Chapter Eight

Introduction to Graphs

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).

$$u \longrightarrow v$$

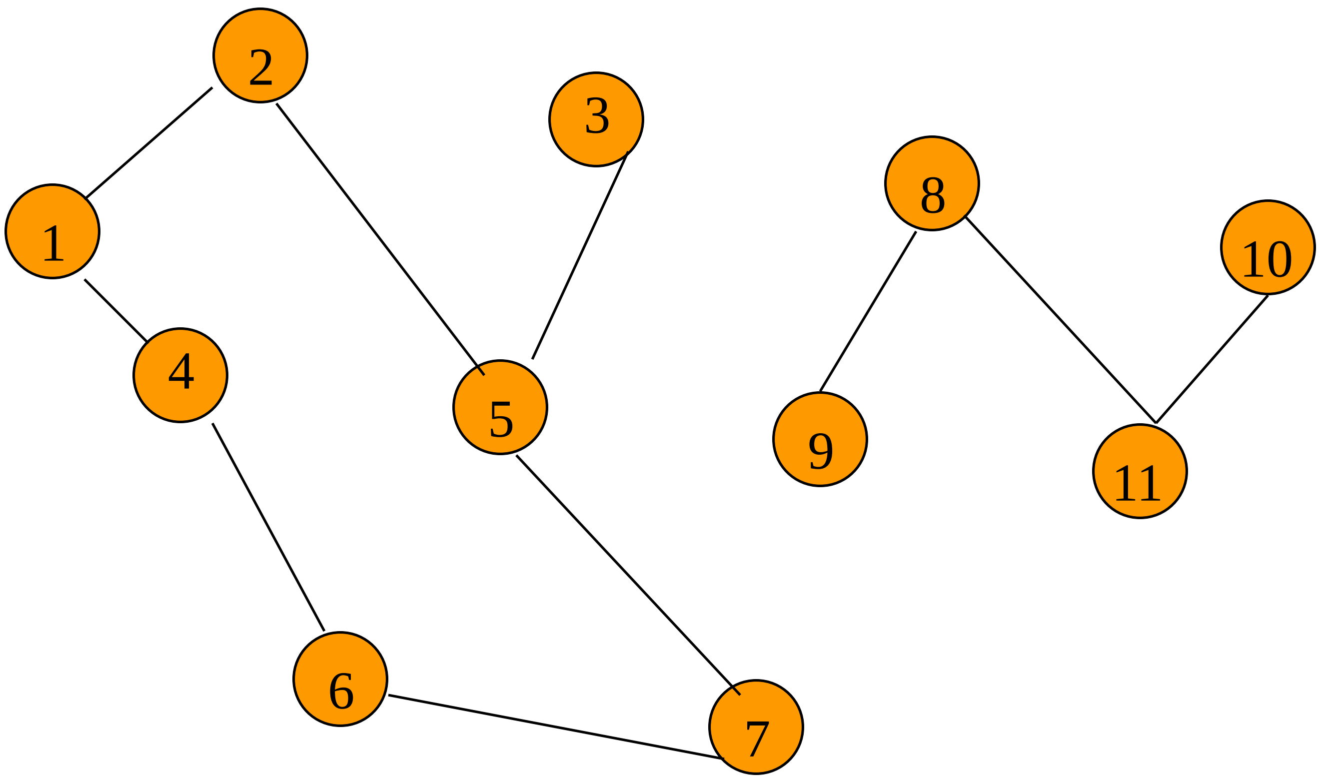
Graphs

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

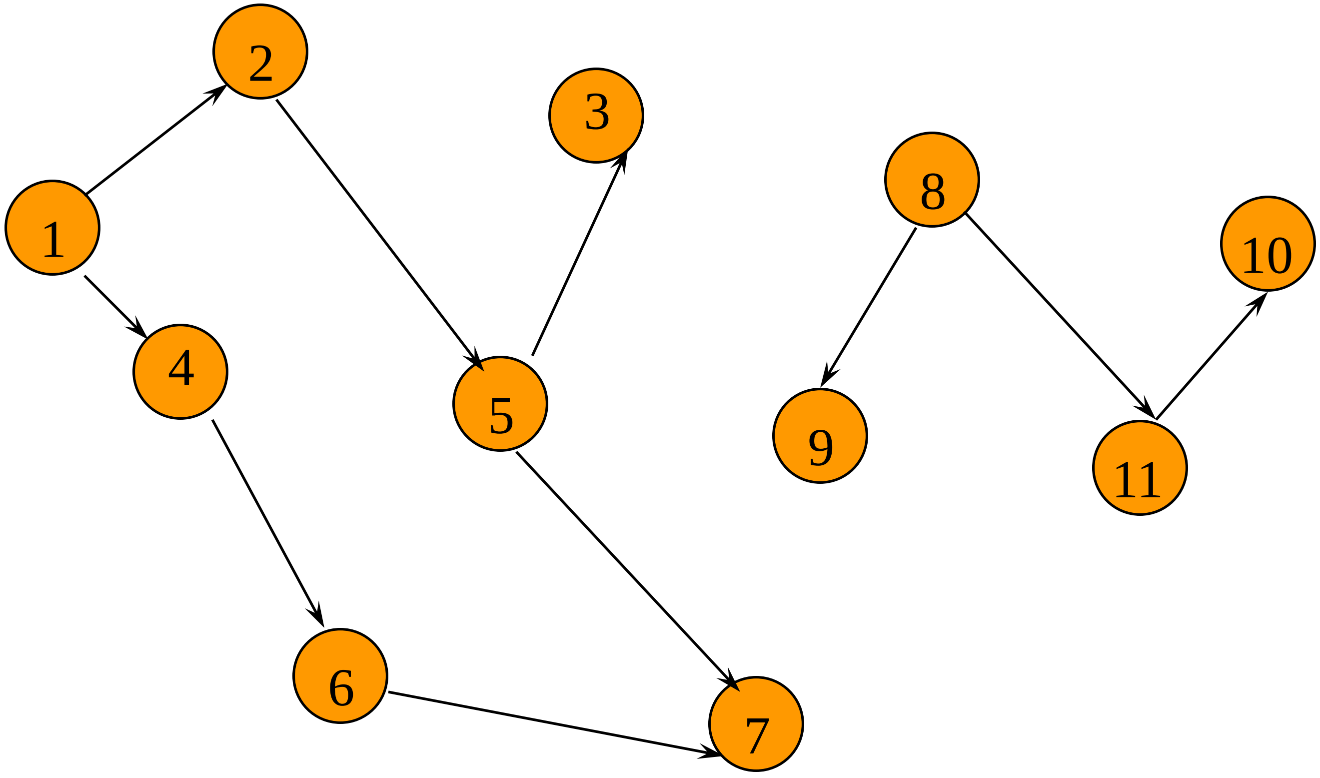
  u ——— v

- Undirected graph => no oriented edge.
- Directed graph => every edge has an orientation.

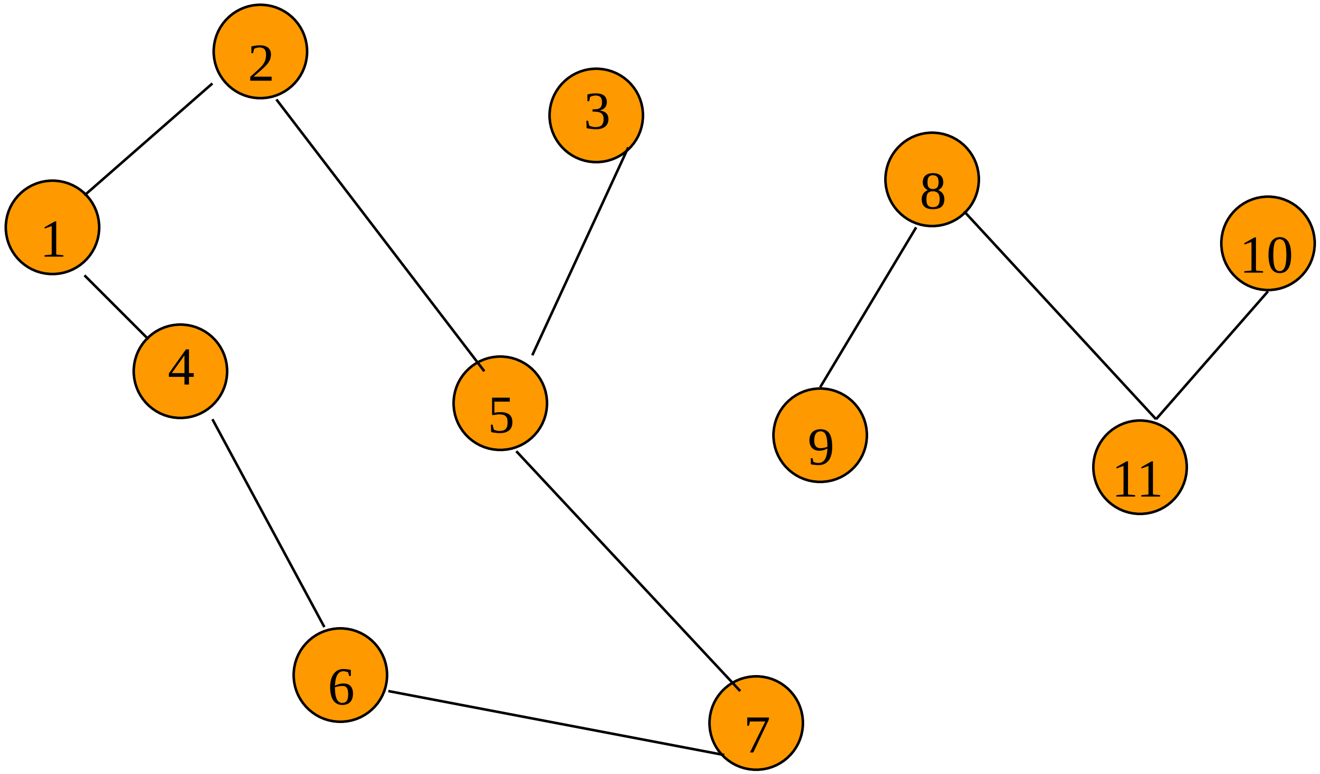
Undirected Graph



Directed Graph (Digraph)

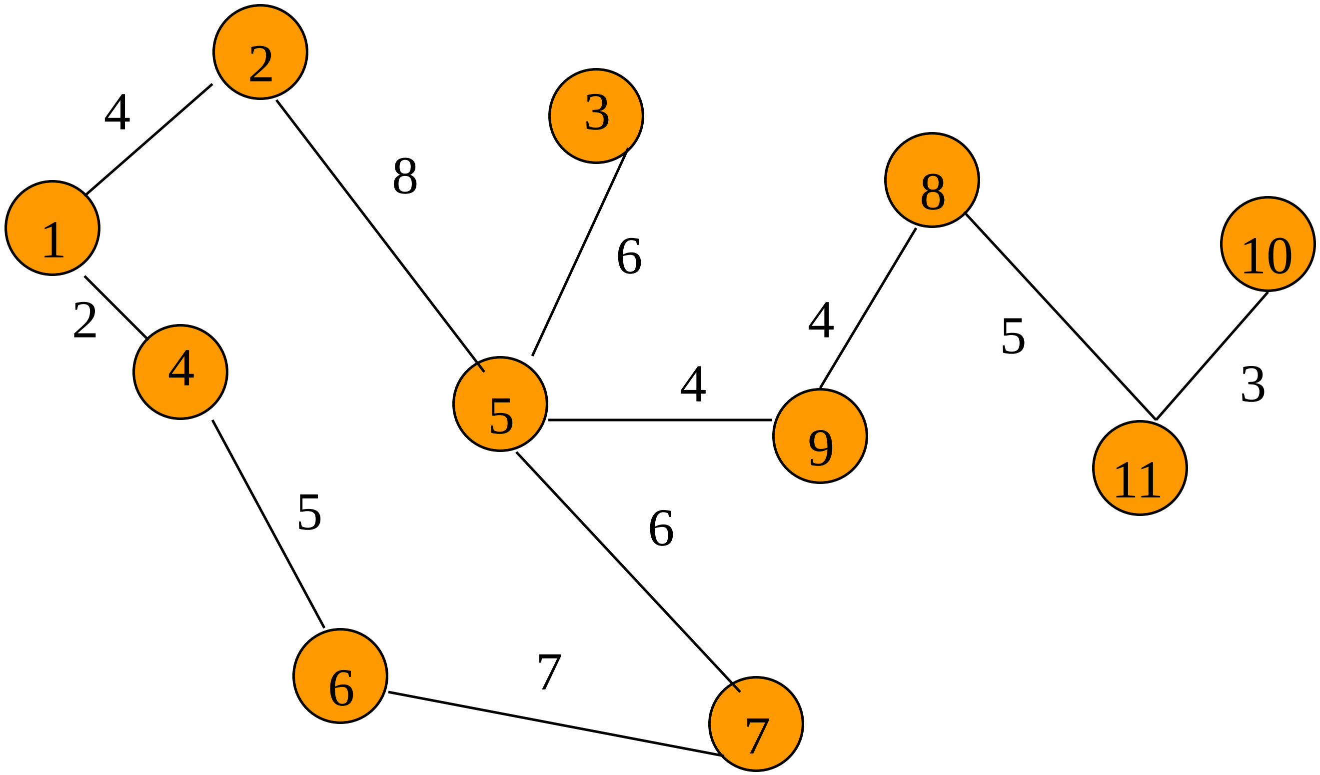


Applications—Communication Network



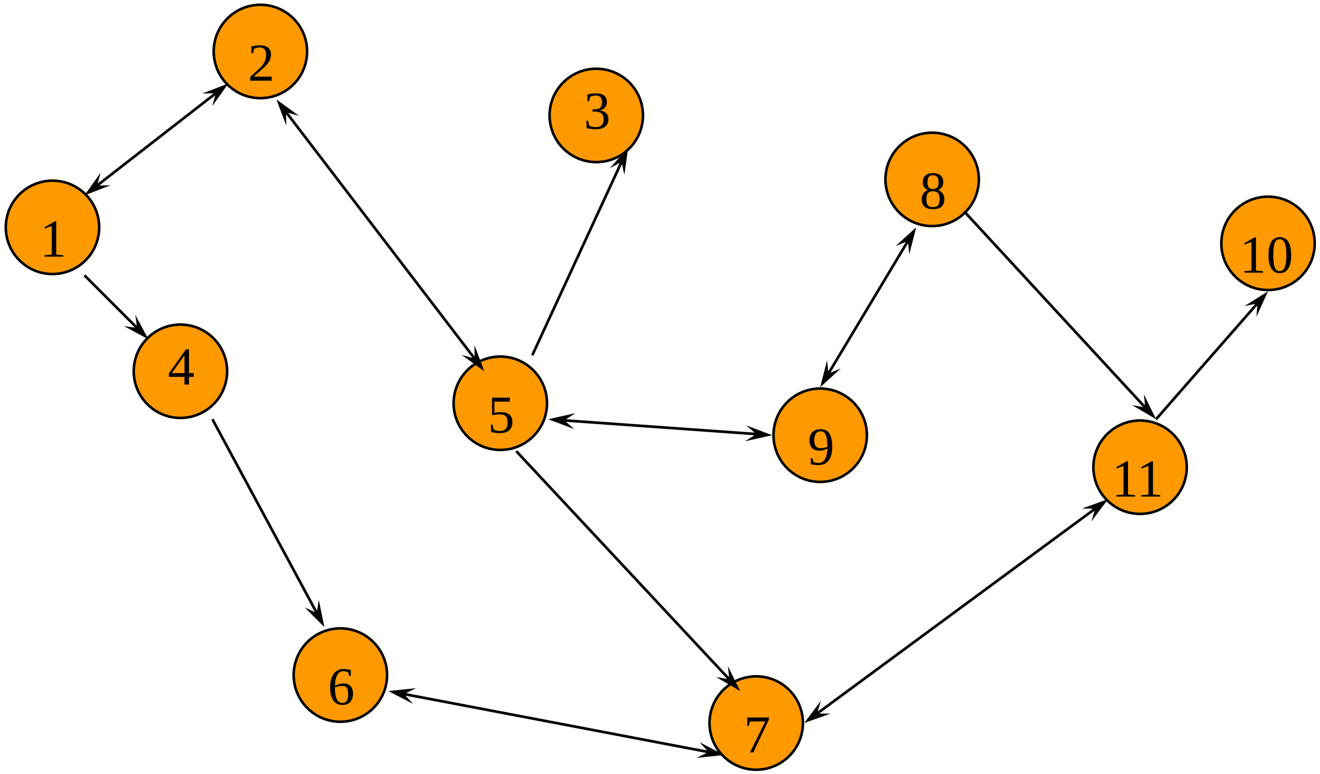
- Vertex = city, edge = communication link.

Driving Distance/Time Map



- Vertex = city, edge weight = driving distance/time.

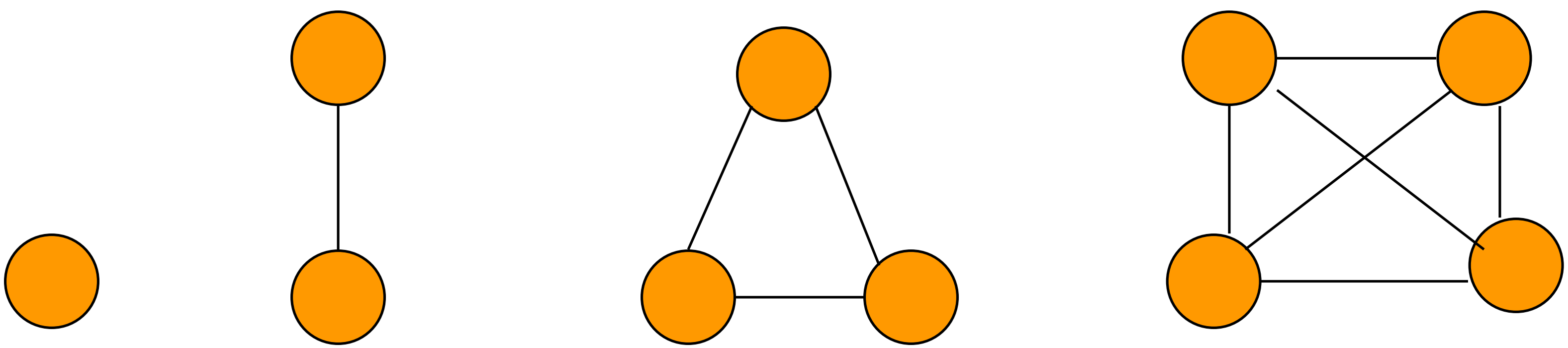
Street Map



- Some streets are one way.

Complete Undirected Graph

Has all possible edges.



n = 1 n = 2 n = 3 n = 4

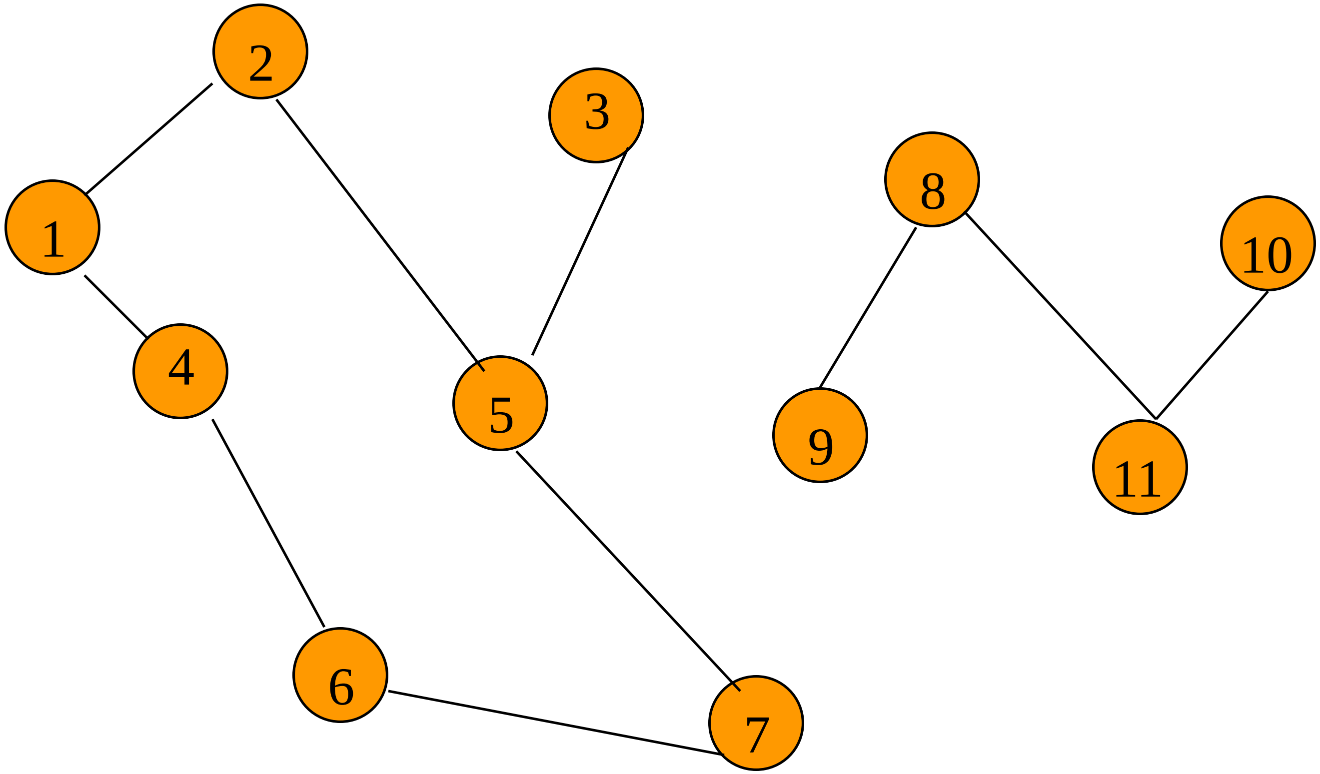
Number Of Edges—Undirected Graph

- 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 <= n(n-1)/2.

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 <= n(n-1).

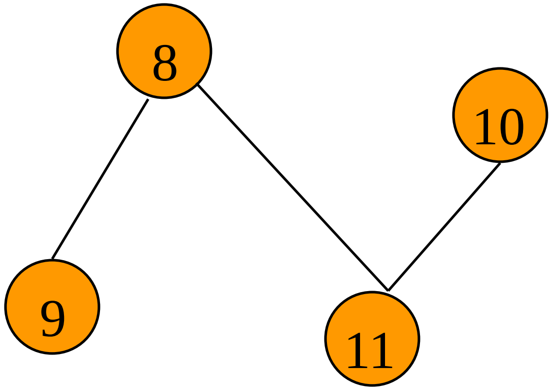
Vertex Degree



Number of edges incident to vertex.

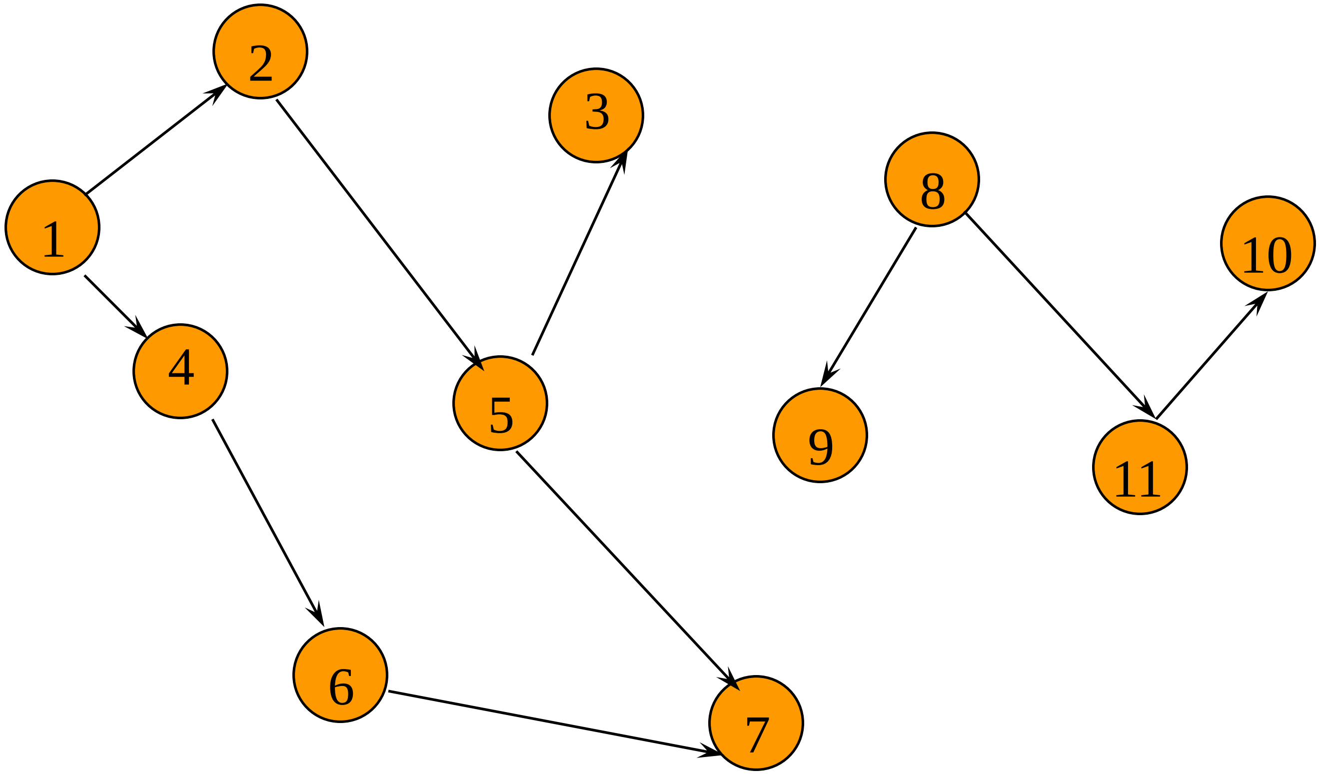
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



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

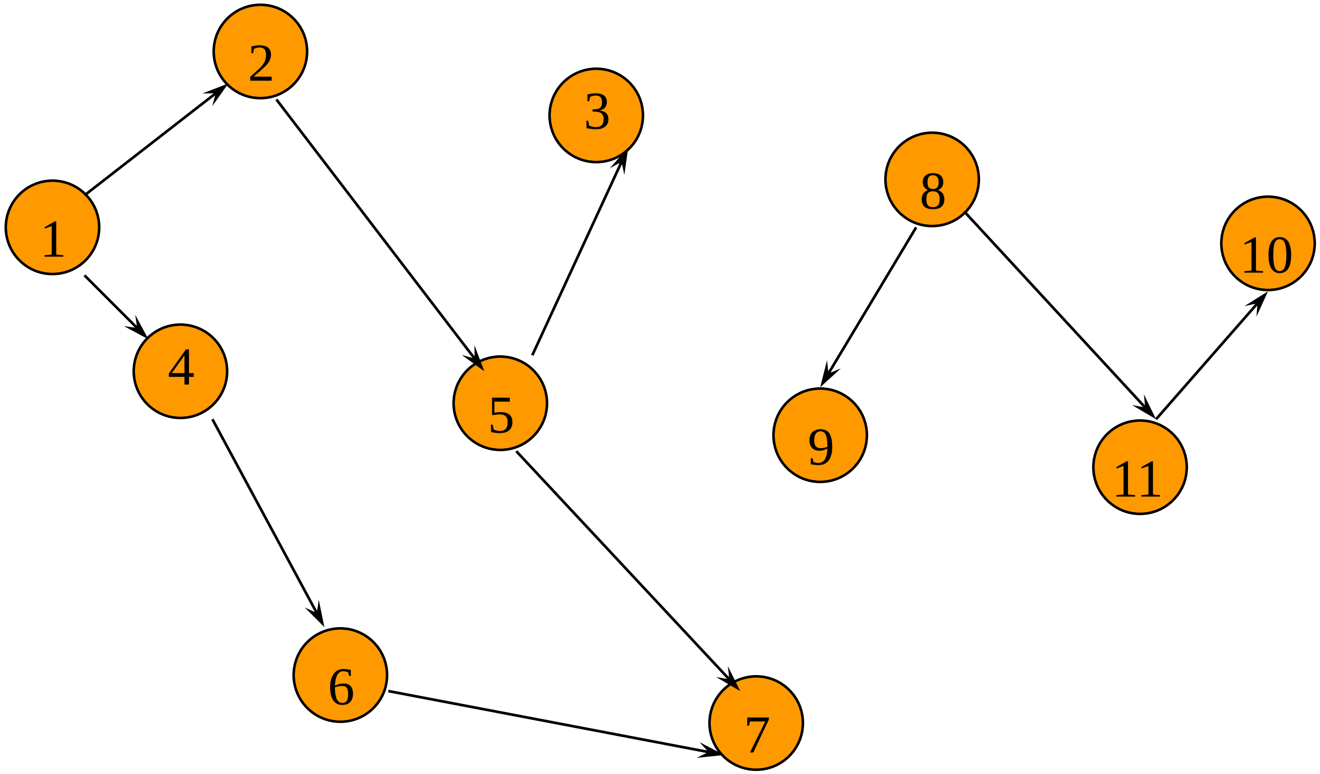
In-Degree Of A Vertex



in-degree is number of incoming edges

indegree(2) = 1, indegree(8) = 0

Out-Degree Of A Vertex



out-degree is number of outbound edges

outdegree(2) = 1, outdegree(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 = e,

where e is the number of edges in the digraph

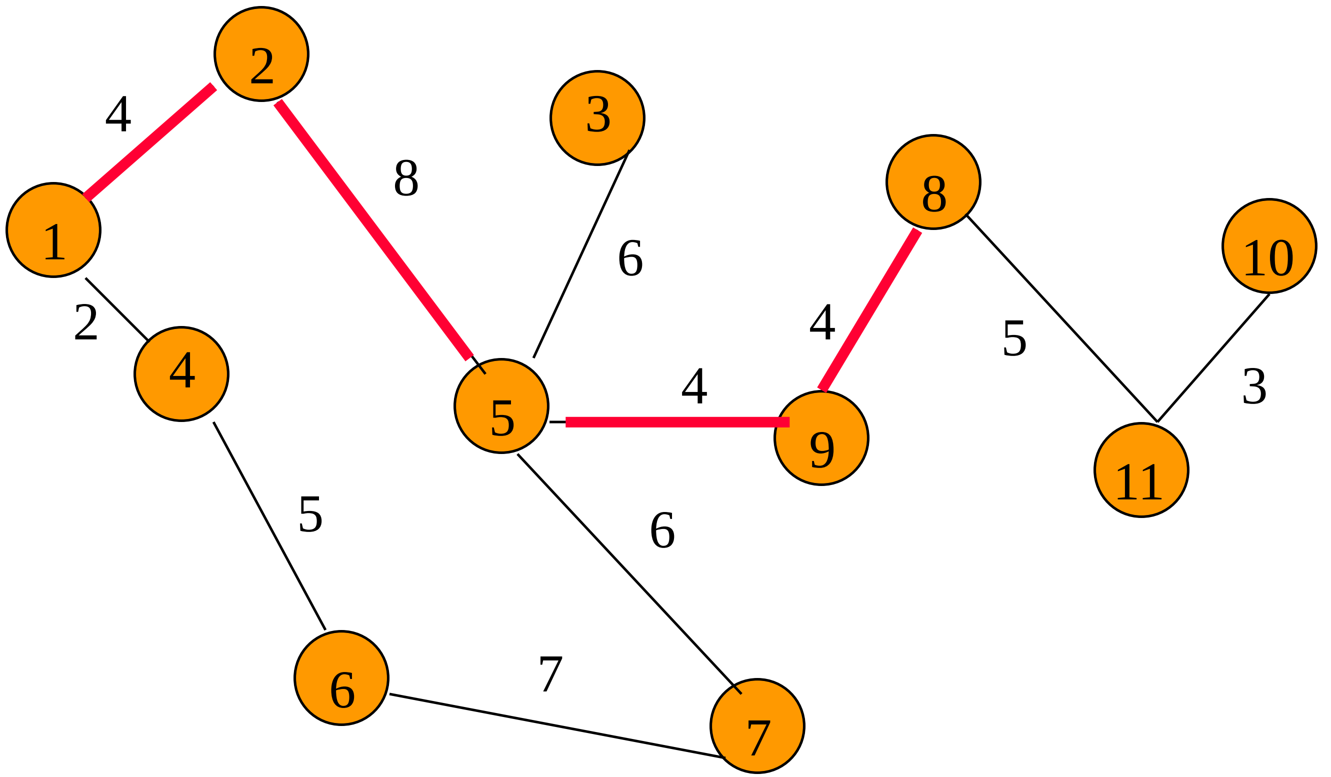
Graph Operations And Representation

Sample Graph Problems

- Path problems.
- Connectedness problems.
- Spanning tree problems.

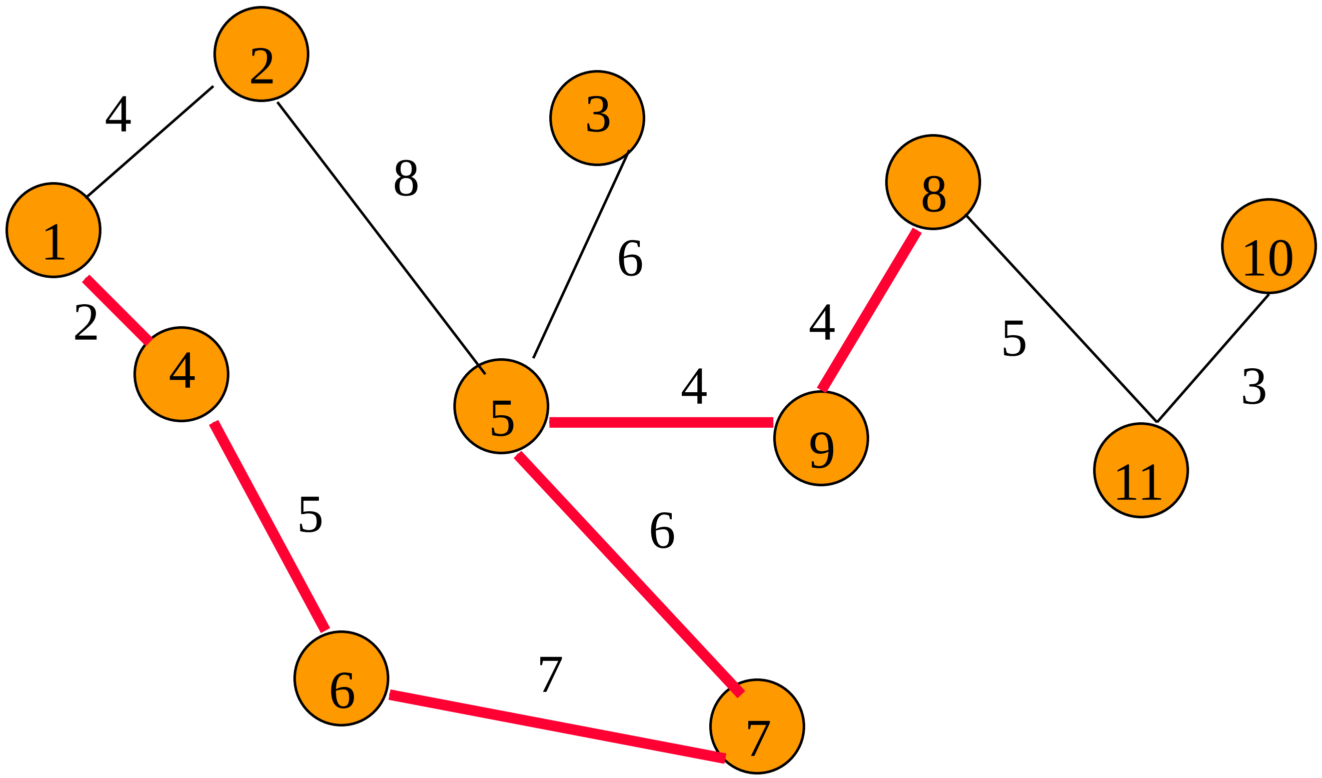
Path Finding

Path between 1 and 8.



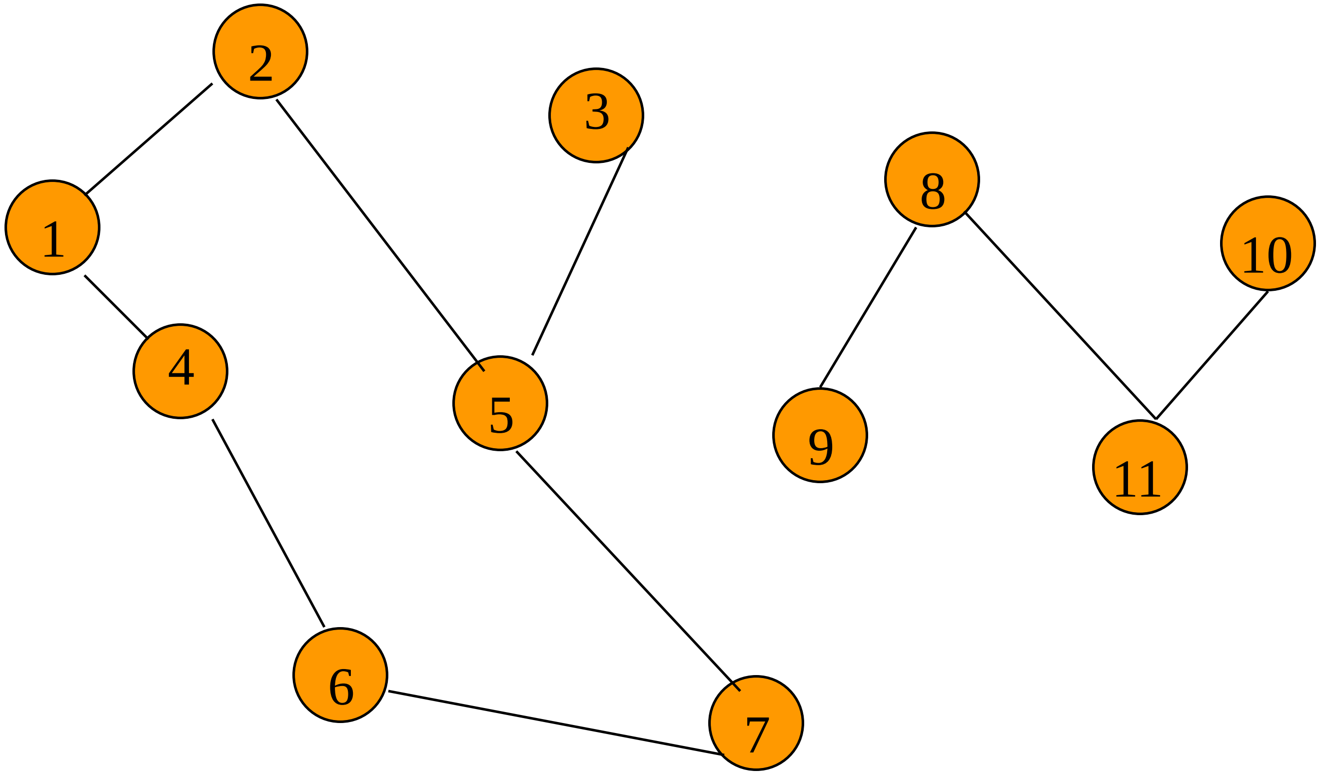
Path length is 20.

Another Path Between 1 and 8



Path length is 28.

Example Of No Path

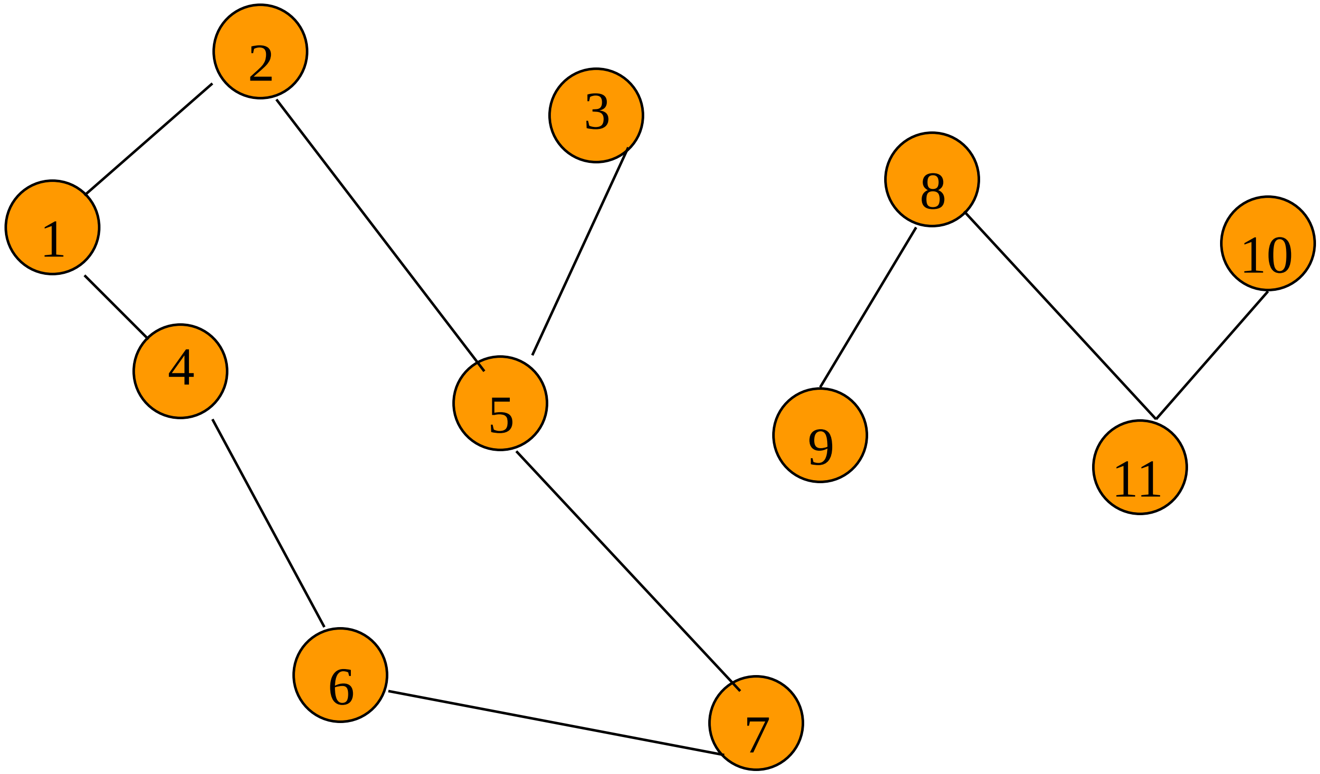


No path between 2 and 9.

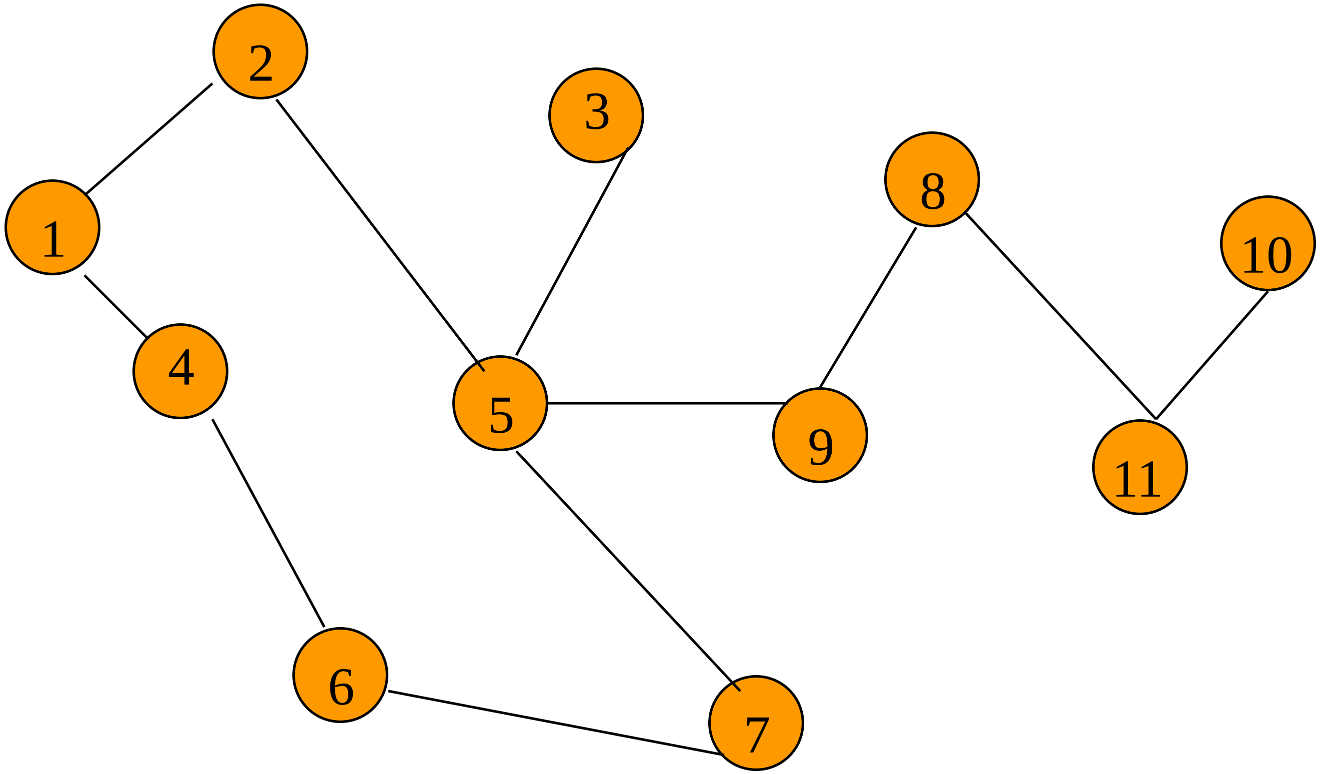
Connected Graph

- Undirected graph.
- There is a path between every pair of vertices.

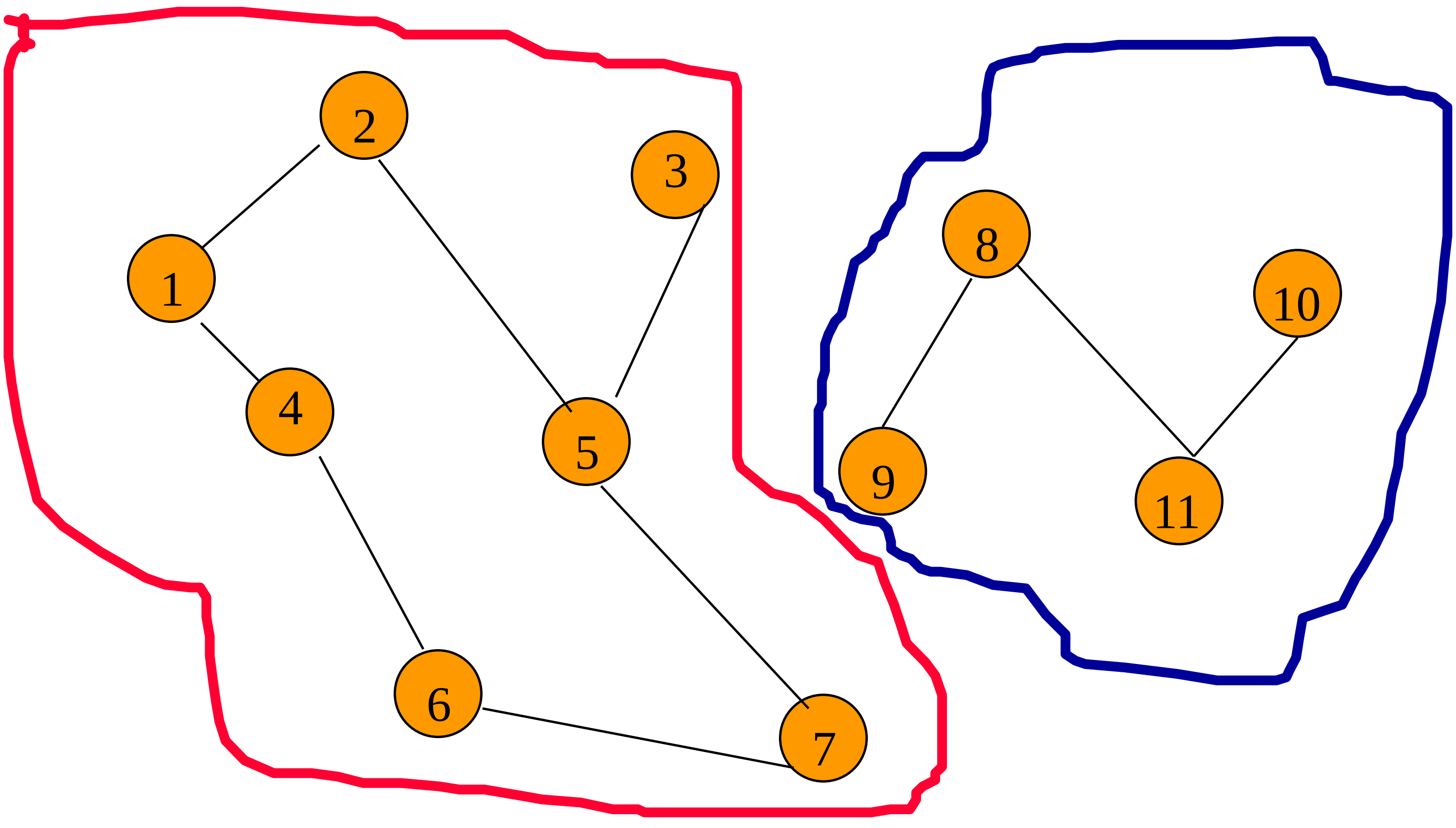
Example Of Not Connected



Connected Graph Example



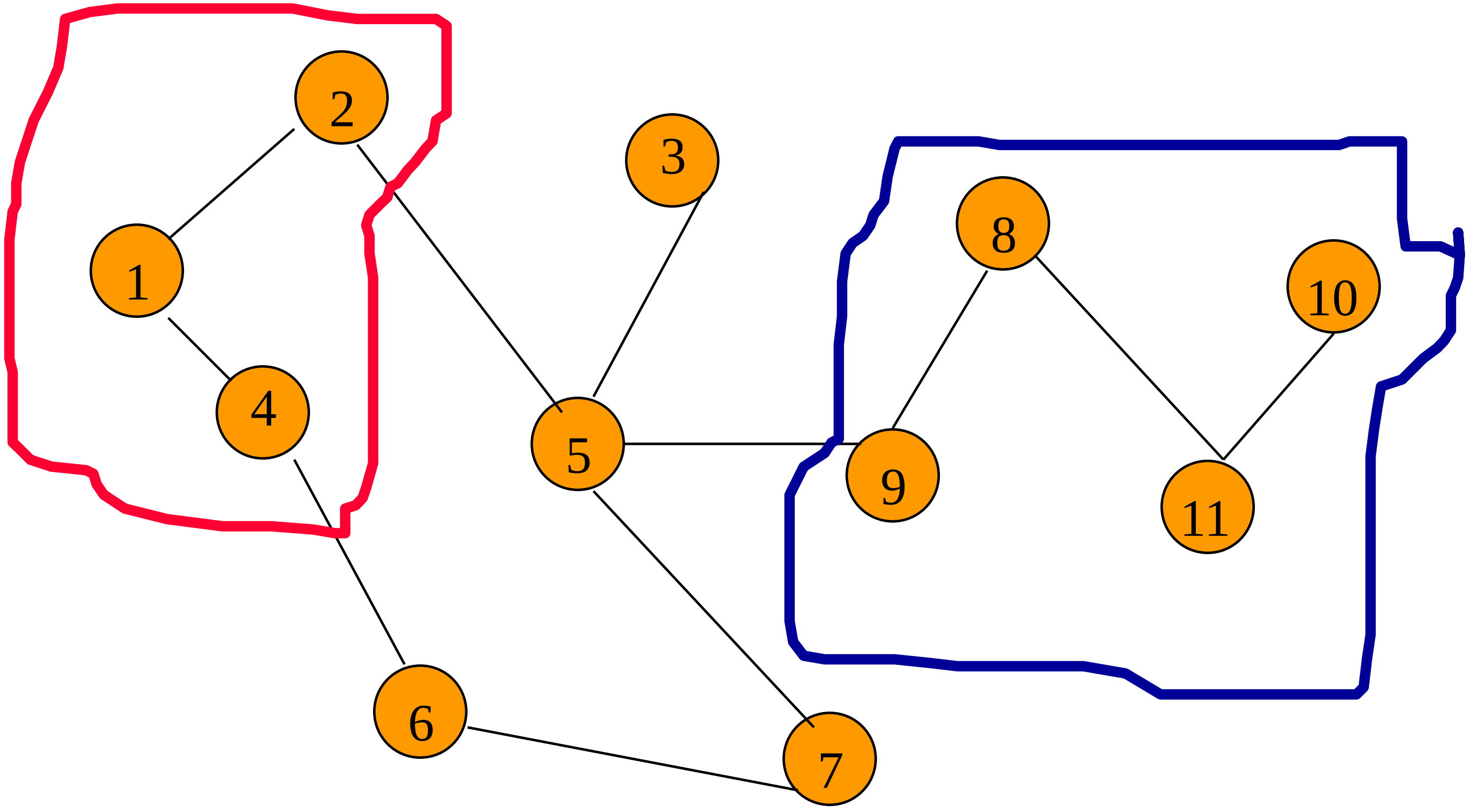
Connected Components



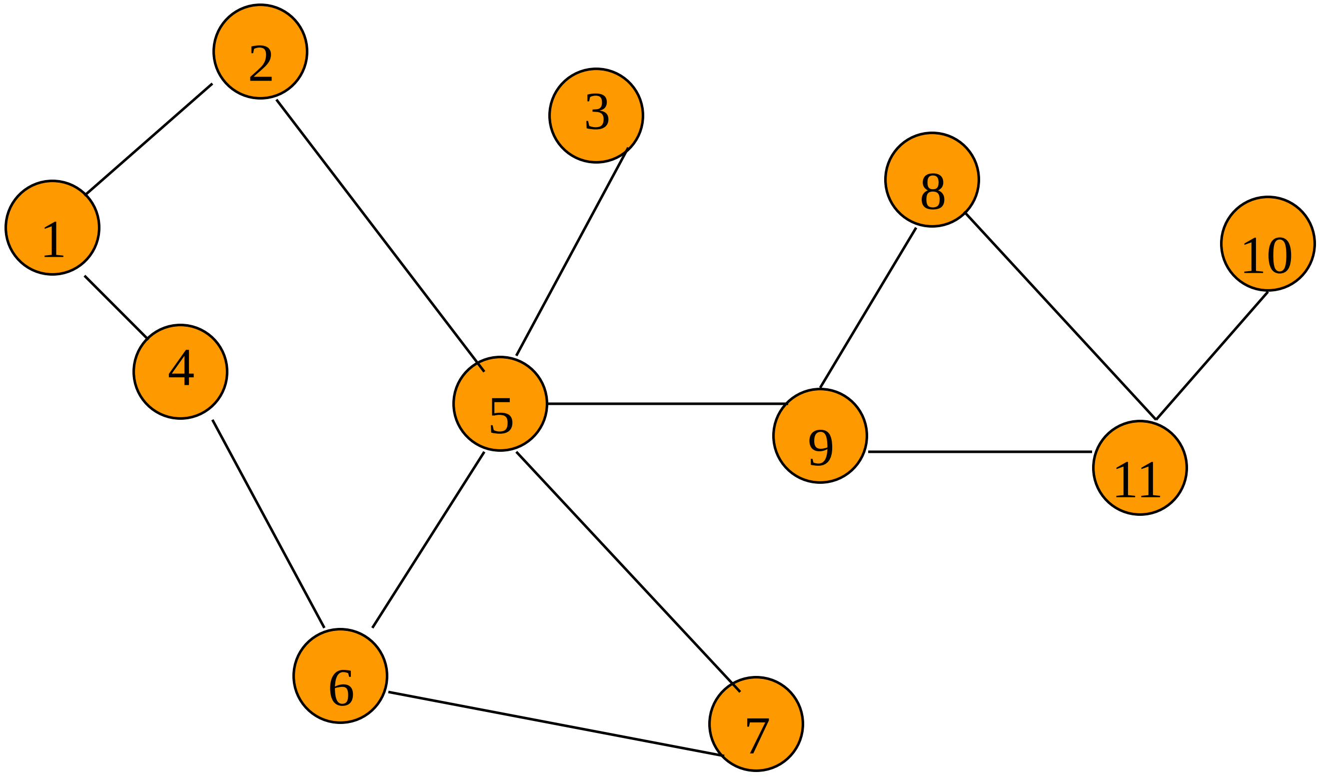
Connected Component

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

Not A Component



Communication Network

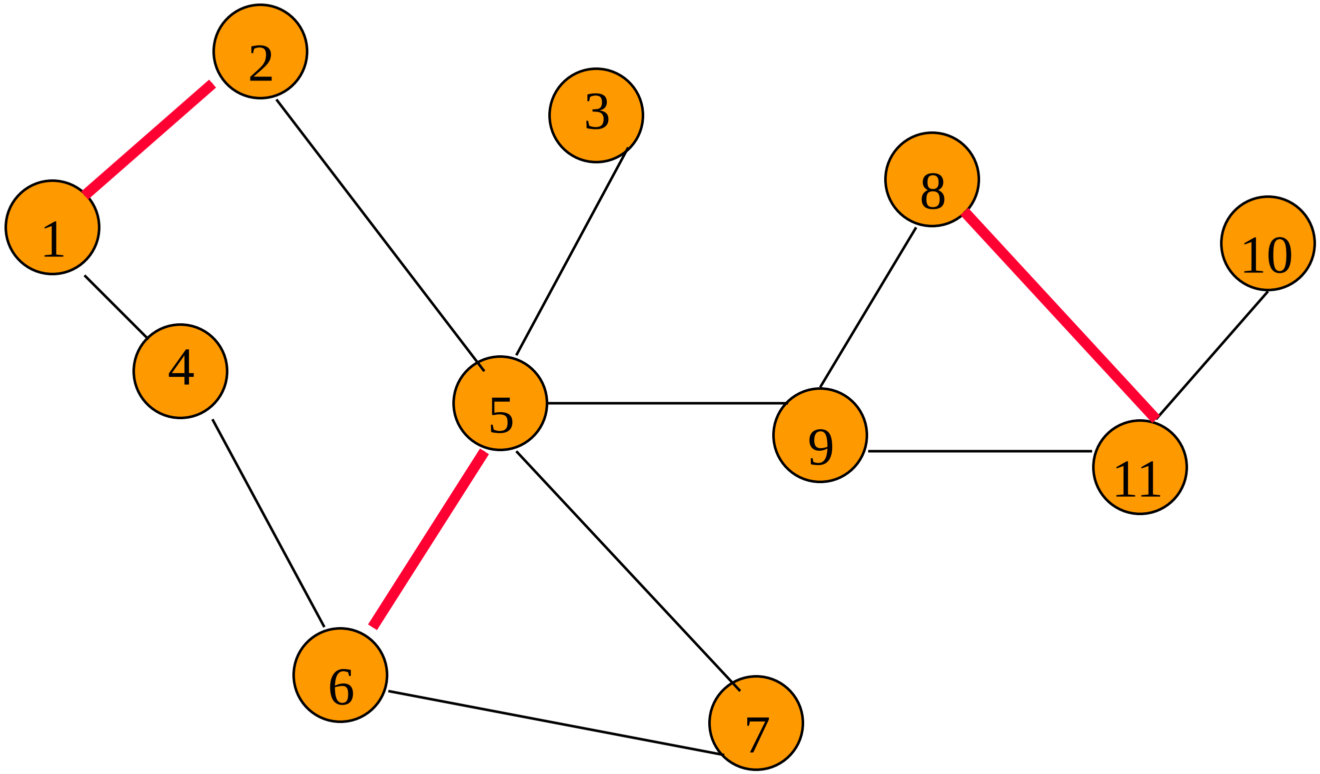


Each edge is a link that can be constructed (i.e., a feasible link).

Communication Network Problems

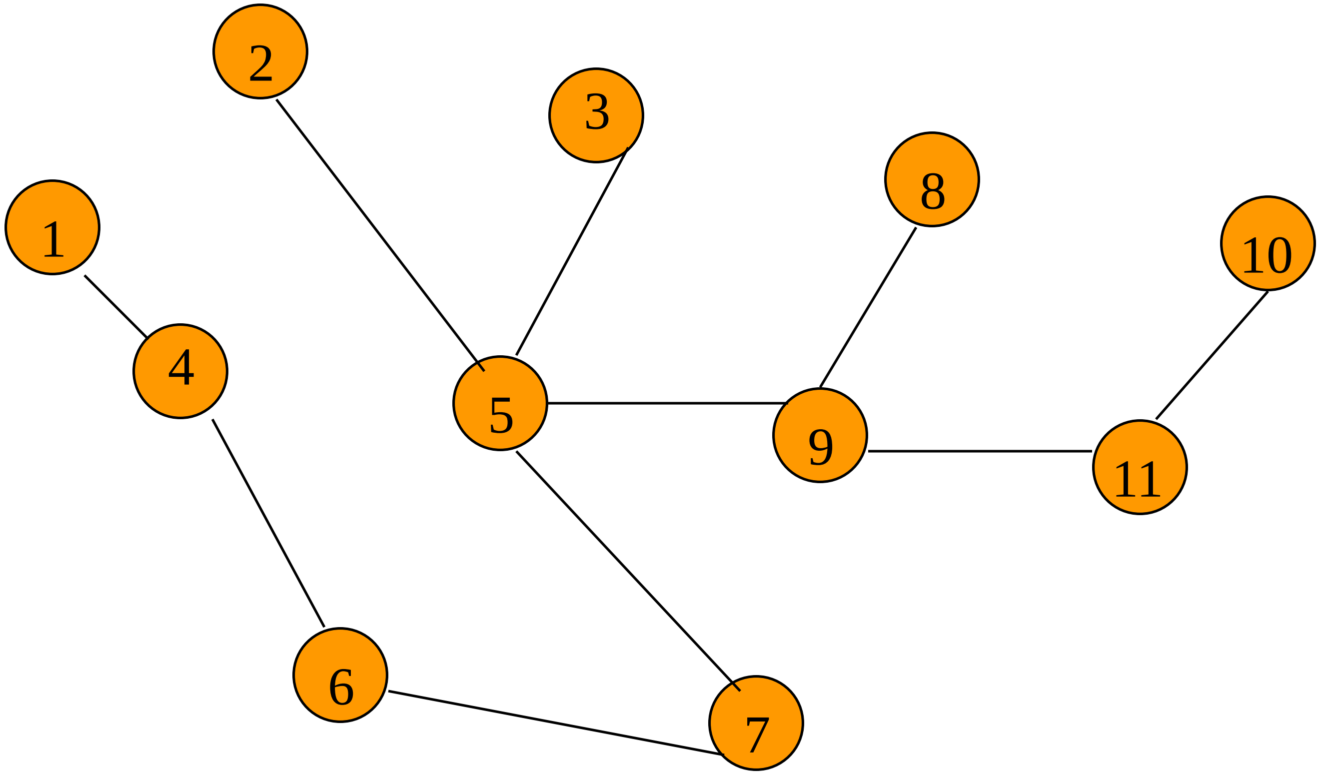
- Is the network connected?
  - Can we communicate between every pair of cities?
- Find the components.
- Want to construct smallest number of feasible links so that resulting network is connected.

Cycles And Connectedness



Removal of an edge that is on a cycle does not affect connectedness.

Cycles And Connectedness



Connected subgraph with all vertices and minimum number of edges has no cycles.

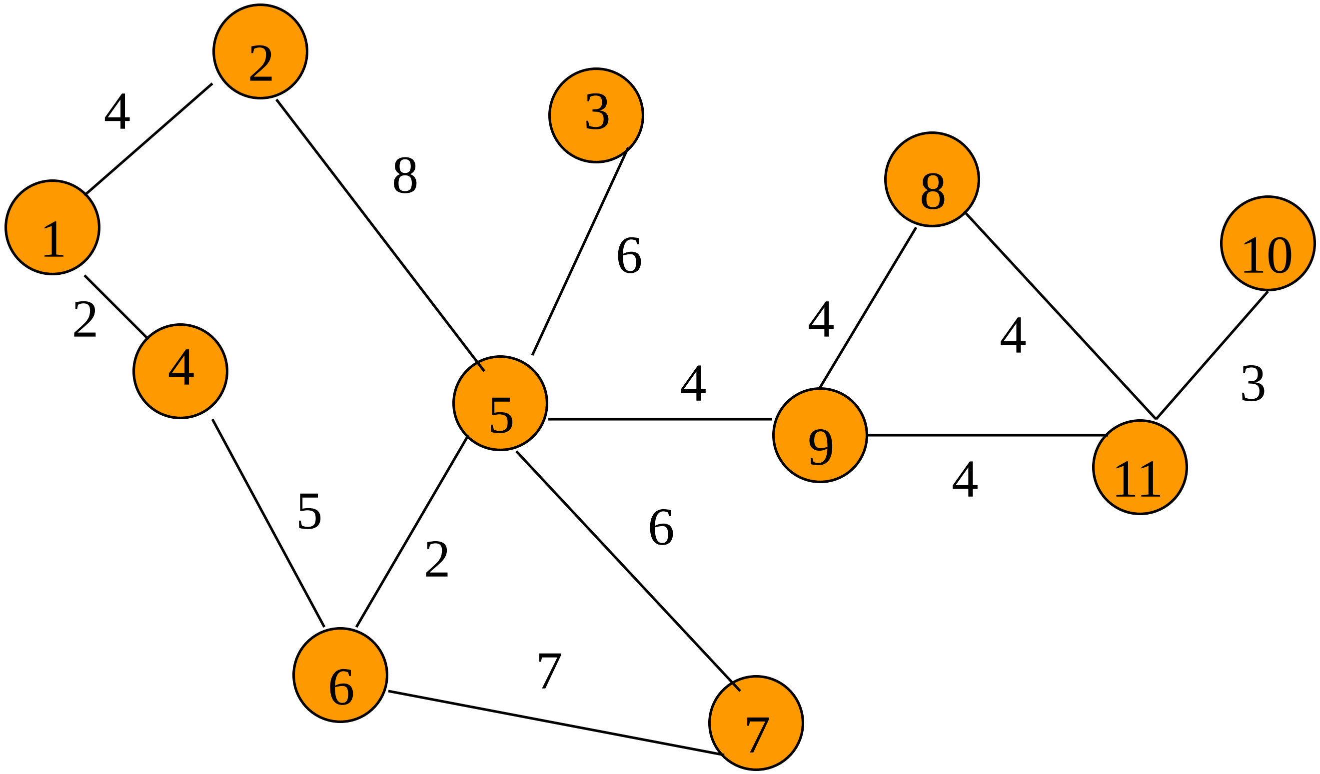
Tree

- Connected graph that has no cycles.
- n vertex connected graph with n-1 edges.

Spanning Tree

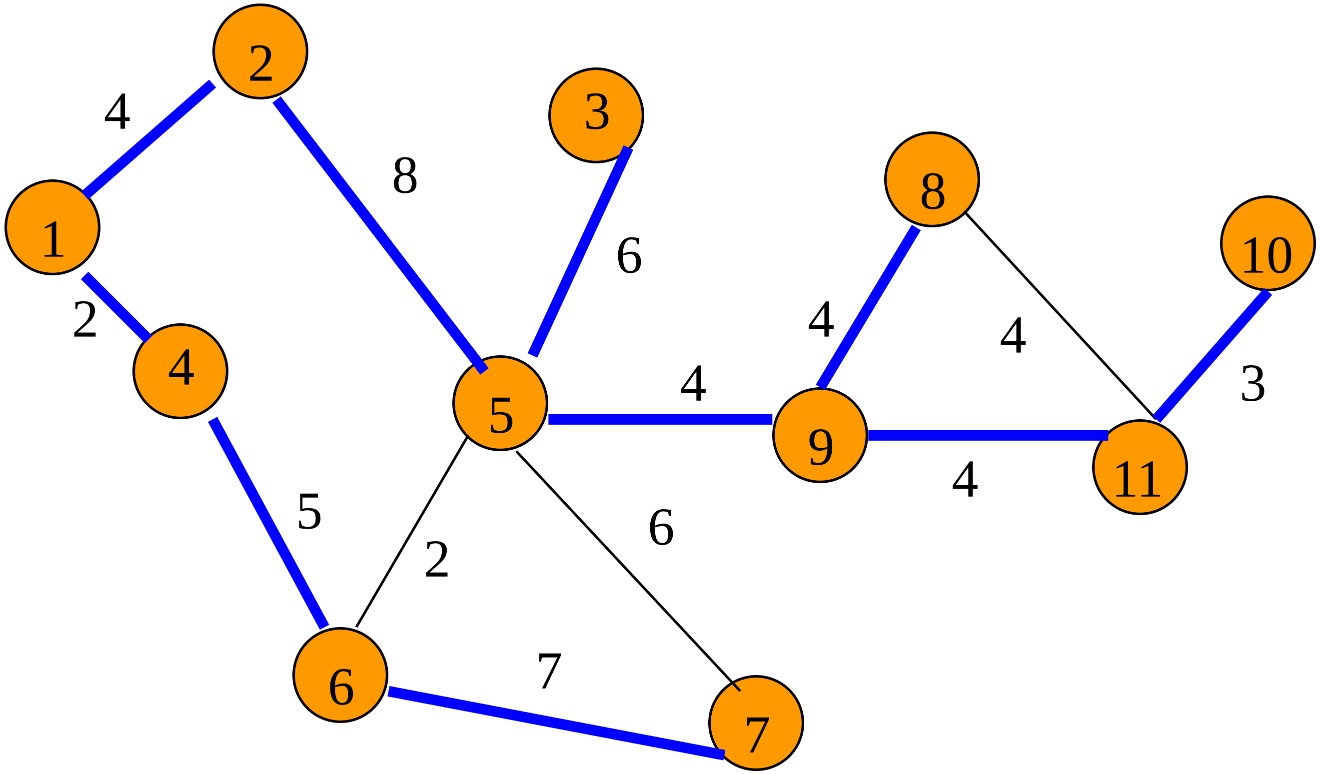
- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

Minimum Cost Spanning Tree



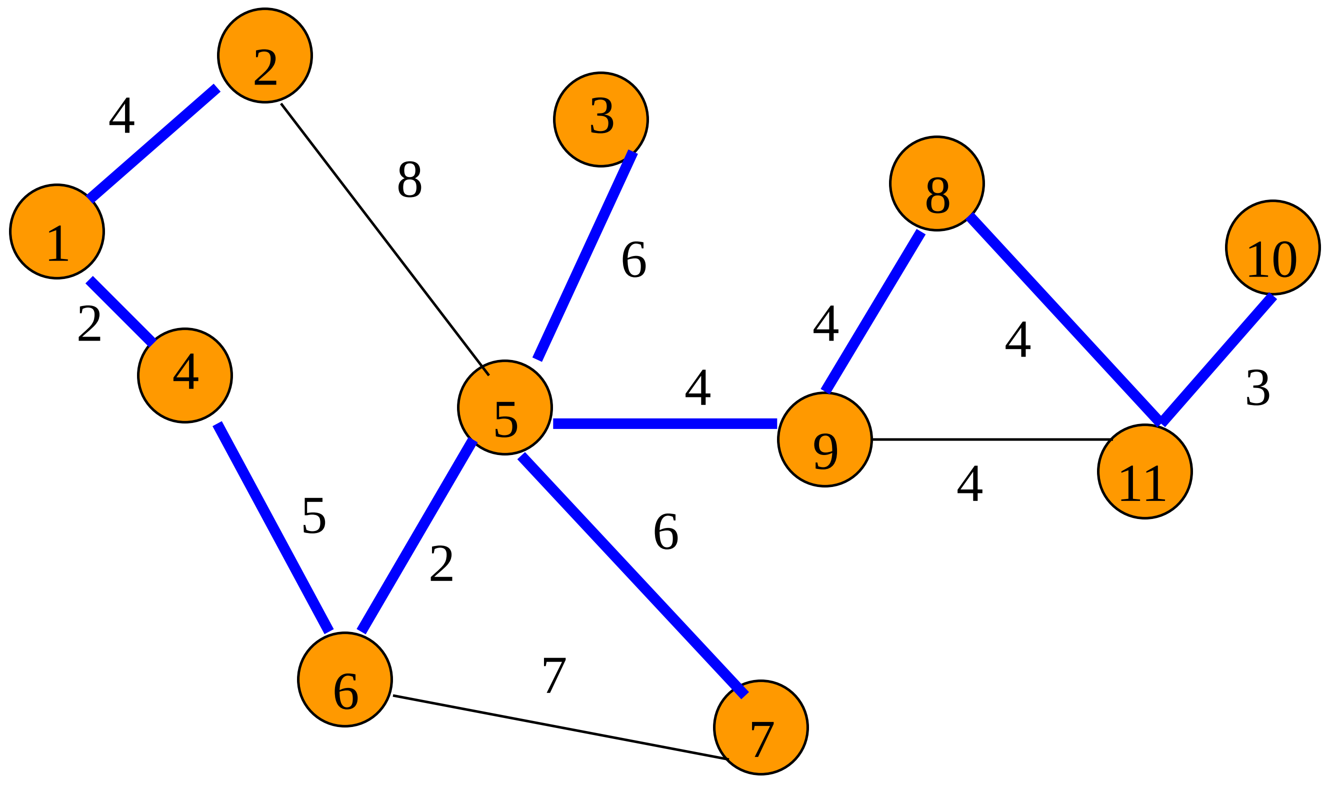
- Tree cost is sum of edge weights/costs.

A Spanning Tree



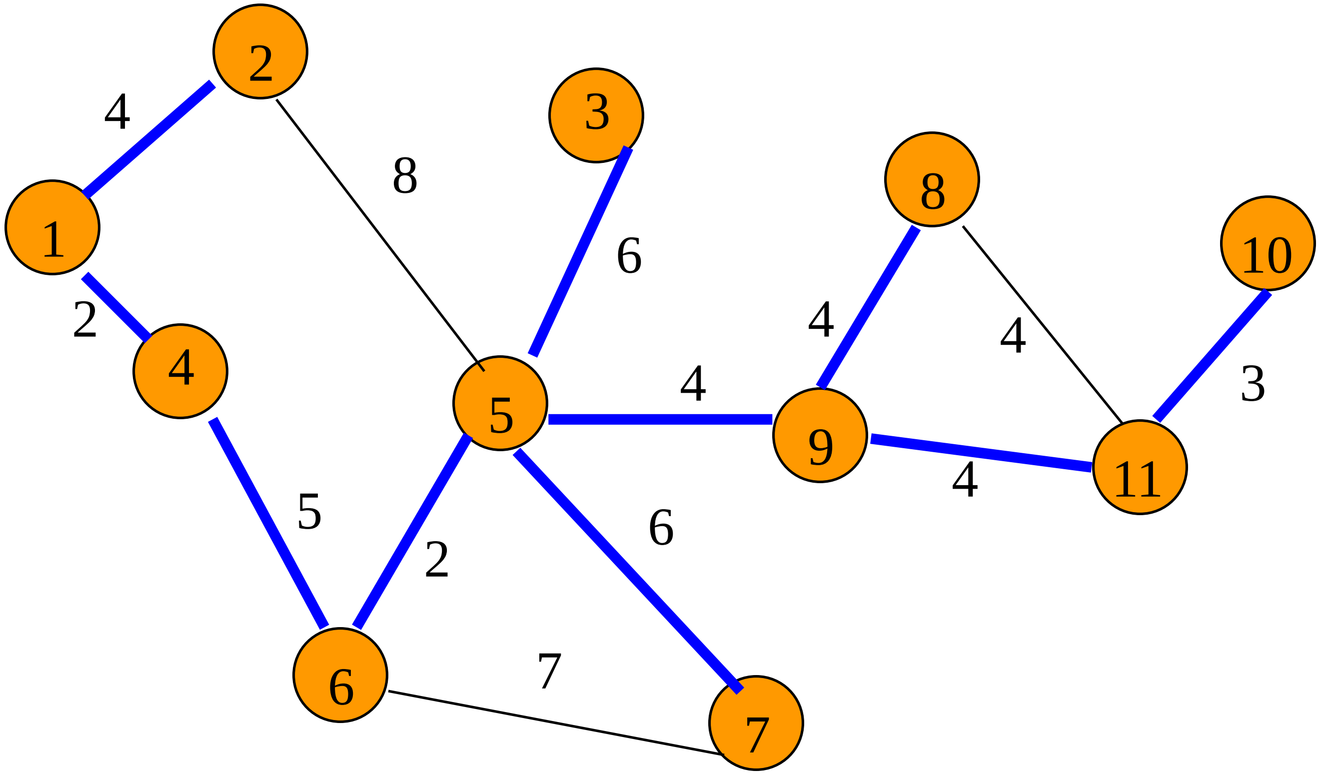
Spanning tree cost = 47.

(**Minimum Cost Spanning Tree



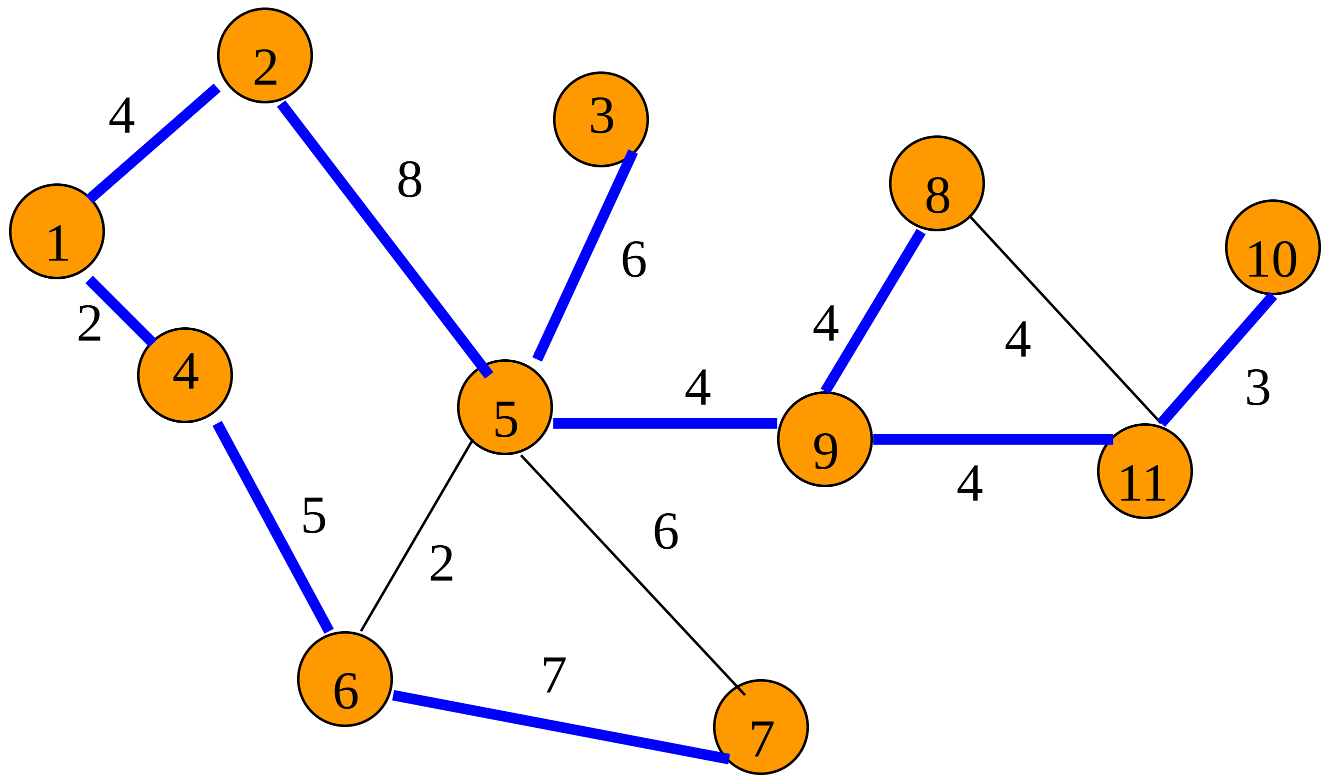
Spanning tree cost = 40.

Another MST



Spanning tree cost = 40.

A Wireless Broadcast Tree
(** Not an MST **)



Source = 1, weights = needed power.

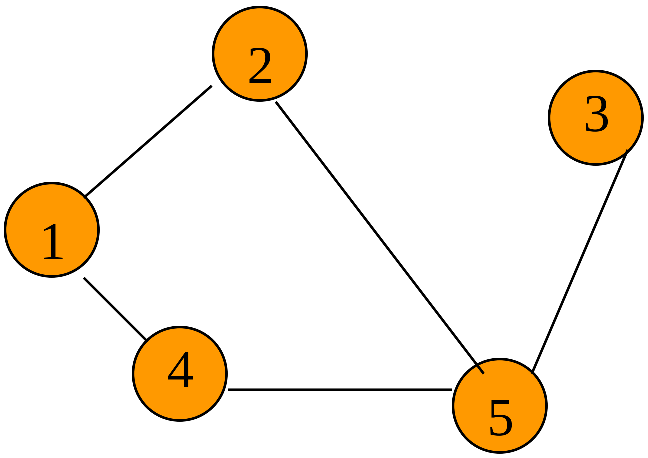
Cost = 4 + 8 + 2 + 5 + 6 + 7 + 4 + 4 +4 + 3 = 47.

Graph Representation

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

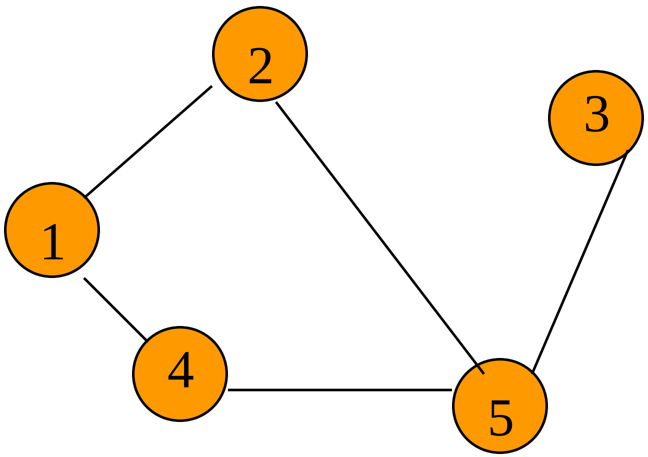
Adjacency Matrix

- 0/1 n x 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

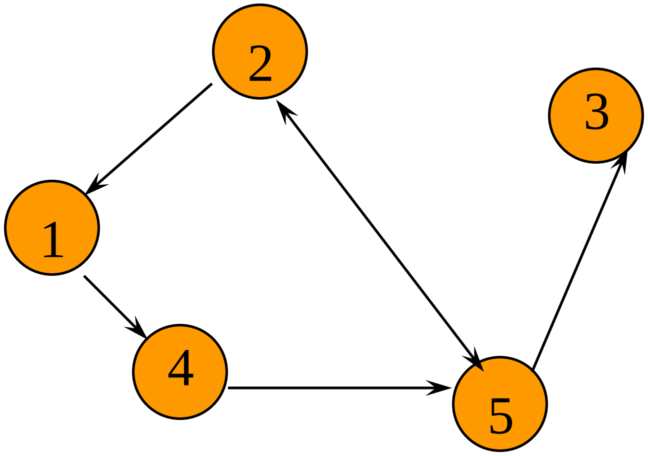
Adjacency Matrix Properties



	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

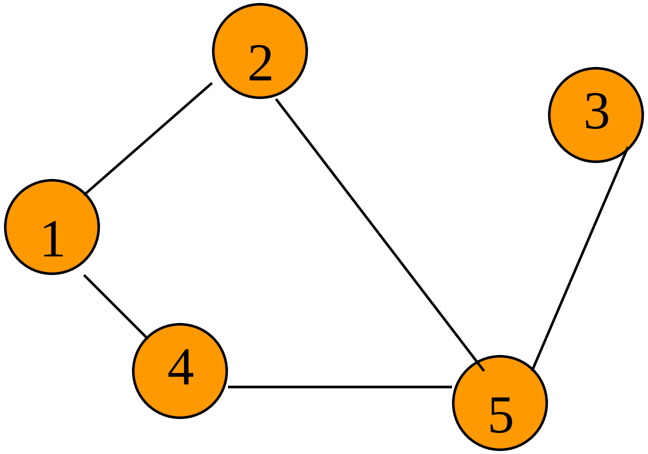
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

Adjacency Matrix

- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ 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)

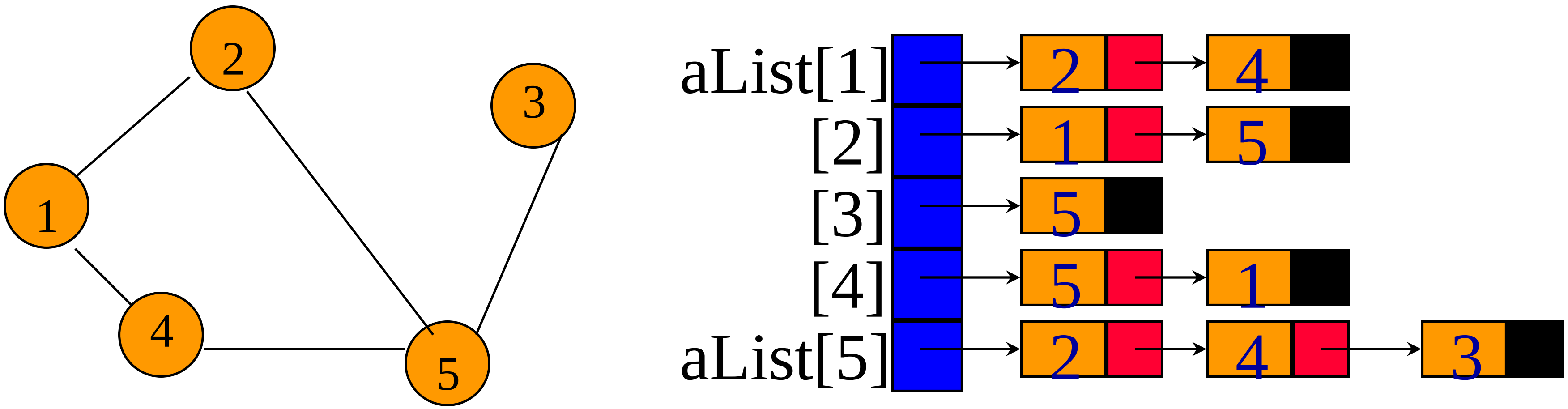
aList[3] = (5)

aList[4] = (5,1)

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

Linked Adjacency Lists

- Each adjacency list is a chain.



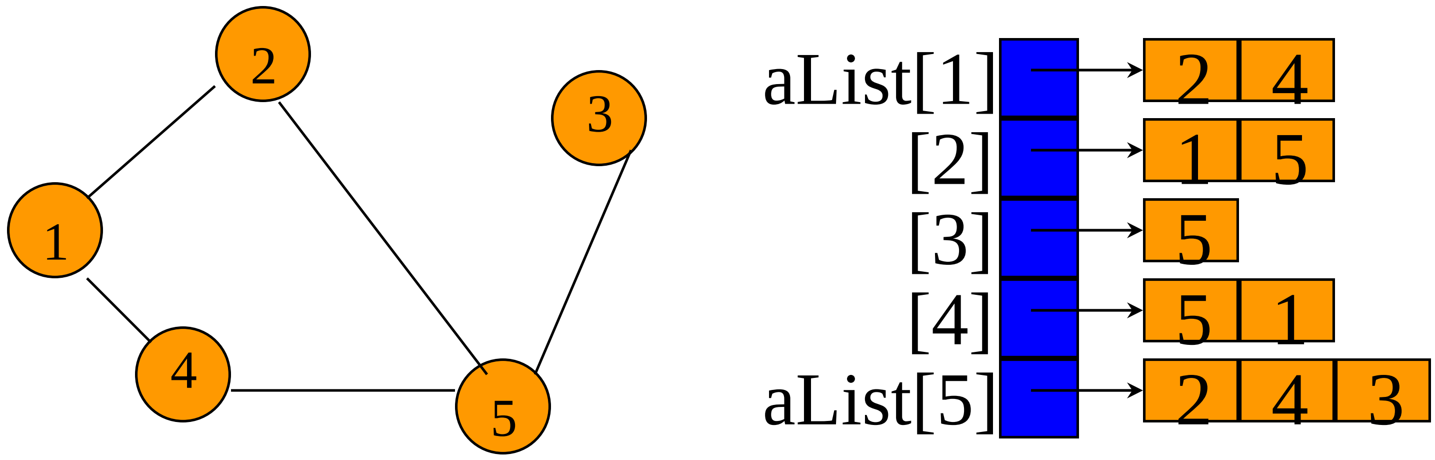
Array Length = n

# 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)

Weighted Graphs

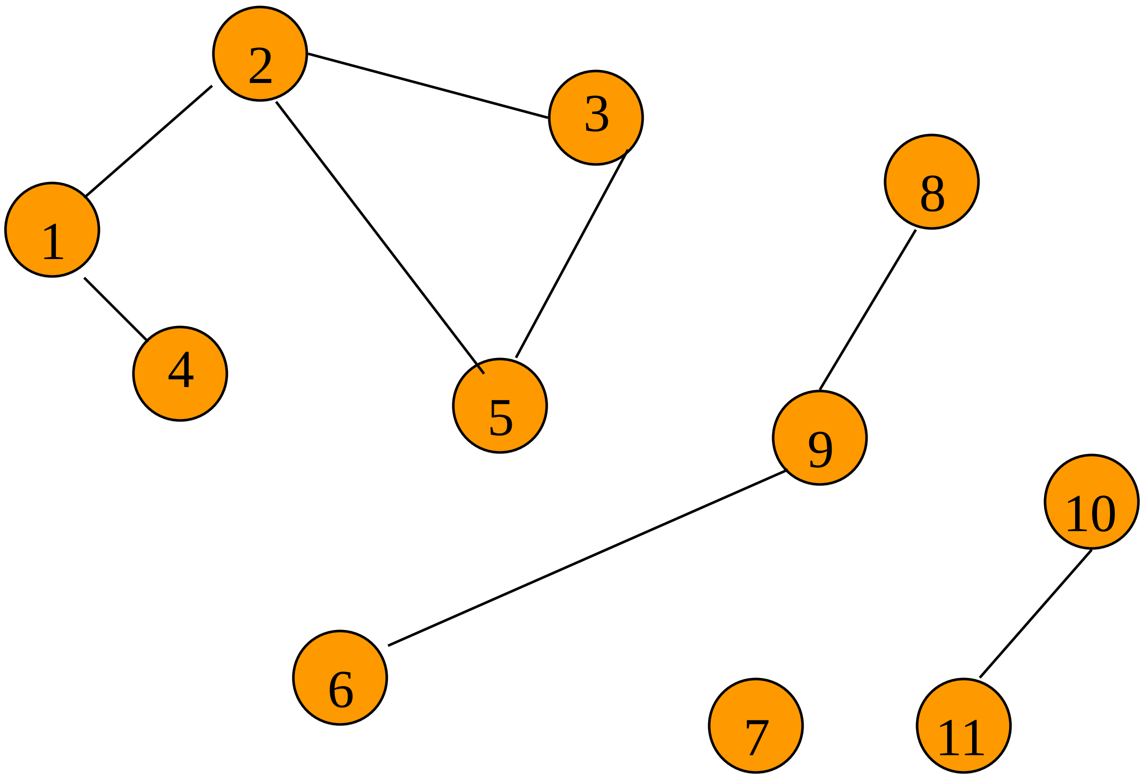
- Cost adjacency matrix.
  - C(i,j) = cost of edge (i,j)
- Adjacency lists => each list element is a pair (adjacent vertex, edge weight)

Number Of C++ Classes Needed

- 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
- 3 x 4 = 12 C++ classes

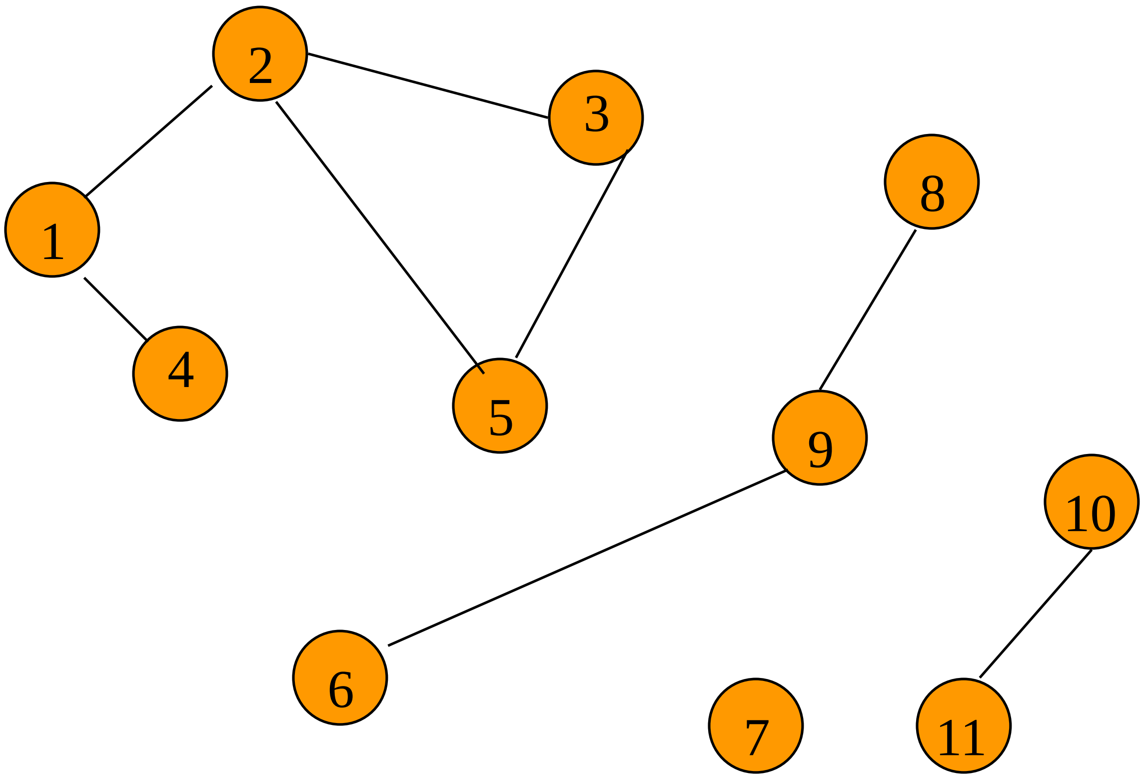
Graph Search Methods

- 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.



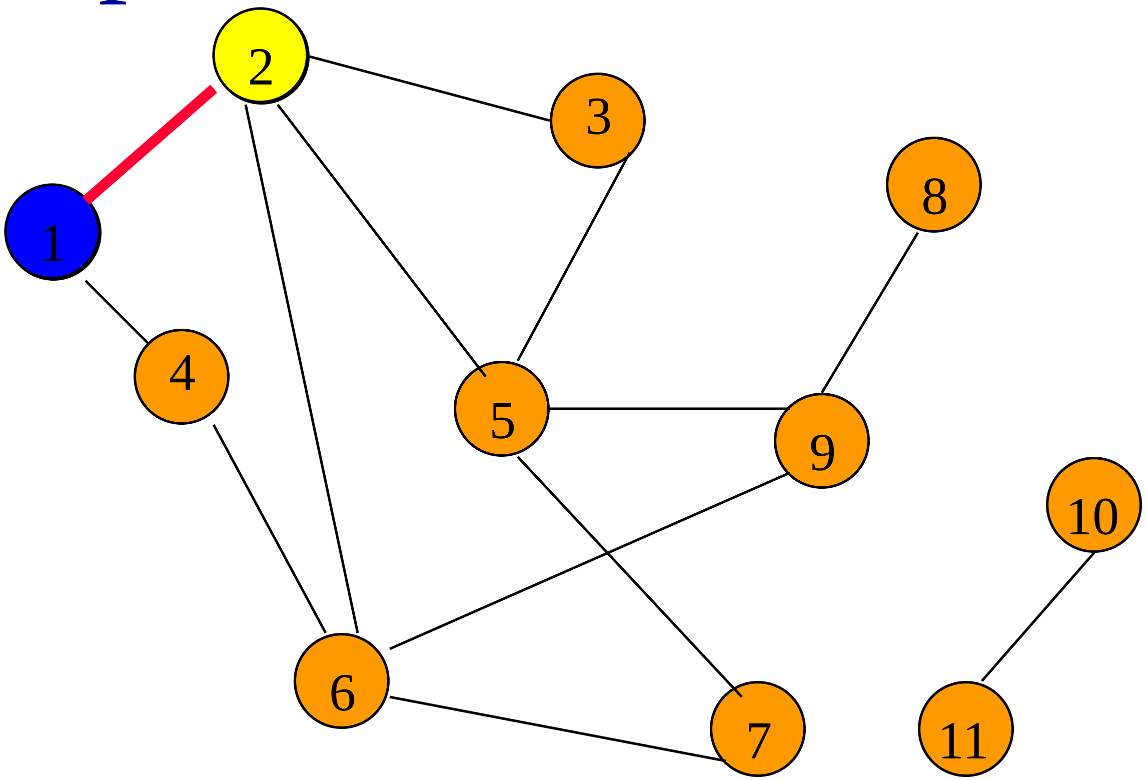
Graph Search Methods

- 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.
  - Breadth-first search.

Depth-First Search

```
DFS(v)
{
  Label vertex v as reached.
  for (each unreached vertex u
                        adjacenct from v)
    DFS(u);
}
```

Depth-First Search Example

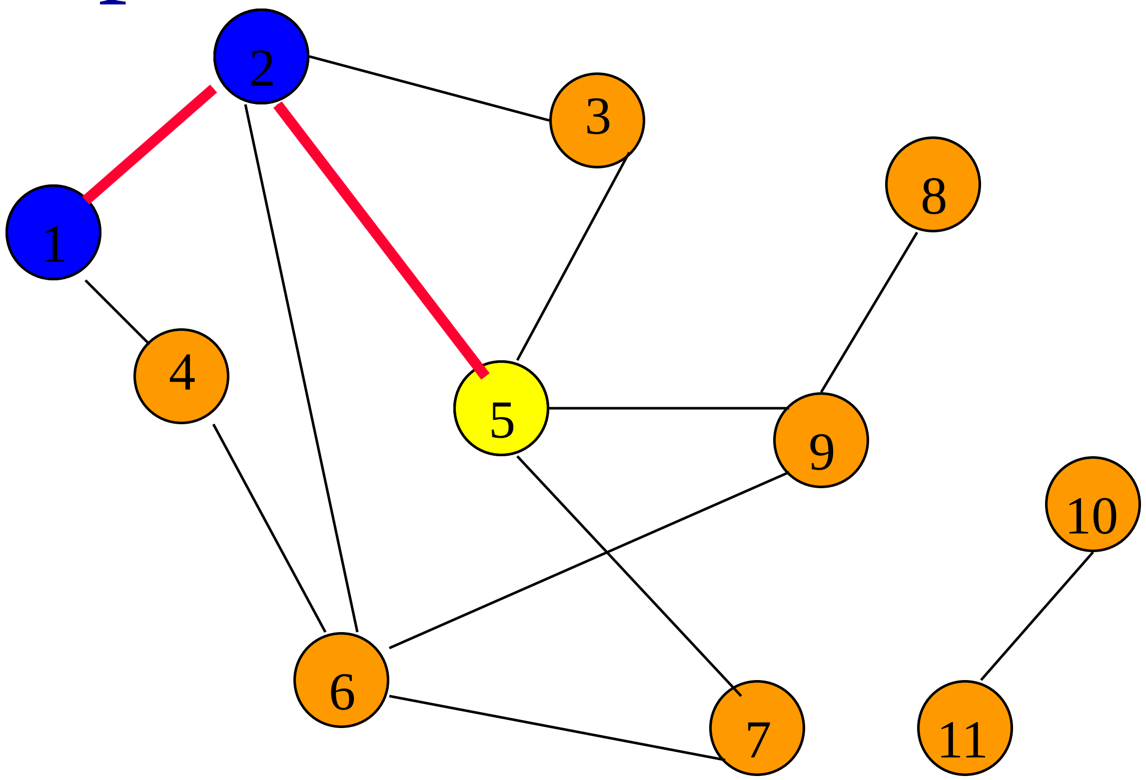


Start search at vertex 1.

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

Suppose that vertex 2 is selected.

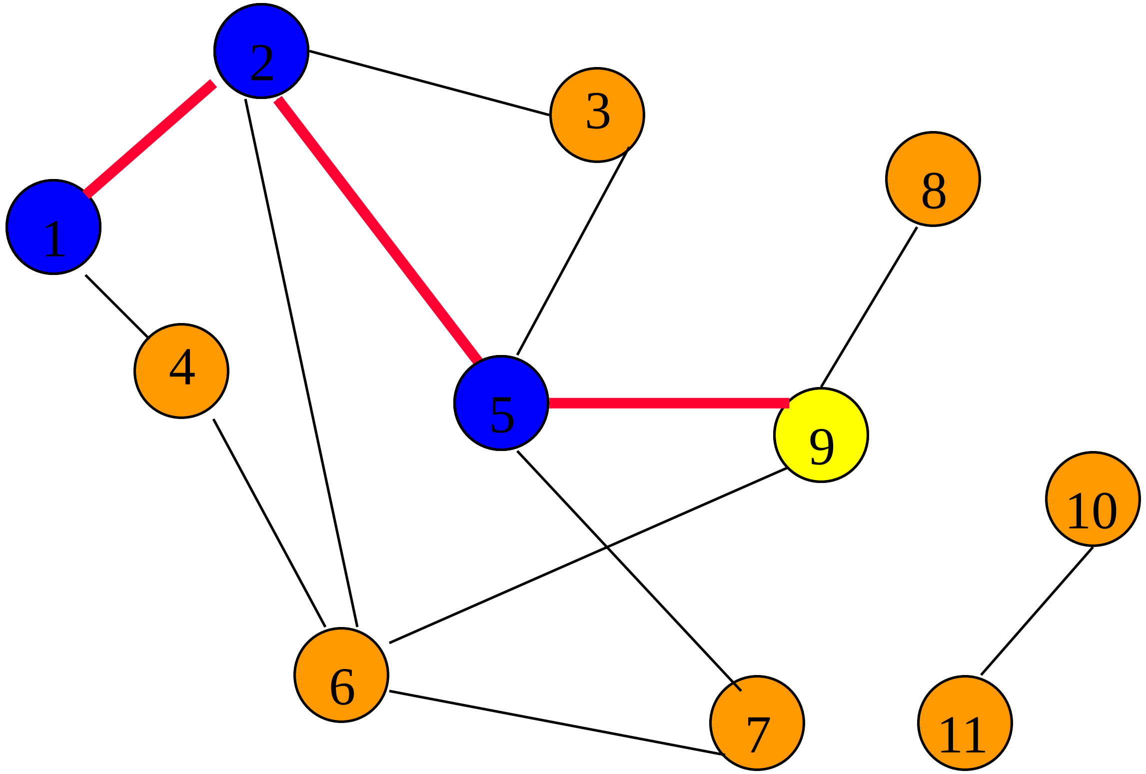
Depth-First Search Example



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

Suppose that vertex 5 is selected.

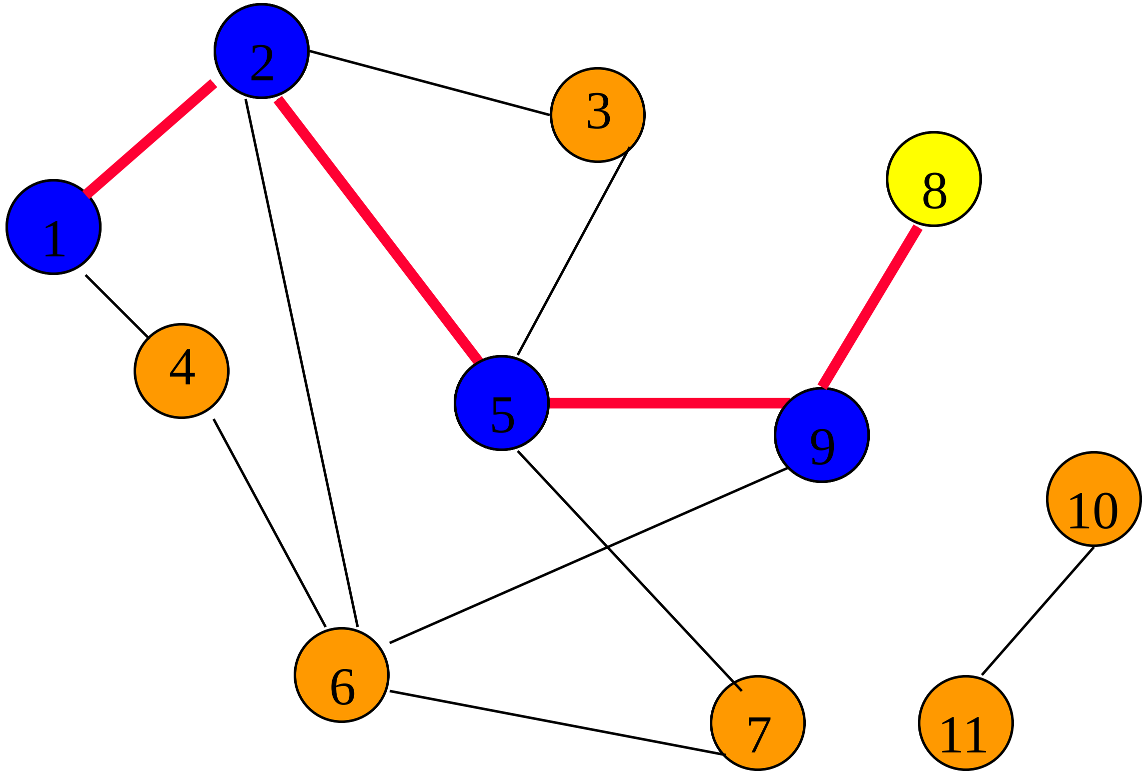
Depth-First Search Example



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

Suppose that vertex 9 is selected.

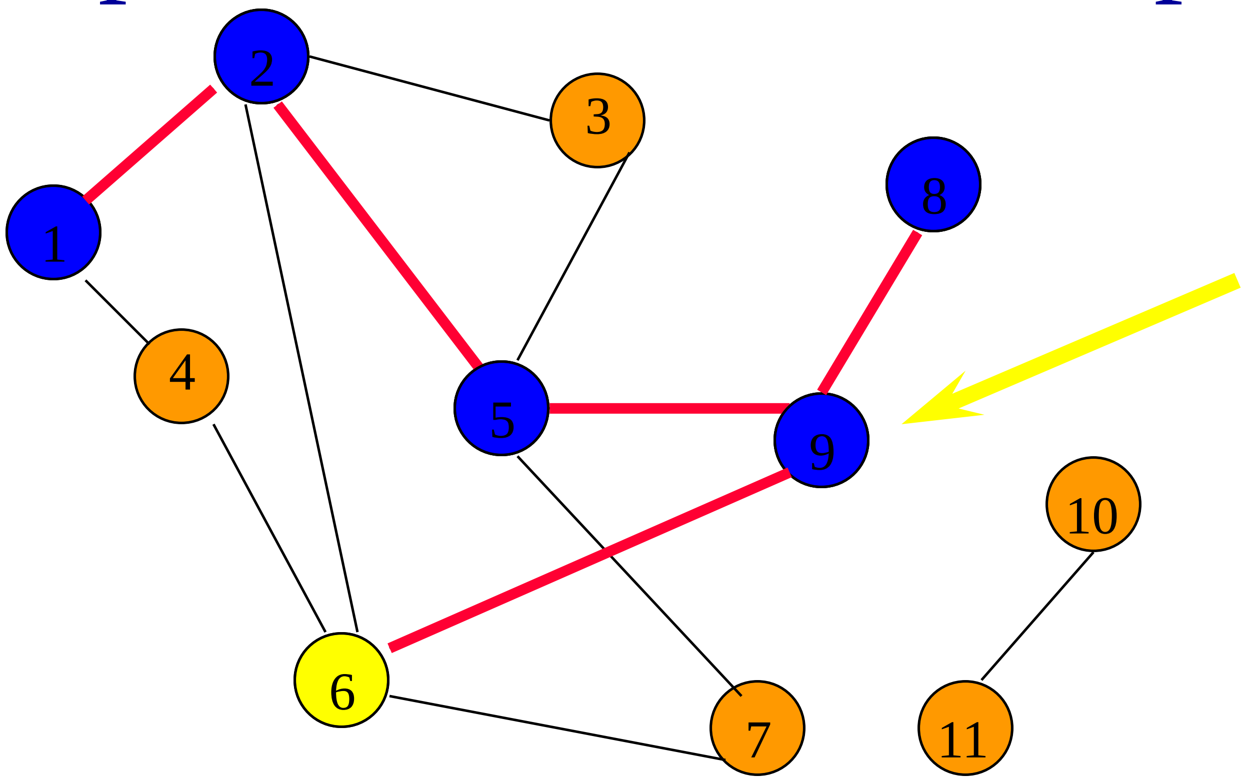
Depth-First Search Example



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

Suppose that vertex 8 is selected.

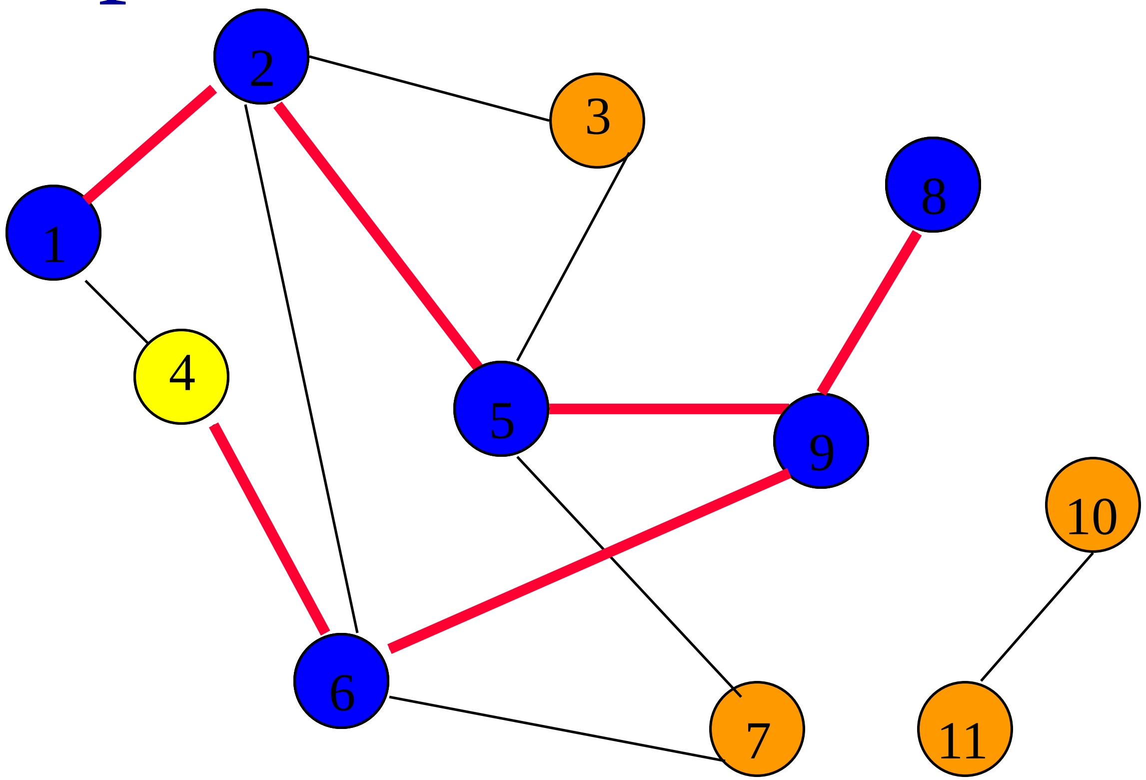
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a DFS(6).

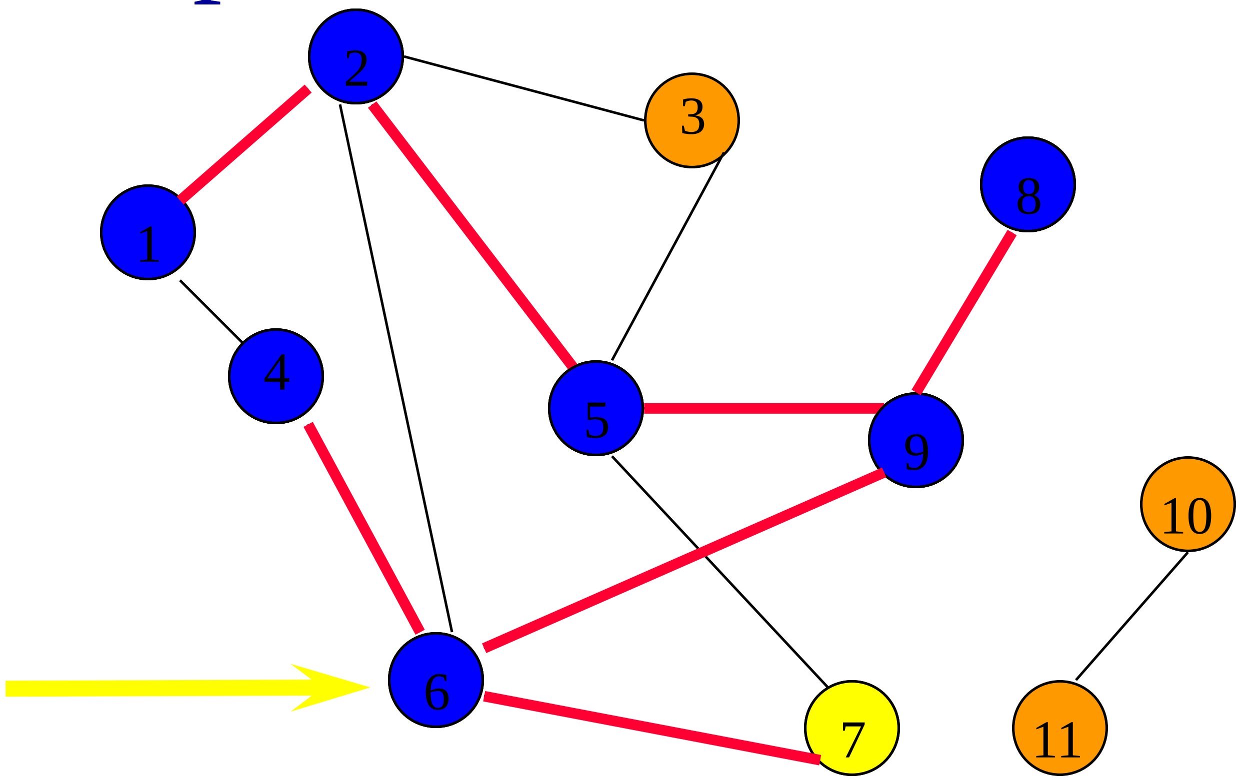
Depth-First Search Example



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

Suppose that vertex 4 is selected.

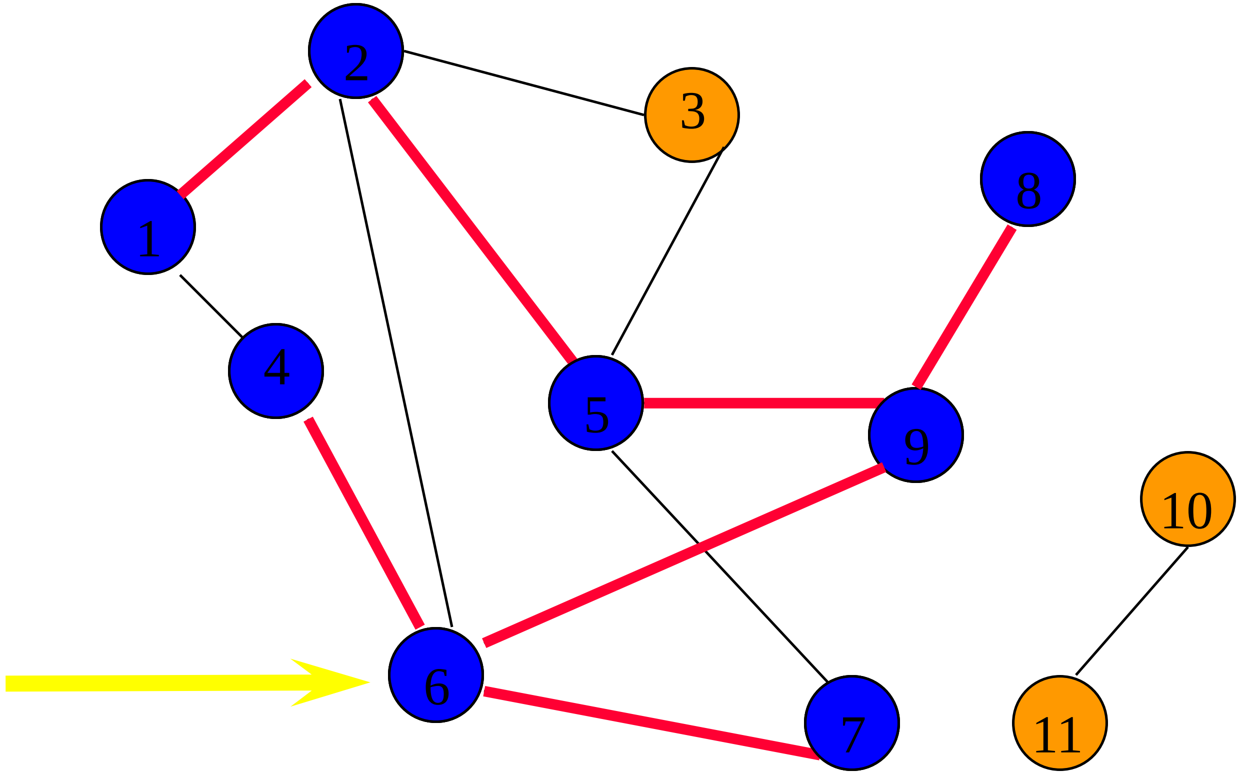
Depth-First Search Example



Label vertex 4 and return to 6.

From vertex 6 do a DFS(7).

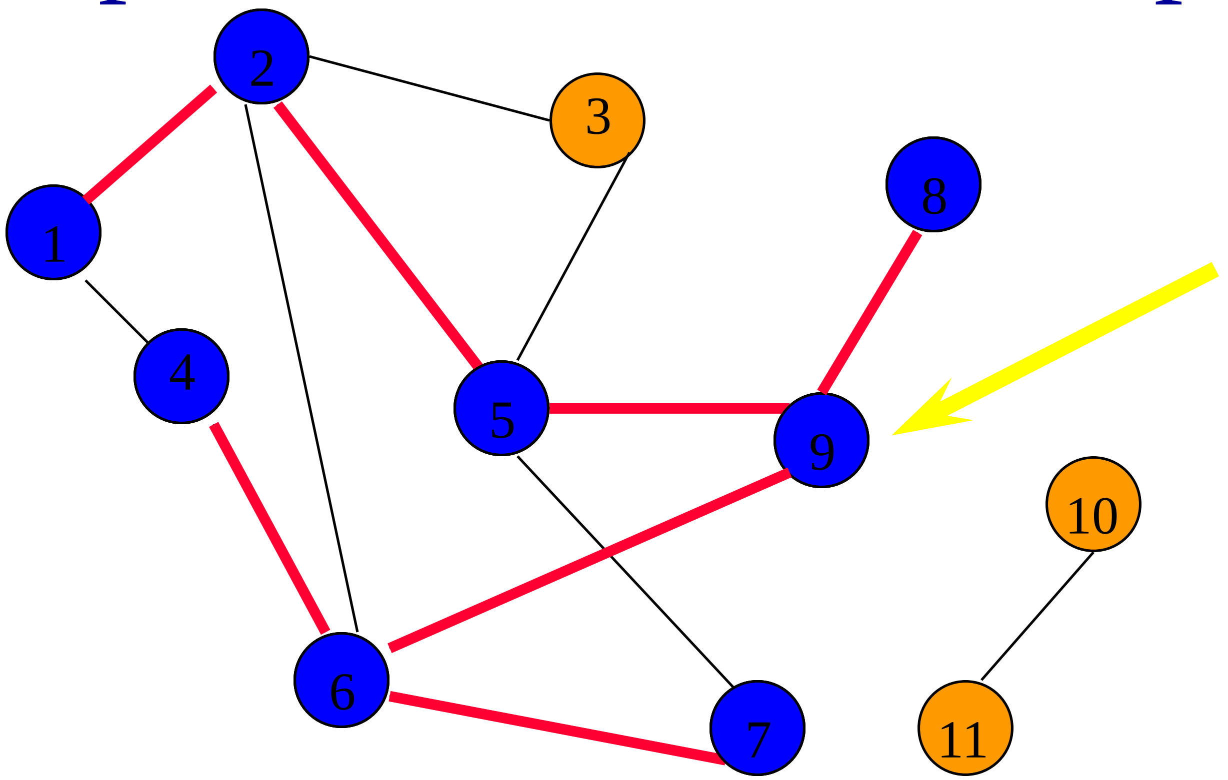
Depth-First Search Example



Label vertex 7 and return to 6.

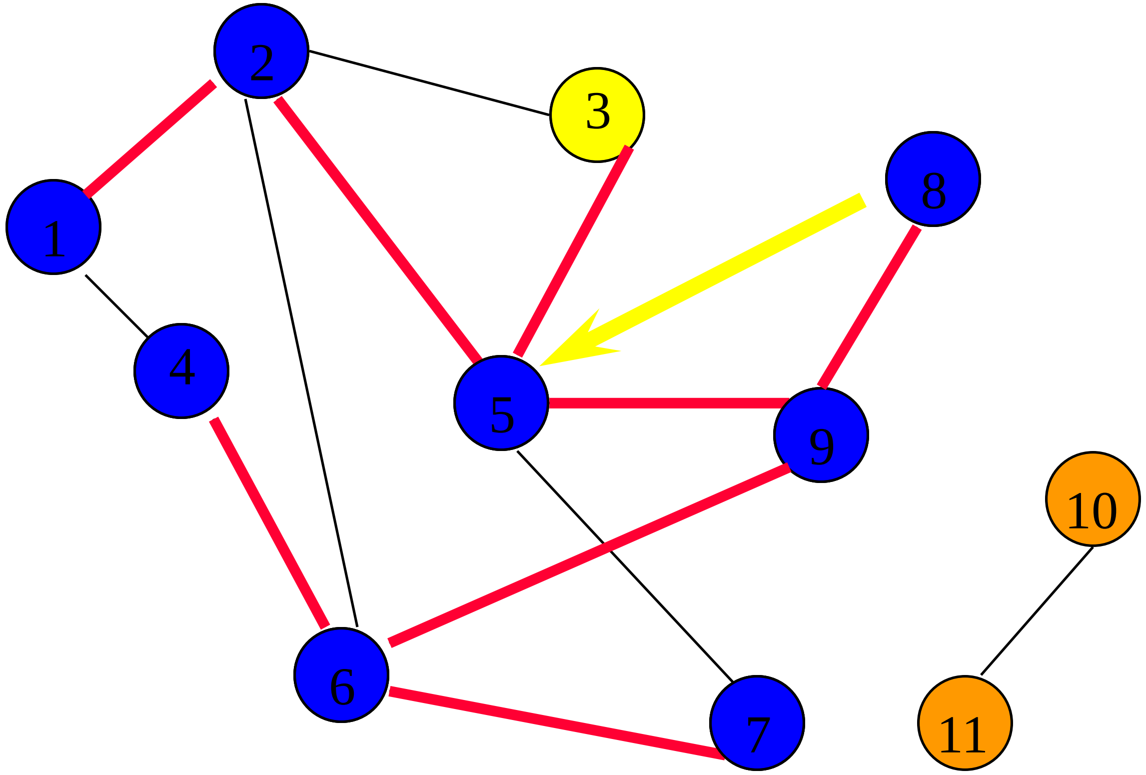
Return to 9.

Depth-First Search Example



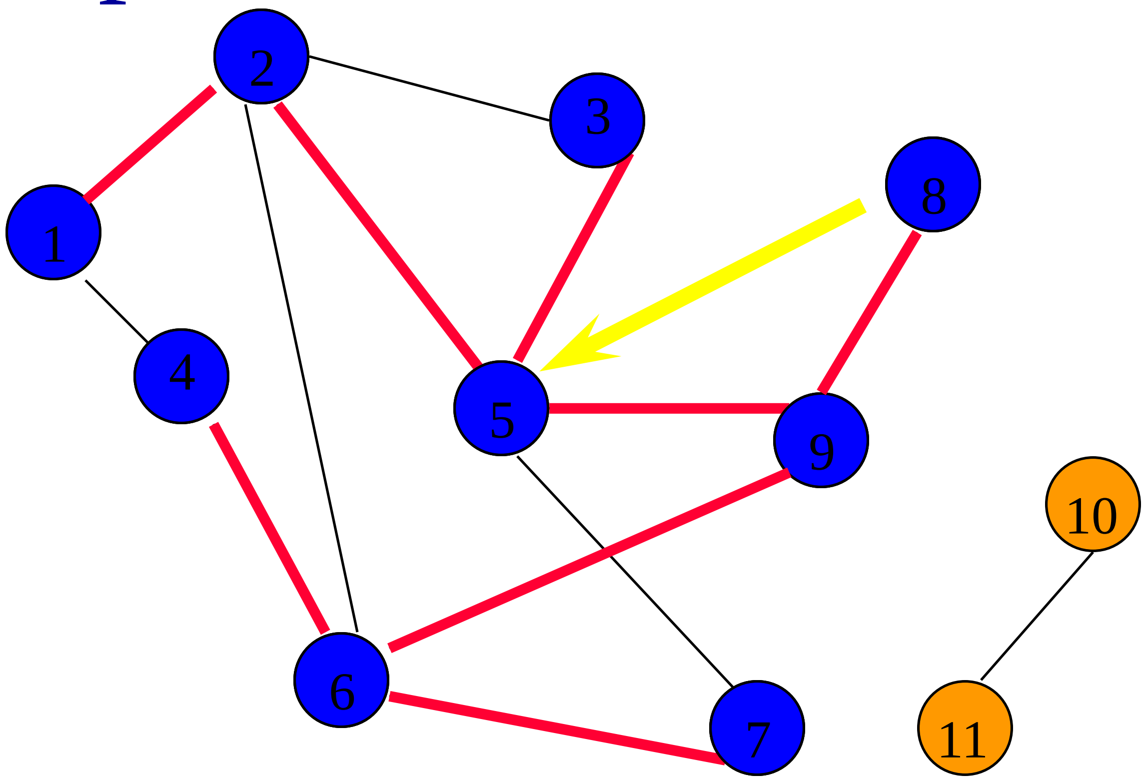
Return to 5.

Depth-First Search Example



Do a DFS(3).

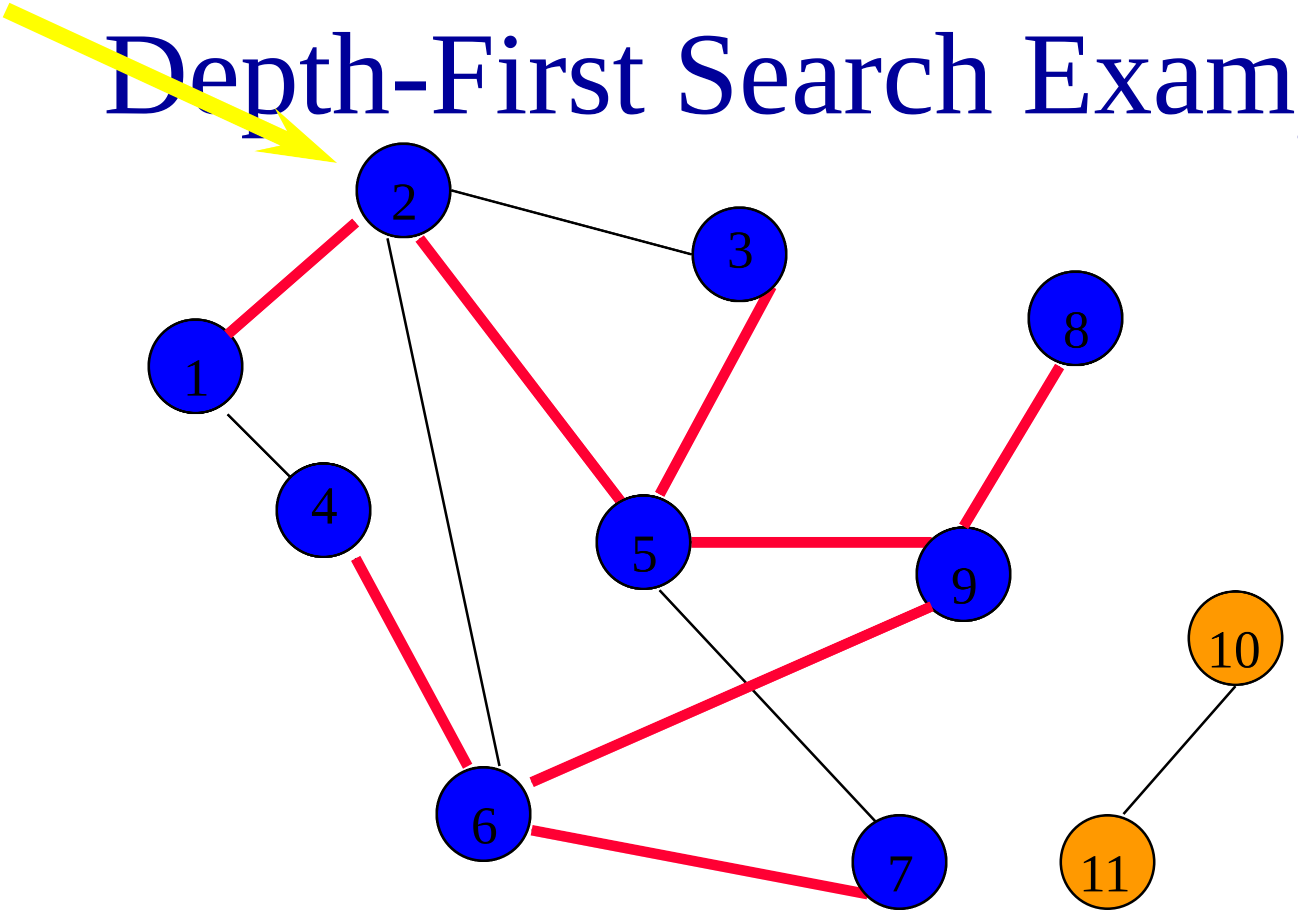
Depth-First Search Example



Label 3 and return to 5.

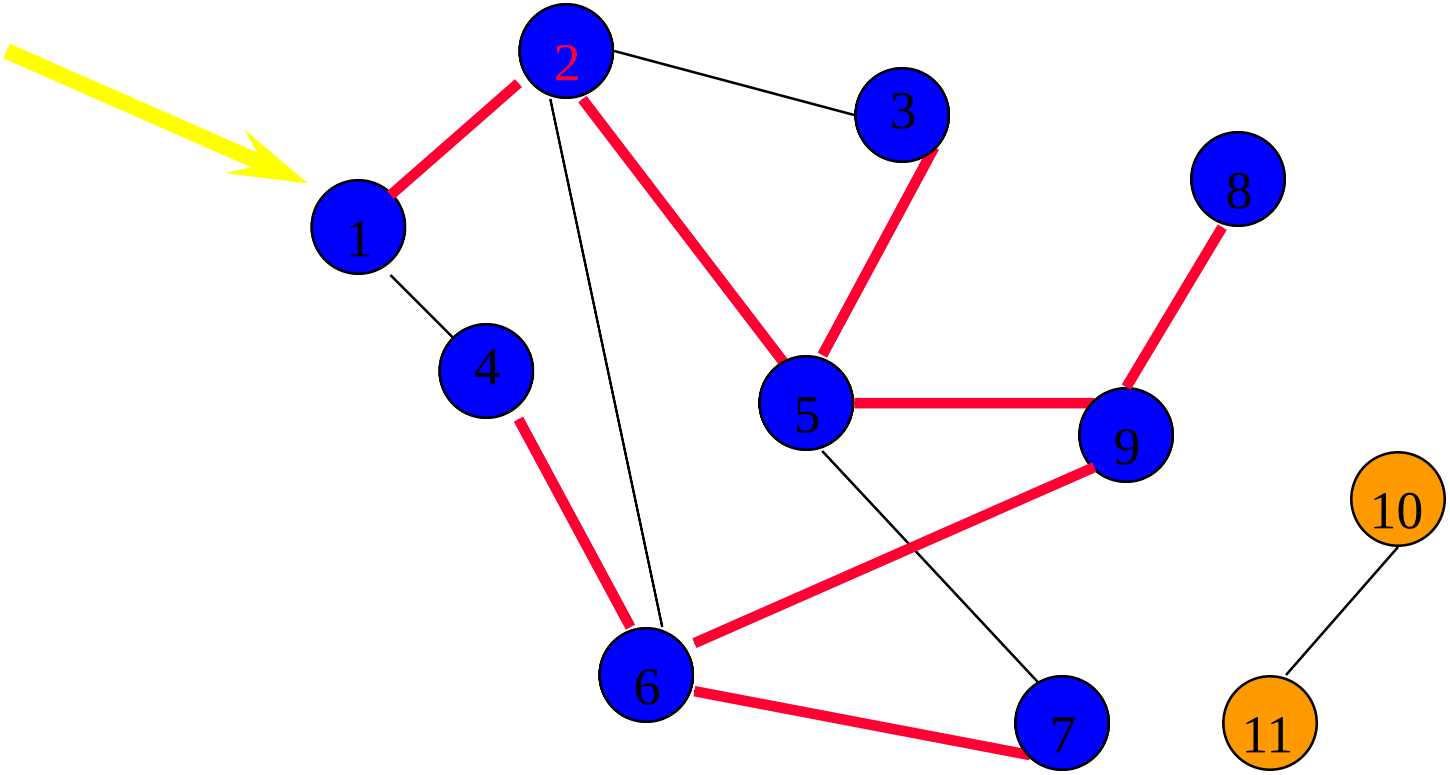
Return to 2.

Depth-First Search Example



Return to 1.

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.

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)

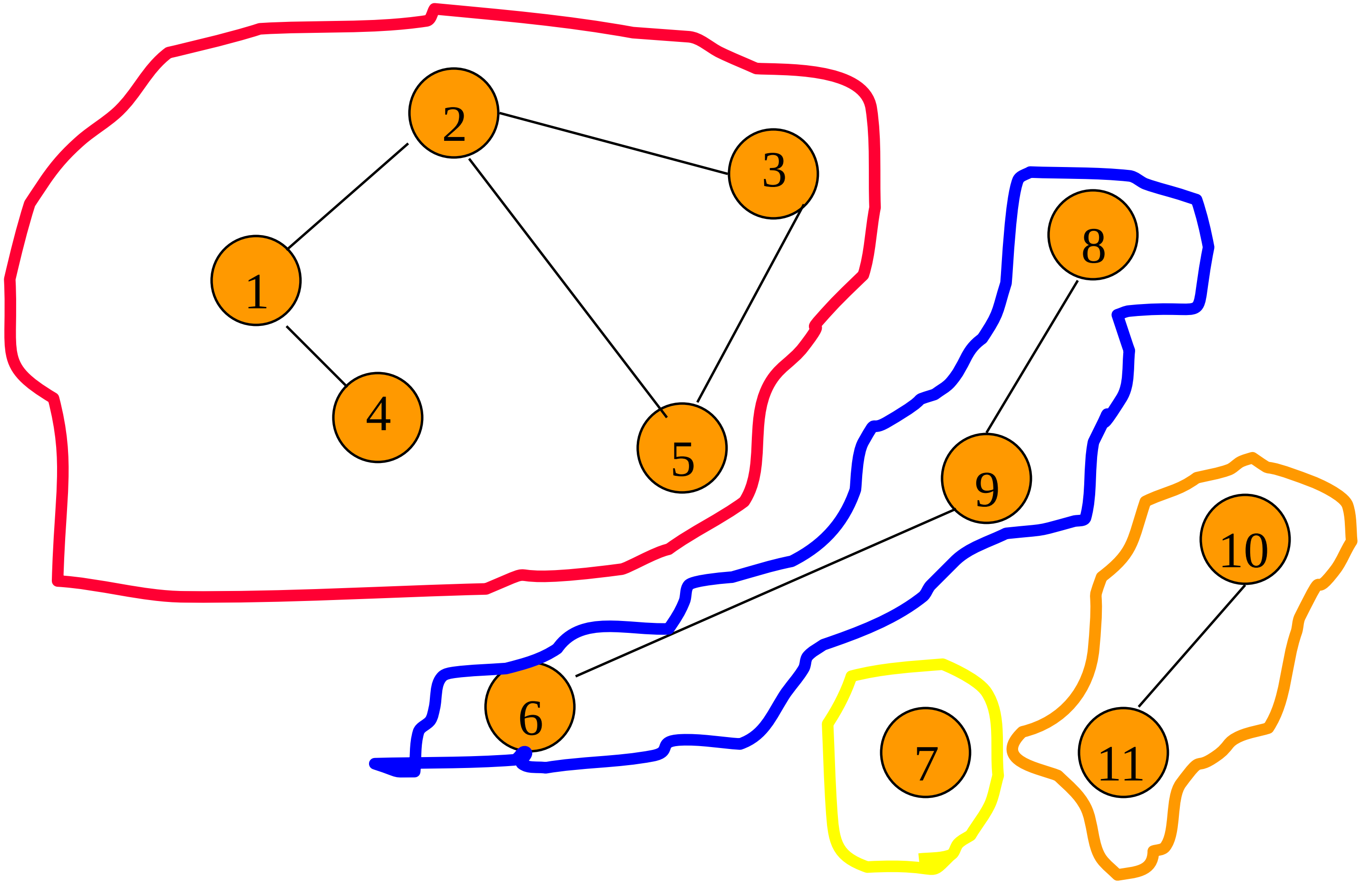
Is The Graph Connected?

- 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)

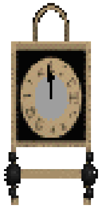
Connected Components

- 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

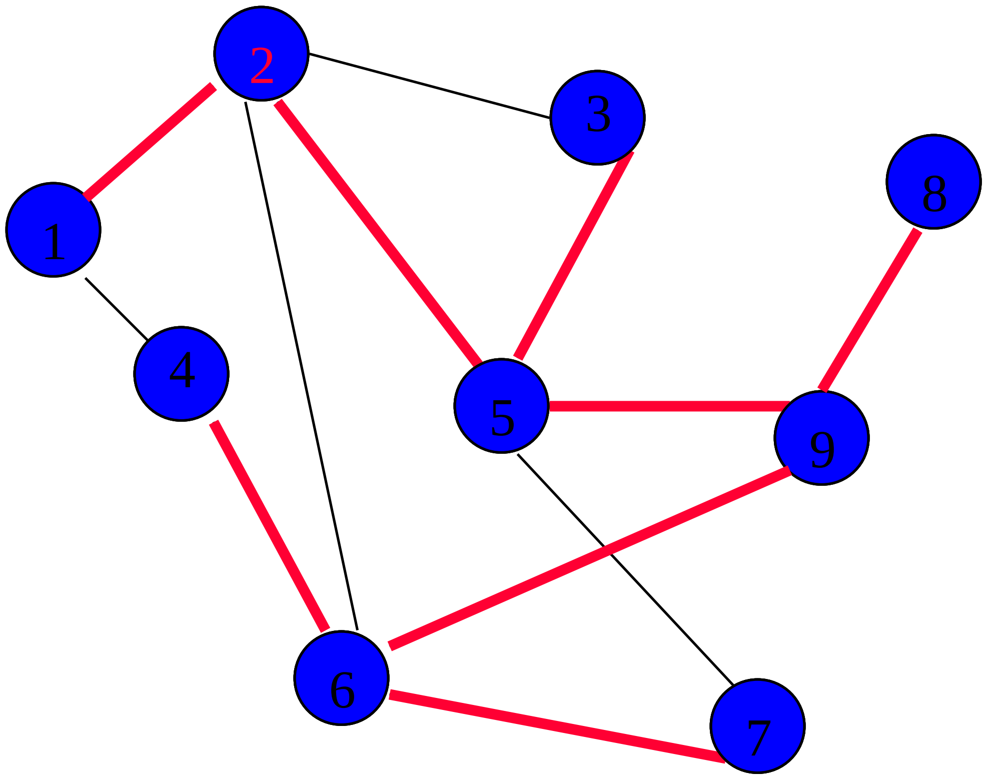


Time Complexity



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

Spanning Tree



Depth-first search from vertex 1.

Depth-first spanning tree.

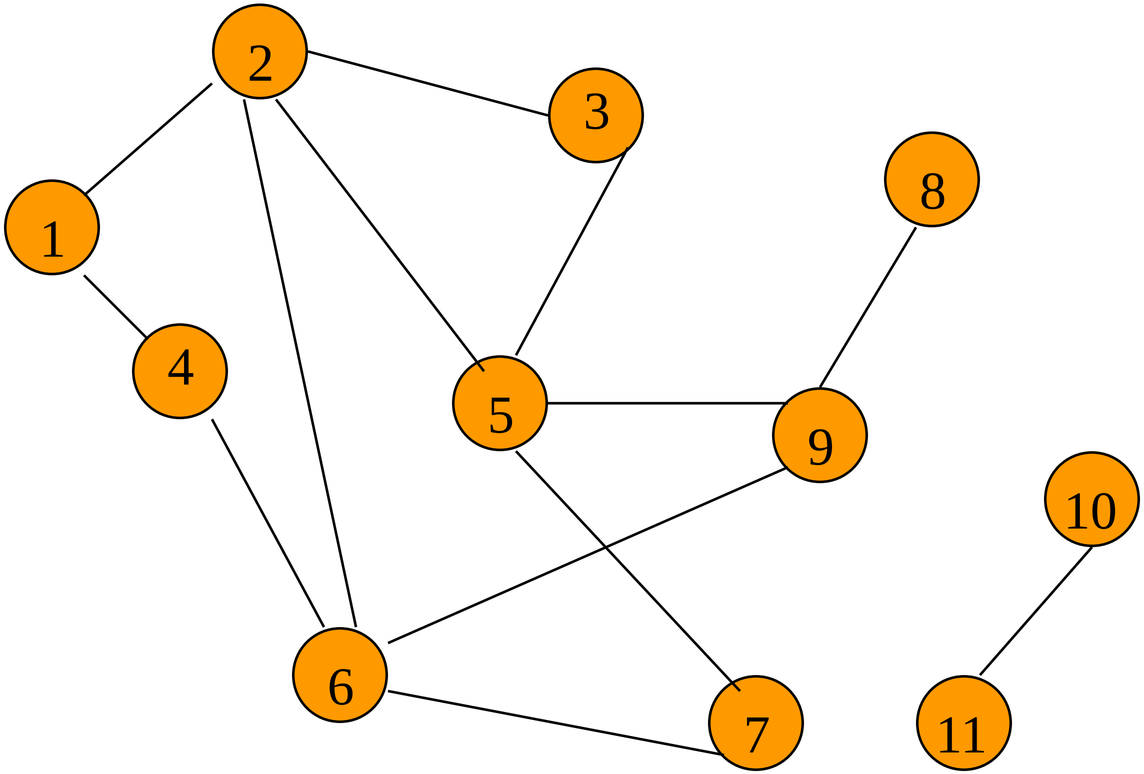
Spanning Tree

- 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

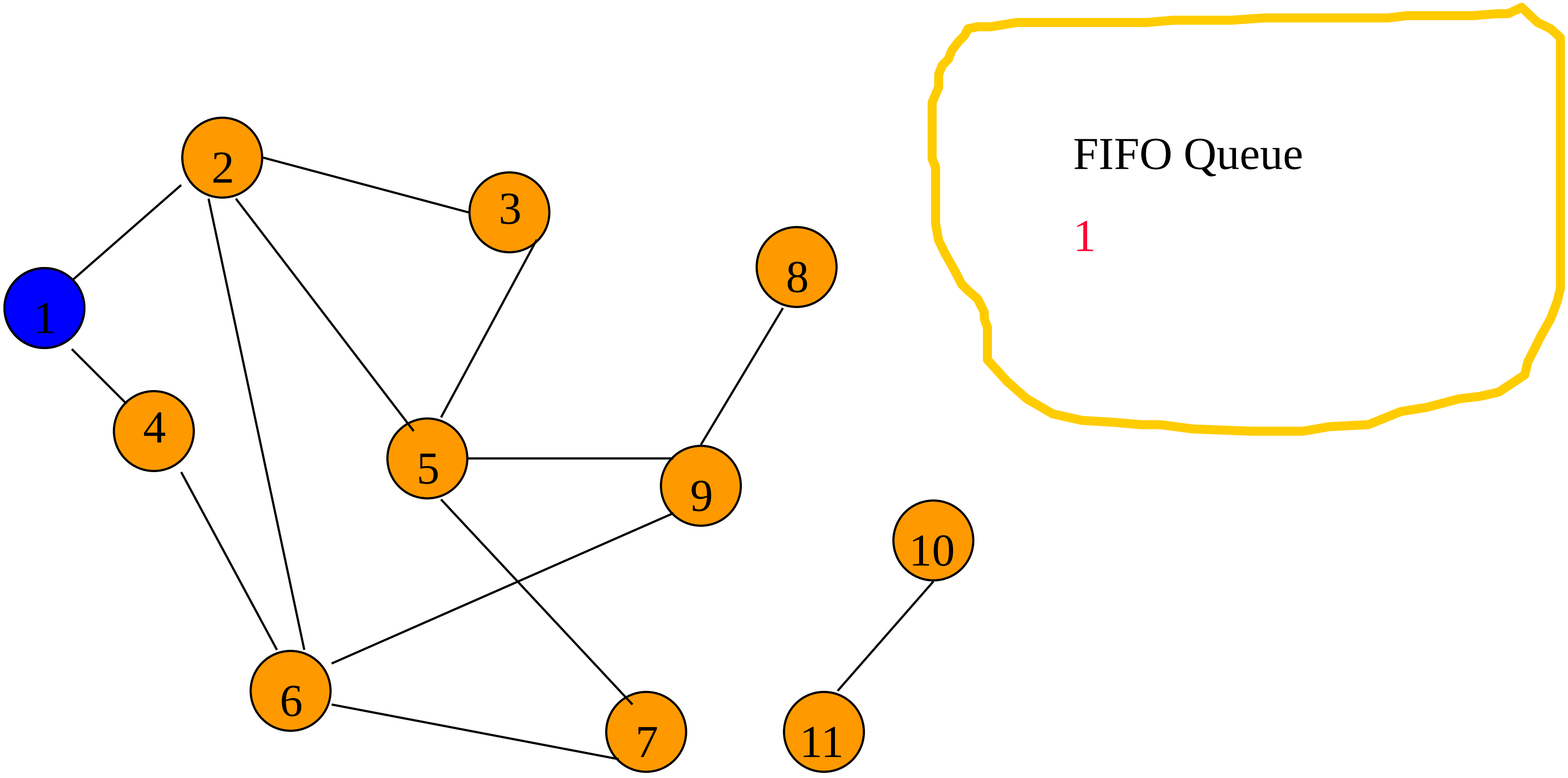
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search Example



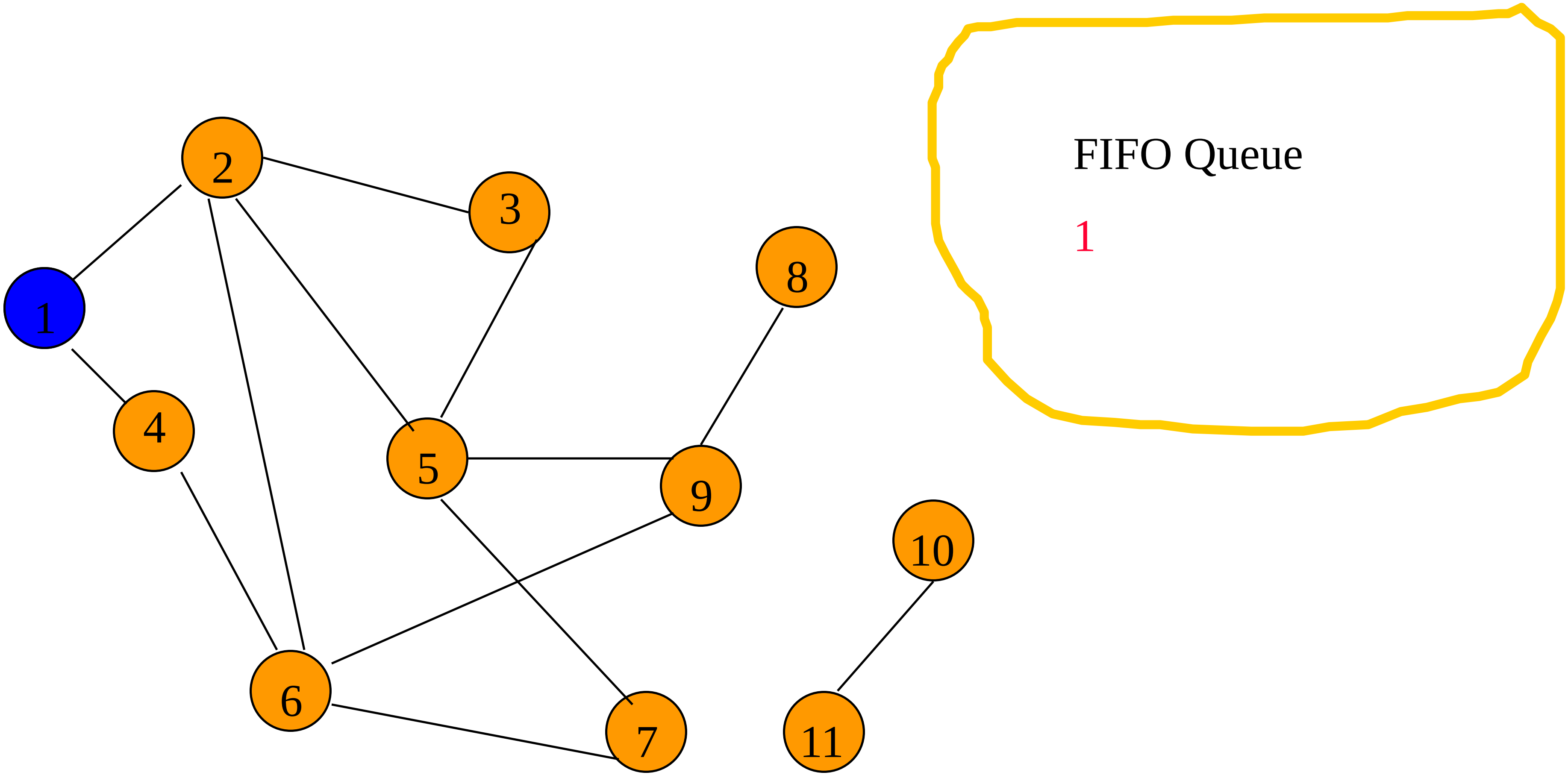
Start search at vertex 1.

Breadth-First Search Example



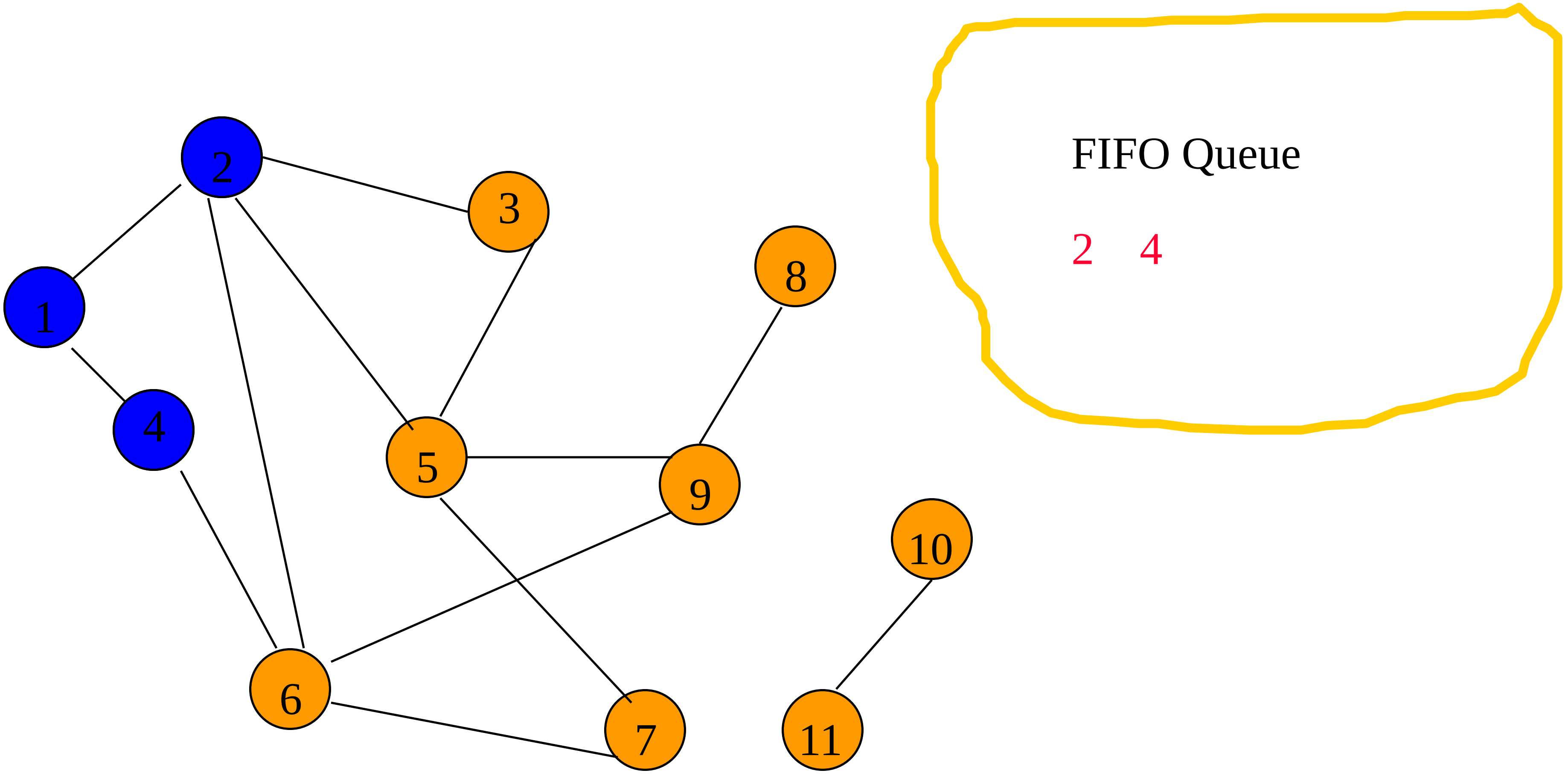
Visit/mark/label start vertex and put in a FIFO queue.

Breadth-First Search Example



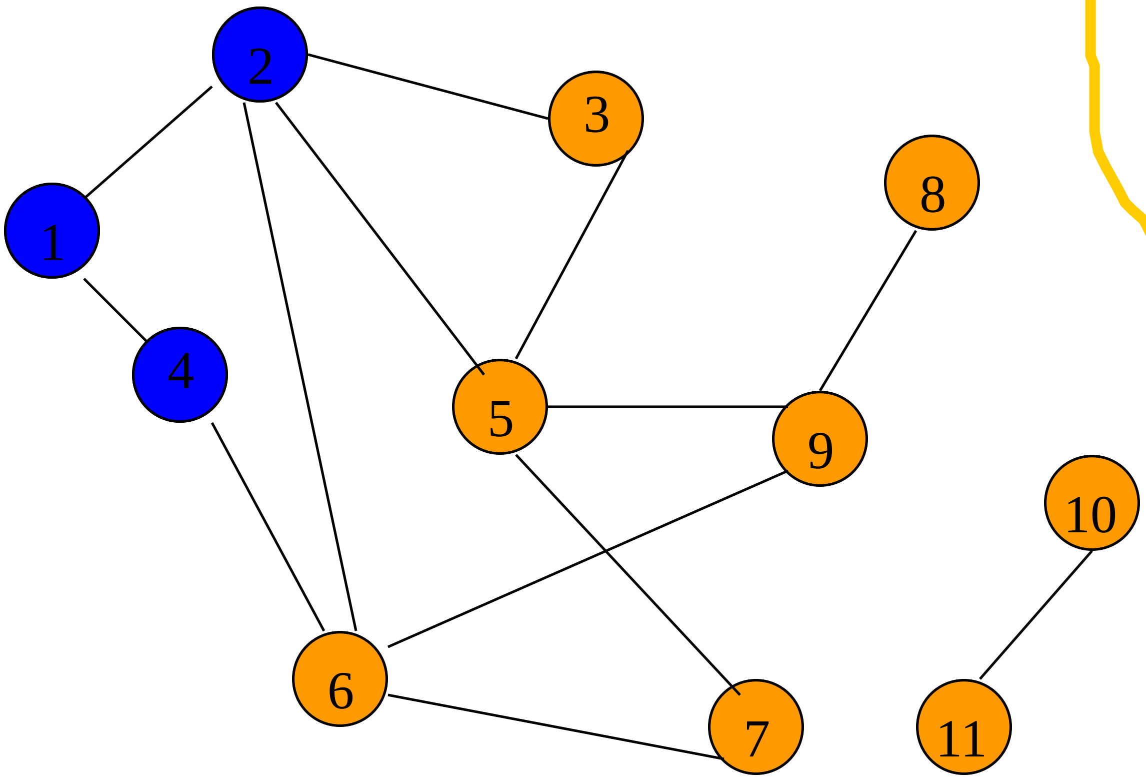
Remove 1 from Q; visit adjacent unvisited vertices; put in Q.

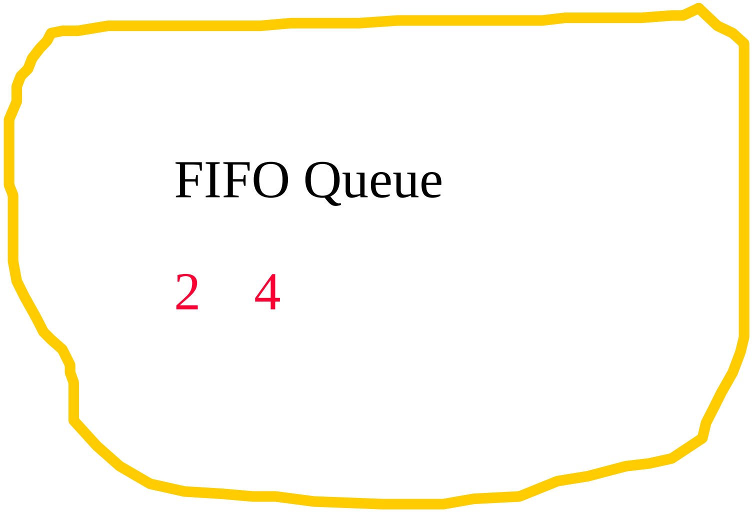
Breadth-First Search Example



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

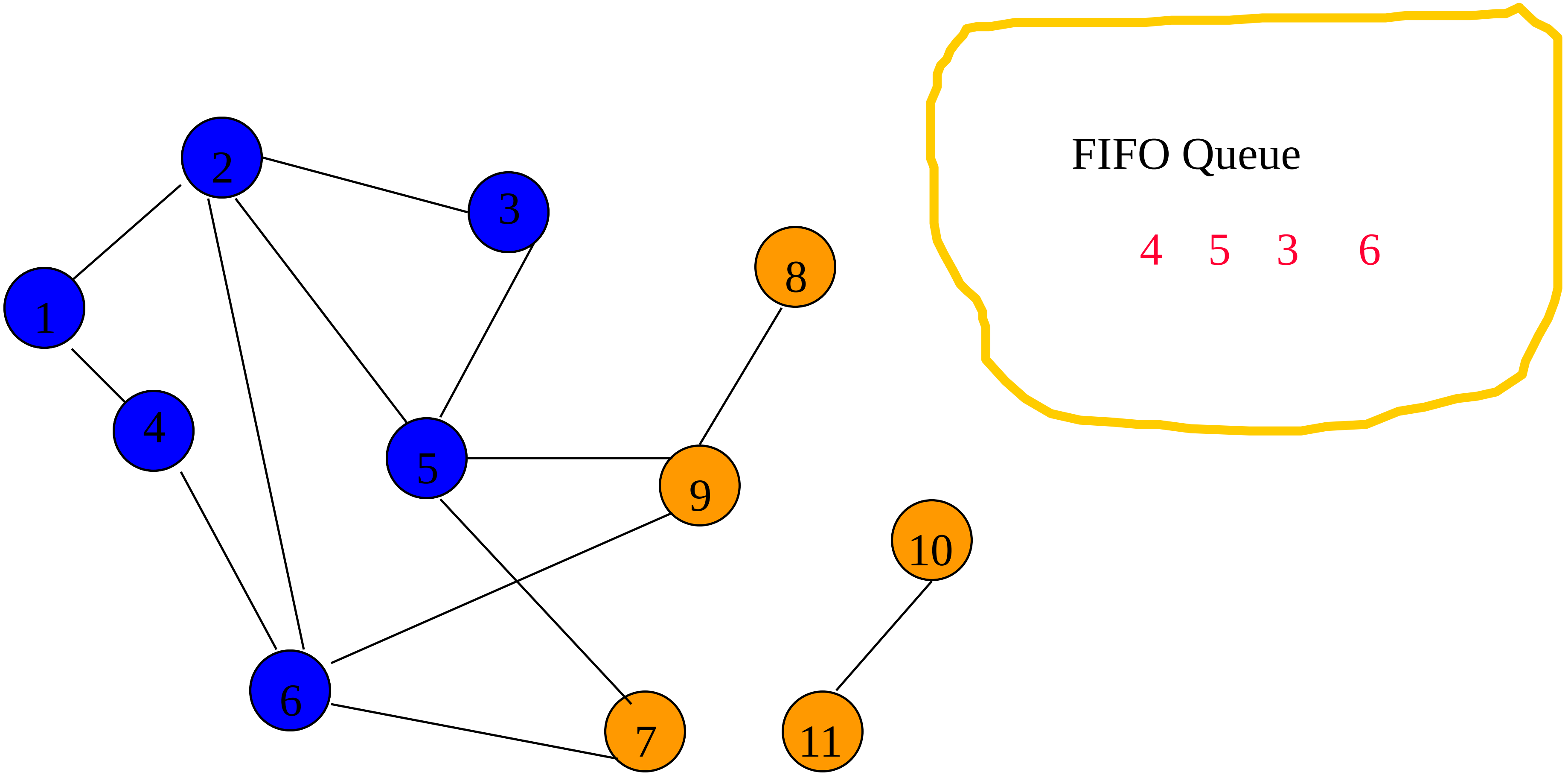
Breadth-First Search Example





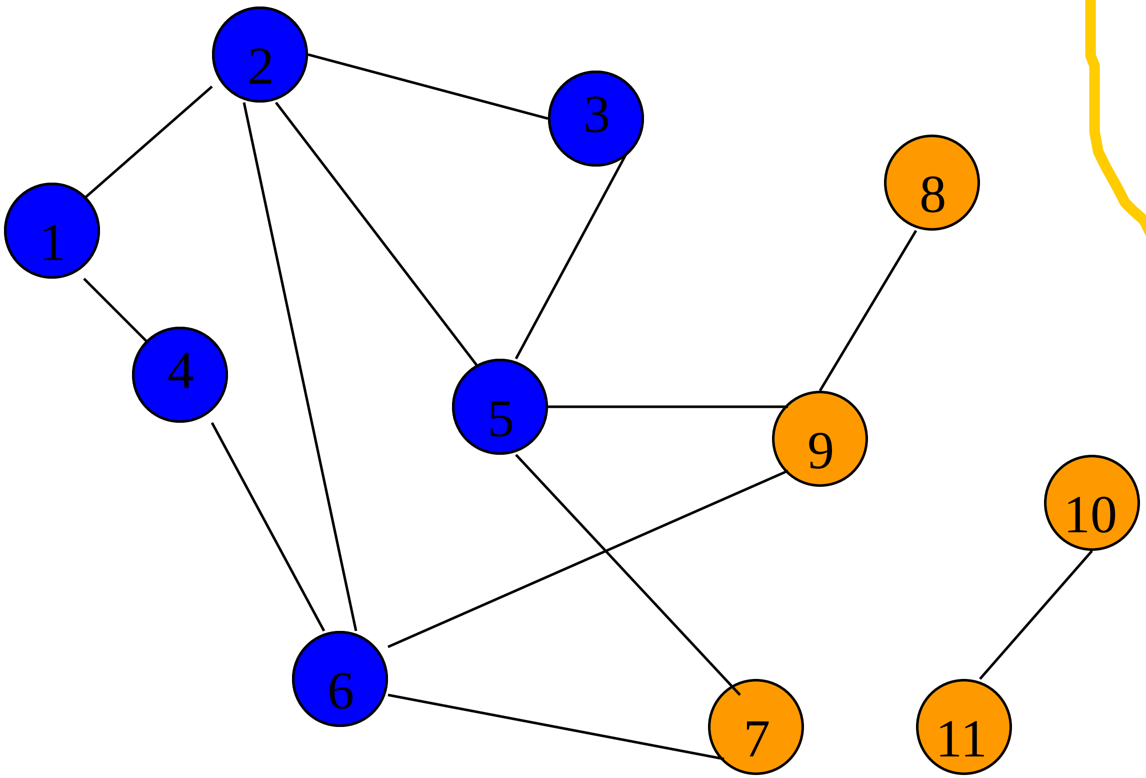
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

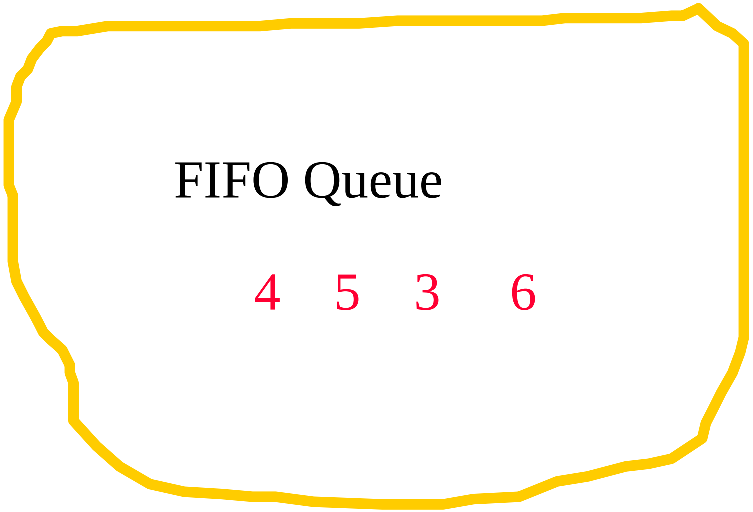
Breadth-First Search Example



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

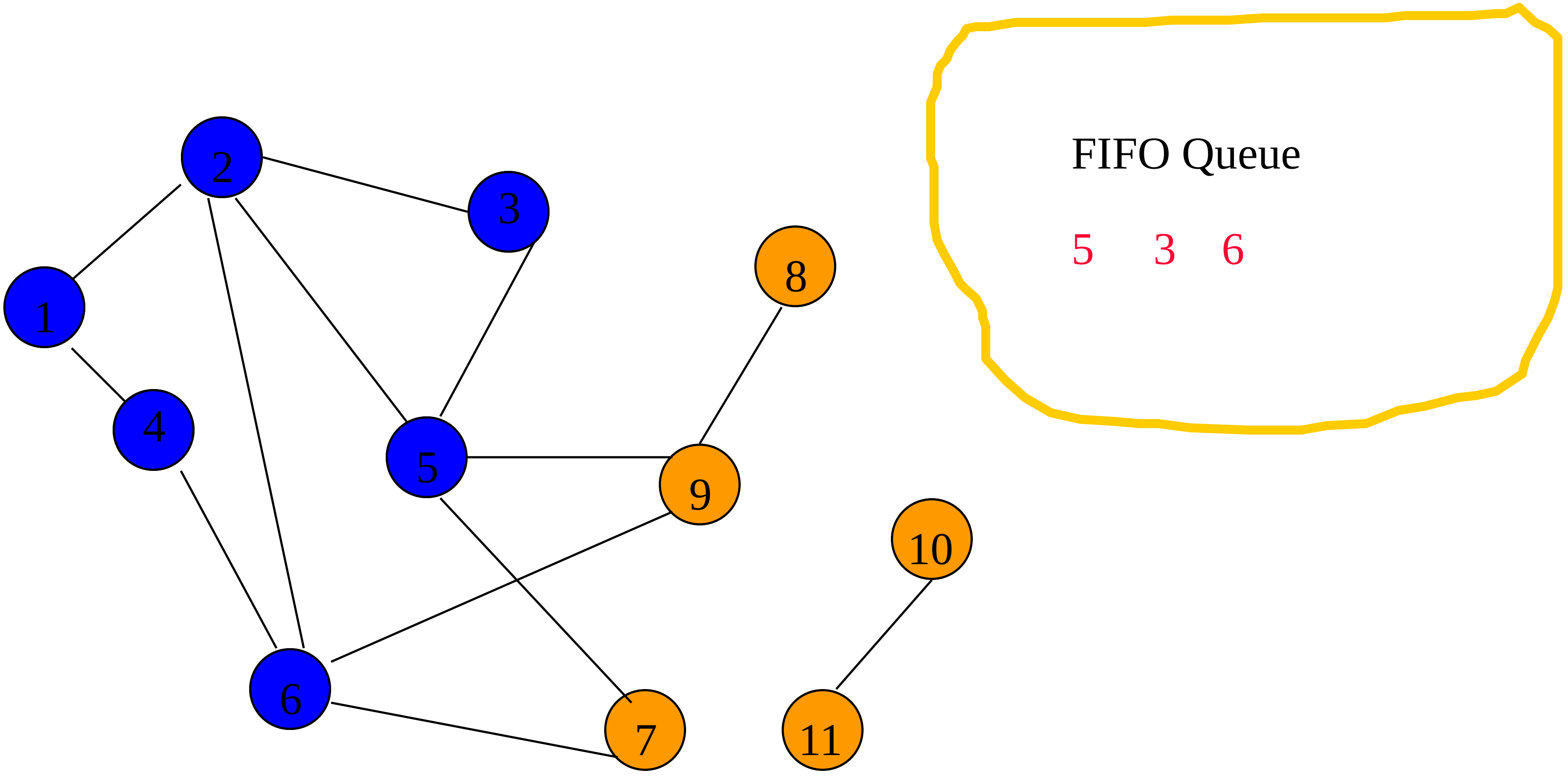
Breadth-First Search Example





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

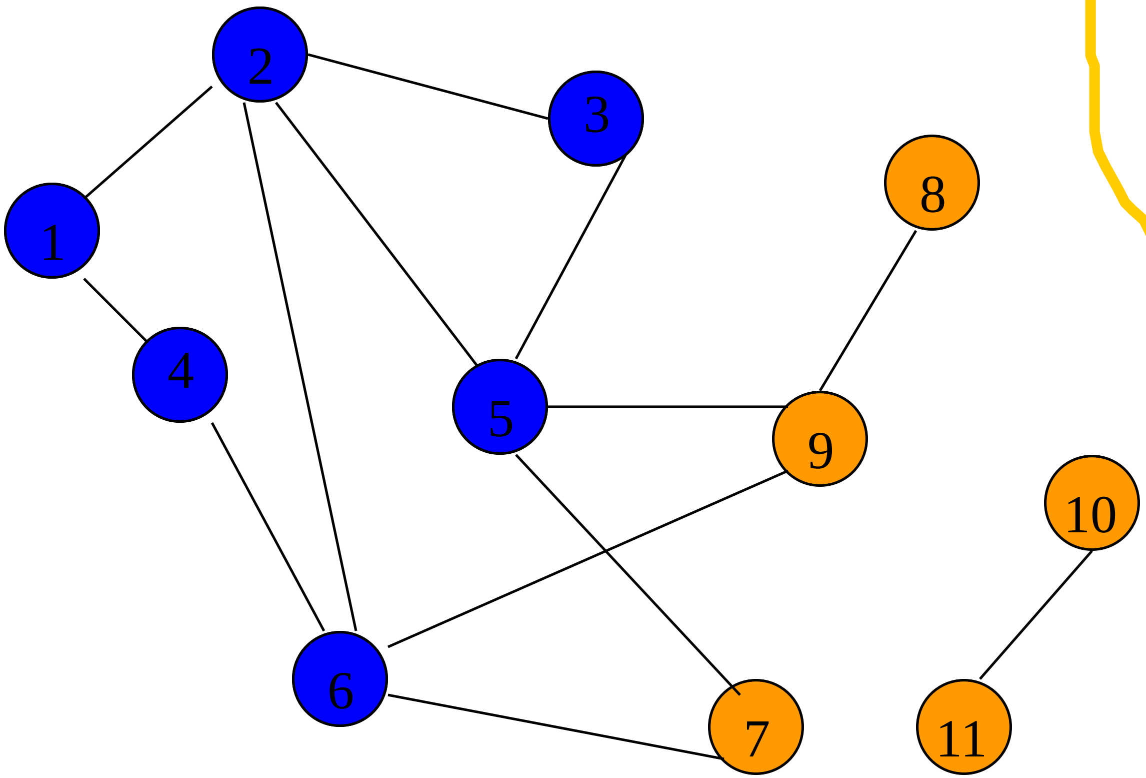
Breadth-First Search Example



Remove 4 from Q; visit adjacent unvisited vertices;

put in Q.

Breadth-First Search Example

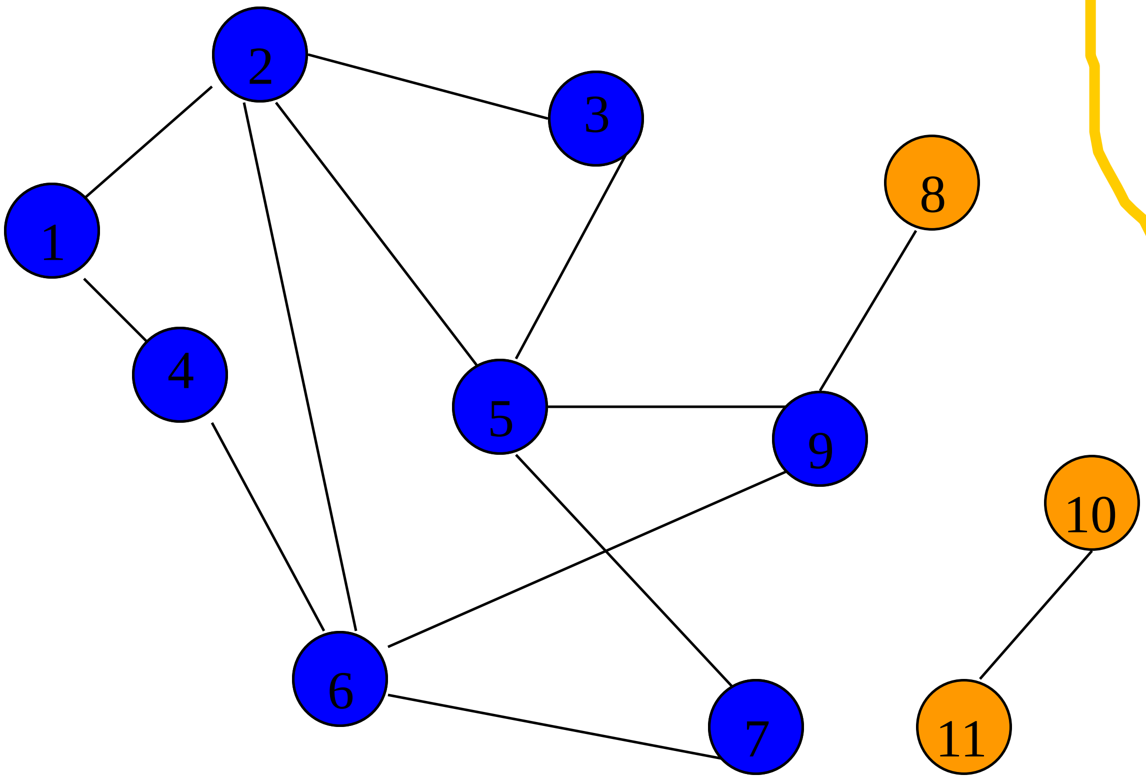


FIFO Queue

5 3 6

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

Breadth-First Search Example

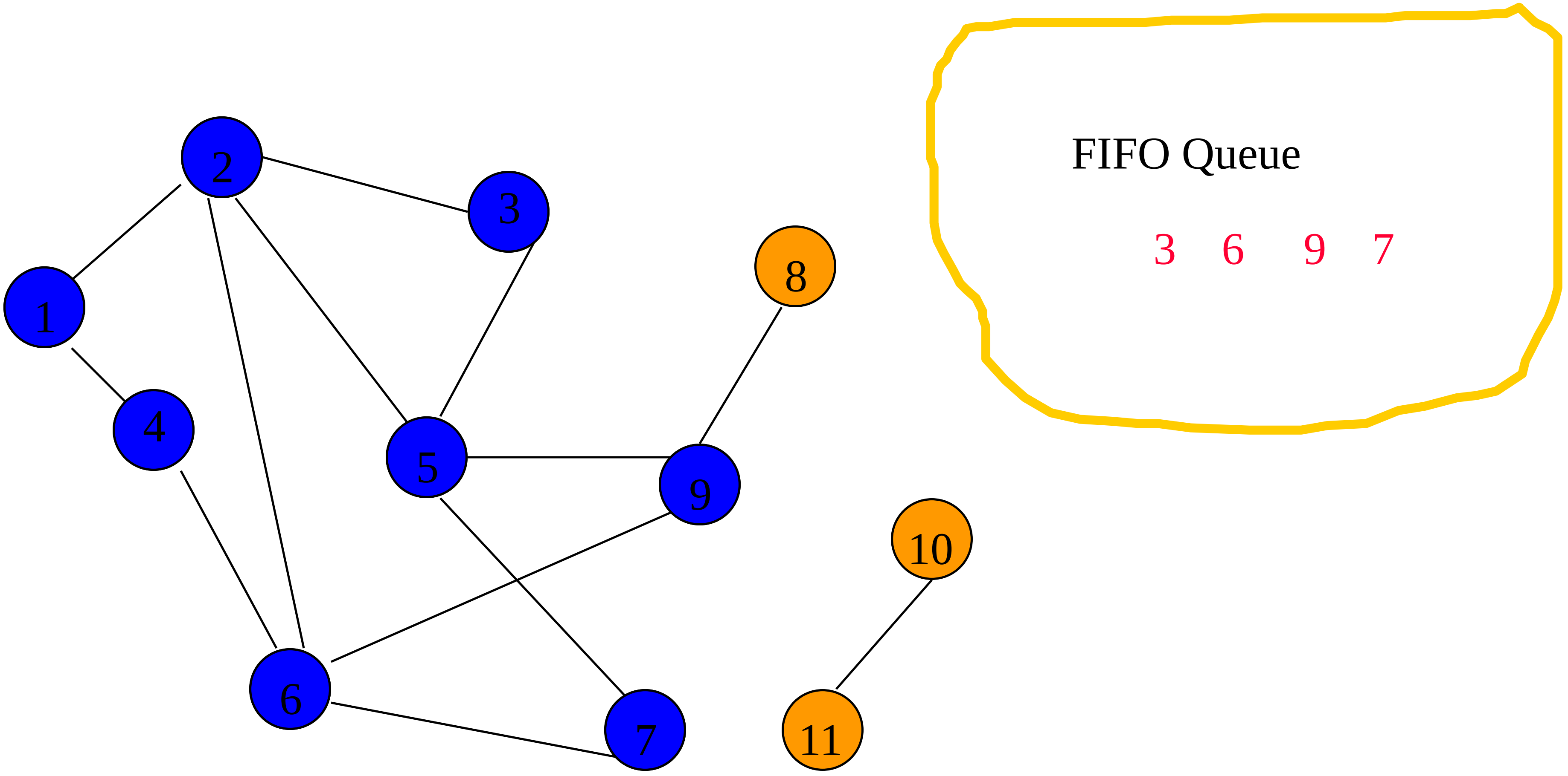


FIFO Queue

3 6 9 7

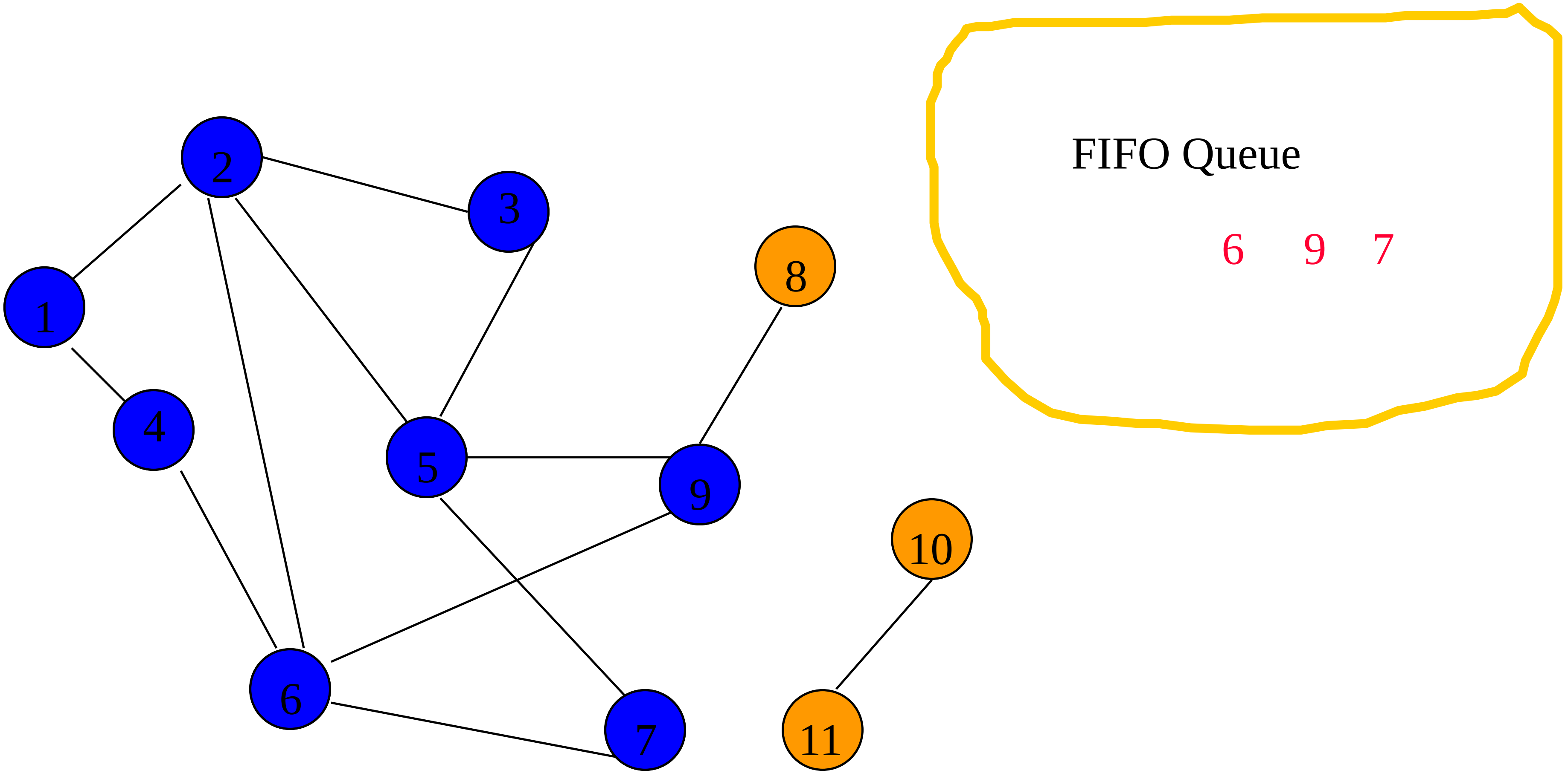
Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



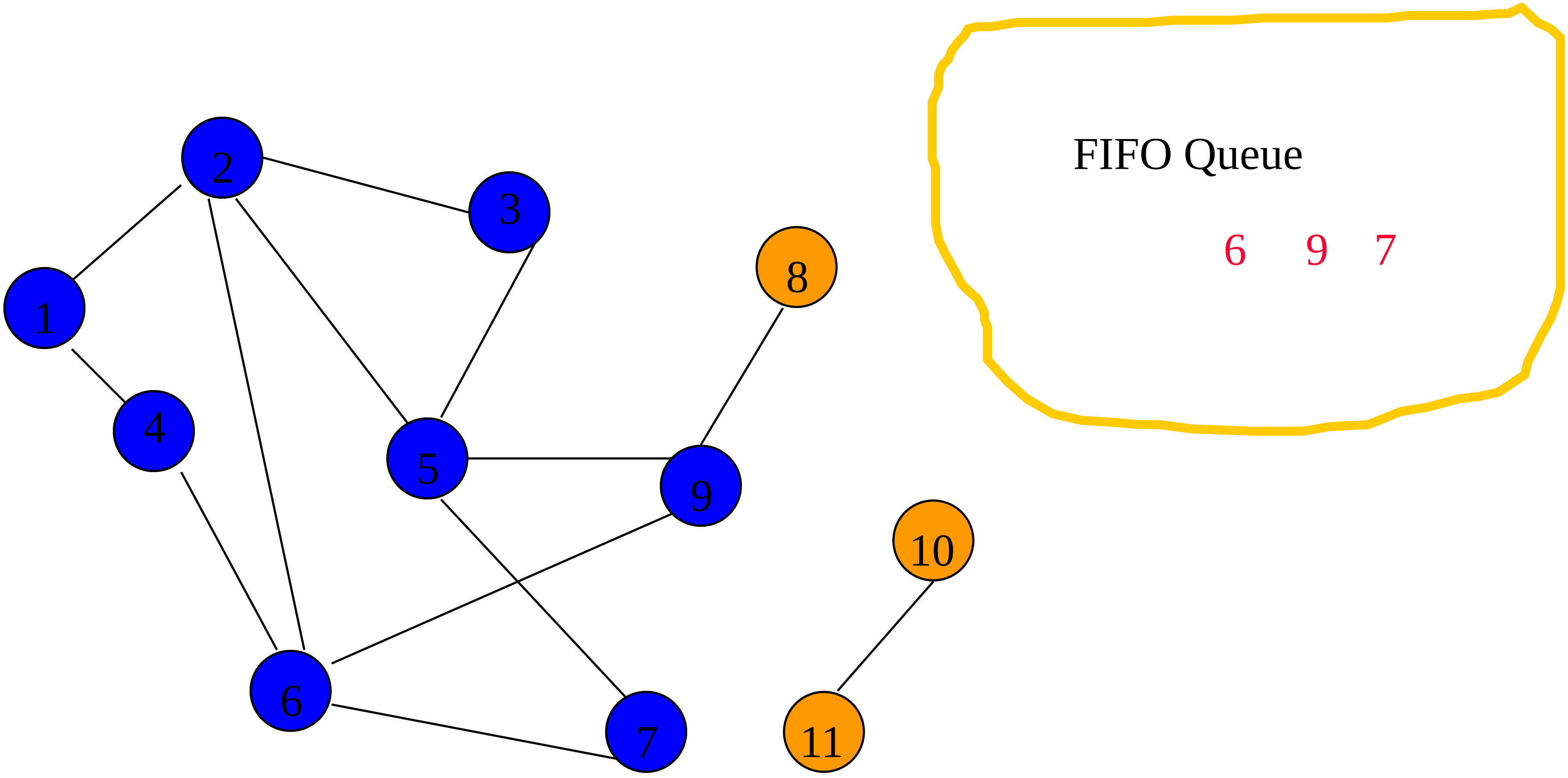
Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



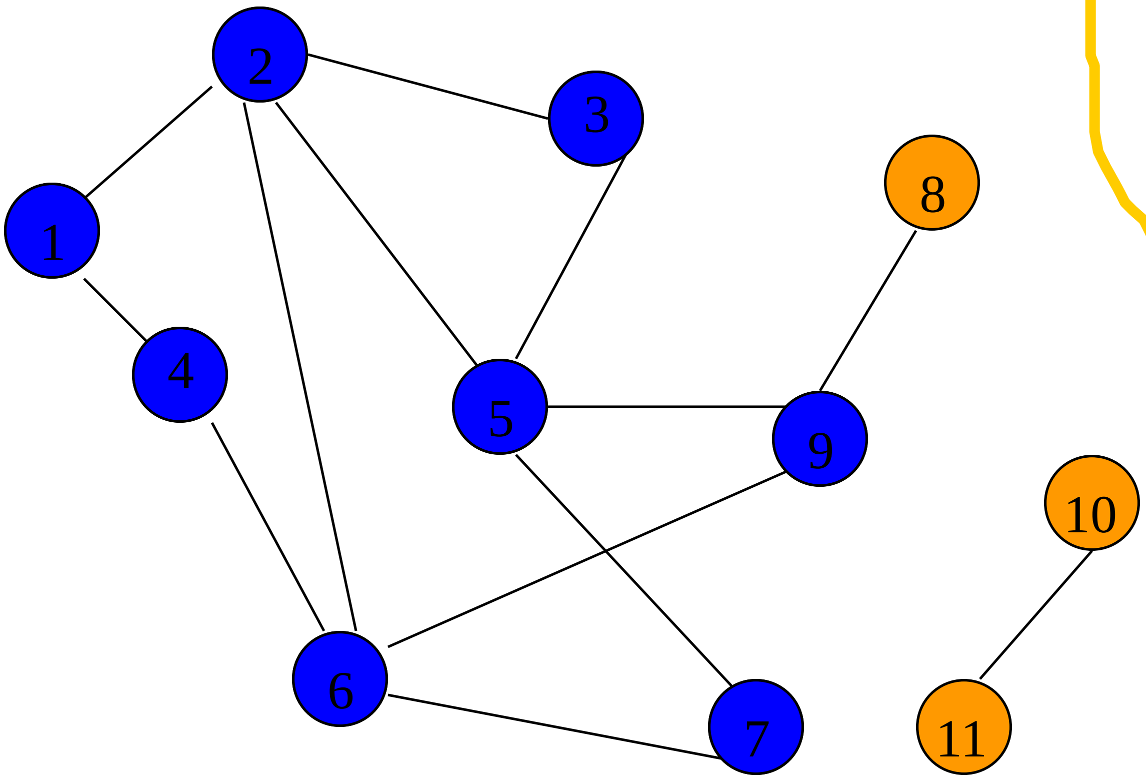
Remove 3 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



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

Breadth-First Search Example

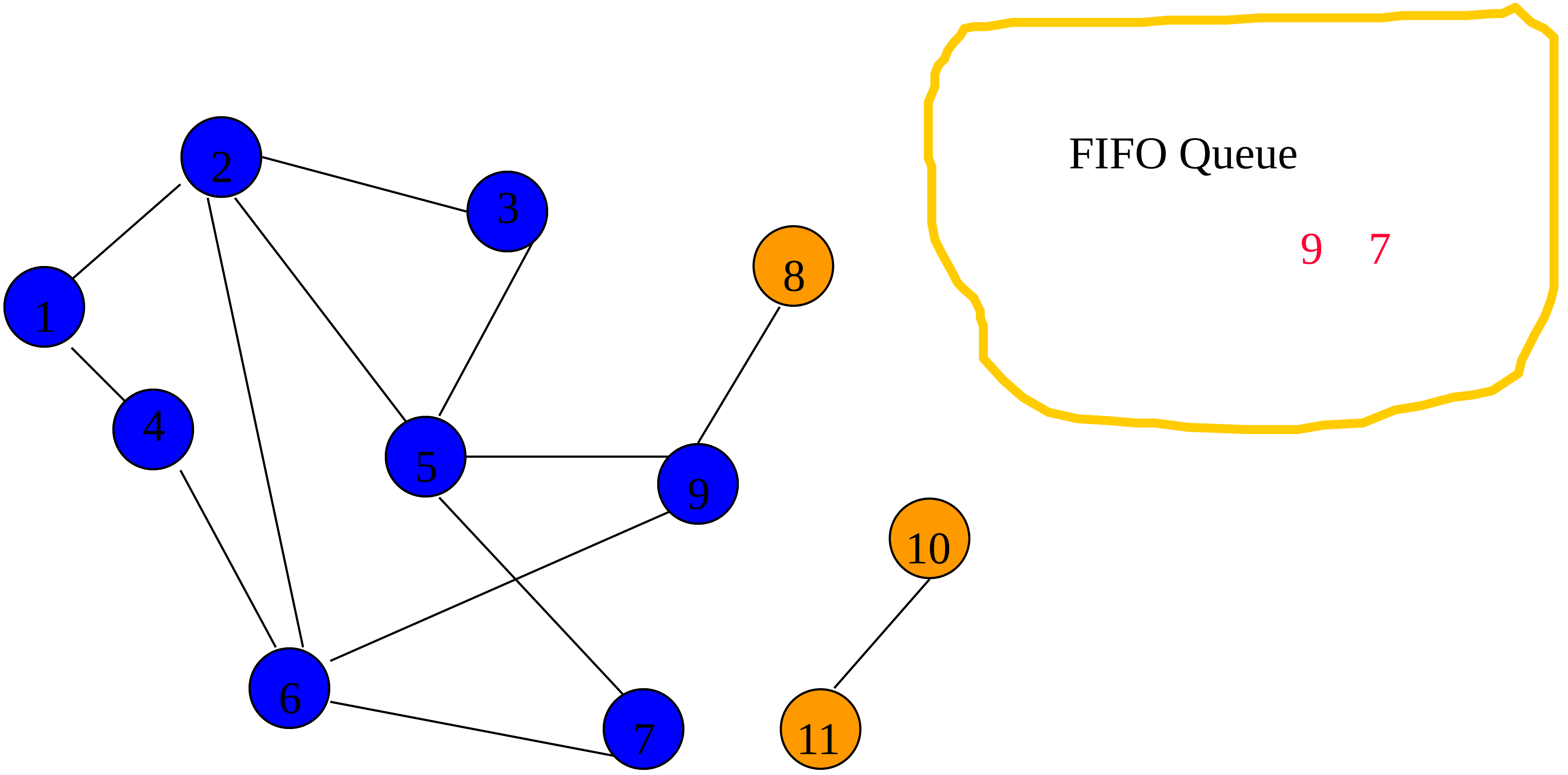


FIFO Queue

9 7

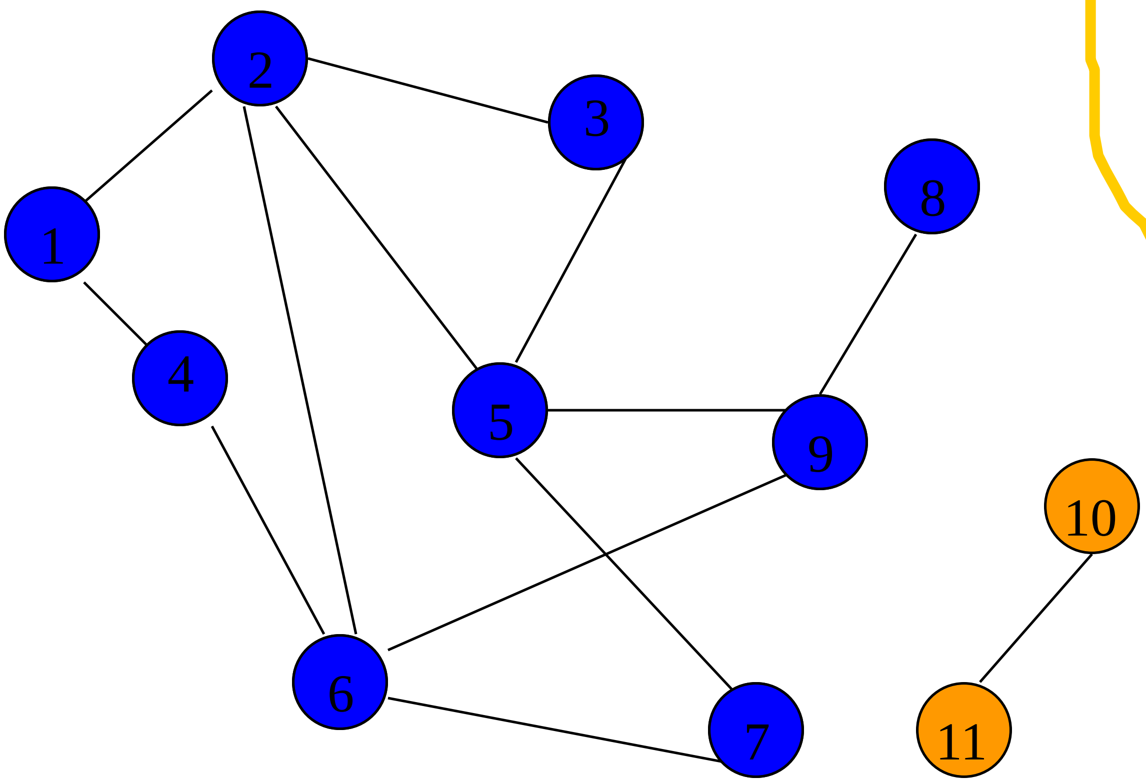
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

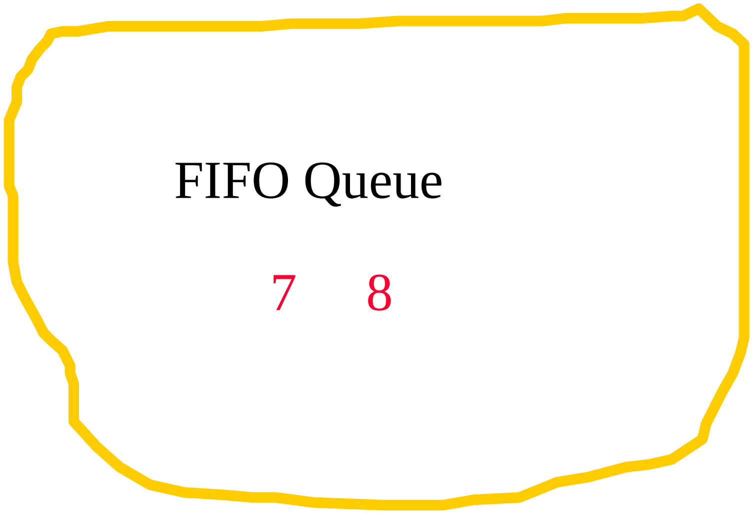
Breadth-First Search Example



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

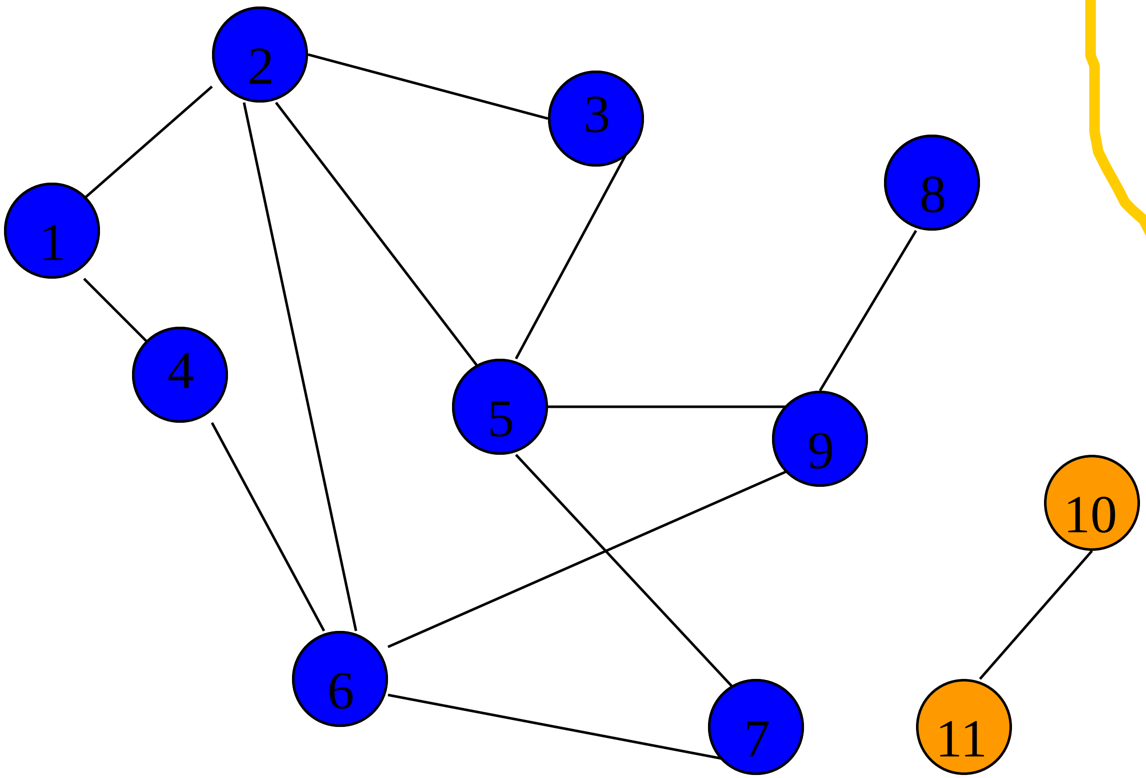
Breadth-First Search Example

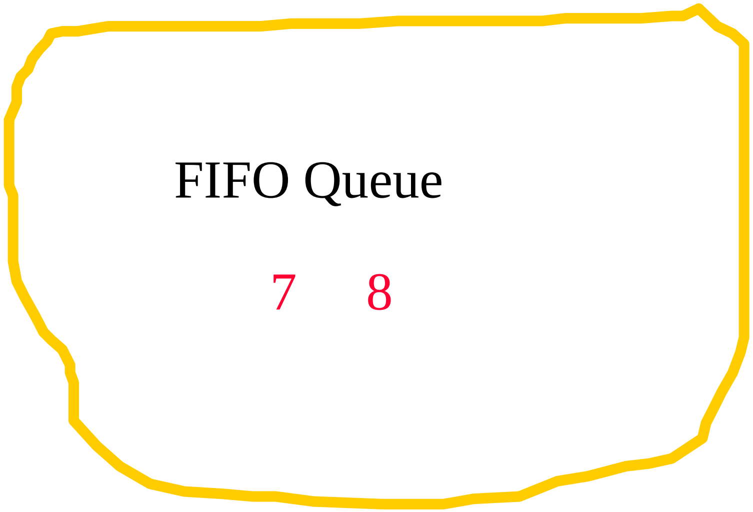




Remove 9 from Q; visit adjacent unvisited vertices;
put in Q.

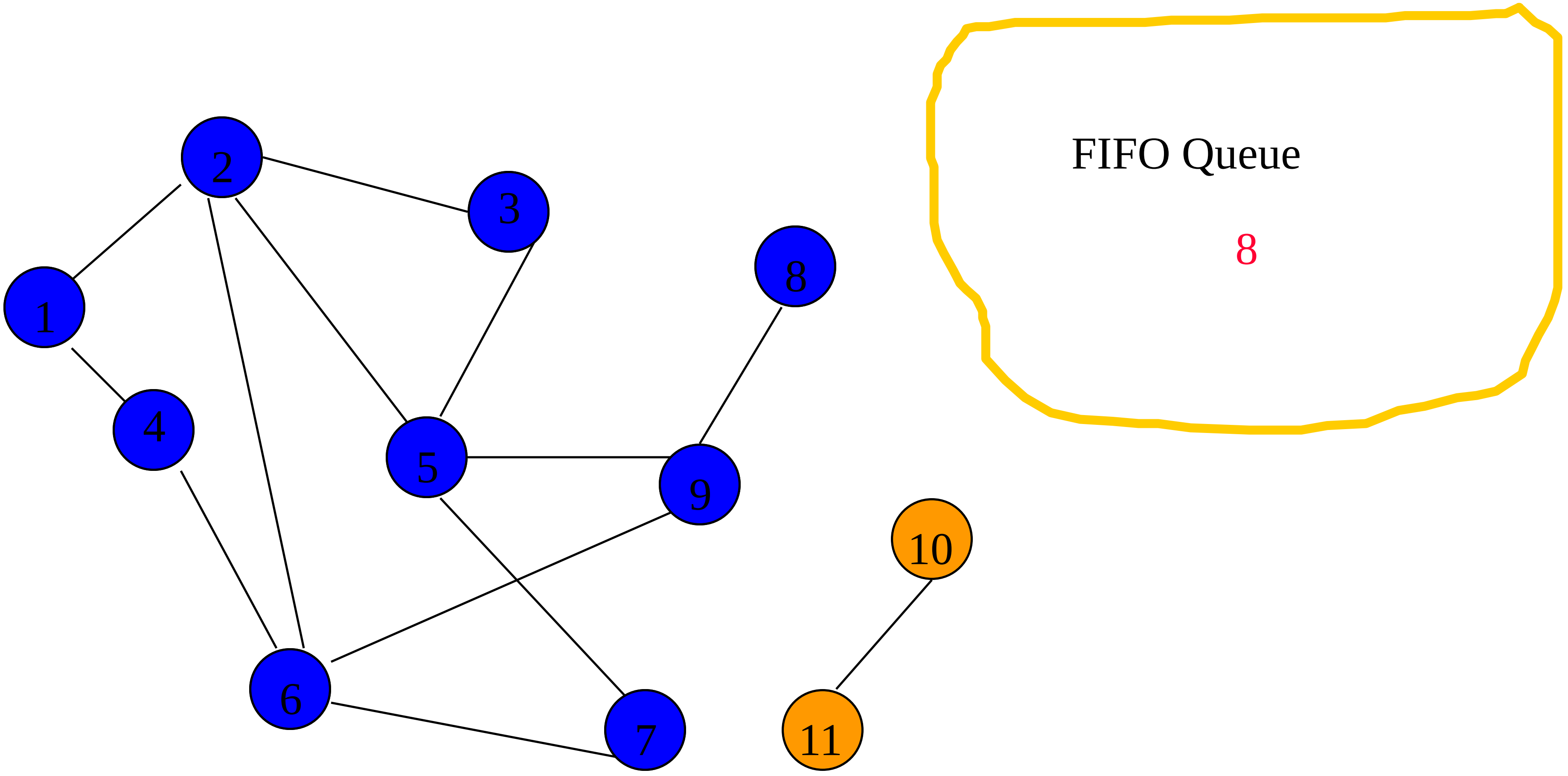
Breadth-First Search Example





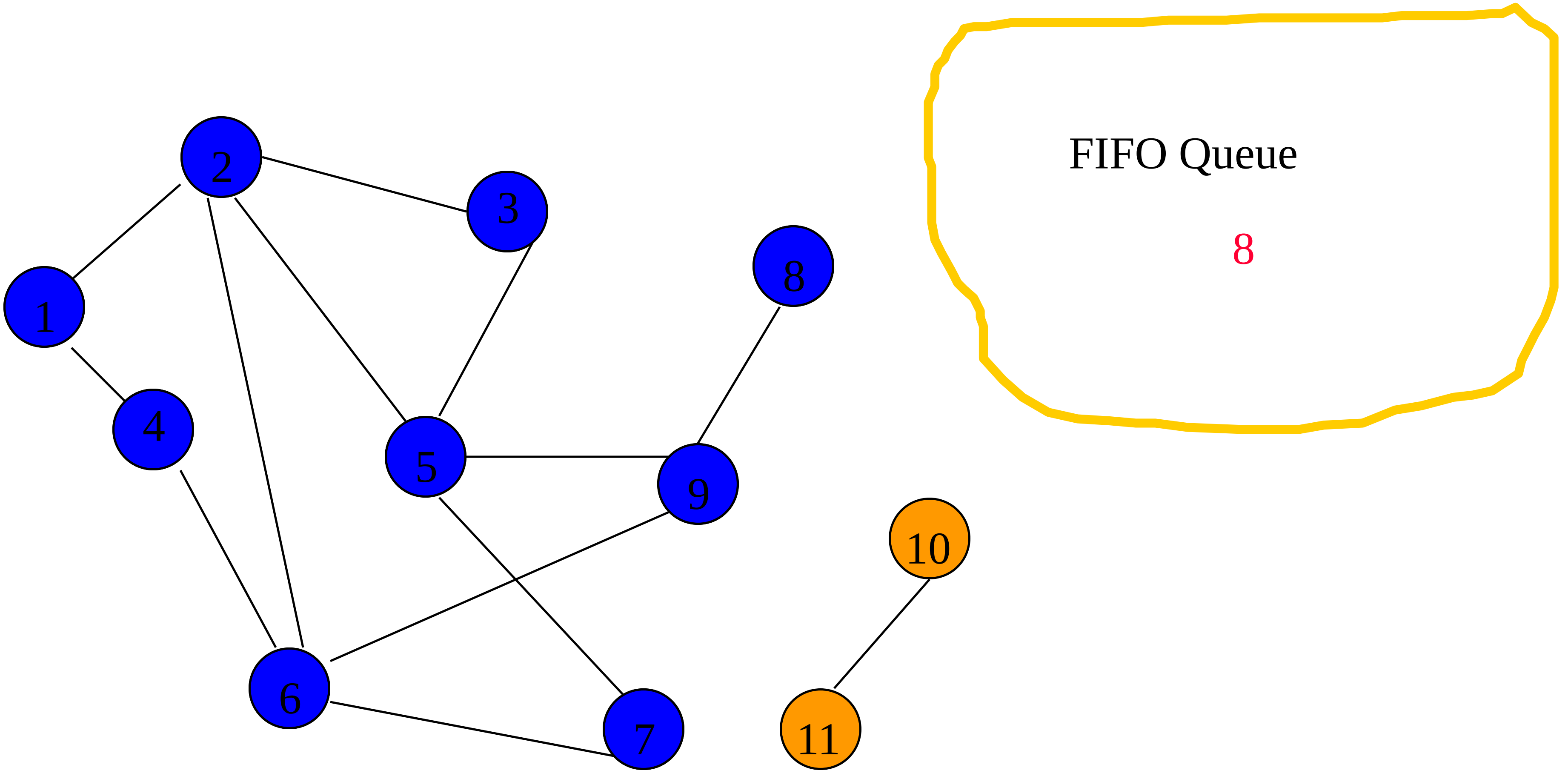
Remove 7 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



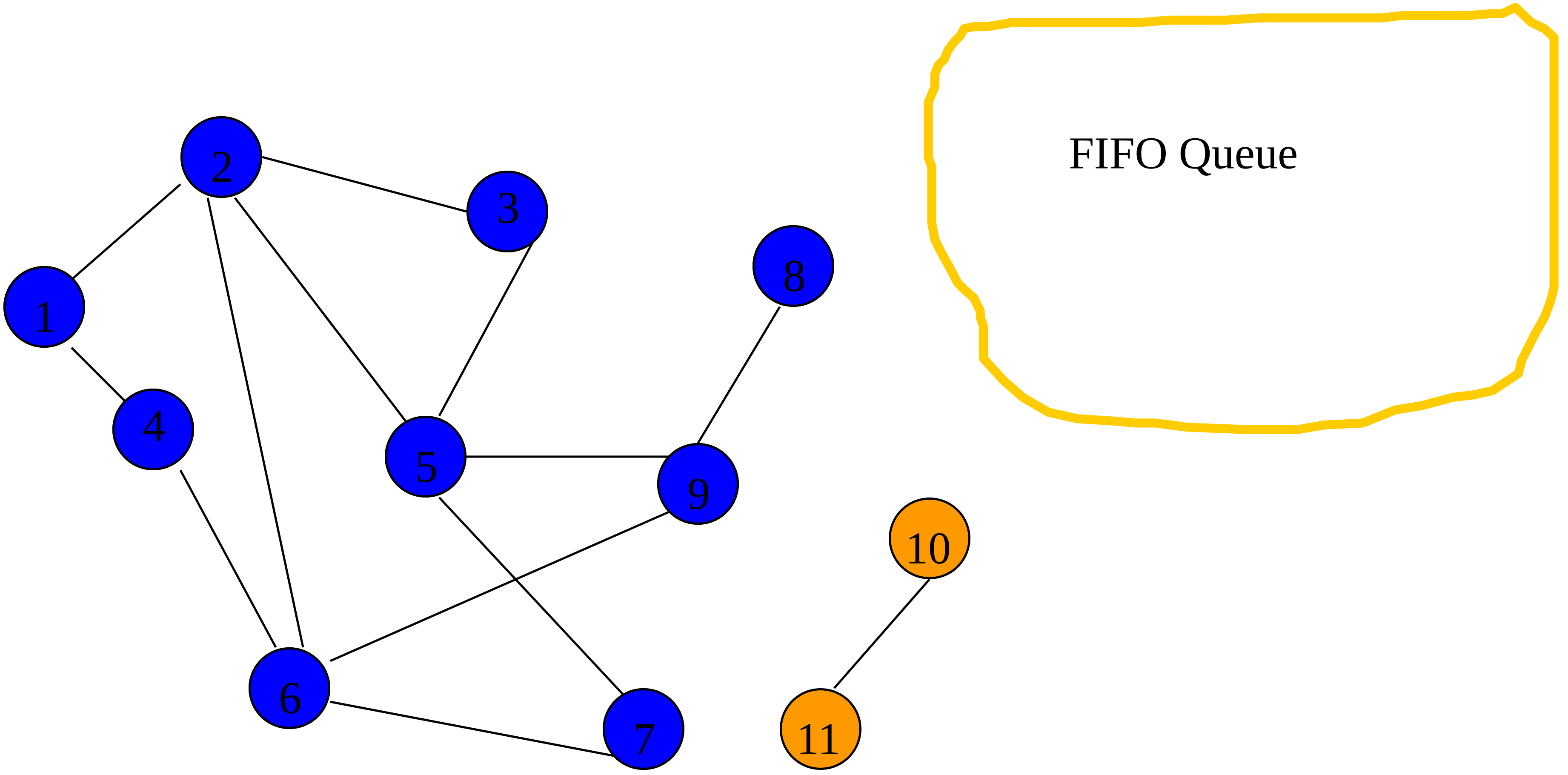
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



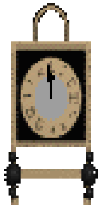
Remove 8 from Q; visit adjacent unvisited vertices;
put in Q.

Breadth-First Search Example



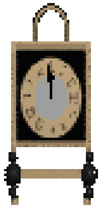
Queue is empty. Search terminates.

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
- Total time
  - O(mn), where m is number of vertices in the component that is searched (adjacency matrix)

Time Complexity



- O(n + sum of component vertex degrees) (adj. lists)

  = O(n + number of edges in component)

Breadth-First Search Properties

- Same complexity as DFS.
- 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.
- There are problems for which bfs is better than dfs and vice versa.