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

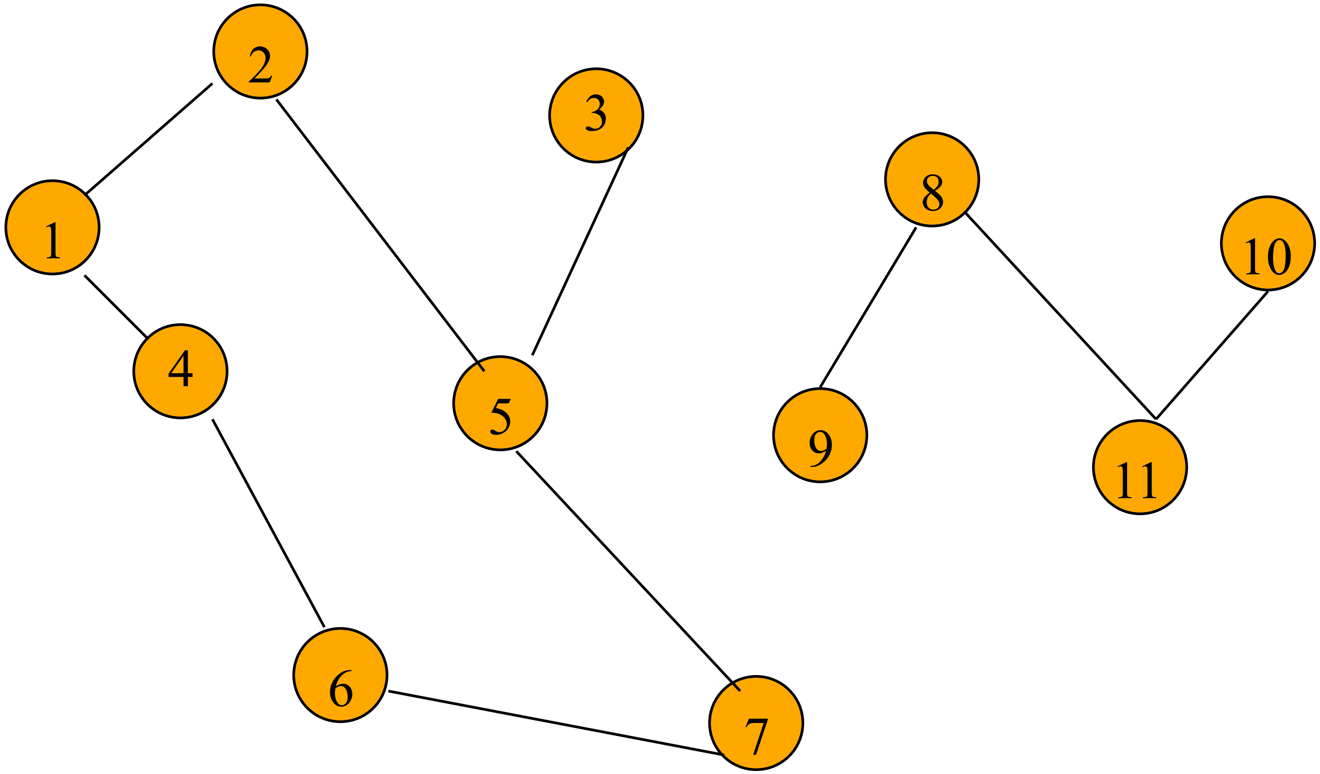
- Undirected edge has no orientation (u,v).

  u ——— v

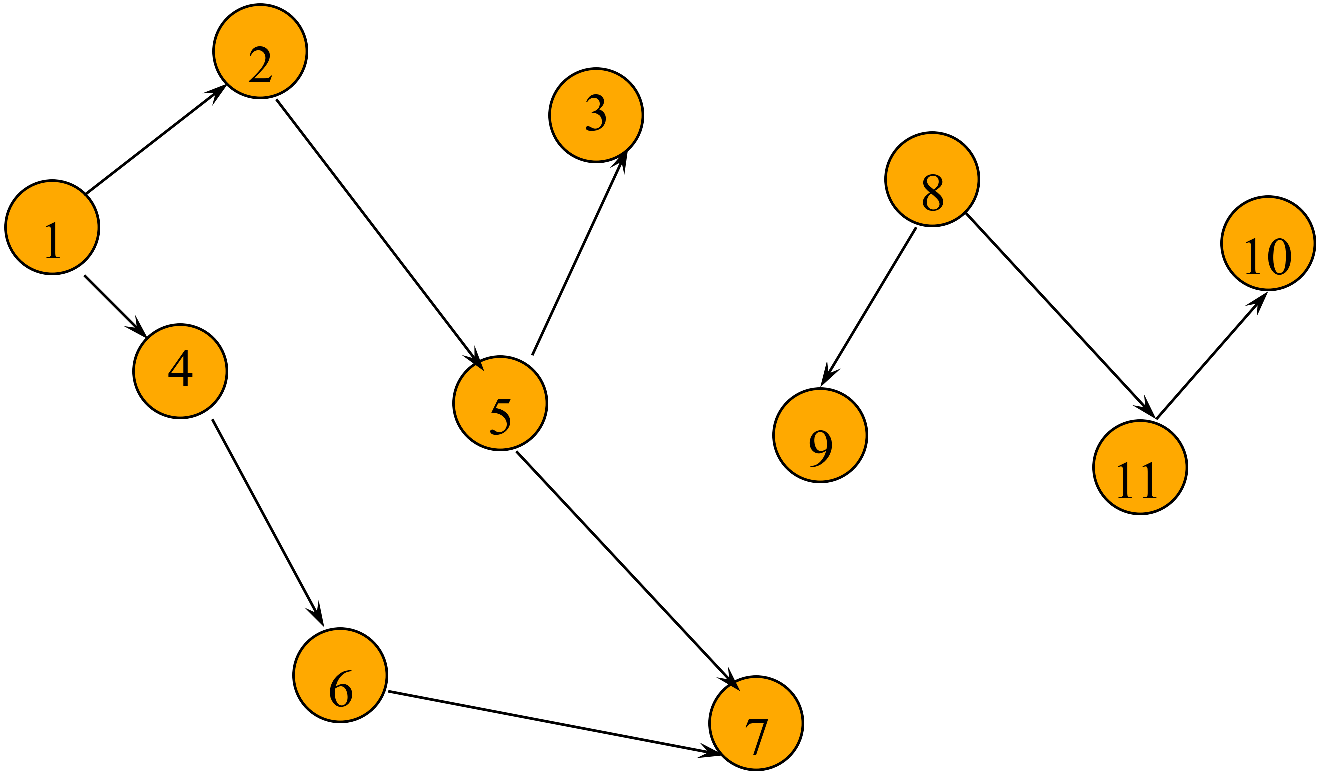
- Undirected graph => no oriented edge.
- Directed graph => every edge has an orientation.

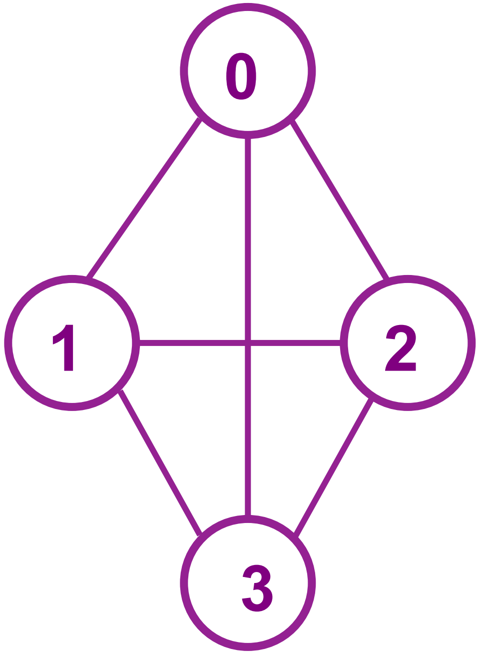
- If (u, v) $\in$E(G), we say u and v are **adjacent** and edge (u, v) is **incident on** vertices u and v. If <u, v> is a directed edge, then vertex u is **adjacent to** v, and v is adjacent from u, <u, v> is **incident to** u and v

Undirected Graph



Directed Graph (Digraph)

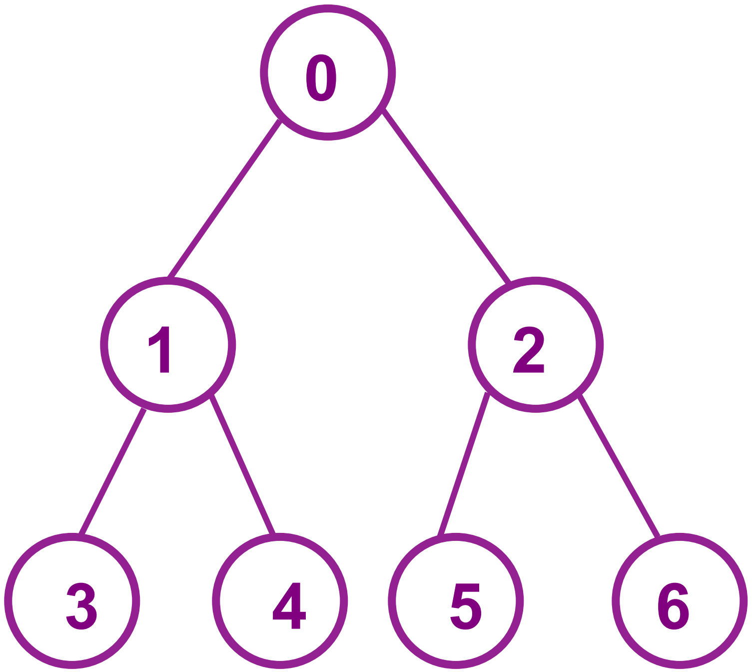




G_1:

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

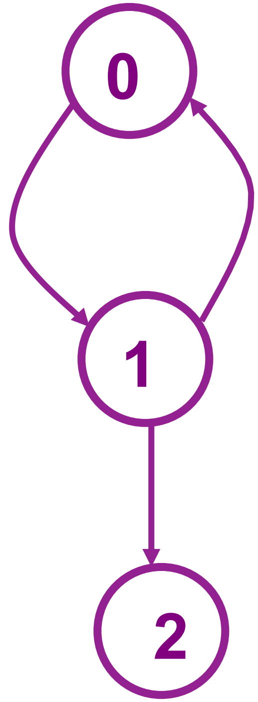
$E(G_1)=\{(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)\}$



G_2:

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

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

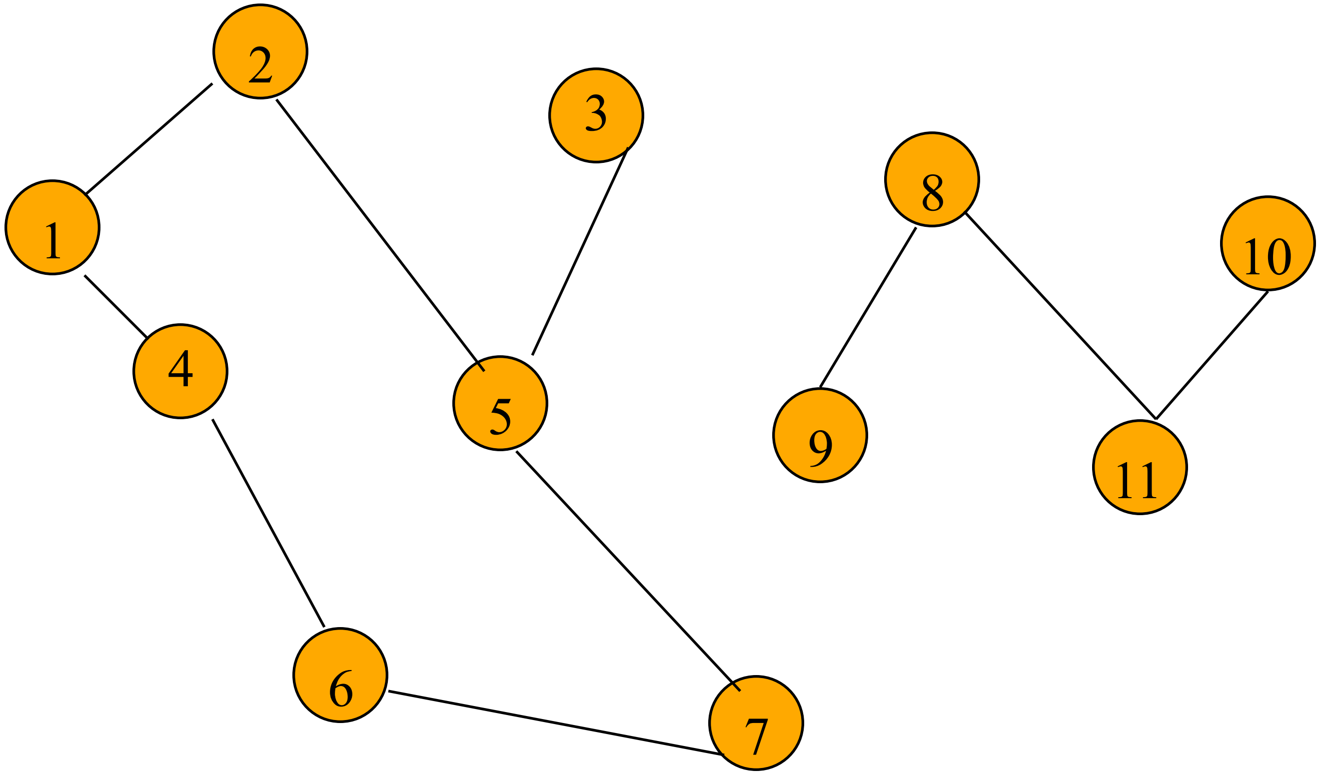


G_3:

$V(G_3) = \{0,1,2\}$

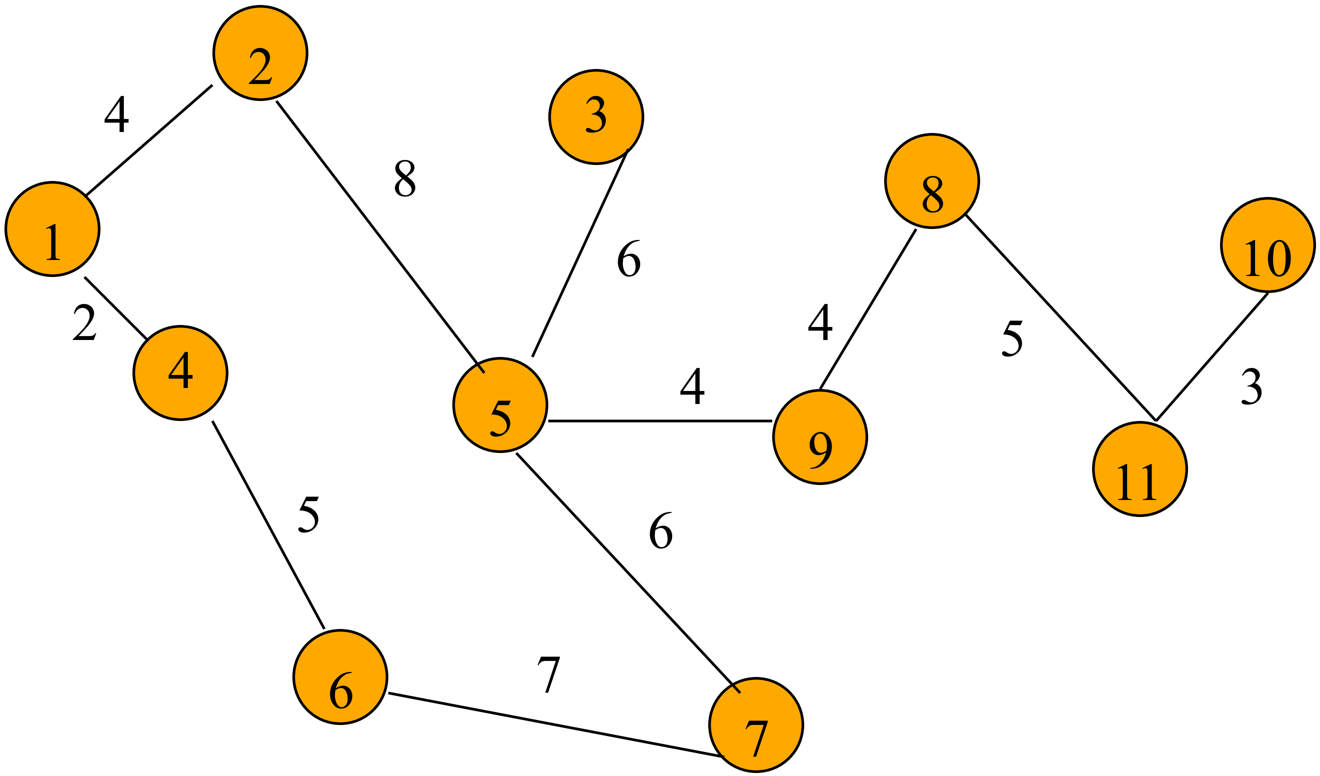
$E(G_3) = \{<0,1>,<1,0>,<1,2>\}$ (directed)

Applications—Communication Network



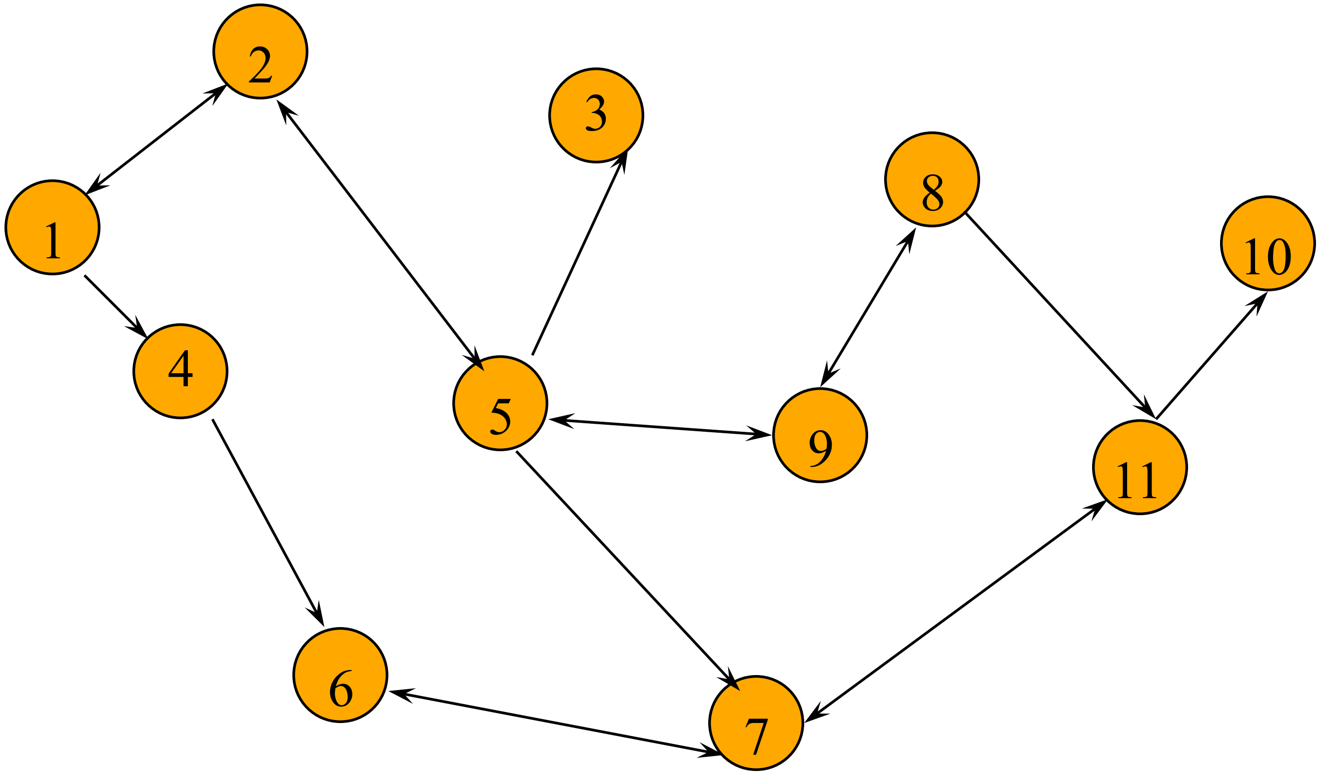
- Vertex = city, edge = communication link.

Driving Distance/Time Map



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

Street Map



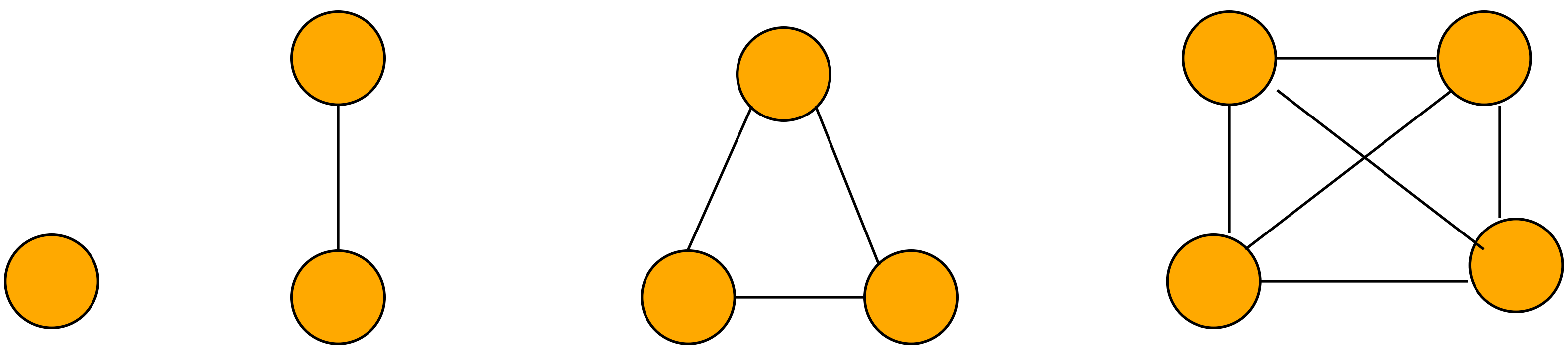
- Some streets are one way.

Restrictions:

- (v, v) or <v, v> is not legal, such edges are known as self edges
- Multiple occurrences of the same edges are not allowed. If allowed, we get a multigraph

Complete Undirected Graph

Has all possible edges.



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

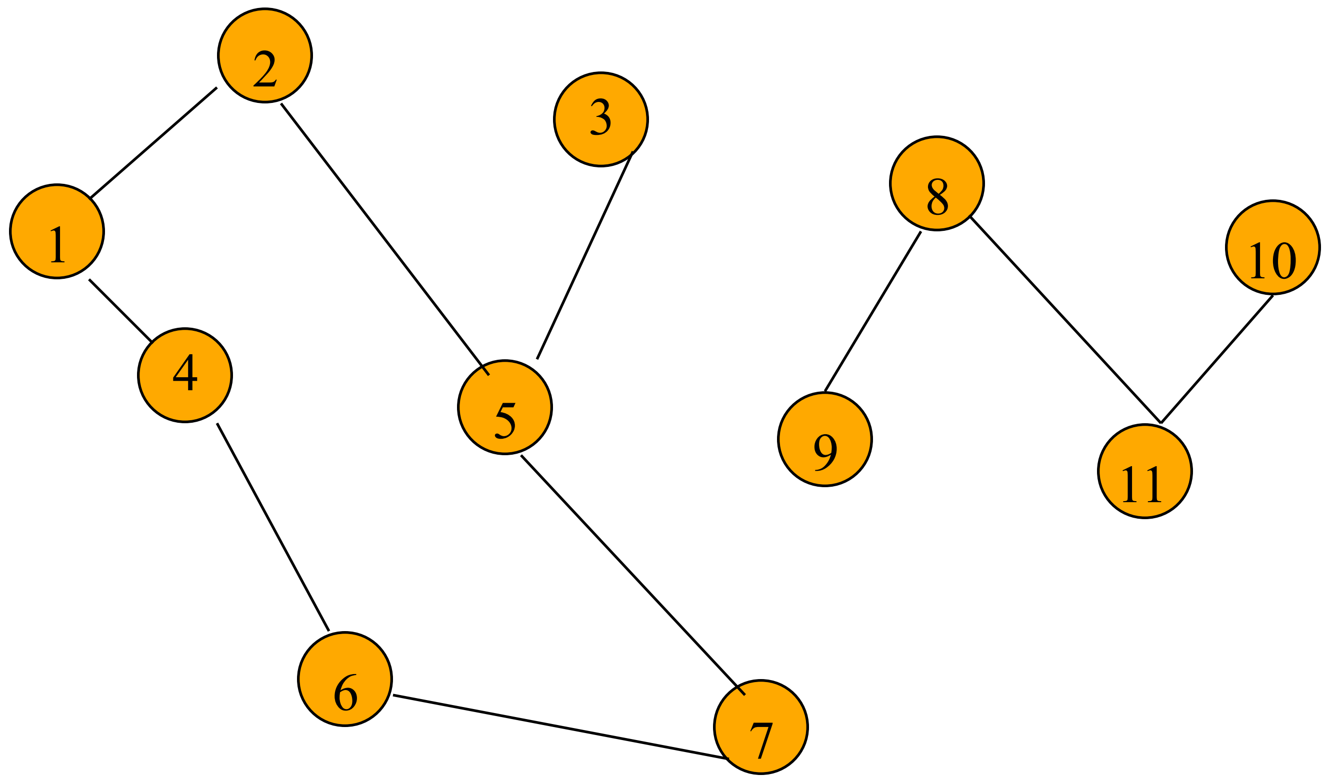
Number Of Edges—Undirected Graph

- 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).

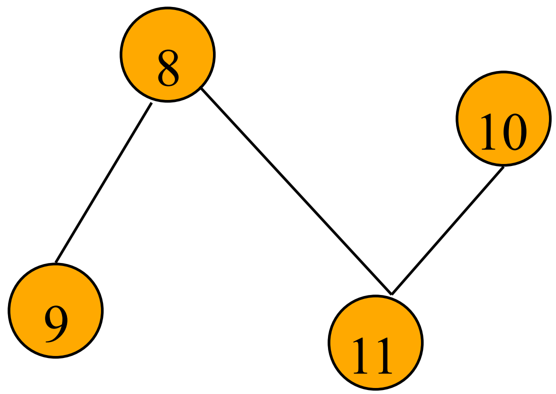
Vertex Degree



Number of edges incident to vertex.

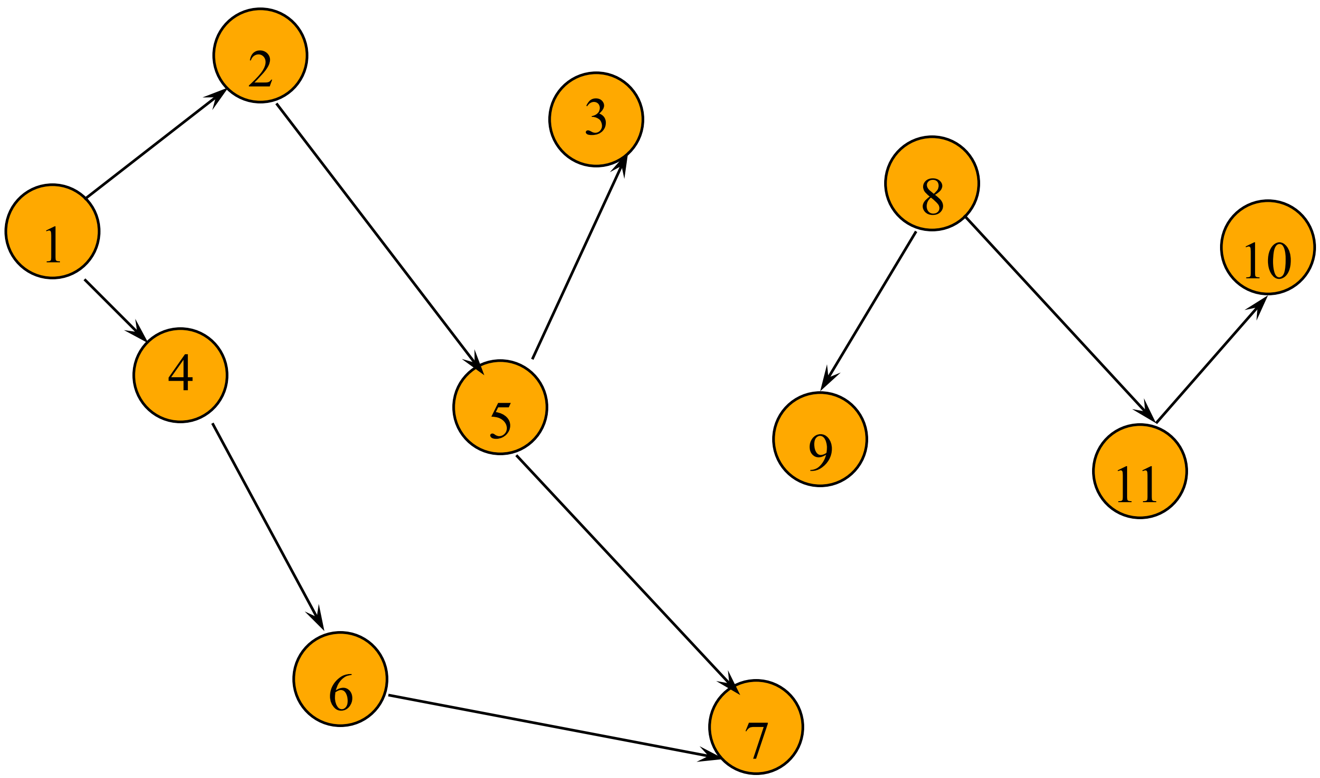
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



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

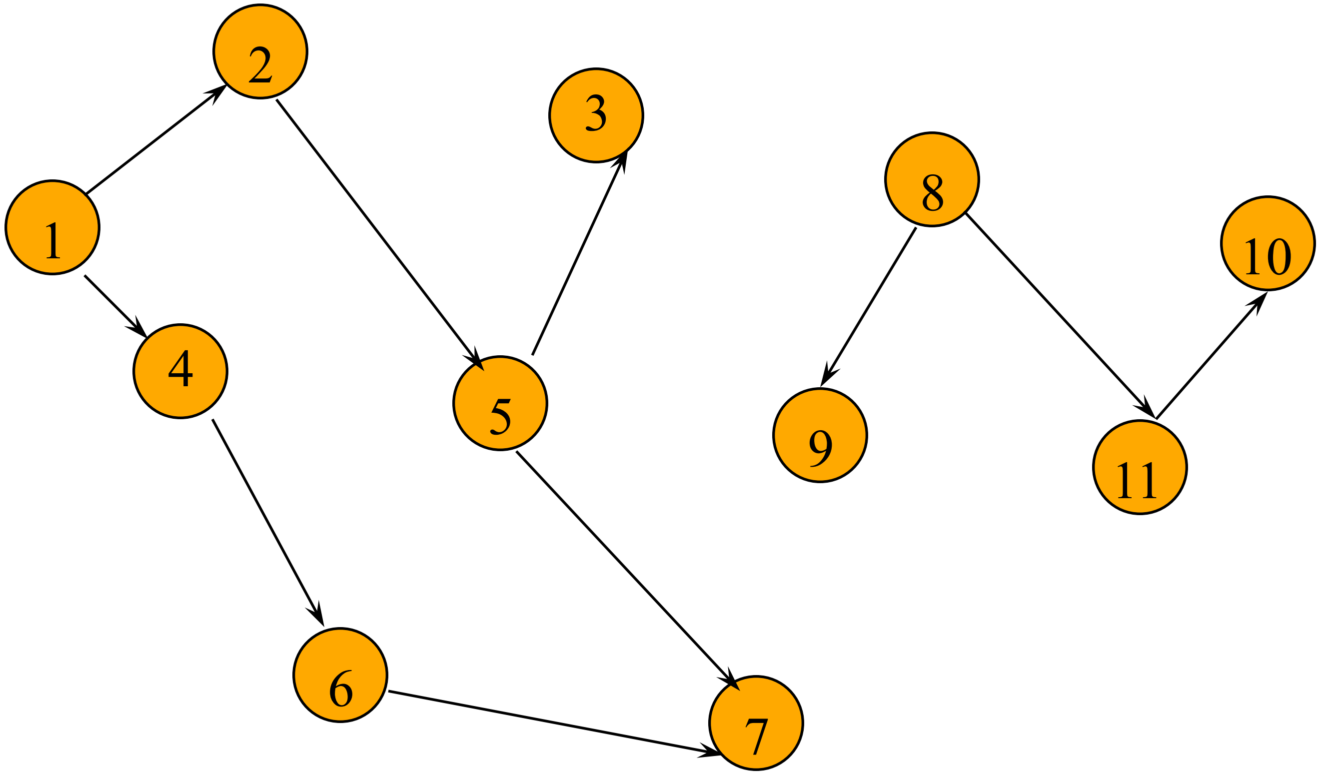
In-Degree Of A Vertex



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

Notations

- A **subgraph** of G is a graph G` such that V(G`) $\subseteq$ V(G) and E(G`) $\subseteq$ E(G).
- A **path** from u to v in 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). If G` is directed, then $<u, i_1>$, $<i_1, i_2>$,…,$<i_k, v>$ are edges in E(G`).

Notations

- A **simple path** is a path in which all vertices except possibly the first and last are distinct.
- A **cycle** is a simple path in which the first and last vertices are the same.
- For directed graph, we have **directed paths** and **cycles**.

Notations

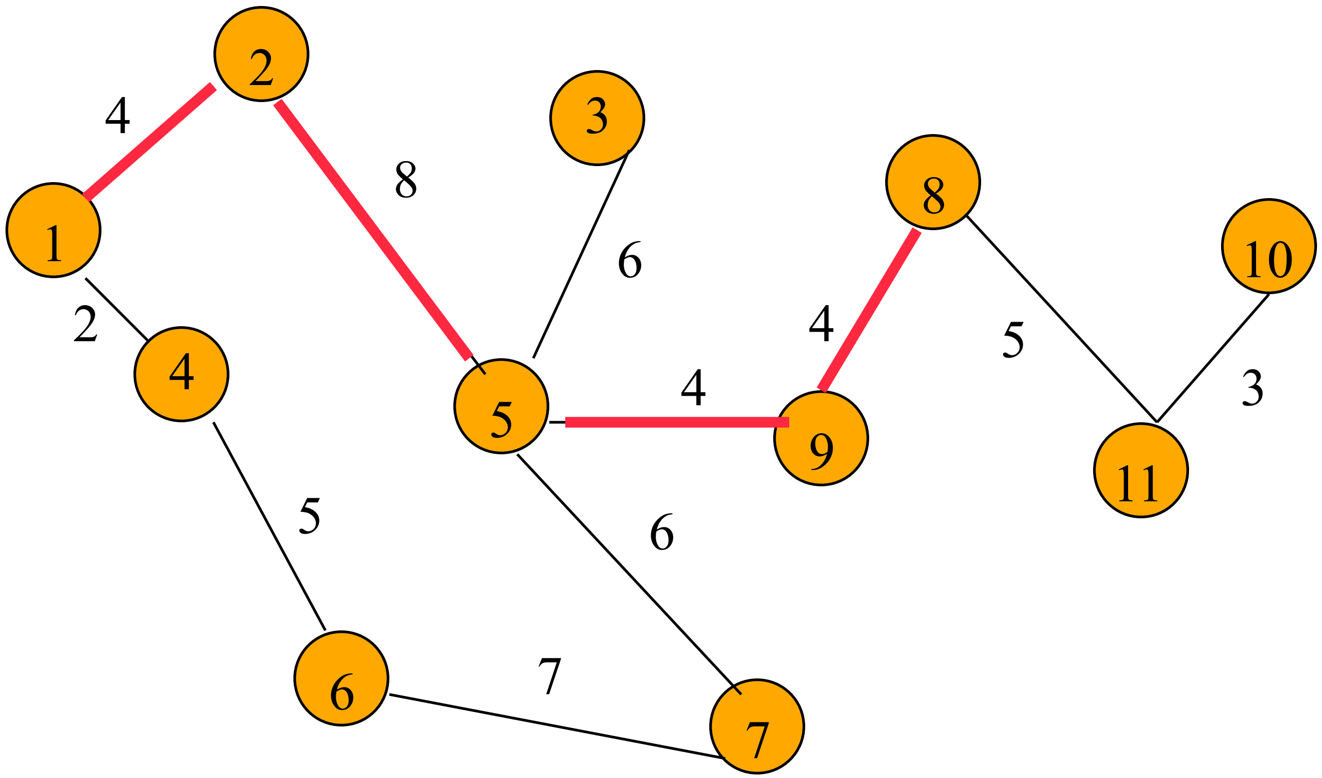
- The **length** of a path is the number of edges on it.
- The **length** of a path is the sum of weights of edges on it.

Sample Graph Problems

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

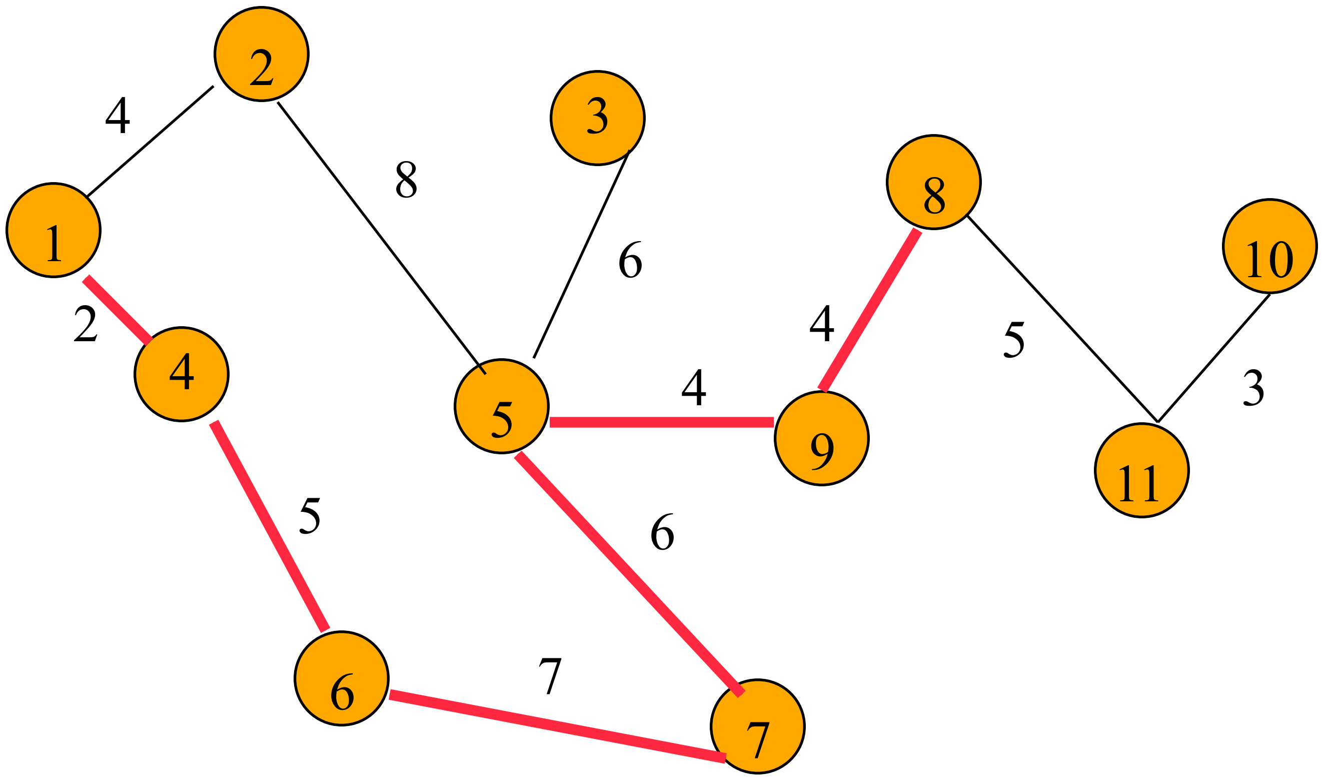
Path Finding

Path between 1 and 8.



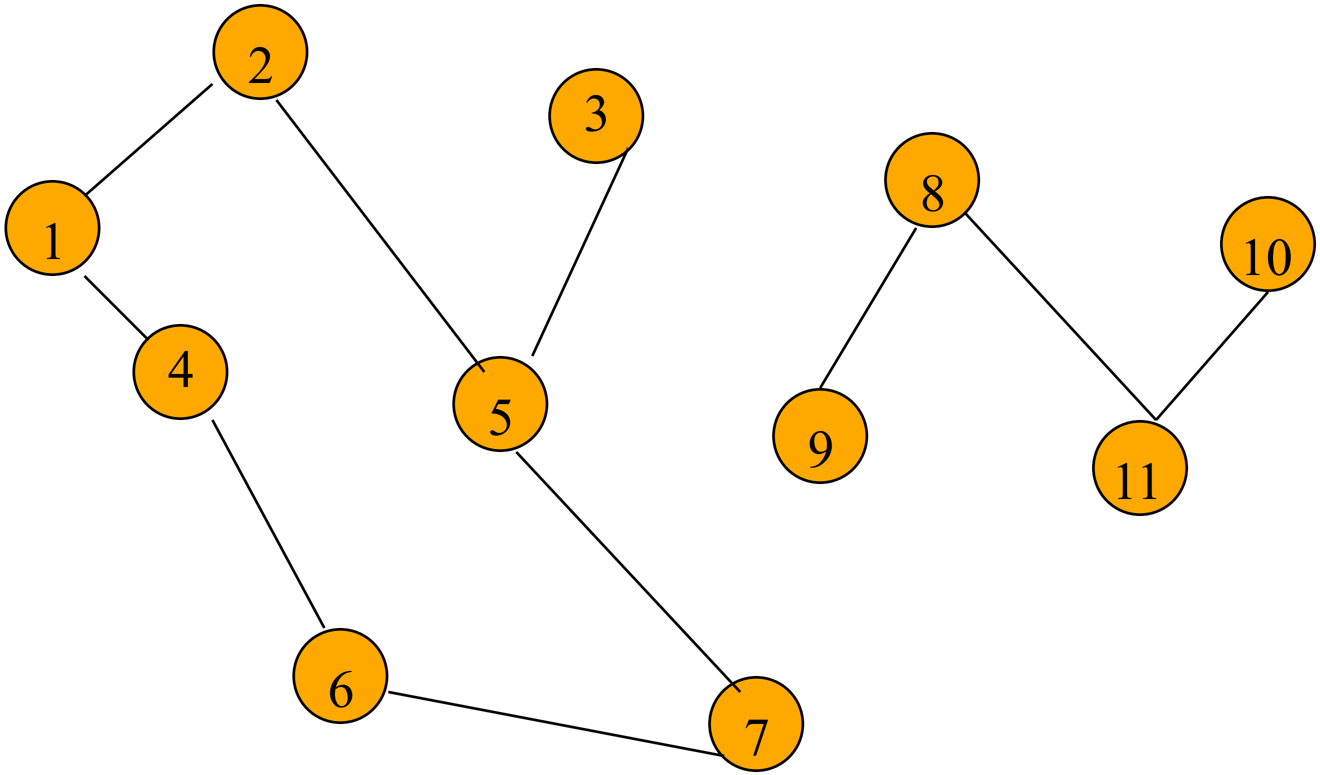
Path length is 20.

Another Path Between 1 and 8



Path length is 28.

Example Of No Path



No path between 2 and 9.

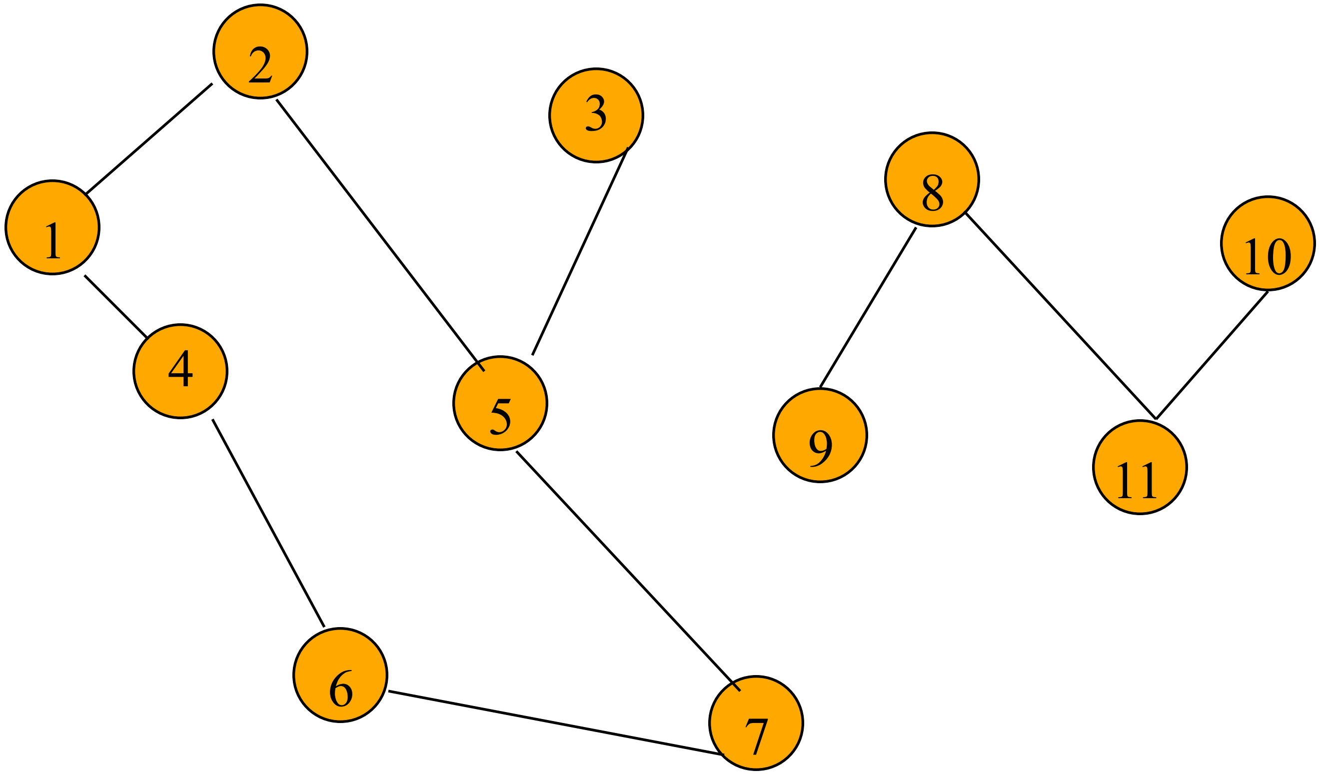
Connected Graph

- Undirected graph.
- u and v are **connected** iff there is a path in G from u to v (also from v to u)
- **Connected Graph**: There is a path between every pair of vertices.

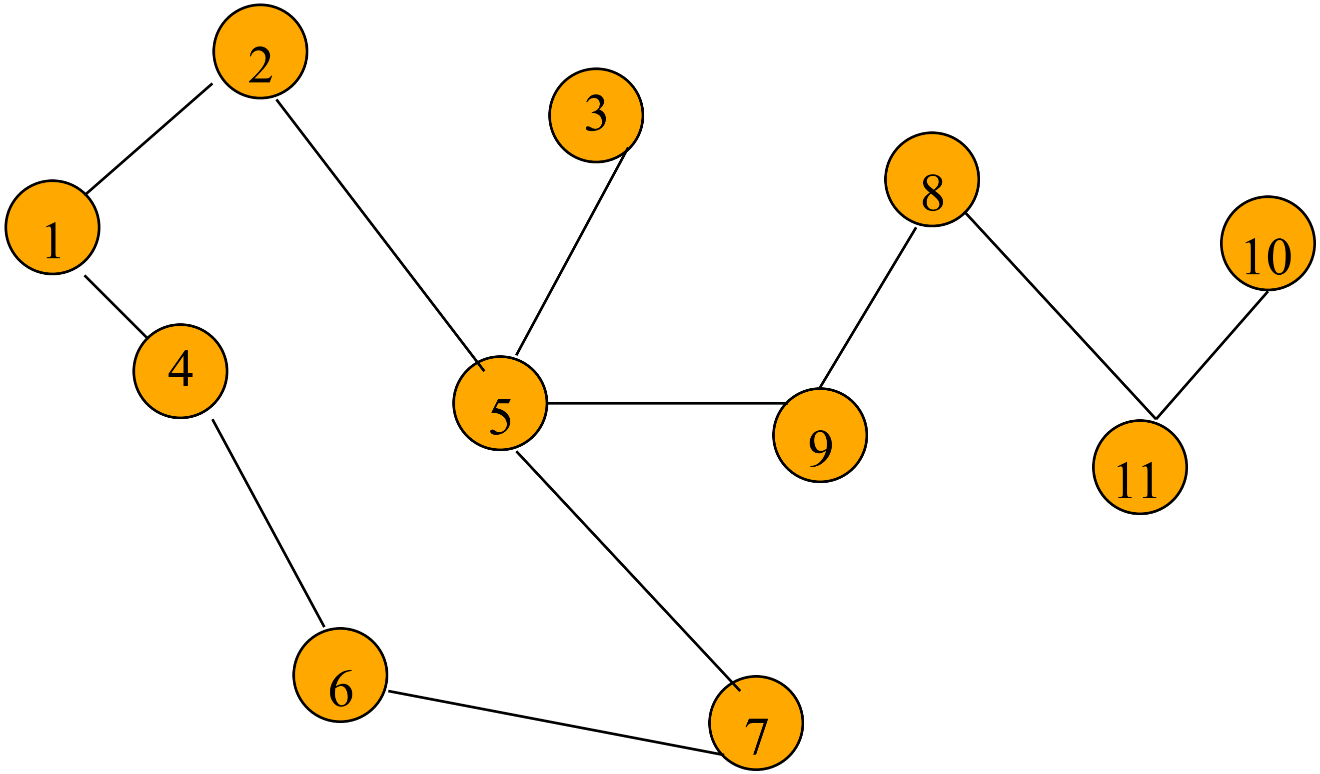
Connected Graph

- Directed graph.
- A directed G is **strongly connected** iff for every pair of distinct u and v in V(G), there is a directed path from u to v and also from v to u.
- A **strongly connected component** is a maximal subgraph that is strongly connected.

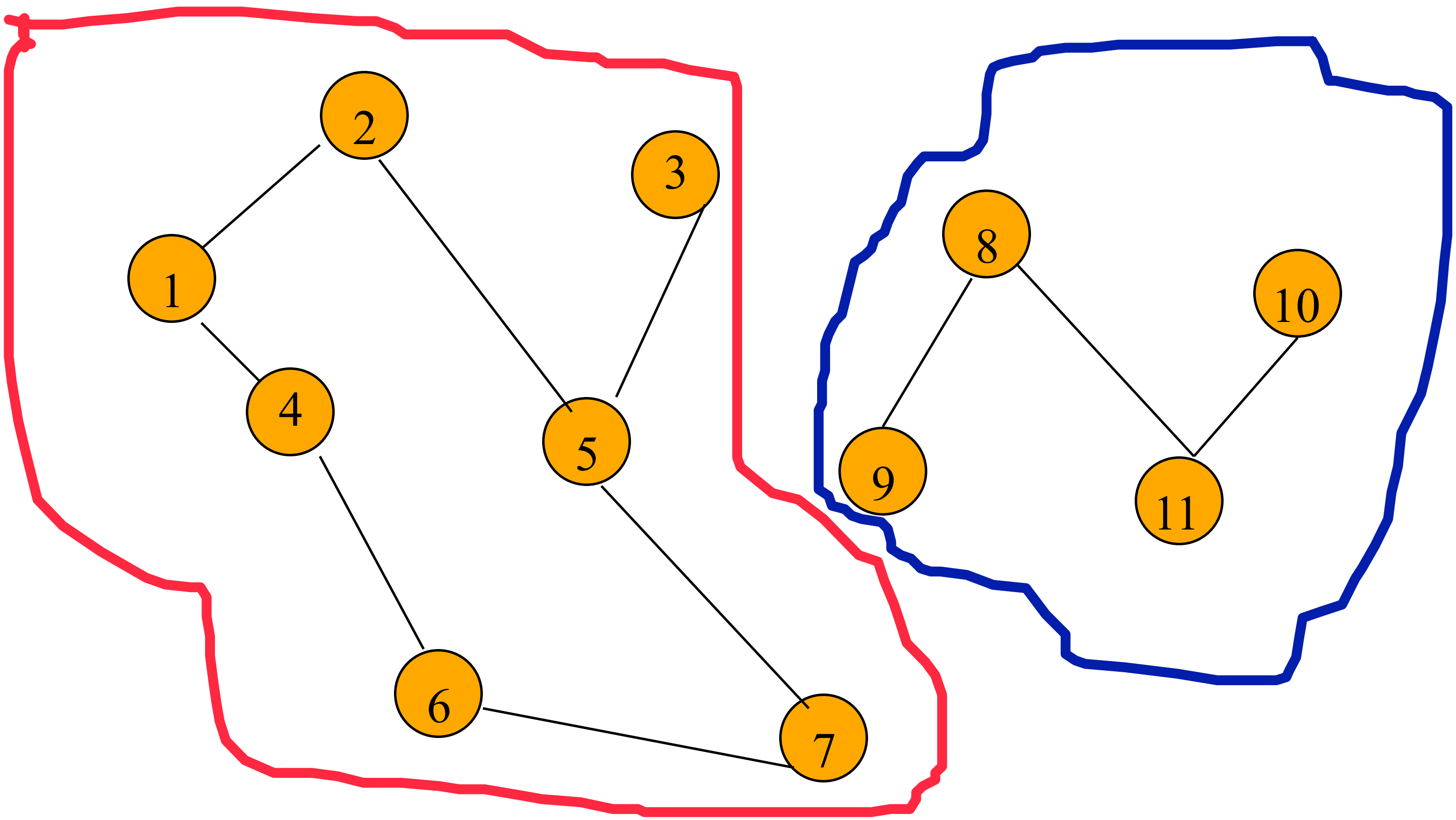
Example Of Not Connected



Connected Graph Example



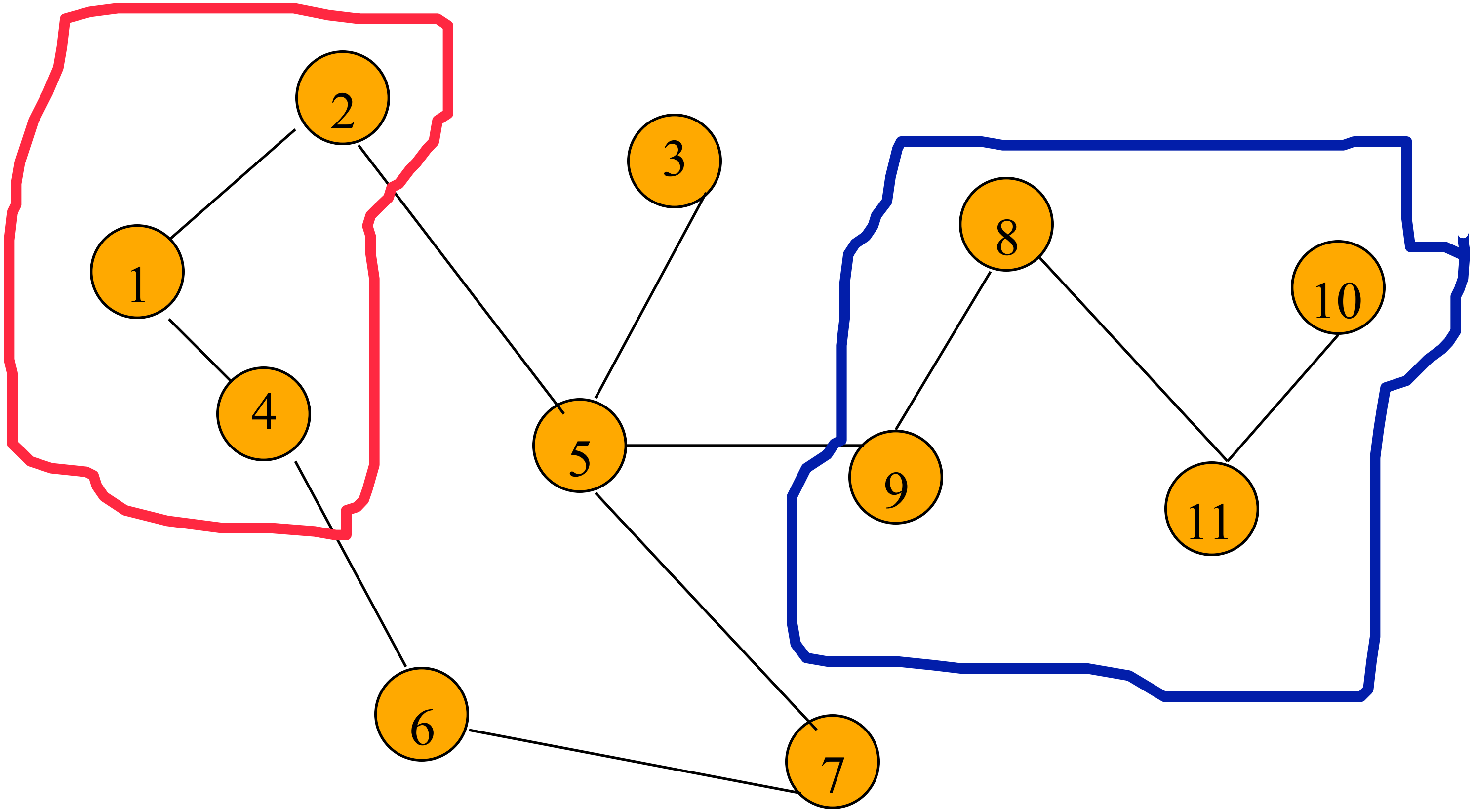
Connected Components



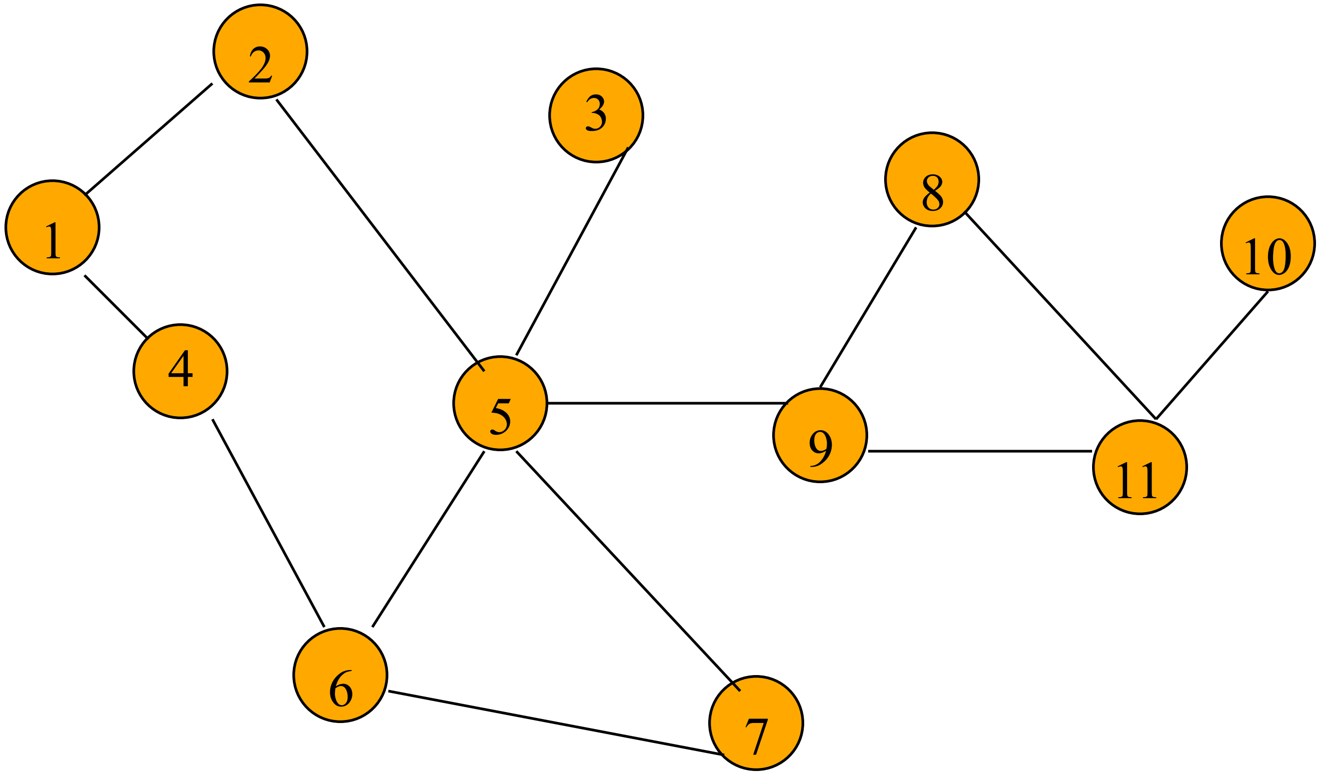
Connected Component

- 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

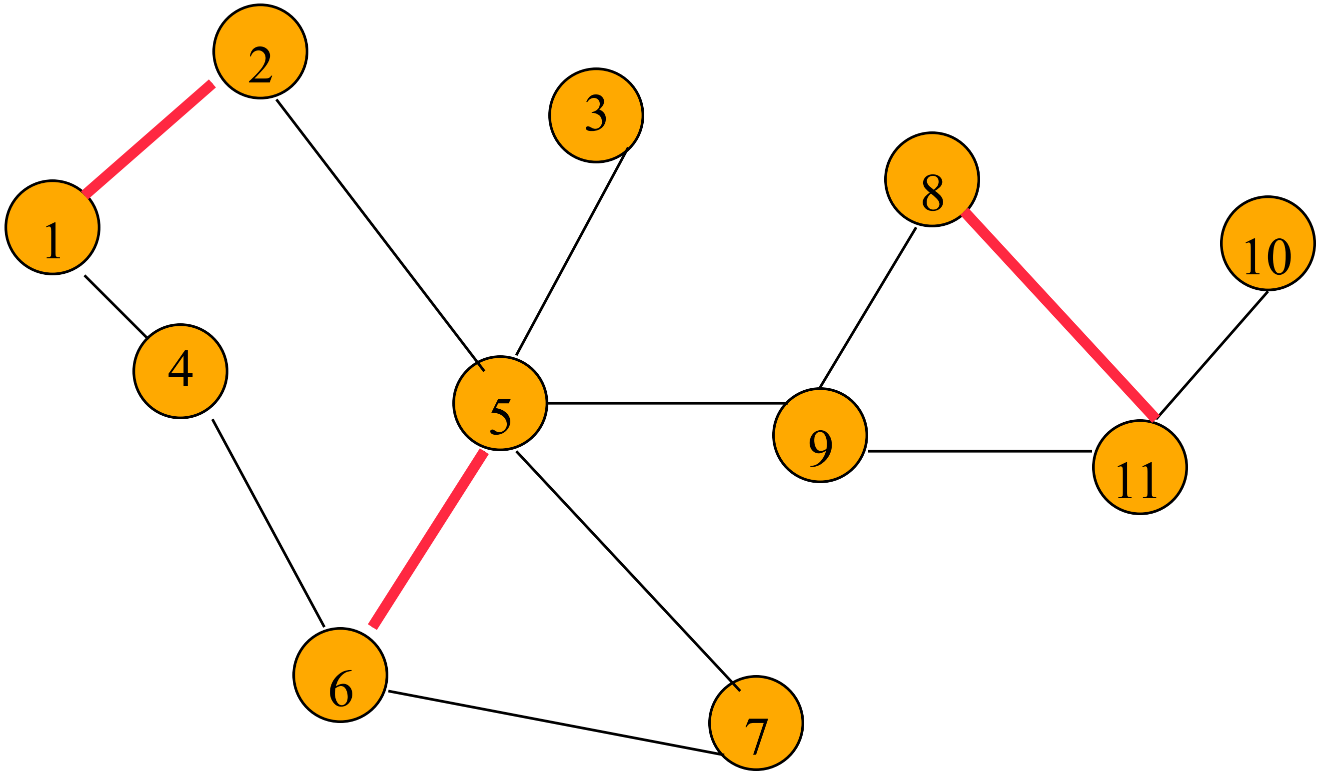


Each edge is a link that can be constructed
(i.e., a feasible link).

Communication Network Problems

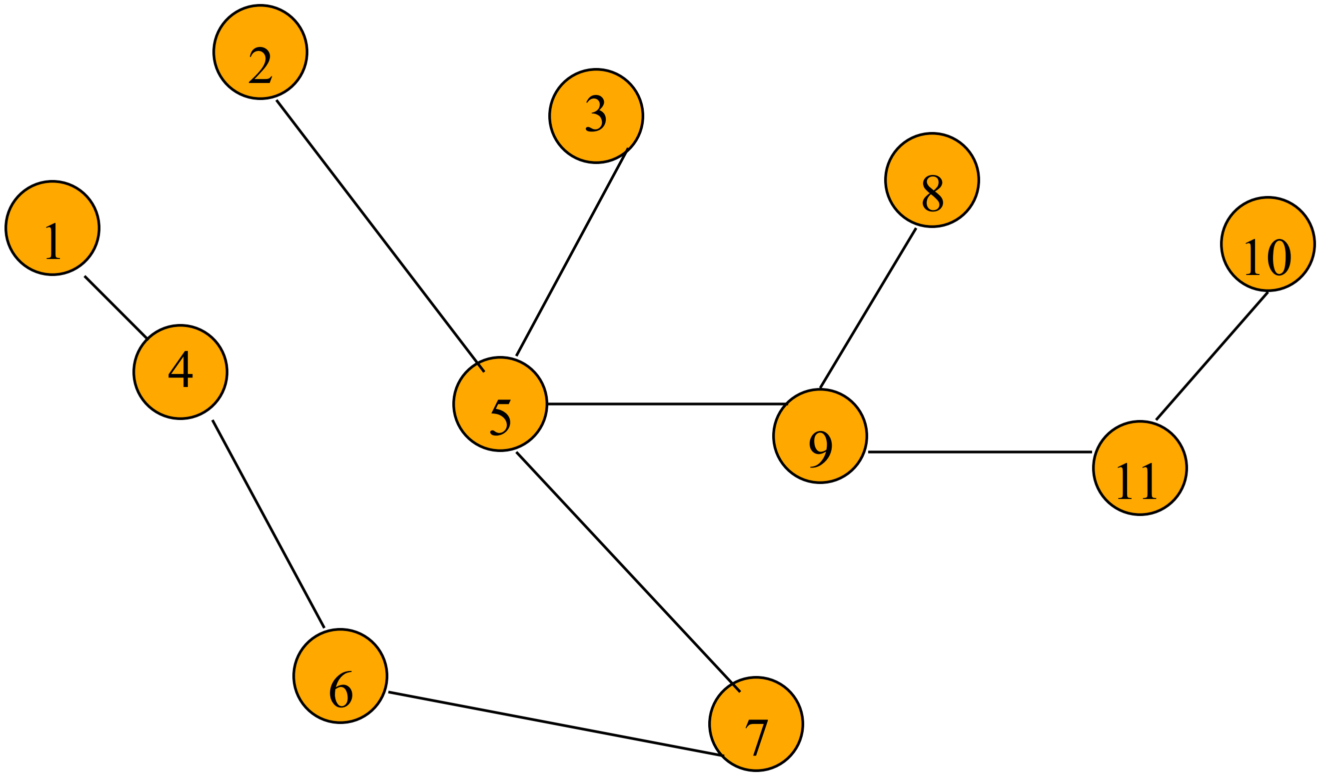
- Is the network connected?
  - Can we communicate between every pair of cities?
- Find the components.
- Want to construct smallest number of feasible links so that resulting network is connected.

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.

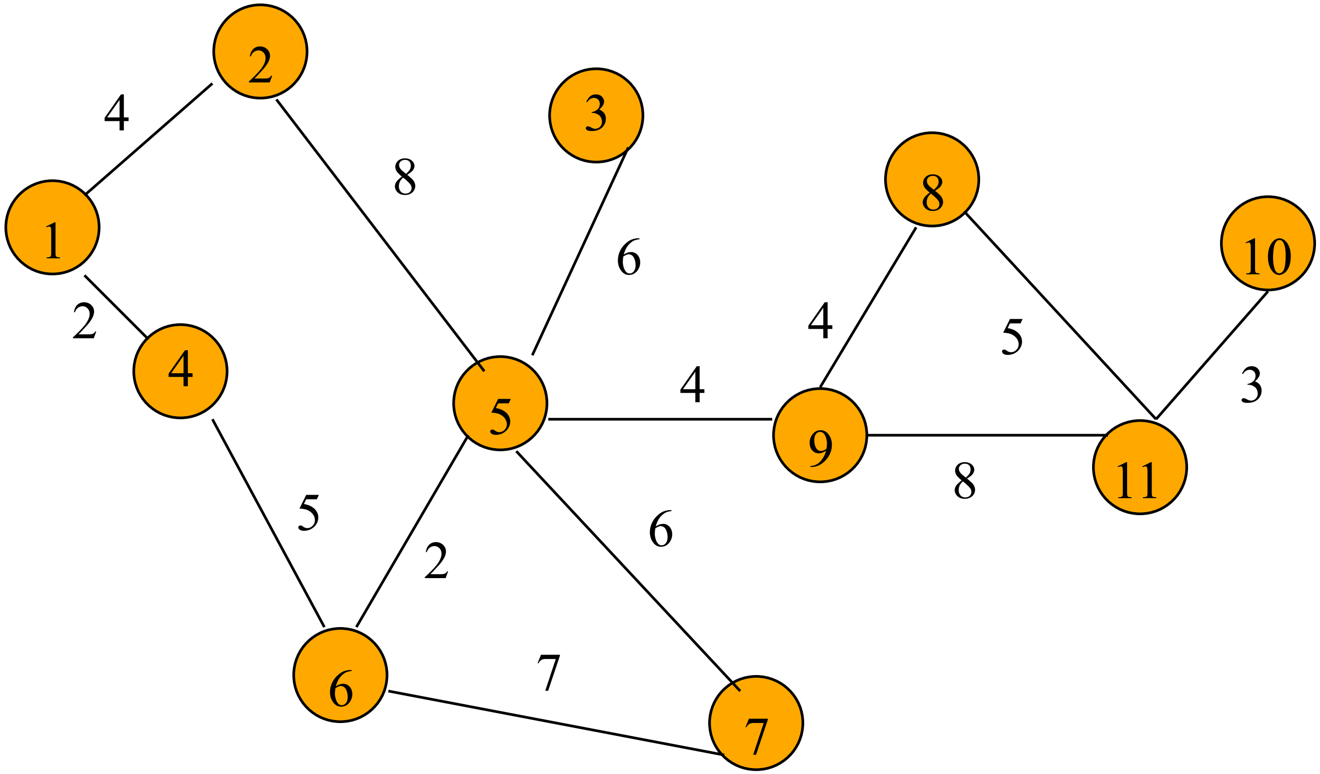
Tree

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

Spanning Tree

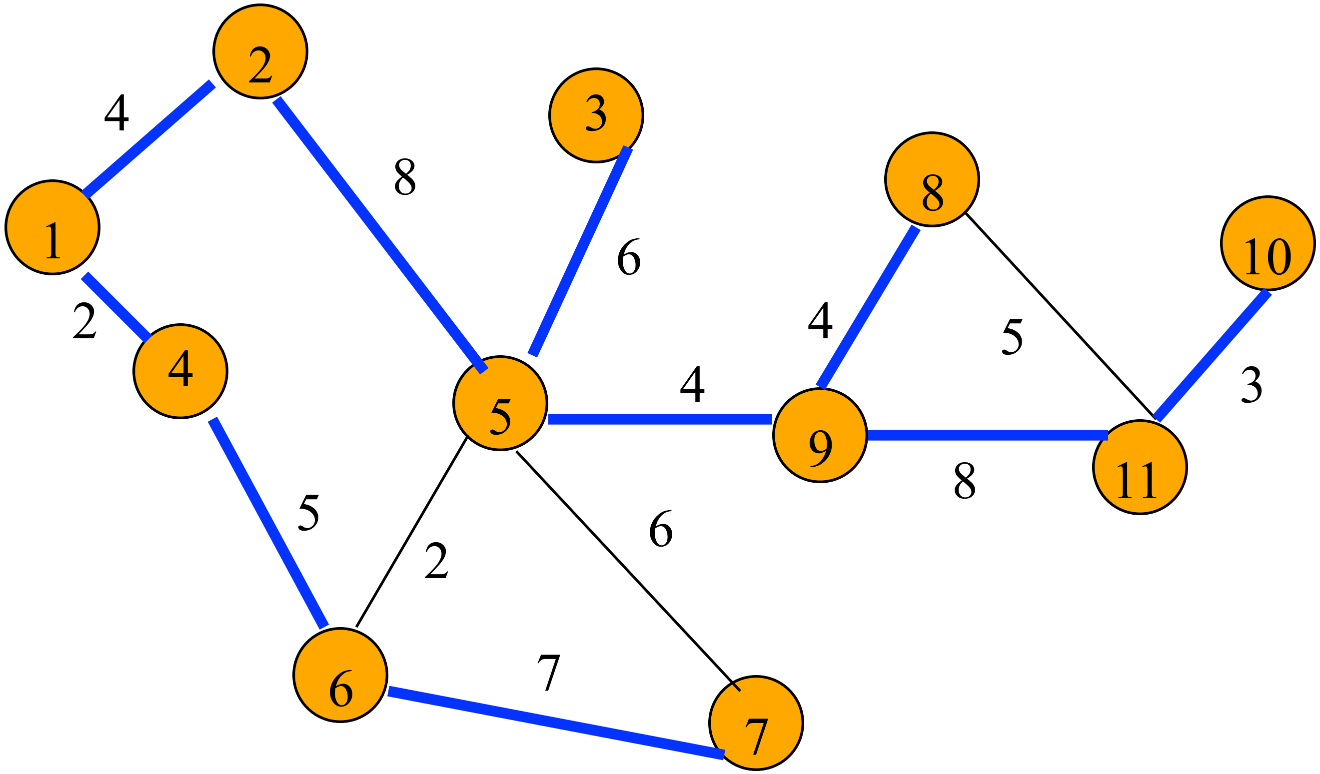
- Subgraph that includes all vertices of the original graph.
- Subgraph is a tree.
  - If original graph has n vertices, the spanning tree has n vertices and n-1 edges.

Minimum Cost Spanning Tree



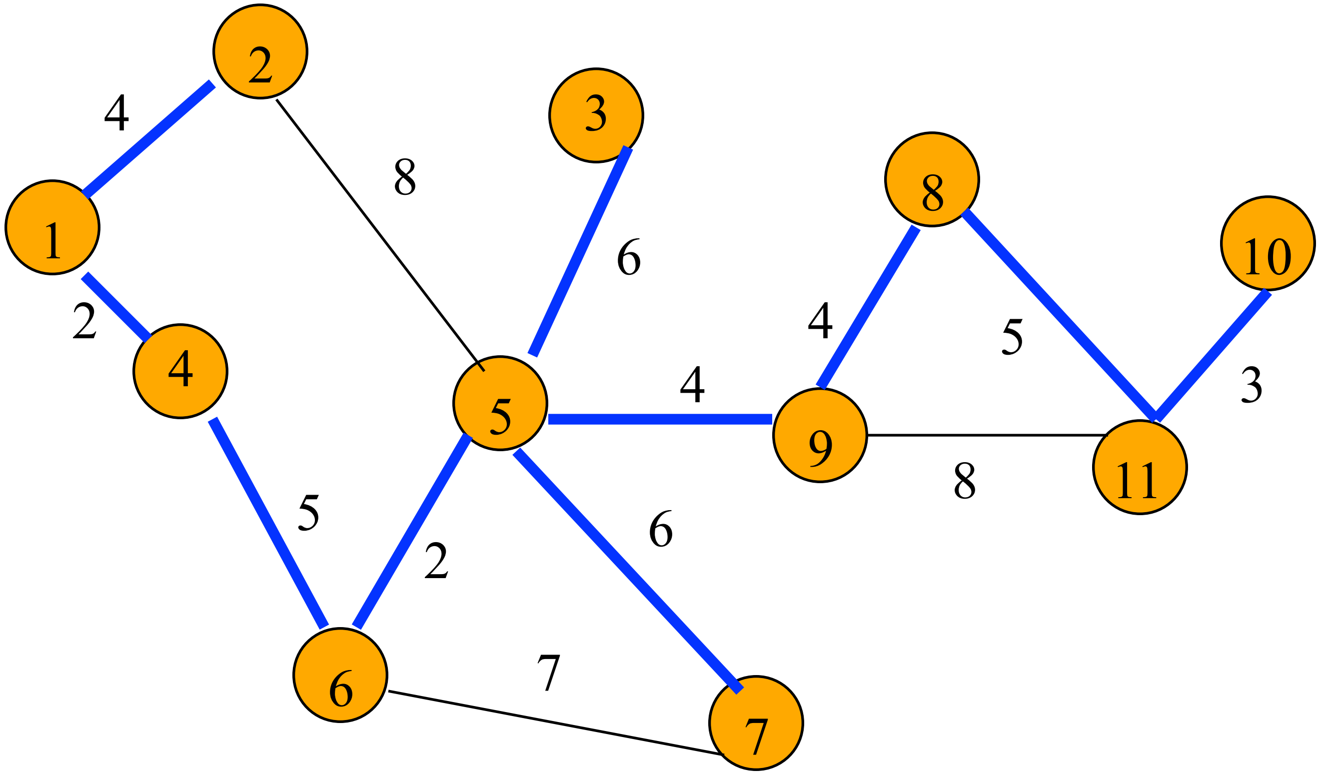
- Tree cost is sum of edge weights/costs.

A Spanning Tree



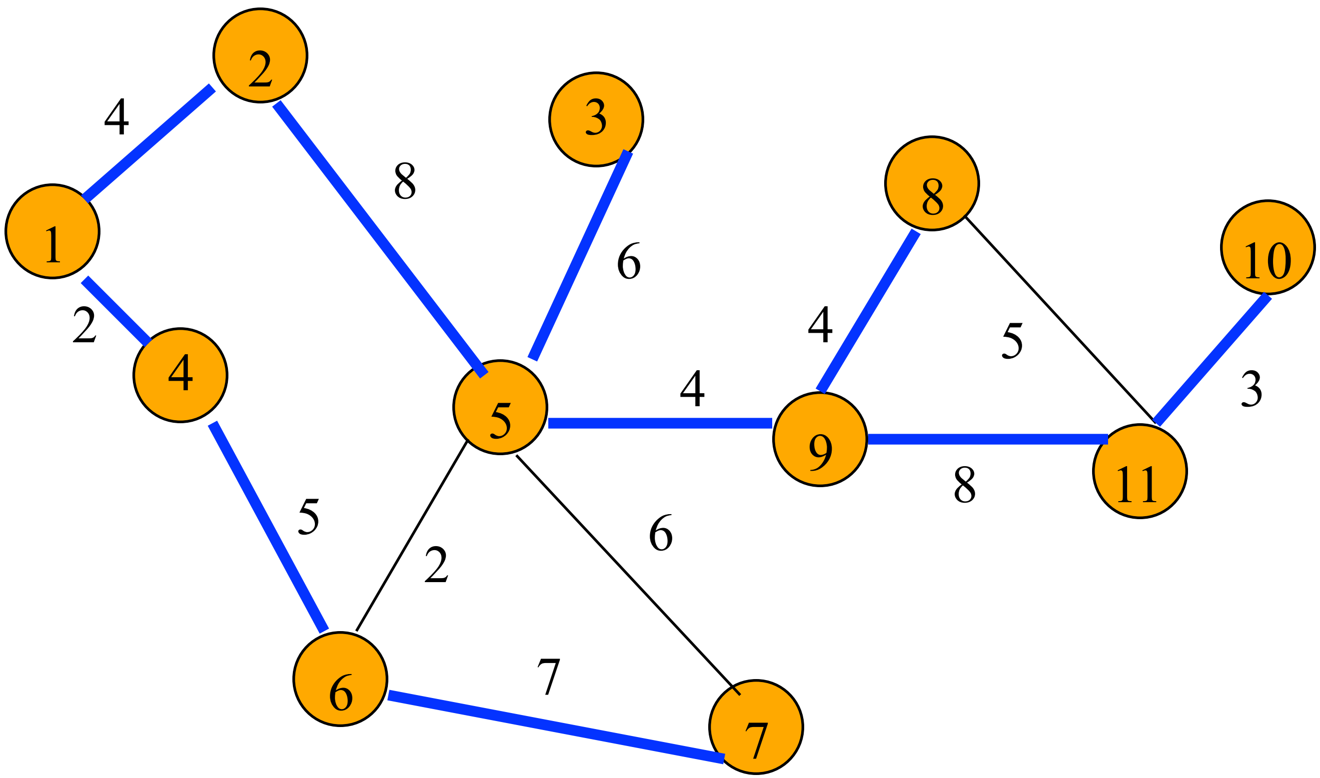
Spanning tree cost = 51.

Minimum Cost Spanning Tree



Spanning tree cost = 41.

A Wireless Broadcast Tree



Source = 1, weights = needed power.

Cost = 4 + 8 + 5 + 6 + 7 + 8 + 3 = 41.

ADT 6.1 Graph

class Graph
{ // A non empty set of vertices and a set of undirected
 // edges, where each edge is a pair of vertices.
public:
  virtual ~Graph()**{ };**
   // virtual destructor
  bool IsEmpty() **const {return** n==0**;};**
   // return **true** iff graph has no vertices
  int NumberOfVertices() **const {return** n**;};**
   // return the number of vertices in the graph
  int NumberofEdges() **const {return** e**;};**
   // return number of edges in the graph
  virtual int Degree**(int** u**) const** =0**;**
   // return number of edges incident to vertex u

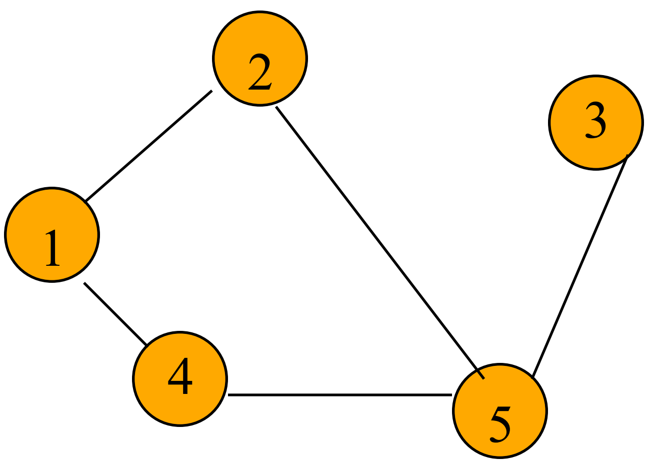
```
    virtual bool ExisteEdge(int u, int v) const =0;
      // return true iff graph has edge (u, v)
    virtual void InsertVertex (int v) =0;
      // insert vertex v into graph, v has no incident edges
    virtual void InsertEdge (int u, int v) =0;
      // insert edge (u, v) into graph
    virtual void DeleteVertex (int v);
      // delete v and all edges incident to it
    virtual void DeleteEdge (int u, int v) =0;
      // delete edge (u, v) from the graph
private:
     int n;    // number of vertices
     int e;    // number of edges
};
```

Graph Representation

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

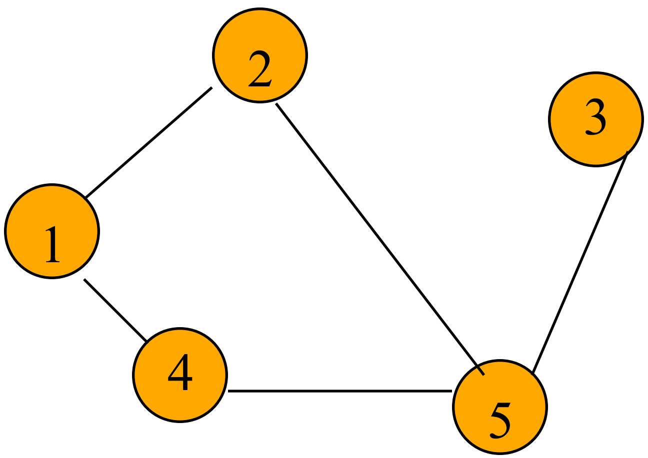
Adjacency Matrix

- 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

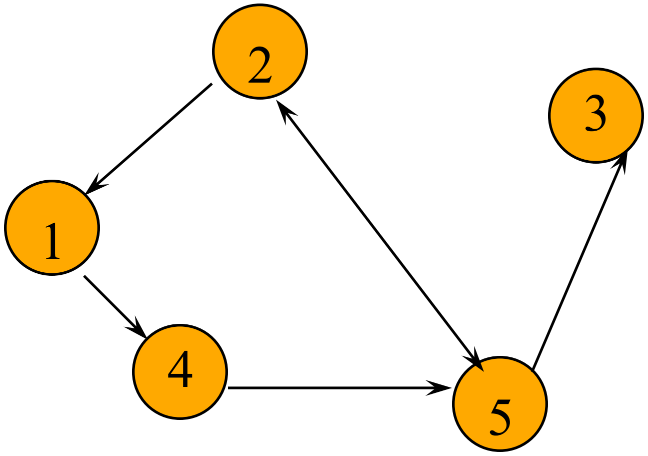
Adjacency Matrix Properties



	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

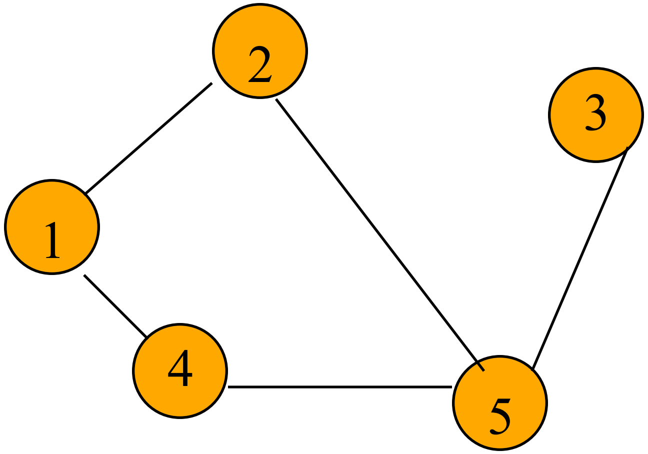
- n^2 bits of space
- For an undirected graph, may store only lower or upper triangle (exclude diagonal).
  - $(n-1)n/2$ bits
- time to find vertex degree and/or vertices adjacent to a given vertex?
  - $O(n)$

Adjacency Matrix

- For an graph
- $$d(i) = \sum_{j=0}^{n-1} a[i][j]$$
- For a digraph
- $$\text{out-d}(i) = \sum_{j=0}^{n-1} a[i][j]$$
- $$\text{in-d}(j) = \sum_{i=0}^{n-1} a[i][j]$$

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)

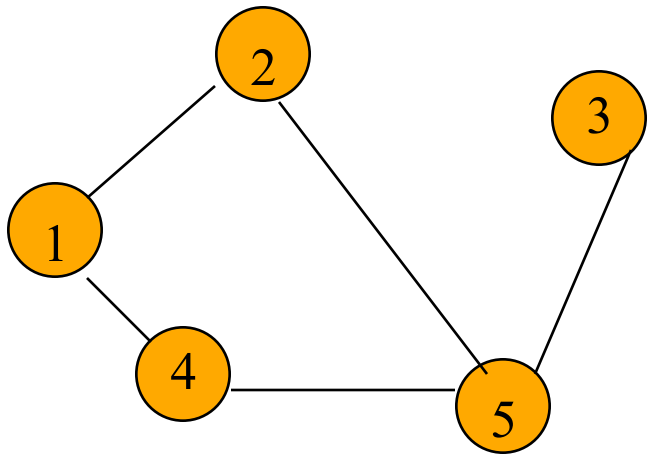
aList[3] = (5)

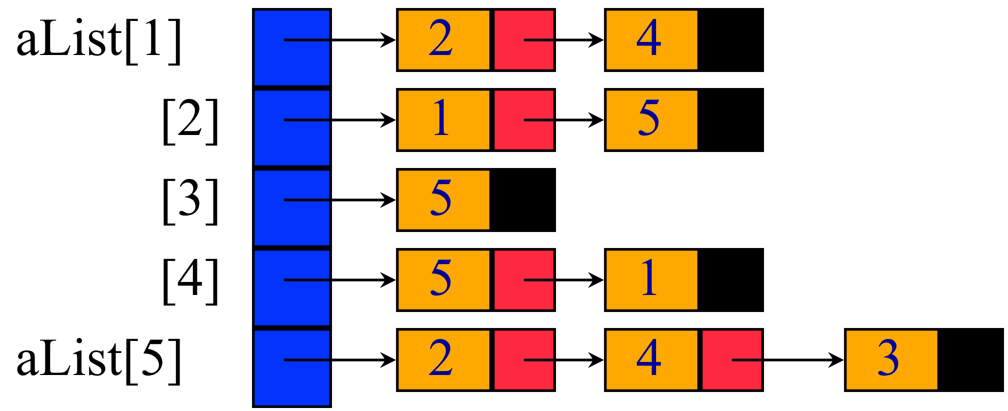
aList[4] = (5,1)

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

Linked Adjacency Lists

- Each adjacency list is a chain.





Array Length = n

# of chain nodes = 2e (undirected graph)

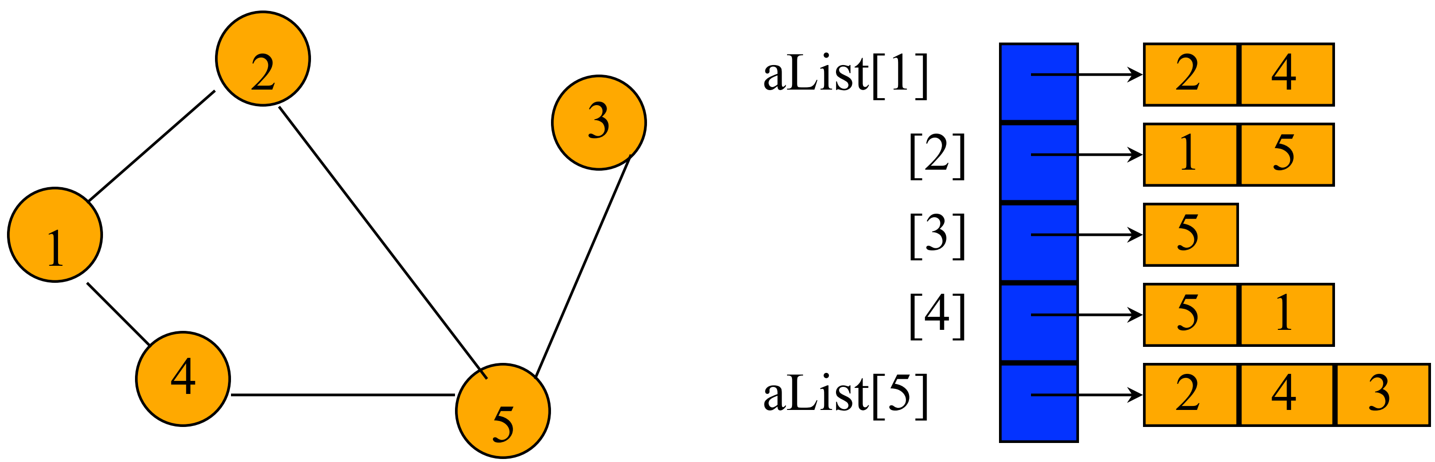
# of chain nodes = e (digraph)

Linked Adjacency Lists

- **class** LinkedGraph {
- **public:**
- LinkedGraph (**const int** vertices): e(0) {
- **if** (vertices < 1) **throw** “Number of vertices must be > 0”**;**
- n = vertices**;**
- adjLists = **new** Chain<**int**>[n]**;**
- }**;**
- **private:**
- Chain<**int**>* adjLists**;**
- **int** n**;**
- **int** e**;**
- }**;**

Array Adjacency Lists

- Each adjacency list is an array list.



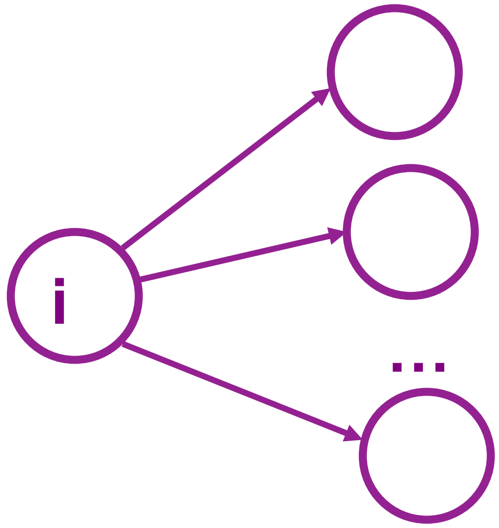
Array Length = n

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

\# of list elements = e (digraph)

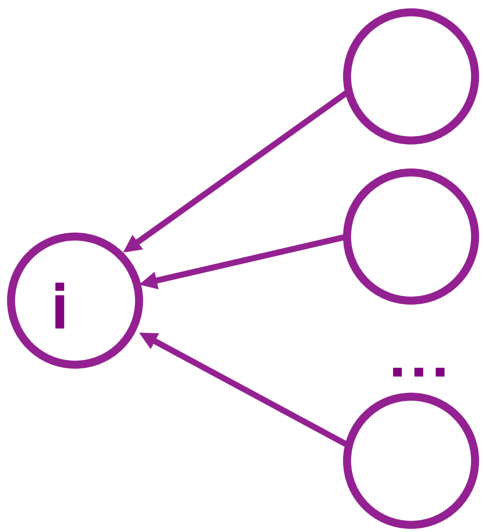
Adjacency Lists

- Digraph

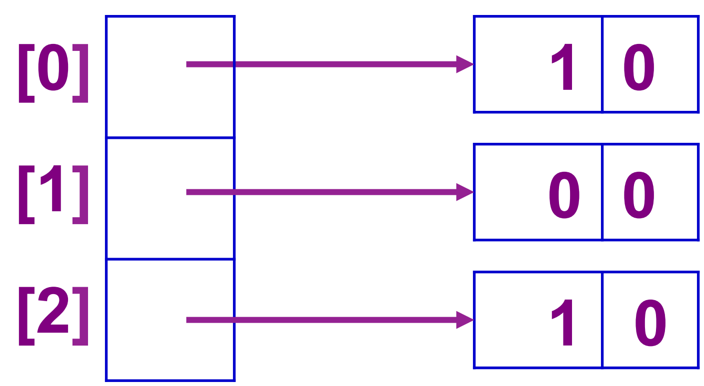


Inverse Adjacency Lists

- Digraph



$$\begin{array}{c|ccc|}
 & 0 & 1 & 2 \\
\hline
0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
2 & 0 & 0 & 0 \\
\hline
\end{array}$$

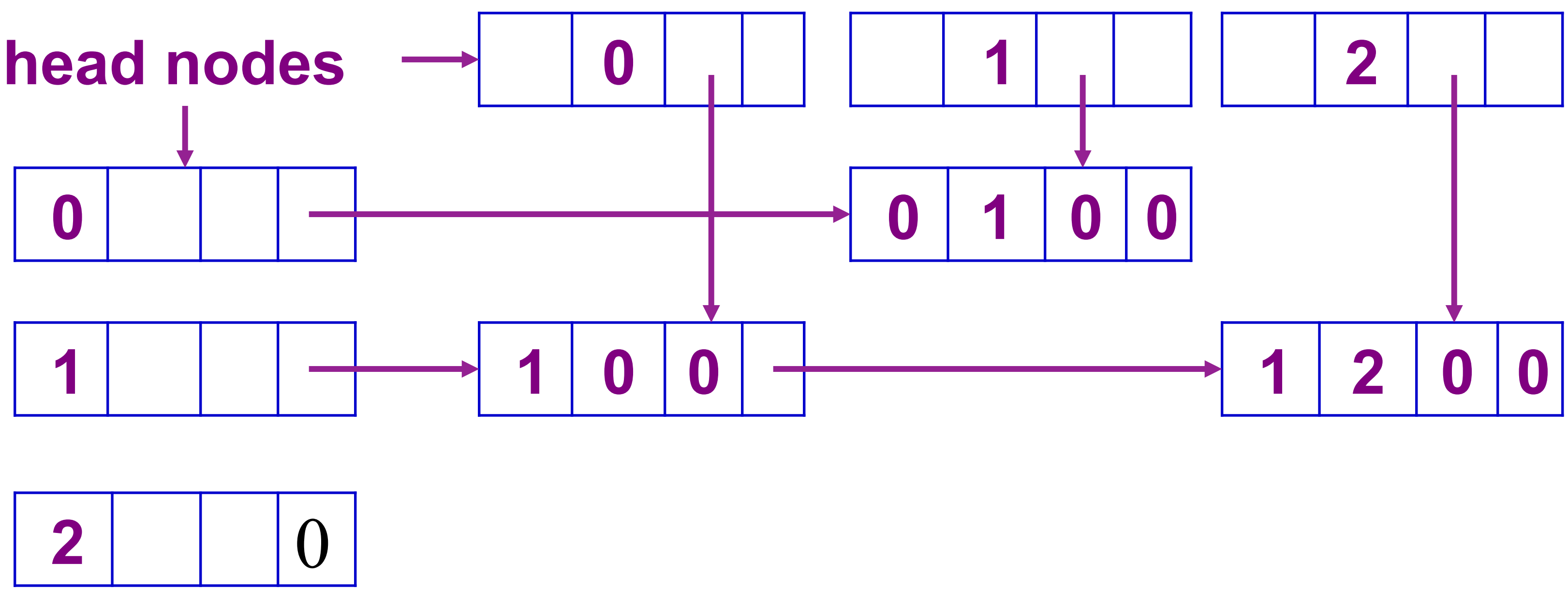


Orthogonal Adjacency Lists

- Digraph

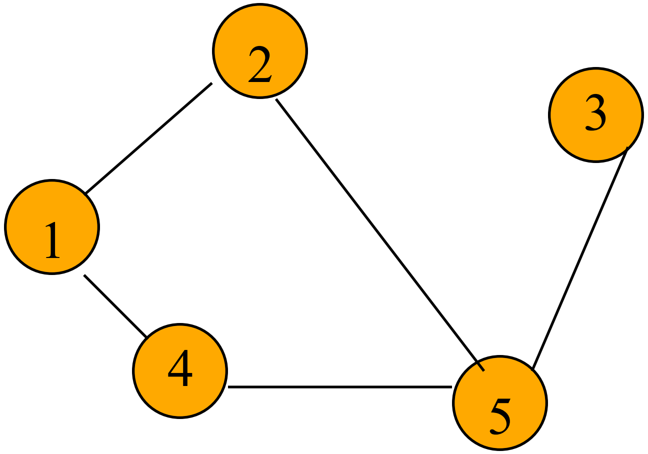
tail	head	column link	row link

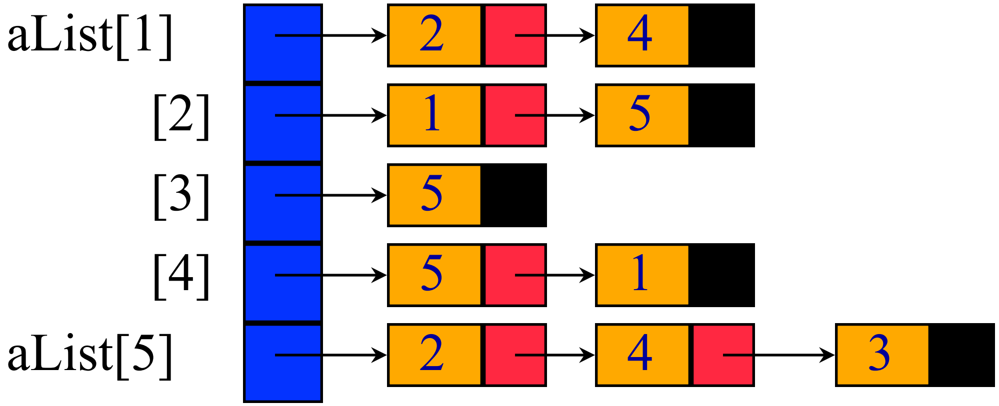
$$
\begin{array}{c|ccc|}
 & 0 & 1 & 2 \\
\hline
0 & 0 & 1 & 0 \\
1 & 1 & 0 & 1 \\
2 & 0 & 0 & 0 \\
\hline
\end{array}
$$



Adjacency Multilists

- Undirected graph





Each (u, v) is represented by 2 entries.

Visit an edge only once?

m	vertex1	vertex2	v1link	v2link
			path1	path2

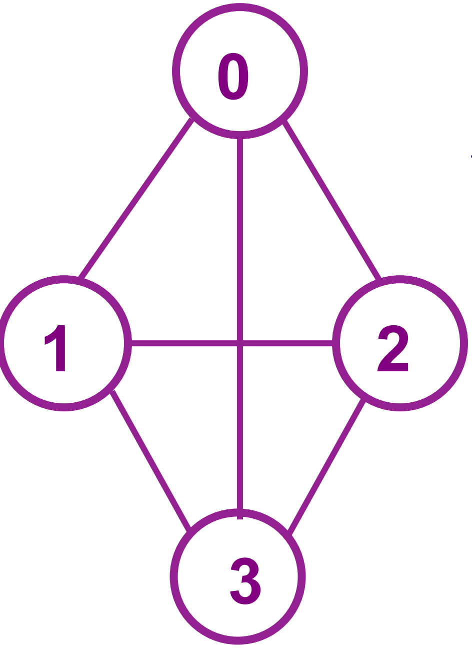
Adjacency Multilists

- **class** MGraphEdge {
- **private:**
-     **bool** m;
-     **int** vertex1, vertex2;
-     MGraphEdge *path1, *path2;
- };
- **typedef** MGraphEdge *EdgePtr ;
- **class** MGraph {
- **public:**
-     MGraph(**const int**);
- **private:**
-     EdgePtr *adjMultiLists;   **int** n;   **int** e;
- };

Adjacency Multilists

- MGraph::MGraph(**const int** vertices) : e(0)
- {
- **if** (vertices < 1) **throw** “Number of vertices must be > 0”**;**
- n = vertices**;**
- adjMultiLists = **new** EdgePtr[n]**;**
- fill(adjMultiLists, adjMultiLists+n,0)**;**
- }

Adjacency Multilists



N0		0	1	N1	N3	edge (0,1)
N1		0	2	N2	N3	edge (0,2)
N2		0	3	0	N4	edge (0,3)
N3		1	2	N4	N5	edge (1,2)
N4		1	3	0	N5	edge (1,3)
N5		2	3	0	0	edge (2,3)

[0]
[1]
[2]
[3]

adjMultiLists

Adjacency Multilists

- **If p points to an MGraphEdge representing (u, v), and given u, to get v we need the following test:**
  - **if** (p→vertex1 == u) v = p→vertex2;
  - **else** v = p→vertex1;
- **And we can insert an edge in O(1):**
- **void** MGraph::InsertEdge(**int** u, **int** v) {
- MGraphEdge *p = **new** MGraphEdge;
- p→m = **false;** p→vertex1 = u; p→vertex2 = v;
- p→path1 = adjMultiLists[u];
- p→path2 = adjMultiLists[v];
- adjMultiLists[u] = adhMultiLists[v] = p;
- }

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 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 classes

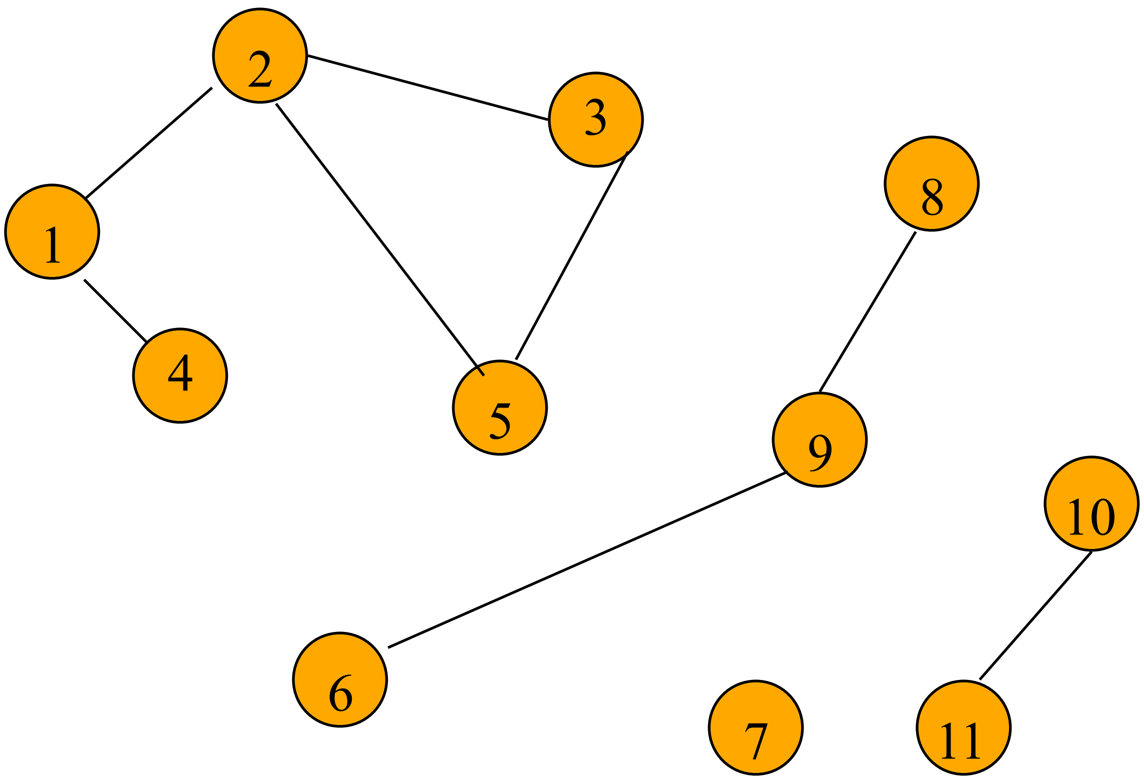
- **Exercises: P340-5, 9**

Graph Search Methods

- Given $G = (V, E)$, and v in $V(G)$, we wish to visit all vertices in G that are reachable from v.
- In the following methods, we assume the graphs are undirected, although they work on the directed as well.

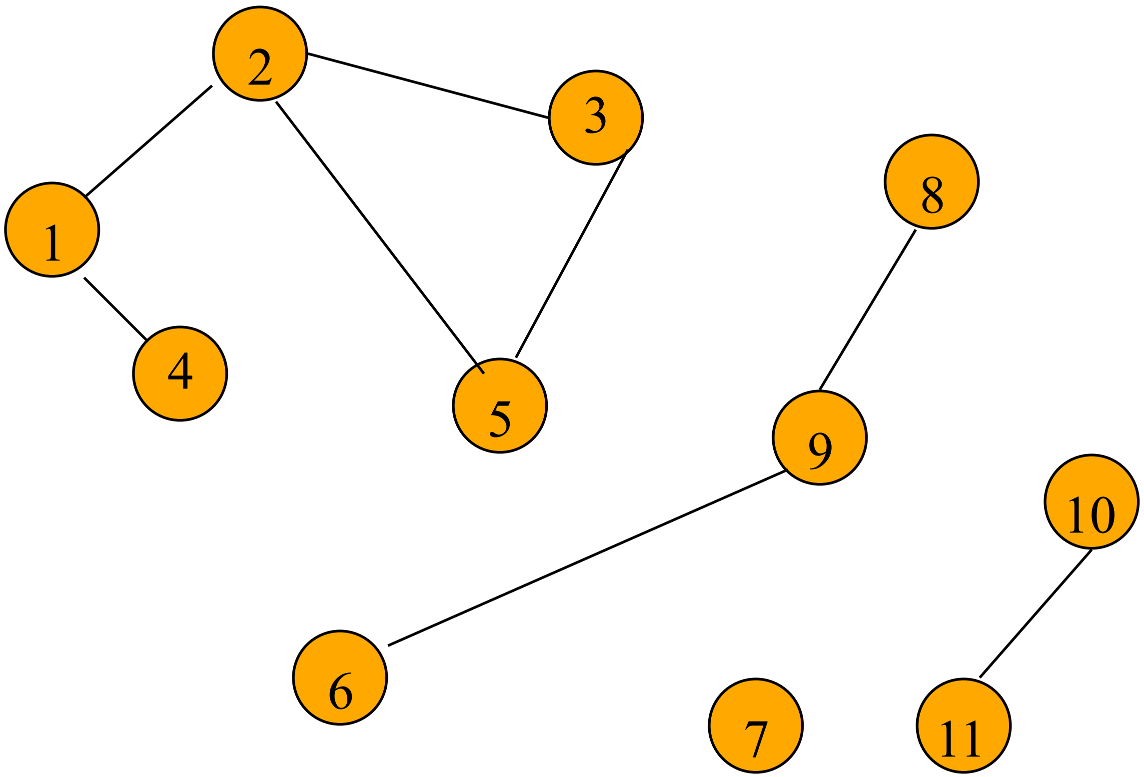
Graph Search Methods

- 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.



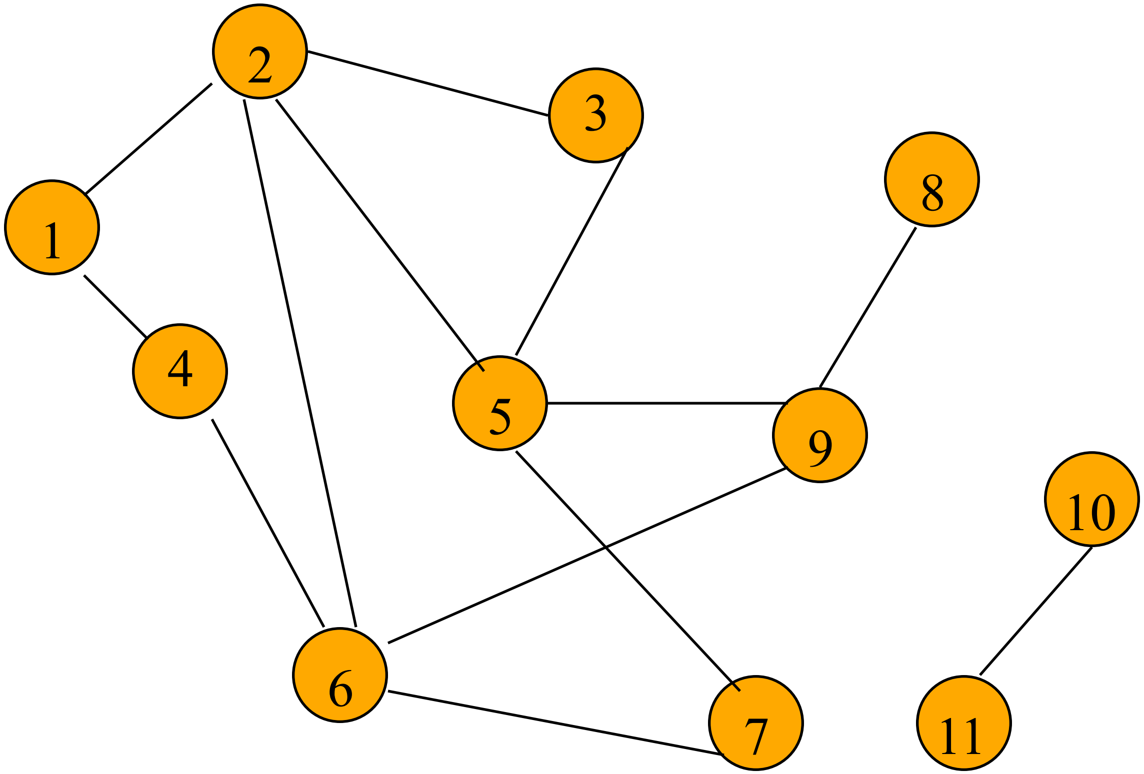
Graph Search Methods

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

Breadth-First Search

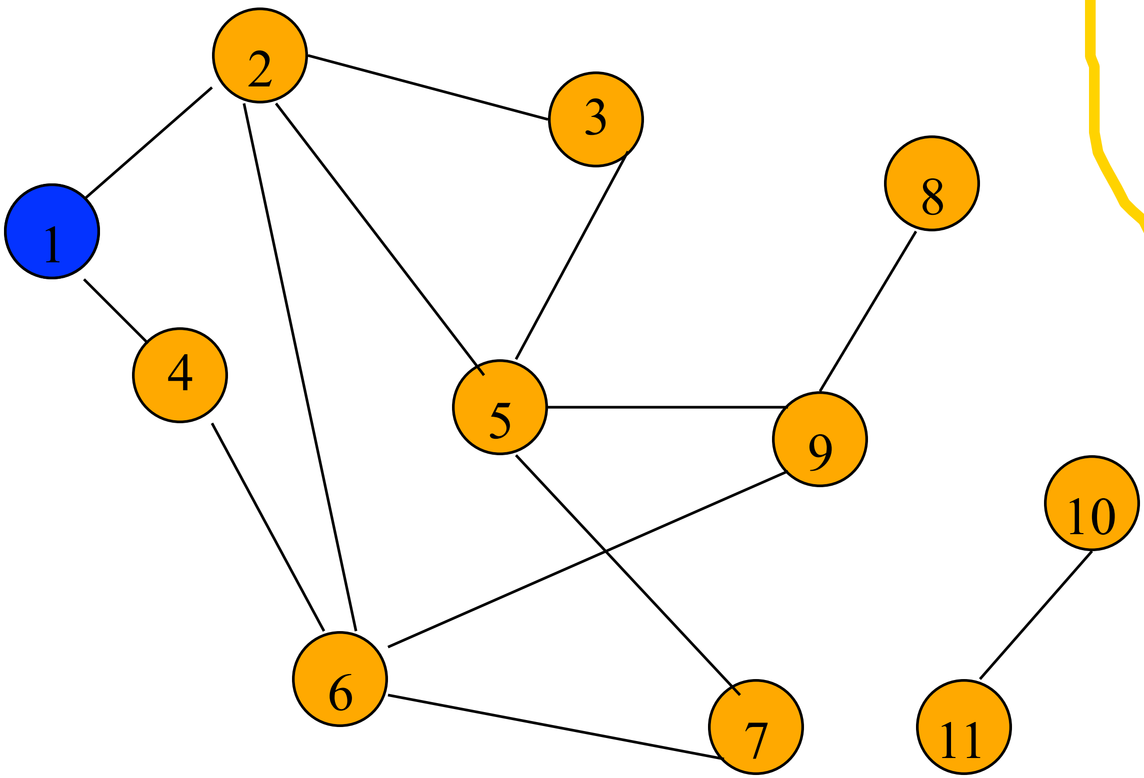
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Breadth-First Search Example



Start search at vertex 1.

Breadth-First Search Example

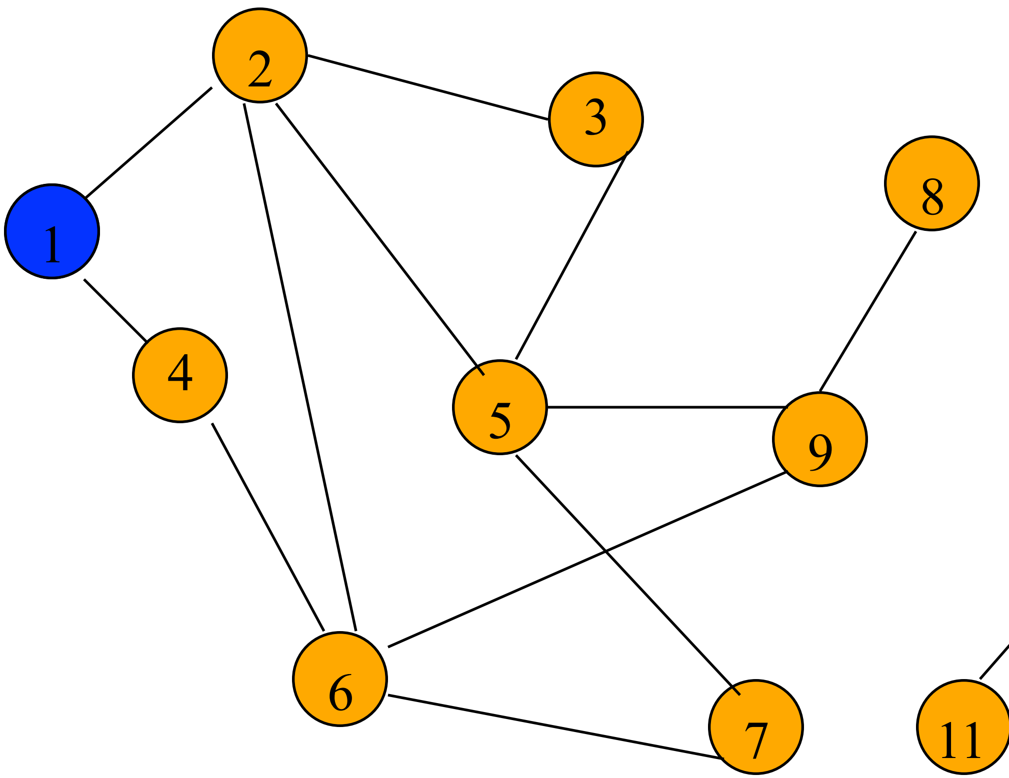


FIFO Queue

1

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

Breadth-First Search Example

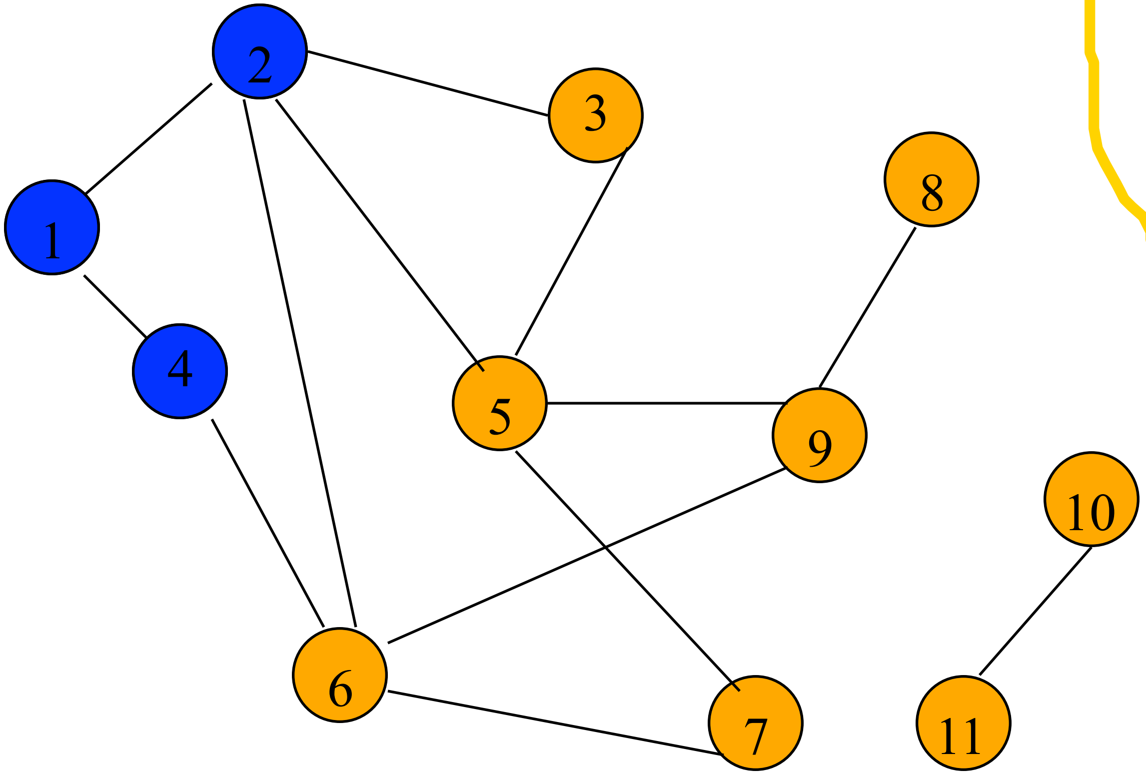


FIFO Queue

1

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

Breadth-First Search Example

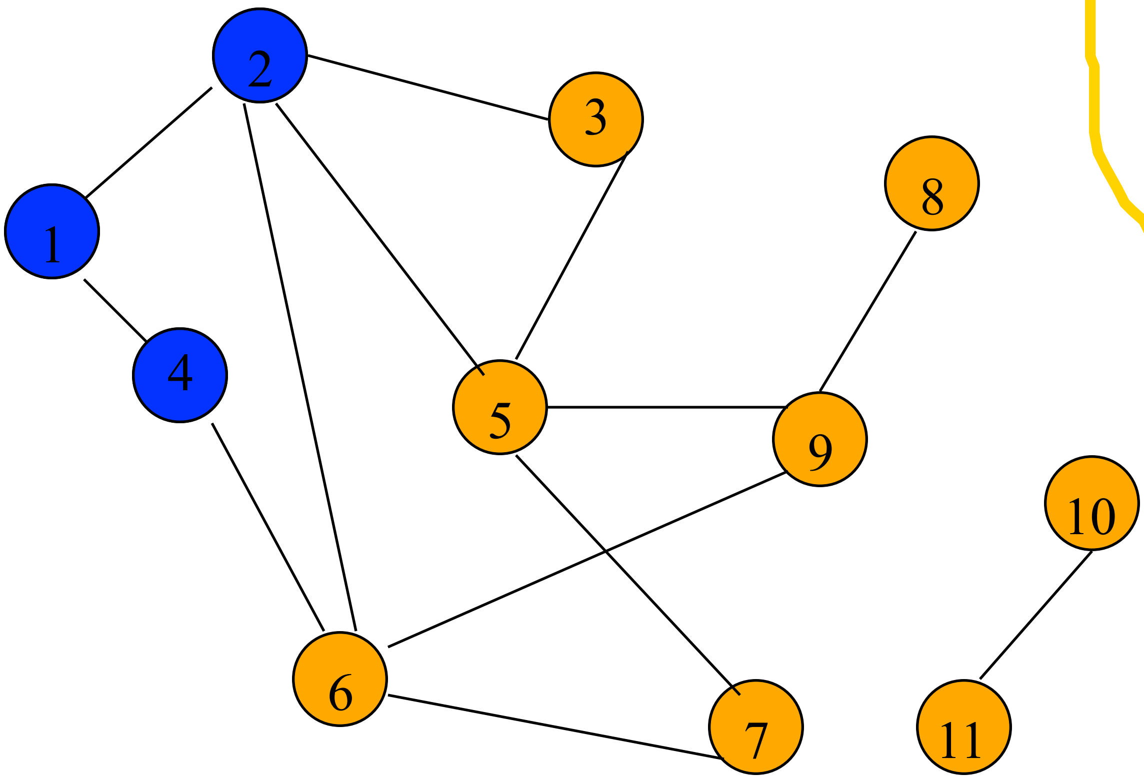


FIFO Queue

2 4

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

Breadth-First Search Example

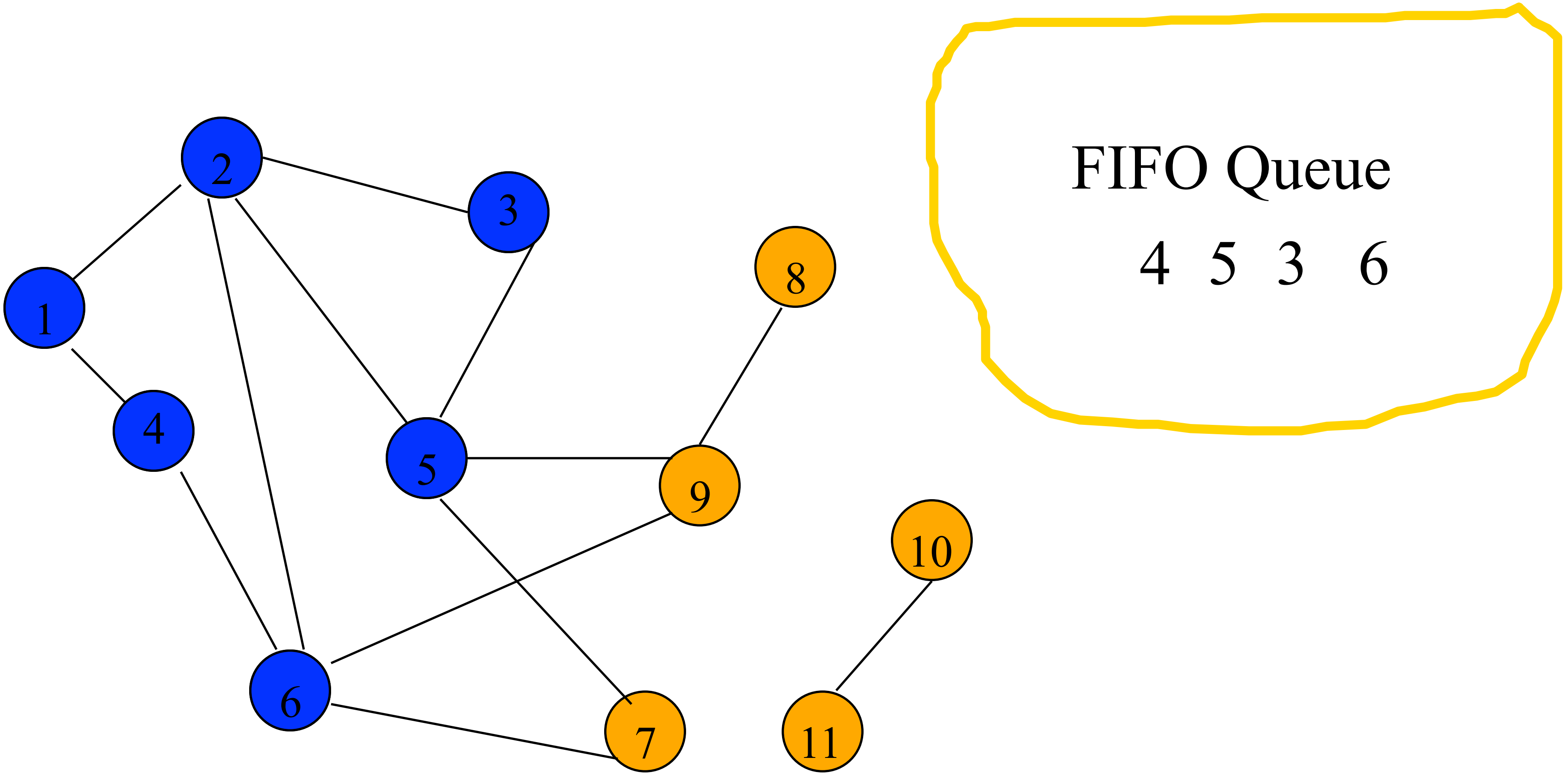


FIFO Queue

2 4

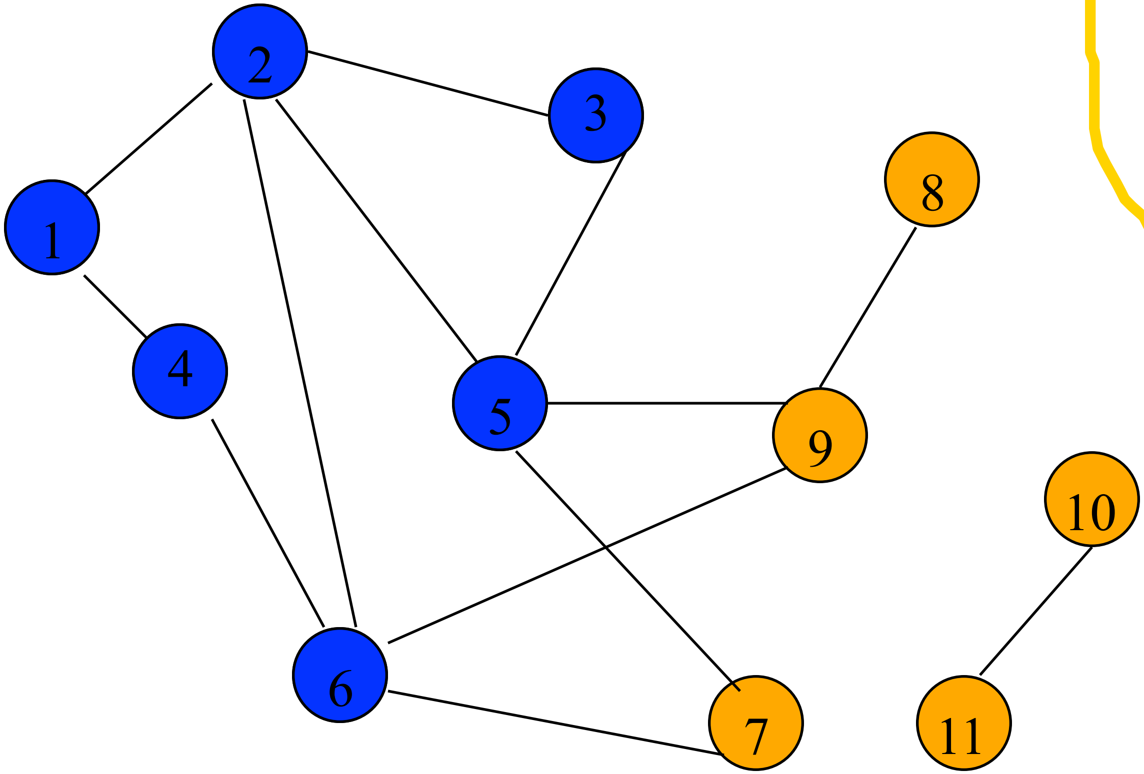
Remove 2 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



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

Breadth-First Search Example

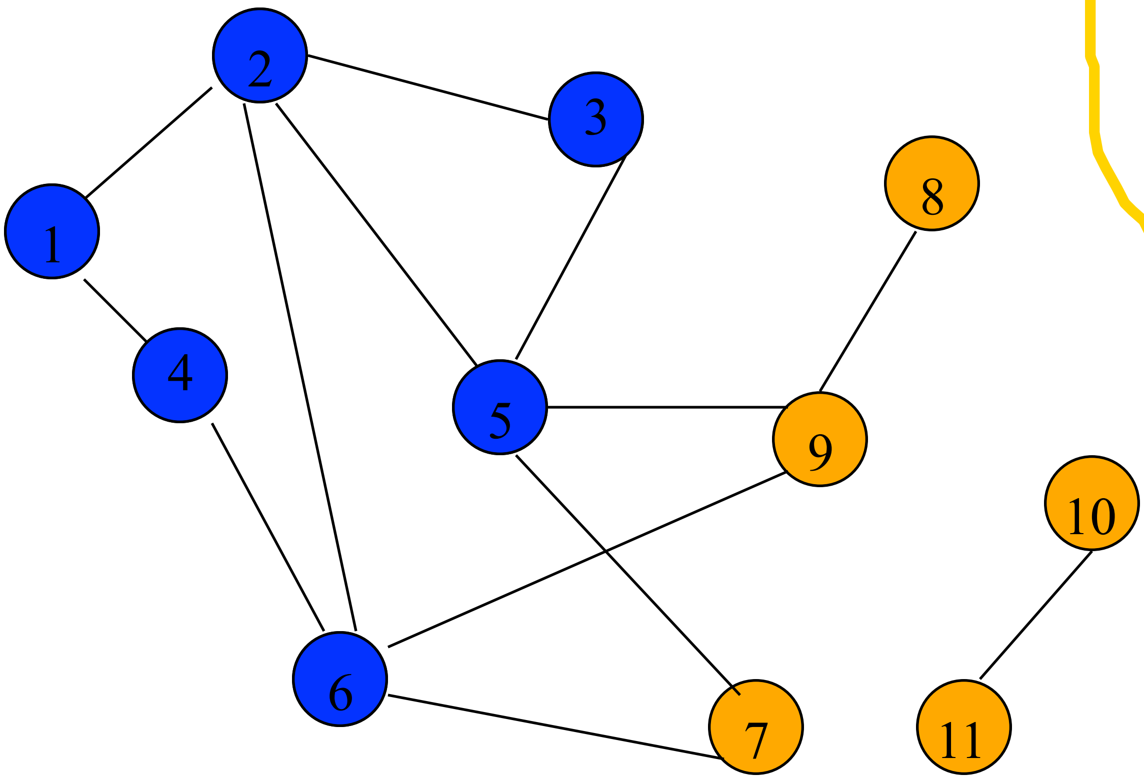


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

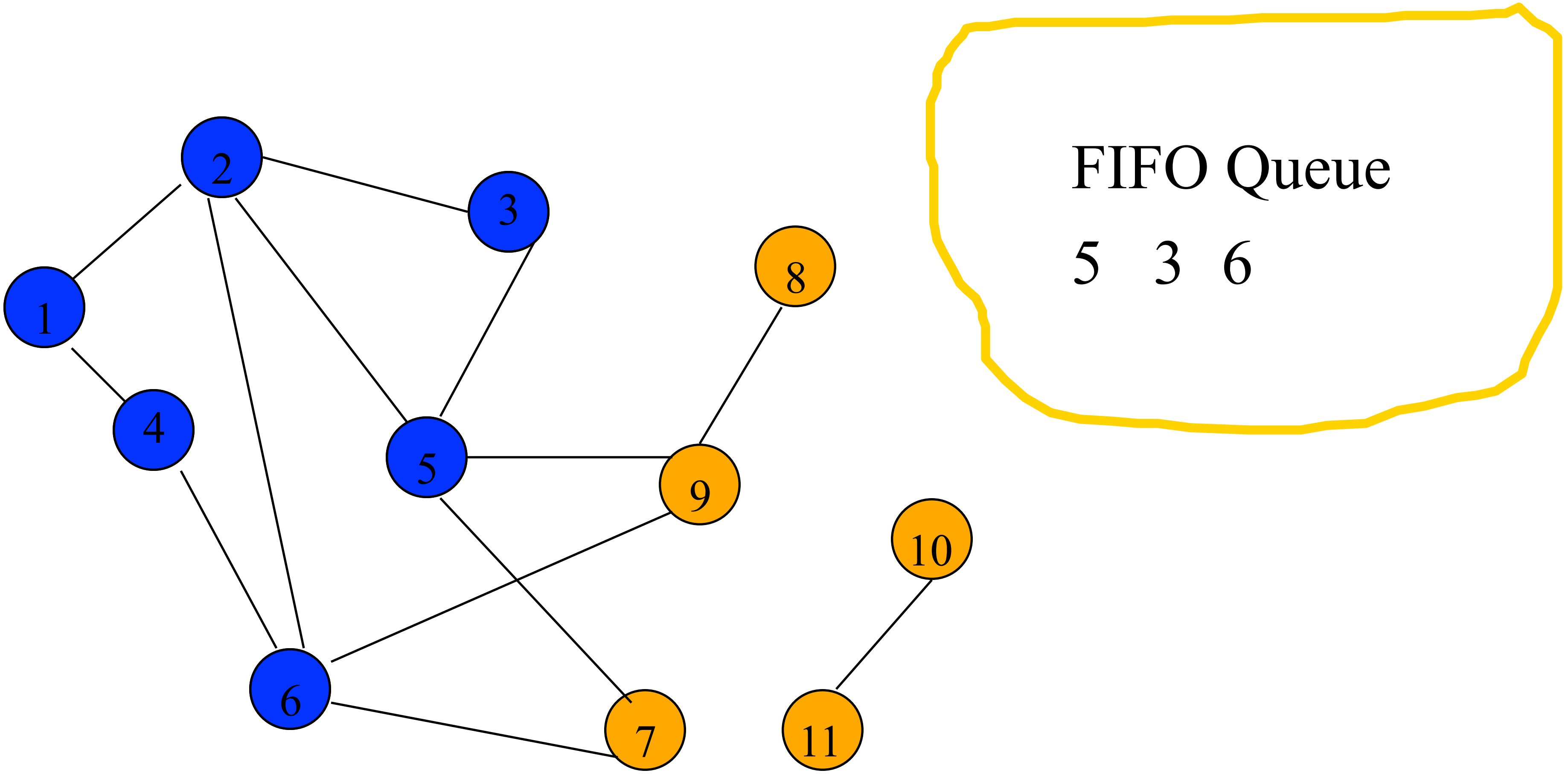


FIFO Queue

5 3 6

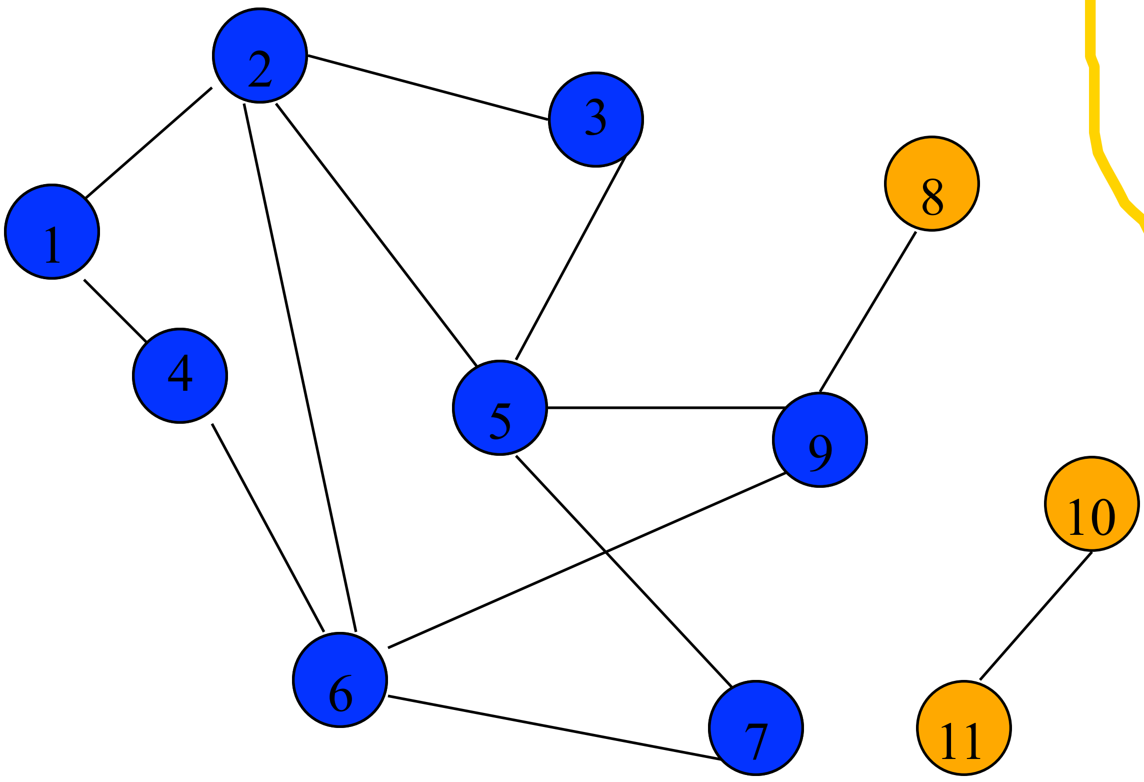
Remove 4 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



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

Breadth-First Search Example

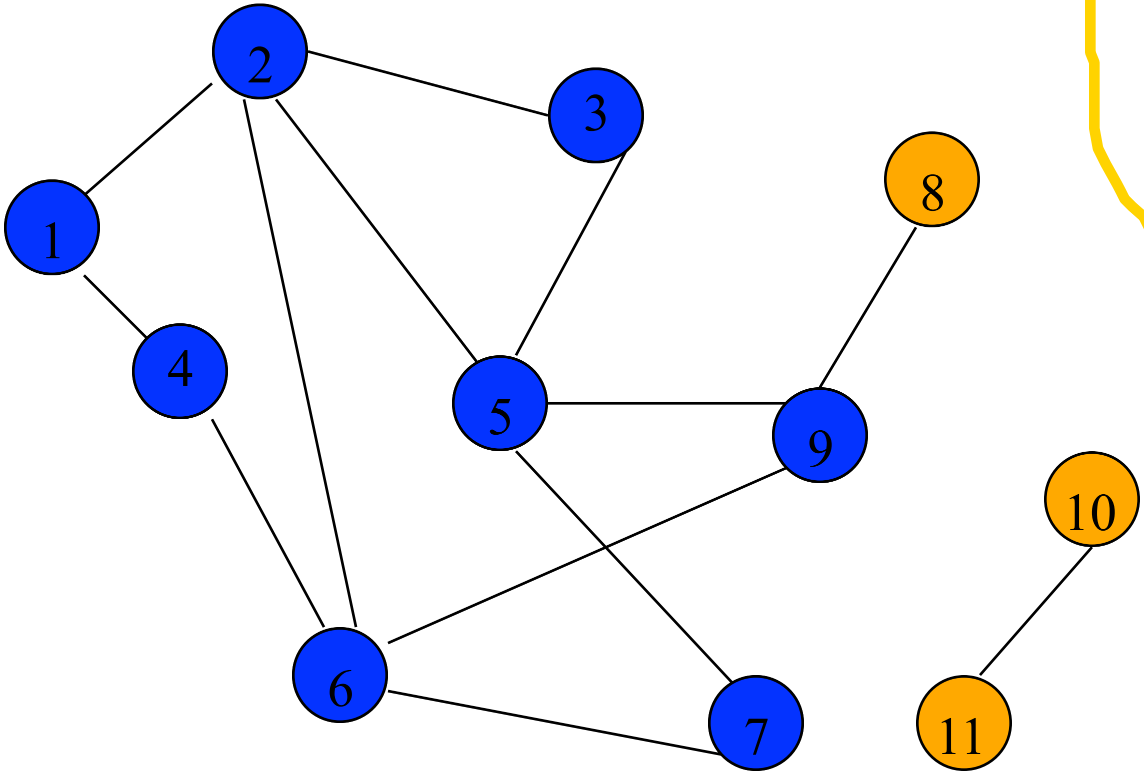


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

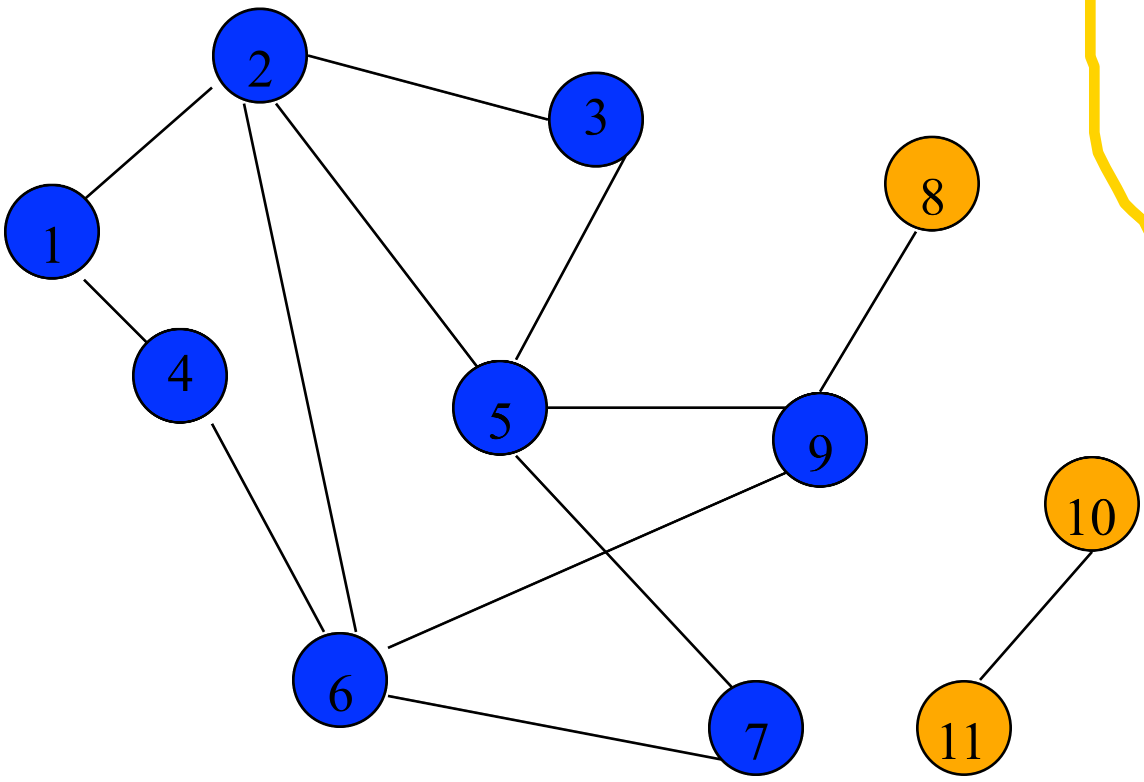


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

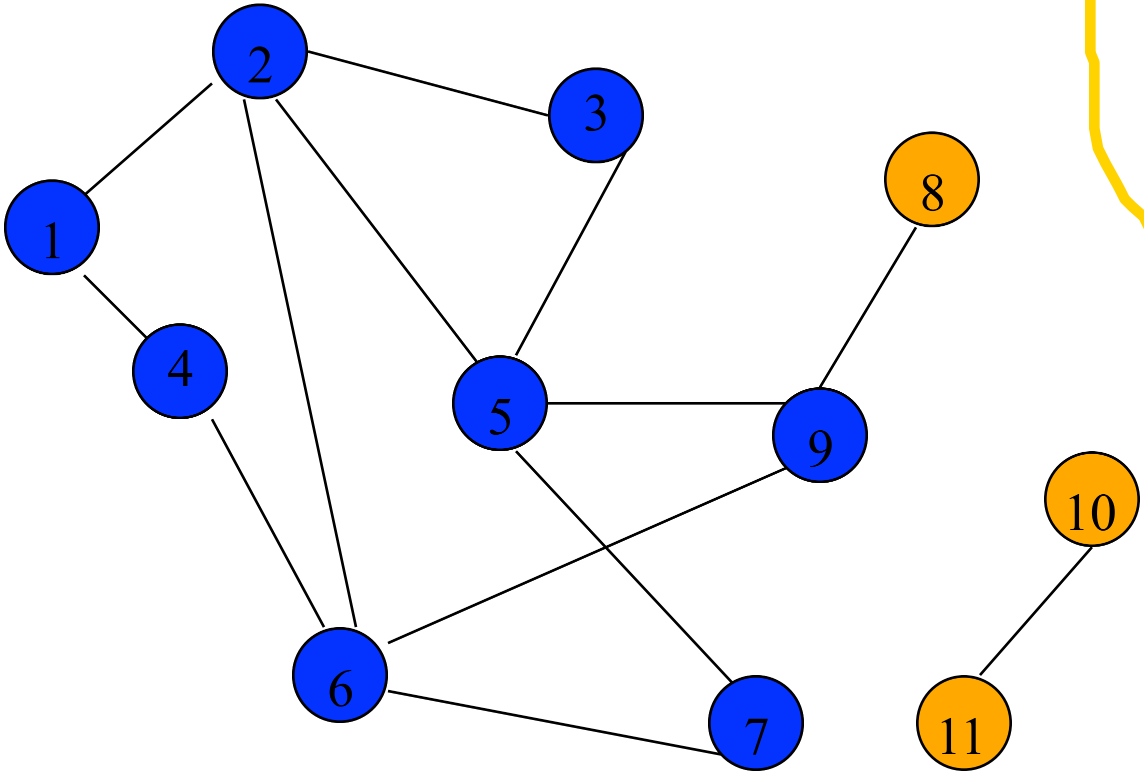


FIFO Queue

6 9 7

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

Breadth-First Search Example

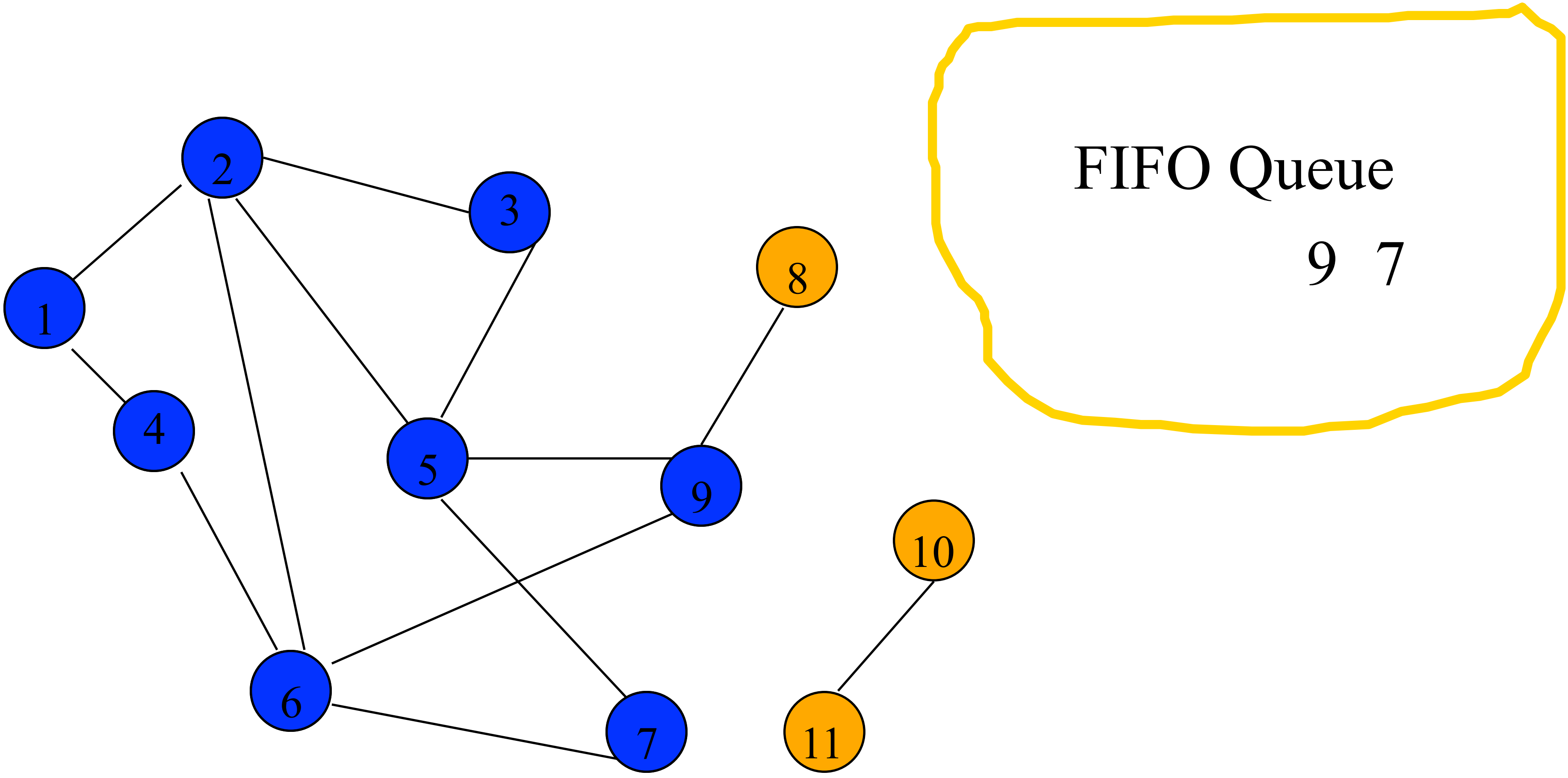


FIFO Queue

6 9 7

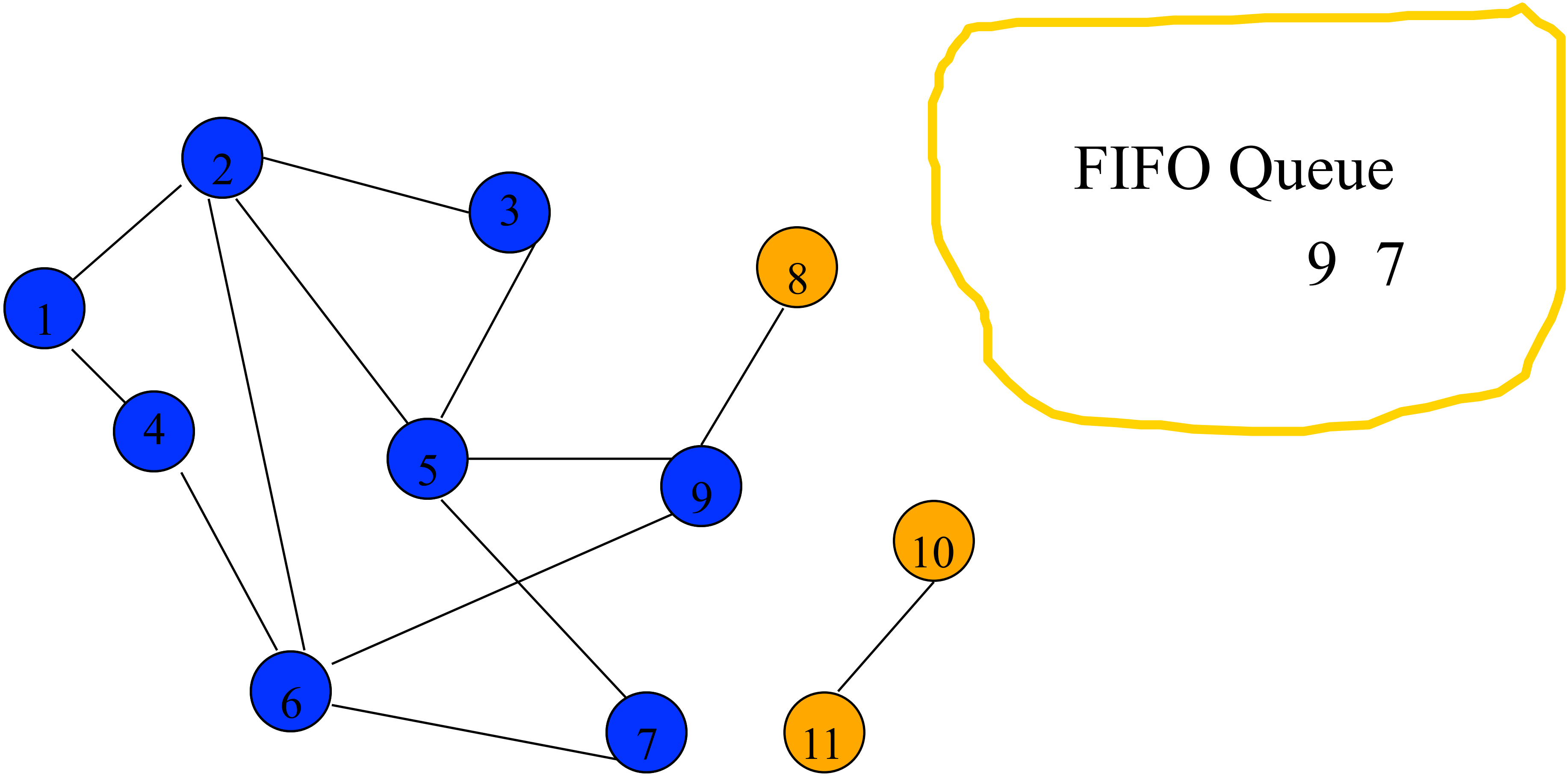
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



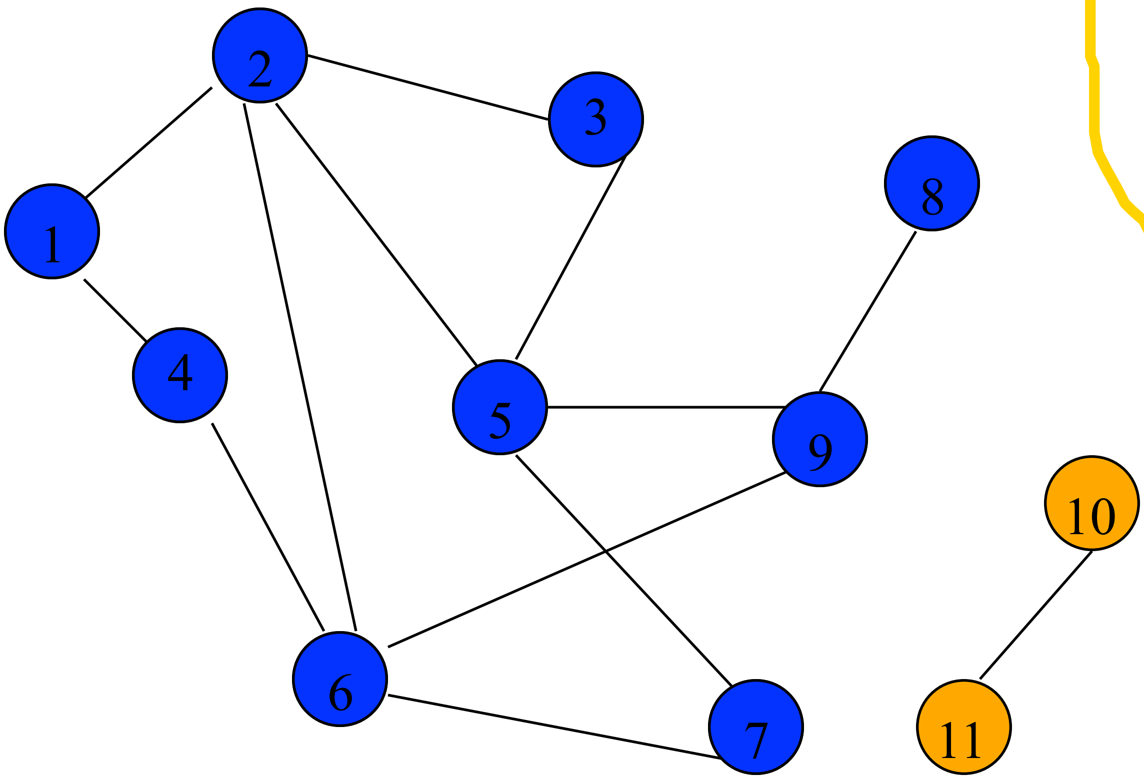
Remove 6 from Q; visit adjacent unvisited vertices; put in Q.

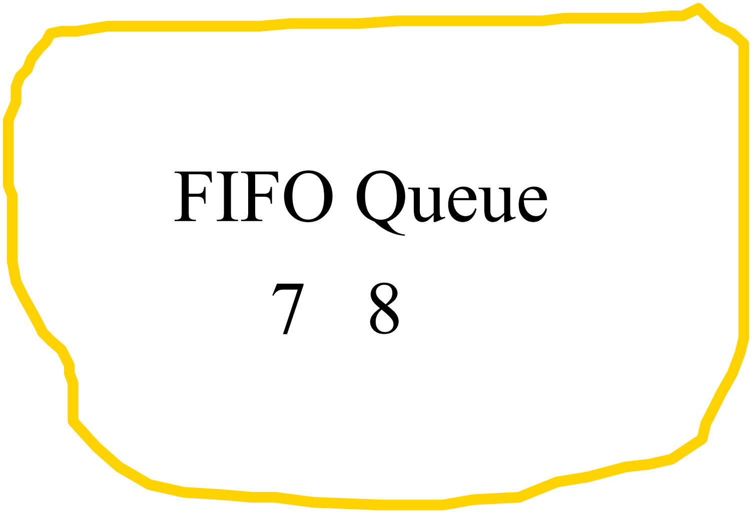
Breadth-First Search Example



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

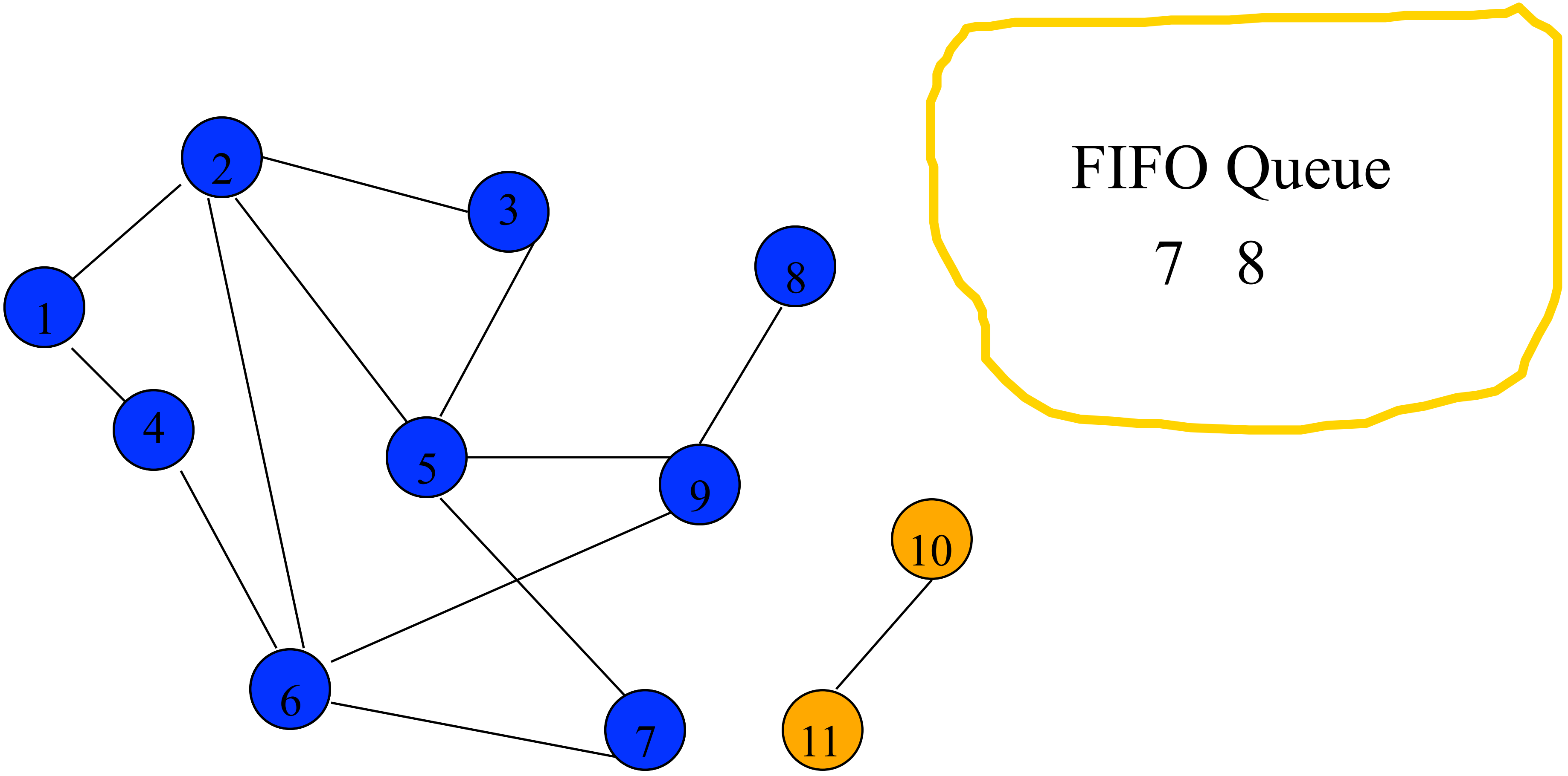
Breadth-First Search Example





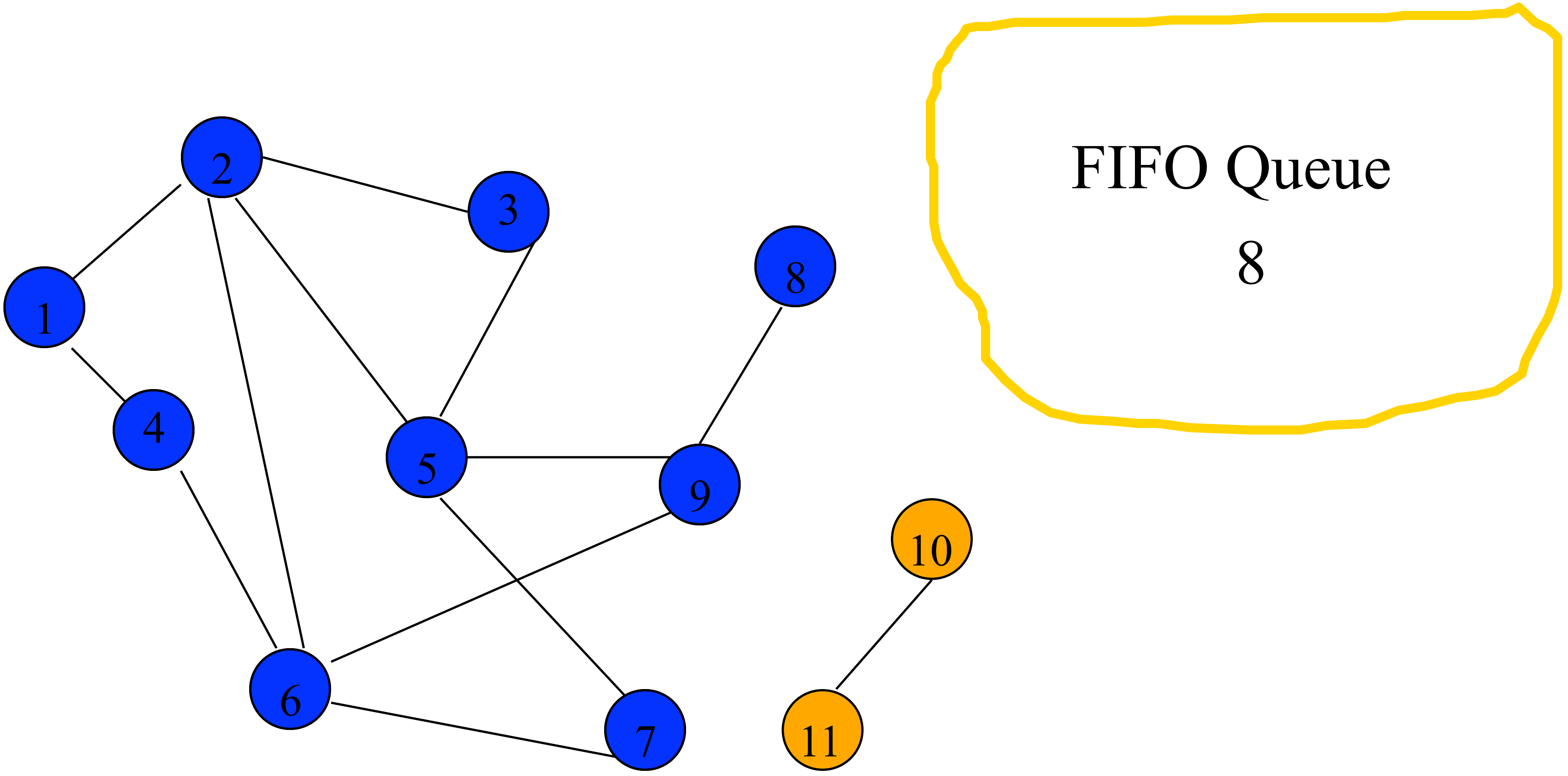
Remove 9 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



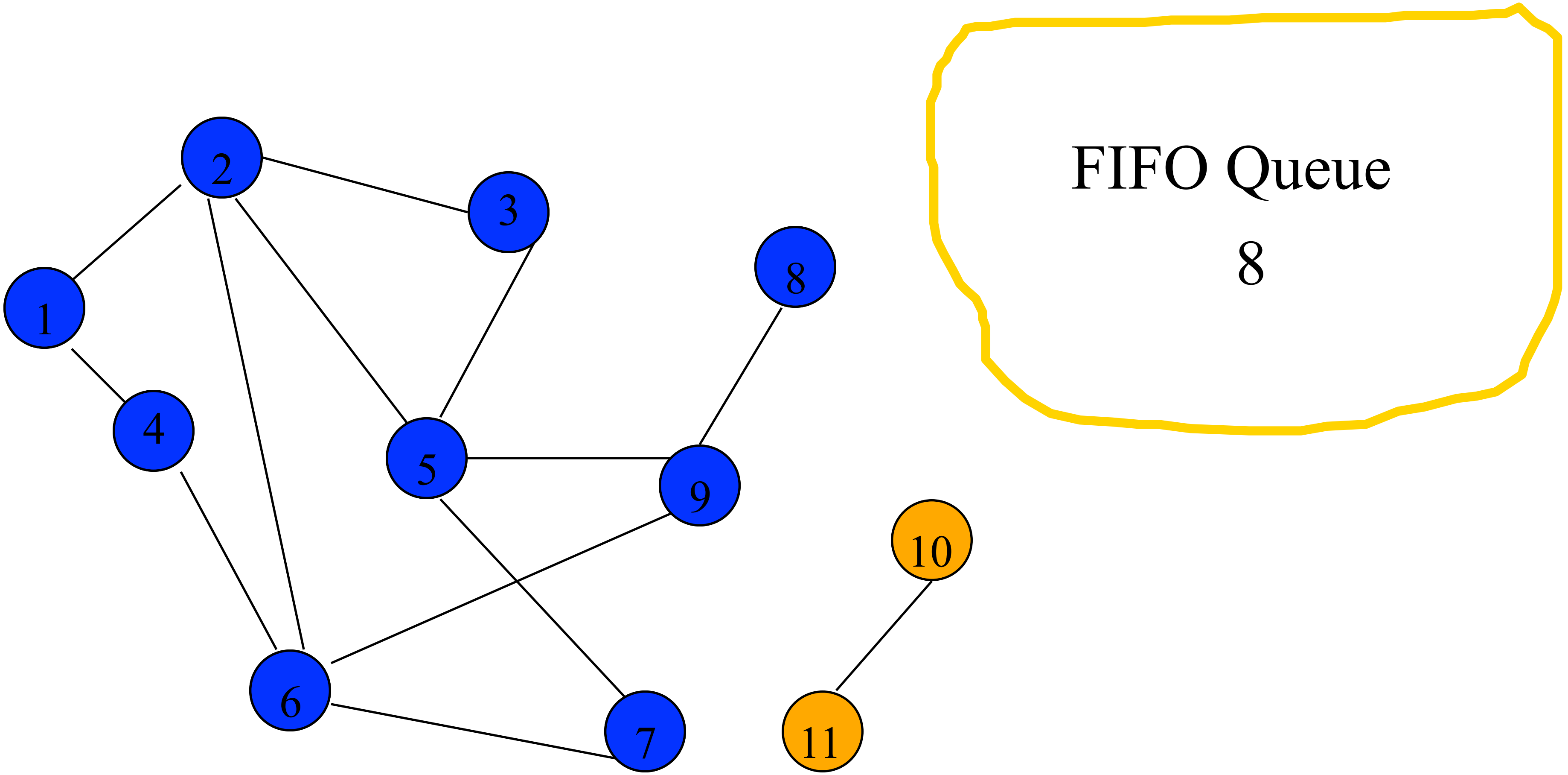
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



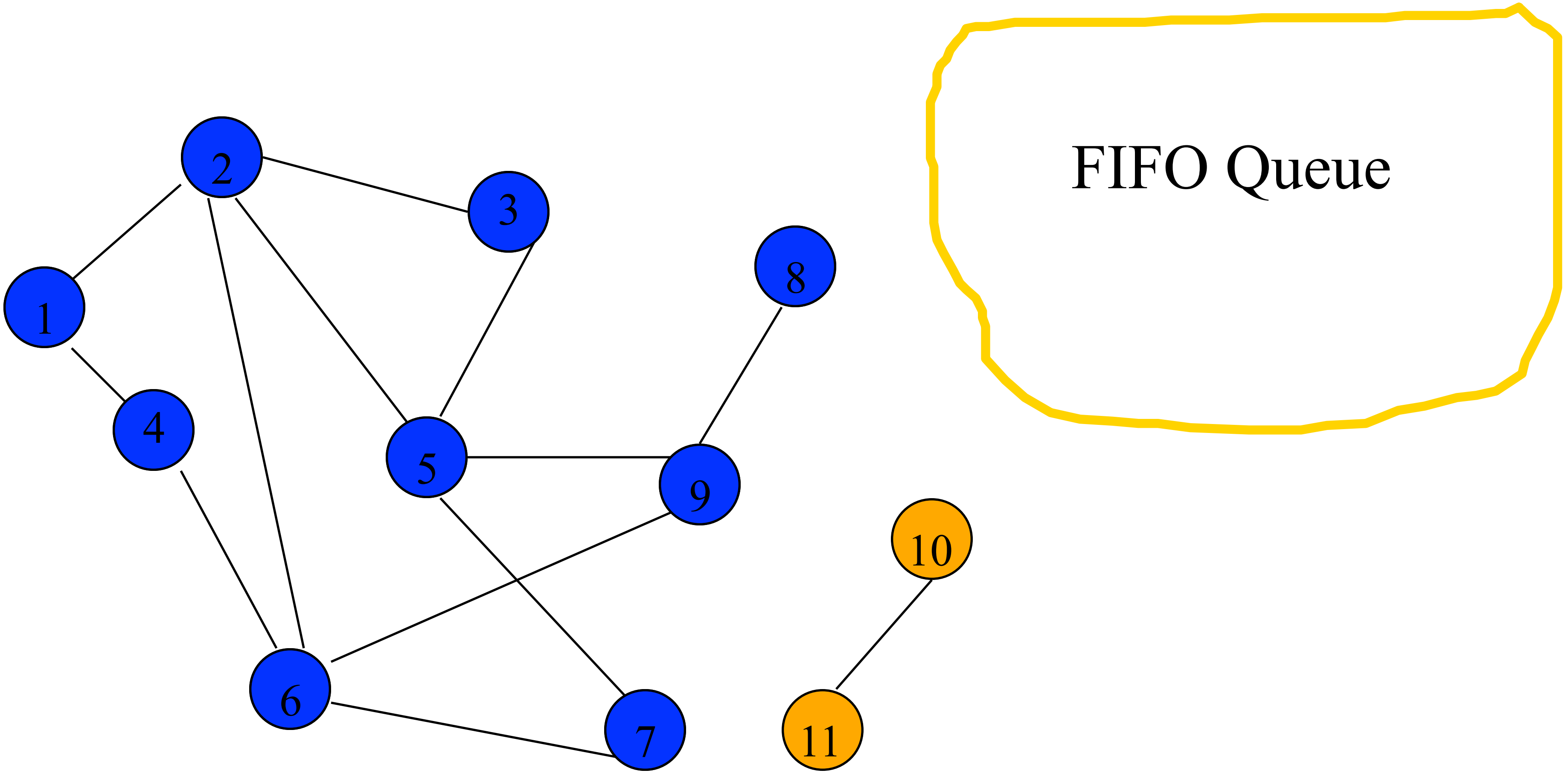
Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

Breadth-First Search Example



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

Breadth-First Search Example



Queue is empty. Search terminates.

Breadth-First Search Property

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

```
virtual void Graph::BFS (int v) {
    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)
            if (!visited[w]) {
                visited[w] = true;
                q.Push(w);
            }
    }       // end of while
    delete [ ] visited;
}
```

m	vertex1	vertex2	v1link	v2link

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)

Path From Vertex v To Vertex u

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

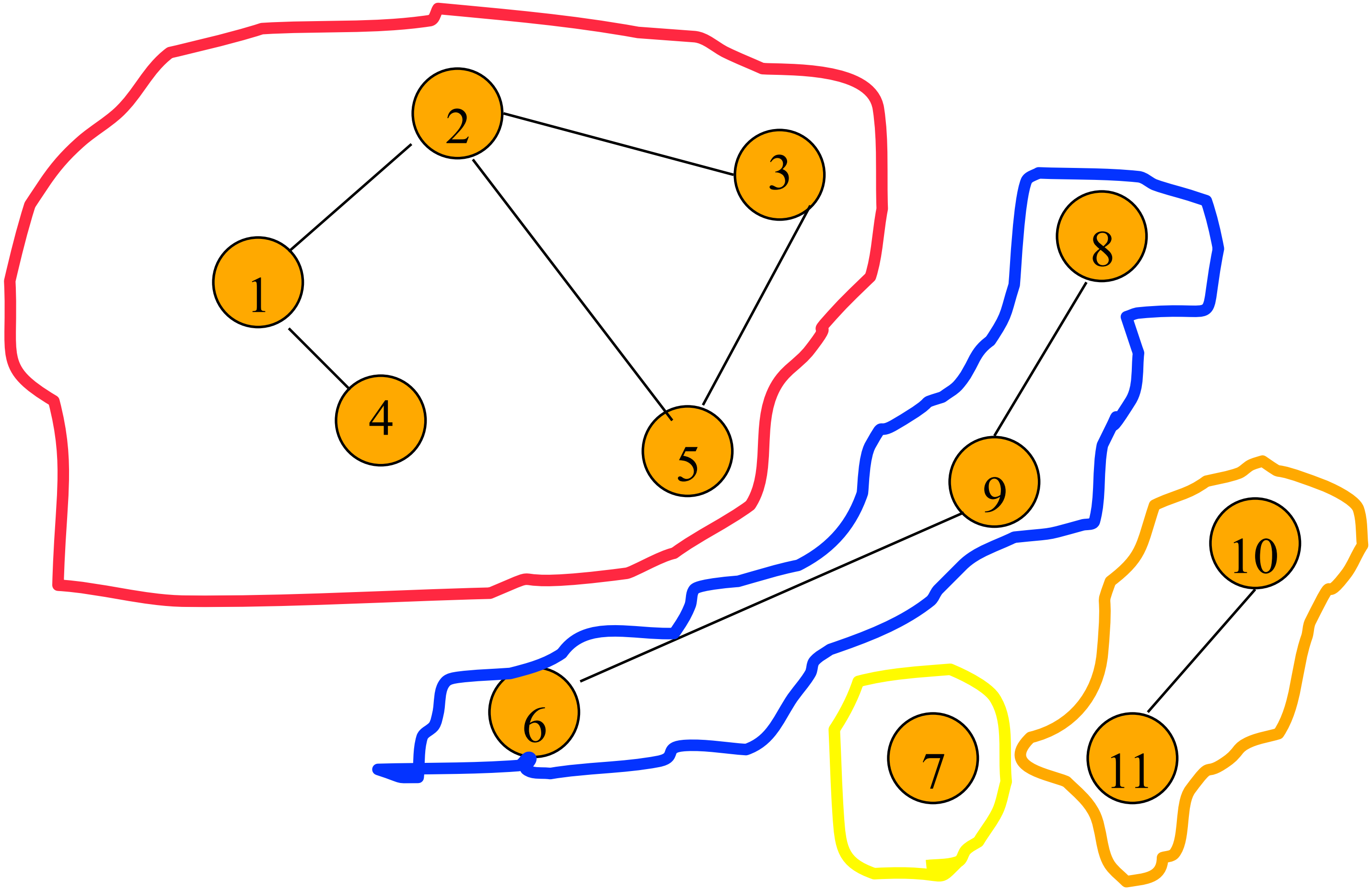
Is The Graph Connected?

- Start a breadth-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 breadth-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



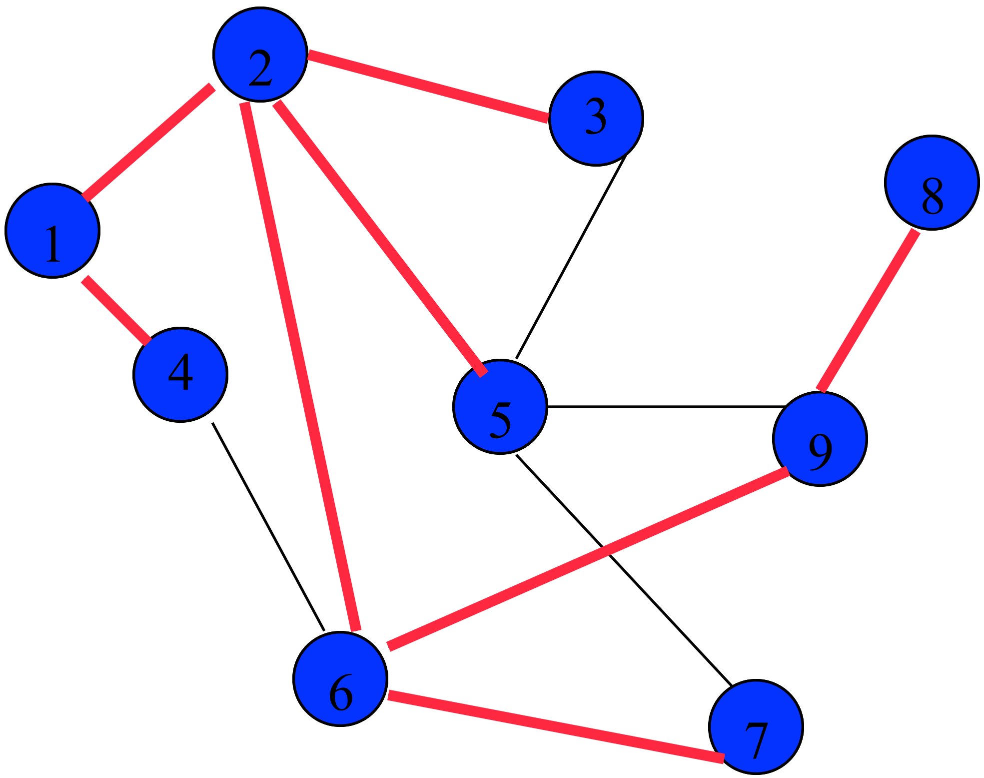
Connected Components

- **virtual void** Graph::Components(){
-     visited = **new bool**[n];
-     fill(visited, visited+n, **false**);
-     **for** (**int** i=0**;** i<n**;** i++)
-         **if** (!visited[i]) {
-             BFS (i)**;** // find a component
-             OutputNewComponent();
-         }
-     **delete** [ ] visited;
- }

Time Complexity

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

Spanning Tree



Breadth-first search from vertex 1.

Breadth-first spanning tree.

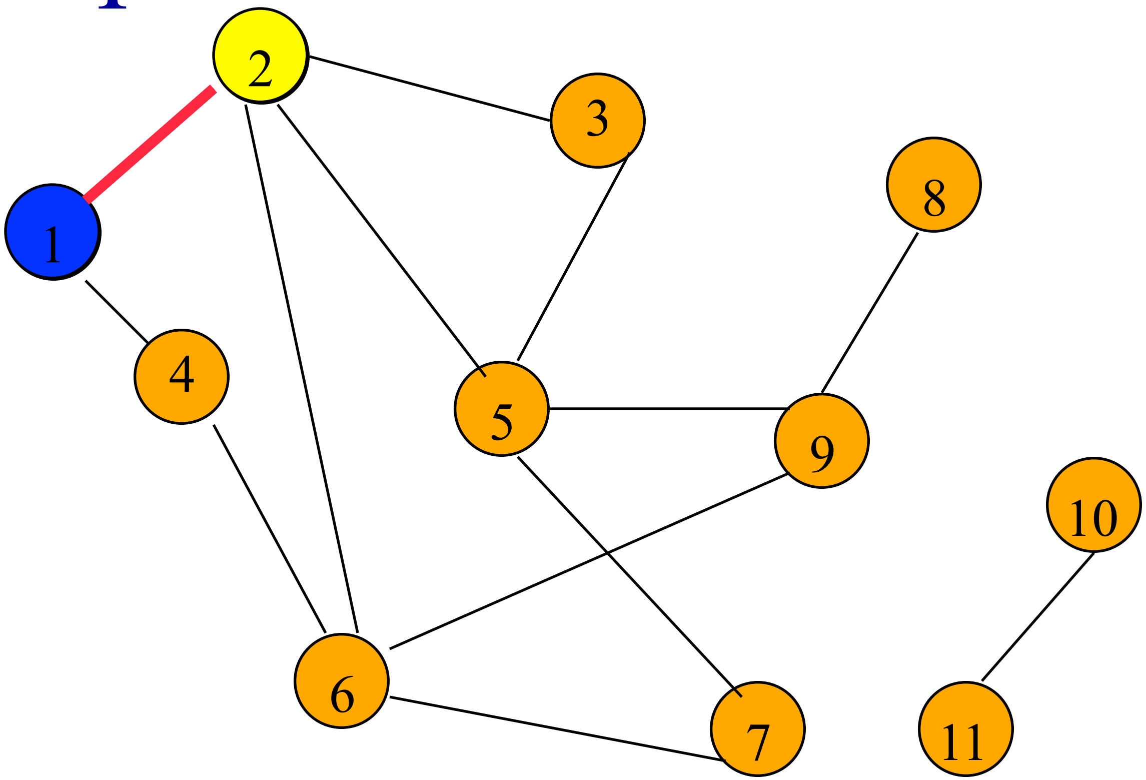
Spanning Tree

- Start a breadth-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 (breadth-first spanning tree).
- Time
  - $O(n^2)$ when adjacency matrix used
  - $O(n+e)$ when adjacency lists used (e is number of edges)

Depth-First Search

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

Depth-First Search Example

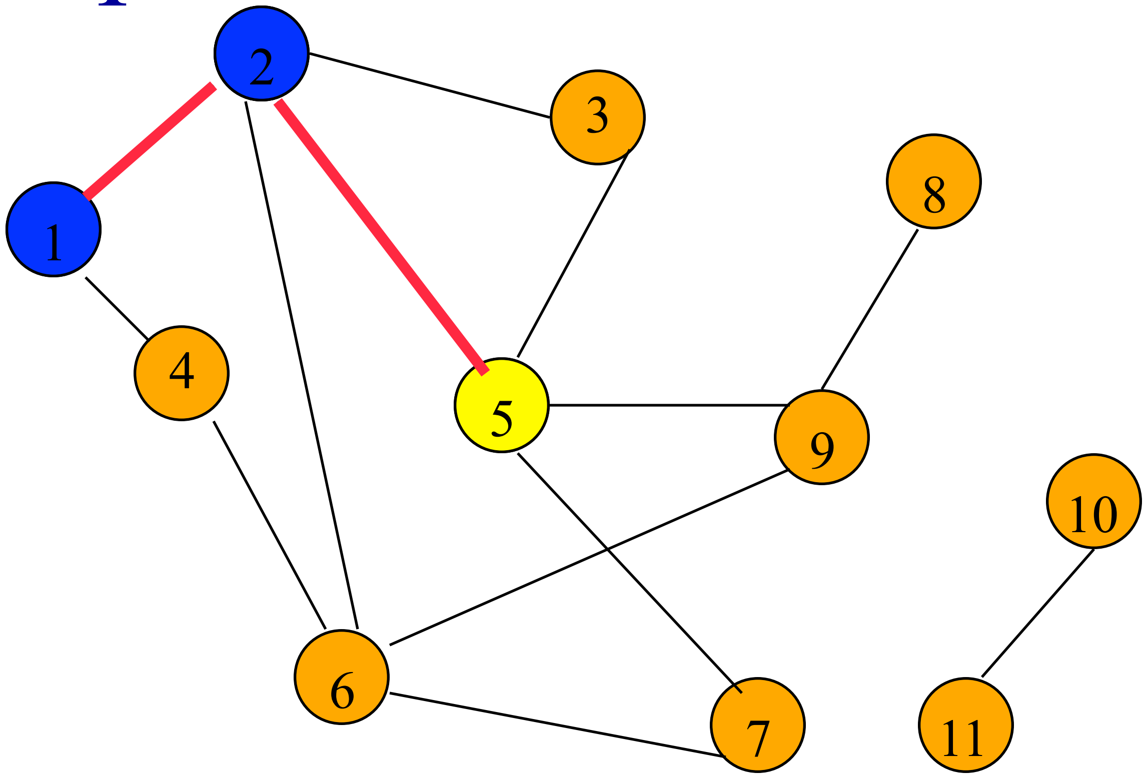


Start search at vertex 1.

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

Suppose that vertex 2 is selected.

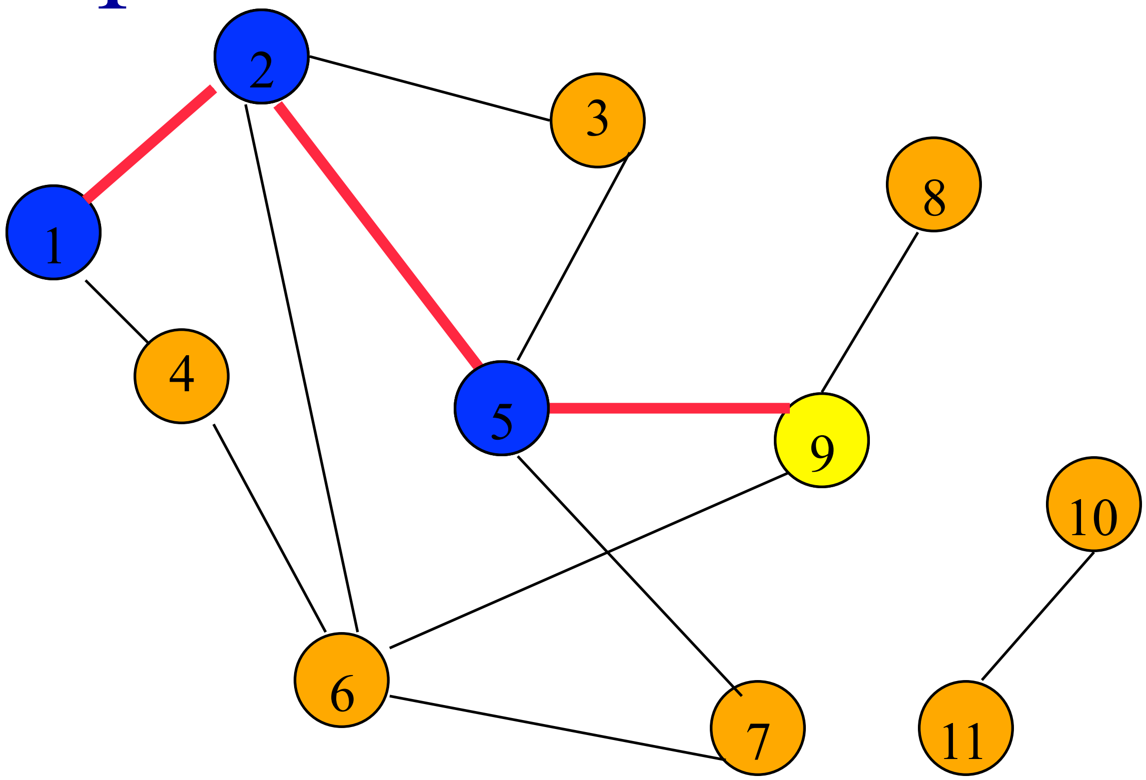
Depth-First Search Example



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

Suppose that vertex 5 is selected.

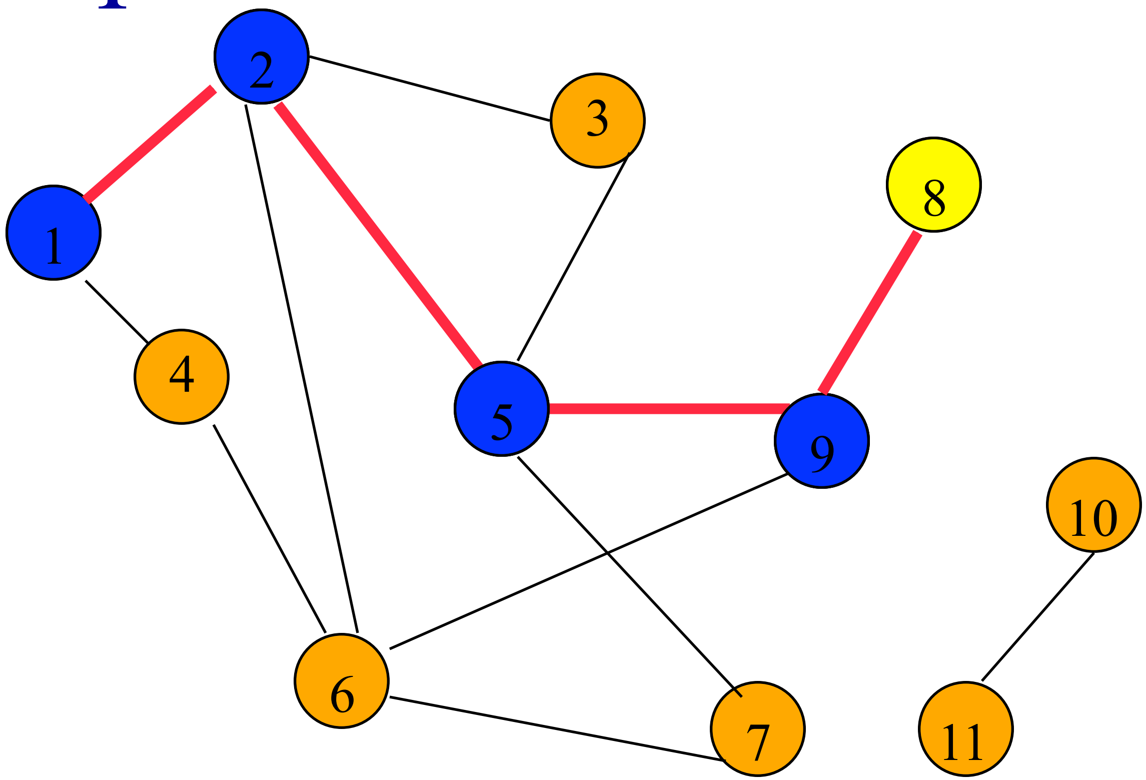
Depth-First Search Example



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

Suppose that vertex 9 is selected.

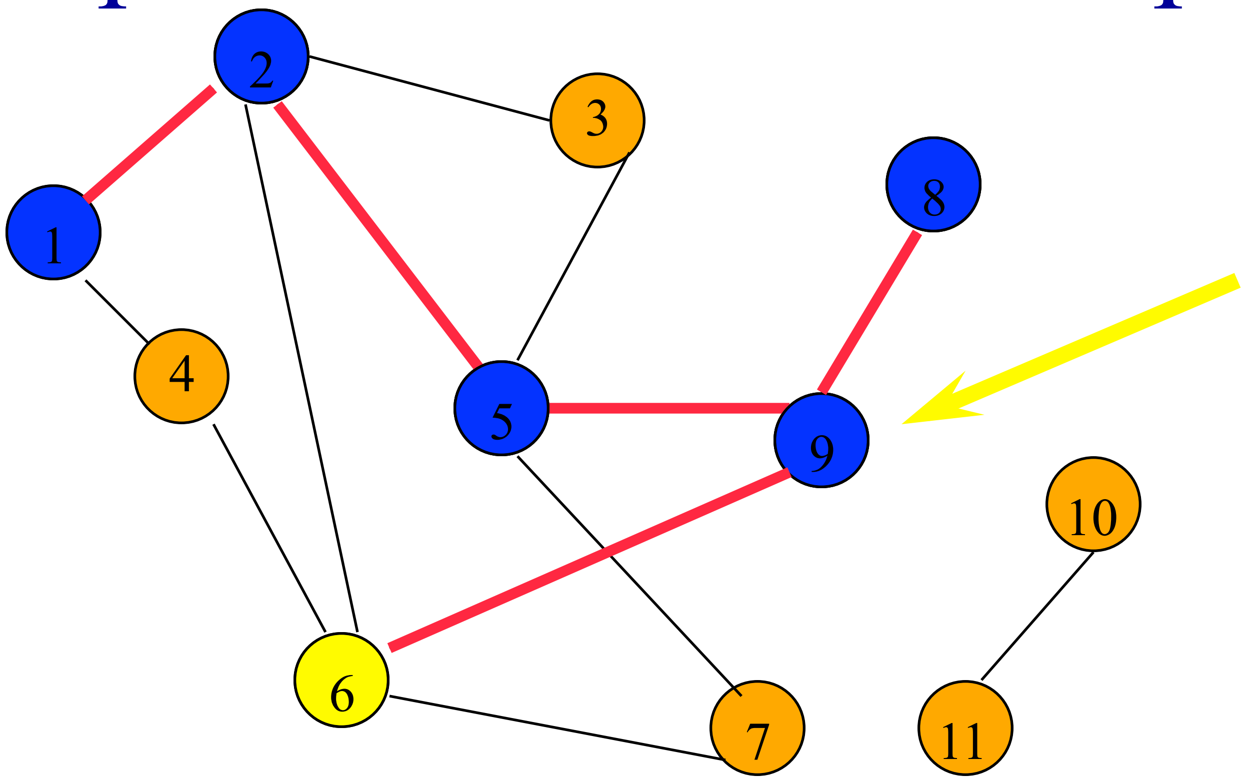
Depth-First Search Example



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

Suppose that vertex 8 is selected.

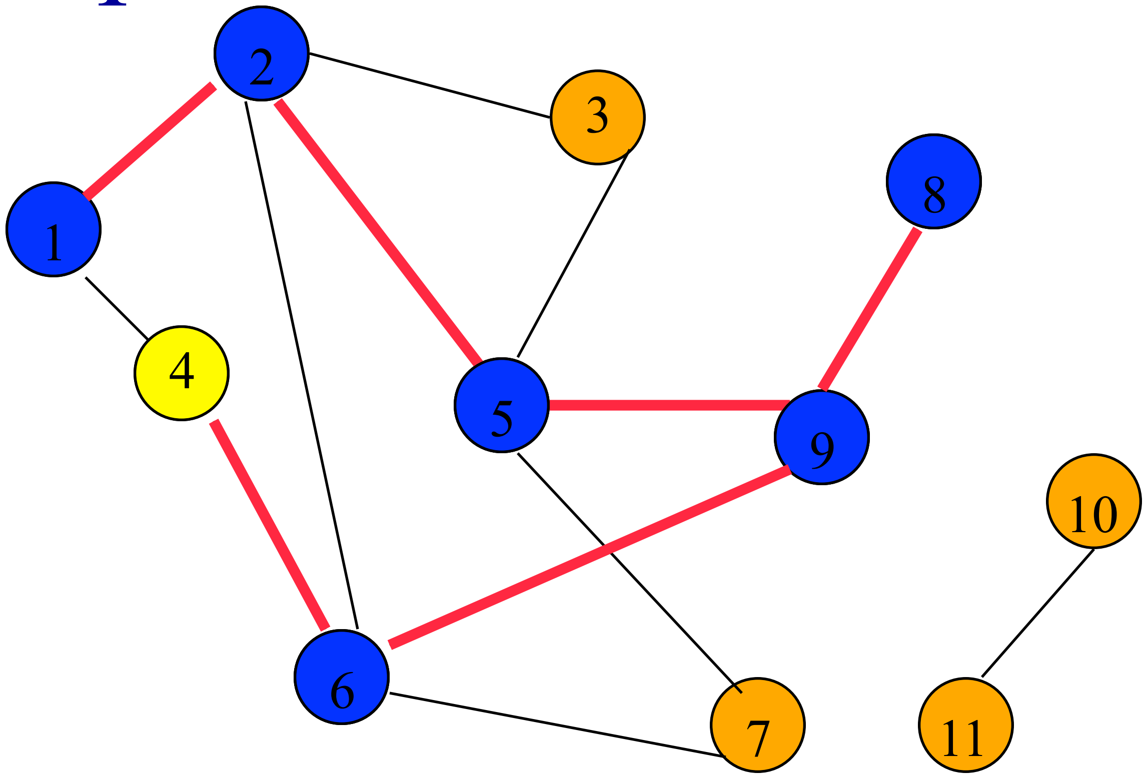
Depth-First Search Example



Label vertex 8 and return to vertex 9.

From vertex 9 do a dfs(6).

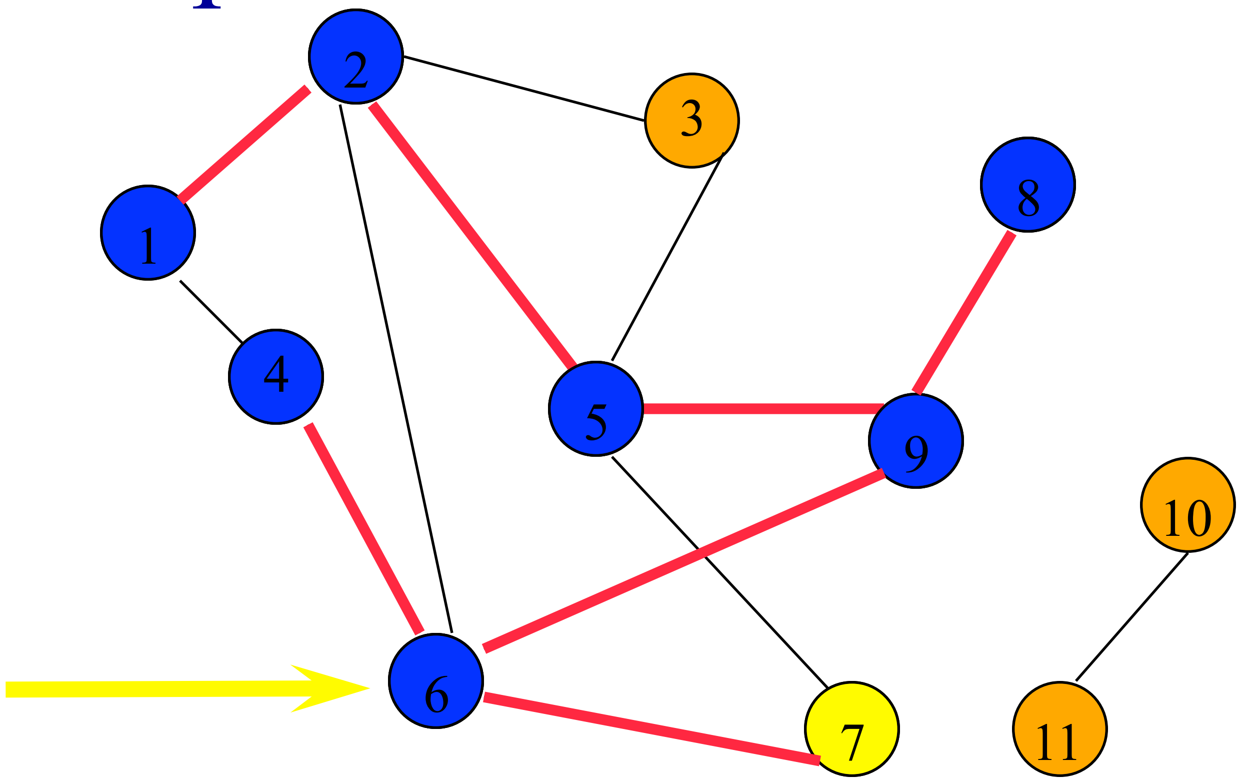
Depth-First Search Example



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

Suppose that vertex 4 is selected.

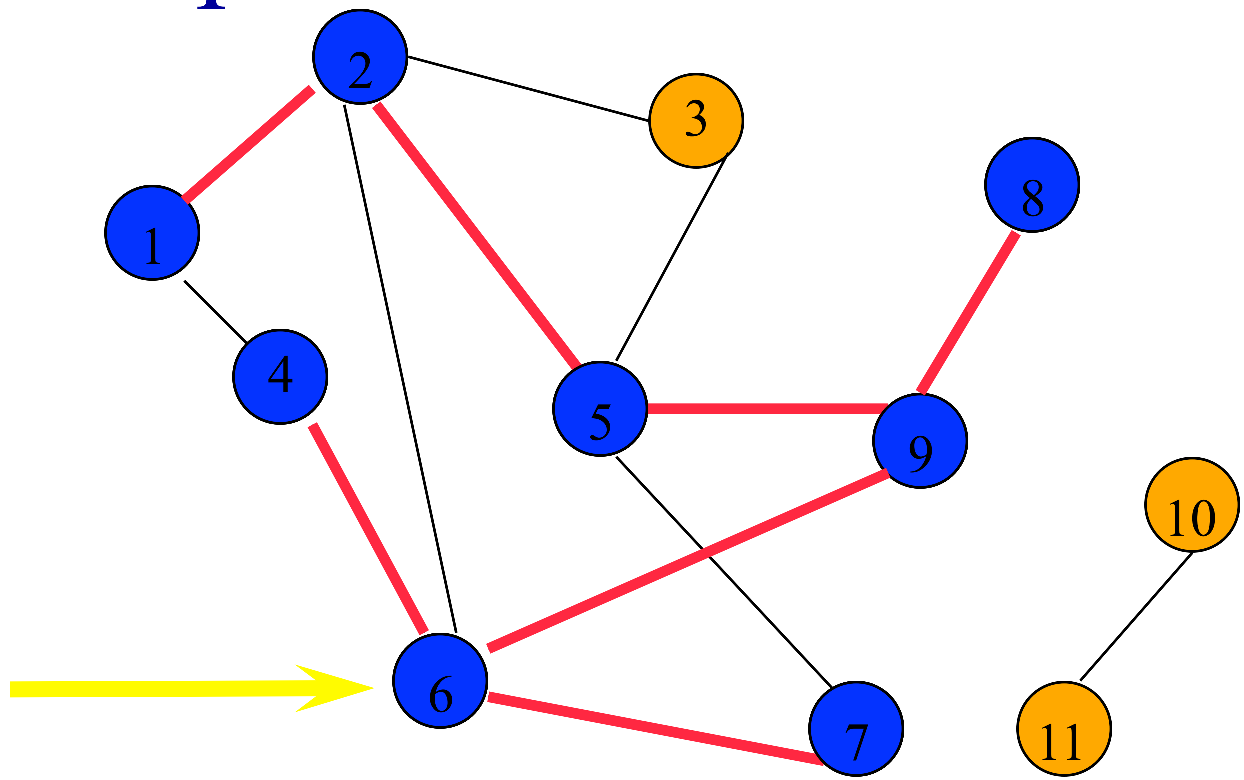
Depth-First Search Example



Label vertex 4 and return to 6.

From vertex 6 do a dfs(7).

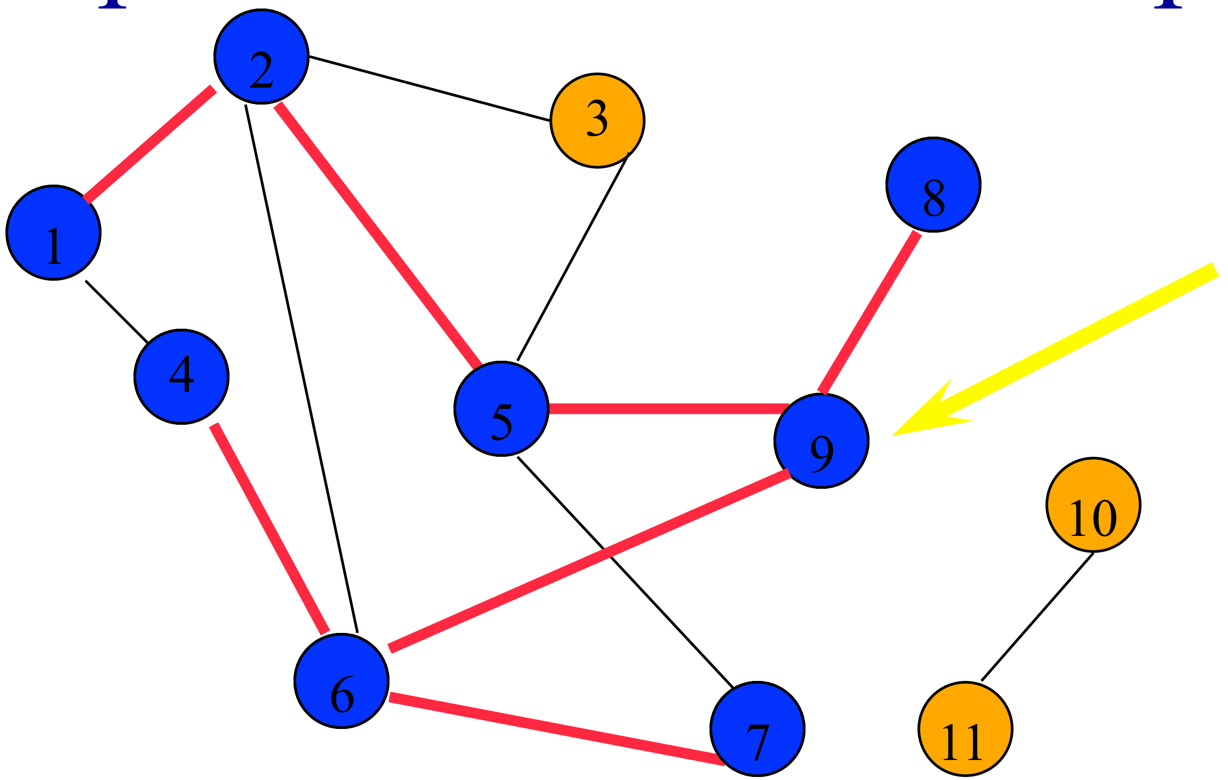
Depth-First Search Example



Label vertex 7 and return to 6.

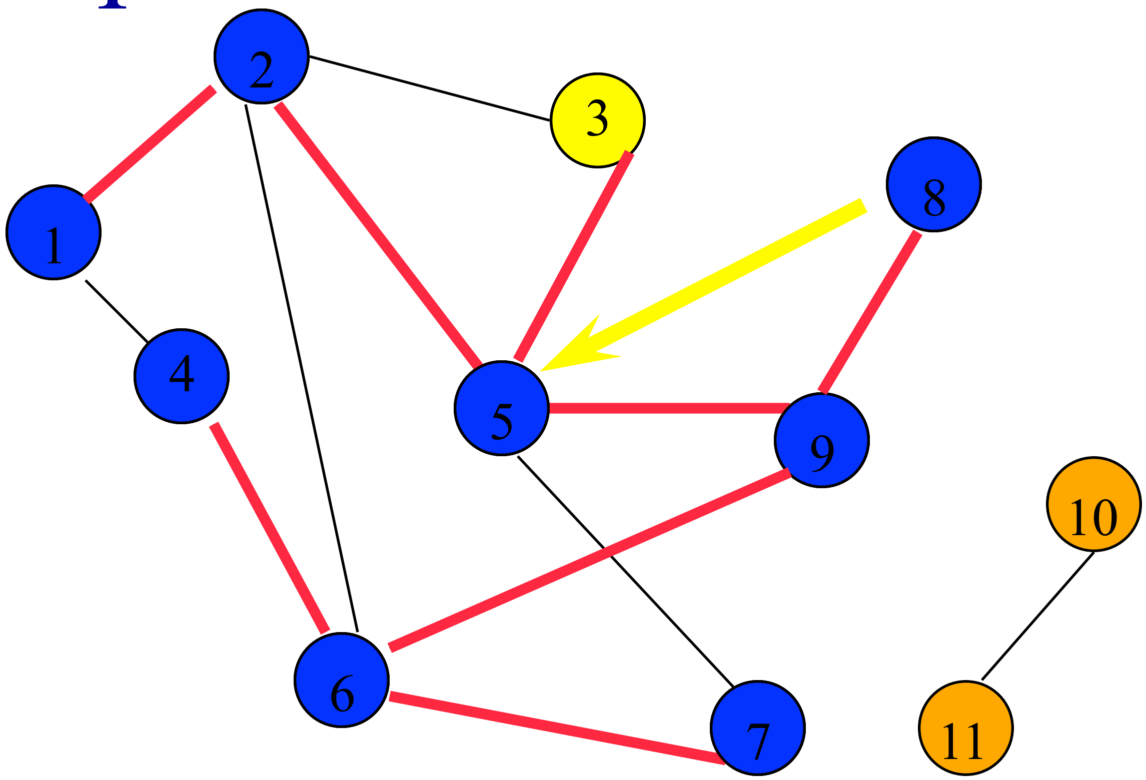
Return to 9.

Depth-First Search Example



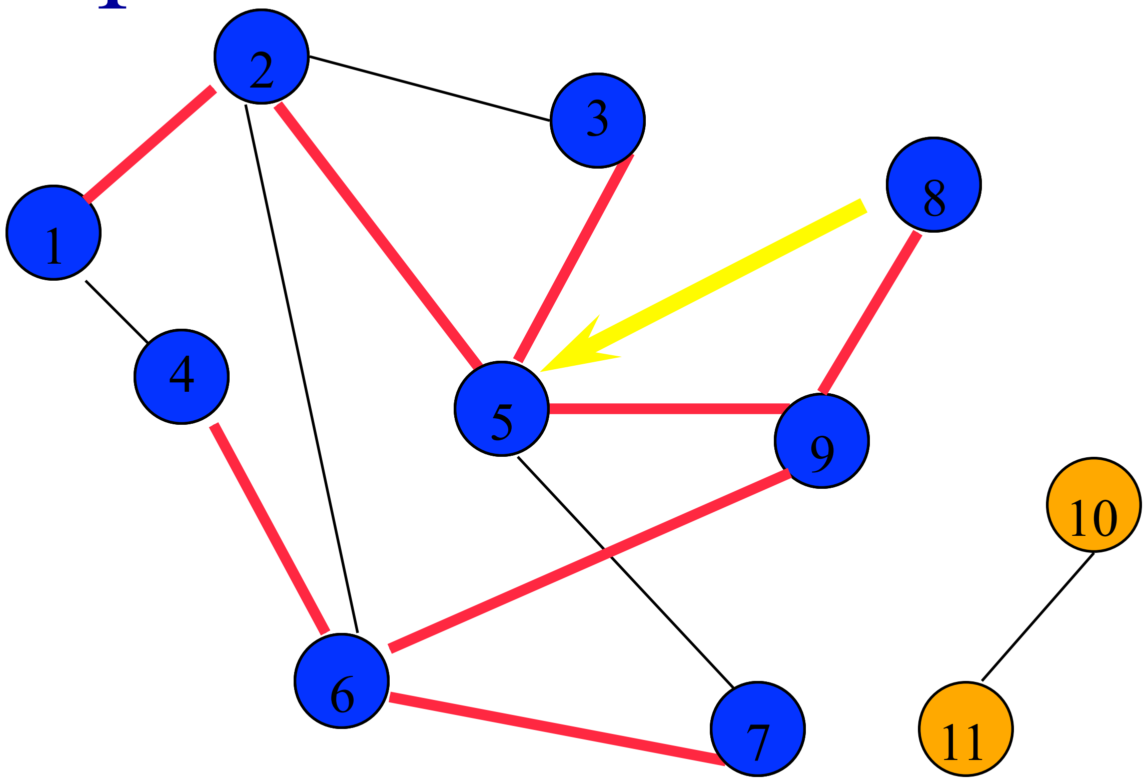
Return to 5.

Depth-First Search Example



Do a dfs(3).

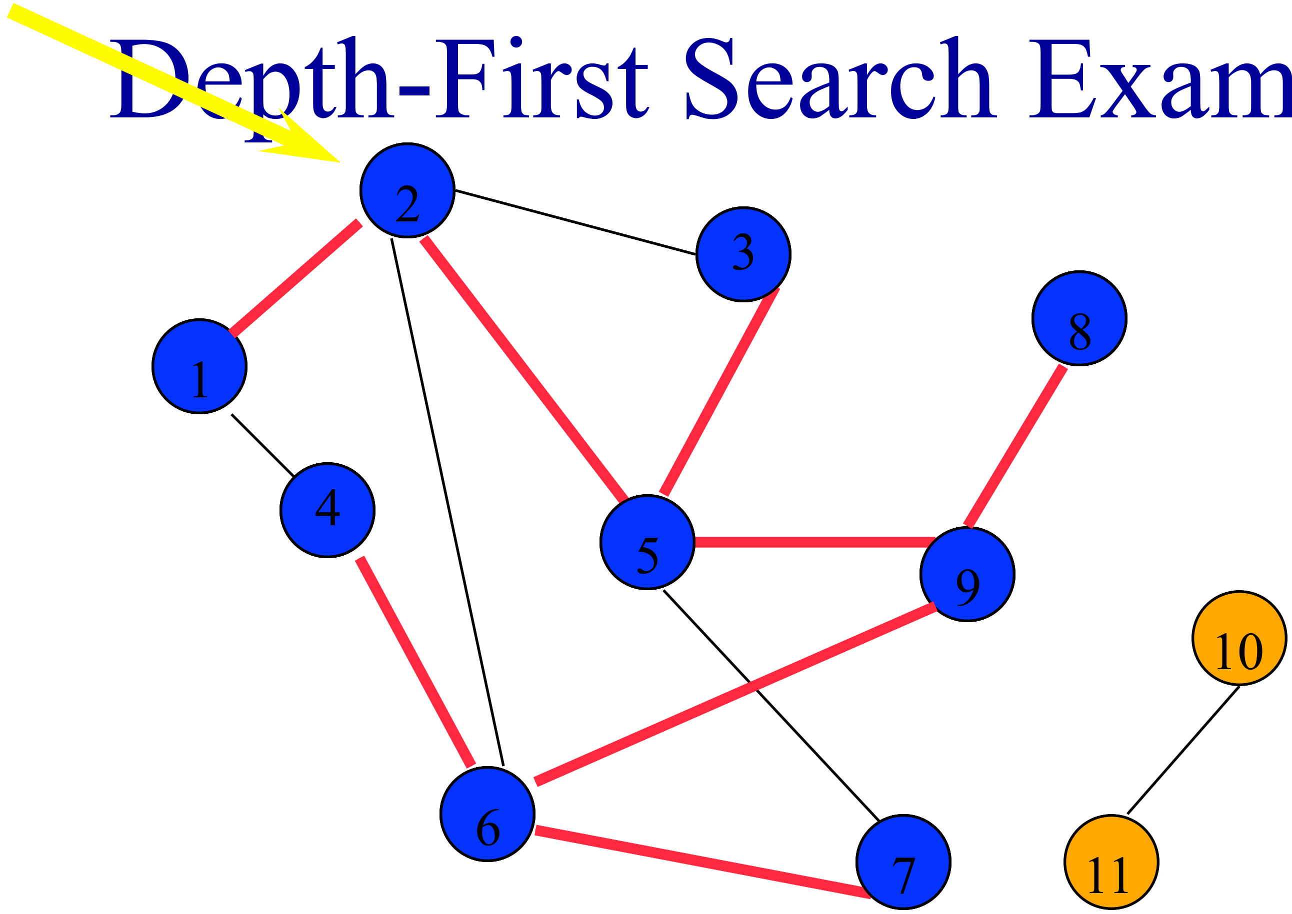
Depth-First Search Example



Label 3 and return to 5.

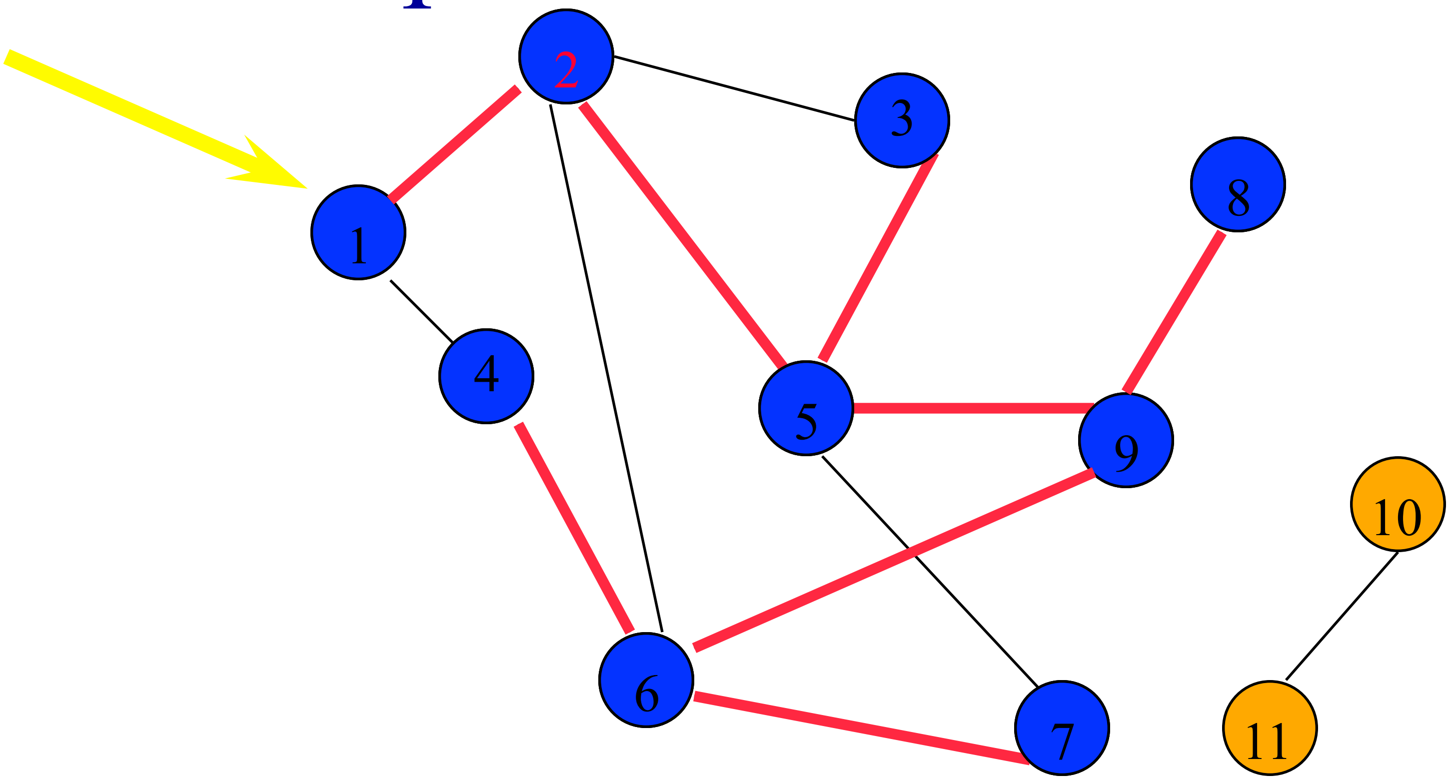
Return to 2.

Depth-First Search Example



Return to 1.

Depth-First Search Example



Return to invoking method.

Depth-First Search Properties

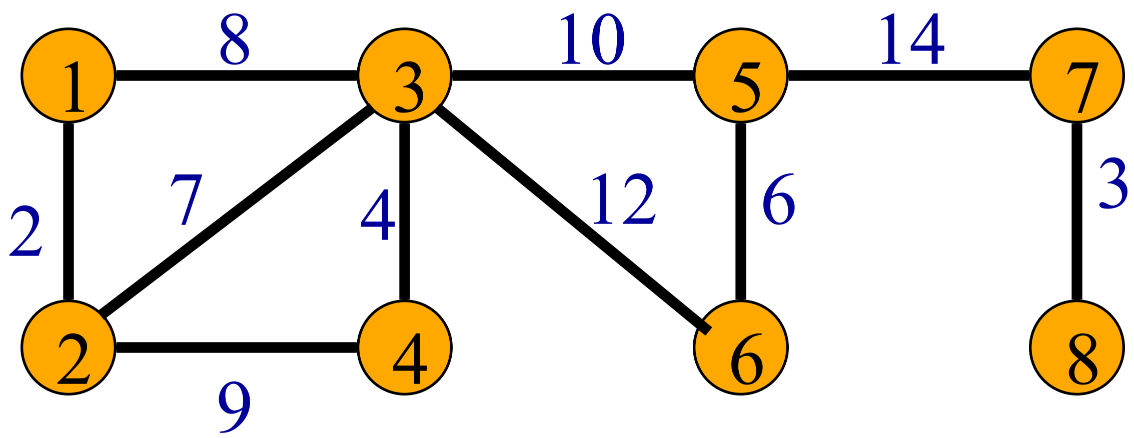
- Same complexity as BFS.
- Same properties with respect to path finding, connected components, and spanning trees.
- Edges used to reach unlabeled vertices define a depth-first spanning tree when the graph is connected.
- There are problems for which bfs is better than dfs and vice versa.

- **Exercises: P352-3, 5, 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.

Greedy Method

- Solve problem by making a sequence of decisions.
- Decisions are made one by one in some order.
- Each decision is made using a greedy criterion.
- A decision, once made, is (usually) not changed later.

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.

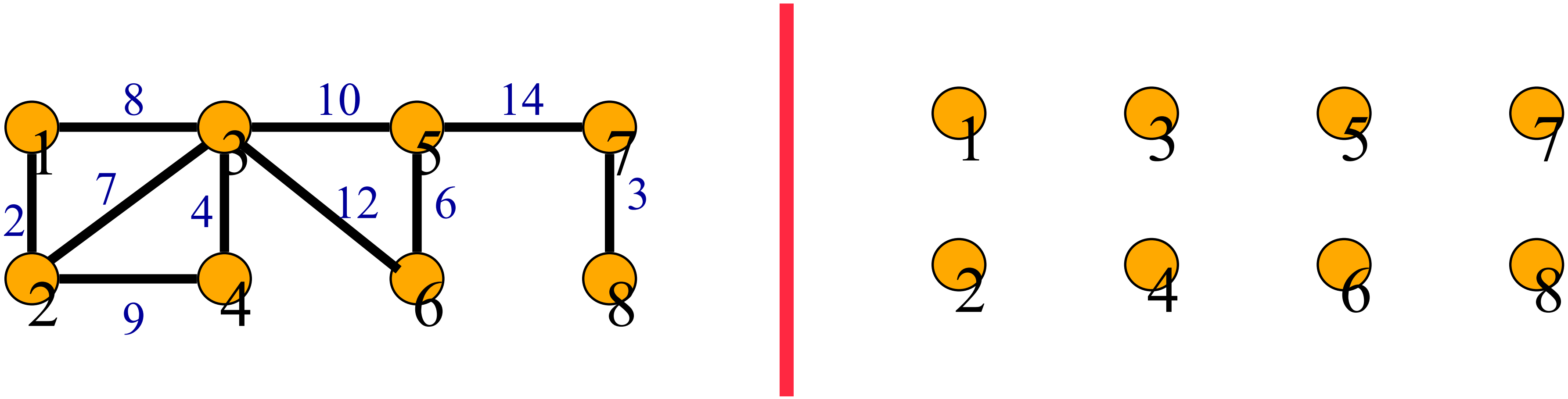
Edge Selection Greedy Strategies

- 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.

Edge Rejection Greedy Strategies

- Start with the connected graph. Repeatedly find a cycle and eliminate the highest cost edge on this cycle. Stop when no cycles remain.
- Consider edges in descending order of cost. Eliminate an edge provided this leaves behind a connected graph.

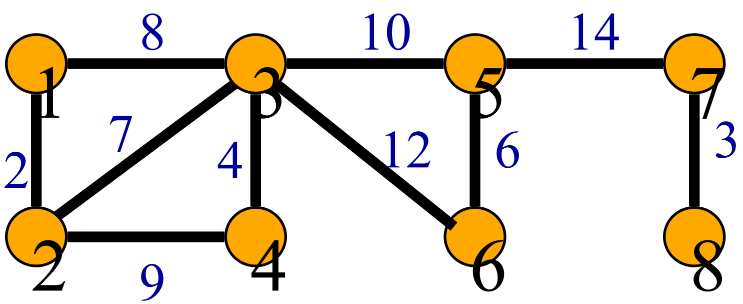
Kruskal’s Method

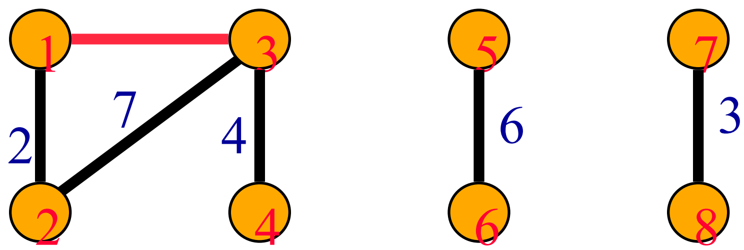


- Start with a forest that has no edges.

- Consider edges in ascending order of cost.
- Edge (1,2) is considered first and added to the forest.

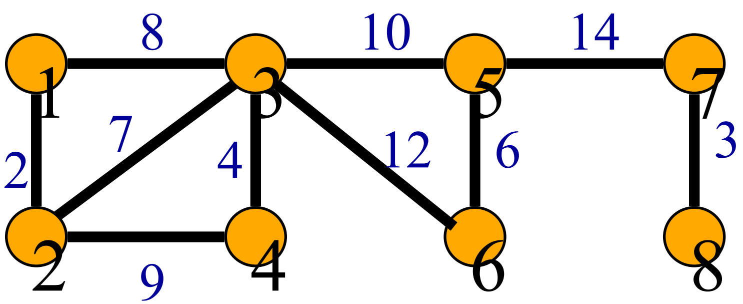
Kruskal's Method

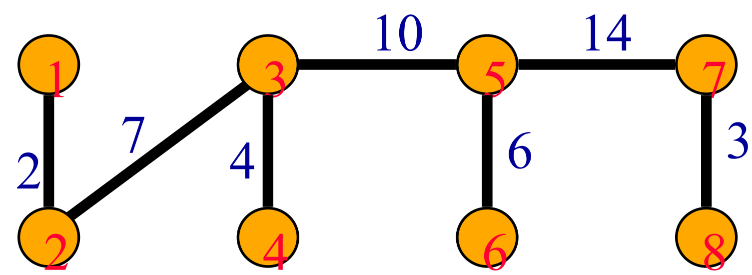




- Edge (7,8) is considered next and added.
- Edge (3,4) is considered next and added.
- Edge (5,6) is considered next and added.
- Edge (2,3) is considered next and added.
- Edge (1,3) is considered next and rejected because it creates a cycle.

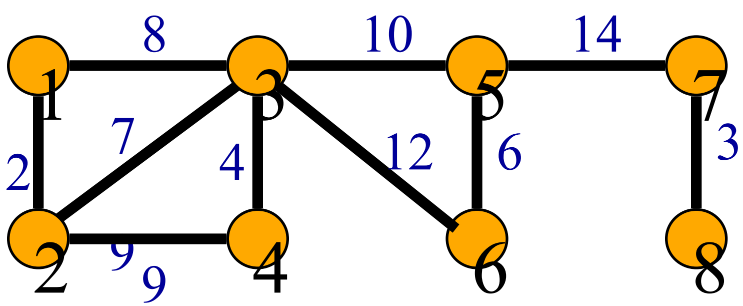
Kruskal’s Method

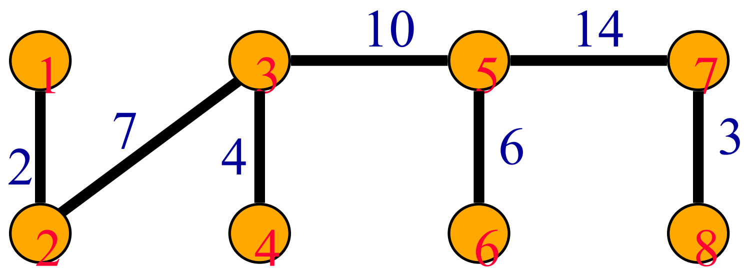




- Edge (2,4) is considered next and rejected because it creates a cycle.
- Edge (3,5) is considered next and added.
- Edge (3,6) is considered next and rejected.
- Edge (5,7) is considered next and added.

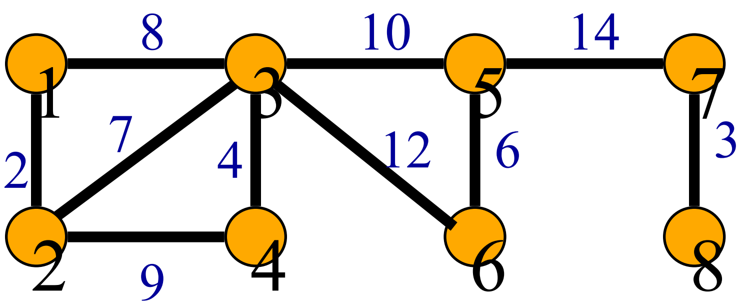
Kruskal's Method

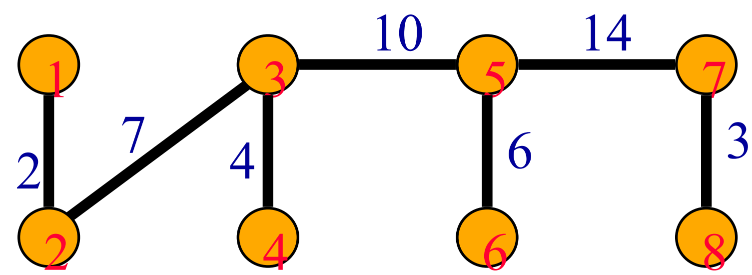




- n - 1 edges have been selected and no cycle formed.
- So we must have a spanning tree.
- Cost is 46.
- Min-cost spanning tree is unique when all edge costs are different.

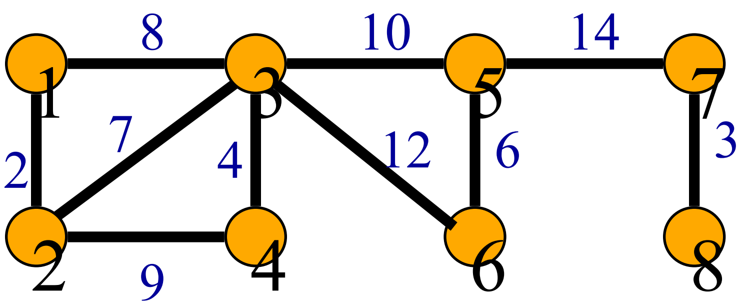
Prim's Method

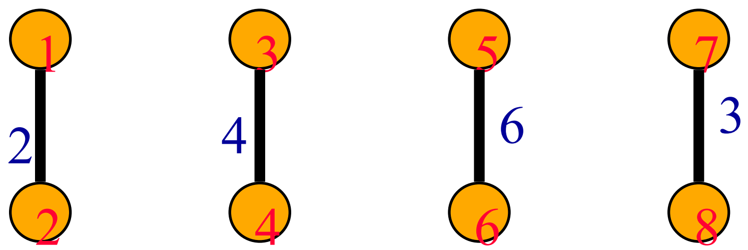




- Start with any single vertex tree.
- Get a 2-vertex tree by adding a cheapest edge.
- Get a 3-vertex tree by adding a cheapest edge.
- Grow the tree one edge at a time until the tree has n - 1 edges (and hence has all n vertices).

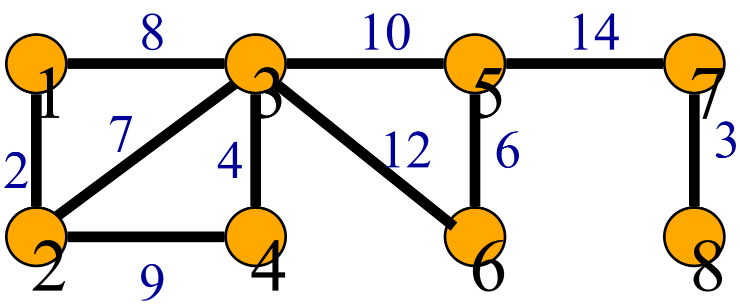
Sollin's Method

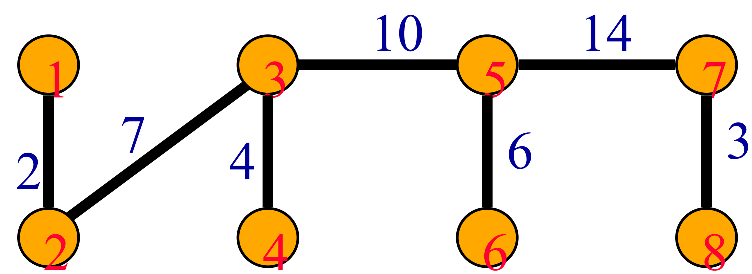




- Start with a forest that has no edges.
- Each component selects a least cost edge with which to connect to another component.
- Duplicate selections are eliminated.
- Cycles are possible when the graph has

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.

Greedy Minimum-Cost Spanning Tree Methods

- Can prove that all result in a minimum-cost spanning tree.
- See Text Book

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.
```

Data Structures For Kruskal’s Method

Edge set E.

Operations are:

- Is E empty?
- Select and remove a least-cost edge.

Use a min heap of edges.

- Initialize. O(e) time.
- Remove and return least-cost edge. O(log e) time.

Data Structures For Kruskal's Method

Set of selected edges T.

Operations are:

- Does T have n - 1 edges?
- Does the addition of an edge (u, v) to T result in a cycle?
- Add an edge to T.

Data Structures For Kruskal's Method

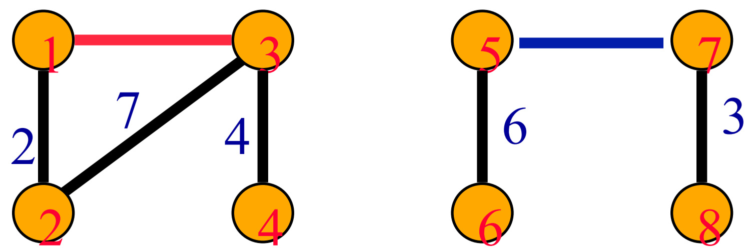
Use an array linear list for the edges of T.

- Does T have n - 1 edges?
  - Check size of linear list. O(1) time.
- Does the addition of an edge (u, v) to T result in a cycle?
  - Not easy.
- Add an edge to T.
  - Add at right end of linear list. O(1) time.

Just use an array rather than ArrayLinearList.

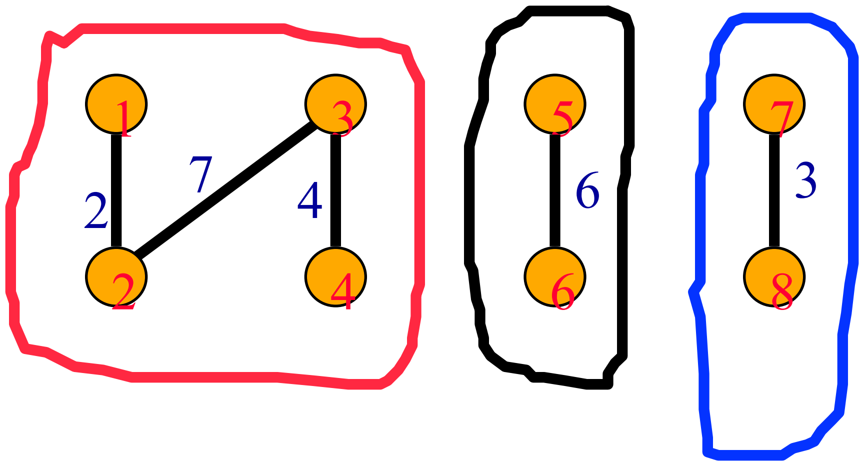
Data Structures For Kruskal's Method

Does the addition of an edge (u, v) to T result in a cycle?



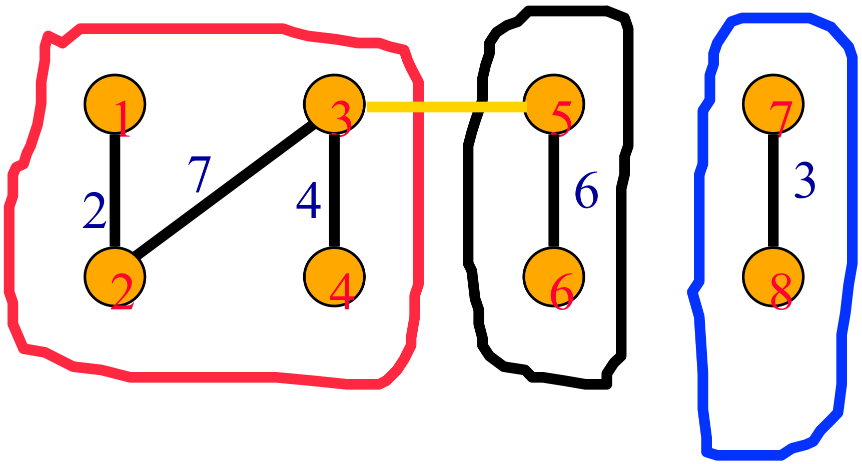
- Each component of T is a tree.
- When u and v are in the same component, the addition of the edge (u,v) creates a cycle.
- When u and v are in the different components, the addition of the edge (u,v) does not create a cycle.

Data Structures For Kruskal's Method



- Each component of T is defined by the vertices in the component.
- Represent each component as a set of vertices.
  - {1, 2, 3, 4}, {5, 6}, {7, 8}
- Two vertices are in the same component iff they are in the same set of vertices.

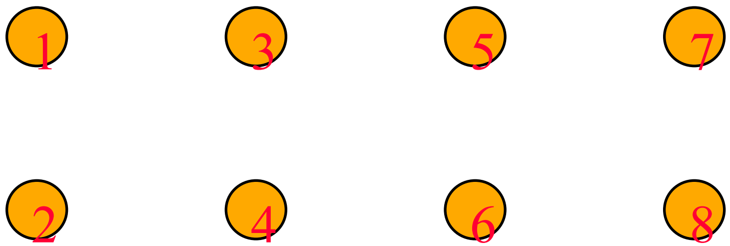
Data Structures For Kruskal's Method



- When an edge (u, v) is added to T, the two components that have vertices u and v combine to become a single component.
- In our set representation of components, the set that has vertex u and the set that has vertex v are united.
  - {1, 2, 3, 4} + {5, 6} => {1, 2, 3, 4, 5, 6}

Data Structures For Kruskal's Method

- Initially, T is empty.



- Initial sets are:
  - {1} {2} {3} {4} {5} {6} {7} {8}
- Does the addition of an edge (u, v) to T result in a cycle? If not, add edge to T.

```
s1 = find(u); s2 = find(v);

if (s1 != s2) union(s1, s2);
```

Data Structures For Kruskal's Method

- Use FastUnionFind.
- Initialize.
  - $O(n)$ time.
- At most $2e$ finds and $n-1$ unions.
  - Very close to $O(n + e)$.
- Min heap operations to get edges in increasing order of cost take $O(e \log e)$.
- Overall complexity of Kruskal's method is $O(n + e \log e)$.

Greedy Minimum-Cost Spanning Tree Methods

- Prim's method is fastest.
  - $O(n^2)$ using an implementation similar to that of Dijkstra's shortest-path algorithm.
  - $O(e + n \log n)$ using a Fibonacci heap.
- Kruskal's uses union-find trees to run in $O(n + e \log e)$ time.

- **Exercises: P359-1**

- Implement a full version algorithm of Kruskal's Method (**Experiment**)

- Implement a BFS algorithm using Adjacency Multilists

Adjacency Multilists

Alist[i] ⟶

m	vertex1	vertex2	v1link	v2link

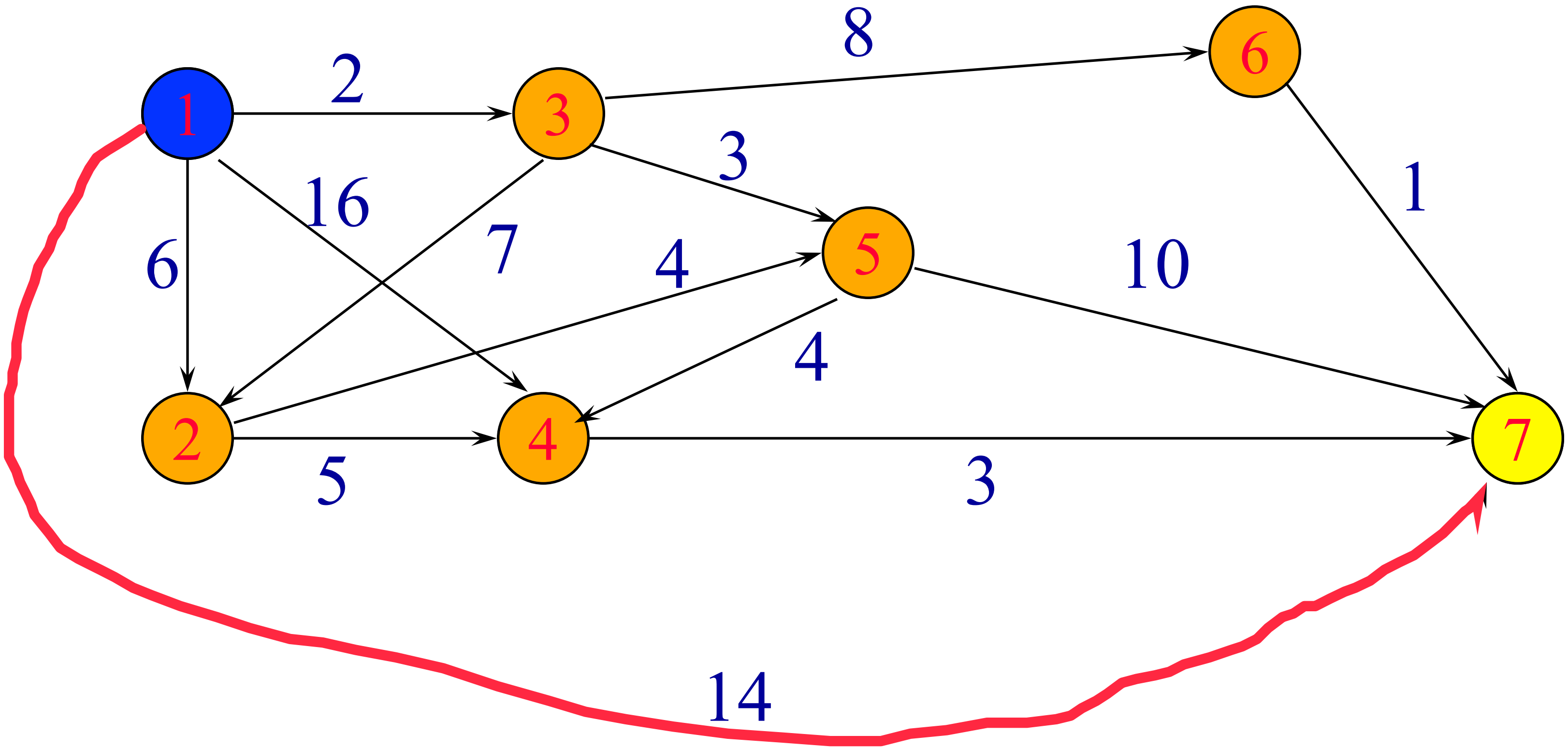
- **virtual void** Graph::BFS (**int** v) {
- visited = **new bool**[n]; fill(visited, visited + n, **false**);
- visited[v] = **true;**
- Queue<**int**> q**;**
- q.Push(v)**;**
- **while** (!q.IsEmpty()) {
- v = q.Front(); q.Pop();
- **ADNode * p = Alist[v];**
- while(p != null){

```
•           int w ;
•           if(p->v1 == v) {
•                 w = p->v2;
•                 p = p->v1link;}
•           else{
•                 w = p->v1;
•                 p = p->v2link;}
•           if (!visited[w]) {
•               q.Push(w);
•               visited[w] = true;}
•         }  // end of while(p)
•           }        // end of while(q)
•     delete [ ] visited; }
```

Shortest Path Problems

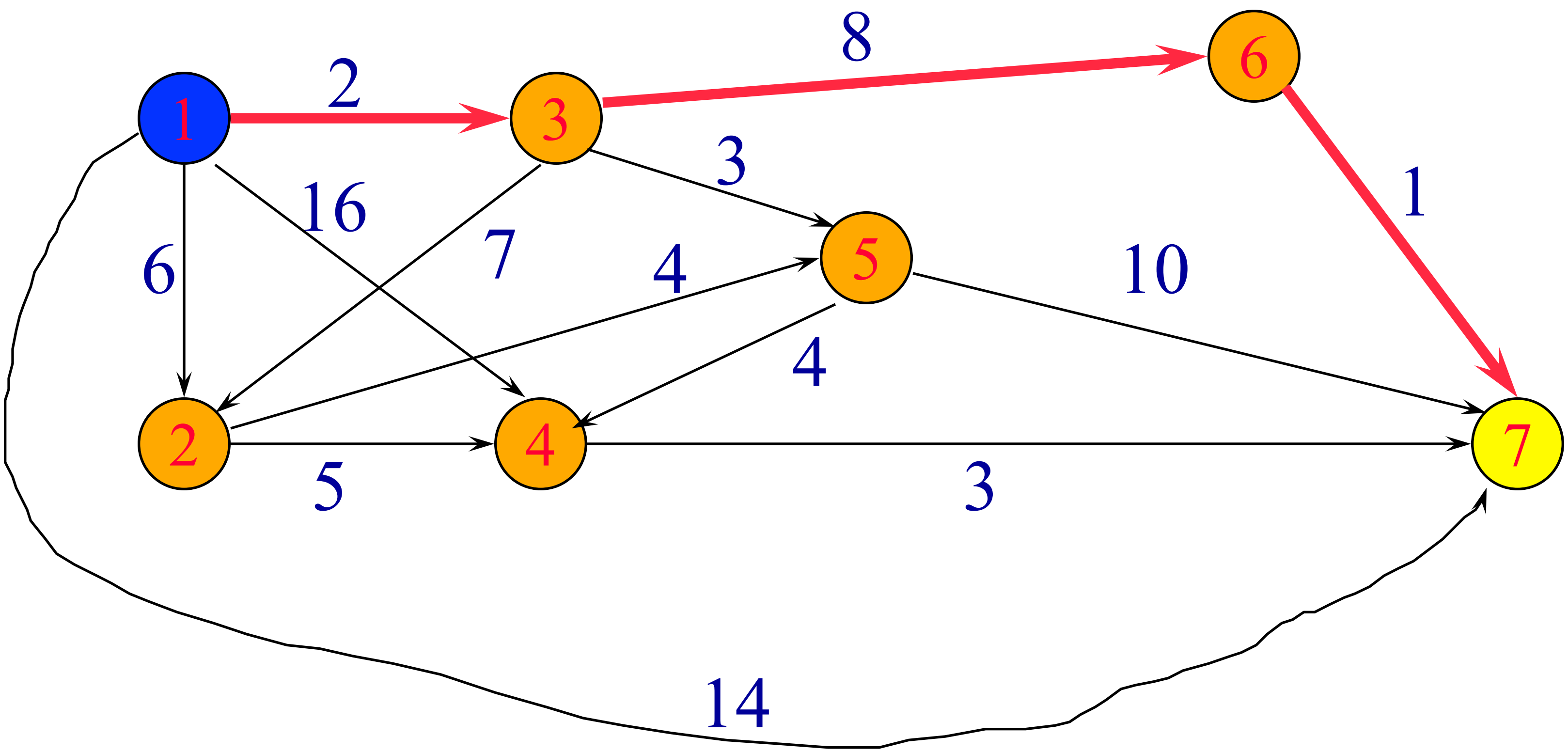
- Directed weighted graph.
- Path length is sum of weights of edges on path.
- The vertex at which the path begins is the source vertex.
- The vertex at which the path ends is the destination vertex.

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

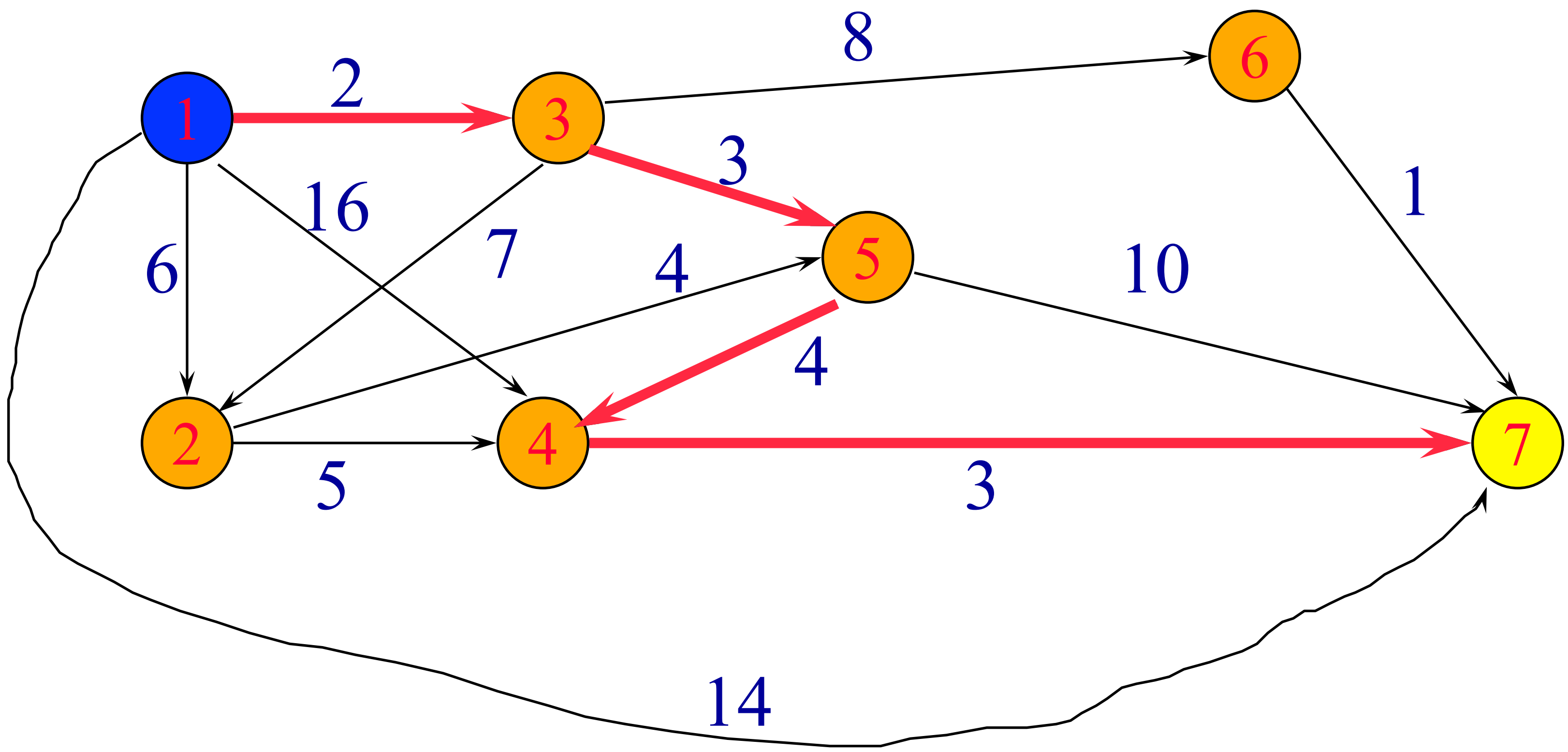
- Single source single destination.
- Single source all destinations.
- All pairs (every vertex is a source and destination).

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!

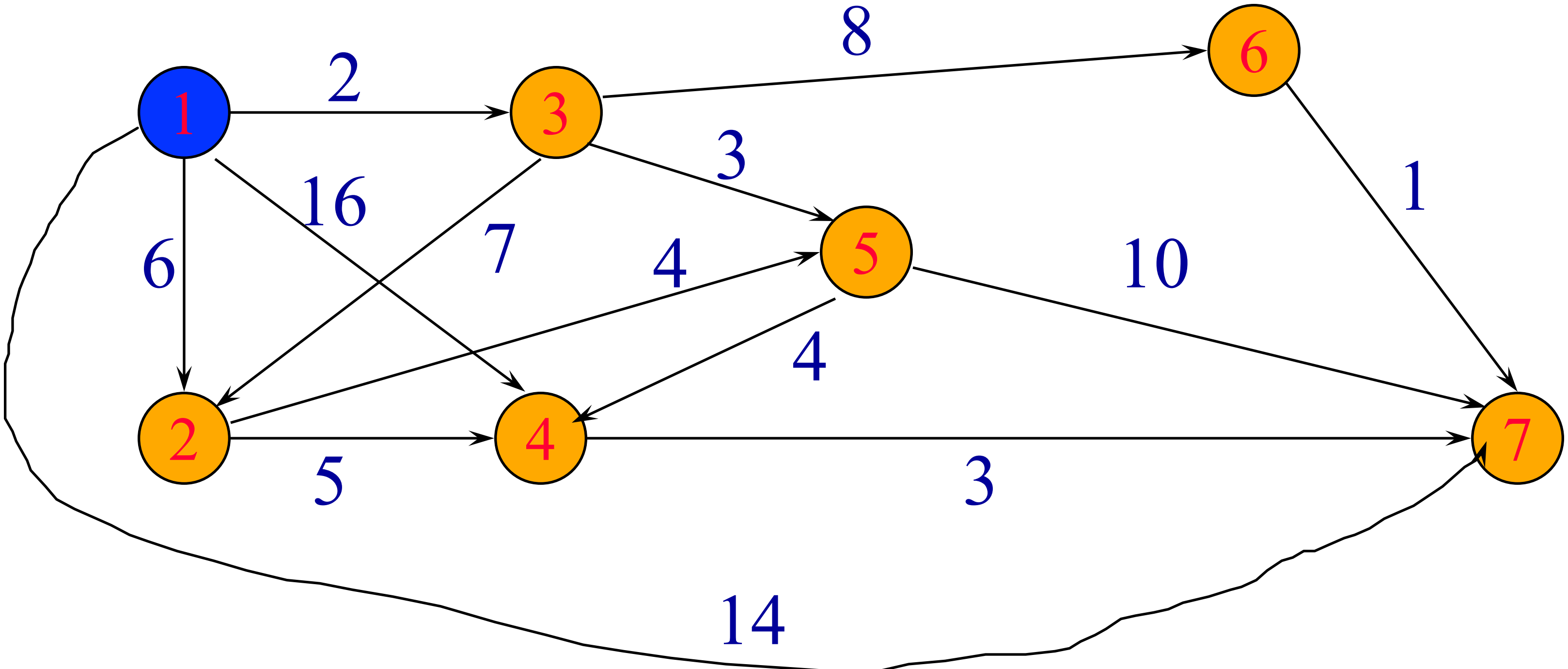
Single Source All Destinations

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

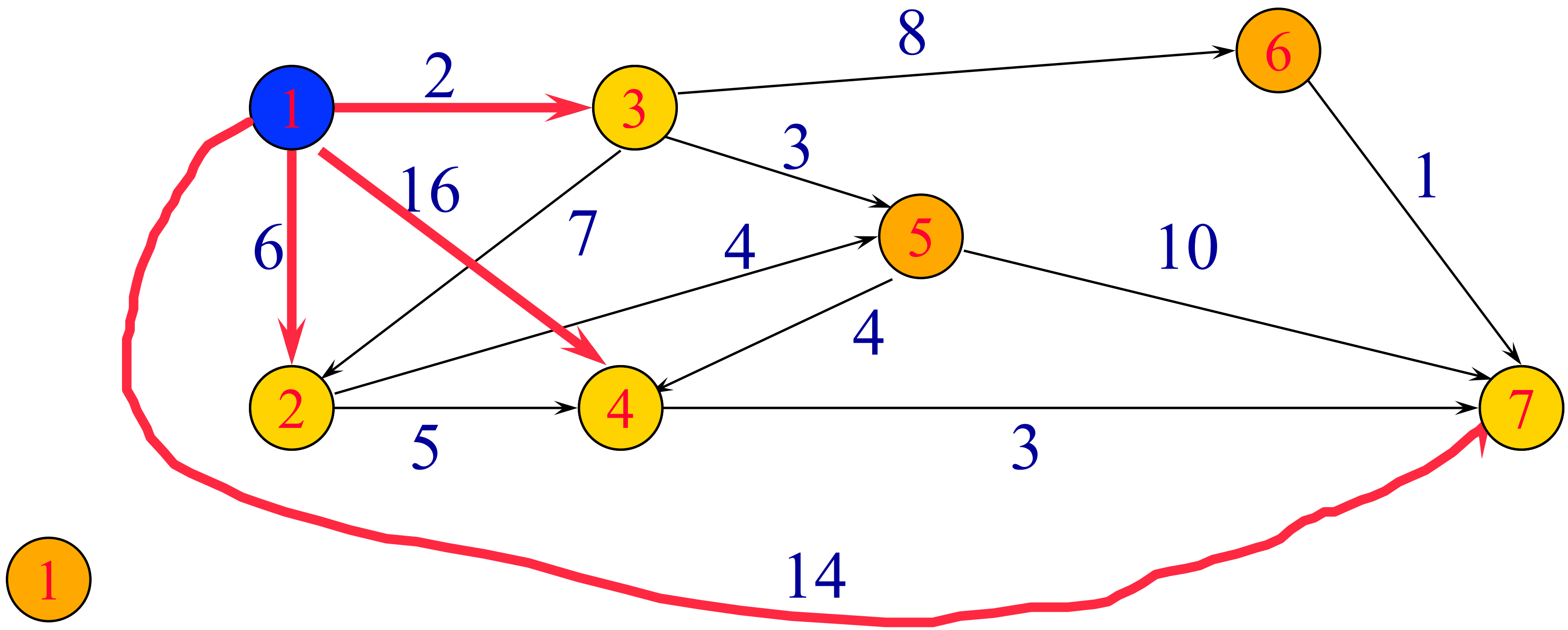
Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

- 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

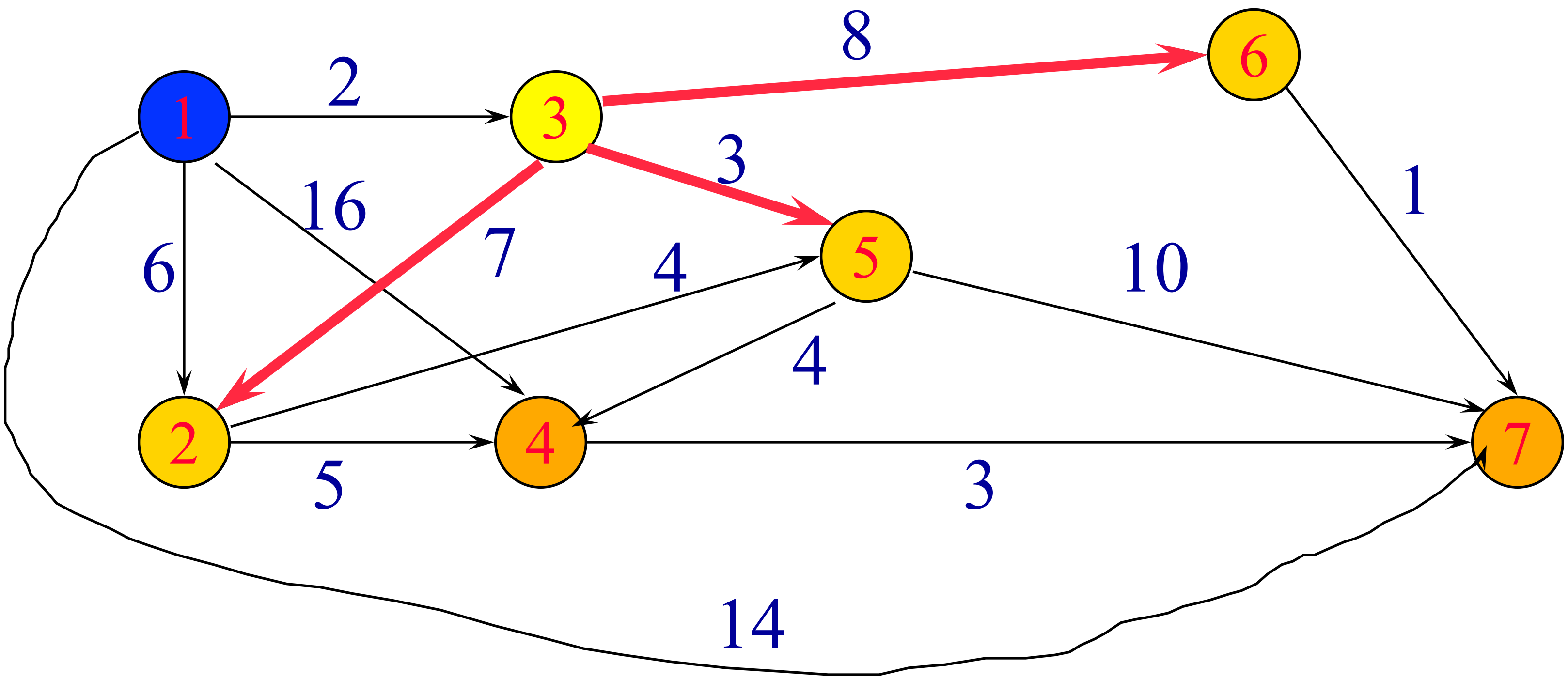
- Let d(i) (distanceFromSource(i)) be the length of a shortest one edge extension of an already generated shortest path, the one edge extension ends at vertex i.
- The next shortest path is to an as yet unreached vertex for which the d() value is least.
- Let p(i) (predecessor(i)) be the vertex just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

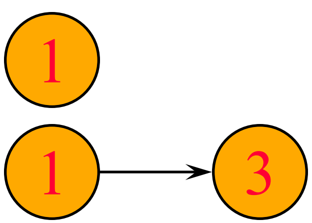
Greedy Single Source All Destinations



	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	$\boxed{2}$	16	-	-	14
p	-	1	1	1	-	-	1

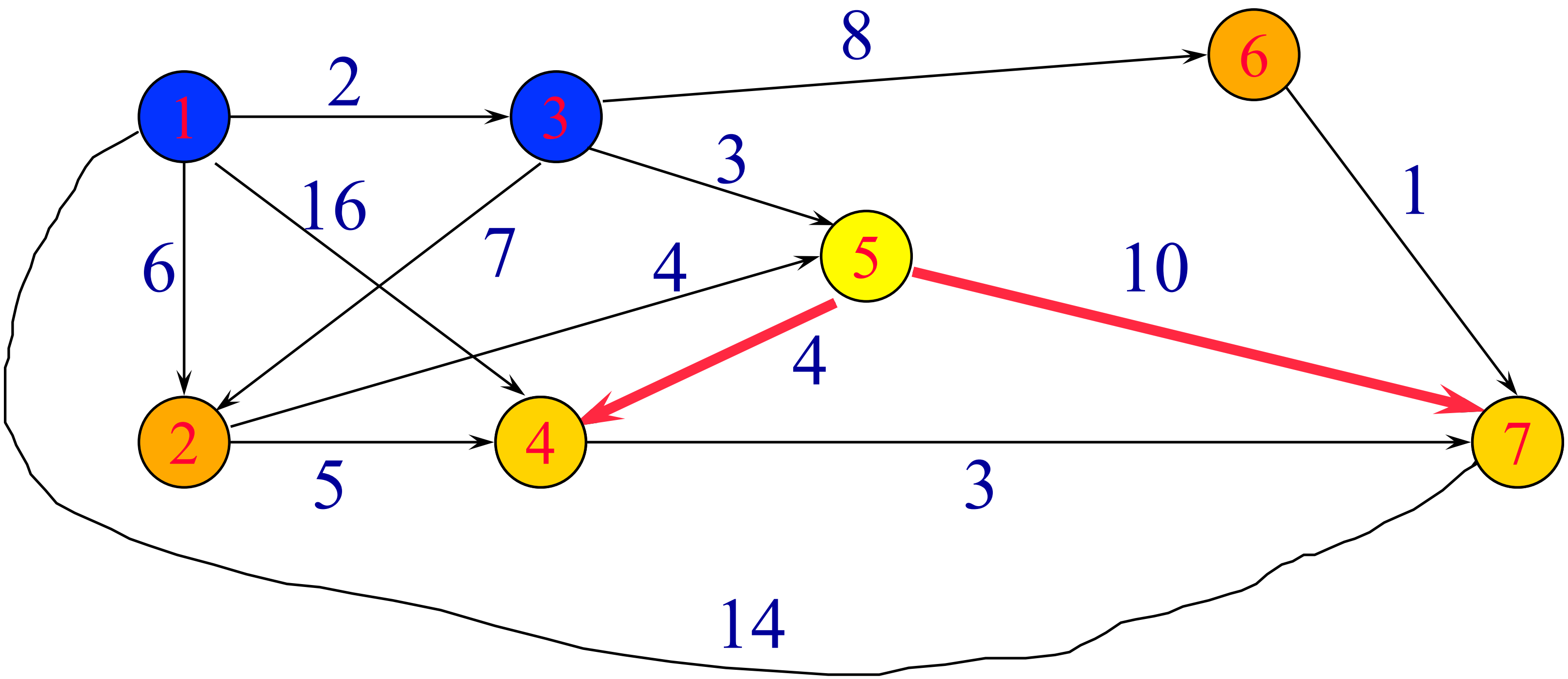
Greedy Single Source All Destinations

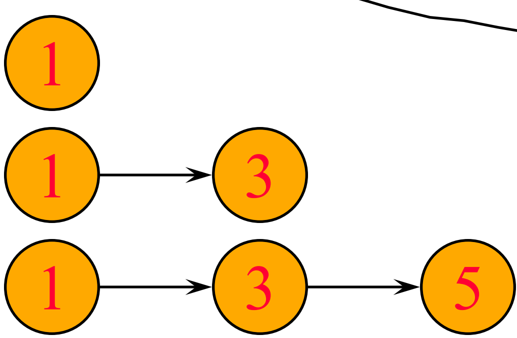




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	16	5	10	14
p	-	1	1	1	3	3	1

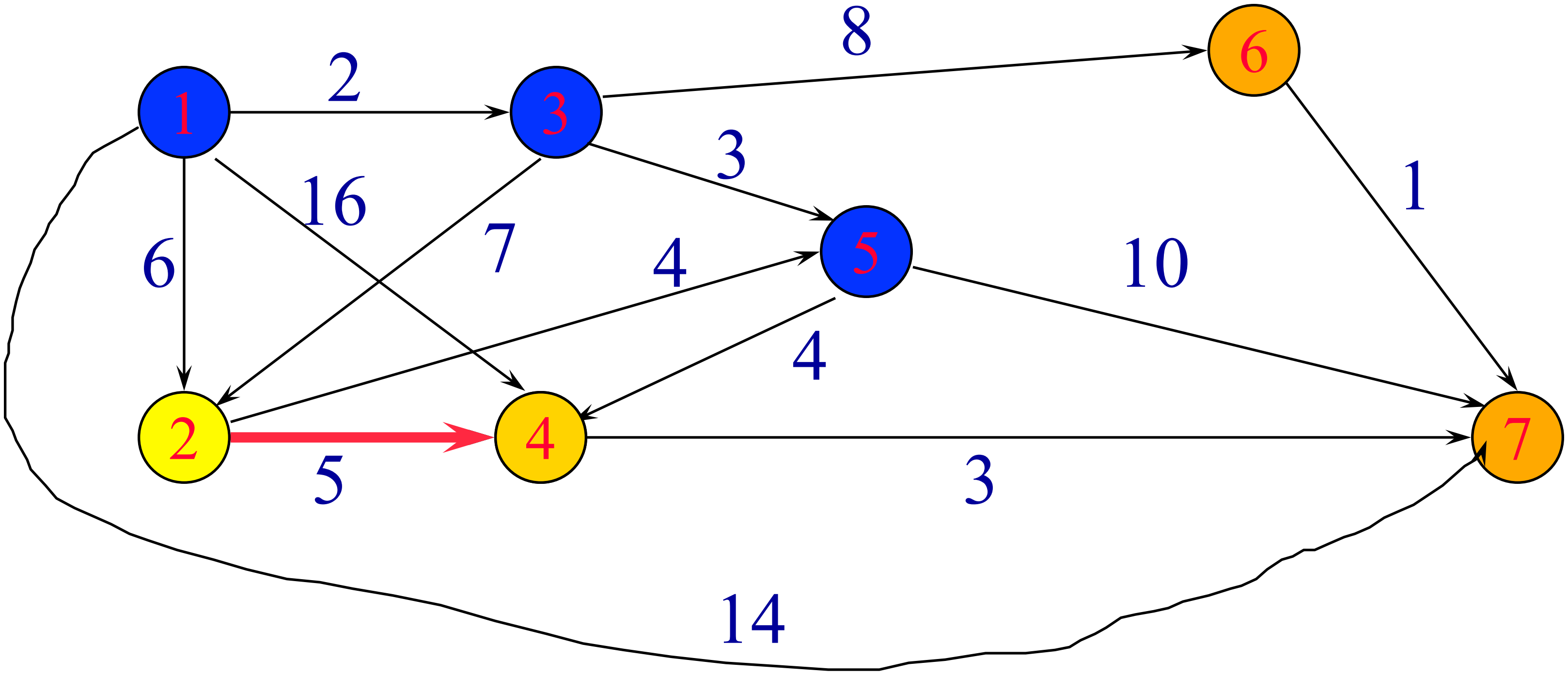
Greedy Single Source All Destinations

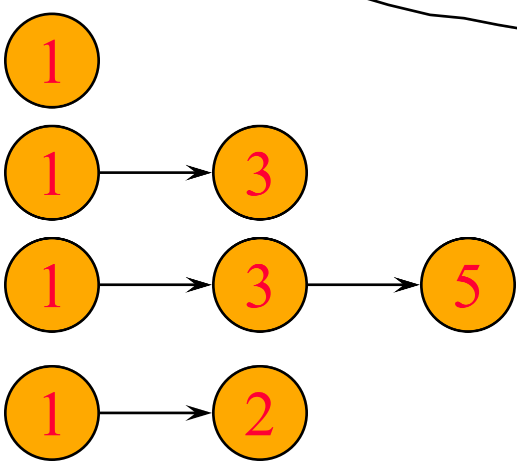




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

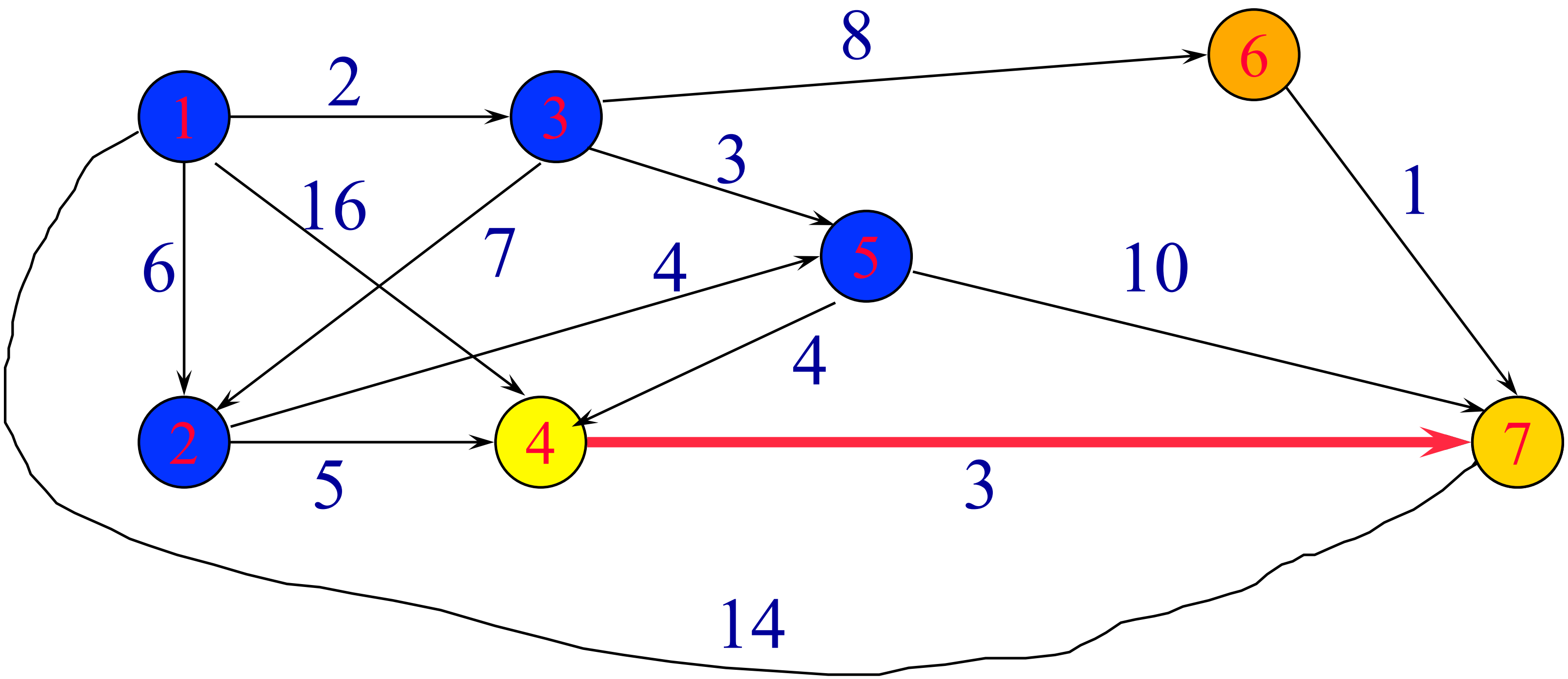
Greedy Single Source All Destinations

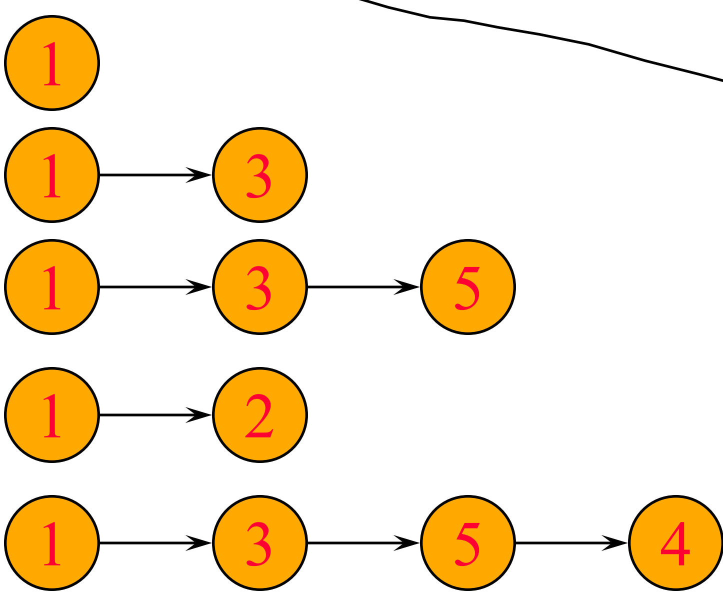




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	14
p	-	1	1	5	3	3	1

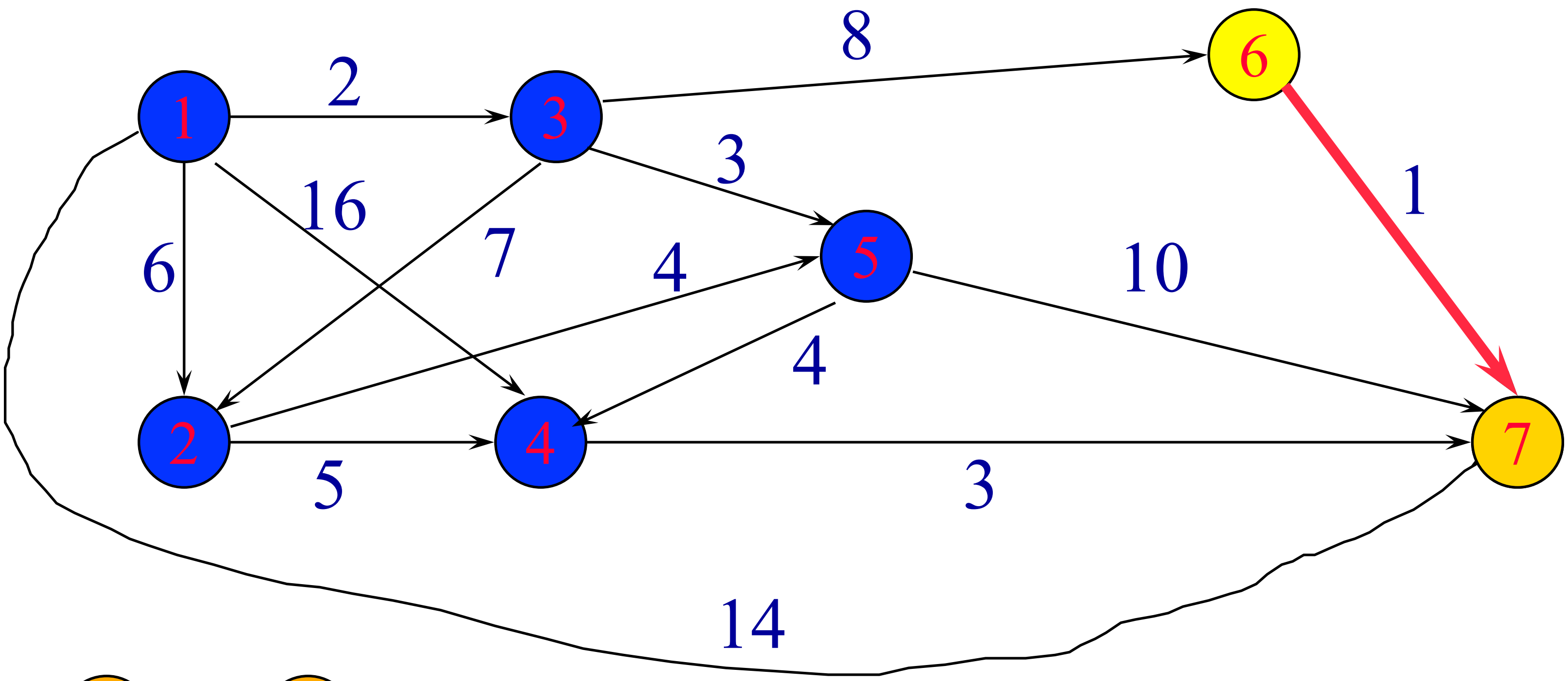
Greedy Single Source All Destinations

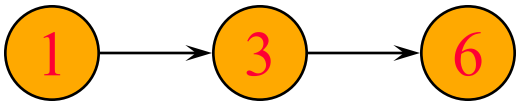




	[1]	[2]	[3]	[4]	[5]	[6]	[7]
d	0	6	2	9	5	10	12
p	-	1	1	5	3	3	4

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

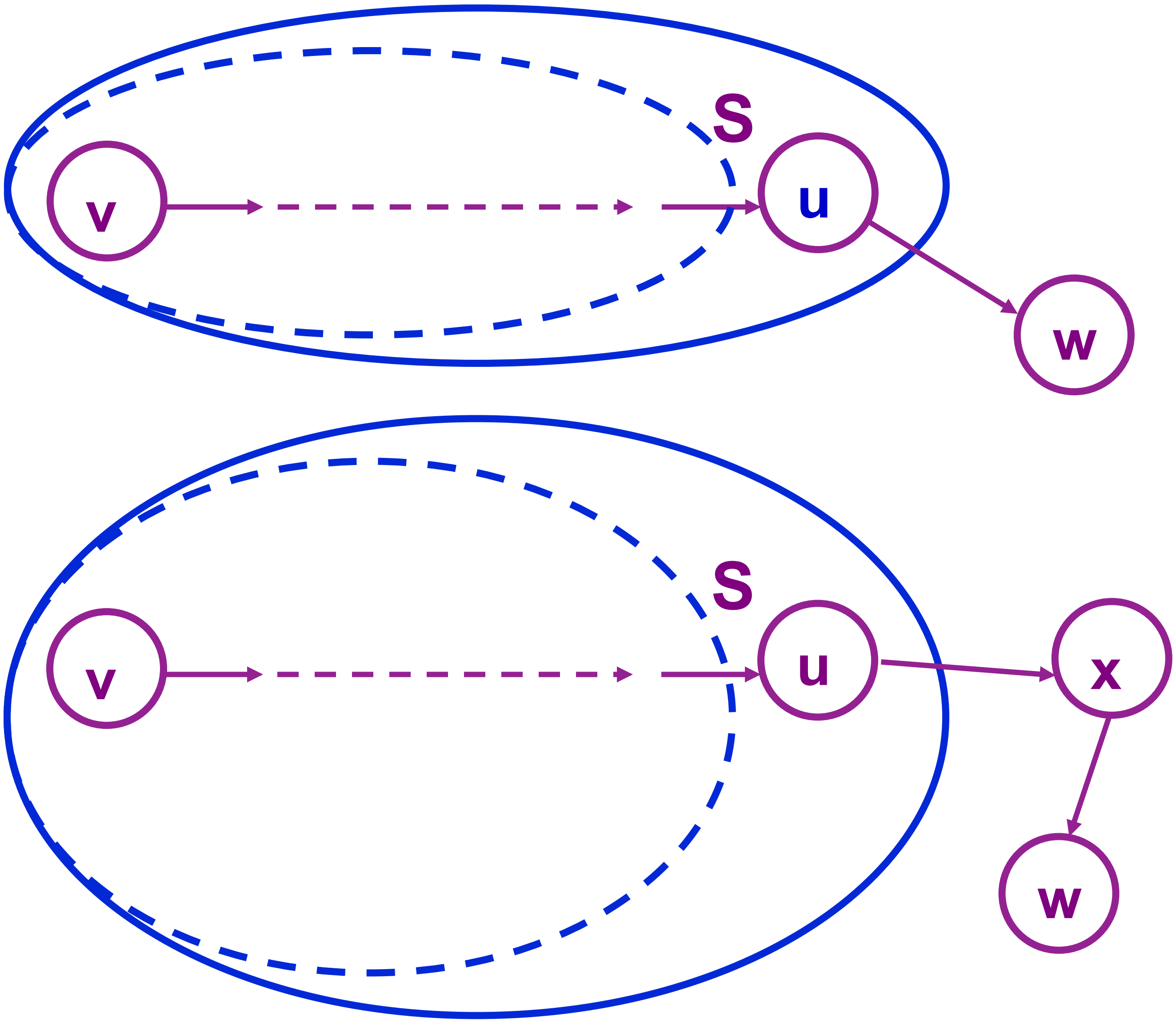
Path	Length
1	0
1 → 3	2
1 → 3 → 5	5
1 → 2	6
1 → 3 → 5 → 4	9
1 → 3 → 6	10
1 → 3 → 6 → 7	11

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

Single Source Single Destination

Terminate single source all destinations greedy algorithm as soon as shortest path to desired vertex has been generated.

Correctness



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.

```
void MatrixDigraph::ShortestPath( int n, int v){
    for (int i=0; i<n; i++) {
                L[i]=false; dist[i]=length[v][i];}
    L[v]=true;
    dist[v]=0;
    for (i=0; i<n-2; i++) { //determine n-1 paths from v
      int u=choose(n);  // IMPORTANT
      L[u]= true;
      for ( int w=0; w<n; w++)
        if (!L[w] && dist[u]+length[u][w]<dist[w])
            dist[w]=dist[u]+length[u][w];
    }
}
```

Complexity

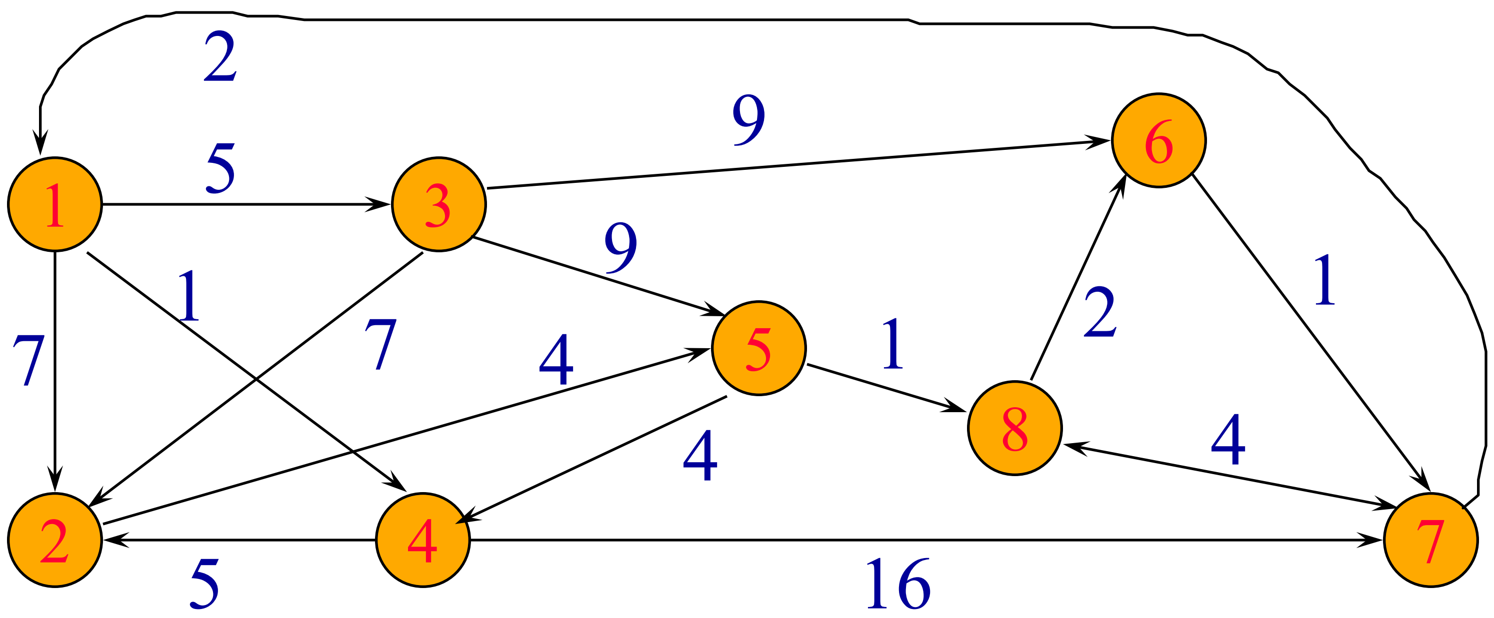
- $O(n)$ to select next destination vertex.
- $O(\text{out-degree})$ to update $d()$ and $p()$ values when adjacency lists are used.
- $O(n)$ to update $d()$ and $p()$ values when adjacency matrix is used.
- Selection and update done once for each vertex to which a shortest path is found.
- Total time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

Complexity

- When a min heap of d() values is used in place of the linear list L of reachable vertices, total time is $O((n+e) \log n)$, because $O(n)$ remove min operations and $O(e)$ change key (d() value) operations are done.
- When e is $O(n^2)$, using a min heap is worse than using a linear list.
- When a Fibonacci heap is used, the total time is $O(n \log n + e)$.

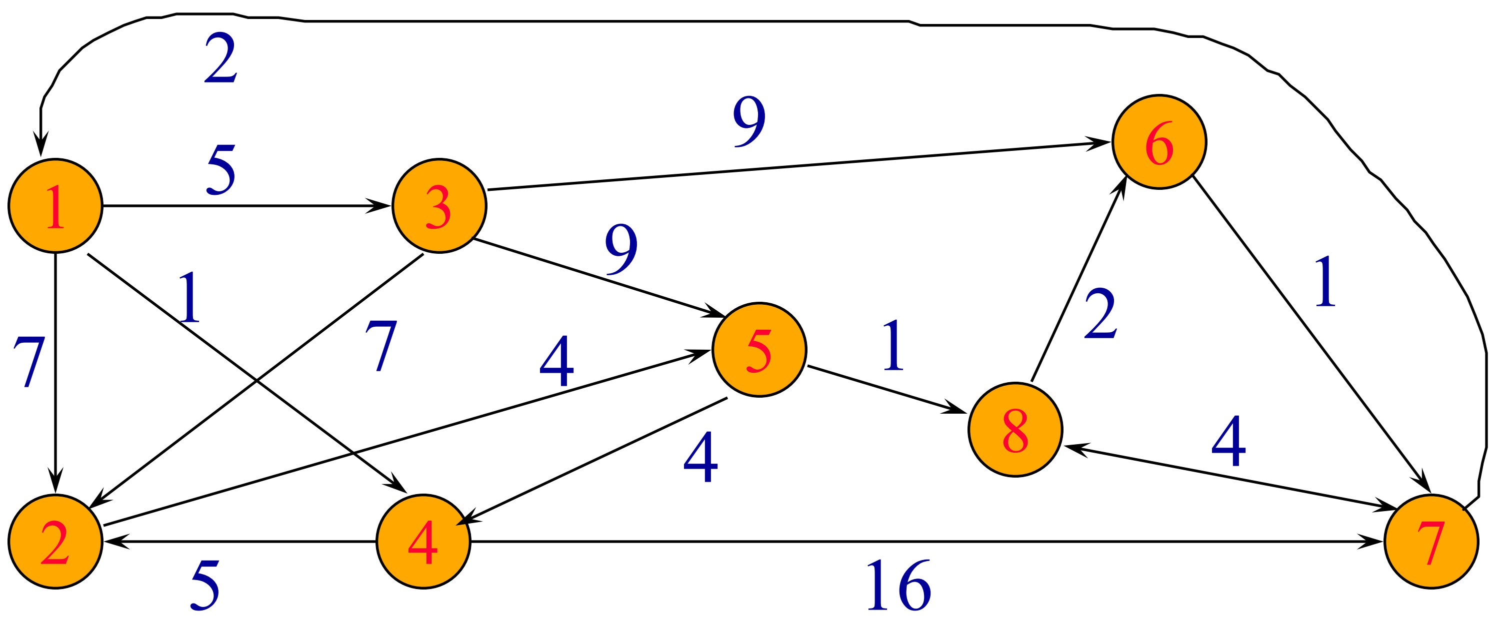
All-Pairs Shortest Paths

- Given an n-vertex directed weighted graph, find a shortest path from vertex i to vertex j for each of the n^2 vertex pairs (i,j).

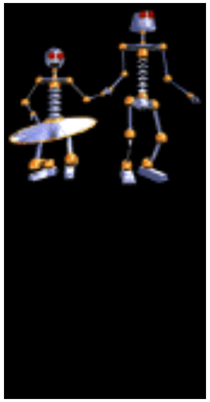


Dijkstra's Single Source Algorithm

- Use Dijkstra's algorithm n times, once with each of the n vertices as the source vertex.



Performance

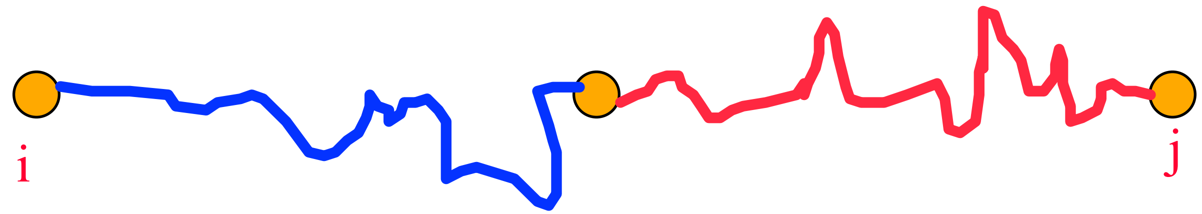


- Time complexity is $O(n^3)$ time.
- Works only when no edge has a cost < 0.

Dynamic Programming Solution

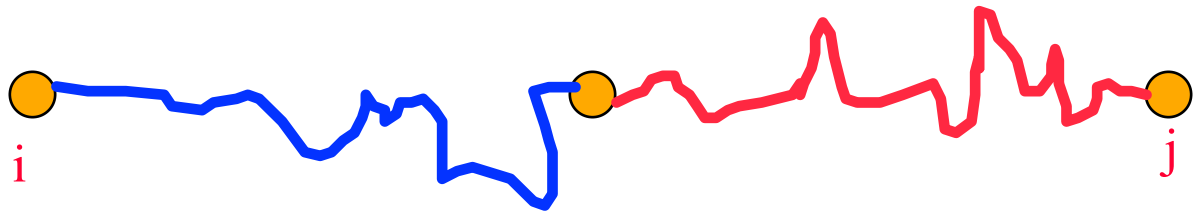
- Time complexity is Theta(n^3) time.
- Works so long as there is no cycle whose length is < 0.
- When there is a cycle whose length is < 0, some shortest paths aren't finite.
  - If vertex 1 is on a cycle whose length is -2, each time you go around this cycle once you get a 1 to 1 path that is 2 units shorter than the previous one.
- Simpler to code, smaller overheads.
- Known as Floyd's shortest paths algorithm.

Decision Sequence



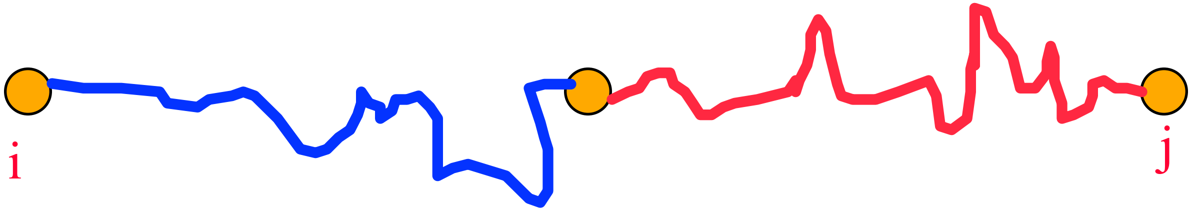
- First decide the highest intermediate vertex (i.e., largest vertex number) on the shortest path from i to j.
- If the shortest path is i, 2, 6, 3, 8, 5, 7, j, the first decision is that vertex 8 is an intermediate vertex on the shortest path and no intermediate vertex is larger than 8.
- Then decide the highest intermediate vertex on the path from i to 8, and so on.

Problem State



- (i,j,k) denotes the problem of finding the shortest path from vertex i to vertex j that has no intermediate vertex larger than k.
- (i,j,n) denotes the problem of finding the shortest path from vertex i to vertex j (with no restrictions on intermediate vertices).

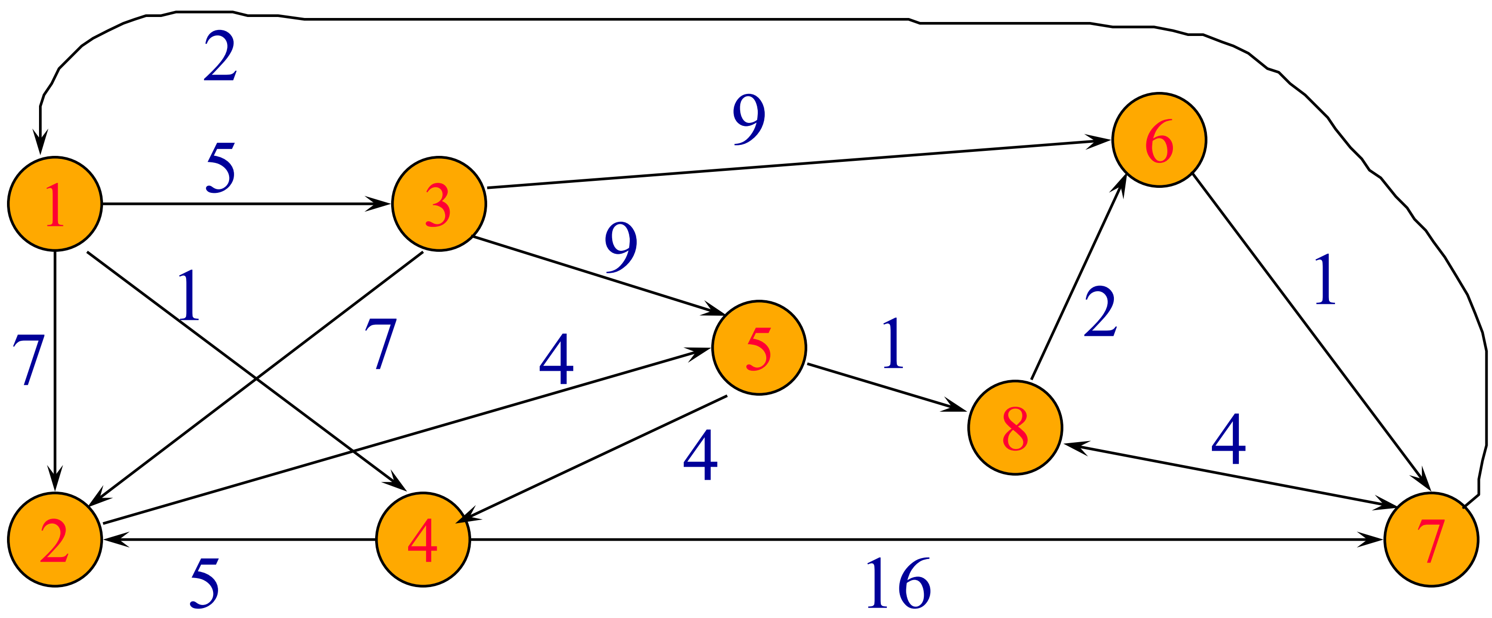
Cost Function



- Let c(i,j,k) be the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than k.

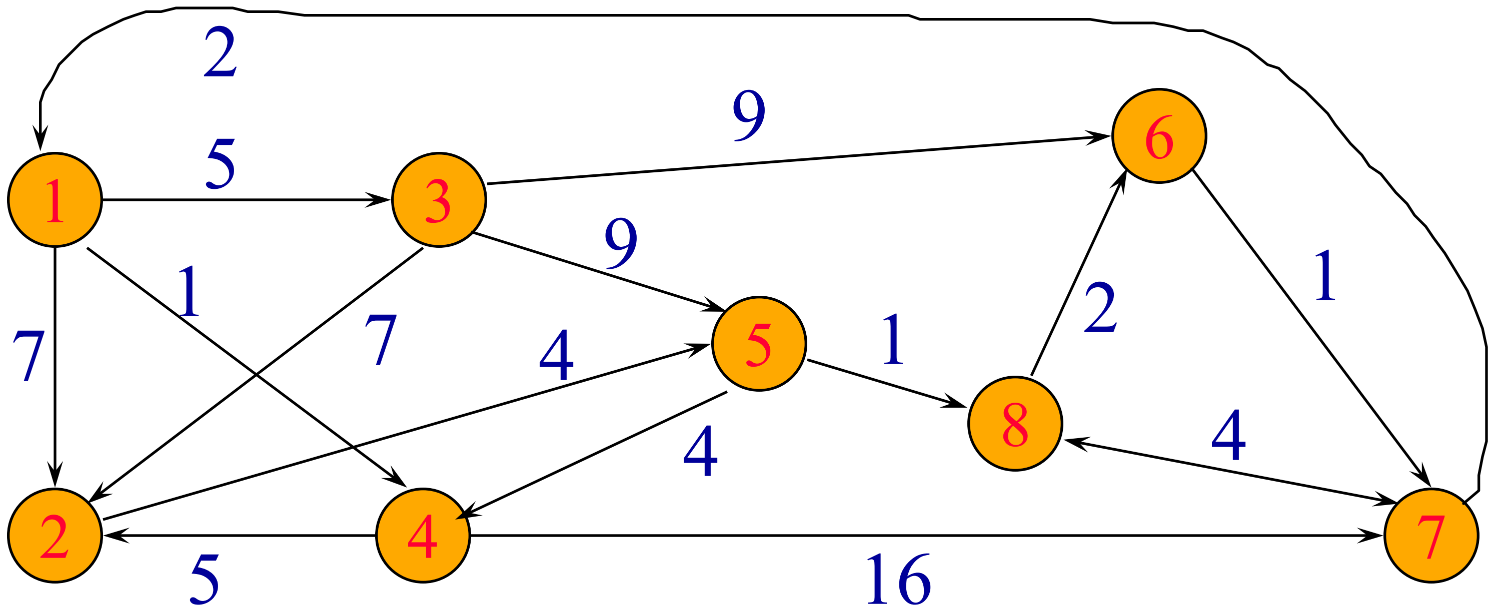
c(i,j,n)

- c(i,j,n) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than n.
- No vertex is larger than n.
- Therefore, c(i,j,n) is the length of a shortest path from vertex i to vertex j.



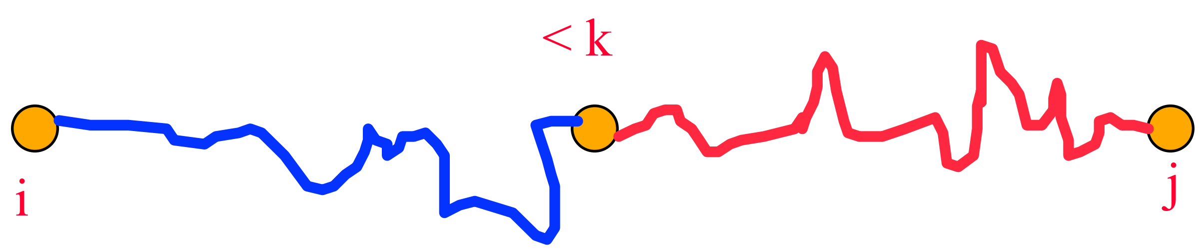
c(i,j,0)

- c(i,j,0) is the length of a shortest path from vertex i to vertex j that has no intermediate vertex larger than 0.
  - Every vertex is larger than 0.
  - Therefore, c(i,j,0) is the length of a single-edge path from vertex i to vertex j.



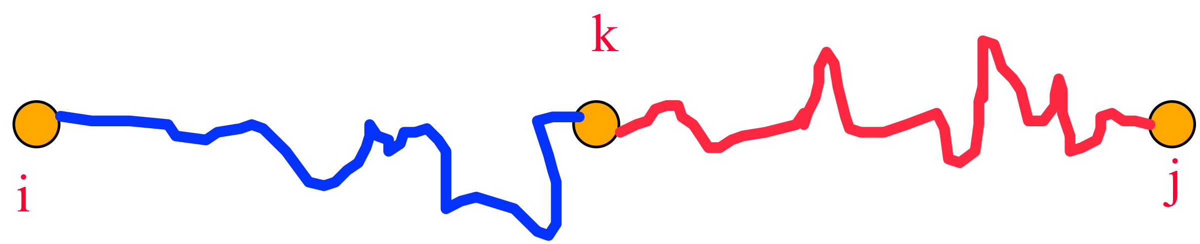
Recurrence For c(i,j,k), k > 0

- The shortest path from vertex i to vertex j that has no intermediate vertex larger than k may or may not go through vertex k.
- If this shortest path does not go through vertex k, the largest permissible intermediate vertex is k-1. So the path length is c(i,j,k-1).



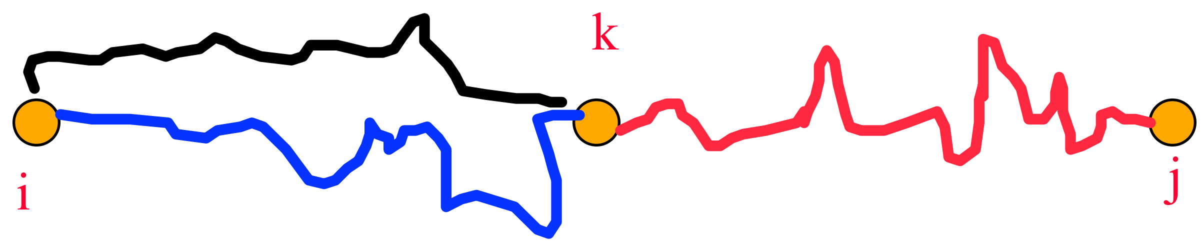
Recurrence For c(i,j,k)), k > 0

- Shortest path goes through vertex k.



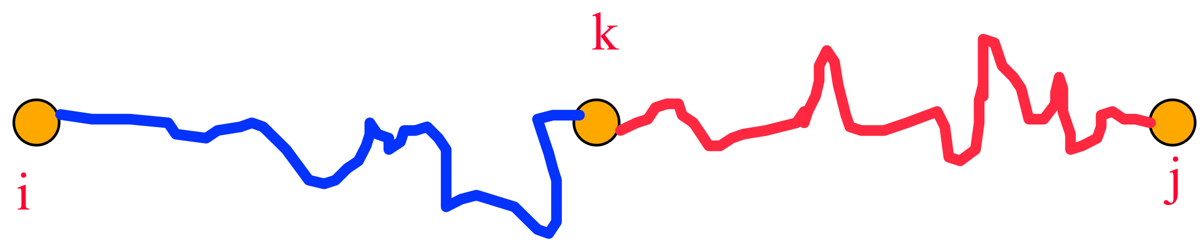
- We may assume that vertex k is not repeated because no cycle has negative length.
- Largest permissible intermediate vertex on i to k and k to j paths is k-1.

Recurrence For c(i,j,k)), k > 0



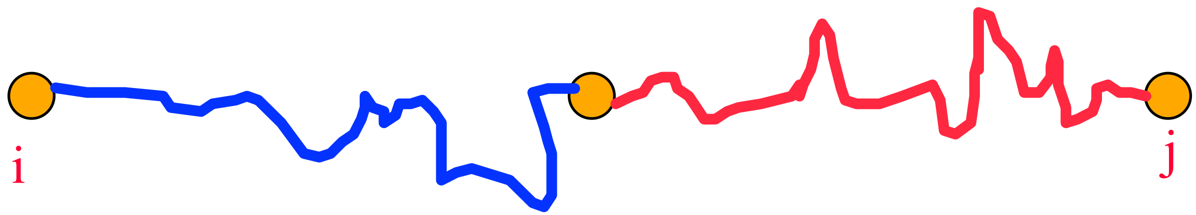
- i to k path must be a shortest i to k path that goes through no vertex larger than k-1.
- If not, replace current i to k path with a shorter i to k path to get an even shorter i to j path.

Recurrence For c(i,j,k)), k > 0



- Similarly, k to j path must be a shortest k to j path that goes through no vertex larger than k-1.
- Therefore, length of i to k path is c(i,k,k-1), and length of k to j path is c(k,j,k-1).
- So, c(i,j,k) = c(i,k,k-1) + c(k,j,k-1).

Recurrence For c(i,j,k)), k > 0



- Combining the two equations for c(i,j,k), we get c(i,j,k) = min{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)}.
- We may compute the c(i,j,k)s in the order k = 1, 2, 3, …, n.

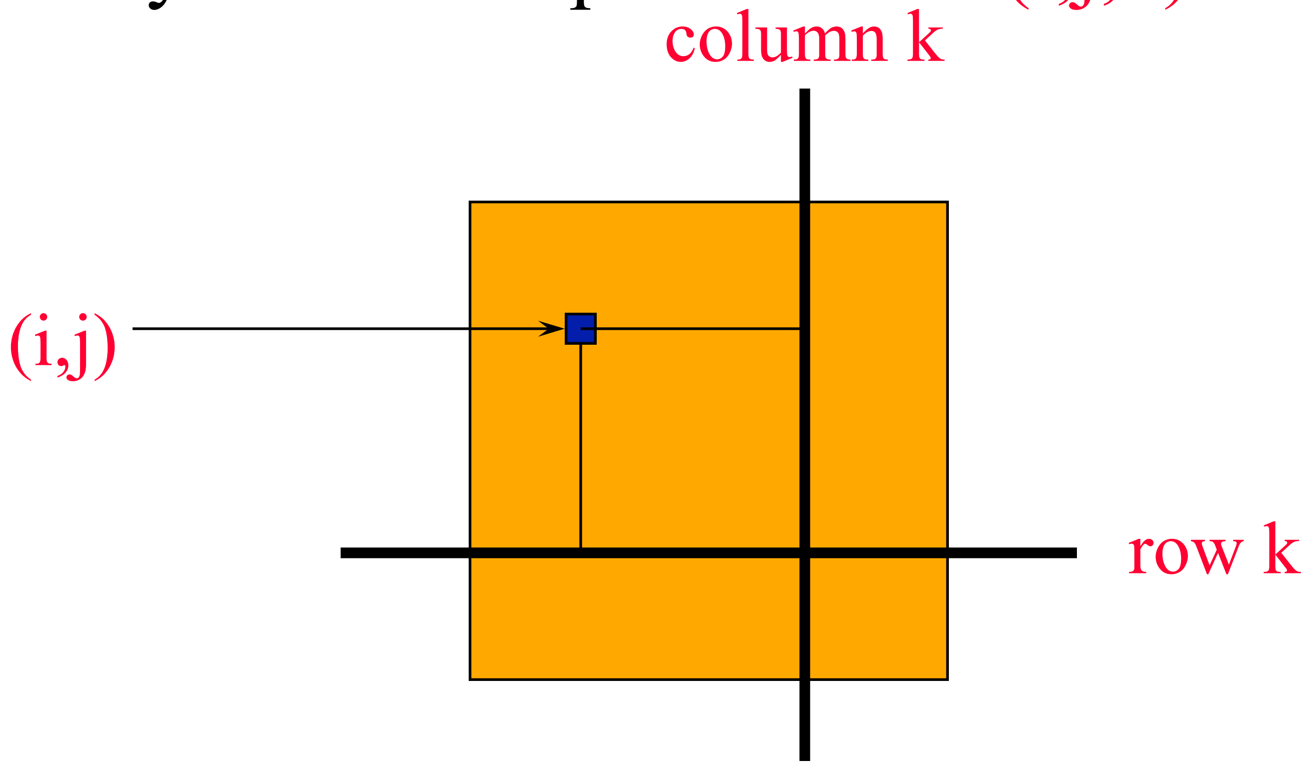
Floyd's Shortest Paths Algorithm

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      c(i,j,k) = min{c(i,j,k-1),
                     c(i,k,k-1) + c(k,j,k-1)};
```

- Time complexity is $O(n^3)$.
- More precisely $Theta(n^3)$.
- $Theta(n^3)$ space is needed for c(*,*,*).

Space Reduction

- $c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\}$
- When neither i nor j equals k, $c(i,j,k-1)$ is used only in the computation of $c(i,j,k)$.



- So $c(i,j,k)$ can overwrite $c(i,j,k-1)$.

Space Reduction

- $c(i,j,k) = \min\{c(i,j,k-1), c(i,k,k-1) + c(k,j,k-1)\}$
- When i equals k, $c(i,j,k-1)$ equals $c(i,j,k)$.
  - $$\begin{aligned} c(k,j,k) &= \min\{c(k,j,k-1), c(k,k,k-1) + c(k,j,k-1)\} \\ &= \min\{c(k,j,k-1), 0 + c(k,j,k-1)\} \\ &= c(k,j,k-1) \end{aligned}$$
- So, when i equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- Similarly when j equals k, $c(i,j,k)$ can overwrite $c(i,j,k-1)$.
- So, in all cases $c(i,j,k)$ can overwrite $c(i,j,k-1)$.

Floyd's Shortest Paths Algorithm

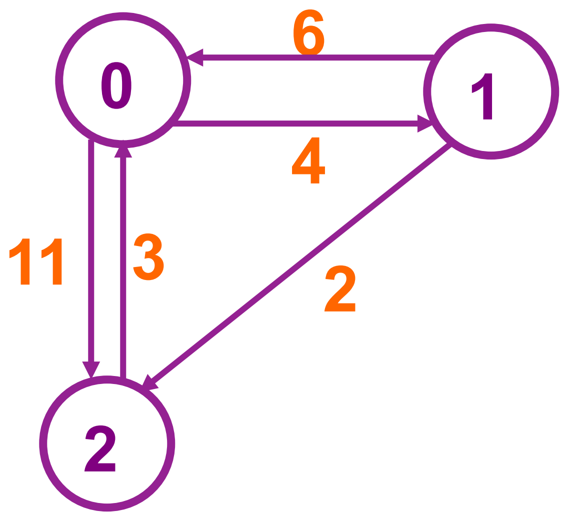
```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      c(i,j) = min{c(i,j), c(i,k) + c(k,j)};
```

- Initially, $c(i,j) = c(i,j,0)$.
- Upon termination, $c(i,j) = c(i,j,n)$.
- Time complexity is $\Theta(n^3)$.
- $\Theta(n^2)$ space is needed for $c(*,*)$.

Building The Shortest Paths

- Let kay(i,j) be the largest vertex on the shortest path from i to j.
- Initially, kay(i,j) = 0 (shortest path has no intermediate vertex).

```
for (int k = 1; k <= n; k++)
  for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
      if (c(i,j) > c(i,k) + c(k,j))
       {kay(i,j) = k; c(i,j) = c(i,k) + c(k,j);}
```



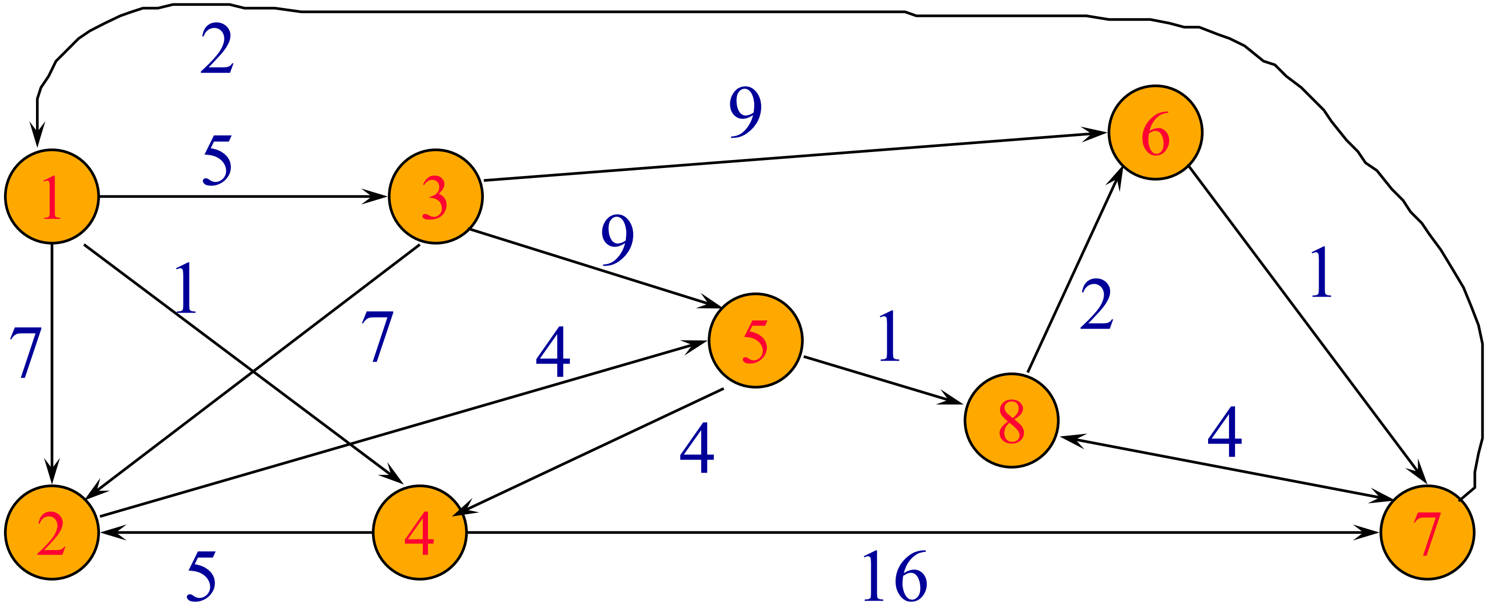
A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{0}	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A^{1}	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A^{2}	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Example



-	7	5	1	-	-	-	-
-	-	-	-	4	-	-	-
-	7	-	-	9	9	-	-
-	5	-	-	-	-	16	-
-	-	-	4	-	-	-	1
-	-	-	-	-	-	1	-
2	-	-	-	-	-	-	4
-	-	-	-	-	2	4	-

Initial Cost Matrix

$c(*,*) = c(*,*,0)$

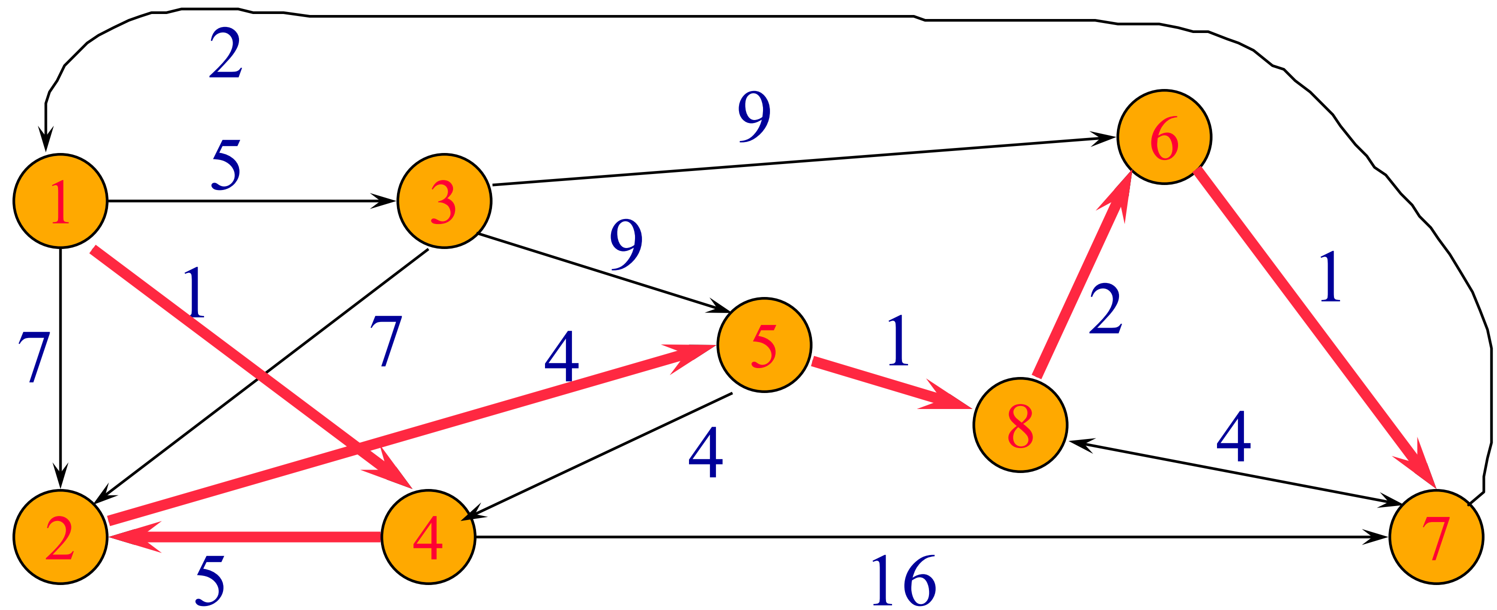
Final Cost Matrix c(*,*) = c(*,*,n)

0	6	5	1	10	13	14	11
10	0	15	8	4	7	8	5
12	7	0	13	9	9	10	10
15	5	20	0	9	12	13	10
6	9	11	4	0	3	4	1
3	9	8	4	13	0	1	5
2	8	7	3	12	6	0	4
5	11	10	6	15	2	3	0

kay Matrix

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

Shortest Path



Shortest path from 1 to 7.

Path length is 14.

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.
- kay(1,7) = 8

$$1 \longrightarrow 8 \longrightarrow 7$$

- kay(1,8) = 5

$$1 \longrightarrow 5 \longrightarrow 8 \longrightarrow 7$$

- kay(1,5) = 4

$$1 \longrightarrow 4 \longrightarrow 5 \longrightarrow 8 \longrightarrow 7$$

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

$1 \rightarrow 4 \rightarrow 5 \rightarrow 8 \rightarrow 7$

- kay(1,4) = 0

$1\ 4 \rightarrow 5 \rightarrow 8 \rightarrow 7$

- kay(4,5) = 2

$1\ 4 \rightarrow 2 \rightarrow 5 \rightarrow 8 \rightarrow 7$

- kay(4,2) = 0

$1\ 4\ 2 \rightarrow 5 \rightarrow 8 \rightarrow 7$

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

$1\ 4\ 2 \rightarrow 5 \rightarrow 8 \rightarrow 7$

- kay(2,5) = 0

$1\ 4\ 2\ 5 \rightarrow 8 \rightarrow 7$

- kay(5,8) = 0

$1\ 4\ 2\ 5\ 8 \rightarrow 7$

- kay(8,7) = 6

$1\ 4\ 2\ 5\ 8 \rightarrow 6 \rightarrow 7$

Build A Shortest Path

0	4	0	0	4	8	8	5
8	0	8	5	0	8	8	5
7	0	0	5	0	0	6	5
8	0	8	0	2	8	8	5
8	4	8	0	0	8	8	0
7	7	7	7	7	0	0	7
0	4	1	1	4	8	0	0
7	7	7	7	7	0	6	0

- The path is 1 4 2 5 8 6 7.

  1 4 2 5 8 → 6 → 7

- kay(8,6) = 0

  1 4 2 5 8 6 → 7

- kay(6,7) = 0

  1 4 2 5 8 6 7

Output A Shortest Path

```
void outputPath(int i, int j)
{// does not output first vertex (i) on path
  if (i == j) return;
  if (kay[i][j] == 0)  // no intermediate vertices on path
          print(j + " ");
  else {// kay[i][j] is an intermediate vertex on the path
          outputPath(i, kay[i][j]);
          outputPath(kay[i][j], j);
        }
}
```

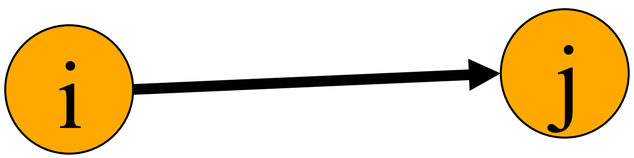
Time Complexity Of outputPath 

O(number of vertices on shortest path)

Exercises: P372-1, P373-2, 5, P375-17

Directed Graphs Usage

- Directed graphs are often used to represent order -dependent tasks
- Cannot start a task before another task finishes
- Model this task dependent constraint using *arcs*
- An *arc (i,j)* means *task j* cannot start until *task i* is finished



Task **j** cannot start until task **i** is finished

- For the system not to hang, the graph must be acyclic.

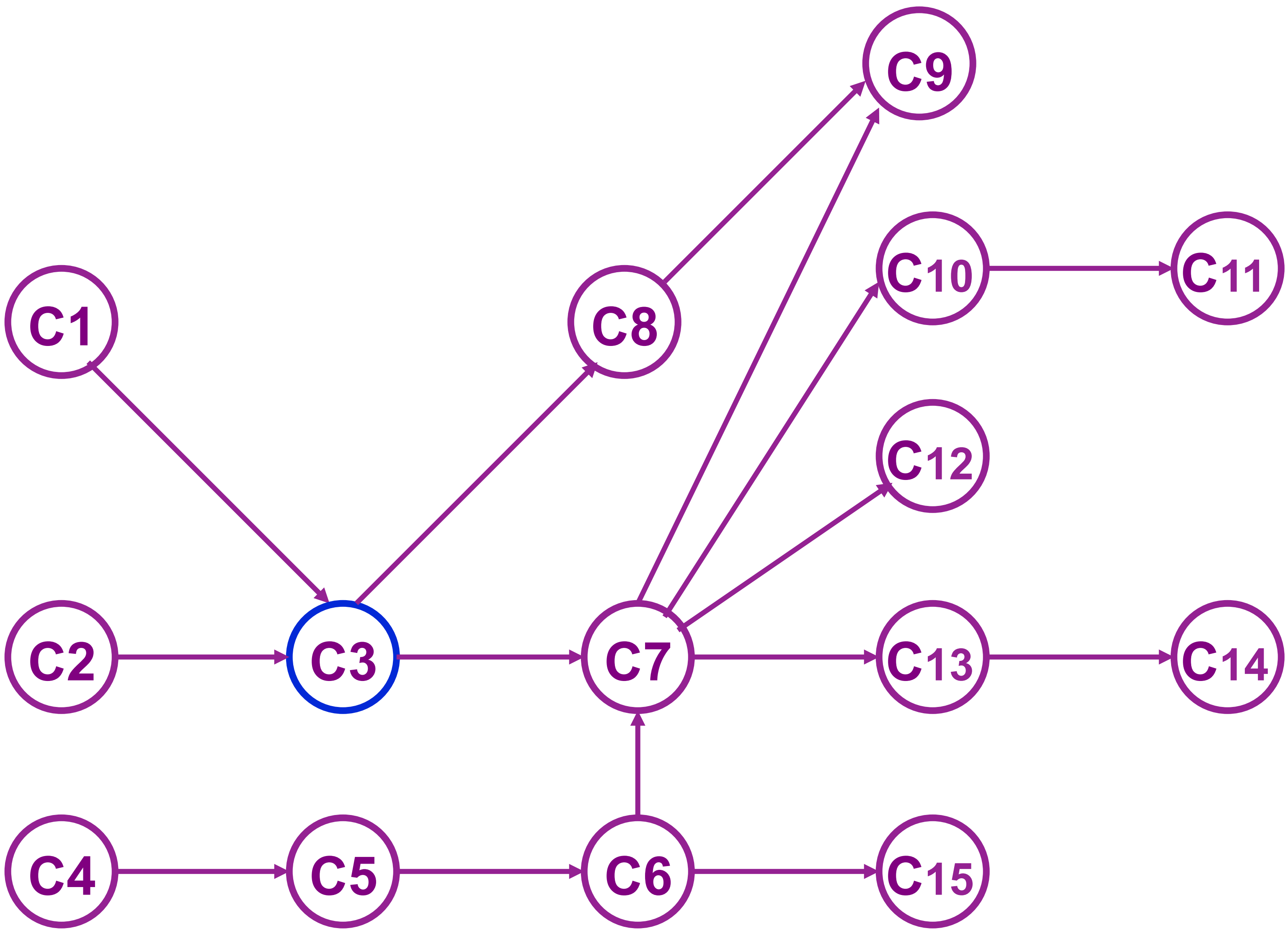
Activity Networks

Activity-on-Vertex (AOV) Networks

- A directed graph G
- Vertices
  - Tasks or activities
- Edges
  - Precedence relations between tasks

Course-No.	Course-Name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	none
C3	Data Structures	C1, C2
C4	Calculus I	none
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating System	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithm	C13
C15	Numerical Analysis	C5

AOV



Definitions

- Vertex i in an AOV network G is a predecessor of j iff there is a directed path from i to j. If <i, j> is an edge in G then i is an immediate predecessor of j and j immediate successor of i.
- A precedence relation that is both transitive and irreflexive is a partial order.
- A directed graph with no cycle is an acyclic graph.

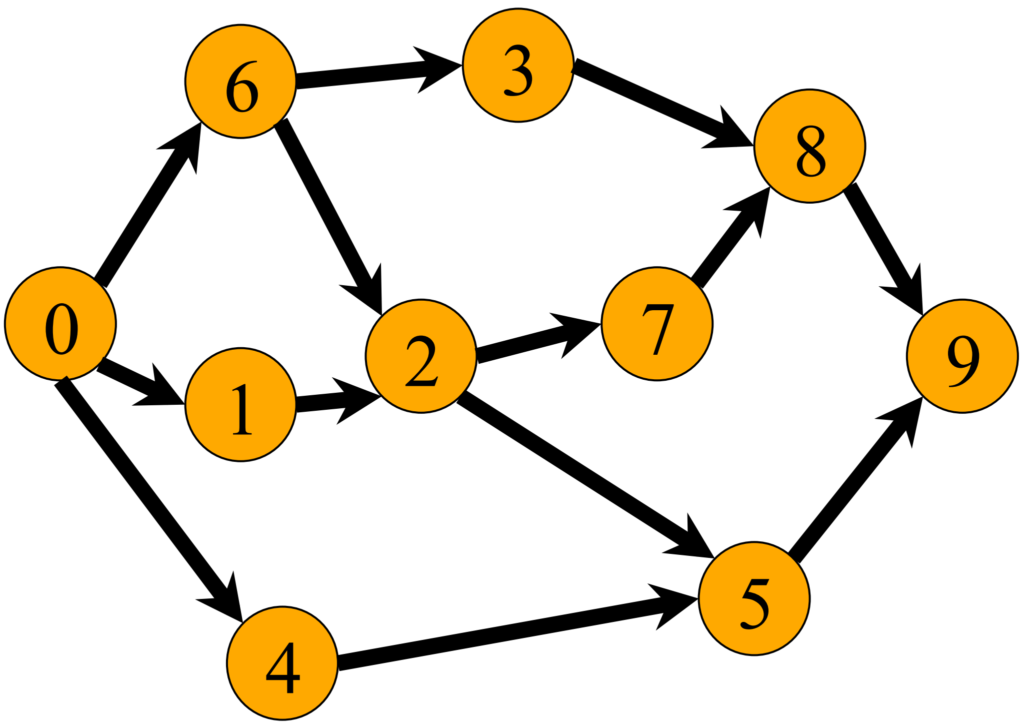
Problem

- Given an AOV network G
  - whether or not it is irreflexive, i.e., acyclic.
- Solution
  - Generate the **topological order** of vertices in G.

Topological order

- A topological order is a linear ordering of vertices of a graph
  - For any two vertices i and j, if i is a predecessor of j in the network, then i precedes j in the linear ordering
- It can be thought of as a way to linearly order the vertices so that the linear order respects the ordering relations implied by the arcs(edges)

Topological order



For example:

0, 1, 2, 5, 9

0, 4, 5, 9

0, 6, 3, 7 ?

Whether a Digraph is acyclic?

- Same to:
  - Does every task can be executed?
- Idea:
  - Tasks have no predecessor can be executed
  - Tasks with all predecessors finished can be executed
  - Starting point must have zero indegree!
  - If it doesn't exist, the graph would not be acyclic

Whether a Digraph is acyclic?

- Vertices with zero *indegree*
  - Can start right away
  - Output it first in the linear order
- A vertex *i* is output
  - Its outgoing arcs *(i, j)* are no longer useful
  - Since tasks *j* does not need to wait for *i* anymore
    - Remove all *i*'s outgoing arcs
- Vertex *i* removed
  - new graph is still a directed acyclic graph
- Repeat step 1-2 until no 0-indegree vertex left

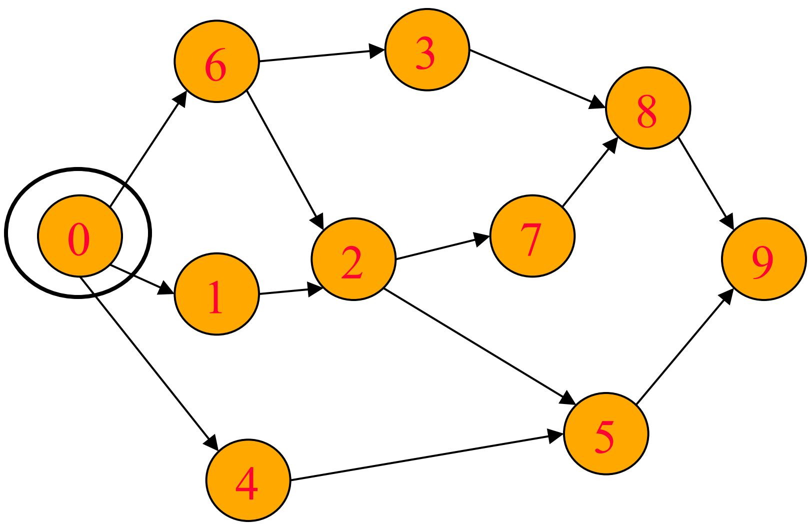
Topological Sort

Algorithm *TSort*(G)
Input: a directed acyclic graph G
Output: a topological ordering of vertices

```
initialize Q to be an empty queue;
for each vertex v
    do if indegree(v) = 0
        then enqueue(Q, v);
while Q is non-empty
    do v := dequeue(Q);
        output v;
        for each arc (v, w)
            do indegree(w) = indegree(w) − 1;
                if indegree(w) = 0
                    then enqueue(w);
```

The running time is $O(n+m)$.

Example



0	6, 1, 4
1	2
2	7, 5
3	8
4	5
5	9
6	3, 2
7	8
8	9
9	

Indegree

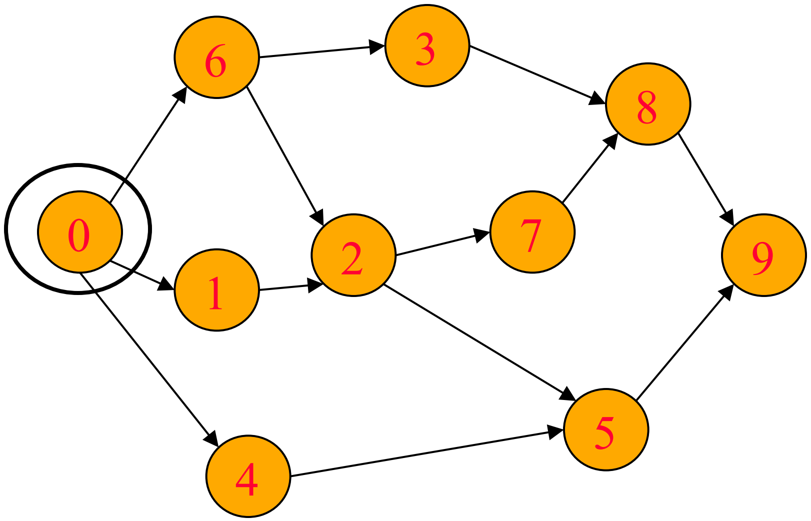
0	0
1	1
2	2
3	1
4	1
5	2
6	1
7	1
8	2
9	2

start

Q = { 0 }

OUTPUT: 0

Example



0	6, 1, 4
1	2
2	7, 5
3	8
4	5
5	9
6	3, 2
7	8
8	9
9	

Indegree

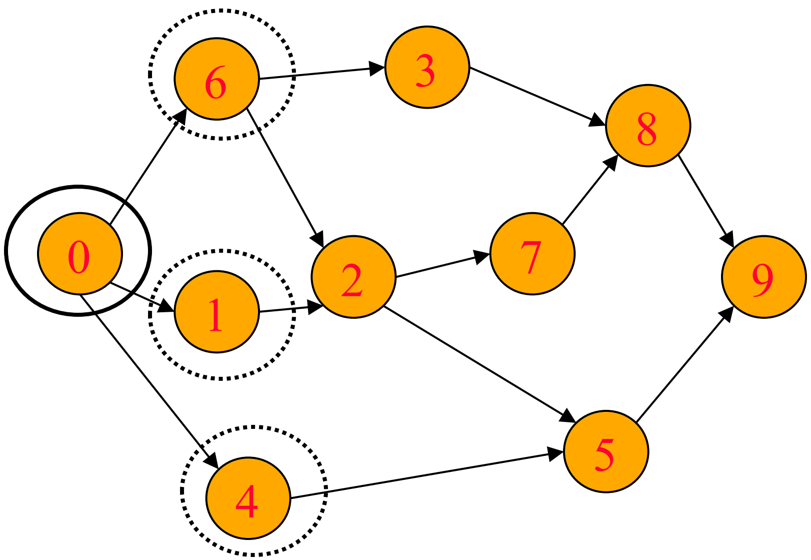
0	0	
1	1	-1
2	2	
3	1	
4	1	-1
5	2	
6	1	-1
7	1	
8	2	
9	2	

Dequeue 0 Q = { }
-> remove 0's arcs – adjust indegrees of neighbors

Decrement 0's neighbors

OUTPUT:

Example



0	6, 1, 4
1	2
2	7, 5
3	8
4	5
5	9
6	3, 2
7	8
8	9
9	

Indegree

0	0
1	0
2	2
3	1
4	0
5	2
6	0
7	1
8	2
9	2

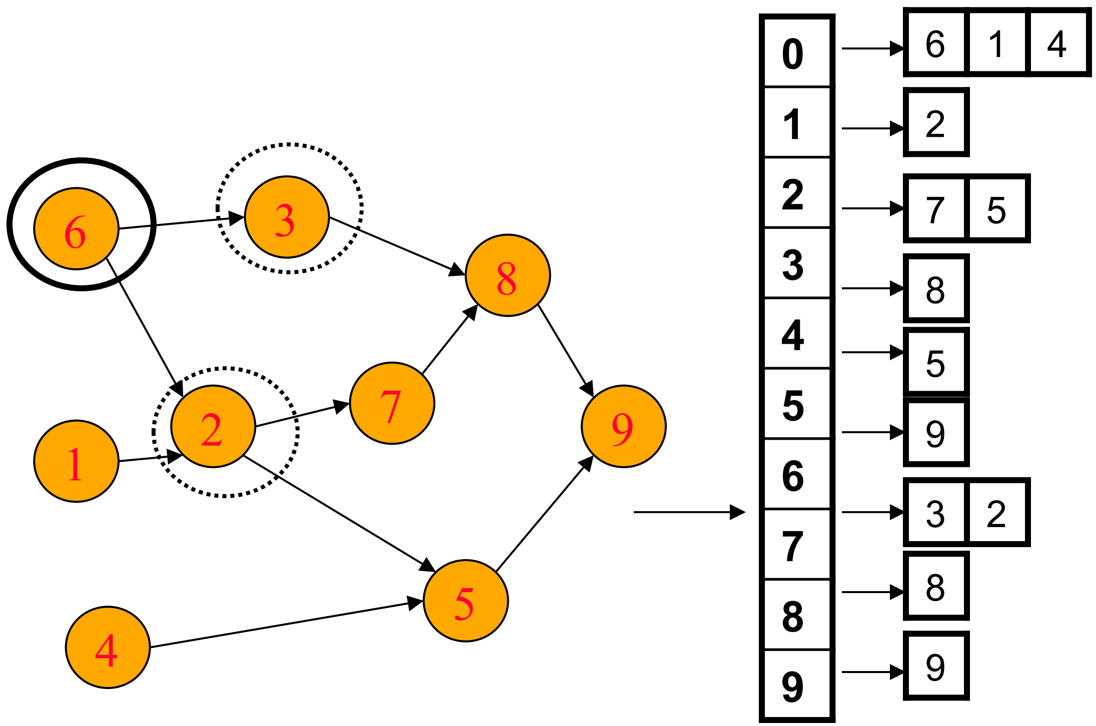
Dequeue 0 Q = { 6, 1, 4 }

Enqueue all starting points

Enqueue all new start points

OUTPUT: 0

Example



Indegree

0	0	
1	0	
2	2	-1
3	1	-1
4	0	
5	2	
6	0	
7	1	
8	2	
9	2	

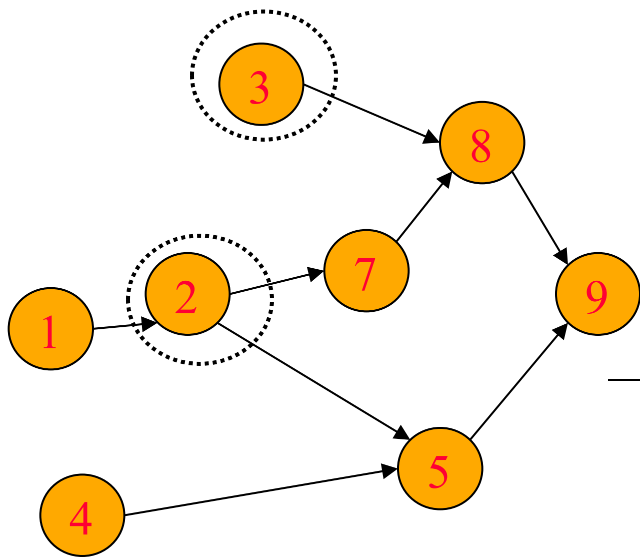
Dequeue 6 Q = { 1, 4 }

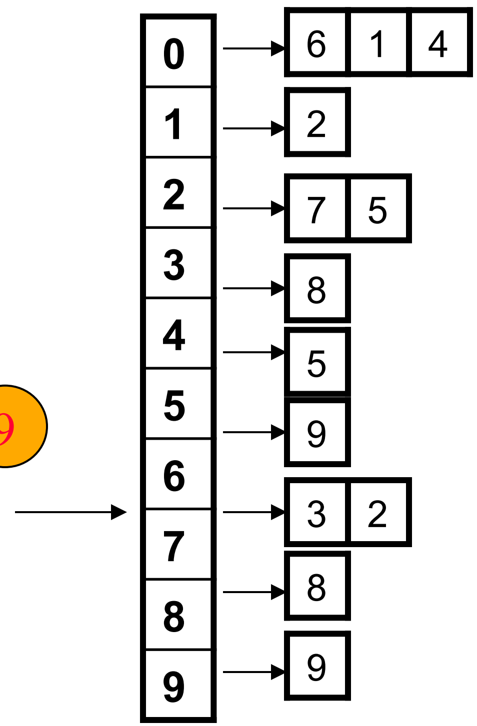
Remove arcs .. Adjust indegrees of neighbors

Adjust neighbors indegree

OUTPUT: 0 6

Example





Indegree

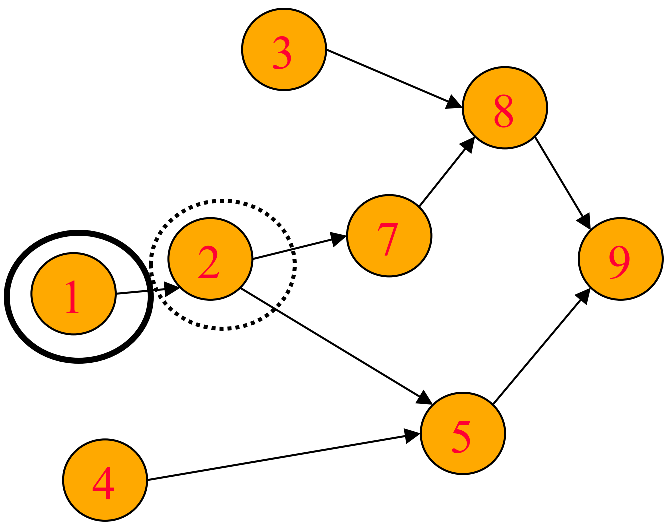
0	0
1	0
2	1
3	0
4	0
5	2
6	0
7	1
8	2
9	2

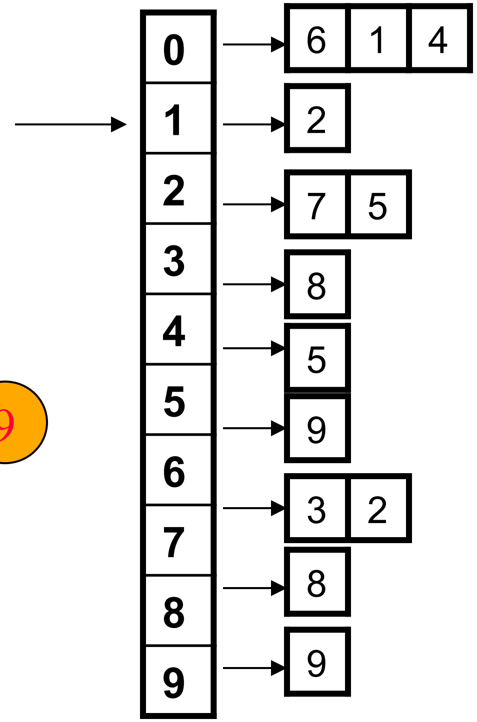
Deque 6 Q = { 1, 4, 3 }
Enqueue 3

Enqueue new start

OUTPUT: 0 6

Example





Indegree

0	0
1	0
2	1
3	0
4	0
5	2
6	0
7	1
8	2
9	2

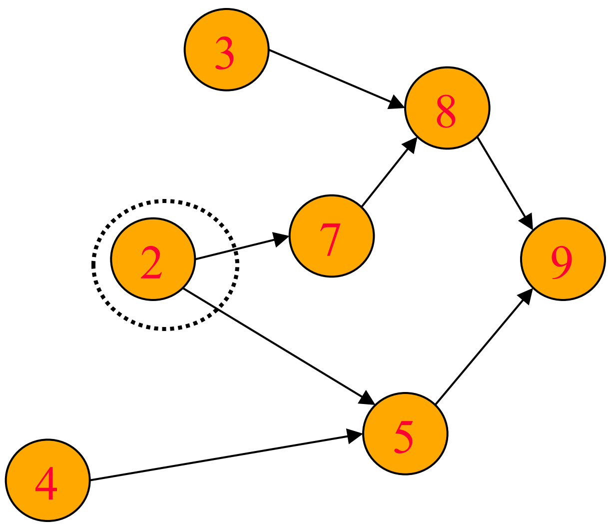
-1

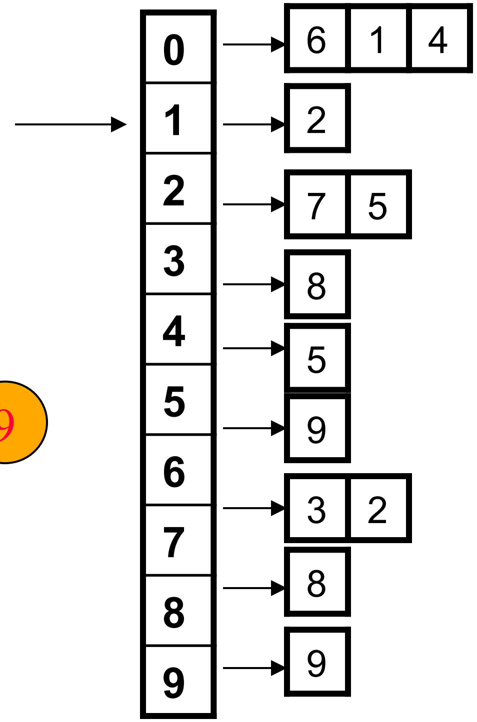
Dequeue 1 Q = { 4, 3 }
Adjust indegrees of neighbors

Adjust neighbors of 1

OUTPUT: 0 6 1

Example





Indegree

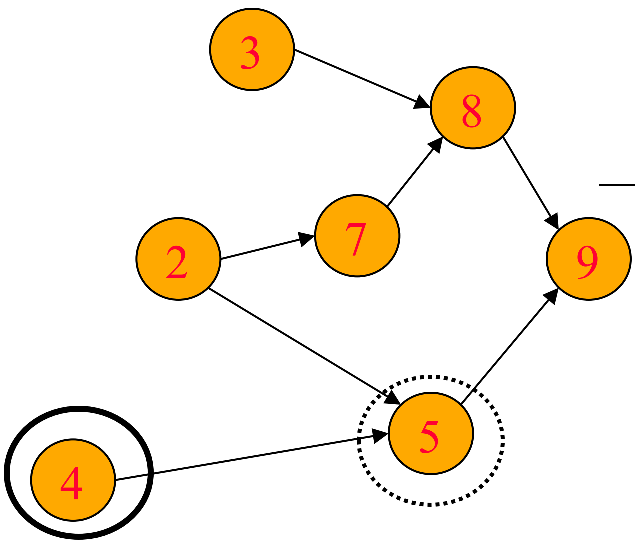
0	0
1	0
2	0
3	0
4	0
5	2
6	0
7	1
8	2
9	2

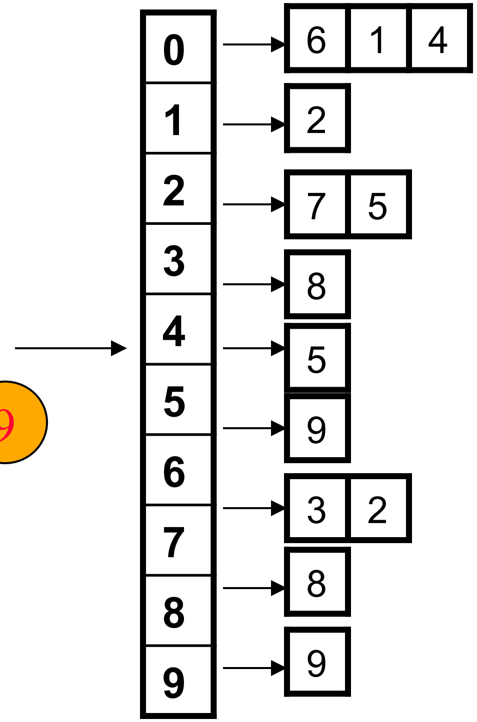
Deque 1 Q = { 4, 3, 2 }
Enqueue 2

Enqueue new starting points

OUTPUT: 0 6 1

Example





Indegree

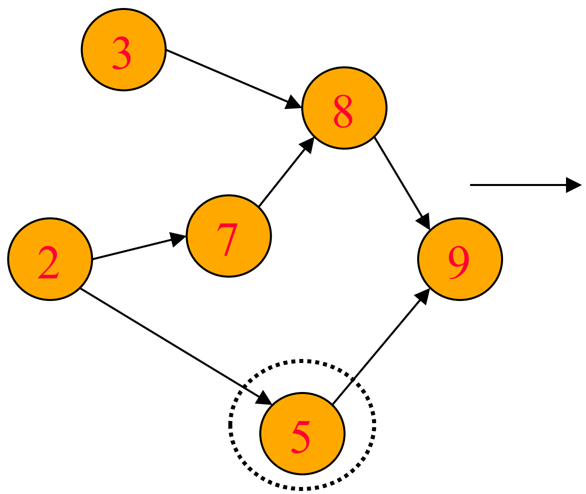
~~0~~	~~0~~
~~1~~	~~0~~
~~2~~	~~0~~
~~3~~	~~0~~
4	0
5	2 -1
~~6~~	~~0~~
7	1
8	2
9	2

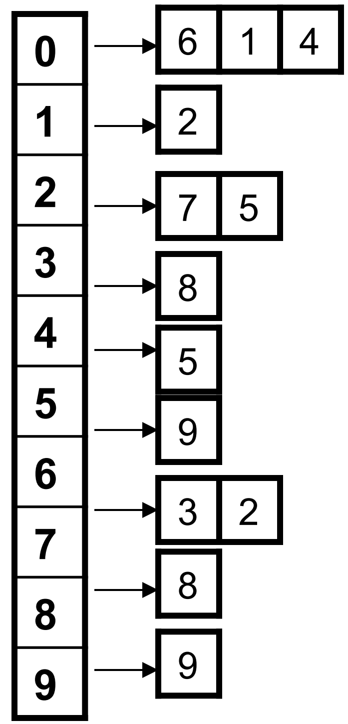
Deque 4 Q = { 3, 2 }
Adjust indegrees of neighbors

Adjust 4's neighbors

OUTPUT: 0 6 1 4

Example





Indegree

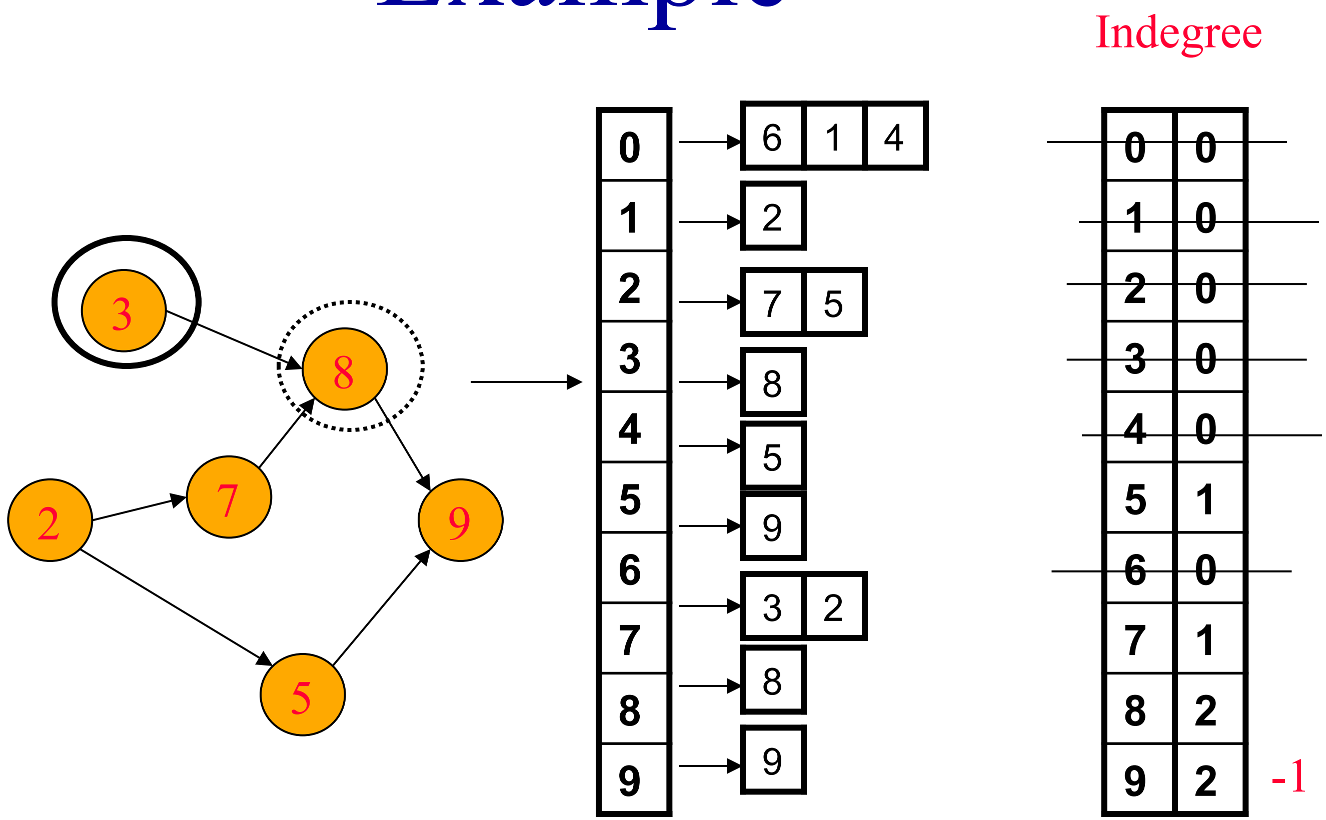
~~0~~	~~0~~
~~1~~	~~0~~
~~2~~	~~0~~
~~3~~	~~0~~
~~4~~	~~0~~
5	1
~~6~~	~~0~~
7	1
8	2
9	2

Deqeue 4 Q = { 3, 2 }
No new start points found

NO new start points

OUTPUT: 0 6 1 4

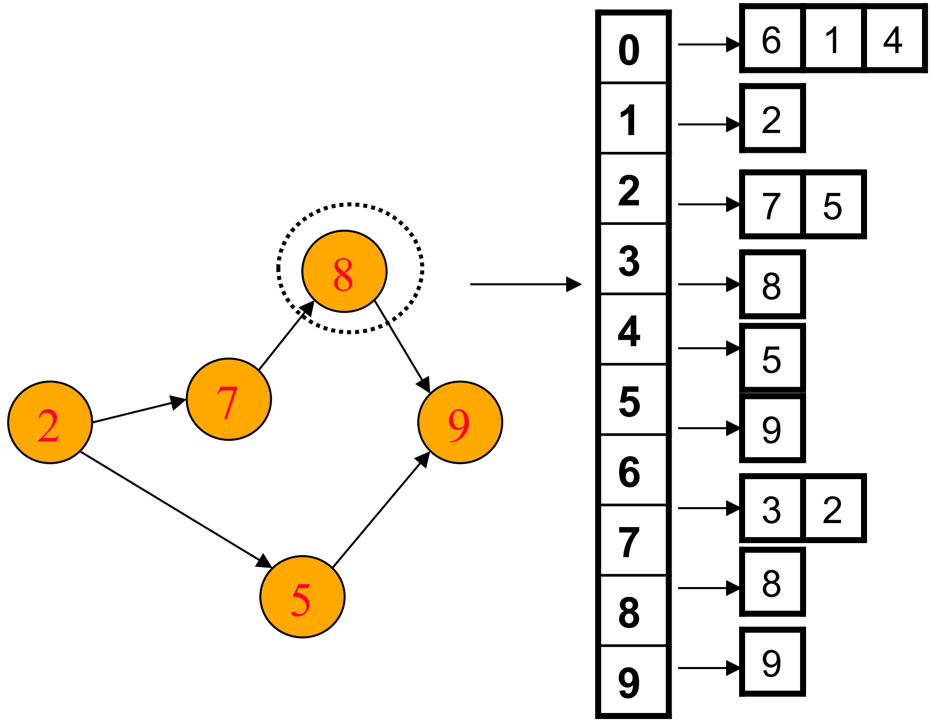
Example



Dequeue 3 Q = { 2 }
Adjust 3's neighbors

OUTPUT: 0 6 1 4 3

Example



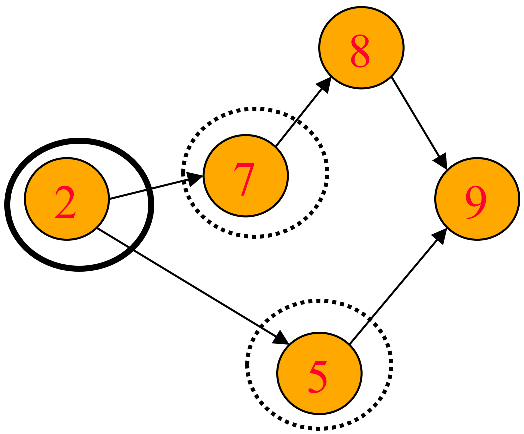
Indegree

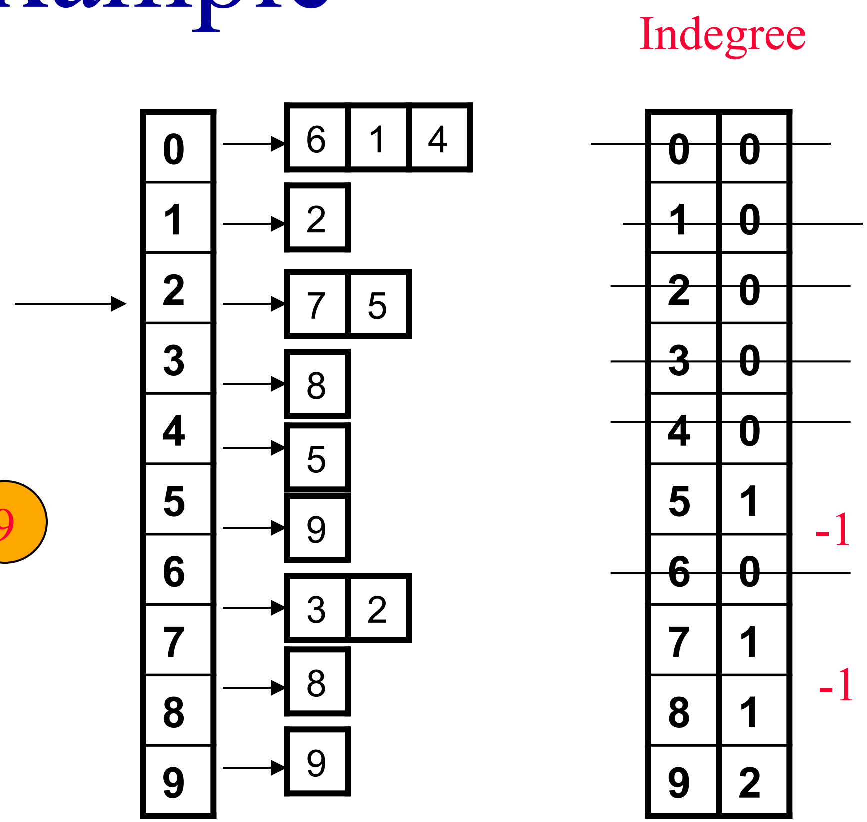
0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	1
9	2

Deque 3 Q = { 2 }
No new start points found

OUTPUT: 0 6 1 4 3

Example

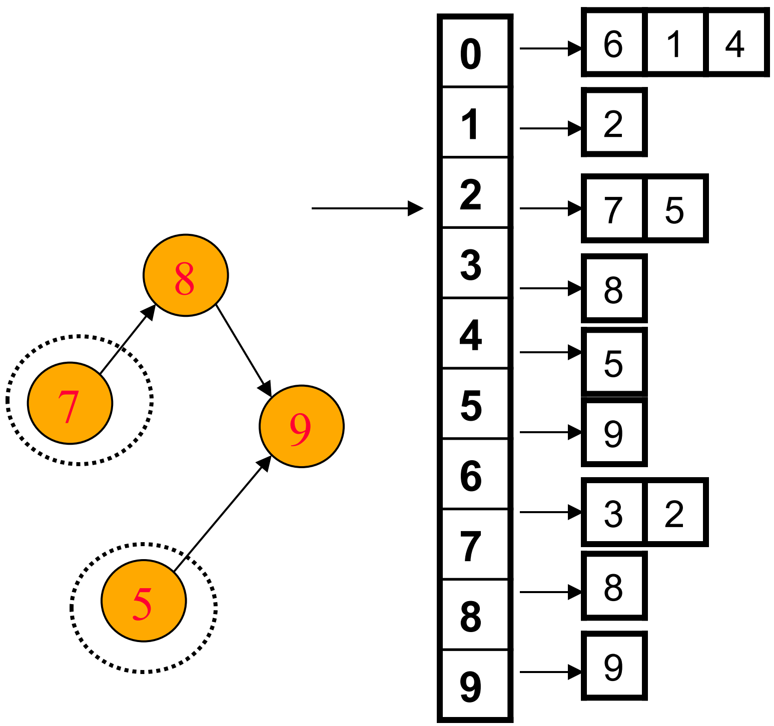




Deqeue 2 Q = { }
Adjust 2's neighbors

OUTPUT: 0 6 1 4 3 2

Example



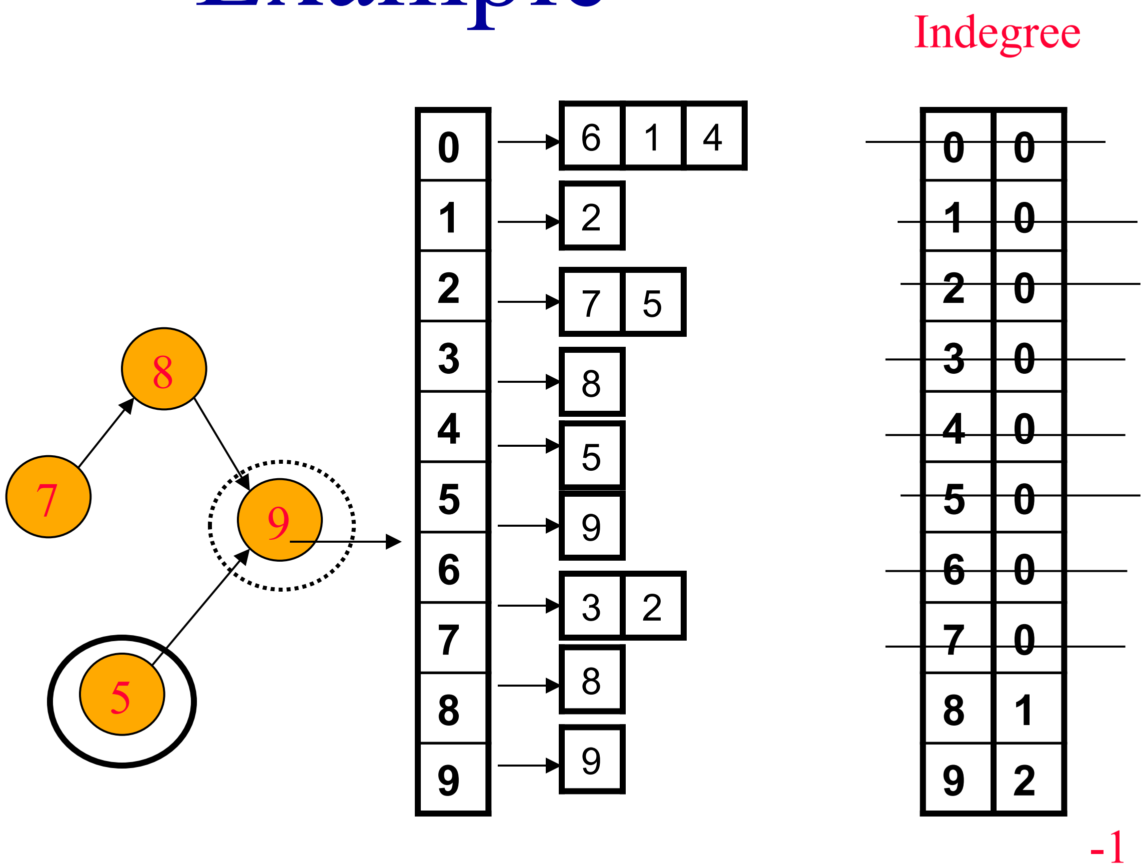
Indegree

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	1
9	2

Dequeue 2 Q = { 5, 7 }
Enqueue 5, 7

OUTPUT: 0 6 1 4 3 2

Example

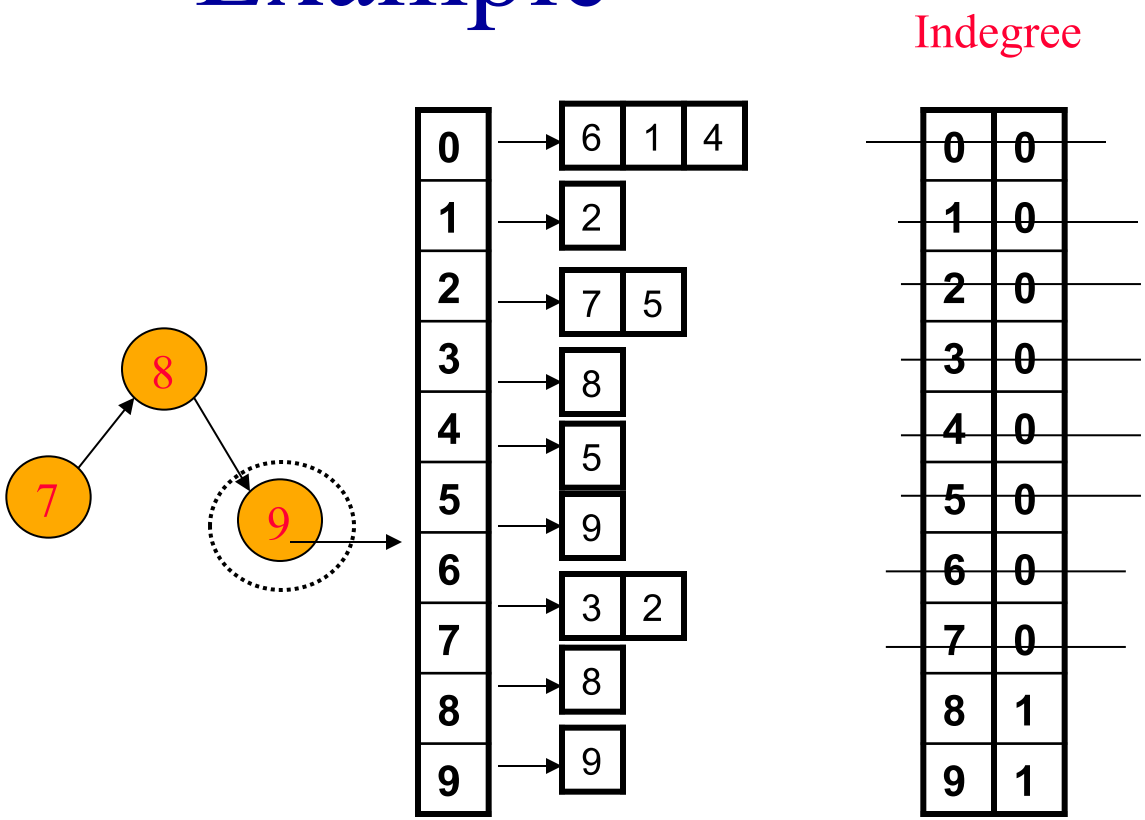


Deque 5 Q = { 7 }

Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5

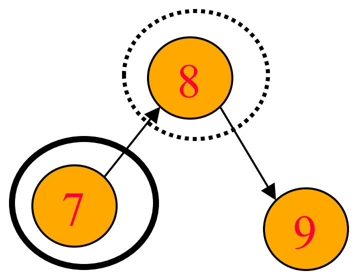
Example



Deque 5 Q = { 7 }
No new starts

OUTPUT: 0 6 1 4 3 2 5

Example



0	6, 1, 4
1	2
2	7, 5
3	8
4	5
5	9
6	3, 2
7	8
8	9
9	

Indegree

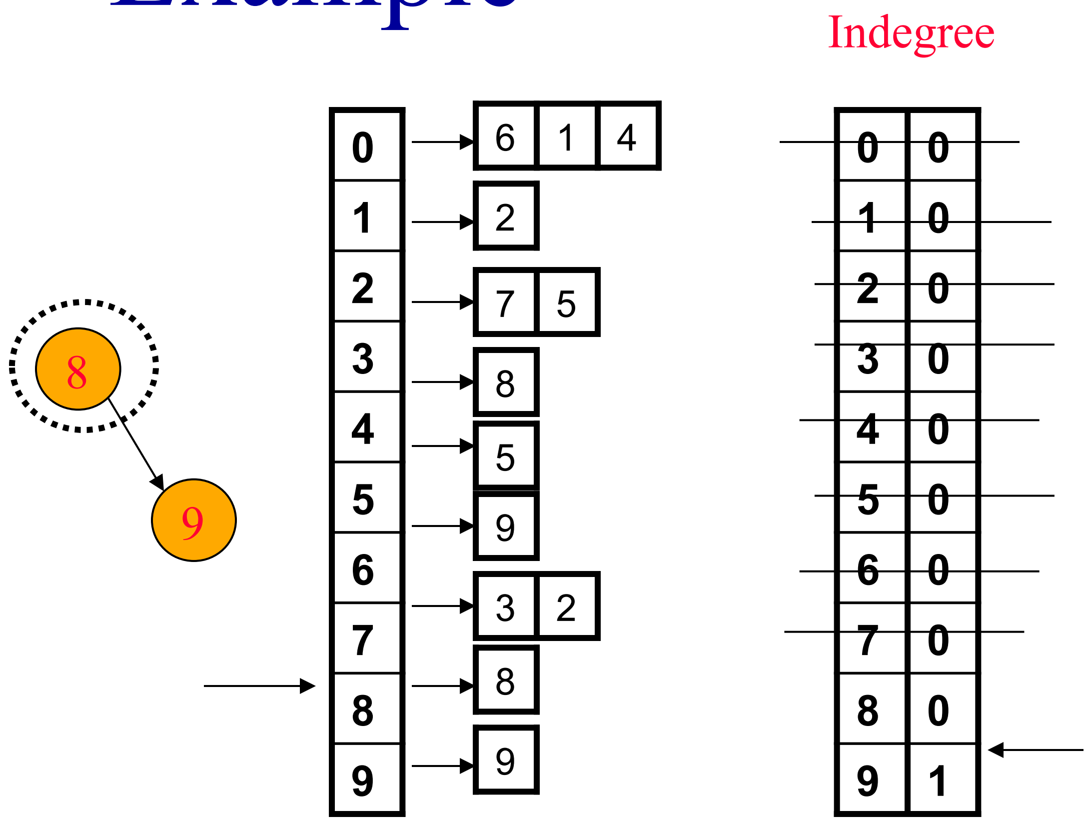
~~0~~	~~0~~
~~1~~	~~0~~
~~2~~	~~0~~
~~3~~	~~0~~
~~4~~	~~0~~
~~5~~	~~0~~
~~6~~	~~0~~
~~7~~	~~0~~
8	1
9	1

-1

Deque 7 Q = { }
Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5 7

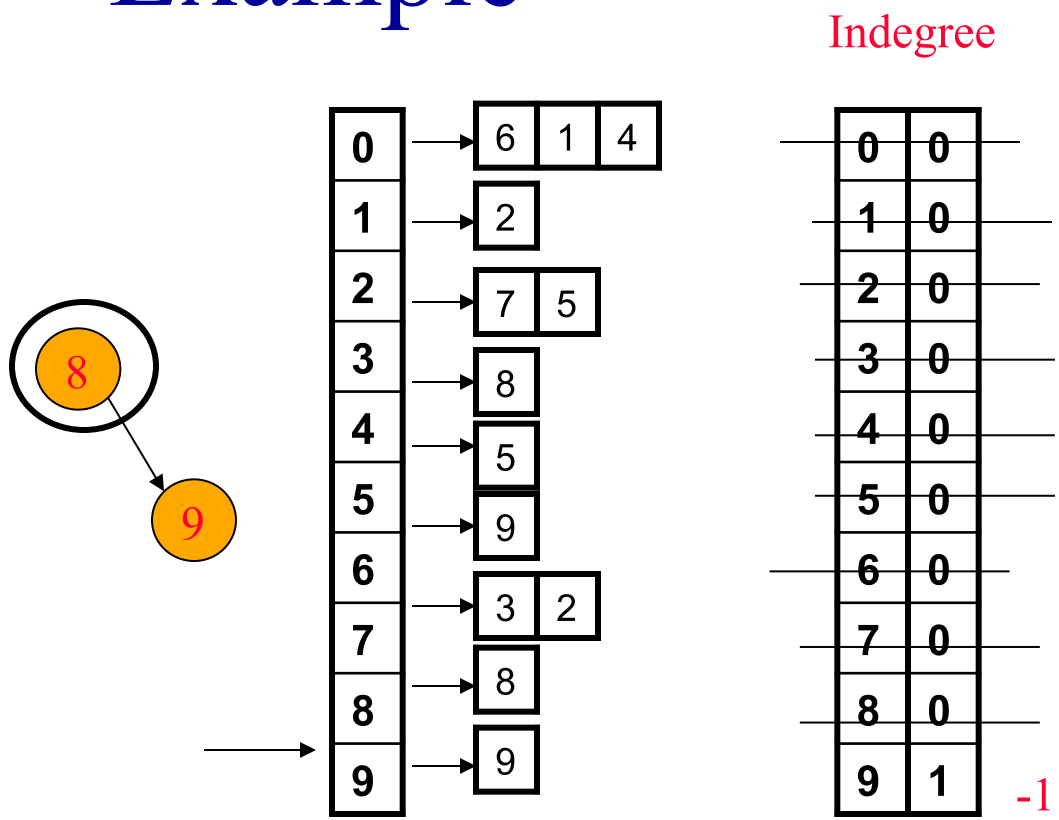
Example



Deque 7 Q = { 8 }
Enqueue 8

OUTPUT: 0 6 1 4 3 2 5 7

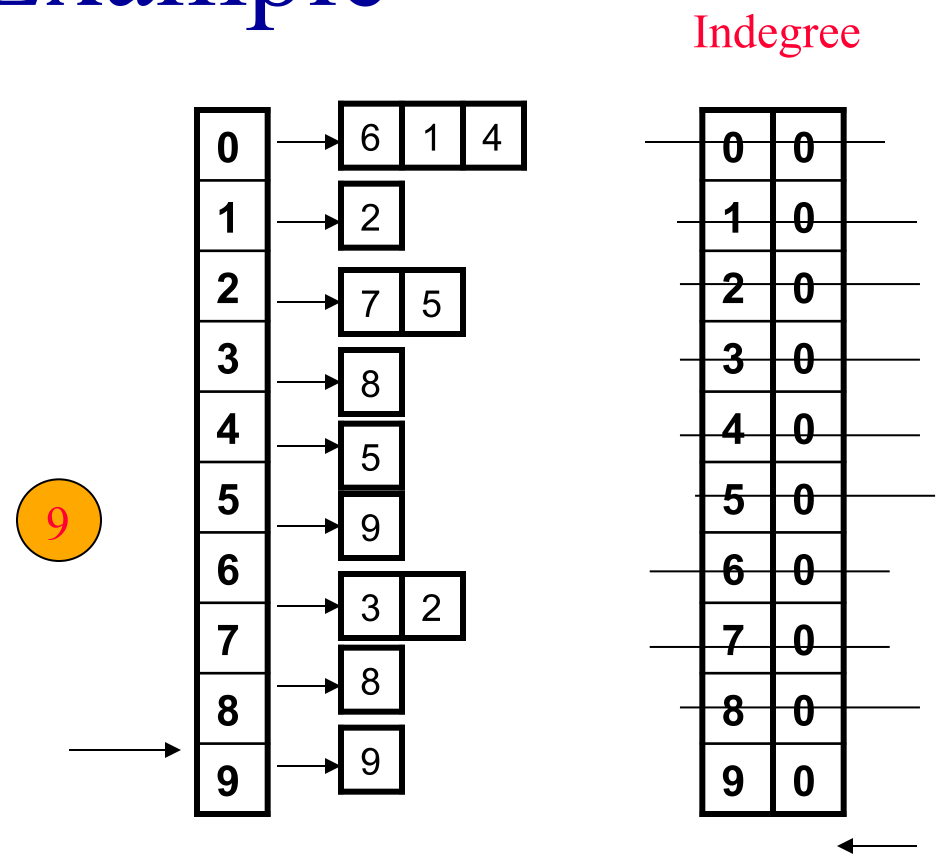
Example



Deqeue 8 Q = { }
Adjust indegrees of neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8

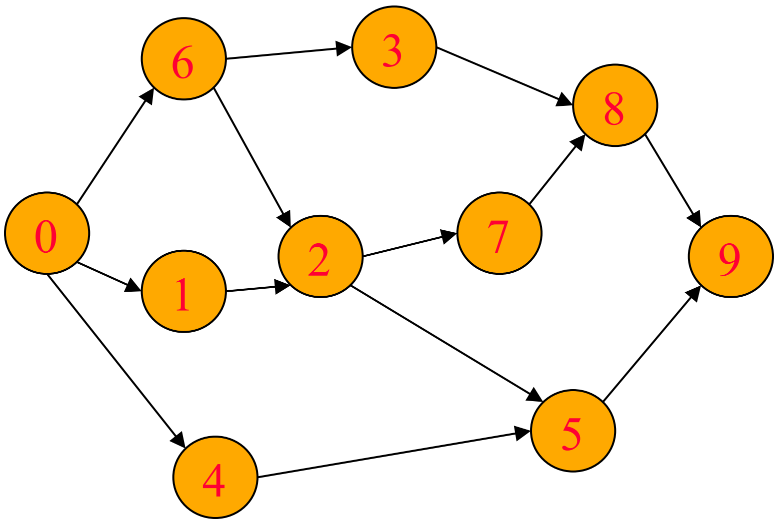
Example



Dequeue 8 Q = { 9 }
    Enqueue 9
Dequeue 9 Q = { }
    STOP – no neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8 9

Example



OUTPUT: 0 6 1 4 3 2 5 7 8 9

Is output topologically correct?

Topological Sort: Complexity

- We never visited a vertex more than one time
- For each vertex, we had to examine all outgoing edges
  - $\Sigma\ outdegree(v) = m$
  - This is summed over all vertices, not per vertex
- So, our running time is exactly
  - $O(n + m)$
- Can we use a stack instead of a queue?

- Input the AOV network, let n be the number of vertices**;**
- **for** (**int** i=0**;** i<n**;** i++) // output the vertices
- {
- **if** (every vertex has a predecessor) **return;**
- // network has a cycle and is infeasible.
- pick a vertex v that has no predecessors**;**
- **cout** << v**;**
- delete v and all edges leading out of v from the network**;**
- }

```
void LinkedGraph::TopologicalOrder() { // count[i] = indegree(i)
    int top = -1, pos = 0;
   for (int i=0; i<n; i++) //create a linked stack of vertices with
      if (count[i]==0) { count[i]=top;  top=i;} //no predecessors
   for (i=0; i<n; i++)
      if (top==-1) throw “network has a cycle.";
     int j=top;  top=count[top]; //unstack a vertex
          t[pos++] = j;  // store vertex j in topological order
     Chain<int>::ChainIterator ji=adjLists[j].begin();
     while (ji != adjLists[j].end()) { // decrease the count of
        count[*ji]--;                    // the successor vertices of j
         if (count[*ji]==0) {count[*ji]=top; top=*ji;} //add to stack
        ji++; // next successor
     }
}
```

Project Planning Problem

- A project
  - Several tasks
  - Task time
  - Task dependencies
- Problem
  - How long at least to finish the project (all tasks)?
  - What tasks are critical to the finish time?

An example

Tasks	Time	Succ
a1	6	a4
a2	4	a5
a3	5	a6
a4	1	a7 a8
a5	1	a7 a8
a6	2	a9

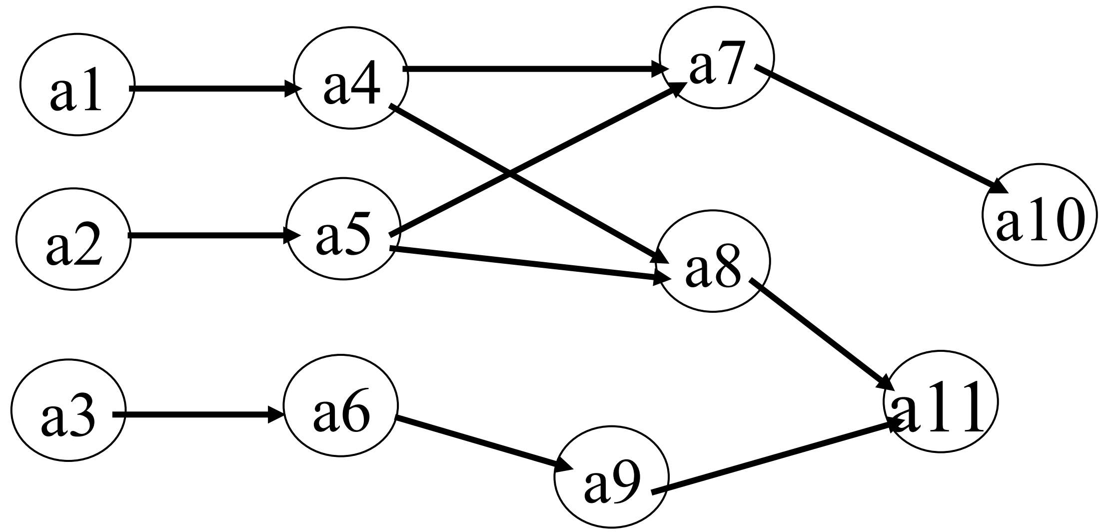
a7	9	a10
a8	7	a11
a9	4	a11
a10	2	
a11	4	

Problem Analysis

- Problem
  - How long at least to finish the project (all tasks)?
  - What tasks are critical to the finish time?
- Key words
  - At Least
    - No delay
  - Critical
    - Delay is not allowed

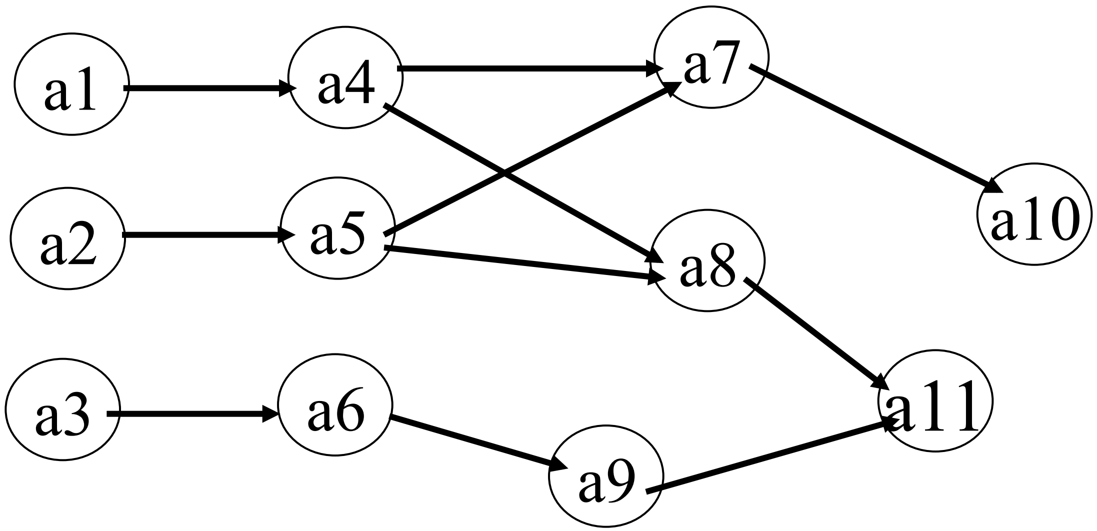
a1	6	a4
a2	4	a5
a3	5	a6
a4	1	a7 a8
a5	1	a7 a8
a6	2	a9
a7	9	a10
a8	7	a11
a9	4	a11
a10	2	
a11	4	

AOV



- Problem
  - How long at least to finish the project (all tasks)?
  - What tasks are critical to the finish time?

Possible Solution



- Topological Sort on AOV?
  - Output task
  - Does not know whether the project is finished or not

Possible Solution

- Analysis
  - We should know what tasks are finished at a given time point
  - Time point
    - Project Phase
    - E.g : after phase 1, task1, 2, 3 are finished
      after phase 2, task1, 2, 3,4,5,6 are finished

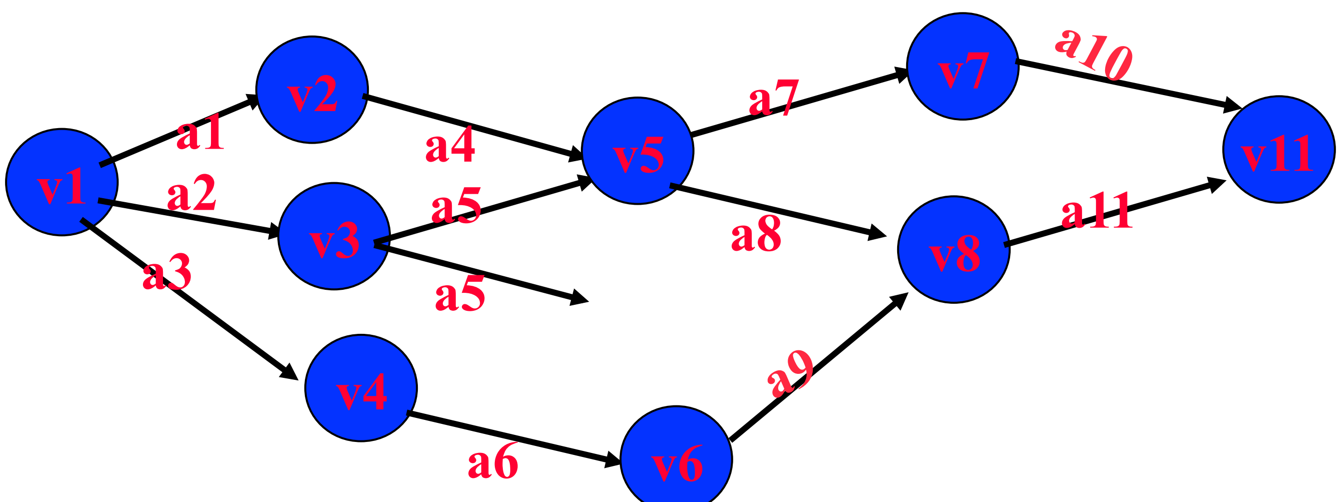
Possible Solution

- If the outputs of topological sort are project phases…
  - We did it!
- How to make it happen
  - Network with project phase as vertex
  - Edges?
    - Tasks!

a1	6	a4
a2	4	a5
a3	5	a6
a4	1	a7 a8
a5	1	a7 a8
a6	2	a9
a7	9	a10
a8	7	a11
a9	4	a11
a10	2	
a11	4	

T	W	Pre
a1	6	
a2	4	
a3	5	
a4	1	a1
a5	1	a2
a6	2	a3
a7	9	a4 a5
a8	7	a4 a5
a9	4	a6
a10	2	a7
a11	4	a8 a9

T	W	Pre
a1	6	
a2	4	
a3	5	
a4	1	a1
a5	1	a2
a6	2	a3
a7	9	a4 a5
a8	7	a4 a5
a9	4	a6
a10	2	a7
a11	4	a8 a9



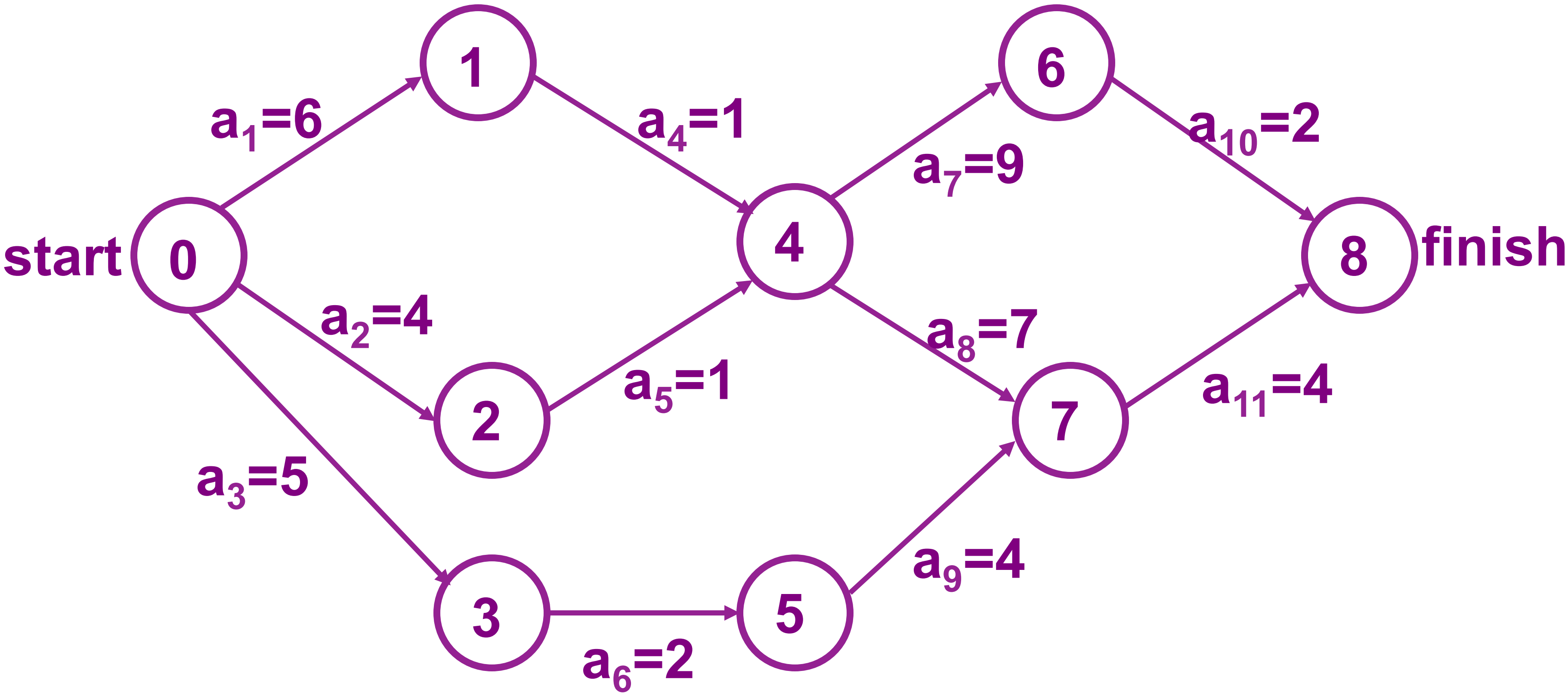
Activity-on-Edge (AOE) Networks

- directed edges --- tasks to be performed
- vertices --- events, signaling the completion of certain activities.
- activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred.
- an event occurs only when all activities entering it have been completed.

Revisit of Project planning

- Problem
  - How long at least to finish the project (all tasks)?
  - What tasks are critical to the finish time?
- Since activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the **longest path** from the start to the finish.
- A path of longest length is a **critical** path.

Another example



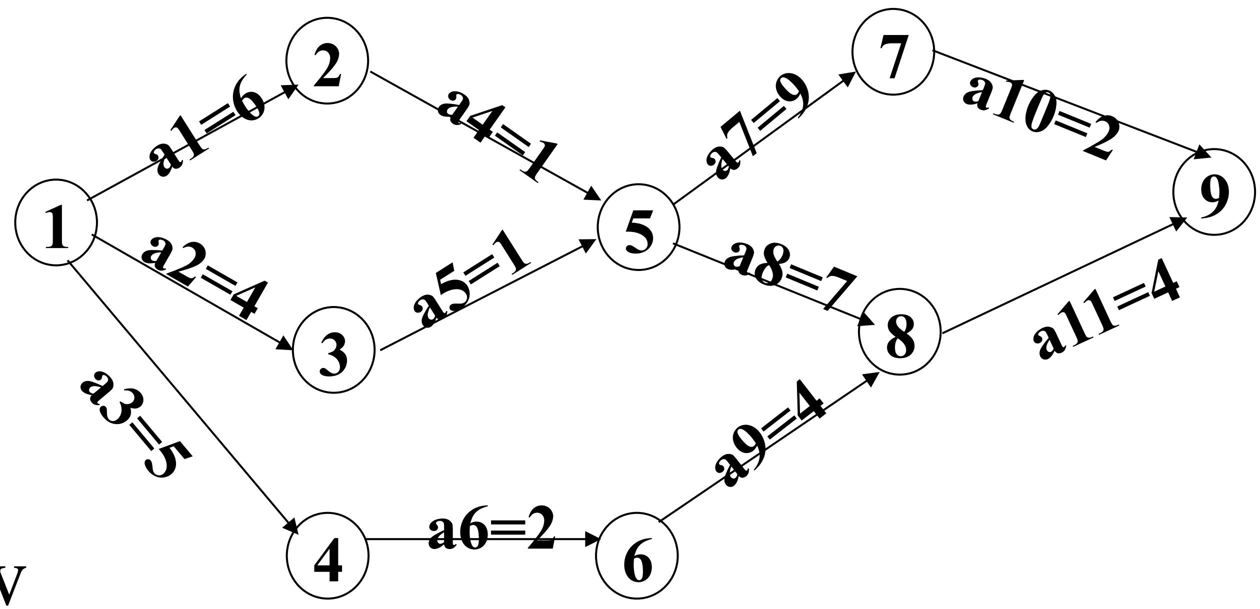
- **Path 0, 1, 4, 6, 8**
- **Path 0, 1, 4, 7, 8**

Critical Activity

- Critical activity
  - Edges in a critical path
  - Cannot delay
  - Starts as soon as possible
- How to identify critical tasks?
  - Given a project time
  - An earliest start time
  - A latest start time
  - If e(i) == l(i), then it is critical

Calculation of Early Activity Times

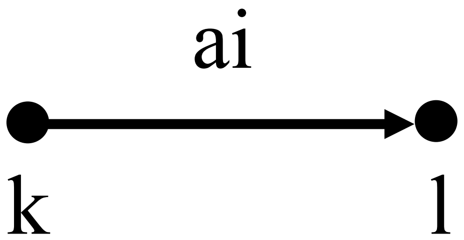
- How to obtain e(i) and l(i)?



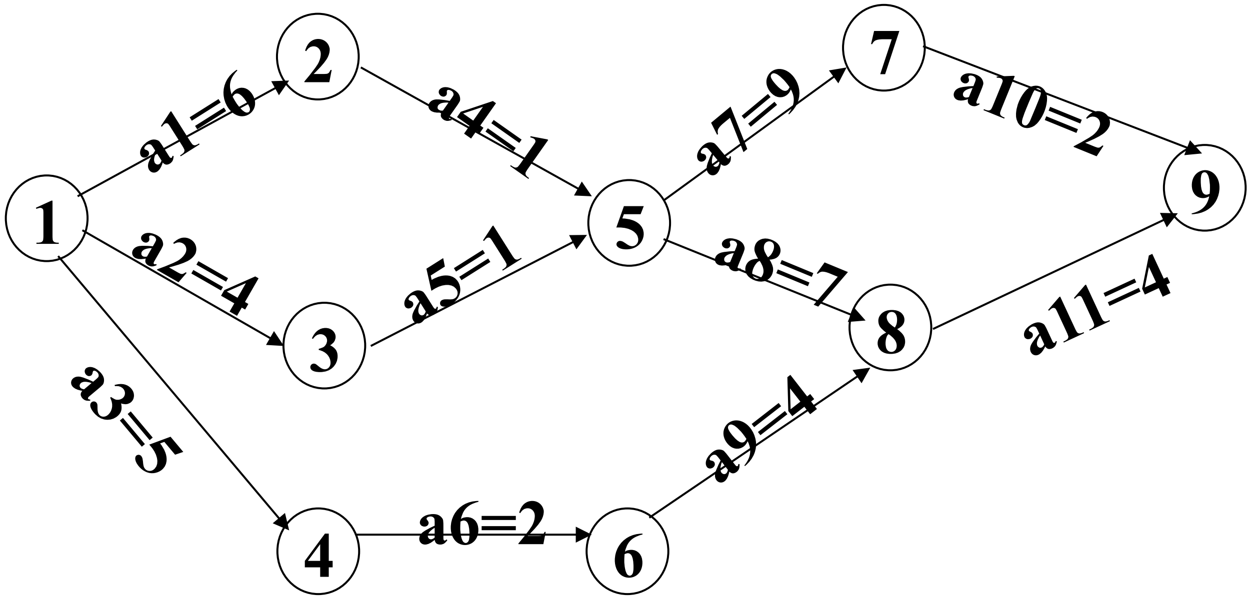
- **a7?**
- If we know
  - Event 5's earliest time
  - Event 7's latest time

Calculation of Early Activity Times

- **If a_i is edge <k, l>, then**
- **（1）e(i)=**
- **Ve(k)**
- **（2）l(i)=**
- **Vl(l)-dut(<k,l>)**



Calculation of Event Times

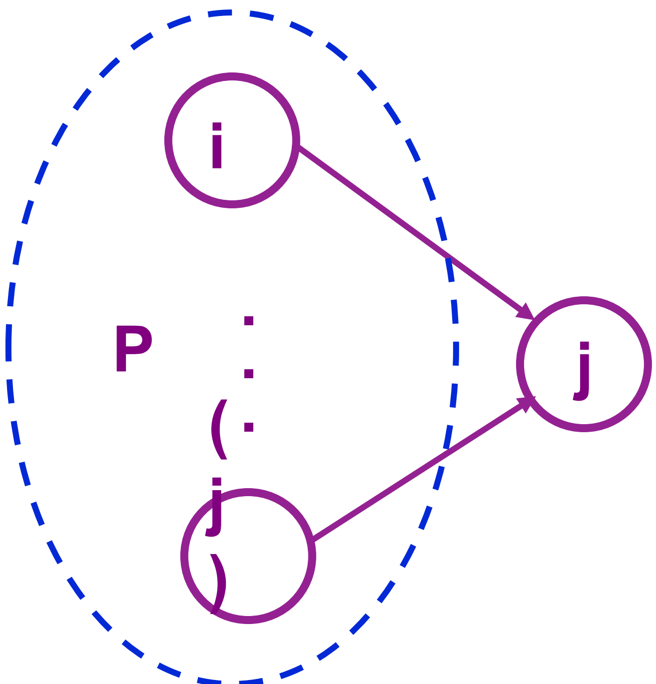


- **E(1) = ?**
  - **0**
- **E(2) = ?**
  - **6**
- **E(3) = ?**
  - **4**
- **E(5) = ?**
  - **7**

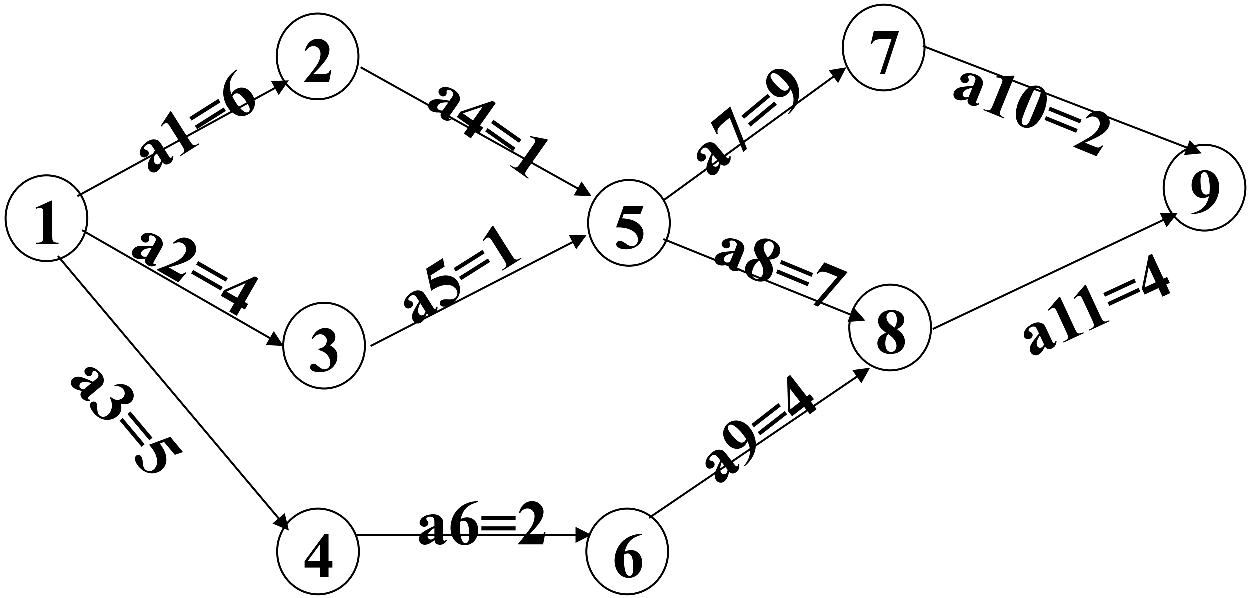
Calculation of Event Times

- P(j) is the set of all vertices adjacent to j.
- ee[0]=0 (suppose 0 is the start)
- $ee[j] = \max_{i \in P(j)} \{ee[i]+\text{duration of } \langle i, j\rangle\}$,

- Topological Order!



Calculation of Event Times

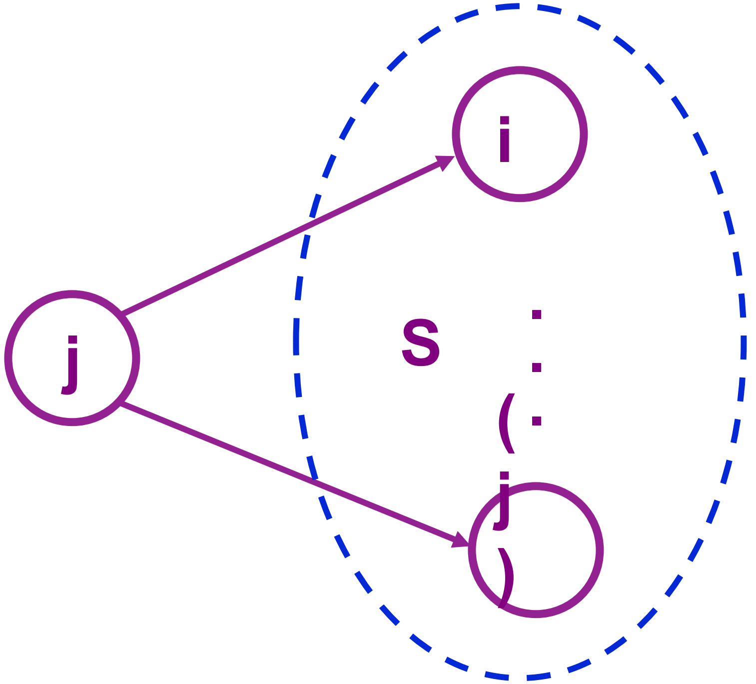


- **L(9) = ?**
  - **E(9)**
- **L(7) = ?**
  - **L(9) – a10**
- **L(8) = ?**
  - **L(9) – a11**
- **L(5) = ?**
  - **Min{L(7)–a7, L(8)-a8}**

Calculation of Event Times

- S(j) is the set of all vertices adjacent from j.
- le[n-1]=ee[n-1]   (suppose n-1 is the finish)
- le[j]= min {le[i]-duration of <j, i>}, i ∈S(j)

- Reverse Topological order!

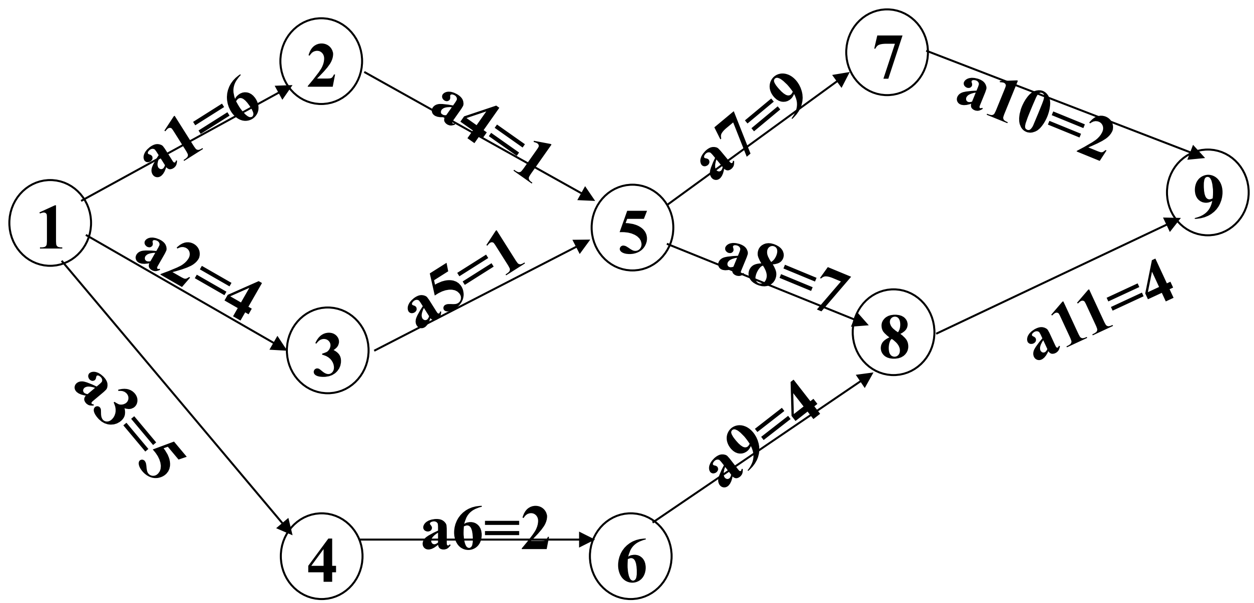


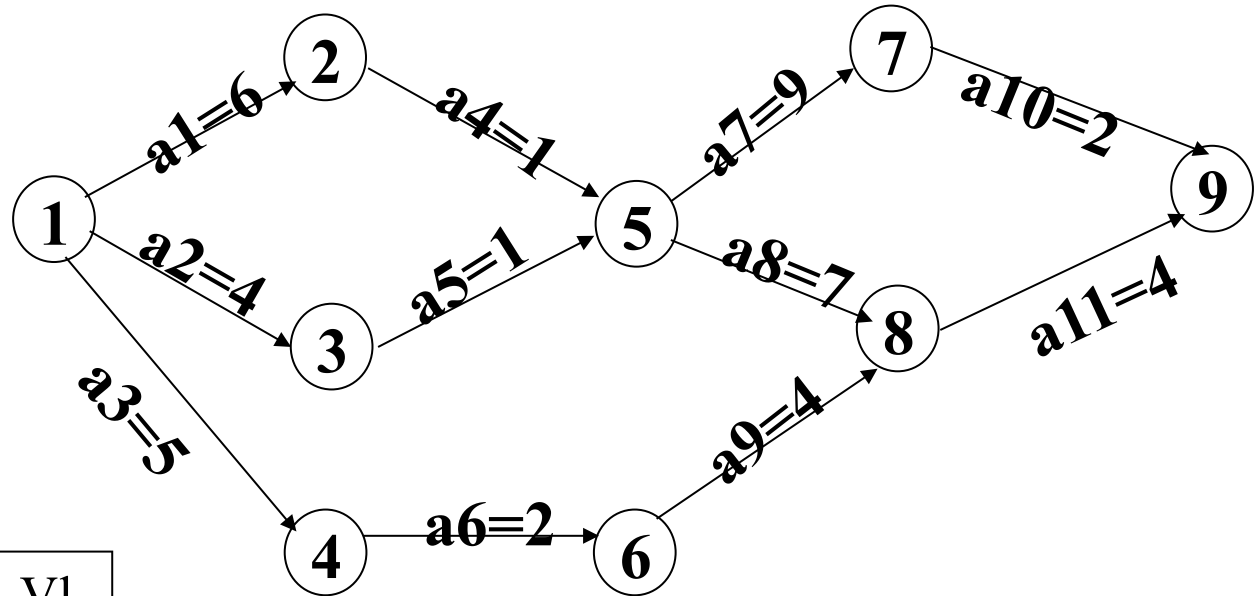
Revisit of Project planning

- Problem
  - How long at least to finish the project (all tasks)?
  - What tasks are critical to the finish time?
- **critical Path**
  - **Path length**
  - **Edges in path**

Critical Path

- Ve(i)
- Vl(i)
- E(i)
- L(i)
- L(i) – E(i)





Vertex	Ve	Vl
V1	0	0
V2	6	6
V3	4	6
V4	5	8
V5	7	7
V6	7	10
V7	16	16
V8	14	14
V9	18	18

Activity	e	l	l-e
a1	0	0	0 ✓
a2	0	2	2
a3	0	3	3
a4	6	6	0 ✓
a5	4	6	2
a6	5	8	3
a7	7	7	0 ✓
a8	7	7	0 ✓
a9	7	10	3
a10	16	16	0 ✓
a11	14	14	0 ✓

- struct Pair
- {
-     int vertex;
-     int dur;   //activity duration
- };

```
class LinkedGraph {
private:
    Chain<Pair> *adjLists;
    int *count, *t, *ee, *le;
    int n;
public:
    LinkedGraph (const int vertices) : {
       if (vertices < 1) throw "Number of vertices must be > 0";
       n = vertices;
       adjLists = new Chain<Pair>[n];
       count = new int[n]; t = new int[n];
       ee = new int[n]; le = new int[n];
    };
   void TopologicalOrder();
   void EarliestEventTime();
   void LatestEventTime();
   void CriticalActivities();
};
```

```
• void LinkedGraph::EarliestEventTime()
• { // assume a topological order has already been in t,
•   // compute ee[j] according to t
•    fill(ee, ee+n, 0);   // initialize ee
•    for (i=0; i<n; i++) {
•        int j=t[i];
•        Chain<Pair>::ChainIterator ji=adjLists[j].begin();
•        while (ji!=adjLists[j].end()) {
•            int k=ji→vertex; //k is successor of j
•            if (ee[k]<ee[j]+ji→dur) ee[k]=ee[j]+ji→dur;
•              ji++;
•        }
• }
```

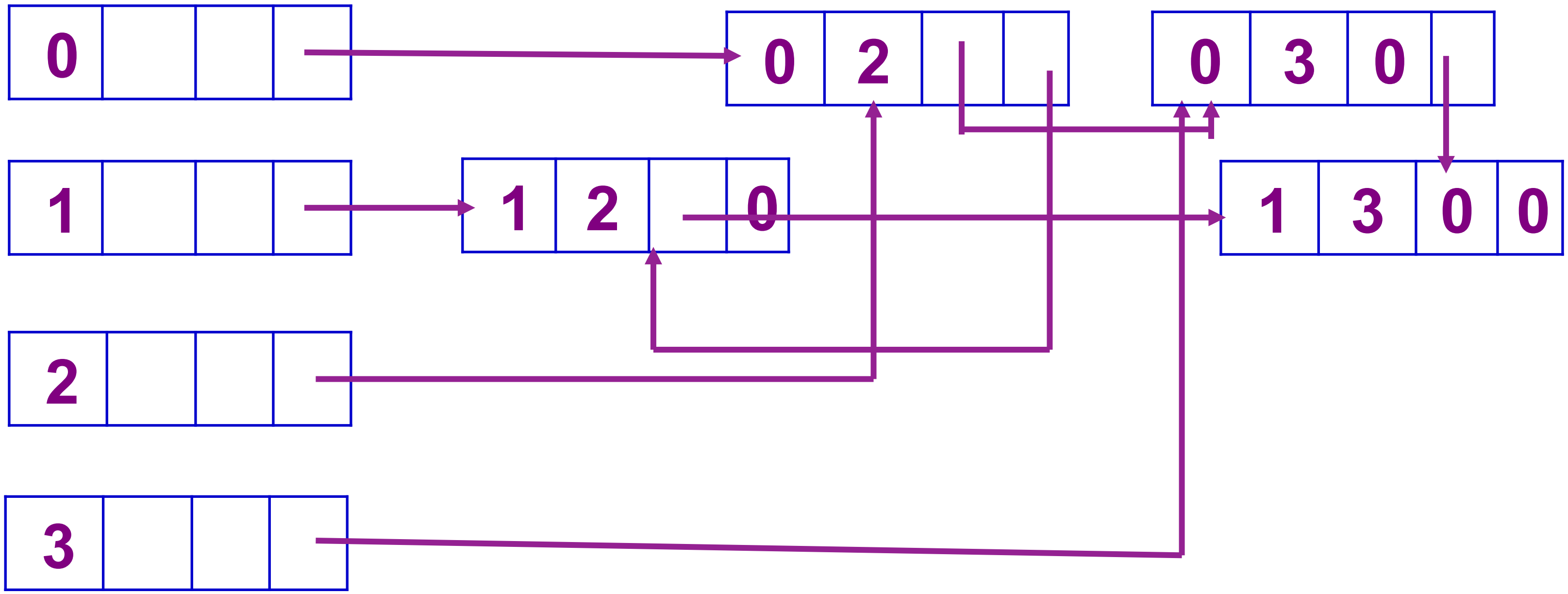
- **void** LinkedGraph::LatestEventTime()
- { // assume a topological order in t, ee has
- // been computed, compute le[j] in the reverse order of t
- fill(le, le+n, ee[n-1]**);** // initialize le
- **for** (i=n-2**;** i>=0**;** i--) {
- **int** j=t[i]**;**
- Chain<Pair>::ChainIterator ji=adjLists[j].begin()**;**
- **while** (ji!=adjLists[j].end()) {
- **int** k=ji→vertex**;** //k is successor of j
- **if** (le[k]-ji→dur<le[j]) le[j]=le[k]-ji→dur**;**
- ji++;
- }
- }

```
void LinkedGraph::CriticalActivities()
{ // compute e[i] and l[i], output critical activities
    int i=1; // the numbering counter for activities
    int u, v, e, l; // e, l are the earliest, latest start time of <u, v>
    for (u=0; u<n; u++) {  // scan the adjacency lists.
       Chain<Pair>::ChainIterator ui=adjLists[u].begin();
       while (ui!=adjLists[u].end()) {
          int v=ui→vertex; // <u, v> is an edge numbered i
          e=ee[u]; l=le[v]-ui→dur;
          if (l==e) cout <<“a”<<i<<“<“<<u<<“,”<<v<<“>”
                              <<“is a critical activity”<<endl;
            ui++; i++;
       }
    }
}
```

Exercises: P389-2, p390-5

Graph

- Definitions
- Representations
- Search algorithms
- Spanning tree
- Shortest path
- AOV
- AOE


0
1
2
3
0 2
1 2 0
0 3 0
1 3 0 0