CMC UNIVERSITY

CHAPTER 5. GRAPHS

CMC UNIVERSITY

Outline

- ✓ Basic concepts
- ✓ Graph representations
- ✓ Graph traversals
- ✓ Shortest paths
- ✓ Minimum Spanning Tree

CMC UNIVERSITY

BASIC CONCEPTS

Graphs

- A **graph** is a set of a finite number of vertices (also known as nodes) and edges, in which the edges are the links between vertices, and each edge in a graph joins two distinct nodes.
- A graph is a formal mathematical representation of a network, i.e. a graph G = (V, E) is an ordered pair of a set V of vertices and a set E of edges.

Example: Graph G = (V, E)

- V = {A, B, C, D, E}
- E = {{A, B}, {A, C}, {B, C}, {B, D}, {C, D}, {D, D}, {B, E}, {D, E}}

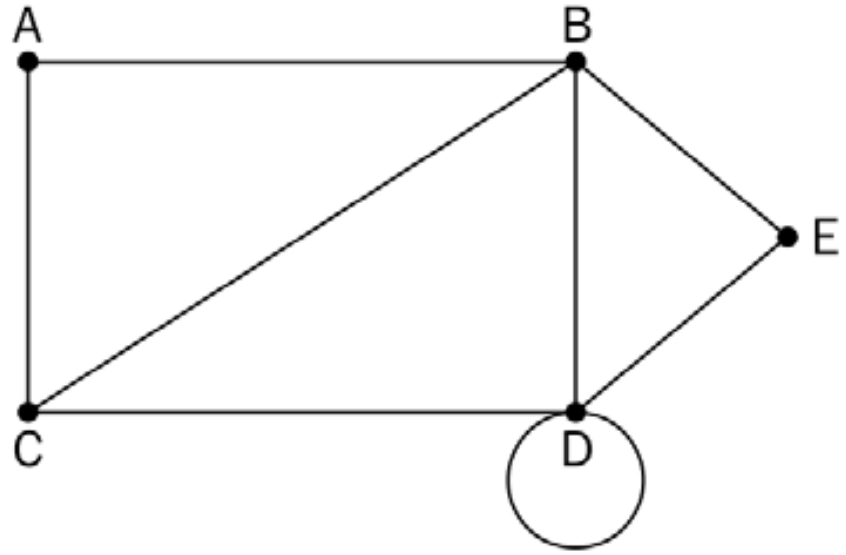


Fig.1: Example of a graph

Graphs

- **Node or vertex**: A point or node in a graph is called a vertex. In the Fig.1, the vertices or nodes are A, B, C, D, and E and are denoted by a dot.
- **Edge**: This is a connection between two vertices. The line connecting A and B is an example of an edge.
- **Loop**: When an edge from a node is returned to itself , that edge forms a loop, e.g. D node.
- **Degree of a vertex/node**: The total number of edges that are incidental on a given vertex is called the degree of that vertex. For example, the degree of the B vertex in the previous diagram is 4.
- **Adjacency**: This refers to the connection(s) between any two nodes; thus, if there is a connection between any two vertices or nodes, then they are said to be adjacent to each other. For example, the C node is adjacent to the A node because there is an edge between them

CMC UNIVERSITY

Graphs

- **Path**: A sequence of vertices and edges between any two nodes represents a path. For example, CABE represents a path from the C node to the E node.
- **Simple path**: a path on which every vertex, except the first and last vertices, is different.
- **Path length**: the number of edges on that path.
- **Leaf vertex** (also called pendant vertex): A vertex or node is called a leaf vertex or pendant vertex if it has exactly one degree.

Directed and undirected Graphs

- If the connecting edges in a graph are undirected, then the graph is called an **undirected graph**, and if the connecting edges in a graph are directed, then it is called a **directed graph**.
- An undirected graph simply represents edges as lines between the nodes. There is no additional information about the relationship between the nodes, other than the fact that they are connected. For example, in Figure 2, we demonstrate an undirected graph of four nodes, A, B, C, and D, which are connected using edges:

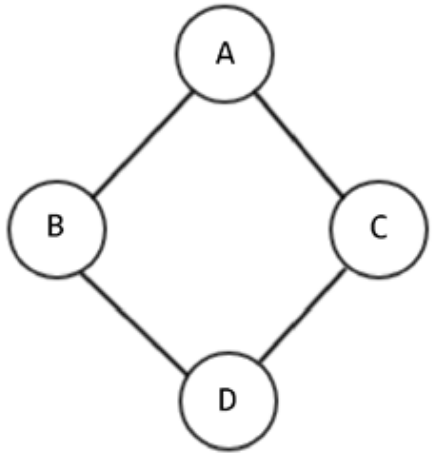


Fig.2 Example of an undirected graph

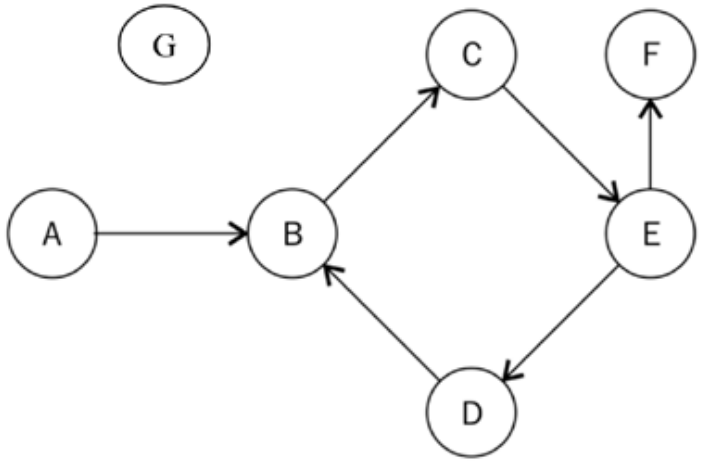


Fig.3. Example of a directed graph

CMC UNIVERSITY

Directed and undirected Graphs

The arrow of an edge determines the flow of direction. One can only move from A to B, as shown in the Fig.3, not B to A. In a directed graph, each node (or vertex) has an *indegree* and an *outdegree*. Let's have a look at what these are

- **Indegree**: The total number of edges that come into a vertex in the graph is called the indegree of that vertex. For example, in the previous diagram, the E node has 1 indegree, due to edge CE coming into the E node.
- **Outdegree**: The total number of edges that go out from a vertex in the graph is called the outdegree of that vertex. For example, the E node in the previous diagram has an outdegree of 2, as it has two edges, EF and ED, going out of that node.

CMC UNIVERSITY

Directed and undirected Graphs

- **Isolated vertex**: A node or vertex is called an isolated vertex when it has a degree of zero, as shown as G node in Fig.3.
- **Source vertex**: A vertex is called a source vertex if it has an indegree of zero. For example, in the previous diagram, the A node is the source vertex.
- **Sink vertex**: A vertex is a sink vertex if it has an outdegree of zero. For example, in the previous diagram, the F node is the sink vertex.

Directed acyclic Graphs

- A **directed acyclic graph** (DAG) is a directed graph with no cycles; in a DAG all the edges are directed from one node to another node so that the sequence of edges never forms a closed loop.
- A **cycle** in a graph is formed when the starting node of the first edge is equal to the ending node of the last edge in a sequence

In a directed acyclic graph, if we start on any path from a given node, we never find a path that ends on the same node. A DAG has many applications, such as in job scheduling, citation graphs, and data compression.

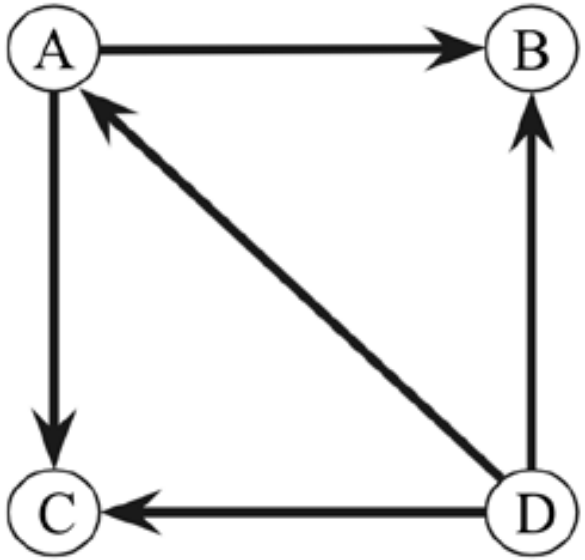


Fig.4. Example of a directed acyclic graph

Connected Graphs

- **Connected**: Two vertices v_i and v_j are said to be connected if there is a path from v_i to v_j (with an undirected graph, there is also a path from v_j to v_i).
- A graph G is said to be connected if for every pair of distinct vertices v_i and v_j in V(G) there is a path from v_i to v_j .

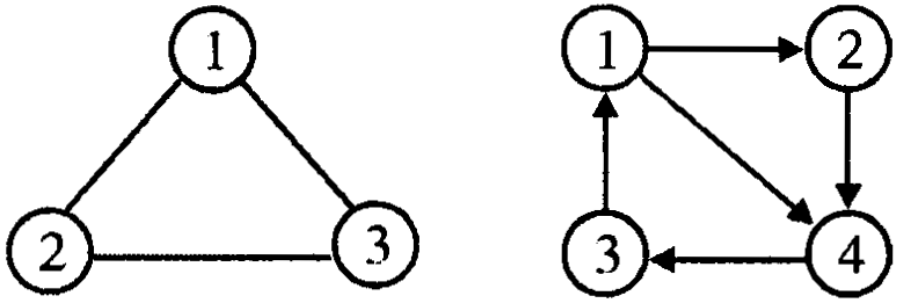


Figure a) Connected graphs

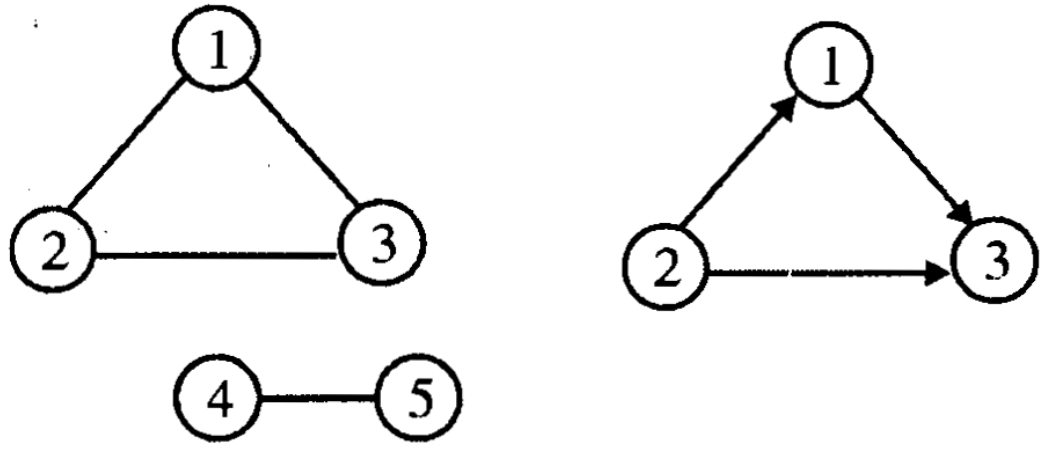


Figure b) Unconnected graphs

Weighted graphs

- A weighted graph is a graph that has a numeric weight associated with the edges in the graph. A weighted graph can be either a directed or an undirected graph. The numeric weight can be used to indicate distance or cost, depending on the purpose of the graph:

The weighted graph in Fig.5 indicates different ways to reach from A node to D node.
There are two possible paths: A-D (cost 40) and A-B-C-D (cost 25)

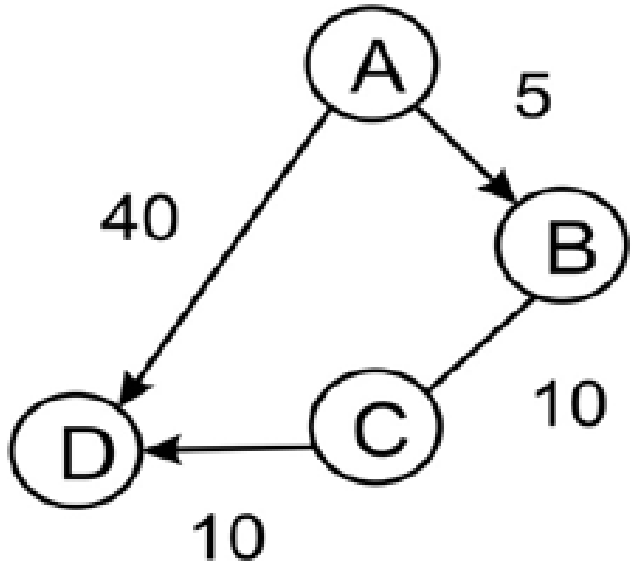


Fig.5. Example of a weighted graph

Bipartite graphs

- A **bipartite graph** (also known as a bigraph) is a special graph in which all the nodes of the graph can be divided into two sets in such a way that edges connect the nodes from one set to the nodes of another set.

The bipartite graph in Fig.6: all the nodes are divided into two independent sets U and V, so that each edge in the graph has one end in set U and another end in set V (e.g. in edge (A, E), one end or one vertex is from set U, and another end or another vertex is from set V).
In bipartite graphs, no edge will connect to the nodes of the same set.

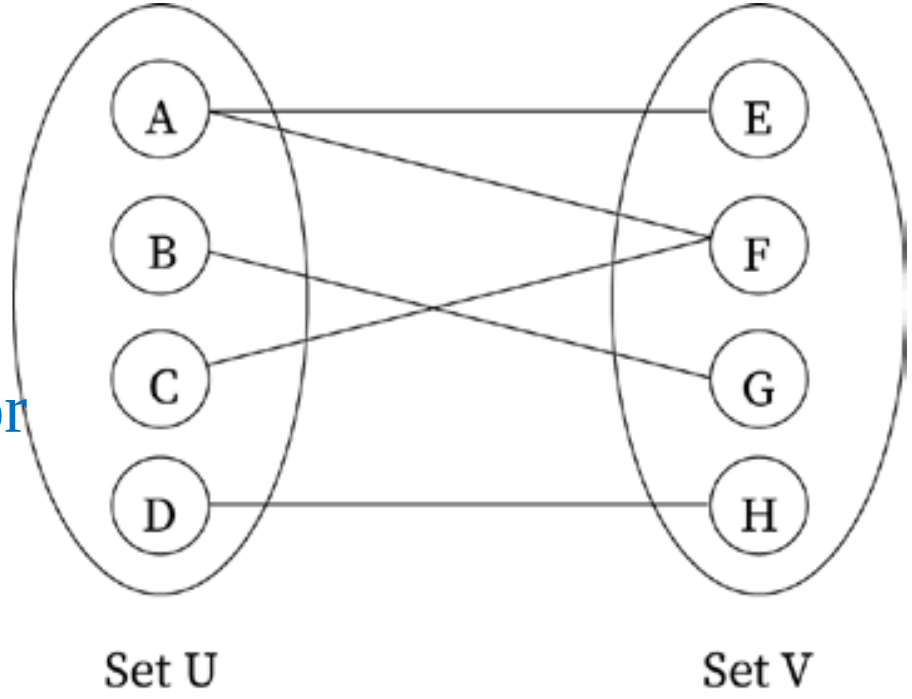


Fig.6. Example of a bipartite graph

Bipartite graphs are useful when we need to model a relationship between two different classes of objects, for example, a graph of applicants and jobs, ...

CMC UNIVERSITY

GRAPHS REPRESENTATIONS

CMC UNIVERSITY

Graph representations

Graphs can be represented with two methods: adjacency list, and adjacency matrix.

- An adjacency list representation is based on a linked list. In this, we represent the graph by maintaining a list of neighbors (also called an adjacent node) for every vertex (or node) of the graph.
- In an adjacency matrix representation of a graph, we maintain a matrix that represents which node is adjacent to which other node in the graph; i.e., the adjacency matrix has the information of every edge in the graph, which is represented by cells of the matrix.

Adjacency lists

- All the nodes directly connected to a node x are listed in its adjacent list of nodes. The graph is represented by displaying the adjacent list for all the nodes of the graph.
- Two nodes, A and B, in the graph shown in Fig.7, are said to be adjacent if there is a direct connection between them:

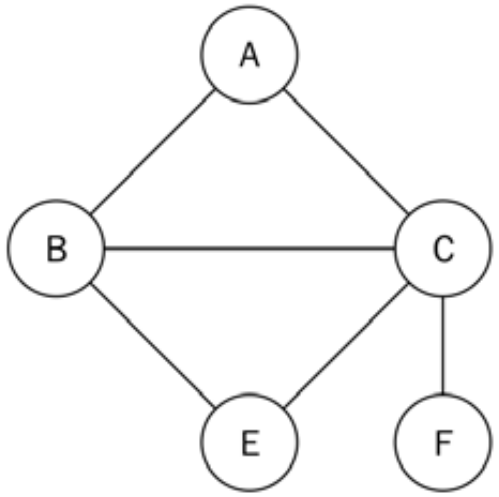


Fig.7. An Example a graph of 5 nodes

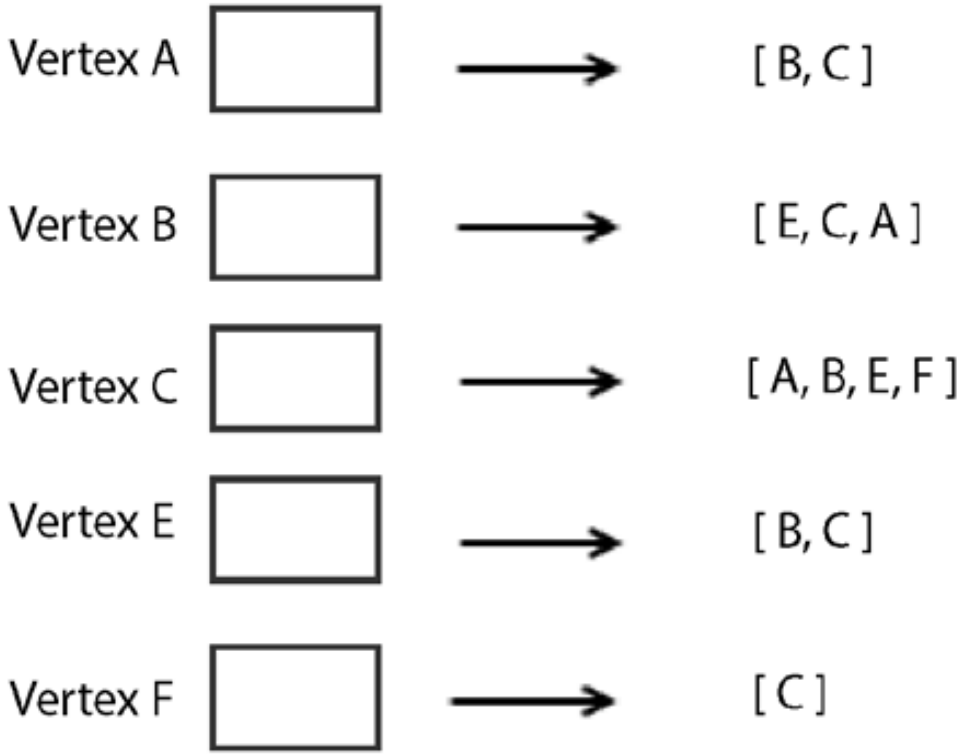


Fig.8. Adjacency list for the graph in Fig.7

CMC UNIVERSITY

Adjacency lists

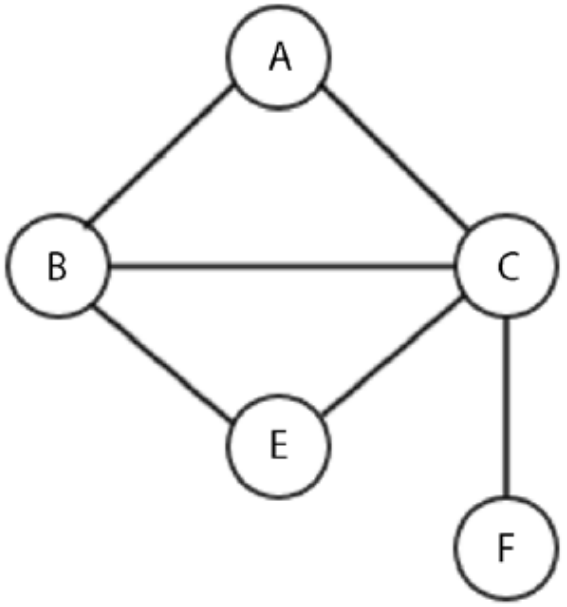
- A linked list can be used to implement the adjacency list. In order to represent the graph, we need the number of linked lists equal to the total number of nodes in the graph. At each index, the adjacent nodes to that vertex are stored. For example, consider the adjacency list shown in Fig.8 corresponding to the sample graph shown in Fig.7

```
graph = dict()
graph['A'] = ['B', 'C']
graph['B'] = ['E','C', 'A']
graph['C'] = ['A', 'B', 'E','F']
graph['E'] = ['B', 'C']
graph['F'] = ['C']
```

- The adjacency list is a preferable graph representation technique when the graph is going to be sparse and we may need to add or delete the nodes in the graph frequently. It is very difficult to check whether a given edge is present in the graph or not using this technique.

Adjacency matrix

- The graph is represented by showing the nodes and their interconnections through edges. The dimensions (V x V) of a matrix are used to represent the graph, where each cell denotes an edge in the graph.
- A matrix is a two-dimensional array, represent the cells of the matrix with a 1 or a 0, depending on whether two nodes are connected by an edge or not. We show an example graph, along with its corresponding adjacency matrix, in Fig.9.



Adjacency Matrix

	A	B	C	E	F
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	1	1
E	0	1	1	0	0
F	0	0	1	0	0

Fig.9. Adjacency matrix for a given graph

CMC UNIVERSITY

Adjacency matrix

- An adjacency matrix can be implemented using the given adjacency list, use the previous dictionary-based implementation of the graph:

- Firstly, we have to obtain the key elements of the adjacency matrix. It is important to note that these matrix elements are the vertices of the graph. We can get the key elements by sorting the keys of the graph.

```
matrix_elements = sorted(graph.keys())
cols = rows = len(matrix_elements)
```

- Next, the length of the keys of the graph will be the dimensions of the adjacency matrix, which are stored in cols and rows. The values of the cols and rows are equal.

- So, now, we create an empty adjacency matrix of the dimensions cols by rows, initially filling all the values with zeros.

```
adjacency_matrix = [[0 for x in range(rows)] for y in range(cols)]
edges_list = []
```

CMC UNIVERSITY

Adjacency matrix

- The edges_list variable will store the tuples that form the edges in the graph. For example, an edge between the A and B nodes will be stored as (A, B). The multidimensional array is filled using a nested for loop:

```
for key in matrix_elements:
    for neighbor in graph[key]:
        edges_list.append((key, neighbor))

print(edges_list)
```

- The neighbors of a vertex are obtained by graph[key]. The key, in combination with the neighbor, is then used to create the tuple stored in edges_list.

```
[('A', 'B'), ('A', 'C'), ('B', 'E'), ('B', 'C'), ('B', 'A'), ('C', 'A'),
('C', 'B'), ('C', 'E'), ('C', 'F'), ('E', 'B'), ('E', 'C'), ('F', 'C')]
```

CMC UNIVERSITY

Adjacency matrix

- The next step in implementing the adjacency matrix is to fill it, using 1 to denote the presence of an edge in the graph. This can be done with the adjacency_matrix[index_of_first_vertex] [index_of_second_vertex] = 1 statement. The full code snippet that marks the presence of edges of the graph is as follows:

```
for edge in edges_list:
    index_of_first_vertex = matrix_elements.index(edge[0])
    index_of_second_vertex = matrix_elements.index(edge[1])
    adjacency_matrix[index_of_first_vertex][index_of_second_vertex] = 1

print(adjacency_matrix)
```

Adjacency matrix

- The matrix_elements array has its rows and cols, starting from A to all other vertices with indices of 0 to 5. The for loop iterates through the list of tuples and uses the index method to get the corresponding index where an edge is to be stored.
- The output of the preceding code is the adjacency matrix for the sample graph shown previously in Fig.9. The adjacency matrix produced looks like the following:

(1, 1): 0 represents the absence of an edge between A and A;

(3, 2): 1 denotes the edge between the C and B vertices in the graph.

```
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
```

- The use of the adjacency matrix for graph representation is suitable when we have to frequently look up and check the presence or absence of an edge between two nodes in the graph (e.g. in creating routing tables in networks, searching routes in public transport applications and navigation systems, etc).
- Adjacency matrices are not suitable when nodes are frequently added or deleted within a graph, in those situations, the adjacency list is a better technique

CMC UNIVERSITY

GRAPH TRAVERSALS

CMC UNIVERSITY

Graph traversals

- A graph traversal means to visit all the vertices of the graph while keeping track of which nodes or vertices have already been visited and which ones have not.
- A graph traversal algorithm is efficient if it traverses all the nodes of the graph in the minimum possible time.
- Graph traversal, also known as a graph search algorithm, is quite similar to the tree traversal algorithms like preorder, inorder, postorder, and level order algorithms; similar to them, in a graph search algorithm we start with a node and traverse through edges to all other nodes in the graph.
- A common strategy of graph traversal is to follow a path until a dead end is reached, then traverse back up until there is a point where we meet an alternative path. We can also iteratively move from one node to another in order to traverse the full graph or part of it.

CMC UNIVERSITY

Breadth-first traversal

- **Breadth-first search** (BFS) works very similarly to how a level order traversal algorithm works in a tree data structure.
- The BFS algorithm: starts by visiting the root node at level 0, and then all the nodes at the first level directly connected to the root node are visited at level 1. Next, the level 2 nodes are visited next. Likewise, all the nodes in the graph are traversed level by level until all the nodes are visited.
- A queue data structure is used to store the information of vertices that are to be visited in a graph:

- Firstly, we visit the starting node, and then we look up all of its neighboring, or adjacent vertices. We first visit these adjacent vertices one by one, while adding their neighbors to the list of vertices that are to be visited.
- We follow this process until we have visited all the vertices of the graph, ensuring that no vertex is visited twice

Breadth-first traversal

Fig.10 presents a graph of five nodes on the left, and on the right, a queue data structure to store the vertices to be visited.

- Start visiting the first node, i.e., A node, and then we add all its adjacent vertices, B, C, and E, to the queue. There are multiple ways of adding the adjacent nodes to the queue: BCE, CEB, CBE, BEC, or ECB, each of which would give us different tree traversal results.

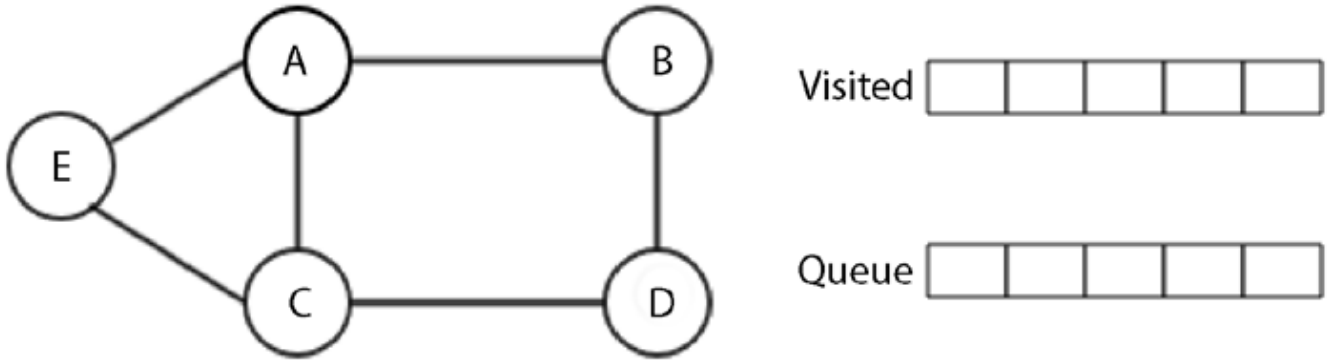


Fig.10. A sample graph

Breadth-first traversal

- We add the nodes in alphabetical order just to keep things simple in the queue, i.e., BCE. The A node is visited as shown in Fig.11.

- Next, we visit its first adjacent vertex, B, and add those adjacent vertices of vertex B that are not already added in the queue or not visited. In this case, we have to add the D vertex (since it has two vertices, A and D nodes, out of which A is already visited) to the queue, as shown in Fig.12

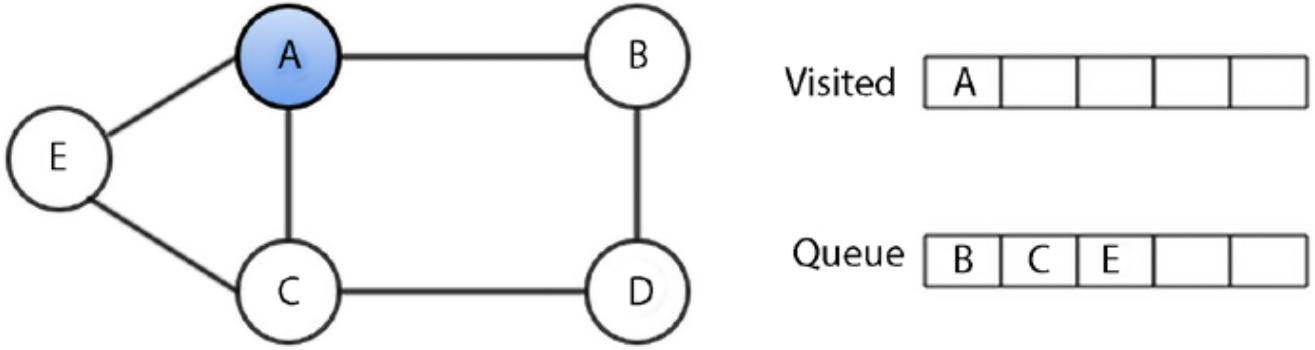


Fig.11. Node A is visited in breadth-first traversal

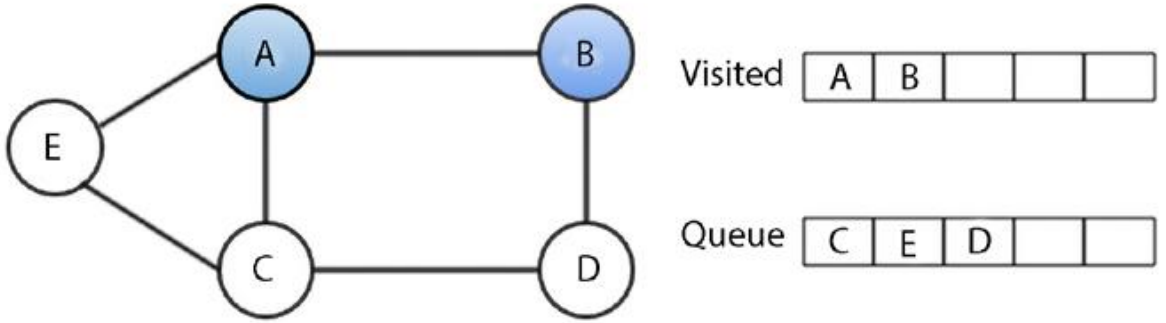
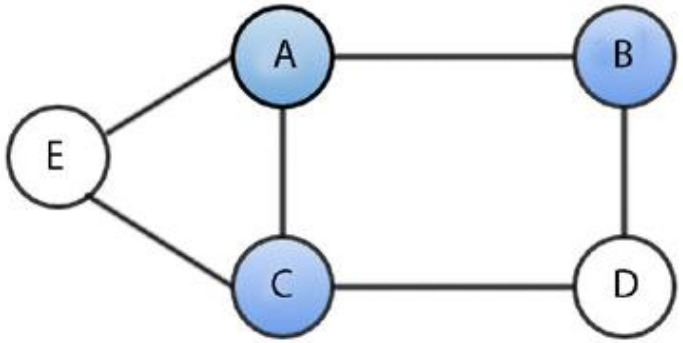


Fig.12. Node B is visited in breadth-first traversal

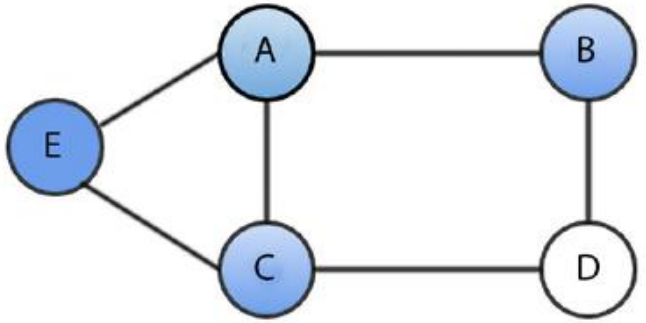
CMC UNIVERSITY

Breadth-first traversal



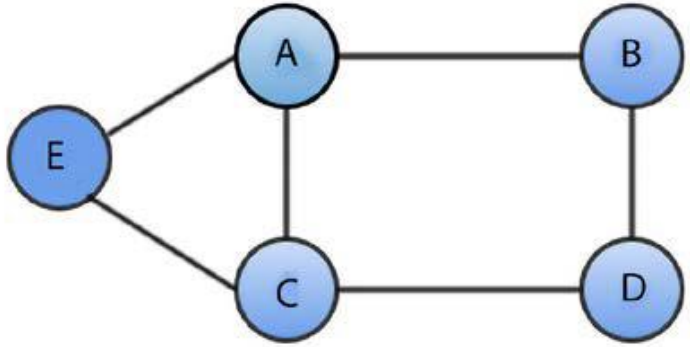
Visited | A | B | C | | |

Queue | E | D | | | |



Visited | A | B | C | E | |

Queue | D | | | | |



Visited | A | B | C | E | D |

Queue | | | | | |

The BFS algorithm for traversing the preceding graph visits the vertices in the order of A-B-C-E-D.

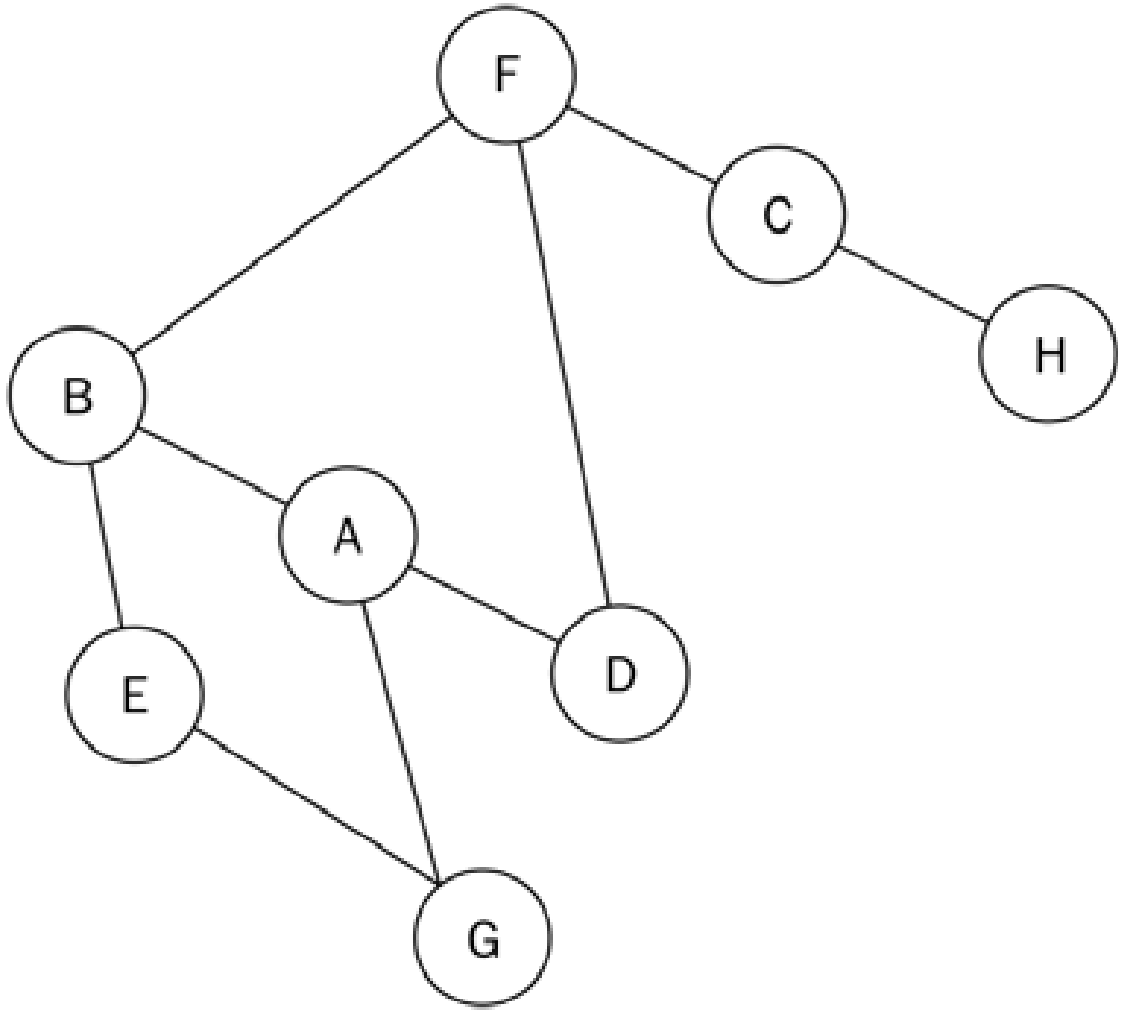
CMC UNIVERSITY

Breadth-first traversal

Example implement the following graph in python.

The adjacency list for the graph is as follows:

```
graph = dict()
graph['A'] = ['B', 'G', 'D']
graph['B'] = ['A', 'F', 'E']
graph['C'] = ['F', 'H']
graph['D'] = ['F', 'A']
graph['E'] = ['B', 'G']
graph['F'] = ['B', 'D', 'C']
graph['G'] = ['A', 'E']
graph['H'] = ['C']
```



CMC UNIVERSITY

Breadth-first traversal

The implementation of the BFS algorithm is as follows

```
from collections import deque

def breadth_first_search(graph, root):
    visited_vertices = list()
    graph_queue = deque([root])
    visited_vertices.append(root)
    node = root
    while len(graph_queue) > 0:
        node = graph_queue.popleft()
        adj_nodes = graph[node]

        remaining_elements = set(adj_nodes).difference(set(visited_
vertices))
```

CMC UNIVERSITY

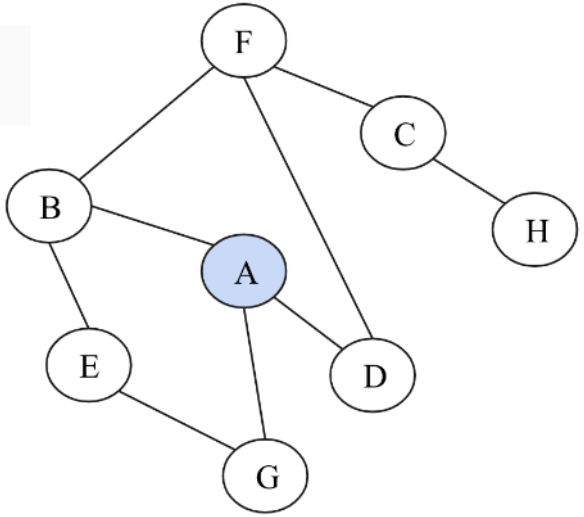
Breadth-first traversal

The implementation of the BFS algorithm is as follows

```
    if len(remaining_elements) > 0:
        for elem in sorted(remaining_elements):
            visited_vertices.append(elem)
            graph_queue.append(elem)

 return visited_vertices
```

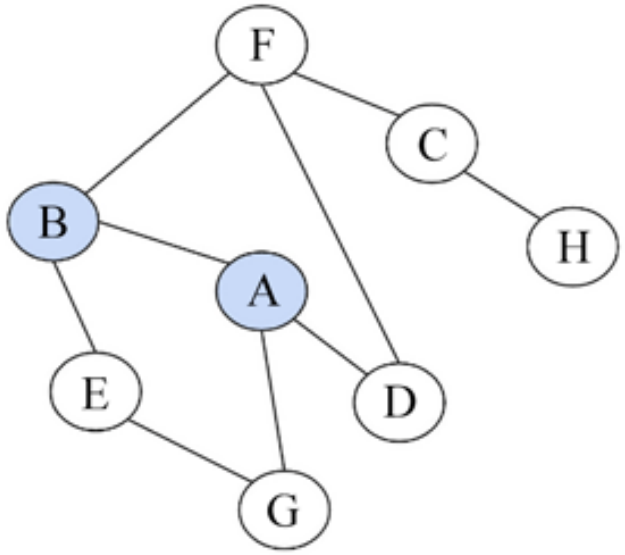
```
print(breadth_first_search(graph, 'A'))
```



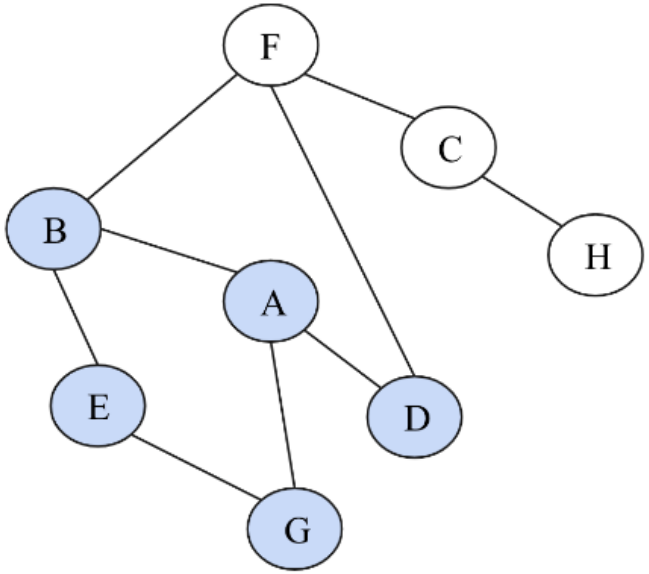
Visited	A						
Queue	B	D	G				

CMC UNIVERSITY

Breadth-first traversal



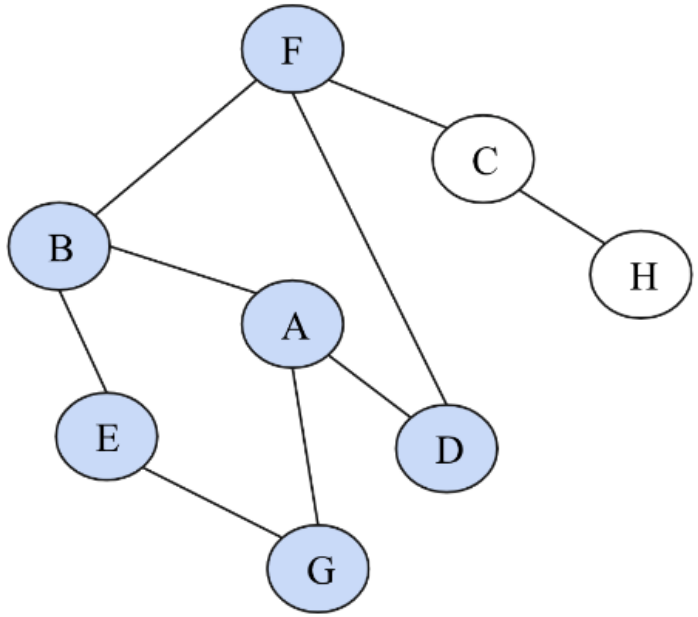
Visited	A	B					
Queue	D	G	E	F			



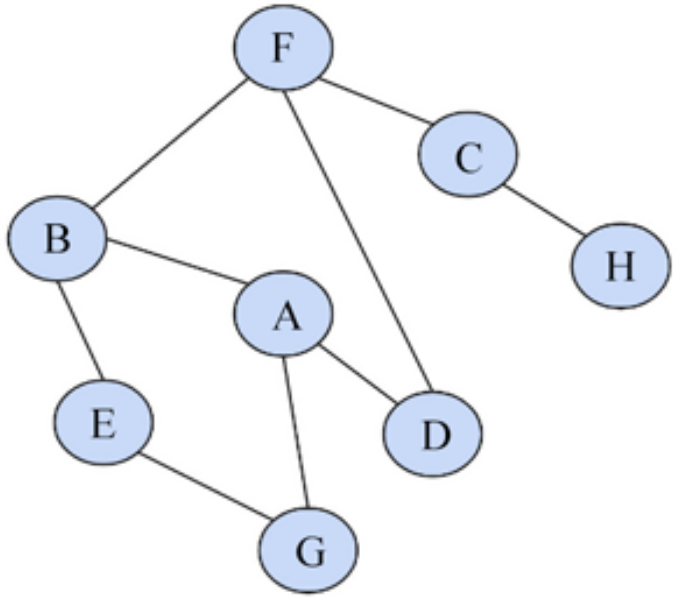
Visited	A	B	D	G	E		
Queue	F						

CMC UNIVERSITY

Breadth-first traversal



Visited	A	B	D	G	E	F	
Queue	C						



Visited	A	B	D	G	E	F	C	H
Queue								

A, B, D, G, E, F, C, H

CMC UNIVERSITY

Breadth-first traversal

- In the worst-case scenario, each node and the edge will need to be traversed, and hence each node will be enqueued and dequeued at least once.
- The time taken for each enqueue and dequeue operation is O(1), so the total time for this is O(V).
- The time spent scanning the adjacency list for every vertex is O(E).
- The total time complexity of the BFS algorithm is O(|V| + |E|), where |V| is the number of vertices or nodes, while |E| is the number of edges in the graph.

CMC UNIVERSITY

Breadth-first traversal

- The BFS algorithm is very useful for constructing the shortest path traversal in a graph with minimal iterations.
- As for some of the real-world applications of BFS, it can be used to create an efficient web crawler in which multiple levels of indexes can be maintained for search engines, and it can maintain a list of closed web pages from a source web page.
- BFS can also be useful for navigation systems in which neighboring locations can be easily retrieved from a graph of different locations.

CMC UNIVERSITY

Depth-first traversal

- The depth-first search (DFS) or traversal algorithm traverses the graph similar to how the preorder traversal algorithm works in trees.
- In the DFS algorithm, we traverse the tree in the depth of any particular path in the graph. As such, child nodes are visited first before sibling nodes.

Algorithm:

- Firstly, visit the root node,
- Then, see all the adjacent vertices of the current node, visit one of the adjacent nodes:

- If the edge leads to a visited node, backtrack to the current node.
- If the edge leads to an unvisited node, go to that node and continue processing from that node.

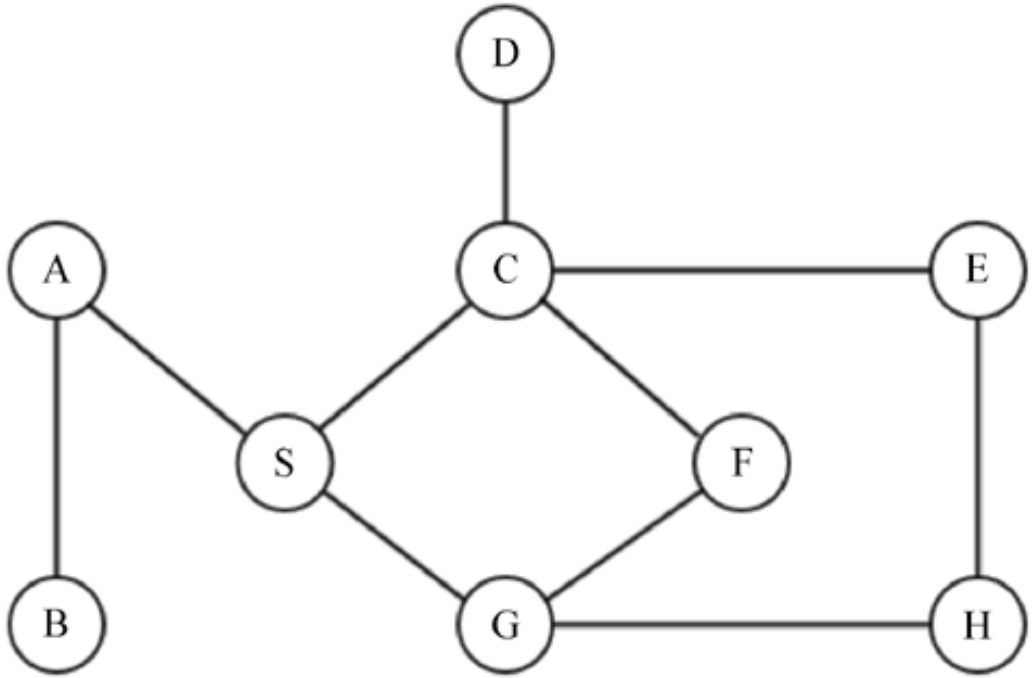
- Continue the same process until reach a dead end when there is no unvisited node; backtrack to previous nodes, and stop when reach the root node while backtracking

CMC UNIVERSITY

Depth-first traversal

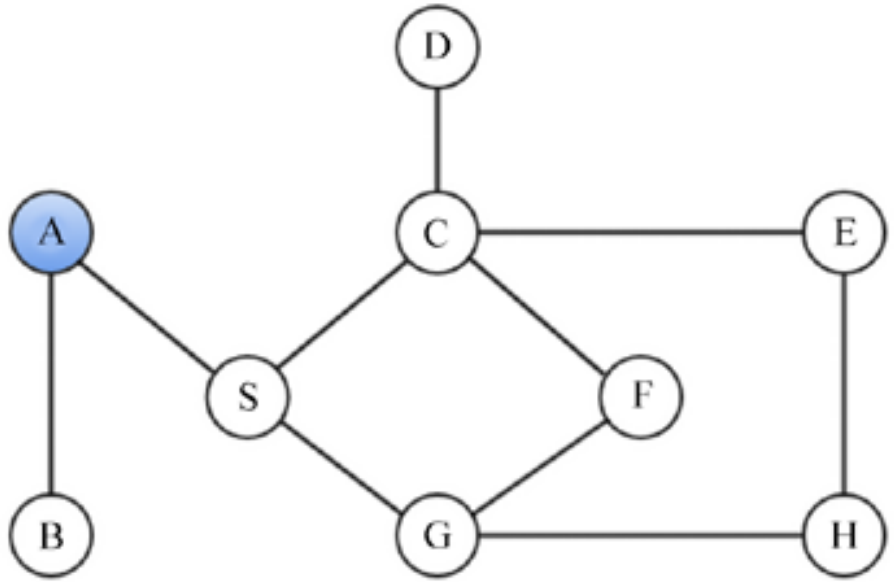
Example: for the following graph

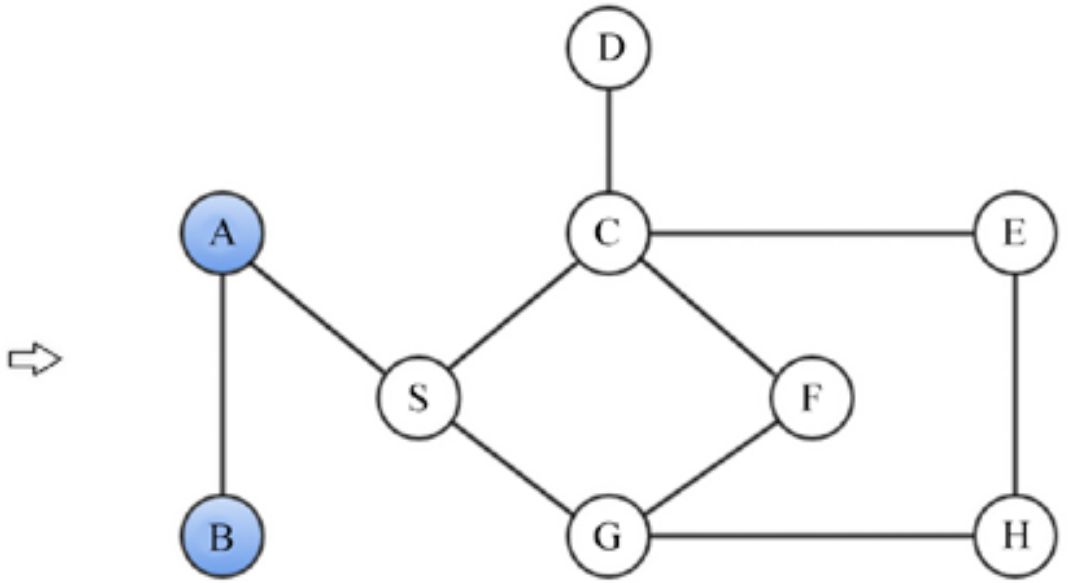
- Start by visiting the A node, and then look at the neighbors of the A vertex, then a neighbor of that neighbor, and so on.
- Visit one of neighbors of A, B (in this example, sort alphabetically; however, any neighbor can be added)

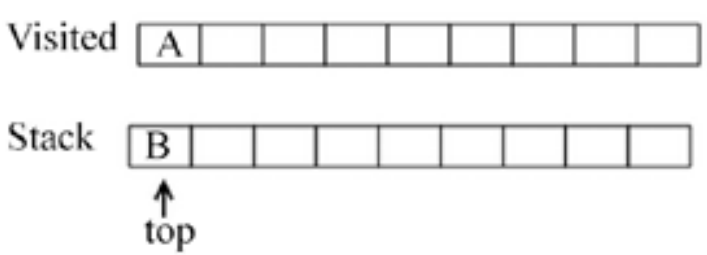


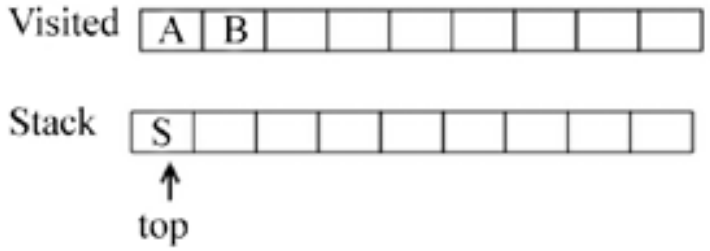
Depth-first traversal

Example: for the following graph



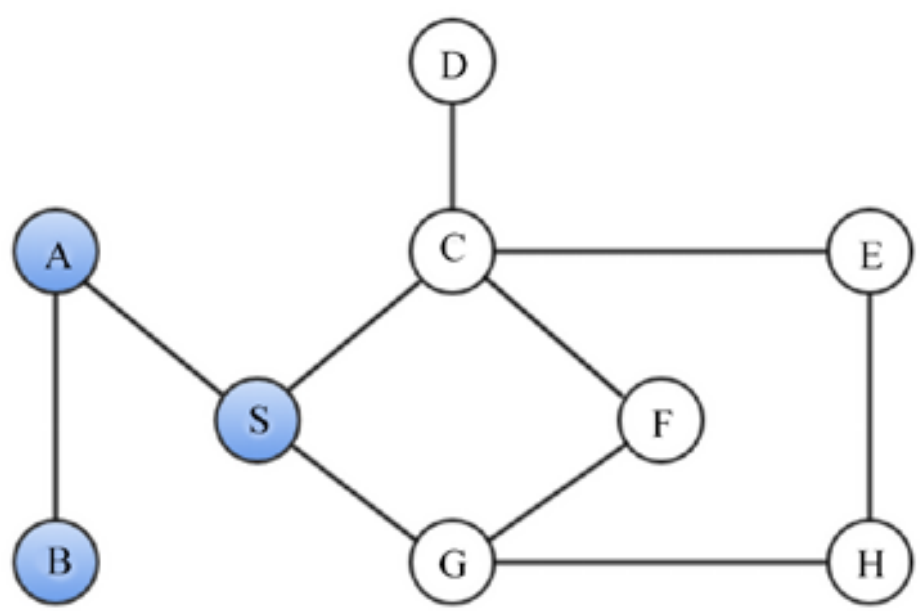


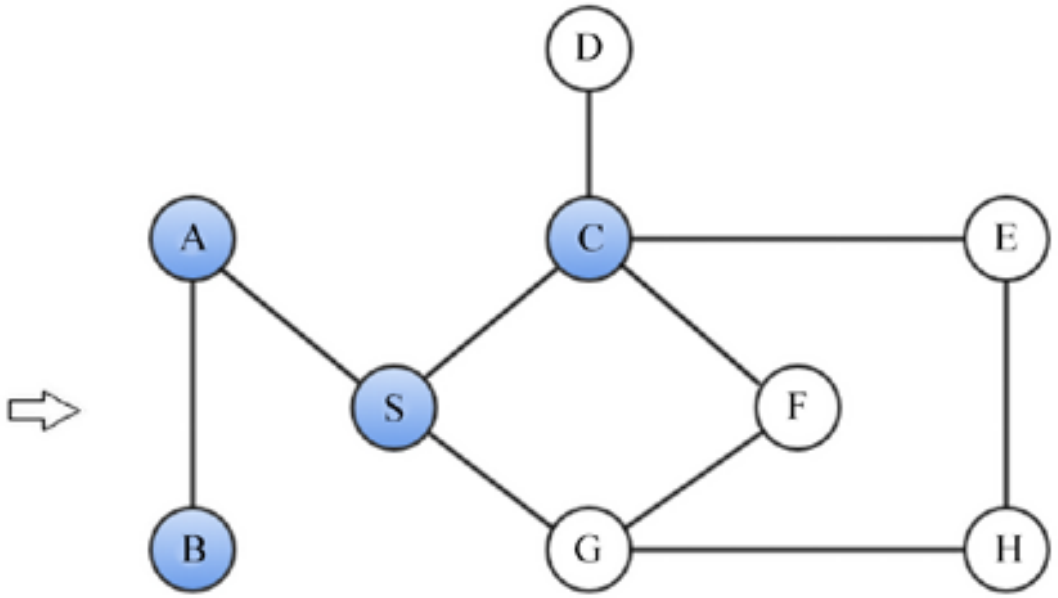


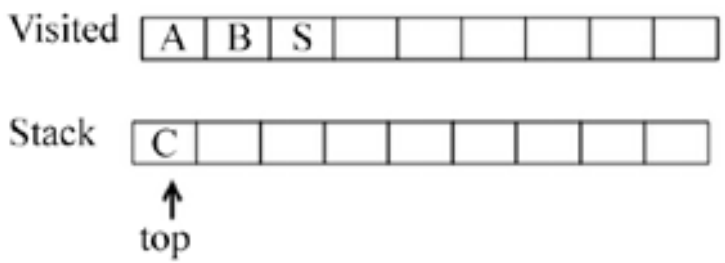


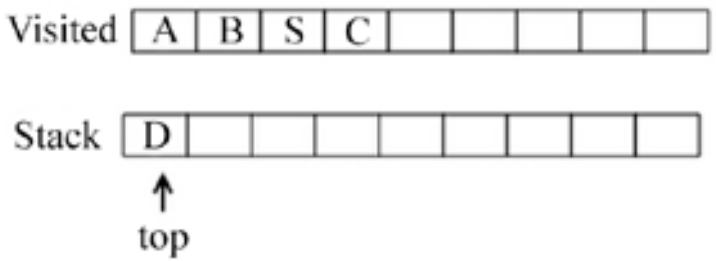
Depth-first traversal

Example: for the following graph



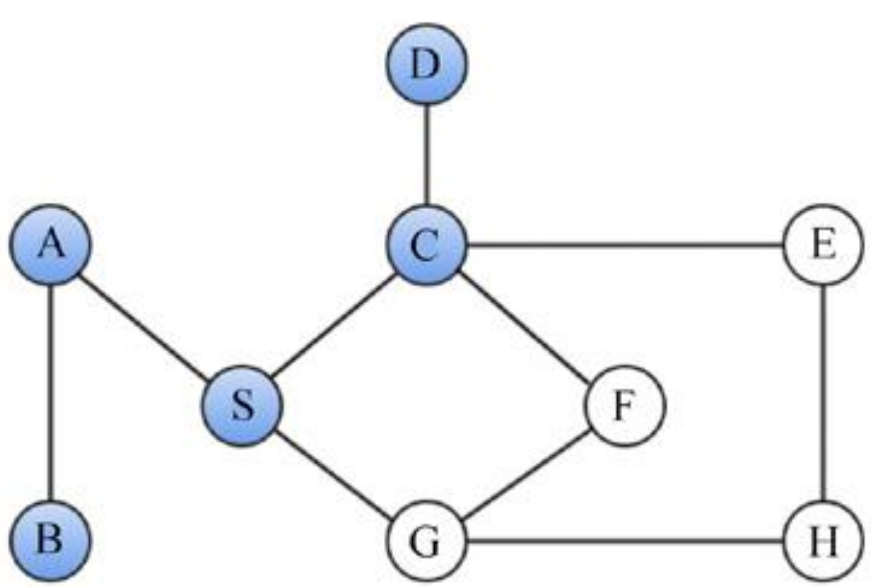


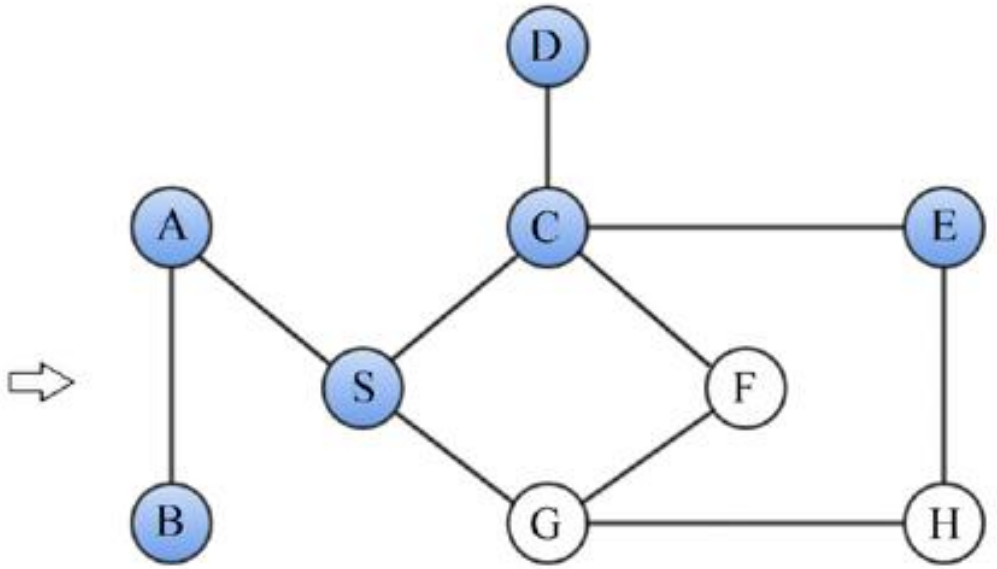


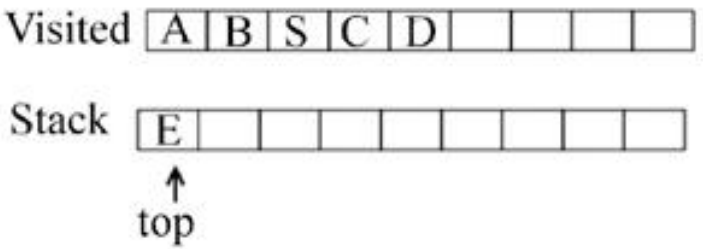


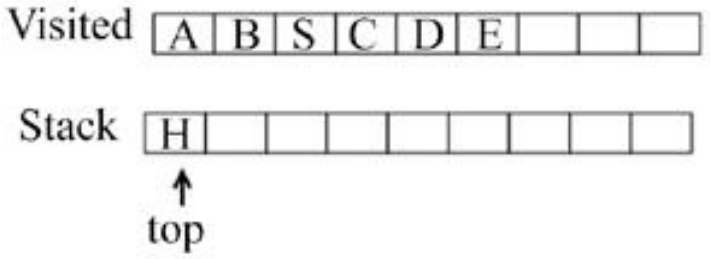
Depth-first traversal

Example: for the following graph



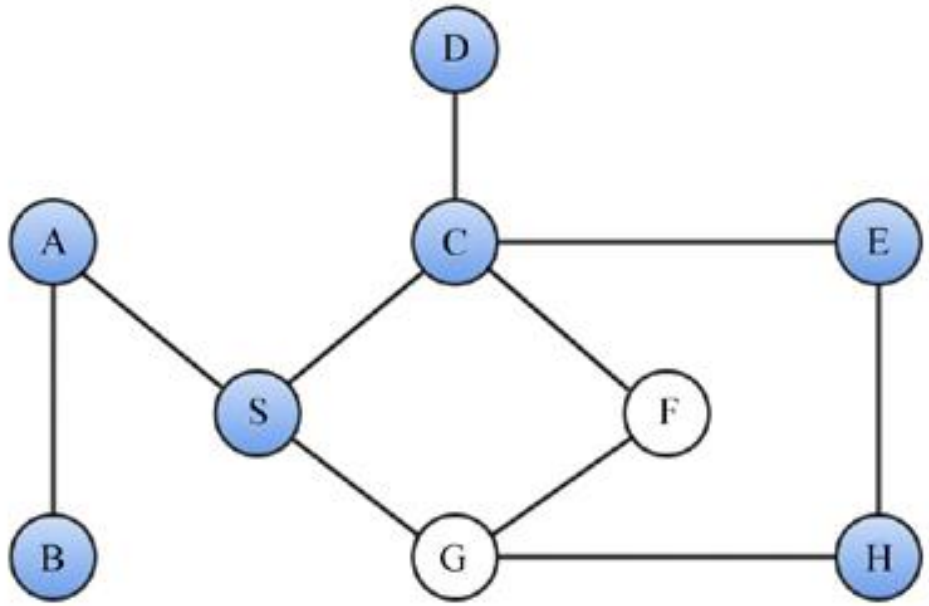


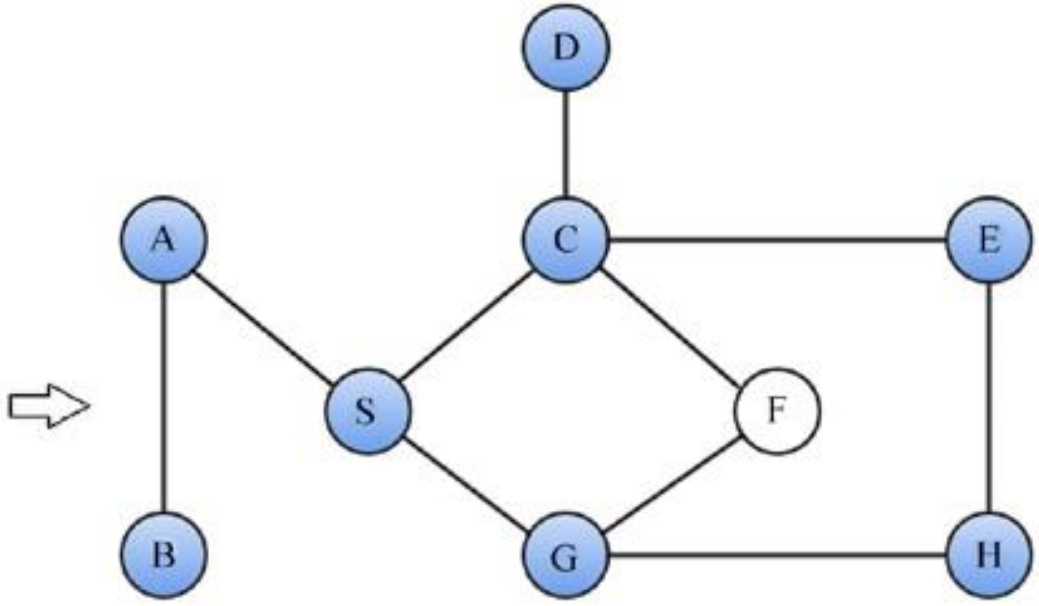


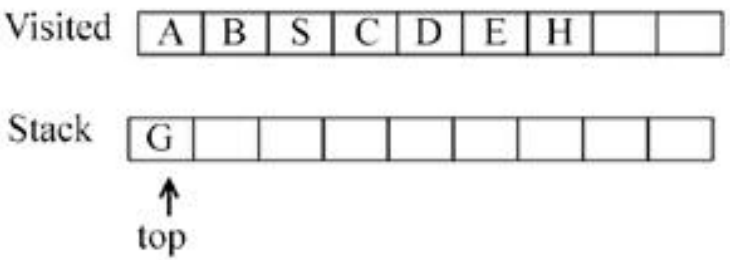


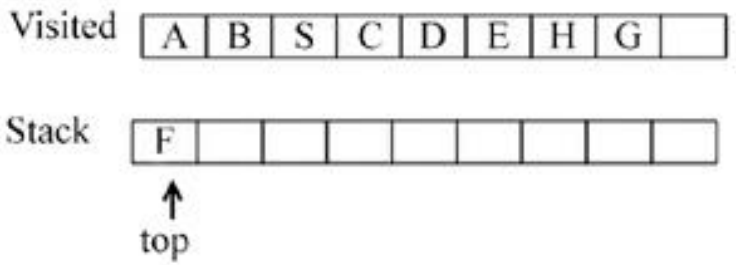
Depth-first traversal

Example: for the following graph



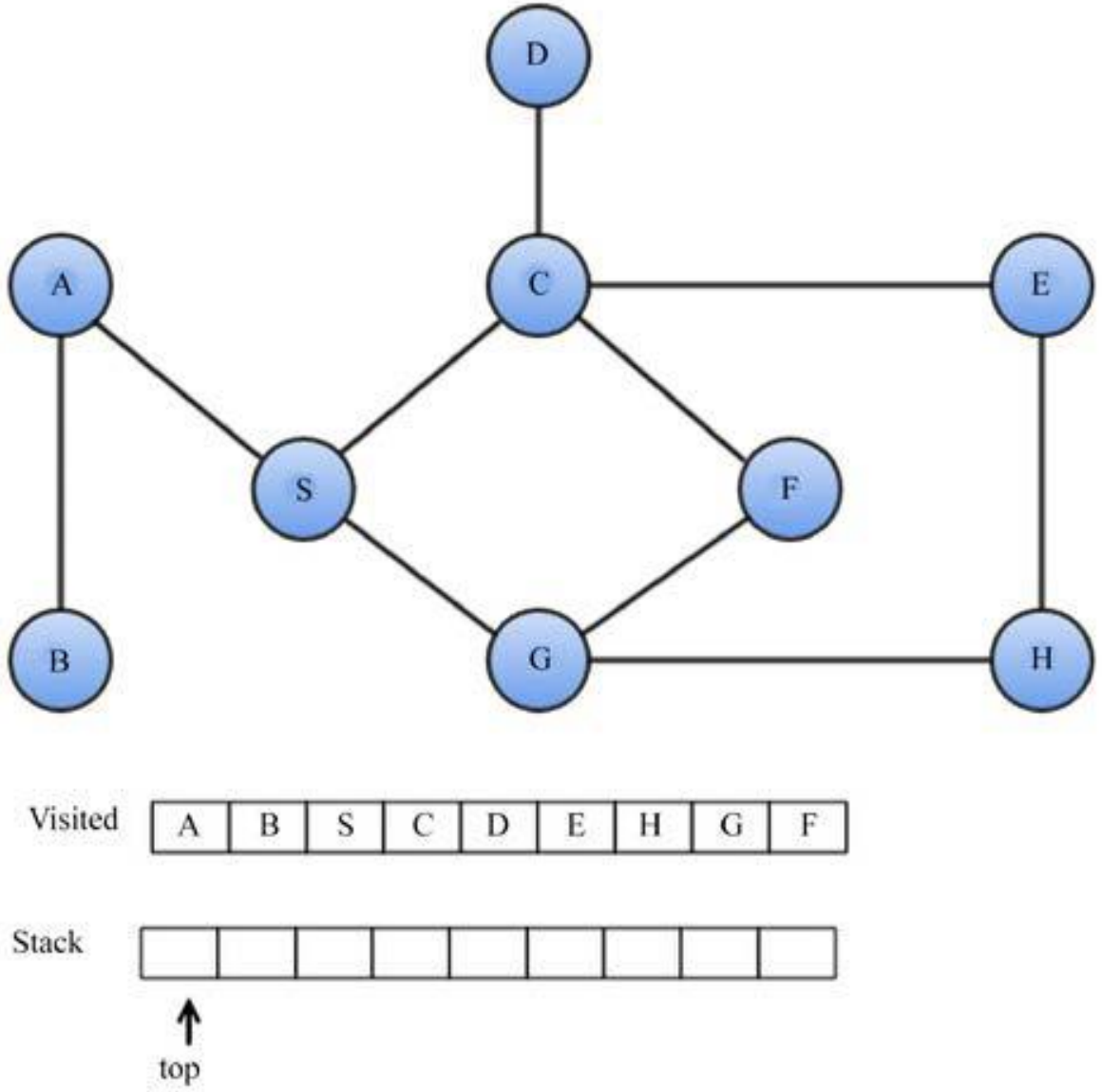






Depth-first traversal

The output of the DFS traversal is
A-B-S-C-D-E-H-G-F.



CMC UNIVERSITY

Depth-first traversal

The implementation of the DFS algorithm is as follows

```
1  def depth_first_search(graph, root):
2      visited_vertices = []
3      graph_stack = []
4
5      graph_stack.append(root)
6      node = root
7      while graph_stack:
8          if node not in visited_vertices:
9              visited_vertices.append(node)
10
11         adj_nodes = graph[node]
12         if set(adj_nodes).issubset(set(visited_vertices)):
13             graph_stack.pop()
14             if len(graph_stack) > 0:
15                 node = graph_stack[-1]
16             continue
17         else:
```

CMC UNIVERSITY

Depth-first traversal

```
17          else:
18              remaining_elements = set(adj_nodes).difference(set
                    (visited_vertices))
19          first_adj_node = sorted(remaining_elements)[0]
20          graph_stack.append(first_adj_node)
21          node = first_adj_node
22      return visited_vertices
```

```
23
24  graph = dict()
25  graph['A'] = ['B', 'S']
26  graph['B'] = ['A']
27  graph['S'] = ['A','G','C']
28  graph['D'] = ['C']
29  graph['G'] = ['S','F','H']
30  graph['H'] = ['G','E']
31  graph['E'] = ['C','H']
32  graph['F'] = ['C','G']
33  graph['C'] = ['D','S','E','F']
34
35  print(depth_first_search(graph, 'A'))
```

```
['A', 'B', 'S', 'C', 'D', 'E', 'H', 'G', 'F']
```

CMC UNIVERSITY

Depth-first traversal

- The time complexity of DFS is $O(V+E)$ when we use an adjacency list, and $O(V^2)$ when we use an adjacency matrix for graph representation.

- The time complexity of DFS with the adjacency list is lower because getting the adjacent nodes is easier, whereas it is not efficient with the adjacency matrix.

- DFS can be applied to solving maze problems, finding connected components, cycle detection in graphs, and finding the bridges of a graph, among other use cases.

We have discussed very important graph traversal algorithms; now let us discuss some more useful graph-related algorithms for finding the spanning tree from the given graph. Spanning trees are useful for several real-world problems such as the traveling salesman problem.

CMC UNIVERSITY

SHORTEST PATHS

CMC UNIVERSITY

Shortest paths

The shortest path from the source node to the destination node:

- The path with the lowest number of edges between the source node and the destination node, for an unweighted graph,

- The path with the lowest value of total weight of passing through a set of edges, for a weighted graph.

CMC UNIVERSITY

Shortest paths

Shortest paths in a Weighted Graph

Let G be a weighted graph. The ***length*** (or weight) of a path is the sum of the weights of the edges of P. That is, if P = ((v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)), then the length of P, denoted w(P), is defined as

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

The ***distance*** from a vertex u to a vertex v in G, denoted d(u,v), is the length of a minimum-length path (also called ***shortest path***) from u to v, if such a path exists.

Dijkstra's algorithm

- Dijkstra's algorithm shares some commonality with depth first search. The algorithm proceeds as depth first search proceeds, but starts with a single source s eventually visiting every node within the graph.

- Each vertex, v, in the graph is assigned a cost which is the sum of the weighted edges on the path from the source to v. Initially the source vertex is assigned cost 0. All other vertices are initially assigned infinite cost.

(Anything greater than the sum of all weights in the graph can serve as an infinite value)

- There are two sets that Dijkstra's algorithm maintains.

+ The first is an *unvisited set*. This is a set of vertices that yet need to be considered while looking for minimum cost paths.

+ The unvisited set serves the same purpose as the stack when performing depth first search on a graph. The visited set is the other set used by the algorithm. The visited set contains all vertices which already have their minimum cost and path computed. The visited set serves the same purpose as the visited set in depth first search of a graph.

CMC UNIVERSITY

Dijkstra's algorithm

1. Remove the vertex we'll call current from the unvisited set with the least cost. All other paths to this vertex must have greater cost because otherwise they would have been in the unvisited set with smaller cost.

2. Add current to the visited set.

3. For every vertex, adjacent, that is adjacent to current, check to see if adjacent is in the visited set or not. If adjacent is in the visited set, then we know the minimum cost of reaching this vertex from the source so don't do anything.

4. If adjacent is not in the visited set, compute a new cost for arriving at adjacent by traversing the edge, e, from current to adjacent. A new cost can be found by adding the cost of getting to current and e's weight. If this new cost is better than the current cost of getting to adjacent, then update adjacent's cost and remember that current is the previous vertex of adjacent. Also, add adjacent to the unvisited set.

Dijkstra's algorithm

Example 1: For the graph as shown below

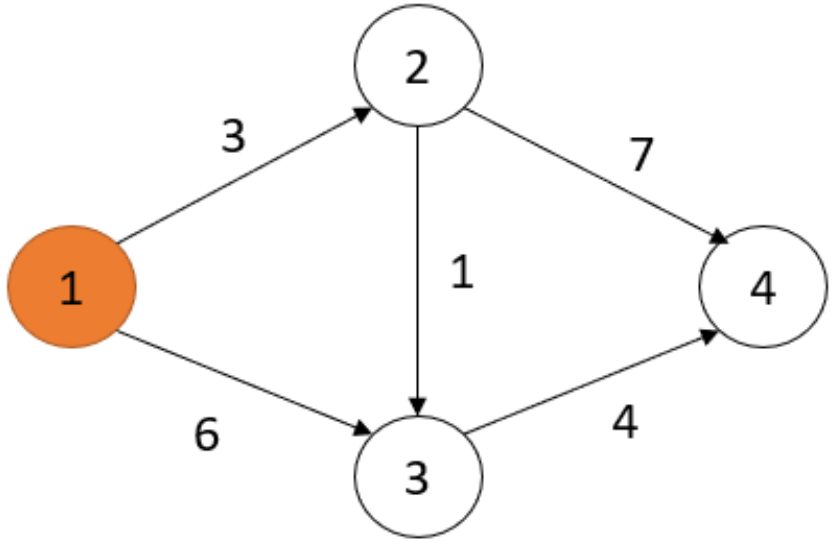
We will need to initialize two arrays with the following meaning:

- dist[u]: shortest path length from source vertex s to vertex u.

Initially dist[u] = $\propto$ or all u, separately dist[s] = 0

- visited[u]: marks whether vertex u has been considered or not. Initially the elements in the array have the value false

dist = [0, $\propto$, $\propto$, $\propto$], visited = [0, 0, 0, 0]

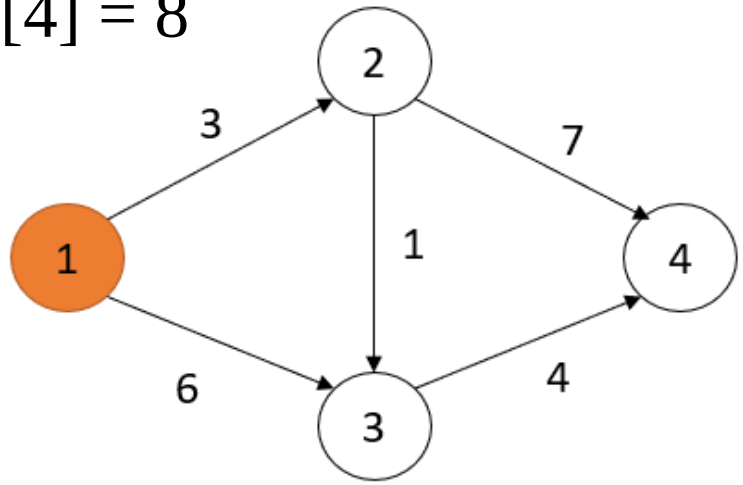


Dijkstra's algorithm

Example 1:

dist = [0, ∝, ∝, ∝], visited = [0, 0, 0, 0]

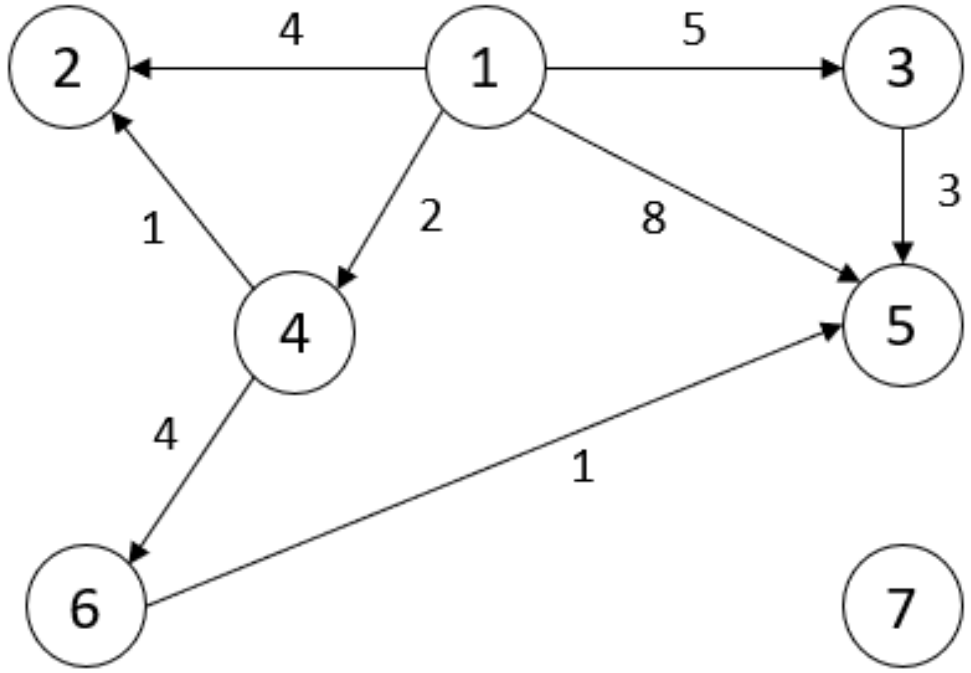
1) u = 1, v = 2, dist[2] = dist[1] + (1, 2) = 3 < dist[2] = ∝, dist[2] = 3,
   u = 1, v = 3, dist[3] = dist[1] + (1, 3) = 6 < dist[3] = ∝, dist[3] = 6
   dist = [0, 3, 6, ∝], visited = [1, 0, 0, 0]
2) u = 2, v = 3, dist[3] = dist[2] + (2, 3) = 4 < dist[3] = 6, dist[3] = 4,
   u = 2, v = 4, dist[4] = dist[2] + (2, 4) = 10 < dist[4] = ∝, dist[4] = 10
   dist = [0, 3, 4, 10], visited = [1, 1, 0, 0]
3) u = 3, v = 4, dist[4] = dist[3] + (3, 4) = 8 < dist[4] = 10, dist[4] = 8
   dist = [0, 3, 4, 8], visited = [1, 1, 1, 0]
4) u = 4,
   dist = [0, 3, 4, 8], visited = [1, 1, 1, 1]



CMC UNIVERSITY

Dijkstra's algorithm

Example 2:



Input	Output
7 8 1 1 2 4 1 3 5 1 4 2 4 2 1 4 6 4 3 5 3 6 5 1 1 5 8	0 3 5 2 7 6 -1

CMC UNIVERSITY

Dijkstra's algorithm

```
1  class Graph:
2      def __init__(self, vertices):
3          self.V = vertices
4          self.graph = [[0 for _ in range(vertices)] for _ in
               range(vertices)]
5
6      def add_edge(self, u, v, w):
7          self.graph[u][v] = w
8          self.graph[v][u] = w
9
10     def find_min_dist(self, dist, visited):
11         min_dist = float('inf')
12         min_dist_vertex = -1
13
14         for v in range(self.V):
15             if not visited[v] and dist[v] < min_dist:
16                 min_dist = dist[v]
17                 min_dist_vertex = v
18         return min_dist_vertex
```

CMC UNIVERSITY

Dijkstra's algorithm

```
19
20      def dijkstra_custom(self, s):
21          dist = [float('inf')] * self.V
22          visited = [False] * self.V
23
24          dist[s] = 0
25
26          for _ in range(self.V):
27              u = self.find_min_dist(dist, visited)
28              visited[u] = True
29
30              for v in range(self.V):
31                  if not visited[v] and self.graph[u][v] > 0:
32                      if dist[u] + self.graph[u][v] < dist[v]:
33                          dist[v] = dist[u] + self.graph[u][v]
34
35          print("Vertex \t Distance from Source s")
36          for i in range(self.V):
37              print(f"{i} \t {dist[i]}")
```

CMC UNIVERSITY

Dijkstra's algorithm

```
38
39  g = Graph(4)
40  g.add_edge(0, 1, 3)
41  g.add_edge(0, 2, 6)
42  g.add_edge(1, 2, 1)
43  g.add_edge(1, 3, 7)
44  g.add_edge(2, 3, 4)
45
46  g.dijkstra_custom(0)
```

```
Vertex   Distance from Source s
0     0
1     3
2     4
3     8
>
```

The complexity of Dijkstra's Algorithm is $O(n^2)$, using a priority queue, Dijkstra's Algorithm will run in O(nlogn) time

CMC UNIVERSITY

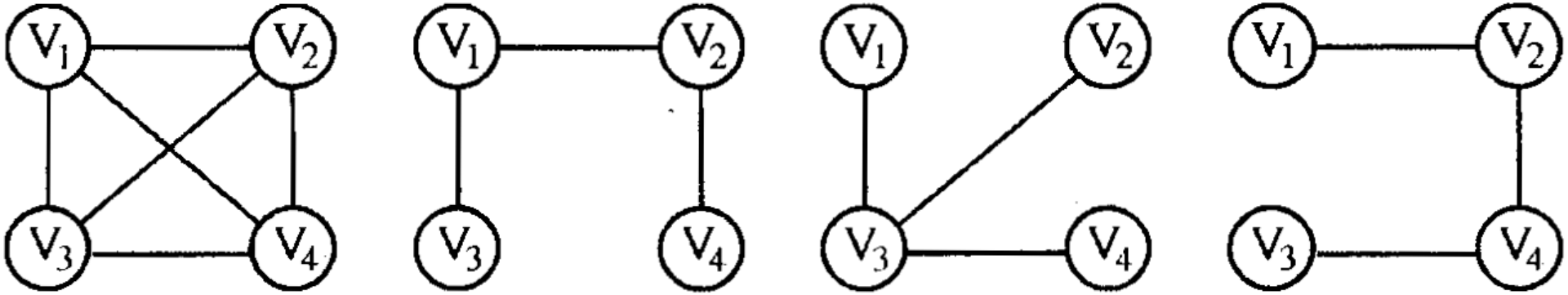
MINIMUM SPANNING TREE

Spanning tree

If the graph G is connected then a breadth or depth search, starting from any vertex, also allows to visit all vertices of G. In this case, the edges of G are divided into two sets:

The set T includes all the edges used or traversed in the search, and the set B includes the remaining edges.

All the edges in T together with their respective vertices will form a tree that includes every vertex (node) of G. Such a tree is called a spanning tree of G.



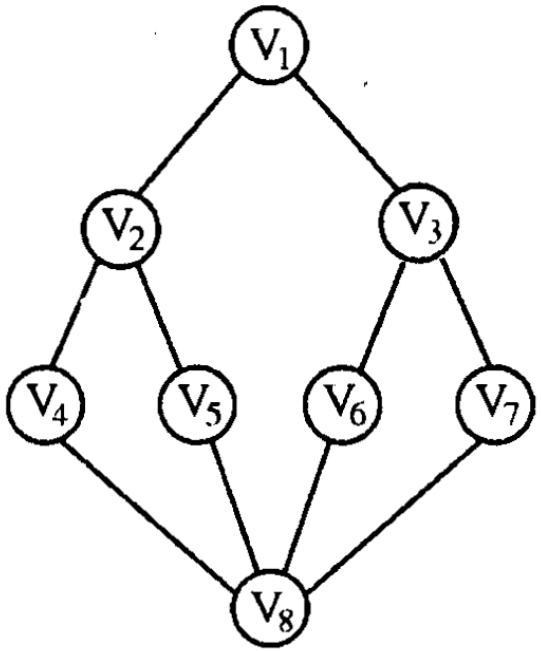
a) b) c)

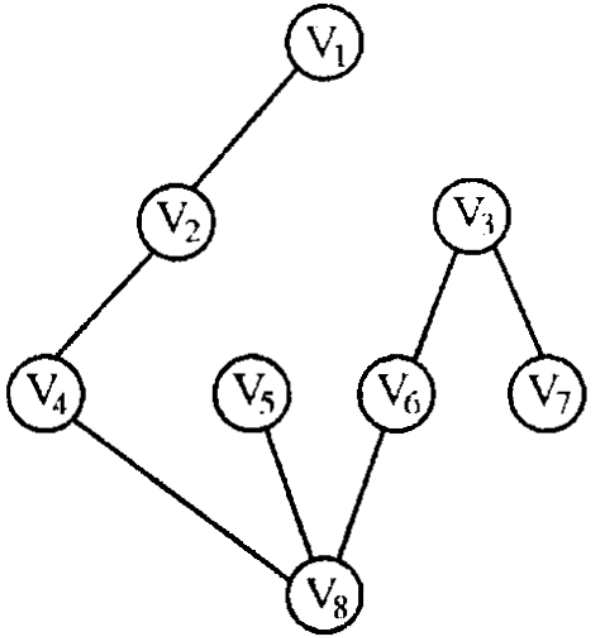
Spanning tree

Depending on the DFS or BFS method used, the corresponding spanning tree will be called a Depth-First Spanning Tree or a Breadth-First Spanning Tree.

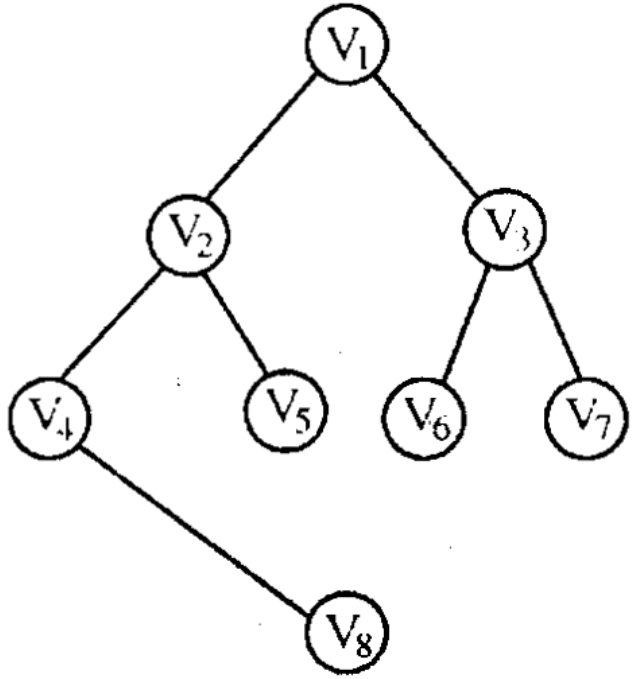
a) DFS Depth-First Spanning Tree (V1)

b) BFS Breadth-First Spanning Tree (V1)





a)

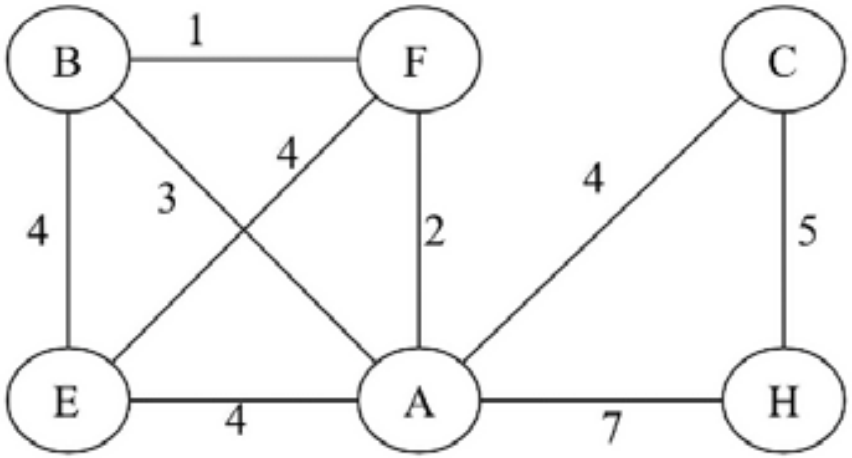


b)

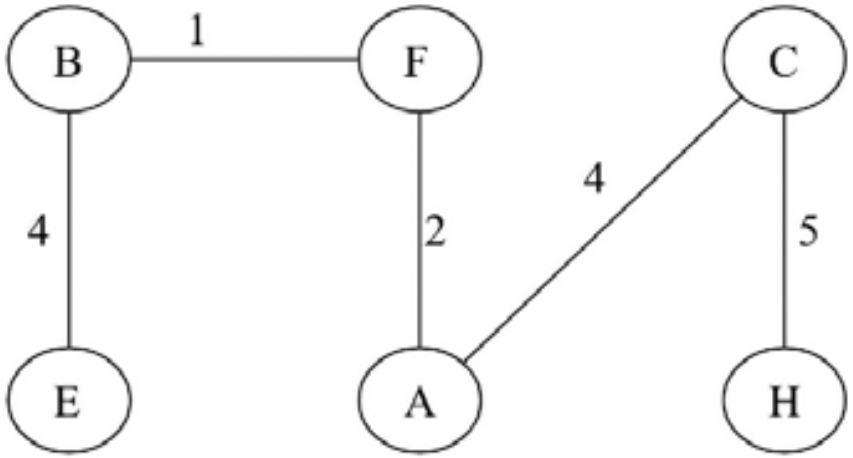
Minimum spanning tree

A Minimum Spanning Tree (MST) is a spanning tree of the connected graph with an edge-weighted graph that connects all the nodes of the graph, with the lowest possible total edge weights and no cycle.

Given a connected graph G, where G = (V, E) with real-valued edge weights, an MST is a subgraph with a subset of the edges $T \subseteq E$ so that the sum of edge weights is minimum and there is no cycle.



A graph



A minimum spanning tree

CMC UNIVERSITY

Minimum spanning tree

There are many possible spanning trees that can connect all the nodes of the graph without any cycle, but the the minimum weight spanning tree is a spanning tree that has the lowest total edge weight (also called cost) among all other possible spanning trees.

The MST has diverse real-world applications. They are mainly used in network design for road congestion, hydraulic cables, electric cable networks, and even cluster analysis.

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

- **Kruskal's algorithm** is a widely used algorithm for finding the spanning tree from a given weighted, connected, and undirected graph.
- It is based on the greedy approach: firstly find the edge with the lowest weight and add it to the tree, and then in each iteration, add the edge with the lowest weight to the spanning tree so that we do not form a cycle.
- In this algorithm, initially, treat all the vertices of the graph as a separate tree, and then in each iteration, select edge with the lowest weight in such a way that it does not form a cycle. These separate trees are combined, and it grows to form a spanning tree. This process is repeated until all nodes are processed.

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

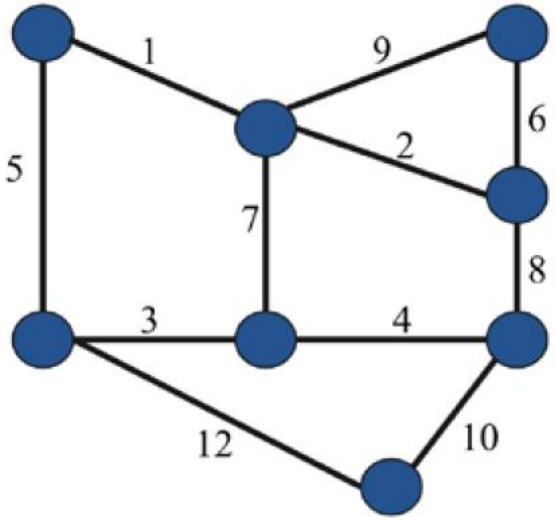
The algorithm works as follows:

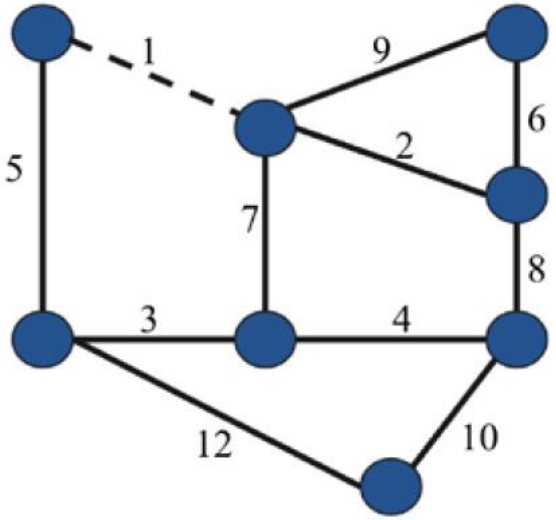
1. Initialize an empty MST (M) with zero edges

2. Sort all the edges according to their weights

3. For each edge from the sorted list, we add them one by one to the MST (M) in such a way that it does not form a cycle.

Example 1:

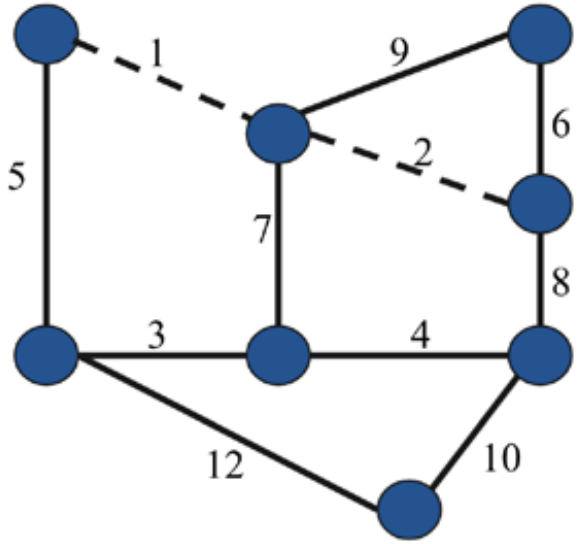


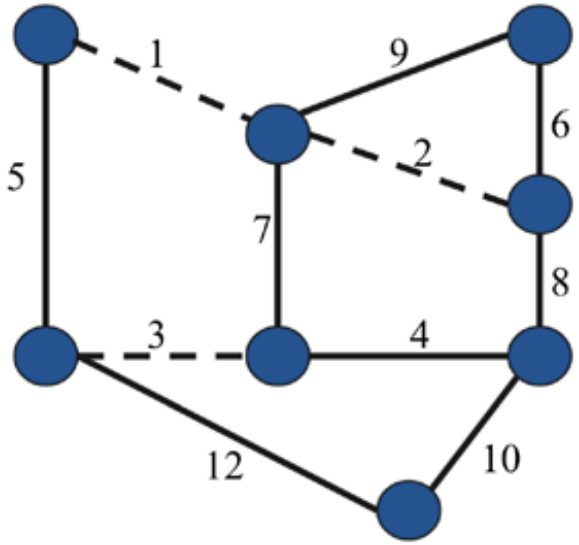


Selecting the first edge with the lowest weight in the spanning tree

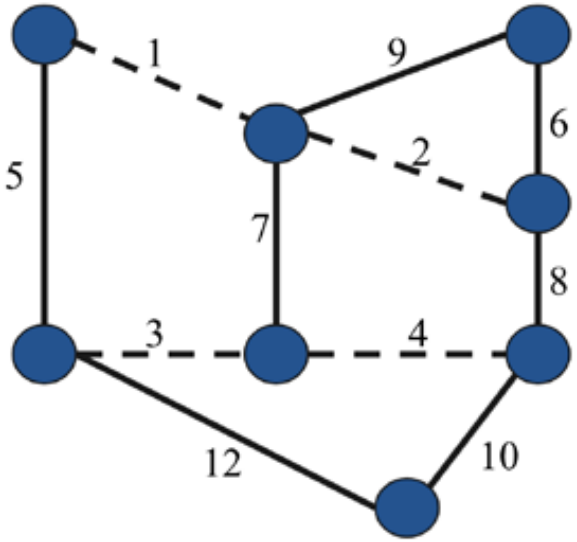
CMC UNIVERSITY

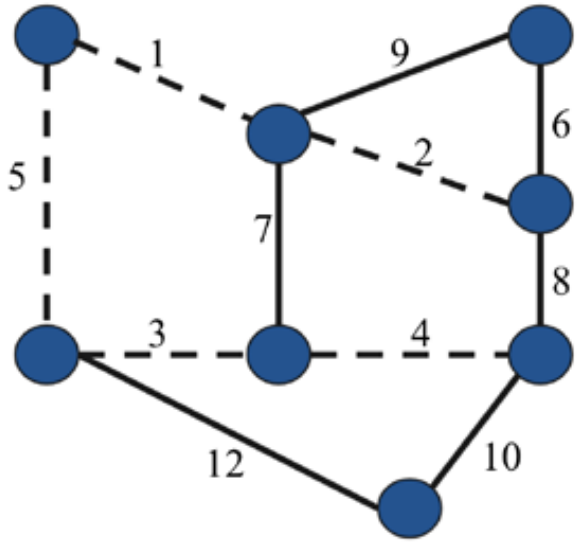
Kruskal's Minimum Spanning Tree algorithm





Selecting edges with weights 2 and 3 in the spanning tree

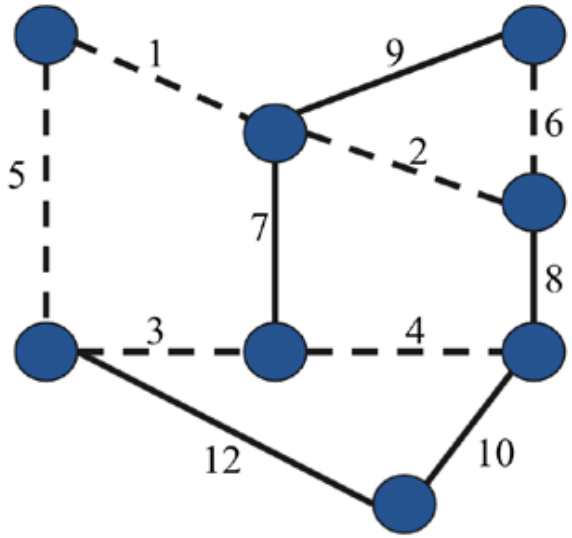


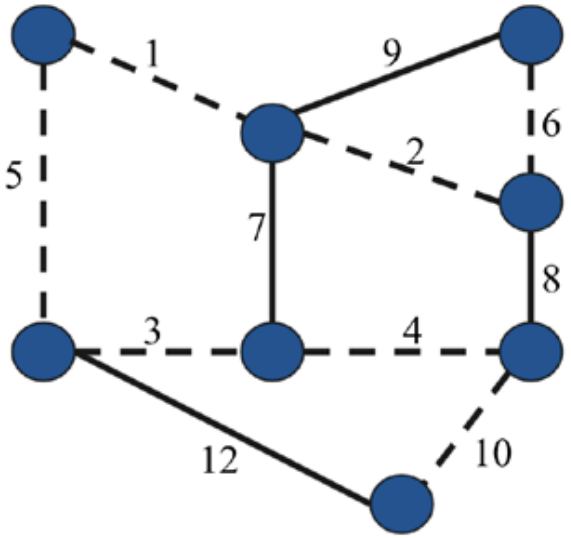


Selecting edges with weights 4 and 5 in the spanning tree

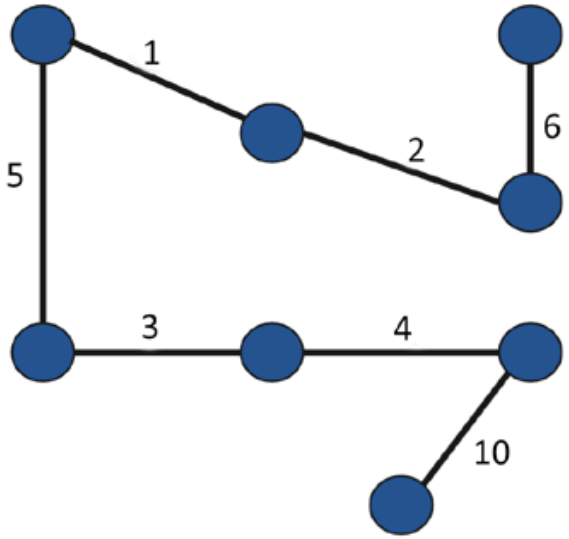
CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm





Selecting edges with weights 6 and 10 in the spanning tree

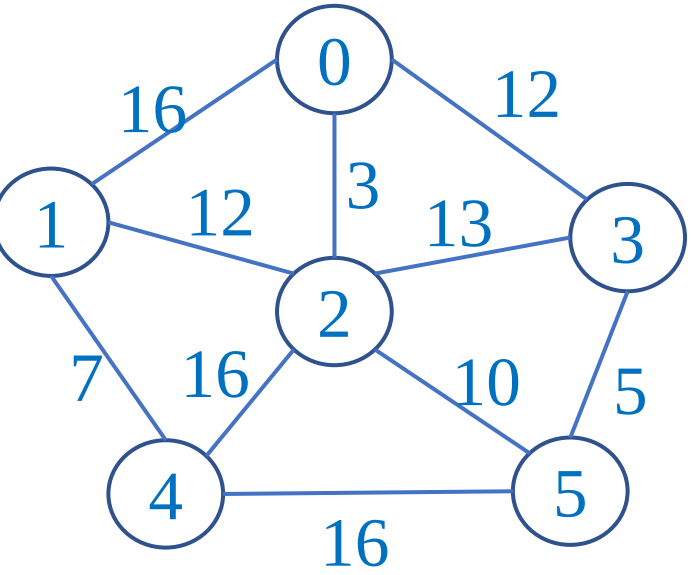


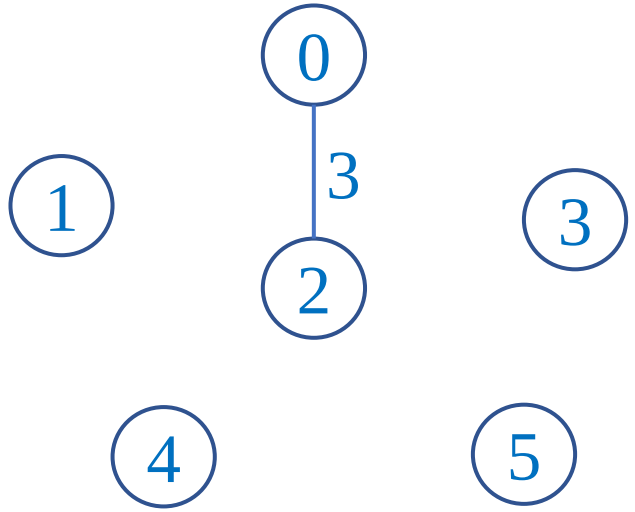
The final spanning tree created using Kruskal's algorithm

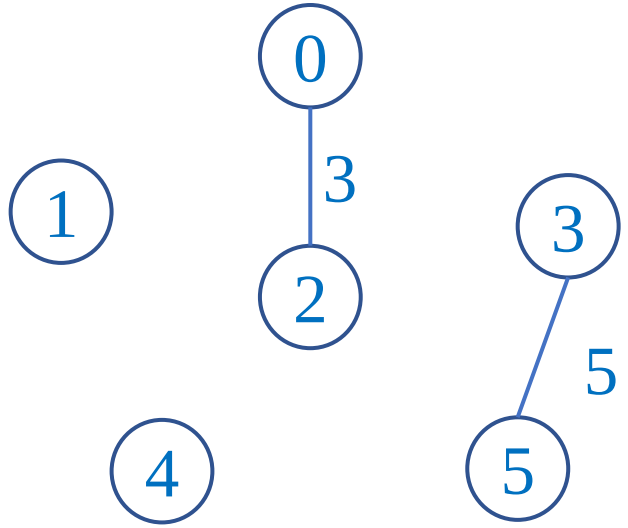
CMC UNIVERSITY

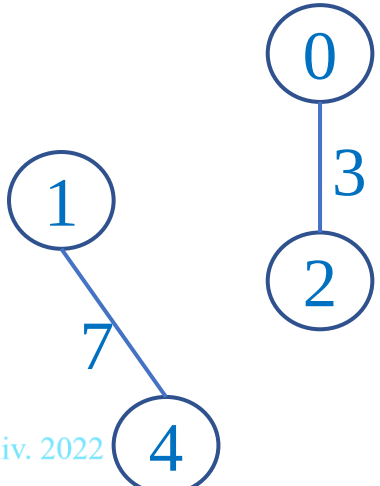
Kruskal's Minimum Spanning Tree algorithm

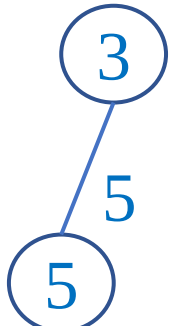
Example 2

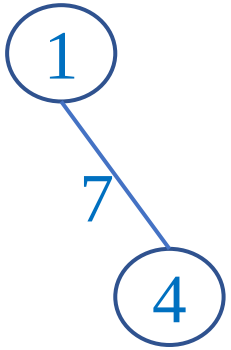


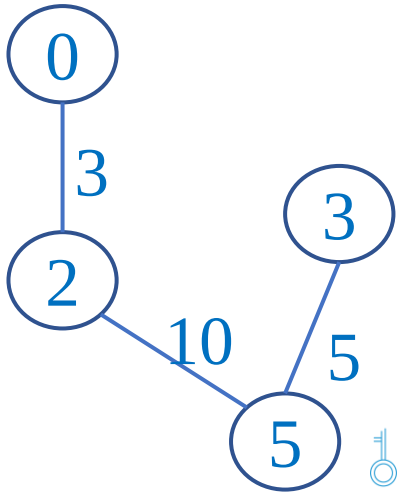


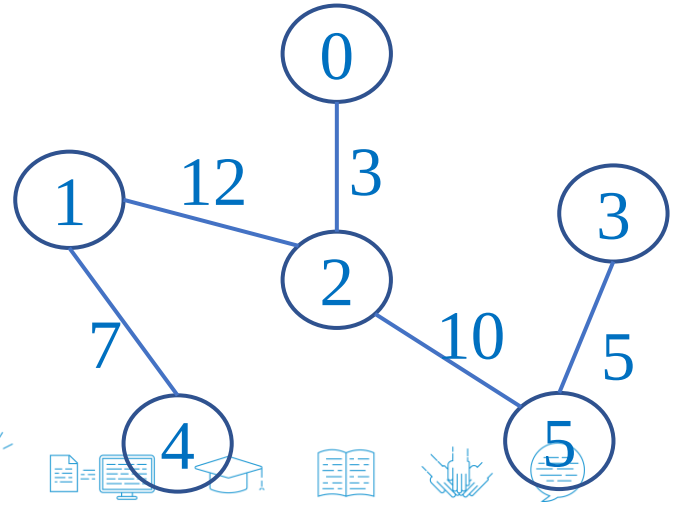












CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append((u, v, w))

    def find_parent(self, parent, i):
        if parent[i] == i:
            return i
        return self.find_parent(parent, parent[i])

    def union(self, parent, rank, x, y):
        x_root = self.find_parent(parent, x)
        y_root = self.find_parent(parent, y)

        if rank[x_root] < rank[y_root]:
            parent[x_root] = y_root
        elif rank[x_root] > rank[y_root]:
```

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

```
20          elif rank[x_root] > rank[y_root]:
21              parent[y_root] = x_root
22          else:
23              parent[y_root] = x_root
24              rank[x_root] += 1
25
26      def kruskal(self):
27          result = []
28          self.graph = sorted(self.graph, key=lambda item:item[2])
29
30          parent = []
31          rank = []
32
33          for node in range(self.V):
34              parent.append(node)
35              rank.append(0)
```

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

```
36
37          i = 0
38          while len(result) < self.V - 1:
39              u, v, w = self.graph[i]
40              i += 1
41              x = self.find_parent(parent, u)
42              y = self.find_parent(parent, v)
43
44              if x != y:
45                  result.append((u, v, w))
46                  self.union(parent, rank, x, y)
47
48          print("Edges in the Minimum Spanning Tree:")
49          for u, v, weight in result:
50              print(f"{u} -- {v} == {weight}")
```

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

```
51
52  g = Graph(6)
53  g.add_edge(0, 1, 16)
54  g.add_edge(0, 2, 3)
55  g.add_edge(0, 3, 12)
56  g.add_edge(1, 2, 12)
57  g.add_edge(1, 4, 7)
58  g.add_edge(2, 3, 13)
59  g.add_edge(2, 4, 16)
60  g.add_edge(2, 5, 10)
61  g.add_edge(3, 5, 5)
62  g.add_edge(4, 5, 16)
63
64  g.kruskal()
```

```
Edges in the Minimum Spanning Tree:
0 -- 2 == 3
3 -- 5 == 5
1 -- 4 == 7
2 -- 5 == 10
1 -- 2 == 12
```

CMC UNIVERSITY

Kruskal's Minimum Spanning Tree algorithm

Kruskal's algorithm has many real-world applications, such as solving the traveling salesman problem (TSP), in which starting from one city, we have to visit all the different cities in a network with the minimum total cost and without visiting the same city twice.

There are many other applications, such as TV networks, tour operations, LAN networks, and electric grids.

The time complexity of Kruskal's algorithm is O (E log (E)) or O (E log(V)), where E is the number of edges and V is the number of vertices.

Prim's Minimum Spanning Tree algorithm

Prim's algorithm is also based on a greedy approach to find the minimum cost spanning tree.

Prim's algorithm is very similar to the Dijkstra algorithm for finding the shortest path in a graph

Start with an arbitrary node as a starting point, and then we check the outgoing edges from the selected nodes and traverse through the edge that has the lowest cost (or weights).

Prim's Minimum Spanning Tree algorithm

The algorithm works as follows:

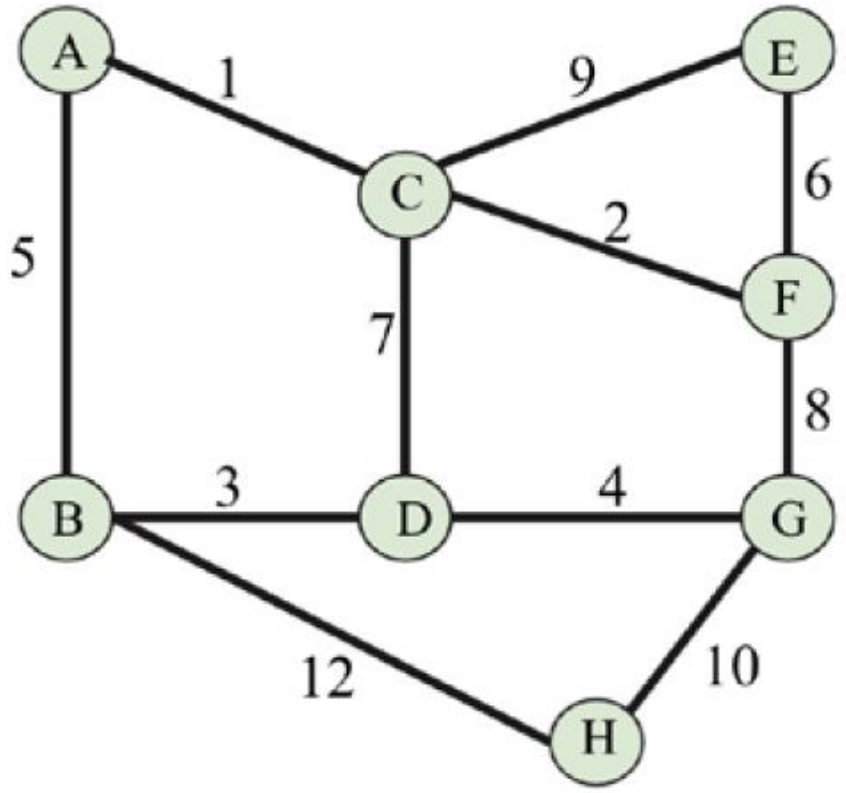
1. Create a dictionary that holds all the edges and their weights

2. Get the edges, one by one, that have the lowest cost from the dictionary and grow the tree in such a way that the cycle is not formed

3. Repeat step 2 until all the vertices are visited

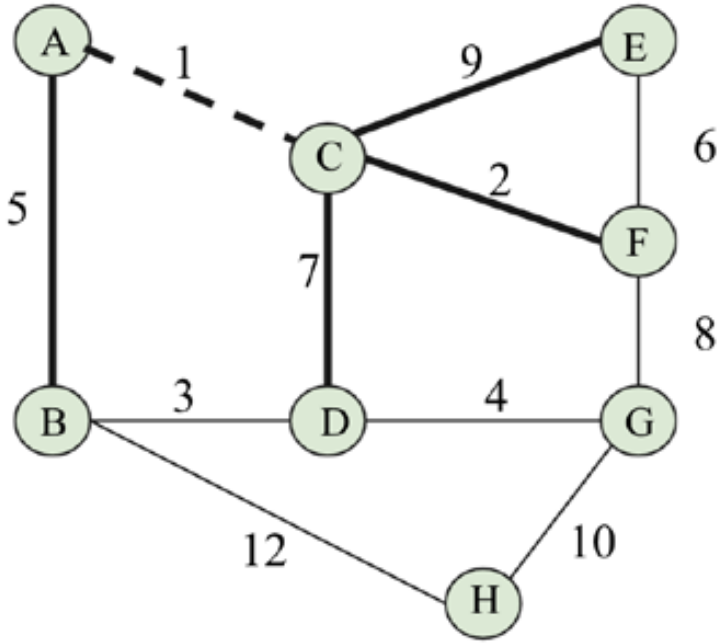
Example:

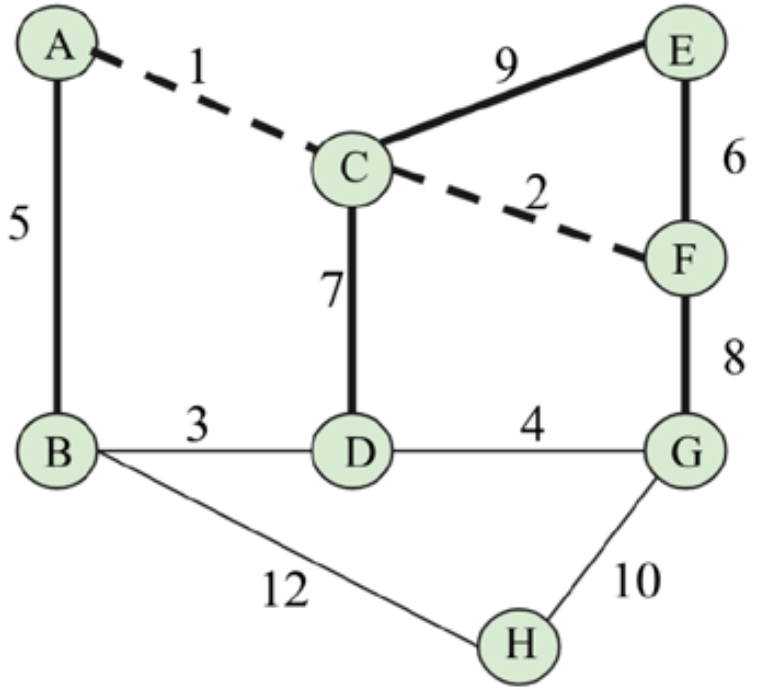
Assuming that we arbitrarily select A node,
we then check all the outgoing edges from A.
Here, we have two options, AB and AC;
we select edge AC since it has less cost/weight
(weight 1),

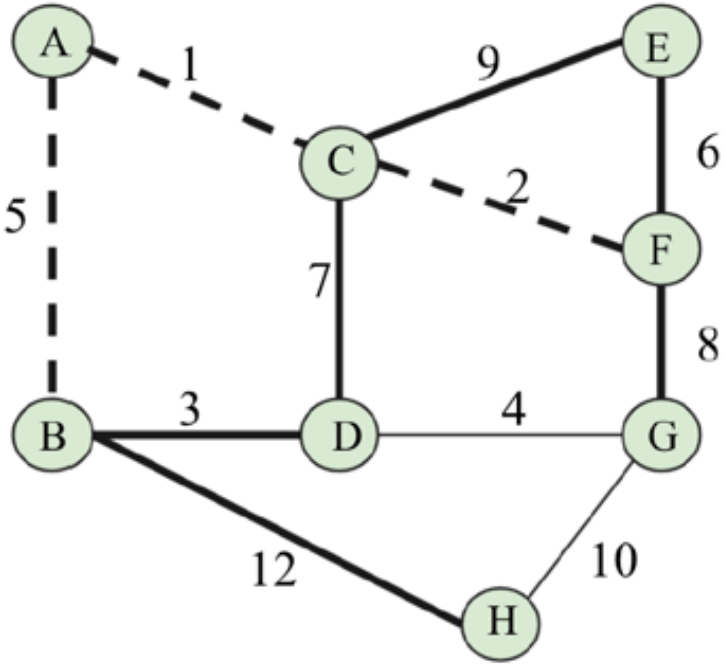


Prim's Minimum Spanning Tree algorithm

Next, we check the lowest outgoing edges from edge AC. We have options AB, CD, CE, CF, out of which we select edge CF, which has the lowest weight of 2. Likewise, we grow the tree, and next we select the lowest weighted edge, i.e., AB

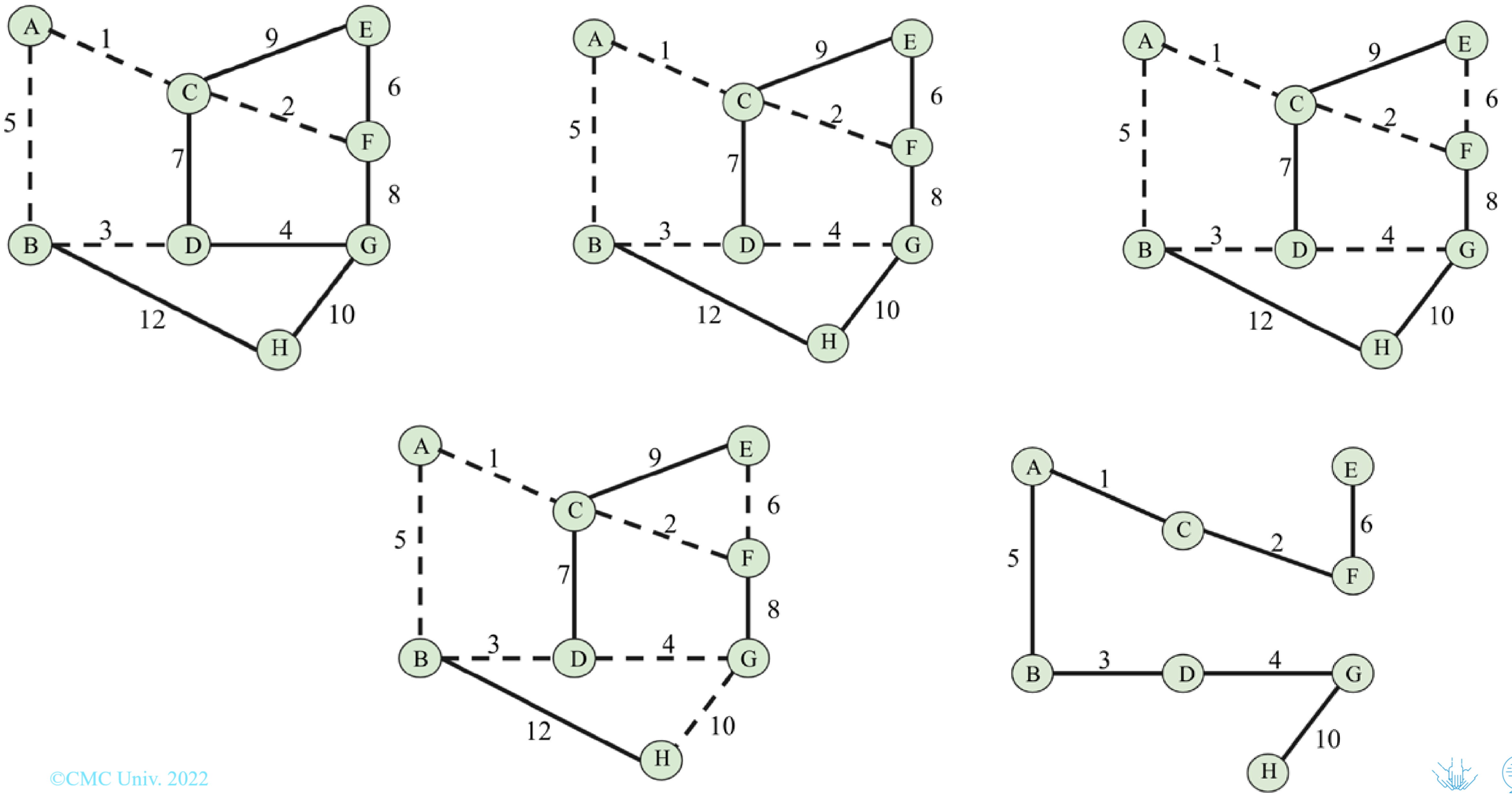






CMC UNIVERSITY

Prim's Minimum Spanning Tree algorithm



Prim's Minimum Spanning Tree algorithm

Prim's algorithm also has many real-world applications. For all the applications where we can use Kruskal's algorithm, we can also use Prim's algorithm. Other applications include road networks, game development, etc.

Since both Kruskal's and Prim's MST algorithms are used for the same purpose, which one should be used? In general, it depends on the structure of the graph. For a graph with n vertices and m edges, Kruskal's algorithm's worst-case time complexity is O(m logn), and Prim's algorithm has a time complexity of O(m + nlogn). So, we can observe that Prim's algorithm works better when we have a dense graph, whereas Kruskal's algorithm is better when we have a sparse graph.

CMC UNIVERSITY

PERT NETWORK

CMC UNIVERSITY

Introduction

- Any project involves planning, scheduling and controlling a number of interrelated activities with use of limited resources, namely, men, machines, materials, money and time.
  - The projects may be extremely large and complex such as construction of a housing, a highway, a shopping complex etc.
  - Introduction of new products, research and development projects
- It is required that managers must have a dynamic planning and scheduling system to produce the best possible results and also to react immediately to the changing conditions and make necessary changes in the plan and schedule.

CMC UNIVERSITY

Introduction

- Network analysis is one of the important tools for project management.
- Whether major or minor a project has to be completed in a definite time & at a definite cost.
- The necessary information of any particular data can be represented as a project network.
- These techniques are very useful for planning, scheduling and executing large-time bound projects involving careful co-ordination of variety of complex and interrelated activities

Introduction

Objectives of network analysis

- Helpful in planning
- Inter-relationship of various activities
- Cost control
- Minimisation of maintenance time
- Reduction of time
- Control on idle resources
- Avoiding delays, interruptions

CMC UNIVERSITY

Introduction

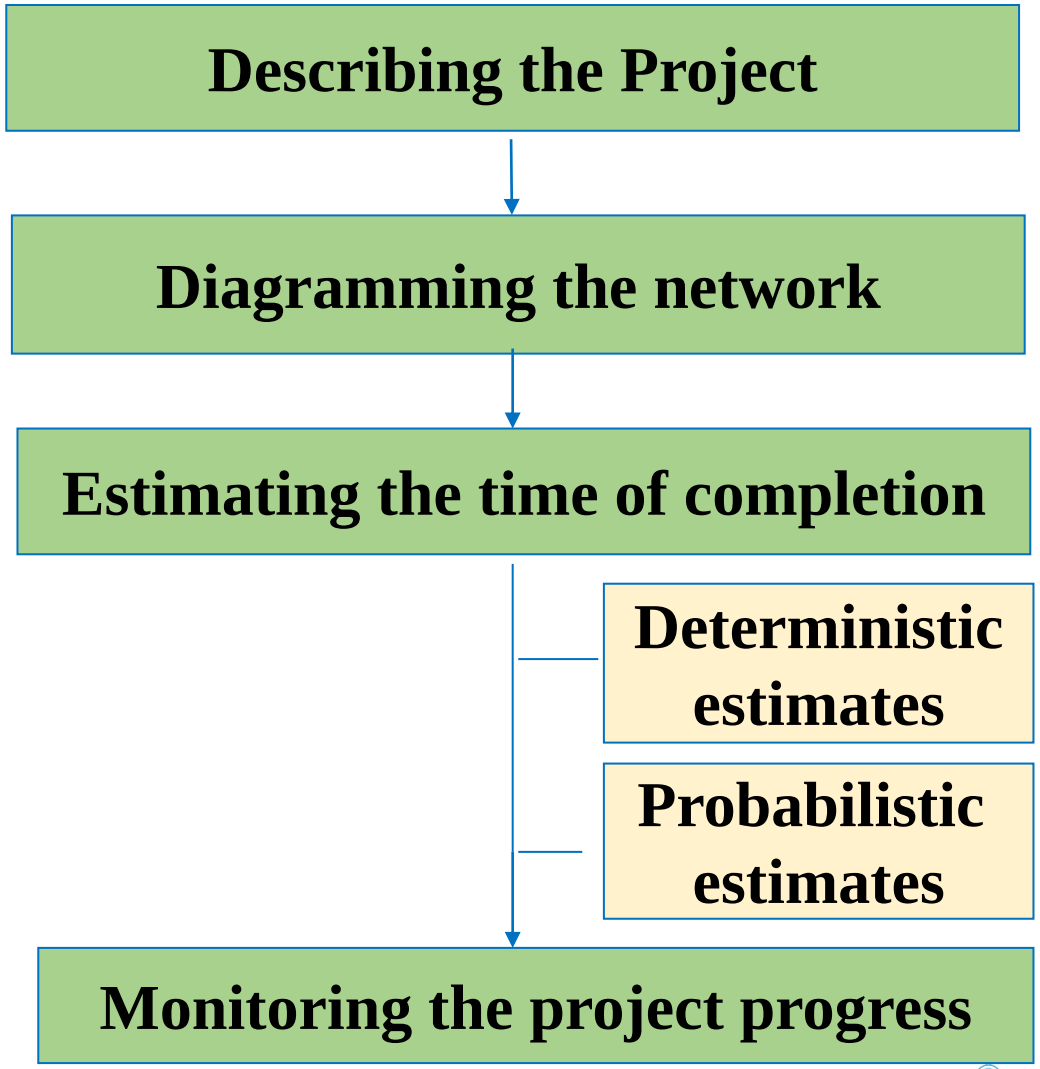
Applications of network analysis

- Planning, scheduling, monitoring and control of large and complex projects.
- Construction of factories, highways, building, bridges, cinemas etc.
- Helpful to army for its missile development.
- Assembly line scheduling
- Installation of computers and high tech machineries
- To make marketing strategies

CMC UNIVERSITY

Introduction

Methodology involved in Network Analysis



CMC UNIVERSITY

PERT

- A convenient analytical and visual technique of PERT and CPM prove extremely valuable in assisting the managers in managing the projects
- PERT (Program Evaluation and Review Technique) was developed in the 1950s. This technique was developed and used in conjunction with the planning and designing of the Polaris missile project.
- CPM (Critical Path Method) was developed by DuPont Company and applied first to the construction projects in the chemical industry
- Both PERT and CPM techniques have conceptual similarities, but PERT is used for analysis of project scheduling problem. CPM has single time estimate and PERT has three time estimates for activities and uses probability theory to find the chance of reaching the scheduled time.

CMC UNIVERSITY

PERT

Difference between PERT & CPM

PERT	CPM
A probability model with uncertainty in activity duration. The duration of each activity is computed from multiple time estimates with a view to take into account time uncertainty.	A deterministic model with well known activity times based upon the past experience.
It is applied widely for planning & scheduling research projects.	It is used for construction projects & business problems.
PERT analysis does not usually consider costs.	CPM deals with cost of project schedules & minimization.

CMC UNIVERSITY

Concepts

Activity

- *Activity*: All projects may be viewed as composed of activities. It is the smallest unit of work consuming both time and resources that project manager should schedule and control.
- An activity is represented by an arrow in network diagram

The head of the arrow shows sequence of activities.

CMC UNIVERSITY

Concepts

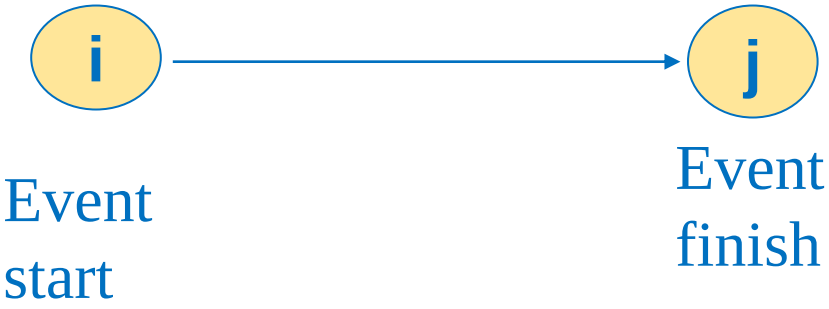
Classification of activities

- *Predecessor activity*: Activities that must be completed immediately prior to the start of another activity are called predecessor activities.
- *Successor activity*: activities that cannot be started until one or more of other activities are completed but immediately succeed them are called successor activities.
- *Concurrent activities*: activities that can be accomplished together are known as concurrent activities.
- *Dummy activity*: An activity which does not consume any resource but merely depicts the dependence of one activity on other is called dummy activity. It is introduced in a network when two or more parallel activities have the same start and finish nodes.

Concepts

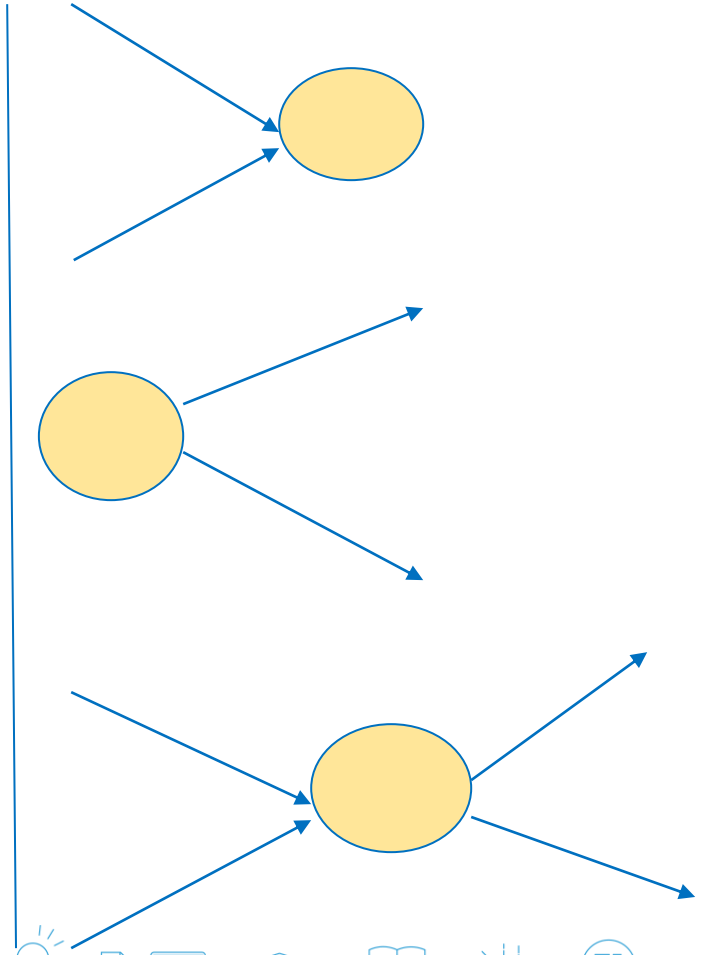
Event

- The beginning and end of an activity are called as events
- Events are represented by numbered circles called nodes.



Types of Events

- Merge event
- Burst event
- Merge & Burst Event

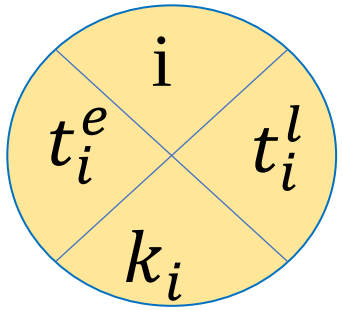


CMC UNIVERSITY

Concepts

Event

- The starting event of project is always numbered 1, the first activities of the project come from this event.
- The ending event has the highest number, the final activities of the project lead to this event.
- The earliest occurrence time of event i is: t_i^e
- The latest occurrence time of event i is: t_i^l
- The reserve time of event is: $t_i^r = t_i^l - t_i^e$



- k_i is the index of the vertex next to vertex i on the longest path from the starting vertex to i

We have: $t_1^e = t_1^l = 0$, $t_n^e = t_n^l = t_p$ (where n is the ending event)

CMC UNIVERSITY

Concepts

Path and Network

- An unbroken chain of activity arrows connecting the initial event to some other event is called a path.
- A network is the graphical representation of logically & sequentially connected arrows & nodes representing activities & events of a project. It is a diagram depicting precedence relationships between different activities.

Errors in network logic

- Looping: looping is known as cycling error and creates an impossible situation and it appears that none of the activities could ever be completed.
- Dangling: sometimes a project network includes an activity which does not fit into the end objective of the project and is carried out without any result related with completion of the project. Such an error in network is called dangling

CMC UNIVERSITY

Concepts

Stages for project management

Planning

- Make a network diagram
- Split the project into separate activities and determine the completion time
- Represent operations by directed arcs, events by nodes

Scheduling

- Build time chart, start and end time of each activity
- Indicate critical activities

Controlling

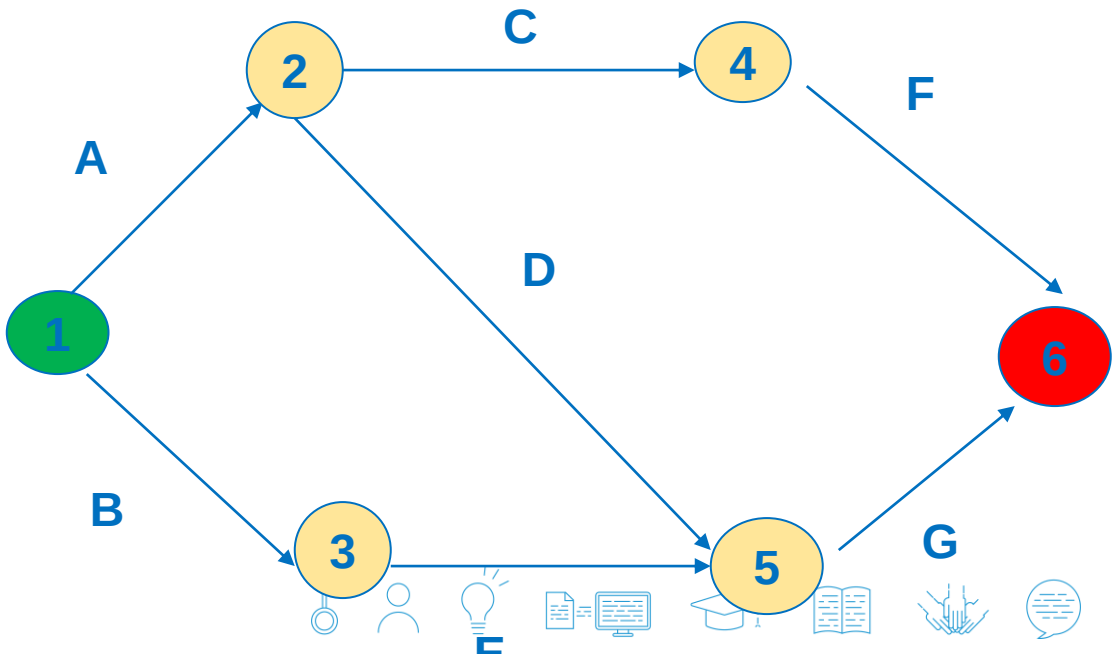
- Use network diagrams to track project progress

Concepts

PERT network diagram: is a directed, single, connected graph containing no cycles. This graph has only one vertex with no input (start vertex) and one vertex with no output (end vertex).

- Vertices: each vertex represents an event, denoted *i*, *j*, ...

- Arcs: An arc represents an activity and the time to do it



CMC UNIVERSITY

PERT Network diagram

Guidelines for network diagram Construction

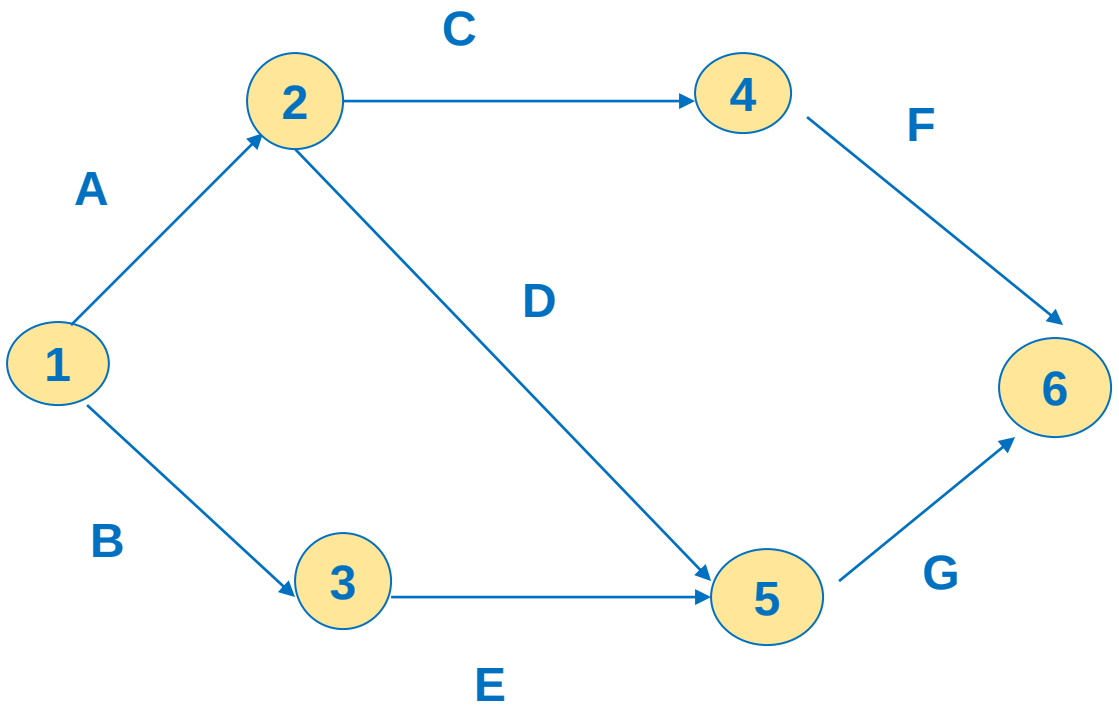
- A complete network diagram should have one stand point & one finish point.
- The flow of the diagram should be from left to right.
- Arrows should not be crossed unless it is completely unavoidable.
- Arrows should be kept straight & not curved or bent.
- Angle between arrows should as large as possible.
- Each activity must have a tail or head event. No two or more activities may have same tail & head events.
- Once the diagram is complete the nodes should be numbered from left to right. It should then be possible to address each activity uniquely by its tail & head event.

CMC UNIVERSITY

PERT Network diagram

Example 1: Draw the network diagram for the following data table

Activity	Predecessor activity
A	none
B	none
C	A
D	A
E	B
F	C
G	D, E

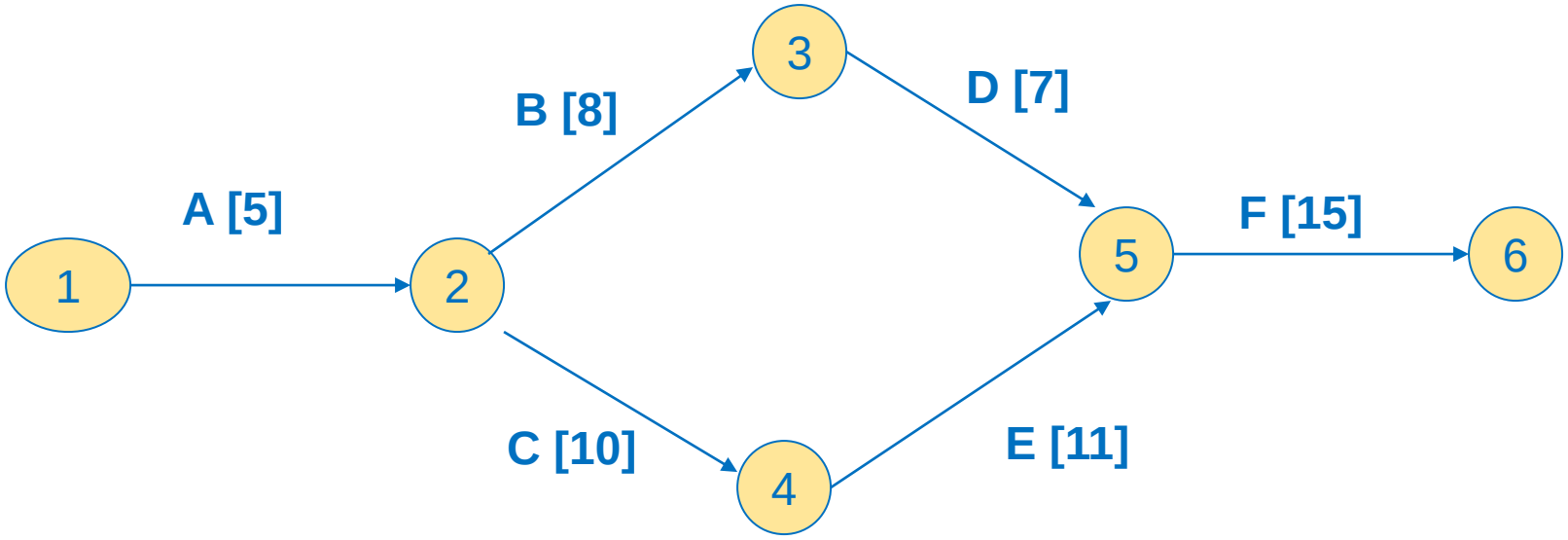


CMC UNIVERSITY

PERT Network diagram

Example 2: Draw the network diagram for the following data table

Activity	Predecessor activity	Time
A	none	5
B	A	8
C	A	10
D	B	7
E	C	11
F	D , E	15



CMC UNIVERSITY

PERT Network diagram

Critical path

- Those activities which contribute directly to the overall duration of the project constitute critical activities, the critical activities form a chain running through the network which is called critical path.
- **Critical event**: the slack of an event is the difference between the latest & earliest events time. The events with zero slack time are called as critical events.
- **Critical activities**: The difference between latest start time & earliest start time of an activity will indicate amount of time by which the activity can be delayed without affecting the total project duration. The difference is usually called total float. Activities with 0 total float are called as critical activities
- The critical path is the longest path in the network from the starting event to ending event & defines the minimum time required to complete the project.
- The critical path is denoted by darker or double lines.

Time calculation on PERT network

Calculate t_i^e *in the direction* of activity deployment

- $t_1^e = 0$
- $t_i^e = \max\{ t_k^e + t_{ki} \}$; (k, i) enters vertex i

Calculate t_i^l *in the opposite direction* of activity deployment

- $t_n^l = t_n^e = t_p$ (t_p is the project completion time)
- $t_i^l = min\{ t_j^l - t_{ij} \}$; (i, j) goes away from vertex i

Note:

- The critical path length tells us the minimum time to complete the project

- If an activity on the critical path is delayed (shortened), the entire project will be delayed (shortened). Therefore, to shorten the project completion time, we only need to shorten the time of the critical tasks.

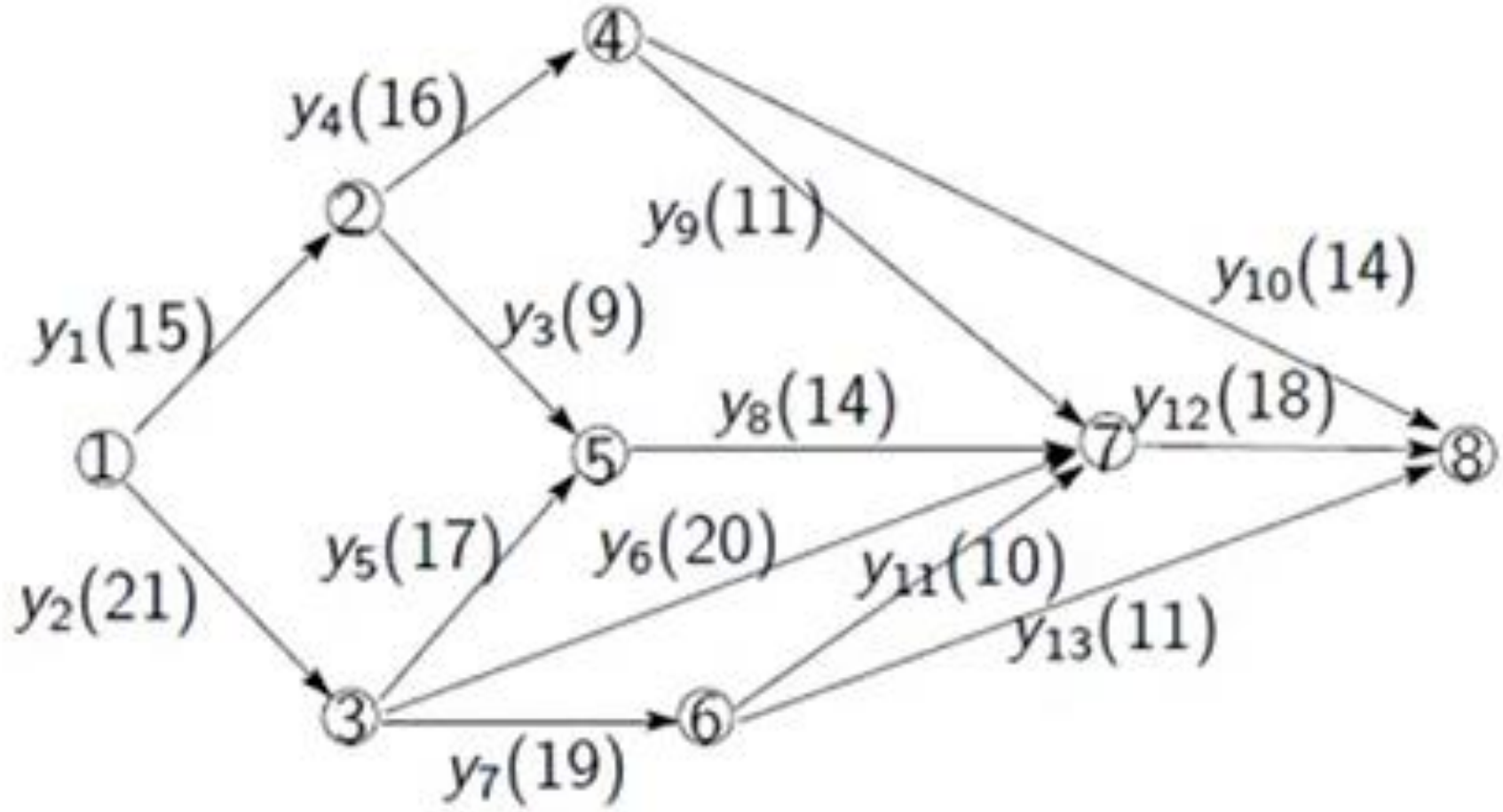
Time calculation on PERT network

Example 3: Draw the network diagram and calculate completion time of the following project:

Activity	Predecessor activity	Time	Activity	Predecessor activity	Time
y_1	-	15	y_8	y_3, y_5	14
y_2	-	21	y_9	y_4	11
y_3	y_1	9	y_{10}	y_4	14
y_4	y_1	16	y_{11}	y_7	10
y_5	y_2	17	y_{12}	y_6, y_8, y_9, y_{11}	18
y_6	y_2	20	y_{13}	y_7	11
y_7	y_2	19			

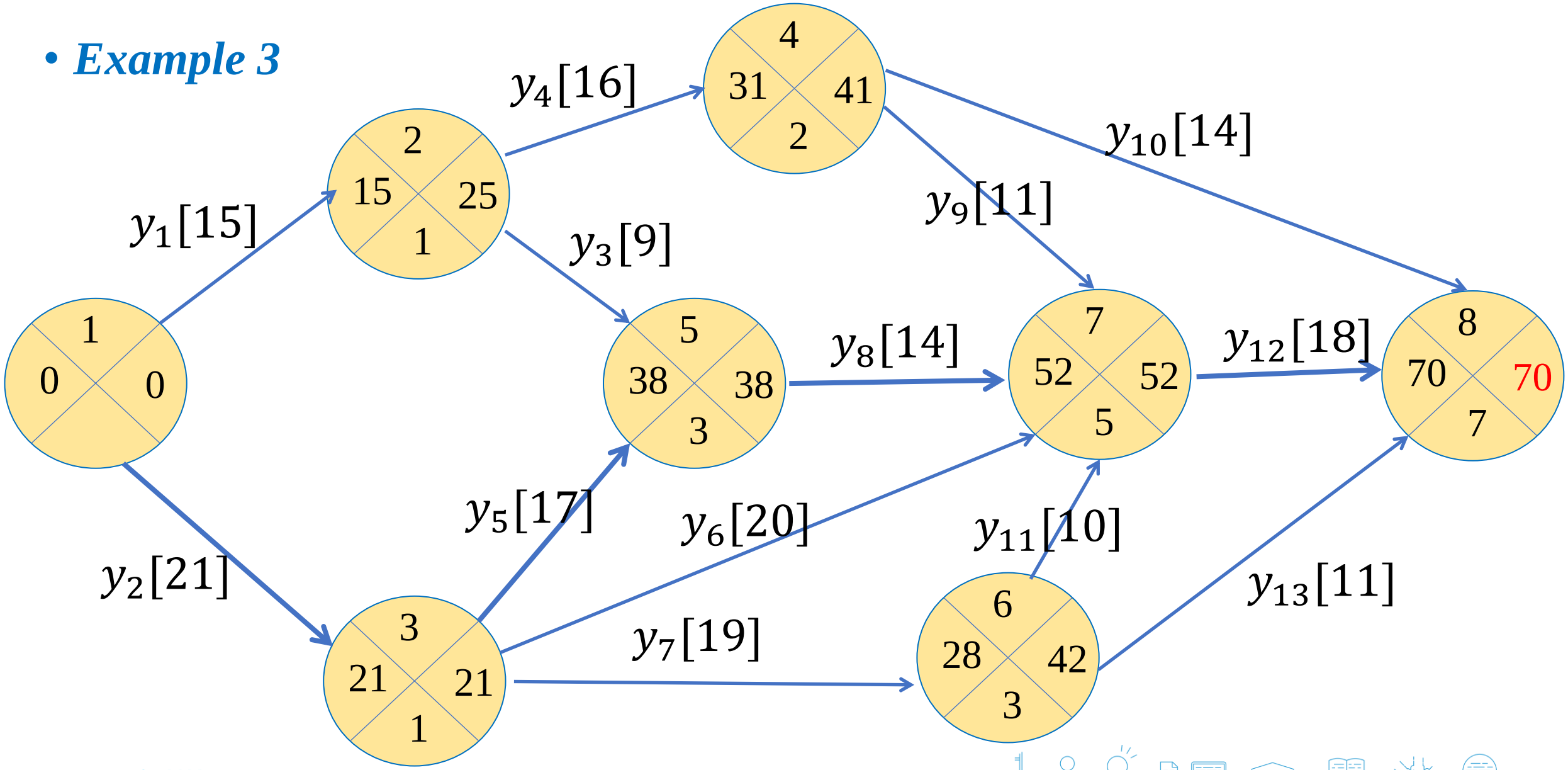
Time calculation on PERT network

- ***Example 3***



Time calculation on PERT network

- ***Example 3***



Time calculation on PERT network

Example 3:
Calculate the earliest start time, earliest finish time, latest finish time, latest start time of activities

Activity	EST_{ij}	EFT_{ij}	LST_{ij}	LFT_{ij}
(1, 2)	0	15	10	25
(1, 3)	0	21	0	21
(2, 4)	15	31	25	41
(2, 5)	15	24	29	38
(3, 5)	21	38	21	38
(3, 6)	21	40	23	42
(3, 7)	21	41	32	52
(4, 7)	31	42	41	52
(4, 8)	31	45	56	70
(5, 7)	48	52	38	52
(6, 7)	40	50	42	52
(6, 8)	40	51	59	70
(7, 8)	52	70	52	70

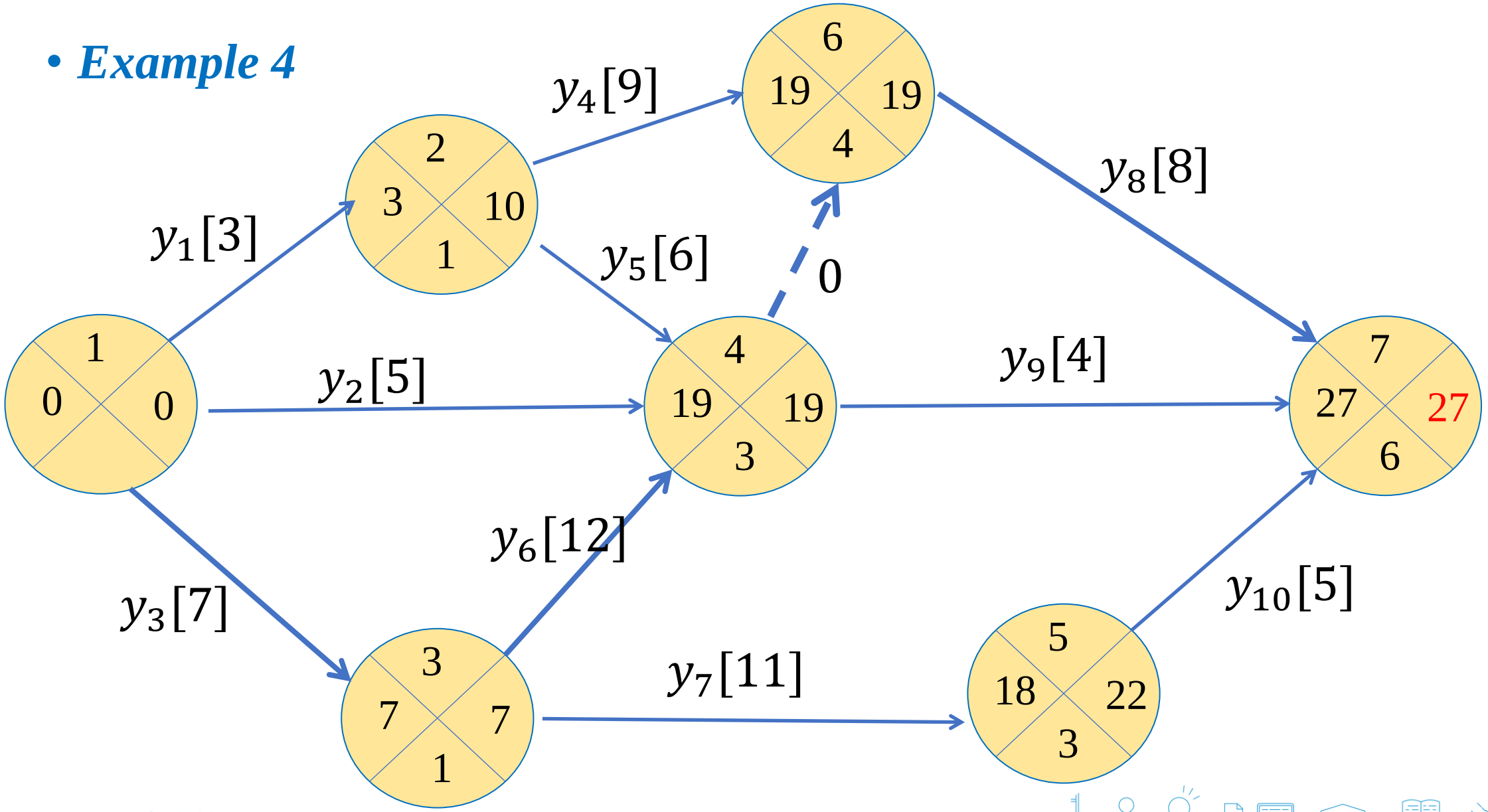
Time calculation on PERT network

Example 4: Draw the network diagram and calculate completion time of the following project

Activity	Predecessor activity	Time
y_1	-	3
y_2	-	5
y_3	-	7
y_4	y_1	9
y_5	y_1	6
y_6	y_3	12
y_7	y_3	11
y_8	y_2, y_4, y_5, y_6	8
y_9	y_2, y_5, y_6	4
y_{10}	y_7	5

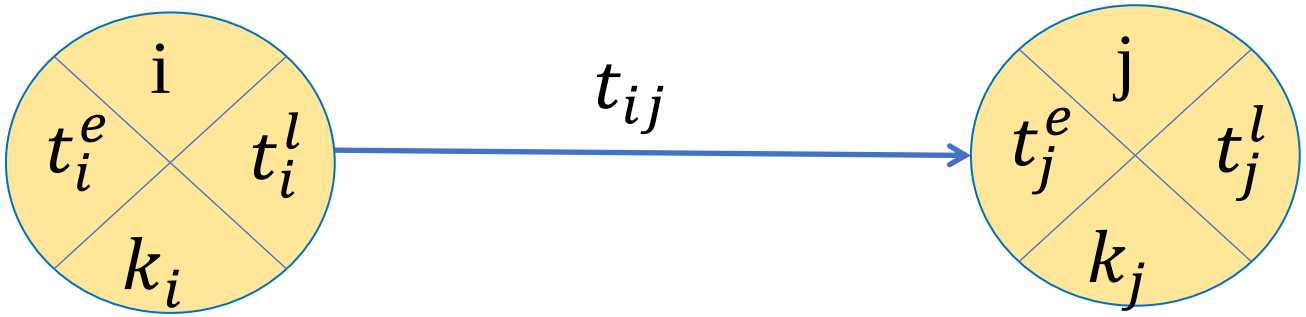
Time calculation on PERT network

- ***Example 4***



Time points of activities

- The activity execution time (i, j): t_{ij}
- The earliest start time of (i, j): $EST_{ij} = t_i^e$
- The earliest finish time of (i, j): $EFT_{ij} = EST_{ij} + t_{ij}$
- The latest start time of (i, j): $LST_{ij} = t_j^l - t_{ij}$
- The latest finish time of (i, j): $LFT_{ij} = t_j^l$



Time points of activities

Reserve time of activity

- The activity reserve time is important in shortening the time to complete the activity without affecting the time to complete the entire project.

General reserve time (maximum reserve time):

- The general reserve time of the activity (i, j) is denoted $t_{ij}^{g} = t_{j}^{l} - (t_{i}^{e} + t_{ij})$

Note:

a) The common reserve time is the type of time that when we use it to change the earliest start time, earliest start time, latest start time, latest finish time, lengthening the duration of the activity (i, j), only affect the before and after activities of (i, j) without affecting the project completion time.

b) If (i,j) is a critical activity, then $t_{ij}^{g} = 0$ $(because\ t_{ij}^{g} = t_{j}^{l} - (t_{i}^{e} + t_{ij}) = t_{j}^{l} - t_{j}^{e} = 0)$

Time points of activities

Reserve time of activity

c) For non-critical activities, there are two types:

- An independent non-critical activity is a non-critical activity that connects two critical vertices
- The relevant non-critical activity is the remaining non-critical activities.

For example: Consider the network diagram in example 3:

- Critical activities: (1, 3), (3, 5), (5, 7), (7, 8)
- Independent non-critical activities: (3, 7)
- Relevant non-critical activities: the remaining non-critical activities

Time points of activities

Independent reserve time (minimum reserve time):

+ *Independent reserve time* of the activity (i, j) is $t_{ij}^{r} = \max\{0, t_j^e - (t_i^l + t_{ij})\}$

+ *Top private reserve time*: is the reserve time that allows us to move the earliest start time or extend the activity time without affecting the earliest start time of the next activities.

$$t_{ij}^{p} = t_j^e - (t_i^e + t_{ij})$$

+ Original private reserve time: is the reserve time that allows us to start late or extend the activity time without affecting the latest start time of the next activities.

$$t_{ij}^{o} = t_j^l - (t_i^l + t_{ij})$$

Time calculation on PERT network

Example 5: Draw the network diagram and determine the critical path of the following project:

Activity	Predecessor activity	Time
y_1	-	2
y_2	-	1
y_3	y_1, y_2	16
y_4	y_3	12
y_5	y_4	4
y_6	y_5	3
y_7	y_6	4
y_8	y_4	10
y_9	y_7, y_8	3
y_{10}	y_9	2

CMC UNIVERSITY

PERT

Advantages

- Planning & controlling projects
- Flexibility
- Designation of responsibilities
- Achievement of objective with least cost
- Better managerial control

PERT

Limitations of PERT /CPM

- Network diagrams should have clear starting & ending points , which are independent of each other which may not be possible in real life.
- Another limitation is that it assumes that manager should focus on critical activities.
- Resources will be available when needed for completion for an an activity is again unreal.

Difficulties

- Difficulty in securing realistic time estimates.
- The planning & implementation of networks requires trained staff.
- Developing clear logical network is troublesome.


CMC UNIVERSITY
THANK YOU