Graph Search Methods

Prof. Ki-Hoon Lee
Dept. of Computer Engineering
Kwangwoon University

Graphs

- G = (V, E)
- V is the vertex set.
- Vertices are also called nodes and points.
- E is the edge set.
- Each edge connects two *different* vertices.
- Edges are also called arcs and lines.
- Directed edge has an orientation <u, v>.

$$u \longrightarrow v$$

- <u, v> and <v, u> represent two different edges.

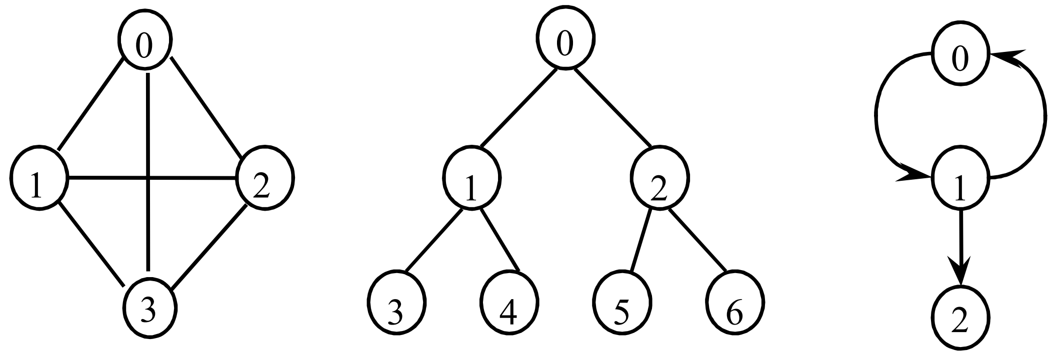
Graphs (cont.)

- Undirected edge has no orientation (u, v).

u —— v

- (u, v) and (v, u) represent the same edges.
- Undirected graph ➔ no oriented edge.
- Directed graph (digraph) ➔ every edge has an orientation.

Examples



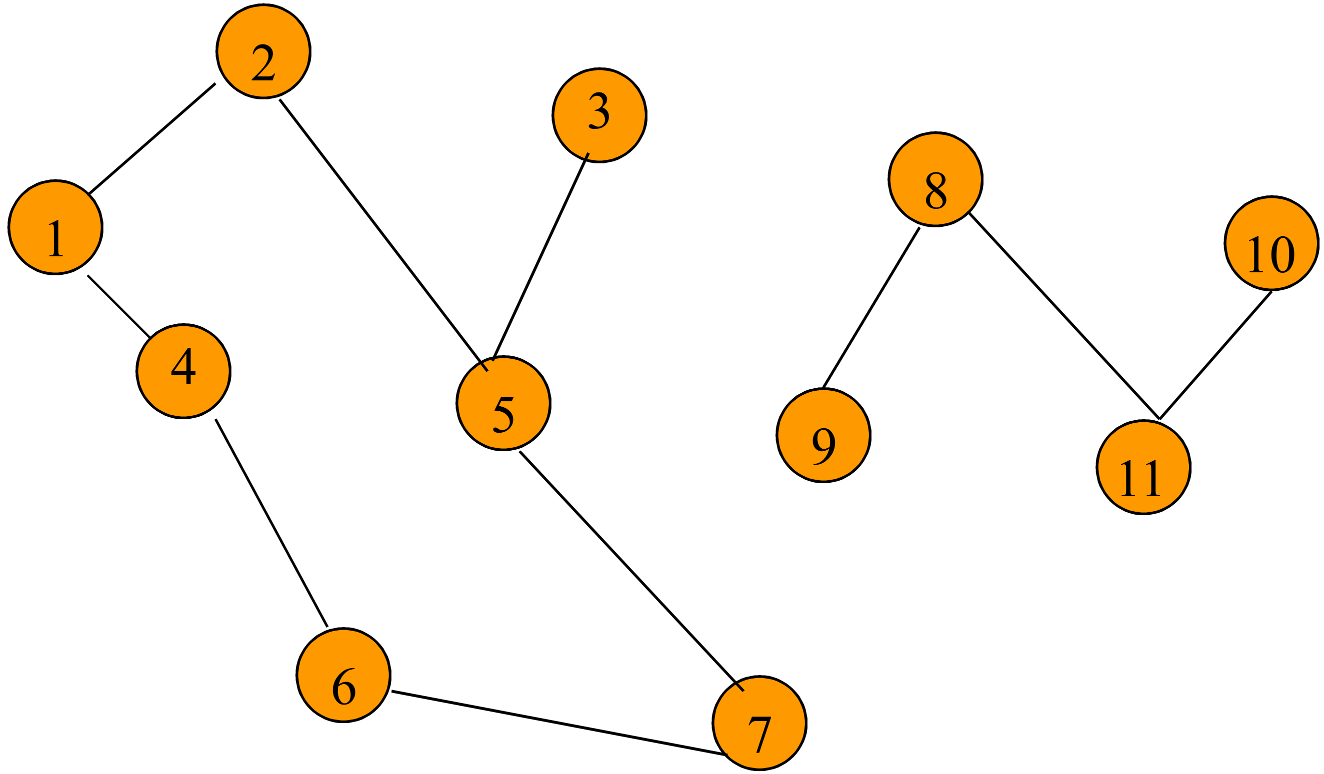
G_1 G_2 G_3

- $V(G_1) = \{0, 1, 2, 3\}$
  $E(G_1) = \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\}$

- $V(G_2) = \{0, 1, 2, 3, 4, 5, 6\}$
  $E(G_2) = \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\}$

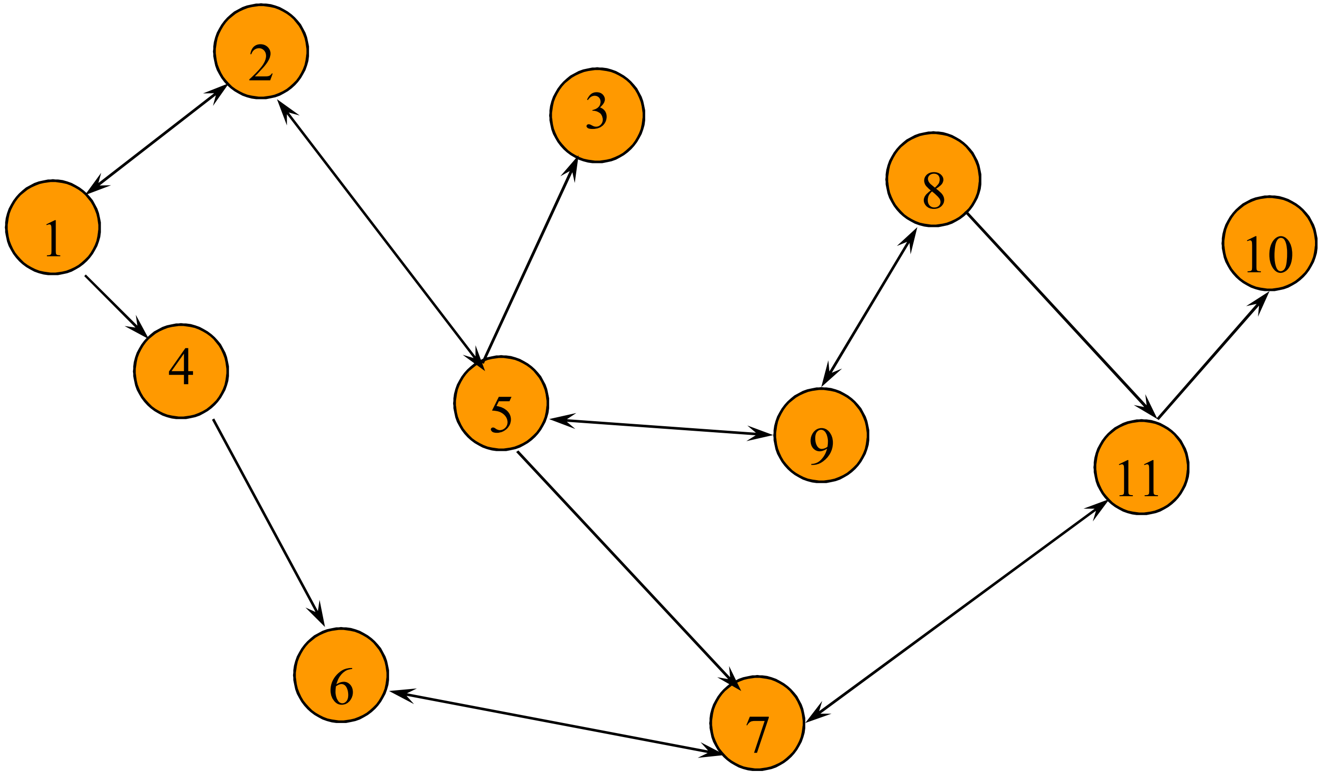
- $V(G_3) = \{0, 1, 2\}$
  $E(G_3) = \{<0, 1>, <1, 0>, <1, 2>\}$

Applications—Communication Network



- Vertex = city, edge = communication link.

Applications—Street Map



- Some streets are one way.

Number of Edges—Directed Graph

- Each edge is of the form <u, v>, u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge <u, v> is not the same as edge <v, u>, the number of edges in a complete directed graph is n(n-1).
- Number of edges in a directed graph is $\leq$ n(n-1).

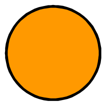
Number of Edges—Undirected Graph

- Each edge is of the form (u, v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u, v) is the same as edge (v, u), the number of edges in a complete undirected graph is n(n-1)/2.
- Number of edges in an undirected graph is $\leq$ n(n-1)/2.

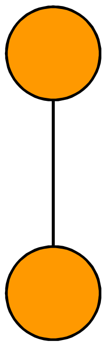
Complete Undirected Graph

Has all possible edges.

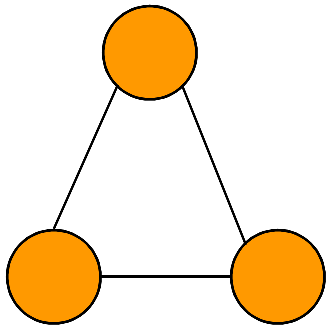
$n(n-1)/2$ ($= {}_nC_2$) edges for a graph with n vertices



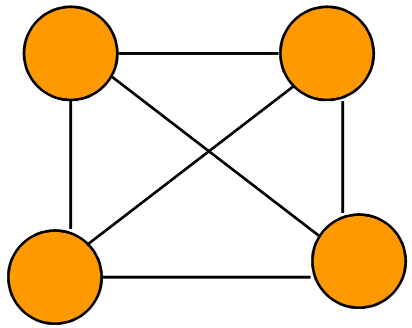
$n = 1$



$n = 2$



$n = 3$

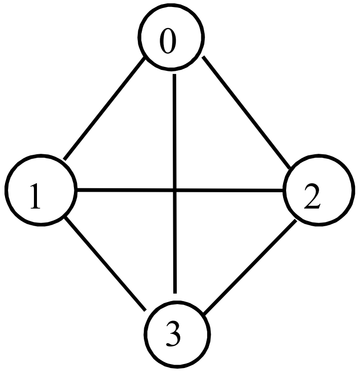


$n = 4$

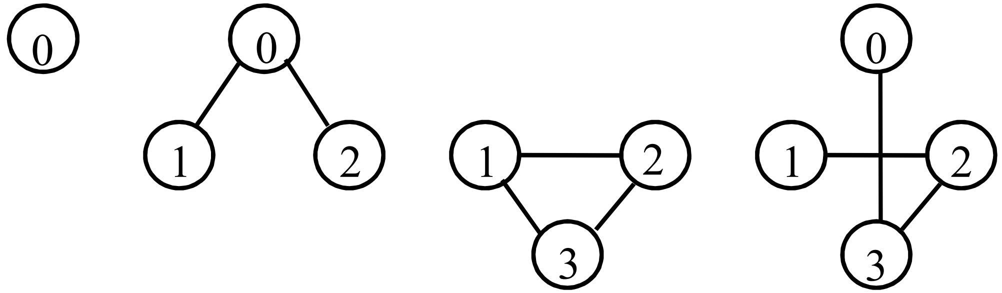
Definitions

- If (u, v) is an edge in E(G),
  - The vertices u and v are *adjacent*
  - The edge (u, v) is *incident* on vertices u and v
- If <u, v> is a directed edge,
  - Vertex u is *adjacent to* v, and v is *adjacent from* u
- A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$

Examples

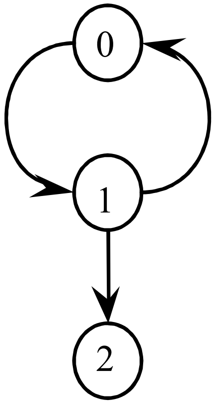


G_1

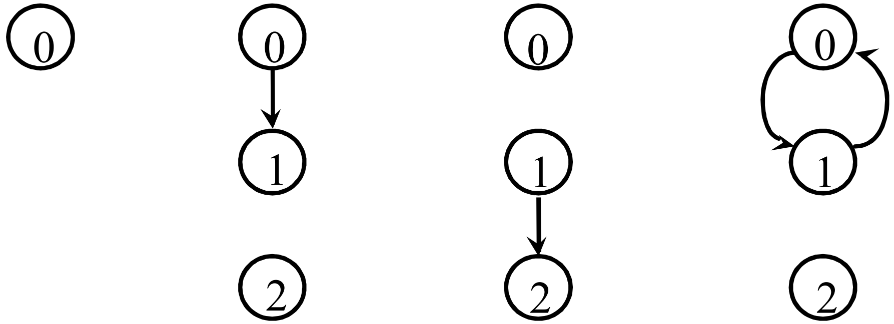


(i) (ii) (iii) (iv)

Some of the subgraphs of G_1



G_3

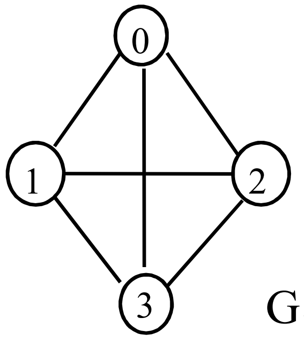


(i) (ii) (iii) (iv)

Some of the subgraphs of G_3

Definitions (cont.)

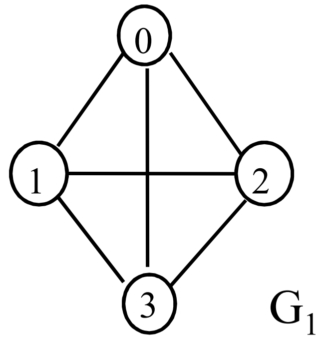
- A *path* from vertex u to vertex v in graph G is a sequence of vertices u, i_1, i_2, …, i_k, v such that (u, i_1), (i_1, i_2), …, (i_k, v) are edges in E(G)
- The *length* of a path is the number of edges on it
- A *simple path* is a path in which all vertices except possibly the first and last are distinct
  - Example: A path 0, 1, 3, 2 of G_1 is a simple path, but a path 0, 1, 3, 1 is not.



G_1

Definitions (cont.)

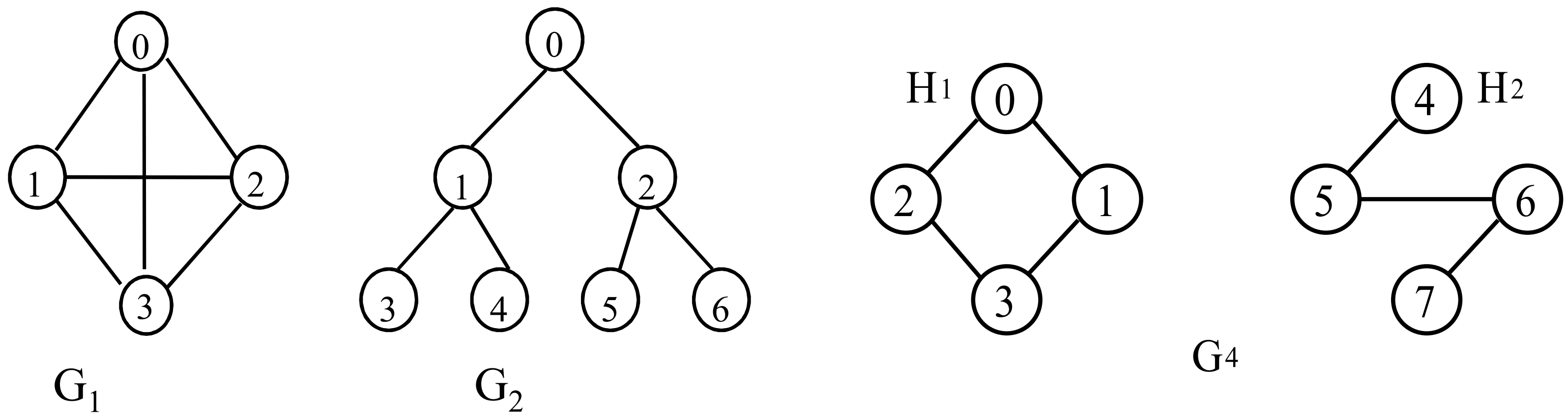
- A *cycle* is a simple path in which the first and last vertices are the same
  - A path 0, 1, 2, 0 is a cycle in G_1



- In an undirected graph, G, two vertices u and v are *connected* iff there is a path in G from u to v (or from v to u)

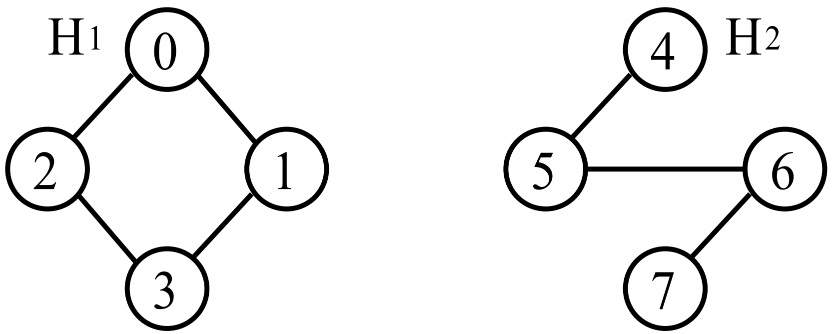
Definitions (cont.)

- An undirected graph *G* is *connected* iff for every pair of distinct vertices u and v in V(G), there is a path from u to v in *G*
  - Example: G_1 and G_2 are connected, whereas G_4 is not



Connect Components

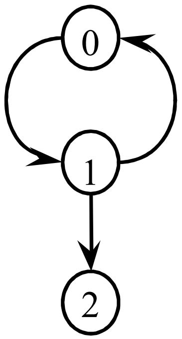
- A *connected component* (or simply a component), H, of an undirected graph is a *maximal* connected subgraph
  - By maximal, we mean that G contains no other subgraph that is both connected and properly contains H
  - G_4 has two components, H_1 and H_2



G4

Strongly Connect Components

- A directed graph G is *strongly connected* iff for every pair of distinct vertices u and v in V(G), there is a directed path from u to v and also from v to u
  - Example: G_3 is not strongly connected, as there is no path from vertex 2 to 1

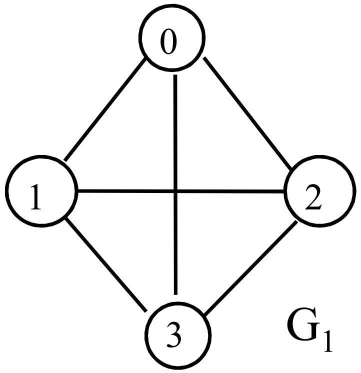


G_3

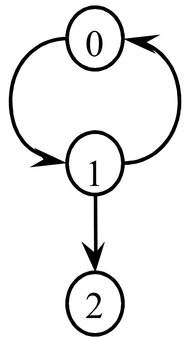
- A *strongly connected component* is a maximal subgraph that is strongly connected
  - Example: G_3 has two strongly connected components

Definitions (cont.)

- A *tree* is a graph that is connected and acyclic (i.e., has no cycles)
- The *degree* of a vertex is the number of edges incident to that vertex
  - Example: The degree of vertex 0 in G_1 is 3
- For a directed graph, the *in-degree* (*out-degree*) of a vertex *v* is the number of edges that have *v* as the head (tail)
  - Example: In G_3, Vertex 1: in-degree 1, out-degree 2, degree 3

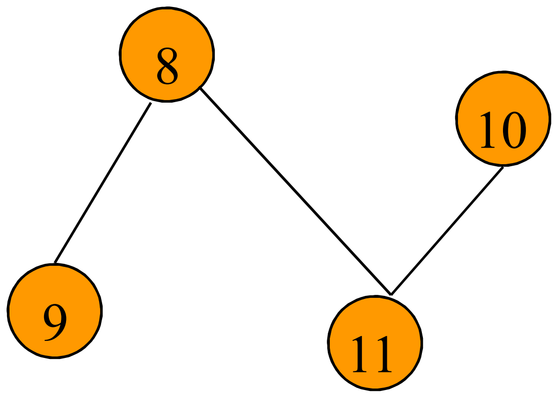


G_1



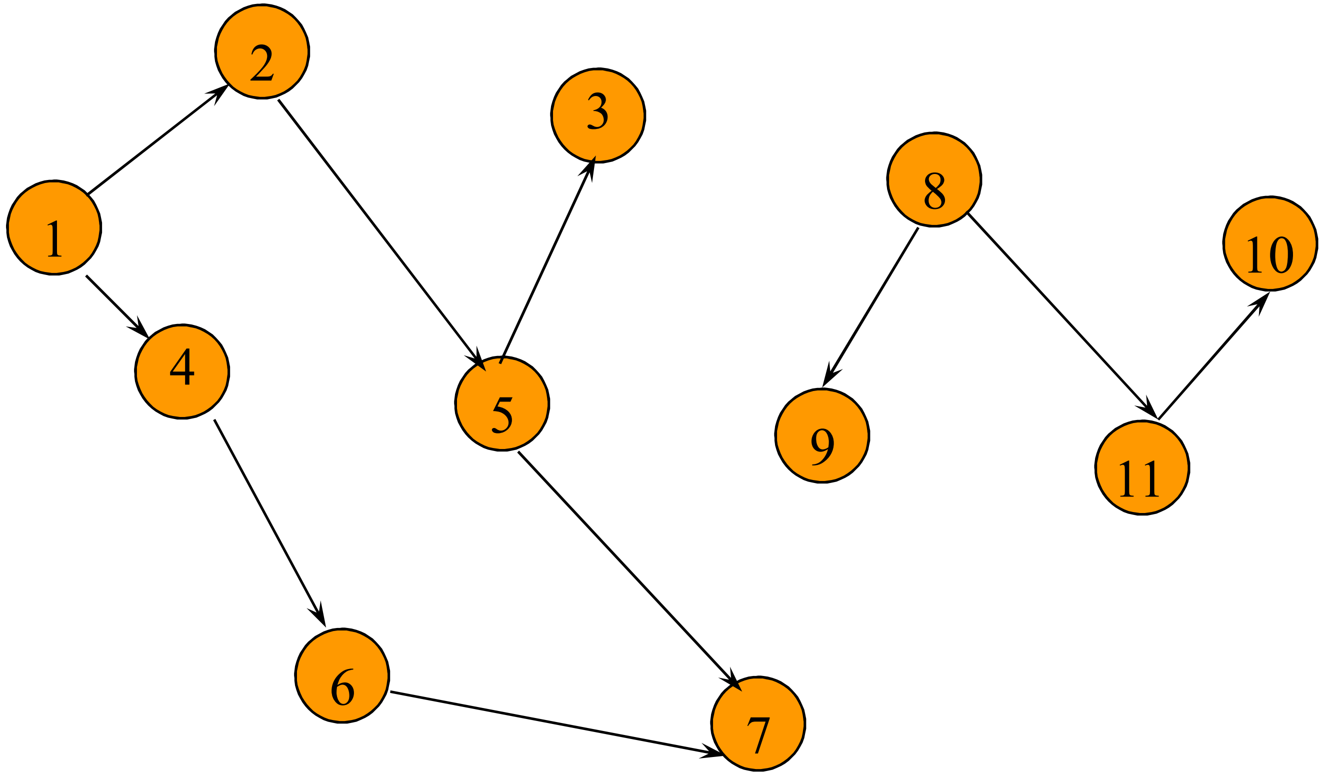
G_3

Sum of Vertex Degrees



Sum of degrees = 2e (e is number of edges)

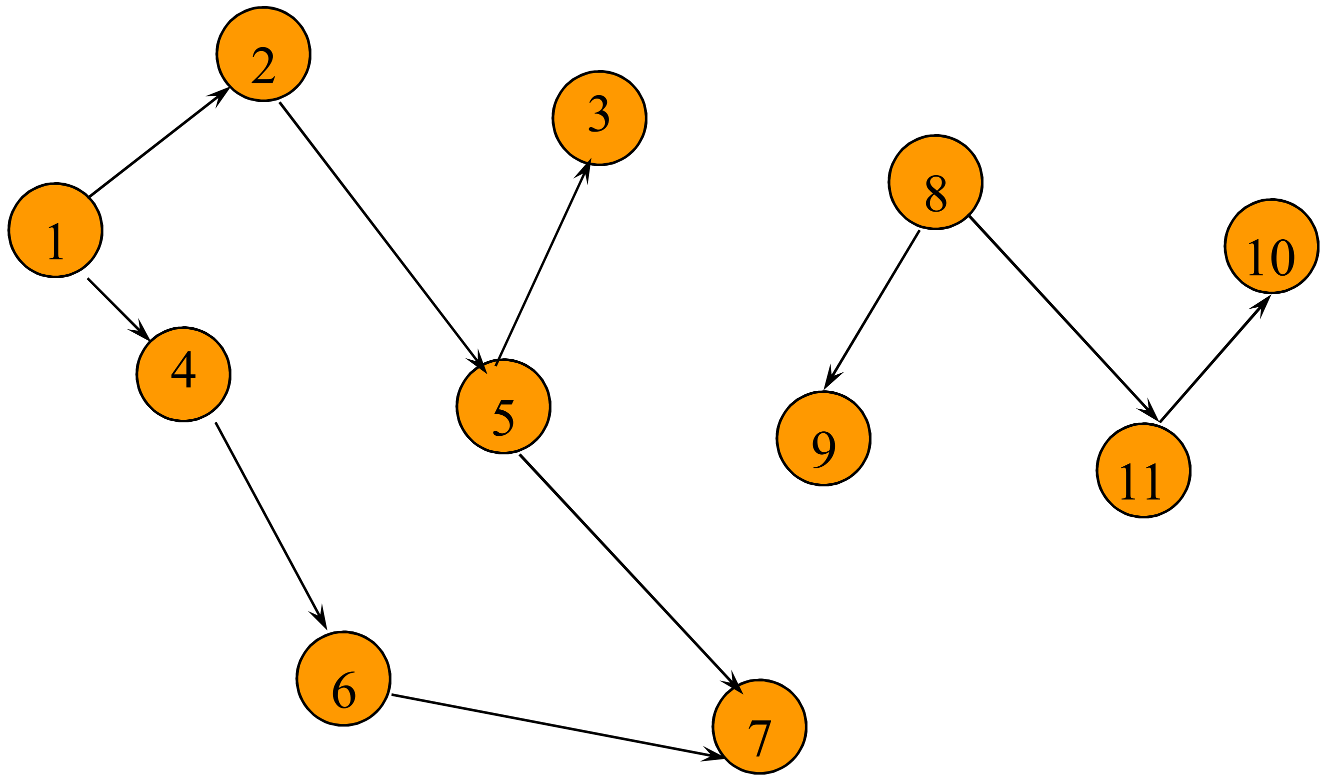
In-Degree of a Vertex



in-degree is number of incoming edges

in-degree(2) = 1, in-degree(8) = 0

Out-Degree of a Vertex



out-degree is number of outgoing edges

out-degree(2) = 1, out-degree(8) = 2

Sum of In- and Out-Degrees

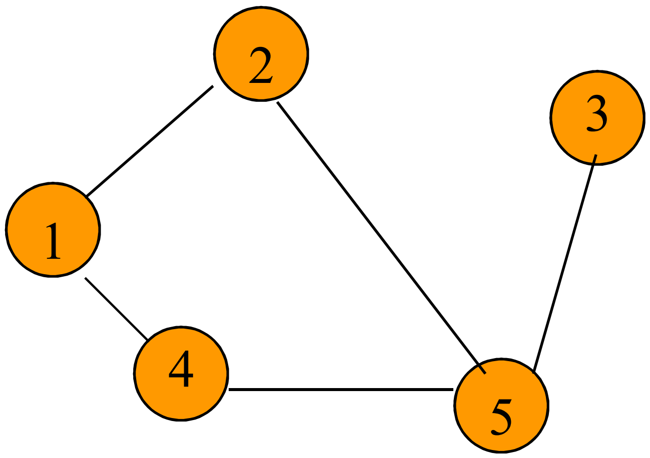
- Each edge contributes 1 to the in-degree of some vertex and 1 to the out-degree of some other vertex
- sum of in-degrees = sum of out-degrees = the number of edges in the digraph

Graph Representation

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

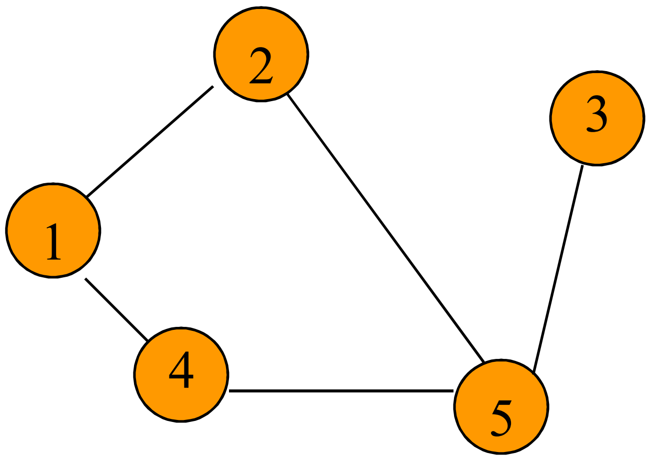
Adjacency Matrix

- $n \times n$ matrix A, where n = # of vertices in G
- A[i][j] = 1 iff (i, j) is an edge in E(G)
- A[i][j] = 0 iff (i, j) is not an edge in E(G)
- (i, j) is changed to <i, j> for a digraph



	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

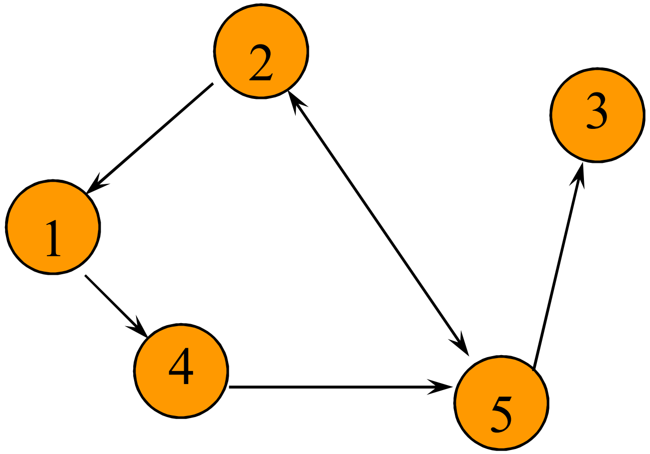
Adjacency Matrix Properties (Undirected Graph)



	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.
- The degree of any vertex i is its row sum

Adjacency Matrix Properties (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 digraph is asymmetric.
- The row sum is the out-degree and the column sum is the in-degree

Adjacency Matrix

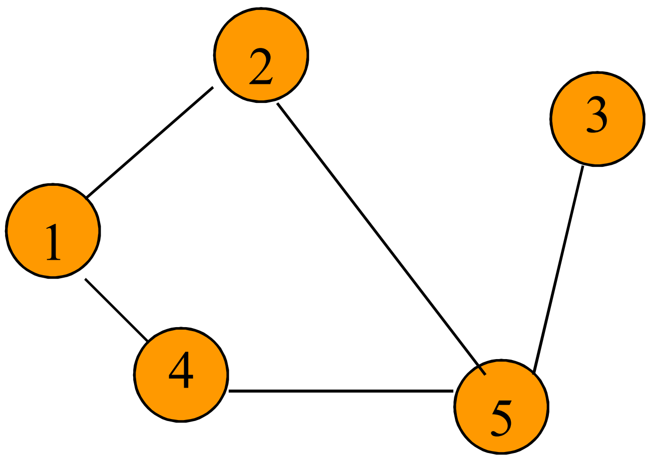
- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ bits
- $O(n)$ time to find vertex degree and/or vertices adjacent to a given vertex.
- $O(n^2)$ time to answer a question about graphs
  - Example: How many edges are there in G?

Adjacency Matrix

- In a *sparse graph*, most of the terms in the adjacency matrix are zero
- Counting the number of edges could be answered in $O(n + e)$ time, where e is the number of edges in G, and $e << n^2$ for sparse graphs
- Such a speed-up can be made possible through the use of a representation in which only the edges that are in G are explicitly stored
- This leads to the next representation for graphs, *adjacency lists*

Adjacency Lists

- Adjacency list for vertex i is a linear list of vertices adjacent from vertex i
  - # of nodes in the adjacency list: the degree of i for an undirected graph or the out-degree of i for digraph
- An array of n adjacency lists



aList[1] = (2,4)

aList[2] = (1,5)

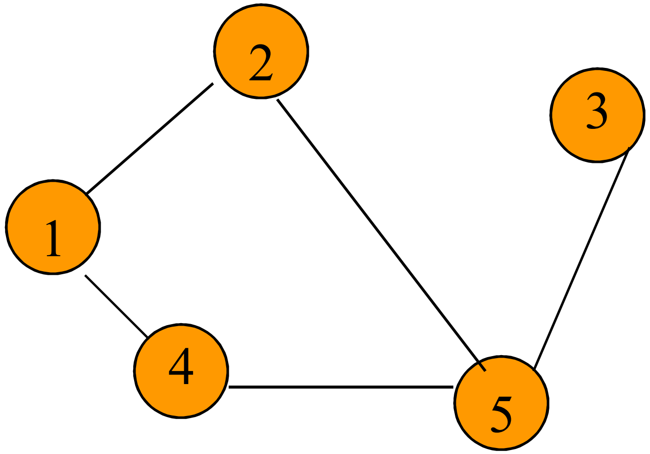
aList[3] = (5)

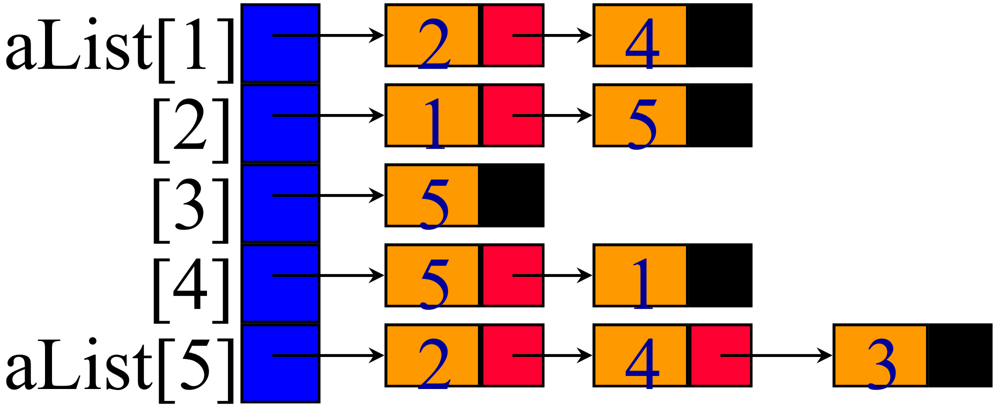
aList[4] = (5,1)

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

Linked Adjacency Lists

- Each adjacency list is a chain





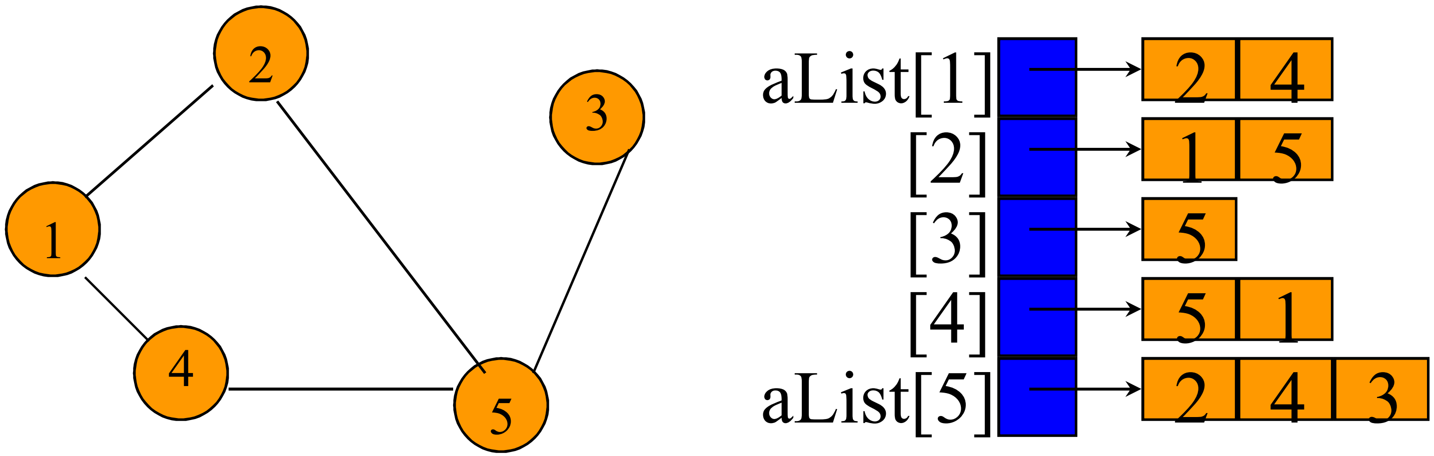
Array Length = n

# 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)

Weighted Graphs

- Weighted adjacency matrix
  - W(i,j) = weight of edge (i,j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)
- A graph with weighted edges is called a *network*

Summary

- Graph representations
  - Adjacency Matrix
  - Adjacency Lists
    - Linked Adjacency Lists
    - Array Adjacency Lists
  - 3 representations

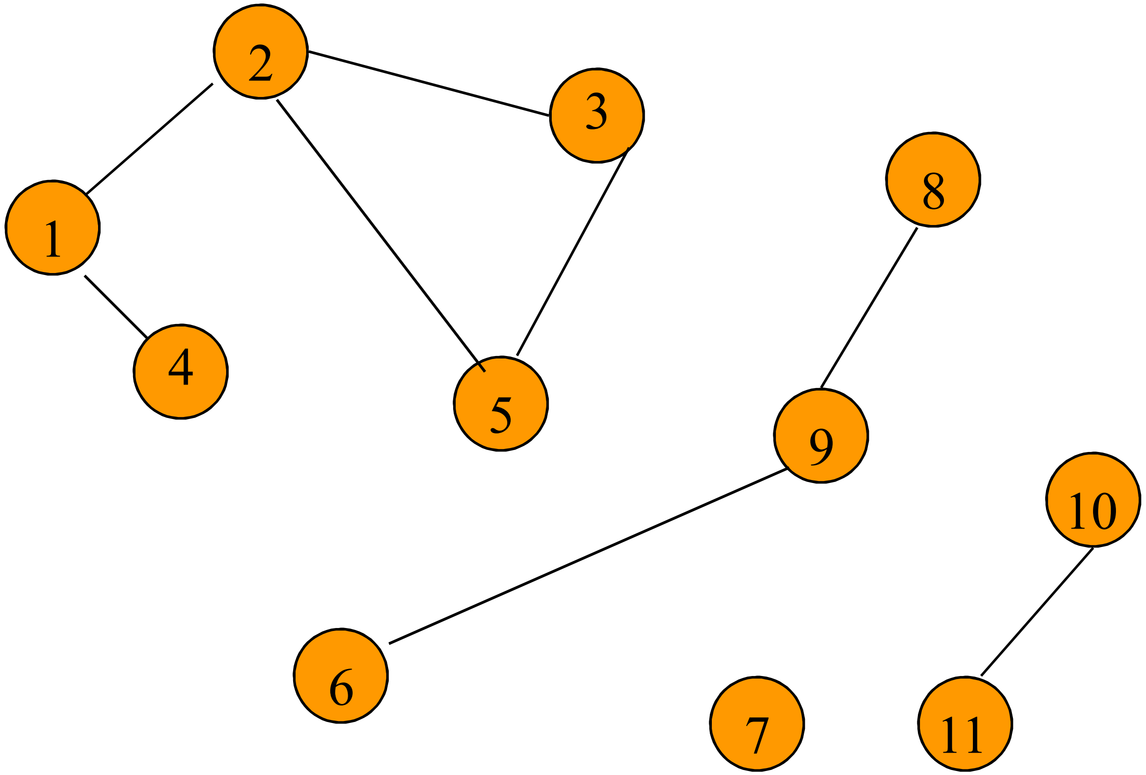
- Graph types
  - Directed and undirected
  - Weighted and unweighted
  - 2 x 2 = 4 graph types

Graph Search Methods

- DFS
- BFS

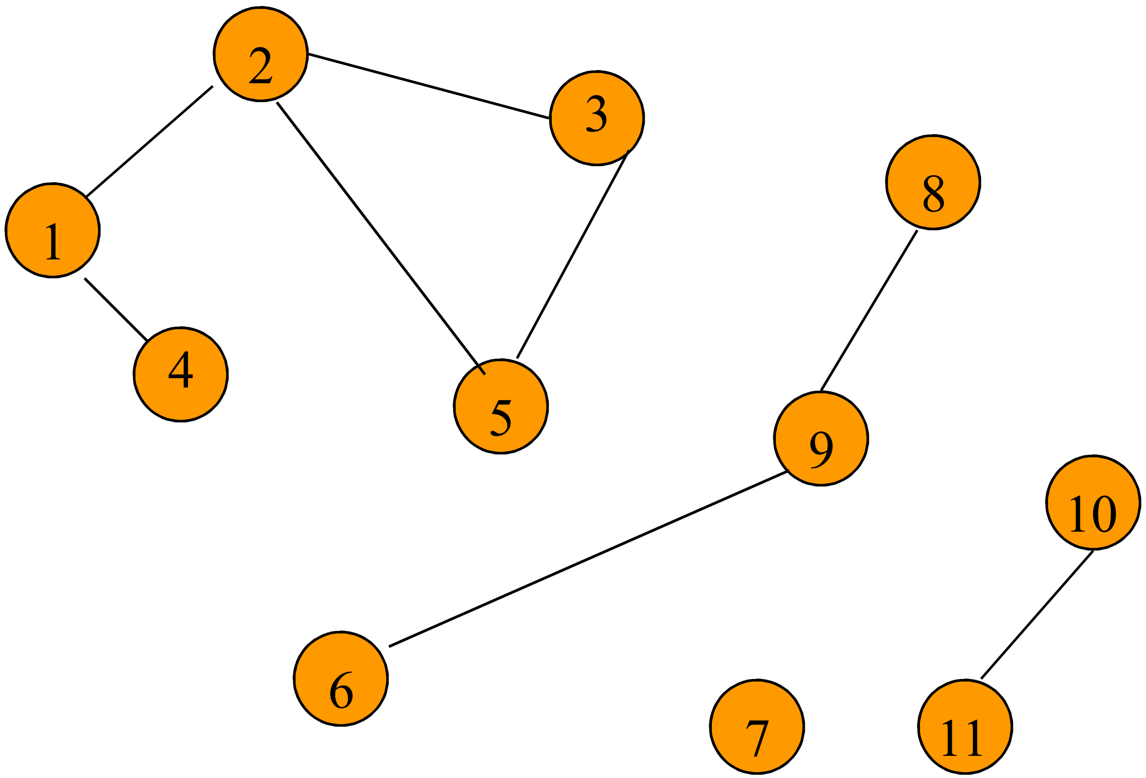
Graph Search Methods

- A vertex u is reachable from vertex v iff there is a path from v to u



Graph Search Methods

- A search method starts at a given vertex v and visits/labels/marks every vertex that is reachable from v



Graph Search Methods

- Many graph problems solved using a search method
  - Path from one vertex to another
  - Is the graph connected?
  - Find a spanning tree
  - Etc.
- Commonly used search methods: depth-first search and breadth-first search
- There are problems for which BFS is better than DFS and vice versa

Depth-First Search (DFS)

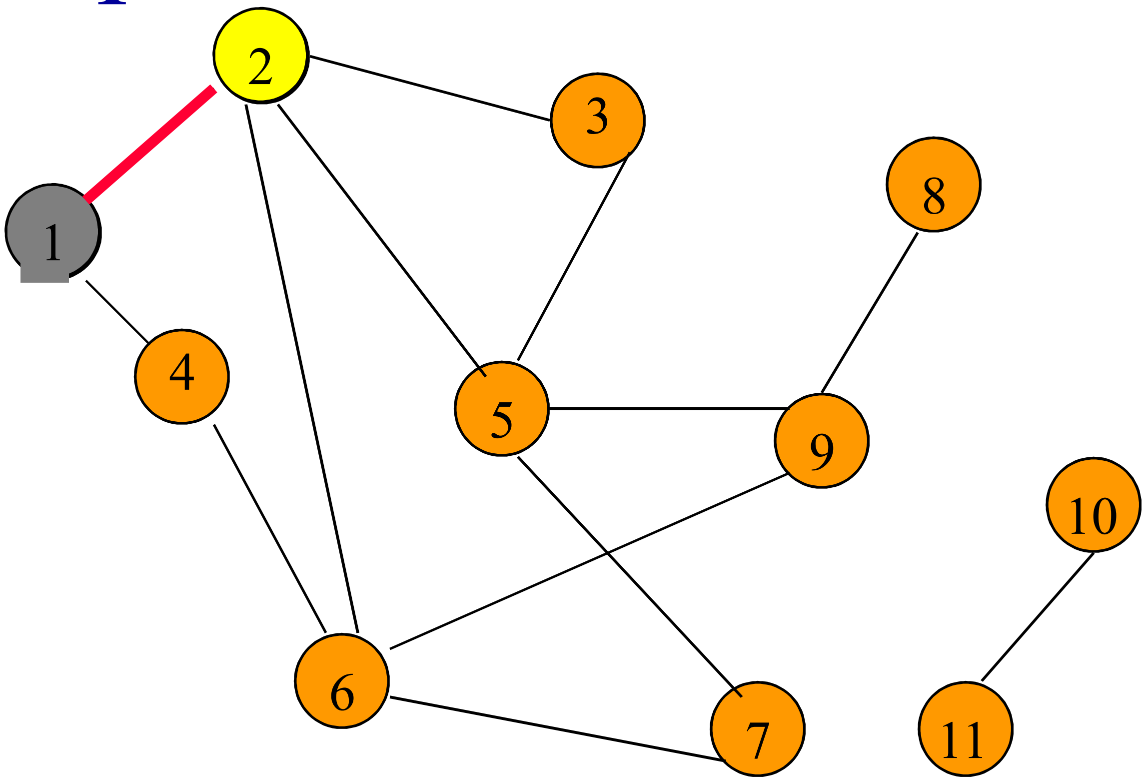
- A recursive graph searching technique
- While doing a DFS, we maintain a set of visited nodes (Initially this set is empty)
- When DFS is called on any vertex (say v), first that vertex is marked as visited and then for every edge going out of that vertex, (v, w), such that w is unvisited, we call DFS on w
- Finally, we return when we have exhausted all the edges going out from v

Depth-First Search

```
virtual void Graph::DFS()  // Driver
{
  visited = new bool [n];
    // visited is declared as a bool* data member of Graph
  fill(visited,visited+n,false);
  DFS(0); // start search at vertex 0
  delete [] visited;
}

virtual void Graph::DFS(const int v) // Workhorse
{// Visit all previously unvisited vertices that are reachable from vertex v.
  visited[v] = true;
  for (each vertex w adjacent to v) // actual code uses an iterator
    if (!visited[w]) DFS(w);
}
```

Depth-First Search Example



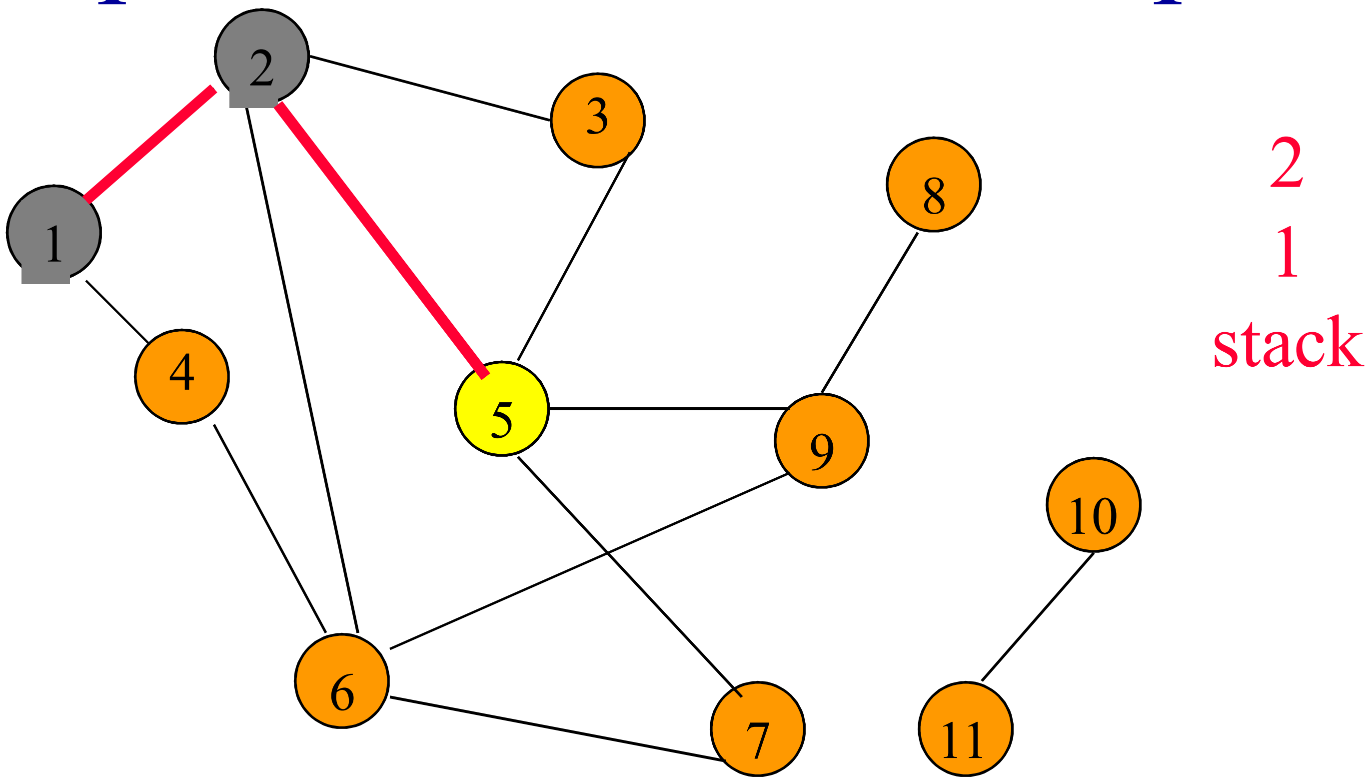
1
stack

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.

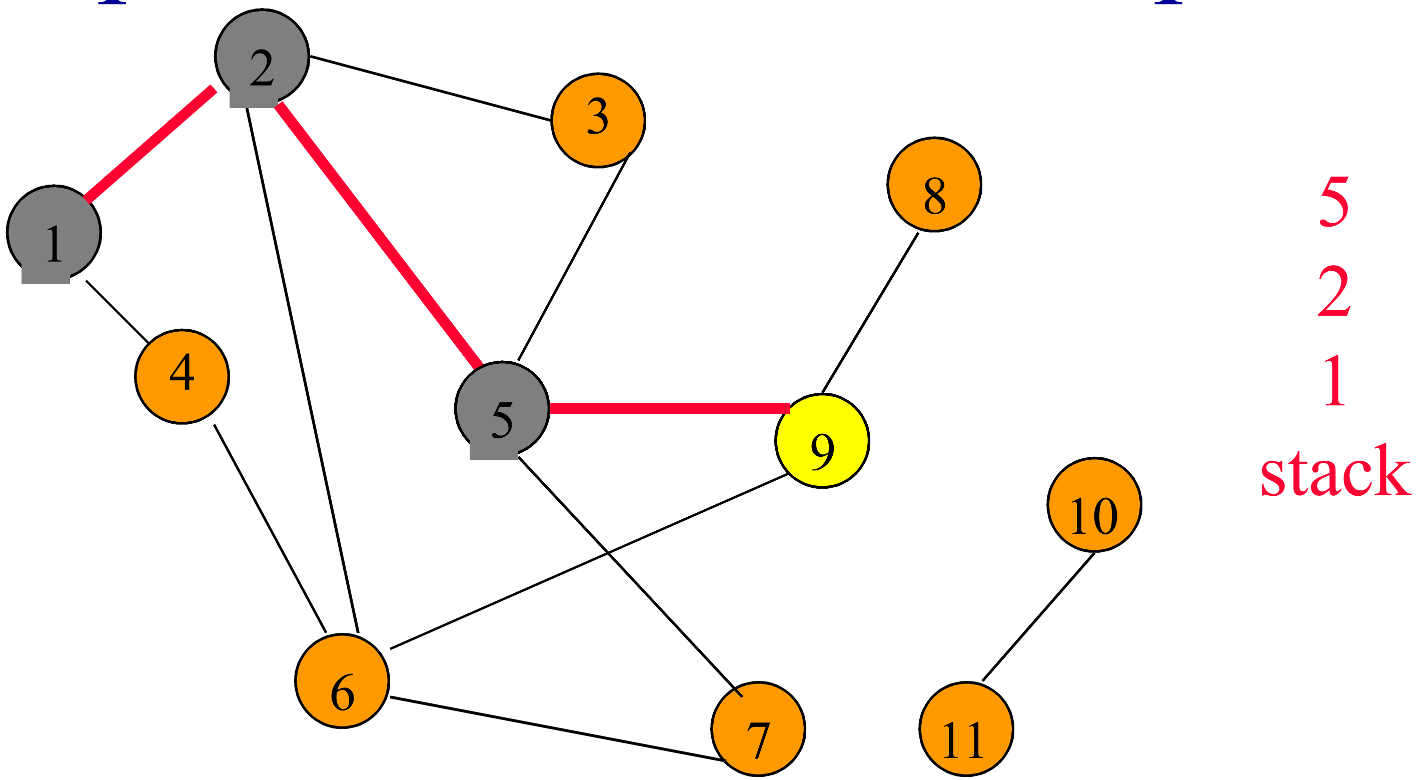
Depth-First Search Example



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

Suppose that vertex 5 is selected.

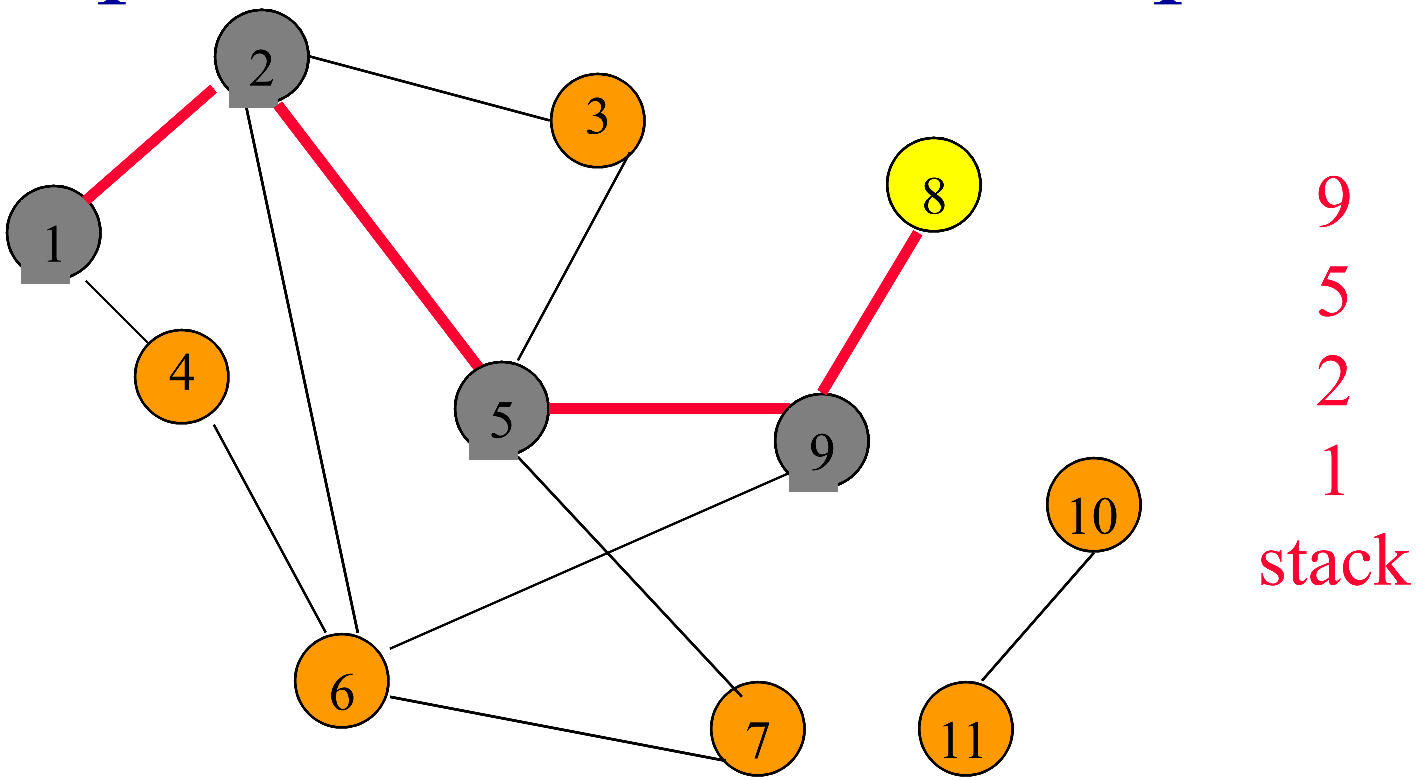
Depth-First Search Example



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

Suppose that vertex 9 is selected.

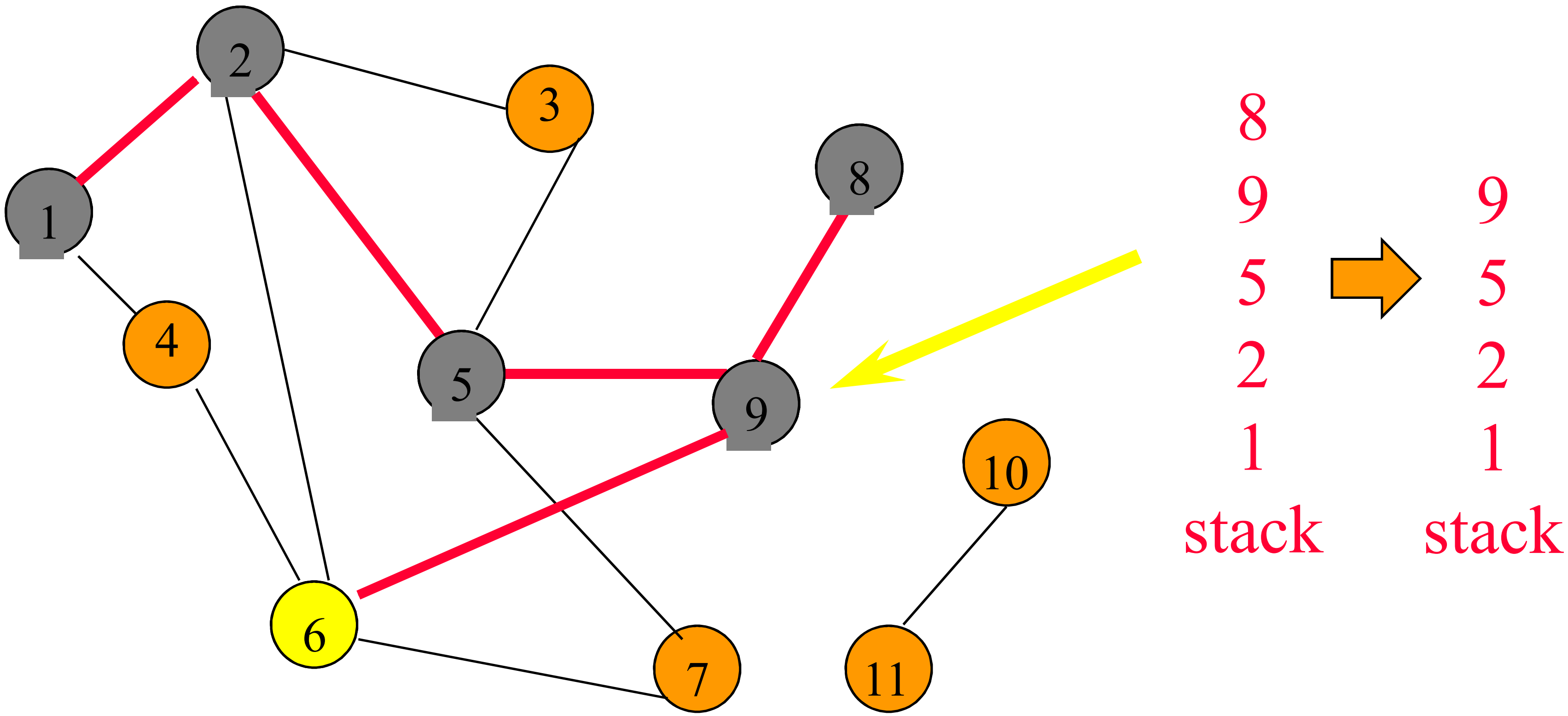
Depth-First Search Example



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

Suppose that vertex 8 is selected.

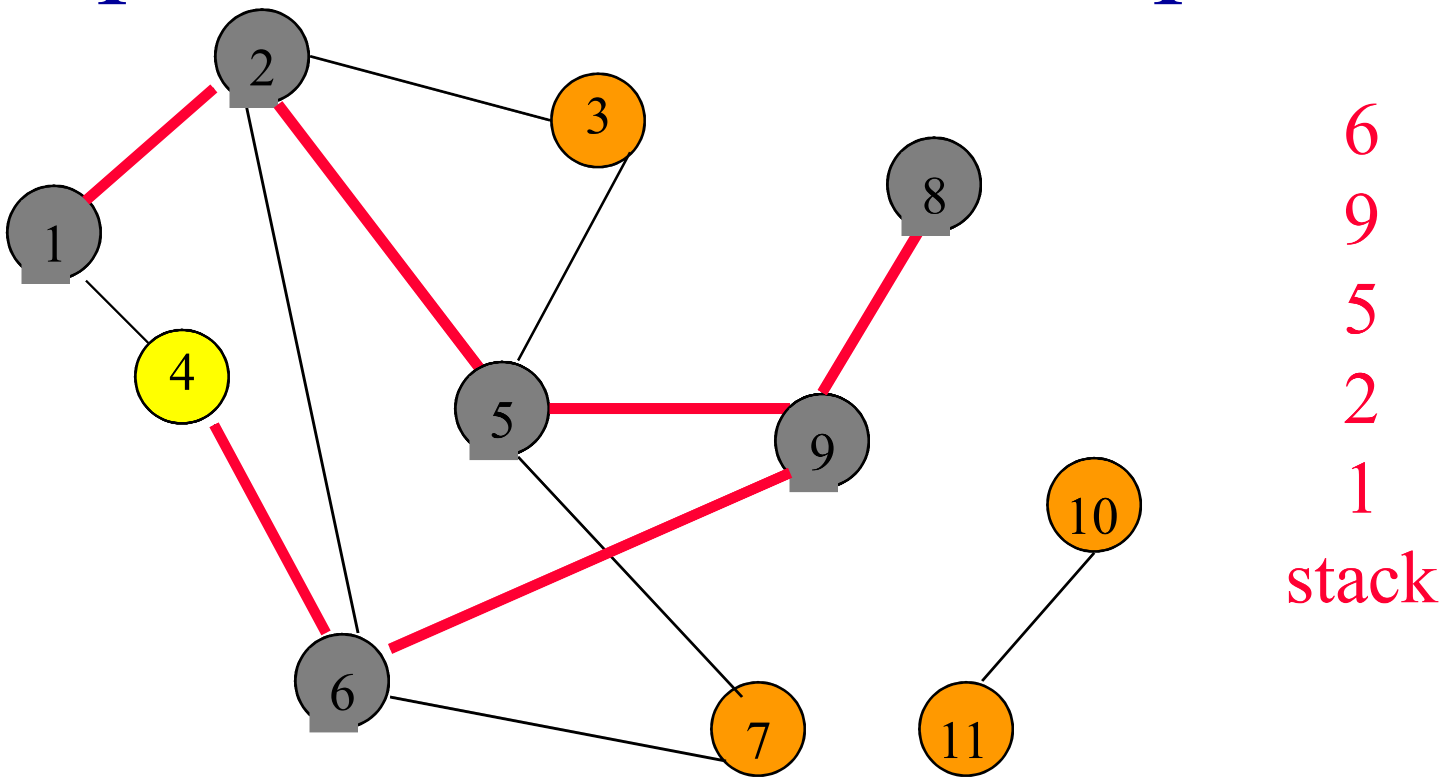
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a DFS(6).

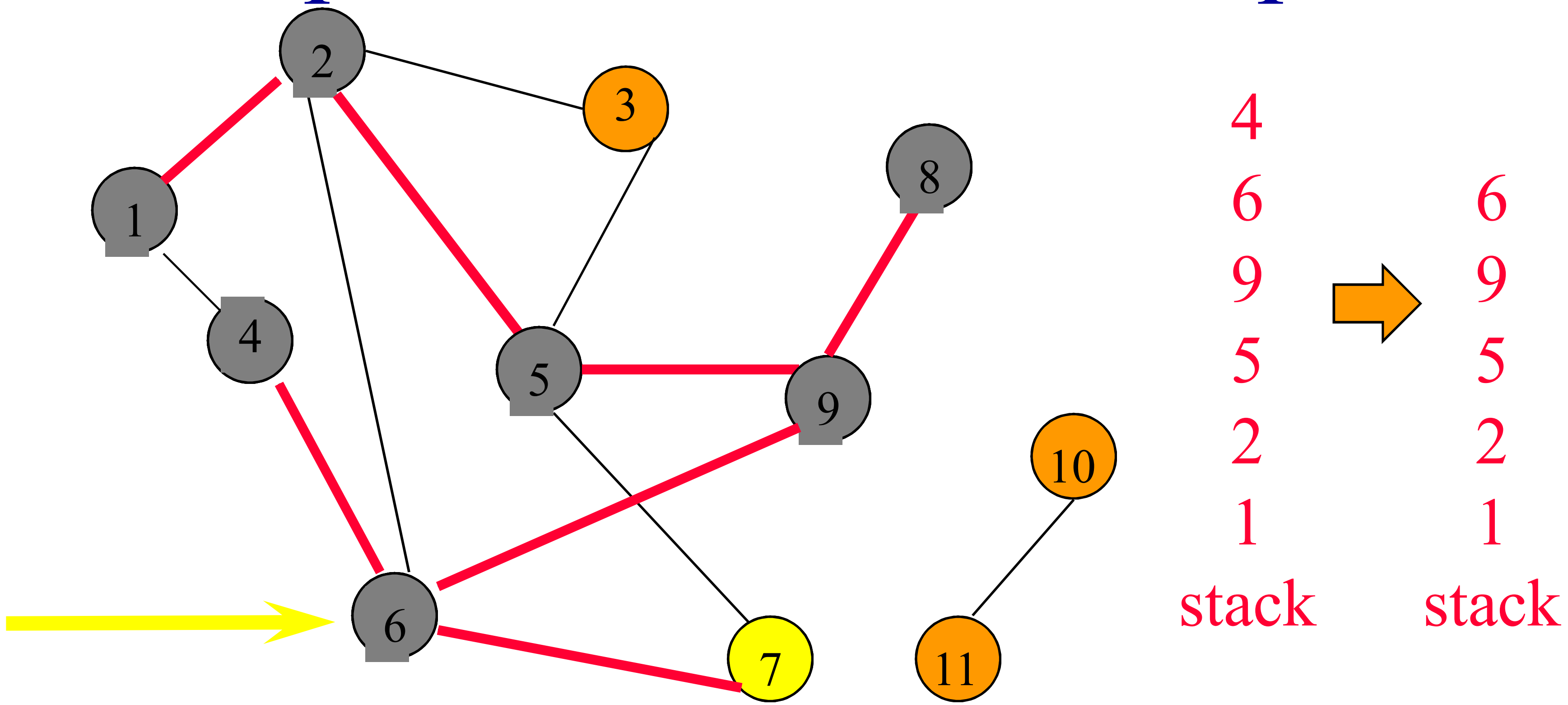
Depth-First Search Example



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

Suppose that vertex 4 is selected.

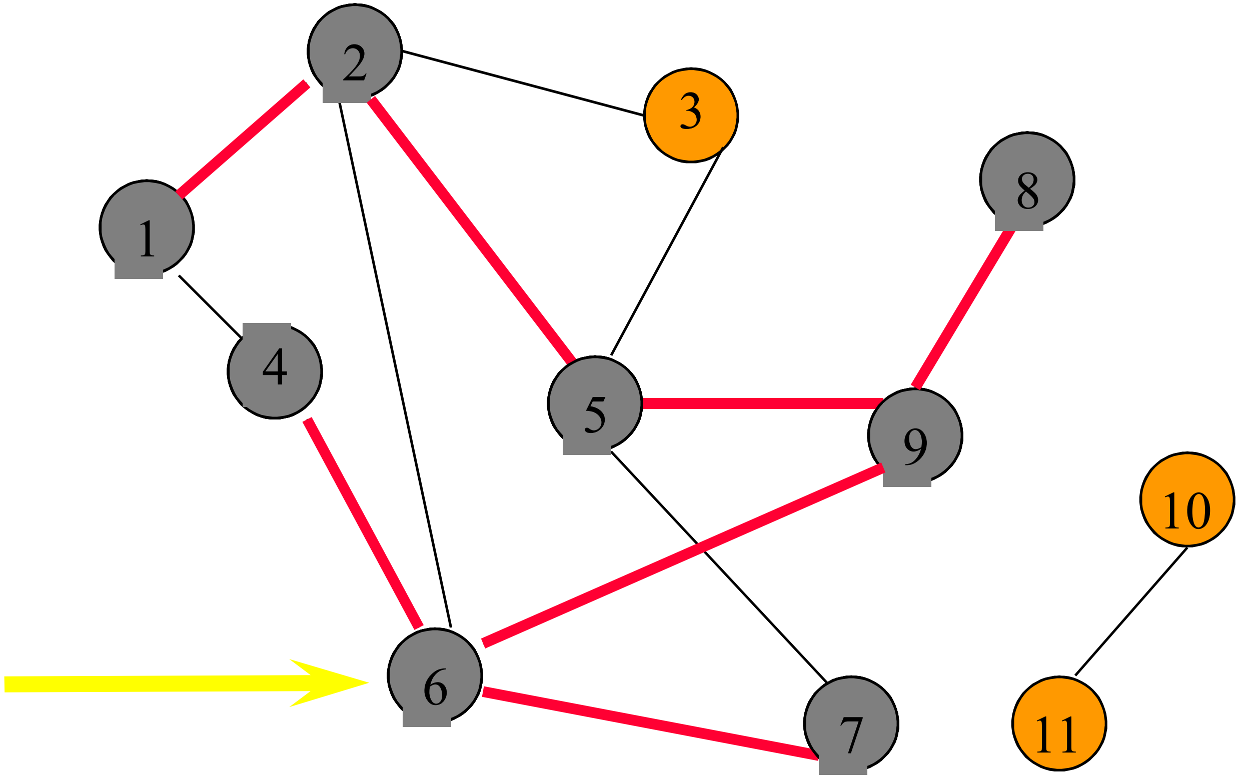
Depth-First Search Example

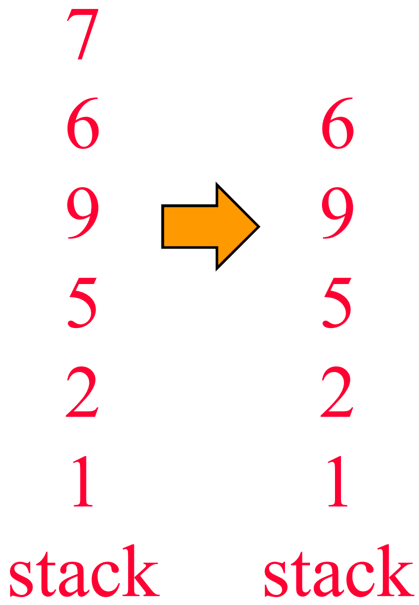


Label vertex 4 and return to 6.

From vertex 6 do a DFS(7).

Depth-First Search Example

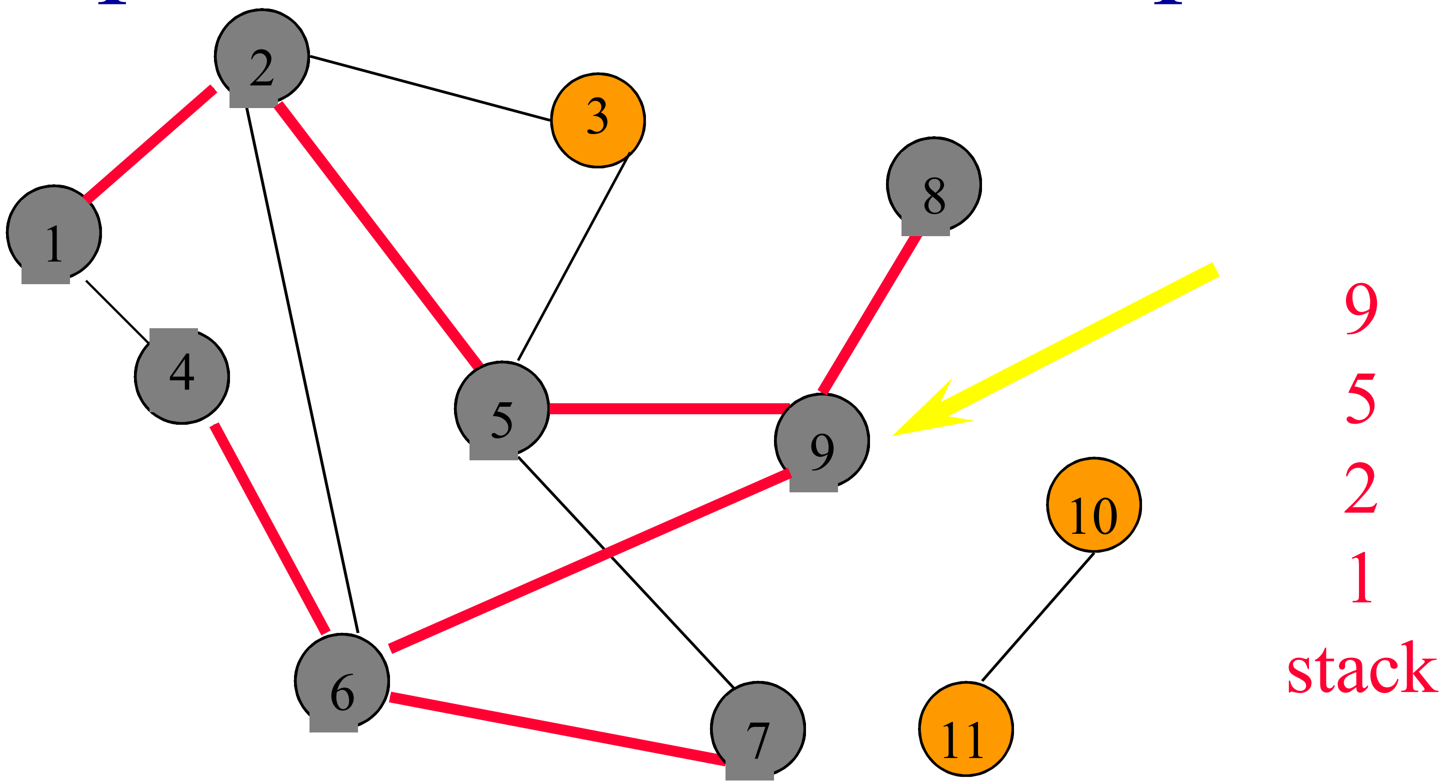




Label vertex 7 and return to 6.

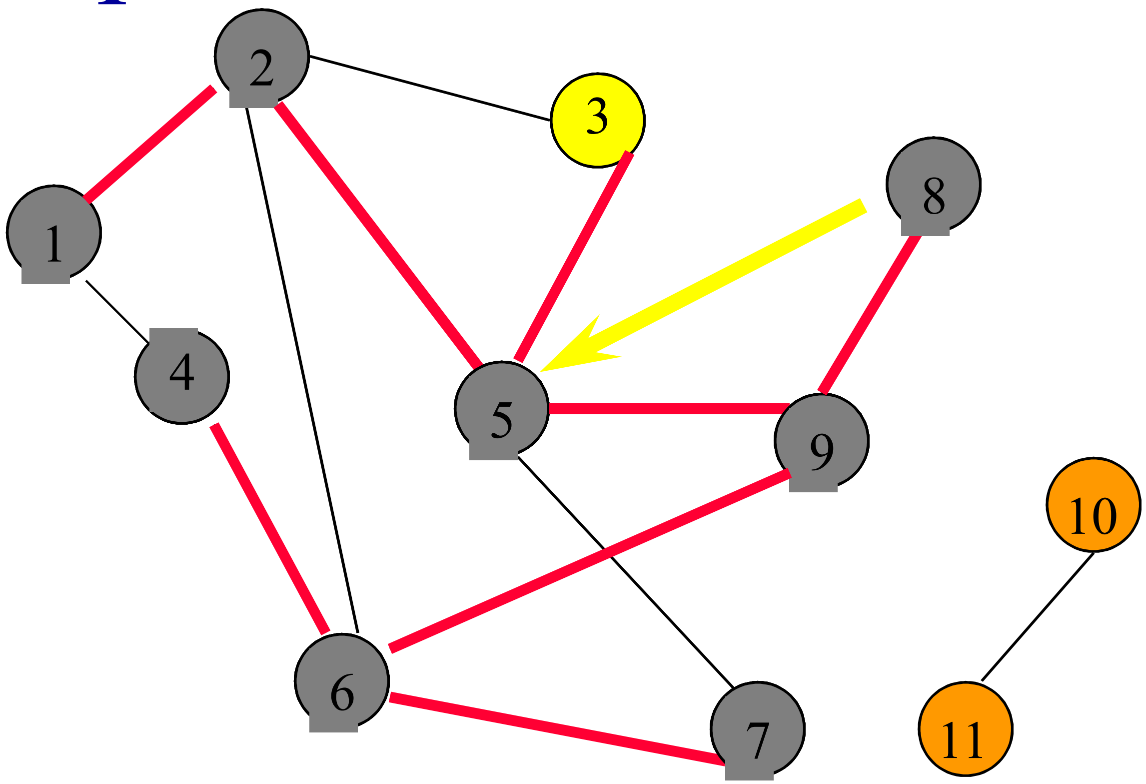
Return to 9.

Depth-First Search Example



Return to 5.

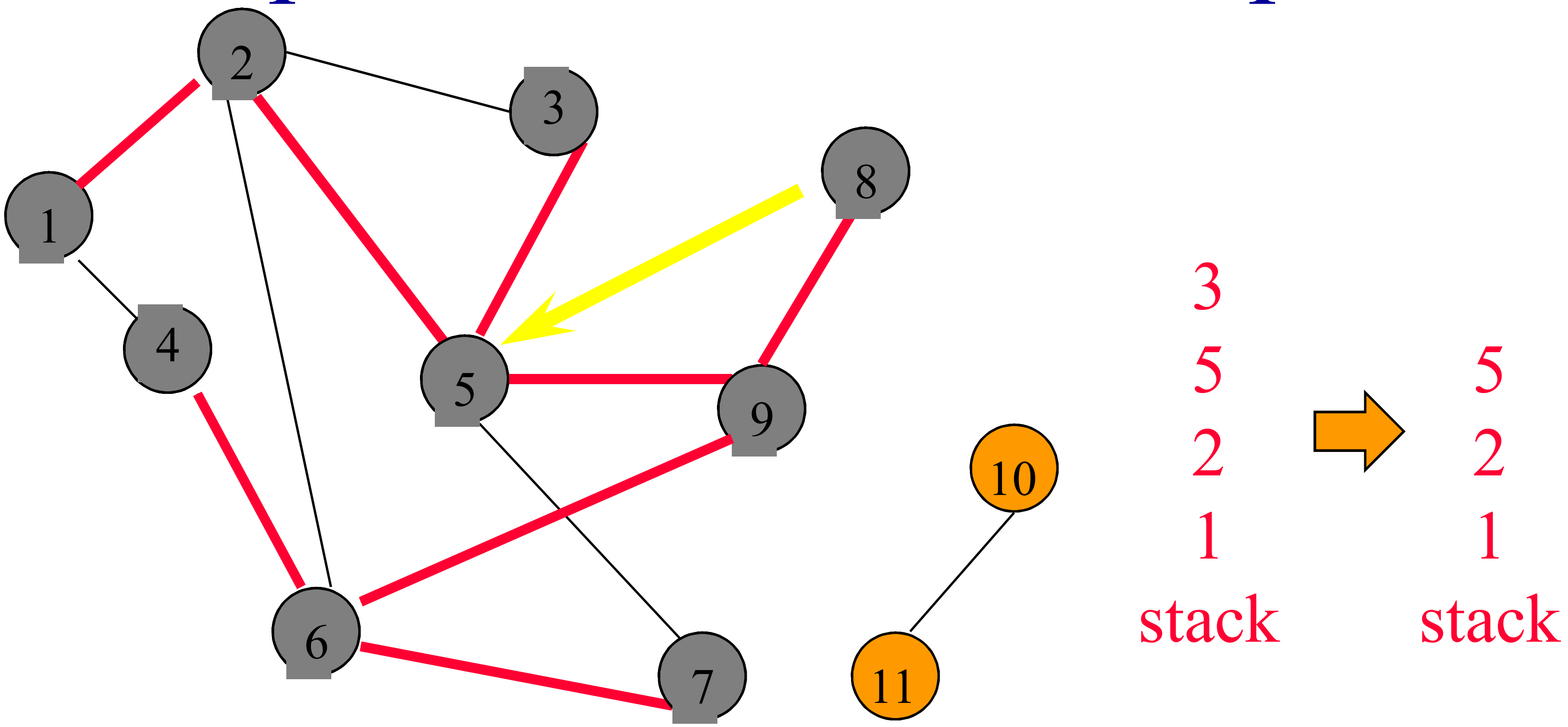
Depth-First Search Example



5
2
1
stack

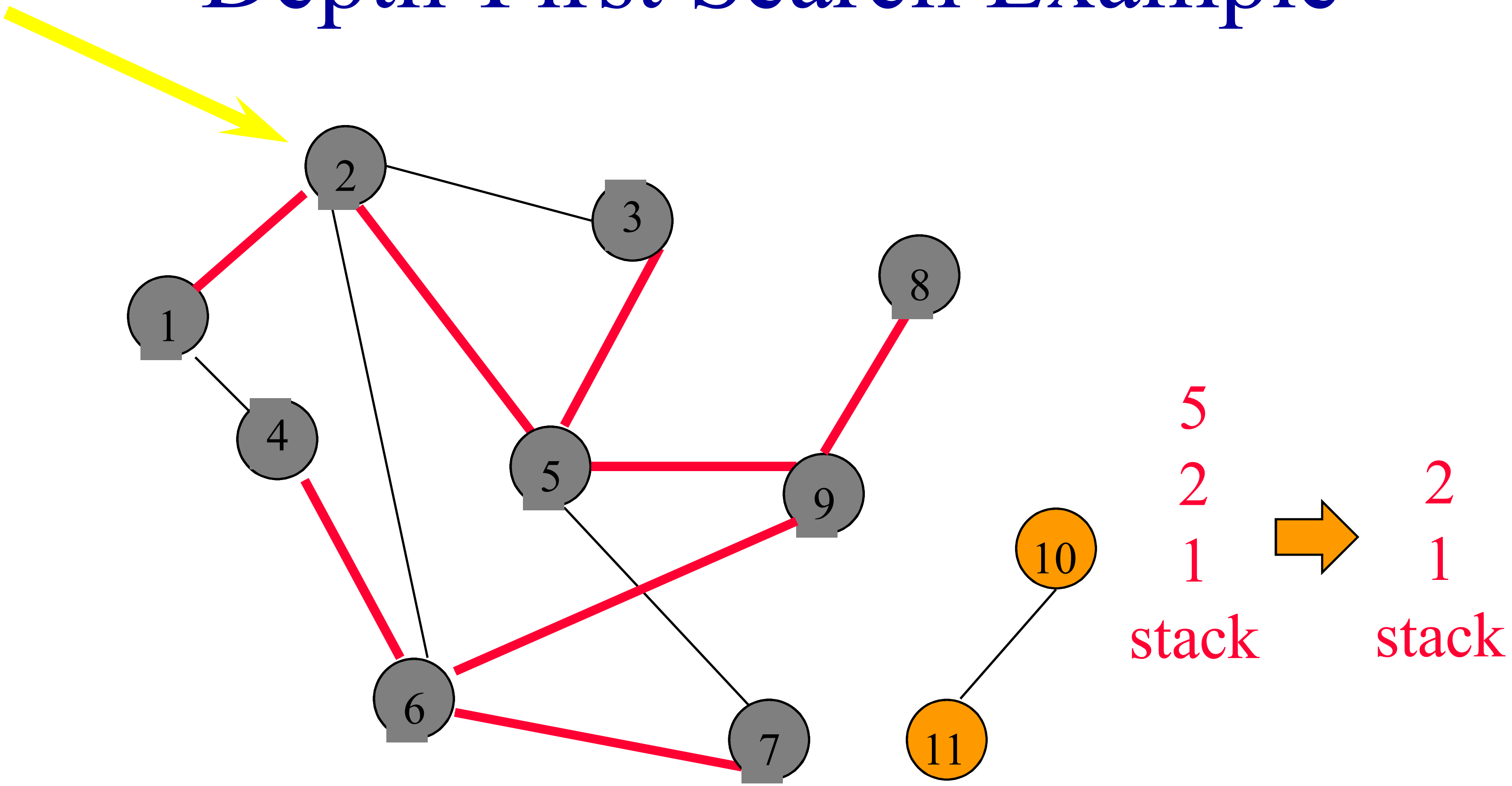
Do a DFS(3).

Depth-First Search Example



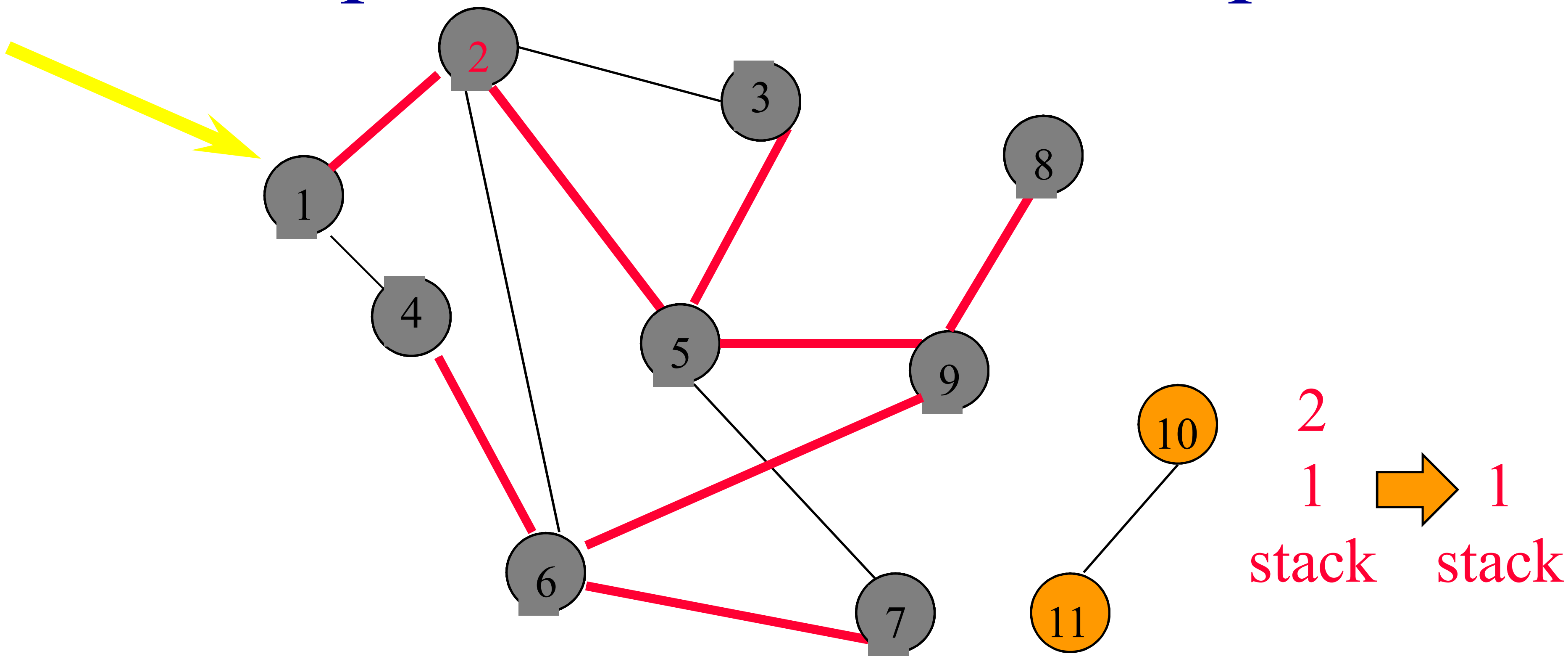
Label 3 and return to 5.

Depth-First Search Example



Return to 2.

Depth-First Search Example



Return to invoking method.

Depth-First Search Property

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

Time Complexity

- $O(n^2)$ when adjacency matrix used
- $O(n+e)$ when adjacency lists used (e is number of edges)

Path from Vertex v to Vertex u

- Start a depth-first search at vertex v.
- Terminate when vertex u is visited or when DFS ends (whichever occurs first).
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

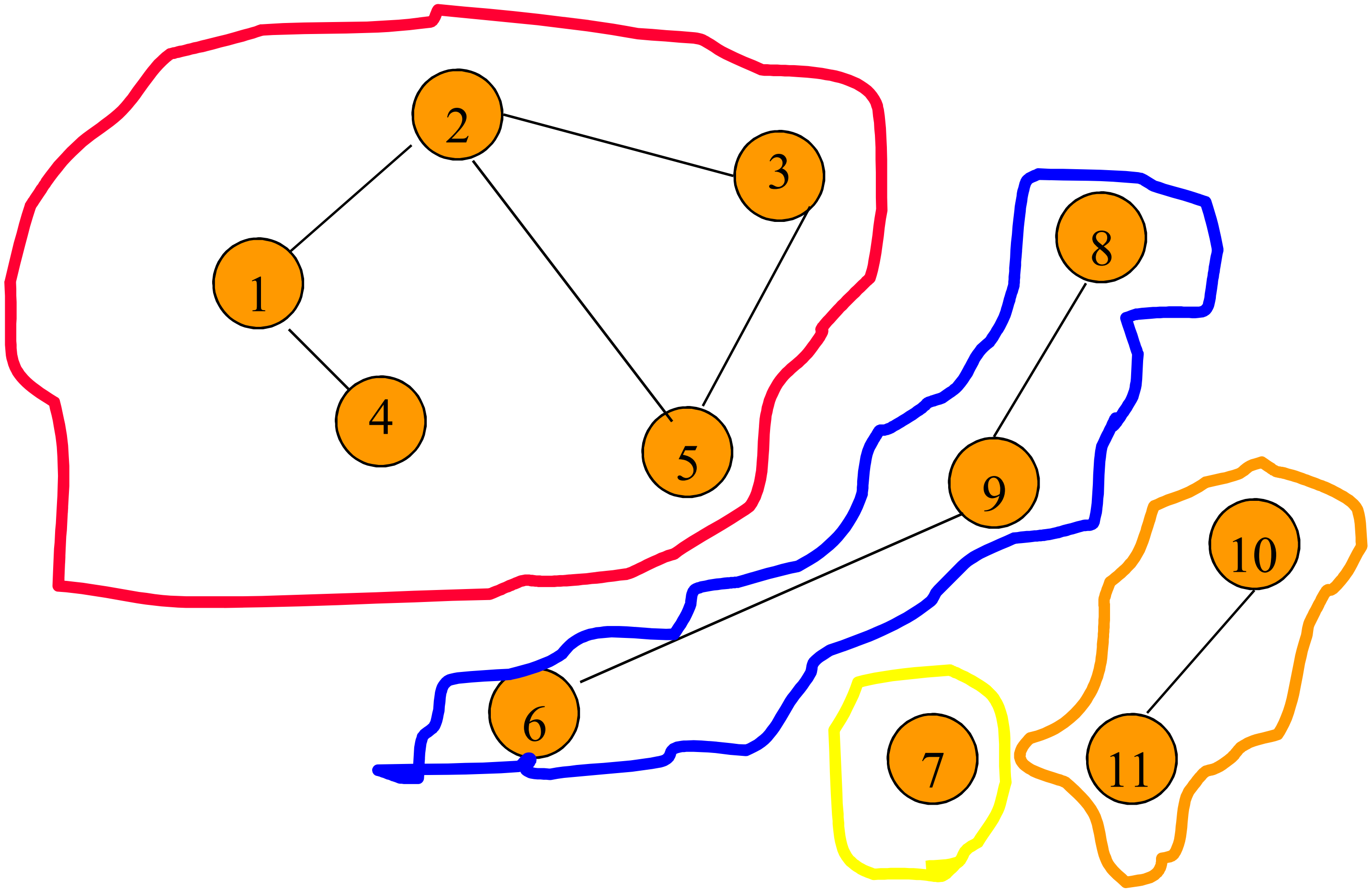
Is the Graph Connected?

- Start a depth-first search at any vertex of the graph.
- Graph is connected iff all n vertices get visited.
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

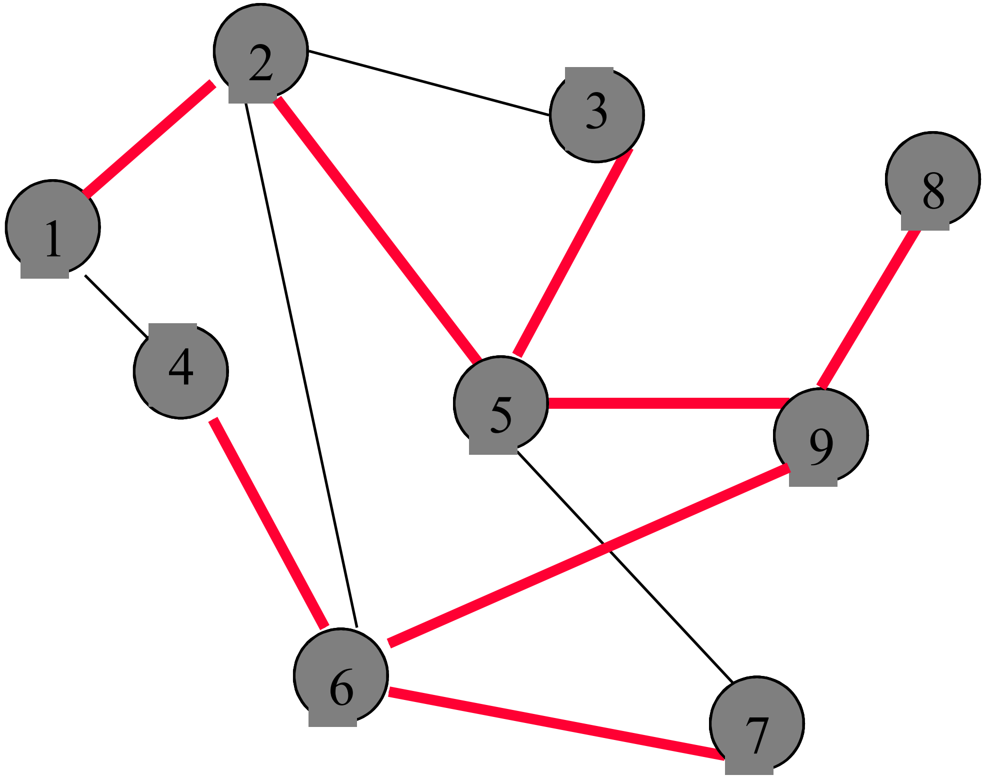
Connected Components

- Start a depth-first search at any as yet unvisited vertex of the graph.
- Newly visited vertices (plus edges between them) define a component.
- Repeat until all vertices are visited.

Connected Components



Spanning Tree



Depth-first search from vertex 1.

Depth-first spanning tree.

Spanning Tree

- Start a depth-first search at any vertex of the graph.
- If graph is connected, the n-1 edges used to get to unvisited vertices define a spanning tree (depth-first spanning tree).
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

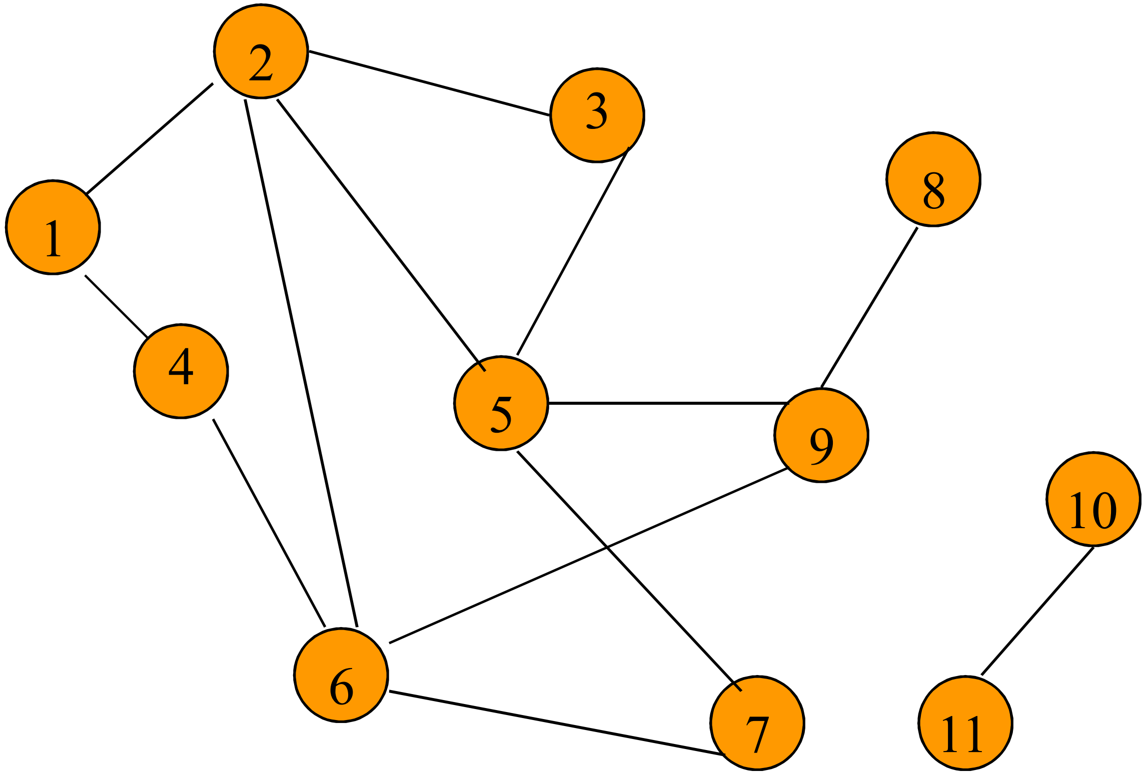
Breadth-First Search

- Visit the start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search (cont.)

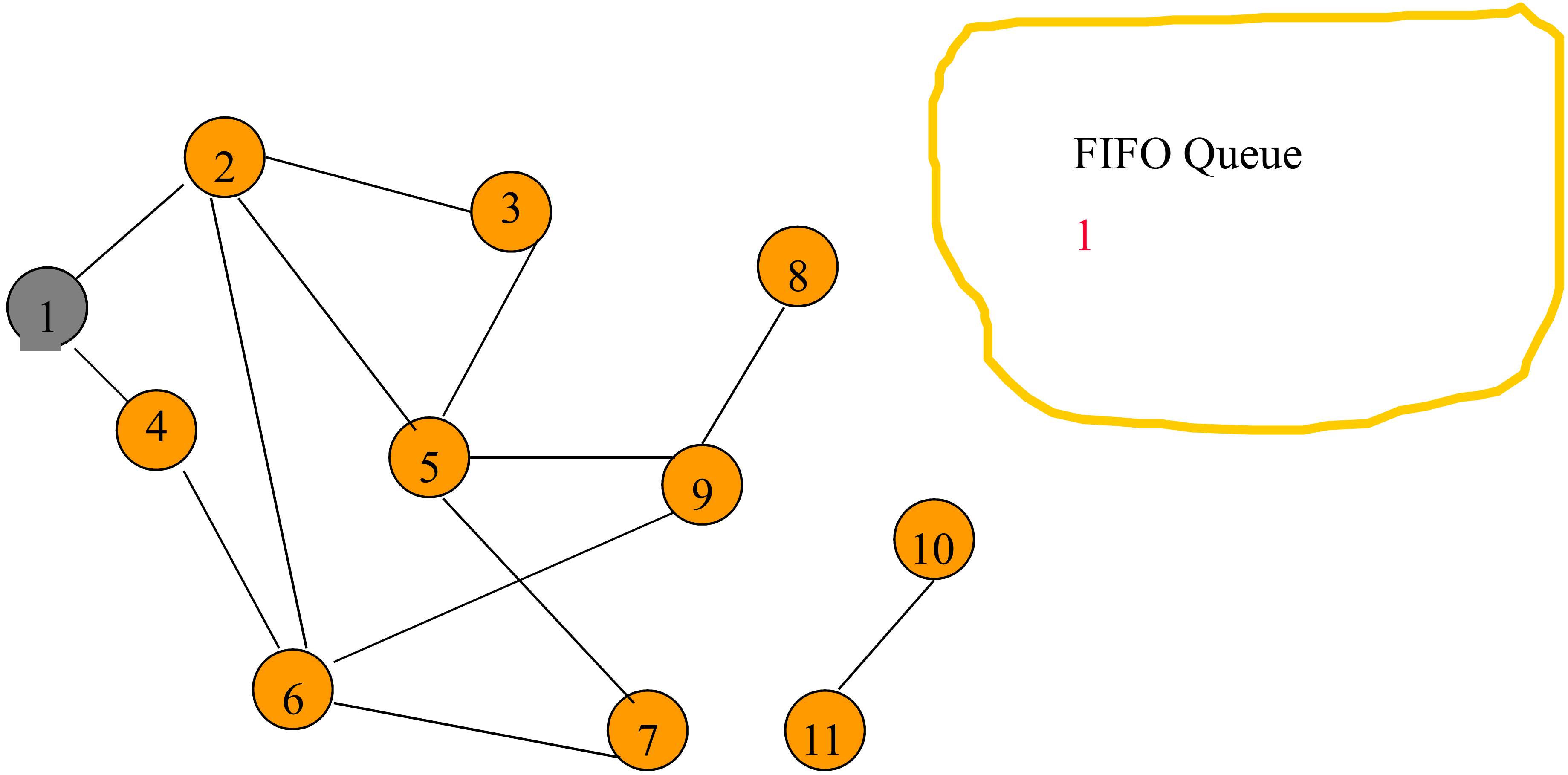
```
virtual void Graph::BFS(int v)
{// A breadth first search of the graph is carried out beginning at vertex v.
// visited[i] is set to true when v is visited. The function uses a queue.
  visited = new bool [n];
  fill(visited,visited+n,false);
  visited[v] = true;
  Queue<int> q;
  q.Push(v);
  while (!q.IsEmpty()) {
    v = q.Front();
    q.Pop();
    for (all vertices w adjacent to v) // actual code uses an iterator
      if (!visited[w]) {
        q.Push(w);
        visited[w] = true;
      }
  } // end of while loop
  delete [] visited;
}
```

Breadth-First Search Example



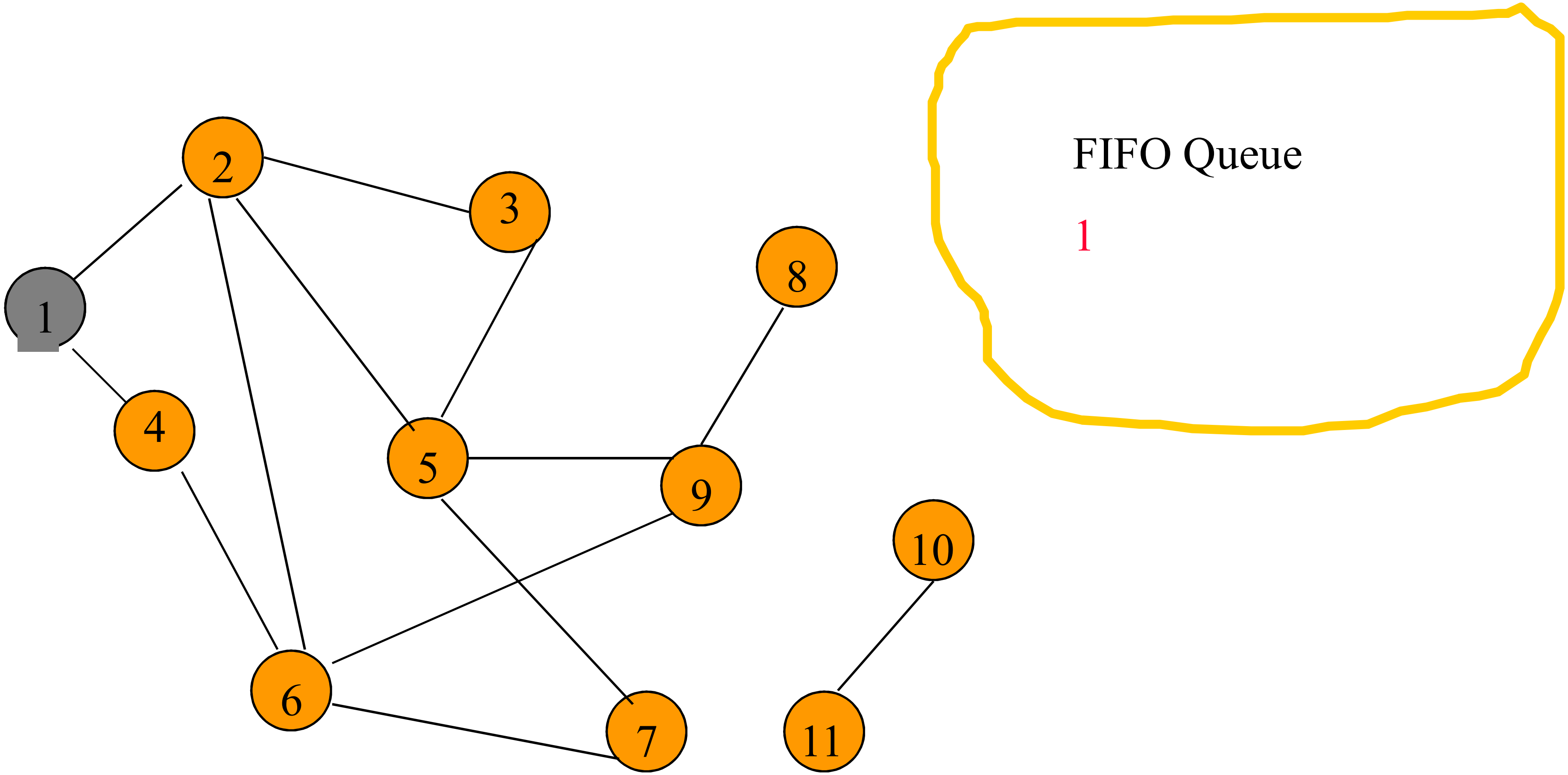
Start search at vertex 1.

Breadth-First Search Example



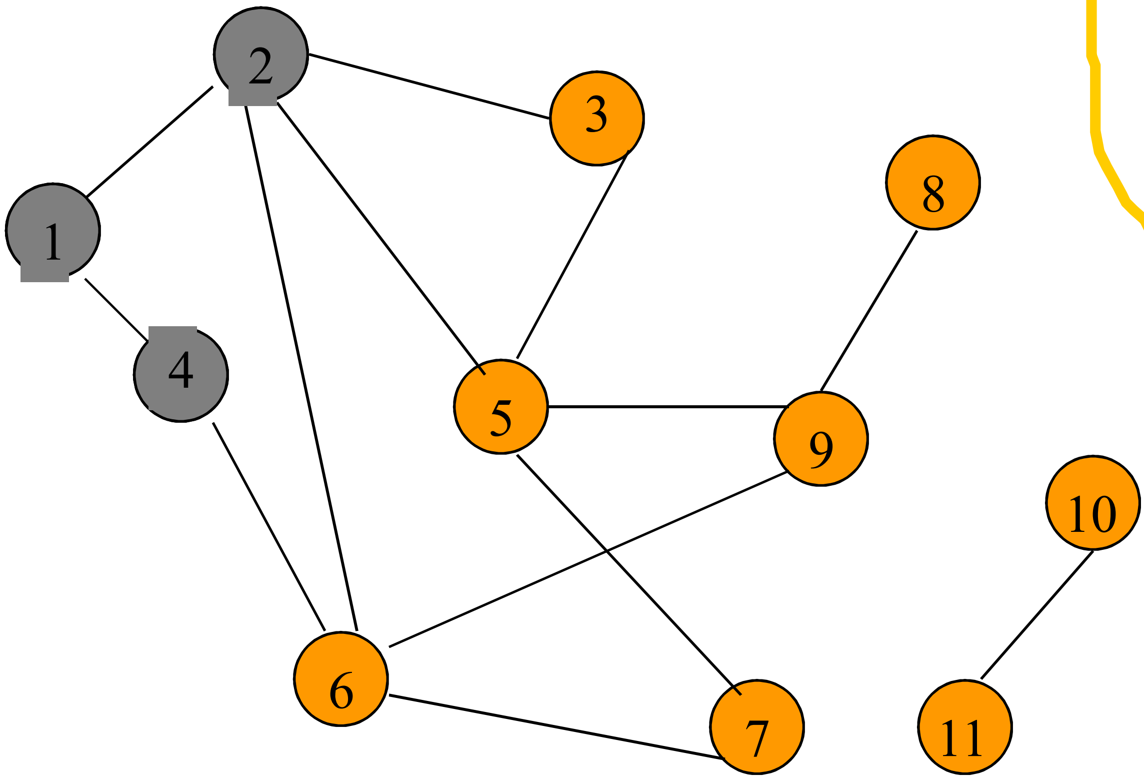
Visit/mark/label start vertex and put in a FIFO queue.

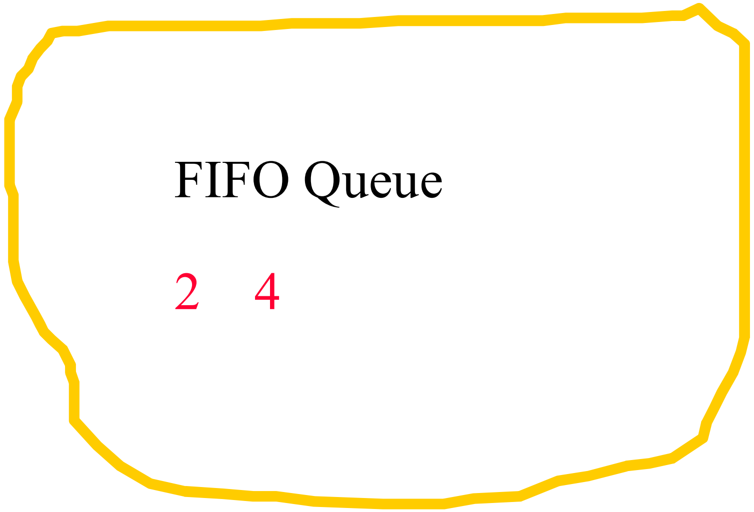
Breadth-First Search Example



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

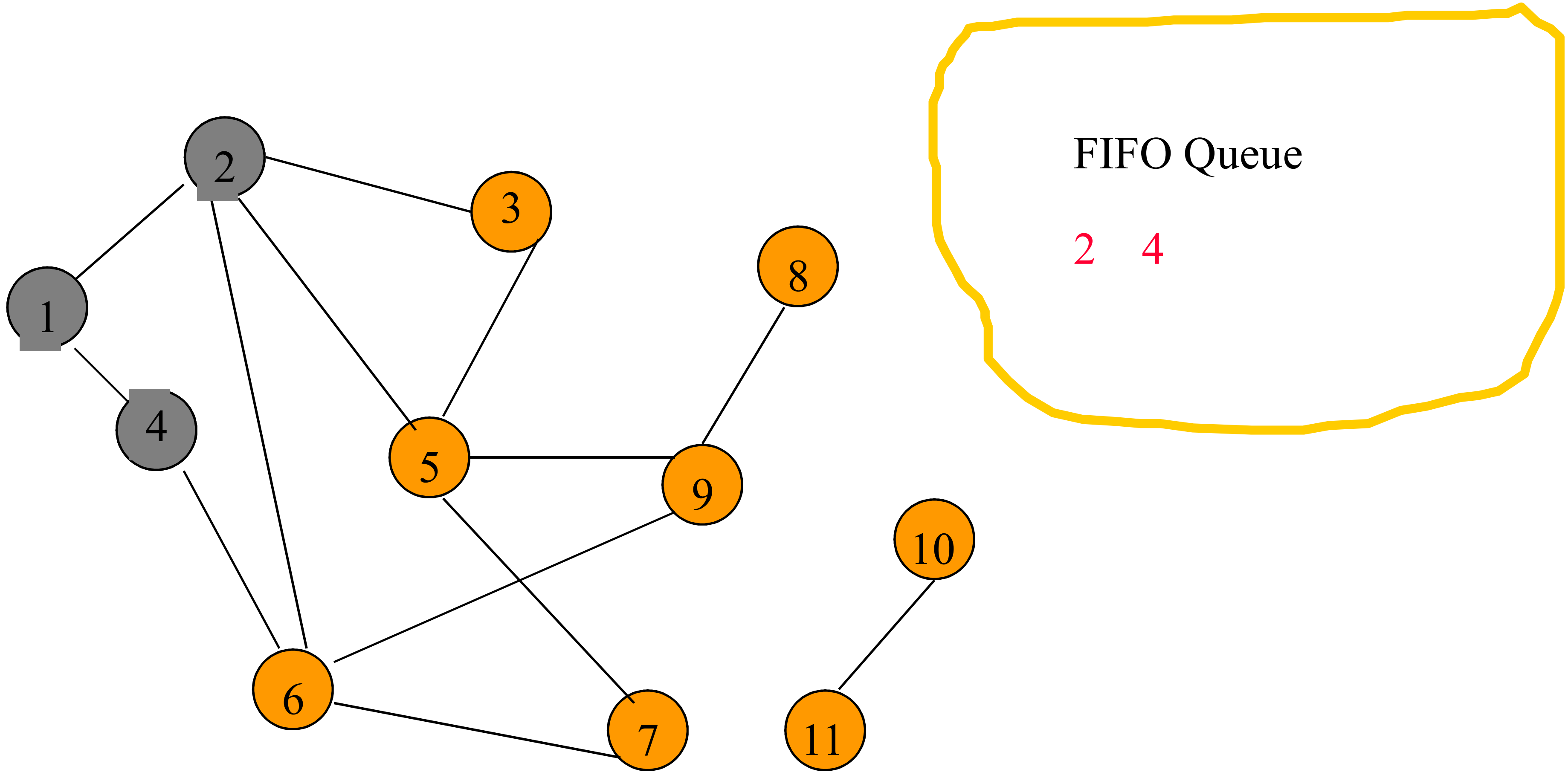
Breadth-First Search Example





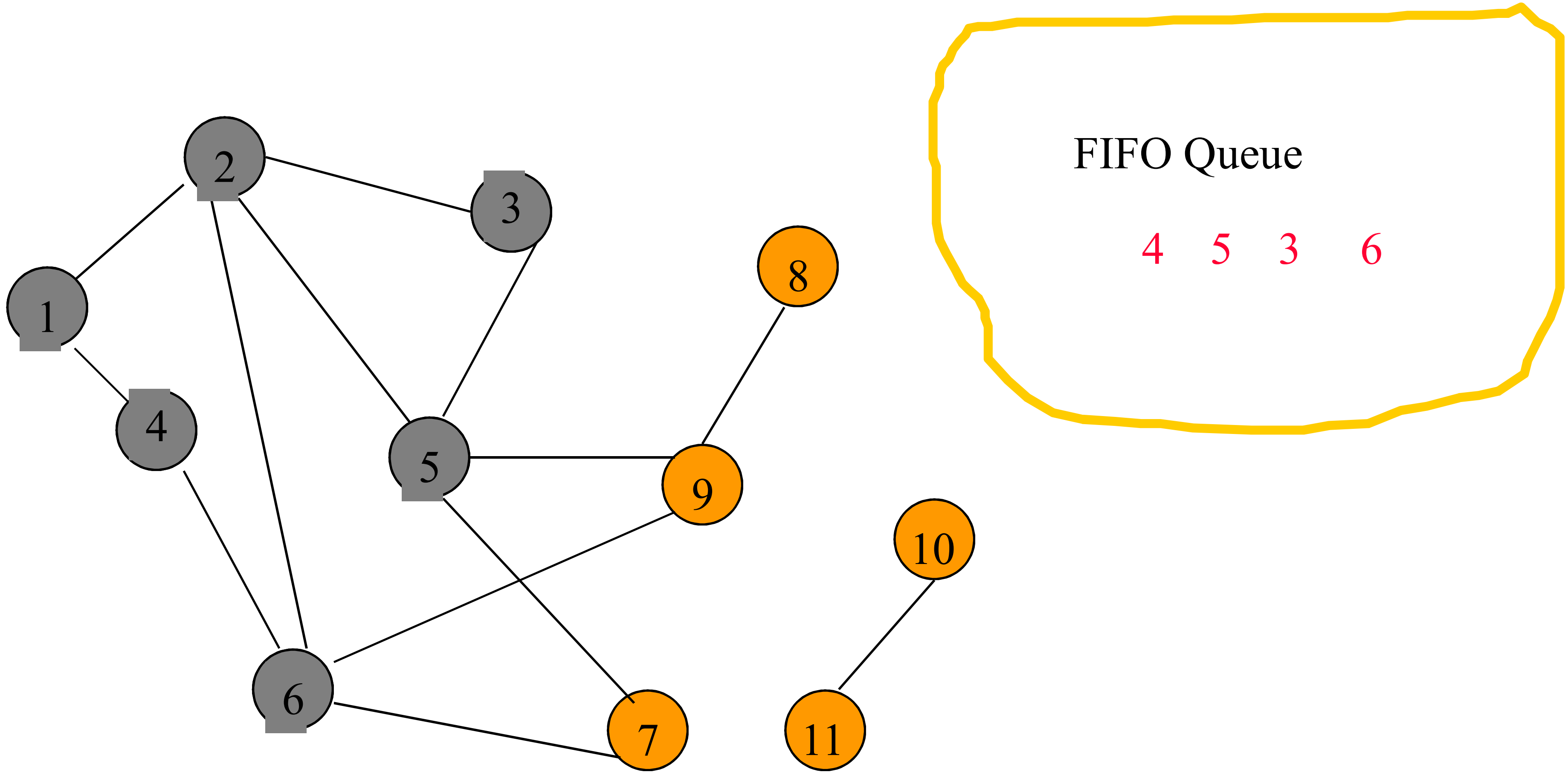
Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



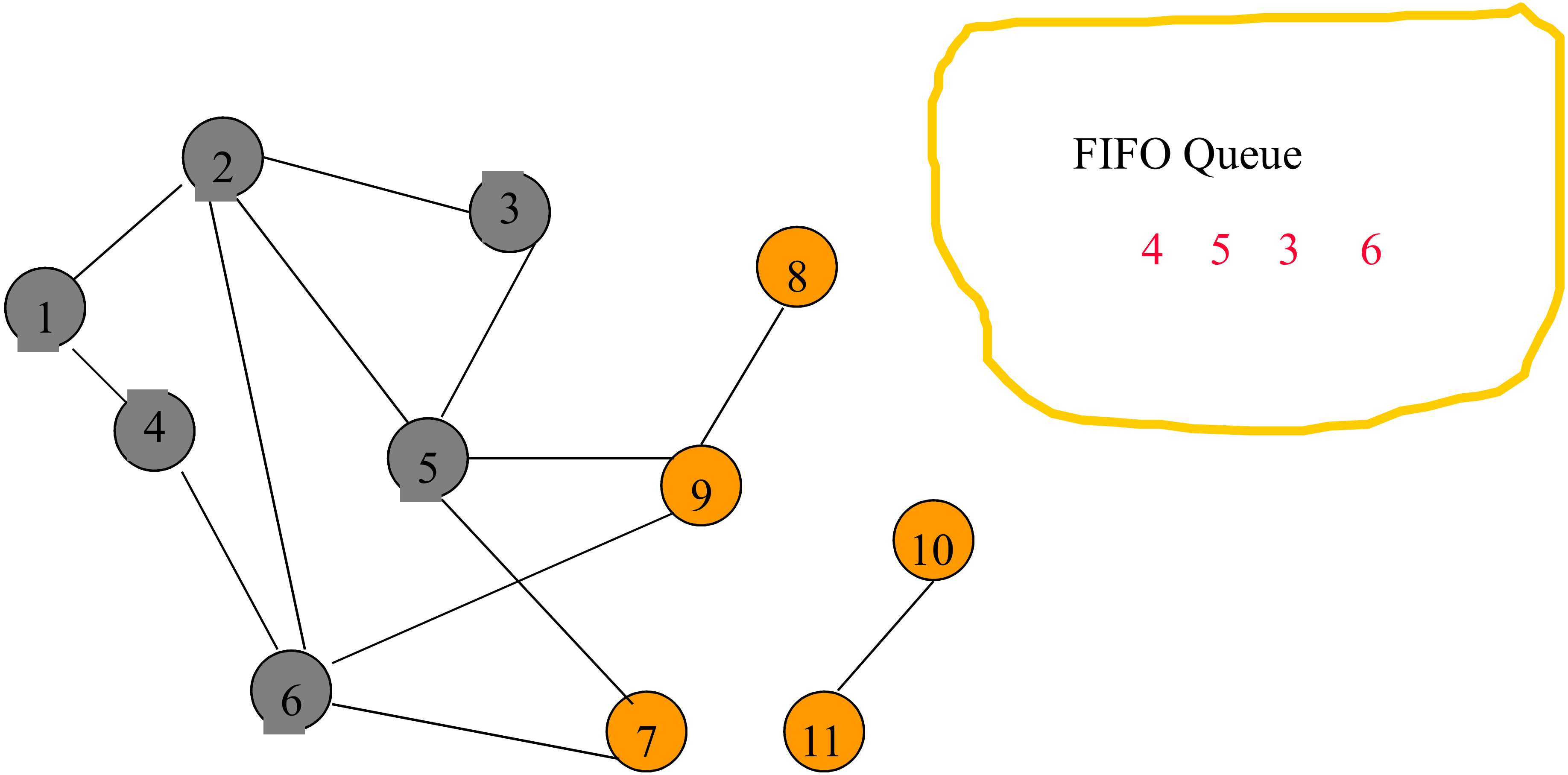
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



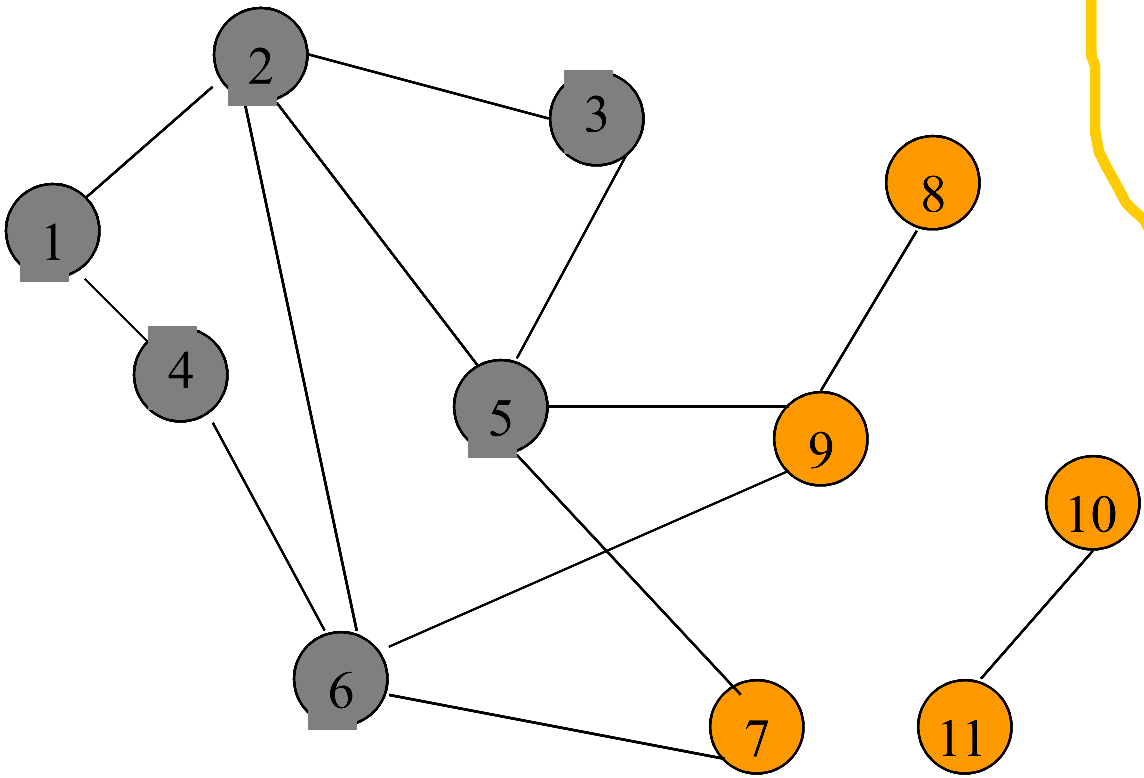
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

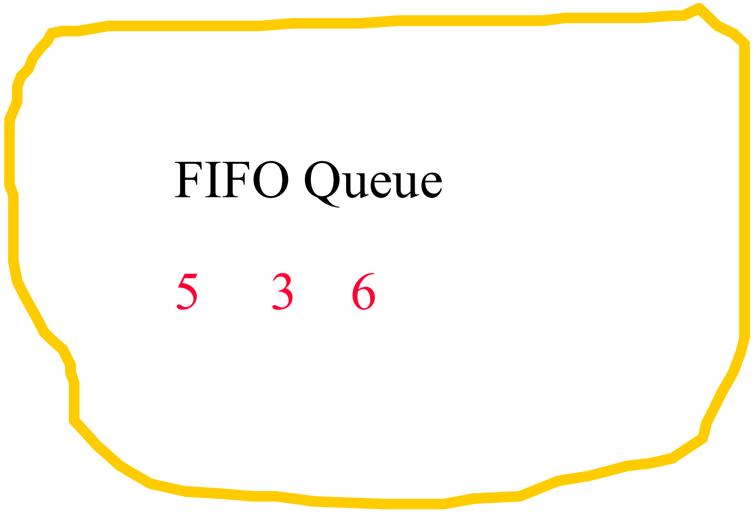
Breadth-First Search Example



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

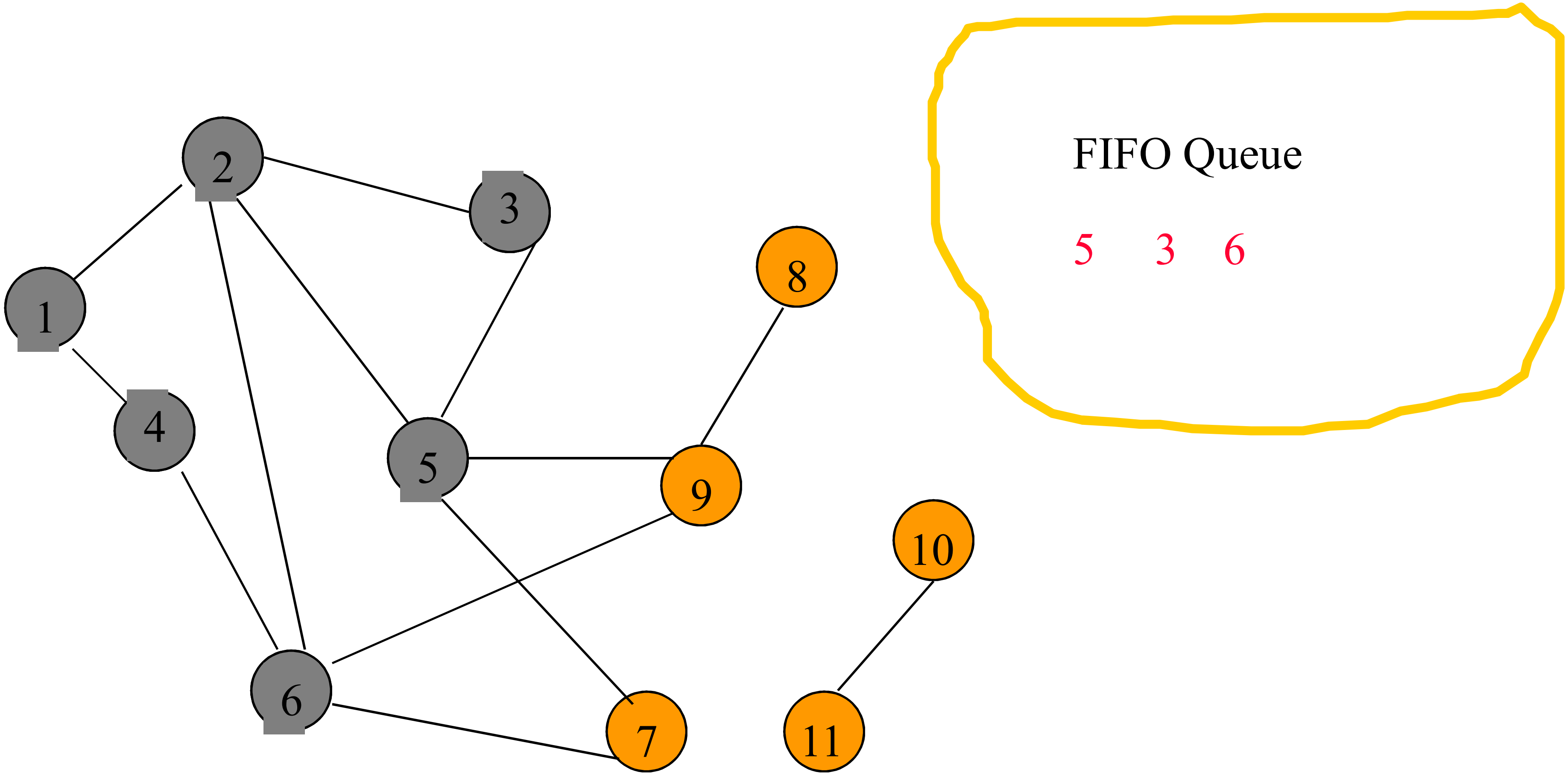
Breadth-First Search Example





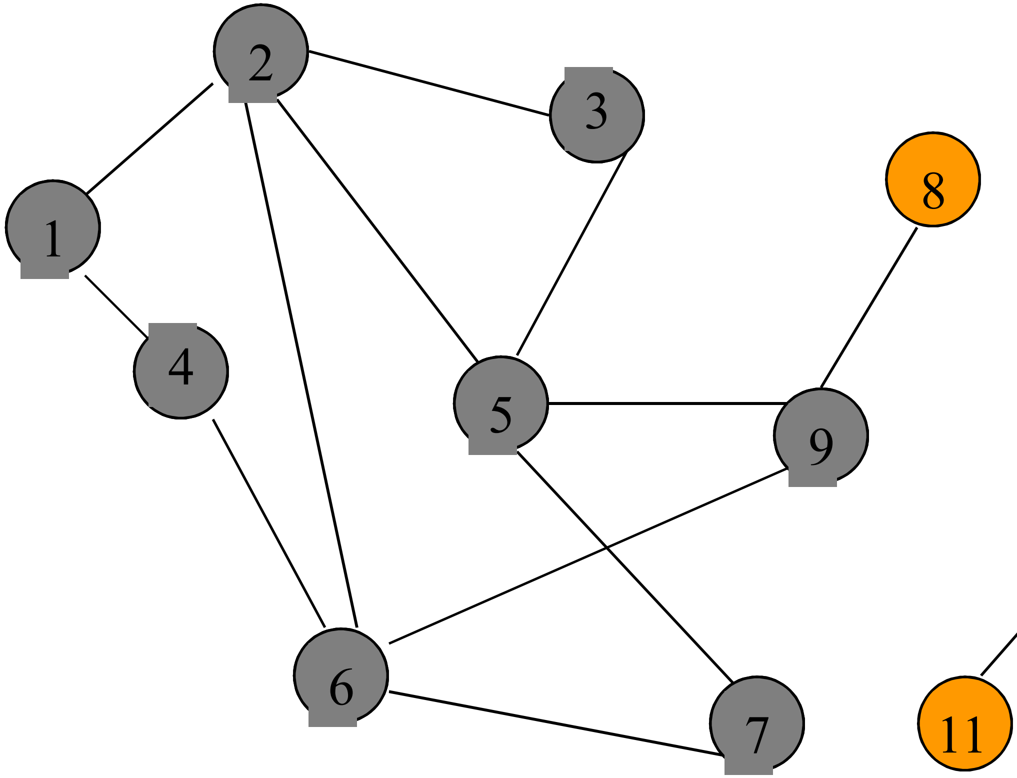
Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

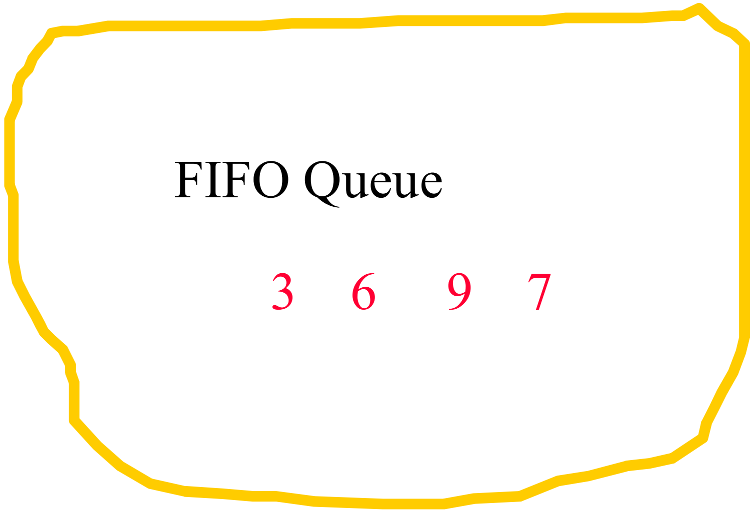
Breadth-First Search Example



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

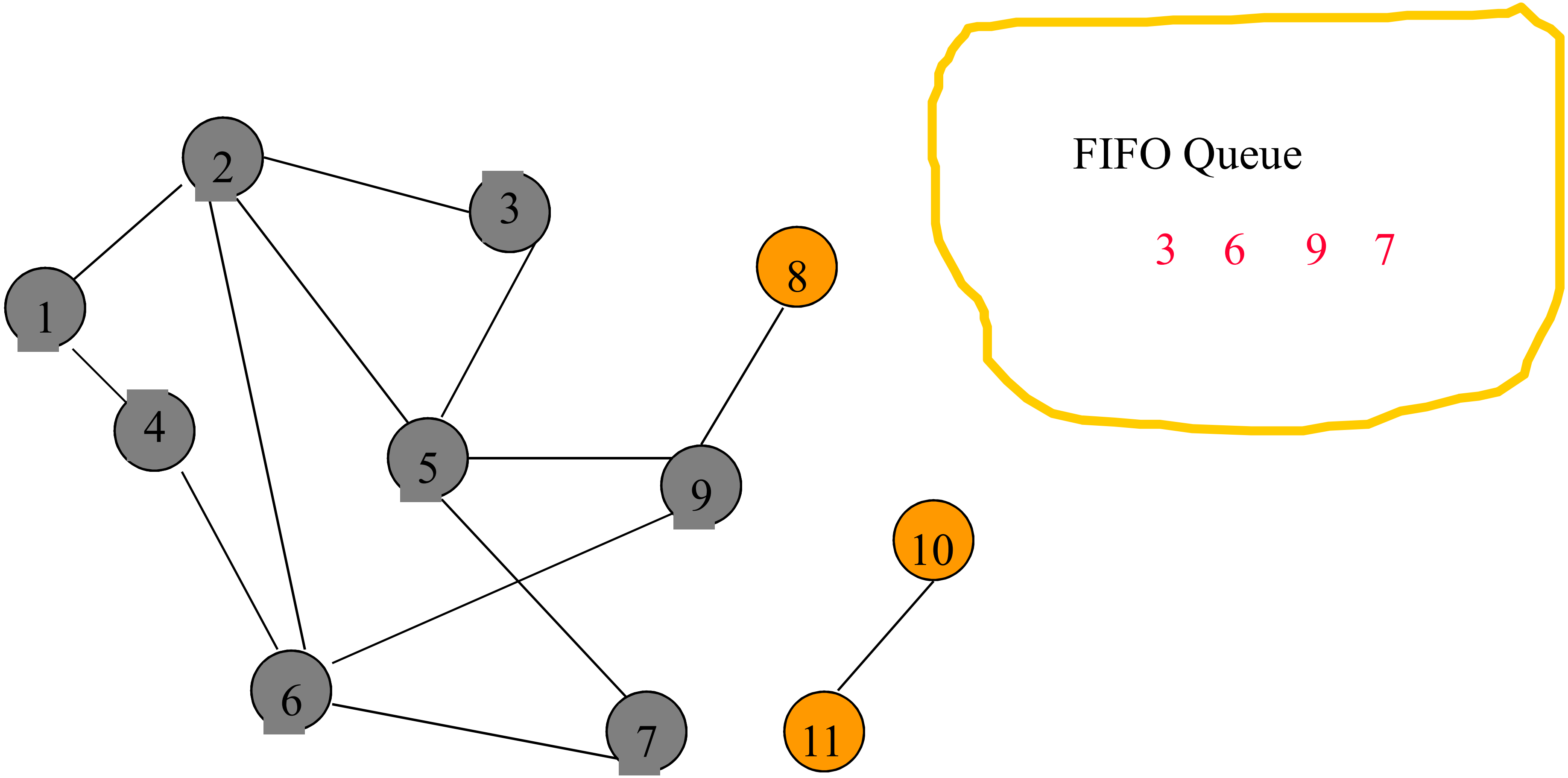
Breadth-First Search Example





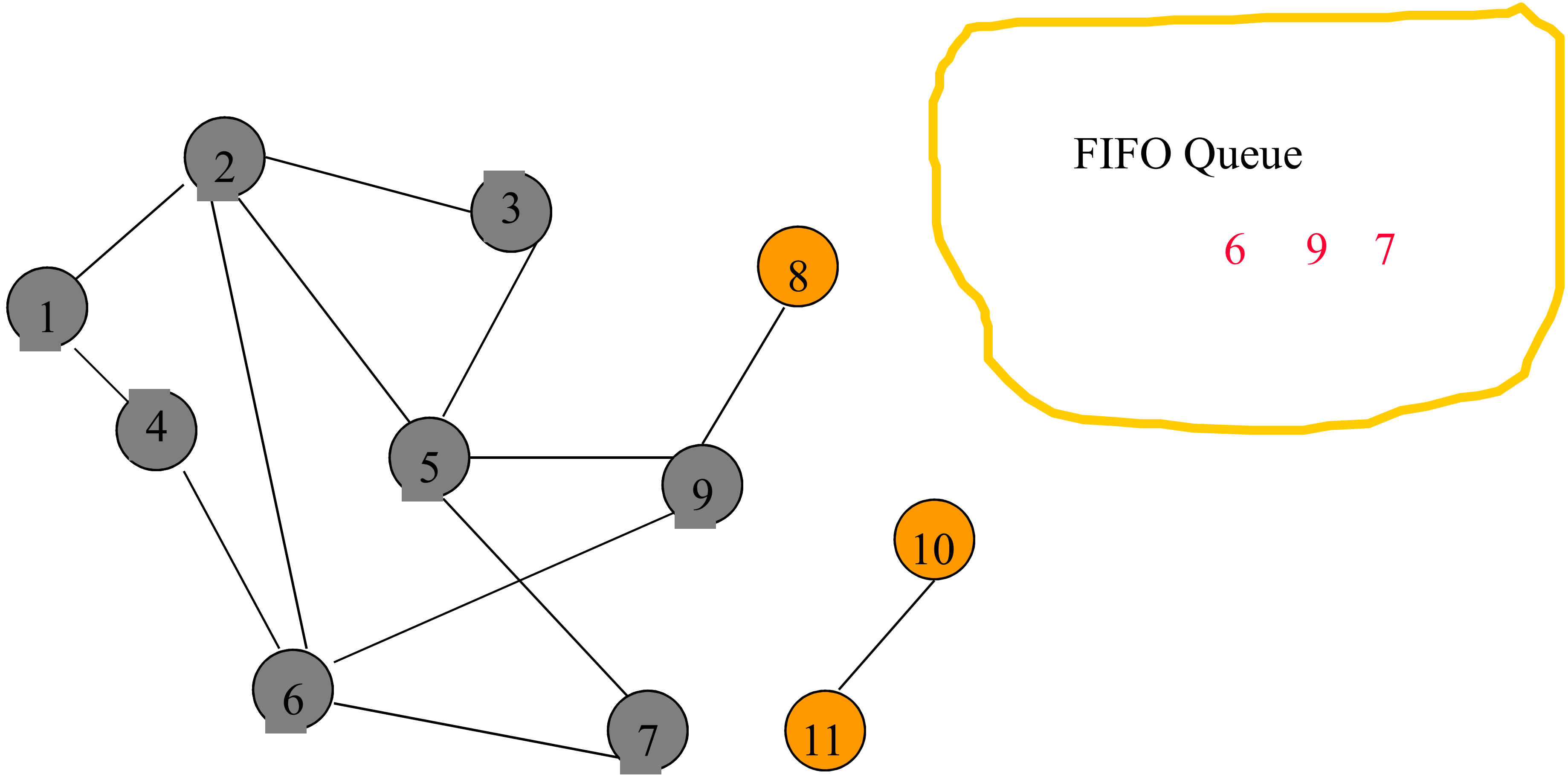
Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



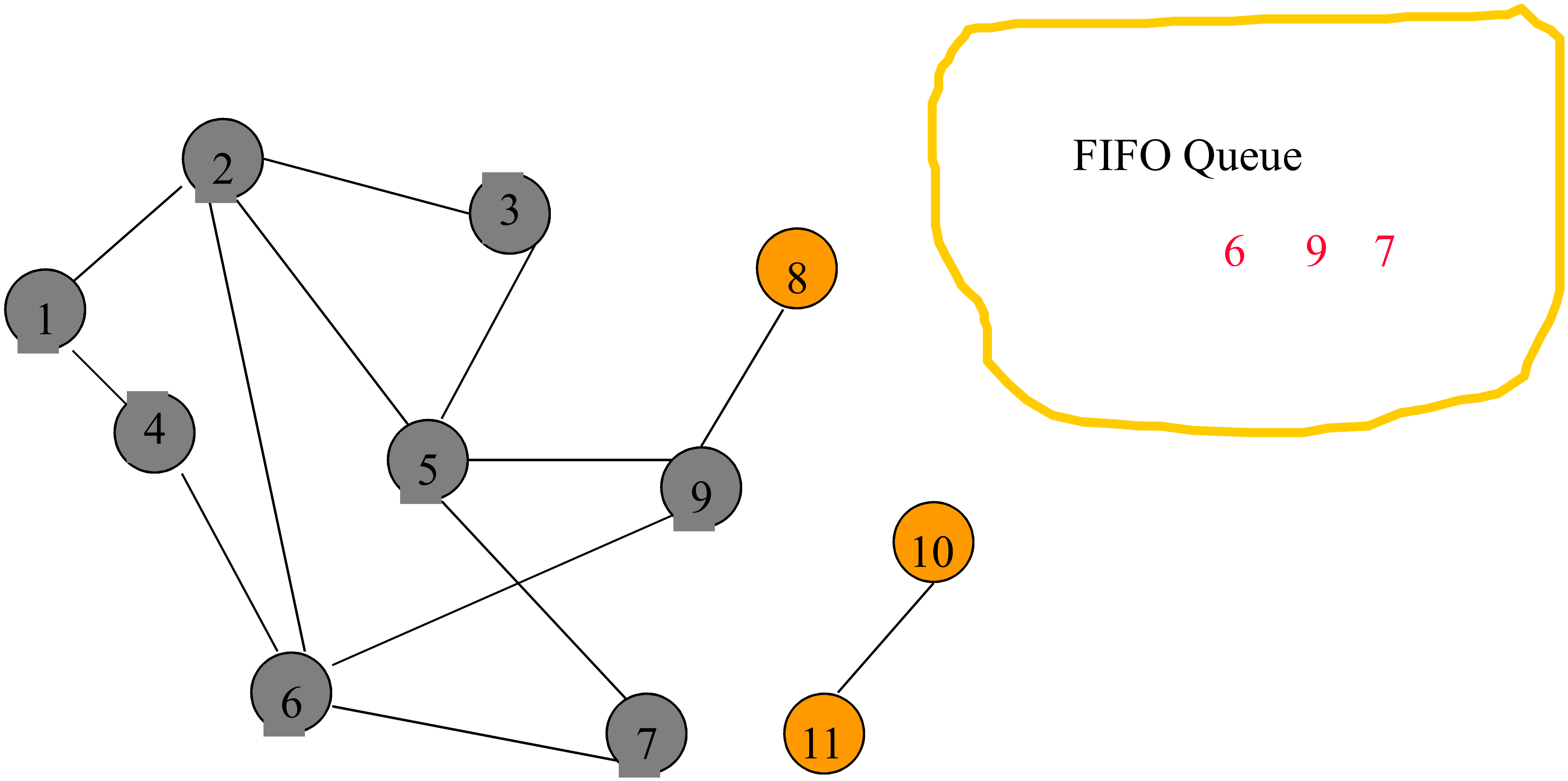
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



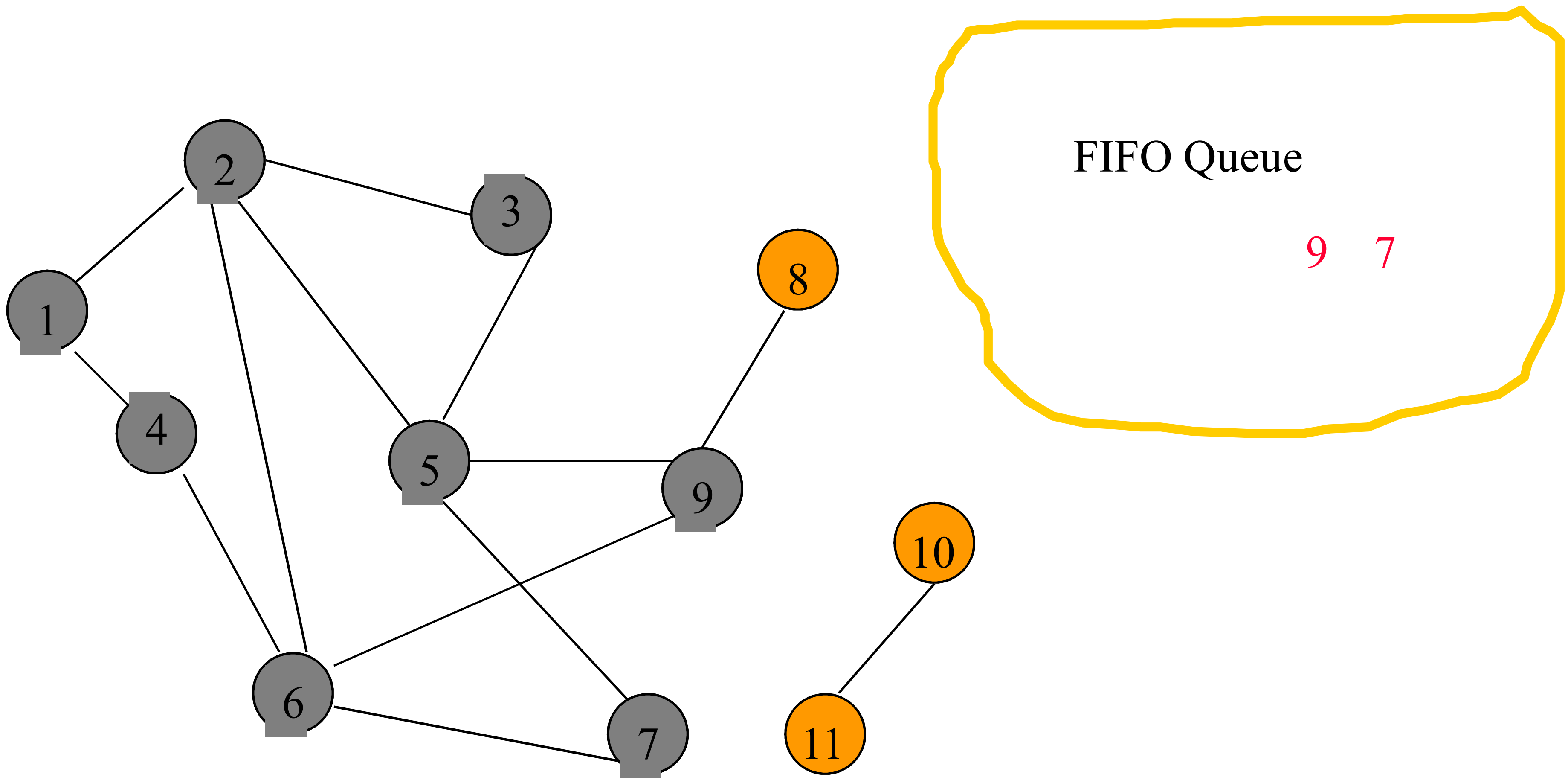
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



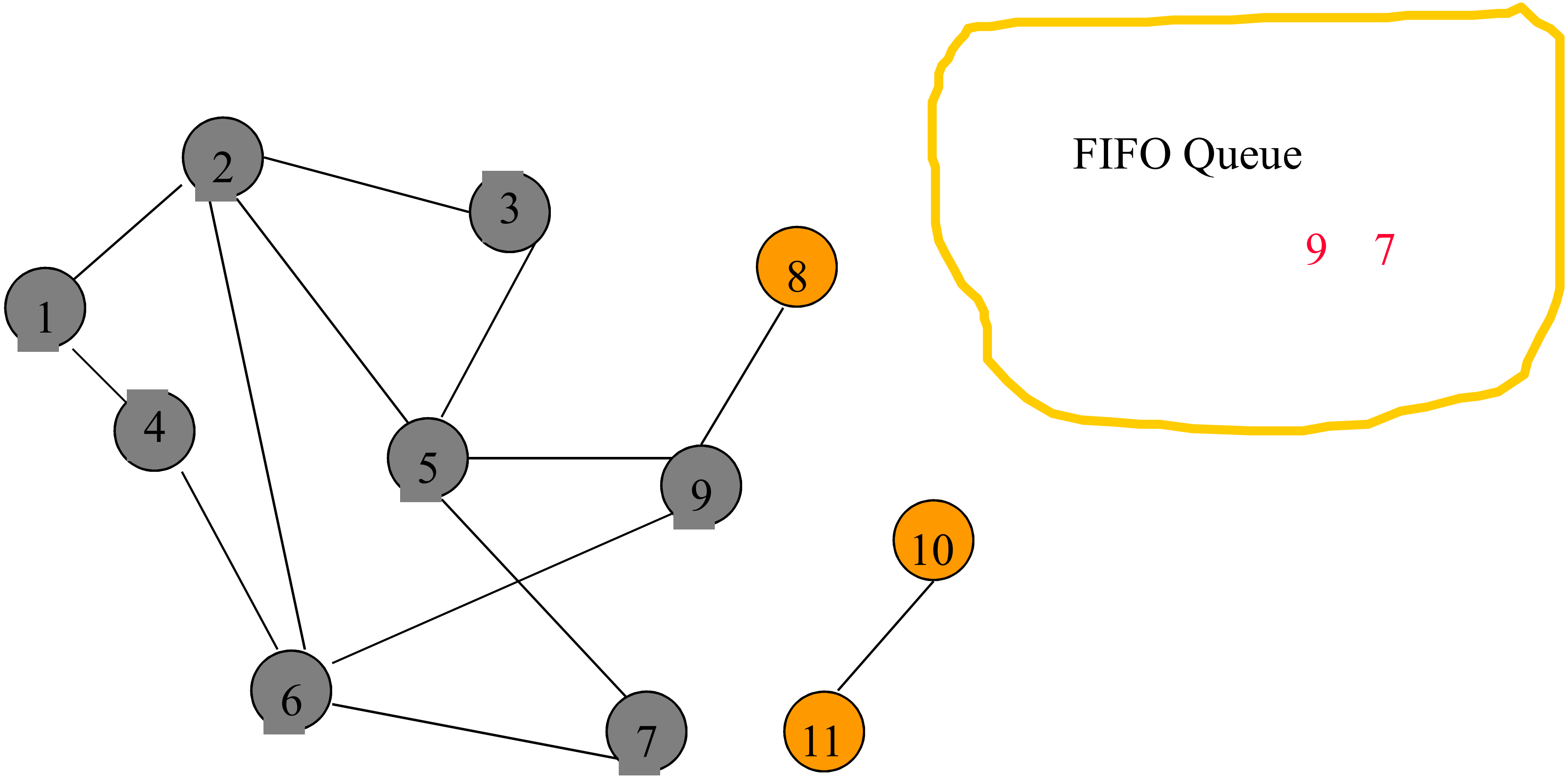
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



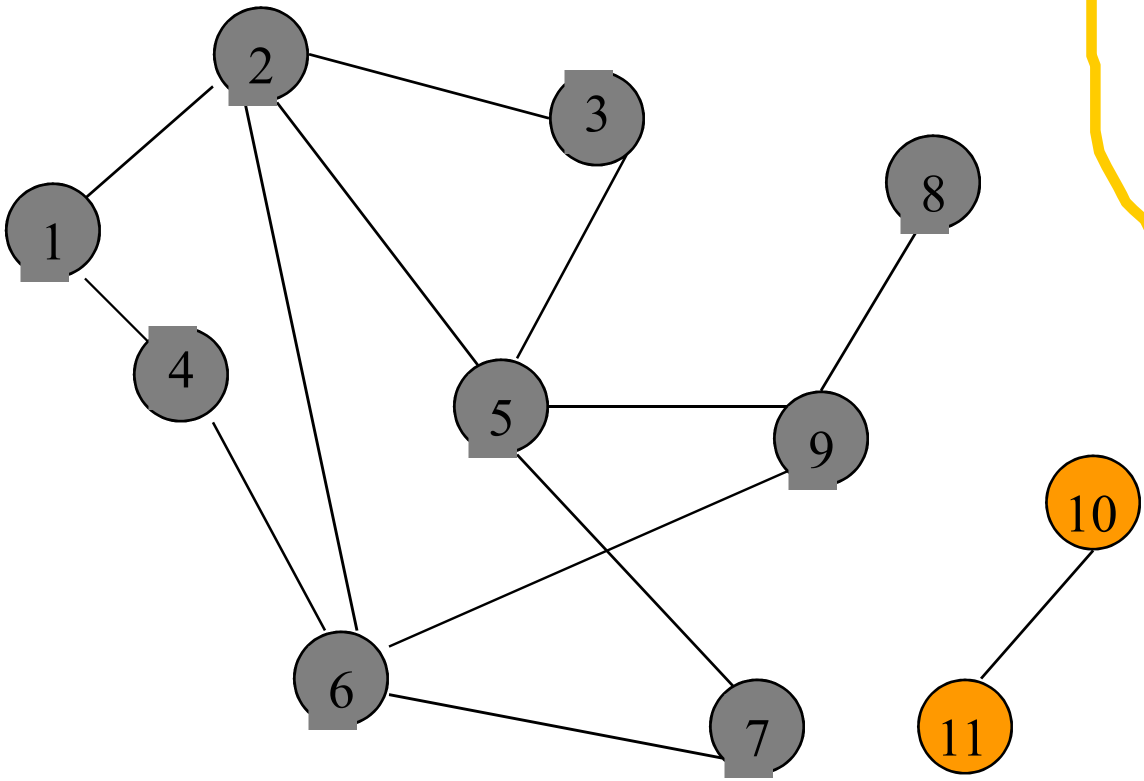
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

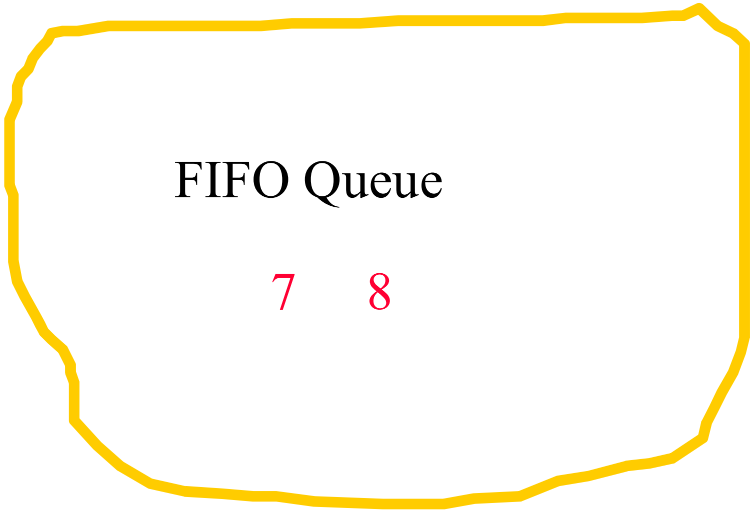
Breadth-First Search Example



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

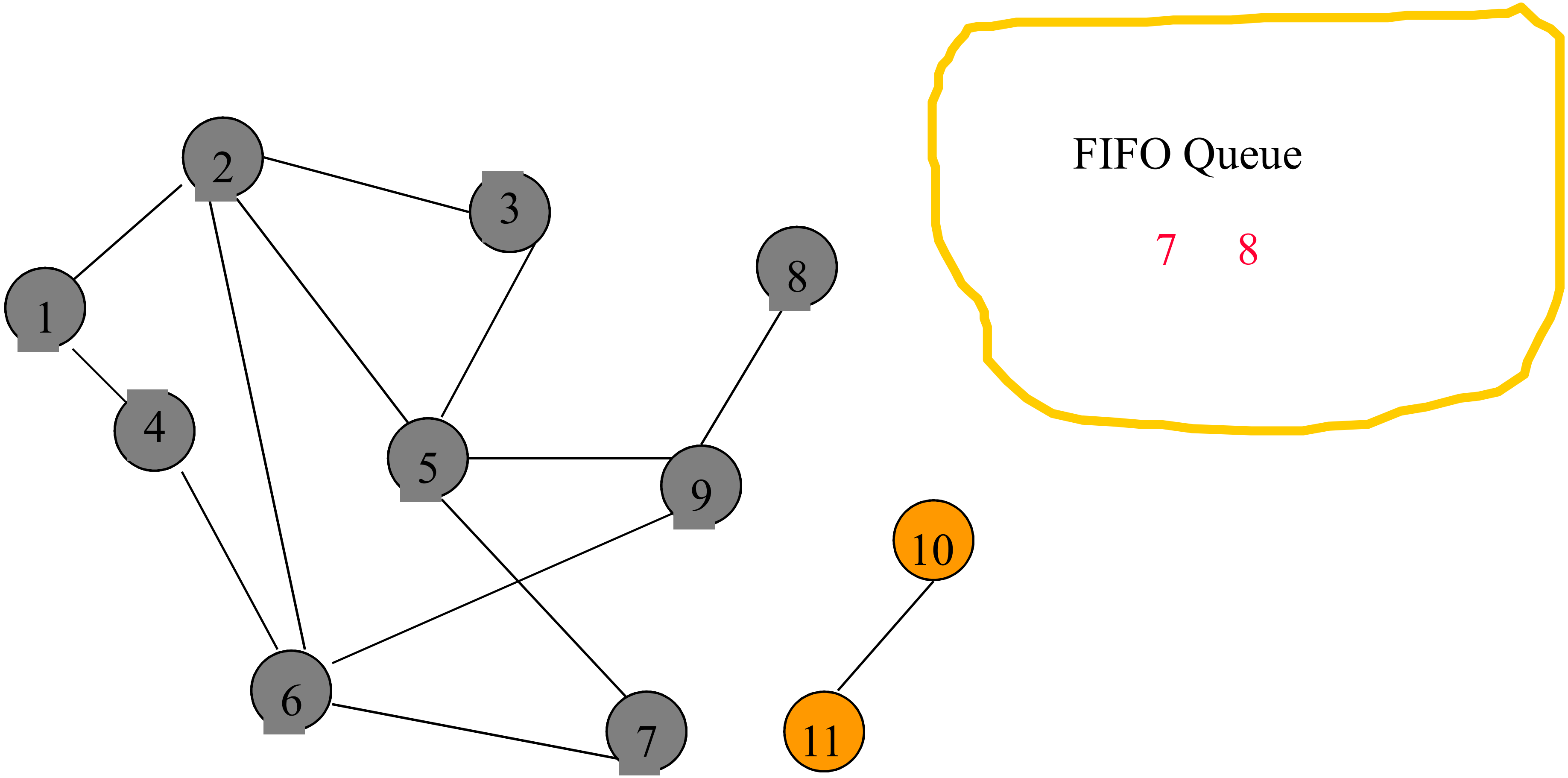
Breadth-First Search Example





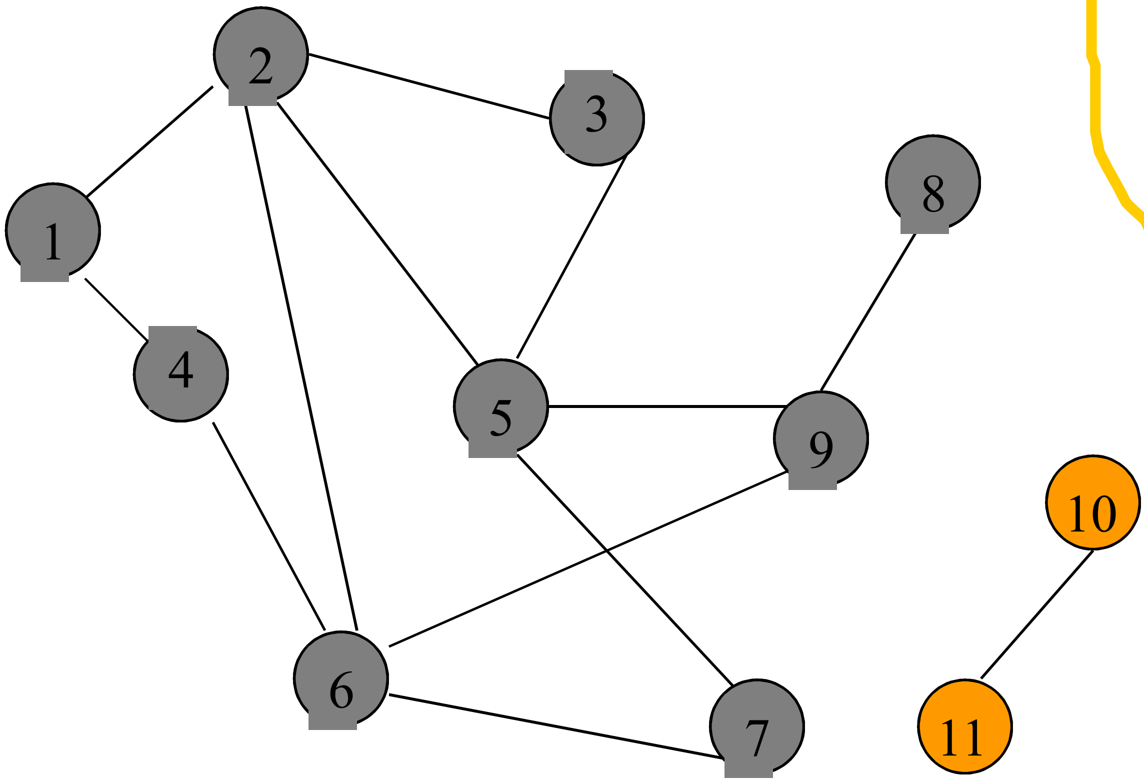
Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

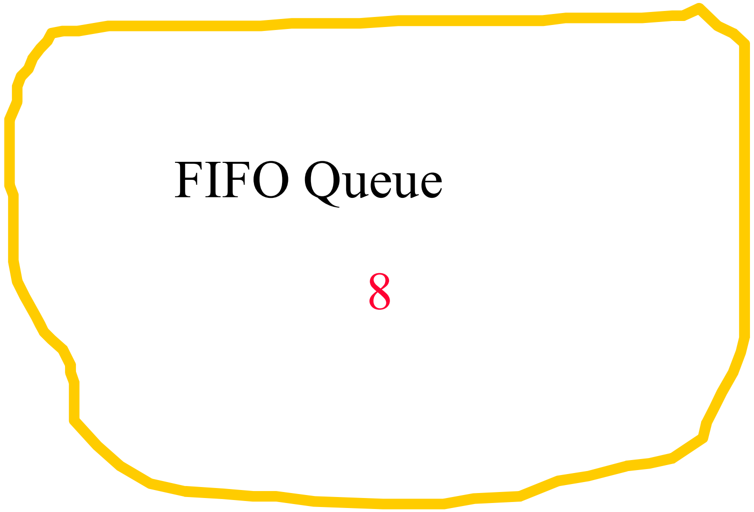
Breadth-First Search Example



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

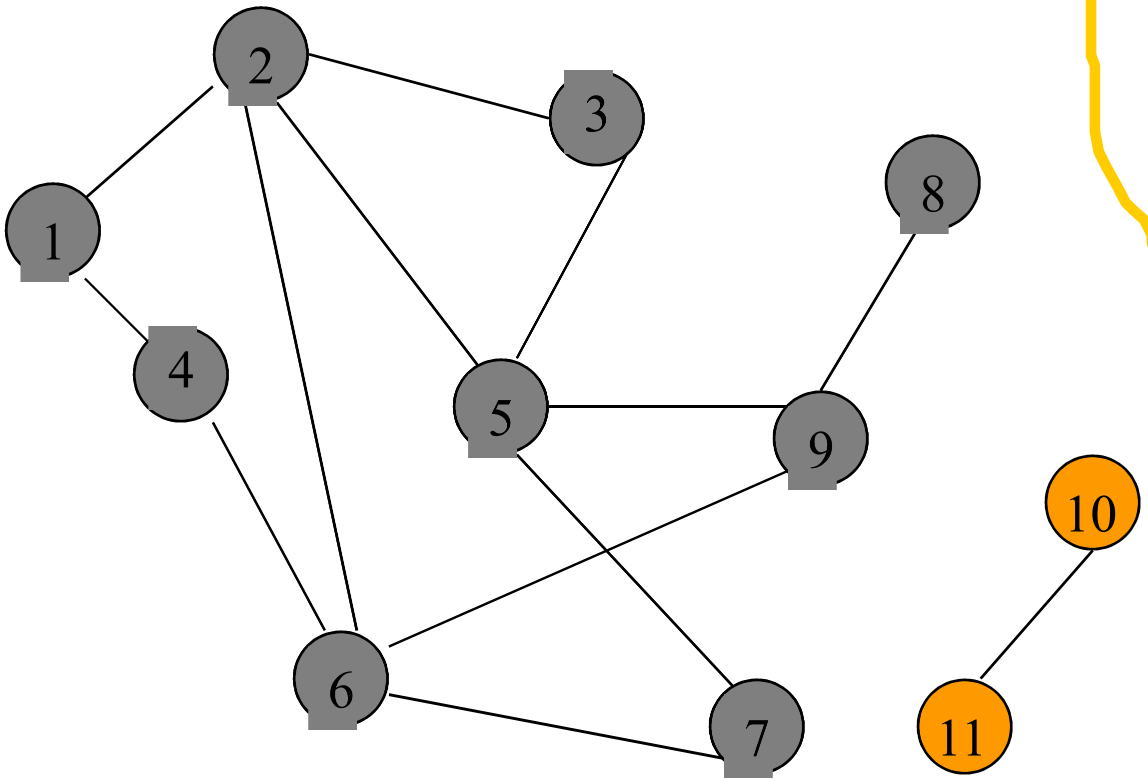
Breadth-First Search Example

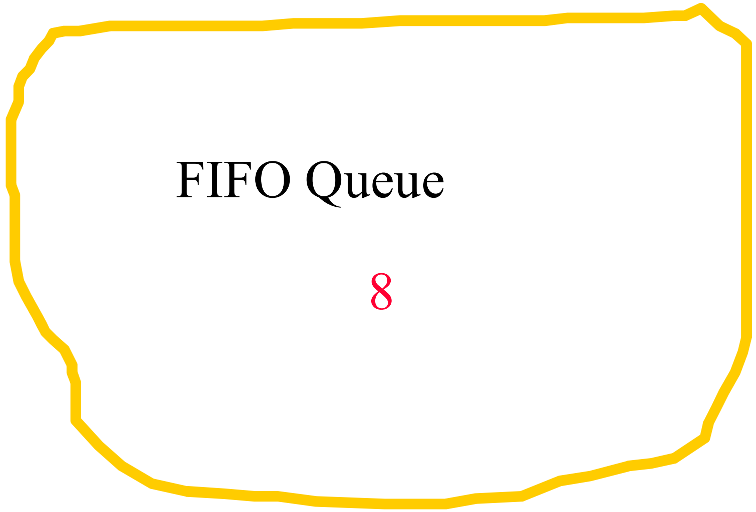




Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

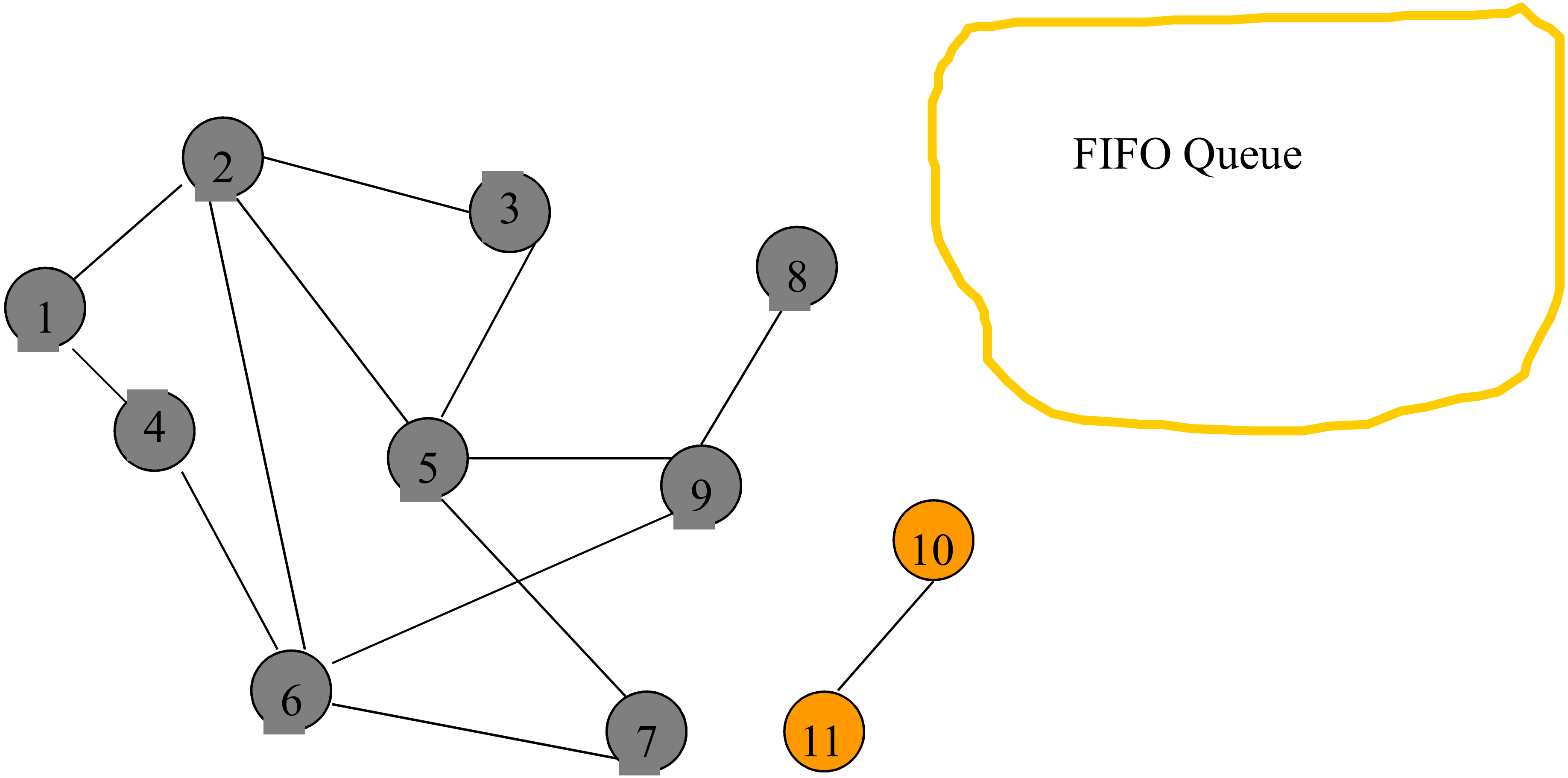
Breadth-First Search Example





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

Breadth-First Search Example



Queue is empty. Search terminates.

Time Complexity

- Each visited vertex is put on (and so removed from) the queue exactly once.
- When a vertex is removed from the queue, we examine its adjacent vertices.
  - O(n) if adjacency matrix used
  - O(vertex degree) if adjacency lists used
- Time
  - $O(n^2)$ when adjacency matrix used
  - O(n+e) when adjacency lists used (e is number of edges)