Graphs

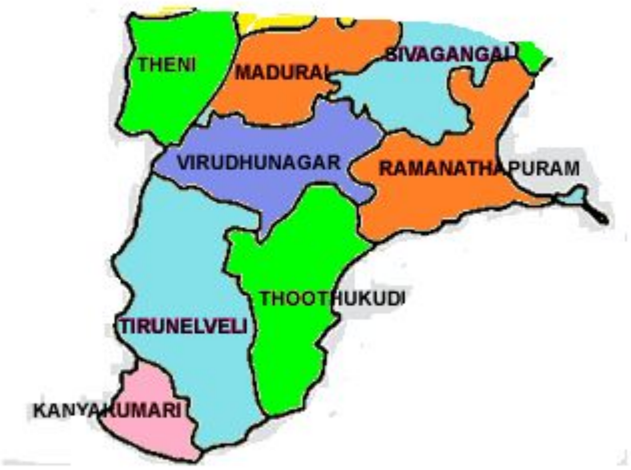
Graphs

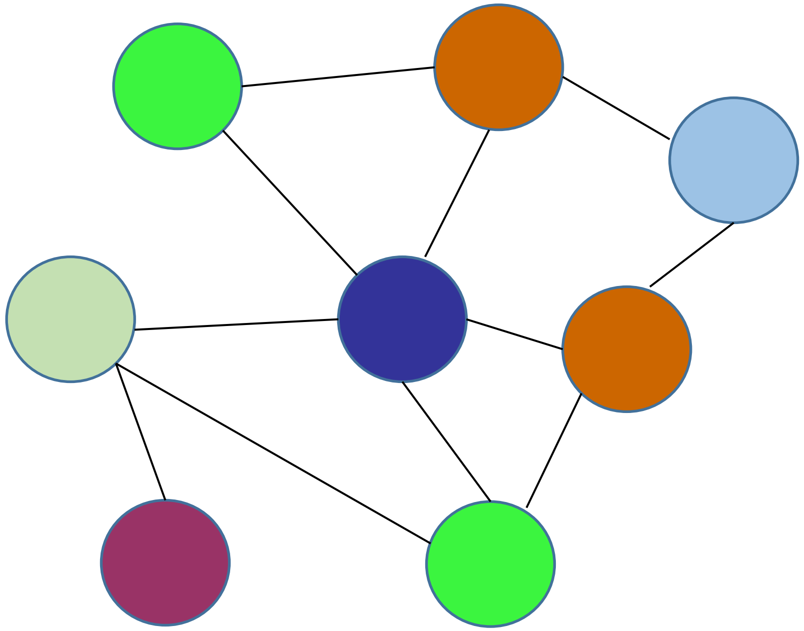
- Non linear Data structures
- Represent many to many relationship among elements, adjacency relation
- Each element has many successors and predecessors

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.

Graph – Adjacency representation





Orientation of edges

- Directed edge has an orientation (u,v)

$$u \longrightarrow v$$

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

$$u \text{ —— } v$$

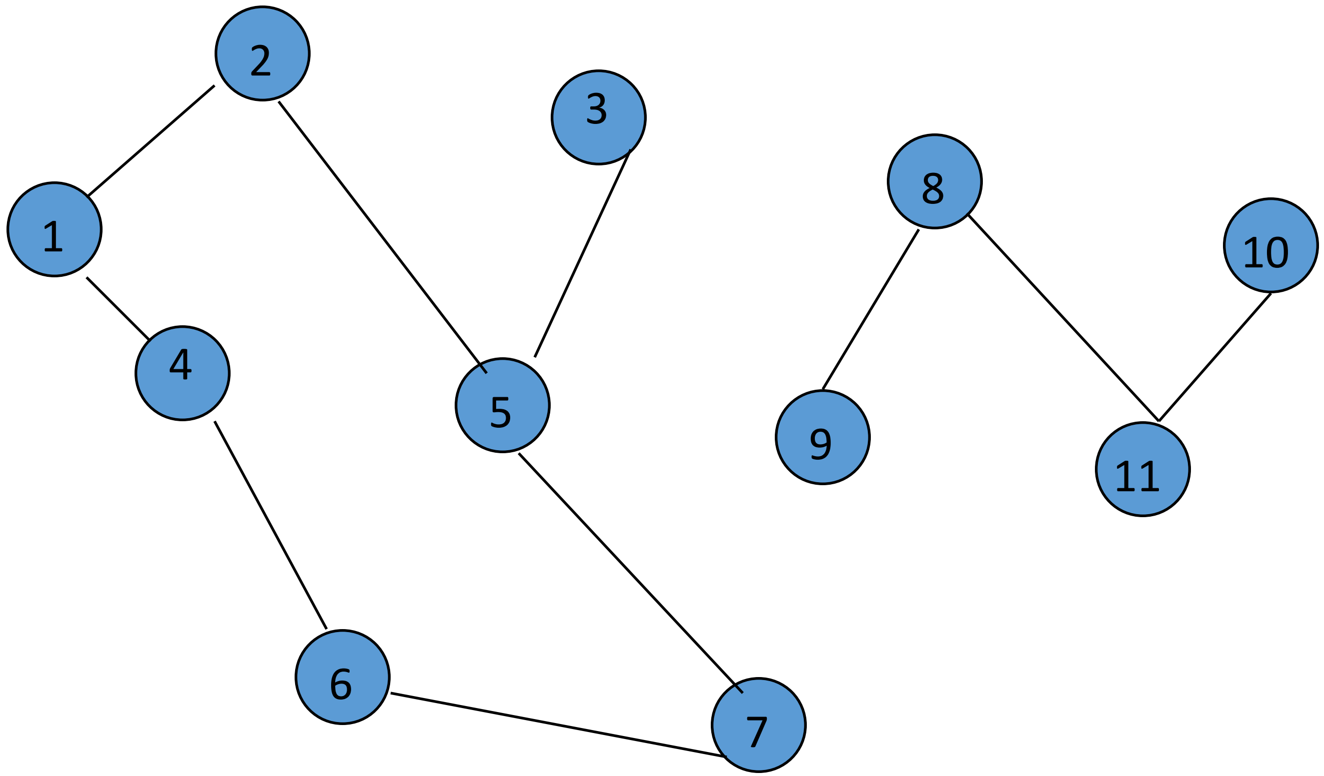
Types

- Directed Graph
- Undirected Graph
- Weighted Graph

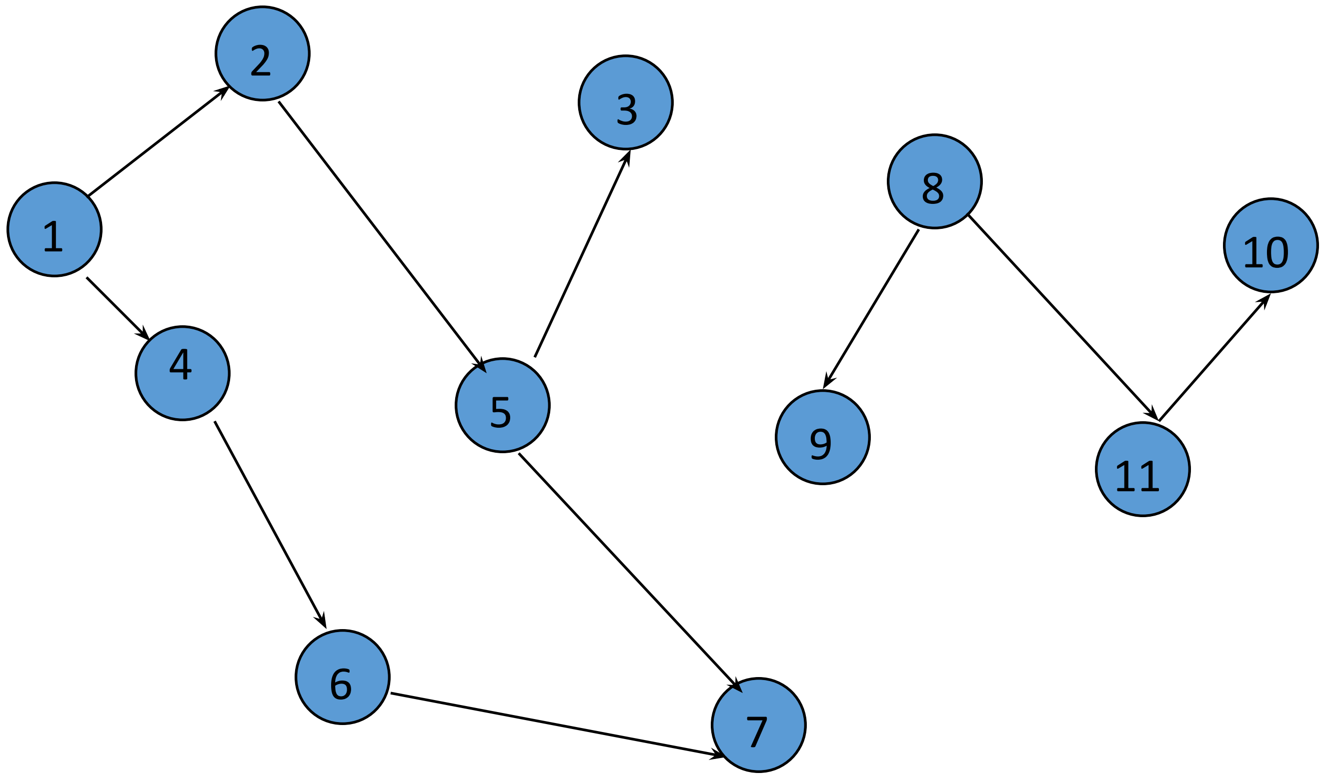
Directed and undirected graph

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

Undirected Graph



Directed Graph (Digraph)



Applications

Vertex	Edge
City	Road
Computer	Communication Link
Railway Station	Tracks
Bus Station	Road Route
Social connect - User	Friendship

Terminology

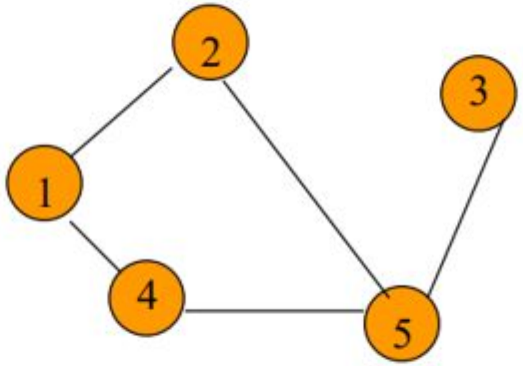
- **Loop**: an edge that connects a vertex to itself.
- **Path**: a sequence of vertices, p_0, p_1, ..., p_m, such that each adjacent pair of vertices p_i and p_{i+1} are connected by an edge.
- **Cycle**: a simple path with no repeated vertices or edges other than the starting and ending vertices. A cycle in a directed graph is called a directed cycle.
- **Multiple edges**: in principle, a graph can have two or more edges connecting the same two vertices in the same direction.
- **Simple graphs**: the graphs that have no loops and no multiple edges.
- **Directed acyclic graphs**: A directed graph with no cycles is a directed acyclic graph (DAG).

Representation of Graphs

- Adjacency matrix
- Adjacency List

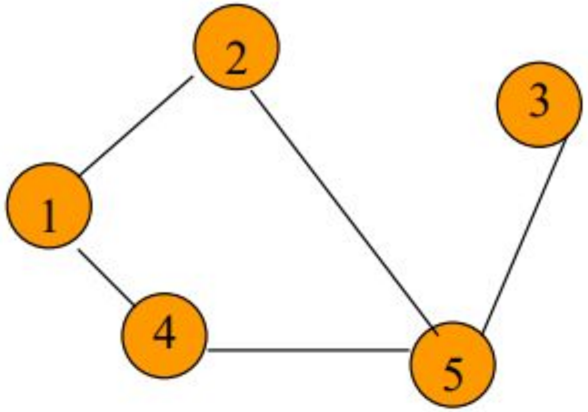
Adjacency Matrix

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



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

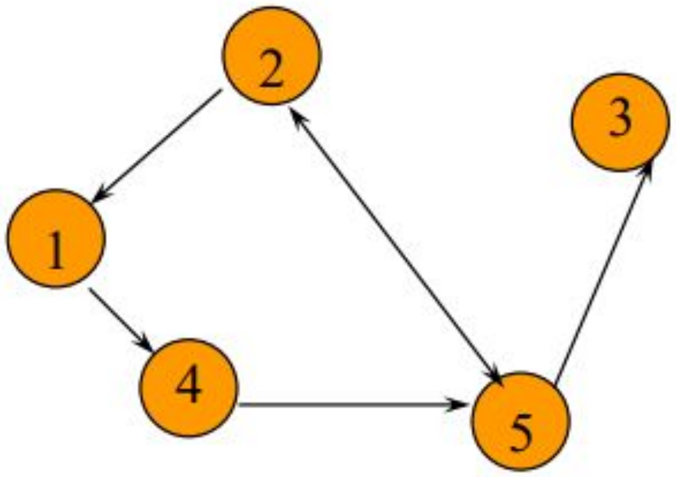
Adjacency Matrix Properties



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

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

Adjacency Matrix (Digraph)



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

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

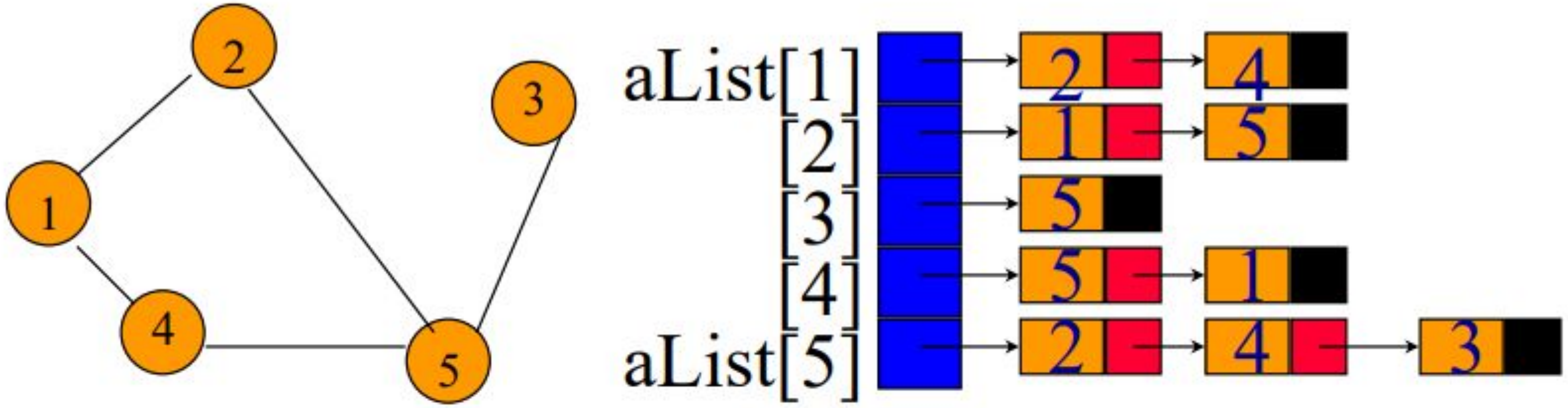
Adjacency Matrix

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

Adjacency List Representation

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

Adjacency List Representation

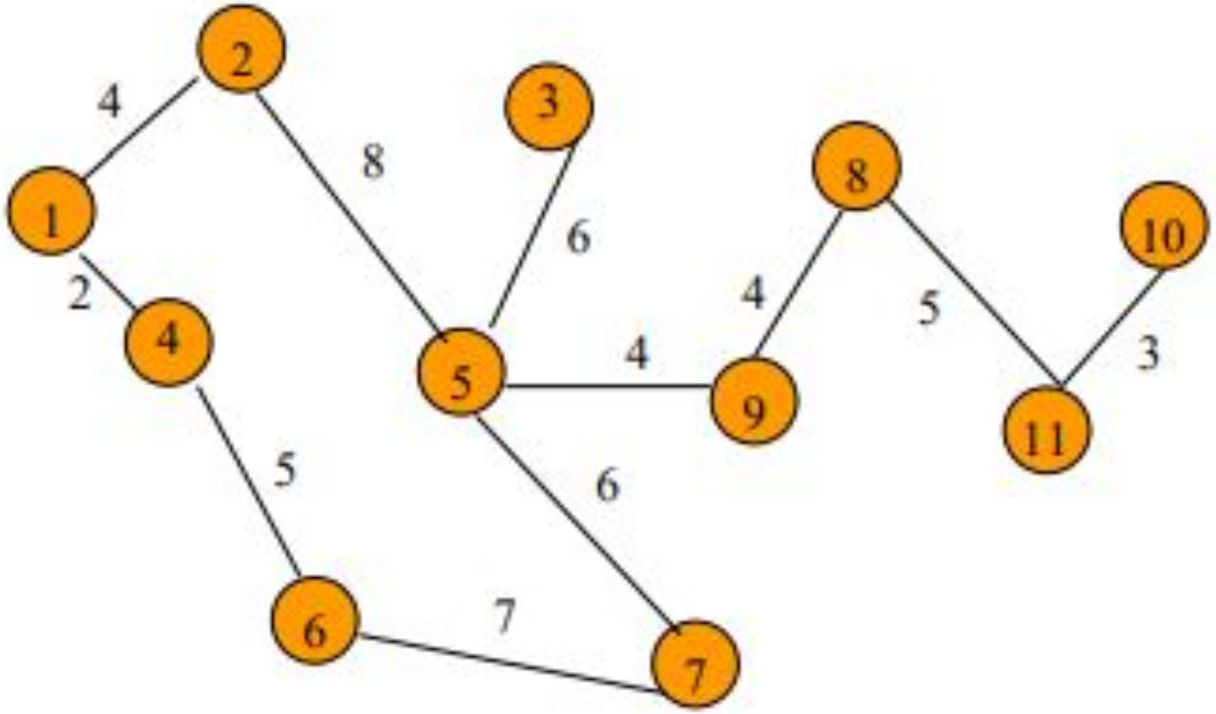


Array Length = n

# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Driving Distance/Time Map



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

Which Representation is Best?

- If space is available, then an adjacency matrix is easier to implement and is generally easier to use than edge lists
- Other operations
    1. Adding or removing edges
    2. Checking whether a particular edge is present
    3. Iterating a loop that executes one time for each edge with a particular source vertex

Choice of representation for efficient implementation of these operations?

Which Representation is Best?

- Operations
  1. Adding or removing edges
  2. Checking whether a particular edge is present
  3. Iterating a loop that executes one time for each edge with a particular source vertex

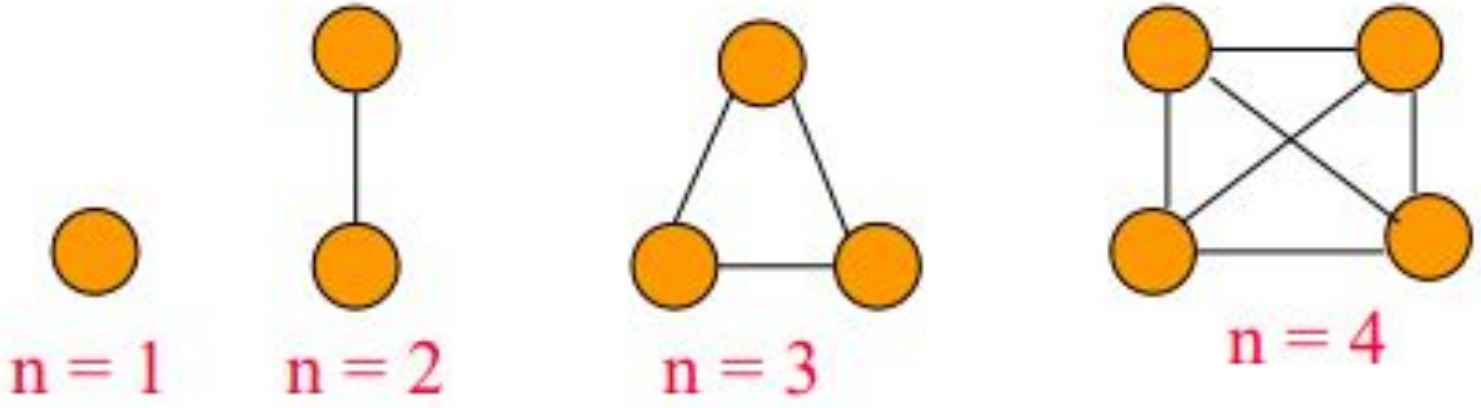
Choice

- Both (1) and (2) require only a small constant amount of time with the adjacency matrices.
- Both (1) and (2) require $O(n)$ operations with the adjacency list representation in the worst case
- With (3), edge lists ($O(e)$, where e is the number of edges that have vertex i as their source) are more efficient than adjacency matrix ($O(n)$).

    If each vertex has only a few edges (sparse graph), then an adjacency matrix will waste space, with many 0

Complete Undirected Graph

Has all possible edges.



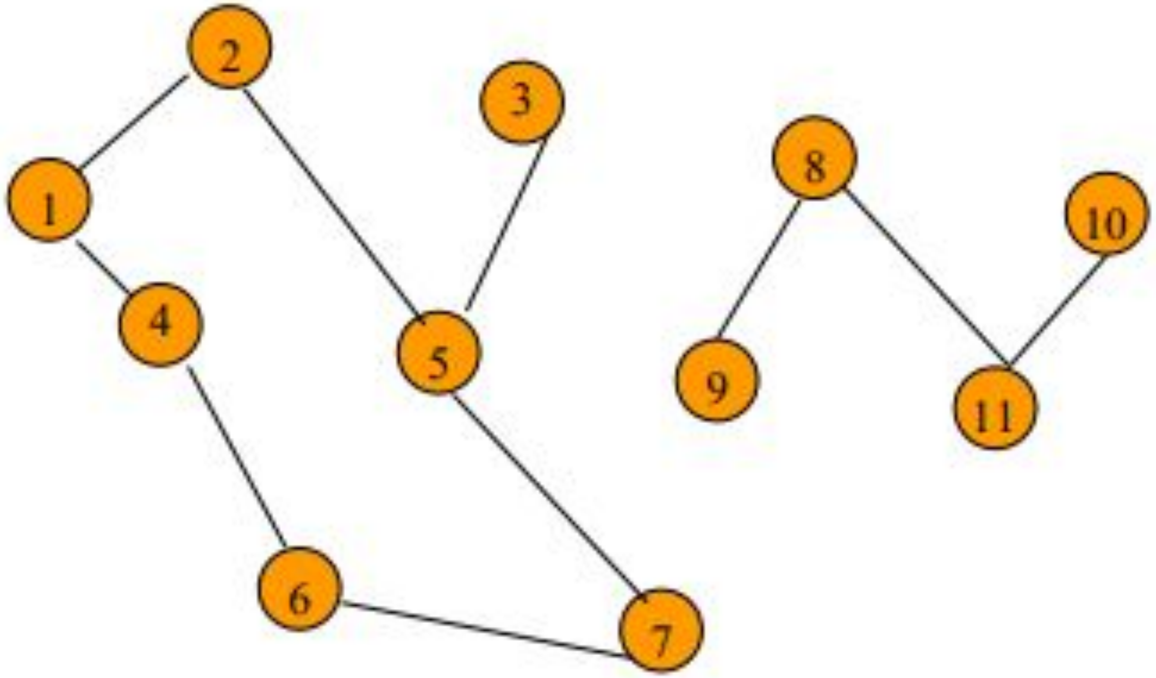
Number Of Edges—Undirected Graph

- 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).

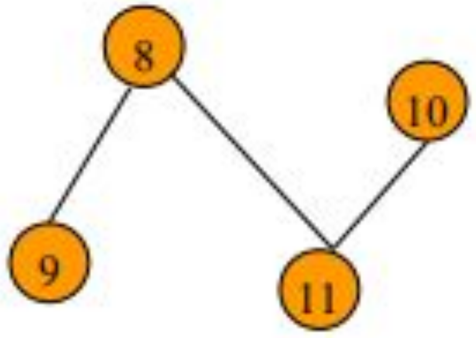
Vertex Degree



Number of edges incident to vertex.

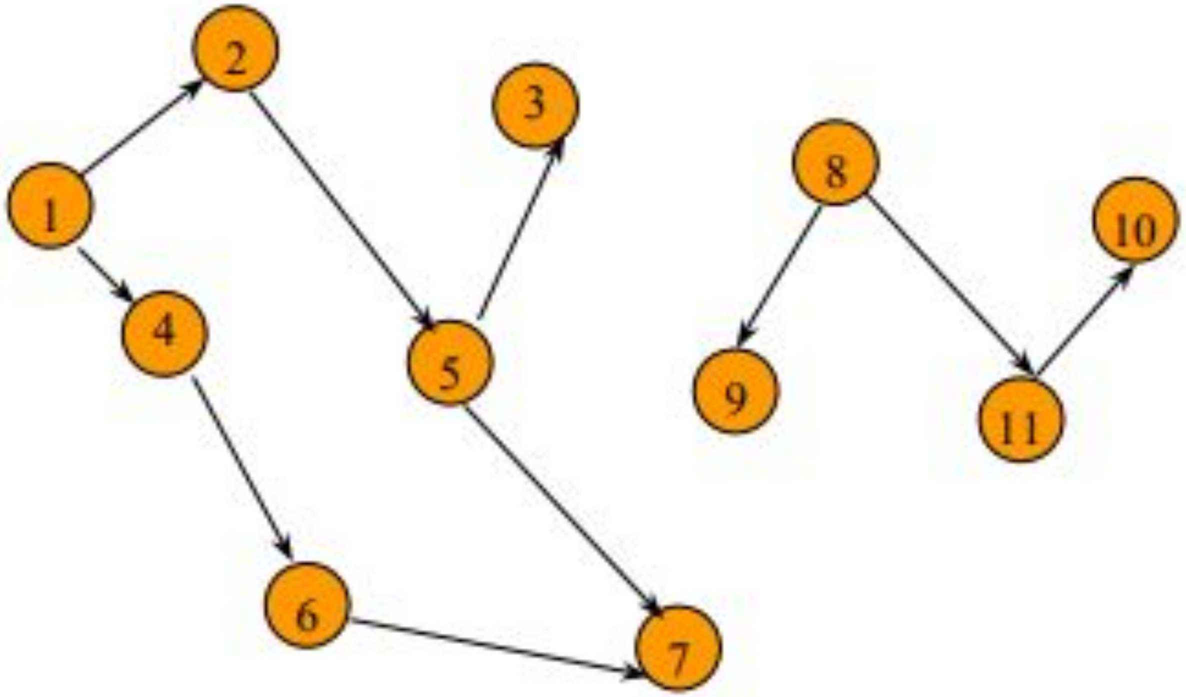
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum Of Vertex Degrees



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

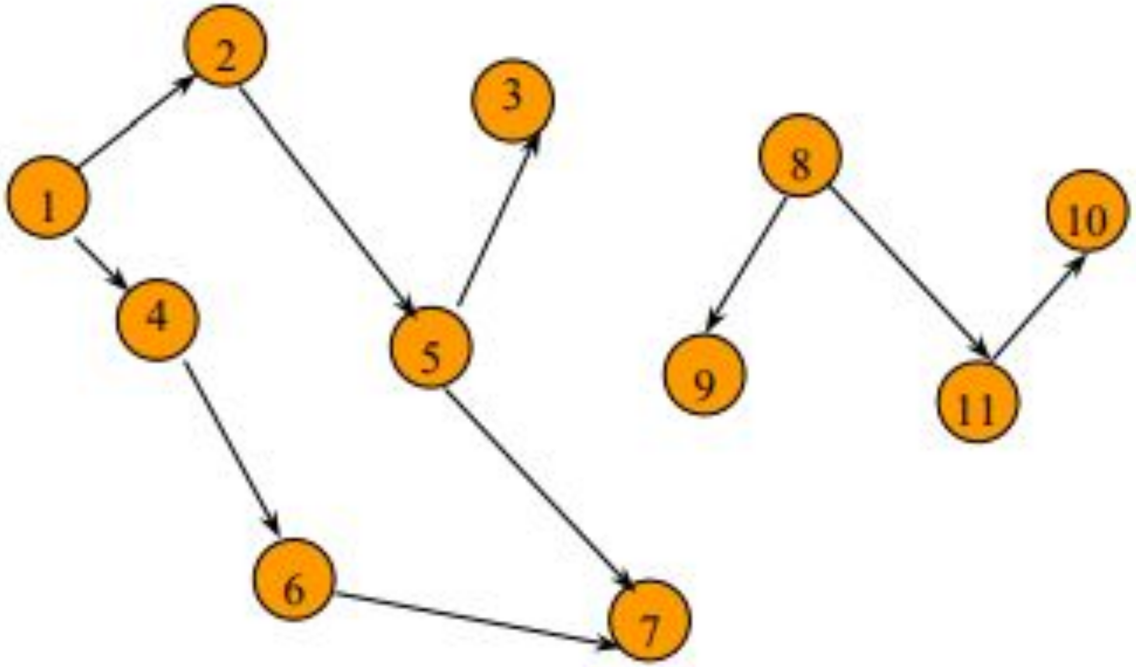
In-Degree Of A Vertex



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 Traversal

- The goal of a graph traversal is to find all nodes reachable from a given set of root nodes.
- In an undirected graph we follow all edges
- In a directed graph we follow only out-edges

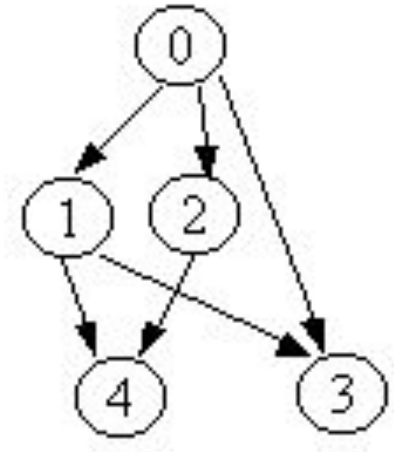
Breadth First Traversal

- Explore neighbours first, before moving to the next level of neighbours.
- Repeatedly explore adjacent vertices using Mark each vertex we visit, so we don't process each more than once.

BFS pseudo code

```
Algorithm BFS(Node start)
 initialize queue q
 insert(q, start )
 mark start as visited
 while(q is not empty) {
        next = delete(q)       // and “process”
        for each node u adjacent to next
              if(u is not marked) {
                     mark u
                     insert ( q , u)
             }  [end if] [end for]
 } [end while]
```

```
Algorithm BFS( start )
 initialize queue q
 insert(q, start )
 mark start as visited
 while(q is not empty) {
       next = delete(q)      // and "process"
       for each node u adjacent to next
            if(u is not marked) {
                   mark u
                   insert ( q , u)
            }  [end if] [end for]
 } [end while]
```



start = 0

q					

q	0				

	0	1	2	3	4
visited	T				

next = 0 BFS = 0

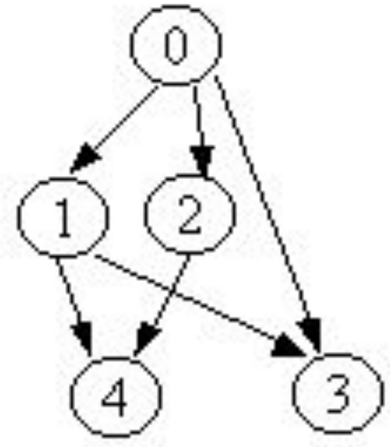
q					

u = 1, 2, 3

	0	1	2	3	4
visited	T	T	T	T	

q	1	2	3		

```
Algorithm BFS( start )
 initialize queue q
 insert(q, start )
 mark start as visited
 while(q is not empty) {
      next = delete(q)      // and "process"
      for each node u adjacent to next
           if(u is not marked) {
                mark u
                insert ( q , u)
           }  [end if] [end for]
 } [end while]
```



u = 1, 2, 3

	0	1	2	3	4
visited	T	T	T	T	

q	1	2	3		

next = 1 BFS = 0 1

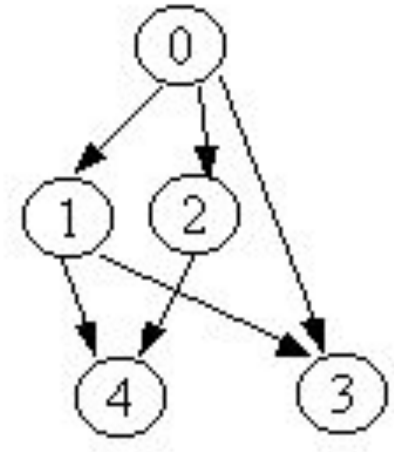
q		2	3		

u = 4, 3

	0	1	2	3	4
visited	T	T	T	T	T

q		2	3	4	

```
Algorithm BFS( start )
 initialize queue q
 insert(q, start )
 mark start as visited
 while(q is not empty) {
       next = delete(q)      // and "process"
       for each node u adjacent to next
            if(u is not marked) {
                   mark u
                   insert ( q , u)
            }  [end if] [end for]
 } [end while]
```



next = 2   u = 4   BFS = 0 1 2

q

		3	4	

visited

0	1	2	3	4
T	T	T	T	T

q

			4	

next = 3   BFS = 0 1 2 3

			4	

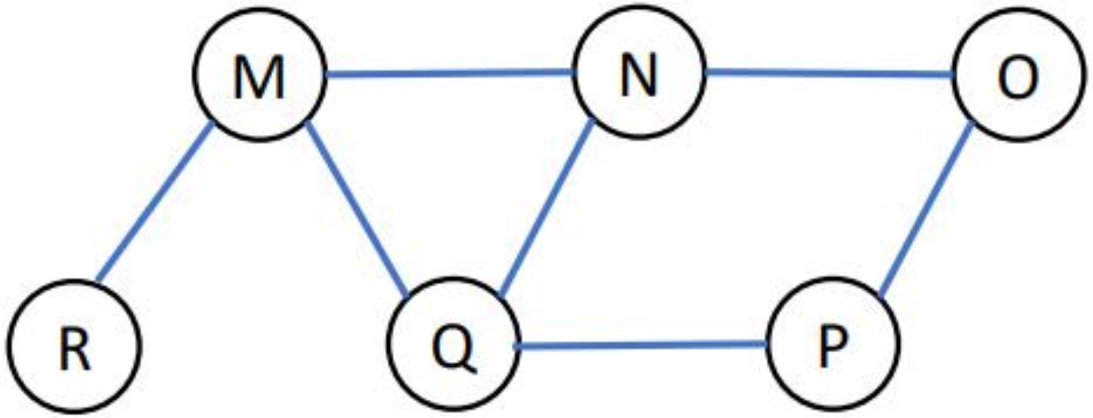
next = 4   **BFS = 0 1 2 3 4**

visited

0	1	2	3	4
T	T	T	T	T

q

What is one possible order of visiting the nodes of the following graph when using Breadth First Search (BFS)?



A) MNOPQR

B) NQMPOR

C) QMNPRO

D) QMNPOR

Depth First Traversal

- Follow a path until it ends, or until a cycle.
- Use a <u>stack</u>

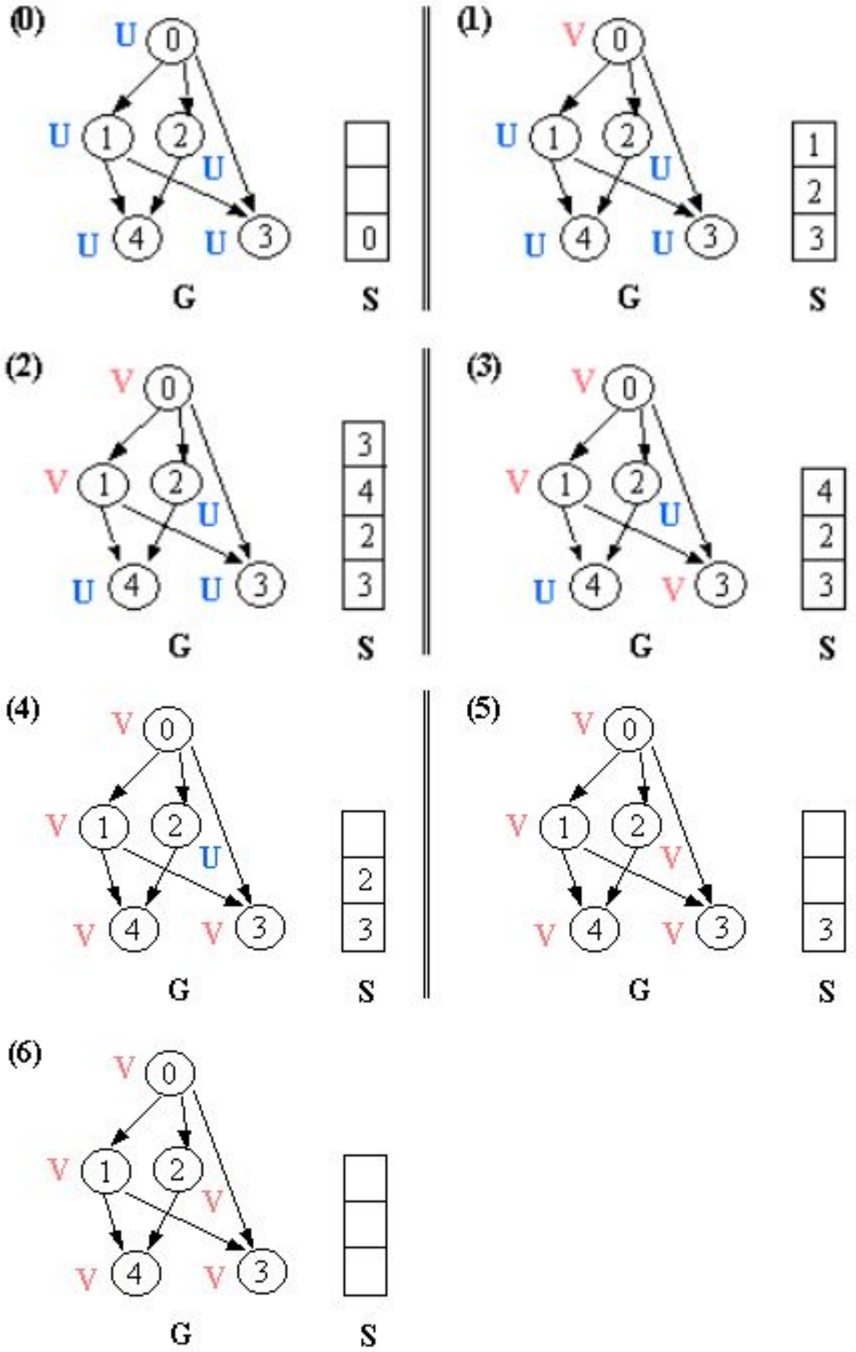
Depth First Search (DFS)

Algorithm DFS(G)

1. Initialize all vertices as "unvisited".
2. Let S be a stack.
3. Push the root on S
4. While (S not empty) {
5. Let n = Pop(S)
7 If (n is marked as "unvisited“) {
8. Mark n as "visited“
9. Print n
10. For each vertex v djacent to n
11. If v is marked as "unvisited”
12. push v on S.
13. end [end if]
14. end [end while]

Algorithm DFS(G)

1. Initialize all vertices as "unvisited".
2. Let S be a stack.
3. Push the root on S
4. While (S not empty) {
5. Let n = Pop(S)
7 If (n is marked as "unvisited“) {
8. Mark n as "visited“
9. Print n
10. For each vertex v djacent to n
11. If v is marked as "unvisited”
12. push v on S.
13. end [end if]
14. end [end while]



Graph Traversal Uses

- In addition to finding paths, we can use graph traversals to answer:
  - What are all the vertices reachable from a starting vertex?
  - Is an graph connected?
- What if we want to actually output the path?
  - Instead of just “marking” a node, store the previous node along the path
  - When you reach the goal, follow path fields back to where you started (and then reverse the answer)
  - If just wanted path length, could put the integer distance at each node instead once

Applications

- Google Maps
- Facebook
- Social networks
- World wide web
- Operating systems - resource allocation graph
- Computer games
- Robot planning
- Semantic networks