

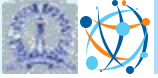
DS221: Introduction to Scalable Systems

Topic: Algorithms and Data Structures



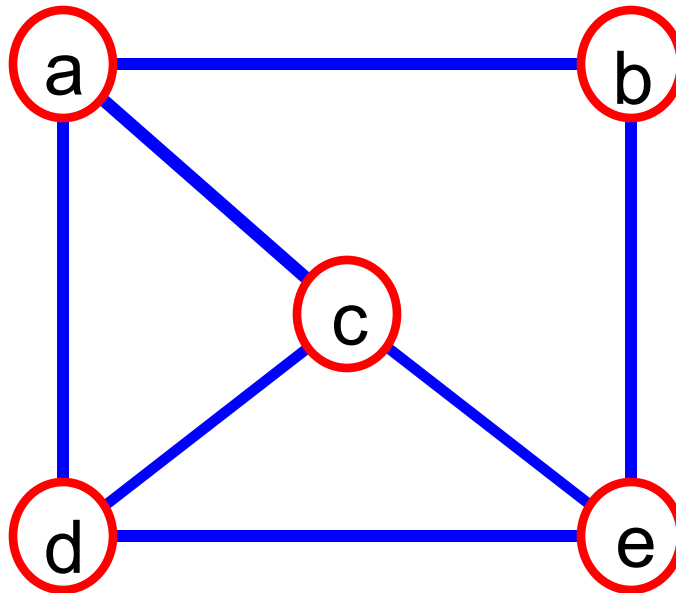
L5: Graphs

Graph ADT, Algorithms



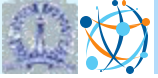
What is a Graph?

- A graph $G = (V, E)$ is composed of:
 - V : set of **vertices**
 - E : set of **edges** connecting the **vertices** in V
- An **edge** $e = (u, v)$ is a pair of **vertices**
- Example:



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

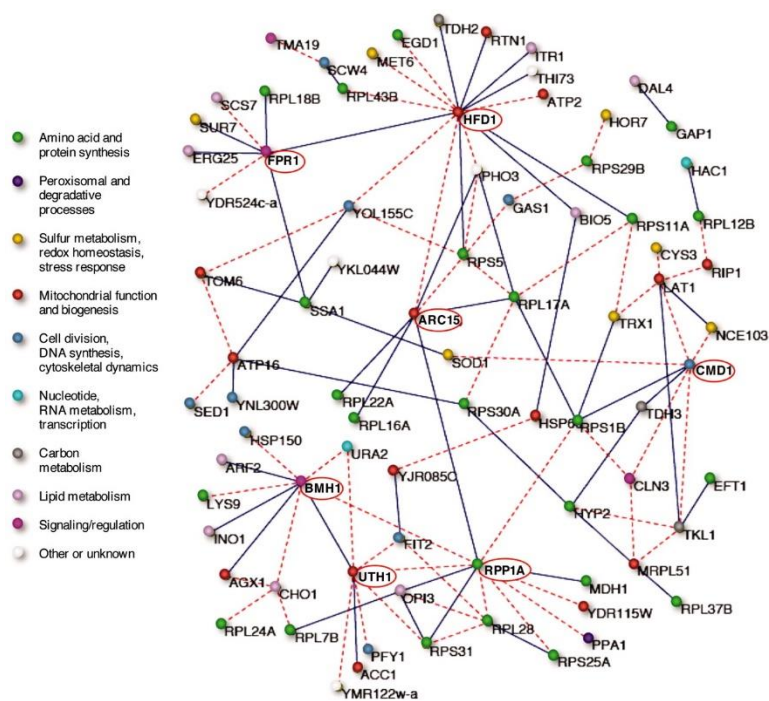
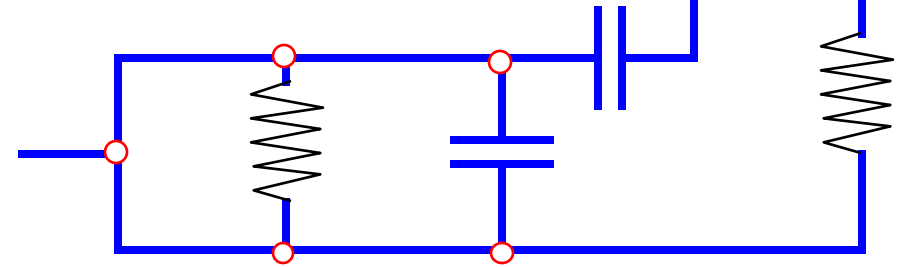
$$E = \{(a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e)\}$$

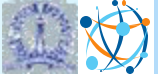


Applications

- Electronic circuit design
- Transport networks
- Biological Networks

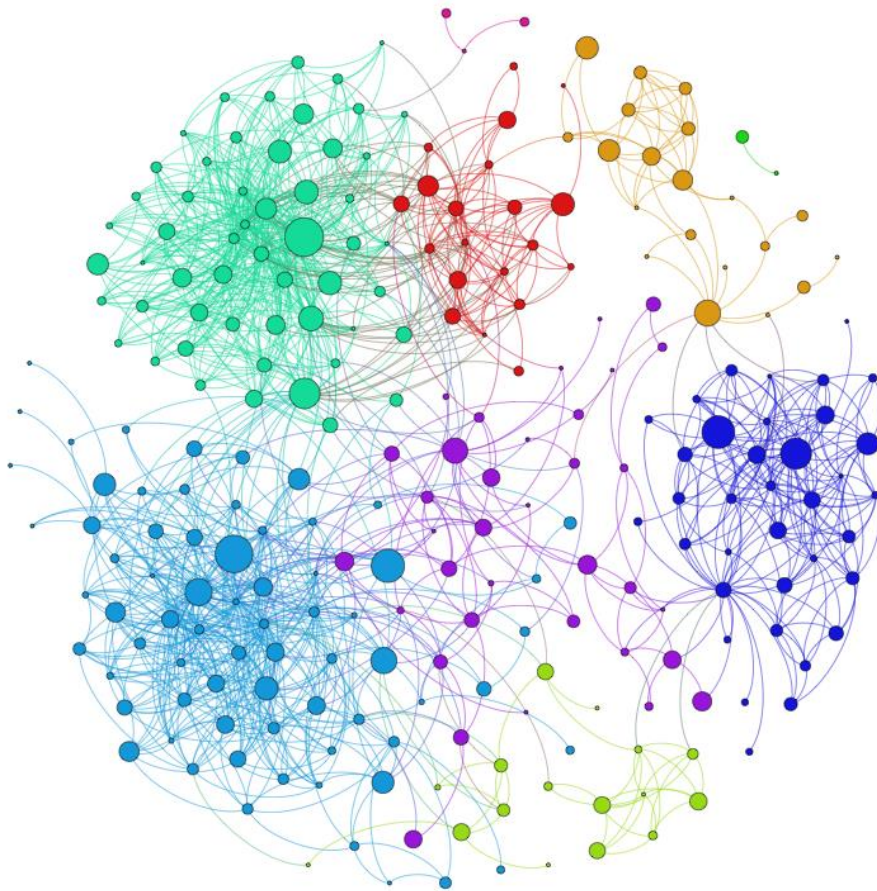
CS16



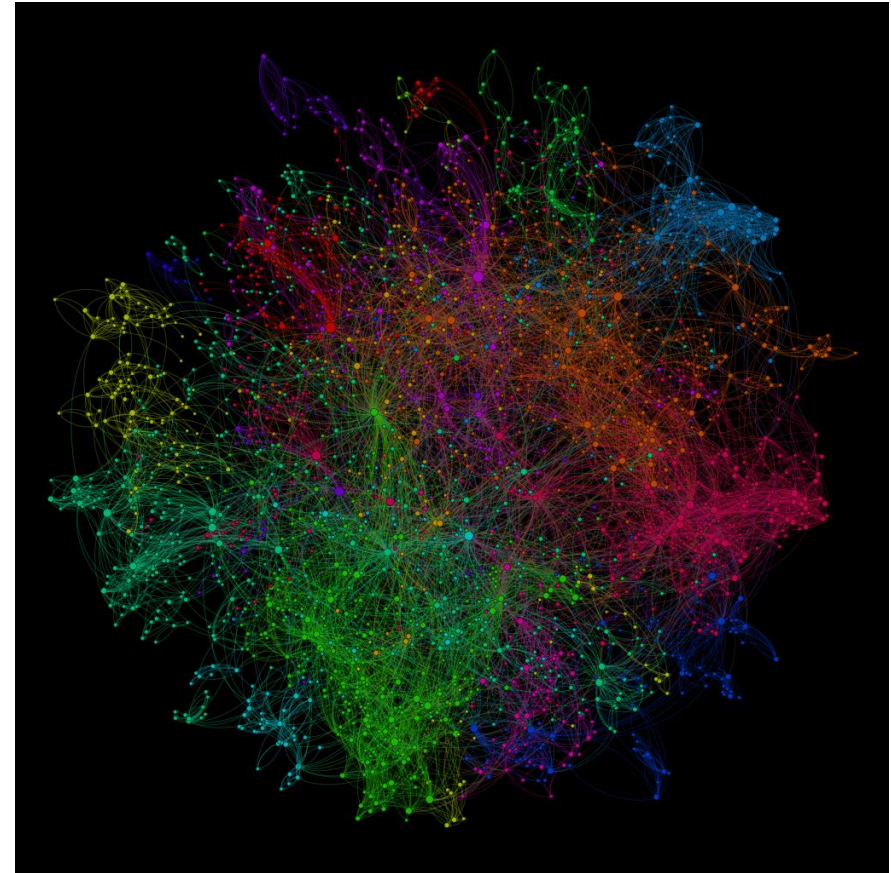


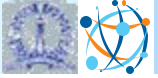
Applications

LinkedIn Social Network Graph



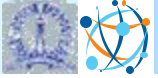
Java Call Graph for Neo4J





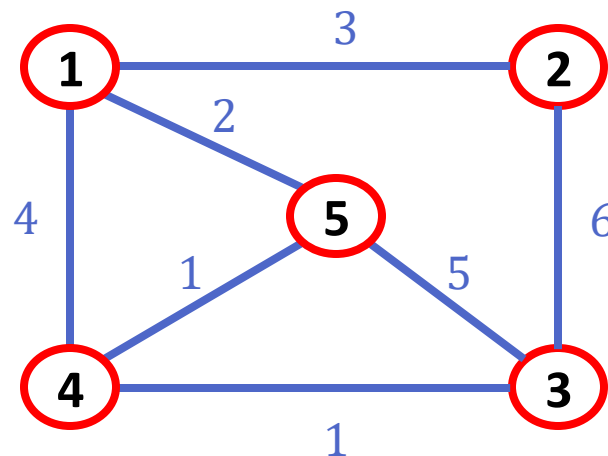
Terminology

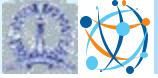
- If (v_0, v_1) is an edge in an **undirected** graph,
 - v_0 and v_1 are **adjacent**, or are **neighbors**
 - The edge (v_0, v_1) is **incident** on vertices v_0 and v_1
- If $\langle v_0, v_1 \rangle$ is an edge in a **directed** graph
 - v_0 is **adjacent to** v_1 , and v_1 is **adjacent from** v_0
 - The edge $\langle v_0, v_1 \rangle$ is **incident** on v_0 and v_1
 - v_0 is the **source vertex** and v_1 is the **destination vertex**



Terminology

- Vertices & edges can have **labels** that uniquely identify them
 - Edge label can be formed from the pair of vertex labels it is incident upon...*assuming only one edge can exist between a pair of vertices*
- Edge **weights** indicate some measure of distance or cost to pass through that edge



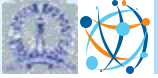


Terminology

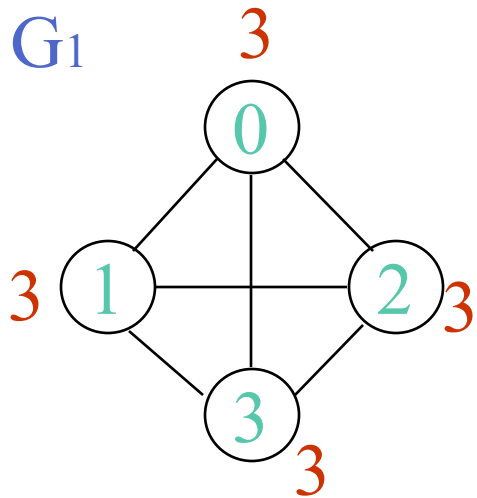
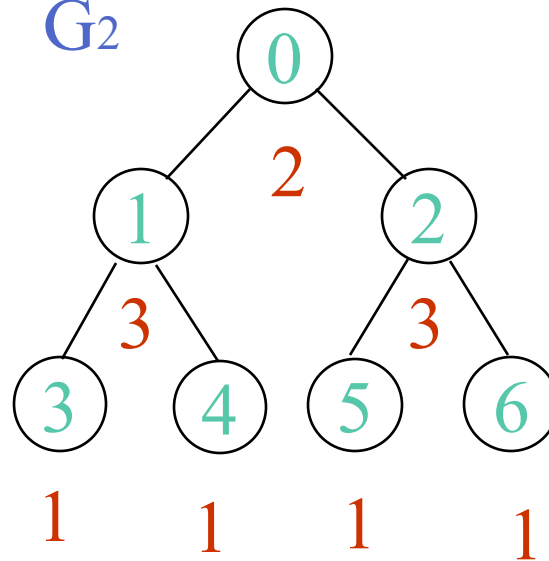
- The **degree** of a vertex is the number of edges incident to that vertex
- For directed graph,
 - the **in-degree** of a vertex v is the number of edges that have v as the sink vertex
 - the **out-degree** of a vertex v is the number of edges that have v as the source vertex
 - if d_i is the degree of a vertex i in a graph G with n vertices and e edges, the number of edges is

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$

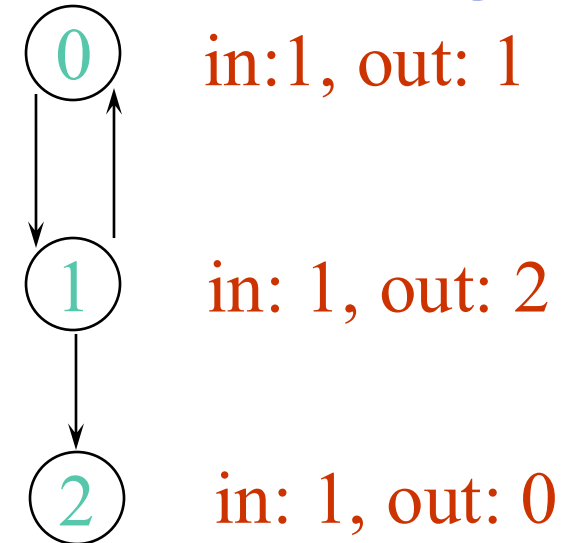
Why? Since adjacent vertices each count the adjoining edge, it will be counted twice



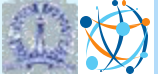
Examples

 G_1  G_2 

undirected graphs

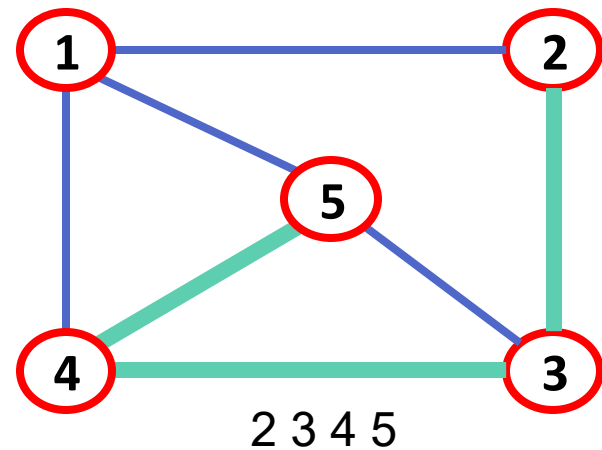
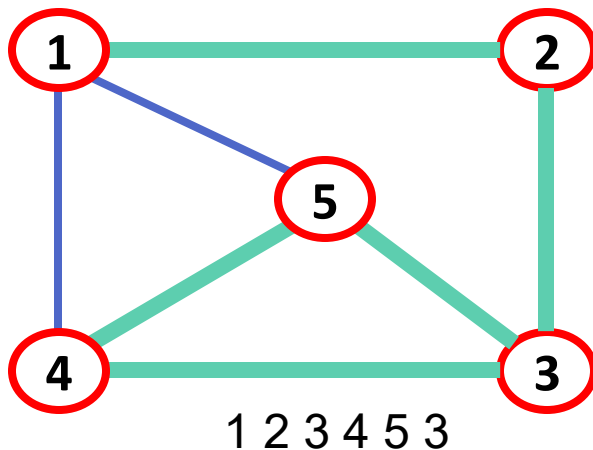
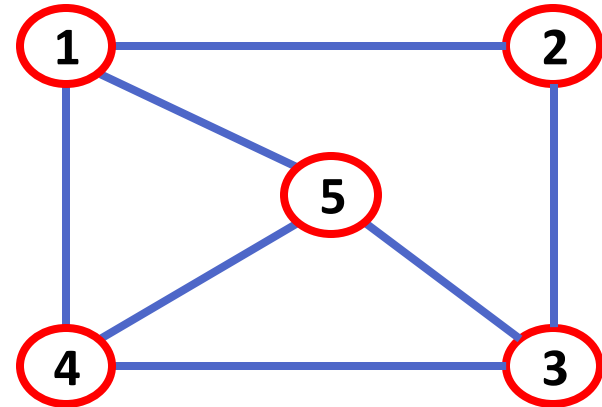
 G_3 

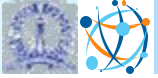
directed graph



Terminology

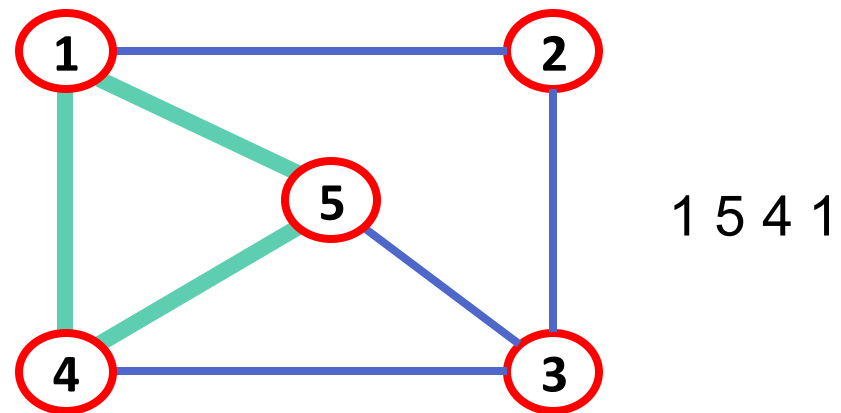
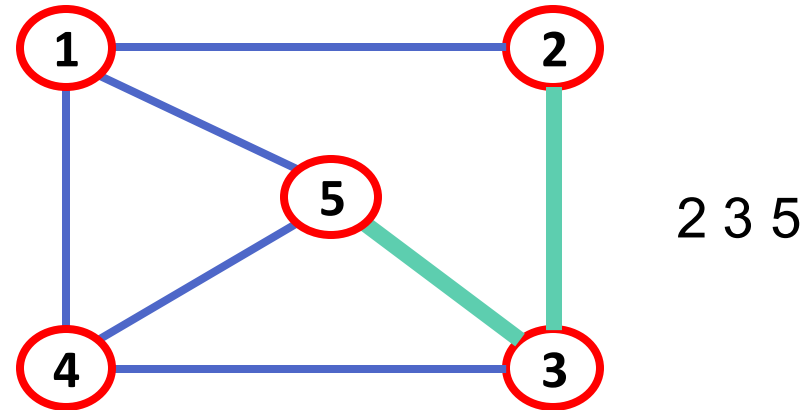
- **path** is a sequence of vertices $\langle v_1, v_2, \dots, v_k \rangle$ such that consecutive vertices v_i and v_{i+1} are adjacent

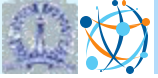




Terminology

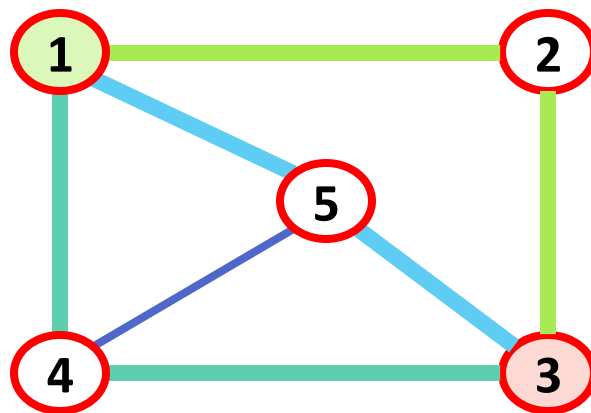
- **simple path**: no repeated vertices
- **cycle**: simple path, except that the last vertex is the same as the first vertex



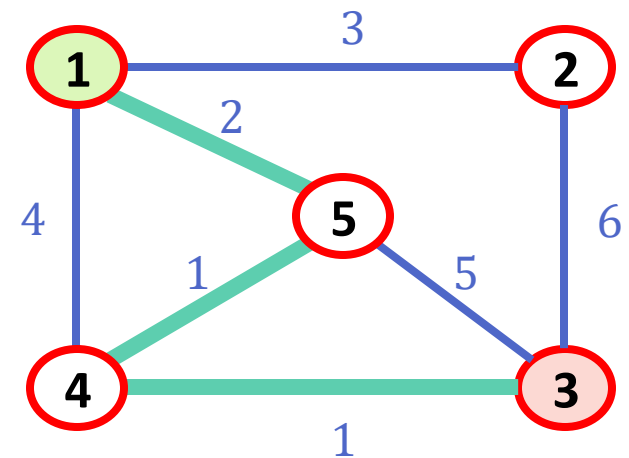


Terminology

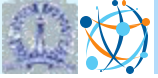
- **Shortest Path:** Path between two vertices where the sum of the edge weights is the smallest
 - Has to be a simple path (why?)
 - Assume “unit weight” for edges if not specified



1 2 3
1 5 3
1 4 3

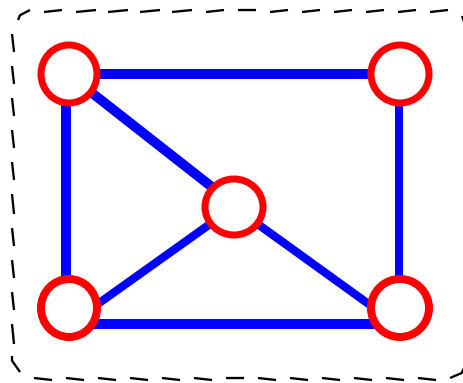


1 5 4 3

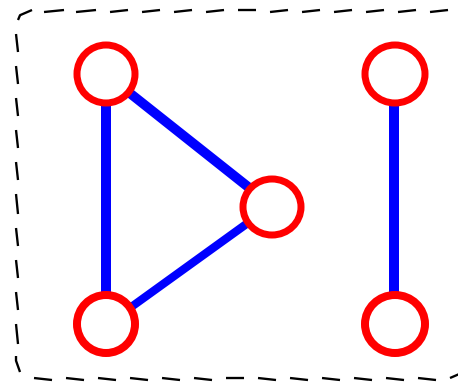


Connected Graph

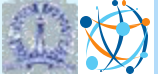
- **connected graph**: For every pair of vertices, there exists a path between them.



connected

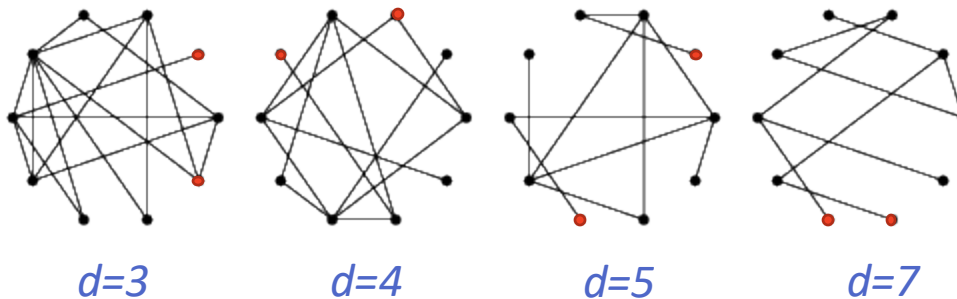


not connected



Graph Diameter

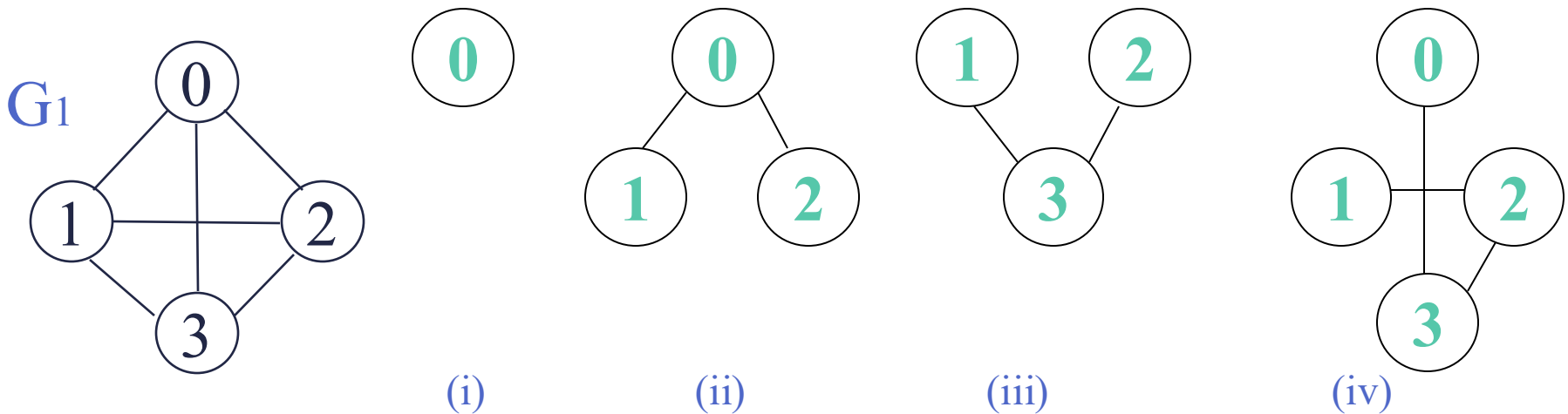
- A **graph's diameter** is the distance of its *longest shortest path*
- if $d(u,v)$ is the distance of the shortest path between vertices u and v , then:
- $diameter = \text{Max}(d(u,v))$, for all u, v in V
- A disconnected graph has an infinite diameter



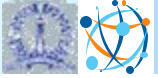


Subgraph

- **subgraph**: subset of vertices and edges forming a graph

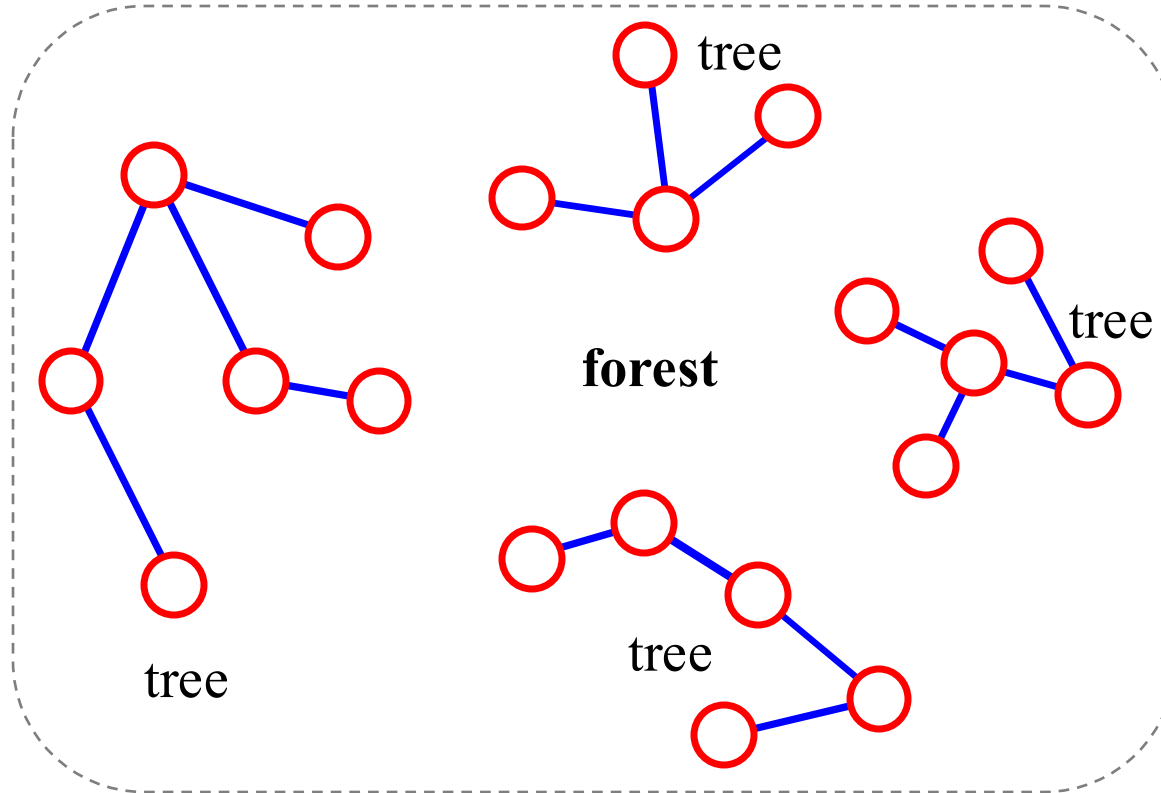


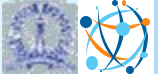
(a) Some of the subgraph of G_1



Trees & Forests

- **tree** - connected graph without cycles
- **forest** - collection of trees



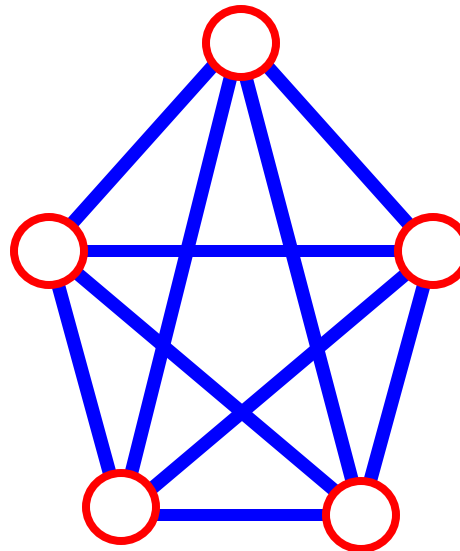


Fully Connected Graph

- Let $n = \text{\#vertices}$, and $m = \text{\#edges}$
- Complete graph (or) Fully connected graph**: One in which all pairs of vertices are adjacent
- How many total edges in a complete graph?*
 - Each of the n vertices is incident to $n-1$ edges, however, we would have counted each edge twice! Therefore, intuitively, $m = n(n-1)/2$.

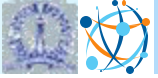
If a graph is not complete:

$$m < n(n-1)/2$$



$$n = 5$$

$$m = (5*4)/2 = 10$$

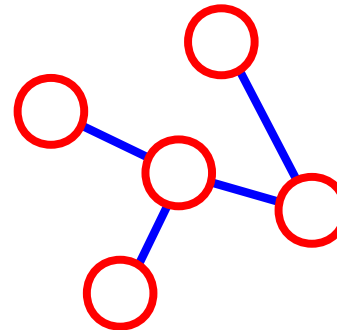


More on Connectivity

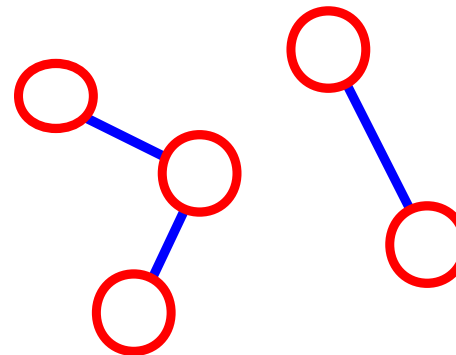
n = #vertices

m = #edges

■ For a tree **m** = **n** - 1

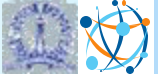


n = 5
m = 4



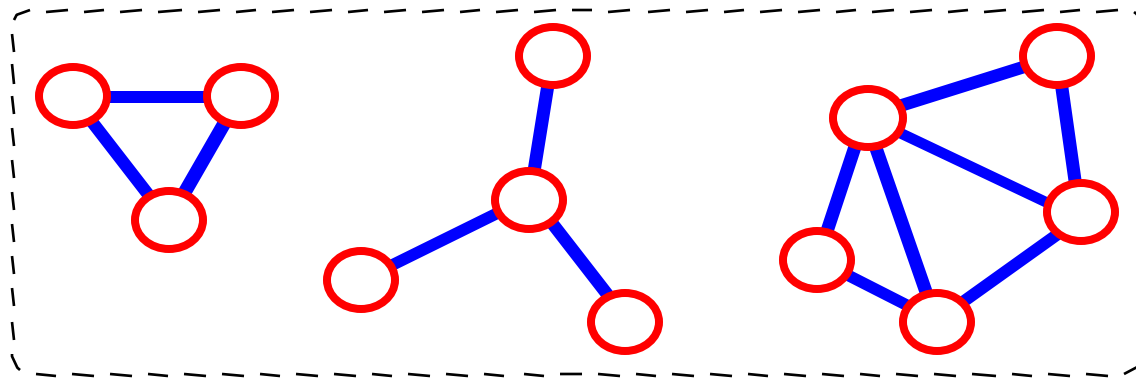
n = 5
m = 3

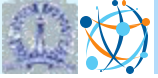
If **m** < **n** - 1, G is
not connected



Connected Component

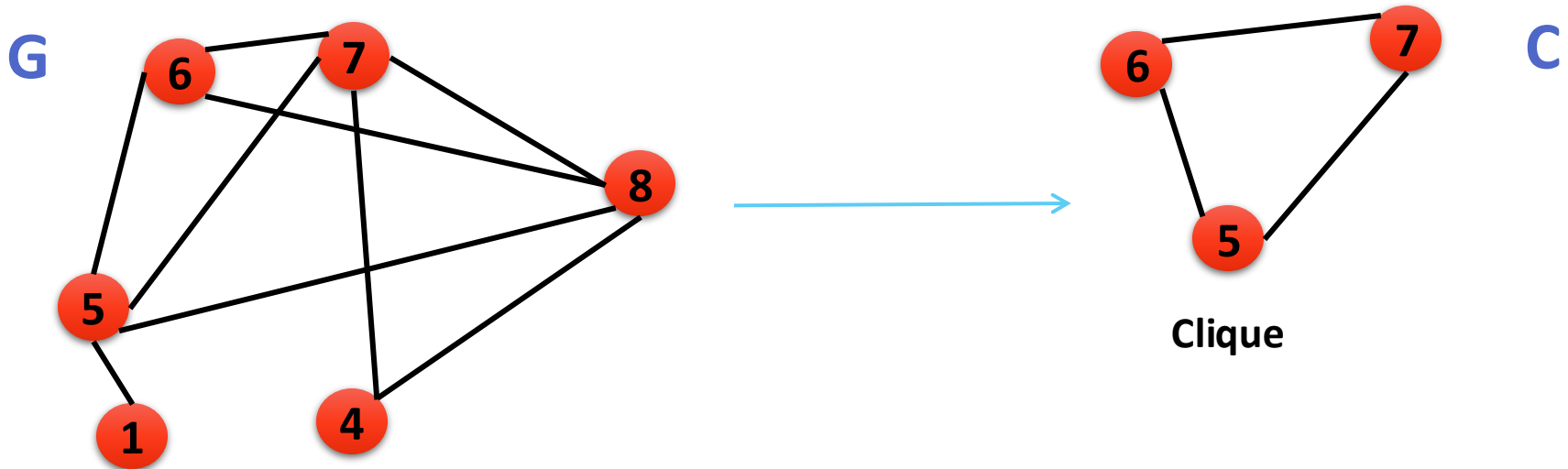
- A **connected component** is a maximal subgraph that is connected.
 - Cannot add vertices and edges from original graph and retain connectedness.
- A connected graph has exactly **1 component**.

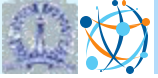




Clique

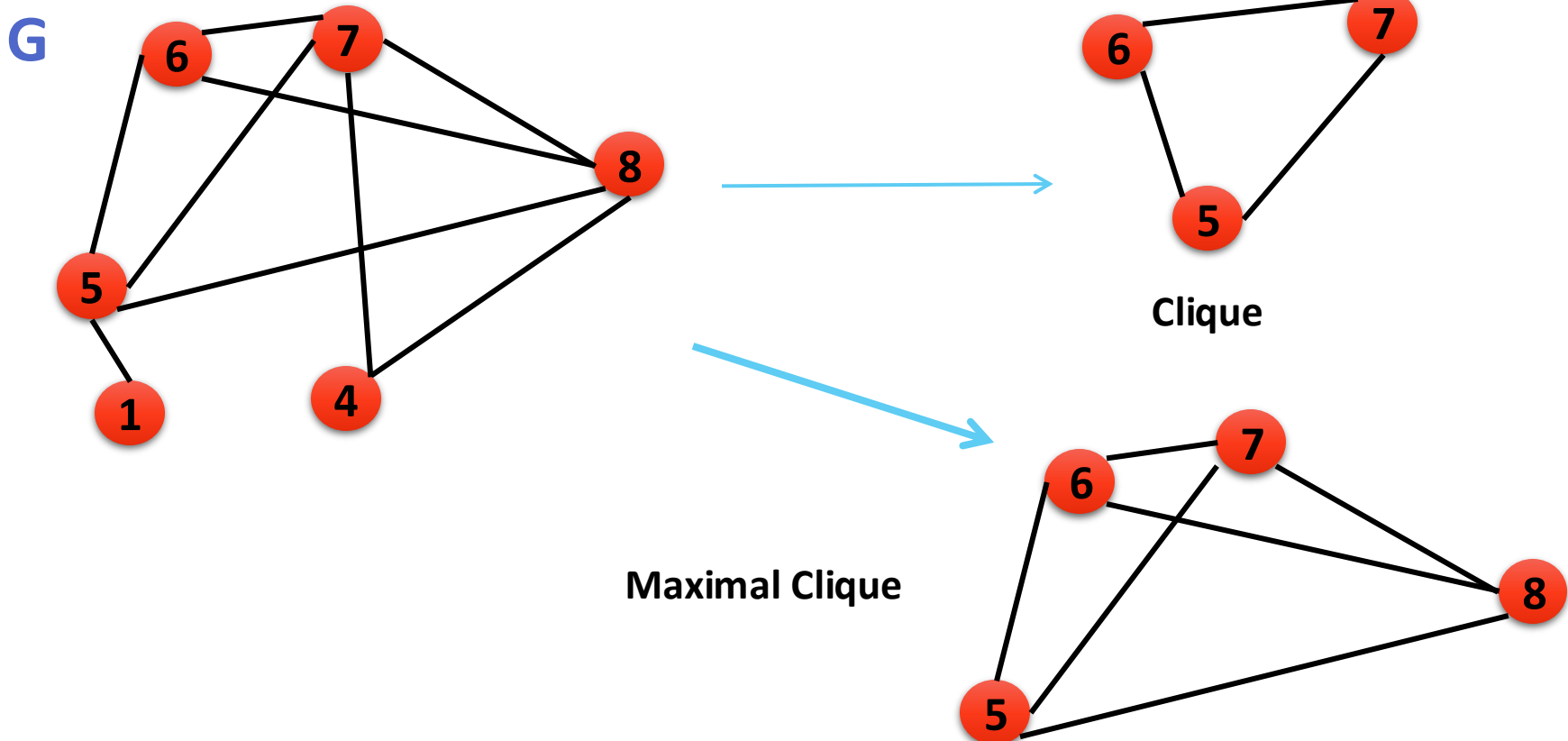
- A subgraph C of a graph G with *edges between all pairs of vertices*

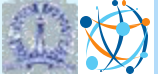




Maximal Clique

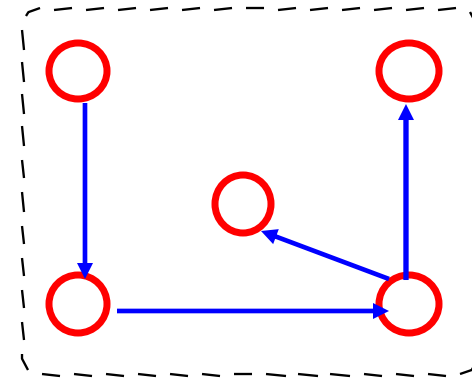
- A maximal clique is a clique that is not part of a larger clique

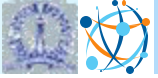




Directed vs. Undirected Graph

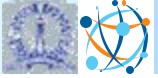
- An **undirected graph** is one in which the pair of vertices in a edge is unordered, $(v_0, v_1) = (v_1, v_0)$
- A **directed graph** (or **Digraph**) is one in which each edge is a directed pair of vertices, $\langle v_0, v_1 \rangle \neq \langle v_1, v_0 \rangle$





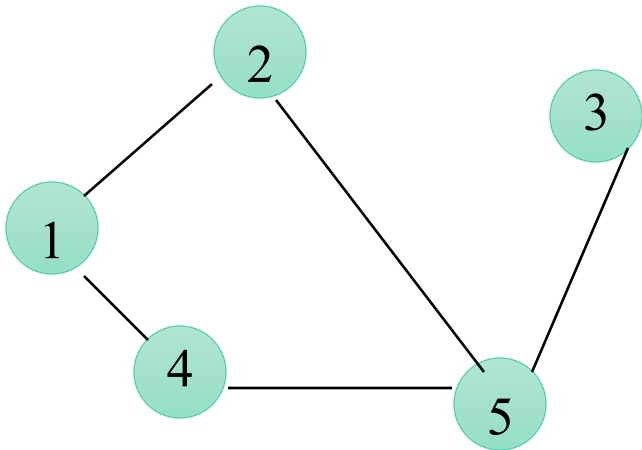
Graph Representation

- Adjacency Matrix
- Adjacency Lists
 - Linked Adjacency Lists
 - Array Adjacency Lists



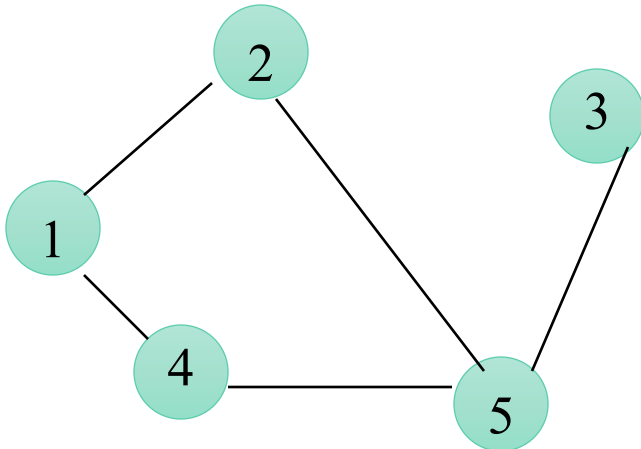
Adjacency Matrix

- **0/1 $n \times n$ matrix**, where $n = \#$ of vertices
- $A(i,j) = 1$ iff (i,j) is an edge



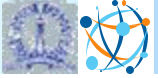
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

Adjacency Matrix Properties

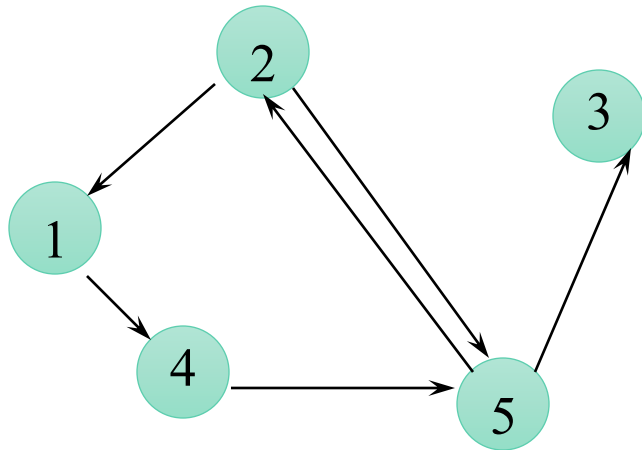


	1	2	3	4	5
1	0	1	0	1	0
2	1	0	0	0	1
3	0	0	0	0	1
4	1	0	0	0	1
5	0	1	1	1	0

- Diagonal entries are zero.
- Adjacency matrix of an *undirected graph* is *symmetric*.
 - $A(i,j) = A(j,i)$ for all i and j .

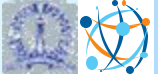


Adjacency Matrix (Digraph)



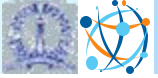
	1	2	3	4	5
1	0	0	0	1	0
2	1	0	0	0	1
3	0	0	0	0	0
4	0	0	0	0	1
5	0	1	1	0	0

- Diagonal entries are zero.
- Adjacency matrix of a directed graph need not be symmetric.



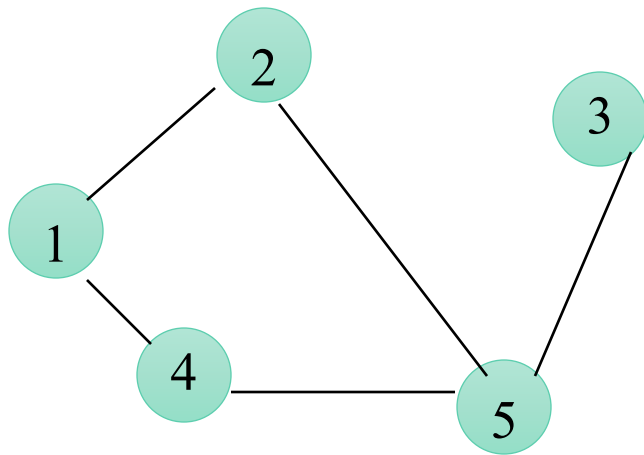
Adjacency Matrix

- n^2 bits of space
- For an *undirected graph*, may store only lower or upper triangle (exclude diagonal)
 - $(n^2 - n)/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.



Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i .
- An array of n adjacency lists.



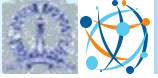
$aList[1] = (2,4)$

$aList[2] = (1,5)$

$aList[3] = (5)$

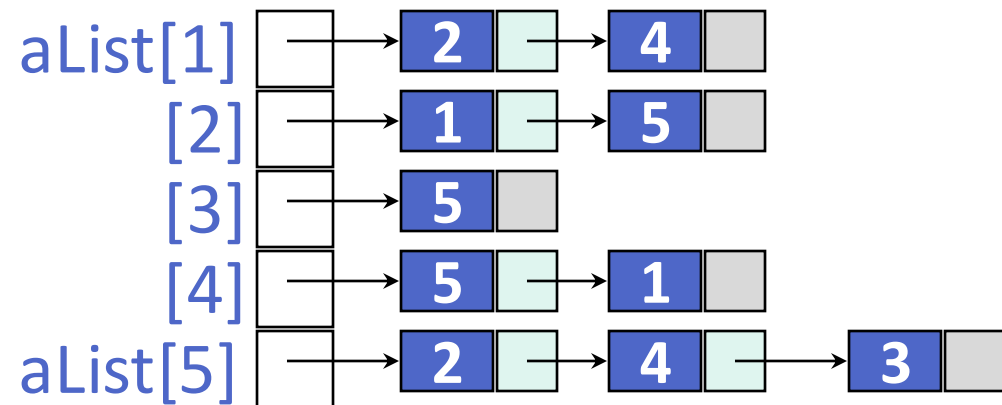
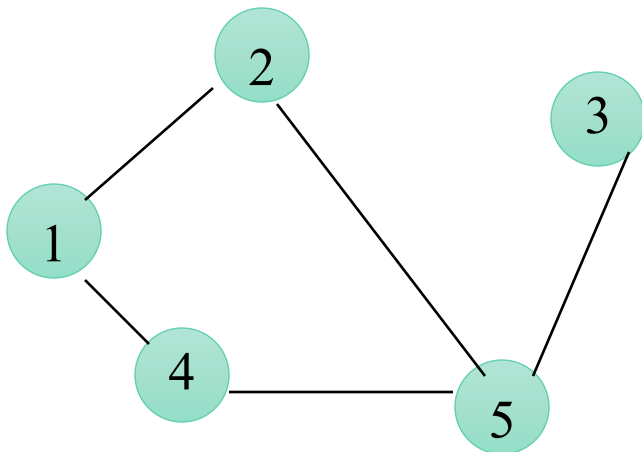
$aList[4] = (5,1)$

$aList[5] = (2,4,3)$

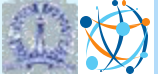


Linked Adjacency Lists

- Each adjacency list is a chain.

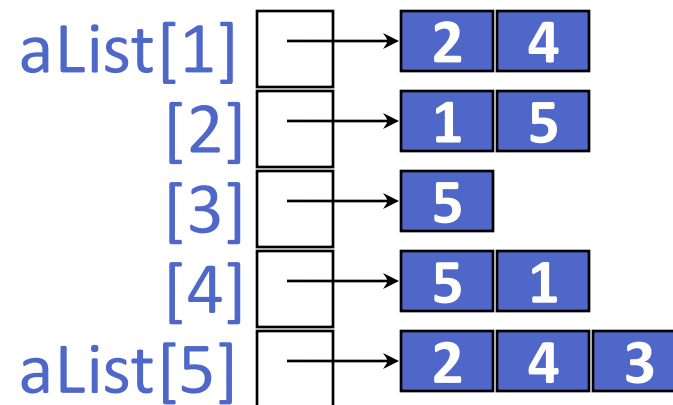
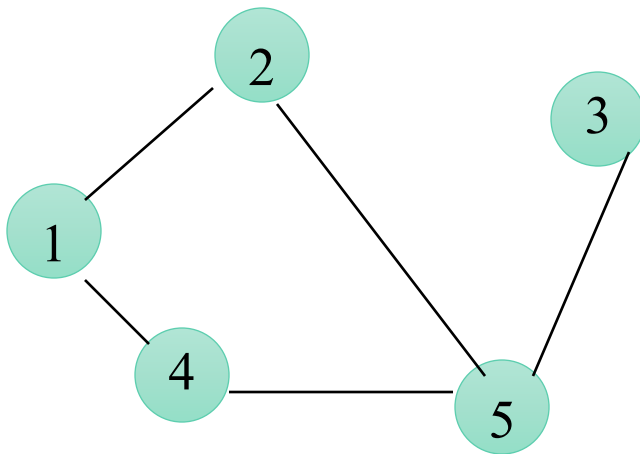


- Array Length = n
- # of chain nodes = $2e$ (undirected graph)
- # of chain nodes = e (digraph)

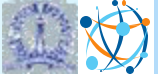


Array Adjacency Lists

- Each adjacency list is an array list.

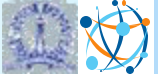


- Array Length = n
- # of list elements = $2e$ (undirected graph)
- # of list elements = e (digraph)



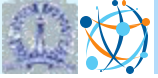
Storing Weighted Graphs

- Cost adjacency matrix
 - $C(i,j)$ = cost of edge (i,j) instead of 0/1
- Adjacency lists
 - Each list element is a pair (adjacent vertex, edge weight)



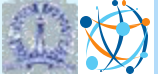
ADT for Graph

```
class Vertex<V,E> {  
    int id;  
    V value;  
    int GetId();  
    V GetValue();  
    List<Edge<V,E>> Neighbors();  
}  
  
class Edge<V,E> {  
    int id;  
    E value;  
    int GetId();  
    E GetValue();  
    Vertex<V,E> GetSource();  
    Vertex<V,E> GetDestination();  
}
```



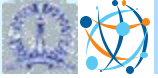
ADT for Graph

```
class Graph<V,E>{  
    List<Vertex<V,E>> vertices;  
    List<Edge<V,E>> edges;  
  
    void InsertVertex(Vertex<V,E> v);  
    void InsertEdge(Edge<V,E> e);  
  
    bool DeleteVertex(int vid);  
    bool DeleteEdge(int eid);  
  
    List<Vertex<V,E>> GetVertices();  
    List<Edge<V,E>> GetEdges();  
  
    bool IsEmpty(graph);  
}
```



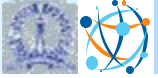
Sample Graph Problems

- Graph traversal
 - Searching
 - Shortest Paths
 - Connectedness
 - Spanning tree
- Graph centrality
 - PageRank
 - Betweenness centrality
- Graph clustering
 - K-means clustering

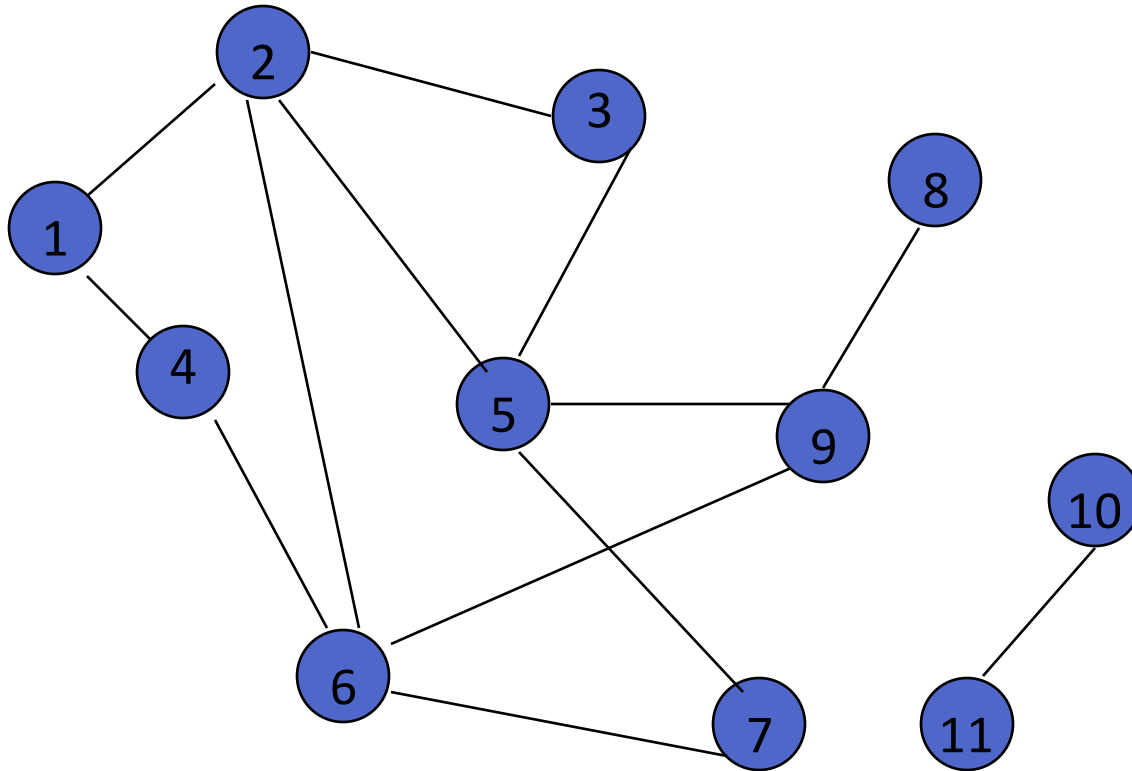


Graph Search & Traversal

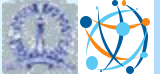
- Find a vertex (or edge) with a given ID or value
 - If list of vertices/edges is available, linear scan!
 - BUT, goal here is to traverse the neighbors of the graph, not scan the list
- Traverse through the graph to list all vertices in a particular order
 - Target item may be discovered as a by-product of the traversal



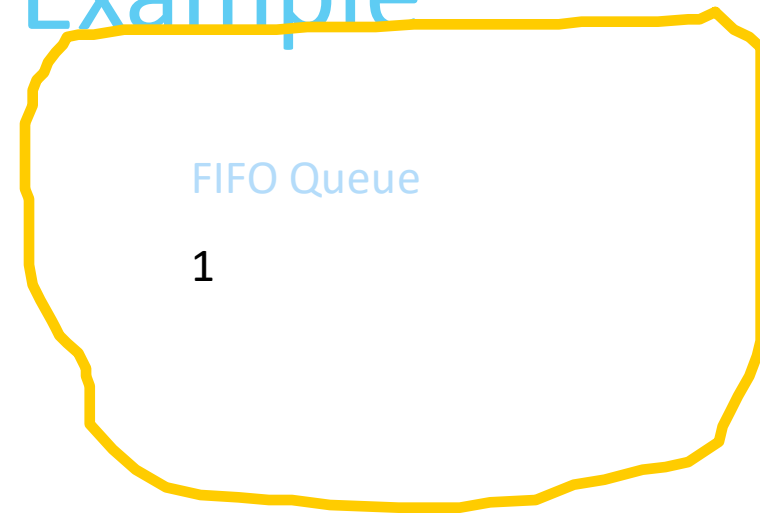
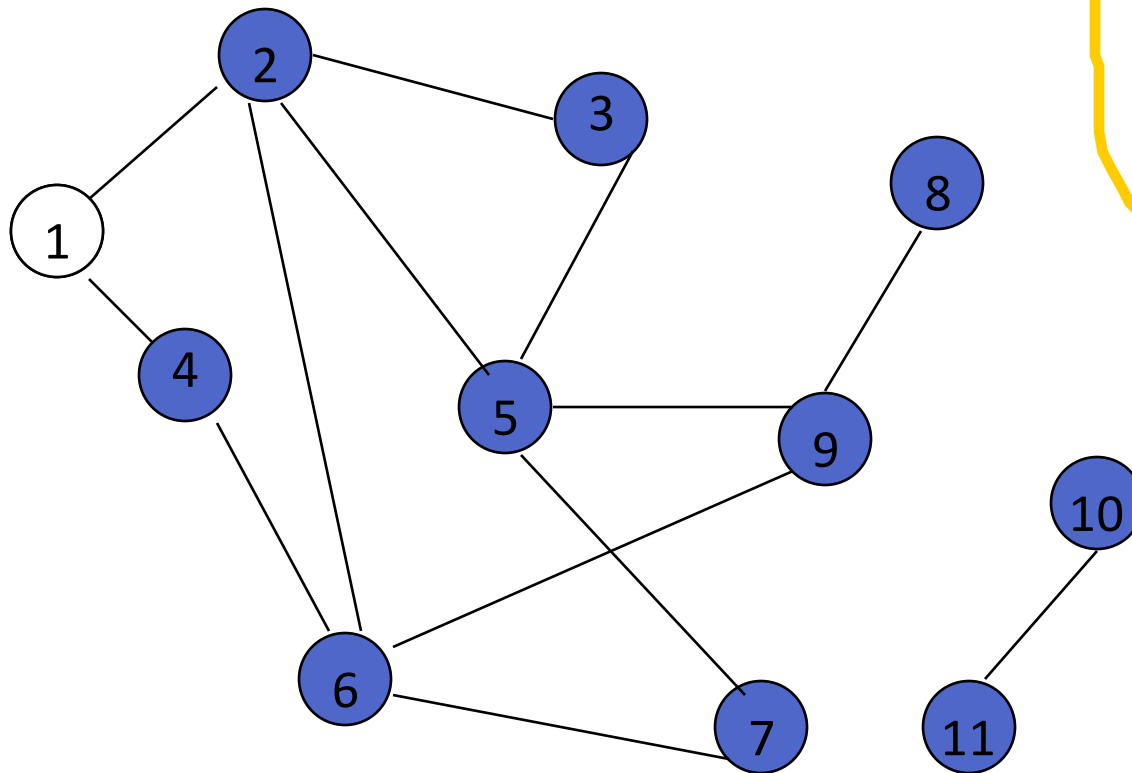
Breadth-First Search Example



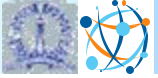
Start search at vertex 1.



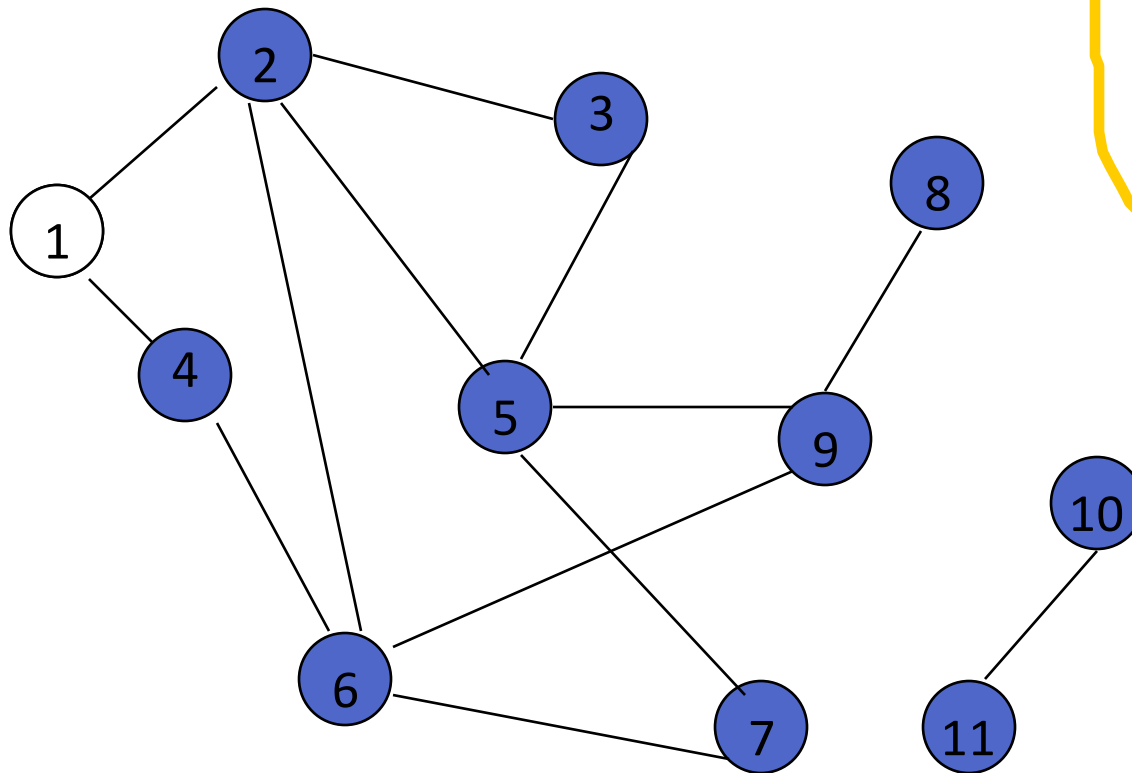
Breadth-First Search Example



Visit/mark/label start vertex and put in a FIFO queue.



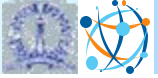
Breadth-First Search Example



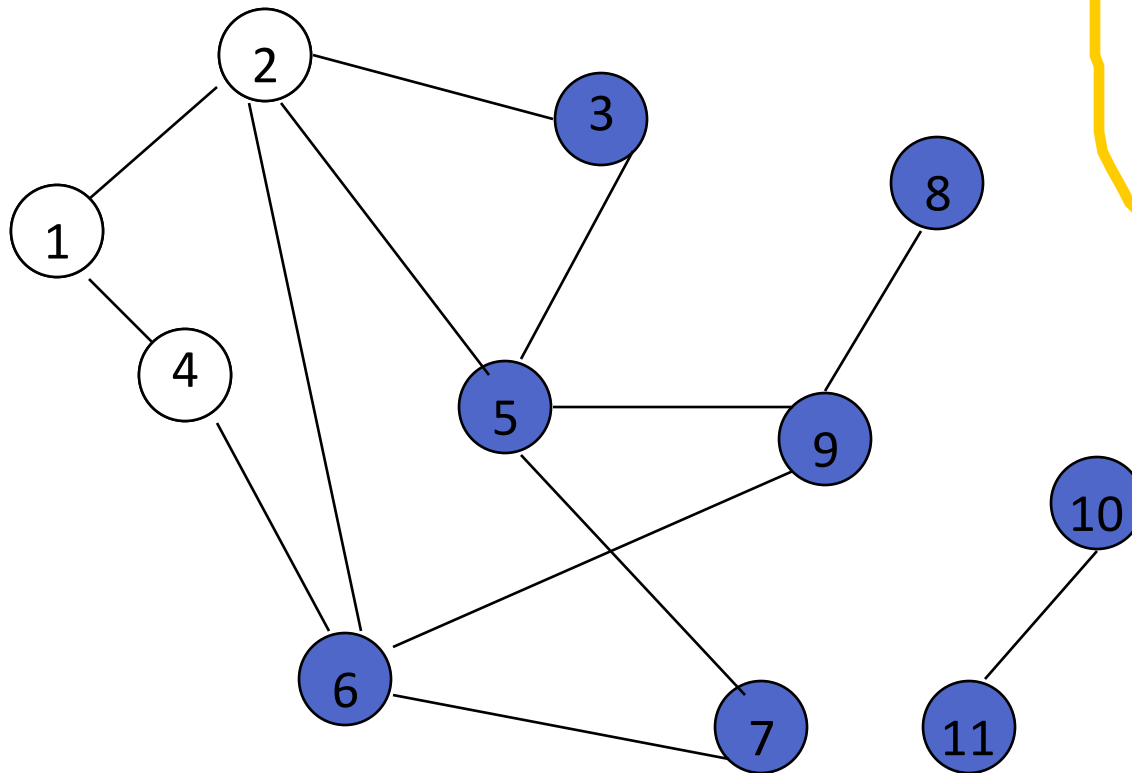
FIFO Queue

1

Remove 1 from Q; visit adjacent unvisited vertices;
put in Q.



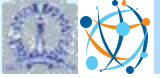
Breadth-First Search Example



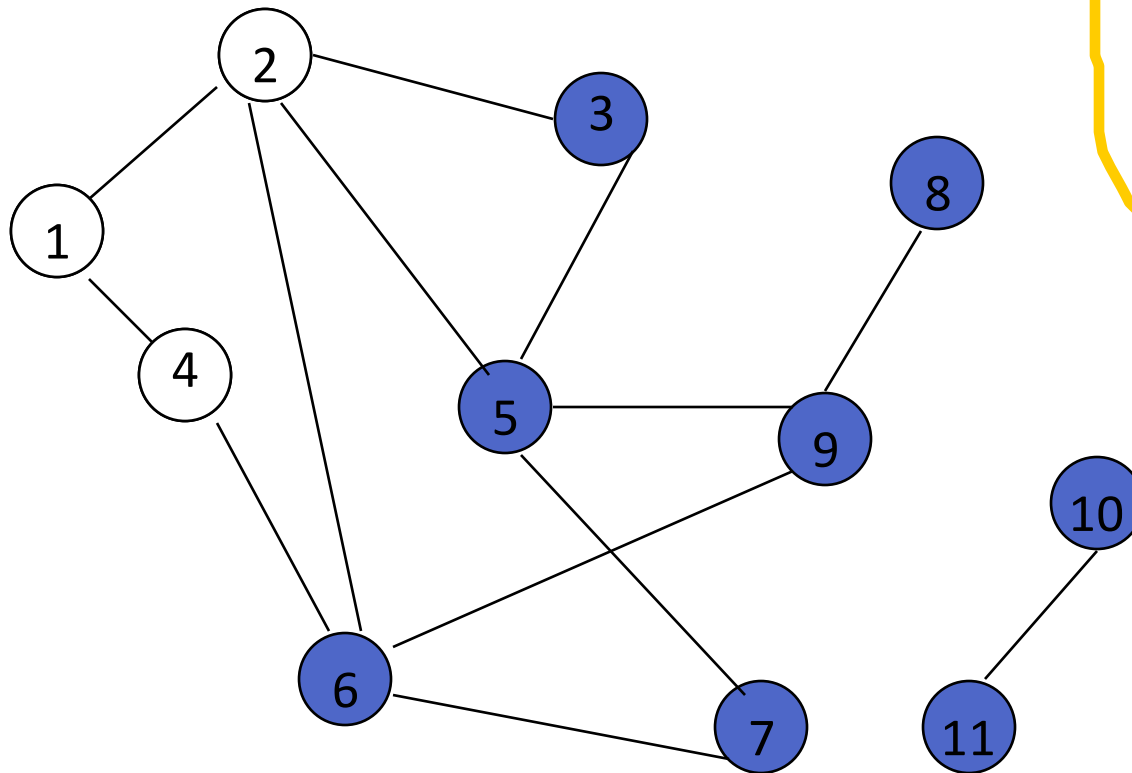
FIFO Queue

2 4

Remove 1 from Q; visit adjacent unvisited vertices;
put in Q.



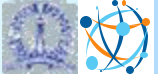
Breadth-First Search Example



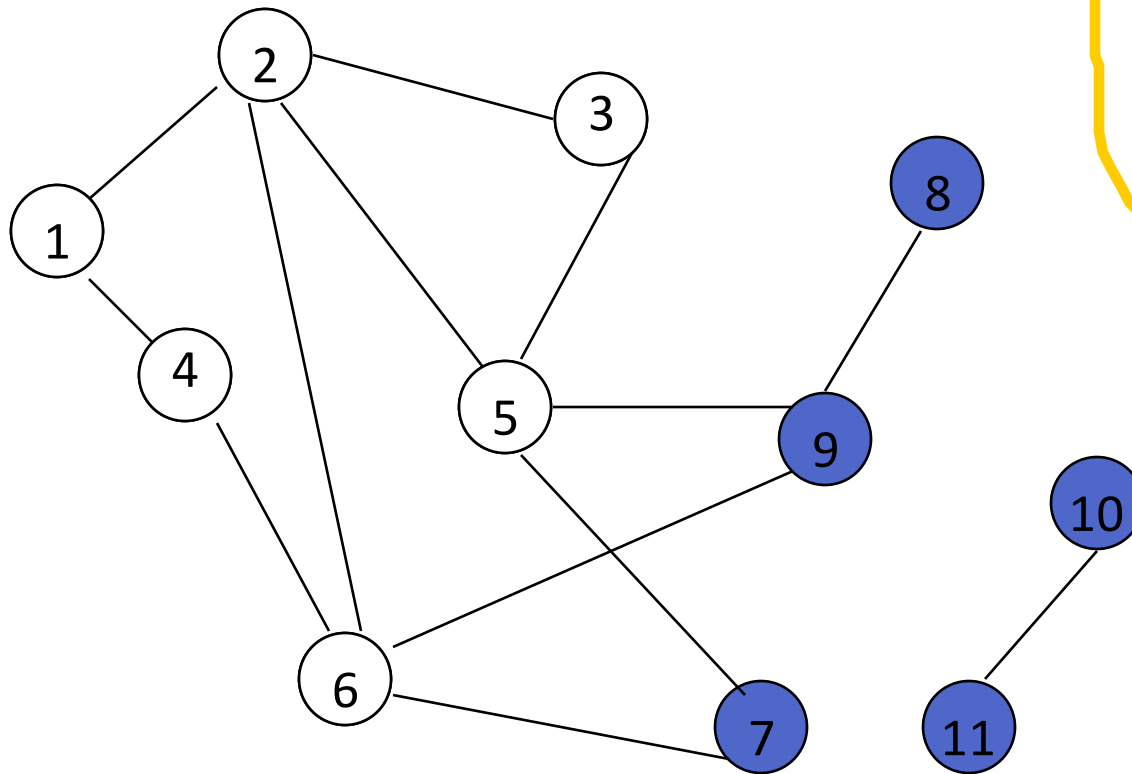
FIFO Queue

2 4

Remove 2 from Q; visit adjacent unvisited vertices;
put in Q.



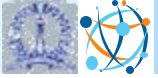
Breadth-First Search Example



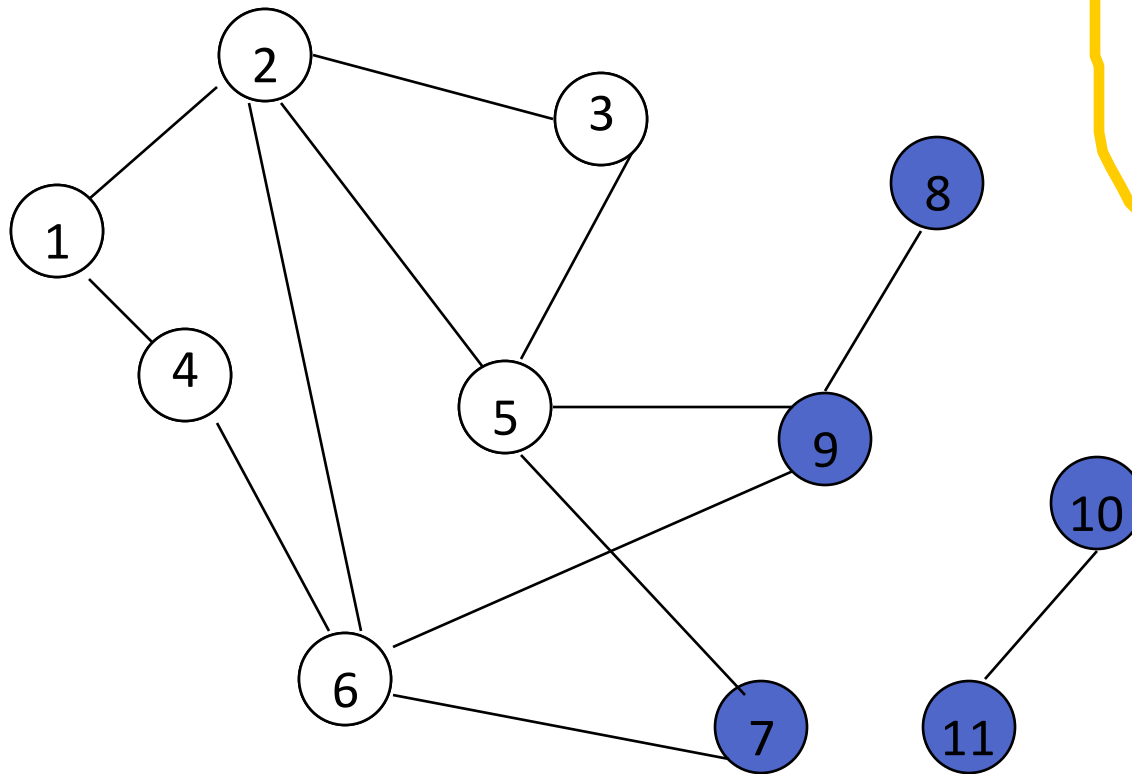
FIFO Queue

4 5 3 6

Remove 2 from Q; visit adjacent unvisited vertices;
put in Q.



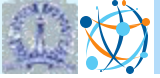
Breadth-First Search Example



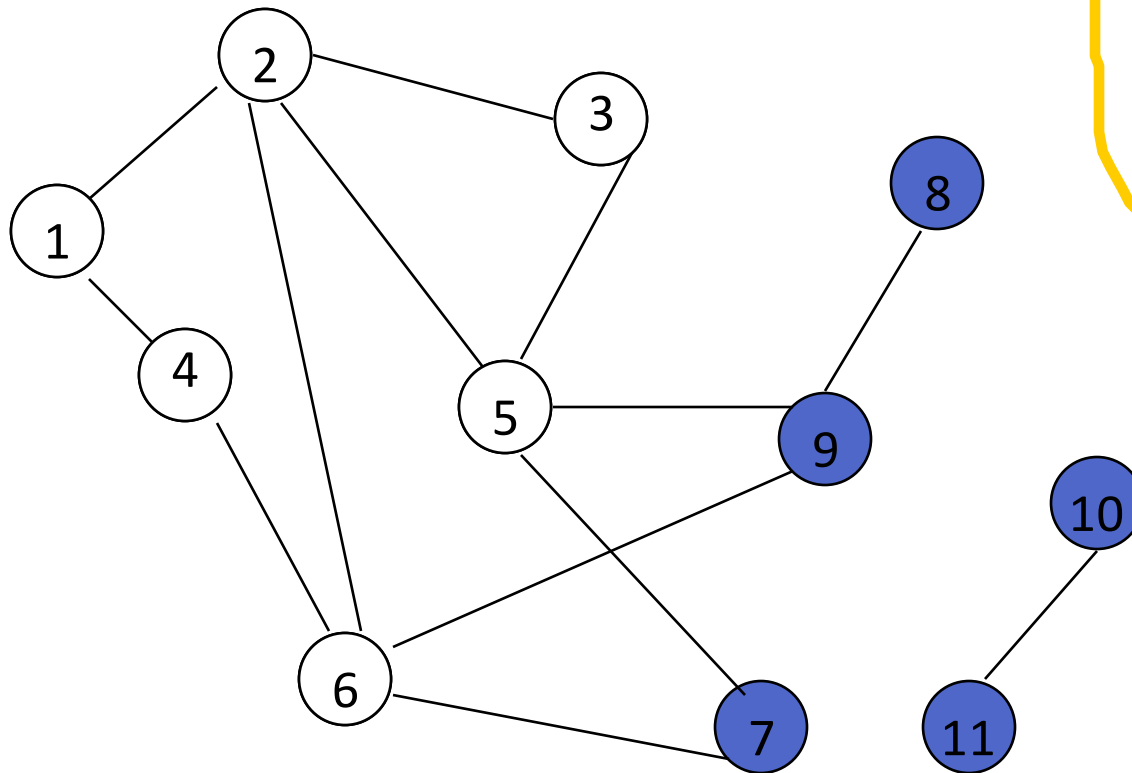
FIFO Queue

4 5 3 6

Remove 4 from Q; visit adjacent unvisited vertices;
put in Q.



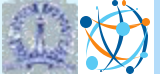
Breadth-First Search Example



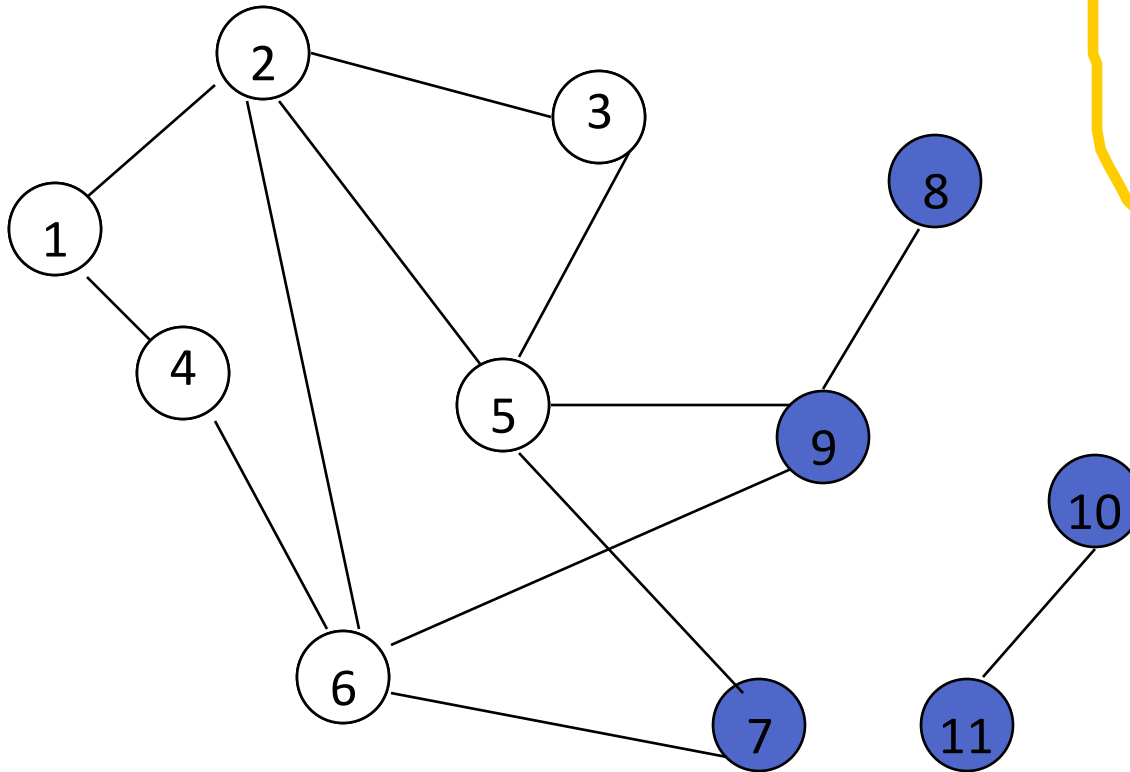
FIFO Queue

5 3 6

Remove 4 from Q; visit adjacent unvisited vertices;
put in Q.



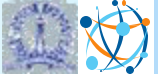
Breadth-First Search Example



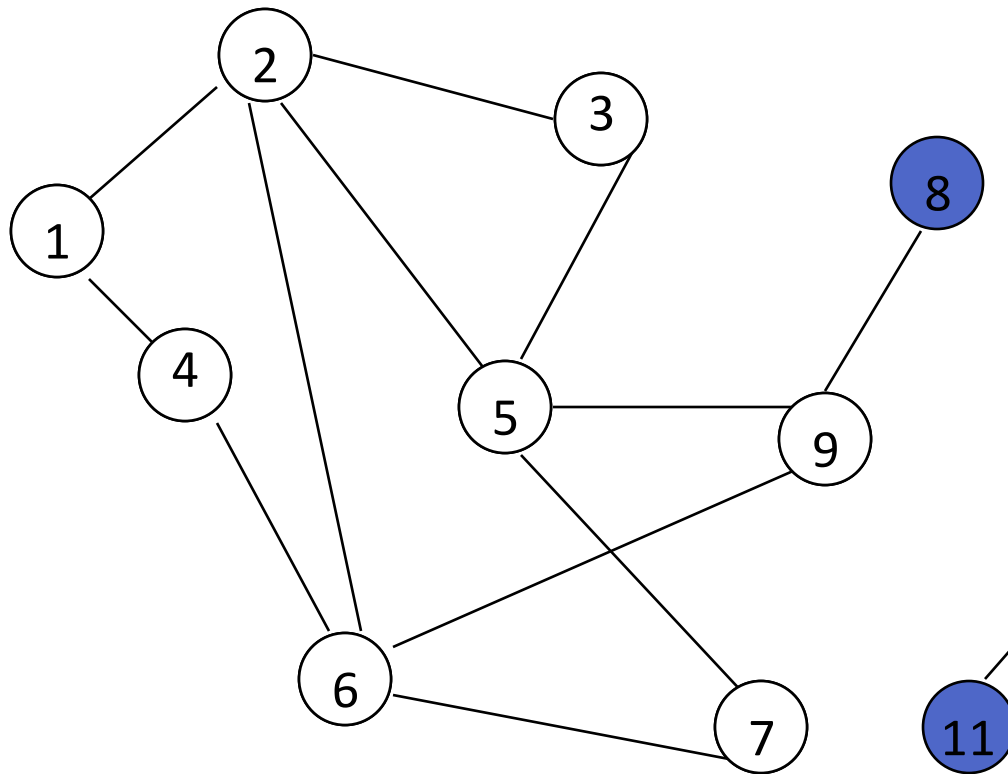
FIFO Queue

5 3 6

Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.



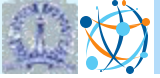
Breadth-First Search Example



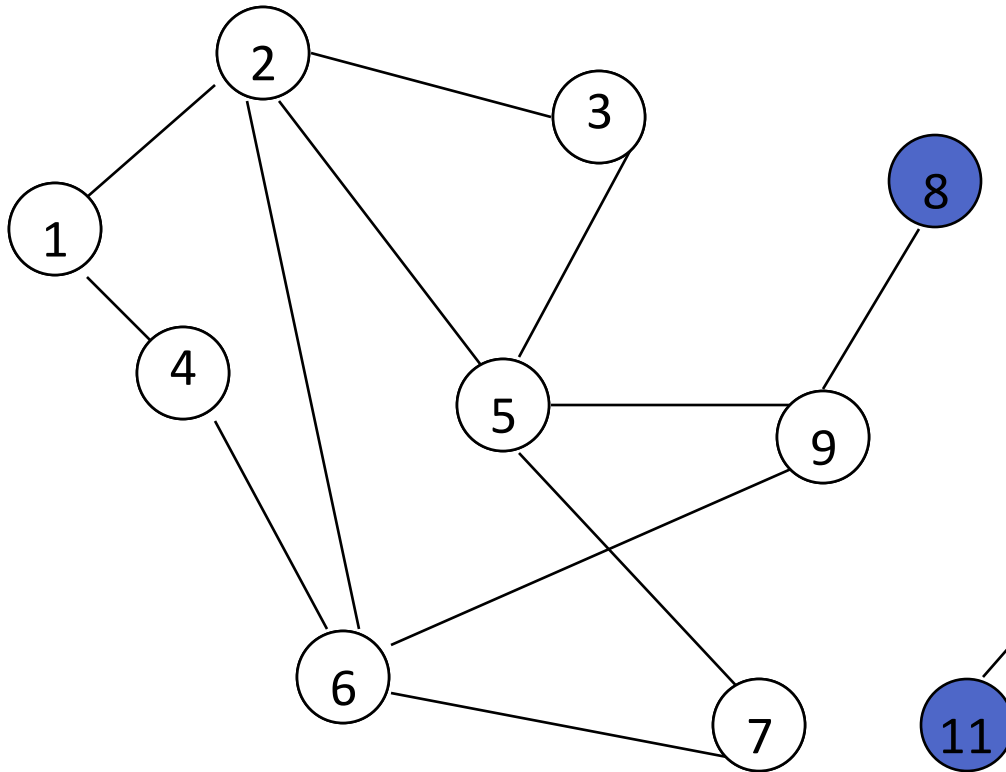
FIFO Queue

3 6 9 7

Remove 5 from Q; visit adjacent unvisited vertices;
put in Q.



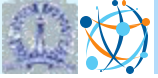
Breadth-First Search Example



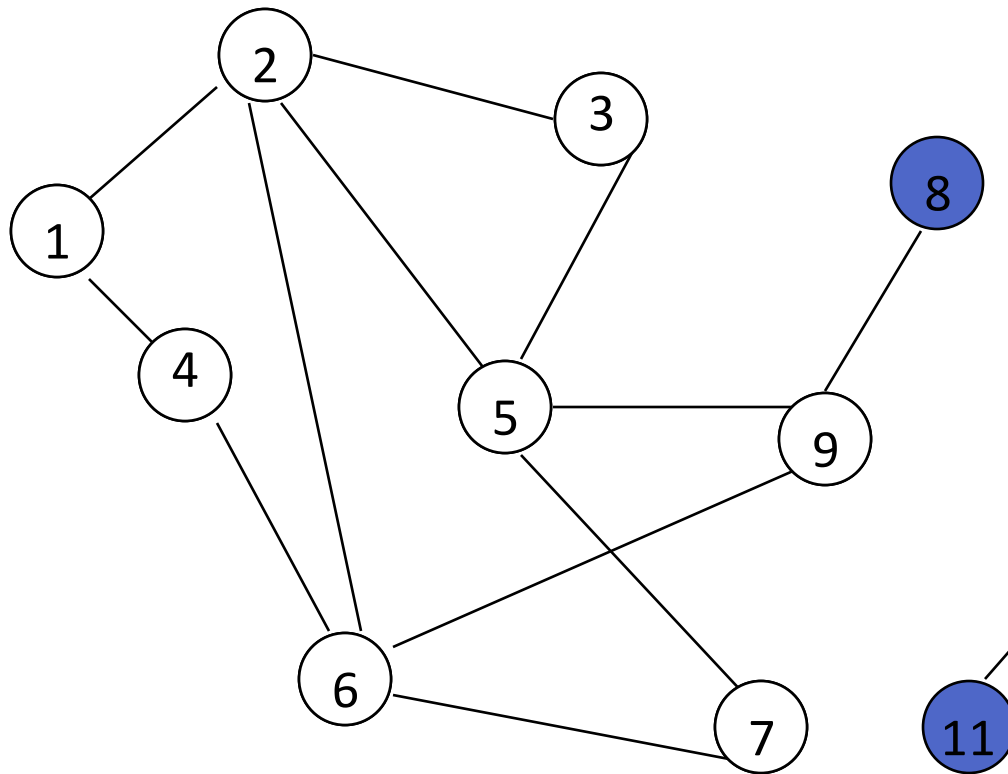
FIFO Queue

3 6 9 7

Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.



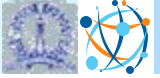
Breadth-First Search Example



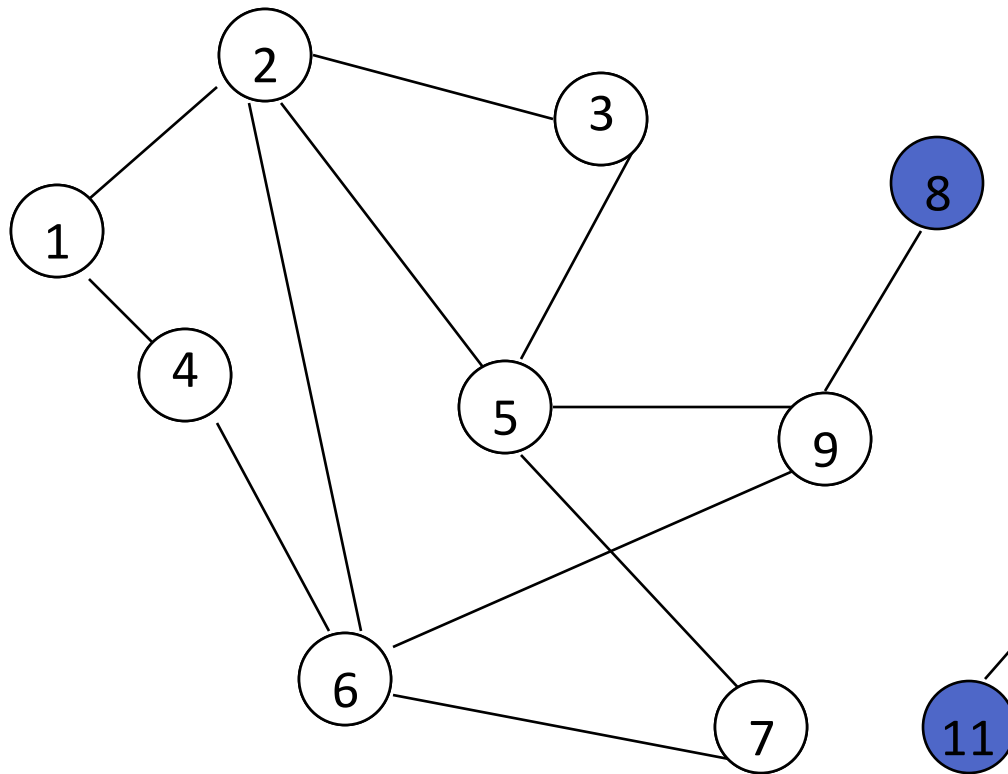
FIFO Queue

6 9 7

Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.



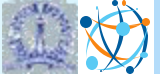
Breadth-First Search Example



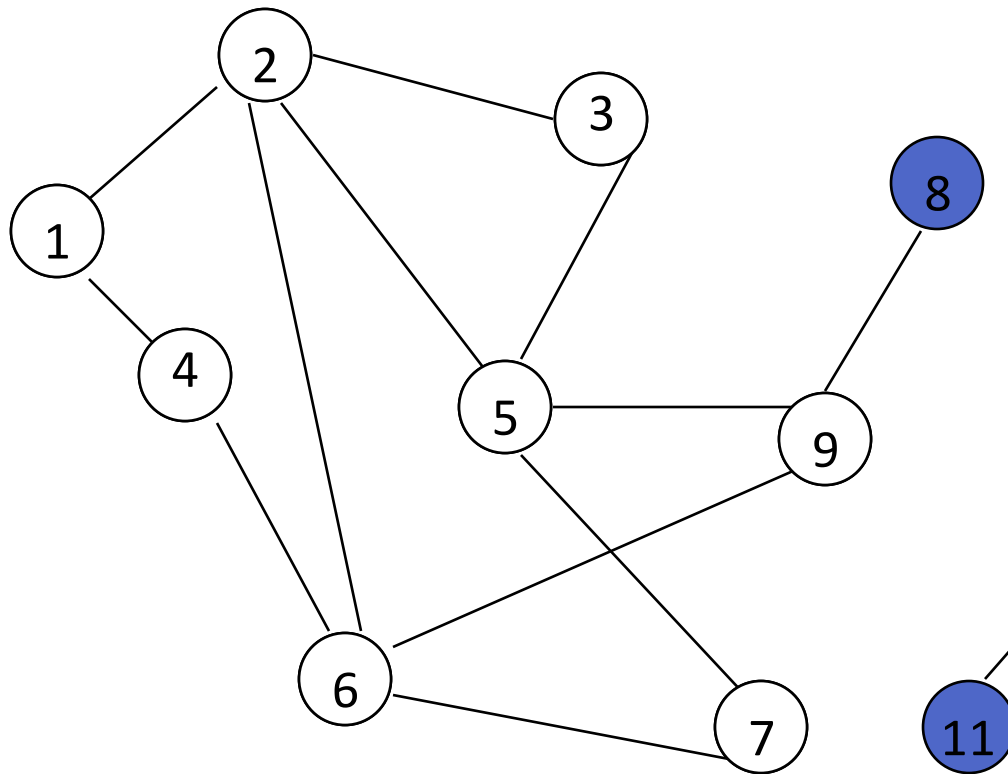
FIFO Queue

6 9 7

Remove 6 from Q; visit adjacent unvisited vertices;
put in Q.



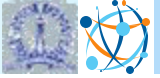
Breadth-First Search Example



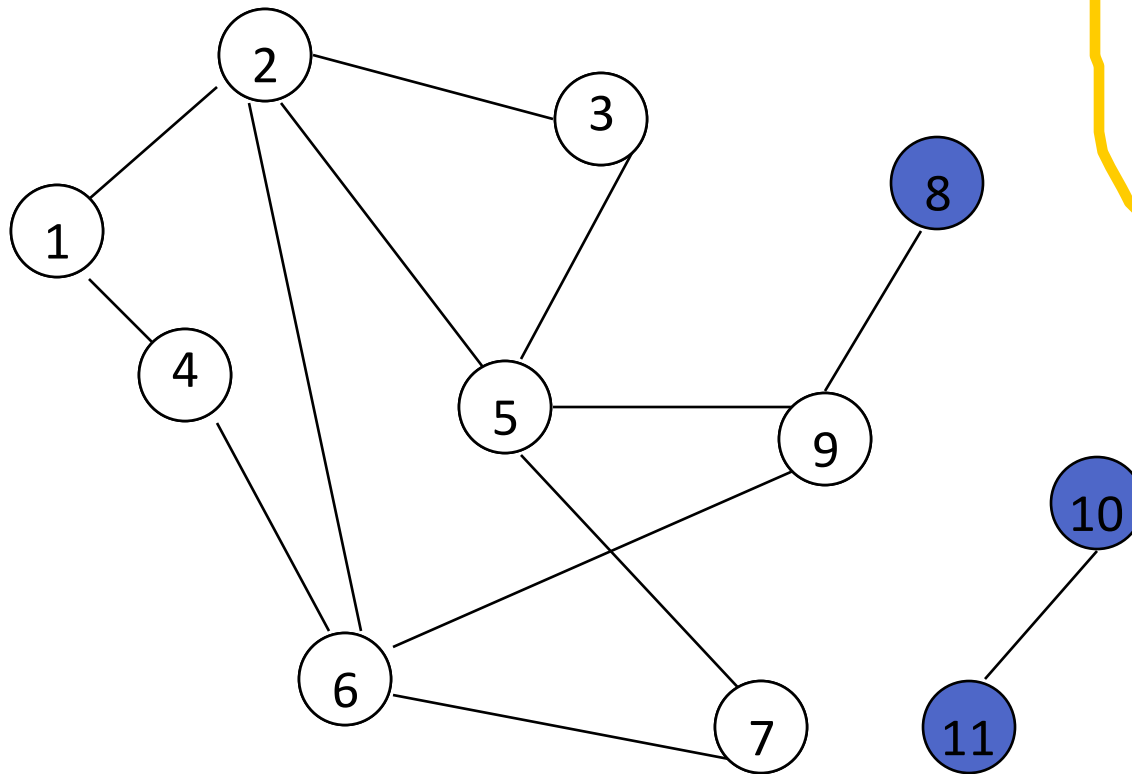
FIFO Queue

9 7

Remove 6 from Q; visit adjacent unvisited vertices;
put in Q.



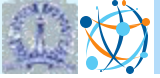
Breadth-First Search Example



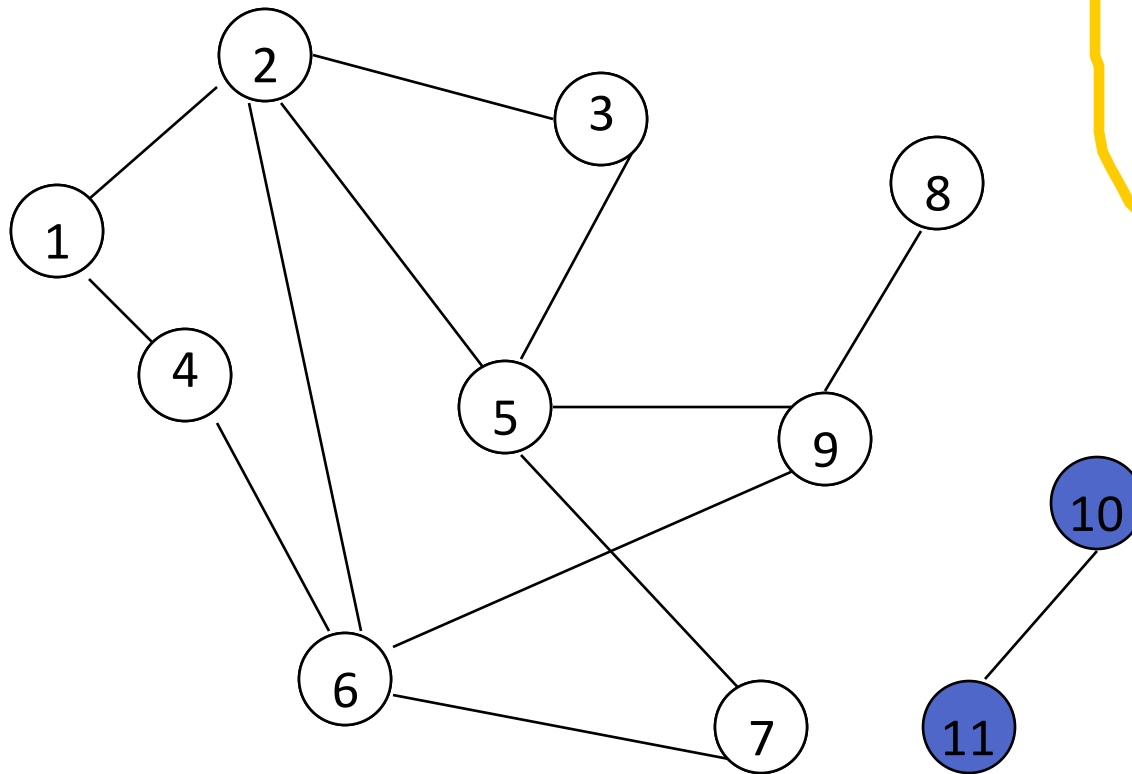
FIFO Queue

9 7

Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.



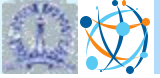
Breadth-First Search Example



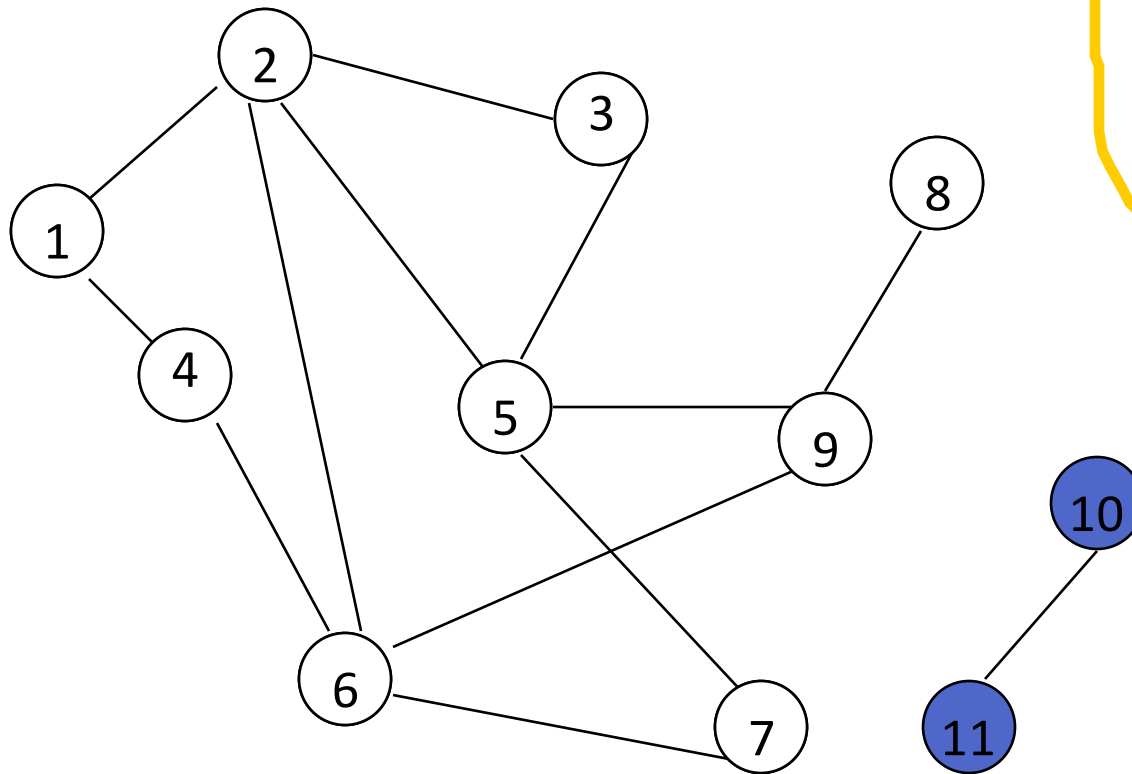
FIFO Queue

7 8

Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.



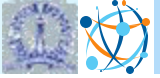
Breadth-First Search Example



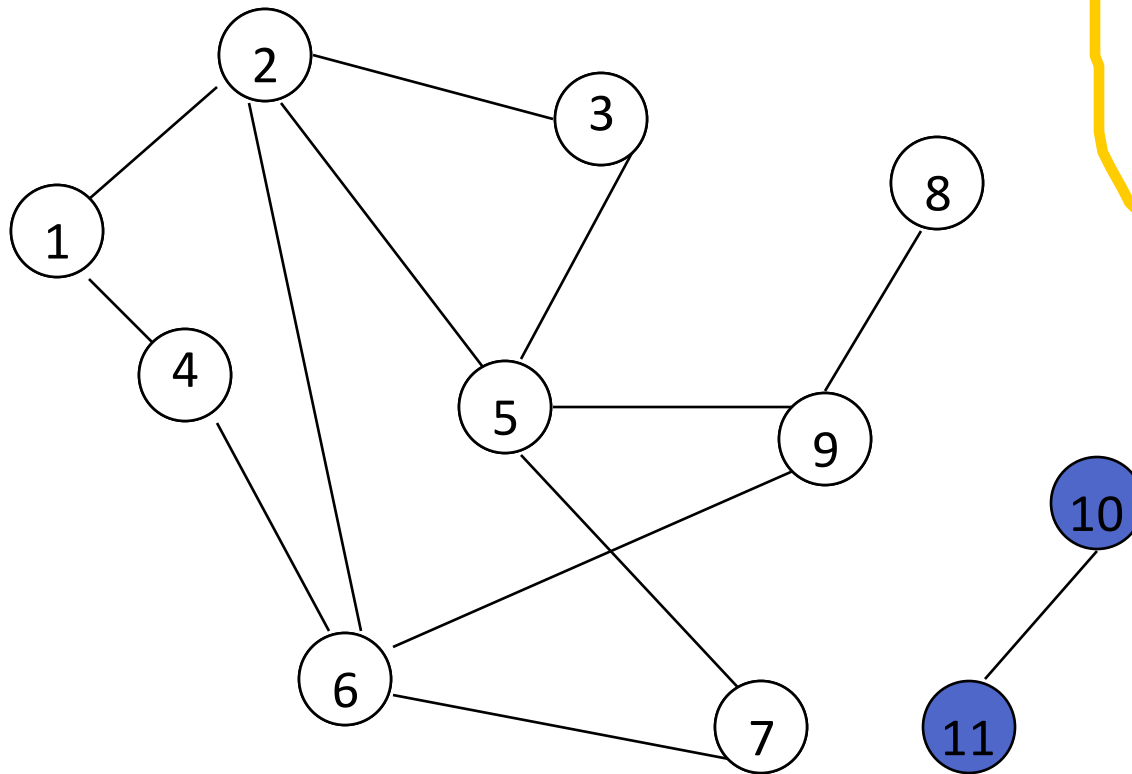
FIFO Queue

7 8

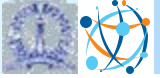
Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.



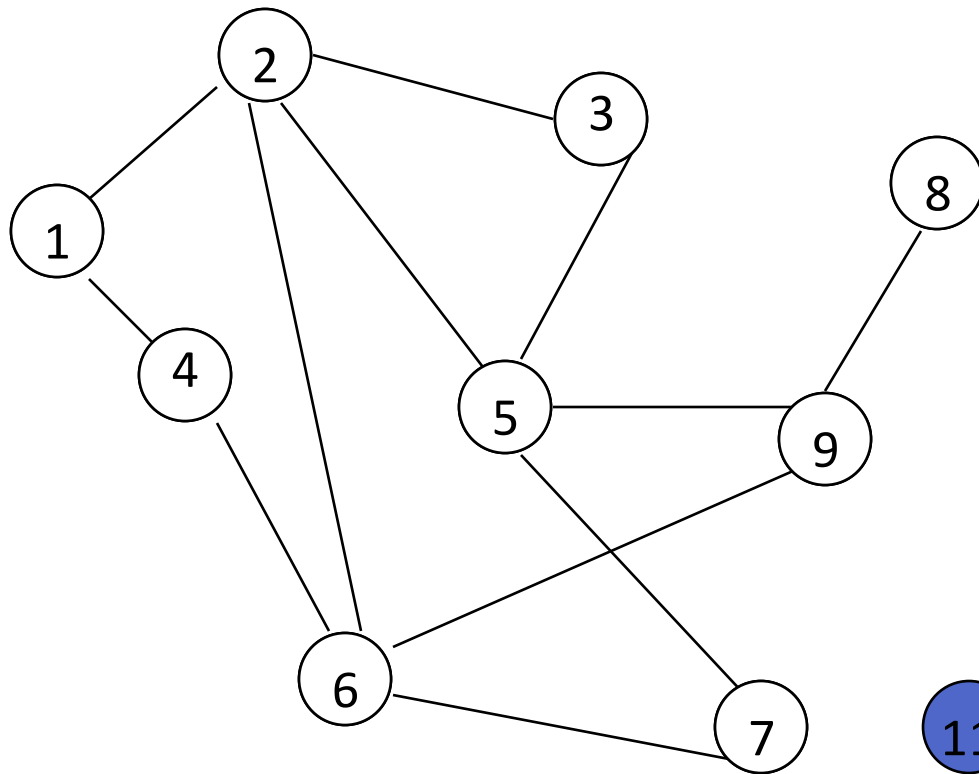
Breadth-First Search Example



Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.



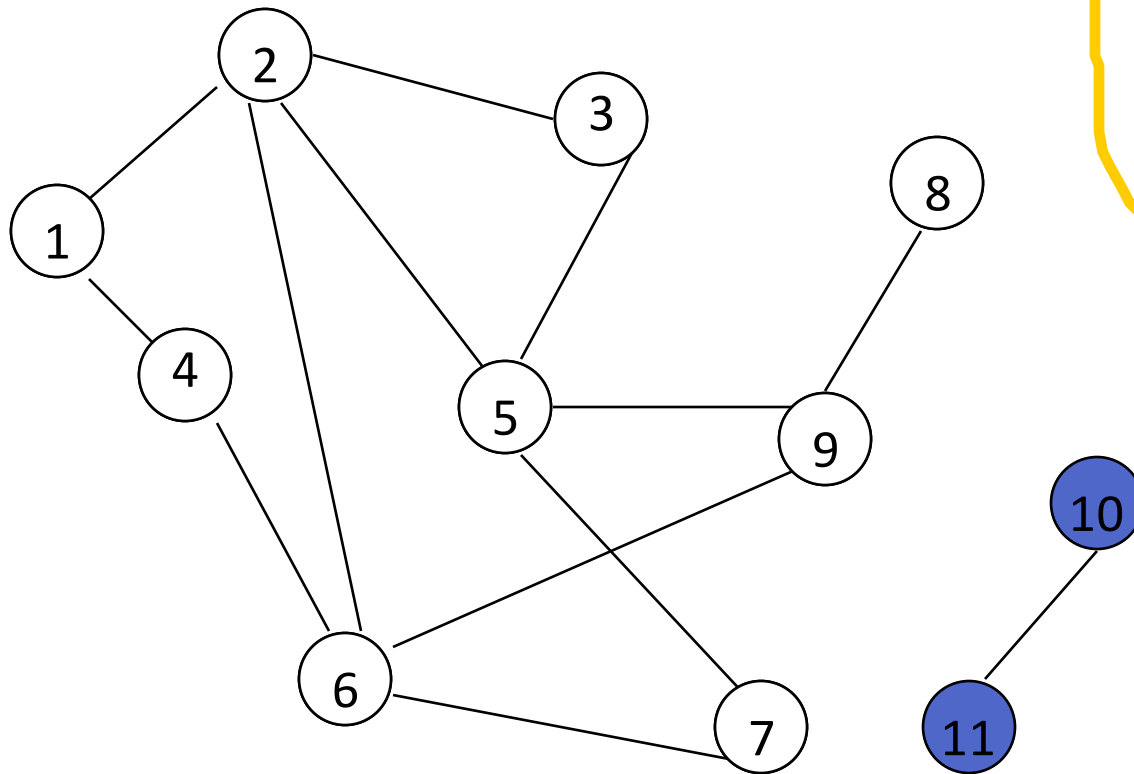
Breadth-First Search Example



Remove 8 from Q; visit adjacent unvisited vertices;
put in Q.

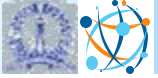


Breadth-First Search Example



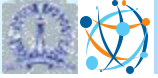
FIFO Queue

Queue is empty. Search terminates.



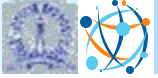
Breadth-First Search Property

- All vertices reachable from the start vertex (including the start vertex) are visited.



Time Complexity

- Each visited vertex is added to (and so removed from) the queue exactly once
- When a vertex is removed from the queue, we examine its adjacent vertices
 - $O(|V|)$ if adjacency matrix is used, where $|V|$ is number of vertices in whole graph
 - $O(d)$ if adjacency list is used, where d is *edge degree*
- Total time
 - Adjacency matrix: $O(w|V|)$, where w is number of vertices in the *connected component* that is searched
 - Adjacency list: $O(w+f)$, where f is number of edges in the *connected component* that is searched

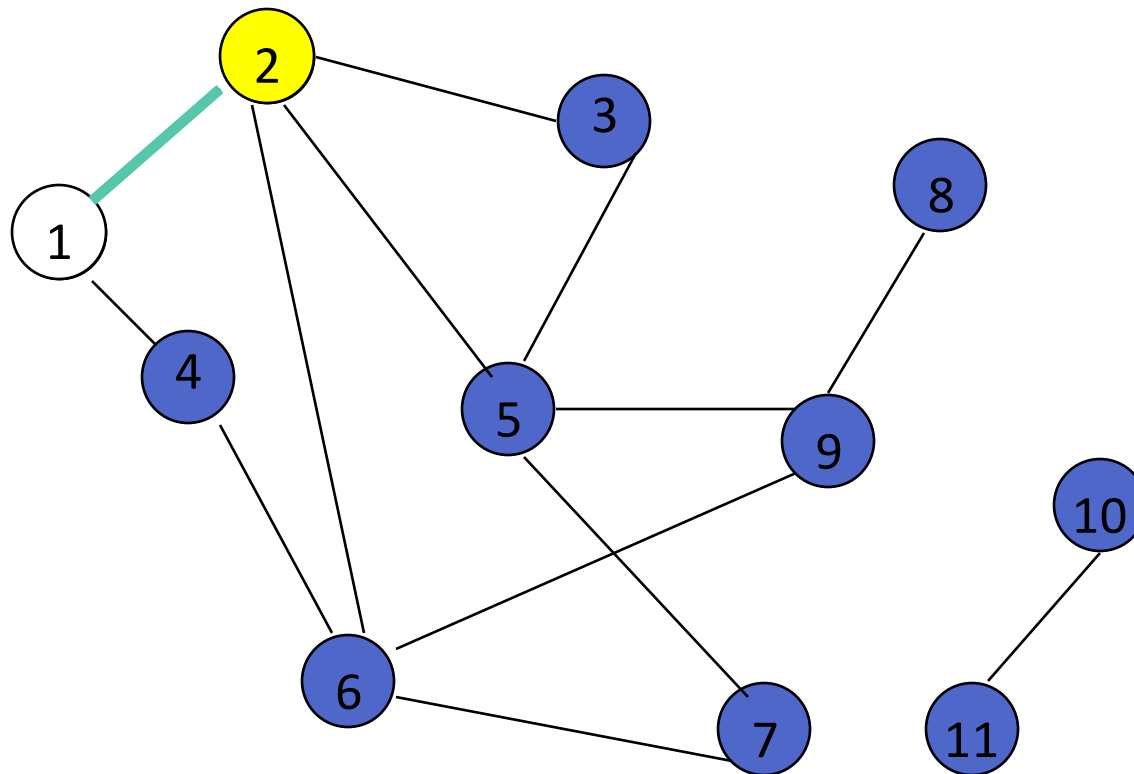


Depth-First Search

```
depthFirstSearch(v) {  
    Label vertex v as reached;  
    for(each unreached vertex u  
        adjacent to v)  
        depthFirstSearch(u);  
}
```



Depth-First Search Example



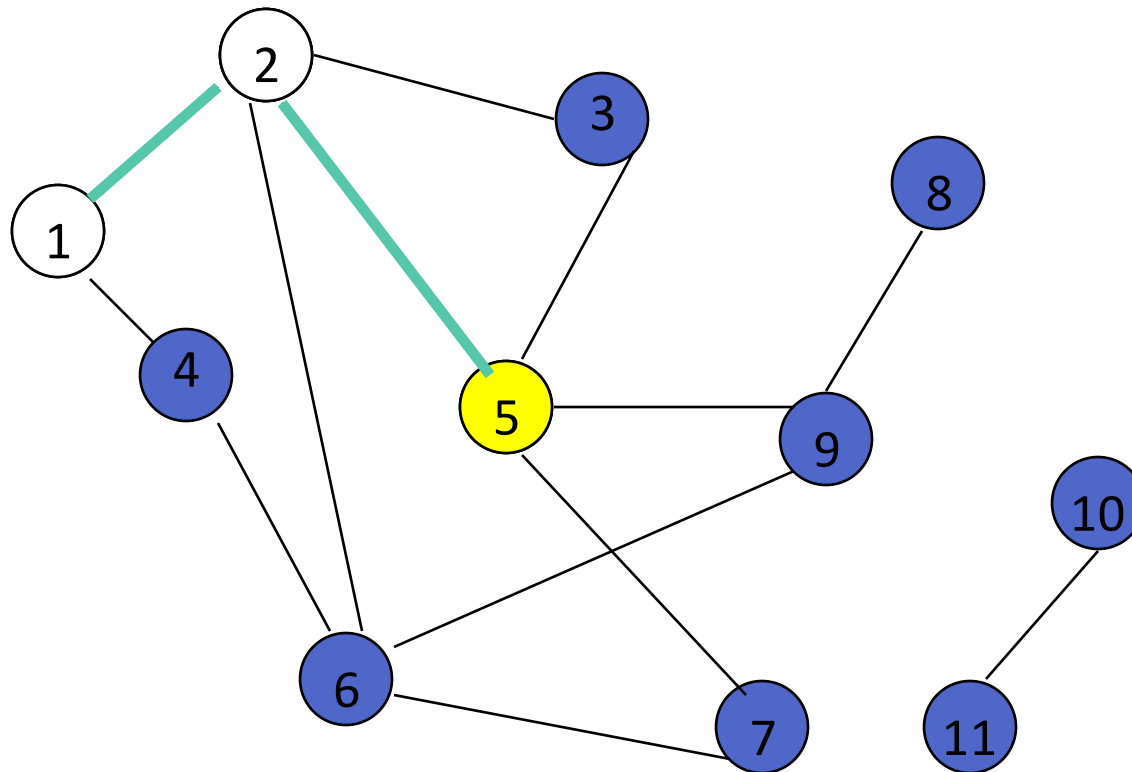
Start search at vertex 1.

Label vertex 1 and do a depth first search from either 2 or 4.

Suppose that vertex 2 is selected.



Depth-First Search Example

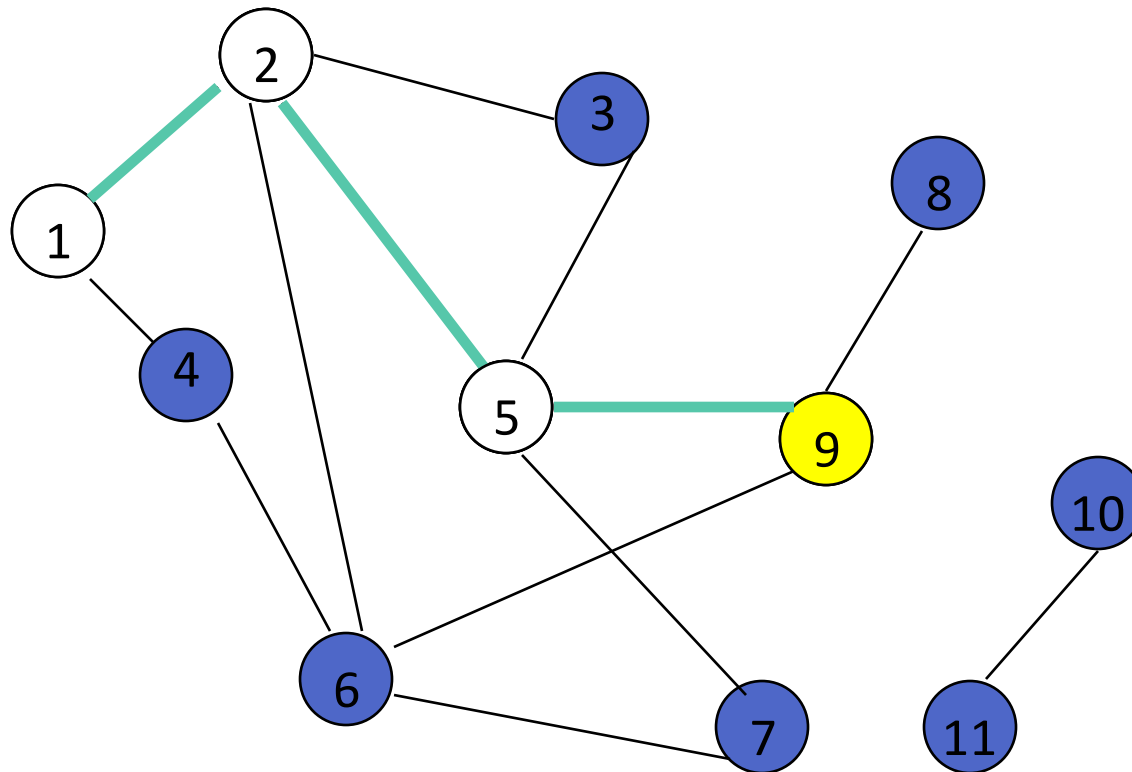


Label vertex 2 and do a depth first search from either 3, 5, or 6.

Suppose that vertex 5 is selected.



Depth-First Search

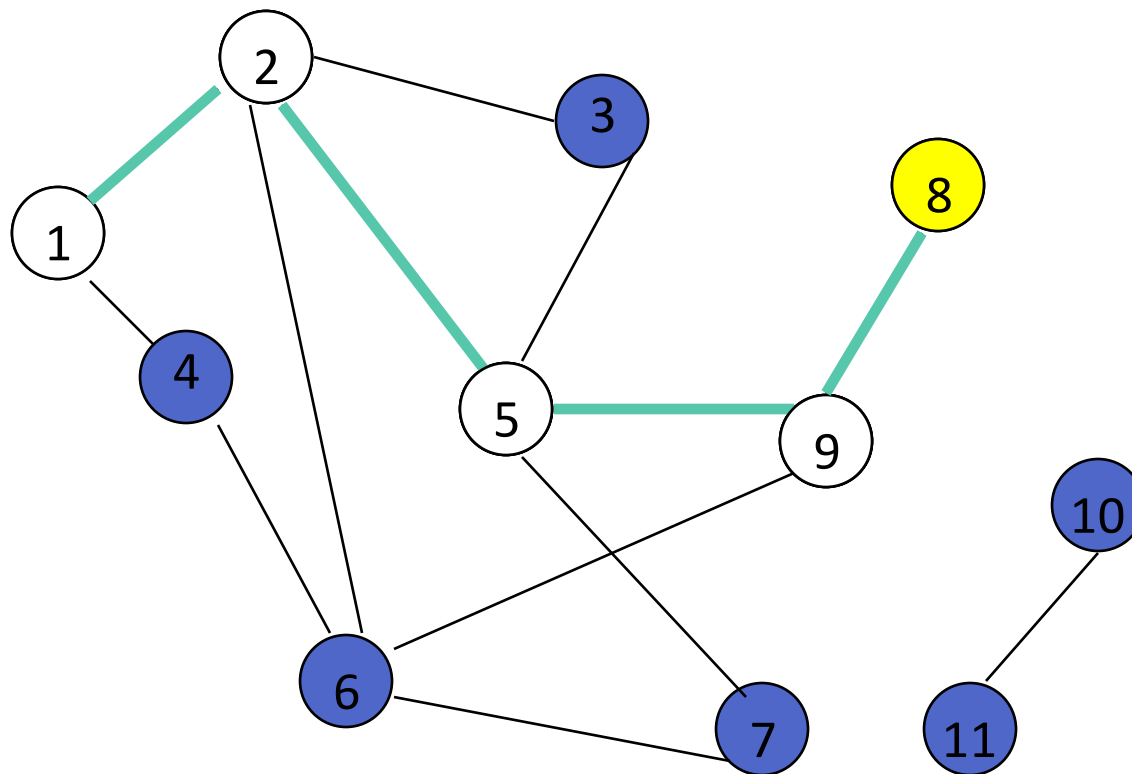


Label vertex 5 and do a depth first search from either 3, 7, or 9.

Suppose that vertex 9 is selected.



Depth-First Search

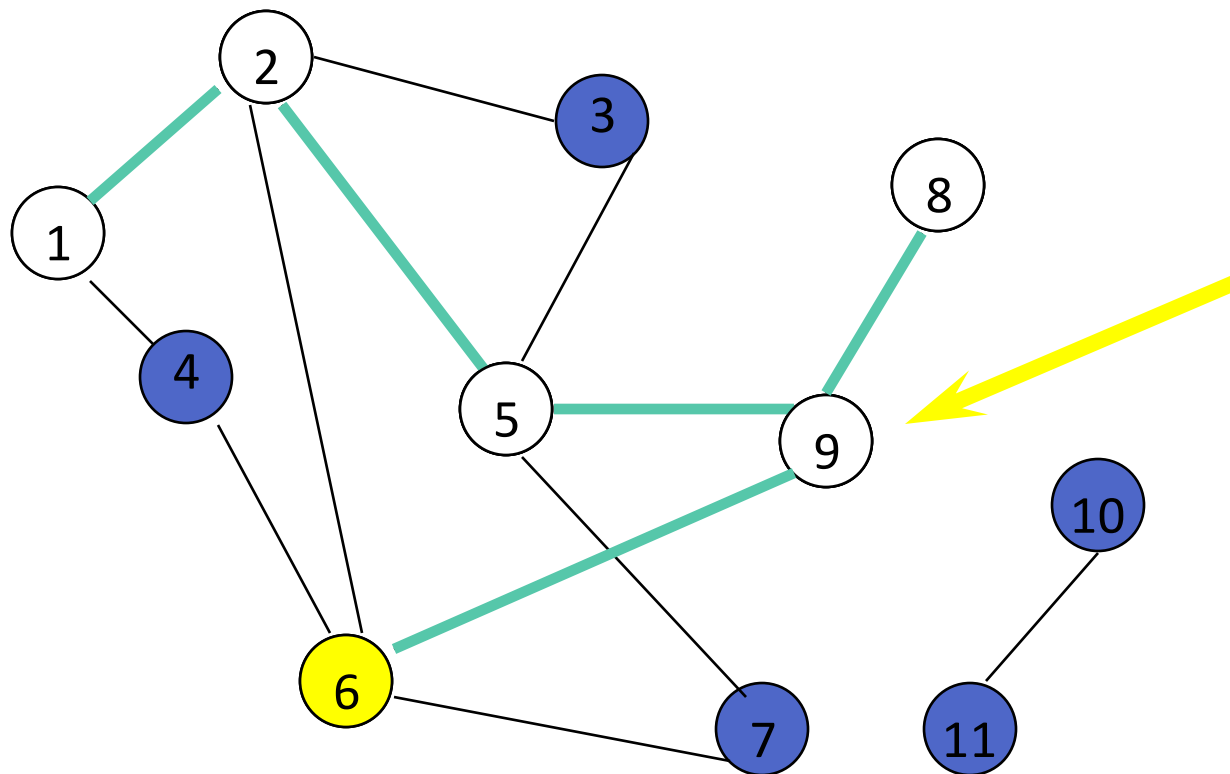


Label vertex 9 and do a depth first search from either 6 or 8.

Suppose that vertex 8 is selected.



Depth-First Search

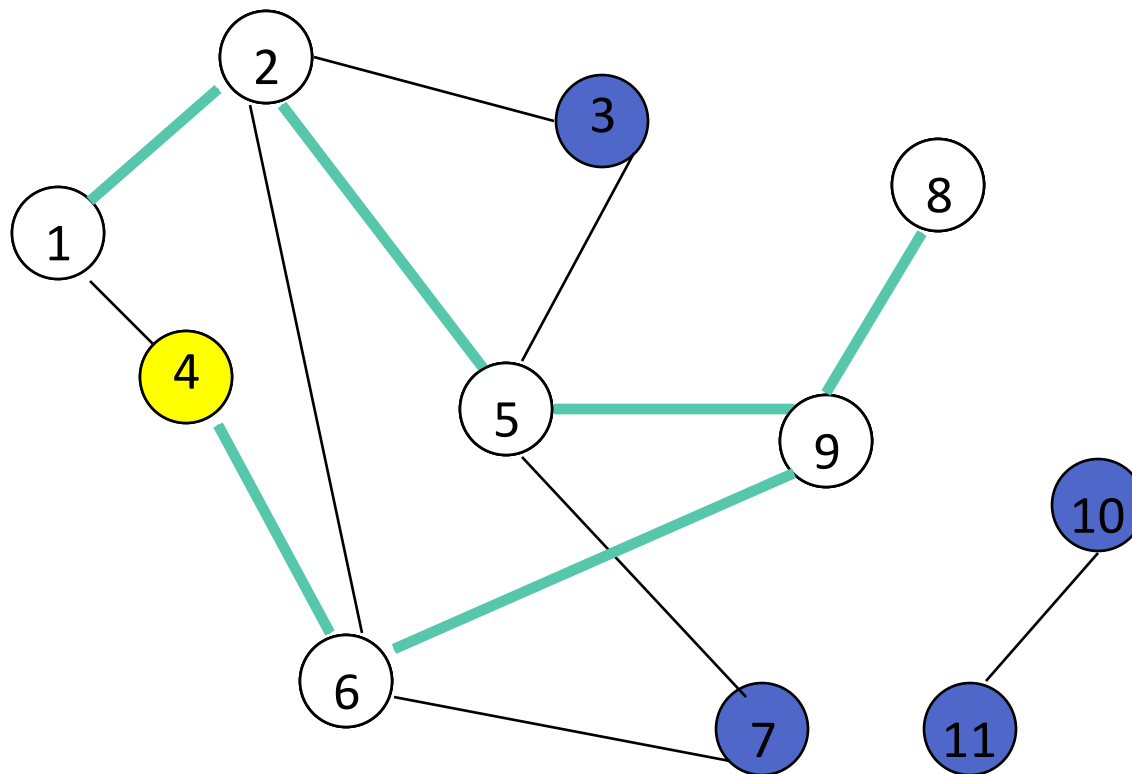


Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6)



Depth-First Search

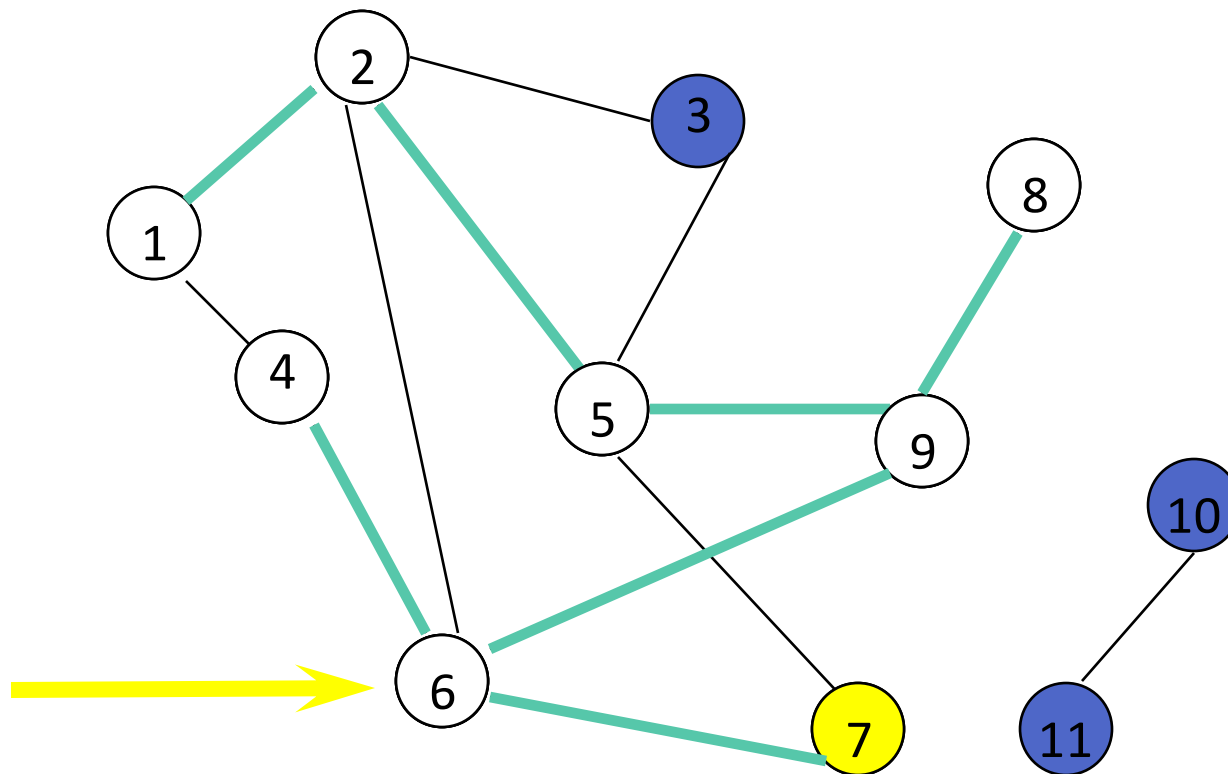


Label vertex 6 and do a depth first search from either 4 or 7.

Suppose that vertex 4 is selected.



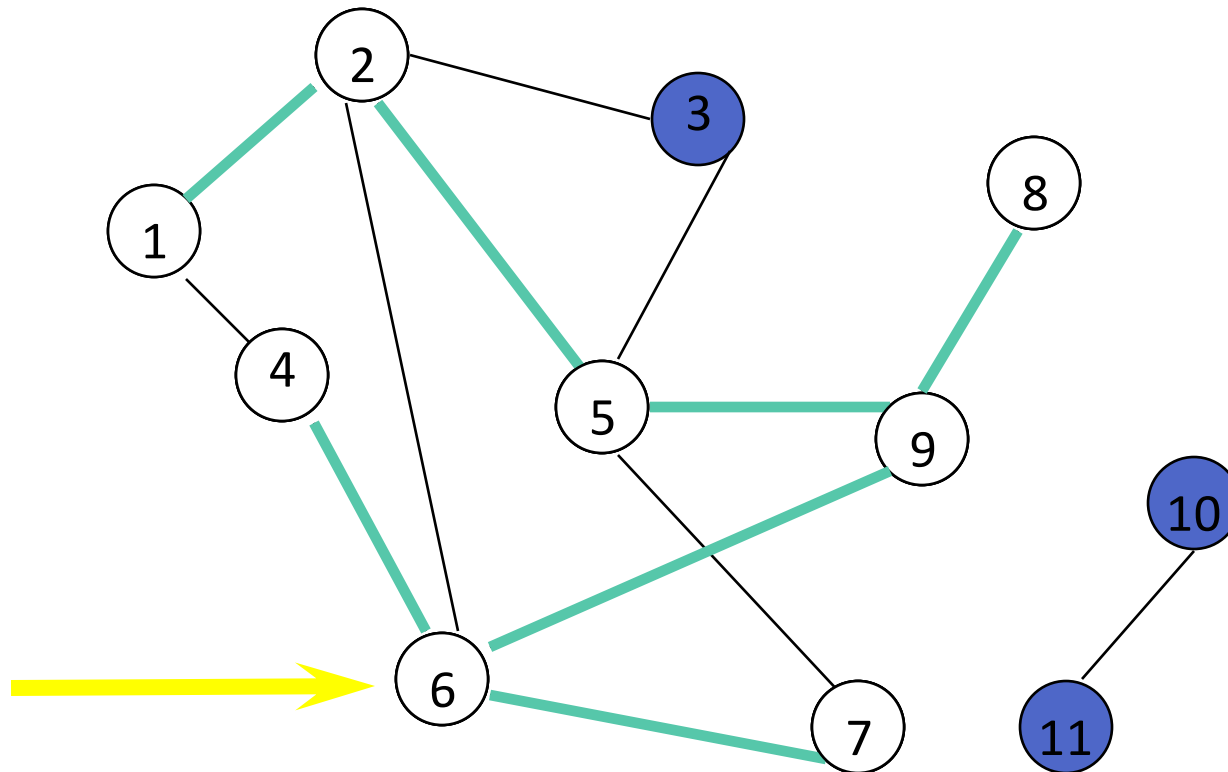
Depth-First Search



Label vertex 4 and return to 6.
From vertex 6 do a dfs(7).



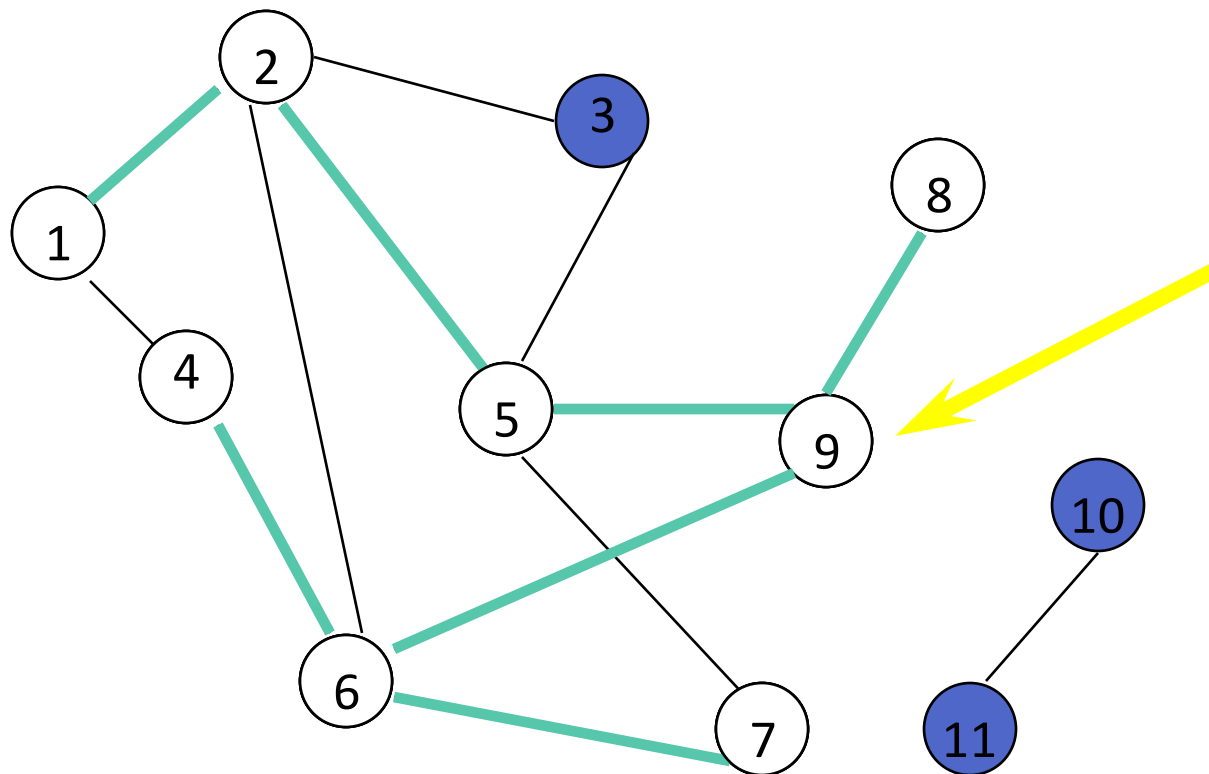
Depth-First Search



Label vertex 7 and return to 6.
Return to 9.



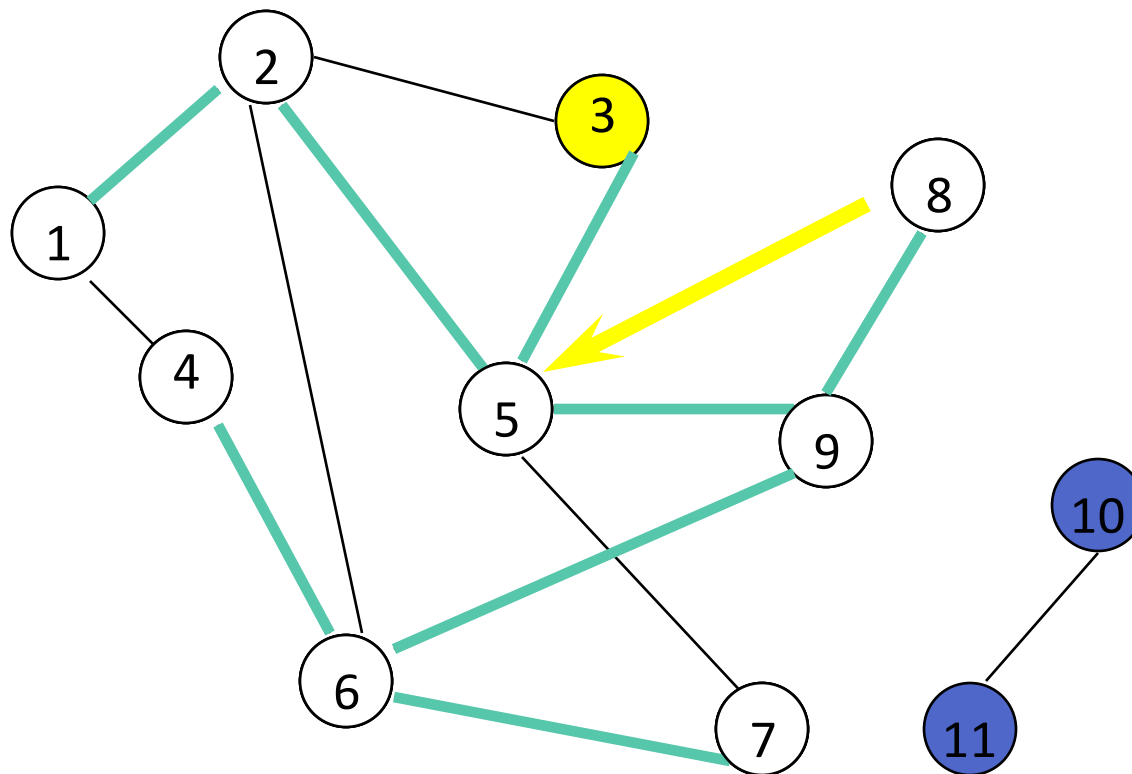
Depth-First Search



Return to 5.



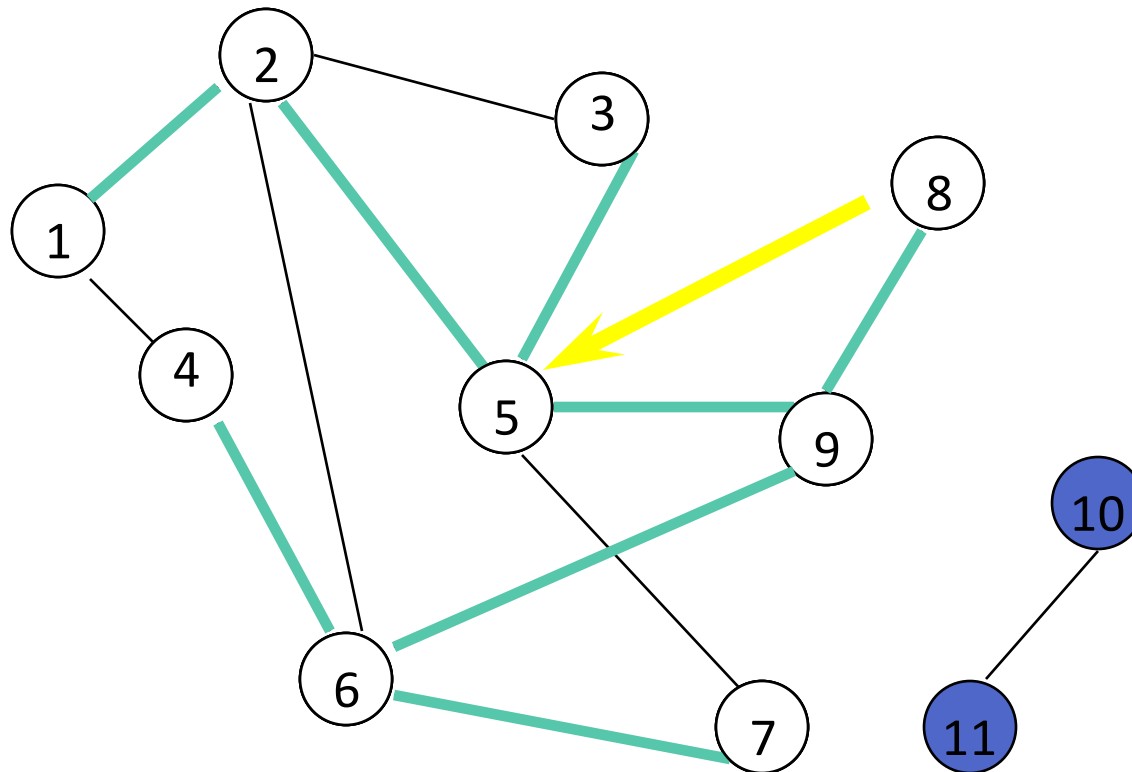
Depth-First Search



Do a dfs(3).



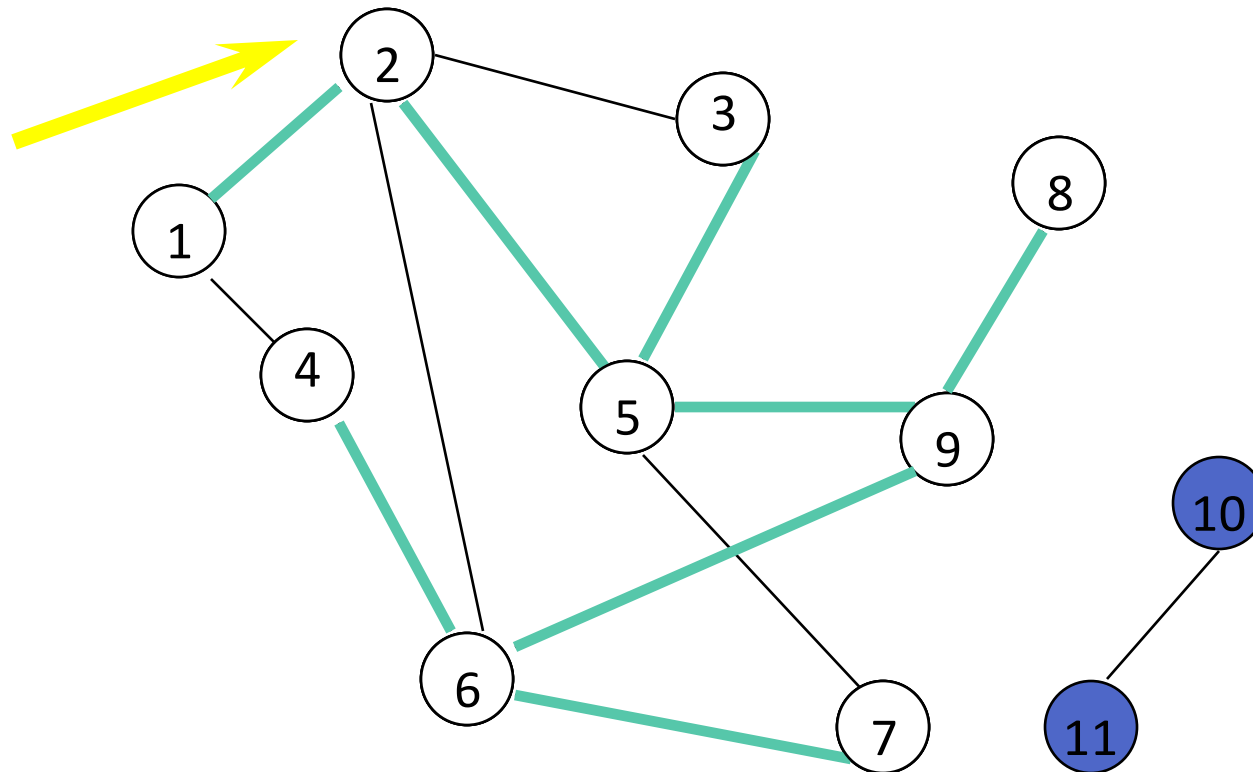
Depth-First Search



Label 3 and return to 5.
Return to 2.



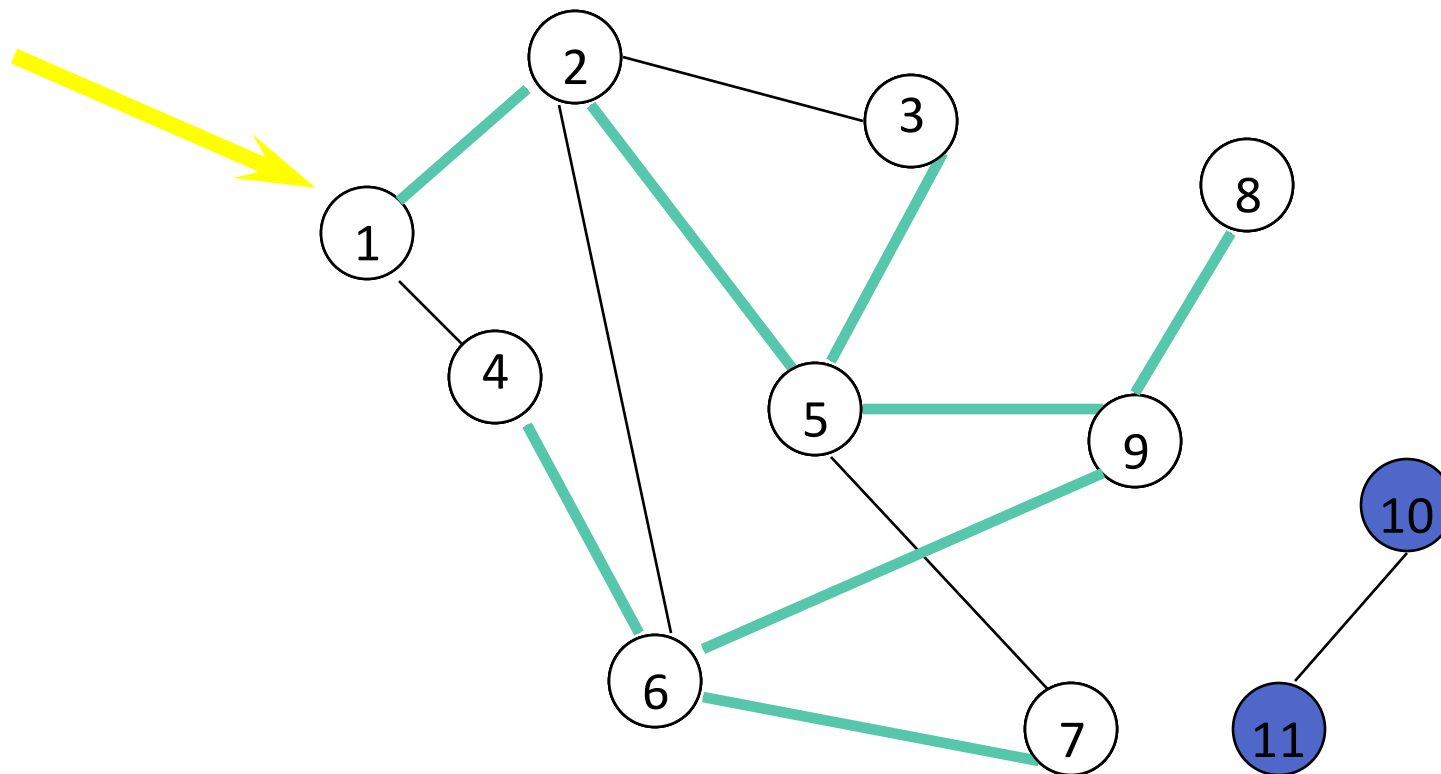
Depth-First Search



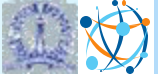
Return to 1.



Depth-First Search

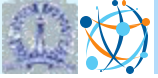


Return to invoking method.



DFS Properties

- DFS has same time complexity as BFS
- DFS requires $O(h)$ memory for recursive function stack calls while BFS requires $O(w)$ queue capacity
- Same properties with respect to path finding, connected components, and spanning trees.
 - Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
- One is better than the other for some problems, e.g.
 - When searching, if the item is far from source (leaves), then DFS may locate it first, and vice versa for BFS
 - BFS traverses vertices at same distance (level) from source
 - DFS can be used to detect cycles (revisits of vertices in current stack)

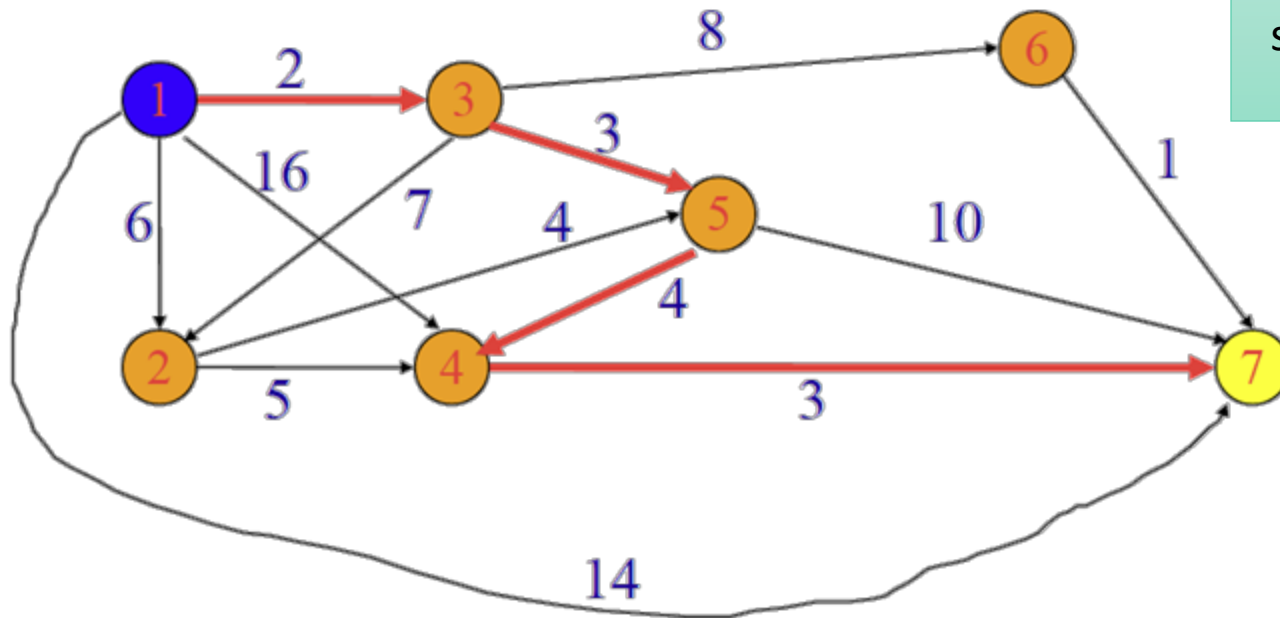


Shortest Path: Single source, single destination

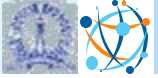
■ Possible greedy algorithm

- ▶ Leave source vertex using *shortest outgoing edge*
- ▶ Leave new vertex again using shortest outgoing edge to an *unvisited vertex*
- ▶ Continue until destination is reached

Greedy Path
from 1 To 7



Length of 12 is not shortest path!

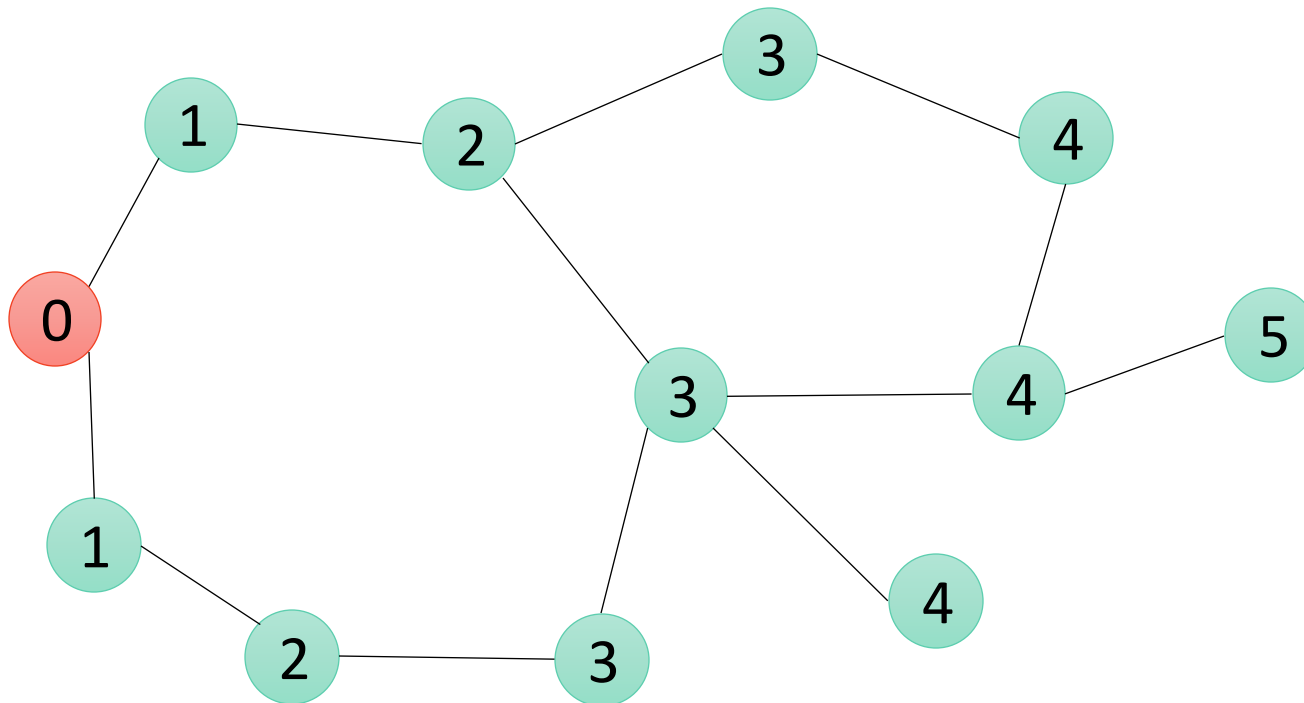


Single Source Shortest Path

- Shortest distance from **one source vertex** to all **destination vertices**
- Is there a simple way to solve this?
- ...Say if you had an unit-weighted graph?
- **Just do Breadth First Search (BFS)! 😊**

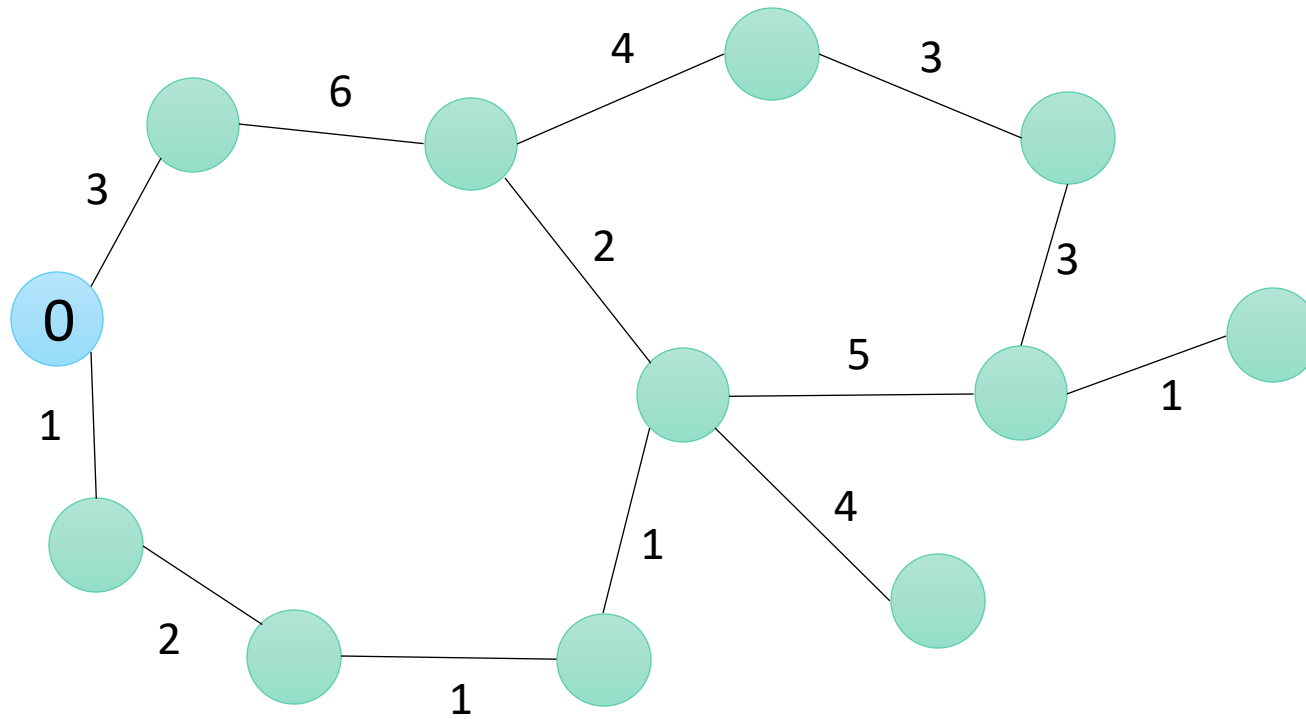


SSSP: BFS on Unweighted Graphs



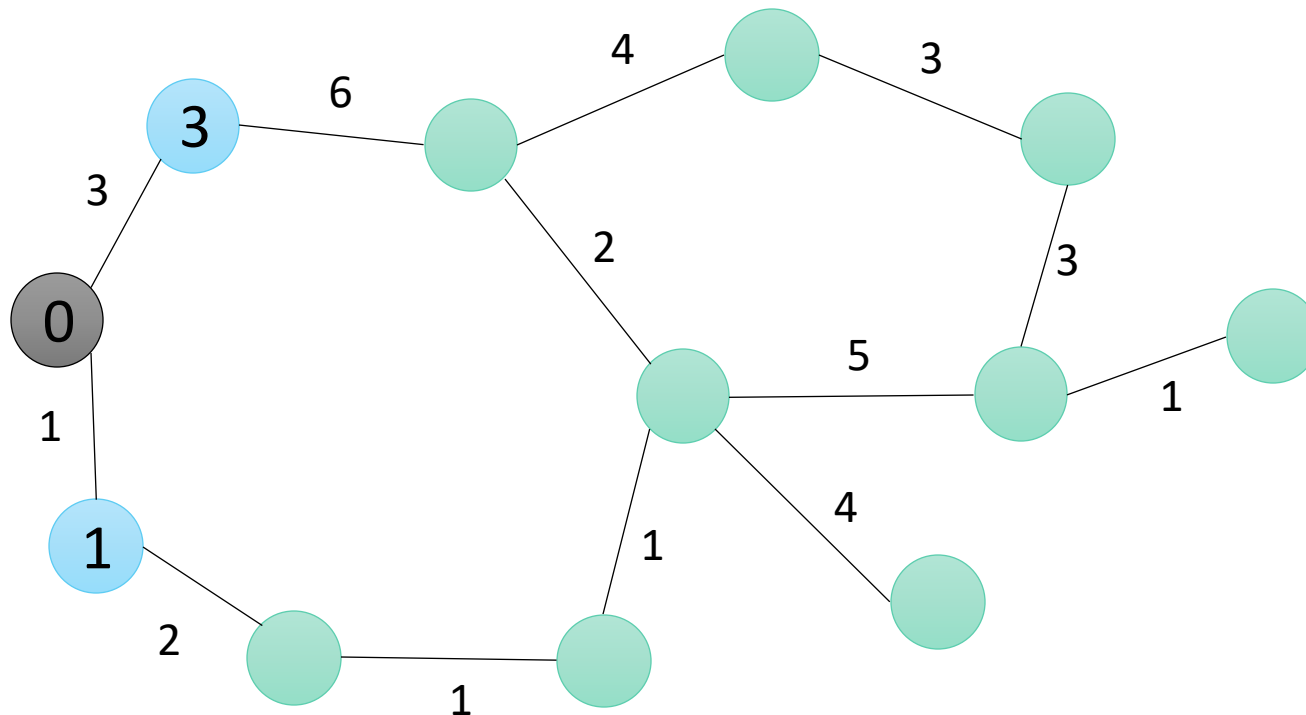


SSSP: *BFS on Weighted Graphs?*



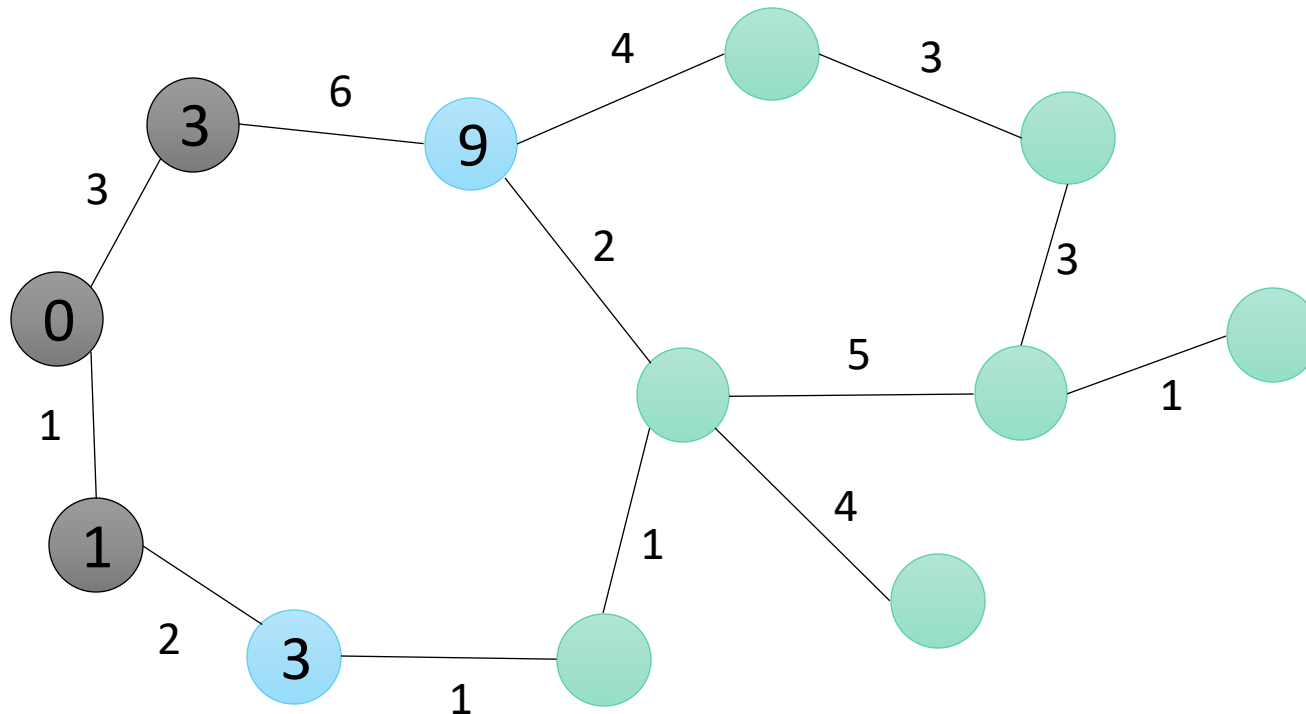


SSSP: BFS on Weighted Graphs?



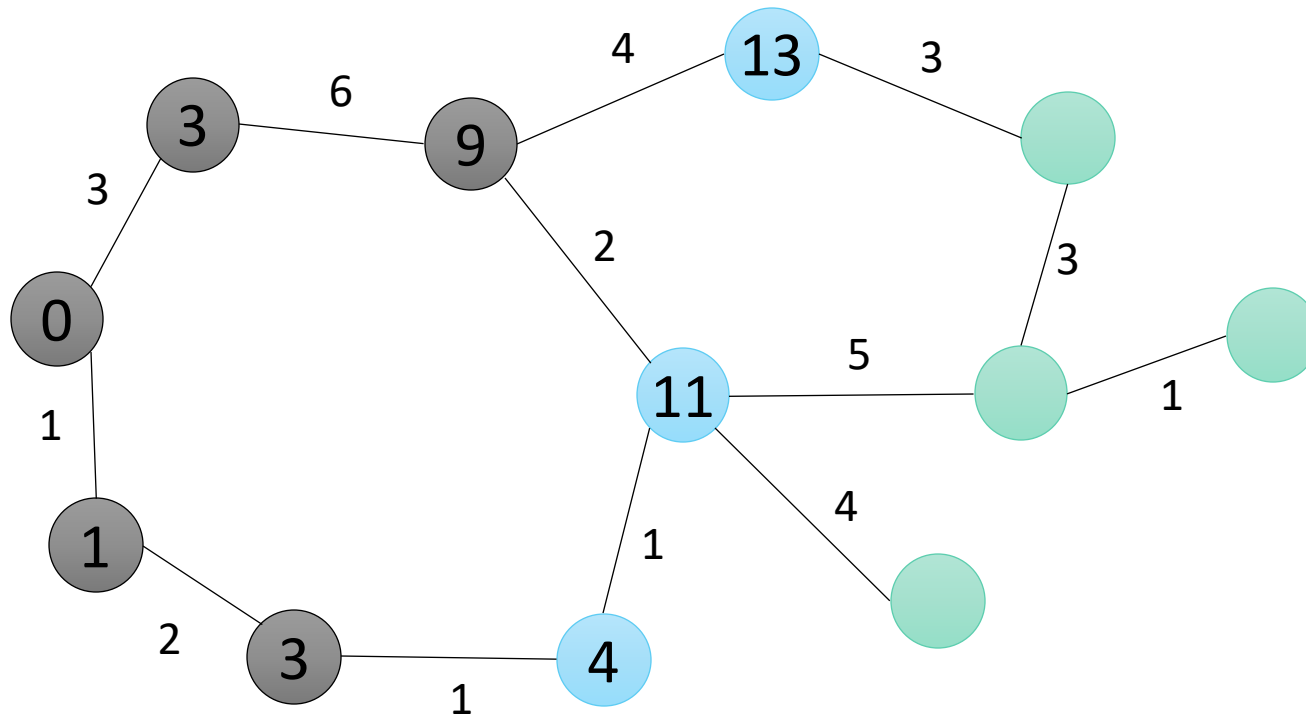


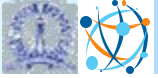
SSSP: BFS on Weighted Graphs?



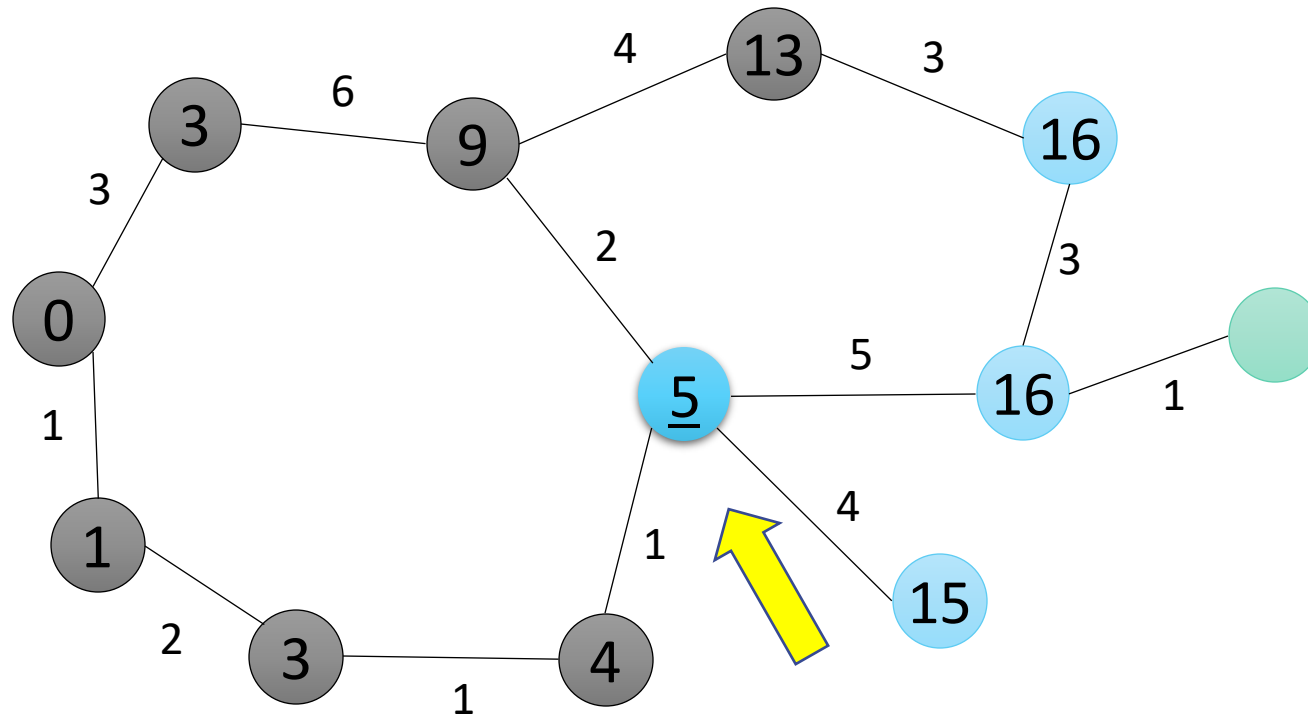


SSSP: BFS on Weighted Graphs?

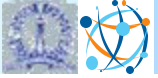




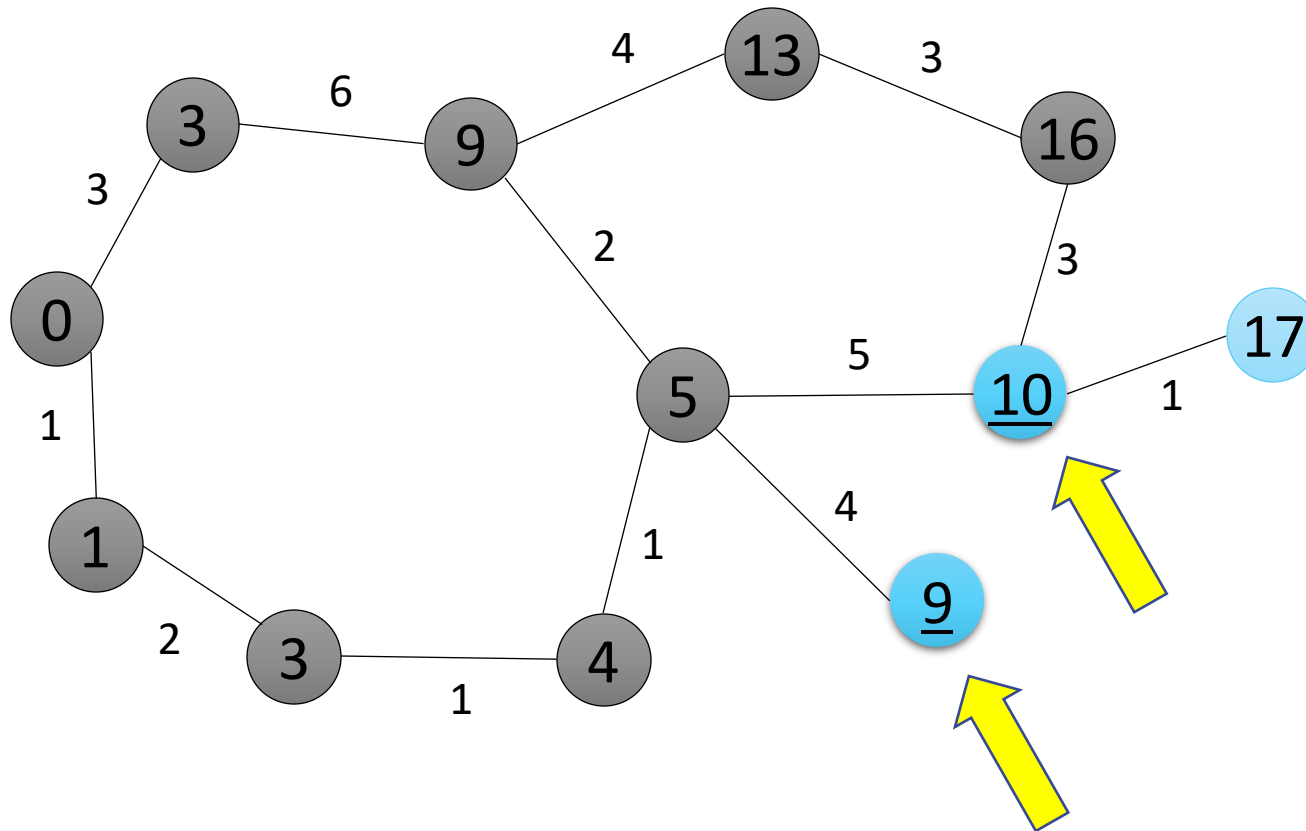
SSSP: BFS on Weighted Graphs?

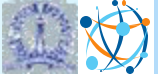


*Revisit, recalculate, re-propagate...
cascading effect*

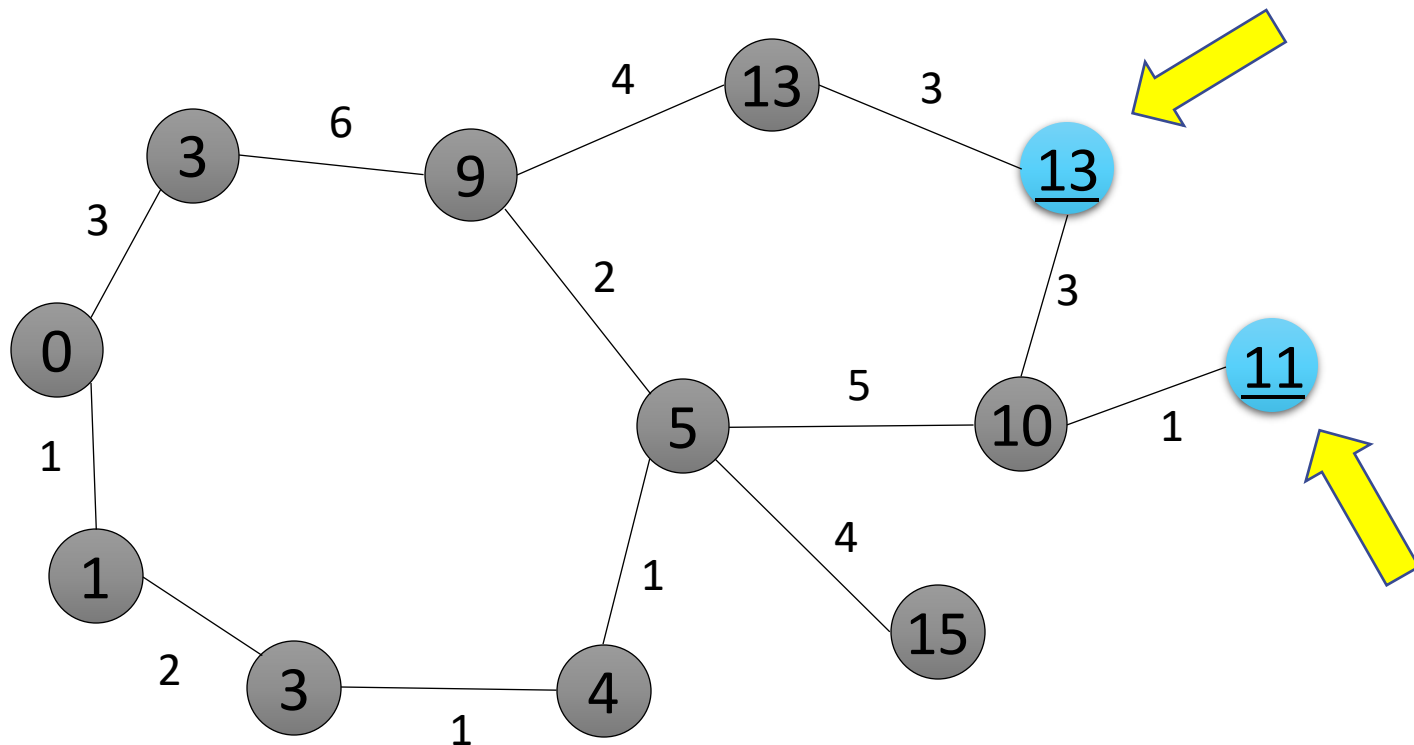


SSSP: BFS on Weighted Graphs?





SSSP: BFS on Weighted Graphs?



BFS with revisits is not efficient. Can we be smart about order of visits?