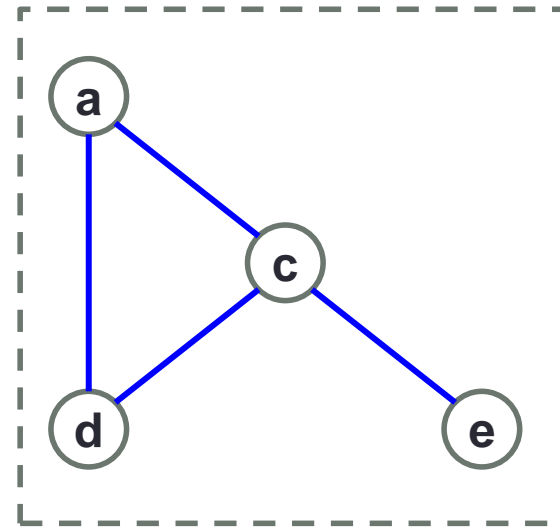
Not Connected

Graph Terminology(cont'd)

- Subgraph: subset of vertices and edges forming a graph



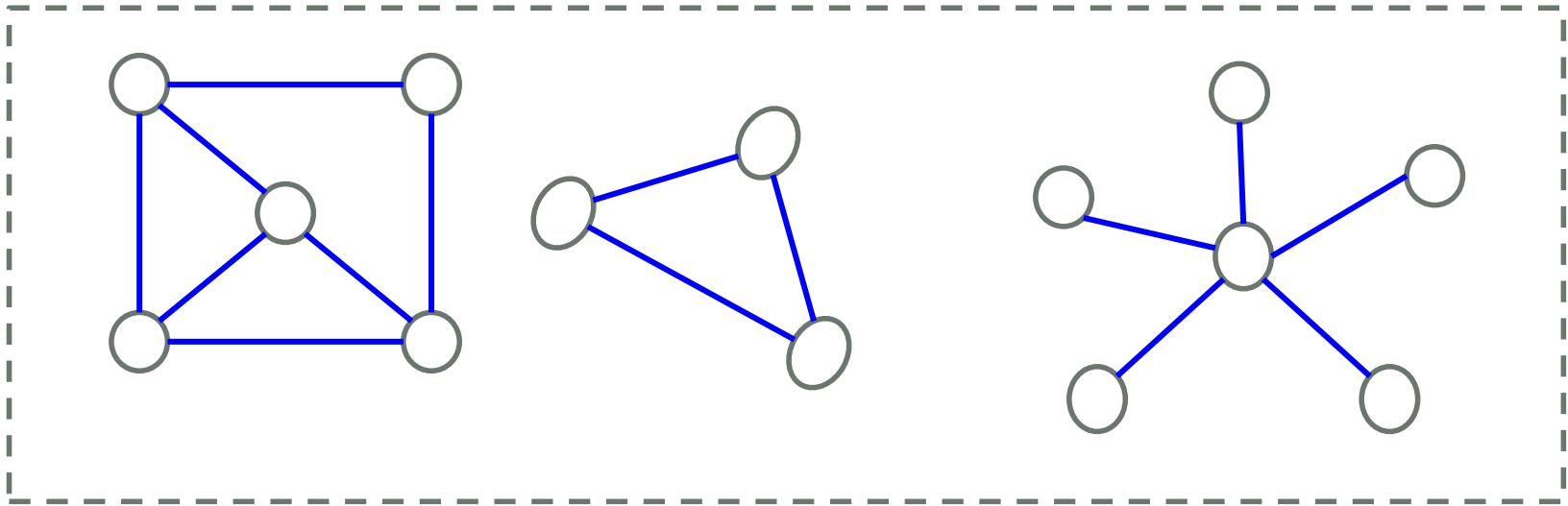
G



G' Subgraph of G

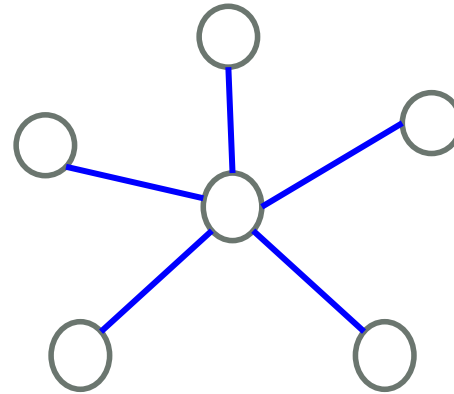
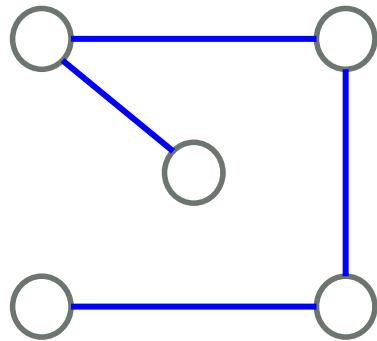
Graph Terminology(cont'd)

- Connected Components: maximal connected subgraph
- Eg. Below graph has 3 connected components



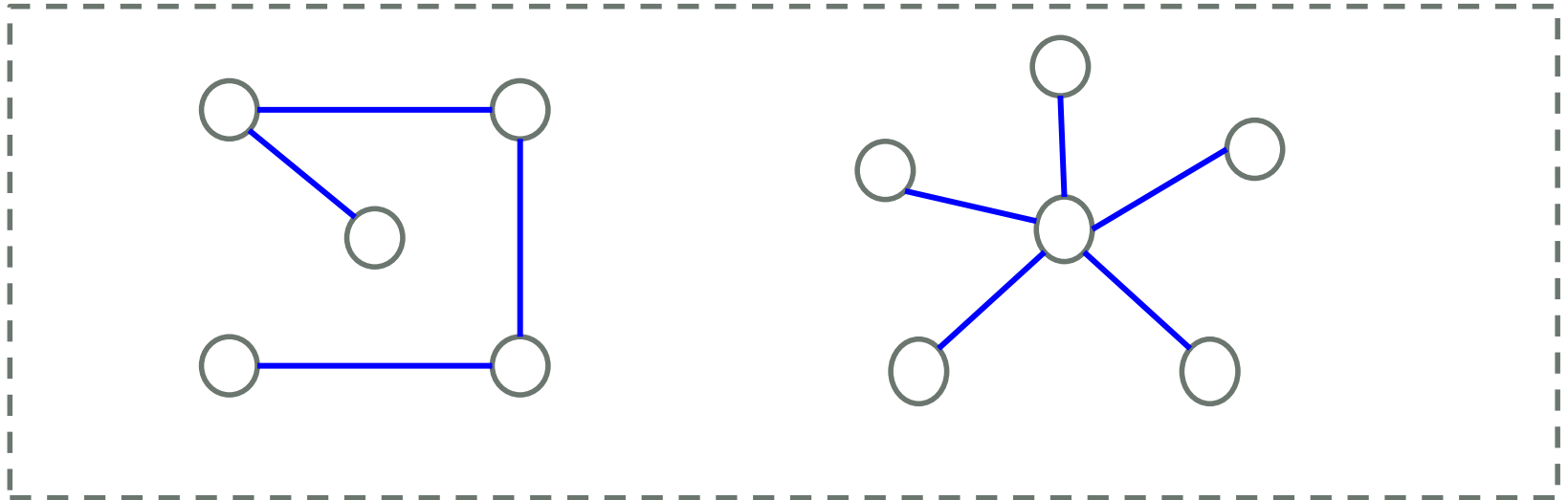
Graph Terminology(cont'd)

- (free)Tree: connected graph without cycles



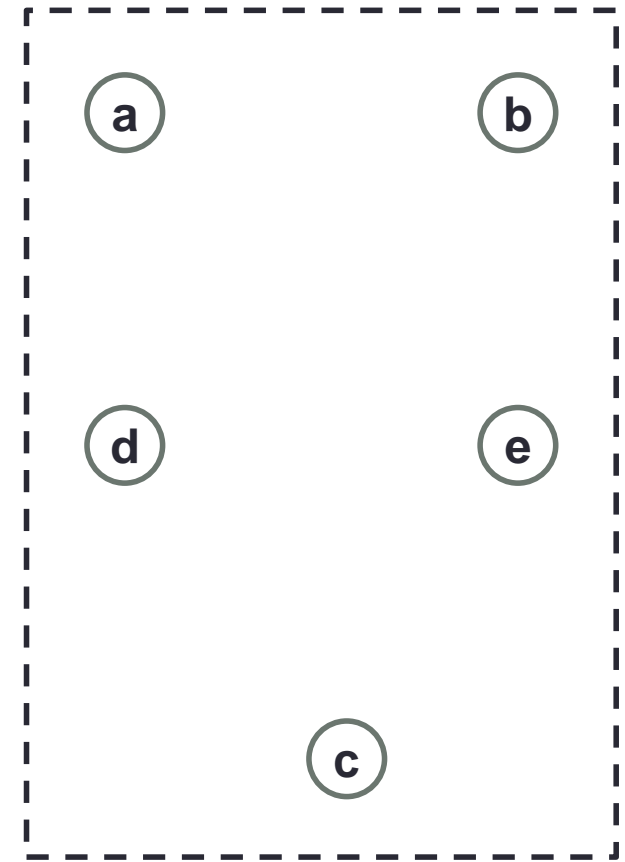
Graph Terminology(cont'd)

- (free)Tree: connected graph without cycles
- Forest: collection of tree



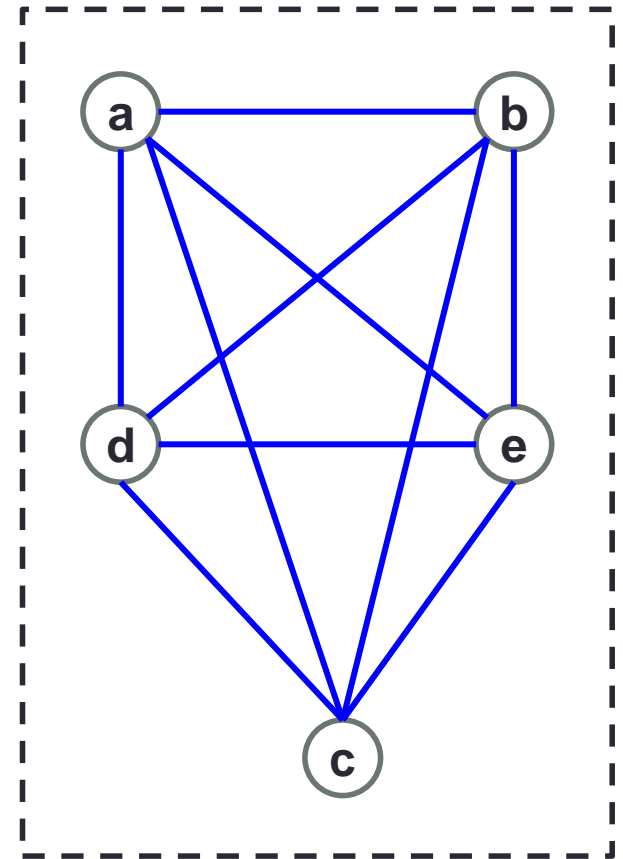
Connectivity in Graph

- If $m = \text{\#edges}$ and $n = \text{\#vertices}$, then
 - Minimum number of edges possible: $m=0$



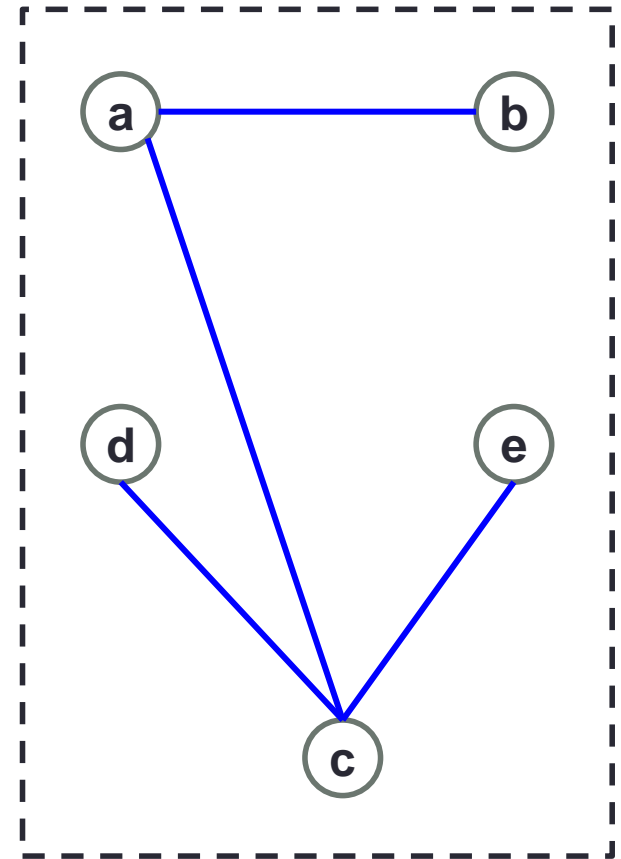
Connectivity in Graph

- If $m = \text{\#edges}$ and $n = \text{\#vertices}$, then
 - Minimum number of edges possible: $m=0$
 - Maximum number of edges possible:
 $m = n(n-1)/2$ i.e. every vertex is adjacent to every other vertex.
 - Complete graph: each pair of vertices are adjacent



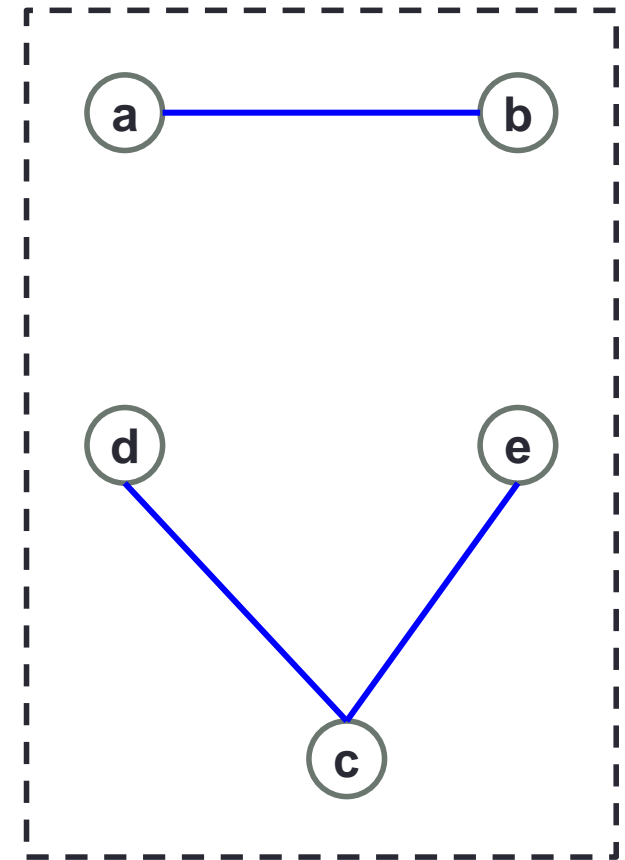
Connectivity in Graph

- If $m = \text{\#edges}$ and $n = \text{\#vertices}$, then
 - Minimum number of edges possible: $m=0$
 - Maximum number of edges possible:
 $m = n(n-1)/2$ i.e. every vertex is adjacent to every other vertex
 - Complete graph: each pair of vertices are adjacent
 - Number of edges for a tree: $m=n-1$



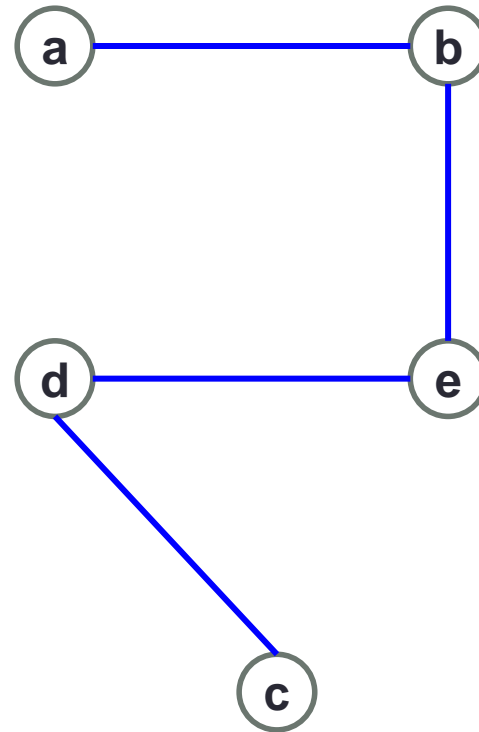
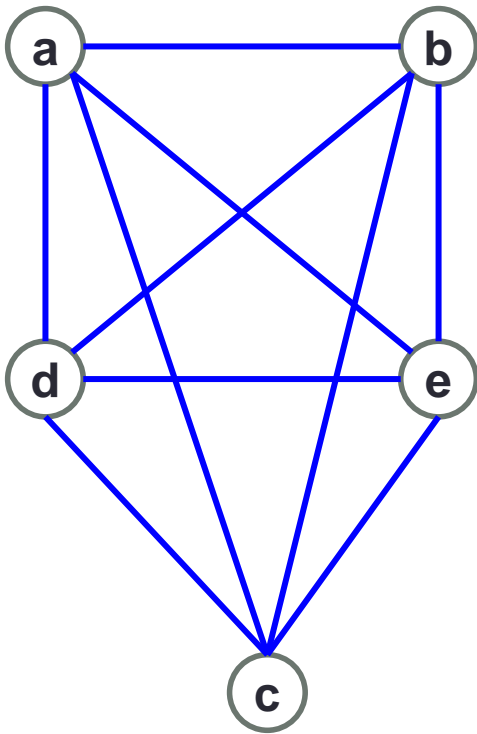
Connectivity in Graph

- If $m = \text{\#edges}$ and $n = \text{\#vertices}$, then
 - Minimum number of edges possible: $m=0$
 - Maximum number of edges possible:
 $m = n(n-1)/2$ i.e. every vertex is adjacent to every other vertex
 - Complete graph: each pair of vertices are adjacent
 - Number of edges for a tree: $m=n-1$
 - If $m < n-1$ then graph is not connected
 - If $m=n-k$, then graph will $\geq k$ connected component(s)



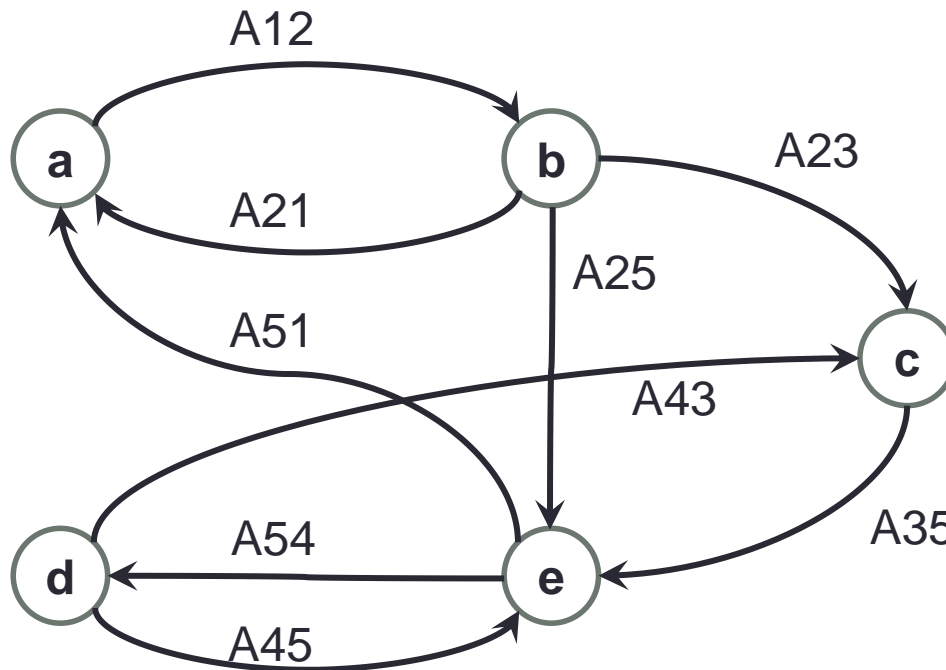
Spanning Tree

- A spanning tree of graph G is a subgraph which is a **tree** and has **all vertices of G**



A Sample Graph

- Consider the given graph. We will represent this graph using various techniques.

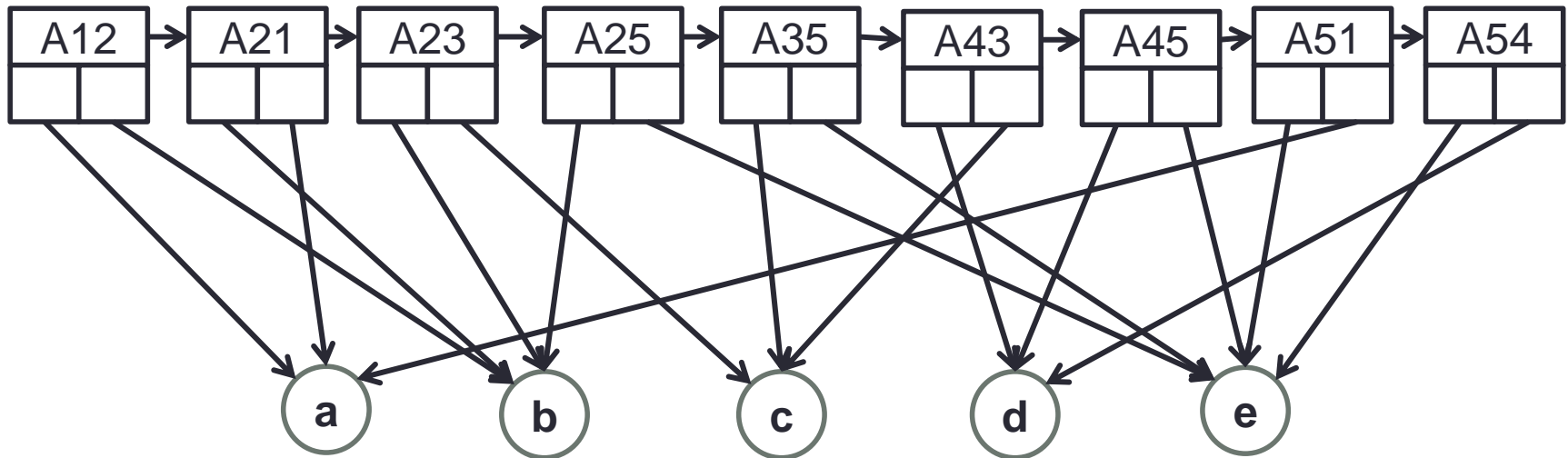


Representation of Graph

- Edge List
- Adjacency List
- Adjacency Matrix

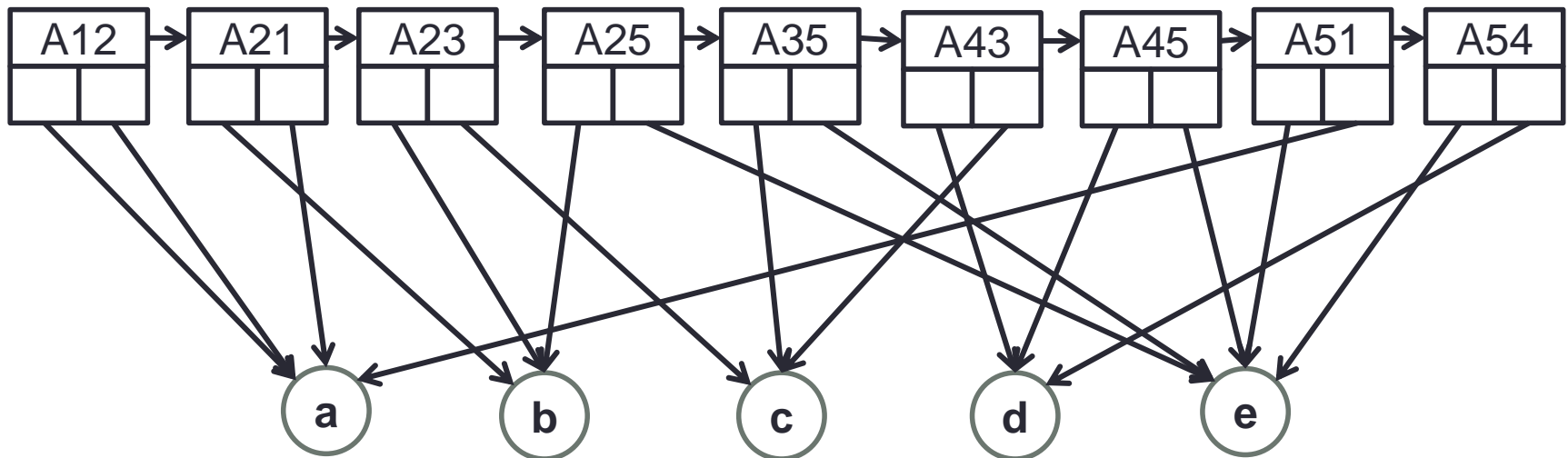
Edge List

- A list of all the edges where each node in list stores some info about the edge
- Each node also stores link to the two vertices and a link to next edge in the list
- Space required: $\Theta(m+n)$



Edge List(cont'd)

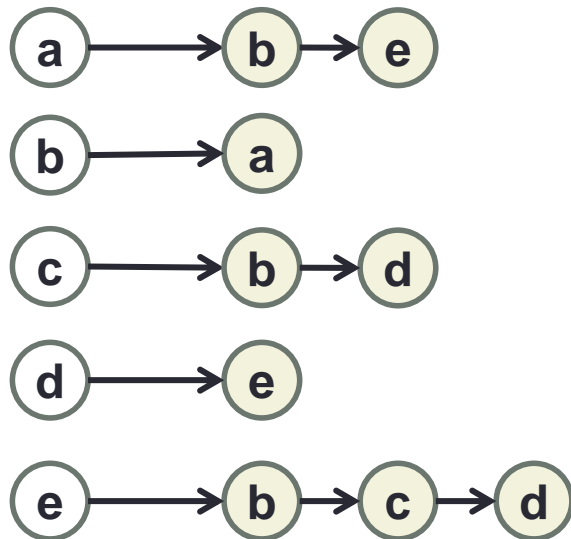
- $O(1)$: insert edge/vertex, both vertices, source, destination
- $O(m)$: edges incidenting on a vertex, number of edges, removing vertex



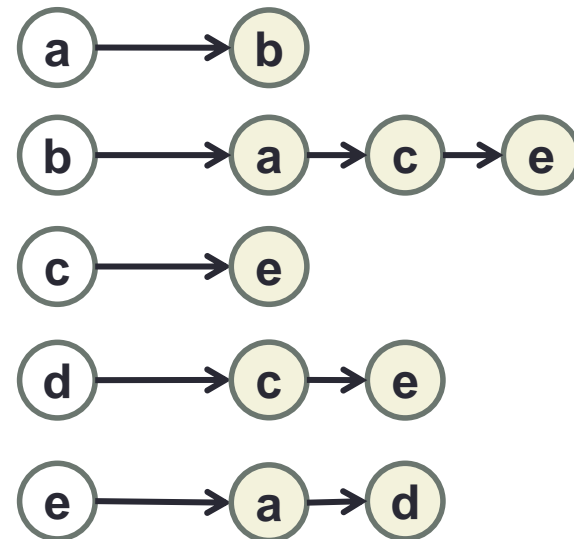
Adjacency List

- For each vertex, store list of adjacent vertices
- If directed graph(digraph), store two list for “in” and “out” edges separately
- Space complexity: $\Theta(n + \sum \text{degree}(v)) = \Theta(n + m)$

IN



OUT

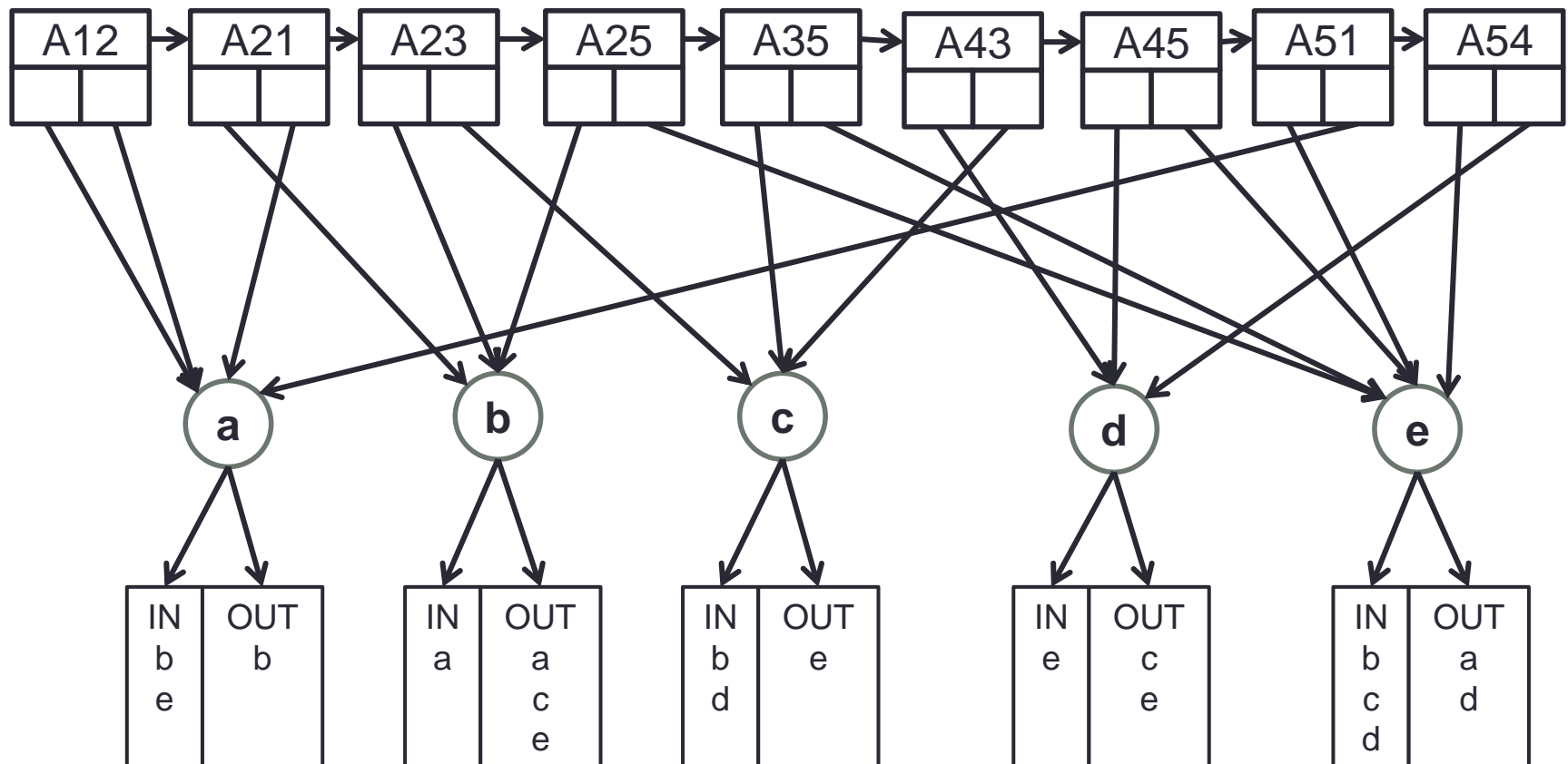


Adjacency List(cont'd)

- Some operations like finding adjacent vertices, degree of a vertex can be done in constant time
- But, as we do not store information about edges, operations like end points of an edge is not possible.
- Eg. end points of edge A12.

Modern Adjacency List

- Extend Edge List to store Adjacency list at each vertex



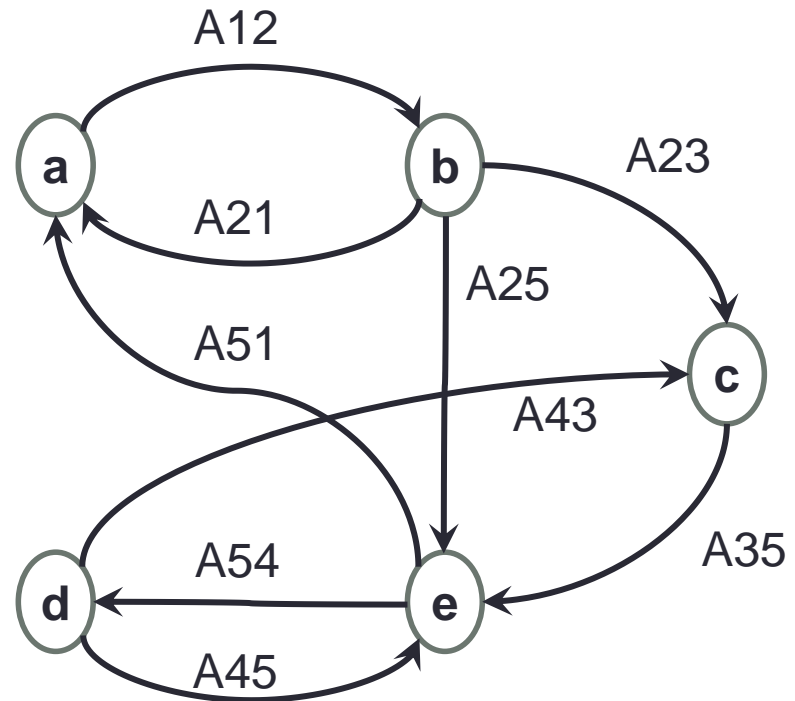
Modern Adjacency List(cont'd)

- Space Complexity: $O(n+m)$
- Finding adjacent vertex is $O(1)$, but operations like finding in/out edges from a vertex is expensive
- If we store list of incident edges in adjacency list, then can reduce time complexity of finding in/out edges from vertex
- Therefore, all such decision depends on which operations we want to optimize

Adjacency Matrix

- An “n x n” matrix where n is #vertex.
- $M[i,j] = 1$ if there is an edge from vertex i to j, else 0
- Space complexity = $\Theta(n^2)$

	a	b	c	d	e
a	0	1	0	0	0
b	1	0	1	0	1
c	0	0	0	0	1
d	0	0	1	0	1
e	1	0	0	1	0



Modern Adjacency Matrix

- Modify Adjacency matrix representation such that each row and column corresponds to a vertex and matrix stores Edge information

	0	1	2	3	4
0	0	A12	0	0	0
1	A21	0	A23	0	A25
2	0	0	0	0	A35
3	0	0	A43	0	A45
4	A51	0	0	A54	0



Breadth First Search(BFS)

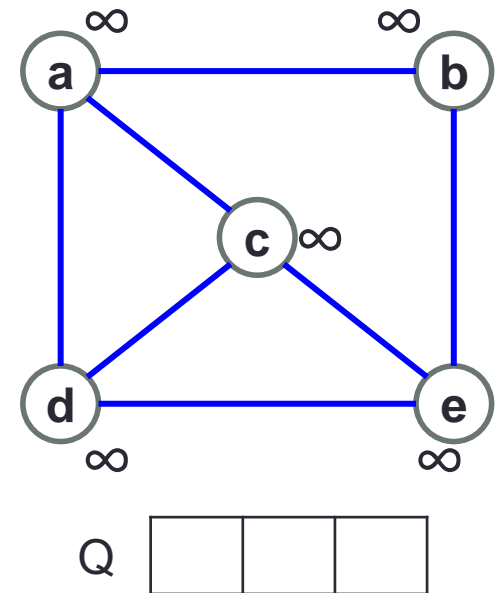
- Starting from a vertex(source), visit all the nodes which are directly connected. Repeat the process of visiting remaining vertices from the recently visited vertices
- BFS produces spanning tree
- Uses of BFS:
 - Determines whether graph is connected or not
 - Can determine the shortest distance from source for every vertex

BFS : Algo

- Initialize all the vertex with label ∞ and predecessor NULL
- Enqueue source vertex in queue “Q” and set label 0
- In each step:
 - Dequeue from Q
 - For each adjacent vertex of dequeued vertex
if label = ∞ then
 label = label of dequeued vertex + 1
 predecessor = dequeued vertex
 - Repeat until Q is empty

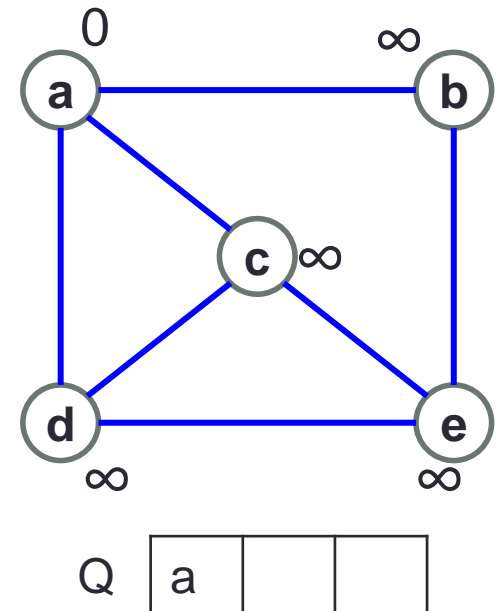
BFS : Example

1. Initialize vertices with label ∞



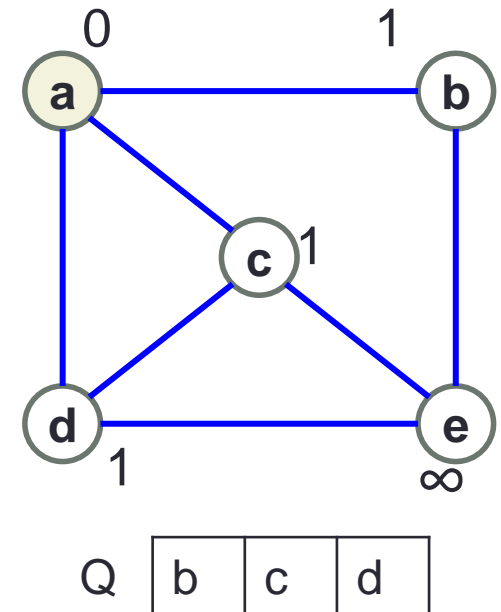
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a



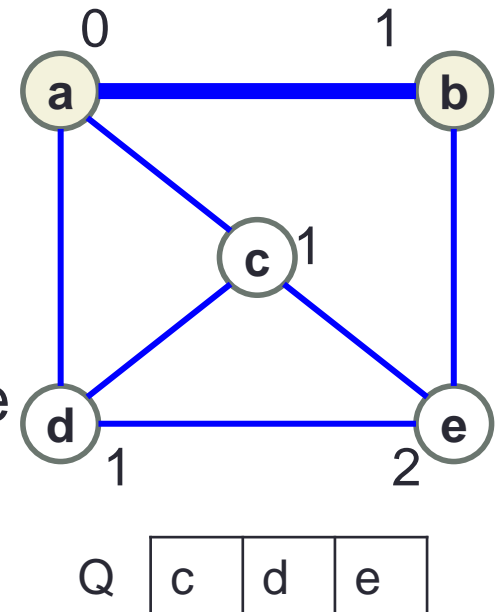
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue



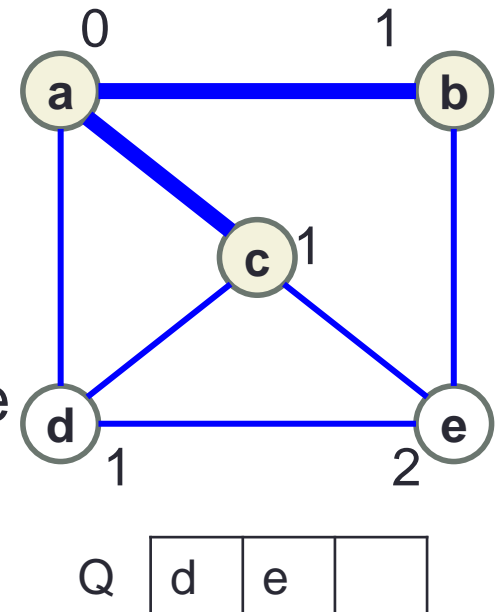
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue
5. dequeue Q, find adjacent of b
6. Ignore a. set label for e 2 and enqueue



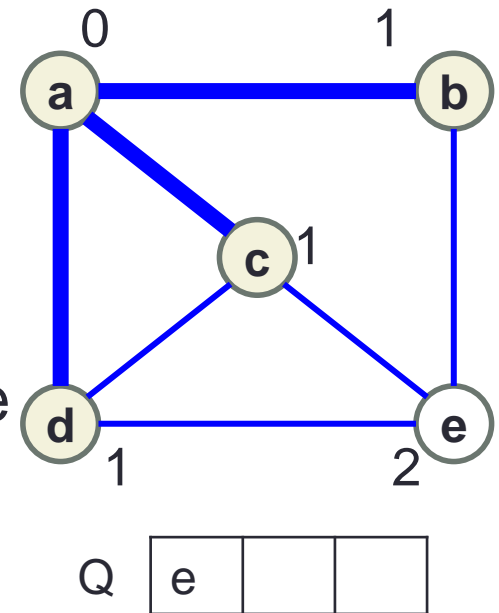
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue
5. dequeue Q, find adjacent of b
6. Ignore a. set label for e 2 and enqueue
7. Dequeue Q, find adjacent of c
8. Ignore all as all are visited



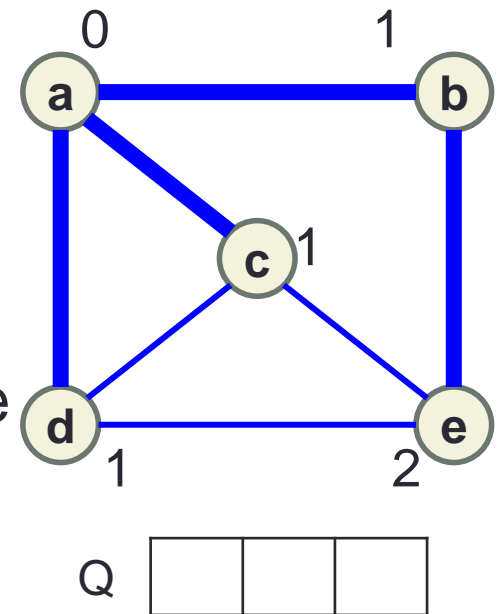
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue
5. dequeue Q, find adjacent of b
6. Ignore a. set label for e 2 and enqueue
7. Dequeue Q, find adjacent of c
8. Ignore all as all are visited
9. Dequeue Q, find adjacent of d. Ignore all.



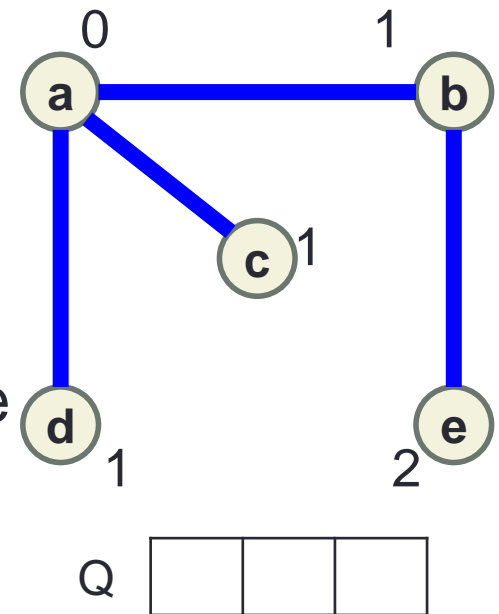
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue
5. dequeue Q, find adjacent of b
6. Ignore a. set label for e 2 and enqueue
7. Dequeue Q, find adjacent of c
8. Ignore all as all are visited
9. Dequeue Q, find adjacent of d. Ignore all.
10. Dequeue Q, find adjacent of e. Ignore all.



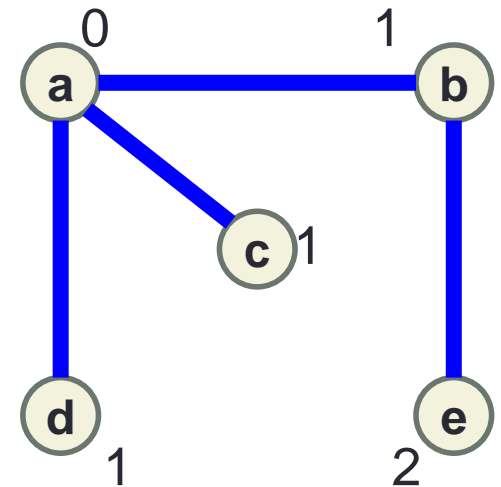
BFS : Example

1. Initialize vertices with label ∞
2. Set label 0 and enqueue a
3. dequeue Q, find adjacent of a
4. Set label for b,c,d 1 and enqueue
5. dequeue Q, find adjacent of b
6. Ignore a. set label for e 2 and enqueue
7. Dequeue Q, find adjacent of c
8. Ignore all as all are visited
9. Dequeue Q, find adjacent of d. Ignore all.
10. Dequeue Q, find adjacent of e. Ignore all.
11. Stop as Q is empty



BFS : Example

- The resultant graph is spanning tree
- Every node has shortest distance from source 'a' stored in label



BFS Analysis

- Initializing vertices takes: $O(V)$ or $O(n)$
- All the vertices are examined once only, hence $O(V)$ again
- Assuming we were using adjacency list representation, each time we dequeue a vertex we fetch its adjacent vertex. In total, accessing adjacent vertices for all the vertices will take $O(E)$ or $O(m)$
- Therefore, examining vertices will take $O(V+E)$
- Total time: $O(V) + O(V+E) = O(V+E)$

Depth First Search(DFS)

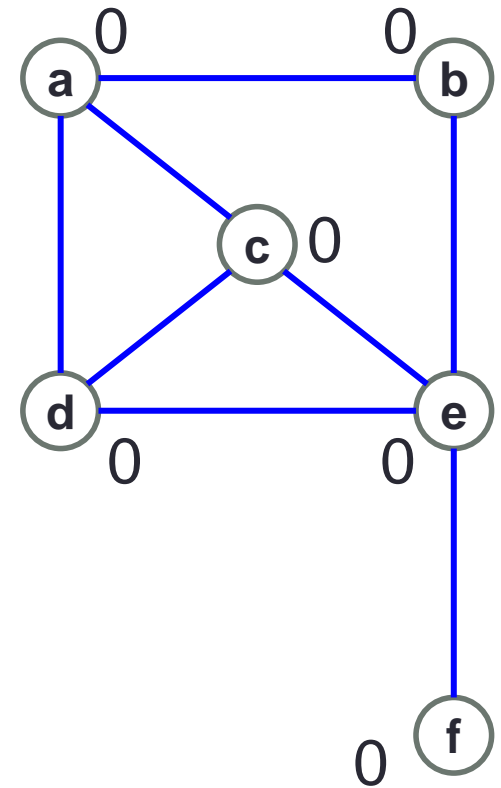
- Similar to Depth First traversal of tree
- Only difference is that graph might have cycle
- Therefore, mark a vertex visited if DFS is applied on it
- DFS also produces spanning tree as output
- DFS traverse each edge and vertex(BFS too)
- DFS application:
 - Finding cycle in the graph
 - All pair shortest path

DFS : Algo

- Initialize label for all vertex as 0(Unvisited)
- Start from any vertex s
- If the label is 0, set it to 1 else ignore
- For all the adjacent vertex of s , recursively perform DFS

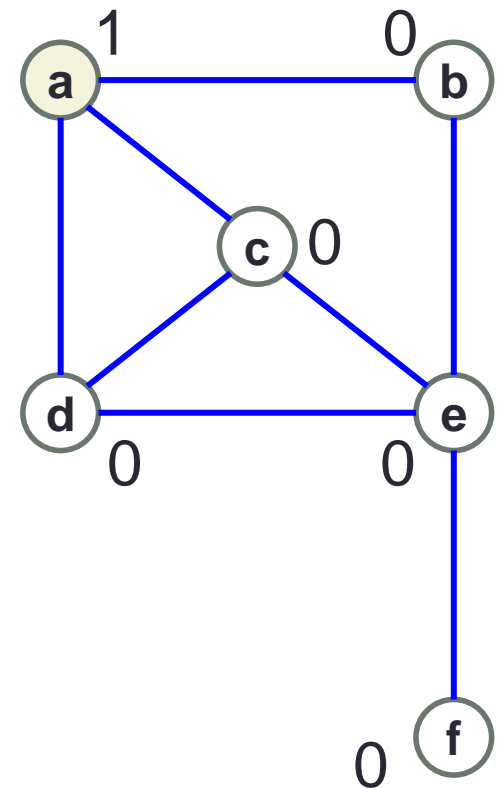
DFS : Example

- Initialize label for each vertex to 0



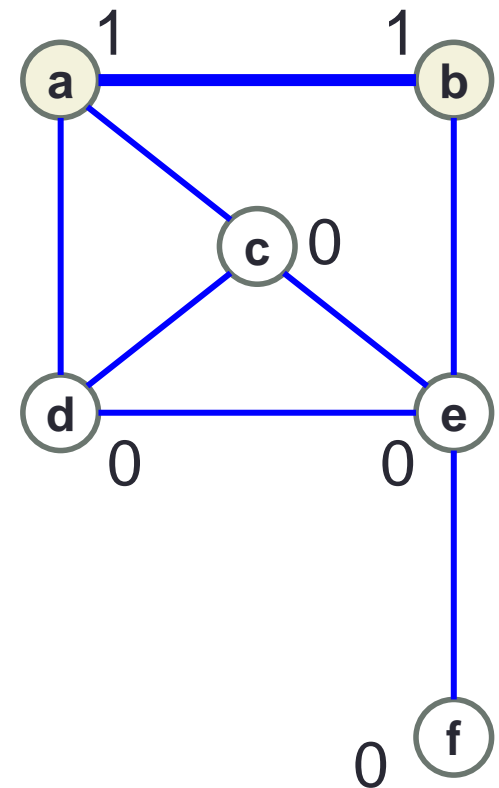
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b



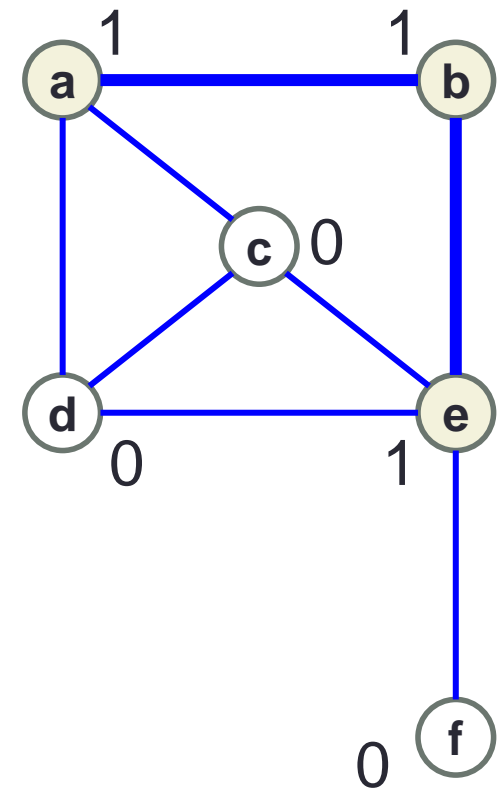
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)



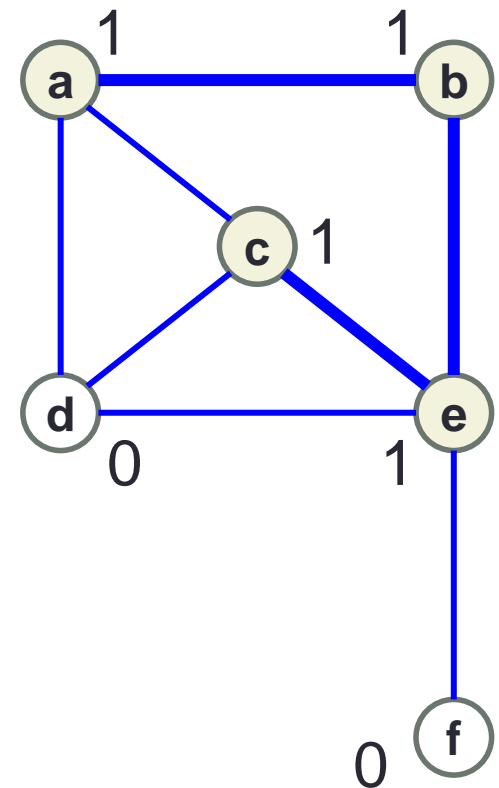
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)
- As $\text{Label}(e) = 0$, therefore mark e visited
- Run DFS(c)



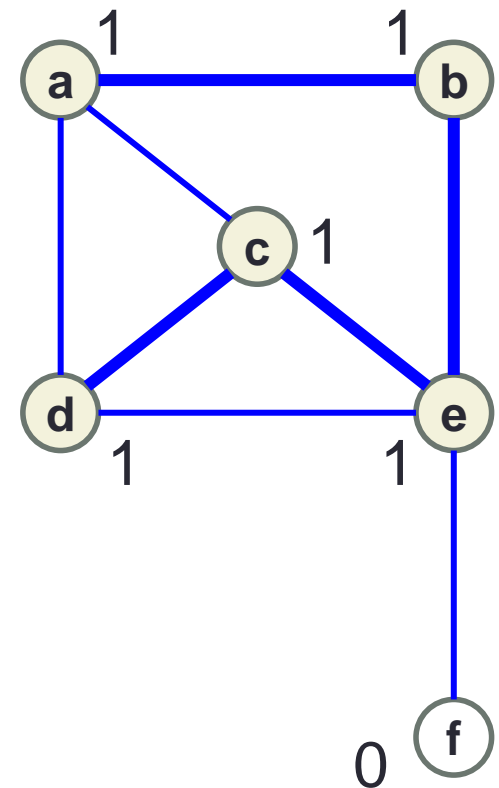
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)
- As $\text{Label}(e) = 0$, therefore mark e visited
- Run DFS(c)
- As $\text{Label}(c) = 0$, mark c visited
- Run DFS(d)



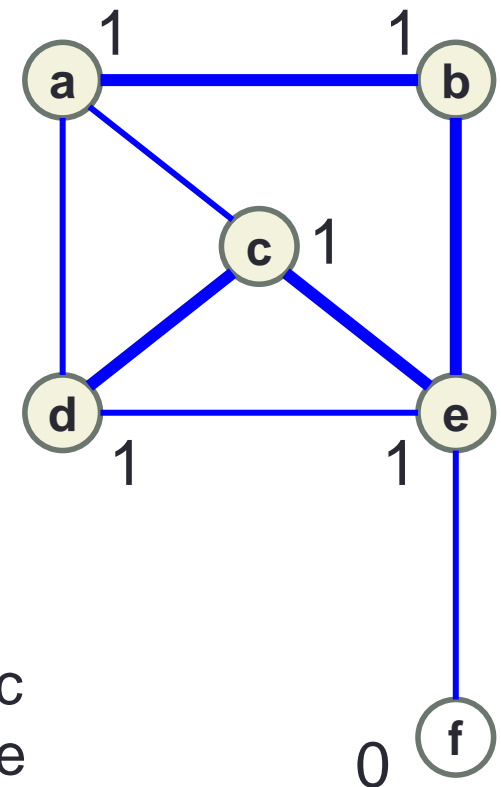
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)
- As $\text{Label}(e) = 0$, therefore mark e visited
- Run DFS(c)
- As $\text{Label}(c) = 0$, mark c visited
- Run DFS(d)
- As $\text{Label}(d) = 0$, mark d visited



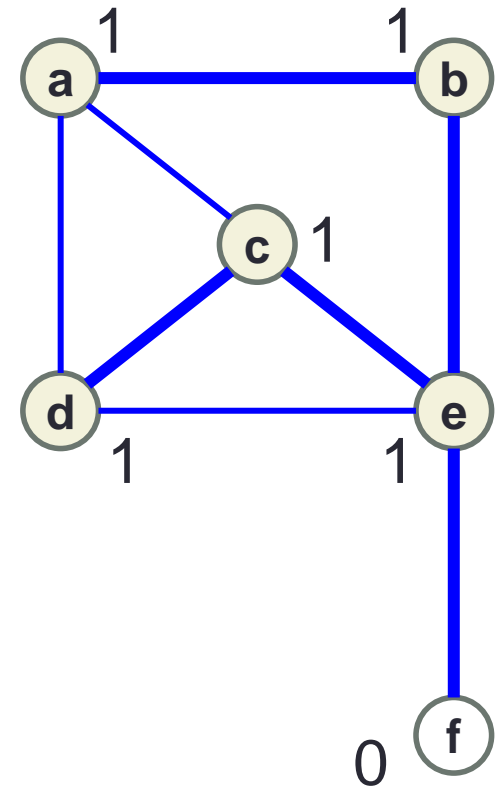
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)
- As $\text{Label}(e) = 0$, therefore mark e visited
- Run DFS(c)
- As $\text{Label}(c) = 0$, mark c visited
- Run DFS(d)
- As $\text{Label}(d) = 0$, mark d visited
- As a and e are already visited, backtrack to c
- As a and e are already visited, backtrack to e
- d is visited. Run DFS(f)



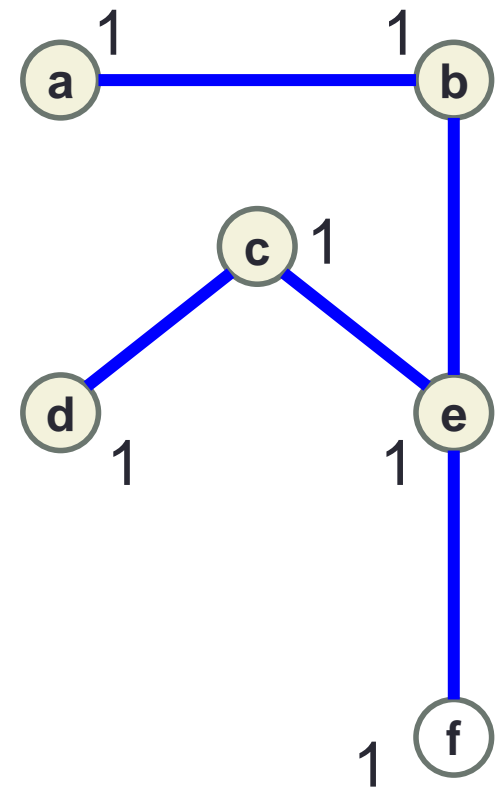
DFS : Example

- Initialize label for each vertex to 0
- Start from vertex a, mark a visited
- Run DFS for adjacent vertex b
- As $\text{Label}(b) = 0$, therefore mark b visited
- Run DFS for adjacent vertex a
- As a is already visited, run DFS(e)
- As $\text{Label}(e) = 0$, therefore mark e visited
- Run DFS(c)
- As $\text{Label}(c) = 0$, mark c visited
- Run DFS(d)
- As $\text{Label}(d) = 0$, mark d visited
- As a and e are already visited, backtrack to c
- As a and e are already visited, backtrack to e
- d is visited. Run DFS(f)
- As $\text{Label}(f) = 0$, mark f visited



DFS : Example(cont'd)

- DFS also produces a spanning tree
- DFS algorithm can be altered to find cycle in a graph and to fit other application



DFS : Analysis

- Initializing vertices : $O(V)$
- In total, we are accessing adjacency list of **every** node **exactly once**. Hence, time required is $O(V+E)$
- Total time required: $O(V)+O(V+E) = O(V+E)$
- If we say that graph is connected, then $V < E$. Thus, $O(E)$

Single Source Shortest Path(SSSP)

- SSSP algorithms find shortest path from single source(vertex) to every other vertex
- For un-weighted graph, we can use BFS to find shortest path to each vertex from a given source.
- For weighted graph, multiple algorithms exists to find the shortest path.
- Dijkstra's SSSP algorithms is one

Dijkstra's SSSP Algo

- This algo is generally used for weighted graphs, though can also be used for un-weighted graphs.
- Precondition for Dijkstra's algo:
 - Graph should not have any negative weight edge

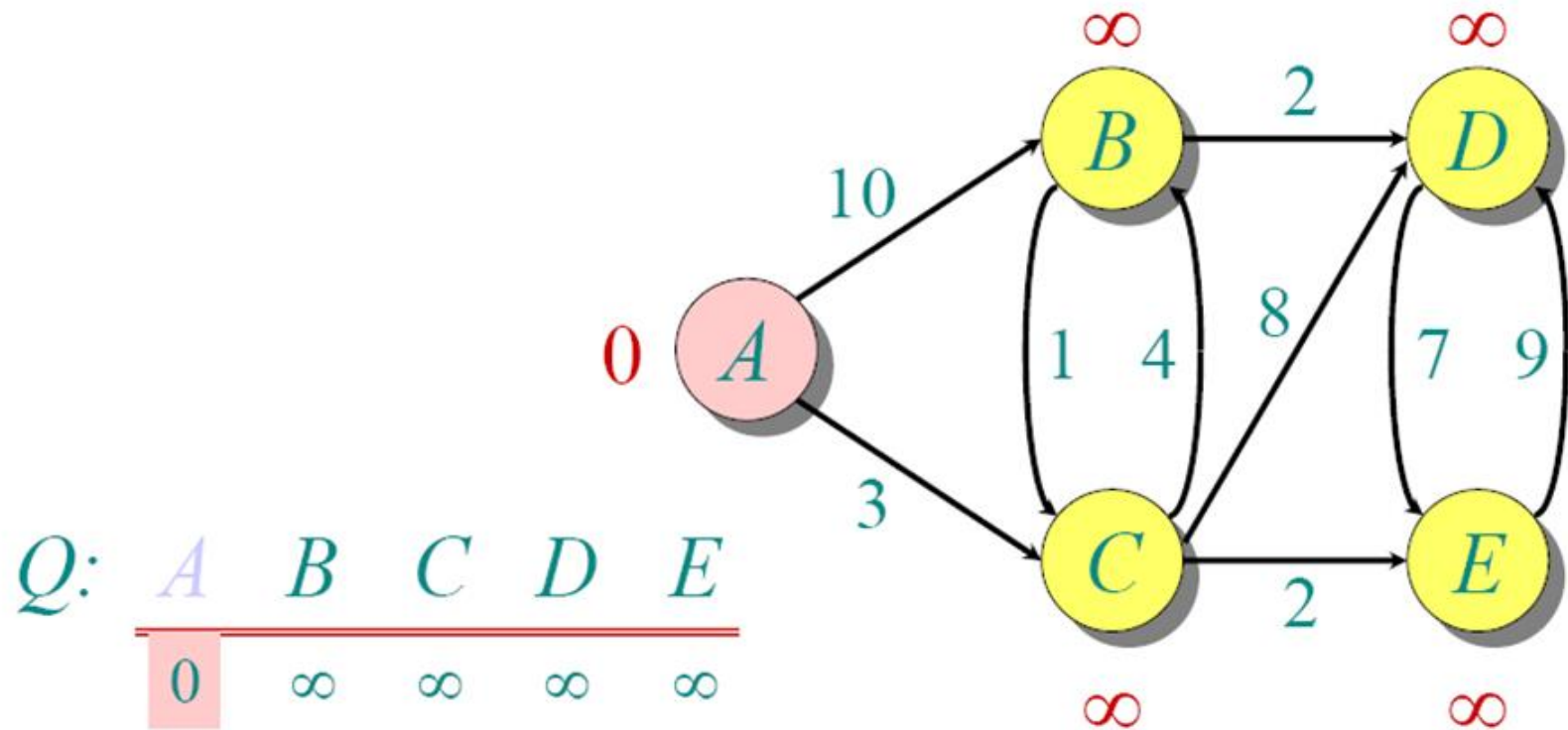
Input: Weighted graph $G=\{E,V\}$ and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

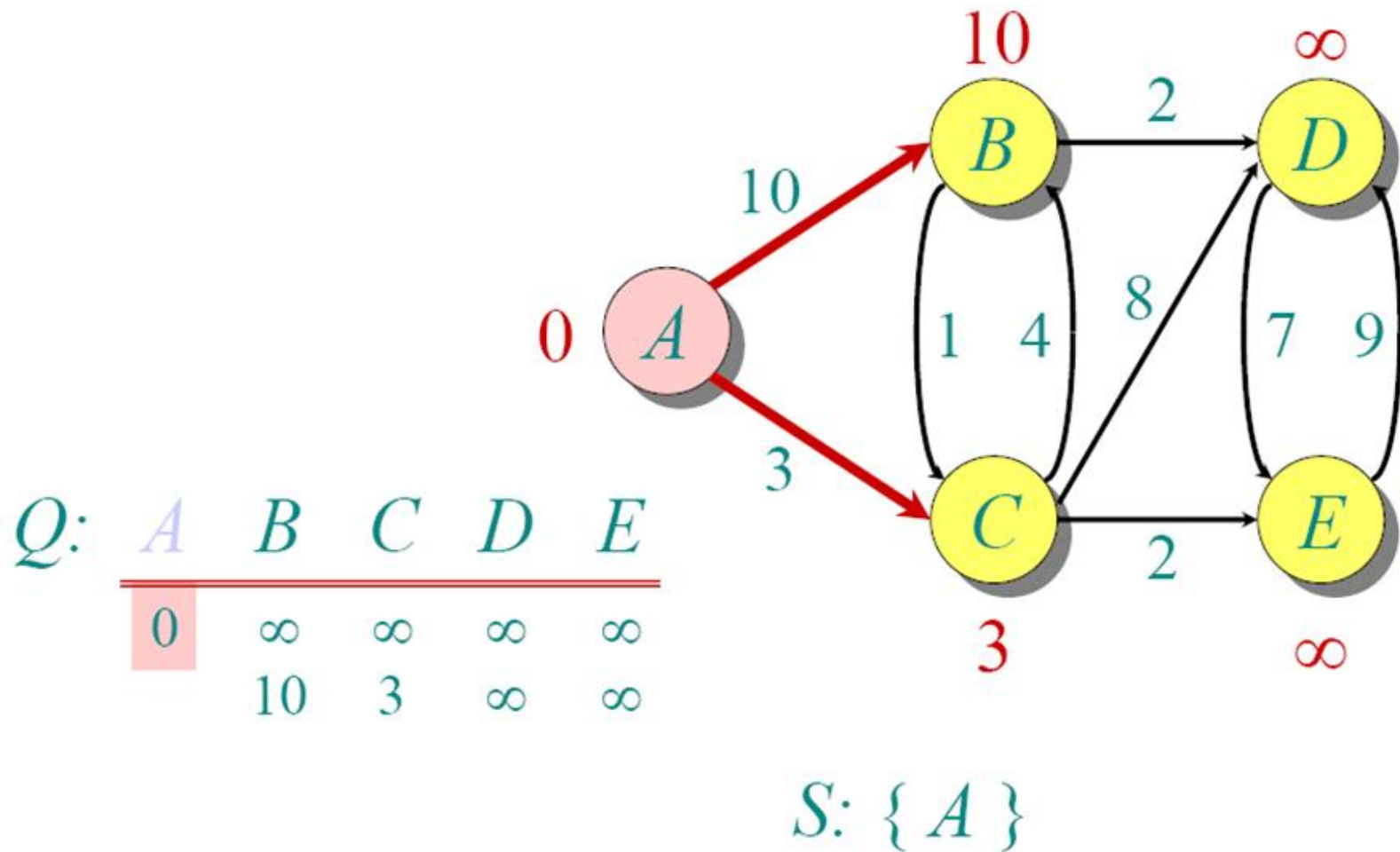
Dijkstra's SSSP Algo(cont'd)

1. Initialize distance label for all vertex to ∞
2. Set distance of source vertex to 0
3. While shortest distance for all vertices are known
 1. Pick vertex with minimum distance label, let v
 2. Mark v as known
 3. For all unknown vertex n adjacent to v
 1. $\text{dist} = D[v] + \text{weight of edge } (v,n)$
 2. If $\text{dist} < D[n]$, then $D[n] = \text{dist}$

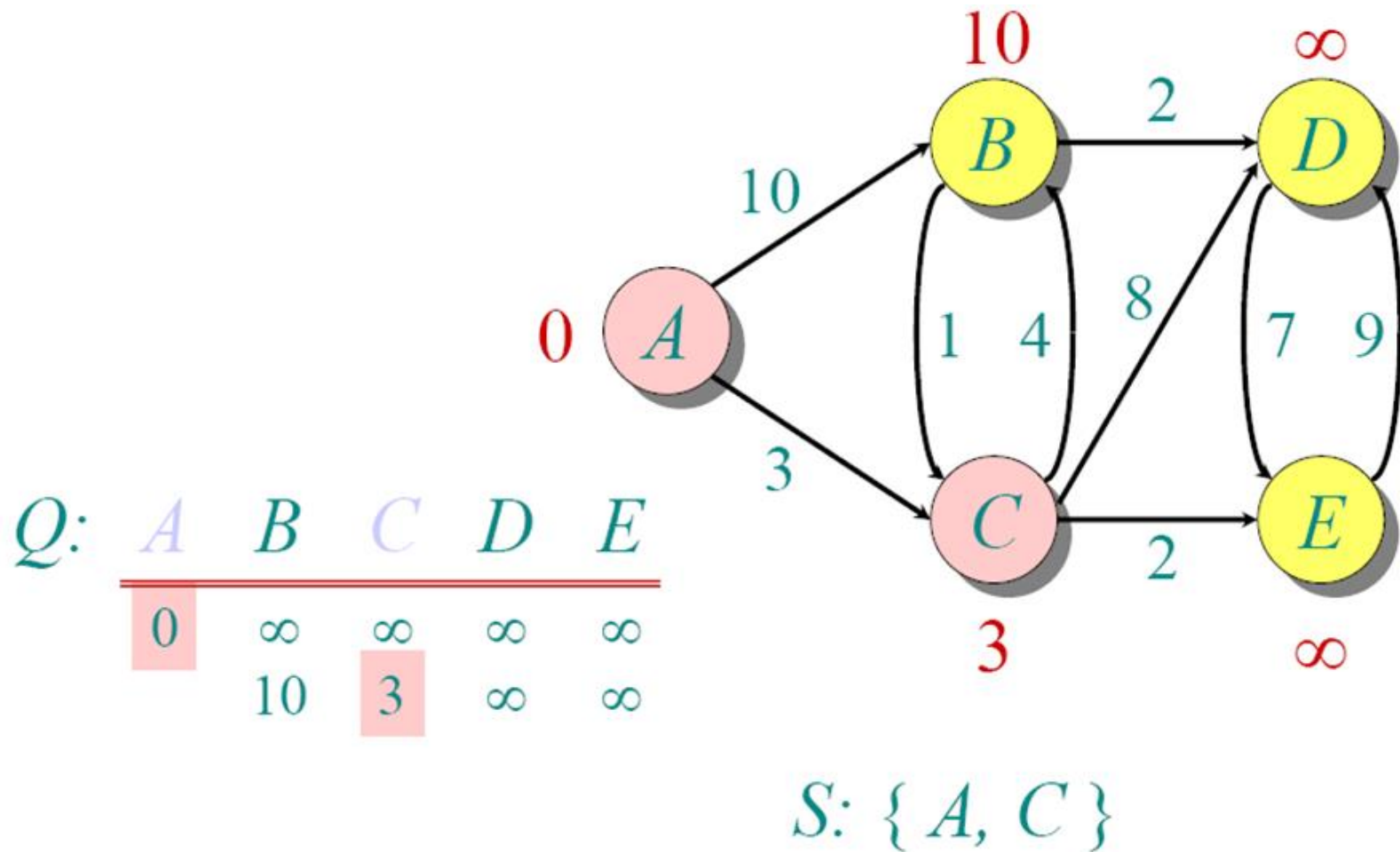
Dijkstra's Algo: Example



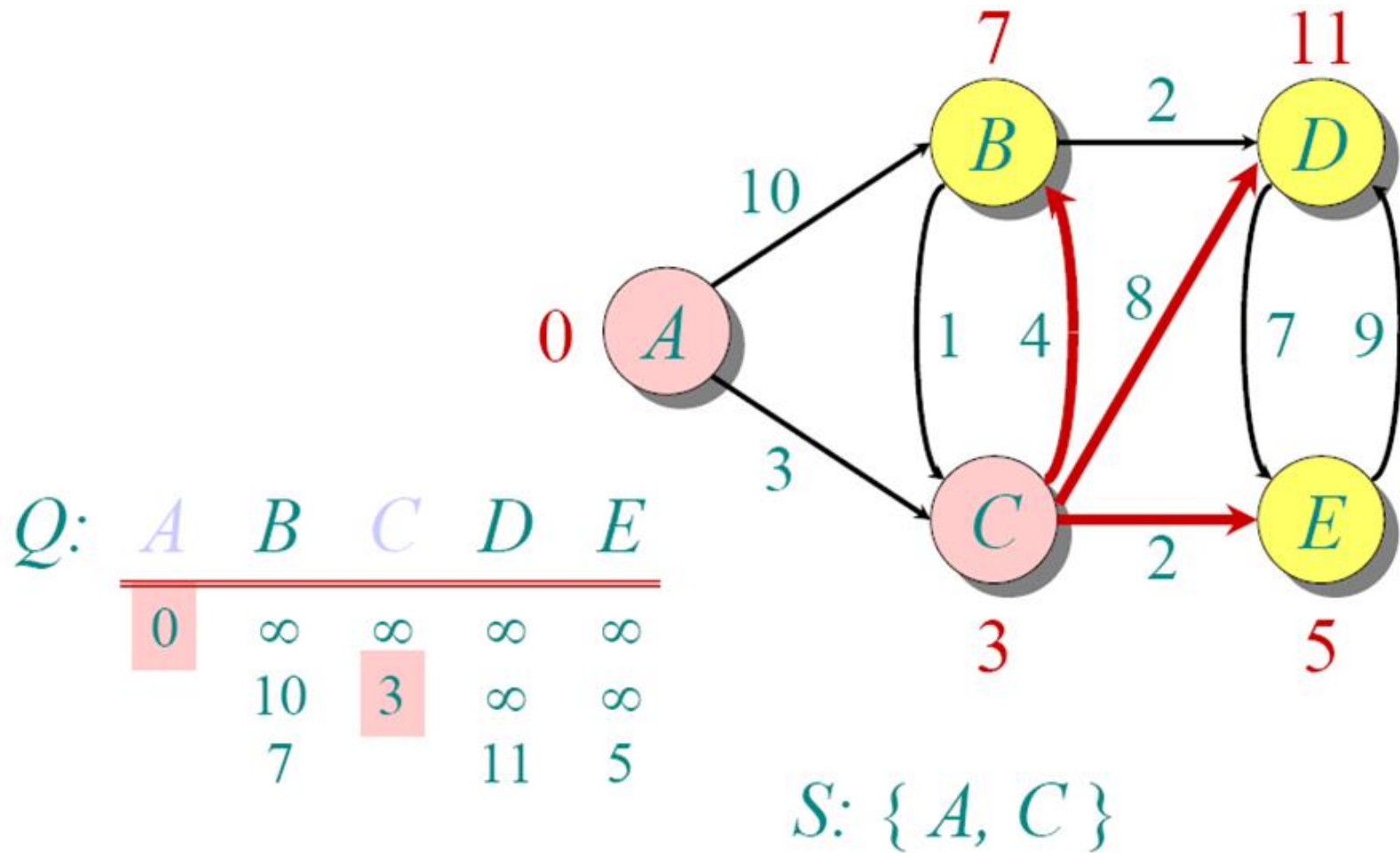
Dijkstra's Algo: Example



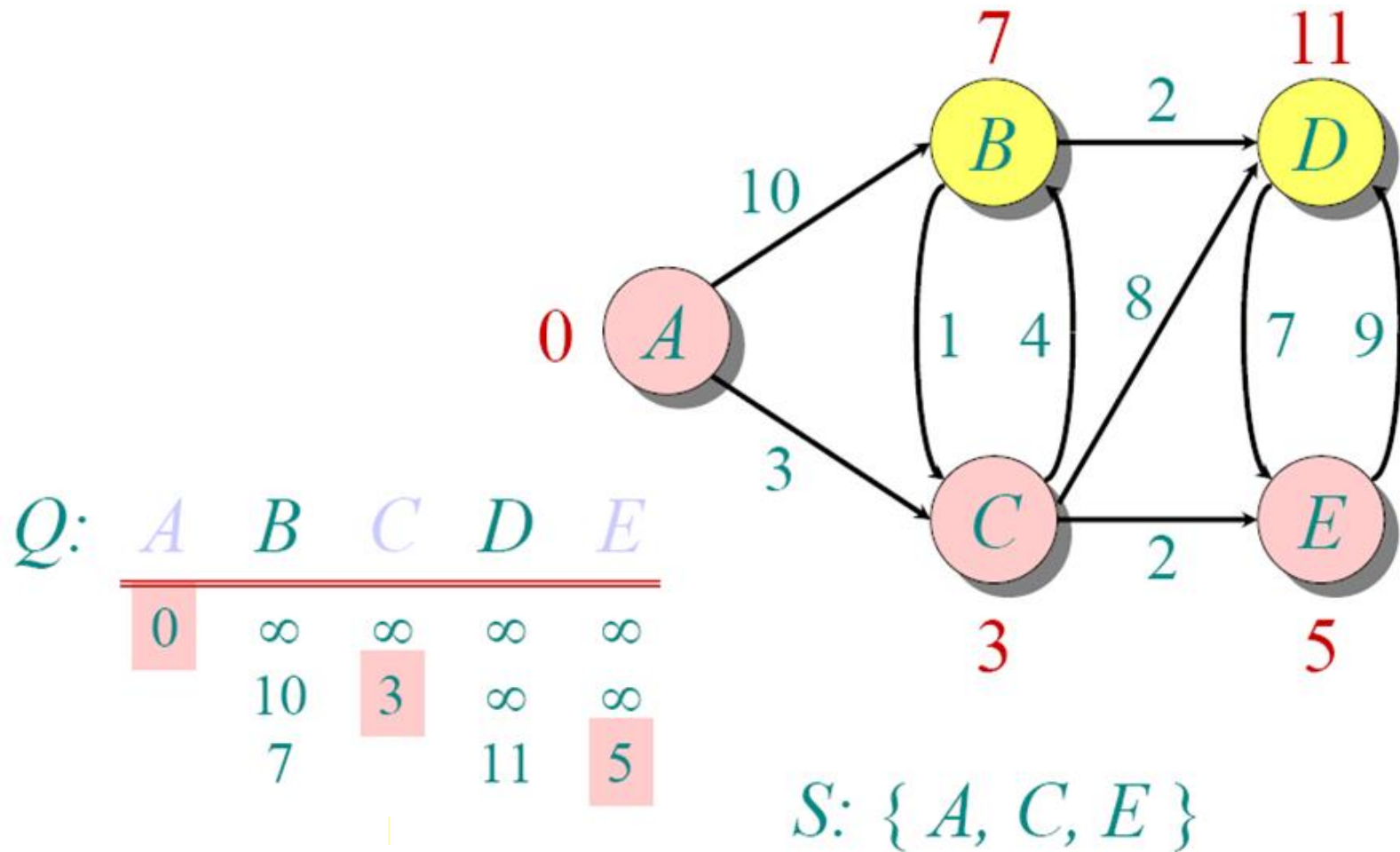
Dijkstra's Algo: Example



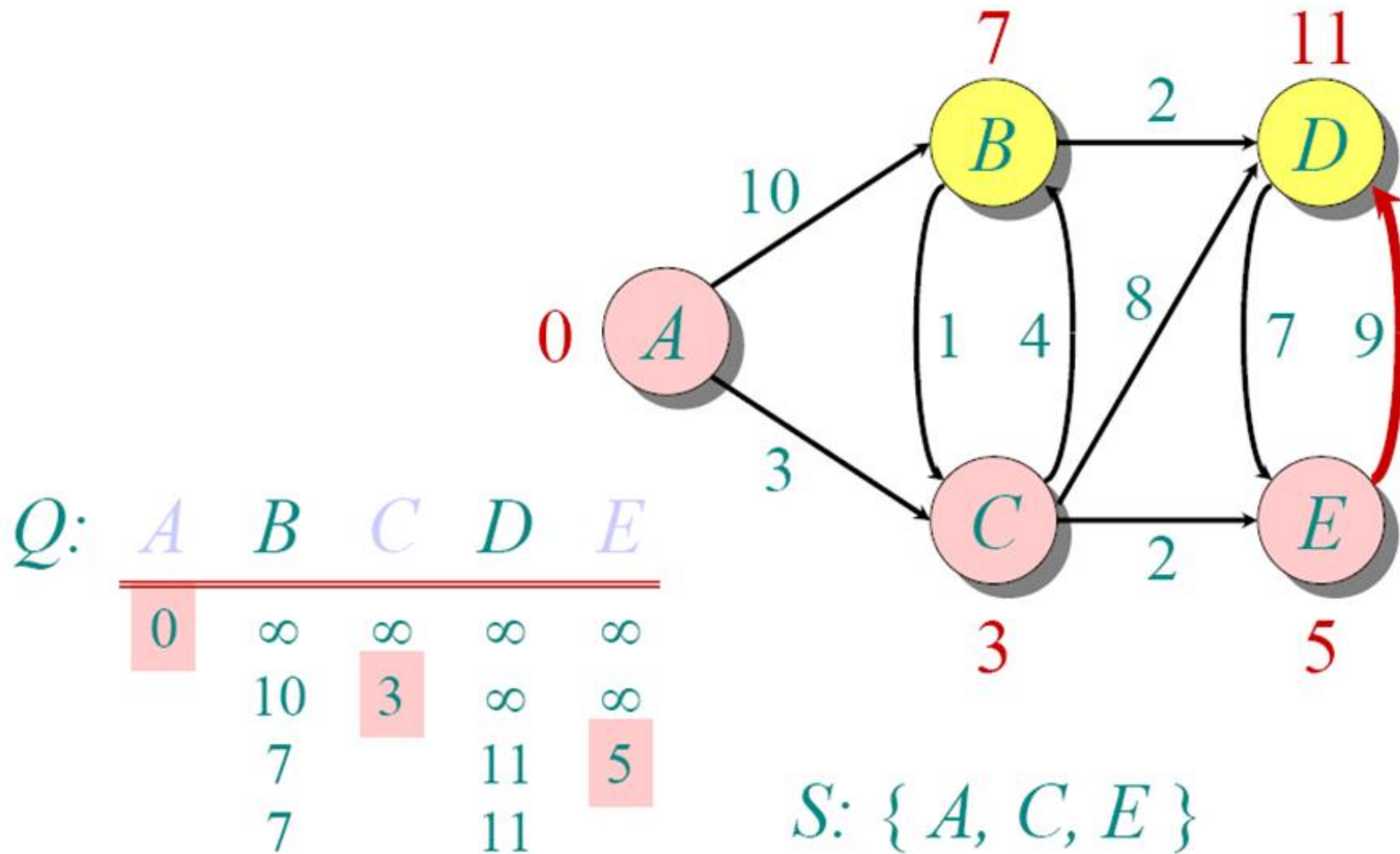
Dijkstra's Algo: Example



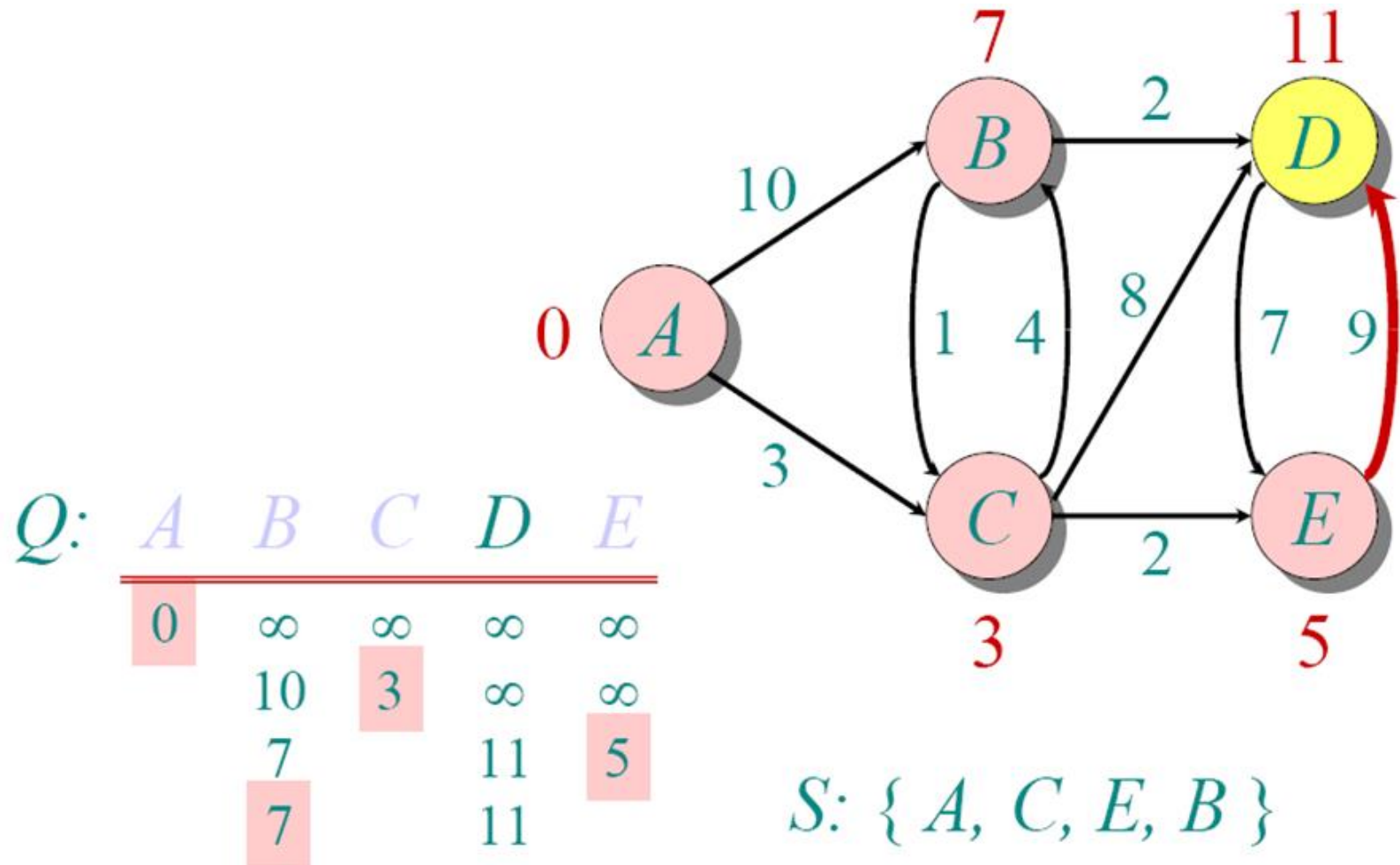
Dijkstra's Algo: Example



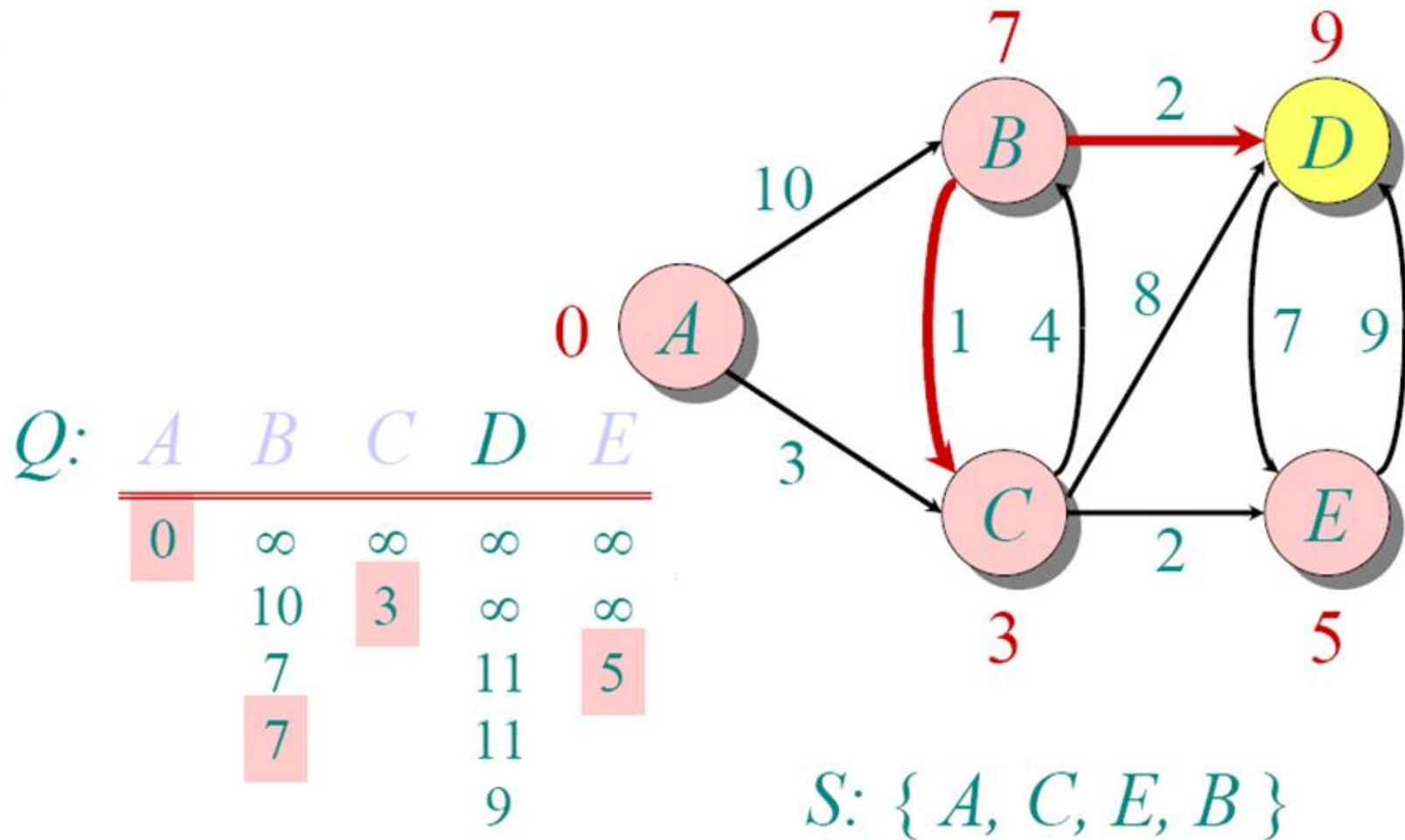
Dijkstra's Algo: Example



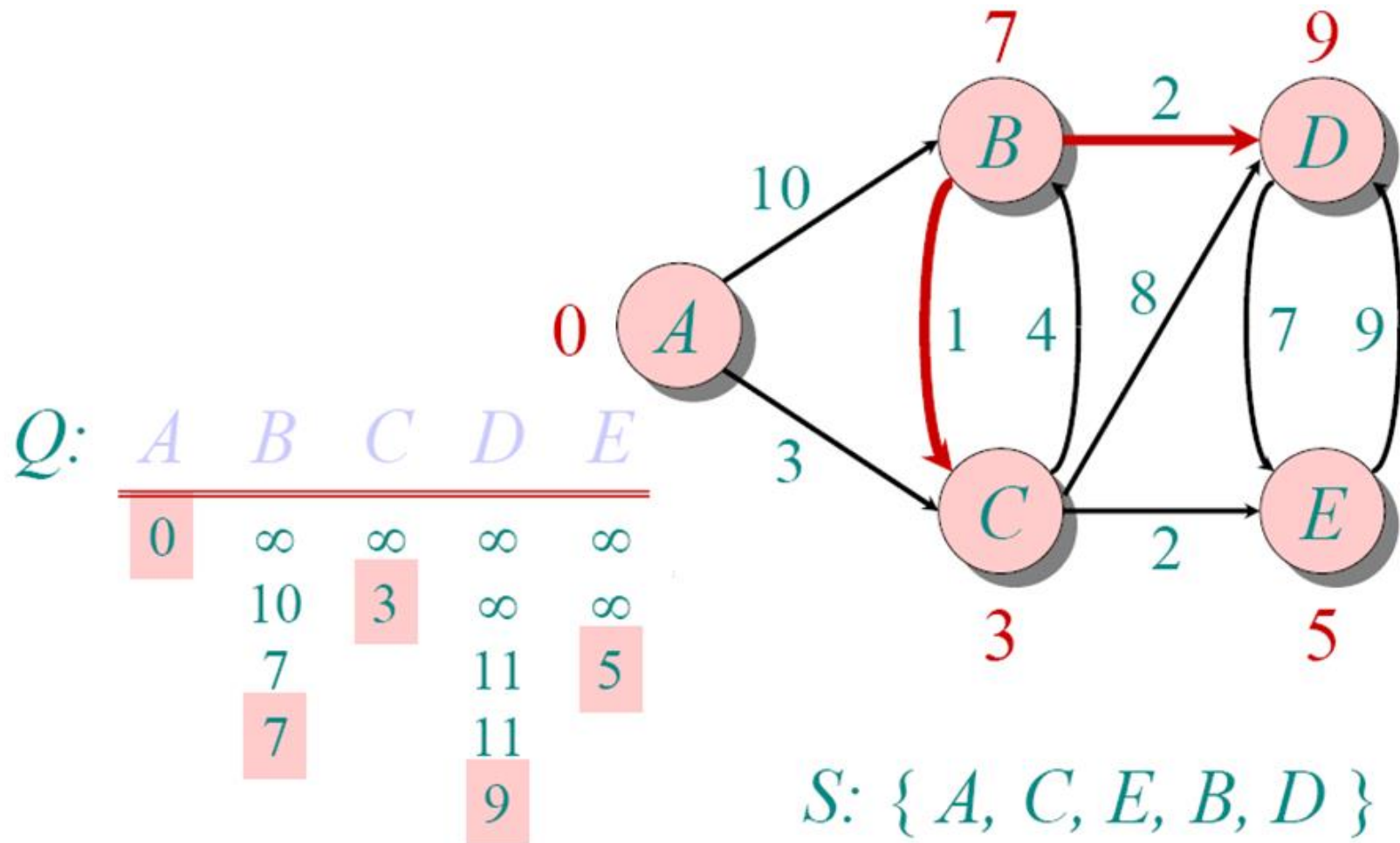
Dijkstra's Algo: Example



Dijkstra's Algo: Example



Dijkstra's Algo: Example



Dijkstra's Algo : Analysis

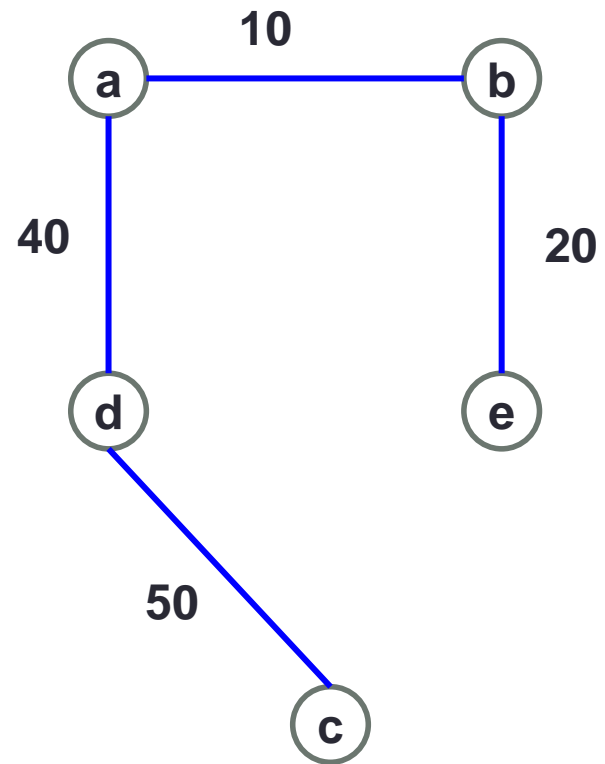
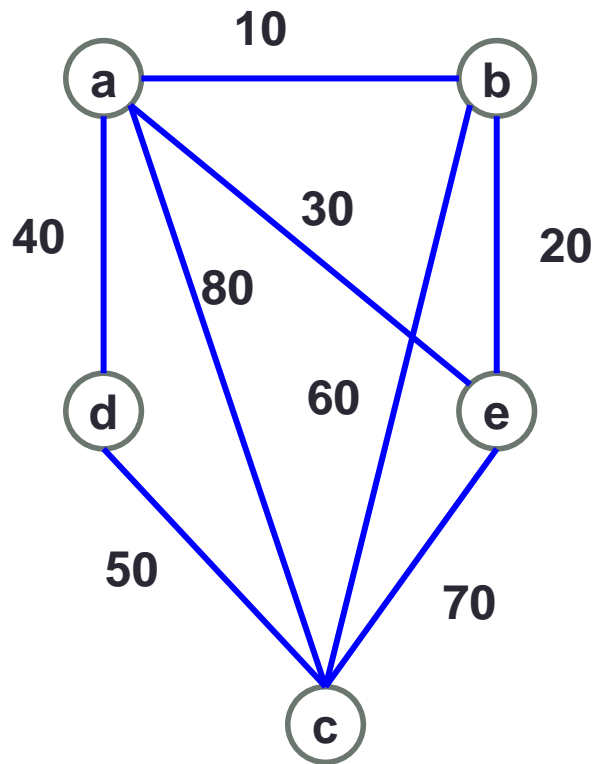
When linked list or Array is used to store vertices and their distance label

- Initialization of all vertices : $O(V)$
- Finding and deleting minimum will take $O(V)$ time
- While loop will run $O(V)$ times and in each step find and delete minimum
- Hence, total is $O(V^2)$
- Optionally, we might need a previous pointer at each vertex to redraw the shortest path tree, which is $O(E)$
- Total time = $O(V^2 + E)$

Minimum Spanning Tree (MST)

- For an un-directed connected weighted graph, MST is a spanning tree with minimum total cost
- Each edge has some cost associated in a weighted graph
- There can be more than MST for a graph, though cost of every MST possible is same
- Use case: BSNL wants to lay underground cables across the city to connect every household. One solution is to connect every household directly with centre. Another is building an MST

MST Example



Algorithms to Find MST

- Kruskal's Algo: In each step include edge with minimum weight in MST from remaining edges such that it does not forms a cycle. Repeat this process until all vertices are covered
- Prim's Algo: Start from any vertex and add it to set S . From all the edges having one endpoint in S , pick the edge with minimum weight not forming cycle and add to MST. Repeat the same process, till all vertices are covered.

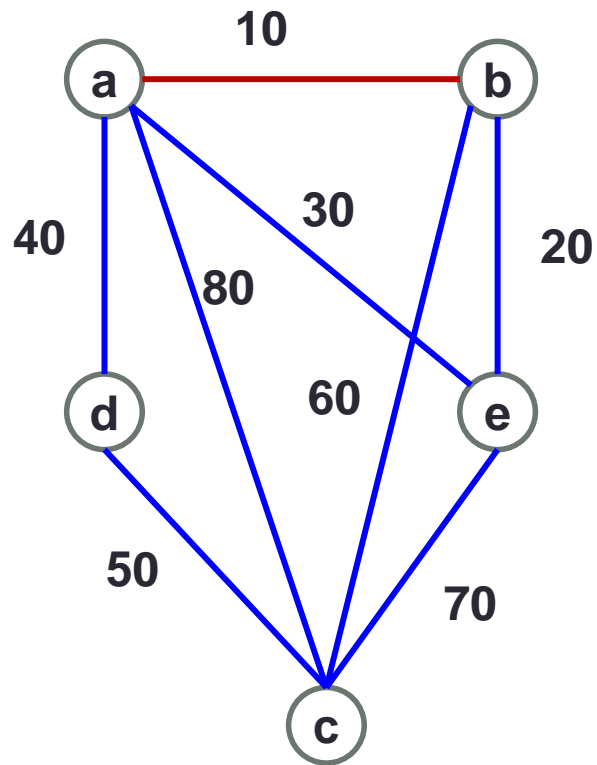
Kruskal's Algorithm

- Kruskal's Algorithm might produce a forest for a connected graph intermediately but eventually will produce MST at end.

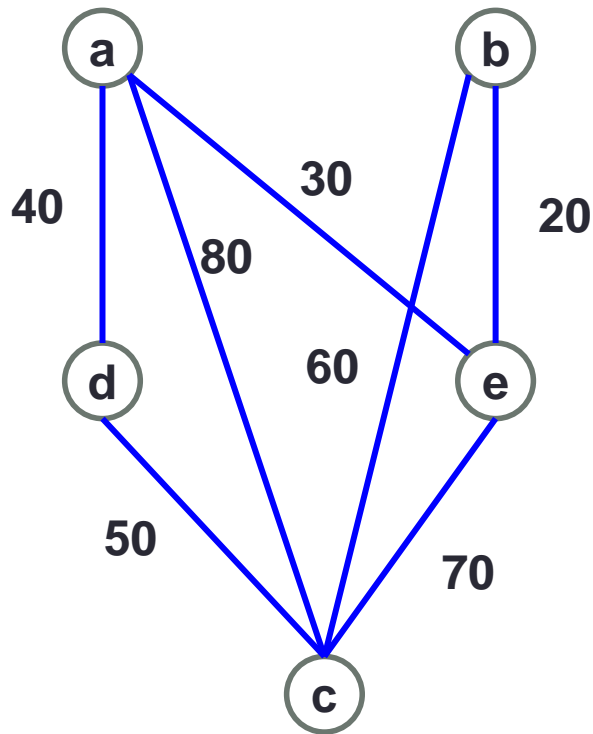
Algo:

1. Start with NULL MST
2. From set E , delete edge with minimum weight e
3. If adding e to MST creates cycle, reject e . Else add e to MST
4. Repeat step 2 for remaining edges in E or until $n-1$ edges are added to MST

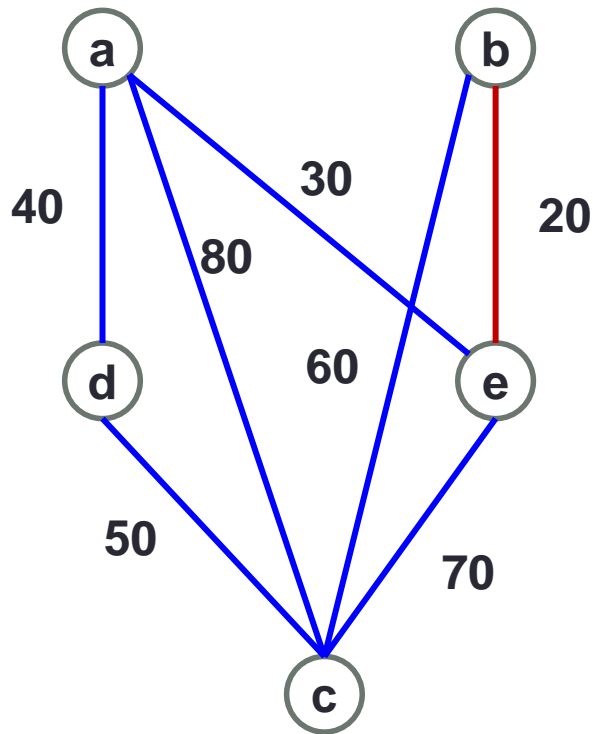
Kruskal's Algo : Example



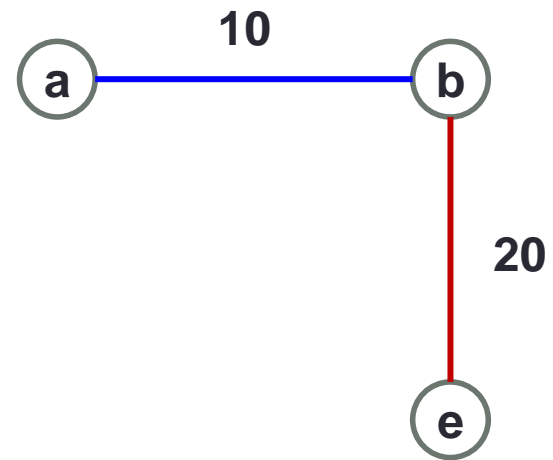
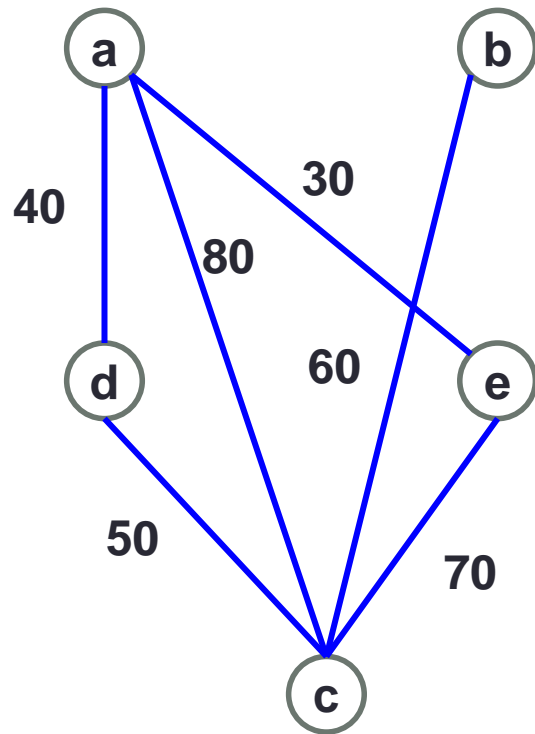
Kruskal's Algo : Example



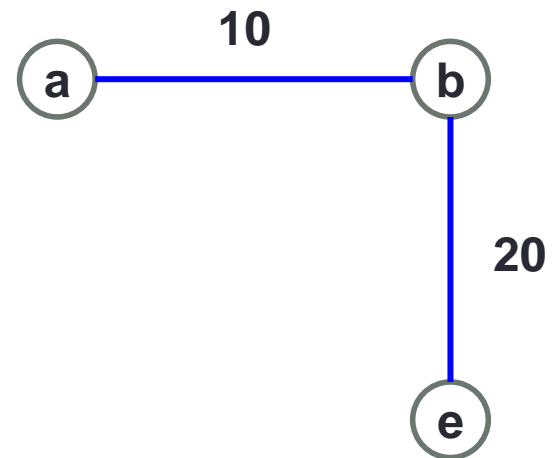
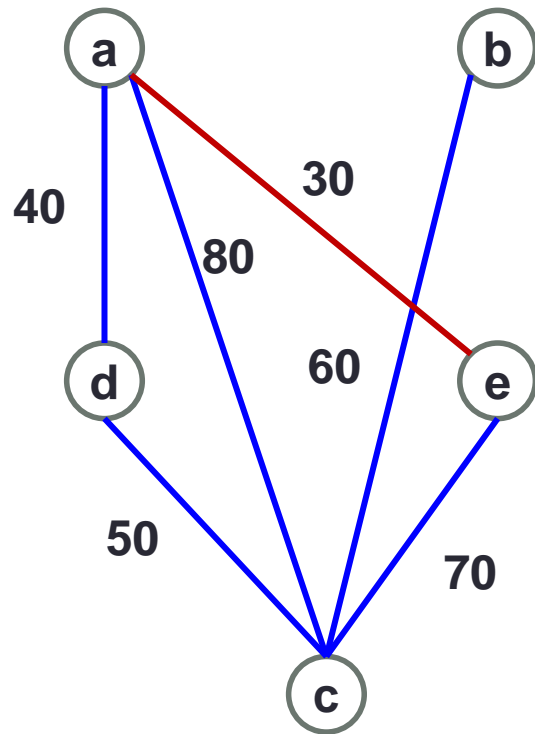
Kruskal's Algo : Example



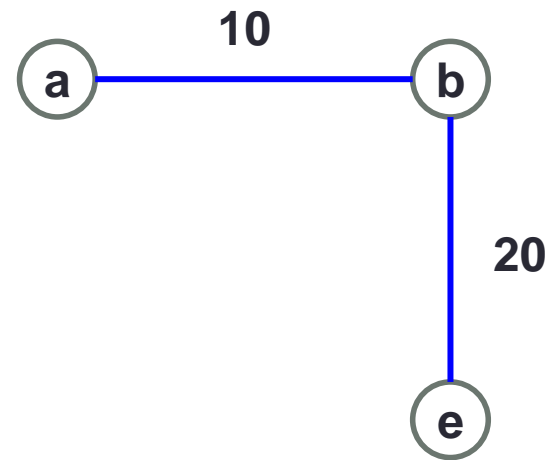
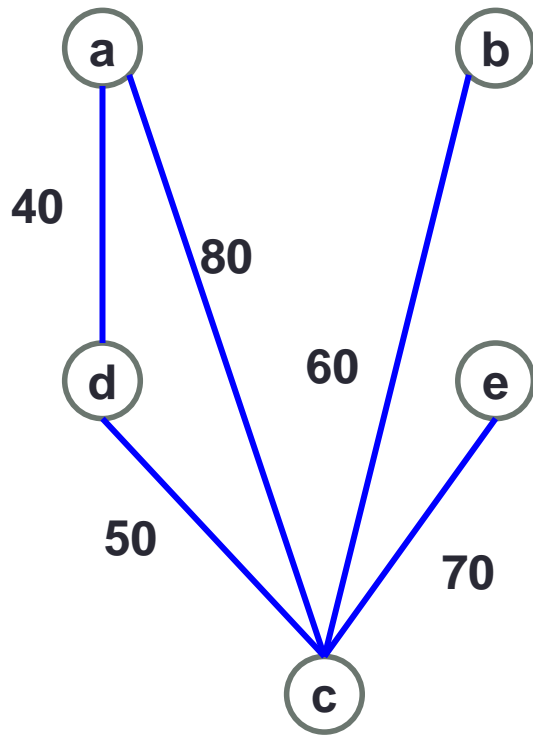
Kruskal's Algo : Example



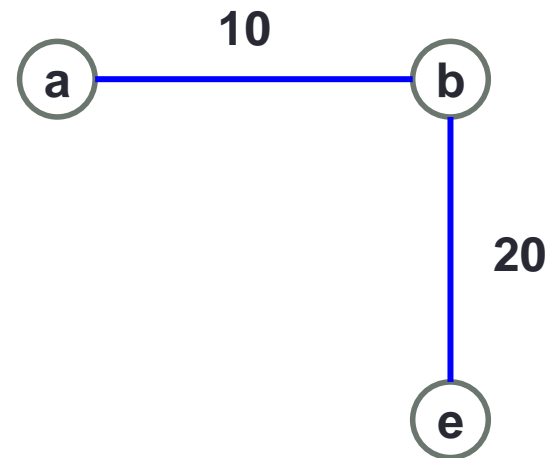
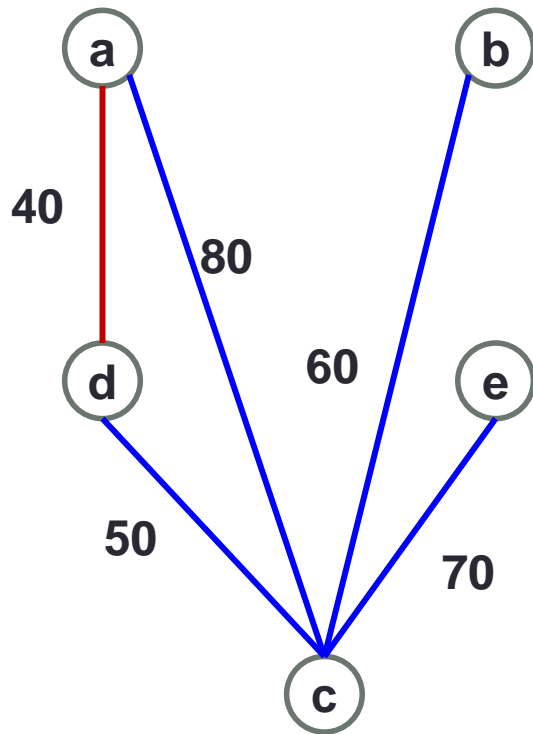
Kruskal's Algo : Example



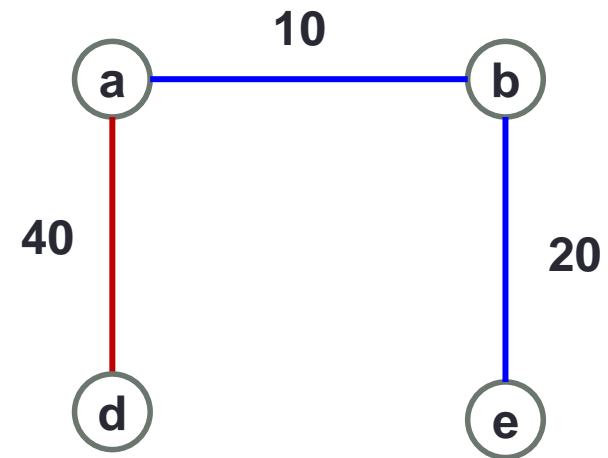
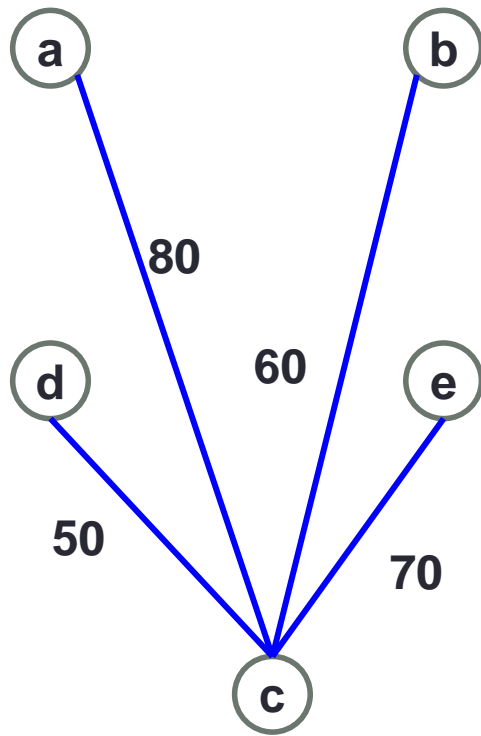
Kruskal's Algo : Example



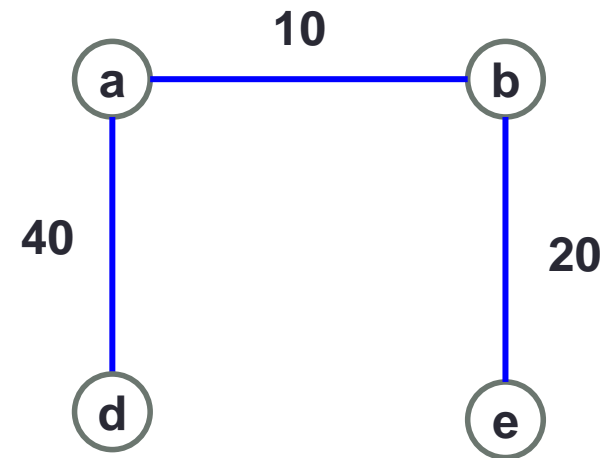
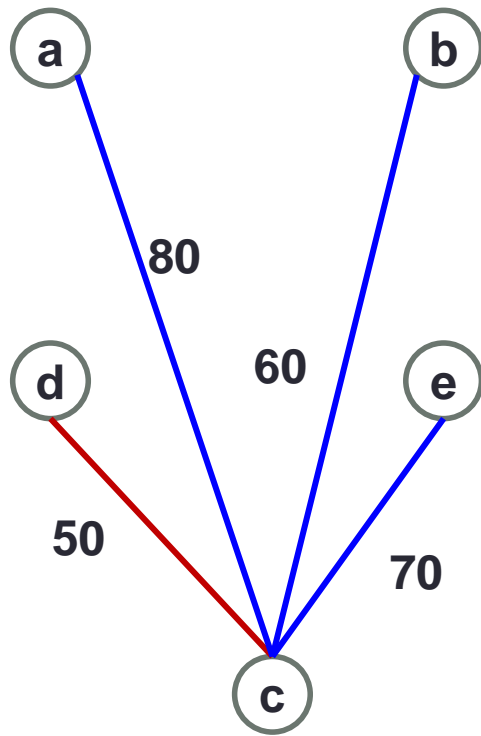
Kruskal's Algo : Example



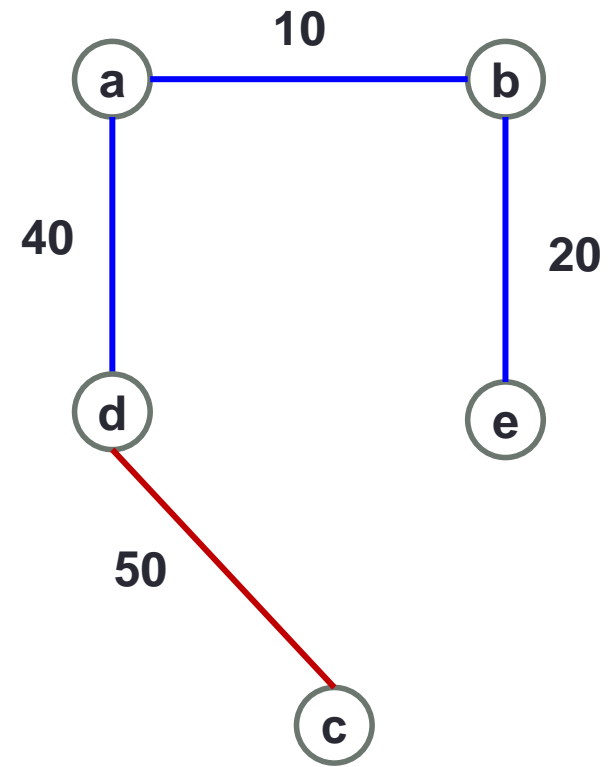
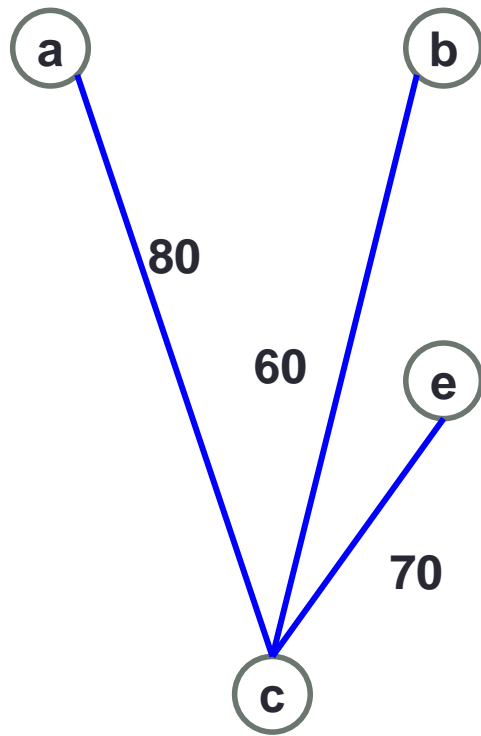
Kruskal's Algo : Example



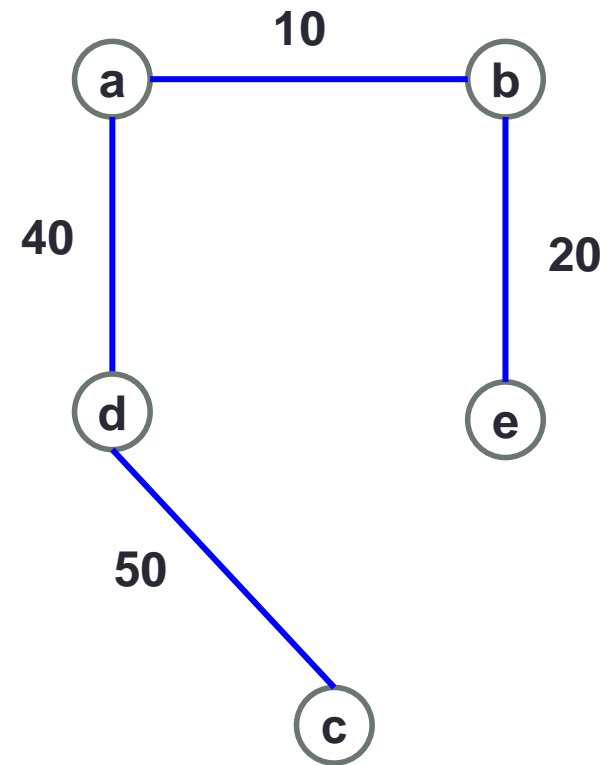
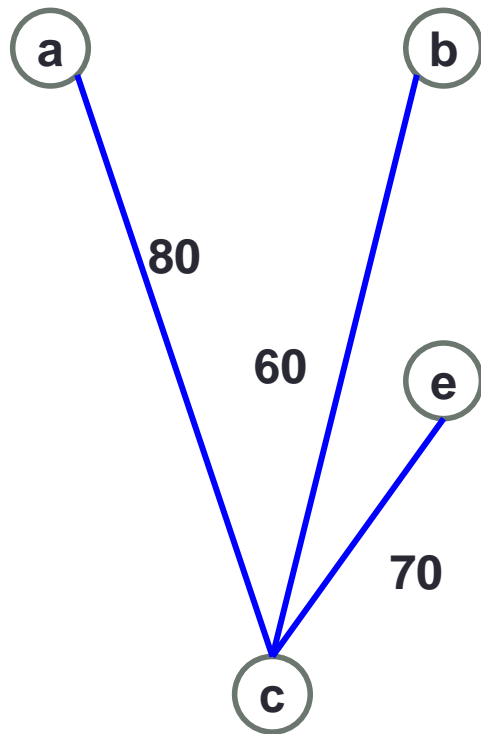
Kruskal's Algo : Example



Kruskal's Algo : Example




Kruskal's Algo : Example




Total cost = 120

Kruskal's Algo : Analysis

- Kruskal's algo requires to delete edge with minimum weight at each step. Therefore, if we sort edges in ascending order : $O(m \log m)$
and deleting smallest element is $O(1)$ time
- For each edge, compare whether adding the edge would create cycle: $O(\log n)$ [using union-find data structure]
 - If not then adding that edge to MST: $O(1)$ [using union-find]
- Time required to run loop: $O(m \log n + n)$



**Loop running
m times**



**Adding only
n-1 edges**
- Total running time = $O(m \log m + m \log n + n)$
= $O(m \log n)$ [$m \leq n^2$]

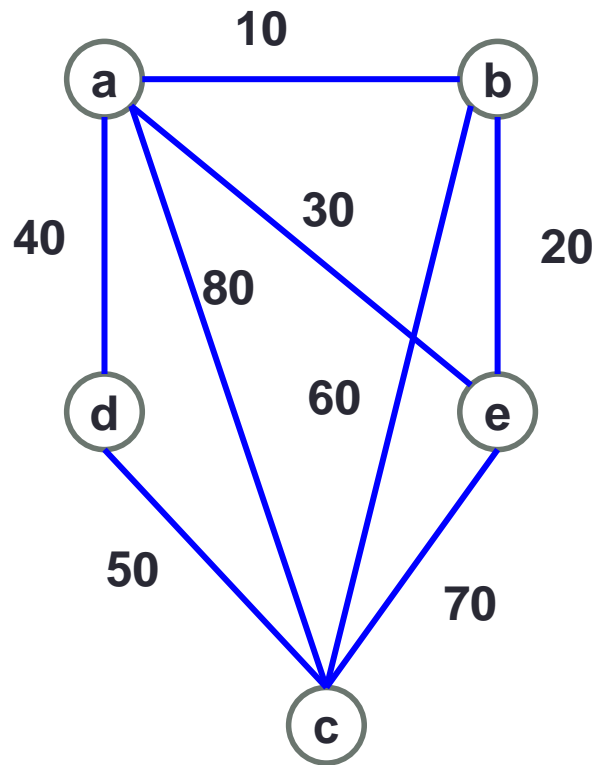
Prim's Algorithm

- Start with arbitrary vertex in the tree. In each step add one more vertex with minimum weight to tree until all vertices are covered.

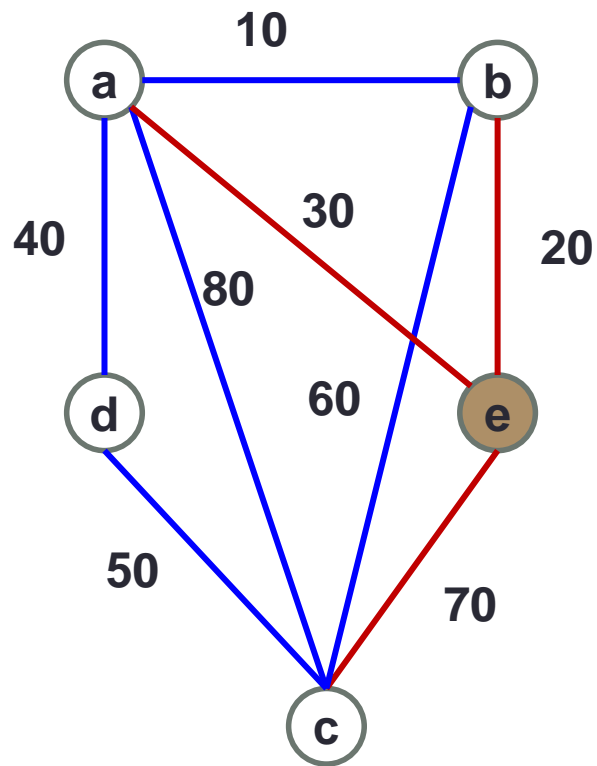
Algo

1. Start with NULL MST
2. Add any arbitrary vertex v to set S
3. Add a new vertex from $V-S$ to S which is adjacent to any vertex $u \in S$ with minimum weight edge
4. Repeat step 3 until all vertices are included in S .

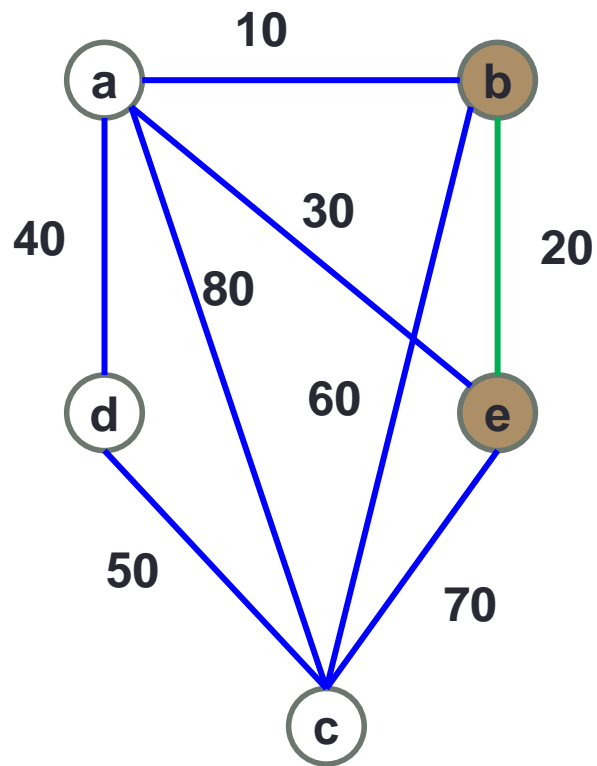
Prim's Algo : Example



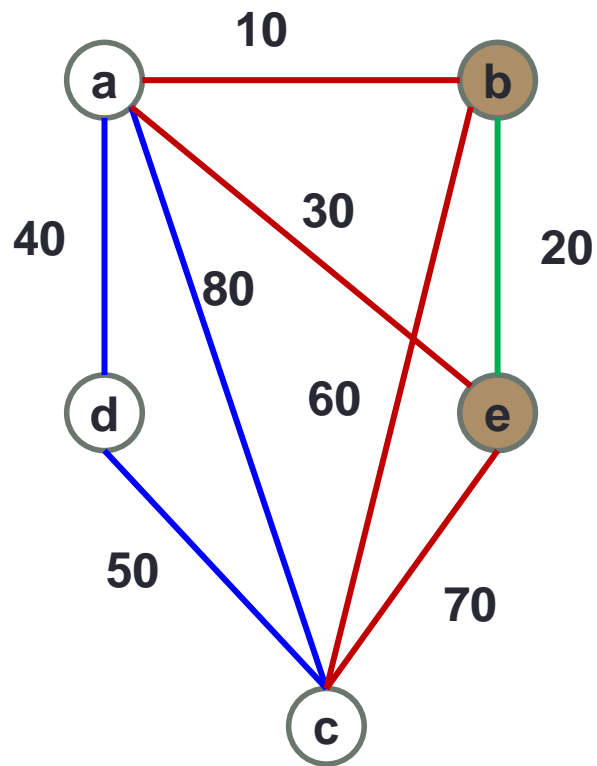
Prim's Algo : Example



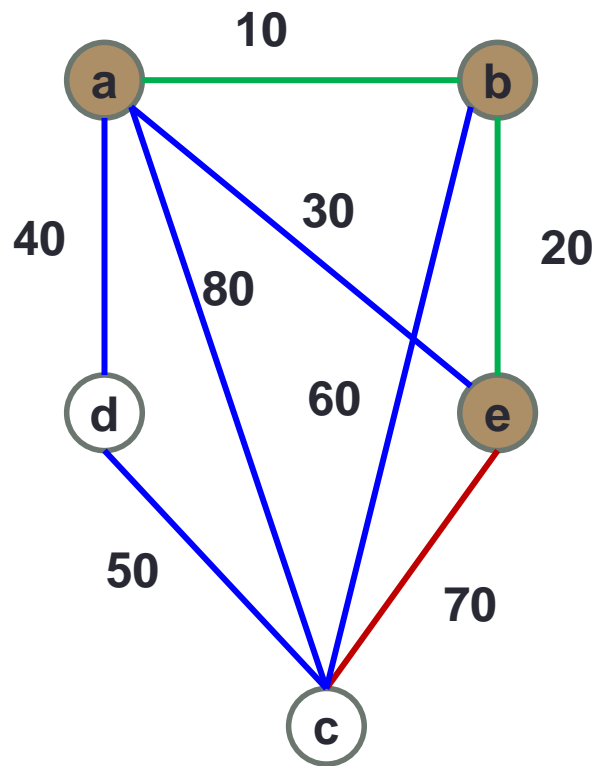
Prim's Algo : Example



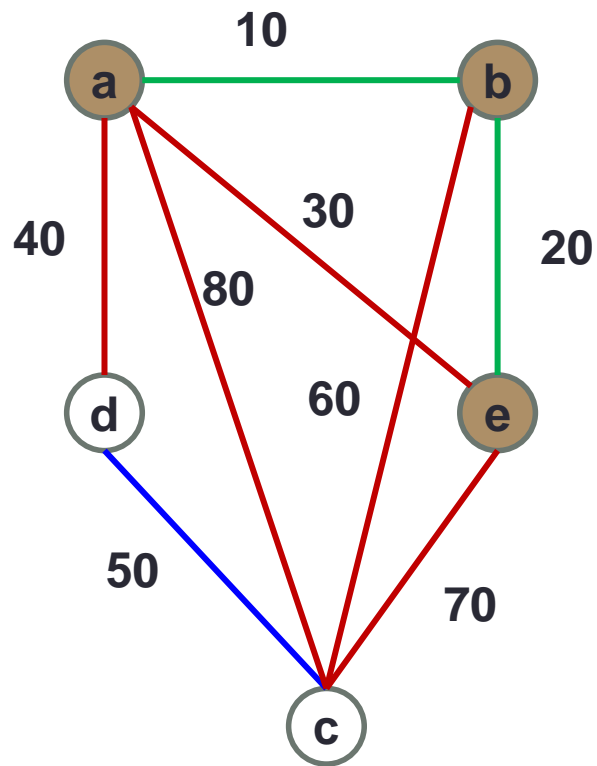
Prim's Algo : Example



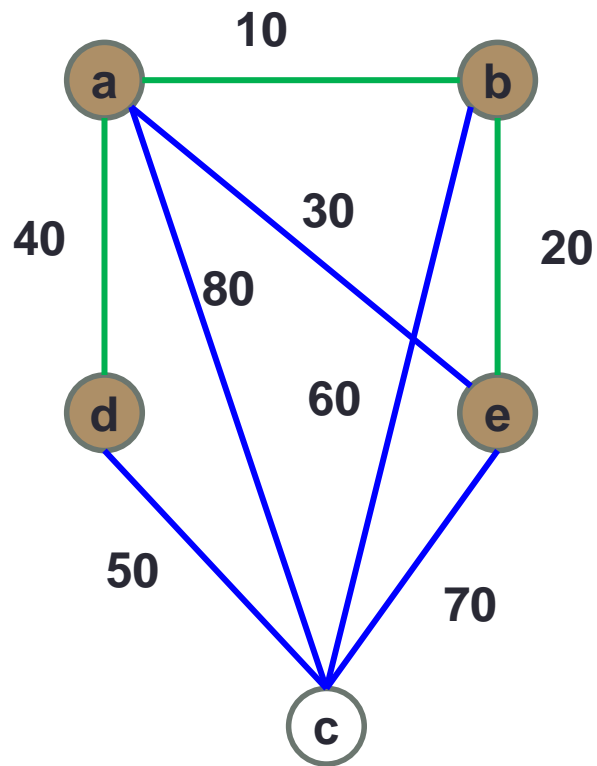
Prim's Algo : Example



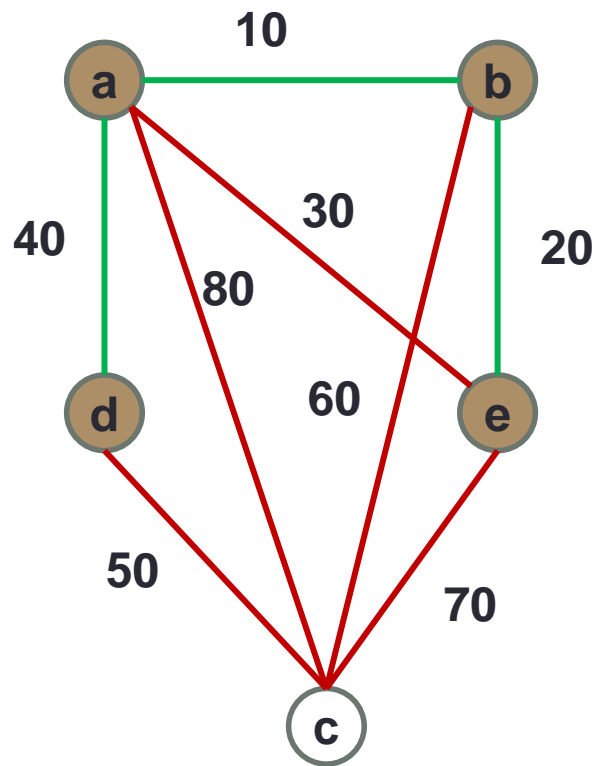
Prim's Algo : Example



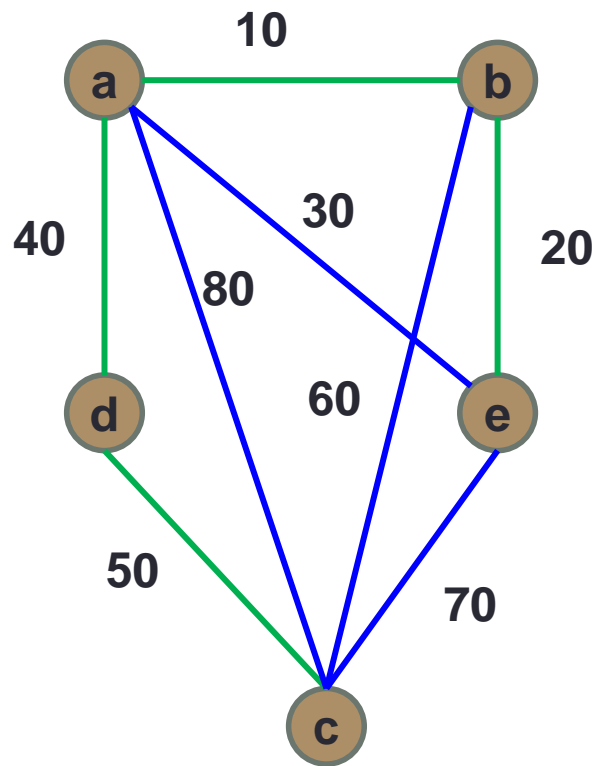
Prim's Algo : Example



Prim's Algo : Example



Prim's Algo : Example



Prim's Algo : Example

