# GRAPHS

Swati Mali

swatimali@somaiya.edu

# Outline

- Graph- Concept
- Graph terminology: vertex, edge, adjacent, incident, degree, cycle, path, connected component, spanning tree
- Types of graphs: undirected, directed, weighted
- Graph representations: adjacency matrix, array adjacency lists, linked adjacency lists
- Graph search methods: breath-first, depth-first search

# What is a graph?

- A data structure that consists of a set of nodes (*vertices*) and a set of edges that relate the nodes to each other
- The set of edges describes relationships among the vertices

# Formal definition of graphs

- A graph $G$ is defined as follows:

$$G=(V,E)$$

*V(G):* a finite, nonempty set of vertices

*E(G):* a set of edges (pairs of vertices)

- Vertices are also called nodes and points.
- Each edge connects two vertices.
- Edges are also called arcs and lines.
- Vertices $i$ and $j$ are adjacent vertices iff ($i$, $j$) is an edge in the graph
- The edge ($i$, $j$) is incident on the vertices $i$ and $j$

# Graphs

- Undirected edge has no orientation (no arrow head)
- Directed edge has an orientation (has an arrow head)
- Undirected graph – all edges are undirected
- Directed graph – all edges are directed

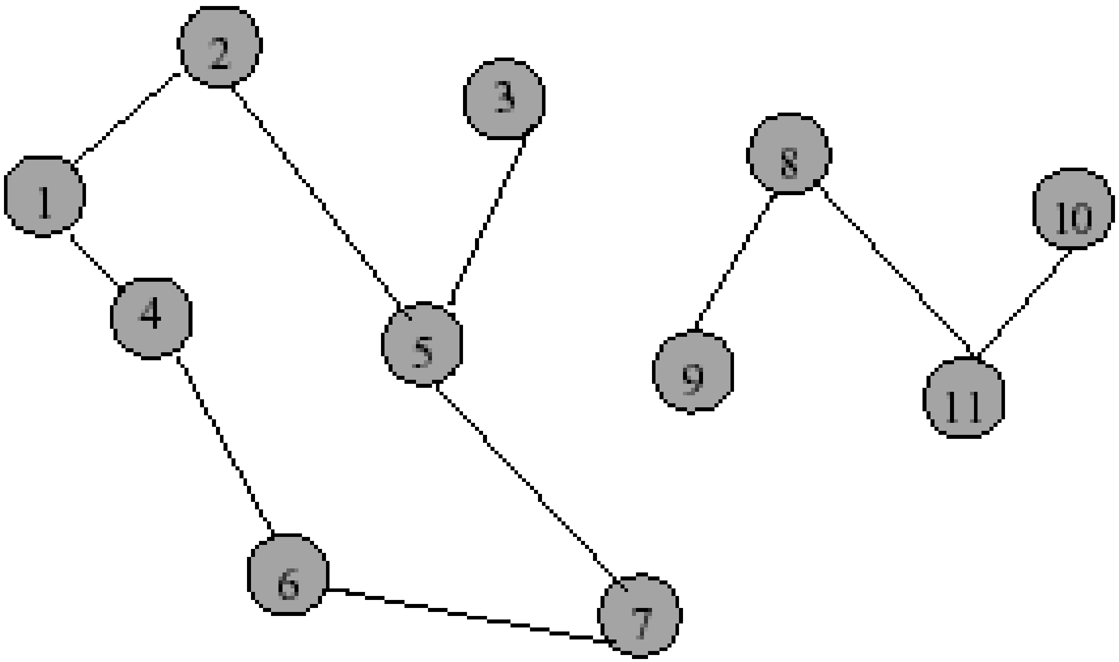
$$u \;\text{———}\; v$$

**undirected edge**

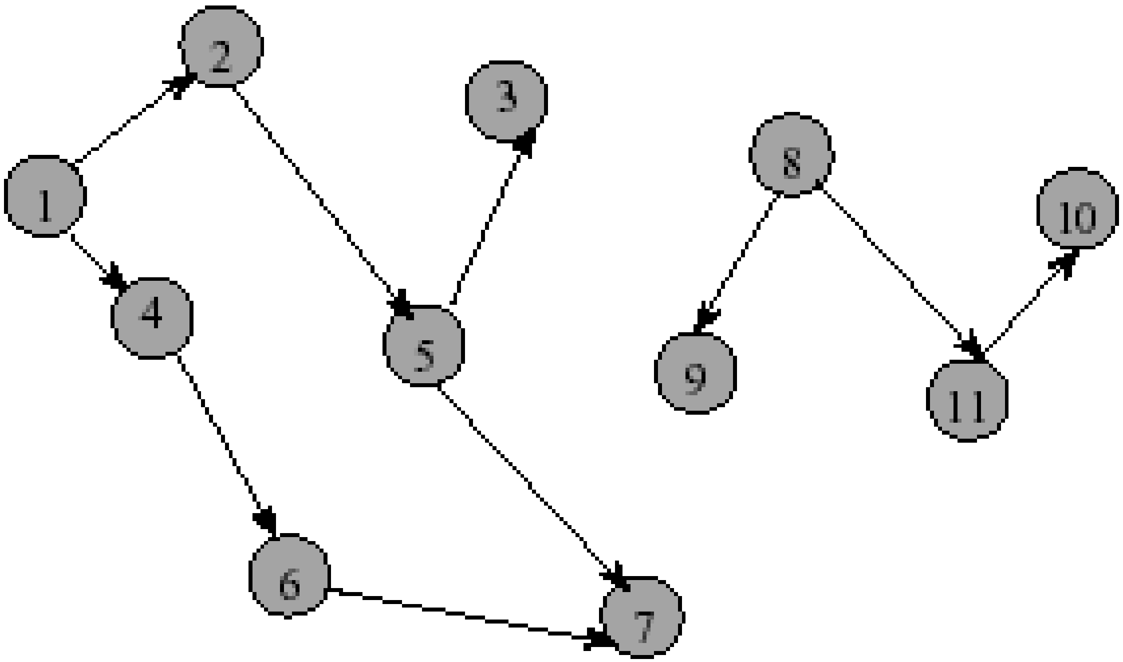
$$u \longrightarrow v$$

**directed edge**

# Undirected Graph

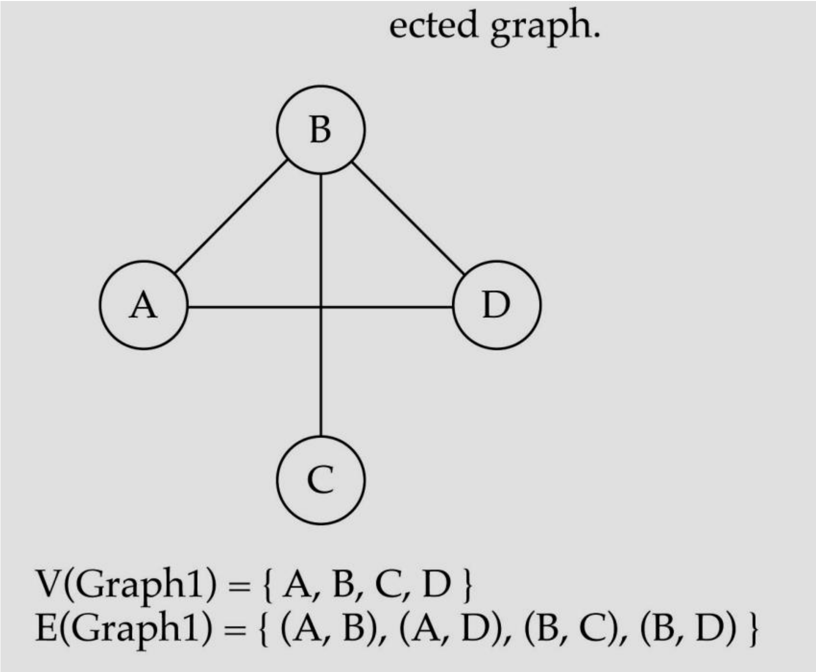


# Directed Graph (Digraph)



# Directed vs. undirected graphs

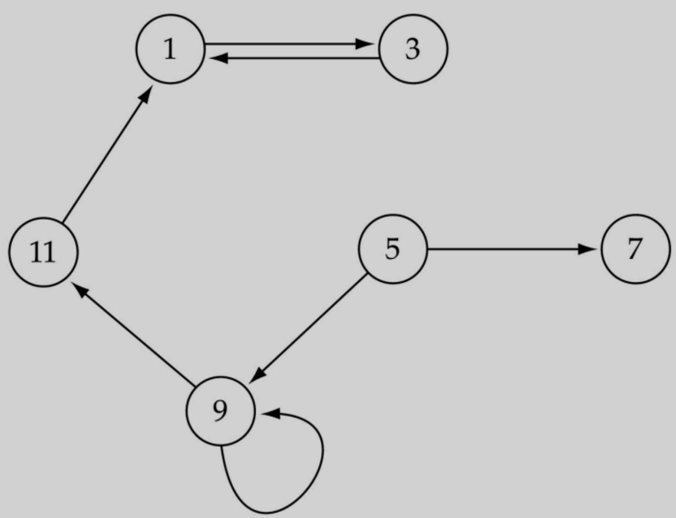
- When the edges in a graph have no direction, the graph is called *undirected*



# Directed vs. undirected graphs (cont.)

- When the edges in a graph have a direction, the graph is called *directed* (or *digraph*)

(b) Graph2 is a directed graph.



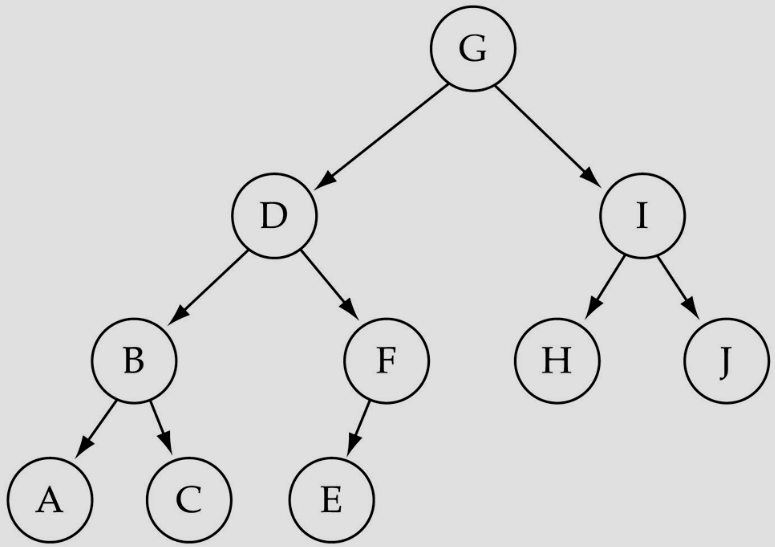

V(Graph2) = { 1, 3, 5, 7, 9, 11 }

E(Graph2) = {(1,3) (3,1) (5,9) (9,11) (5,7)  1), (9, 9), (11, 1) }

# Trees vs graphs

- Trees are special cases of graphs!!

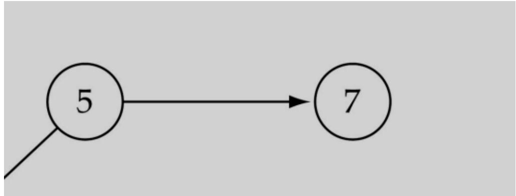
(c) Graph3 is a directed graph.




V(Graph3) = { A, B, C, D, E, F, G, H, I, J }
E(Graph3) = { (G, D), (G, J), (D, B), (D, F) (I, H), (I, J), (B, A), (B, C), (F, E) }

# Graph terminology

- Adjacent nodes: two nodes are adjacent if they are connected by an edge




5 is adjacent to 7
7 is adjacent from 5

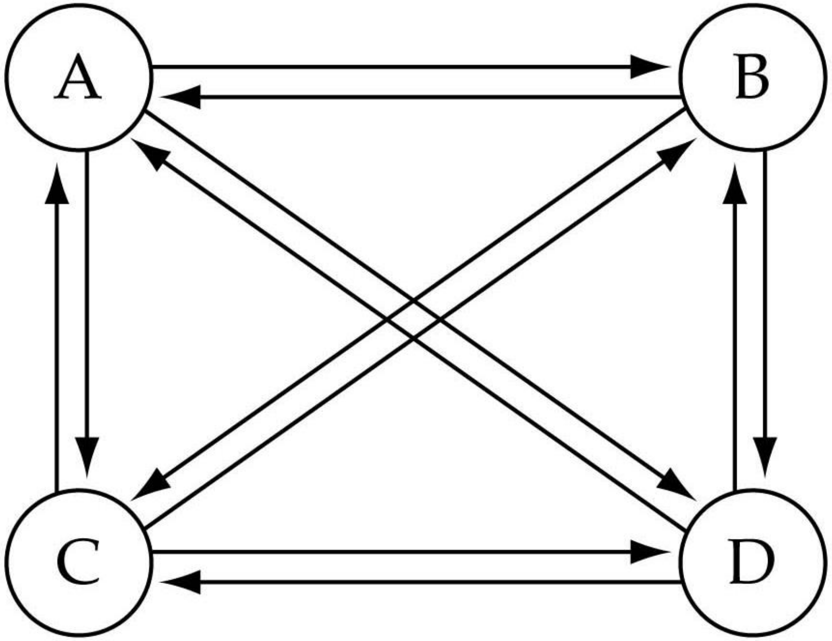
- Path: a sequence of vertices that connect two nodes in a graph
- Complete graph: a graph in which every vertex is directly connected to every other vertex

# Graph terminology (cont.)

- What is the number of edges in a complete directed graph with N vertices?

*N * (N-1)*

$$O(N^2)$$



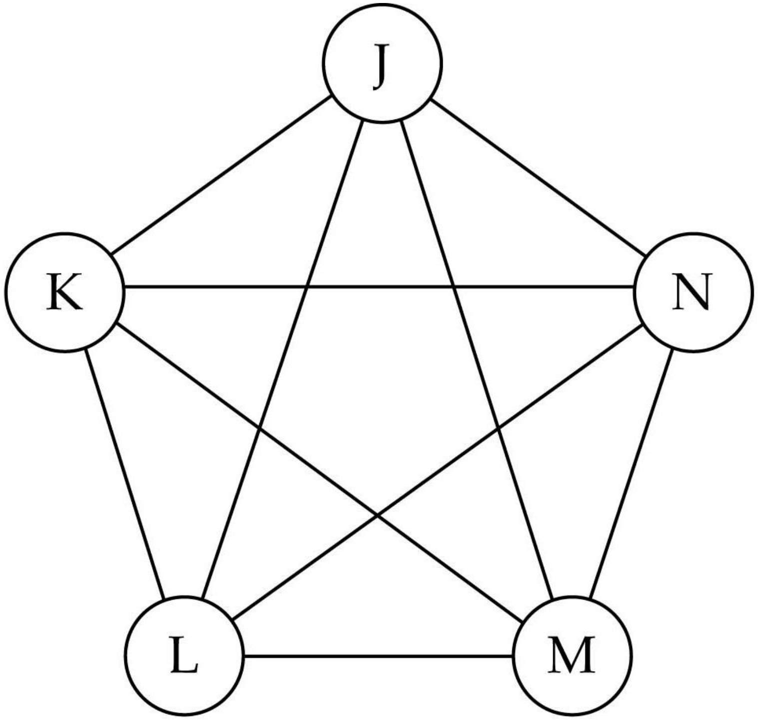

(a) Complete directed graph.

# Graph terminology (cont.)

- What is the number of edges in a complete undirected graph with N vertices?

*N * (N-1) / 2*

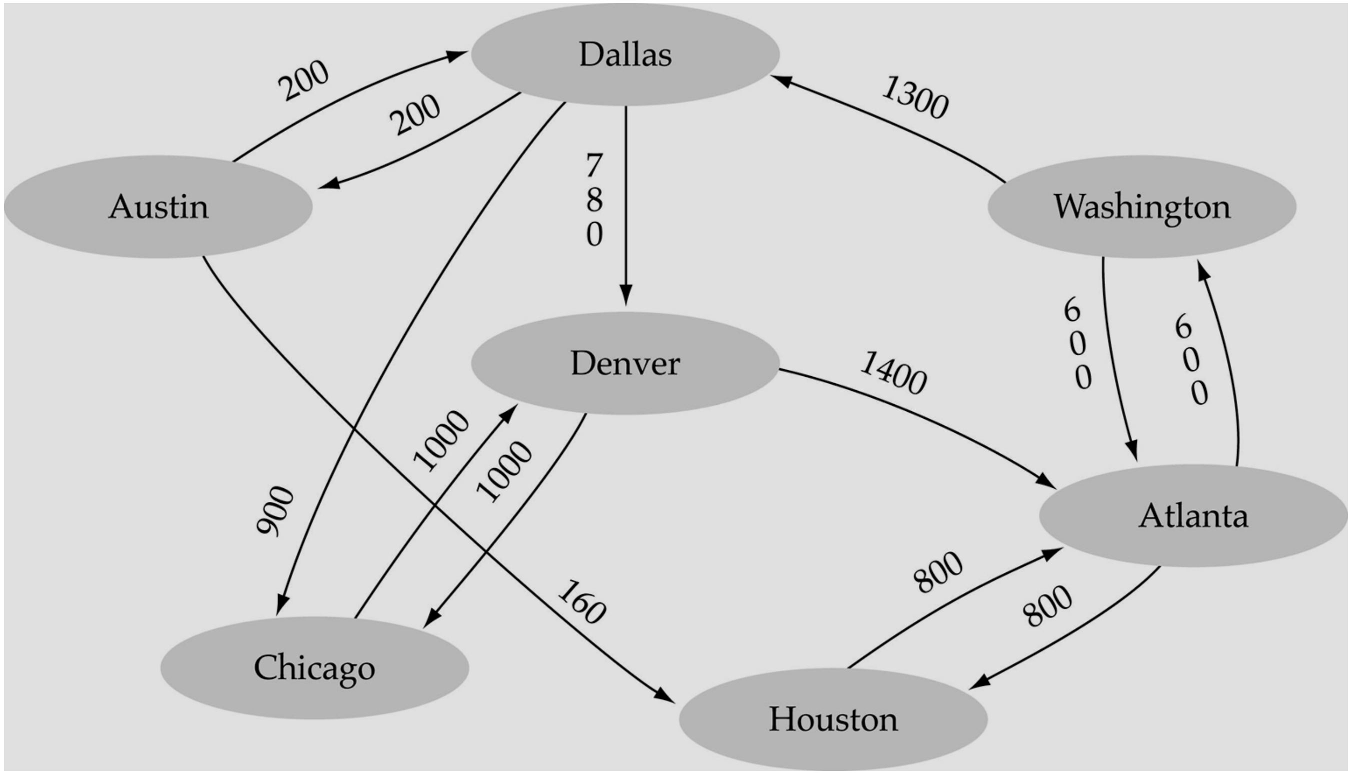
$O(N^2)$



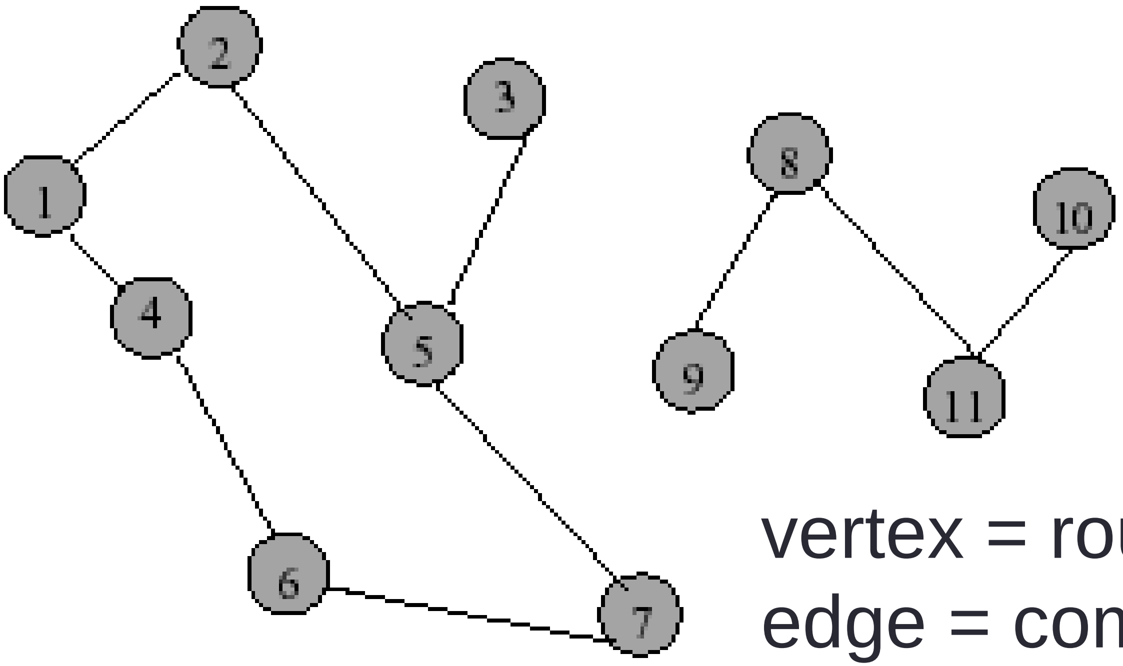

(b) Complete undirected graph.

# Graph terminology (cont.)

- Weighted graph: a graph in which each edge carries a value

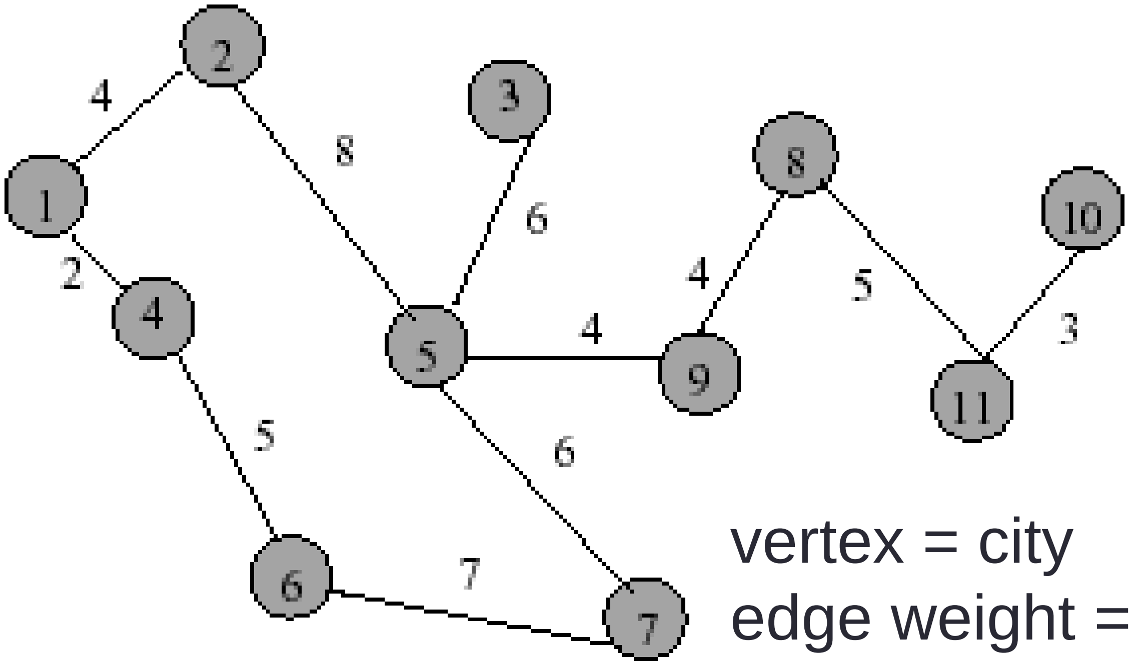


# Applications – Communication Network




vertex = router
edge = communication link

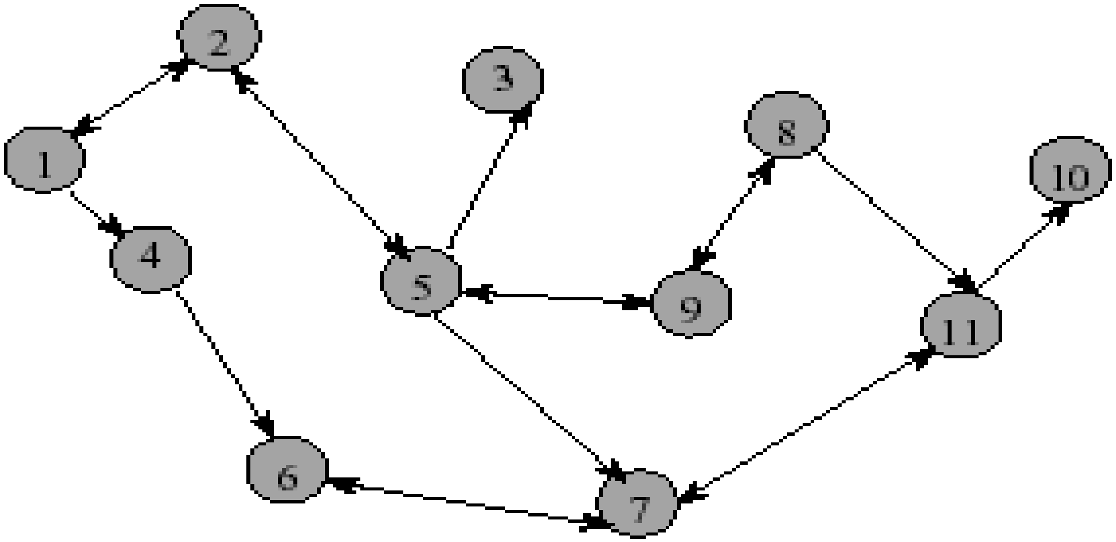
# Applications - Driving Distance/Time Map




vertex = city

edge weight = driving distance/time

# Applications - Street Map




- Streets are one- or two-way.
- A single directed edge denotes a one-way street
- A two directed edge denotes a two-way street
- Read Example 16.1 and see Figure 16.2

# Path

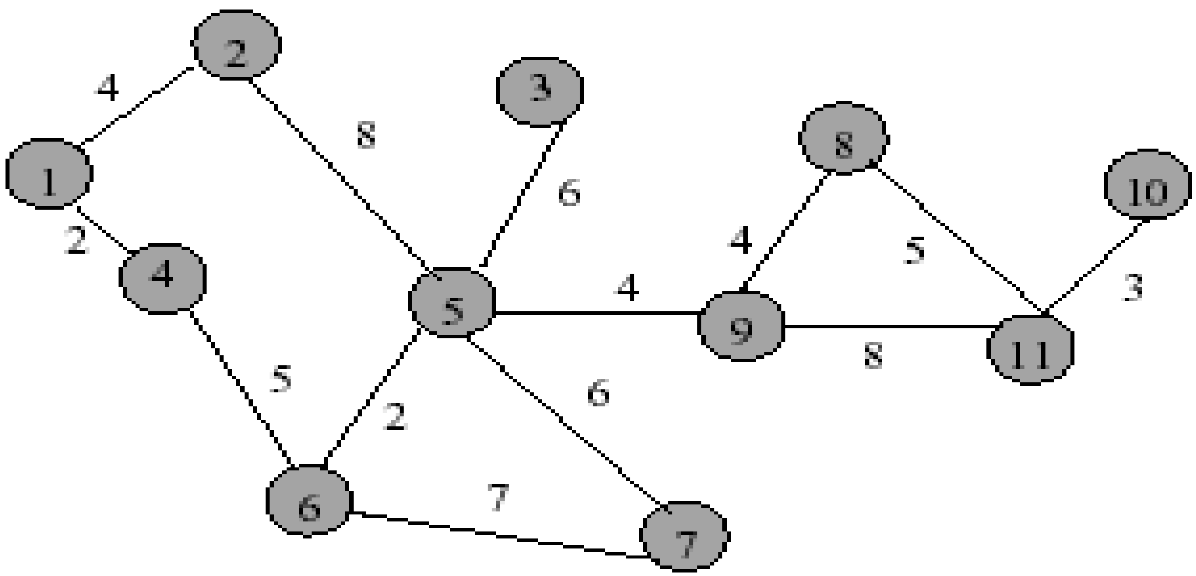
- A sequence of vertices $P = i_1, i_2, \ldots, i_k$ is an $i_1$ to $i_k$ **path** in the graph $G=(V, E)$ iff the edge $(i_j, i_{j+1})$ is in $E$ for every $j$, $1 \leq j < k$
- What are possible paths in Figure 16.2(b)?

# Simple Path

- A **simple path** is a path in which all vertices, except possibly in the first and last, are different

# Length (Cost) of a Path

- Each edge in a graph may have an associated **length (or cost).** The length of a path is the sum of the lengths of the edges on the path
- What is the length of the path 5, 9, 11, 10?



# Subgraph & Cycle

- Let $G = (V, E)$ be an undirected graph
- A graph $H$ is a **subgraph** of graph $G$ iff its vertex and edge sets are subsets of those of $G$
- A **cycle** is a simple path with the same start and end vertex
- List all cycles of the graph of Figure 16.1(a)?
  - 1, 2, 3, 1
  - 1, 4, 3, 1
  - 1, 2, 3, 4, 1

# Graph Properties

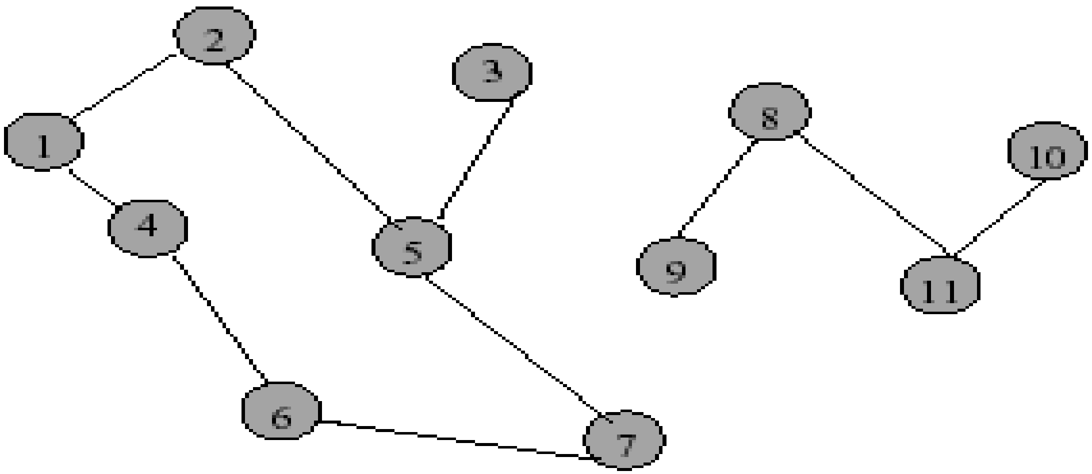
# Number of Edges – Undirected Graph

- Each edge is of the form *(u,v)*, u != v.
- The no. of possible 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 an undirected graph is n*(n-1)/2
- Thus, the 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.
- The no. of possible 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 directed graph is n*(n-1)
- Thus, the number of edges in a directed graph is **≤ n*(n-1)**

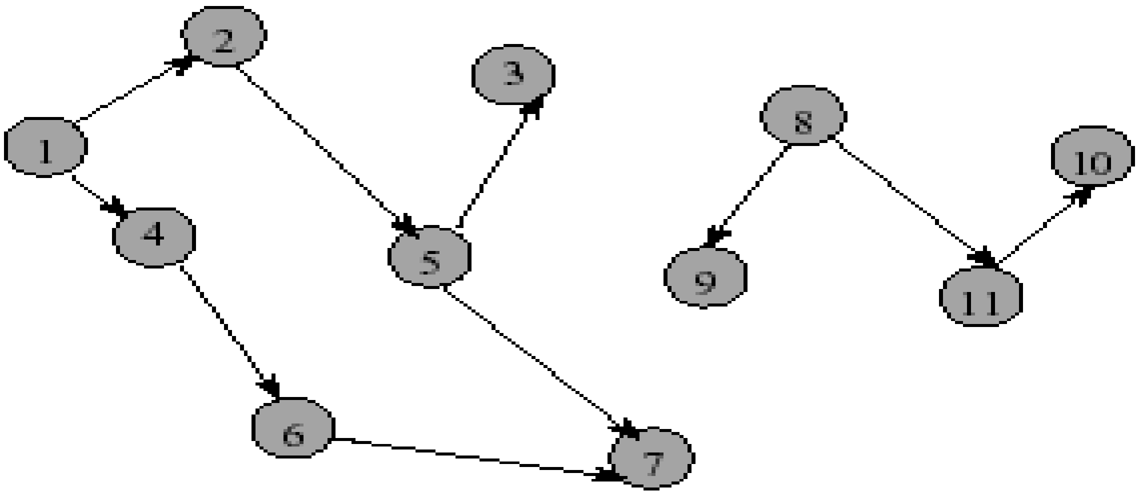
# Vertex Degree




- The **degree** of vertex *i* is the no. of edges incident on vertex *i*.

e.g., degree(2) = 2, degree(5) = 3, degree(3) = 1

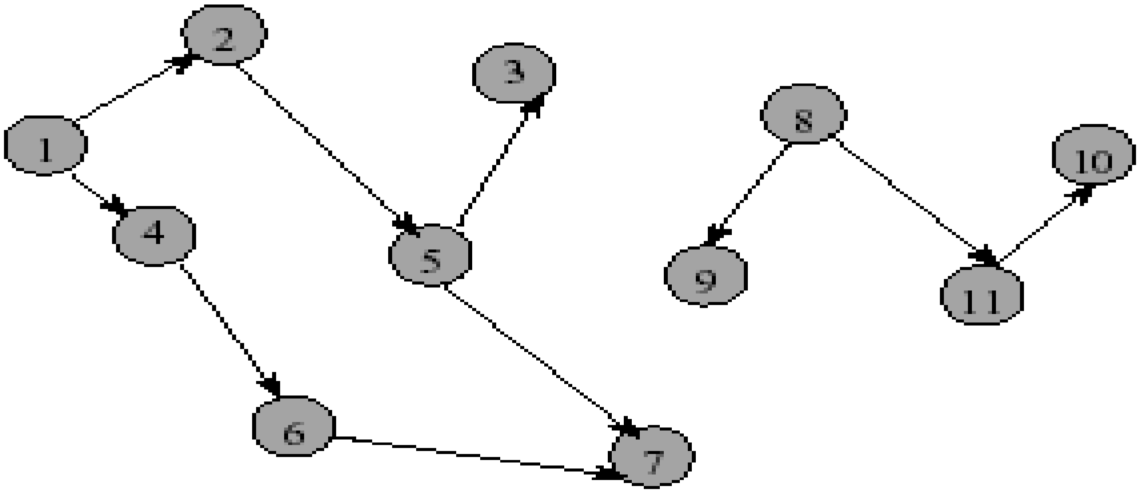
# In-Degree of a Vertex




- **In-degree** of vertex *i* is the number of edges incident to *i* (i.e., the number of incoming edges).

e.g., indegree(2) = 1, indegree(8) = 0

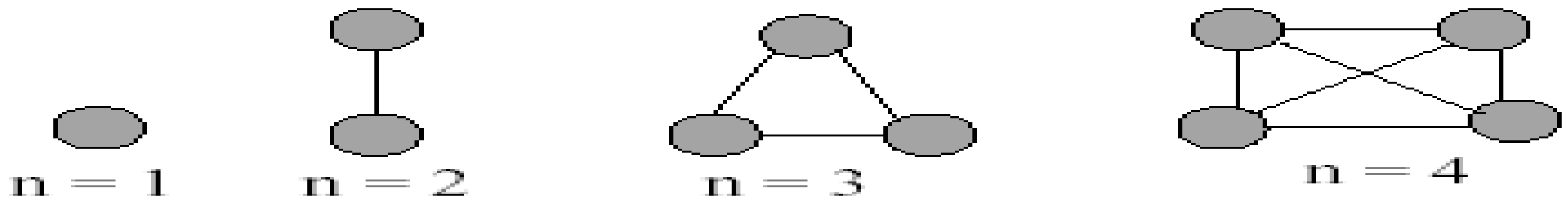
# Out-Degree of a Vertex



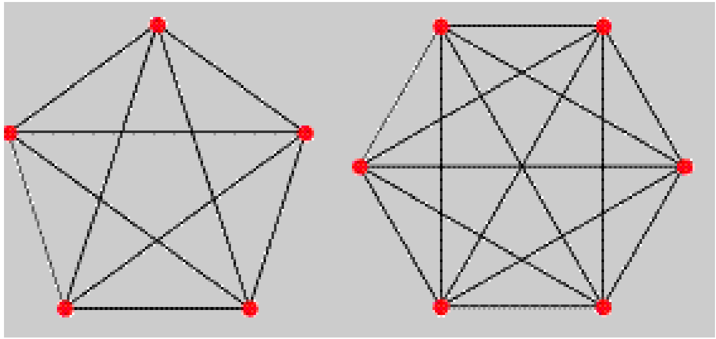

- **Out-degree** of vertex *i* is the number of edges incident from *i* (i.e., the number of outgoing edges).
- e.g., outdegree(2) = 1, outdegree(8) = 2

# Complete Undirected Graphs

- A **complete undirected graph** has n(n-1)/2 edges (i.e., all possible edges) and is denoted by $K_n$



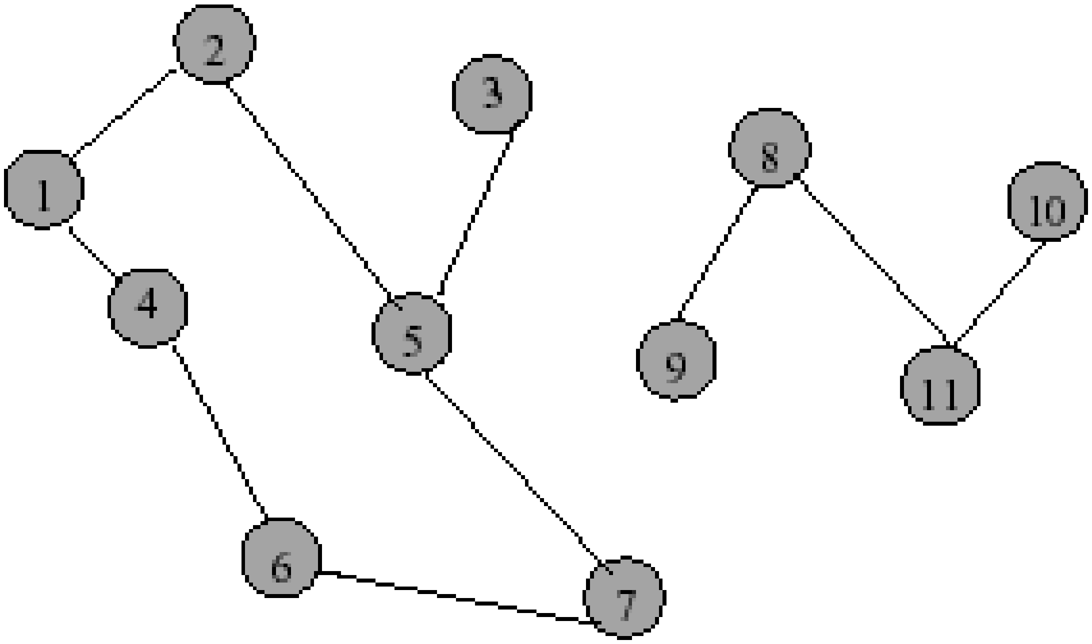

- What would a complete undirected graph look like when n=5? When n=6?



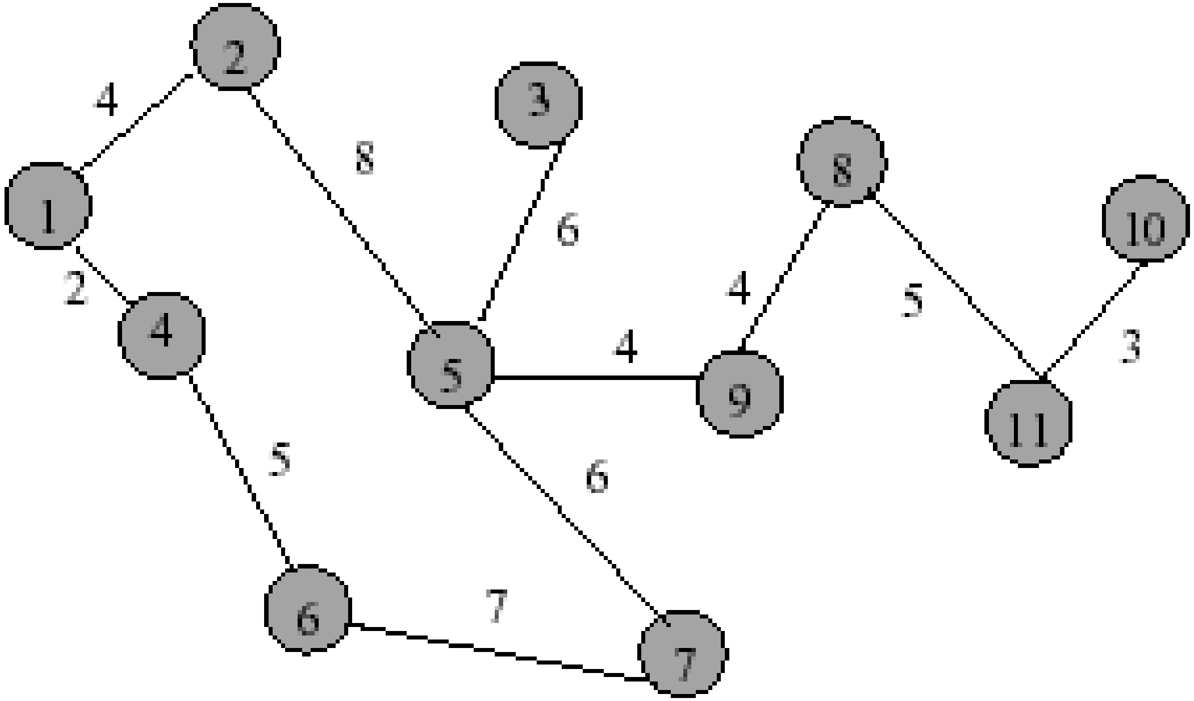
# Connected Graph

- Let $G = (V, E)$ be an undirected graph
- $G$ is **connected** iff there is a path between every pair of vertices in $G$

# Example of Not Connected



# Example of Connected Graph

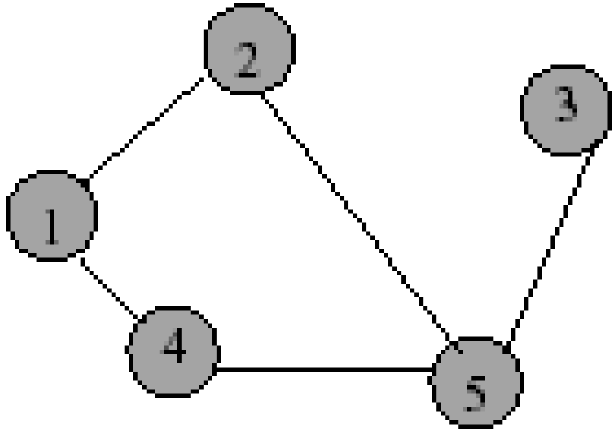


# Representation of Unweighted Graphs

- The most frequently used representations for unweighted graphs are
  - Adjacency Matrix
  - Linked adjacency lists
  - Array adjacency lists

# Adjacency Matrix

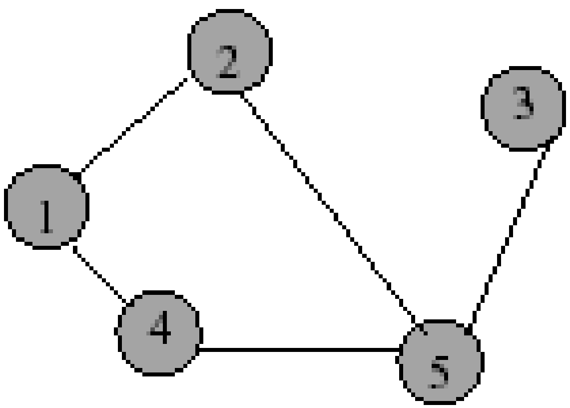
- 0/1 n x n matrix, where n = # of vertices
- A(i, j) = 1 iff (i, j) is an edge.




| | 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

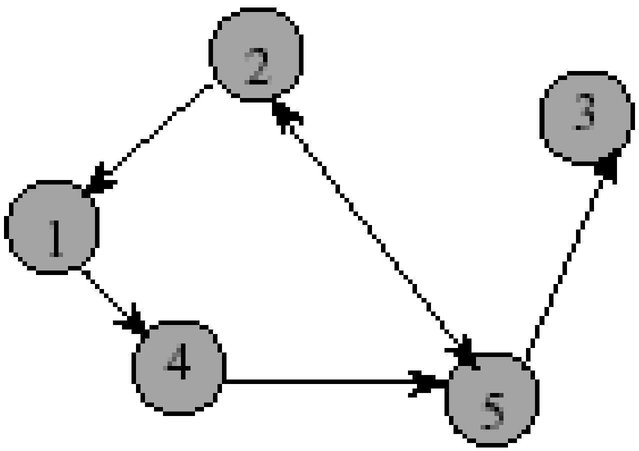
- Diagonal entries are zero.
- Adjacency matrix of an undirected graph is symmetric (A(i,j) = A(j,i) for all i and j).




| | 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 for Digraph

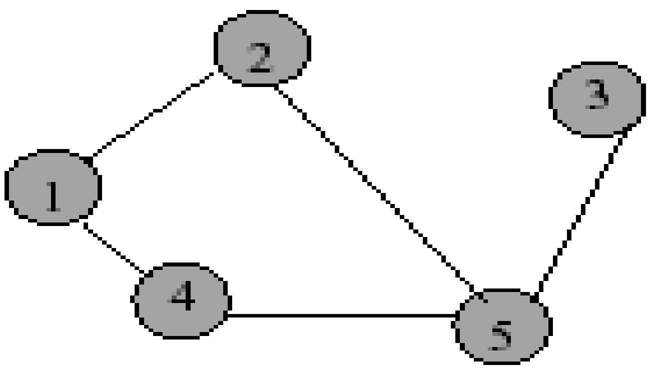
- Diagonal entries are zero.
- Adjacency matrix of a digraph need not be symmetric.




| | 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 |

# Adjacency Lists

- Adjacency list for vertex *i* is a linear list of vertices adjacent from vertex *i*.
- An array of n adjacency lists for each vertex of the graph.




aList[1] = (2,4)

aList[2] = (1,5)

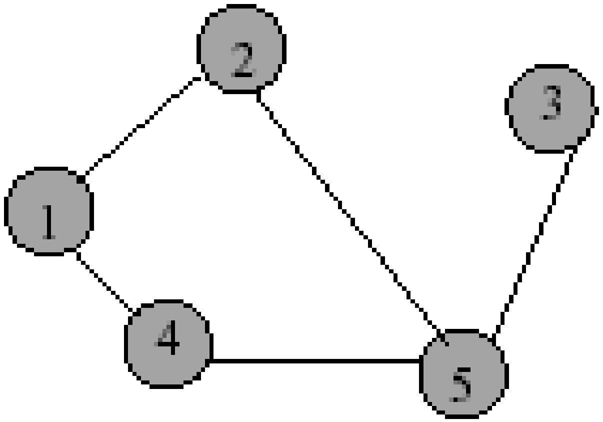
aList[3] = (5)

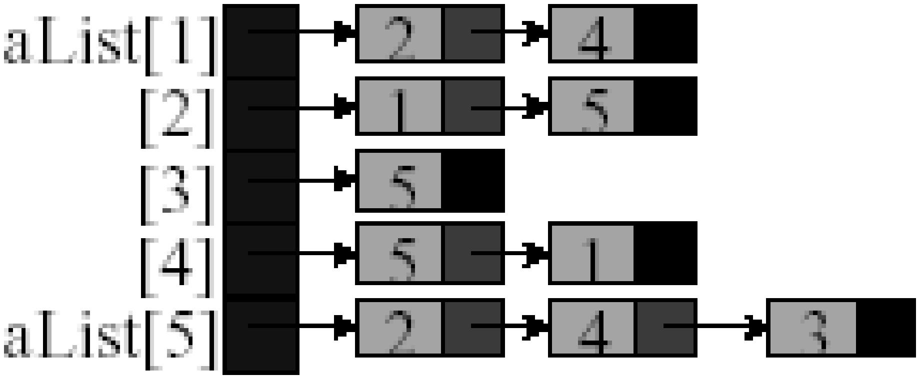
aList[4] = (5,1)

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

# Linked Adjacency Lists

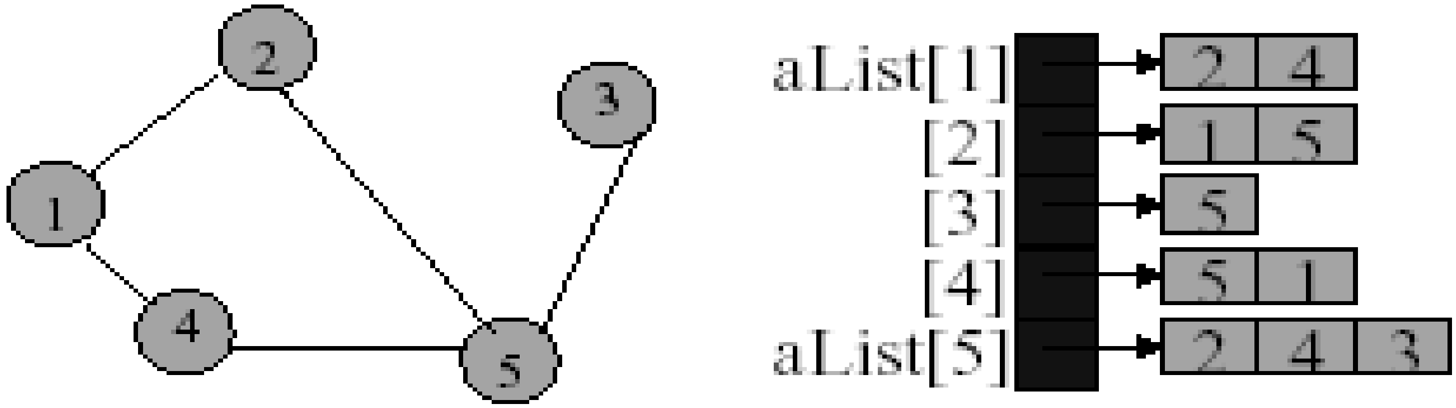
- Each adjacency list is a chain.
  Array length = n.
  # of chain nodes = 2e (undirected graph)
  # of chain nodes = e (digraph)



# Array Adjacency Lists

- Each adjacency list is an array list.
  Array length = n.
  # of chain nodes = 2e (undirected graph)
  # of chain nodes = e (digraph)



# Representation of Weighted Graphs

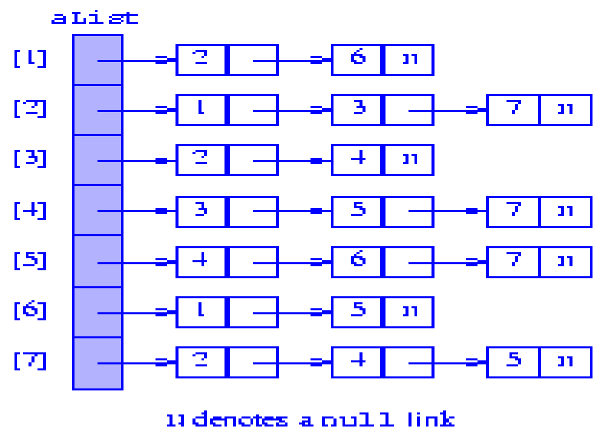
- **Weighted graphs** are represented with simple extensions of those used for unweighted graphs
- The **cost-adjacency-matrix** representation uses a matrix C just like the adjacency-matrix representation does
- Cost-adjacency matrix: C(i, j) = cost of edge (i, j)
- Adjacency lists: each list element is a pair (adjacent vertex, edge weight)

# For the digraph Figure 16.2(b)

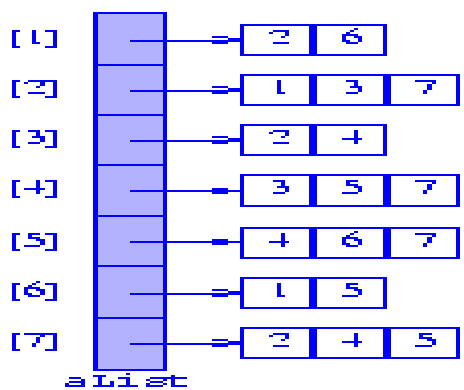
(a) adjacency matrix

$$
\begin{array}{c|ccccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
5 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
6 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
7 & 0 & 1 & 0 & 1 & 1 & 0 & 0
\end{array}
$$

(b) Linked adjacency list




(c) Array adjacency list



# Graph Traversals (Search)

- We have covered some of these with binary trees
  - Breadth-first search (BFS)
  - Depth-first search (DFS)
- A traversal (search):
  - An algorithm for systematically exploring a graph
  - Visiting (all) vertices
  - Until finding a goal vertex or until no more vertices

Only for connected graphs

# Breadth-first search

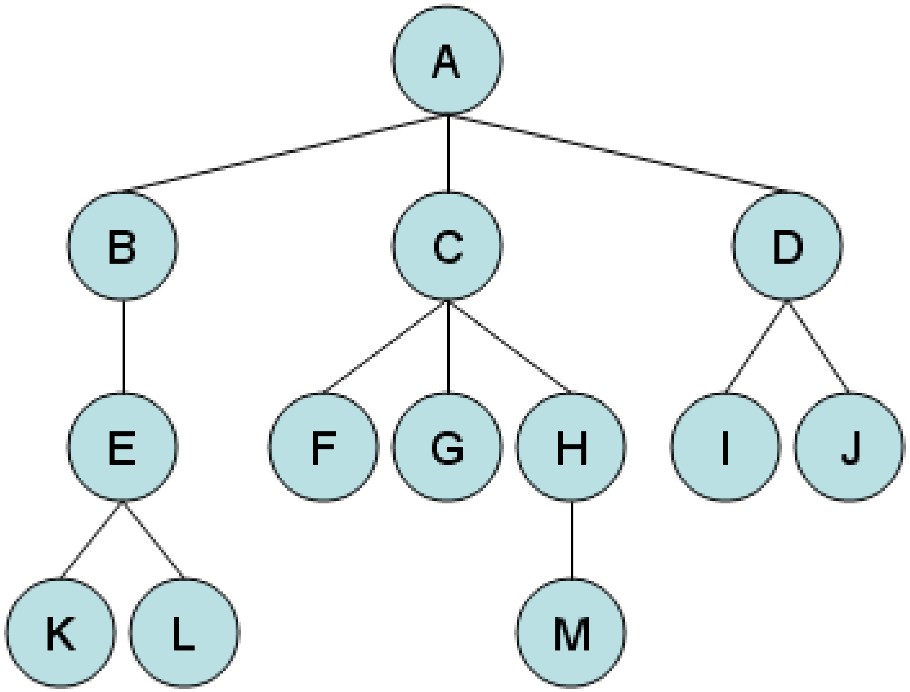
- One of the simplest algorithms
- Also one of the most important
  - It forms the basis for MANY graph algorithms

# BFS: Level-by-level traversal

- Given a starting vertex s
- Visit all vertices at increasing distance from s
  - Visit all vertices at distance k from s
  - Then visit all vertices at distance k+1 from s
  - Then ….

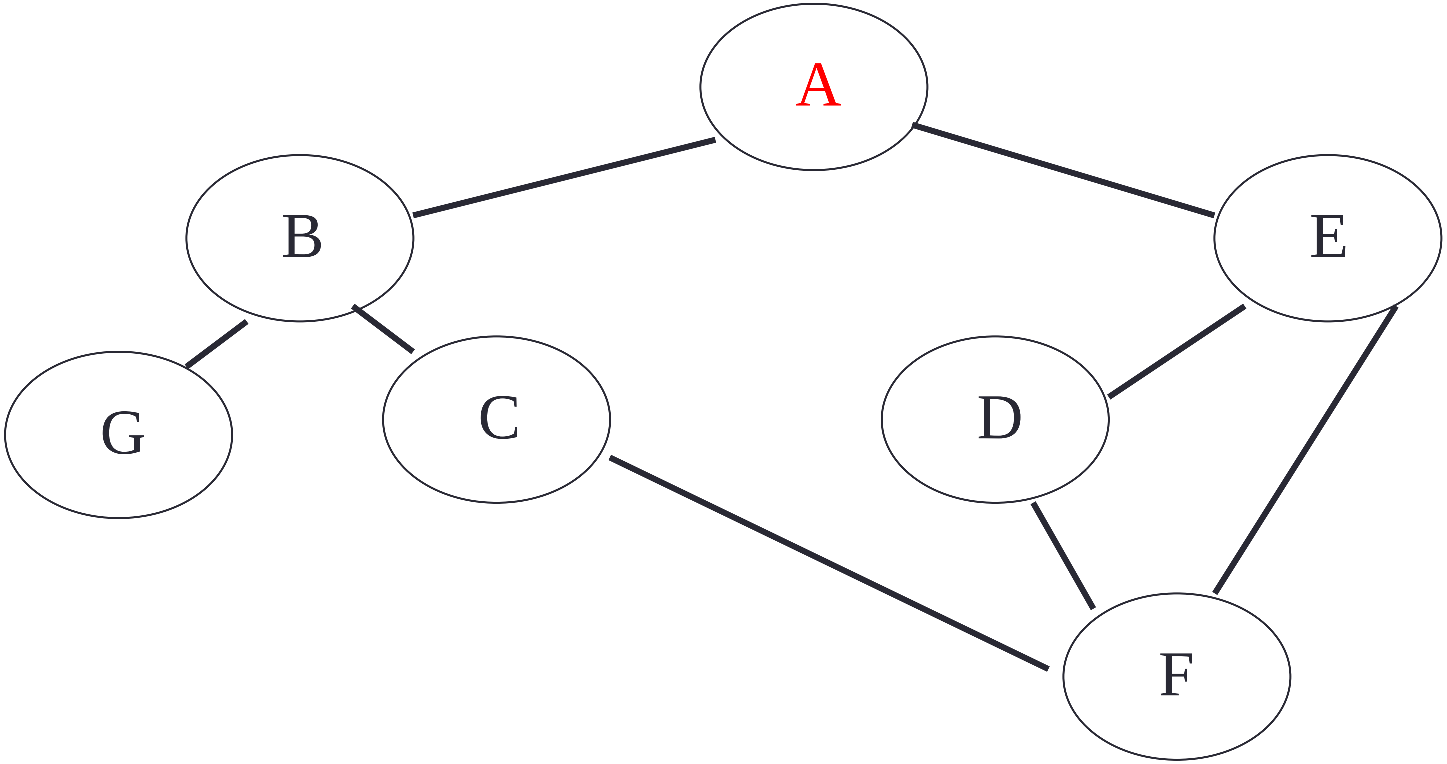
# BFS in a tree

BFS: visit all siblings before their descendants




A B C D E F G H I J K L M

# BFS: Graph

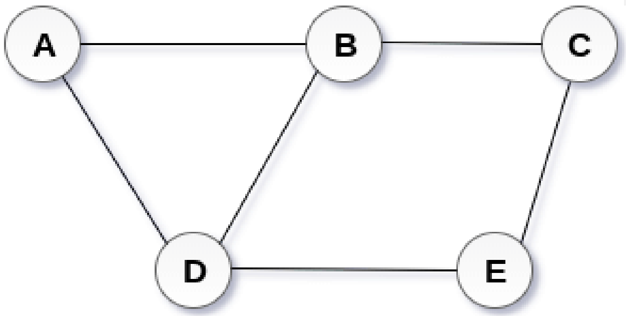



A B E G C D F

# BFS(graph g, vertex s)

```
unmark all vertices in G
q ← new queue
mark s // s is starting vertex
enqueue(q, s)
while (not empty(q))
   curr ← dequeue(q)
   visit curr // e.g., print its data
   for each edge <curr, V>
      if V is unmarked
         mark V
         enqueue(q, V)
```

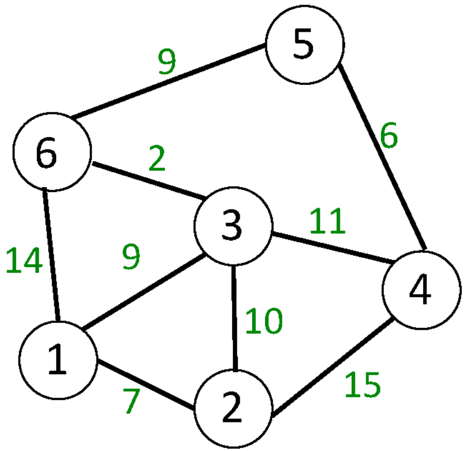
# BFS algorithm




**Undirected Graph**

Starting vertex = d

| Queue | Marked | Curr | BFS |
|---|---|---|---|
| {} | {} | - | - |
| {d} | {d} | d | d |
| {a, b, e} | {d, a,b,e} | a | d, a |
| {b,e} | {d,a,b,e} | b | d,a,b |
| {e,c} | {d,a,b,e,c} | e | d,a,b,e |
| {c} | {d,a,b,e,c} | c | d,a,b,e,c |
| {} | {d,a,b,e,c} | | |




| queue | Marked | Curr | BFS |
|---|---|---|---|
| {} | {} | | |
| ~~{1}~~ | {1} | 1 | 1 |
| {~~2~~,3,6} | {1,2,3,6} | 2 | 1, 2 |
| {~~3~~,6,4} | {1,2,3,6,4} | 3 | 1,2,3 |
| {~~6~~,4} | {1,2,3,6,4} | 6 | 1,2,3,6 |
| {~~4~~,5} | {1,2,3,6,4,5} | 4 | 1,2,3,6,4 |
| ~~{5}~~ | {1,2,3,6,4,5} | 5 | 1,2,3,6,4,5 |
| empty | algo terminates | | BFS= 1,2,3,6,4,5 |

# Interesting features of BFS

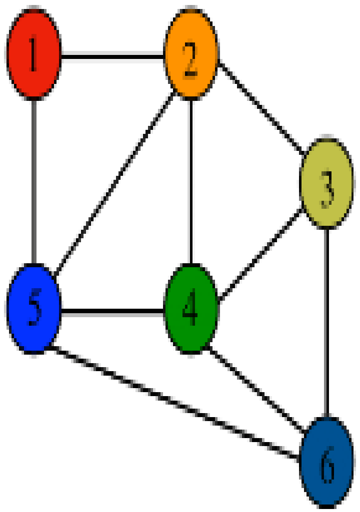
- Complexity: O(|V| + |E|)
  - All vertices put on queue exactly once
  - For each vertex on queue, we expand its edges
  - In other words, we traverse all edges once
- BFS finds shortest path from s to each vertex
  - Shortest in terms of number of edges
  - Why does this work?

  - Takes too much memory.
  - Runs out of memory before it runs out of time.

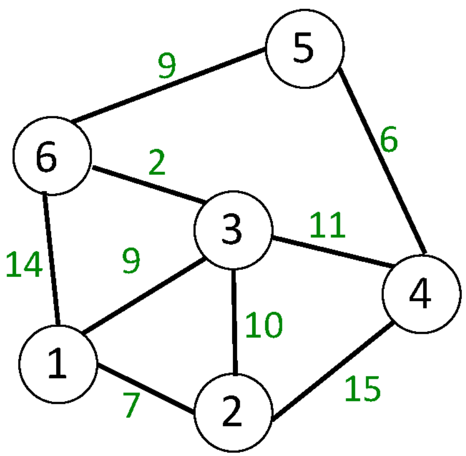
# BFS- Time Complexity

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 1100 | .11 seconds | 1 megabyte |
| 4 | 111,100 | 11 seconds | 106 megabytes |
| 6 | $10^7$ | 19 minutes | 10 gigabytes |
| 8 | $10^9$ | 31 hours | 1 terabytes |
| 10 | $10^{11}$ | 129 days | 101 terabytes |
| 12 | $10^{13}$ | 35 years | 10 petabytes |
| 14 | $10^{15}$ | 3,523 years | 1 exabyte |

**Figure 3.11** Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 10,000 nodes/second; 1000 bytes/node.




| stack | Marked | curr | DFS |
| --- | --- | --- | --- |
| {} | {} | - | - |
| {1} | {1} | 1 | 1 |
| {2,5} | {1,2,5} | 2 | 1,2 |
| {3,4,5} | {1,2,5,3,4} | 3 | 1,2,3 |
| {6,4,5} | {1,2,5,3,4,6} | 6 | 1,2,3,6 |
| {4,5} | {1,2,5,3,4,6} | 4 | 1,2,3,6,4 |
| {5} | {1,2,5,3,4,6} | 5 | 1,2,3,6,4,5 |
| Empty | {1,2,5,3,4,6} | - | DFS: 1,2,3,6,4,5 |




| Stack | Marked | Curr | DFS |
|---|---|---|---|
| {} | {} | | |
| {1} | {1} | 1 | 1 |
| {2,3,6} | {1,2,3,6} | 2 | 1,2 |
| {4,3,6} | {1,2,3,6,4} | 4 | 1,2,4 |
| {5,3,6} | {1,2,3,6,4,5} | 5 | 1,2,4,5 |
| {3,6} | {1,2,3,6,4,5} | 3 | 1,2,4,5,3 |
| {6} | {1,2,3,6,4,5} | 6 | 1,2,4,5,3,6 |
| empty | algo terminates | DFS: 1,2,4,5,3,6 | |

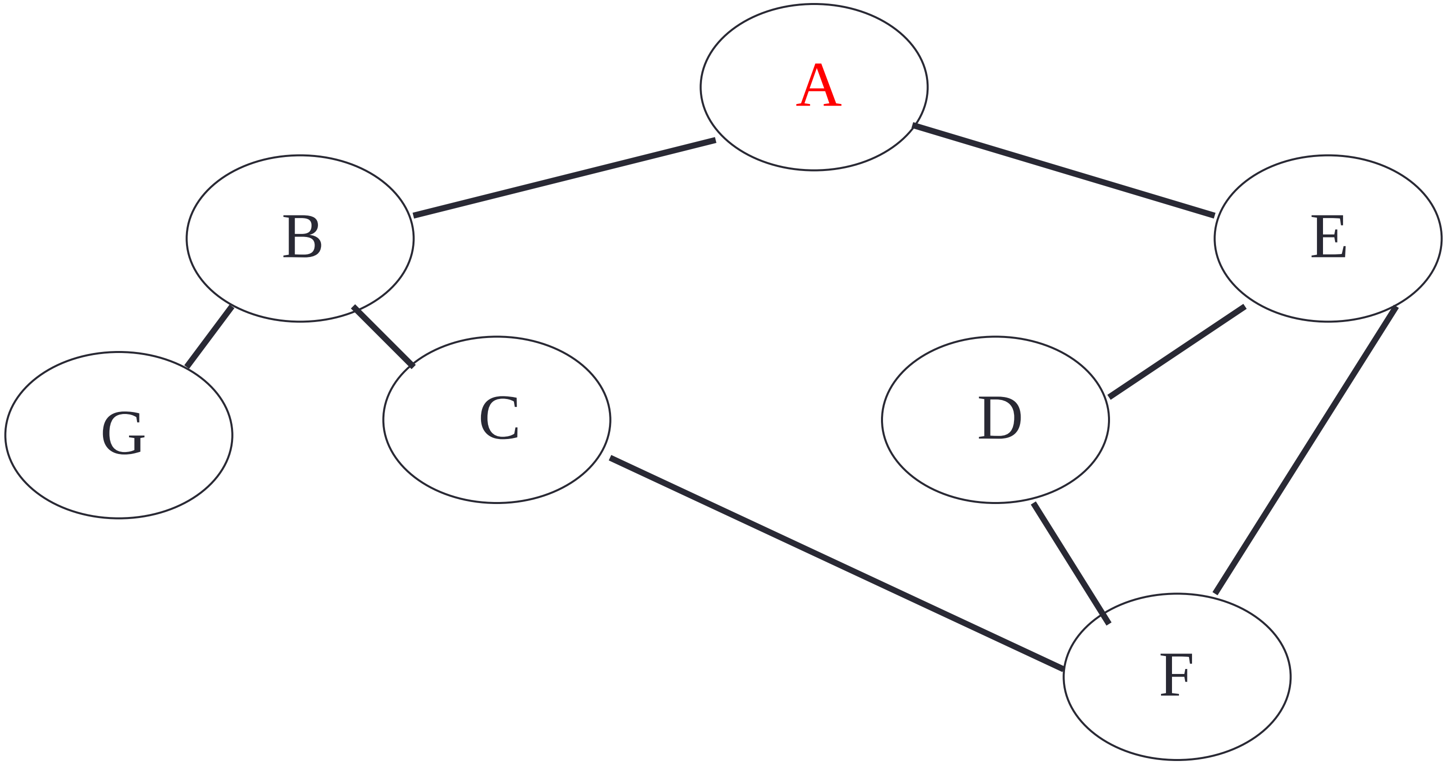
# Depth-first search

- Again, a simple and powerful algorithm
- Given a starting vertex s
- Pick an adjacent vertex, visit it.
  - Then visit one of its adjacent vertices
  - …..
  - Until impossible, then backtrack, visit another

# DFS(graph g, vertex s)

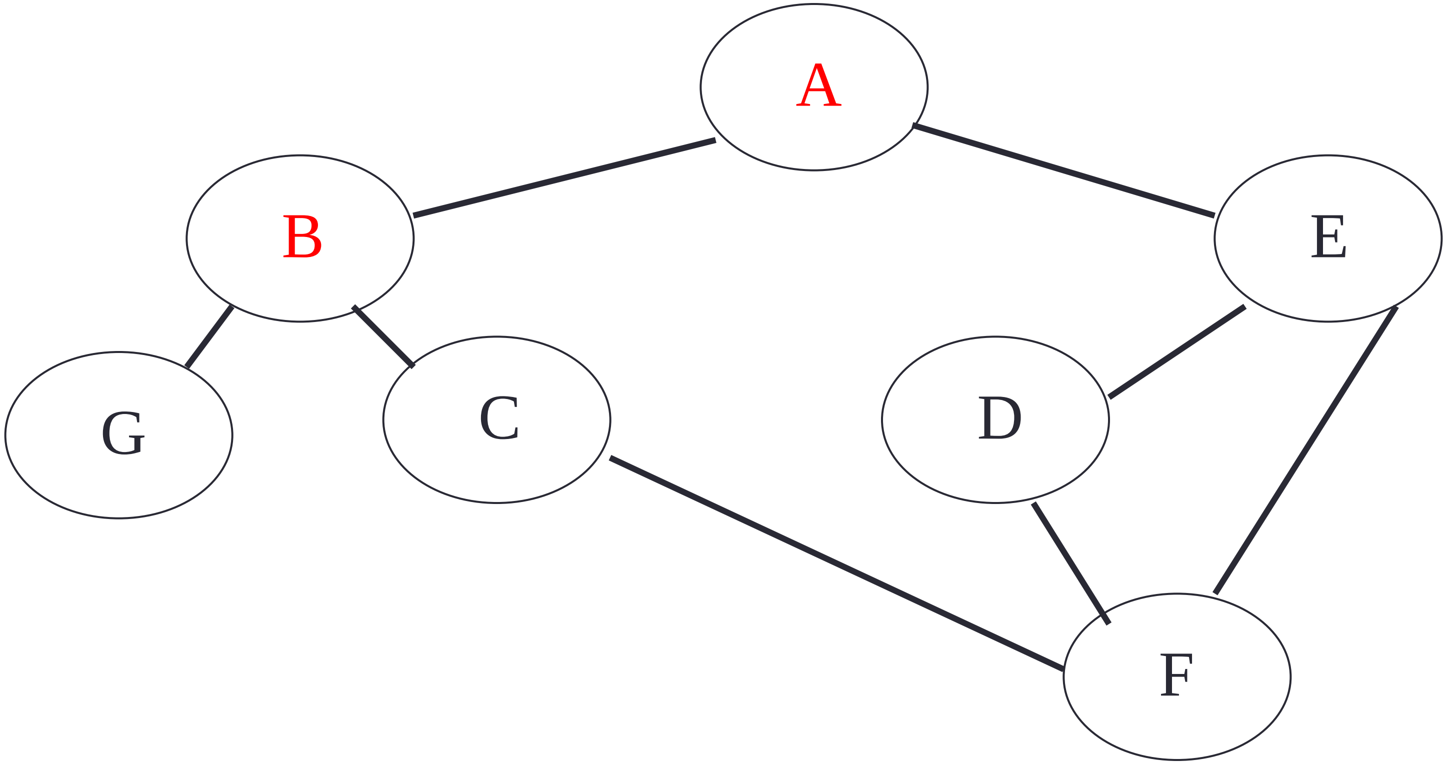
```
unmark all vertices in G
Stack ← new stack
mark s
Push(stack, s)
while (not empty(stack))
    curr ← pop(stack)
    visit curr // e.g., print its data
    for each edge <curr, V>
        if V is unmarked
            mark V
            push(stack, V)
  Print curr
```

# Current vertex: A




Start with A. Mark it.

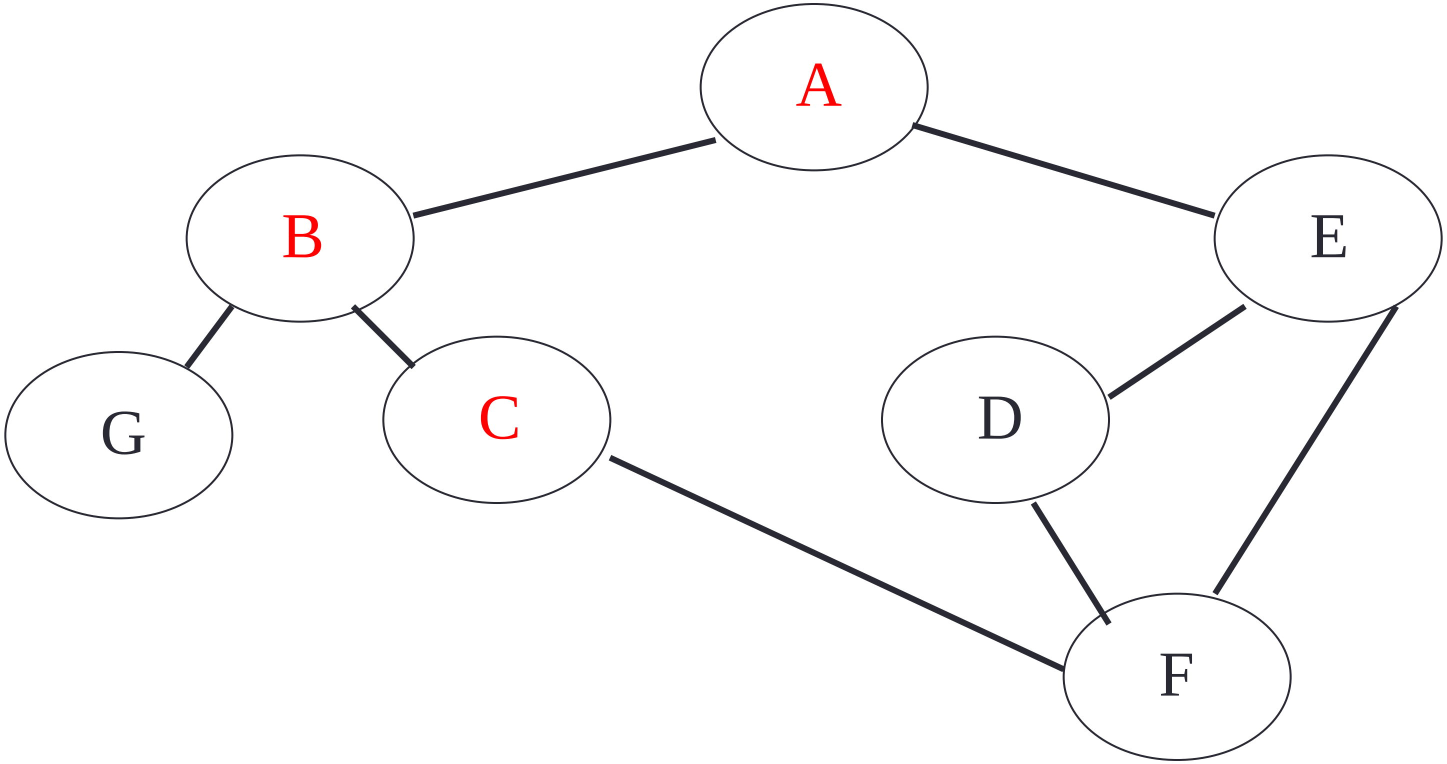
Current: B




Expand A’s adjacent vertices.  Pick one (B).

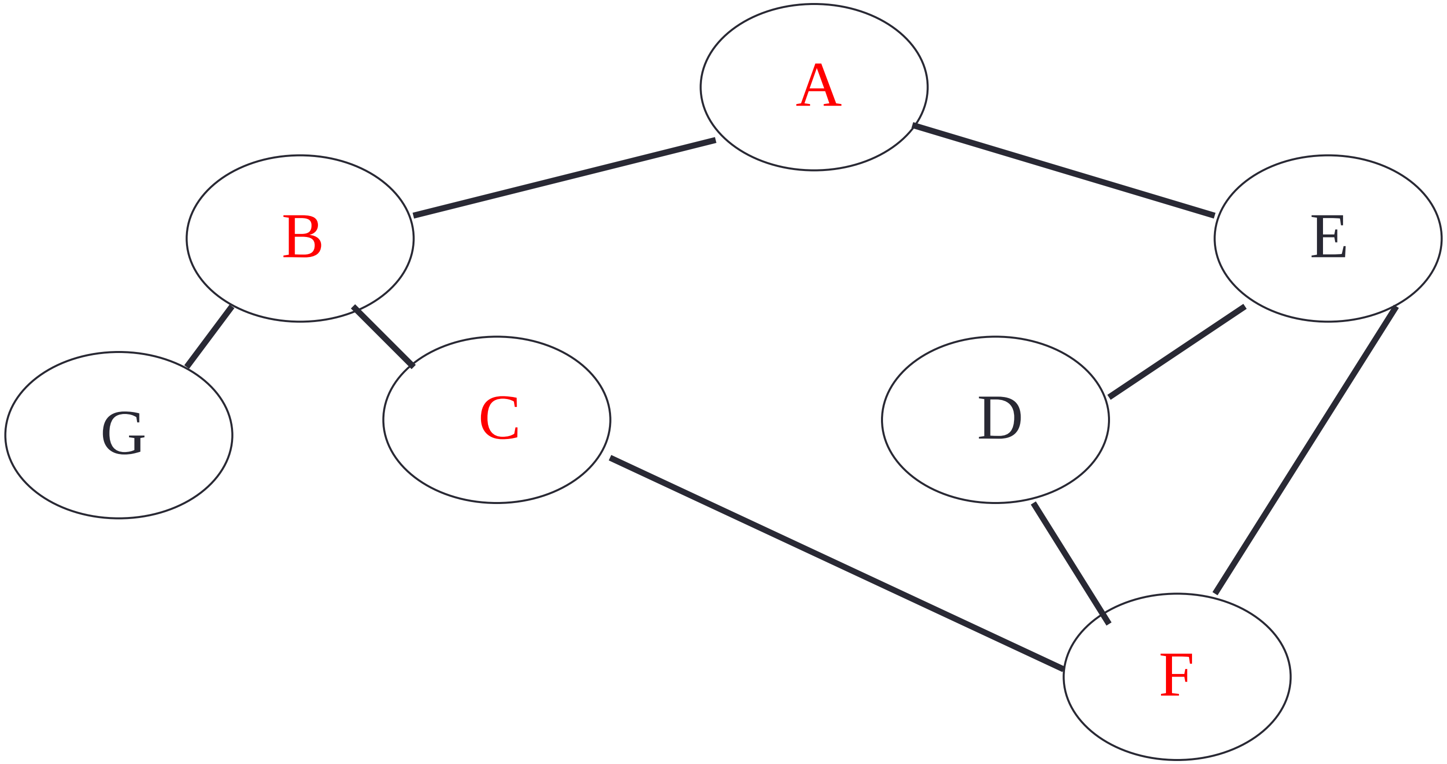
Mark it and re-visit.

Current: C




Now expand B, and visit its neighbor, C.

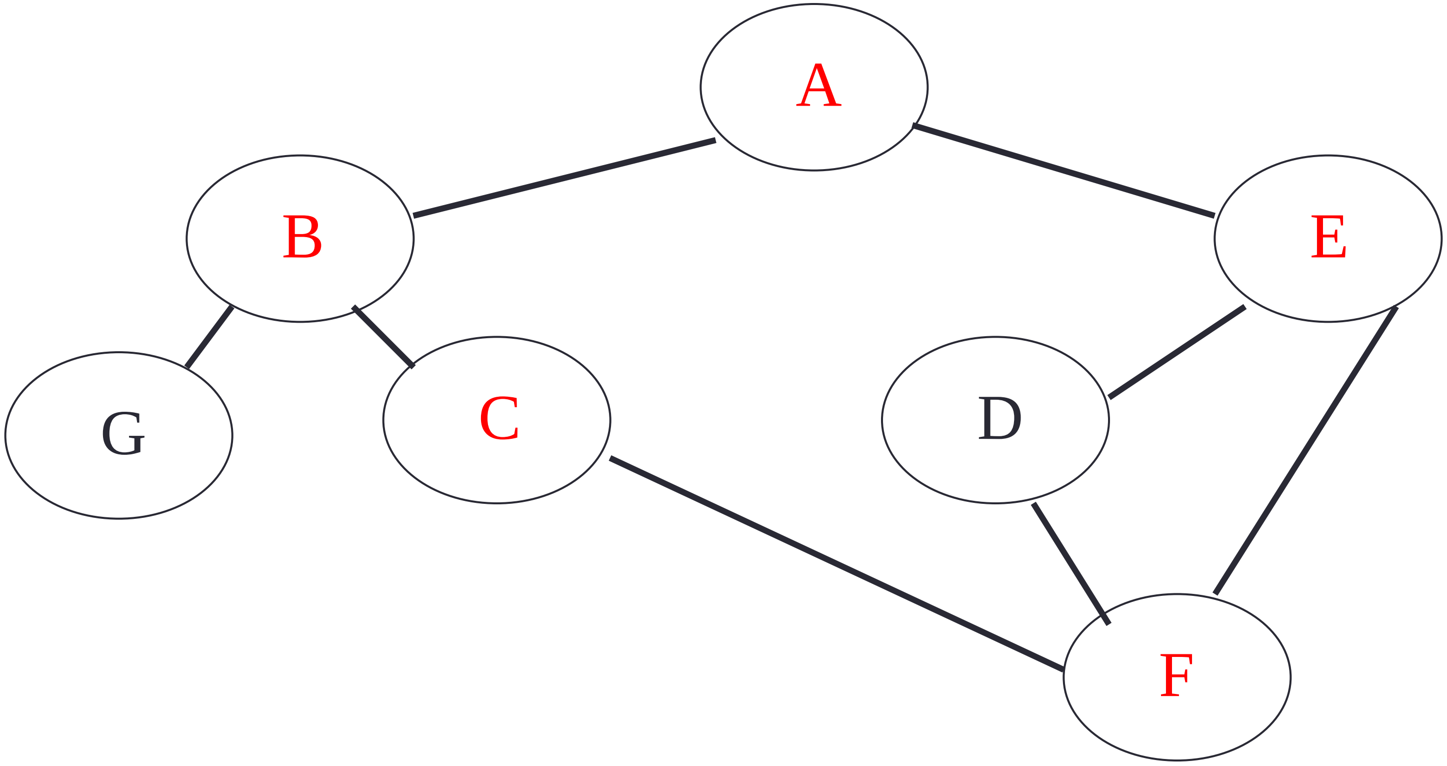
Current: F




Visit F.

Pick one of its neighbors, E.

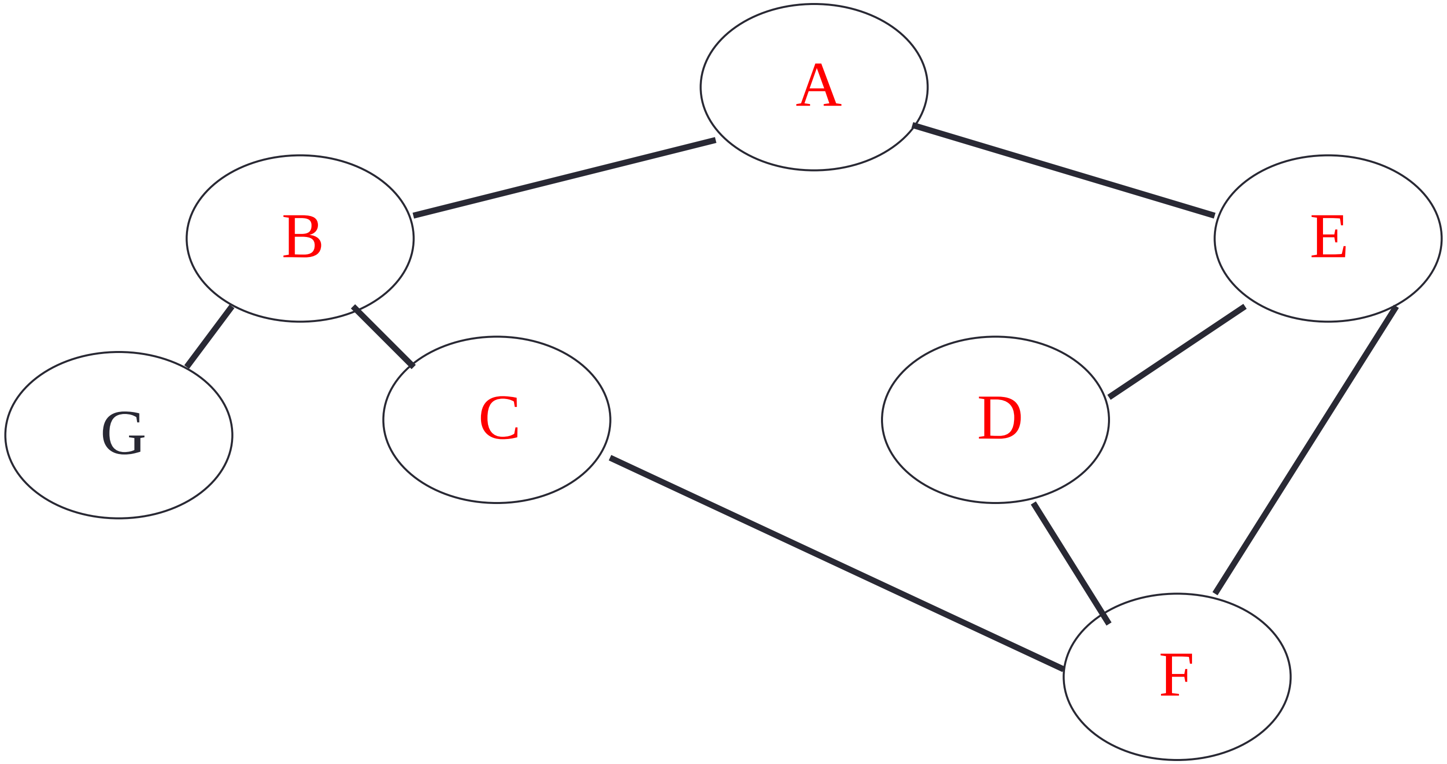
Current: E




E’s adjacent vertices are A, D and F.

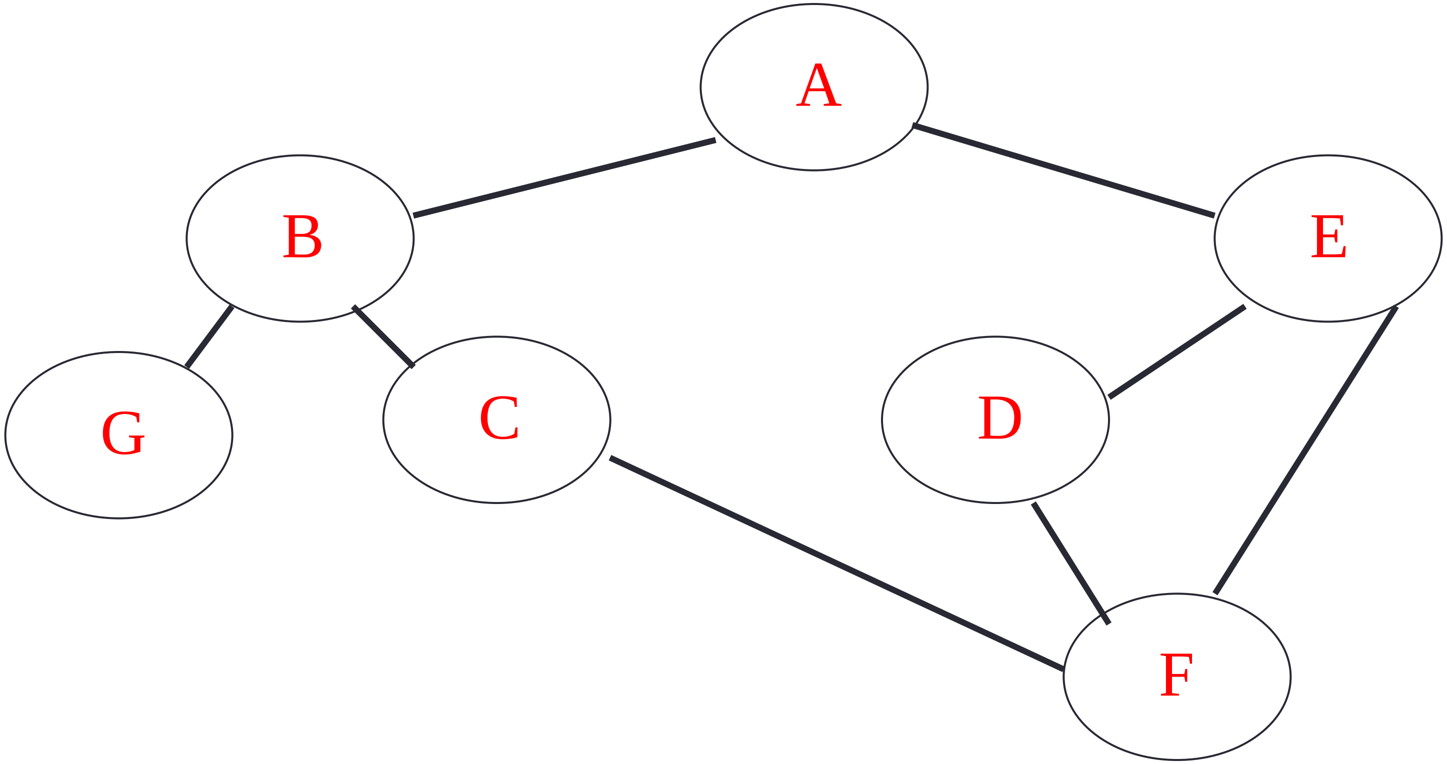
A and F are marked, so pick D.

Current: D



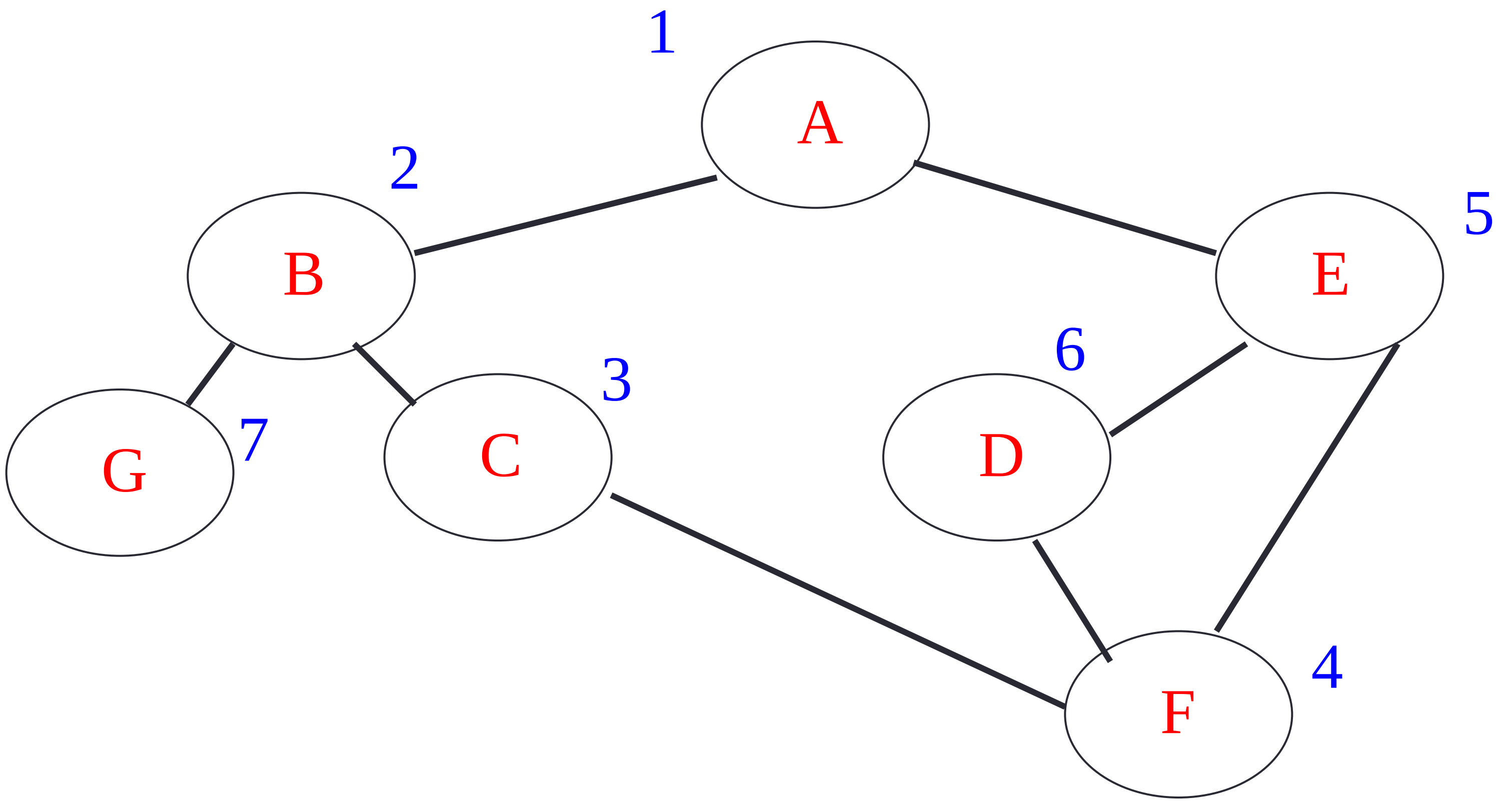

Visit D. No new vertices available. Backtrack to E. Backtrack to F. Backtrack to C. Backtrack to B

Current: G




Visit G. No new vertices from here. Backtrack to B. Backtrack to A. E already marked so no new.

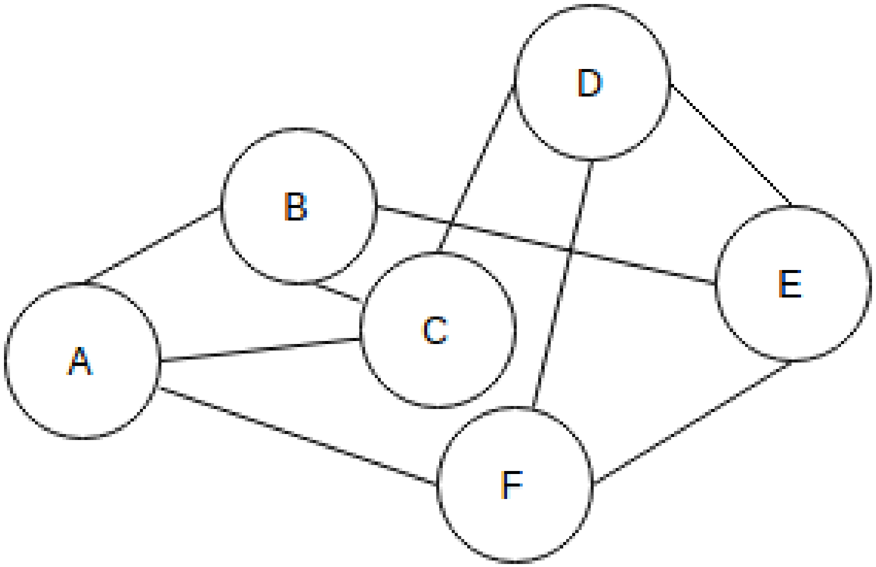
Current:




Done. We have explored the graph in order:

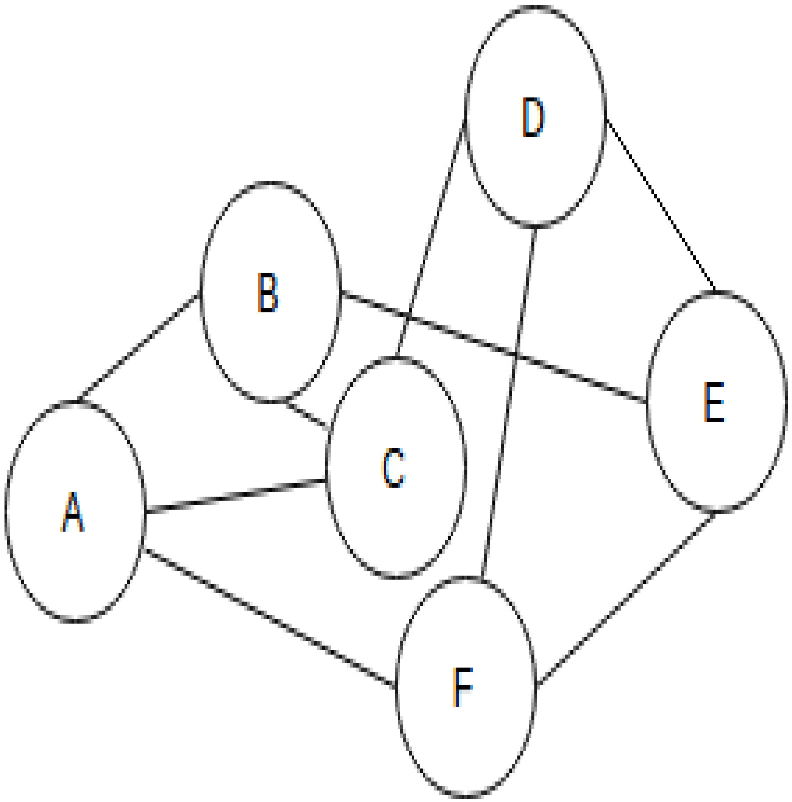
A B C F E D G

# Method 1




| Stack | Marked | Curr | DFS |
|---|---|---|---|
| ~~A~~ | A | A | A |
| ~~B~~,C,F | A,B,C,F | B | A,B |
| ~~E~~,C,F | A,B,C,F,E | E | A,B,E |
| ~~D~~,C,F | A,B,C,F,E | D | A,B,E,D |
| ~~C~~,F | A,B,C,F,E | C | A,B,E,D,C |
| ~~F~~ | A,B,C,F,E | F | A,B,E,D,C,F<==DFS SEQUENCE |

# Method 2




| stack | curr | DFS |
|---|---|---|
| ~~A~~ | A | A |
| ~~B~~ | B | A,B |
| ~~E~~ | E | A,B,E |
| ~~D~~ | D | A,B,E,D |
| ~~C~~ | C | A,B,E,D,C |
| ~~F~~ | F | A,B,E,D,C,F<==DFS SEQUEN |

Marked= {A, B,E,D,C, F }

# Interesting features of DFS

- Complexity: O(|V| + |E|)
  - All vertices visited once, then marked
  - For each vertex on stack, we examine all edges
  - In other words, we traverse all edges once
- DFS does not necessarily find shortest path
  - Why?
- Not a good choice when the goal node is at shallow level on right side of the graph

# QUERIES?

# THANK YOU