Chapter 12 Graphs

数据科学与计算机学院　　　黄方军

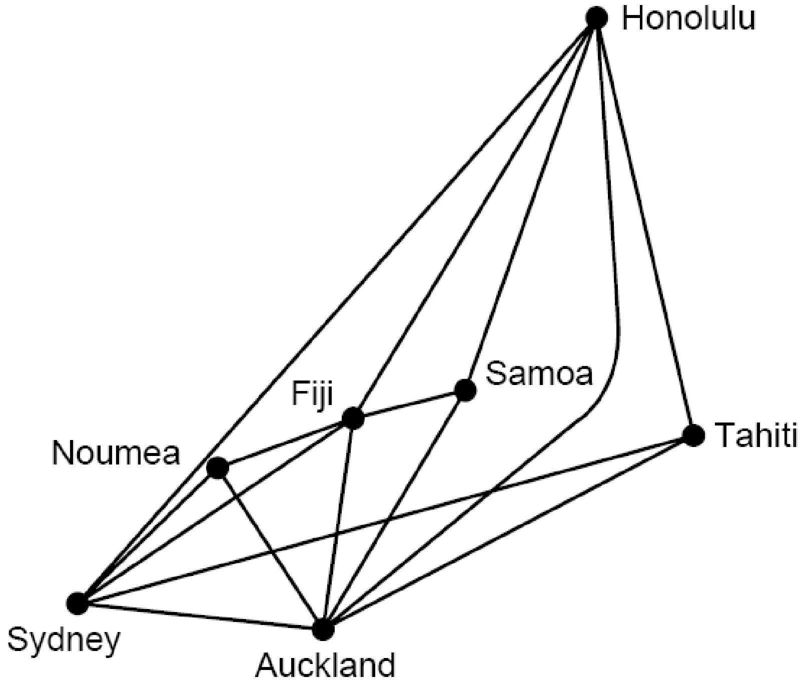
data_structures@163.com

东校区实验中心B502

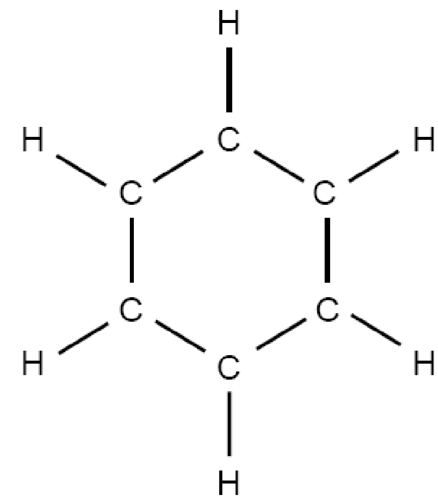
12.1.1 Definitions and Examples

- Terms: vertex, edge, adjacent, incident, degree, cycle, path, connected component, spanning tree;
- Three types of graphs: undirected, directed, weighted;
- Common graph representations: adjacency matrix, adjacency lists.

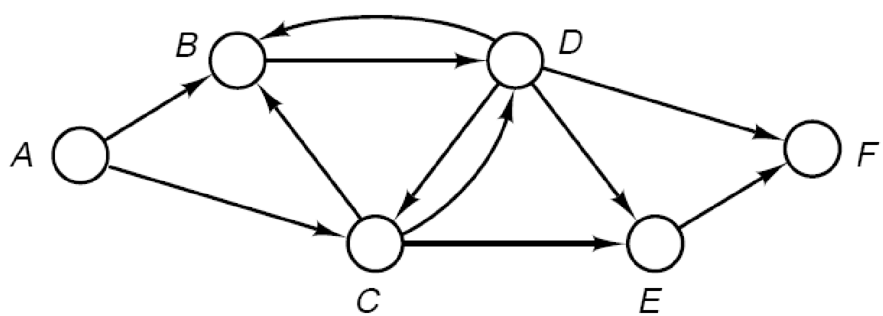
12.1.1 Definitions and Examples



Selected South Pacific air routes



Benzene molecule



Message transmission in a network

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).

$$u \longrightarrow v$$

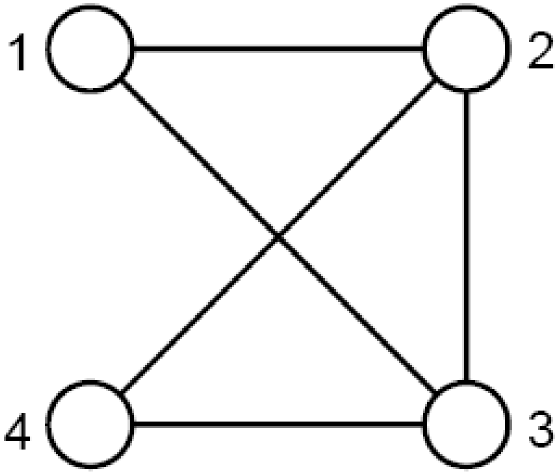
Graphs

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

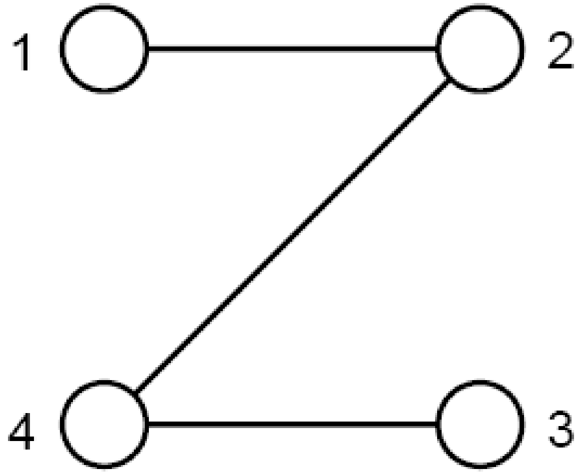
u —— v

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

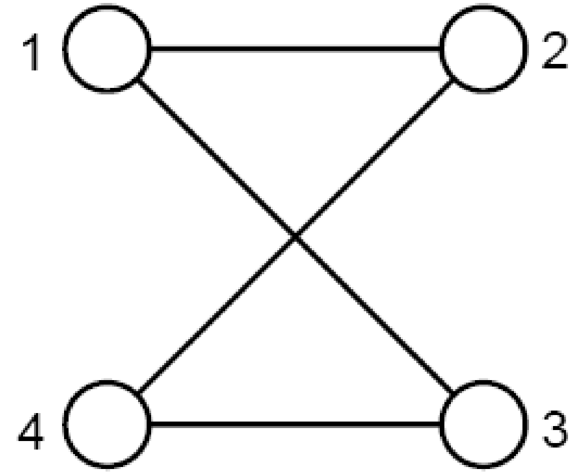
12.1.2 Undirected Graphs



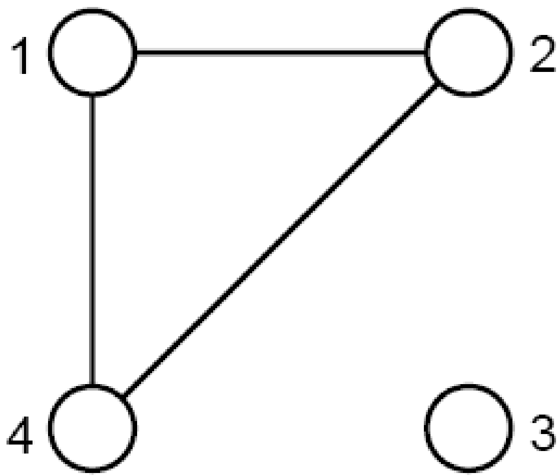
Connected



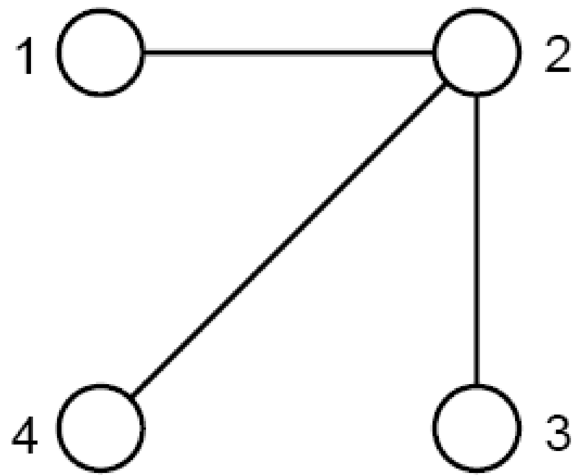
Path



Cycle

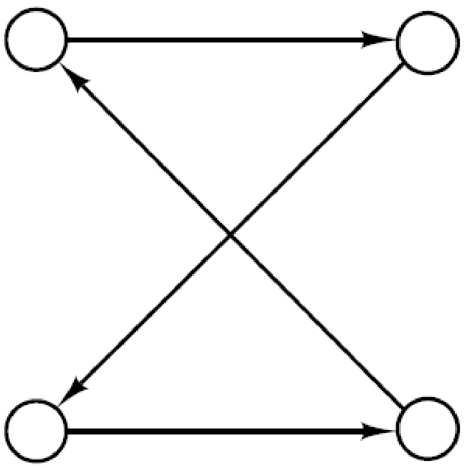


Disconnected



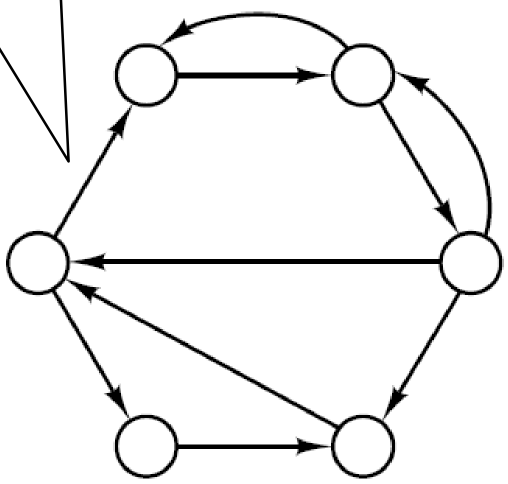
Tree

12.1.3 Directed Graphs

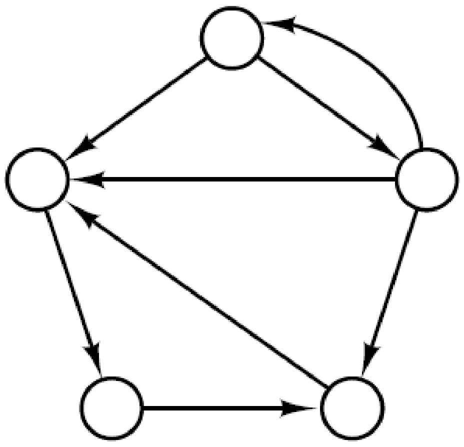


Directed cycle

There is a directed path from any vertex to any other vertex.

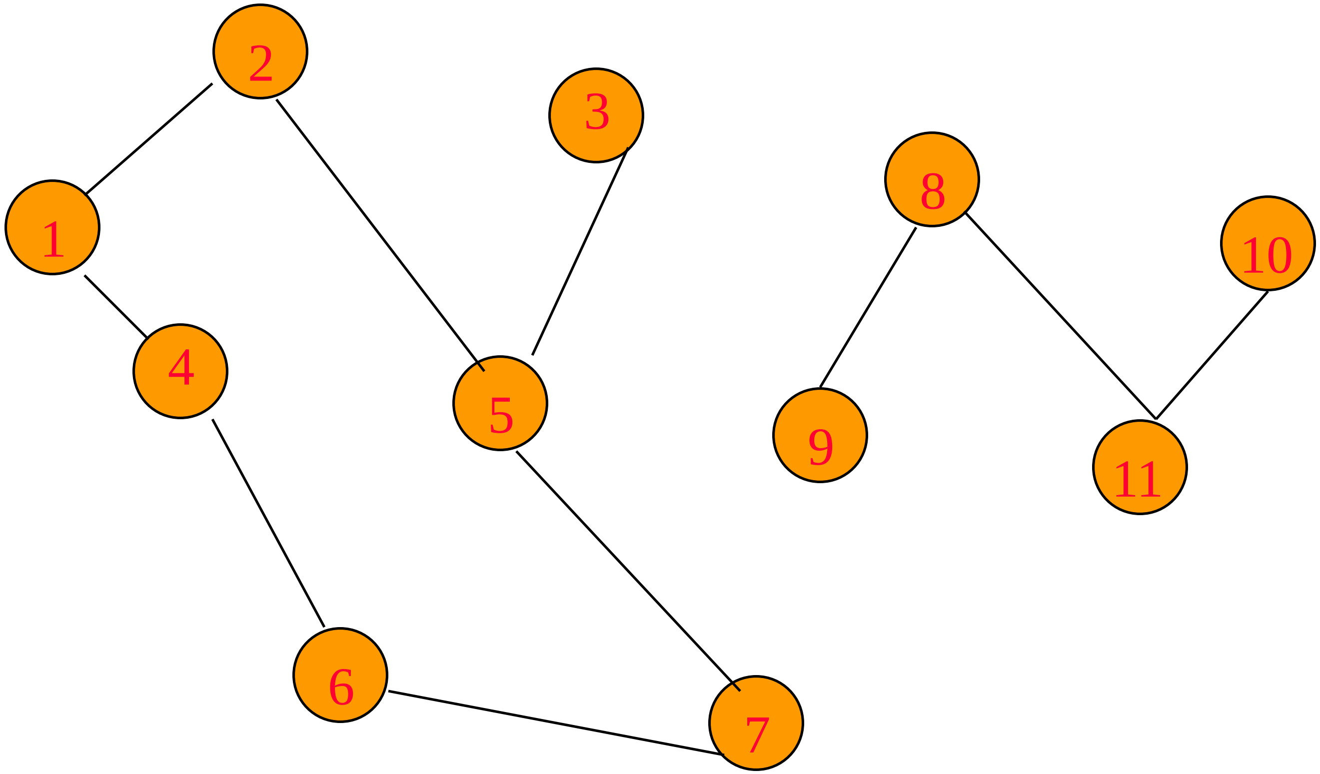


Strongly connected



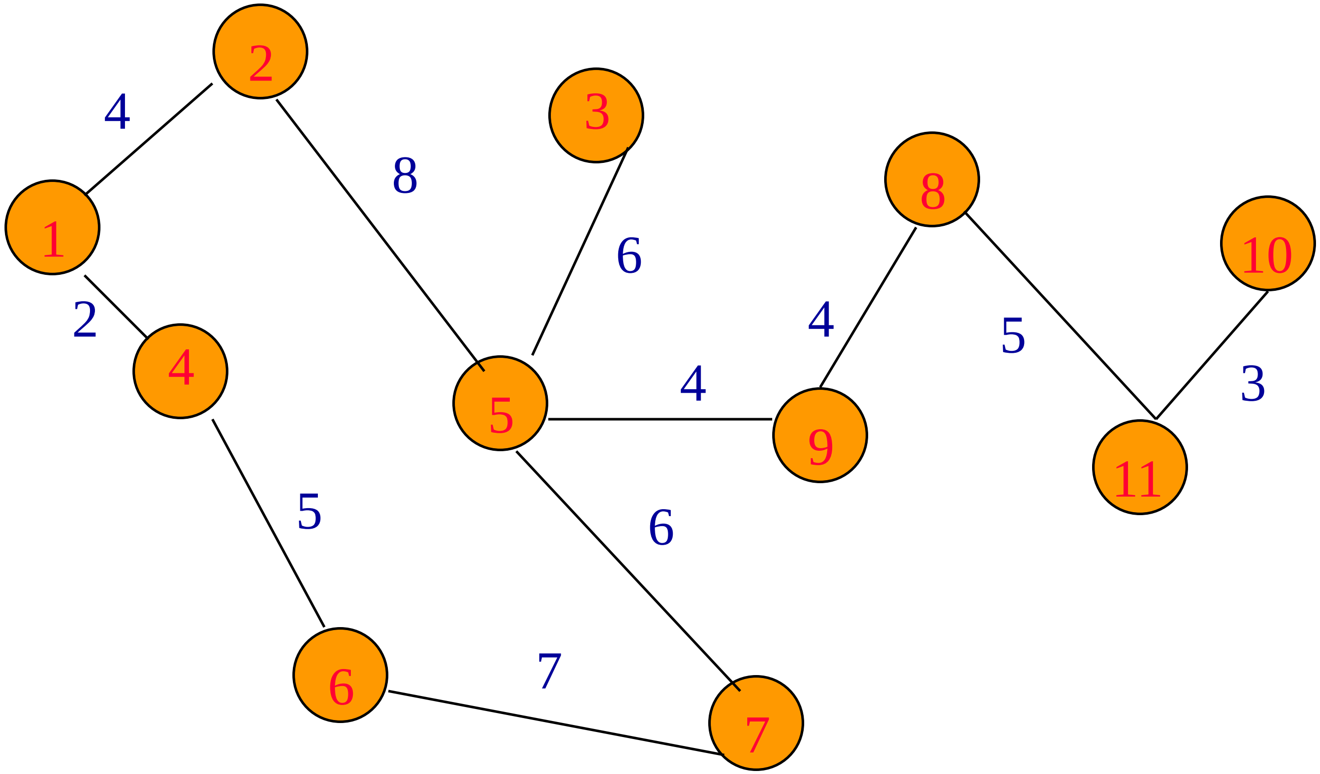
Weakly connected

Applications—Communication Network



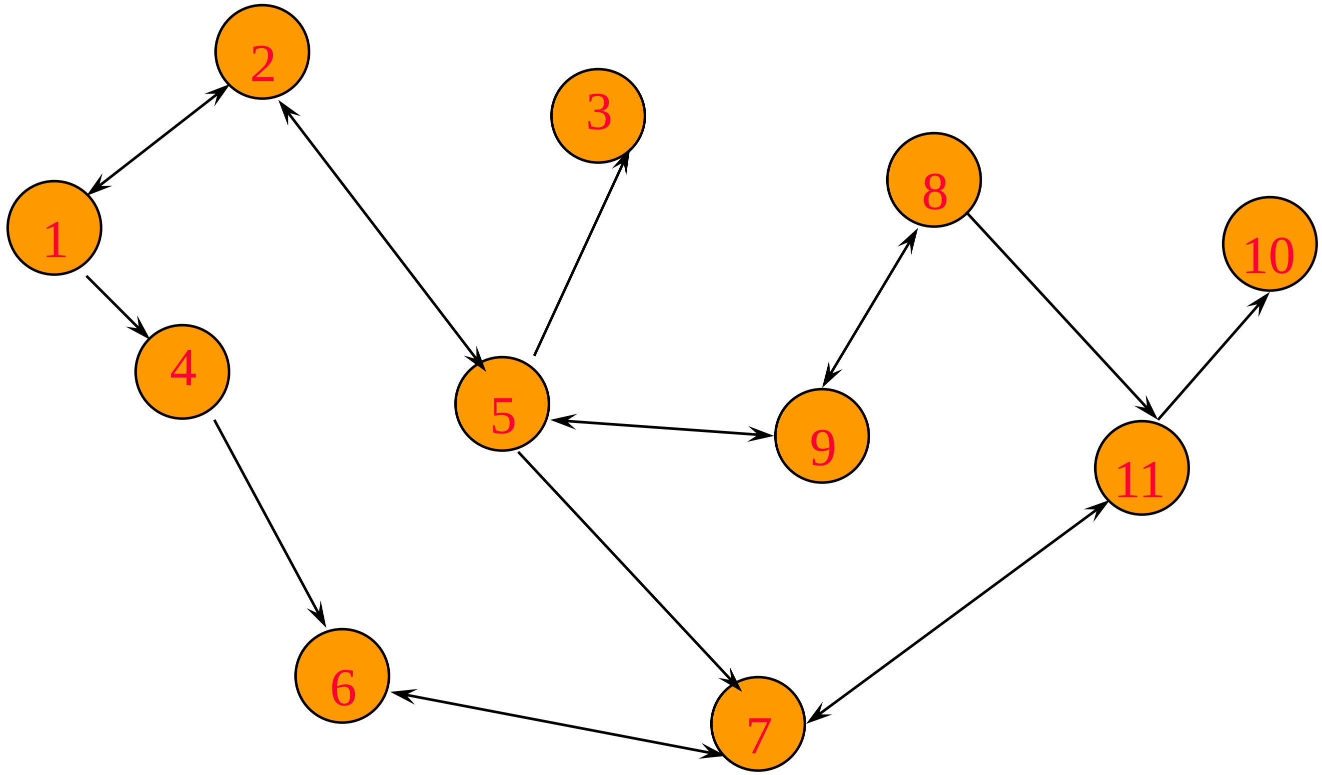
- Vertex = city, edge = communication link.

Driving Distance/Time Map



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

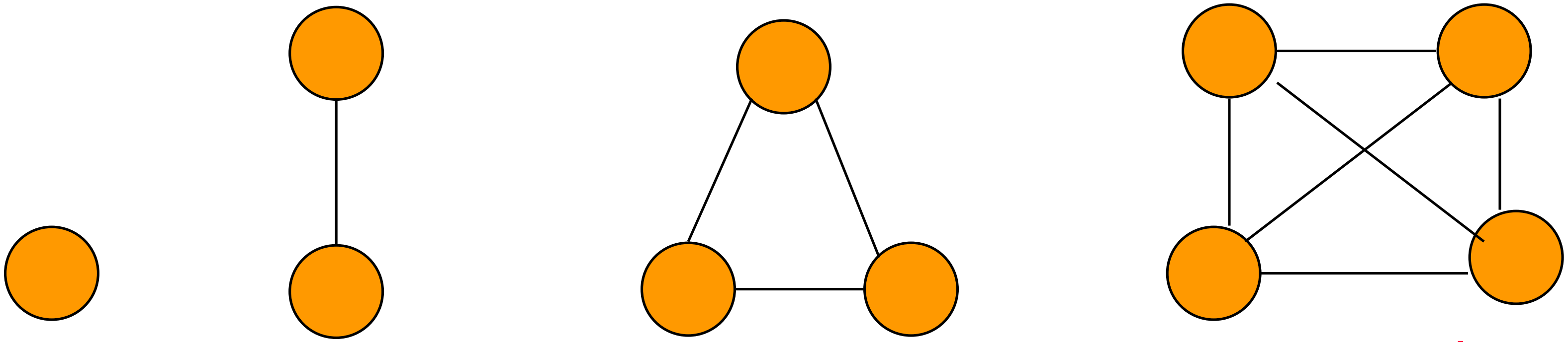
Street Map



- Some streets are one way.

Complete Undirected Graph

Has all possible edges.



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

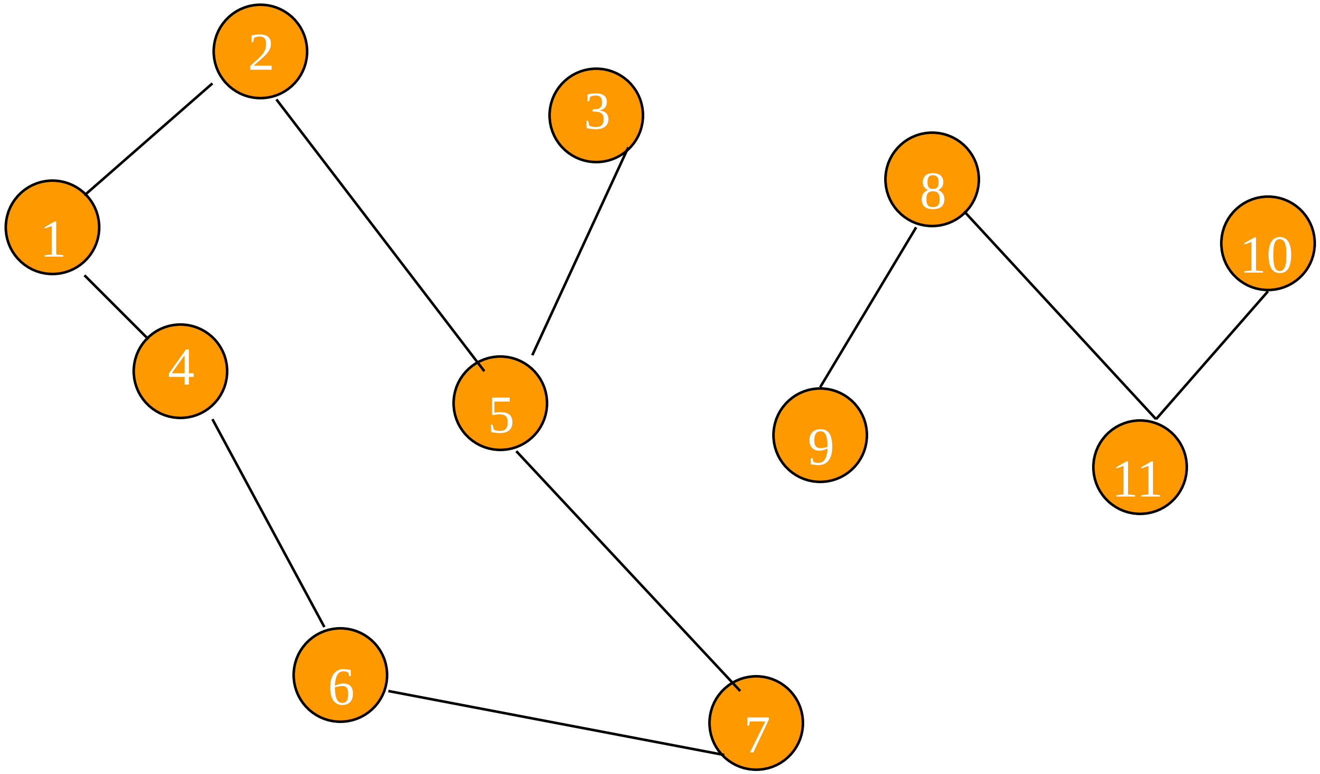
Number Of Edges—Undirected Graph

- 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), $u \mathrel{!}= v$.
- 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 $<=$ $n(n-1)$.

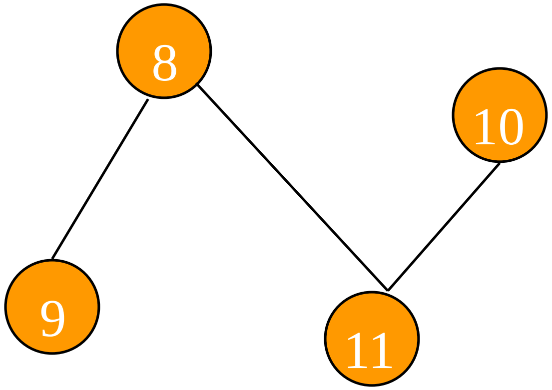
Vertex Degree



Number of edges incident to vertex.

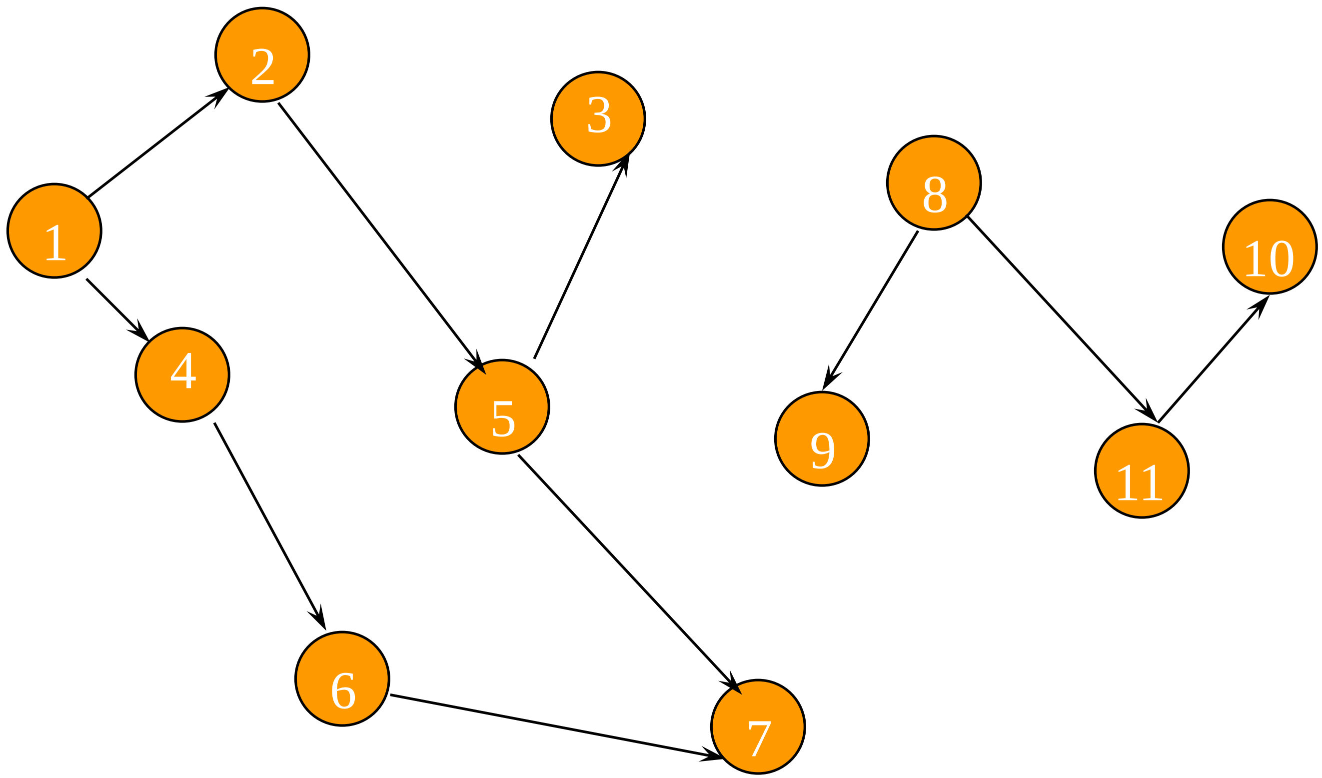
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



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

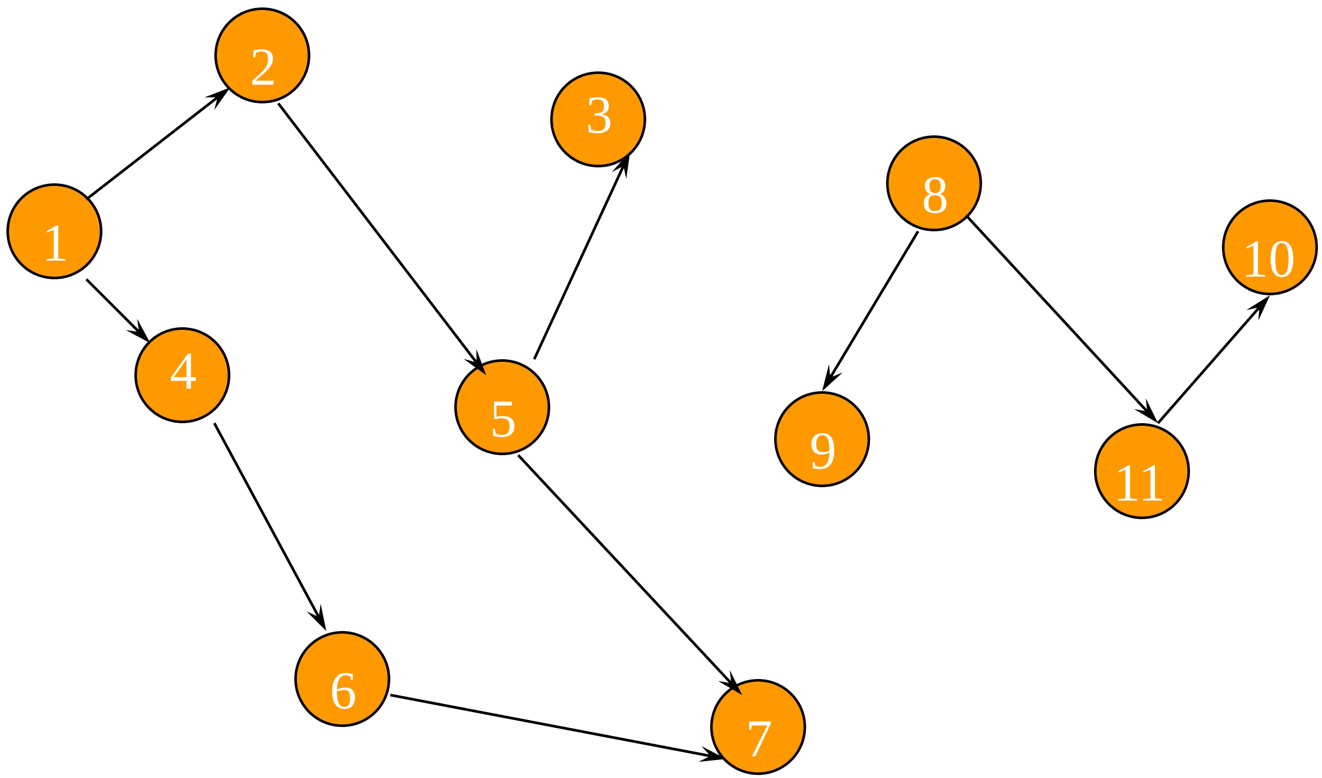
In-Degree Of A Vertex



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

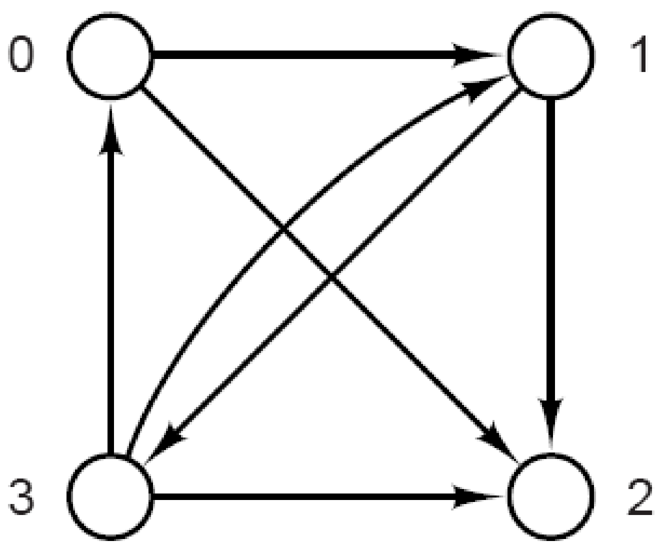
sum of in-degrees = sum of out-degrees = e, where e is the number of edges in the digraph

12.2 Computer Representation

- Adjacency Tables (or Matrixes)
- Adjacency Lists
  - Linked Adjacency Lists
  - Array Adjacency Lists

12.2.1 The Set Representation

Directed graph



Adjacency sets

vertex	Set
0	{ 1, 2 }
1	{ 2, 3 }
2	∅
3	{ 0, 1, 2 }

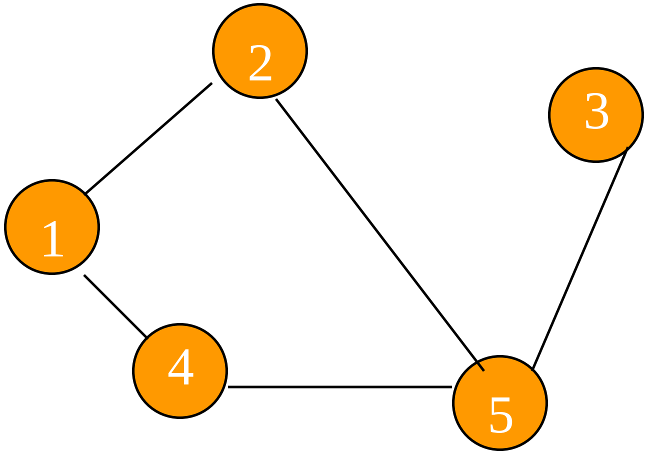
Adjacency table

	0	1	2	3
0	F	T	T	F
1	F	F	T	T
2	F	F	F	F
3	T	T	T	F

Figure 12.4. Adjacency set and an adjacency table

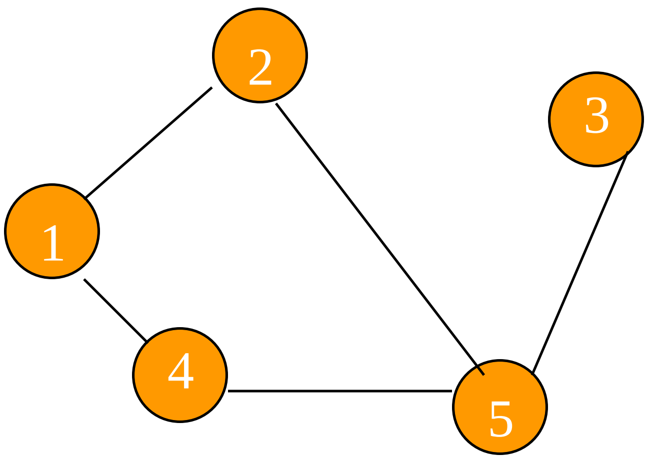
Adjacency Matrix

- 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

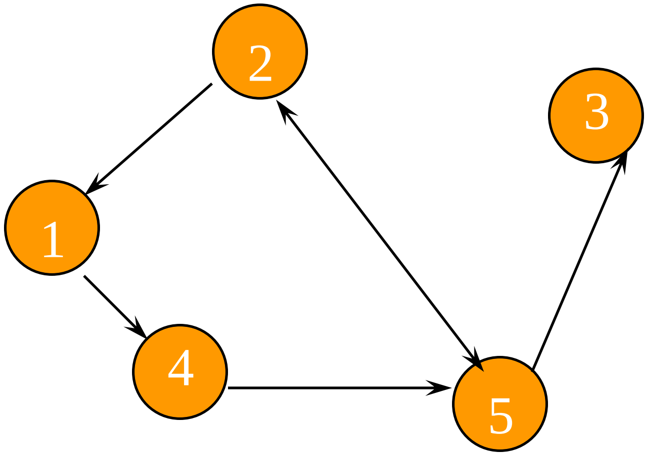
Adjacency Matrix Properties



	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

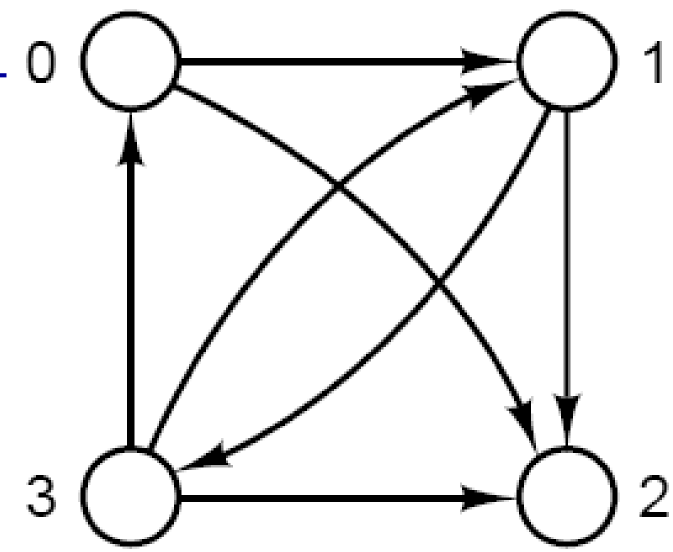
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.

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.

12.2.2 Adjacency Lists

Directed graph



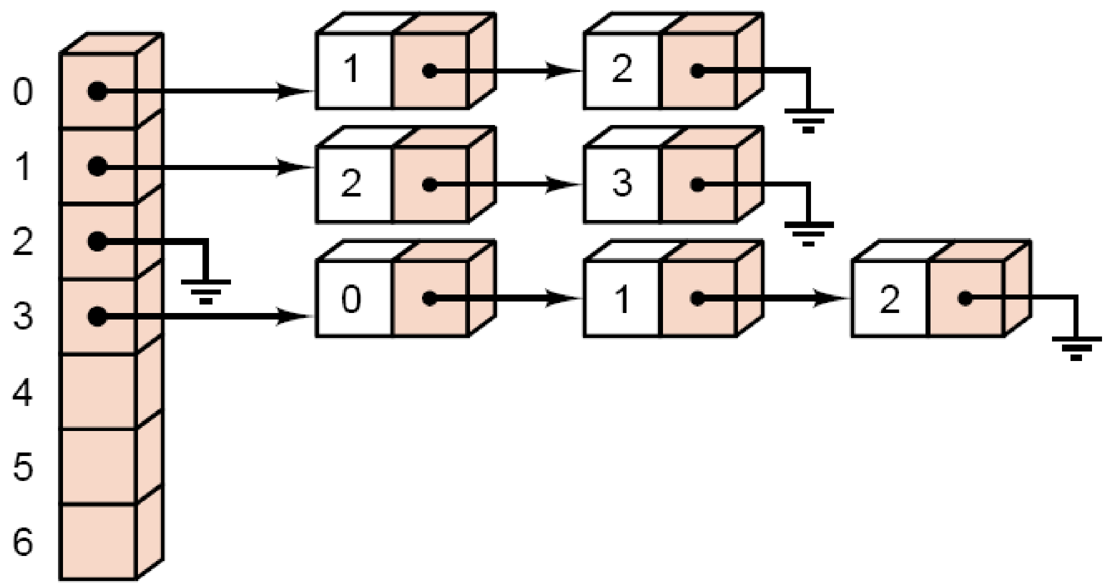
count = 4

vertex	adjacency list						
0	1	2	–	–	–	–	–
1	2	3	–	–	–	–	–
2	–	–	–	–	–	–	–
3	0	1	2	–	–	–	–
4	–	–	–	–	–	–	–
5	–	–	–	–	–	–	–
6	–	–	–	–	–	–	–

(b) Contiguous lists

count = 4

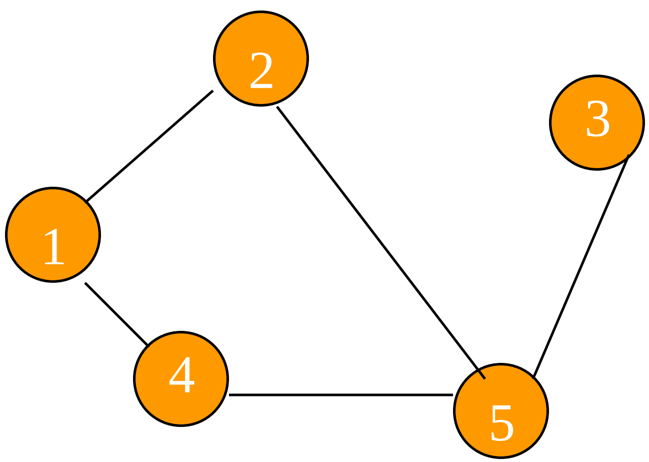
first_edge



(c) Mixed

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)

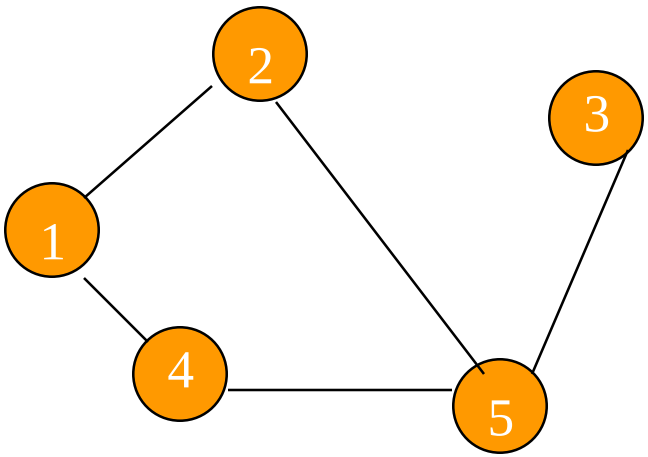
aList[3] = (5)

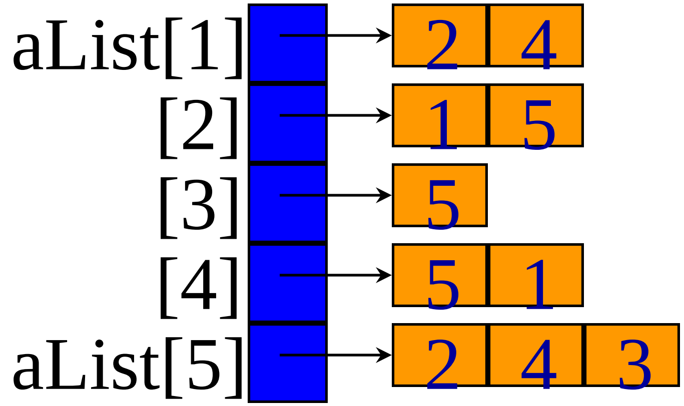
aList[4] = (5,1)

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

Array Adjacency Lists

- Each adjacency list is an array list.





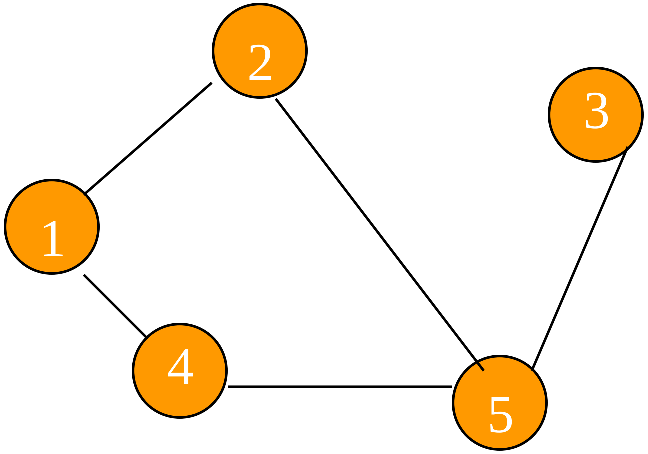
Array Length = n

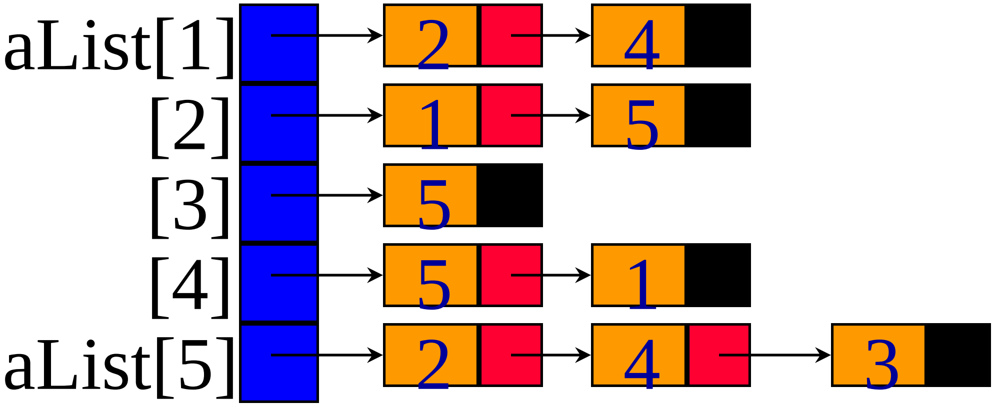
# of list elements = $2e$ (undirected graph)

# of list elements = e (digraph)

Linked Adjacency Lists

- Each adjacency list is a chain.





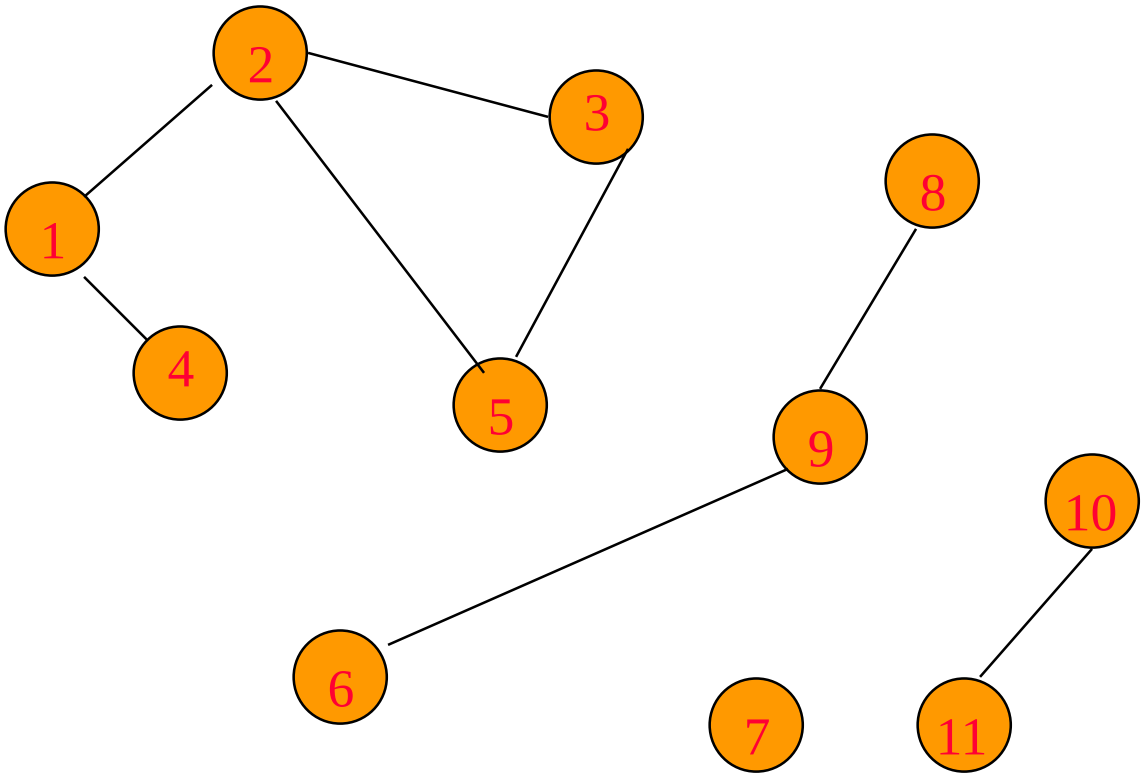
Array Length = n

# of chain nodes = $2e$ (undirected graph)

# of chain nodes = e (digraph)

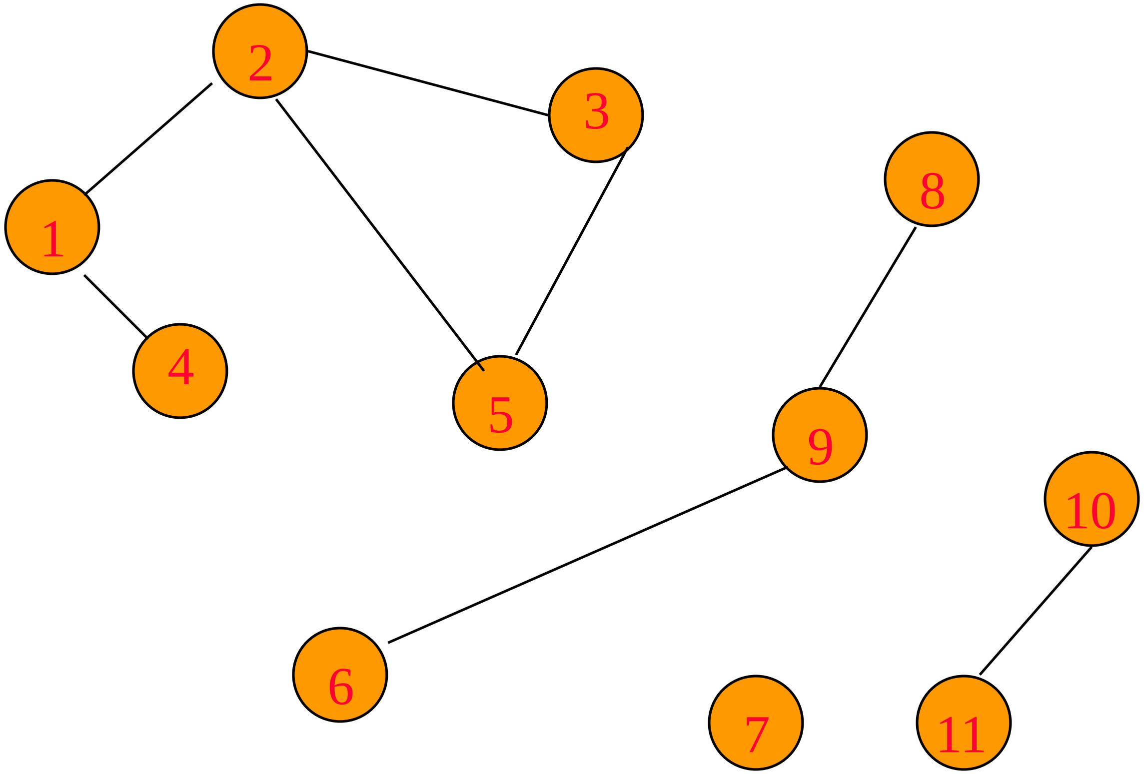
12.3 Graph Traversal

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*.



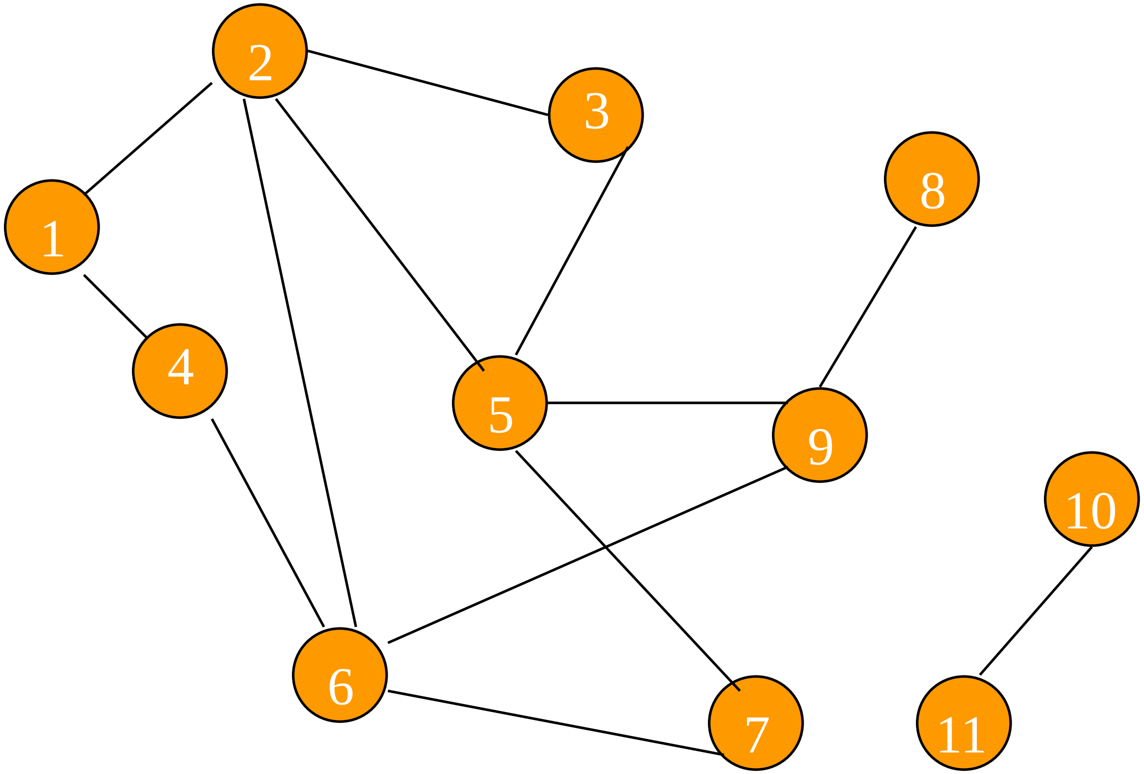
Graph Search Methods

- 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:
  - Breadth-first search.
  - Depth-first search.

Breadth-First Search

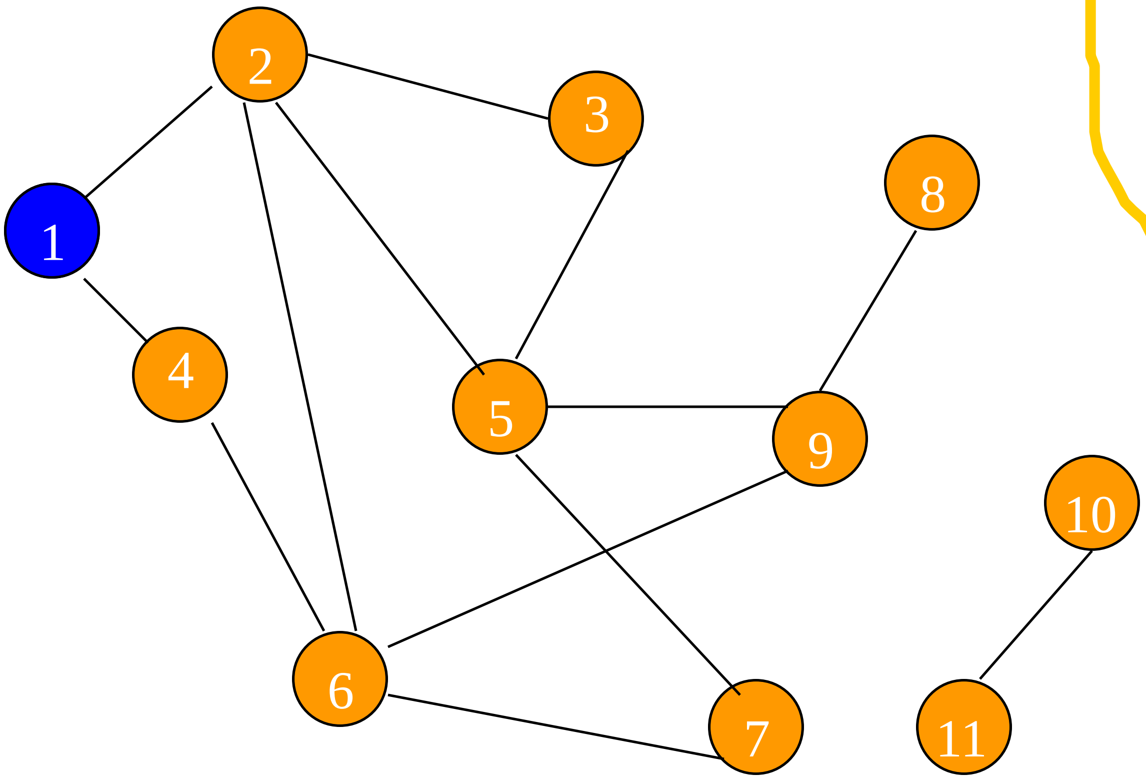
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search Example



Start search at vertex 1.

Breadth-First Search Example

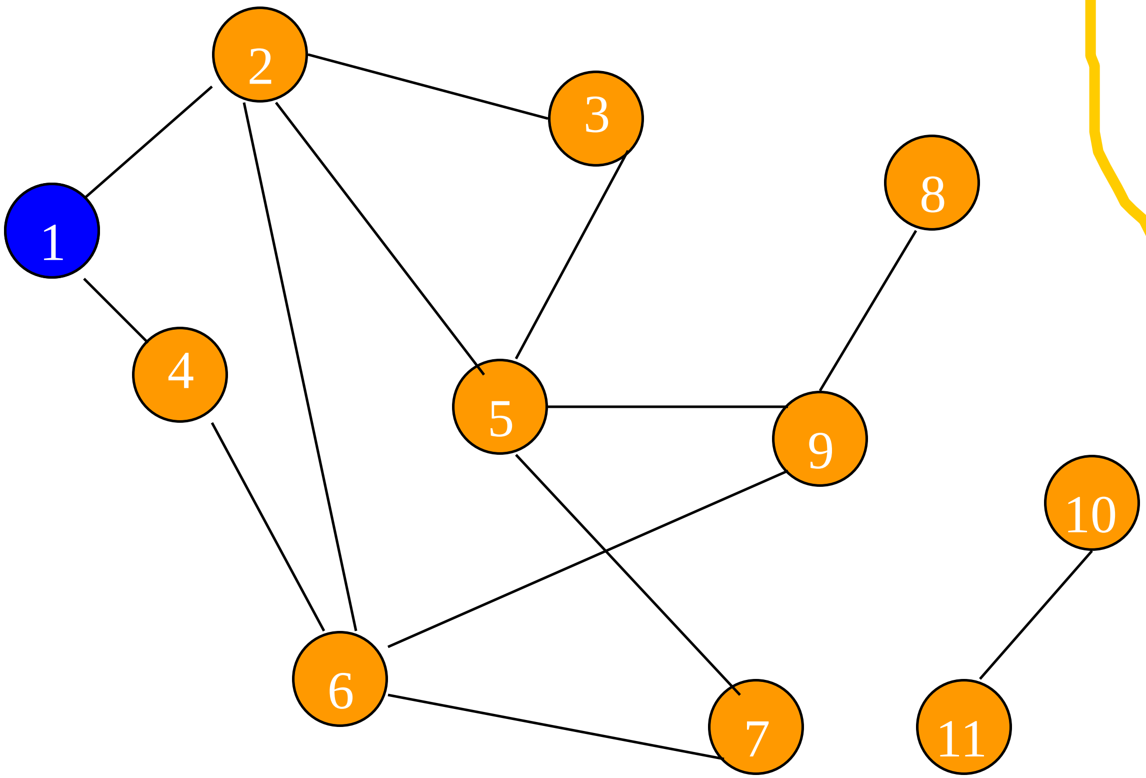


FIFO Queue

1

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

Breadth-First Search Example

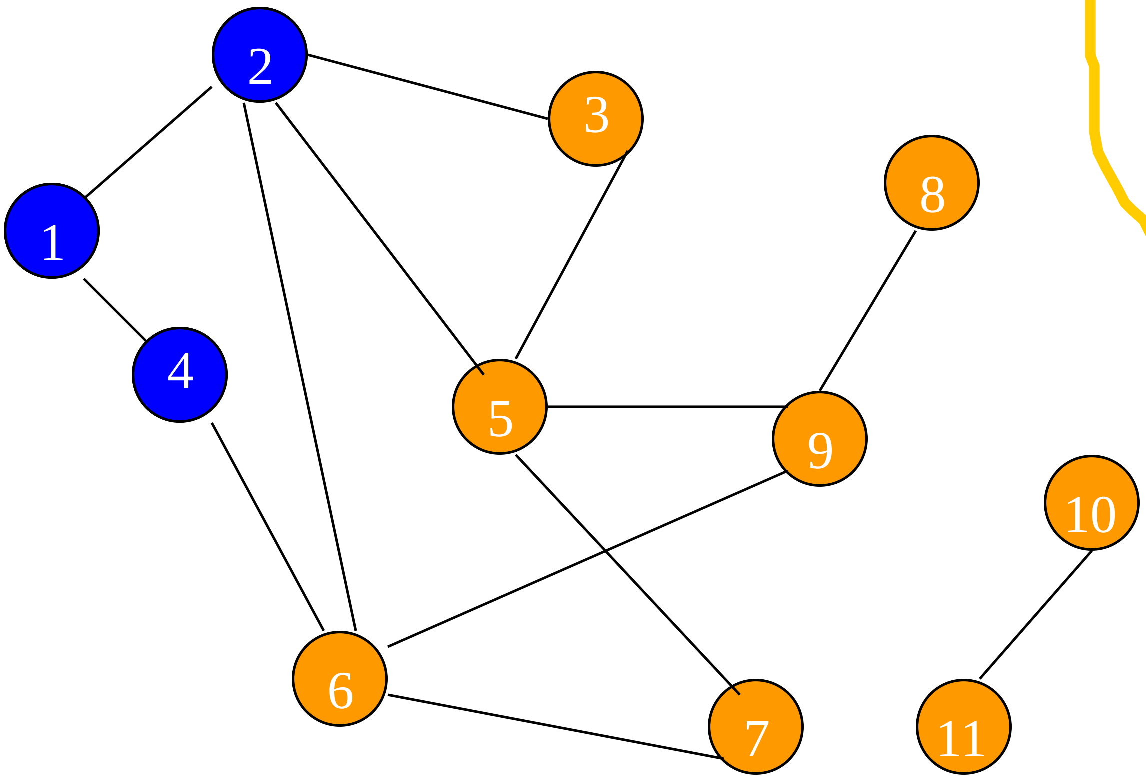


FIFO Queue

1

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

Breadth-First Search Example

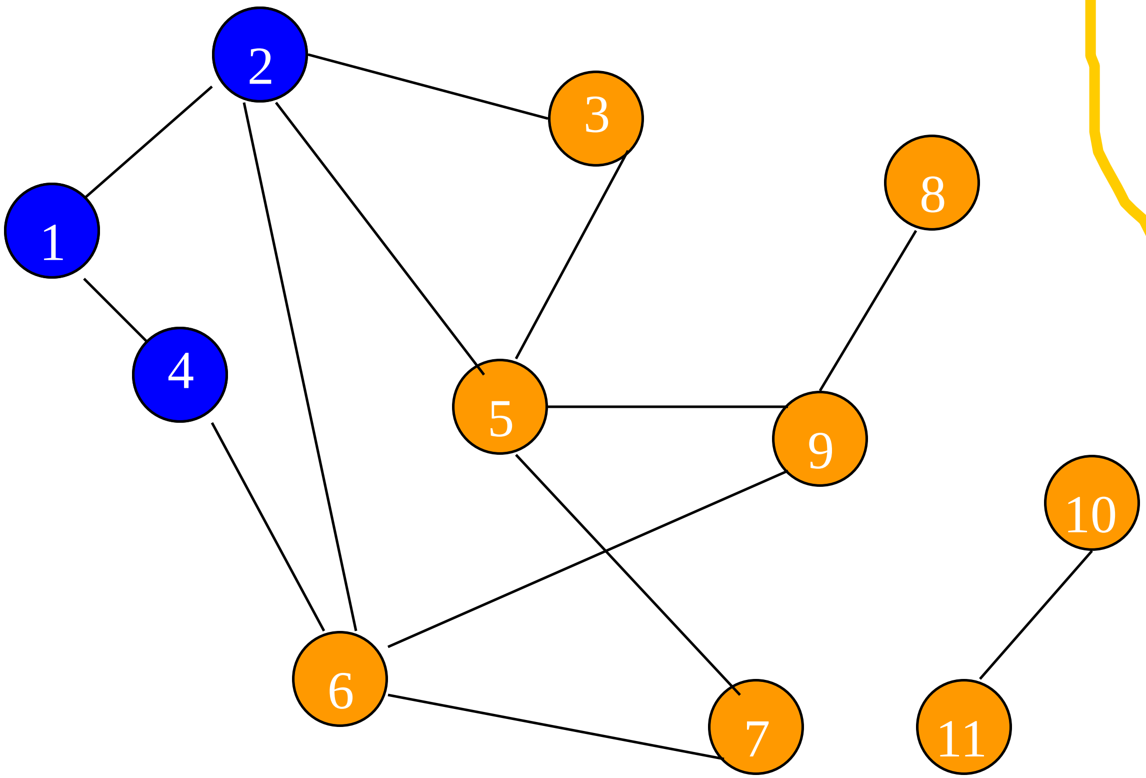


FIFO Queue

2 4

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

Breadth-First Search Example

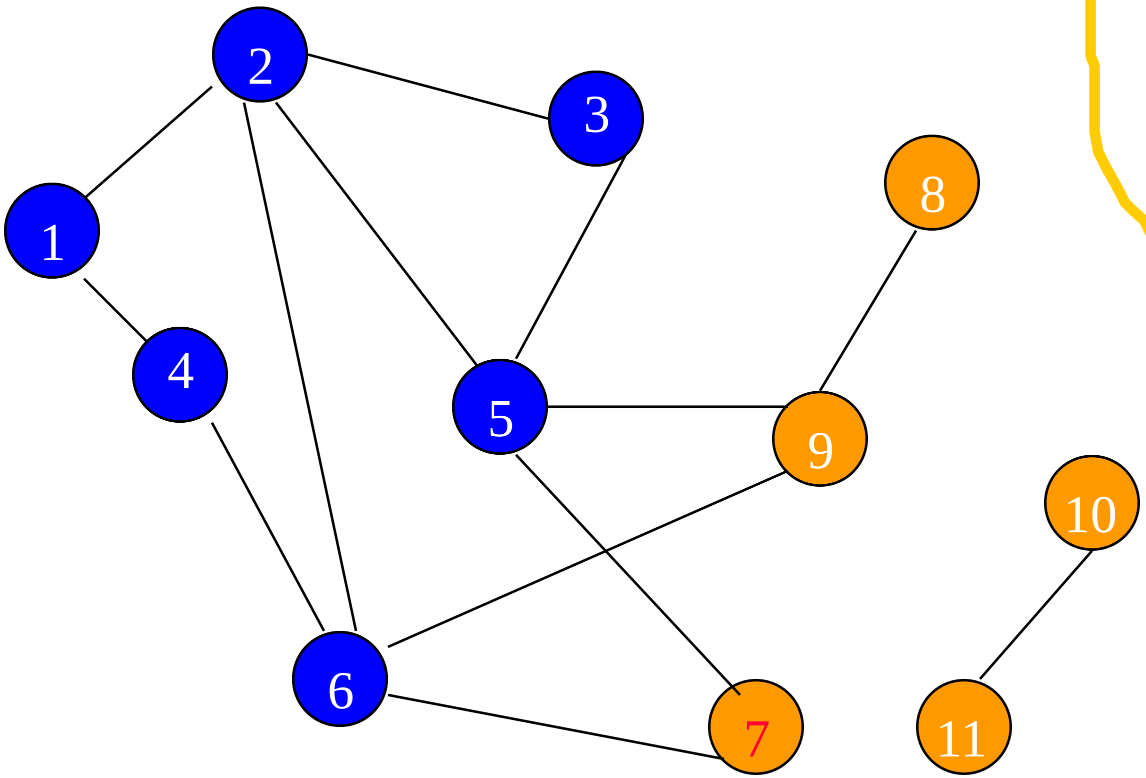


FIFO Queue

2 4

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

Breadth-First Search Example

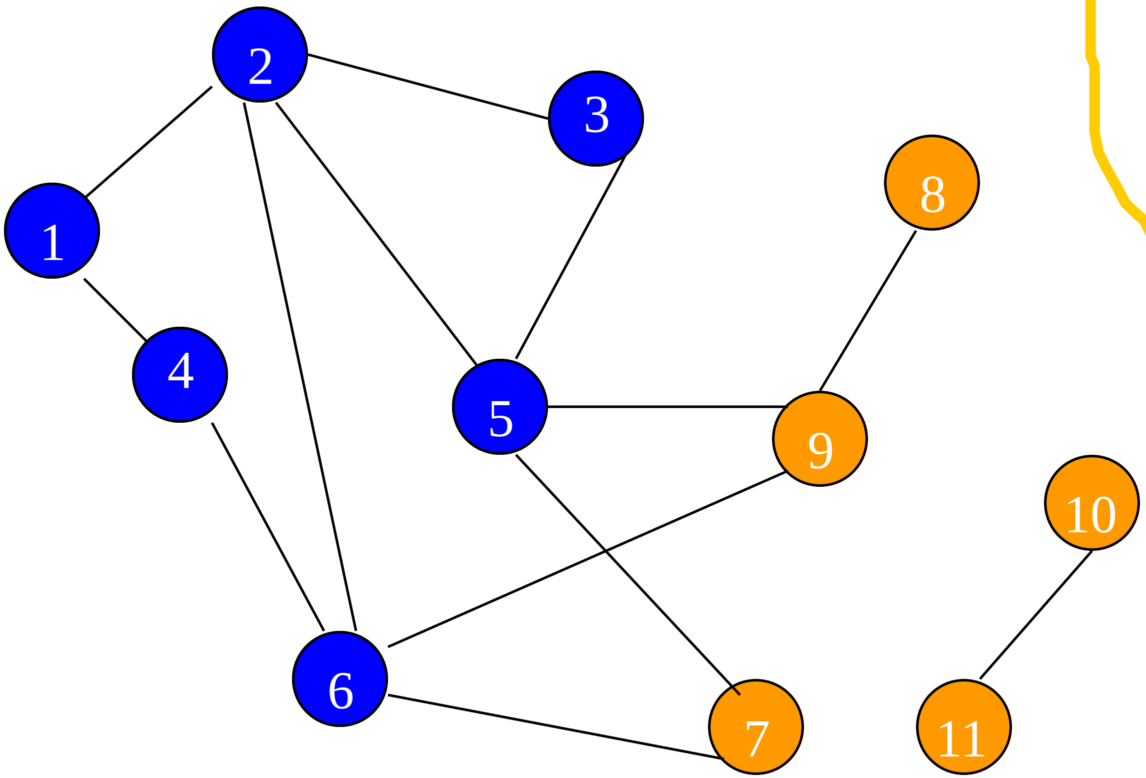


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

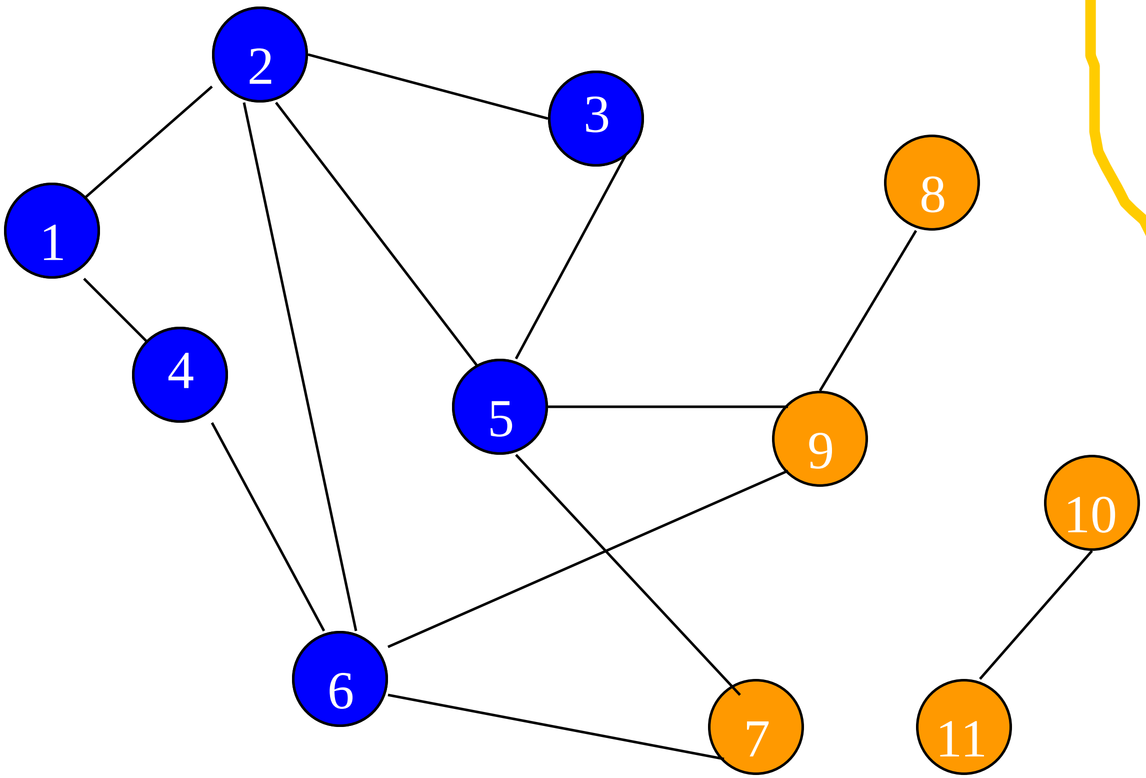


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

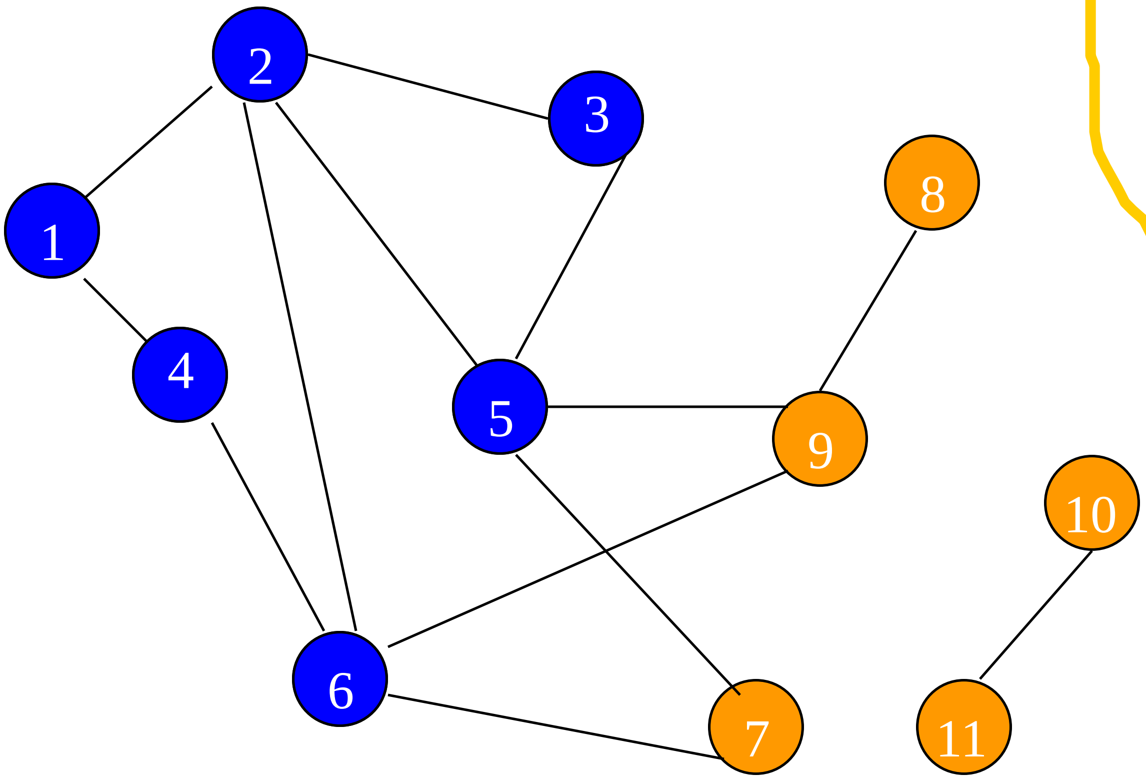


FIFO Queue

5 3 6

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

Breadth-First Search Example

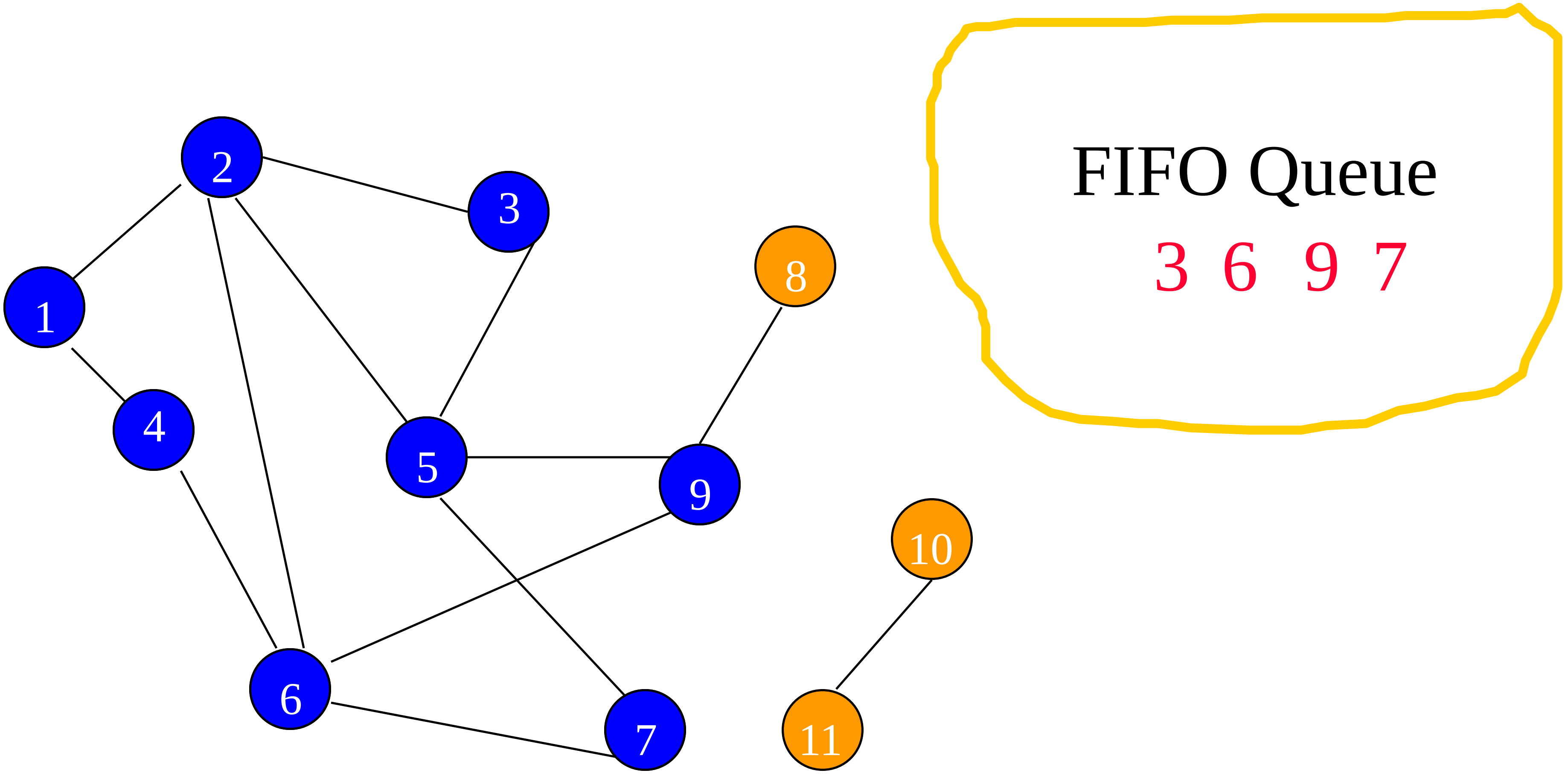


FIFO Queue

5 3 6

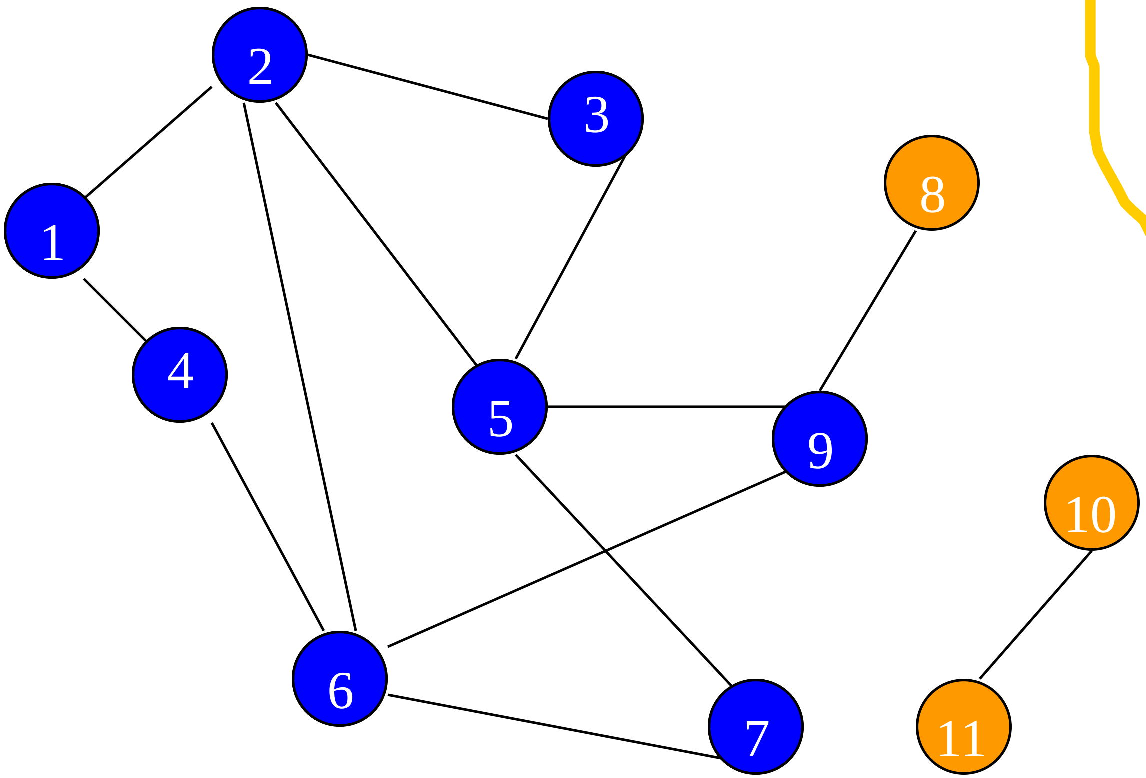
Remove 5 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



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

Breadth-First Search Example

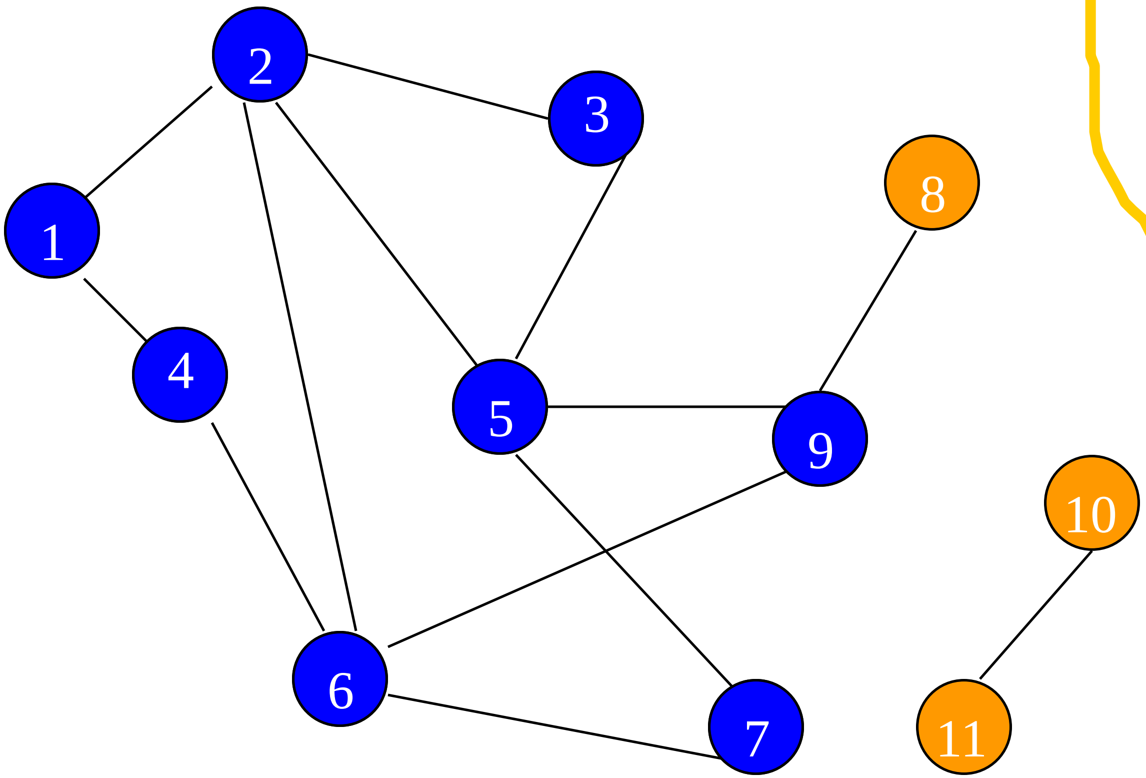


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

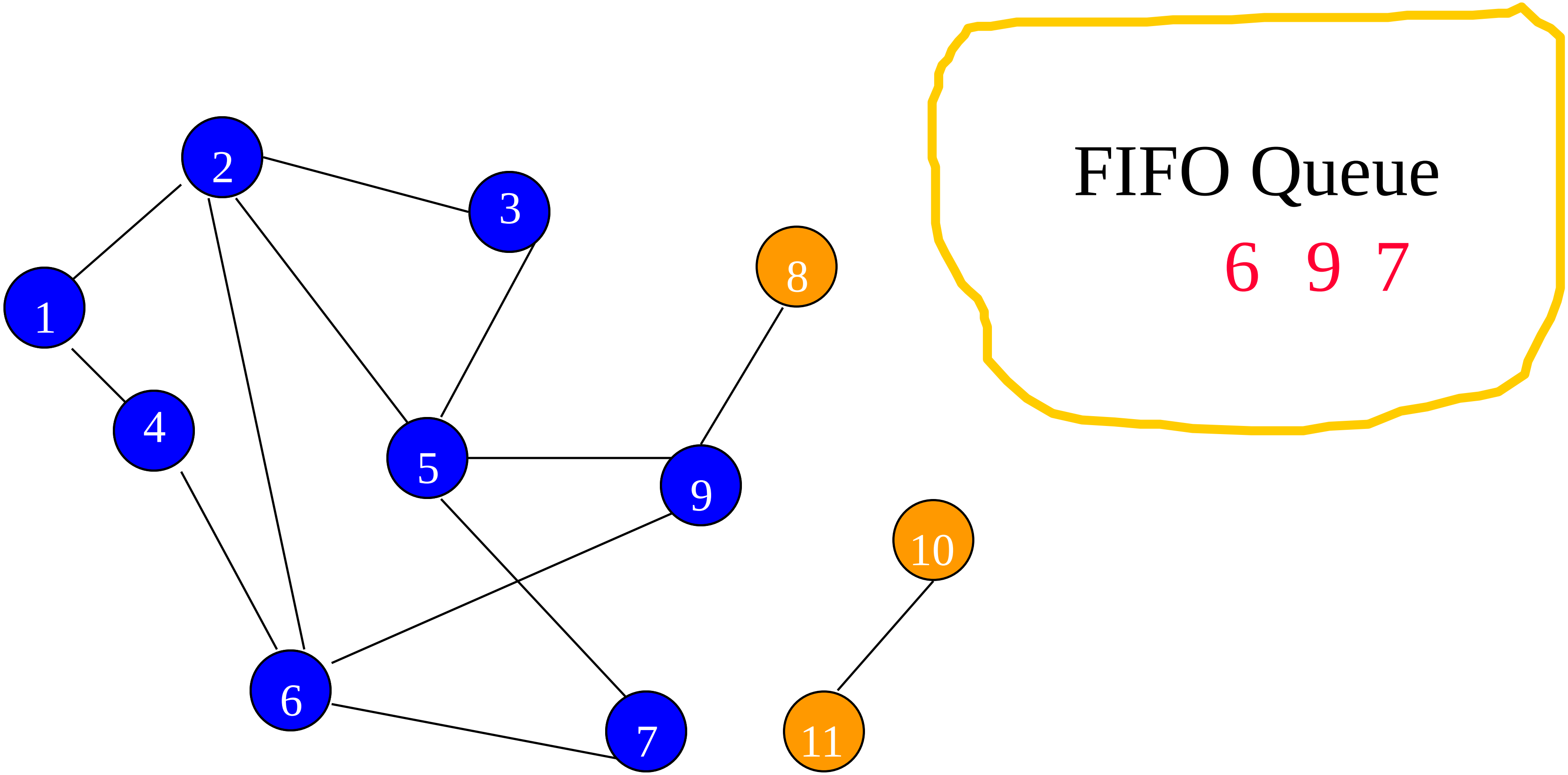


FIFO Queue

6 9 7

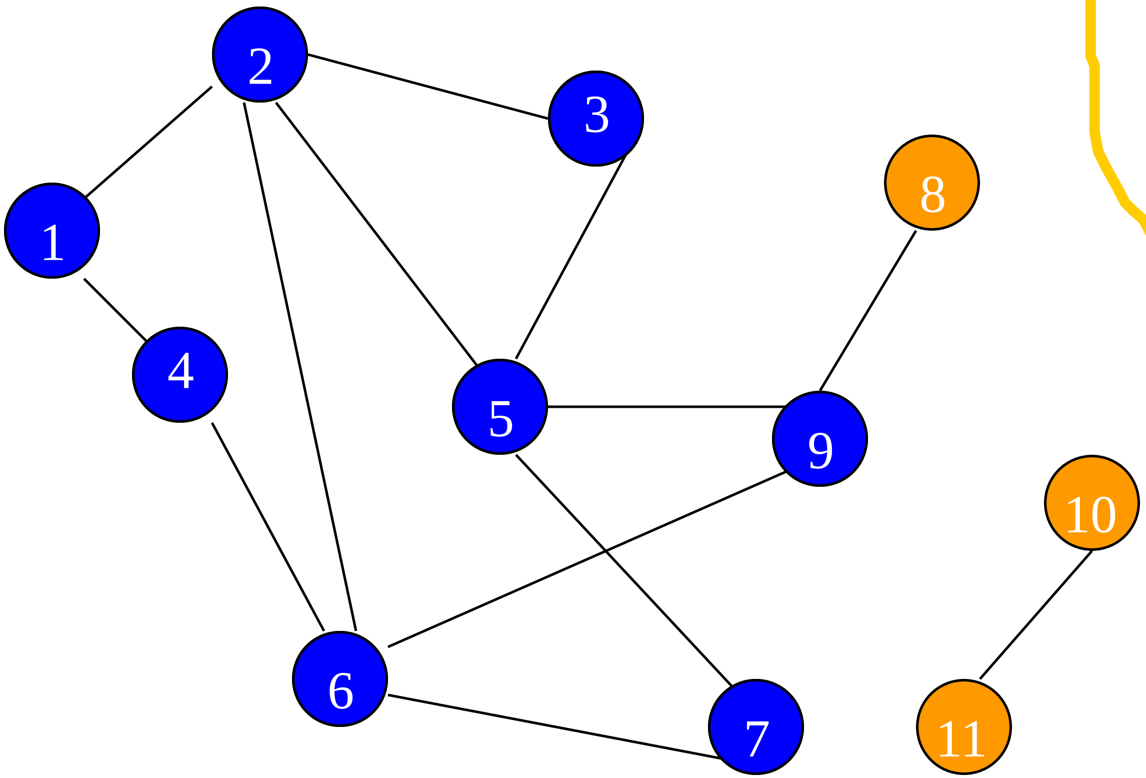
Remove 3 from Q; visit adjacent unvisited vertices; put in Q.

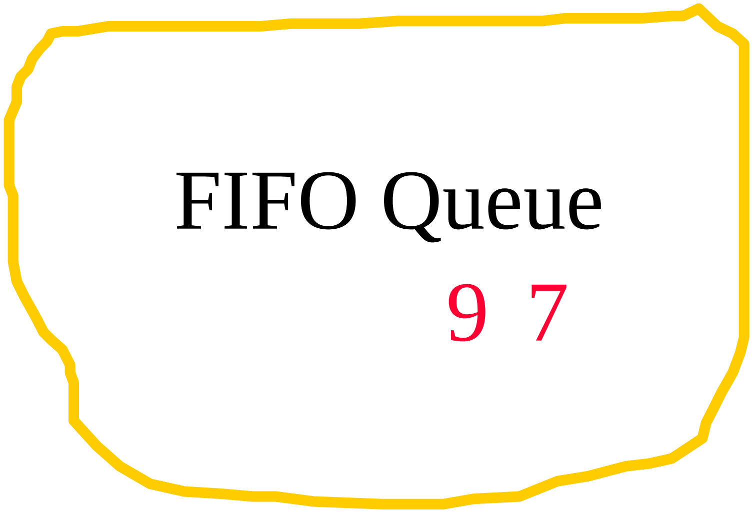
Breadth-First Search Example



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

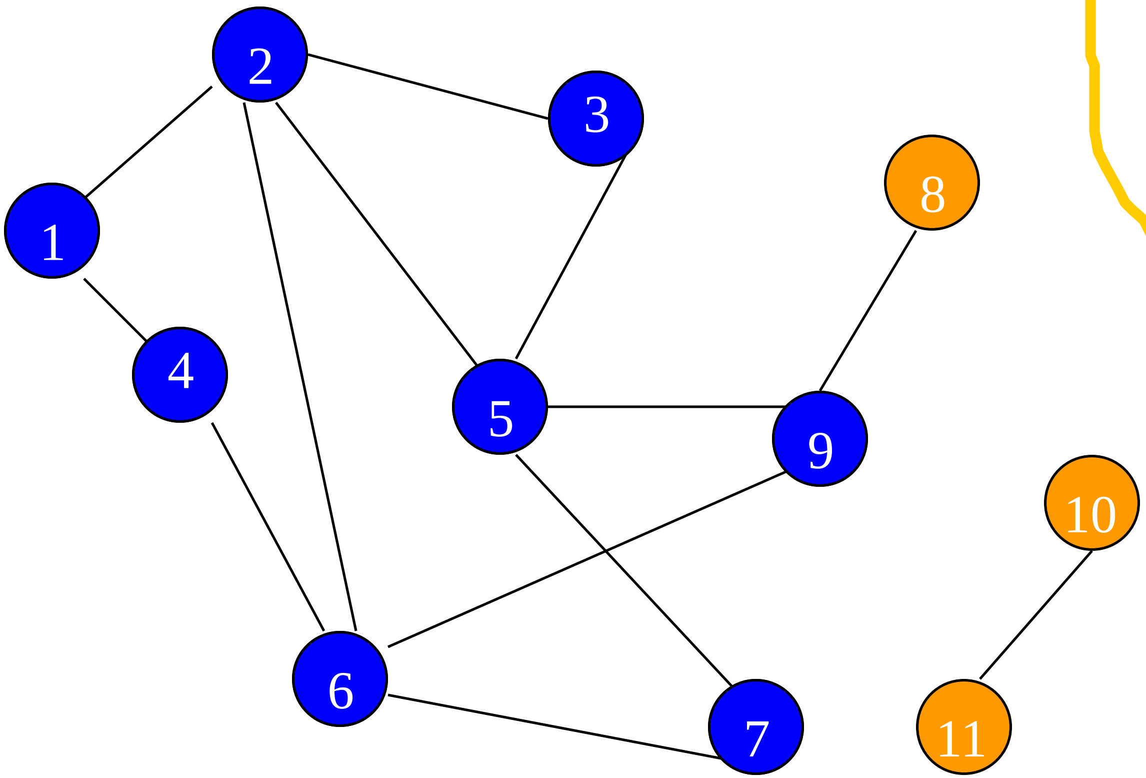
Breadth-First Search Example

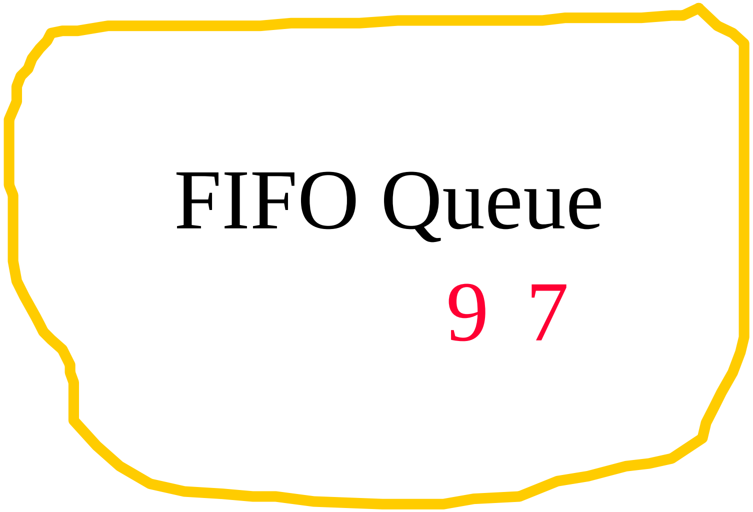




Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

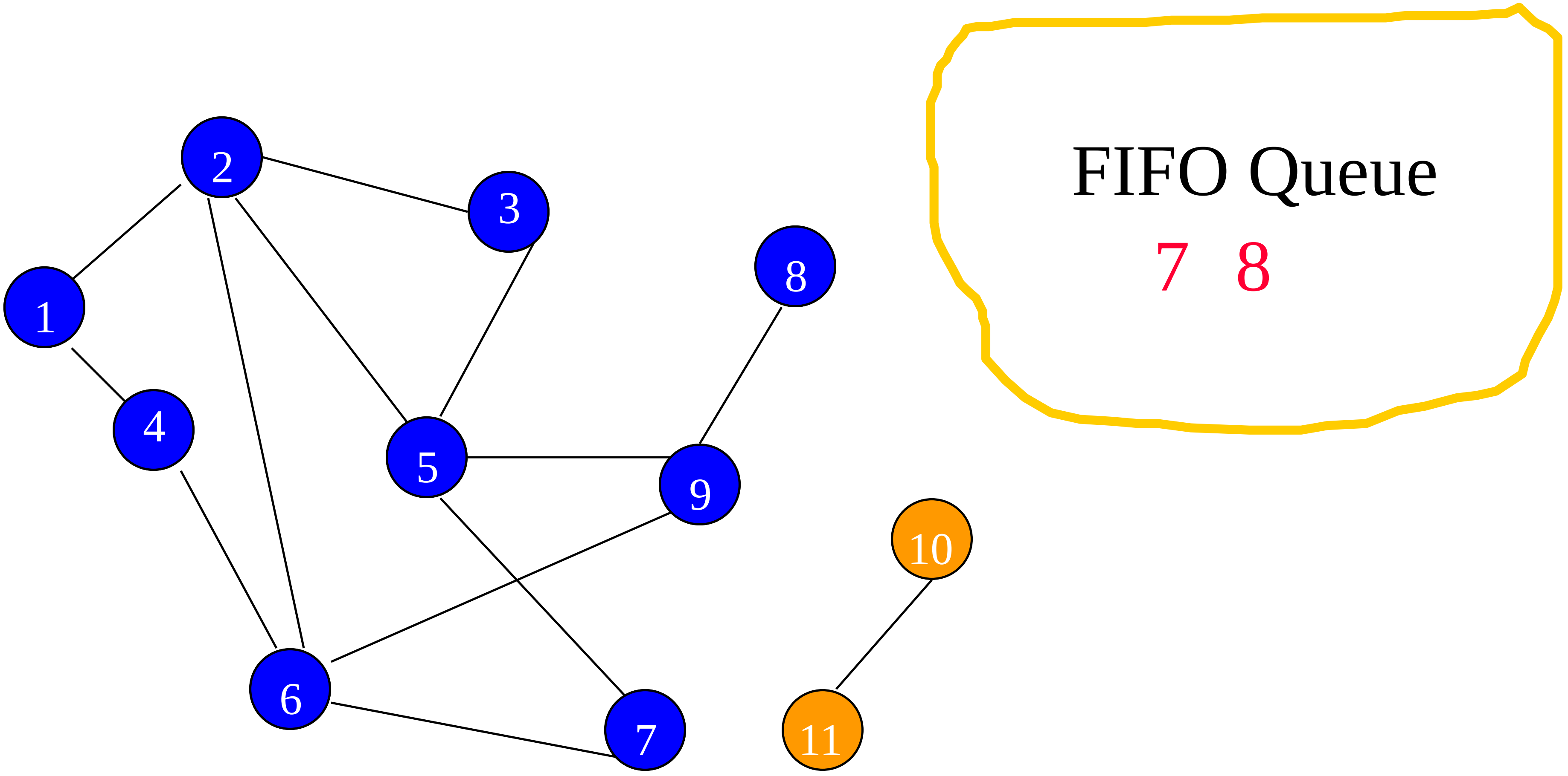
Breadth-First Search Example





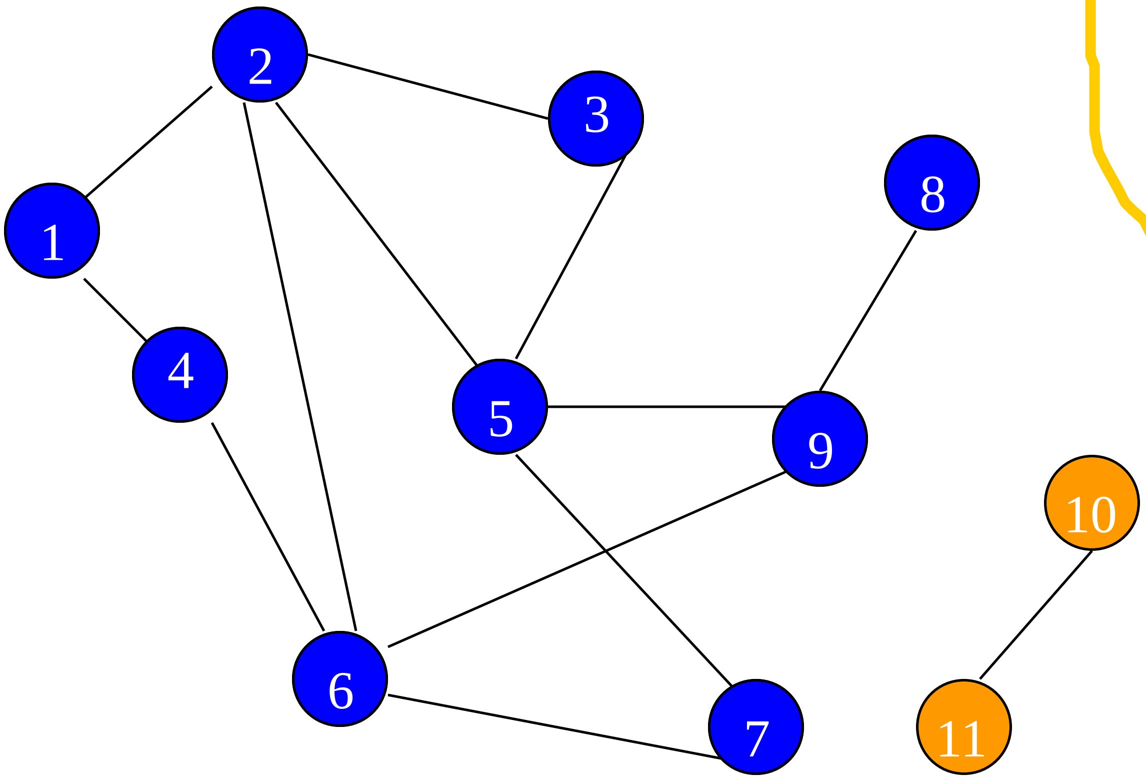
Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



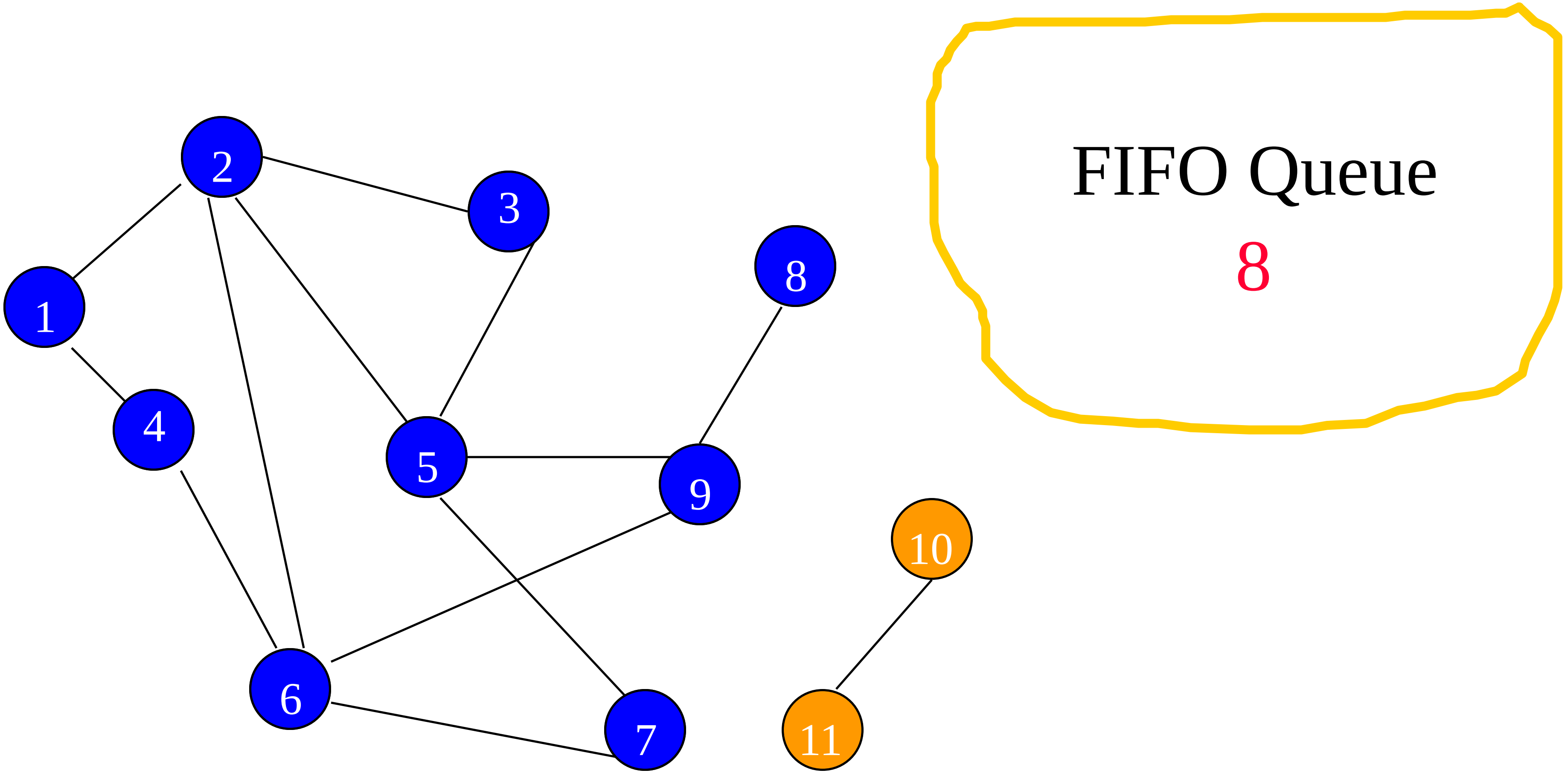
Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



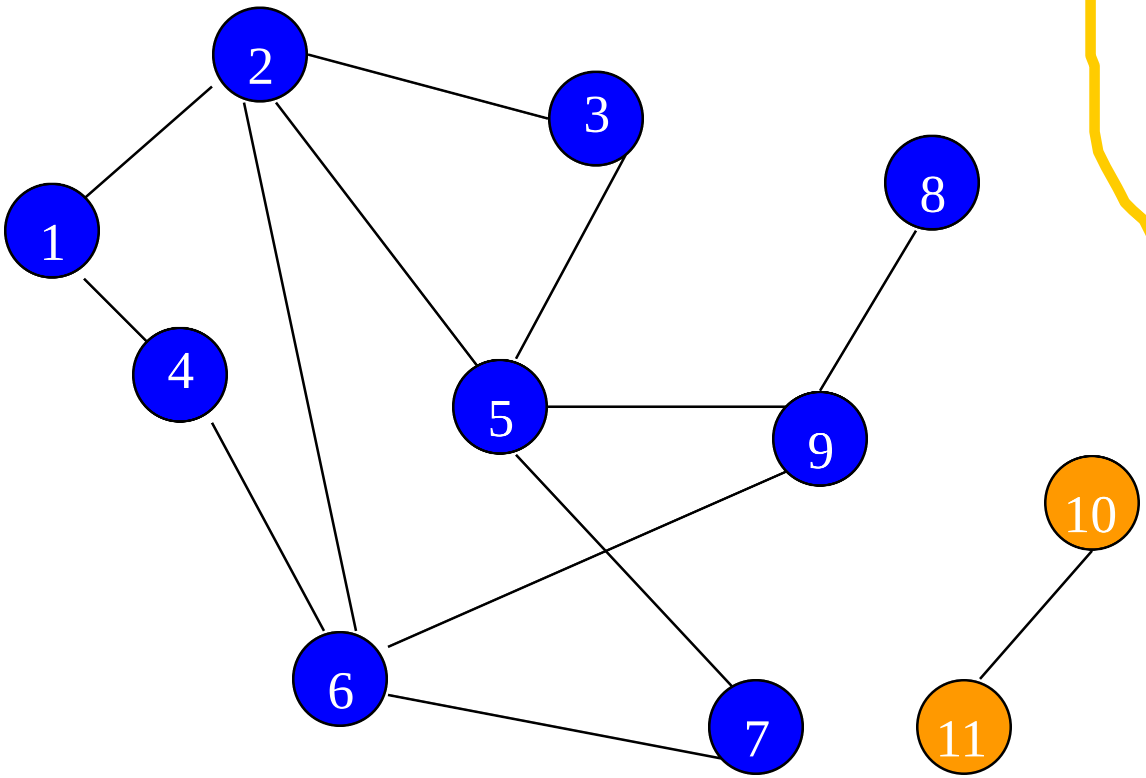
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



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

Breadth-First Search Example

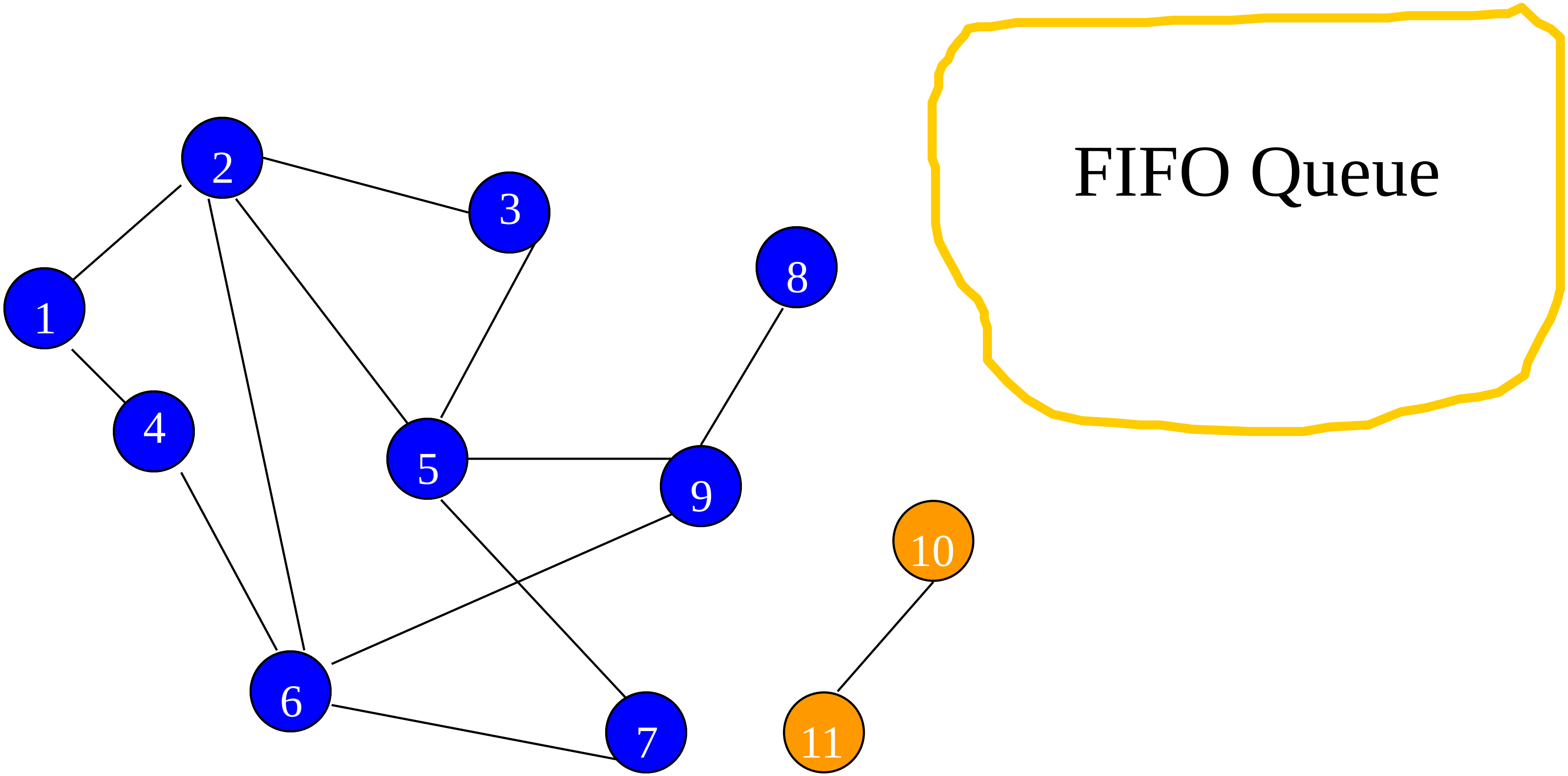


FIFO Queue

8

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

Breadth-First Search Example

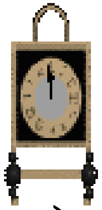


Queue is empty. Search terminates.

Breadth-First Search Property

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

Time Complexity

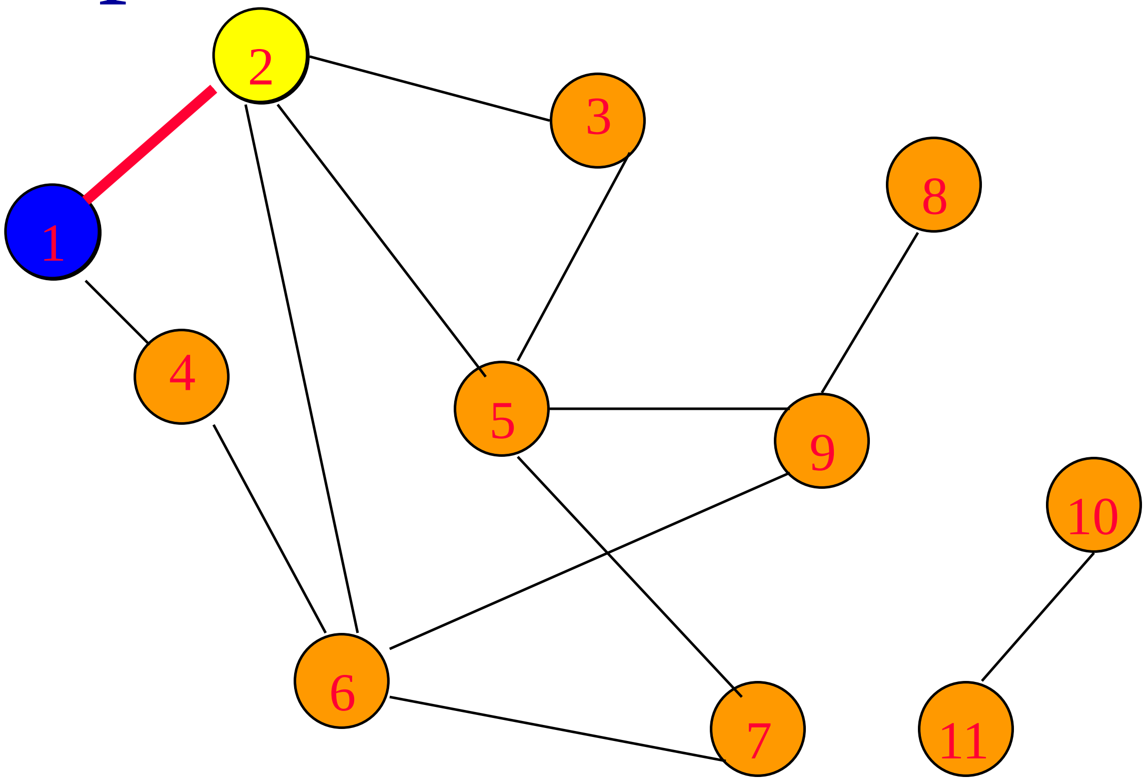


- Each visited vertex is put on (and so removed from) the queue exactly once.
- When a vertex is removed from the queue, we examine its adjacent vertices.
  - O(n) if adjacency matrix used
  - O(*vertex degree*) if adjacency lists used
- Total time
  - $\Theta(sn)$, where s is number of vertices in the component that is searched (adjacency matrix)
  - $\Theta(\sum_i d_i^{out})$ (adjacency lists)

Depth-First Search

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

Depth-First Search Example

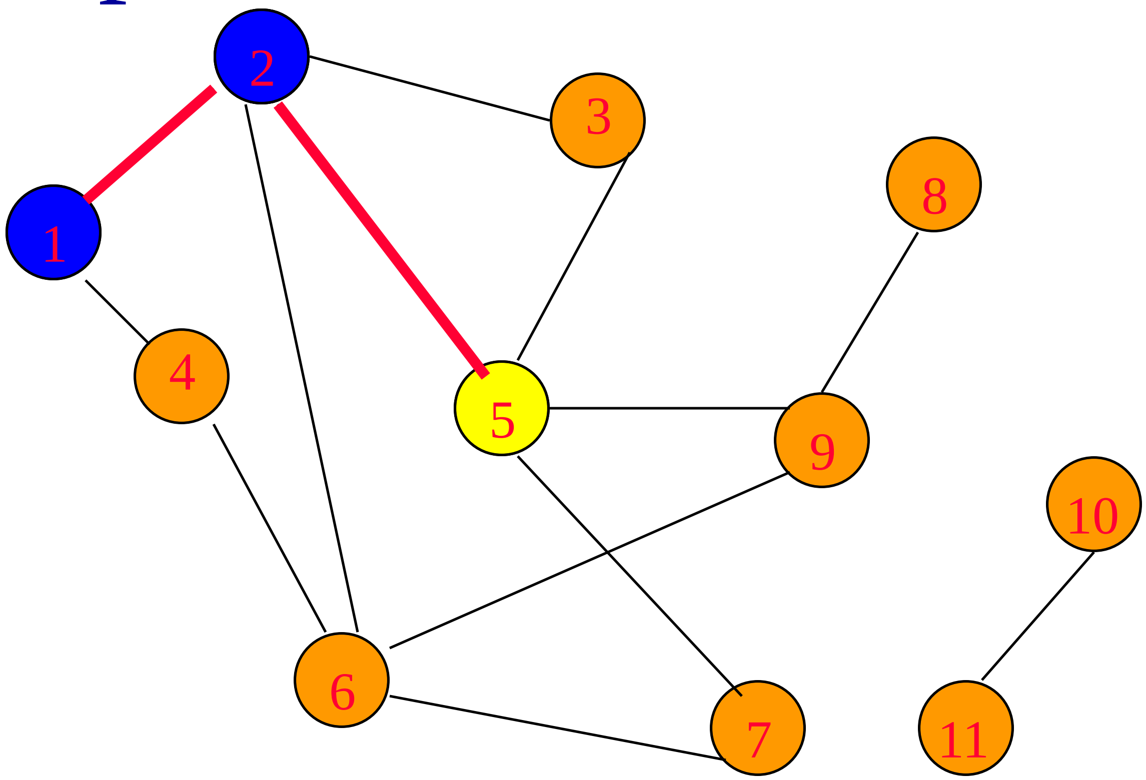


Start search at vertex 1.

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

Suppose that vertex 2 is selected.

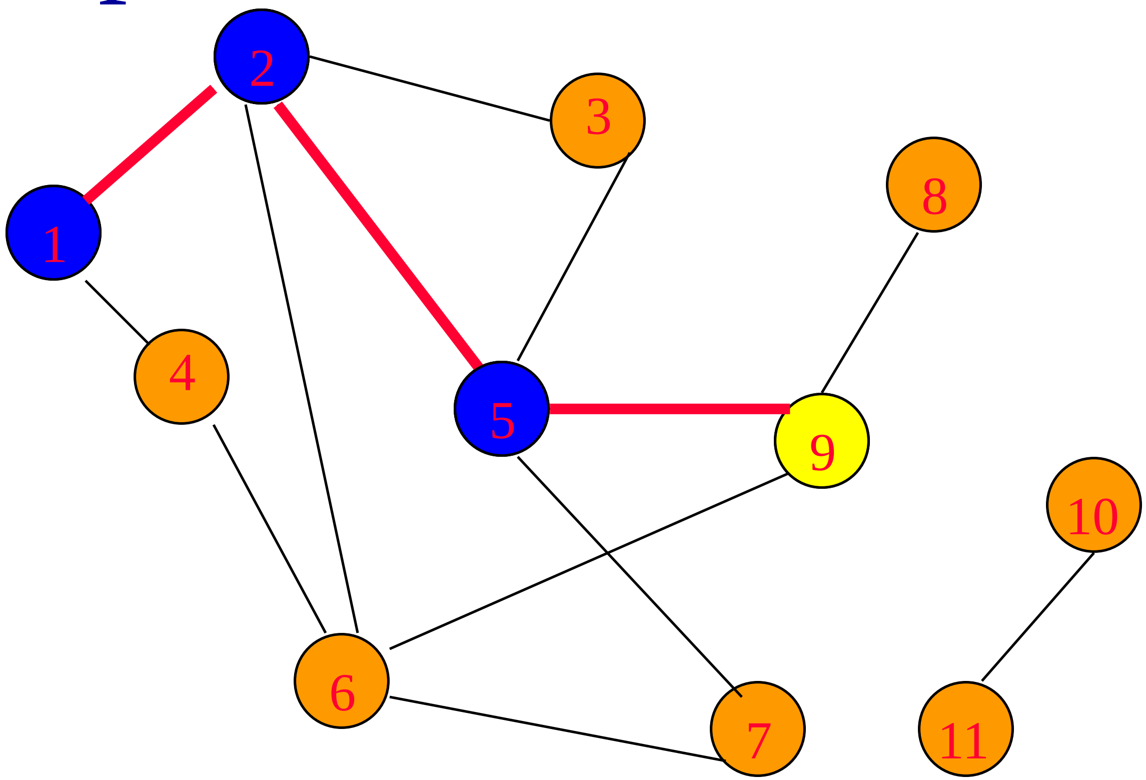
Depth-First Search Example



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

Suppose that vertex 5 is selected.

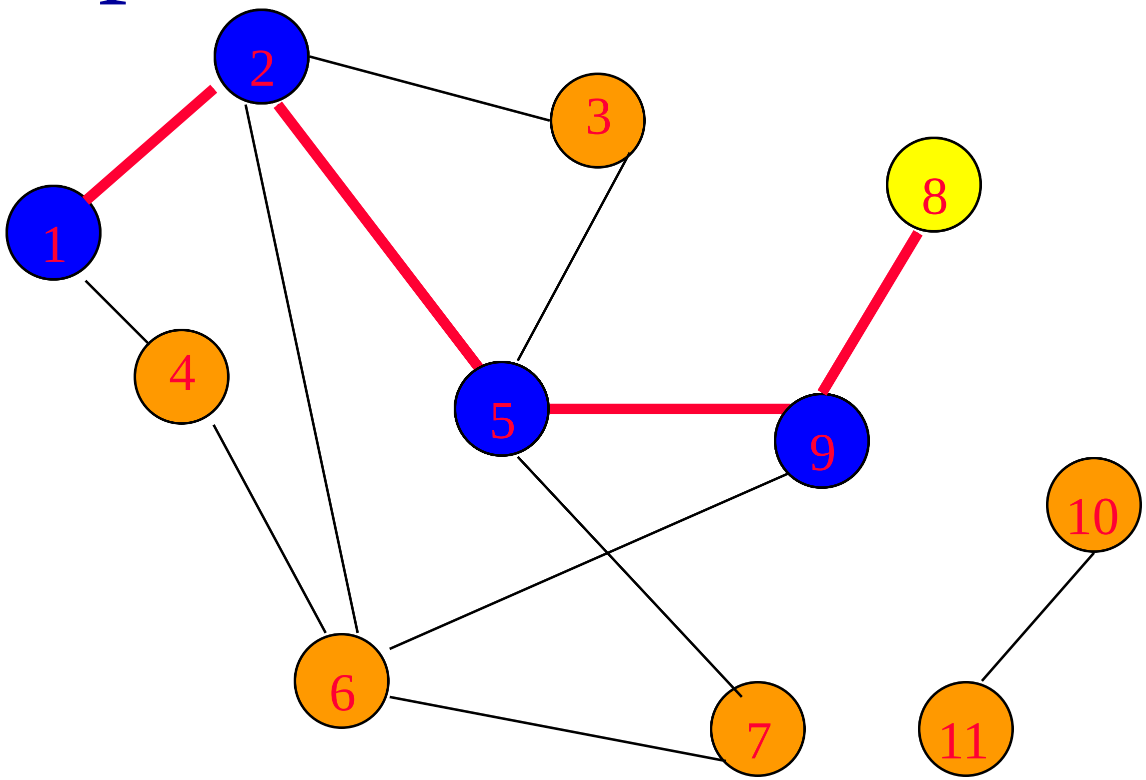
Depth-First Search Example



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

Suppose that vertex 9 is selected.

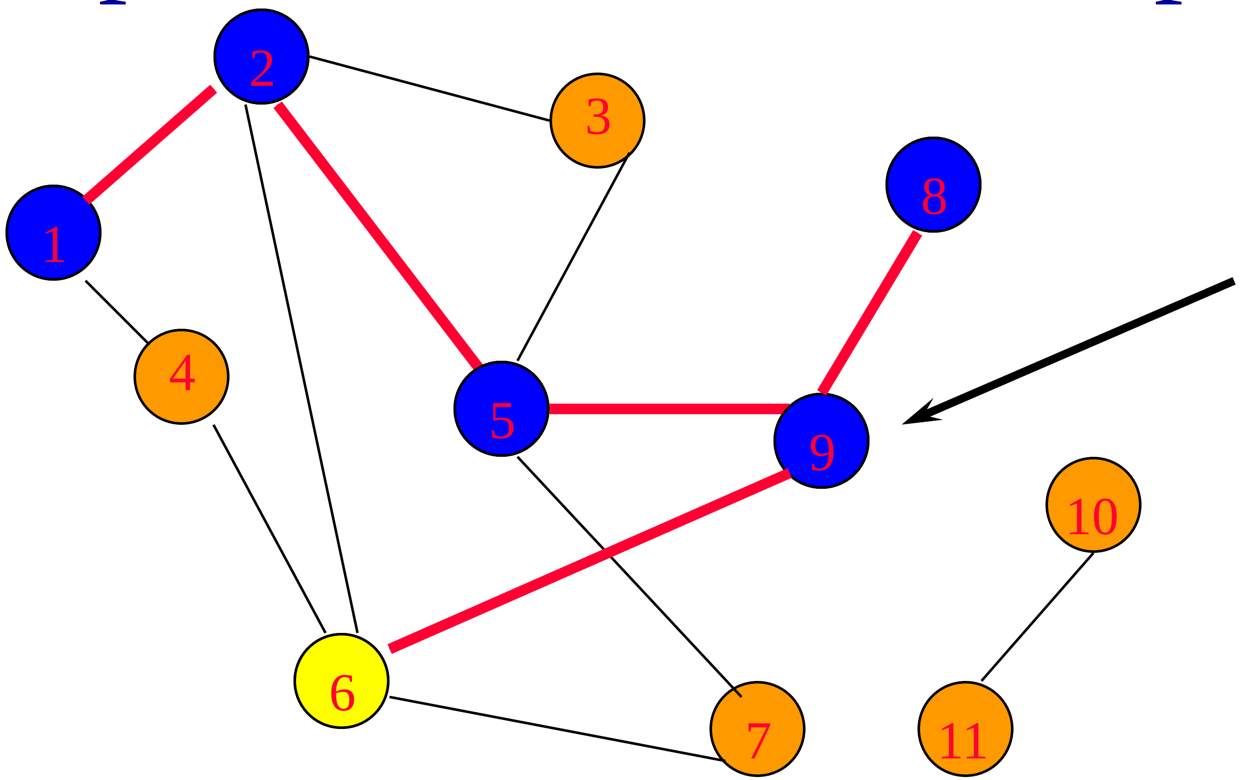
Depth-First Search Example



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

Suppose that vertex 8 is selected.

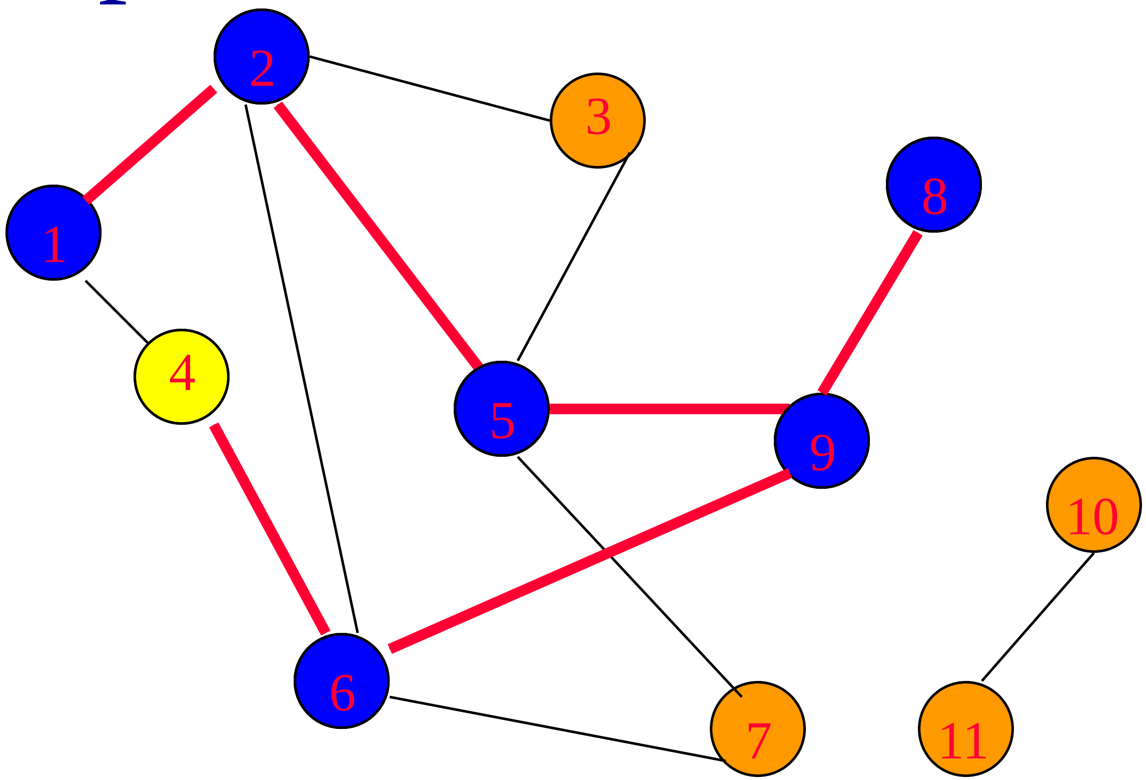
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6).

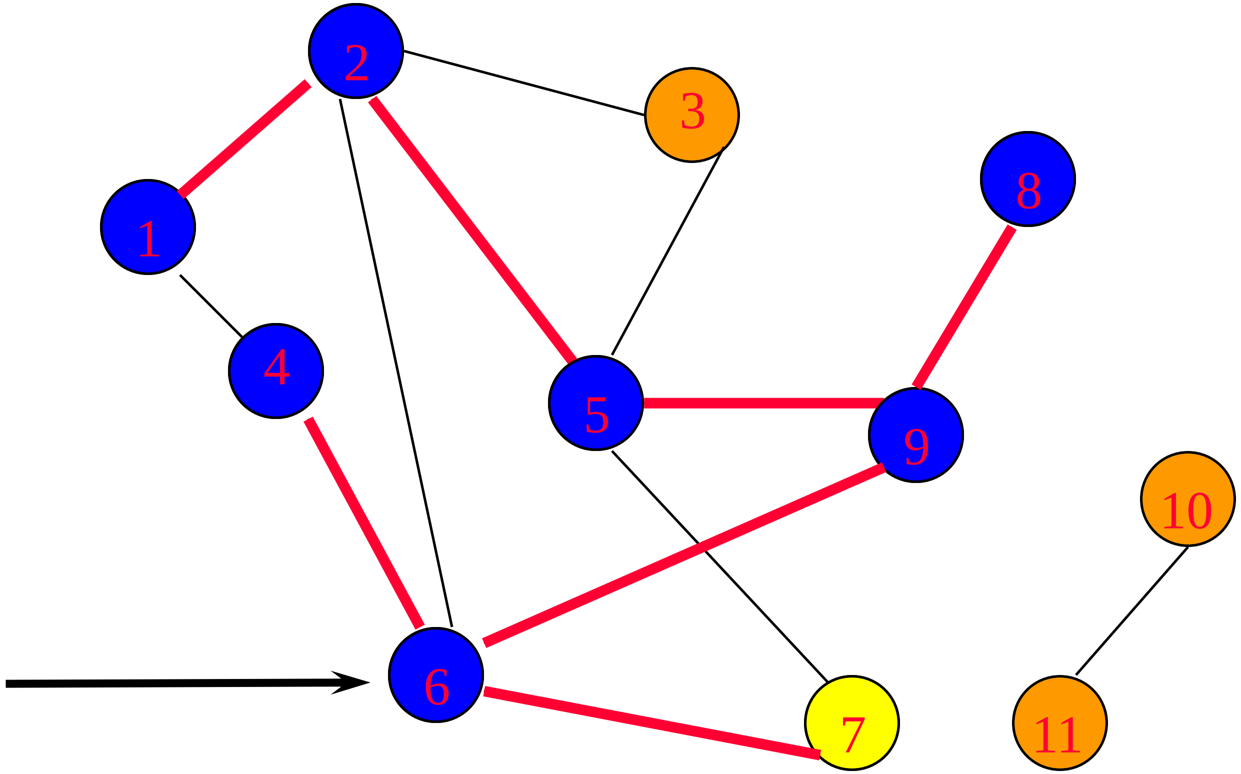
Depth-First Search Example



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

Suppose that vertex 4 is selected.

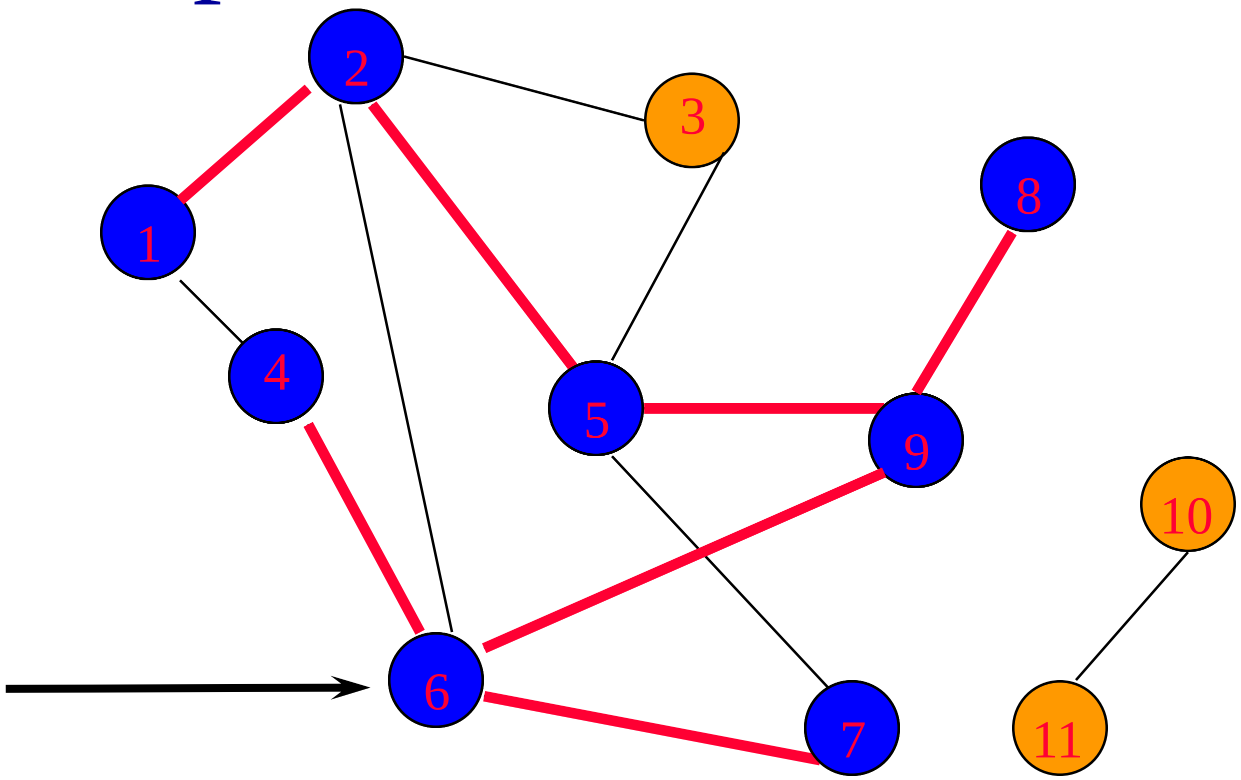
Depth-First Search Example



Label vertex 4 and return to 6.

From vertex 6 do a dfs(7).

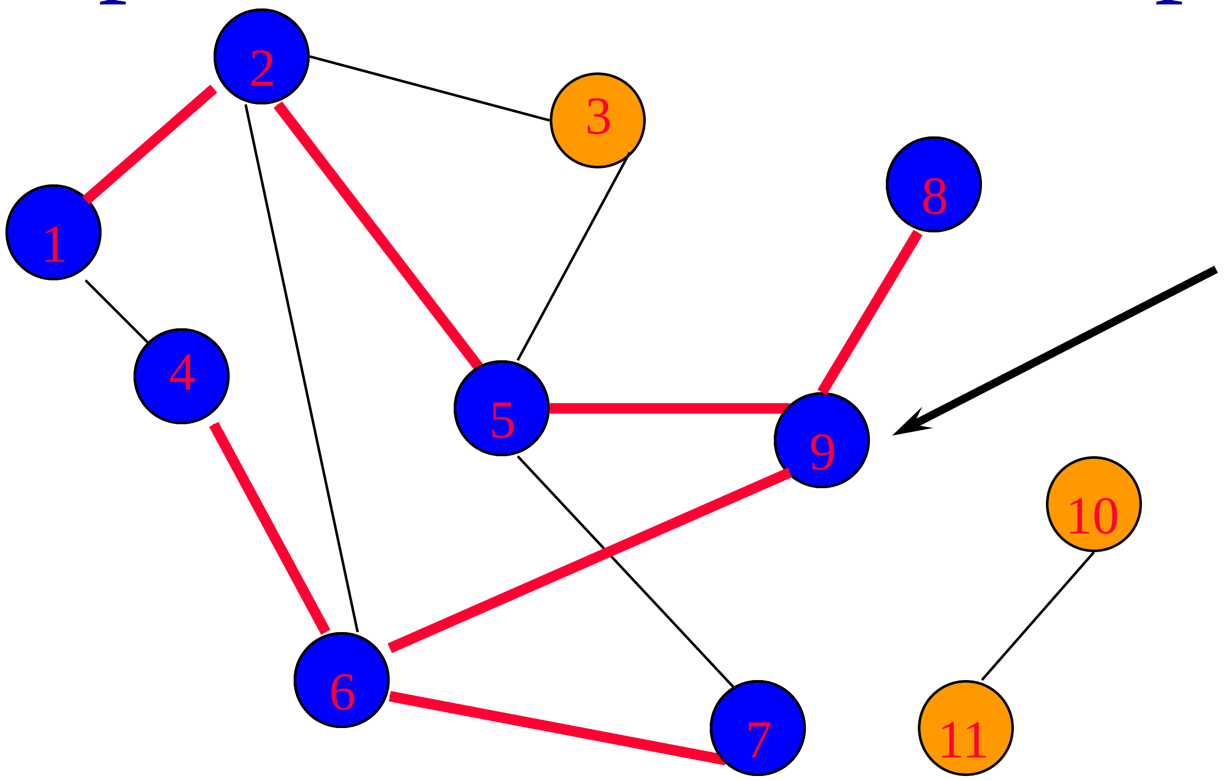
Depth-First Search Example



Label vertex 7 and return to 6.

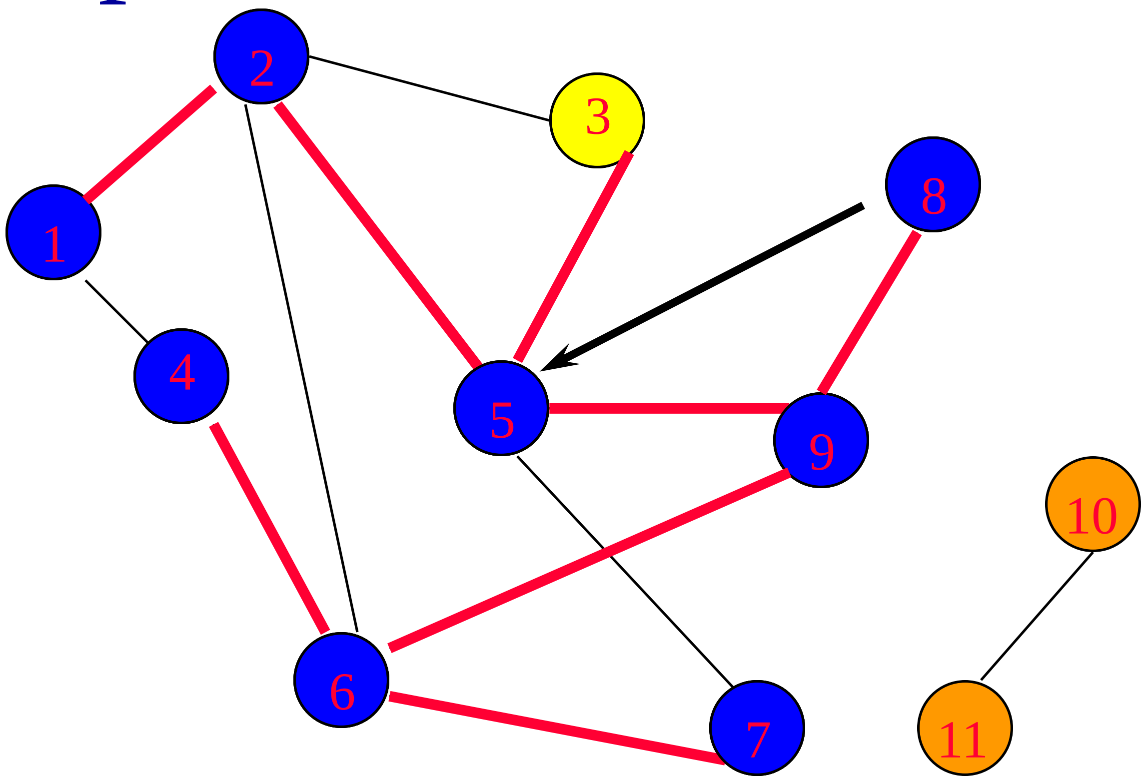
Return to 9.

Depth-First Search Example



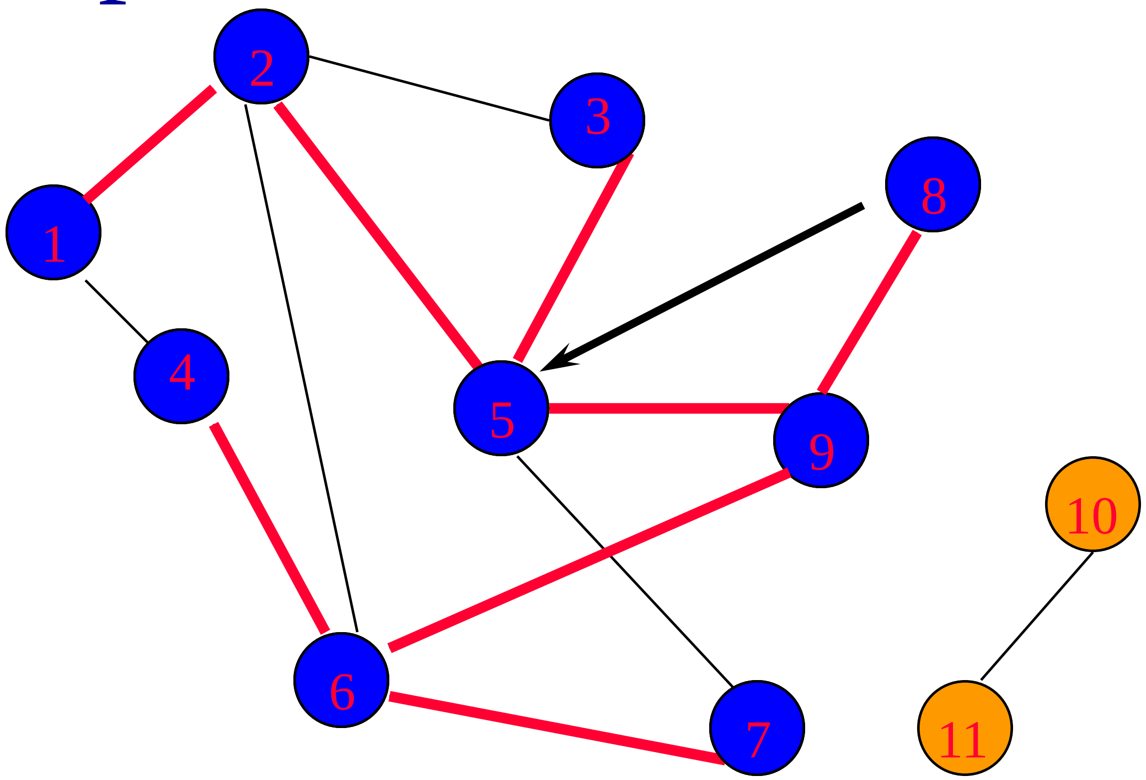
Return to 5.

Depth-First Search Example



Do a dfs(3).

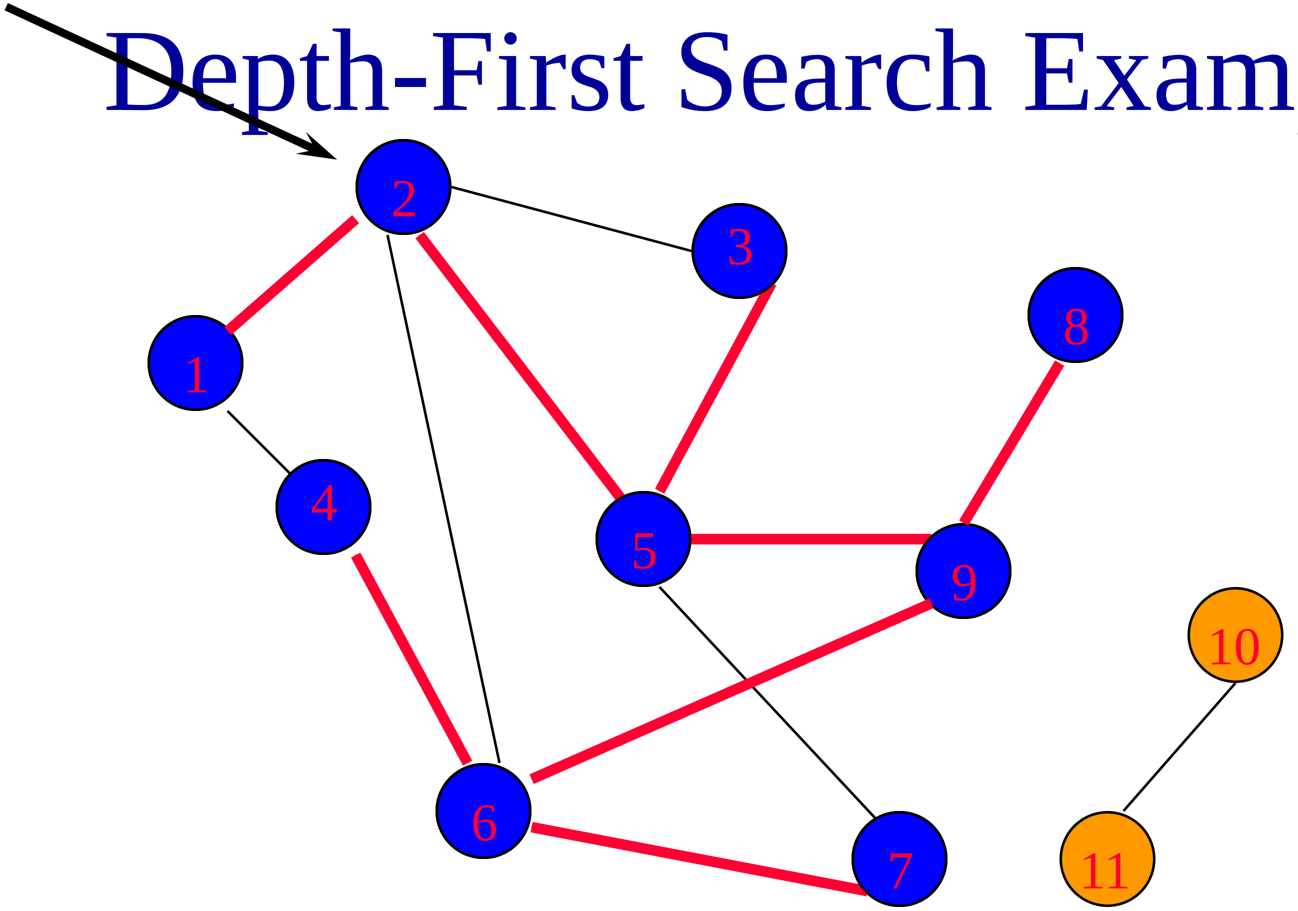
Depth-First Search Example



Label 3 and return to 5.

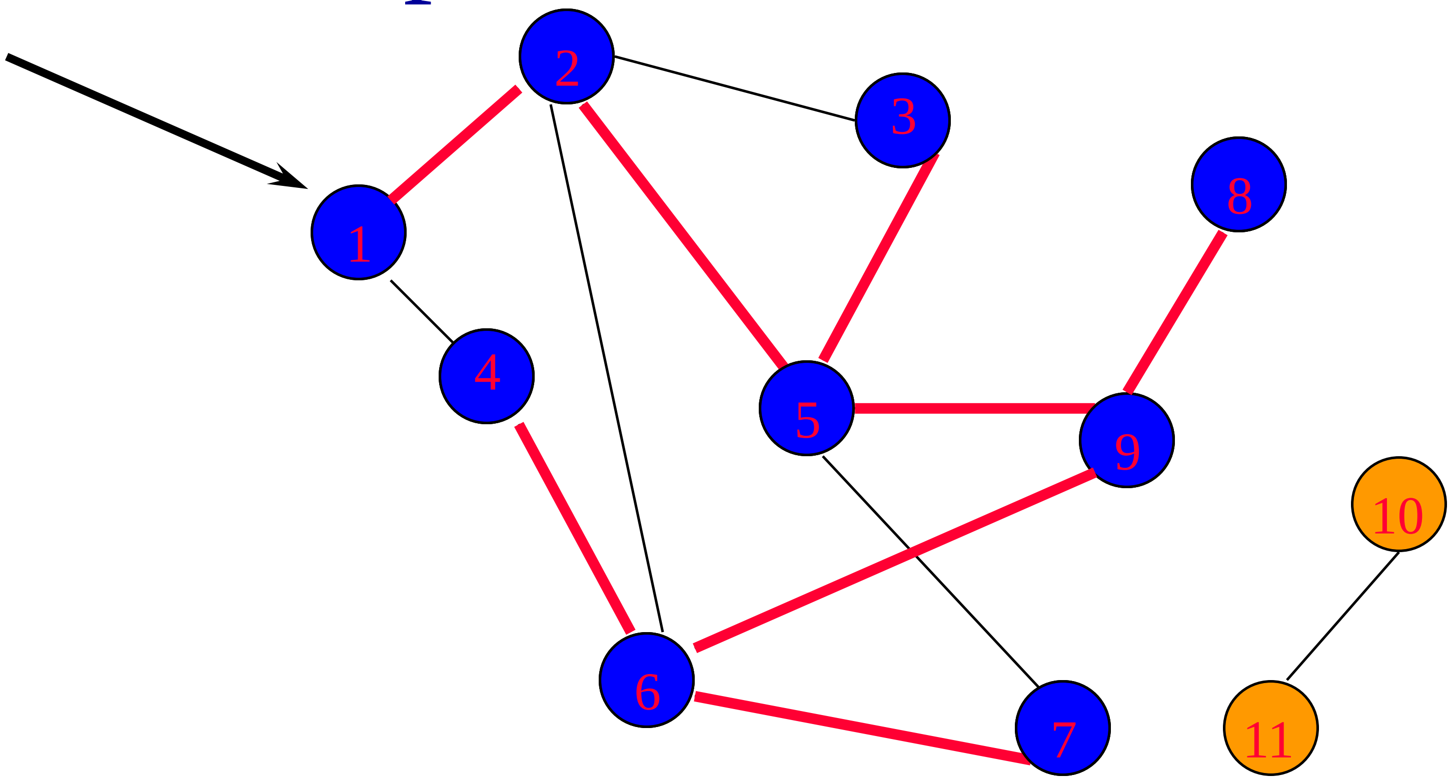
Return to 2.

Depth-First Search Example



Return to 1.

Depth-First Search Example



Return to invoking method.

Depth-First Search Properties

- Same complexity as BFS.
- 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.

Comparisons

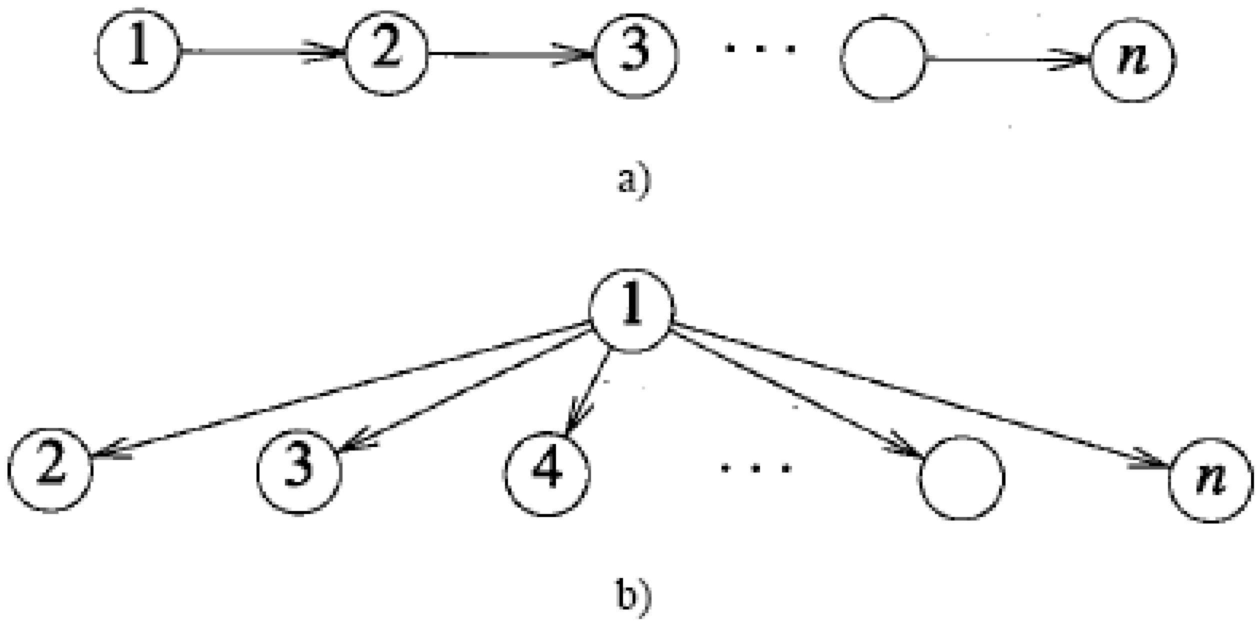
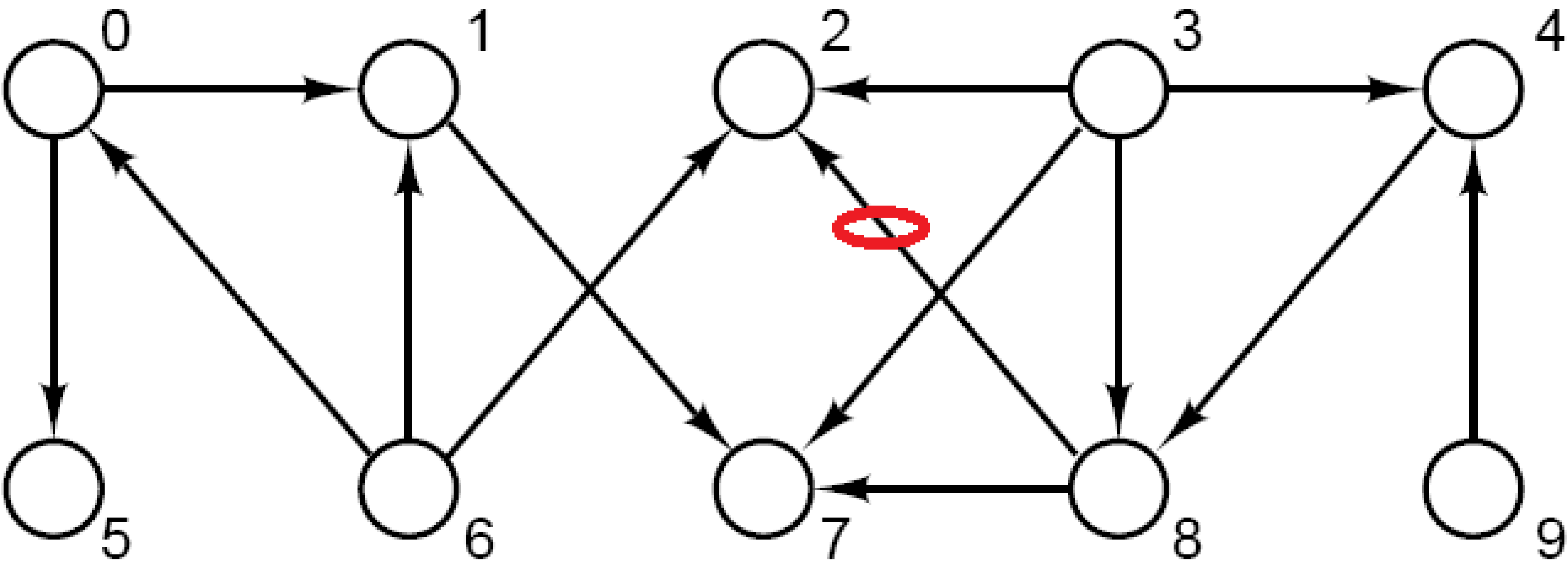


图12-21　产生最好和最坏空间复杂性的图例

a) DepthFirstSearch(1) 的最坏情况；BreadthFirst Search(1) 的最好情况

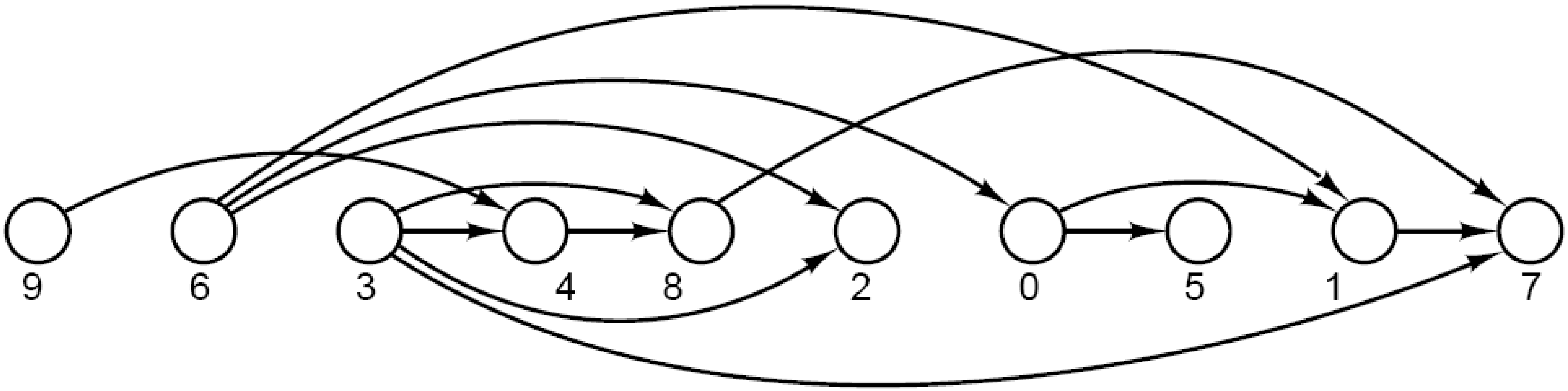
b) DepthFirstSearch(1) 的最好情况；BreadthFirst Search(1) 的最坏情况

12.4 Topological Sorting



Directed graph with no directed cycles

12.4.2 Depth-First Algorithm



Depth-first ordering

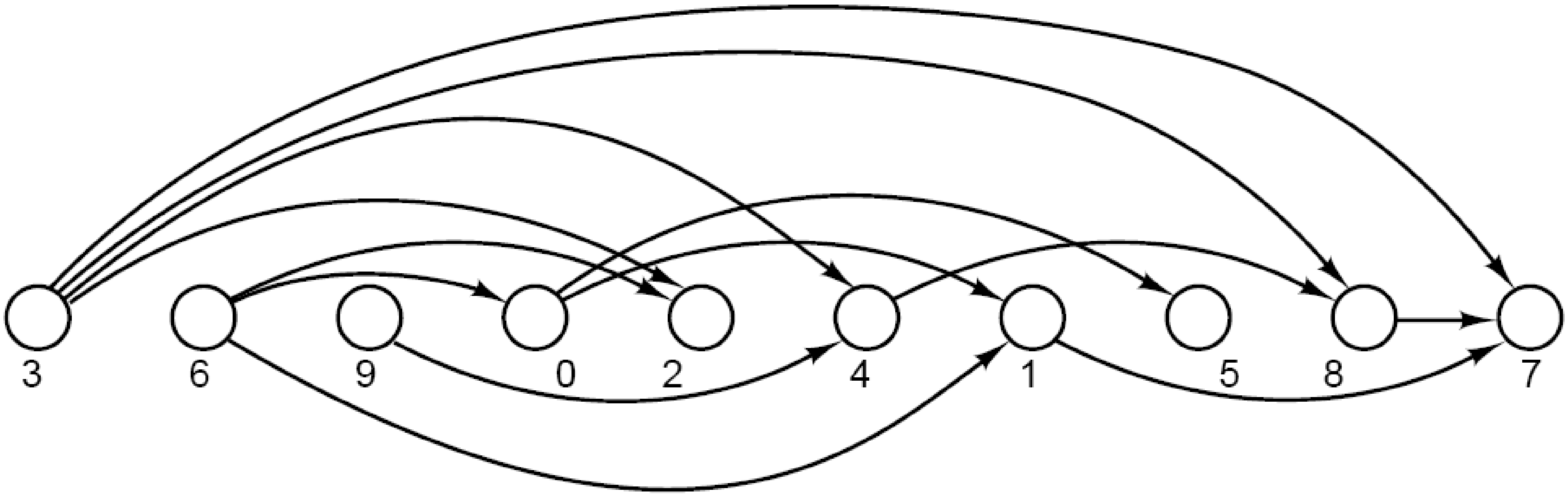
12.4.2 Depth-First Algorithm

```
template <int graph_size>
void Digraph<graph_size> :: depth_sort(List<Vertex> &topological_order)
/* Post: The vertices of the Digraph are placed into List topological_order with a
         depth-first traversal of those vertices that do not belong to a cycle.
   Uses: Methods of class List, and function recursive_depth_sort to perform depth-
         first traversal. */
{
   bool visited[graph_size];
   Vertex v;
   for (v = 0; v < count; v++) visited[v] = false;
   topological_order.clear();
   for (v = 0; v < count; v++)
      if (!visited[v])              // Add v and its successors into topological order.
         recursive_depth_sort(v, visited, topological_order);
}
```

12.4.2 Depth-First Algorithm

```
template <int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool *visited,
                                               List<Vertex> &topological_order)
/* Pre:  Vertex v of the Digraph does not belong to the partially completed List
         topological_order.
   Post: All the successors of v and finally v itself are added to topological_order
         with a depth-first search.
   Uses: Methods of class List and the function recursive_depth_sort. */
{
   visited[v] = true;
   int degree = neighbors[v].size();
   for (int i = 0; i < degree; i++) {
      Vertex w;                        //  A (neighboring) successor of v
      neighbors[v].retrieve(i, w);
      if (!visited[w])                 //  Order the successors of w.
         recursive_depth_sort(w, visited, topological_order);
   }
   topological_order.insert(0, v);     //  Put v into topological_order.
}
```

12.4.3 Breadth-First Algorithm



Breadth-first ordering

Figure 12.7. Topological orderings of a directed graph

12.4.3 Breadth-First Algorithm

一、无前趋的顶点优先的拓扑排序方法

该方法的每一步总是输出当前无前趋(即入度为零)的顶点，其抽象算法可描述为：

```
NonPreFirstTopSort(G){//优先输出无前趋的顶点
  while(G中有入度为0的顶点)do{
      从G中选择一个入度为0的顶点v且输出之；
      从G中删去v及其所有出边；
  }
  if(输出的顶点数目<|V(G)|)
      Error("G中存在有向环，排序失败！");
}
```

|V(G)|为图中所有顶点数目，如输出顶点个数小于|V(G)|，表明有有向环存在。

12.4.3 Breadth-First Algorithm

二、无后继的顶点优先拓扑排序方法

1、思想方法

该方法的每一步均是输出当前无后继(即出度为0)的顶点。对于一个有向图，按此方法输出的序列是**逆拓扑次序**。因此设置一个栈(或向量)T来保存输出的顶点序列，即可得到拓扑序列。若T是栈，则每当输出顶点时，只需做入栈操作，排序完成时将栈中顶点依次出栈即可得拓扑序列。若T是向量，则将输出的顶点从T[n-1]开始依次从后往前存放，即可保证T中存储的顶点是拓扑序列。

2、抽象算法描述

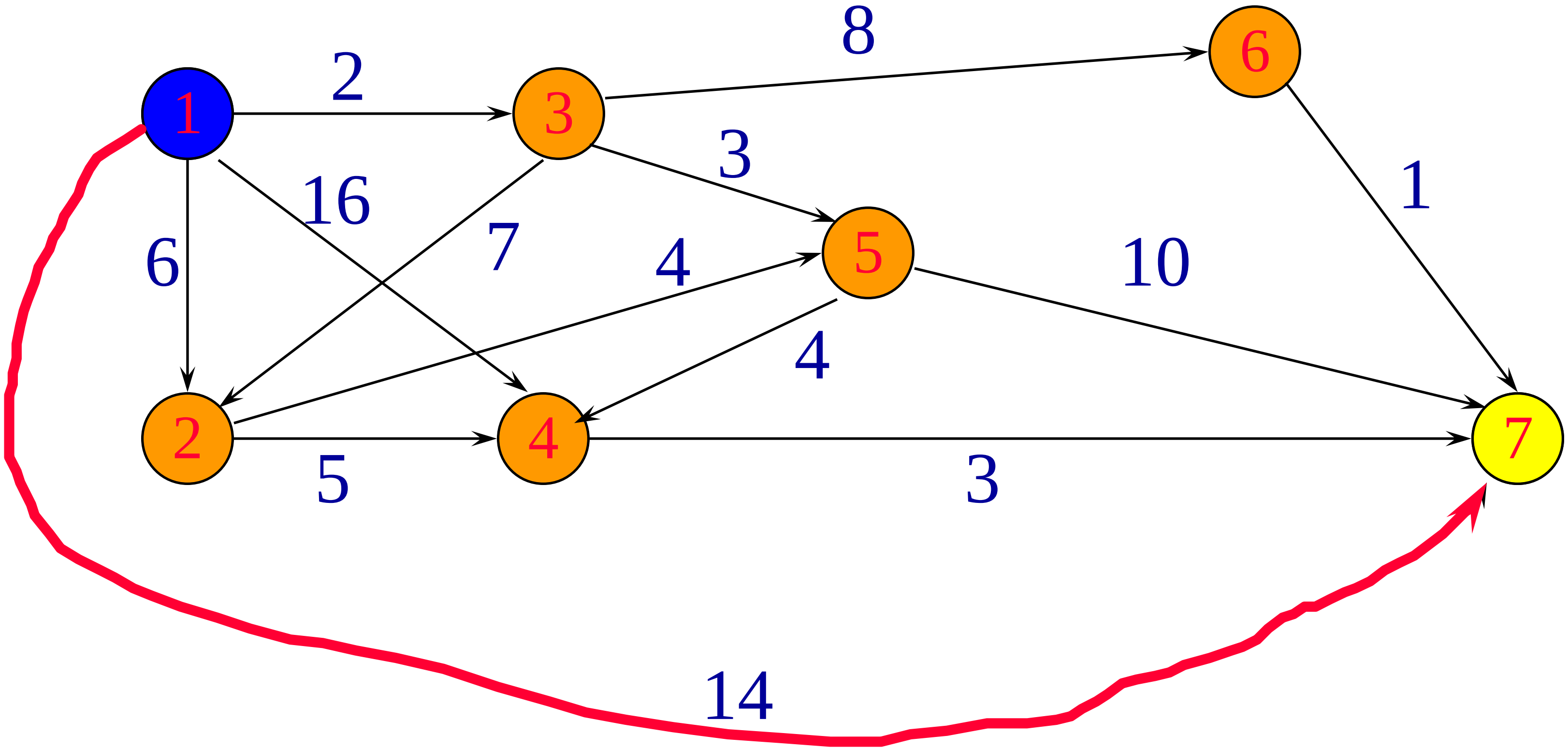
算法的抽象描述为：

```
NonSuccFirstTopSort(G){//优先输出无后继的顶点
 while(G中有出度为0的顶点)do {
  从G中选一出度为0的顶点v且输出v;
  从G中删去v及v的所有人边
  }
 if(输出的顶点数目<|V(G)|)
   Error("G中存在有向环，排序失败!")；}
```

12.5 A Greedy Algorithm: Shortest Paths

- Directed weighted graph.
- Path length is sum of weights of edges on path.
- The vertex at which the path begins is the source vertex.
- The vertex at which the path ends is the destination vertex.

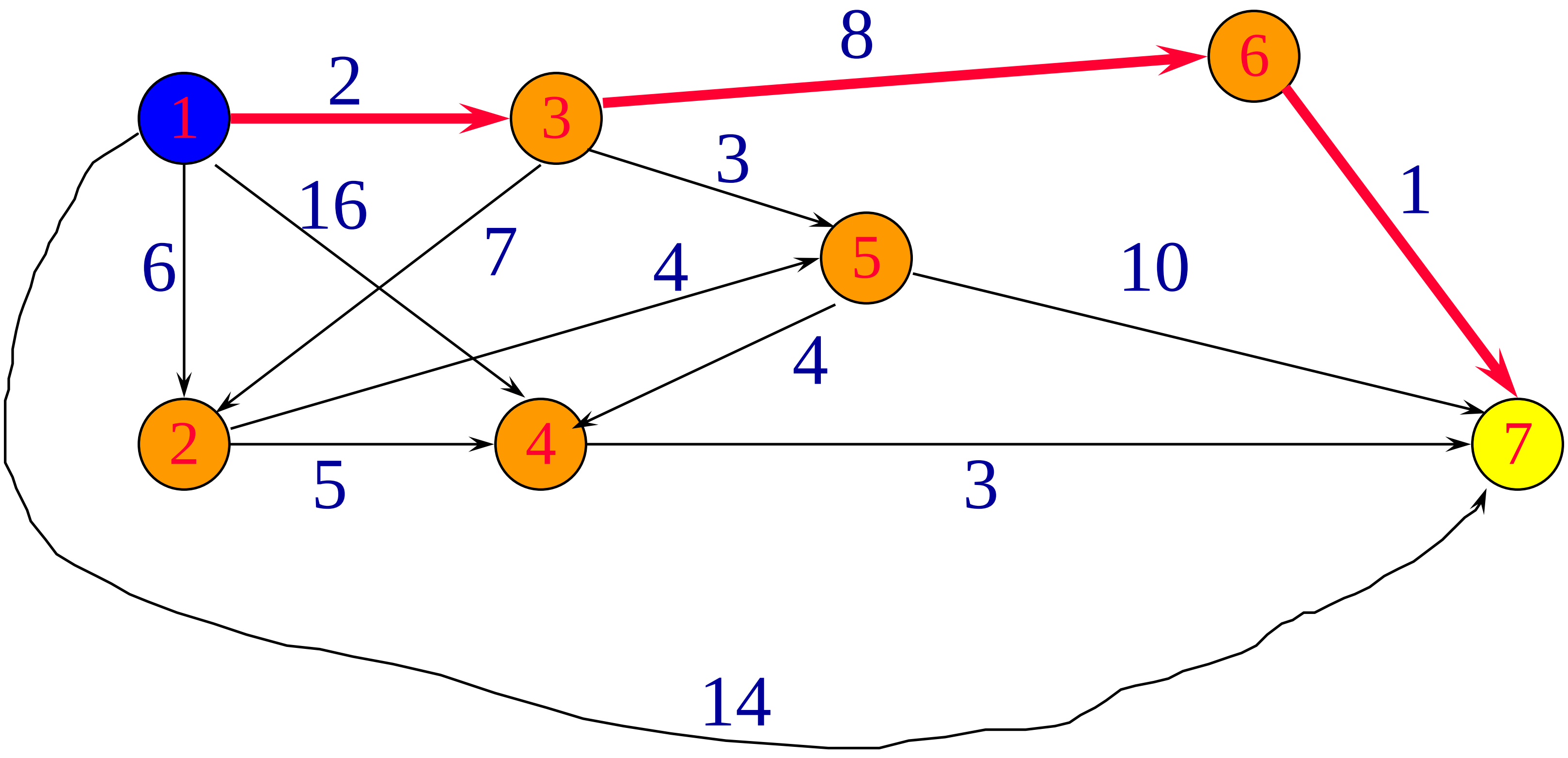
Example



A path from 1 to 7.

Path length is 14.

Example



Another path from 1 to 7.

Path length is 11.

Shortest Path Problems

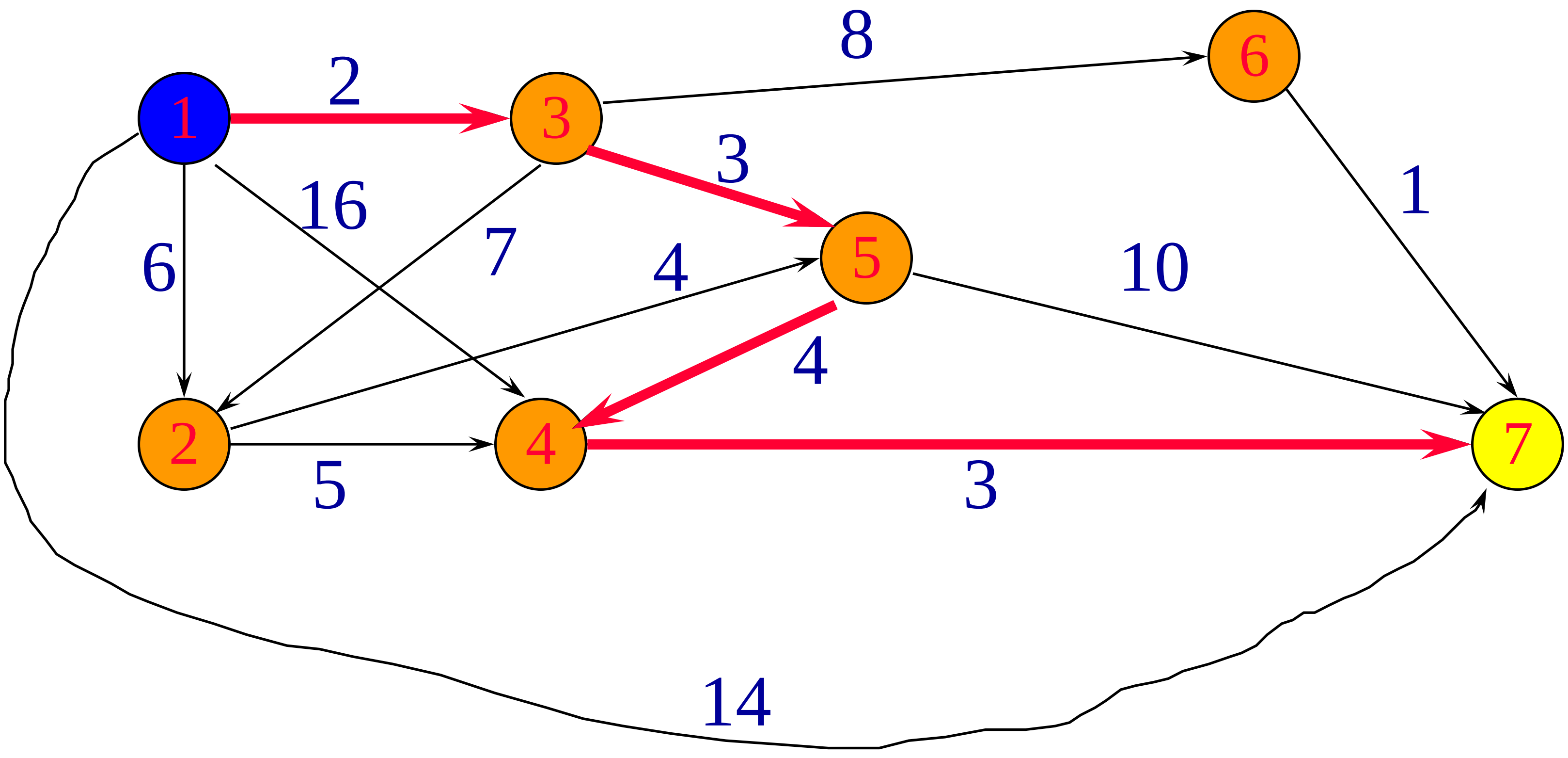
- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

Single Source Single Destination

Possible greedy algorithm:

- Leave source vertex using cheapest/shortest edge.
- Leave new vertex using cheapest edge subject to the constraint that a new vertex is reached.
- Continue until destination is reached.

Greedy Shortest 1 To 7 Path



Path length is 12.

Not shortest path. Algorithm doesn't work!

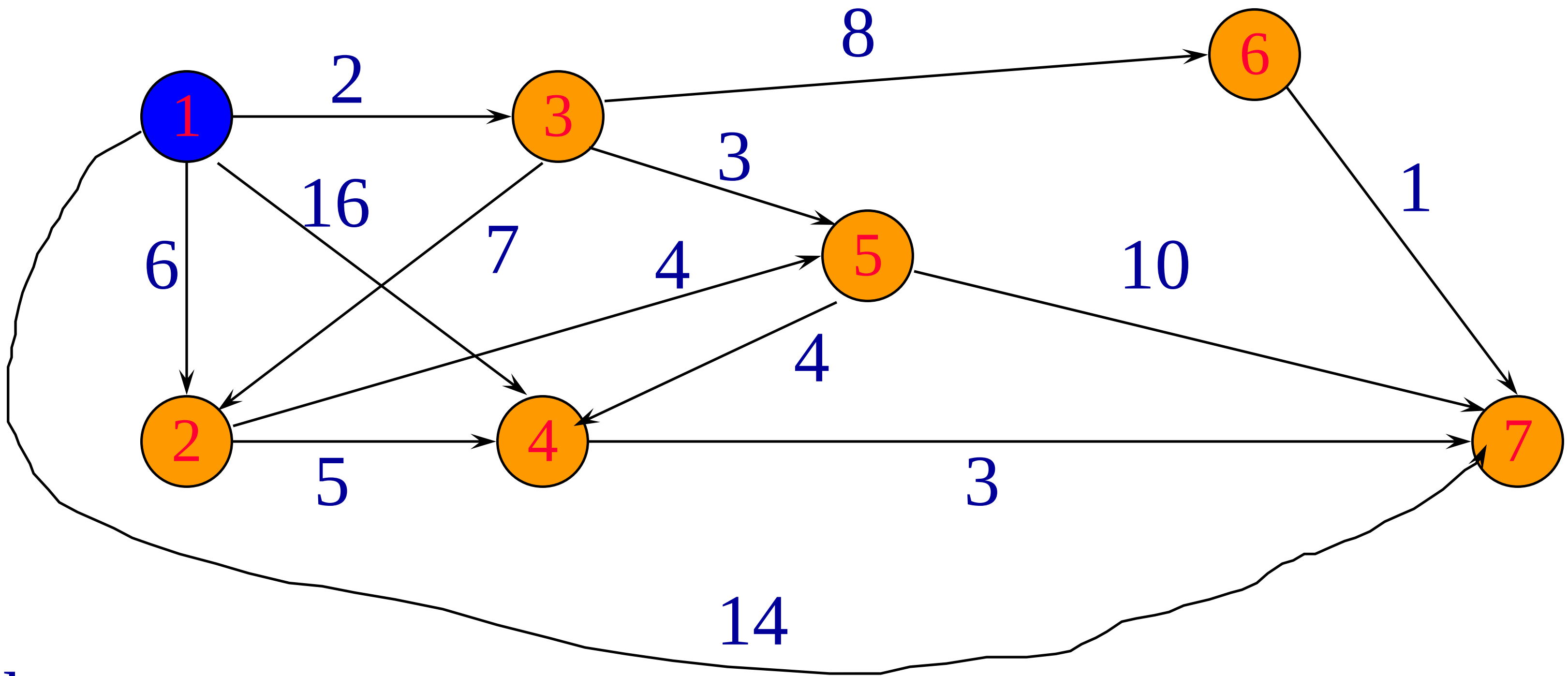
Single Source All Destinations

Need to generate up to *n* (*n* is number of vertices) paths (including path from source to itself).

Greedy method:

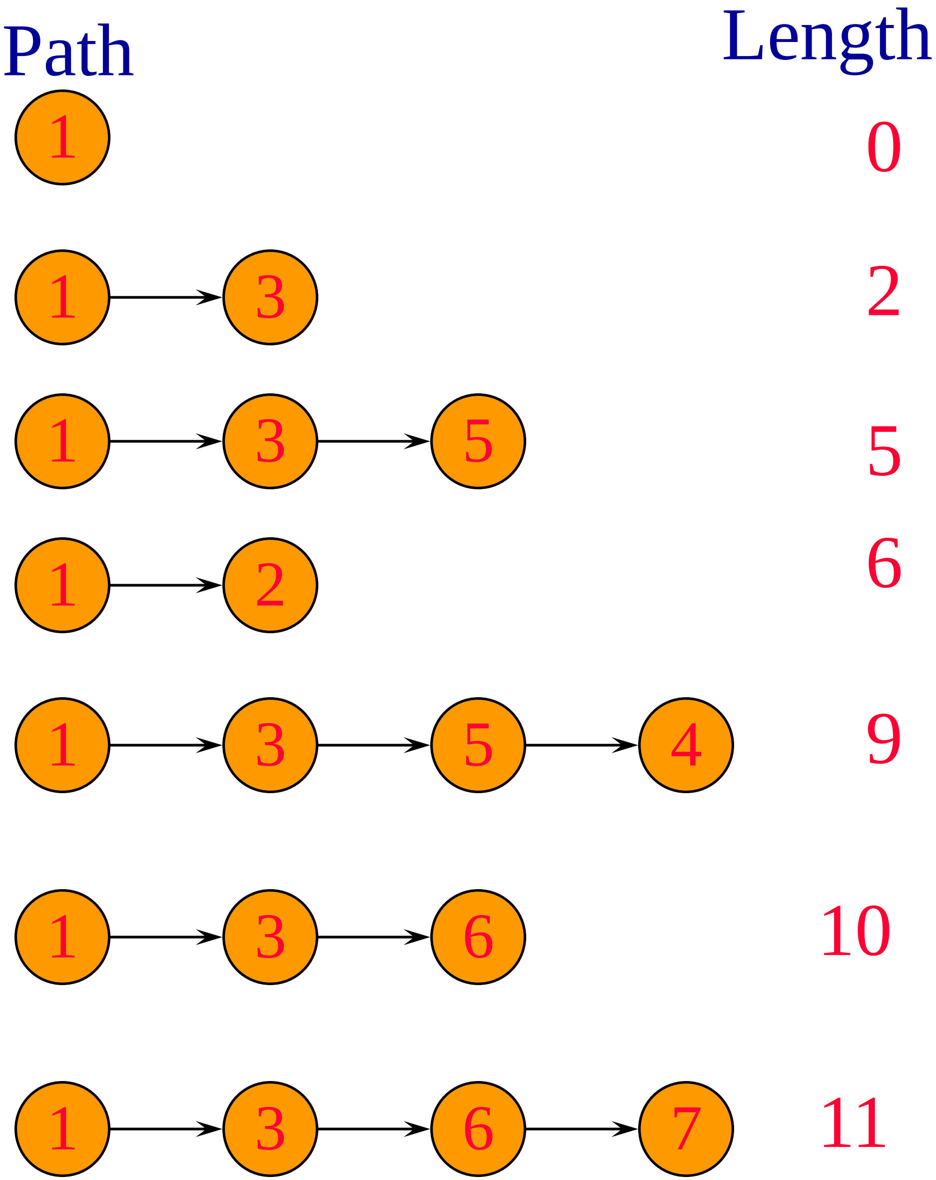
- Construct these up to *n* paths in order of increasing length.
- Assume edge costs (lengths) are >= 0.
- So, no path has length < 0.
- First shortest path is from the source vertex to itself. The length of this path is 0.

Greedy Single Source All Destinations



Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

Greedy Single Source All Destinations

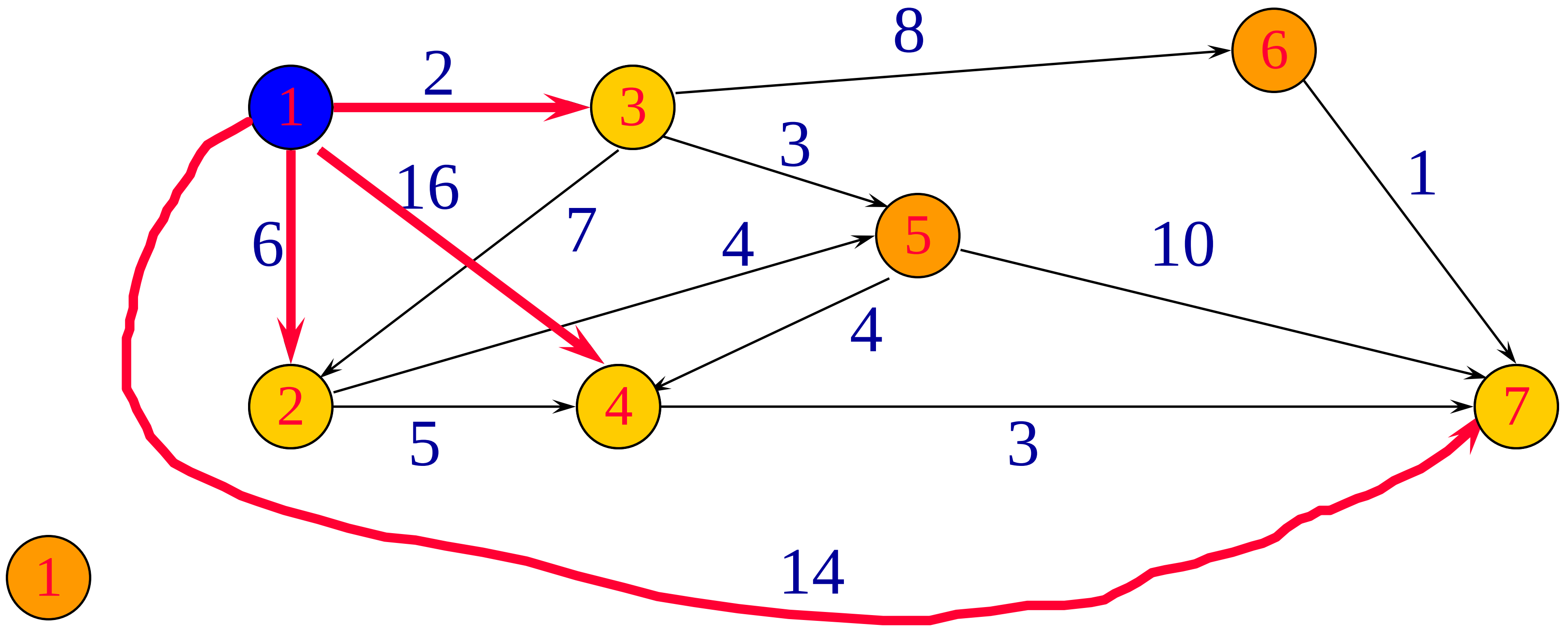


- Each path (other than first) is a one edge extension of a previous path.
- Next shortest path is the shortest one edge extension of an already generated shortest path.

Greedy Single Source All Destinations

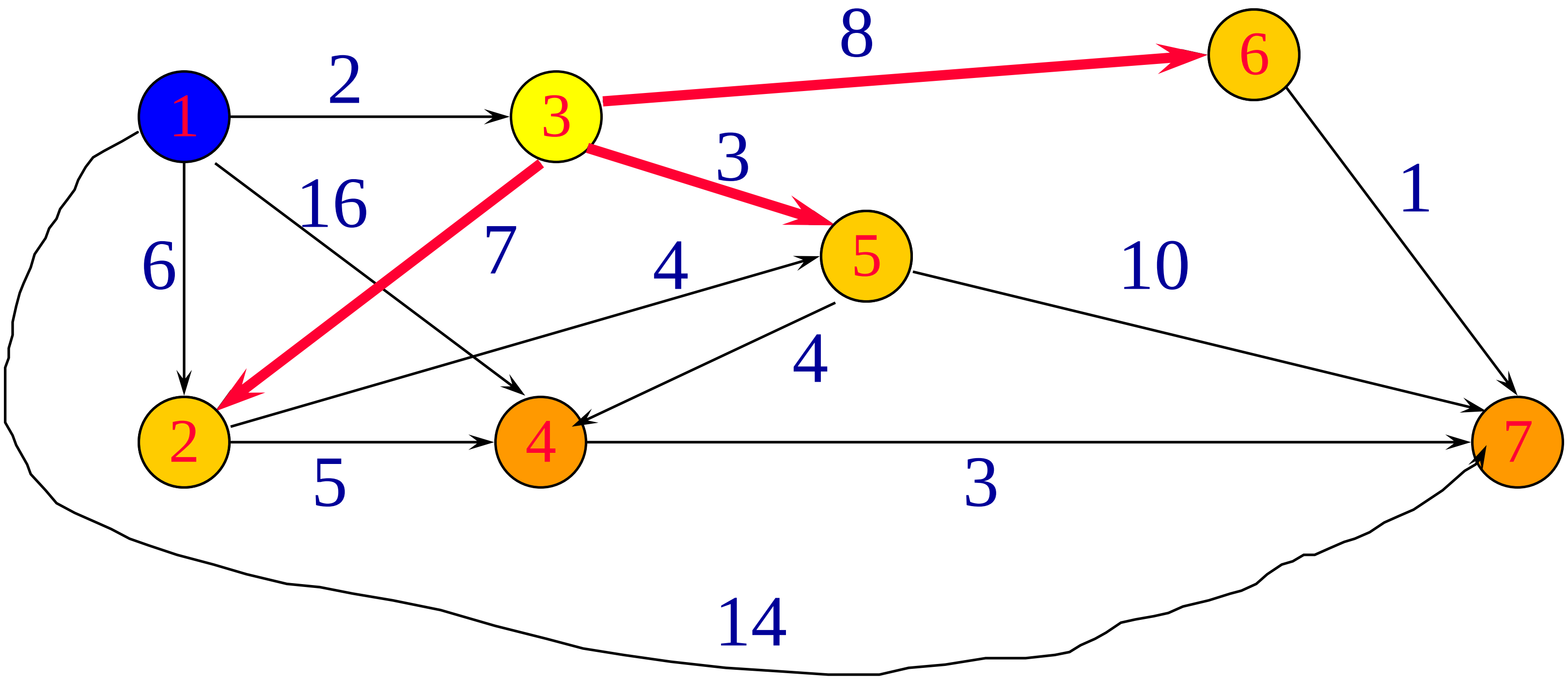
- Let $d(i)$ (distanceFromSource(i)) be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the $d()$ value is least.
- Let $p(i)$ (predecessor(i)) be the vertex just before vertex i on the shortest one edge extension to i.

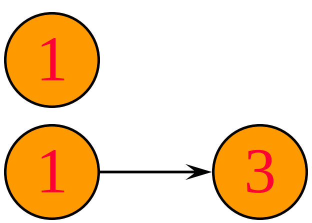
Greedy Single Source All Destinations



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	-	-	14
p	-	1	1	1	-	-	1

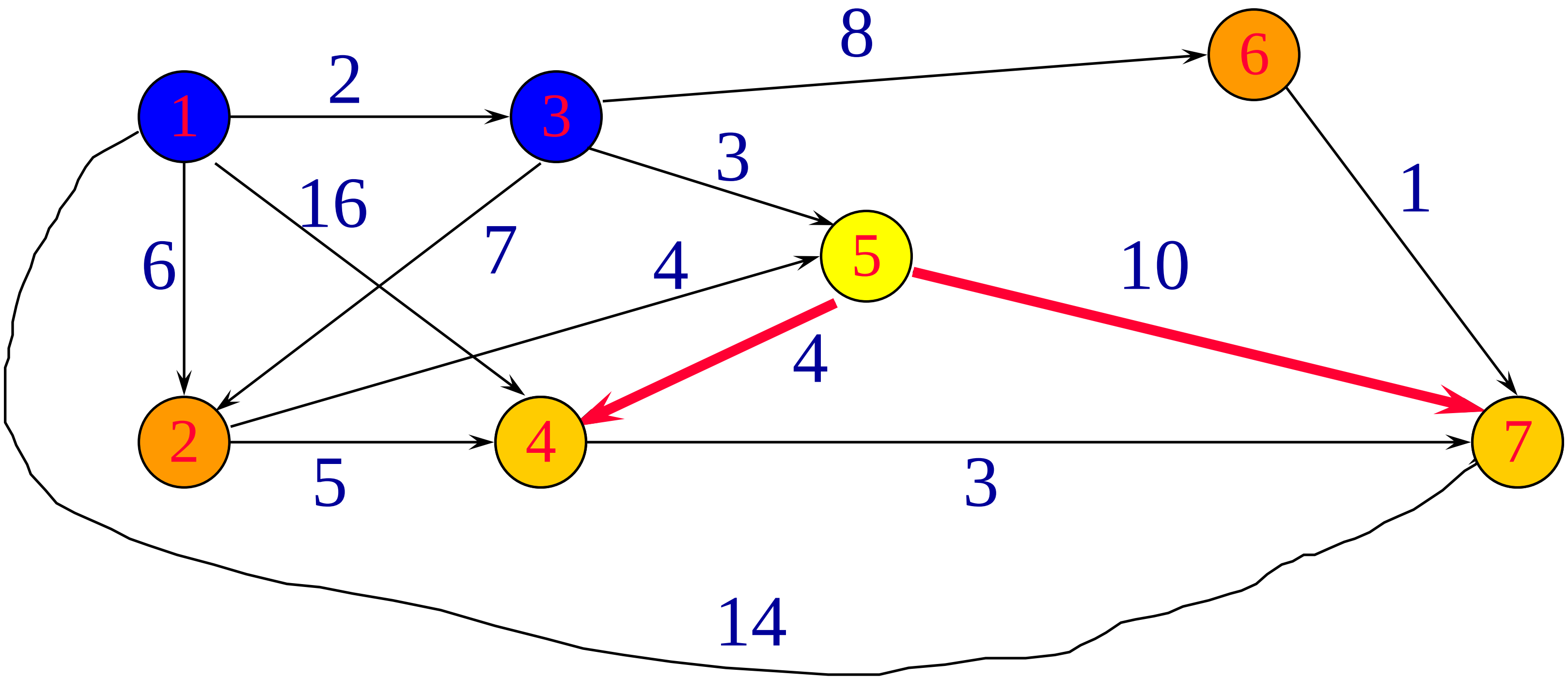
Greedy Single Source All Destinations

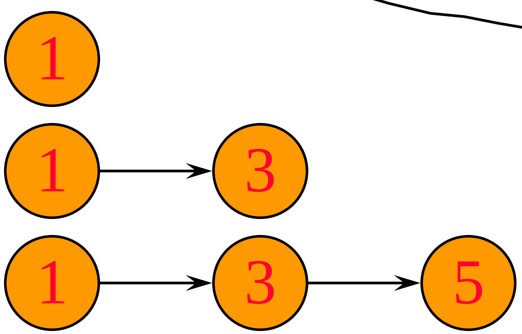




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	5	10	14
p	-	1	1	1	3	3	1

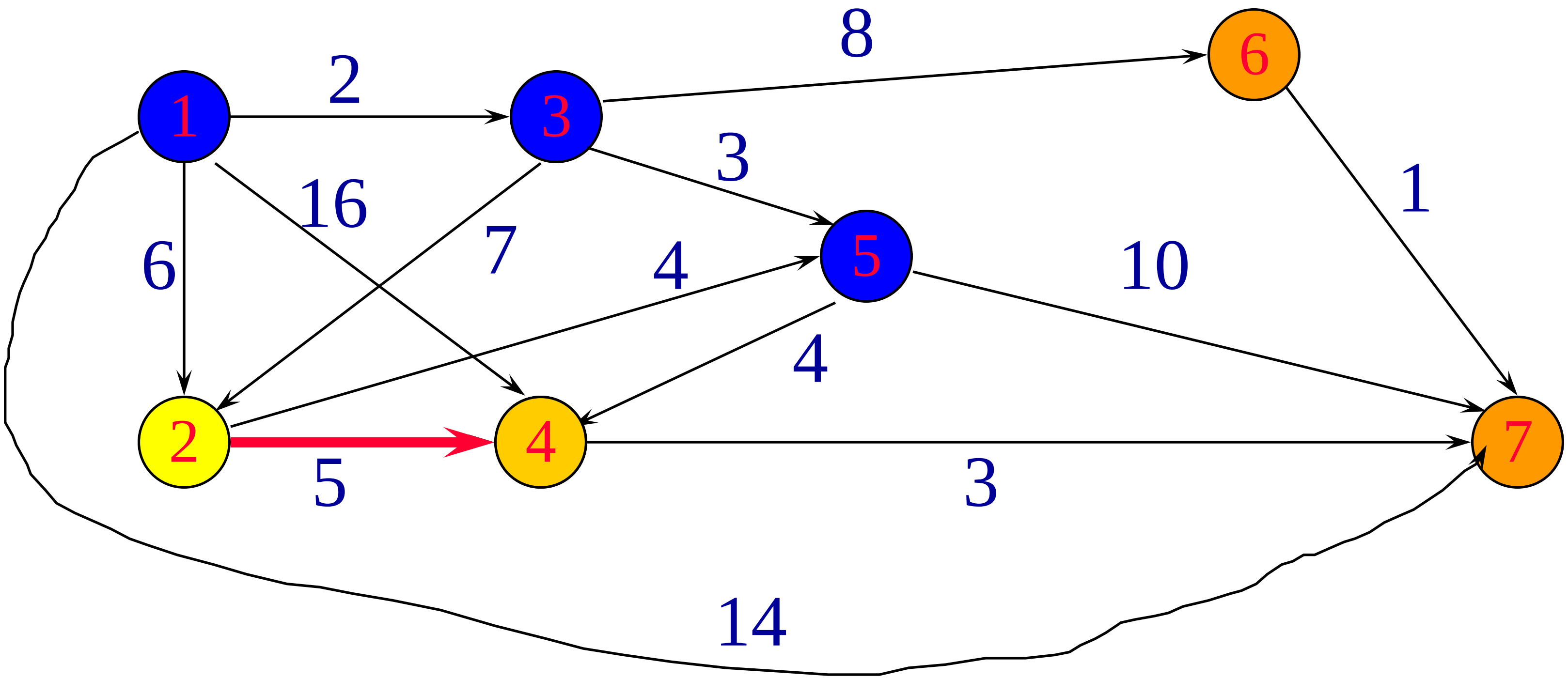
Greedy Single Source All Destinations

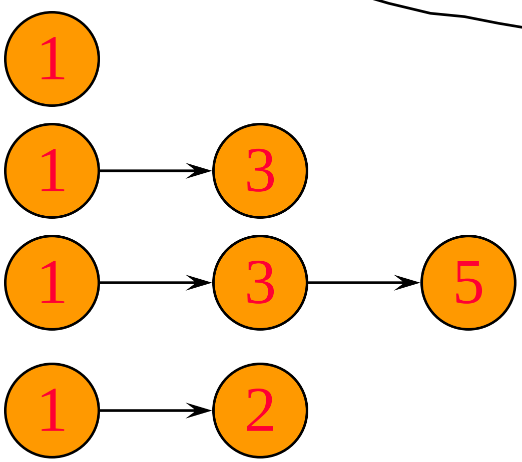




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

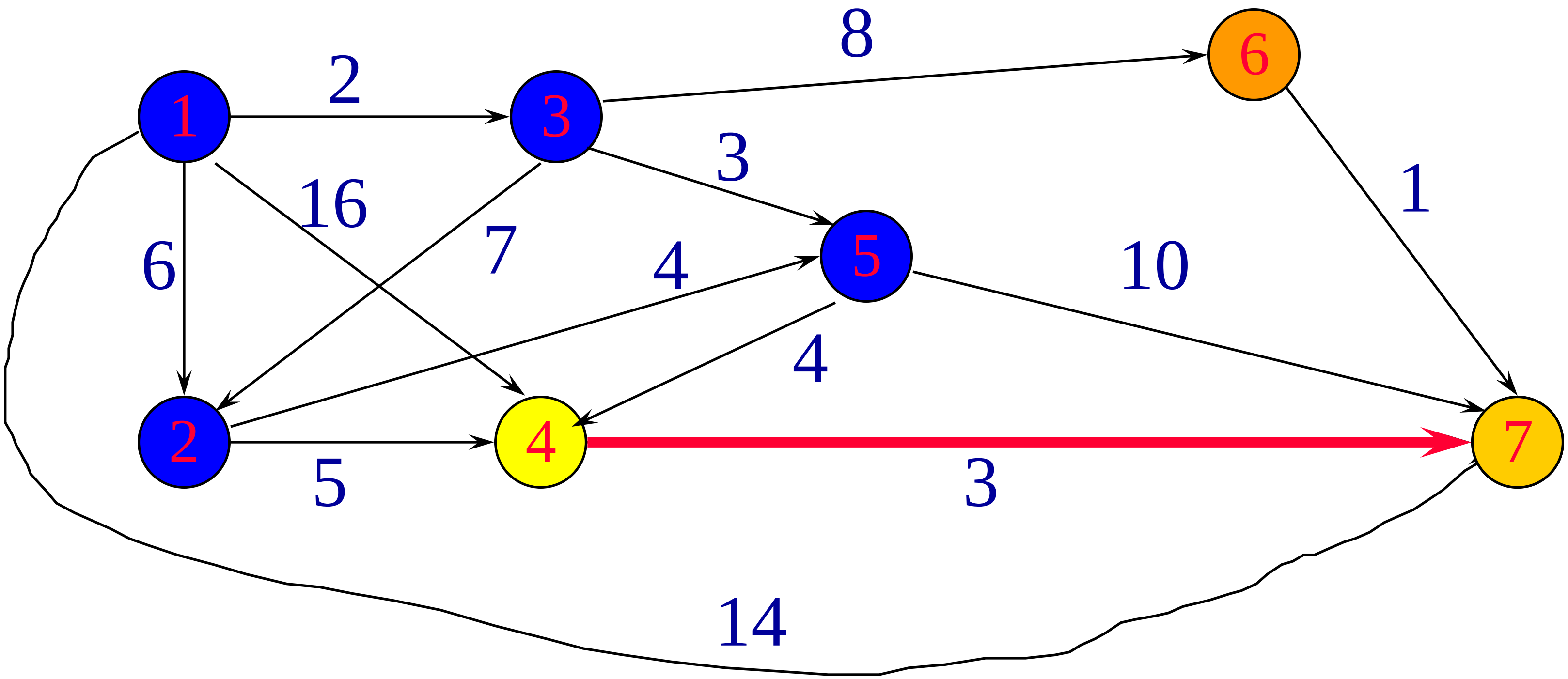
Greedy Single Source All Destinations

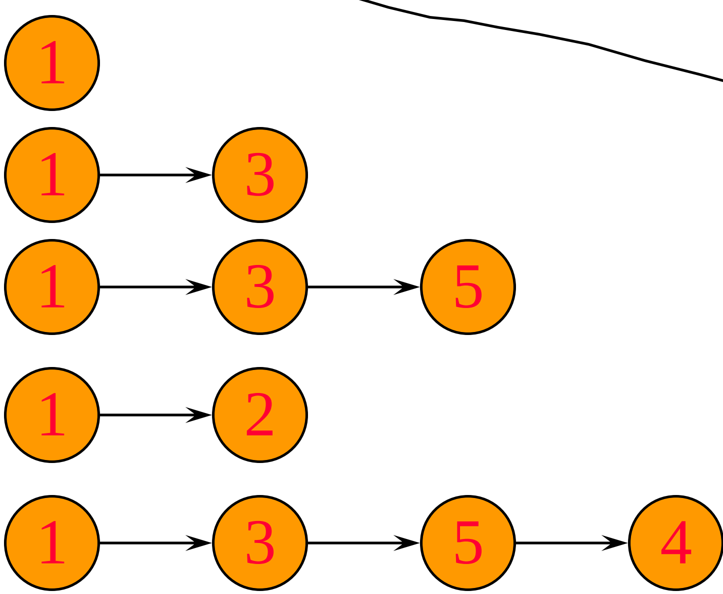




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

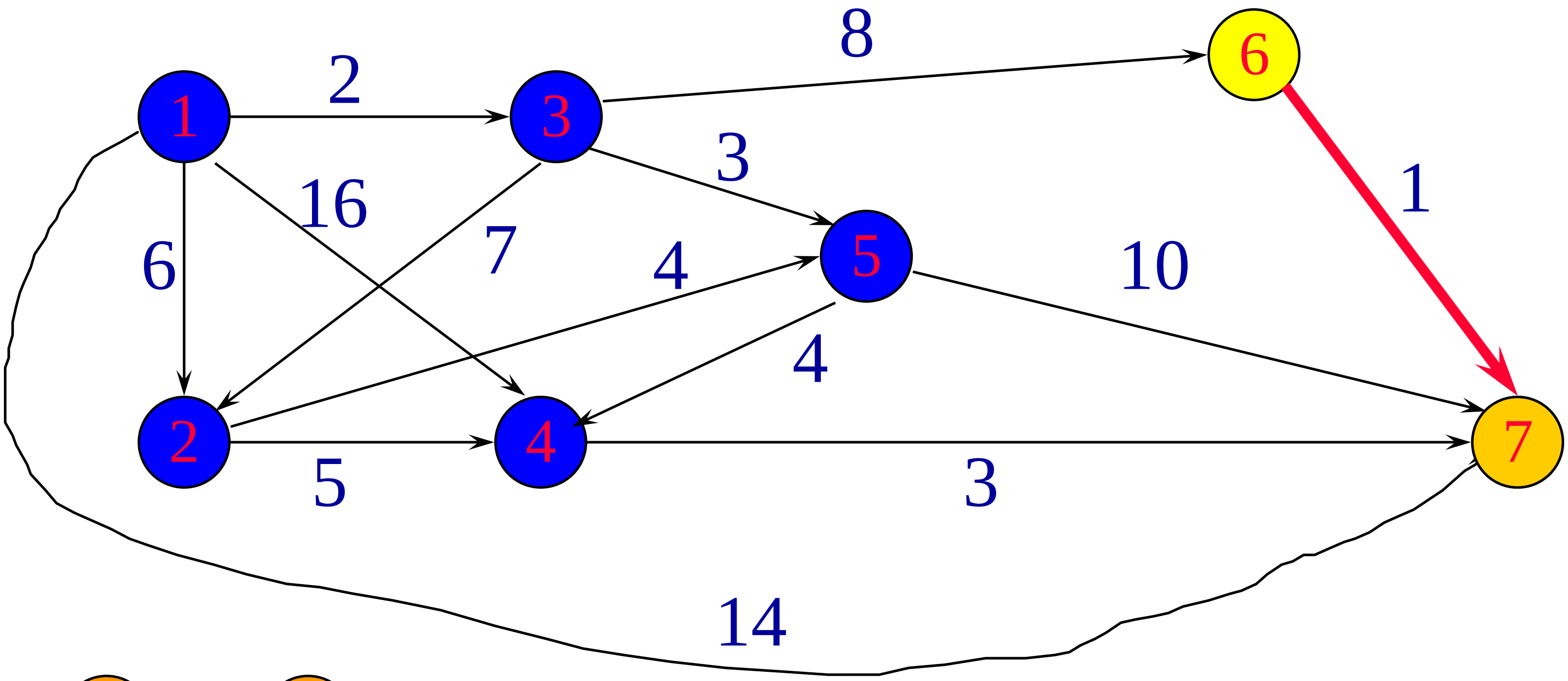
Greedy Single Source All Destinations

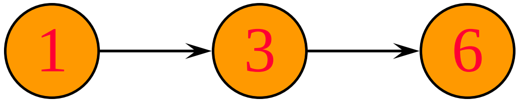




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	12
p	-	1	1	5	3	3	4

Greedy Single Source All Destinations





	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	11
p	-	1	1	5	3	3	6

Greedy Single Source All Destinations

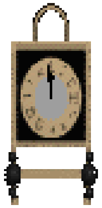
Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

[1]	[2]	[3]	[4]	[5]	[6]	[7]
0	6	2	9	5	10	11
-	1	1	5	3	3	6

Data Structures For Dijkstra’s Algorithm

- The greedy single source all destinations algorithm is known as Dijkstra’s algorithm.
- Implement d() and p() as 1D arrays.
- Keep a linear list L of reachable vertices to which shortest path is yet to be generated.
- Select and remove vertex v in L that has smallest d() value.
- Update d() and p() values of vertices adjacent to v.

Complexity

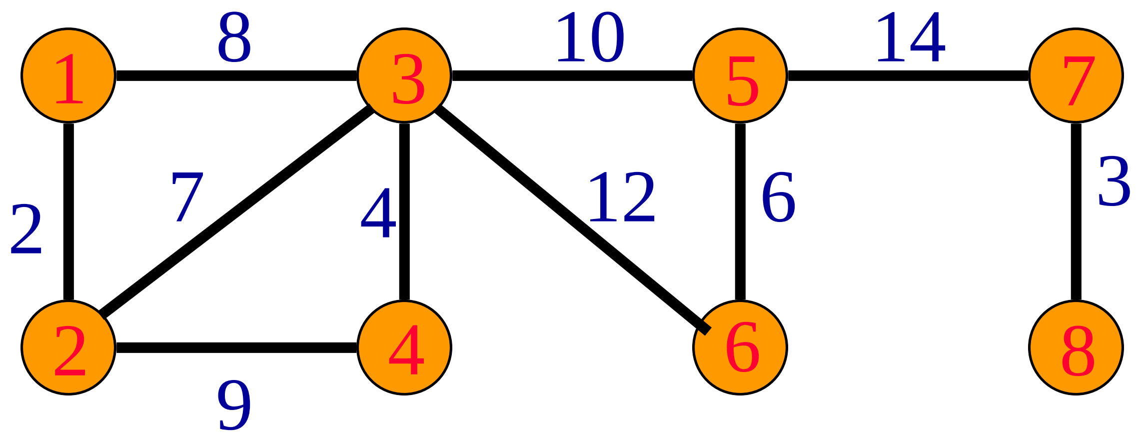


- O(n) to select next destination vertex.
- O(out-degree) to update d() and p() values when adjacency lists are used.
- O(n) to update d() and p() values when adjacency matrix is used.
- Selection and update done once for each vertex to which a shortest path is found.
- Total time is $O(n^2 + e) = O(n^2)$.

12.6 Minimum-Cost Spanning Tree

- weighted connected undirected graph
- spanning tree
- cost of spanning tree is sum of edge costs
- find spanning tree that has minimum cost

Example



- Network has 10 edges.
- Spanning tree has only n - 1 = 7 edges.
- Need to either select 7 edges or discard 3.

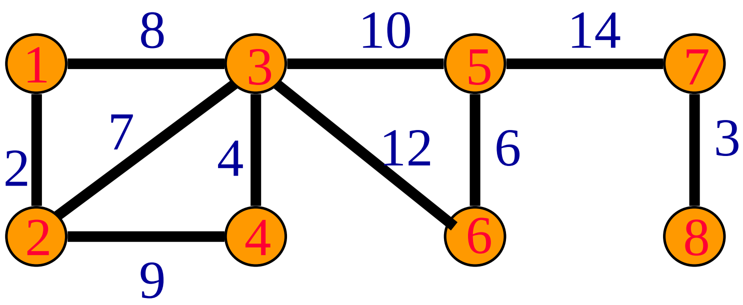
Edge Selection Greedy Strategies

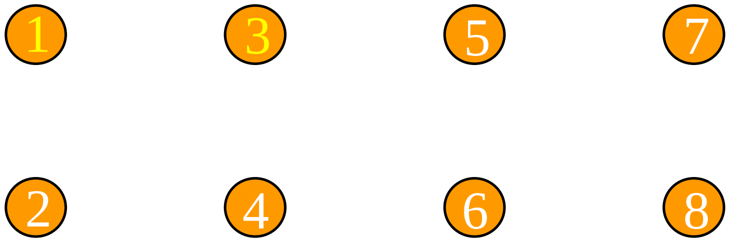
- Start with an n-vertex 0-edge forest. Consider edges in ascending order of cost. Select edge if it does not form a cycle together with already selected edges.
  - Kruskal's method.
- Start with a 1-vertex tree and grow it into an n-vertex tree by repeatedly adding a vertex and an edge. When there is a choice, add a least cost edge.
  - Prim's method.

Edge Selection Greedy Strategies

- Start with an *n*-vertex forest. Each component/tree selects a least cost edge to connect to another component/tree. Eliminate duplicate selections and possible cycles. Repeat until only 1 component/tree is left.
  - Sollin's method.

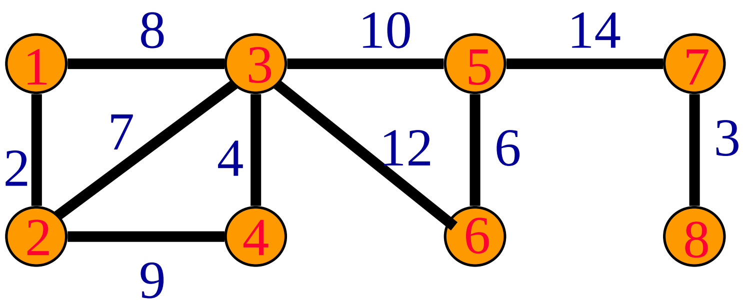
Kruskal's Method

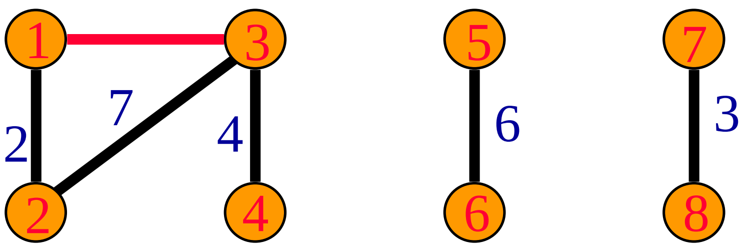




- Start with a forest that has no edges.
- Consider edges in ascending order of cost.
- Edge (1,2) is considered first and added to the forest.

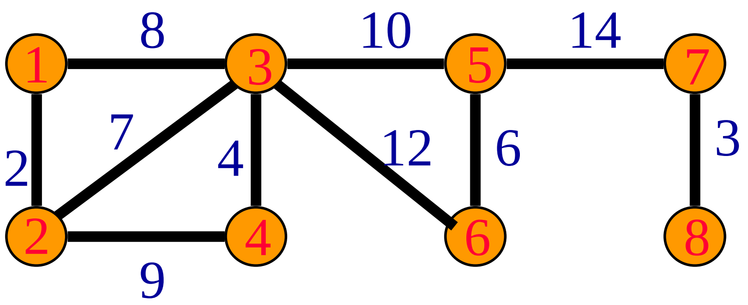
Kruskal's Method

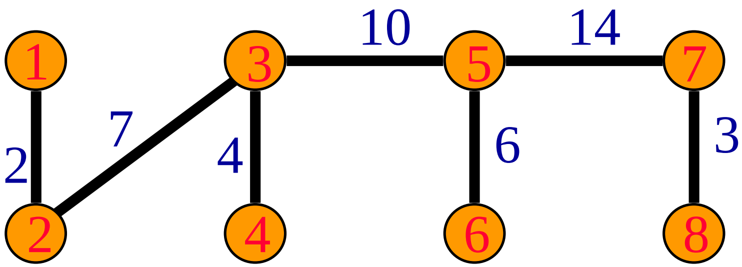




- Edge (7,8) is considered next and added.
- Edge (3,4) is considered next and added.
- Edge (5,6) is considered next and added.
- Edge (2,3) is considered next and added.
- Edge (1,3) is considered next and rejected because it creates a cycle.

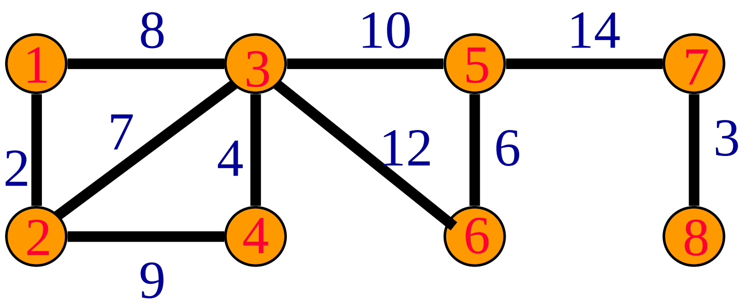
Kruskal's Method

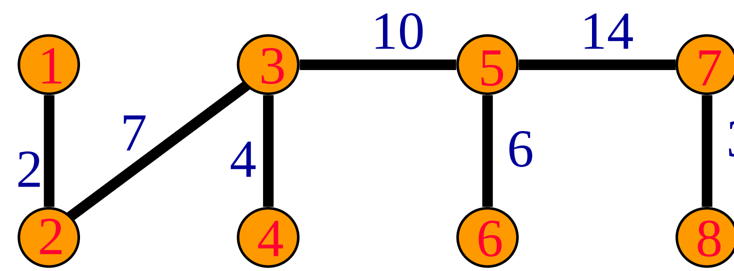




- Edge (2,4) is considered next and rejected because it creates a cycle.
- Edge (3,5) is considered next and added.
- Edge (3,6) is considered next and rejected.
- Edge (5,7) is considered next and added.

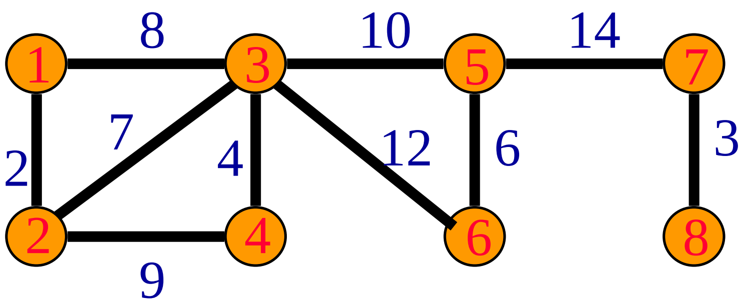
Kruskal’s Method

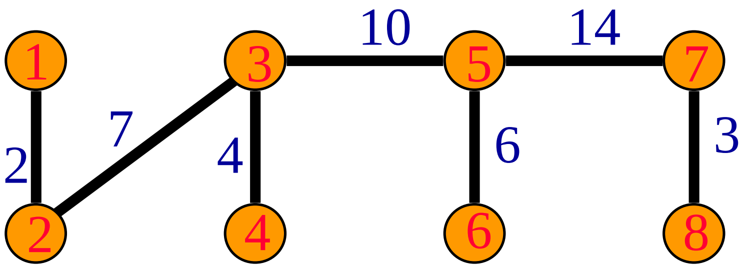




- *n* - 1 edges have been selected and no cycle formed.
- So we must have a spanning tree.
- Cost is 46.
- Min-cost spanning tree is unique when all edge costs are different.

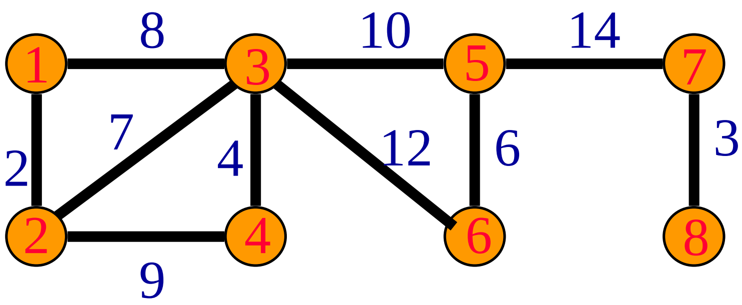
Prim's Method

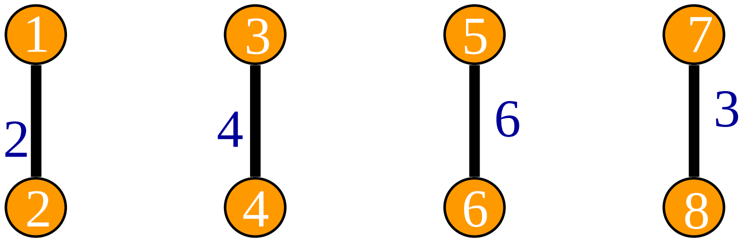




- Start with any single vertex tree.
- Get a 2-vertex tree by adding a cheapest edge.
- Get a 3-vertex tree by adding a cheapest edge.
- Grow the tree one edge at a time until the tree has n - 1 edges (and hence has all n vertices).

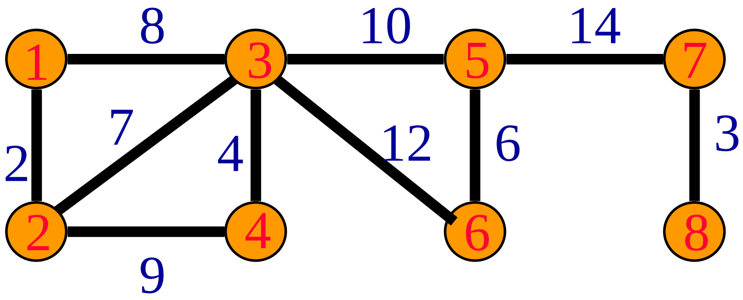
Sollin’s Method

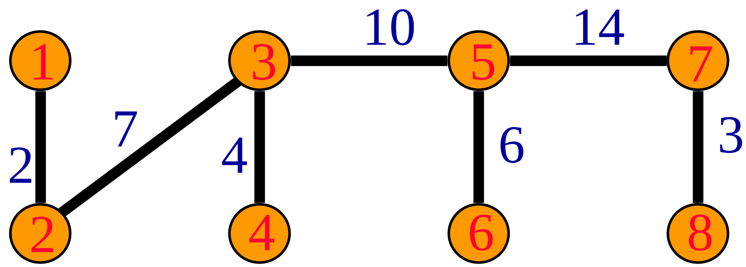




- Start with a forest that has no edges.
- Each component selects a least cost edge with which to connect to another component.
- Duplicate selections are eliminated.
- Cycles are possible when the graph has some edges that have the same cost.

Sollin's Method



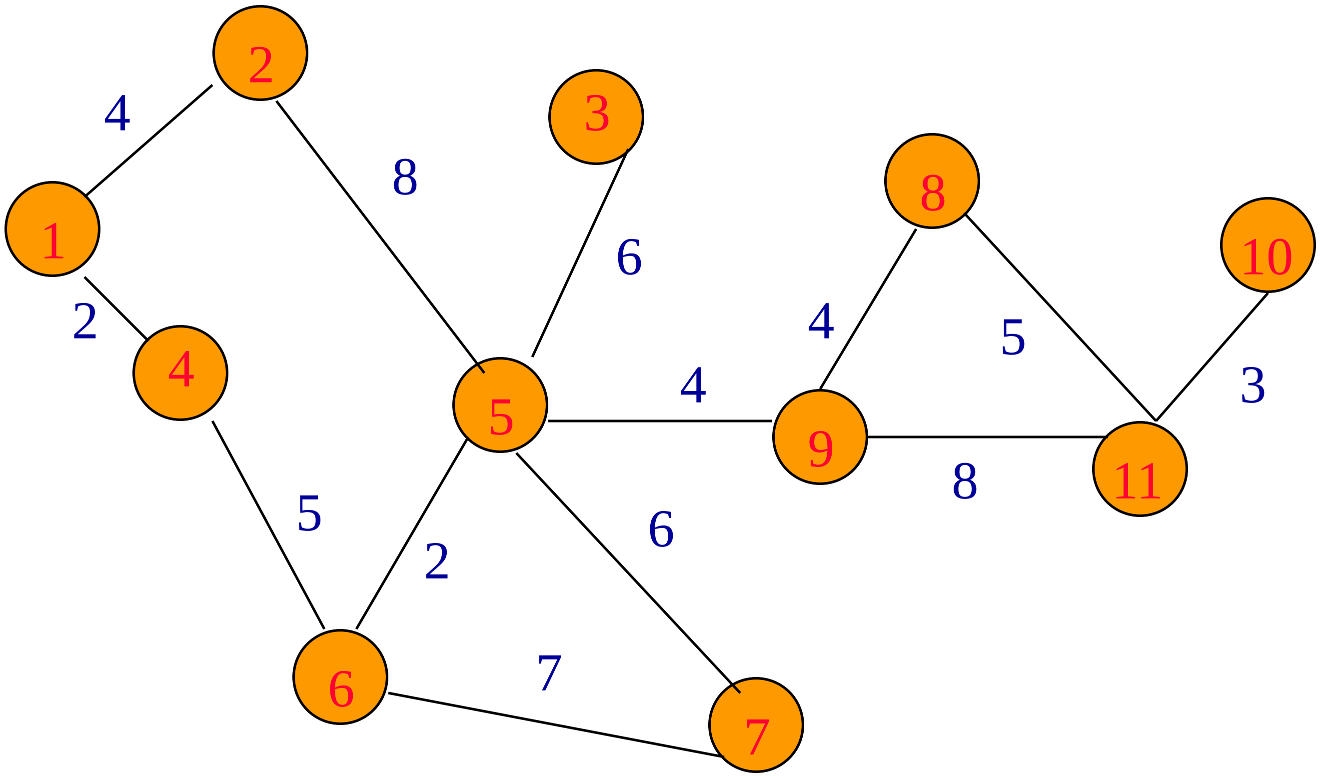


- Each component that remains selects a least cost edge with which to connect to another component.
- Beware of duplicate selections and cycles.

Pseudocode For Kruskal's Method

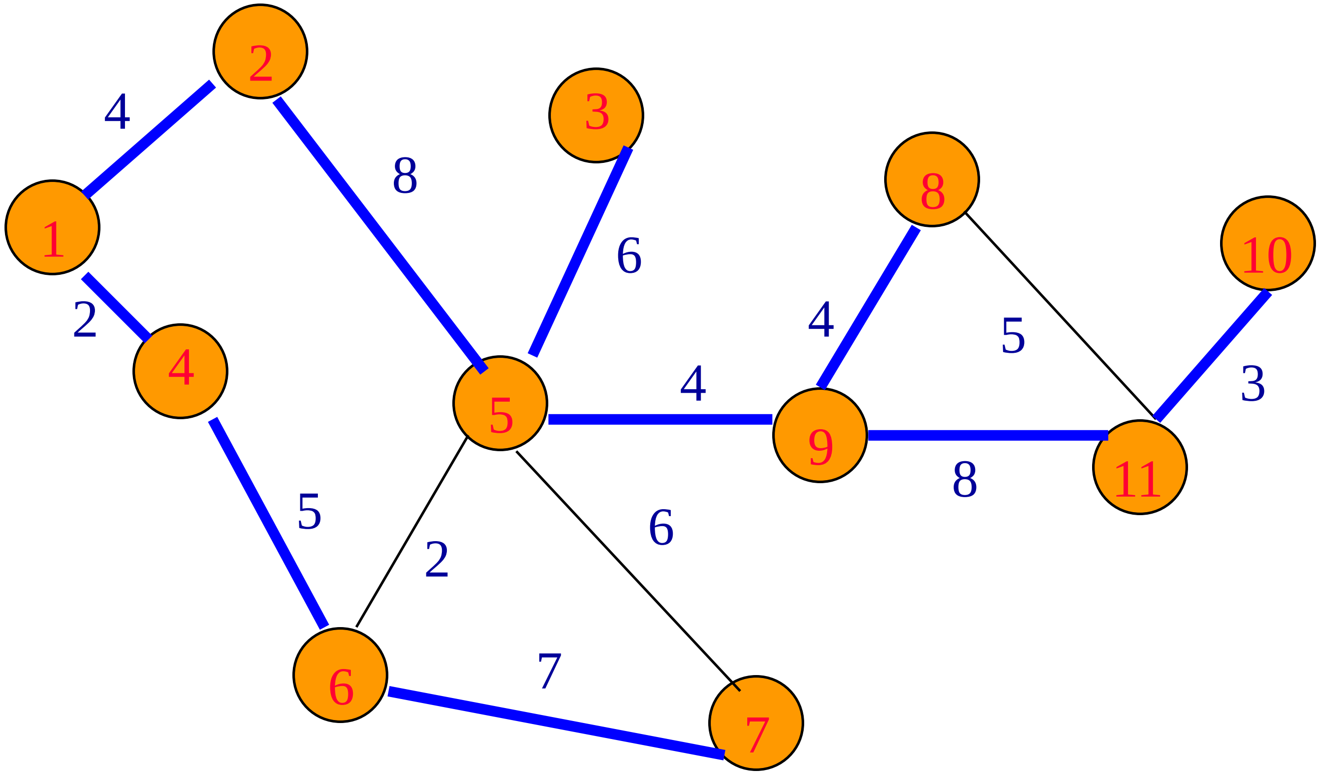
```
Start with an empty set T of edges.
while (E is not empty && |T| != n-1)
{
    Let (u,v) be a least-cost edge in E.
    E = E - {(u,v)}. // delete edge from E
    if ((u,v) does not create a cycle in T)
      Add edge (u,v) to T.
}
if (| T | == n-1) T is a min-cost spanning tree.
else Network has no spanning tree.
```

Minimum Cost Spanning Tree



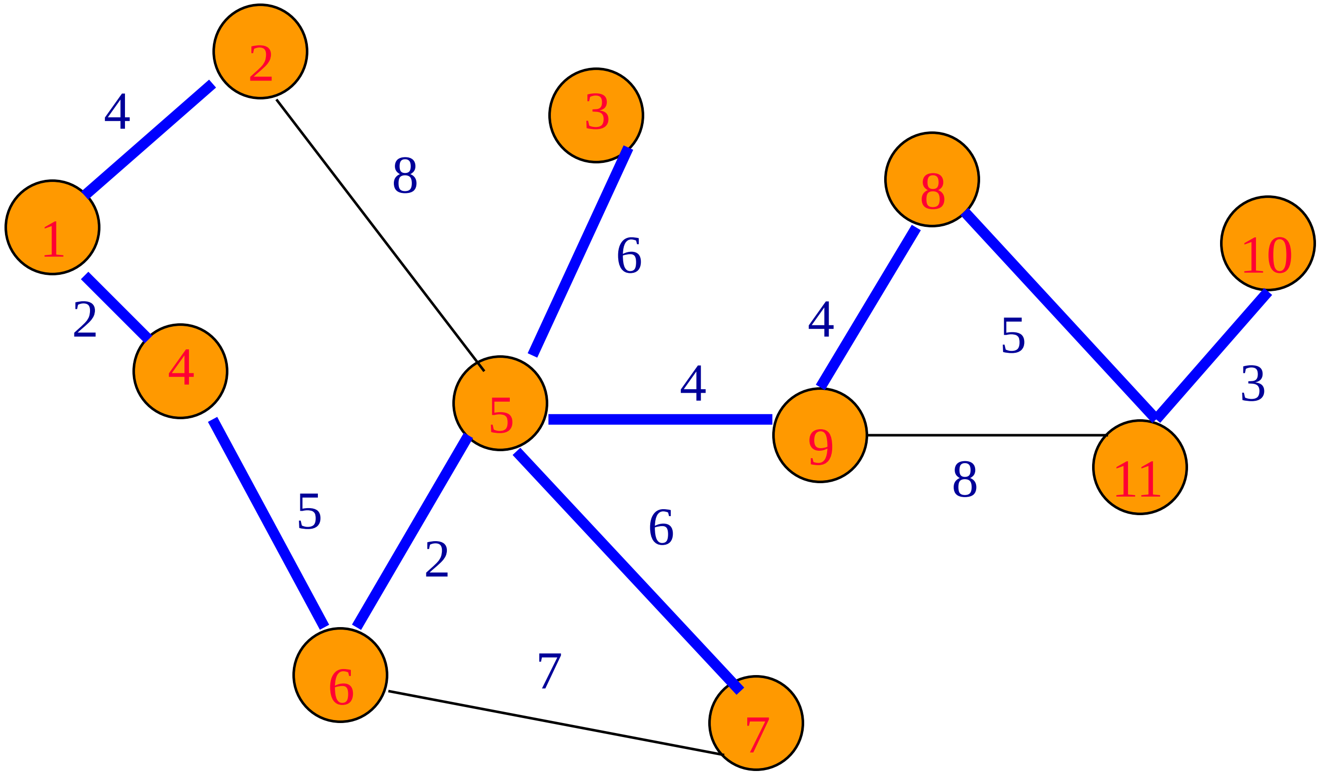
- Tree cost is sum of edge weights/costs.

A Spanning Tree



Spanning tree cost = 51.

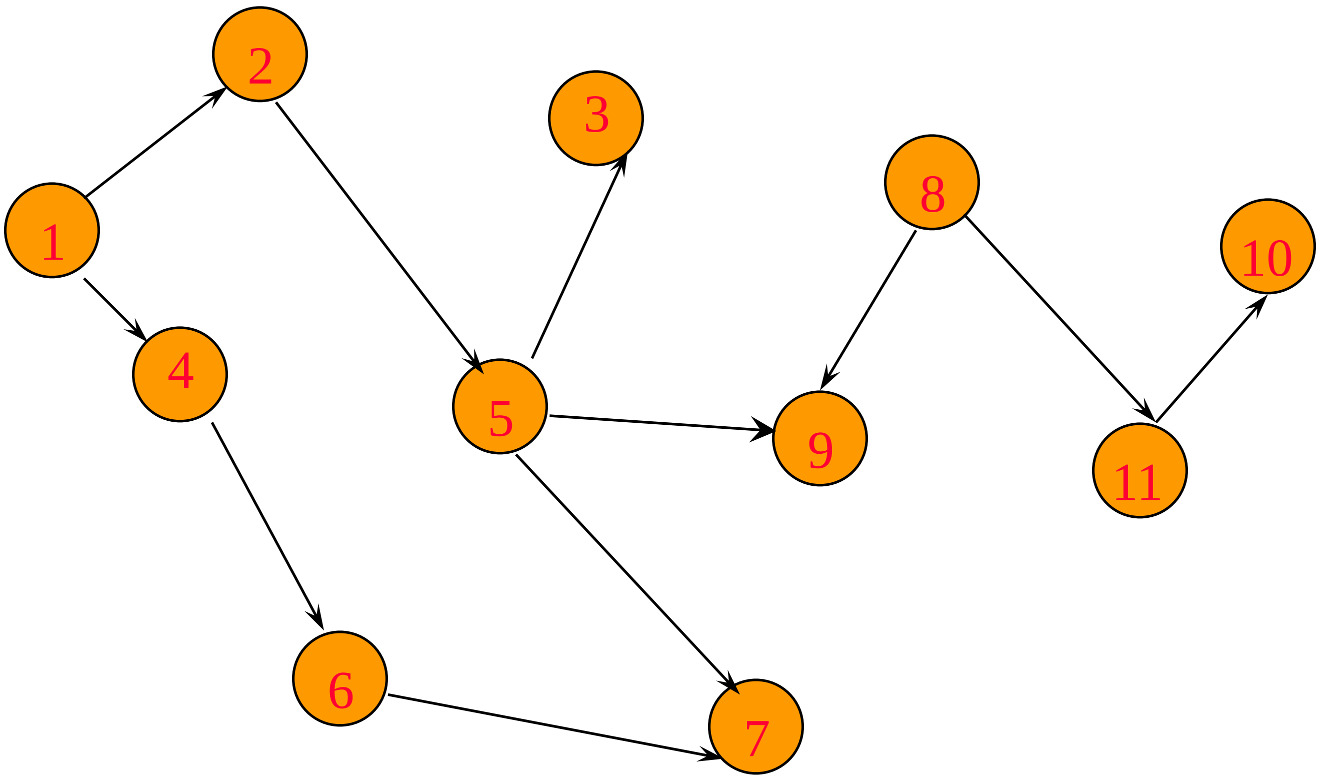
Minimum Cost Spanning Tree



Spanning tree cost = 41.

课后练习

（1）写出该图的邻接矩阵及邻接表；



（2）按所写的邻接表求出从顶点 D开始的深度和广度优先搜索遍历序列。