Data Structures and Algorithms

Lecture 9

Graphs

Introduction

- We have studied many different data structures.
- We started by looking at “linear list” data structures – each node had a single predecessor and single successor (e.g., linked lists, stacks, queues).
- Then we looked at tree structures in which each node could have multiple successors but only one predecessor (e.g., BSTs, heaps).

Introduction

- Now we will begin studying graphs.
- Graphs differ from the other data structures in that each node in the graph may have multiple predecessors as well as multiple successors.

Applications

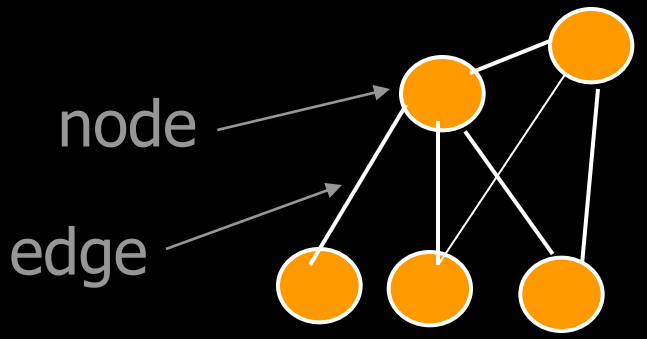
- Routing problems
  - Internet packets
  - Airlines among various airports
- Mapping
  - Mapquest and GPS navigation systems
  - Calculate shortest or fastest route
- Minimize VLSI layouts
  - Transistors/gates = vertices
  - Traces = arcs
- Land development
  - Determine most profitable areas to build up
  - Roads = arcs
- Gene-expression data
  - Determine which genes to cluster to minimize computational cost

Terminology

- A ***graph*** is a collection of nodes, called ***vertices***, and a collection of line segments, called ***lines, edges,*** or ***arcs***, connecting pairs of vertices.
- In other words, a graph consists of 2 sets:
  - A set of lines
  - A set of vertices

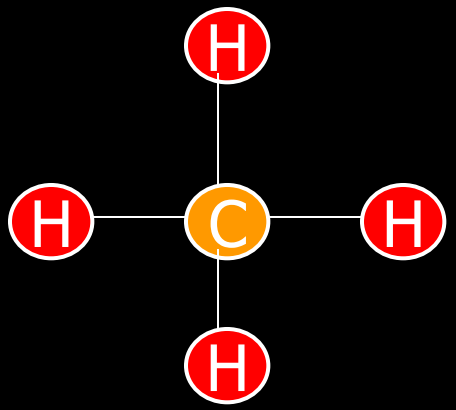
What is a graph?

- Graphs represent the relationships among data items
- A graph G consists of
  - a set V of nodes (vertices)
  - a set E of edges: each edge connects two nodes
- Each node represents an item
- Each edge represents the relationship between two items

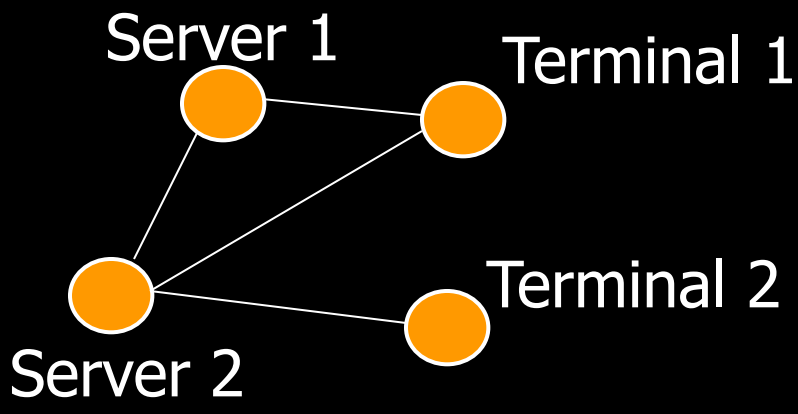


Examples of graphs

Molecular Structure



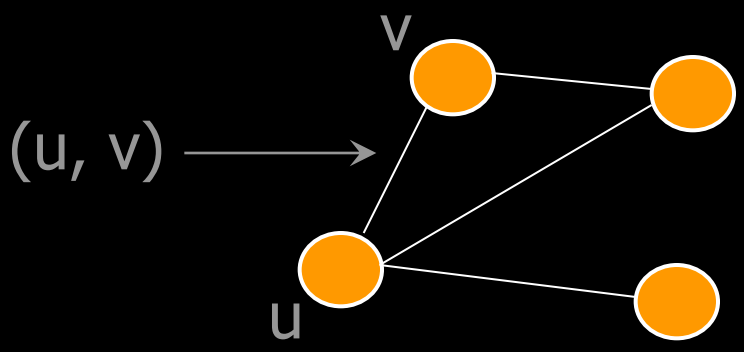
Computer Network



Other examples: electrical and communication networks, airline routes, flow chart, graphs for planning projects

Formal Definition of graph

- The set of nodes is denoted as V
- For any nodes u and v, if u and v are connected by an edge, such edge is denoted as (u, v)



- The set of edges is denoted as E
- A graph G is defined as a pair (V, E)

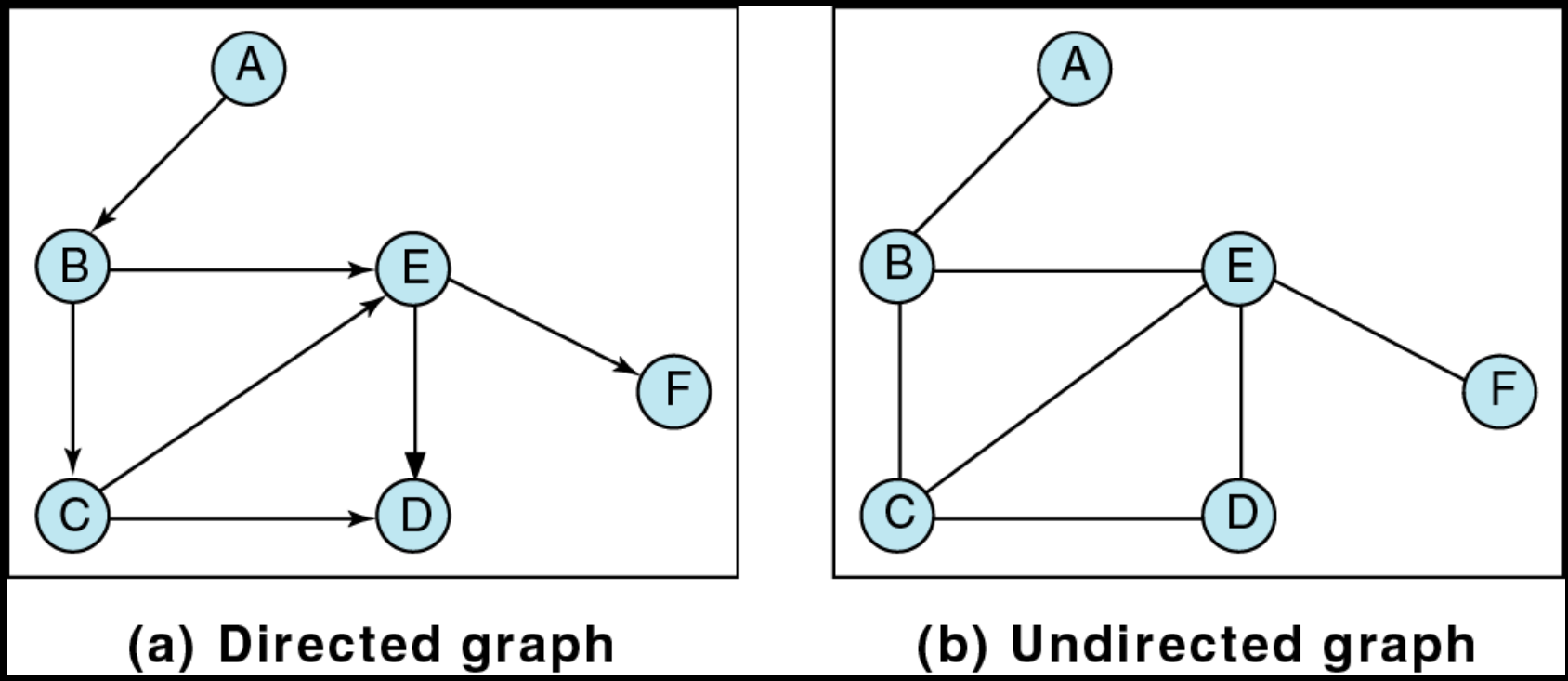
Terminology

- Graphs may be either directed or undirected.
- An ***undirected graph*** is a graph in which there is no direction associated with the lines.
- In an undirected graph, the flow between two vertices can go in either direction.

Terminology

- A ***directed graph*** or ***digraph*** is a graph in which each line has a direction associated with it.
- In a directed graph, the flow along an edge between 2 vertices can only follow the indicated direction.

Terminology



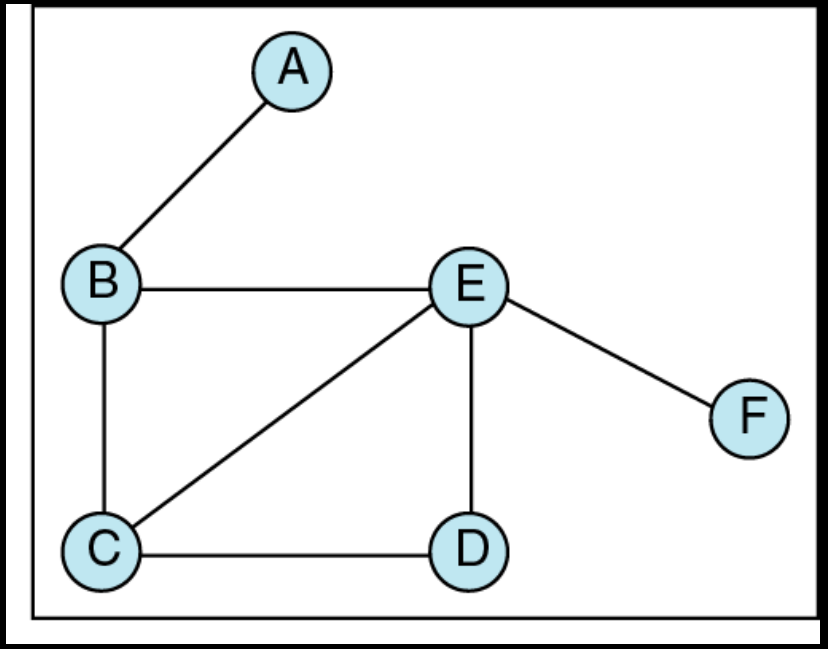
(a) Directed graph

(b) Undirected graph

Terminology

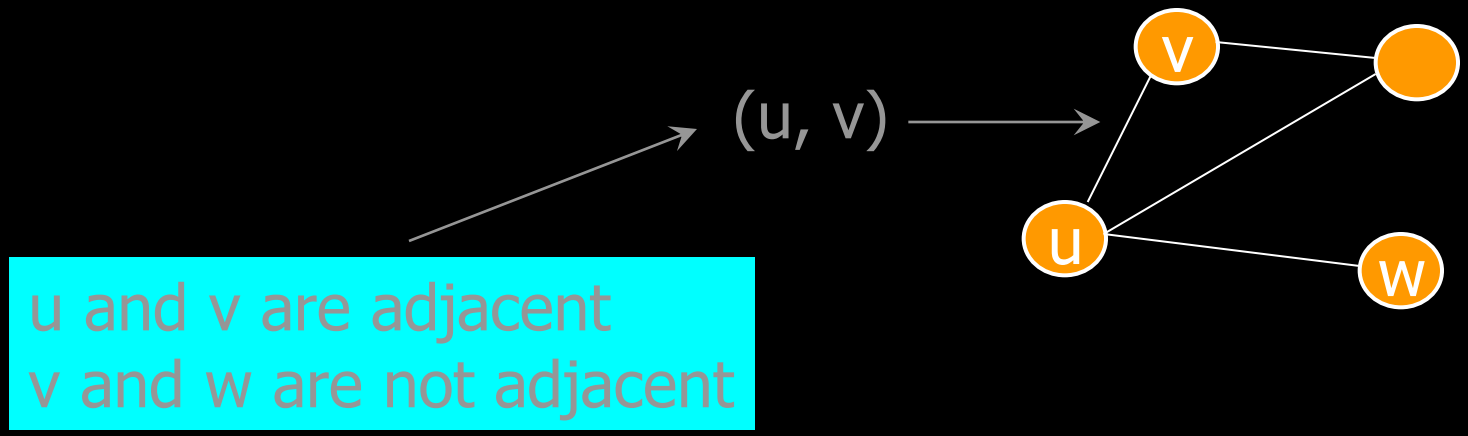
- 2 vertices in a graph are said to be ***adjacent*** (or neighbors) if an edge directly connects them.

A and B are adjacent
D and F are not adjacent



Adjacent

- Two nodes u and v are said to be adjacent if (u, v) $\in$ E

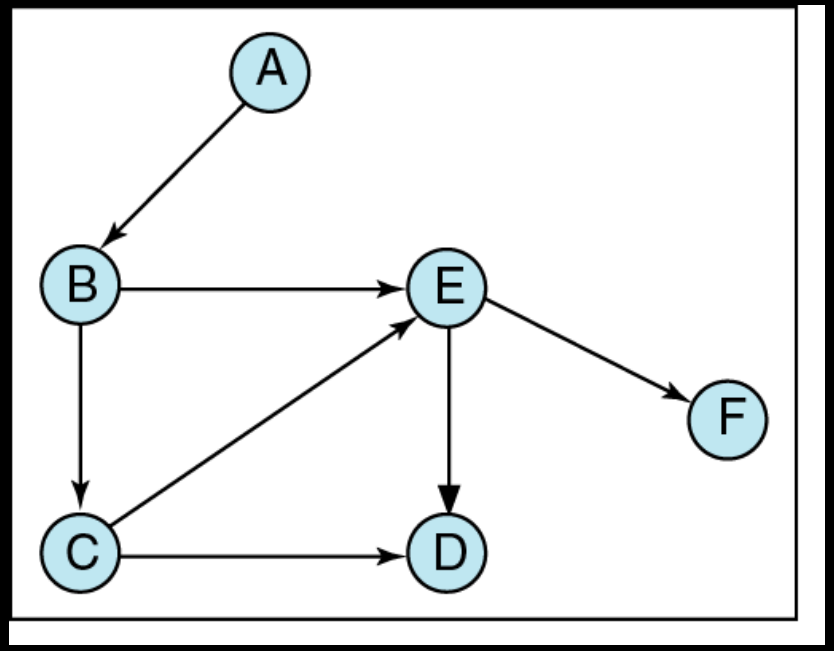


Terminology

- A ***path*** is a sequence of vertices in which each vertex is adjacent to the next one.

A,B,C,E is a path
A,B,E,F is a path

A,B,E,C is *NOT* a path.
Although E is adjacent to C, C is not adjacent to E (note direction).

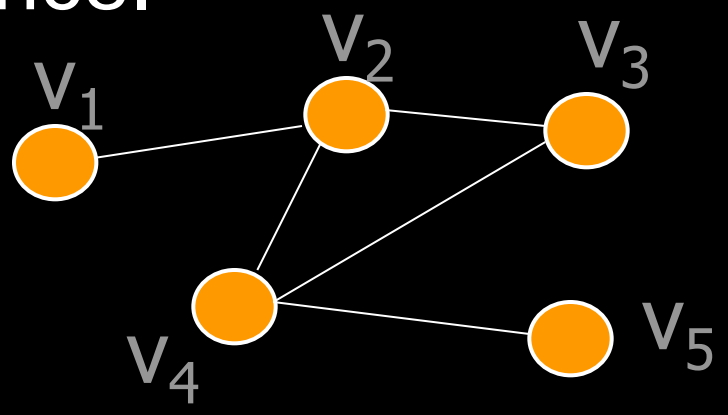


Terminology

- Note that both directed and undirected graphs have paths.
- In an undirected graph, you may travel in either direction.
- In a directed graph, you may only travel along the given direction of the arcs.

Path and simple path

- A path from v_1 to v_k is a sequence of nodes $v_1, v_2, \ldots, v_k$ that are connected by edges $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$
- A path is called a simple path if every node appears at most once.

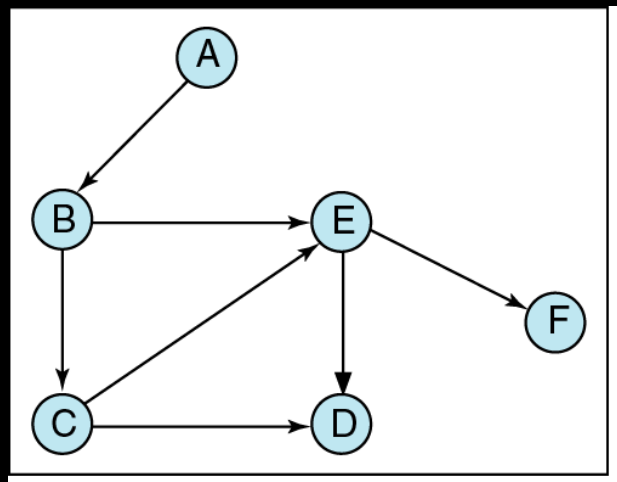


- v_2, v_3, v_4, v_2, v_1 is a path
- v_2, v_3, v_4, v_5 is a path, also it is a simple path

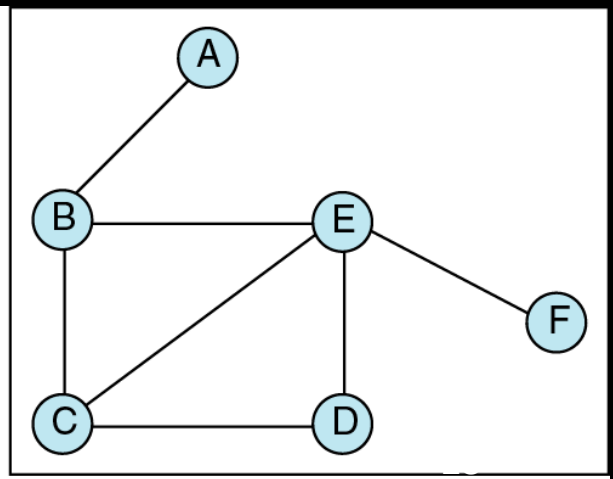
Terminology

- A ***noncyclic*** or ***simple path*** is a path that does not touch the same node twice.
- A ***cycle*** is a path that starts and ends at the same vertex.
- Ignoring the start/end vertex, a cycle is a simple path.
- Note: a single vertex is not a cycle.

B, C, D, E, B is a cycle in the the undirected graph but not in the directed graph.



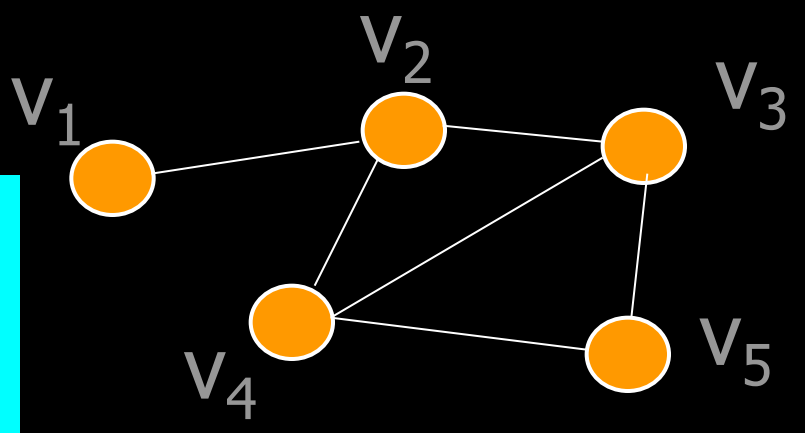
(a) Directed graph



(b) Undirected graph

Cycle and simple cycle

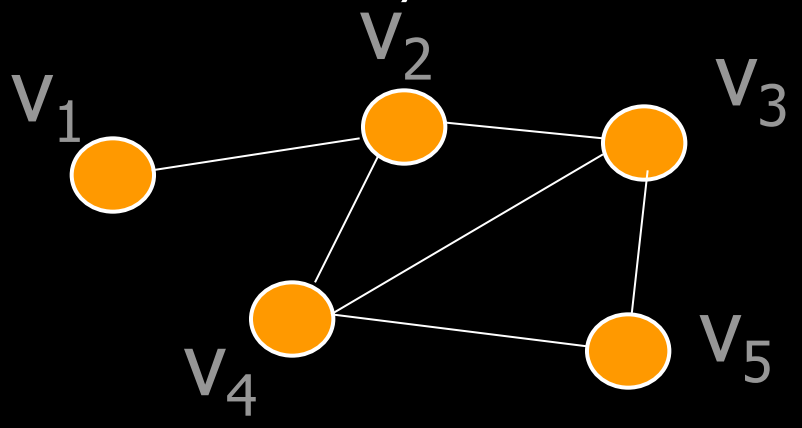
- A cycle is a path that begins and ends at the same node
- A simple cycle is a cycle if every node appears at most once, except for the first and the last nodes



- $v_2, v_3, v_4, v_5, v_3, v_2$ is a cycle
- v_2, v_3, v_4, v_2 is a cycle, it is also a simple cycle

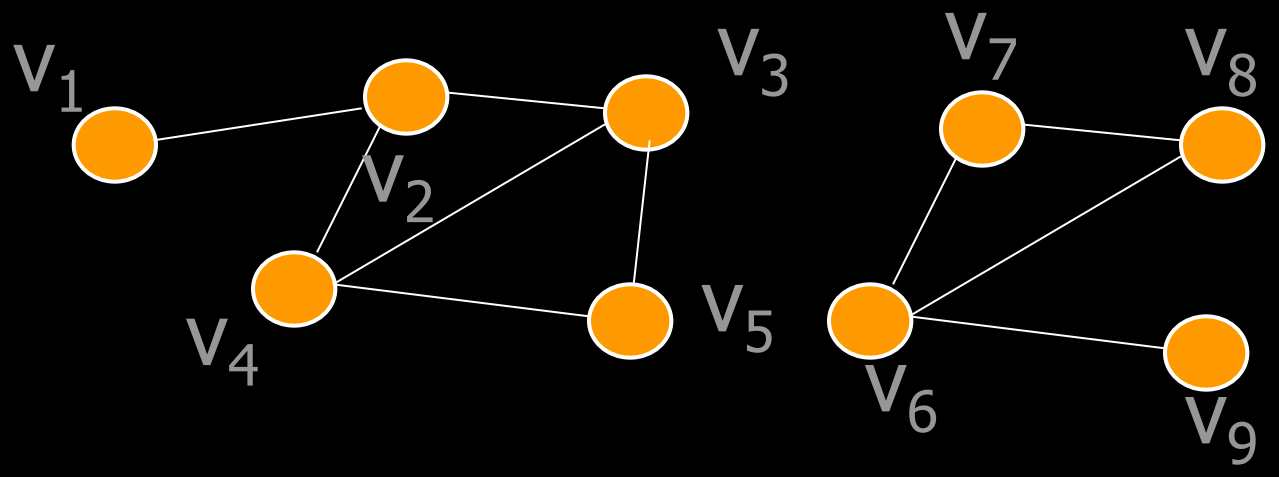
Connected graph

- A graph G is connected if there exists path between every pair of distinct nodes; otherwise, it is disconnected



This is a connected graph because there exists path between every pair of nodes

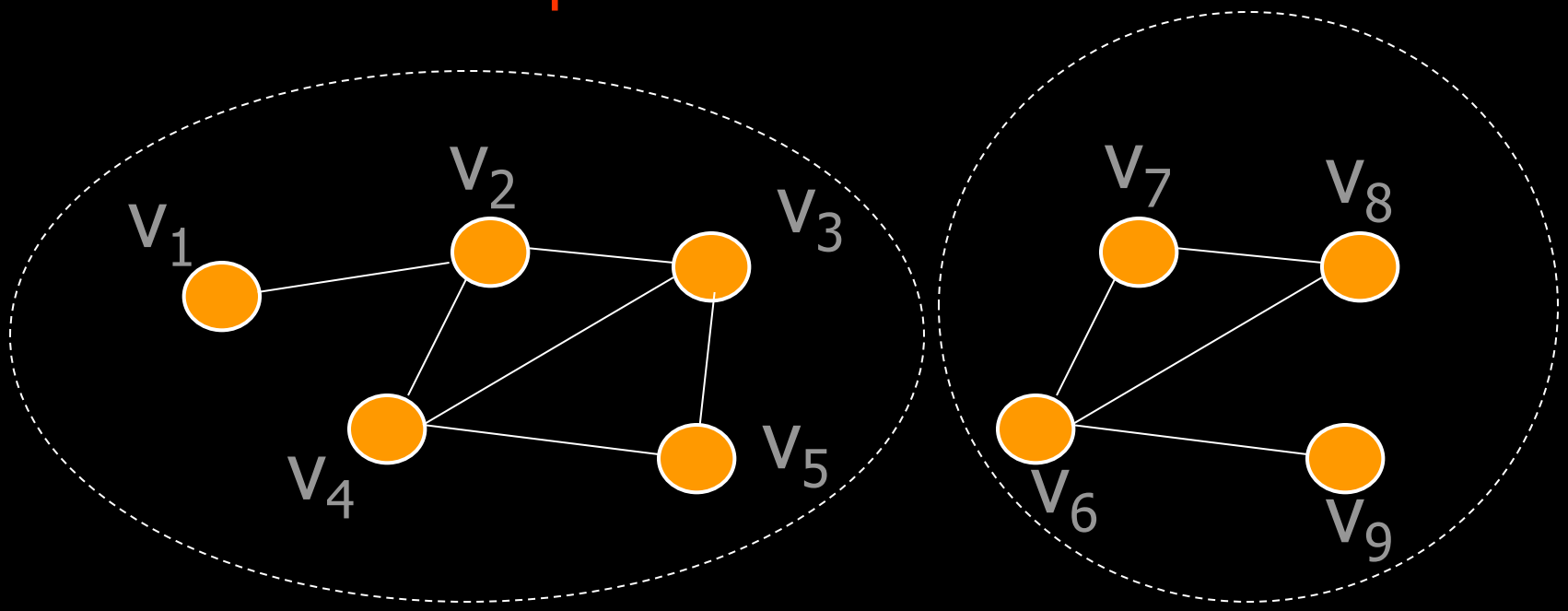
Example of disconnected graph



This is a disconnected graph because there does not exist path between some pair of nodes, says, v_1 and v_7

Connected component

- If a graph is disconnect, it can be partitioned into a number of graphs such that each of them is connected. Each such graph is called a connected component.



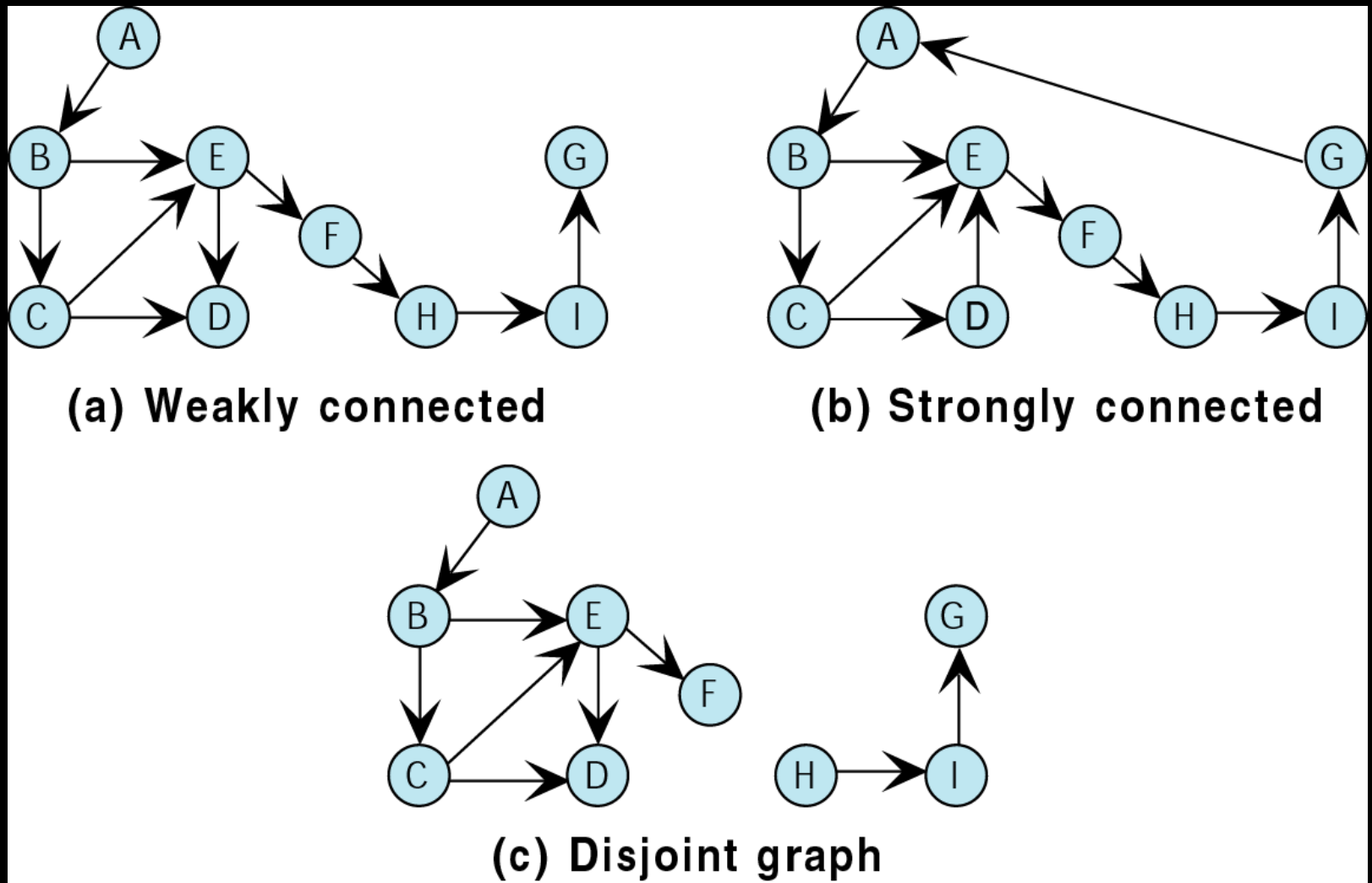
Terminology

- Two vertices are said to be ***connected*** if there is a path between them.
- A graph is said to be connected if, suppressing direction, there is a path from each vertex to every other vertex.
- Furthermore, a directed graph is ***strongly connected*** if there is a path from each vertex to every other vertex.

Terminology

- A directed graph is ***weakly connected*** if at least 2 vertices are not connected.
- A graph is ***disconnected*** or ***disjoint*** if it is not connected.

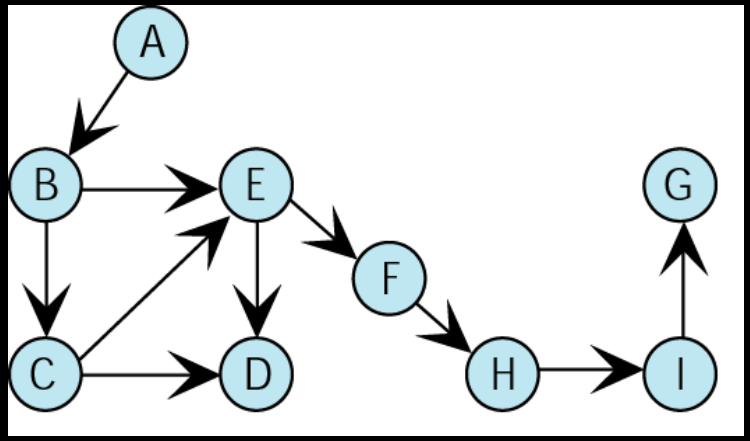
Terminology



Terminology

- The ***degree*** of a vertex is the number of edges incident to it.
- The ***outdegree*** of a vertex in a digraph is the number of arcs leaving the vertex.
- The ***indegree*** of a vertex in a digraph is the number of arcs entering the vertex.

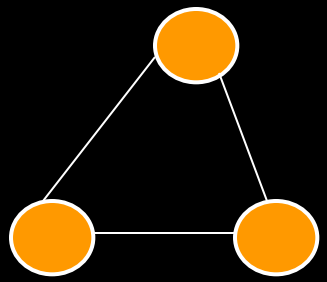
Terminology



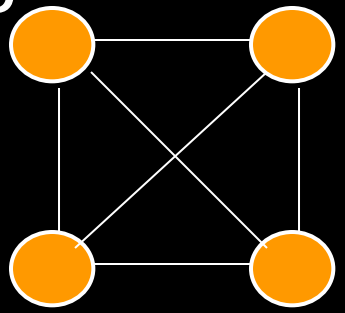
- The degree of vertex B is … 3
- The outdegree of vertex B is … 2
- The indegree of vertex B is … 1

Complete graph

- A graph is complete if each pair of distinct nodes has an edge



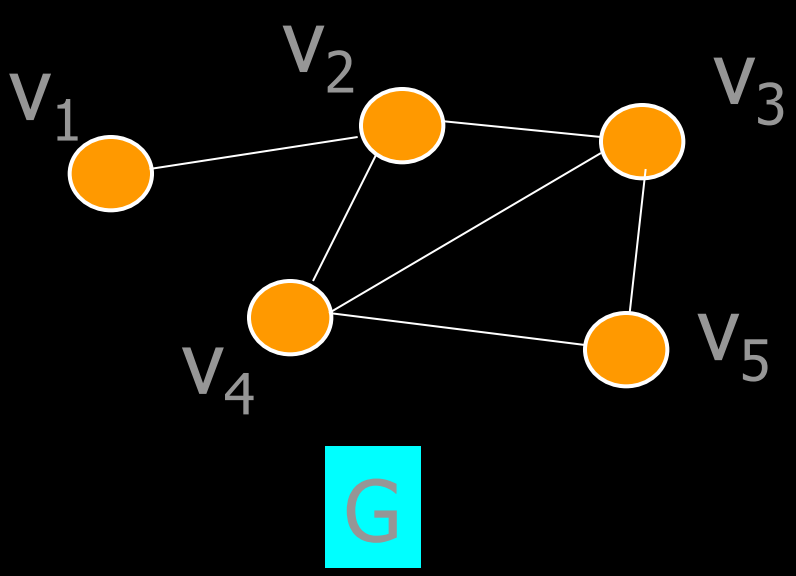
Complete graph with 3 nodes

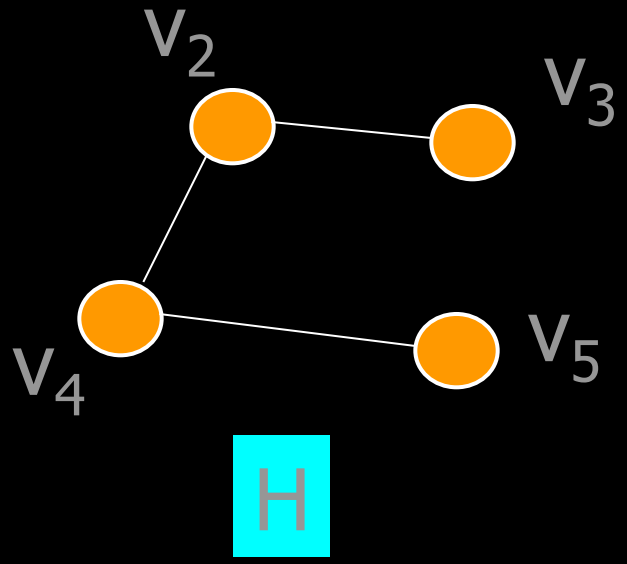


Complete graph with 4 nodes

Subgraph

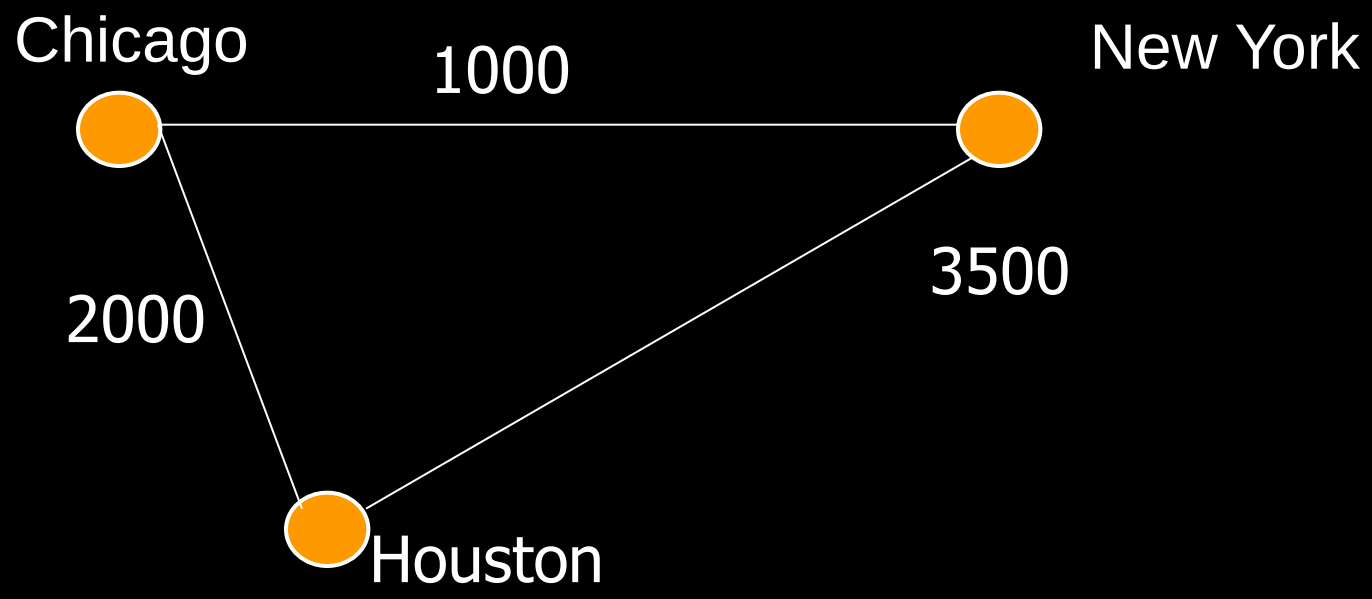
- A subgraph of a graph G =(V, E) is a graph H = (U, F) such that $U \subseteq V$ and $F \subseteq E$.





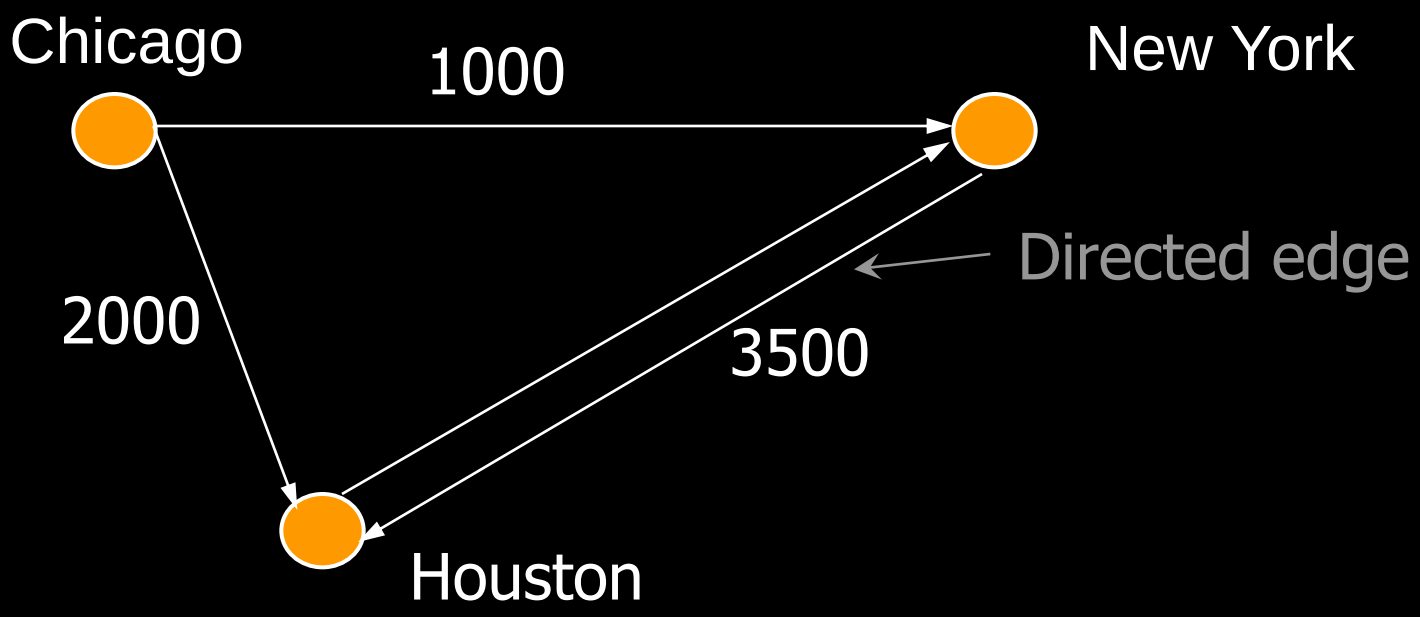
Weighted graph

- If each edge in G is assigned a weight, it is called a weighted graph



Directed graph (digraph)

- All previous graphs are undirected graph
- If each edge in E has a direction, it is called a directed edge
- A directed graph is a graph where every edges is a directed edge

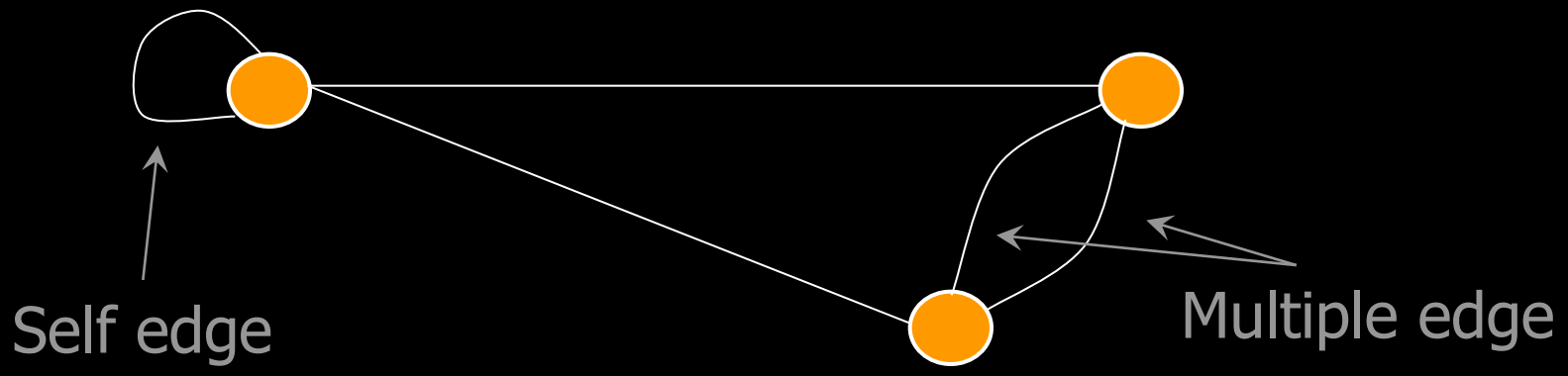


More on directed graph

- If (x, y) is a directed edge, we say
  - y is adjacent to x
  - y is successor of x
  - x is predecessor of y
- In a directed graph, directed path, directed cycle can be defined similarly

Multigraph

- A graph cannot have duplicate edges.
- Multigraph allows multiple edges and self edge (or loop).



Property of graph

- A undirected graph that is connected and has no cycle is a tree.
- A tree with n nodes have exactly n-1 edges.
- A connected undirected graph with n nodes must have at least n-1 edges.

Operations

- Now we will discuss 6 graph operations that provide the basic functionality needed to maintain a graph:
  - Add vertex
  - Delete vertex
  - Add edge
  - Delete edge
  - Find vertex
  - Traverse graph

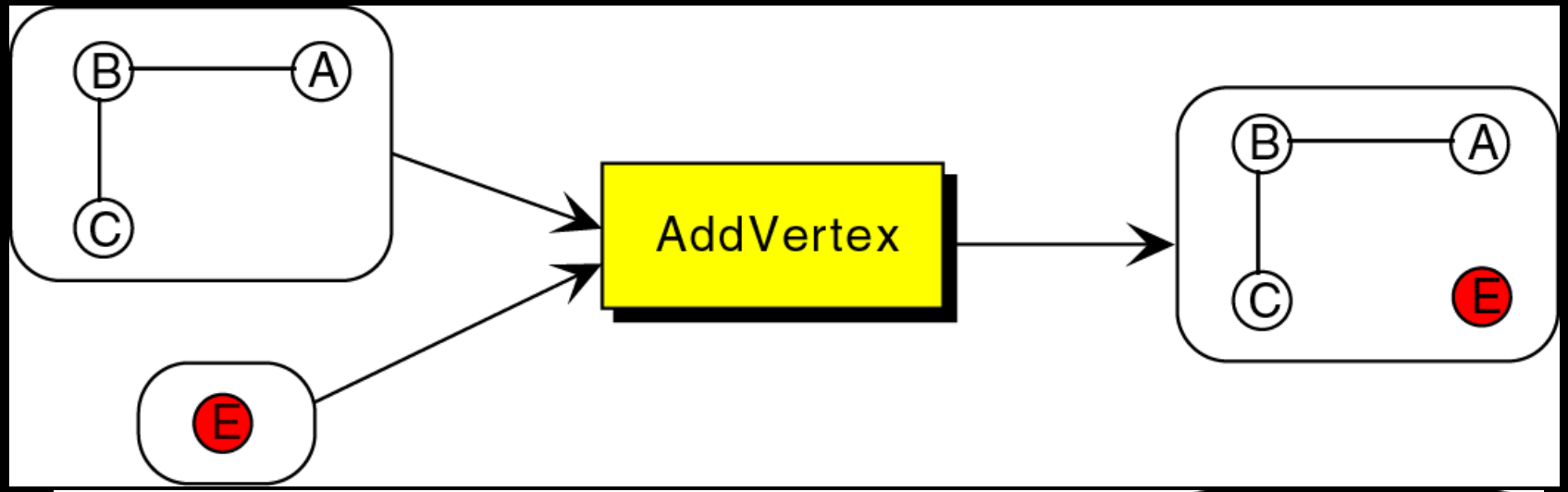
Add Vertex

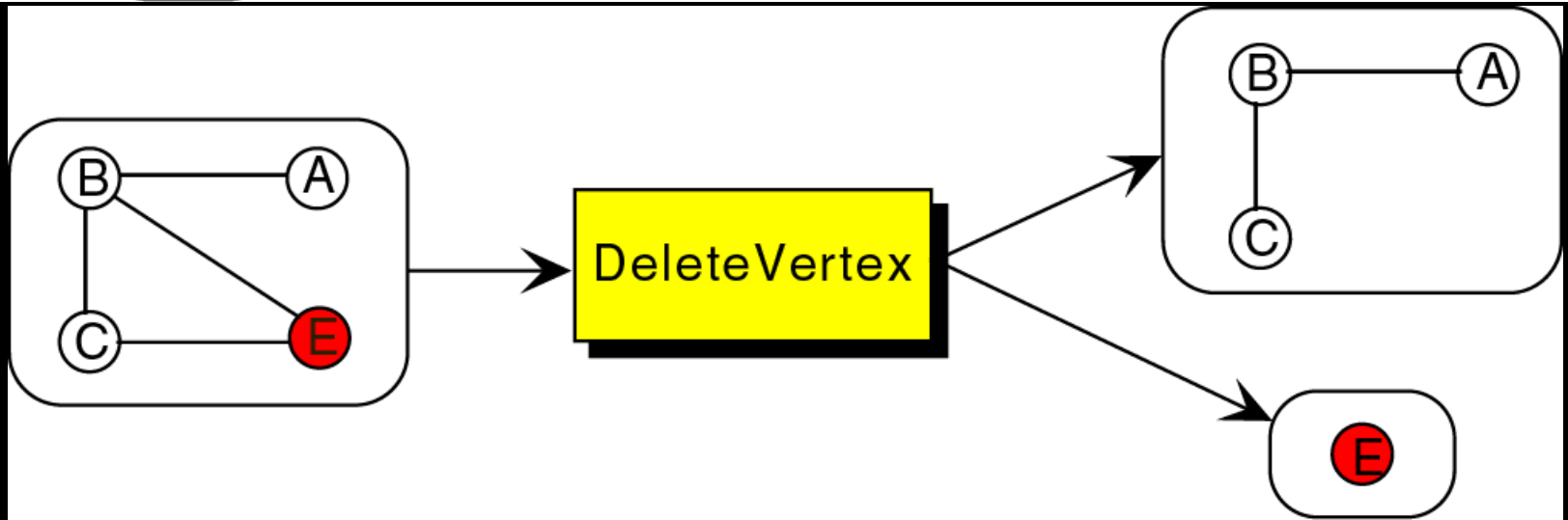
- Add vertex inserts a new vertex into a graph.
- When a vertex is added, it is disjoint; that is, it is not connected to any other vertices in the graph.
- Obviously, adding a vertex is just the first step in the insertion process.
- After a vertex is added, it must be connected.

Delete Vertex

- Delete vertex removes a vertex from the graph.
- When a vertex is deleted, all connecting edges also need to be removed.

Add and Delete Vertex





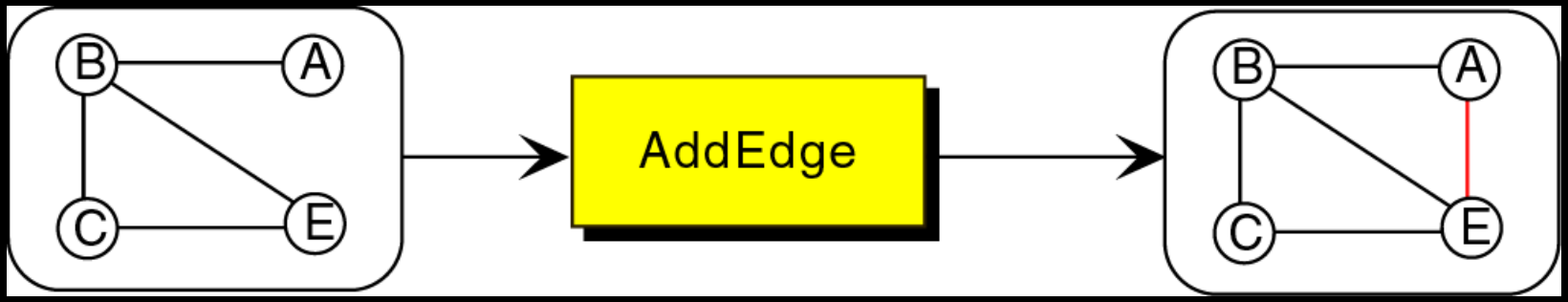
Add Edge

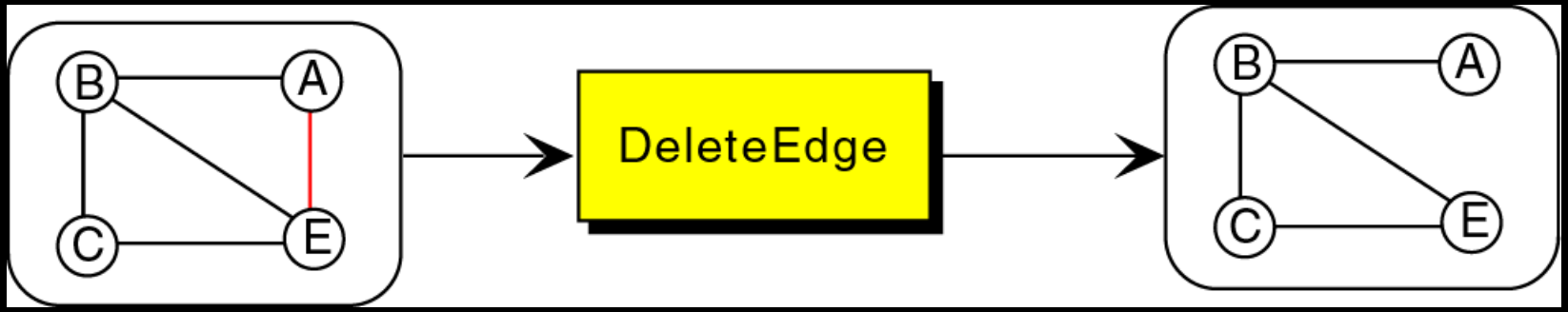
- Add edge connects 2 vertices.
- If a vertex requires multiple edges, add edge must be called once for each adjacent vertex.
- To add an edge, two vertices must be specified.
- If the graph is a digraph, one of the vertices must be specified as the source and one as the destination.

Delete Edge

- Delete edge removes one edge from a graph.

Add and Delete Edge





Find Vertex

- Find vertex traverses a graph looking for a specific vertex.
- If the vertex is found, its data is returned.
- If it is not found, an error is indicated.

Traverse Graph

- Because a vertex in a graph can have multiple parents, the traversal of a graph presents some problems not found in the traversal of linear lists and trees.
- We must ensure that we process the data in each vertex once and only once.

Traverse Graph

- Because there are multiple paths to a vertex, we may arrive at it from more than one direction as we traverse the graph.
- The traditional solution to this problem is to include a ‘visited’ flag at each vertex.
- Before the traversal, we set the visited flag in each vertex to ‘off.’
- Then, as we traverse the graph, we set the visited flag to ‘on’ to indicate that the data in a given vertex has been processed.

Traverse Graph

- Note that a graph traversal algorithm is slightly different from a “print all vertices” algorithm.
- A graph-traversal algorithm will only visit all of the nodes that it can reach.
- In other words, a graph traversal that starts at vertex V will visit all vertices W for which there exists a path between V and W.

Traverse Graph

- Hence, unlike a tree traversal which always visits all of the nodes in a tree, a graph traversal does not necessarily visit all of the vertices in the graph (unless the graph is connected).
- If the graph is not connected, a graph traversal that begins at vertex V will visit only a subset of the graph's vertices.
- This subset is called the ***connected component*** containing V.

Implementing Graph

- Adjacency matrix
  - Represent a graph using a two-dimensional array
- Adjacency list
  - Represent a graph using n linked lists where n is the number of vertices

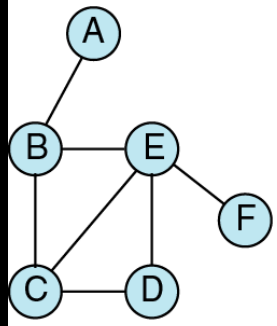
Adjacency Matrix

- The ***adjacency matrix*** uses a vector (1-dimensional array) to store the vertices and a matrix (2-dimensional array) to store the edges.
- If 2 vertices are adjacent – that is, if there is an edge between them, the matrix intersect has a value of 1.
- If there is no edge between them, the intersect is set to 0.

Adjacency Matrix

- If the graph is directed, then the intersection chosen in the adjacency matrix indicates the direction.

Adjacency Matrix

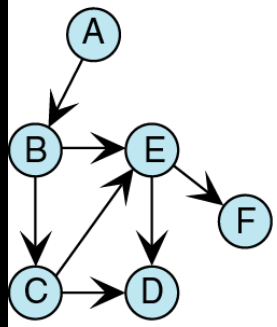


	A	B	C	D	E	F
A	0	1	0	0	0	0
B	1	0	1	0	1	0
C	0	1	0	1	1	0
D	0	0	1	0	1	0
E	0	1	1	1	0	1
F	0	0	0	0	1	0

Vertex vector

Adjacency matrix

(a) Adjacency matrix for non-directed graph



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	1	0	1	0
C	0	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	1	0	1
F	0	0	0	0	0	0

Vertex vector

Adjacency matrix

(a) Adjacency matrix for directed graph

Here we see the vertex vectors and the adjacency matrices for the graphs shown.

The vertex vector contains an entry for each node in the graph.

The adjacency matrix contains a '1' at the appropriate intersection to represent an edge.

matrix[i][j] = 1 if there is an

The intersection at which the '1' appears is determined by the direction of the arc between adjacent nodes of the digraph.

It doesn't matter which intersection gets the '1' – just remain consistent.

I like the notation:

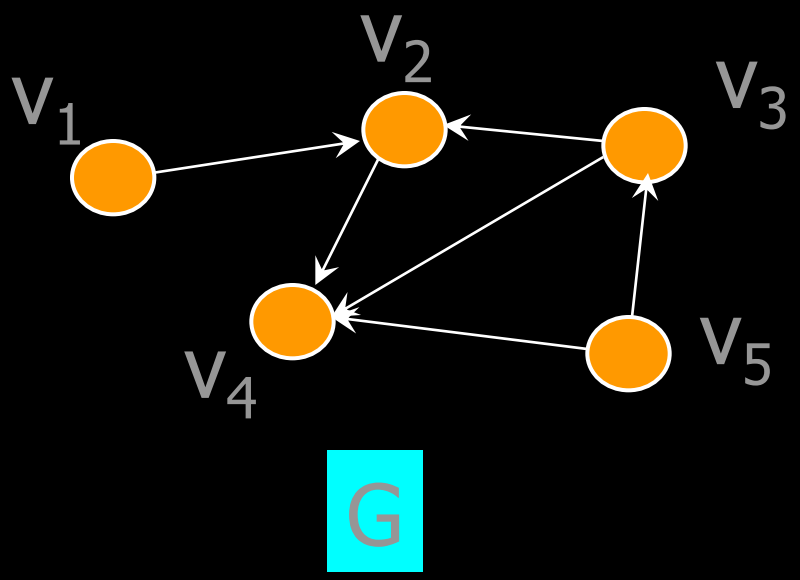
Adjacency Matrix

- Another limitation of adjacency matrices: only 1 edge can be stored between any 2 vertices.
- This can be an issue in modeling some structures (e.g., 2 different flights between the same cities).

Adjacency matrix for directed graph

Matrix[i][j] = 1 if $(v_i, v_j) \in E$

0 if $(v_i, v_j) \notin E$



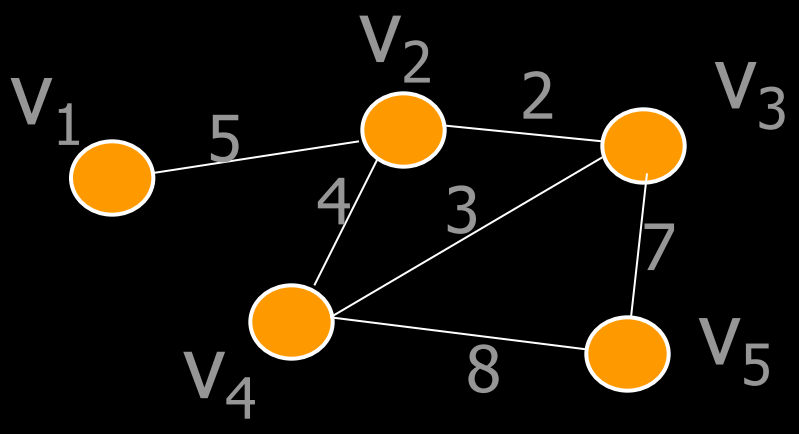
G

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	0	1	0	0	0
2	v_2	0	0	0	1	0
3	v_3	0	1	0	1	0
4	v_4	0	0	0	0	0
5	v_5	0	0	1	1	0

Adjacency matrix for weighted undirected graph

Matrix[i][j] = $w(v_i, v_j)$ if $(v_i, v_j) \in E$ or $(v_j, v_i) \in E$

∞ otherwise



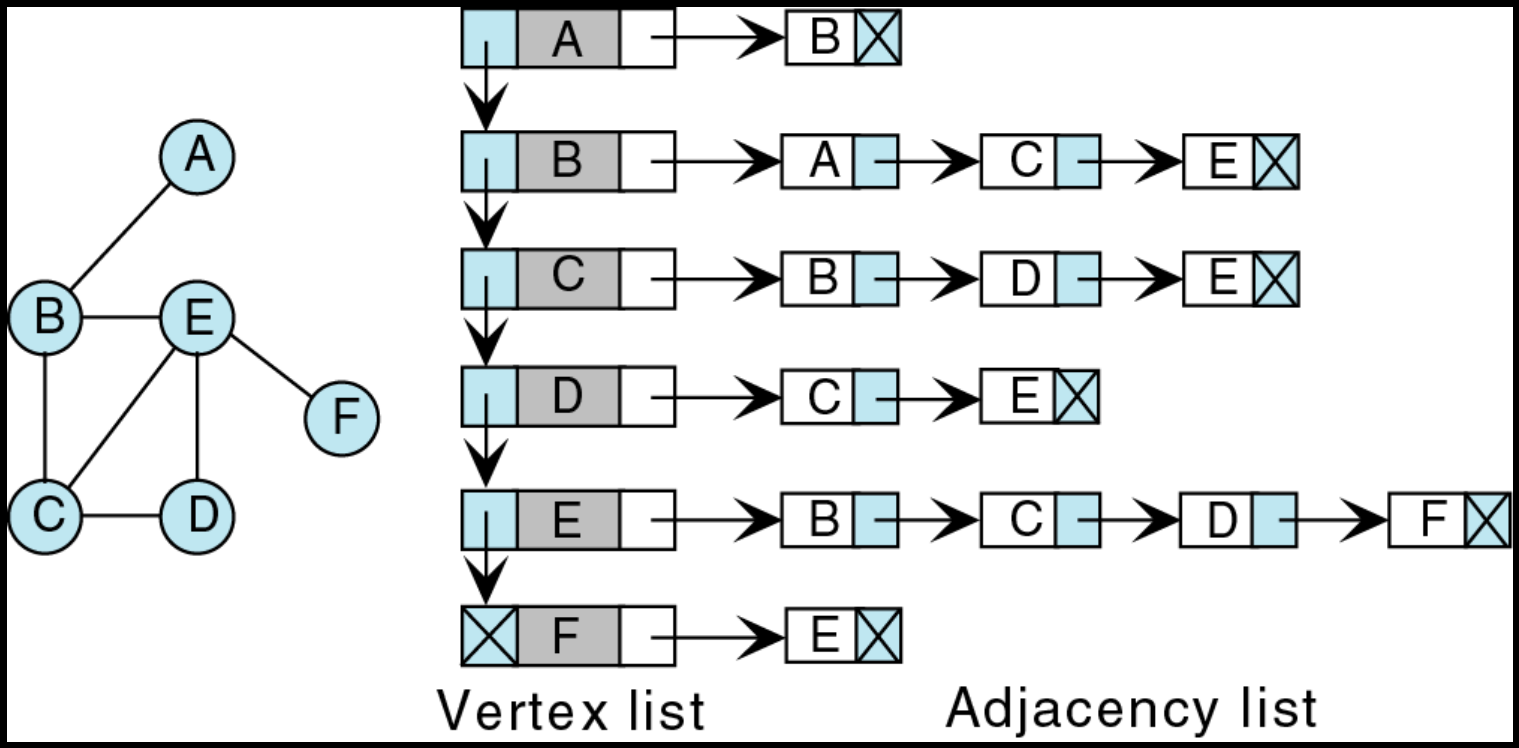
G

		1	2	3	4	5
		v_1	v_2	v_3	v_4	v_5
1	v_1	∞	5	∞	∞	∞
2	v_2	5	∞	2	4	∞
3	v_3	0	2	∞	3	7
4	v_4	∞	4	3	∞	8
5	v_5	∞	∞	7	8	∞

Adjacency List

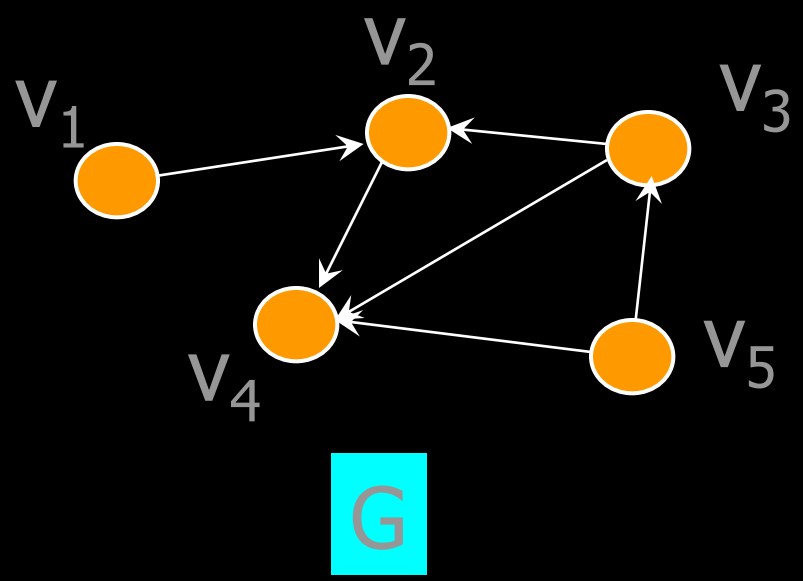
- If the graph structure is unknown ahead of time, we can use linked lists to implement the vertex vector and adjacency matrix.
- In this case, they are referred to as a vertex list and an adjacency list.

Adjacency List



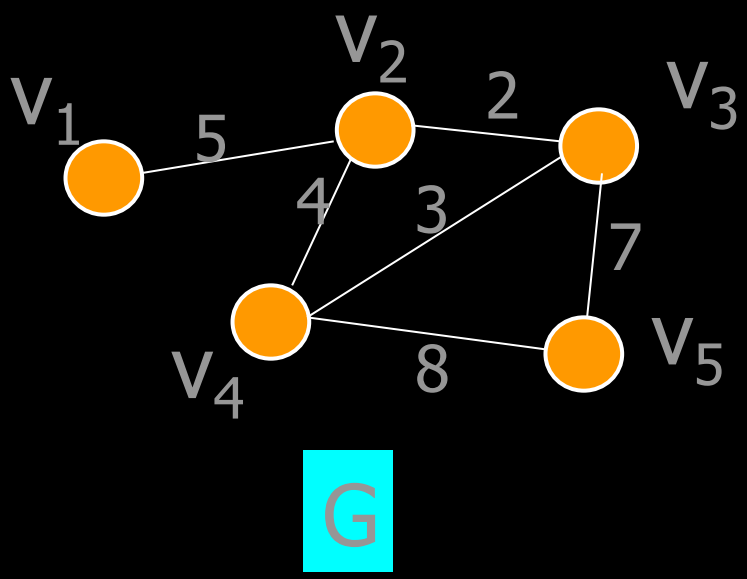
Here, we have a linked list of the nodes in the graph. Each node in the graph contains a pointer to the head of an adjacency list. The adjacency list is a linked list containing adjacent nodes.

Adjacency list for directed graph



1	v_1	$\rightarrow v_2$
2	v_2	$\rightarrow v_4$
3	v_3	$\rightarrow v_2 \rightarrow v_4$
4	v_4	
5	v_5	$\rightarrow v_3 \rightarrow v_4$

Adjacency list for weighted undirected graph



1	v_1	→	$v_2(5)$					
2	v_2	→	$v_1(5)$	→	$v_3(2)$	→	$v_4(4)$	
3	v_3	→	$v_2(2)$	→	$v_4(3)$	→	$v_5(7)$	
4	v_4	→	$v_2(4)$	→	$v_3(3)$	→	$v_5(8)$	
5	v_5	→	$v_3(7)$	→	$v_4(8)$			

Pros and Cons

- Adjacency matrix
  - Allows us to determine whether there is an edge from node i to node j in O(1) time
- Adjacency list
  - Allows us to find all nodes adjacent to a given node j efficiently
  - If the graph is scarce, adjacency list requires less space

Problems related to Graph

- Graph Traversal
- Topological Sort
- Spanning Tree
- Minimum Spanning Tree
- Shortest Path

Graph Traversal Algorithm

- To traverse a tree, we use tree traversal algorithms like pre-order, in-order, and post-order to visit all the nodes in a tree
- Similarly, graph traversal algorithm tries to visit all the nodes it can reach.
- If a graph is disconnected, a graph traversal that begins at a node v will visit only a subset of nodes, that is, the connected component containing v.

Two basic traversal algorithms

- Two basic graph traversal algorithms:
  - Depth-first-search (DFS)
    - After visit node v, DFS strategy proceeds along a path from v as deeply into the graph as possible before backing up
  - Breadth-first-search (BFS)
    - After visit node v, BFS strategy visits every node adjacent to v before visiting any other nodes

Depth-first search (DFS)

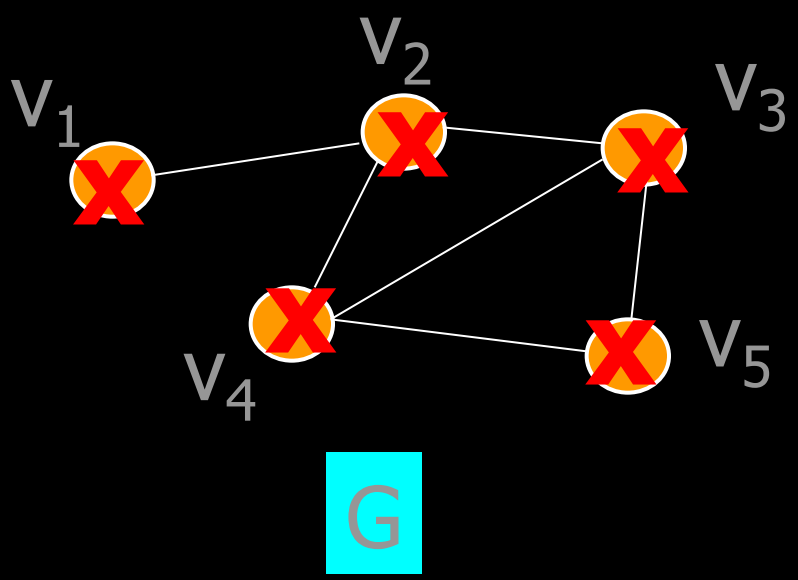
- DFS strategy looks similar to pre-order. From a given node v, it first visits itself. Then, recursively visit its unvisited neighbours one by one.
- DFS can be defined recursively as follows.

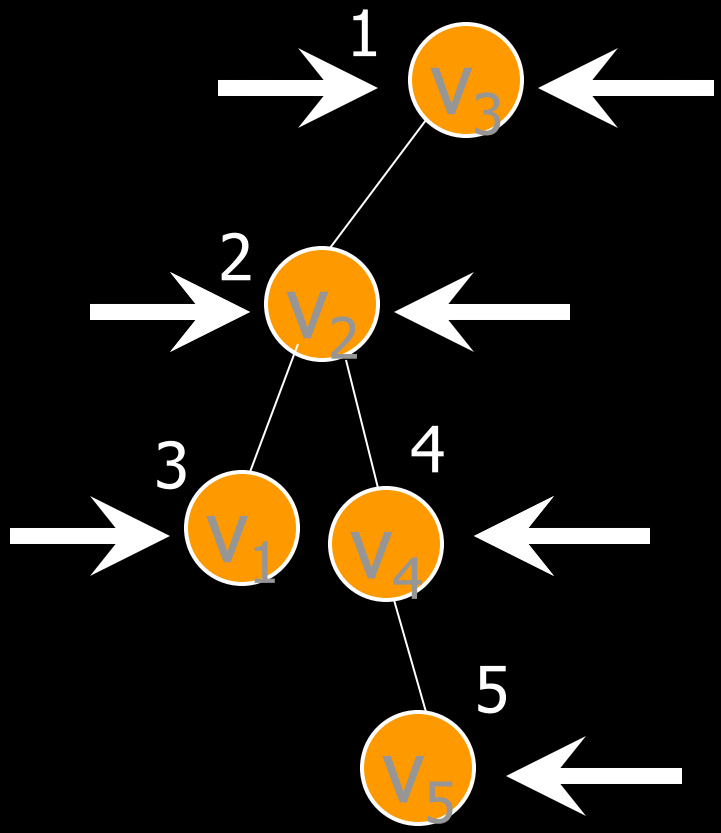
Algorithm dfs(v)

```
print v ;                    // you can do other things!
mark v as visited;
for (each unvisited node u adjacent to v)
    dfs(u);
```

DFS example

- Start from v_3



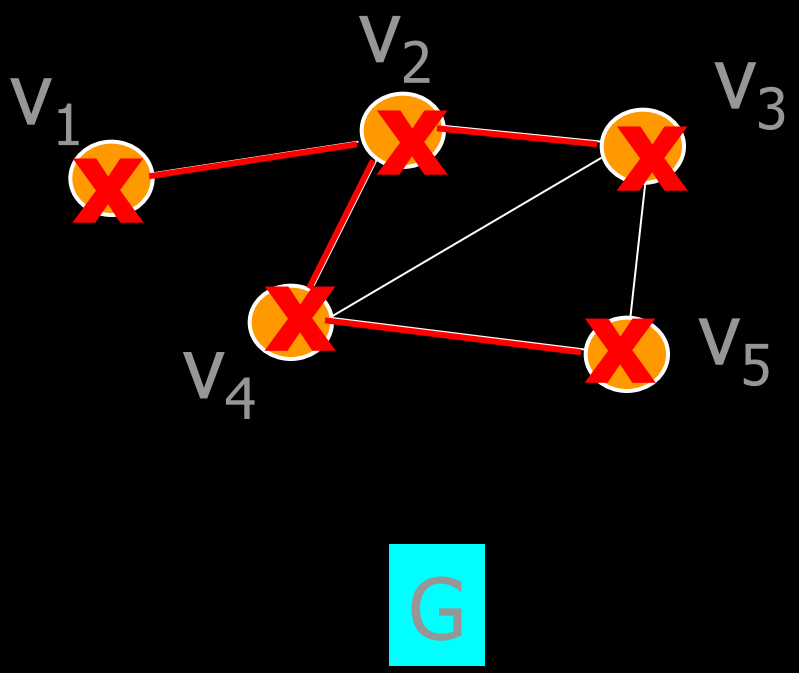


Non-recursive version of DFS algorithm

```
Algorithm dfs(v)
s.createStack();
s.push(v);
mark v as visited;
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x)
            s.pop();            // backtrack
    else {
            select an unvisited node u adjacent to x;
            s.push(u);
            mark u as visited;
    }
}
```

Non-recursive DFS example

visit	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4 , v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty

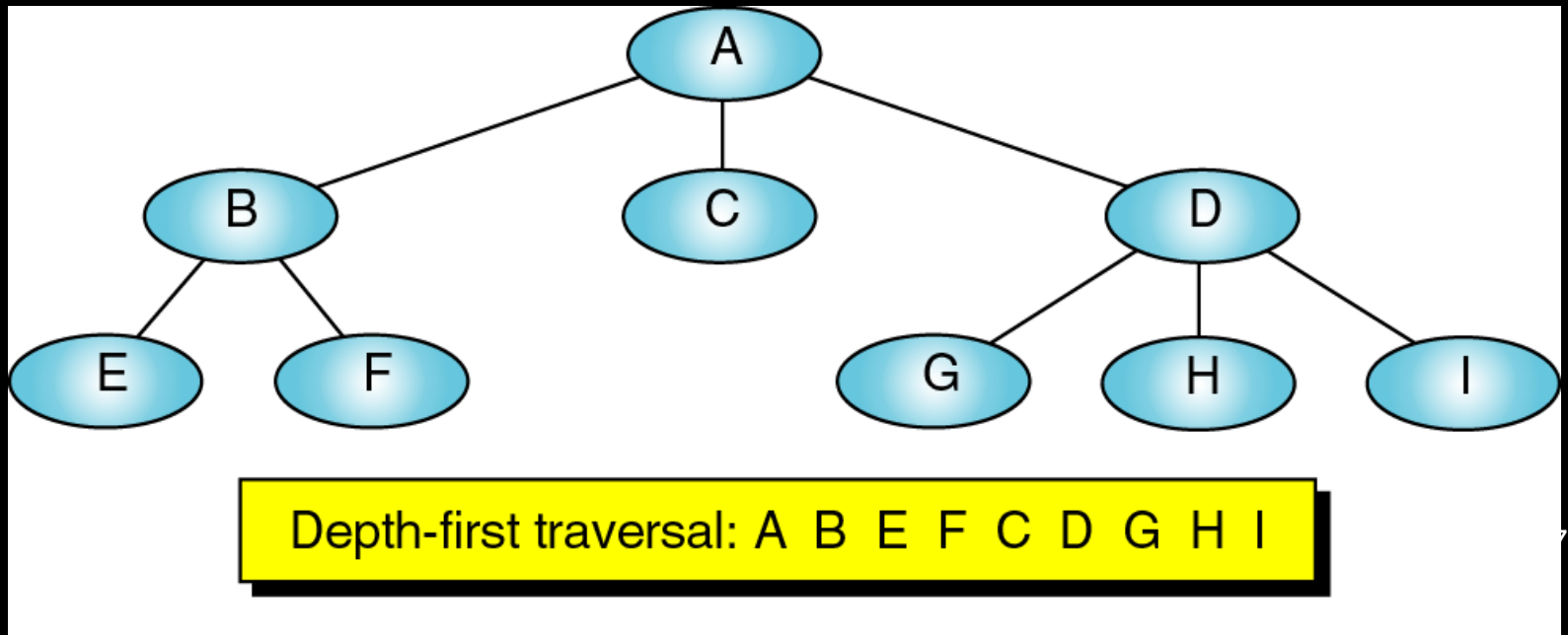


Depth-First Traversal

- In the depth-first traversal, we process all of a vertex's descendants before we move to an adjacent vertex.
- This concept is most easily seen when the graph is a tree.

Depth-First Traversal

- Here we show the preorder traversal, one of the standard depth-first traversals.



Depth-First Traversal

- In a similar manner, the depth-first traversal of a graph starts by processing the first vertex of the graph.
- After processing the first vertex, we select any vertex adjacent to the first vertex and process it.
- As we process each vertex, we select an adjacent vertex until we reach a vertex with no adjacent entries.
- This is similar to reaching a leaf in a tree.

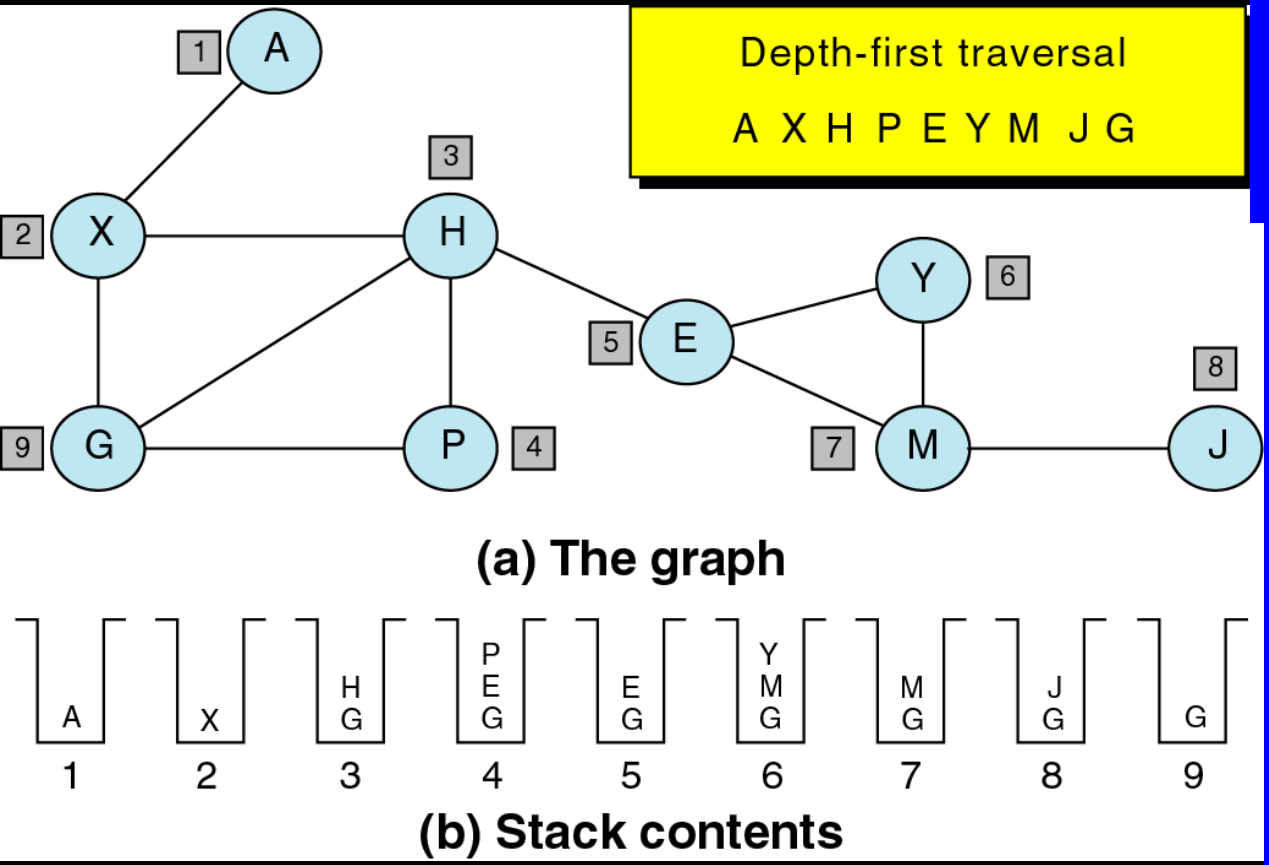
Depth-First Traversal

- We then back out of the structure, processing adjacent vertices as we go.
- It should be obvious that this logic requires a stack (or recursion) to complete the traversal.
- The order in which the adjacent vertices are processed depends on how the graph is physically stored.

Depth-First Traversal

- On the next slide, we'll walk through a depth-first traversal.
- The number in the box next to a vertex indicates the processing order.
- The stacks below the graph show the stack contents as we work our way down the graph and then as we back out.

Depth-First Traversal



3. When the stack is empty, the traversal is complete.
push all of its adjacent vertices onto the stack.
To process X at Step 2, therefore, we pop X from the stack, process it, and then push G and H onto the stack.

Breadth-first search (BFS)

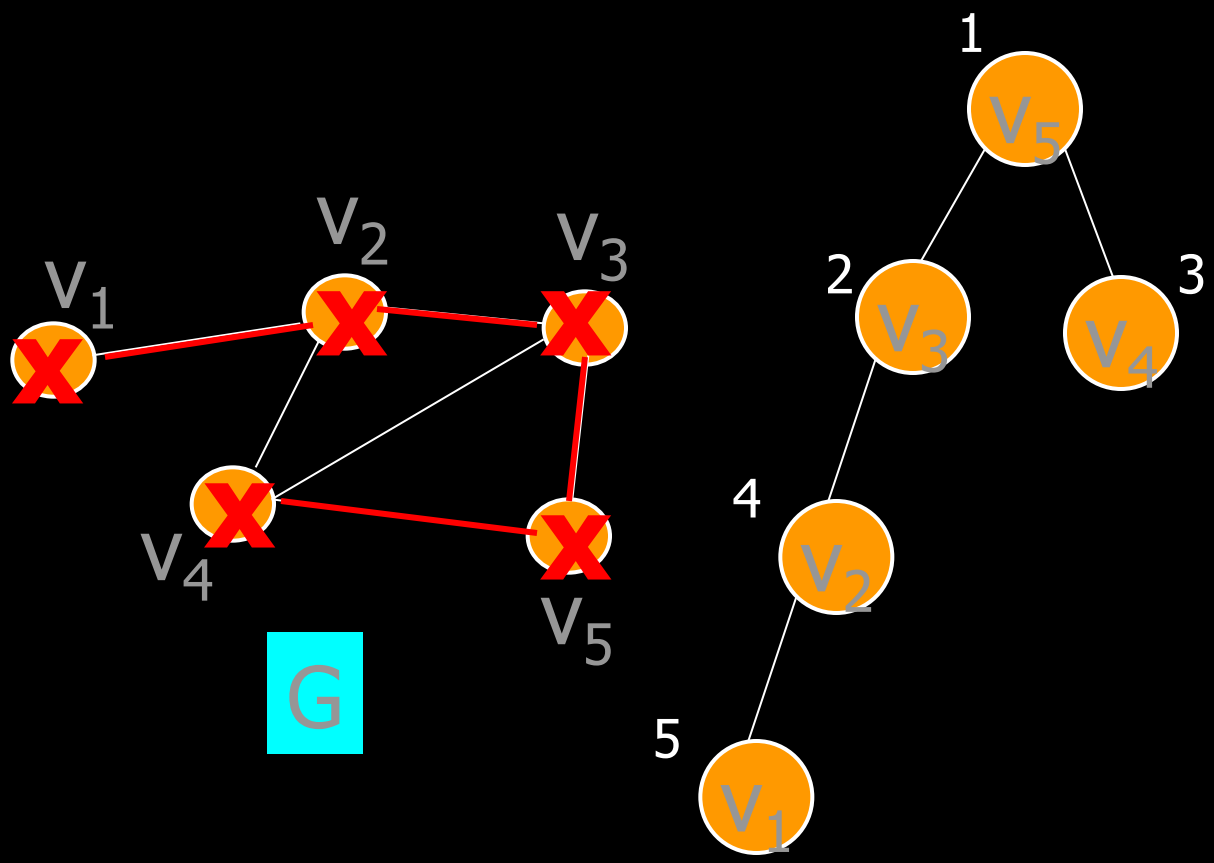
- BFS strategy looks similar to level-order. From a given node v, it first visits itself. Then, it visits every node adjacent to v before visiting any other nodes.
  - 1. Visit v
  - 2. Visit all v's neigbours
  - 3. Visit all v's neighbours' neighbours
  - …
- Similar to level-order, BFS is based on a queue.

Algorithm for BFS

```
Algorithm bfs(v)
q.createQueue();
q.enqueue(v);
mark v as visited;
while(!q.isEmpty()) {
    w = q.dequeue();
    for (each unvisited node u adjacent to w) {
        q.enqueue(u);
        mark u as visited;
    }
}
```

BFS example

- Start from v_5

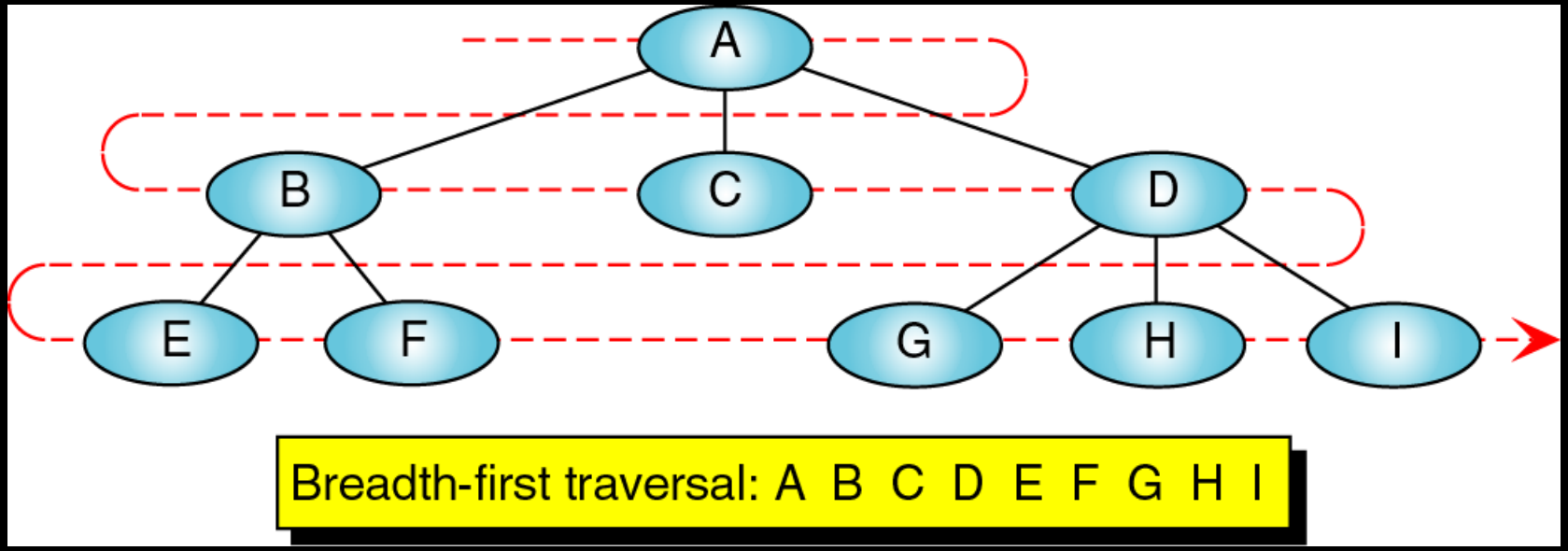


Visit	Queue (front to back)
v_5	v_5
	empty
v_3	v_3
v_4	v_3, v_4
	v_4
v_2	v_4, v_2
	v_2
	empty
v_1	v_1
	empty

Breadth-First Traversal

- In the breadth-first traversal of a graph, we process all adjacent vertices of a vertex before going to the next level.
- We also saw the breadth-first traversal of a tree.

Breadth-First Traversal



- Here we see that the breadth-first traversal starts at level 0 and then processes all the vertices in level 1 before going on to process the vertices in level 2.

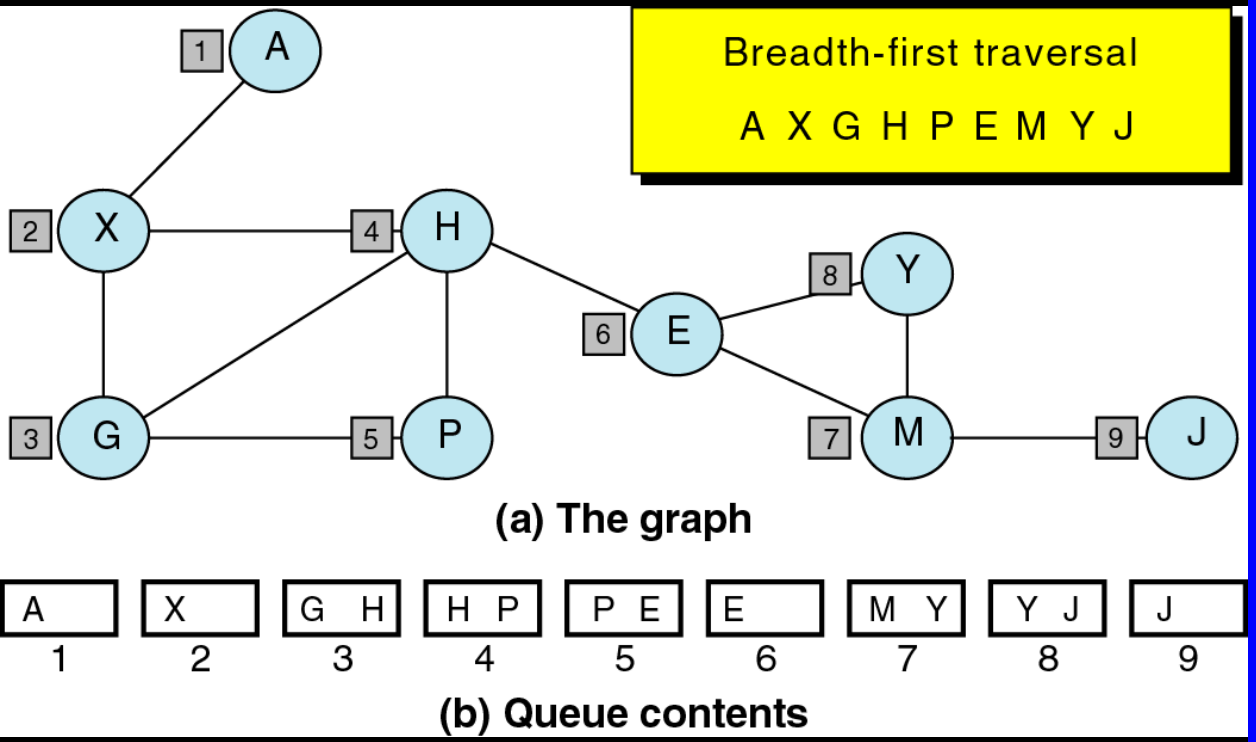
Breadth-First Traversal

- The breadth-first traversal of a graph follows the same approach as the BFT of a tree.
- We begin by picking a starting vertex.
- After processing it, we process all of its adjacent vertices.
- After we process all of the first vertex's adjacent vertices, we pick the first adjacent vertex and process all of its adjacent vertices, then the 2nd adjacent vertex and process all of its adjacent vertices, etc.

Breadth-First Traversal

- The breadth-first traversal uses a queue rather than a stack.
- As we process each vertex, we place all of its adjacent vertices in a queue.
- Then, to select the next vertex to be processed, we remove a vertex from the queue and process it.

Breadth-First Traversal

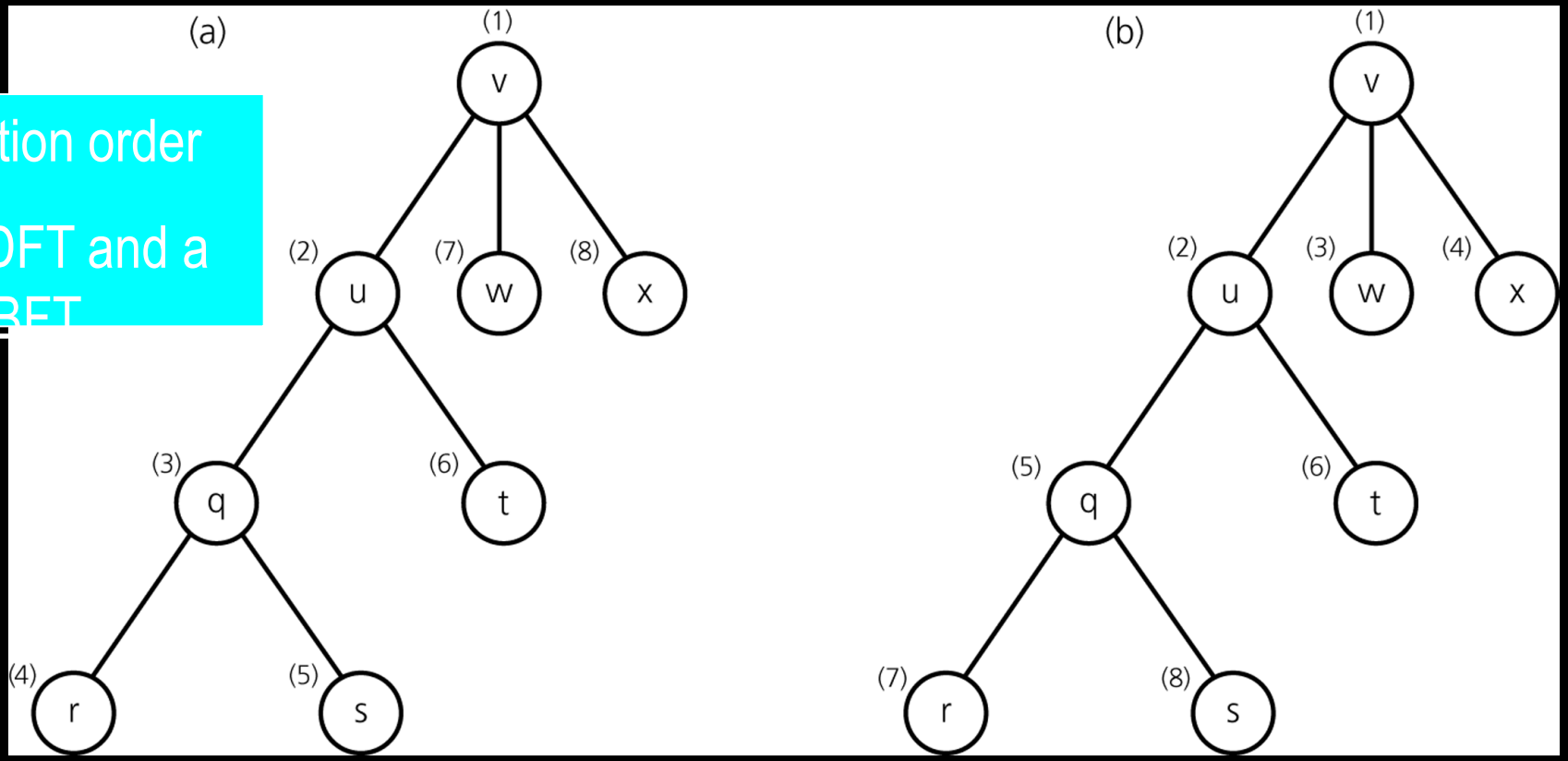


3. When the queue is empty, the traversal is complete.

the vertex at the front of the queue, enqueue all of its adjacent vertices. To process X at Step 2, therefore, we dequeue X, process it, and then enqueue G and H.

Traversals Example

Visitation order for a DFT and a BFT

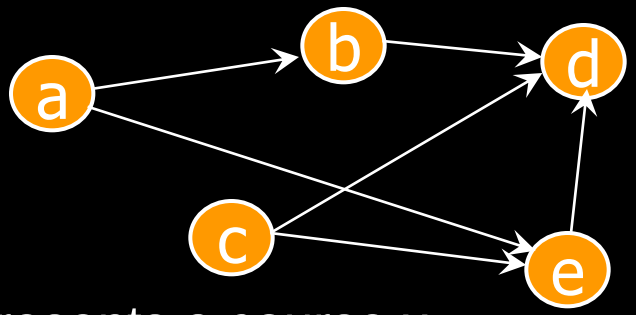


(a) Depth-first traversal

(b) Breadth-first traversal

Topological order

- Consider the prerequisite structure for courses:



- Each node x represents a course x
- (x, y) represents that course x is a prerequisite to course y
- Note that this graph should be a directed graph without cycles (called a directed acyclic graph).
- A linear order to take all 5 courses while satisfying all prerequisites is called a topological order.
- E.g.
  - a, c, b, e, d
  - c, a, b, e, d

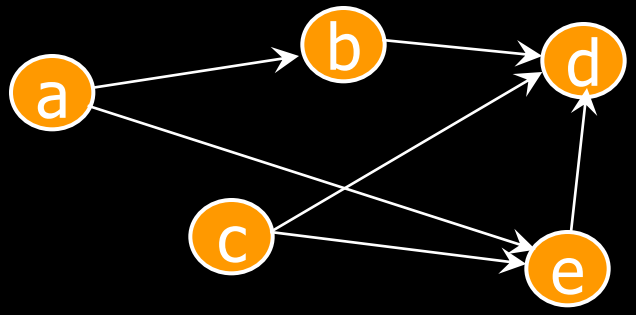
Topological sort

- Arranging all nodes in the graph in a topological order

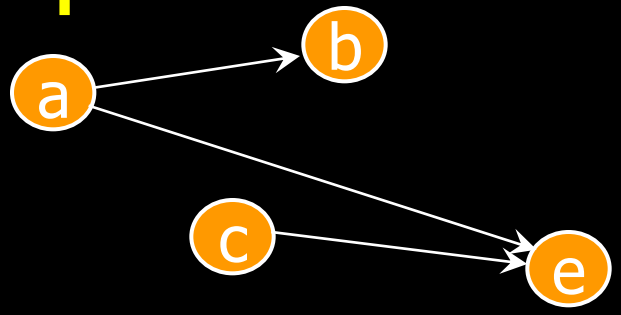
Algorithm topSort

```
n = |V|;
for i = 1 to n {
    select a node v that has no successor;
    aList.add(1, v);
    delete node v and its edges from the graph;
}
return aList;
```

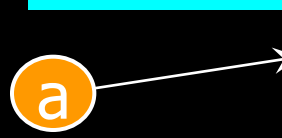
Example



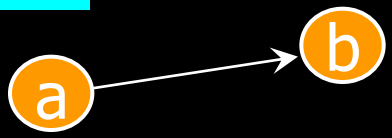
1. d has no successor! Choose d!



2. Both b and e have no successor! Choose e!



3. Both b and c have no successor! Choose c!



4. Only b has no successor! Choose b!

5. Choose a!
The topological order is **a,b,c,e,d**

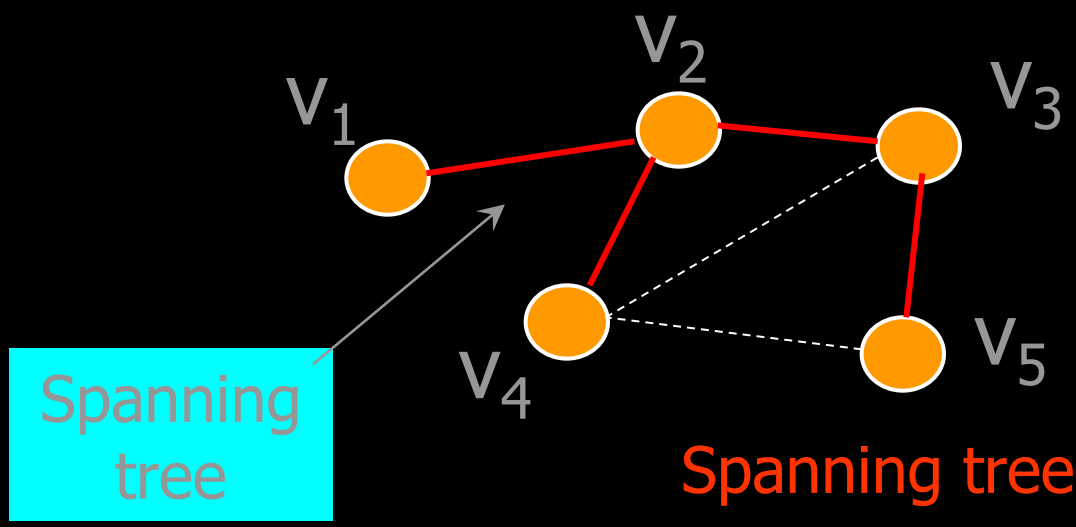
Topological sort algorithm 2

- This algorithm is based on DFS

```
Algorithm topSort2
s.createStack();
for (all nodes v in the graph) {
    if (v has no predecessors) {
        s.push(v);
        mark v as visited;
    }
}
while (!s.isEmpty()) {
    let x be the node on the top of the stack s;
    if (no unvisited nodes are adjacent to x) {        // i.e. x has no unvisited successor
        aList.add(1, x);
        s.pop(); // backtrack
    } else {
        select an unvisited node u adjacent to x;
        s.push(u);
        mark u as visited;
    }
}
return aList;
```

Spanning Tree

- Given a connected undirected graph G, a spanning tree of G is a subgraph of G that contains all of G's nodes and enough of its edges to form a tree.



Spanning tree is not unique!

DFS spanning tree

- Generate the spanning tree edge during the DFS traversal.

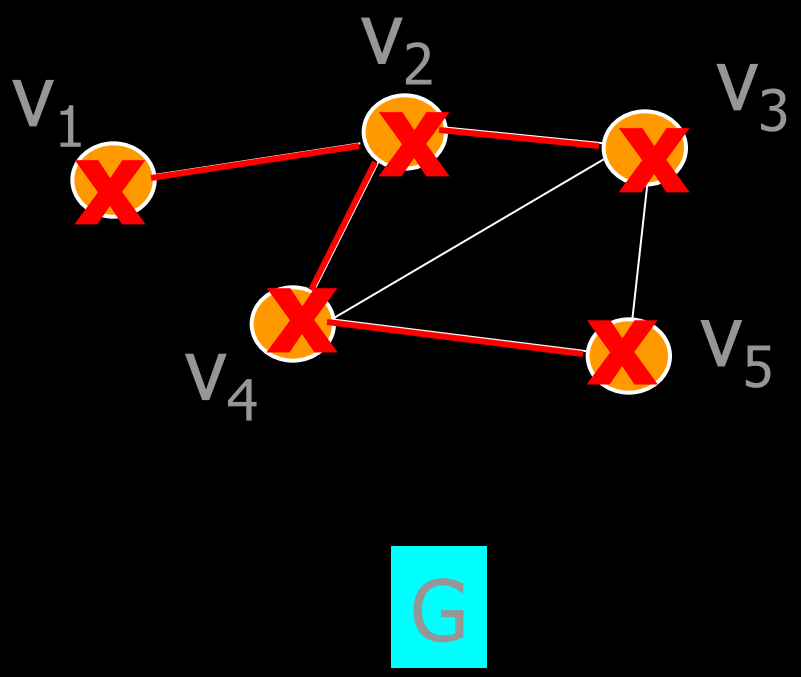
Algorithm dfsSpanningTree(v)

```
mark v as visited;
for (each unvisited node u adjacent to v) {
    mark the edge from u to v;
    dfsSpanningTree(u);
}
```

- Similar to DFS, the spanning tree edges can be generated based on BFS traversal.

Example of generating spanning tree based on DFS

	stack
v_3	v_3
v_2	v_3, v_2
v_1	v_3, v_2, v_1
backtrack	v_3, v_2
v_4	v_3, v_2, v_4
v_5	v_3, v_2, v_4 , v_5
backtrack	v_3, v_2, v_4
backtrack	v_3, v_2
backtrack	v_3
backtrack	empty

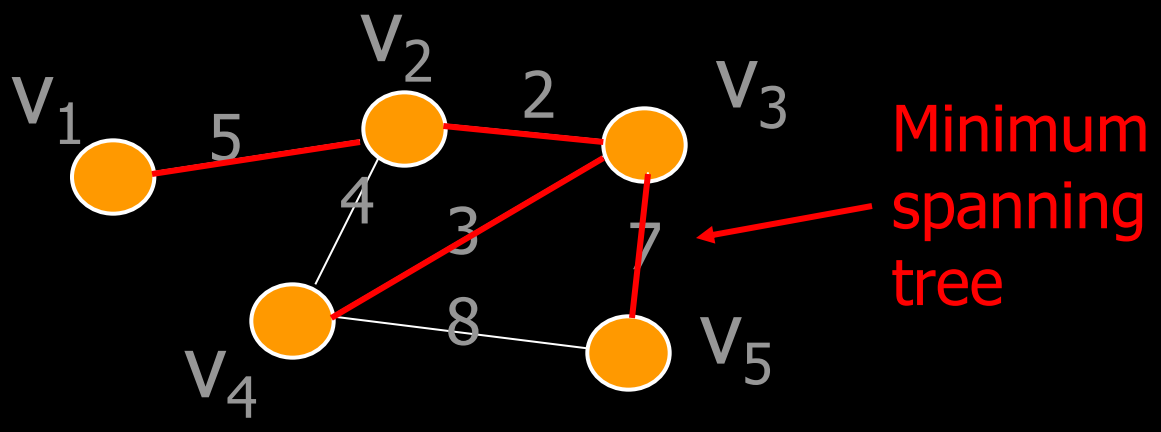


Minimum Spanning Tree

- Consider a connected undirected graph where
  - Each node x represents a country x
  - Each edge (x, y) has a number which measures the cost of placing telephone line between country x and country y
- Problem: connecting all countries while minimizing the total cost
- Solution: find a spanning tree with minimum total weight, that is, minimum spanning tree

Formal definition of minimum spanning tree

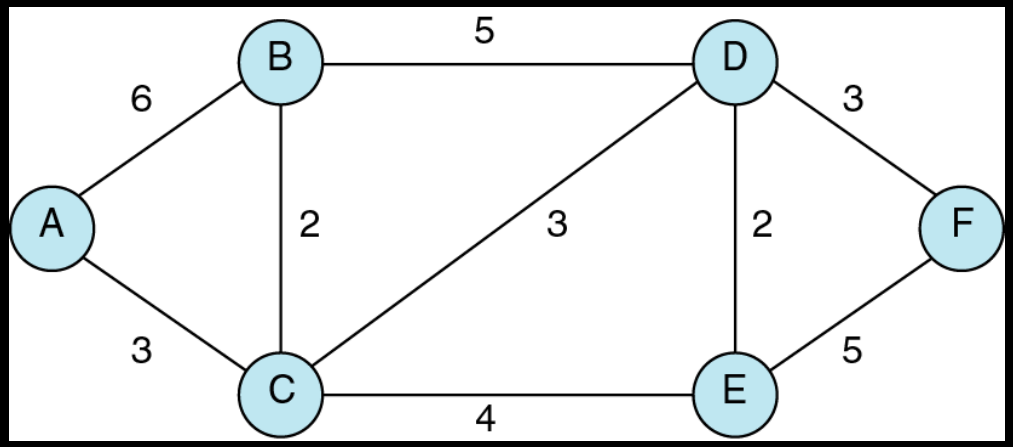
- Given a connected undirected graph G.
- Let T be a spanning tree of G.
- cost(T) = $\sum_{e \in T}$weight(e)
- The minimum spanning tree is a spanning tree T which minimizes cost(T)

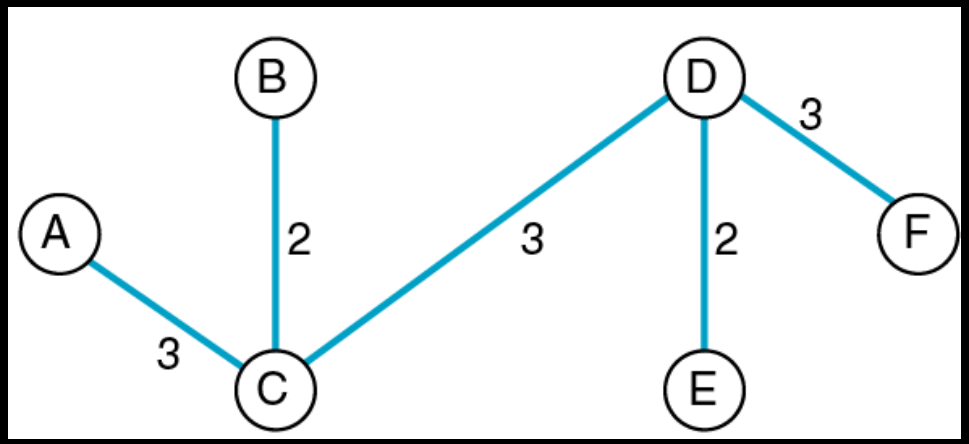


Minimum Spanning Tree

- A ***spanning tree*** is a tree that contains all of the vertices in the graph and enough of its edges to form a tree.
- The ***minimum spanning tree*** of a network is a spanning tree in which the sum of its edge weights is guaranteed to be minimal.
- It is possible to have more than one minimum spanning tree. However, the weights of the different MSTs will be the same.

Minimum Spanning Tree





The bottom graph is a spanning tree of the top graph.

Note that every node from the top graph is represented in the spanning tree.

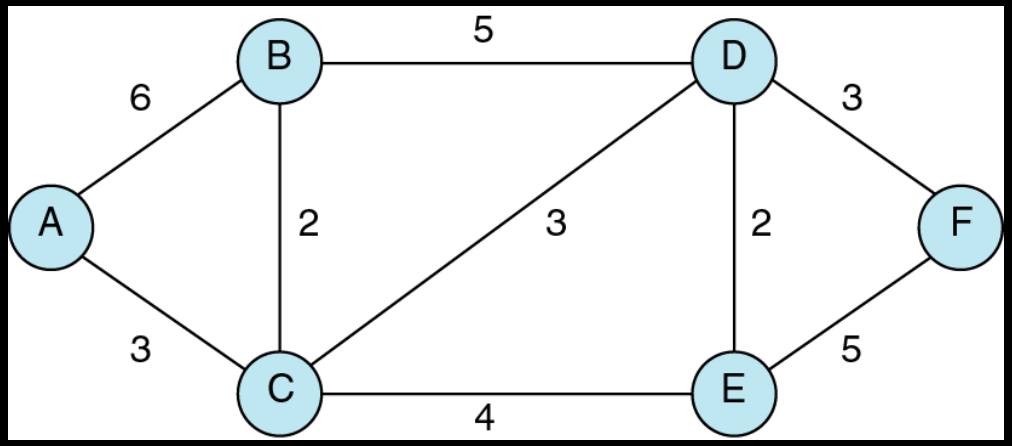
The bottom graph also happens to be the minimum spanning tree.

Minimum Spanning Tree

- There are many applications for minimum spanning trees, all with the requirement to minimize some aspect of the graph, such as the distance among all of the vertices in the graph.
- For example, given a network of computers, we could create a graph that connects all of the computers (think internet hardware).
- The MST gives us the shortest length of cable that can be used to connect all of the computers while ensuring that there is a path between any two computers.

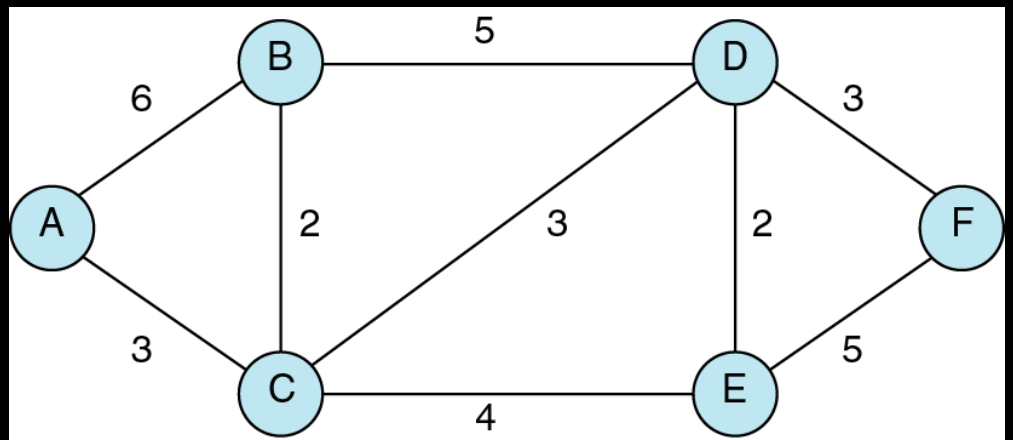
Minimum Spanning Tree

- Before we develop the code to determine the MST of a graph, let's run through the process by hand.

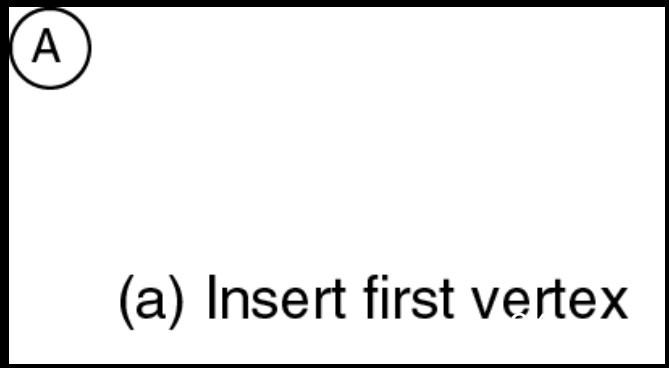


Minimum Spanning Tree

- We can start with any vertex, so we'll just start with A.
- We then add the vertex that gives the minimum-weighted edge with A.
- Our options are AB or AC – we choose AC because 3 is less than 6

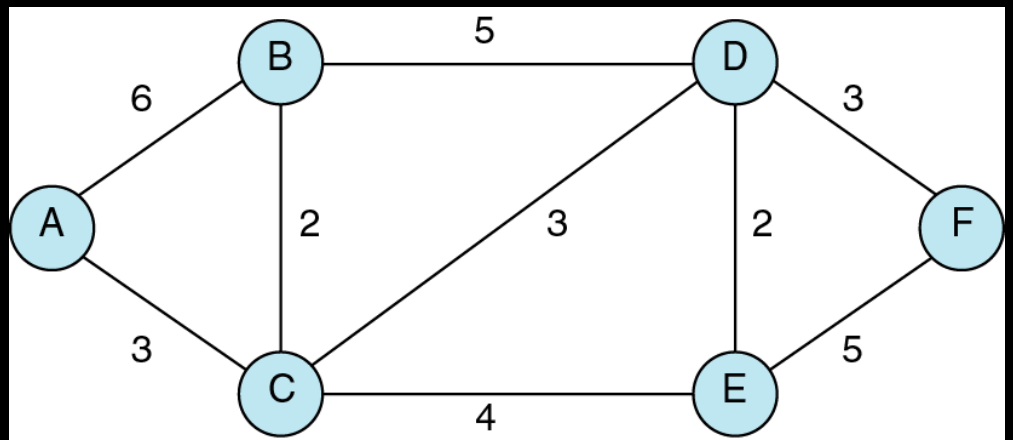


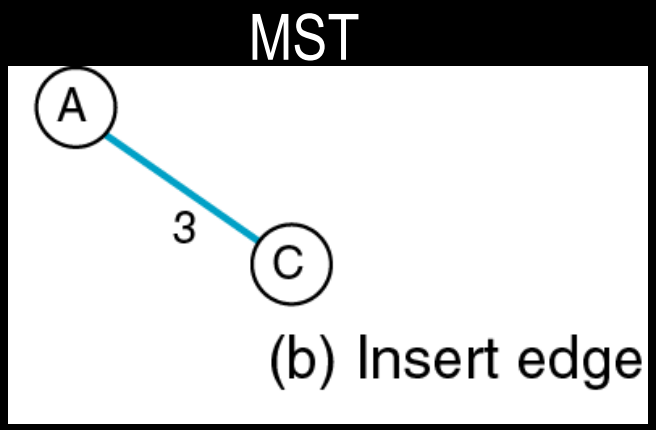
MST



Minimum Spanning Tree

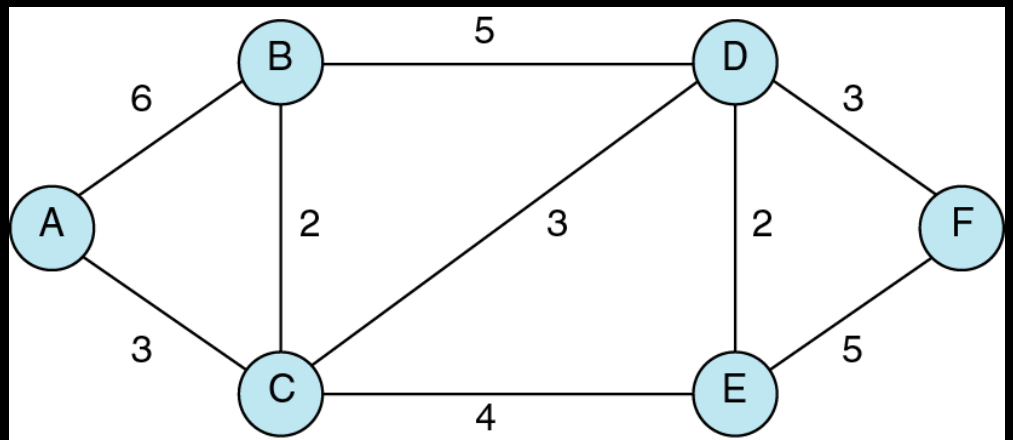
- Now, we have vertices A and C in our MST.
- From these 2, we choose the edge with the minimum weight that connects a vertex in our MST to a vertex not already included in our MST.



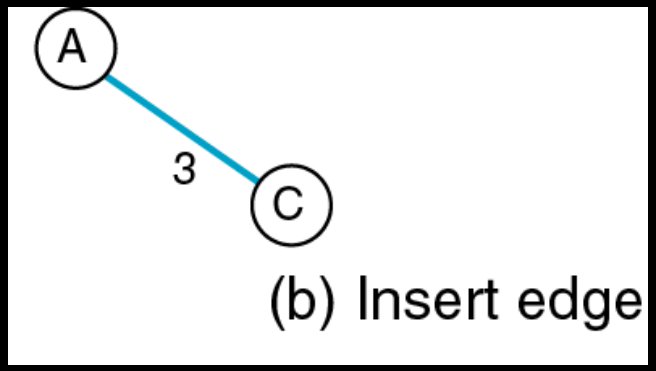


Minimum Spanning Tree

- Our options are
  - CB, weight = 2
  - CD, weight = 3
  - CE, weight = 4
  - AB, weight = 6
- We choose CB because it has the minimum weight.

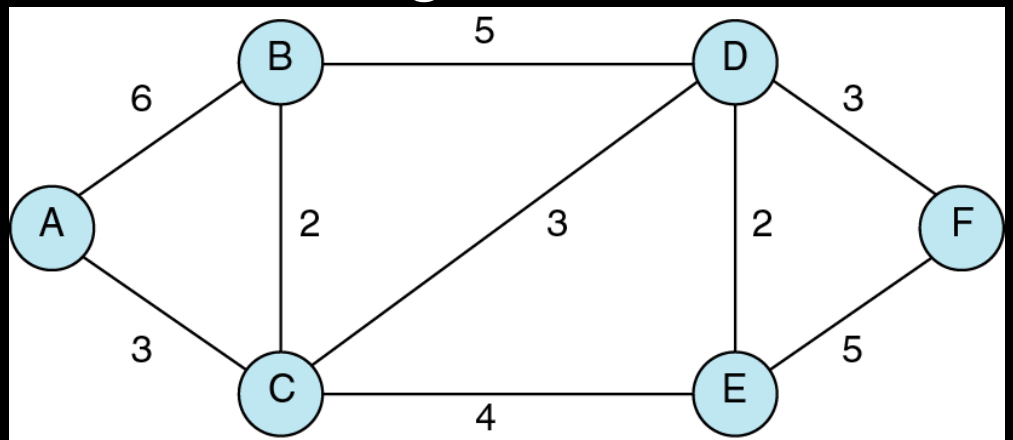


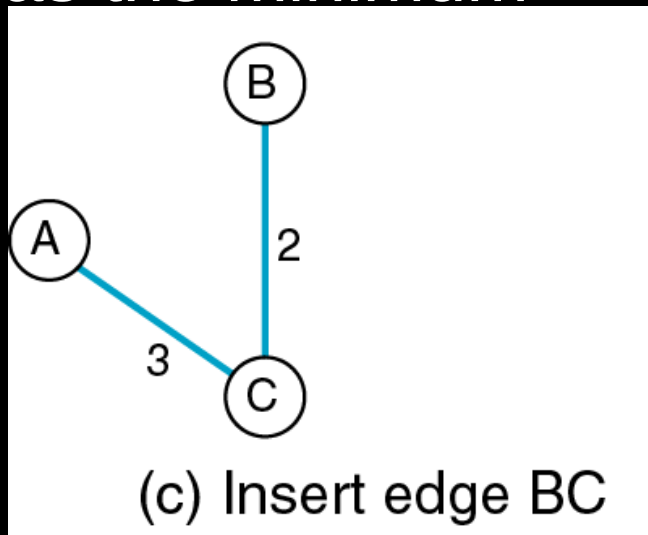
MST



Minimum Spanning Tree

- Now we continue with the same process – here are our choices:
  - BD, weight = 5
  - CD, weight = 3
  - CE, weight = 4
- We choose CD because it has the minimum weight.

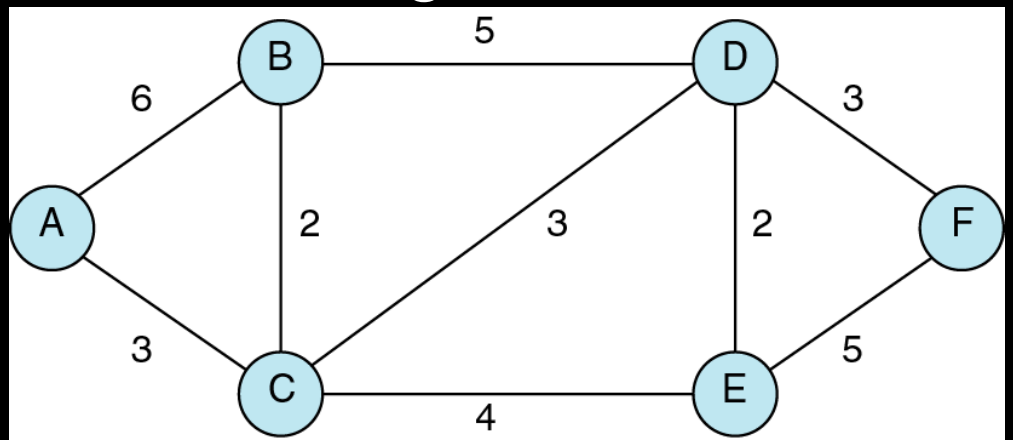


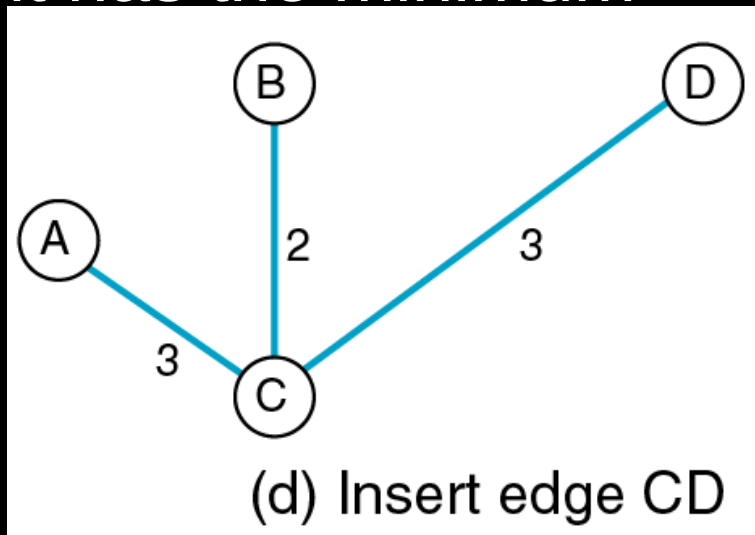


(c) Insert edge BC

Minimum Spanning Tree

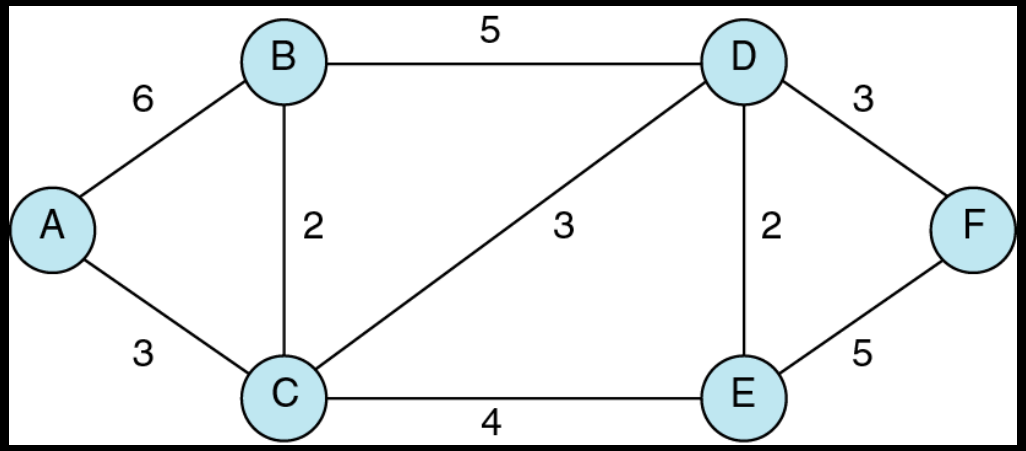
- Now we continue with the same process – here are our choices:
  - CE, weight = 4
  - DE, weight = 2
  - DF, weight = 3
- We choose DE because it has the minimum weight.

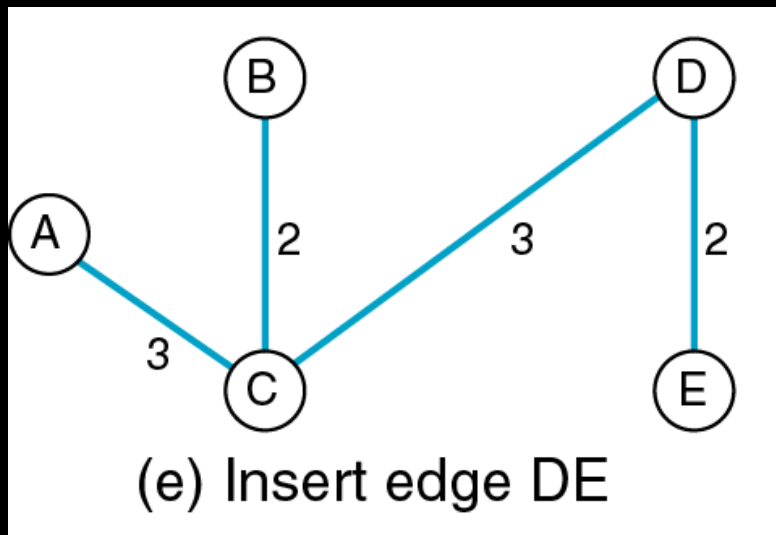




Minimum Spanning Tree

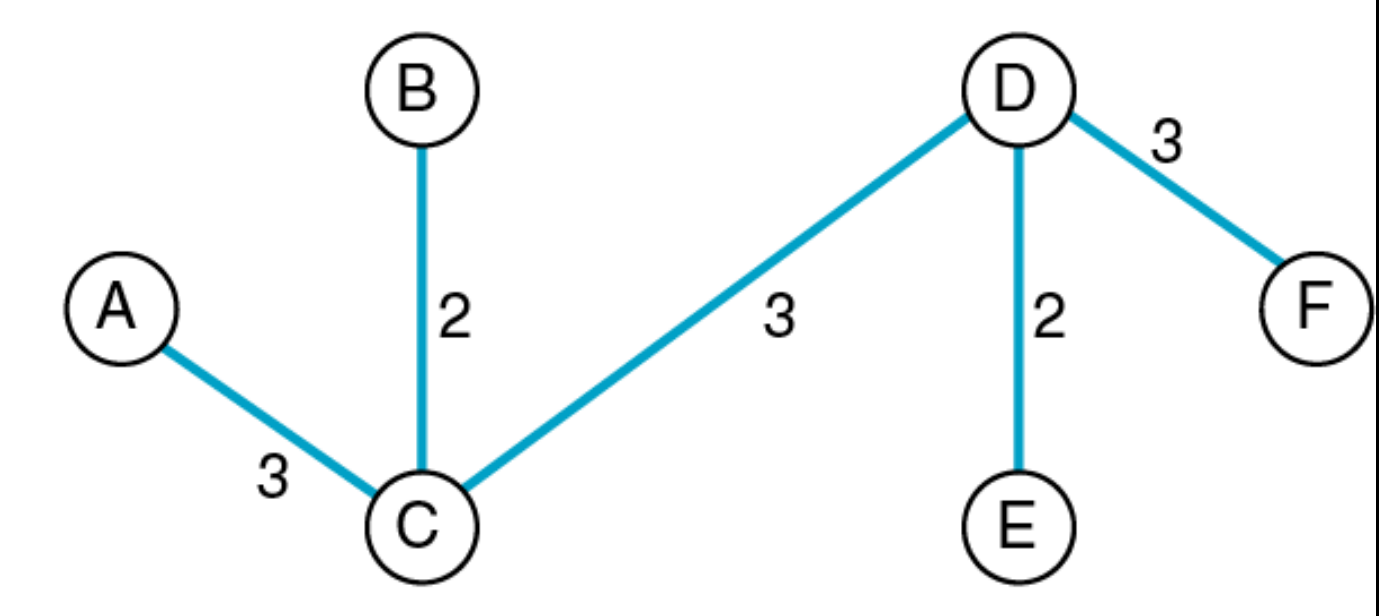
- Now we are down to our last node. Our choices are:
  - EF, weight = 5
  - DF, weight = 3
- We choose DF because it has the minimum weight.





Minimum Spanning Tree

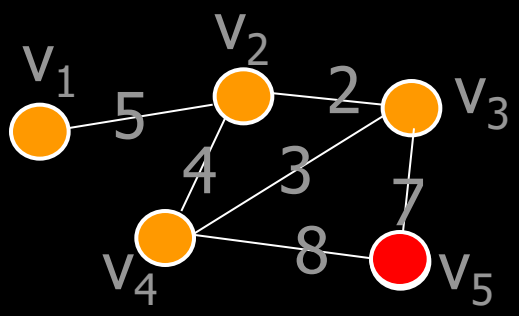
- Here is our final MST.



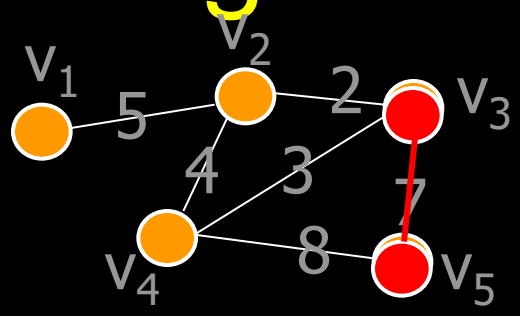
Minimum Spanning Tree

- The algorithm just described is Prim's algorithm.
- Prim's algorithm finds a minimum spanning tree that begins at any vertex.
- Hence, the MST produced by Prim's algorithm may vary depending on which vertex you start at (or if you have two edges that are weighted the same).

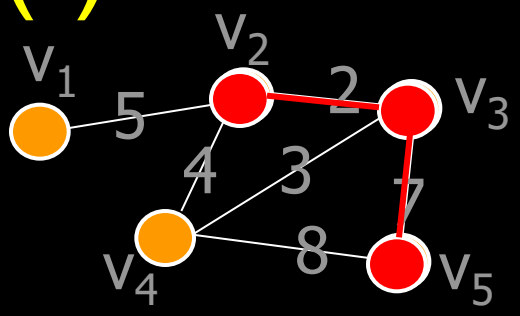
Prim's algorithm (I)



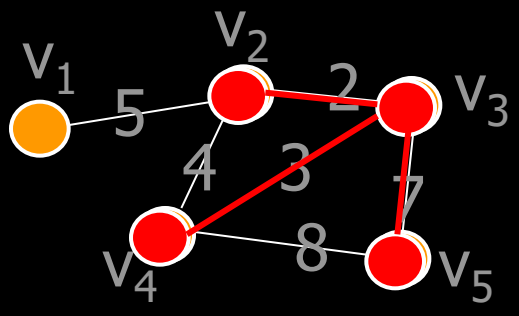
Start from v_5, find the minimum edge attach to v_5



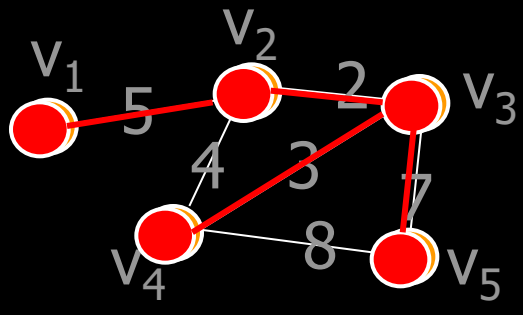
Find the minimum edge attach to v_3 and v_5



Find the minimum edge attach to v_2, v_3 and v_5



Find the minimum edge attach to v_2, v_3 , v_4 and v_5



Prim's algorithm (II)

Algorithm PrimAlgorithm(v)

- Mark node v as visited and include it in the minimum spanning tree;
- while (there are unvisited nodes) {
  - find the minimum edge (v, u) between a visited node v and an unvisited node u;
  - mark u as visited;
  - add both v and (v, u) to the minimum spanning tree;

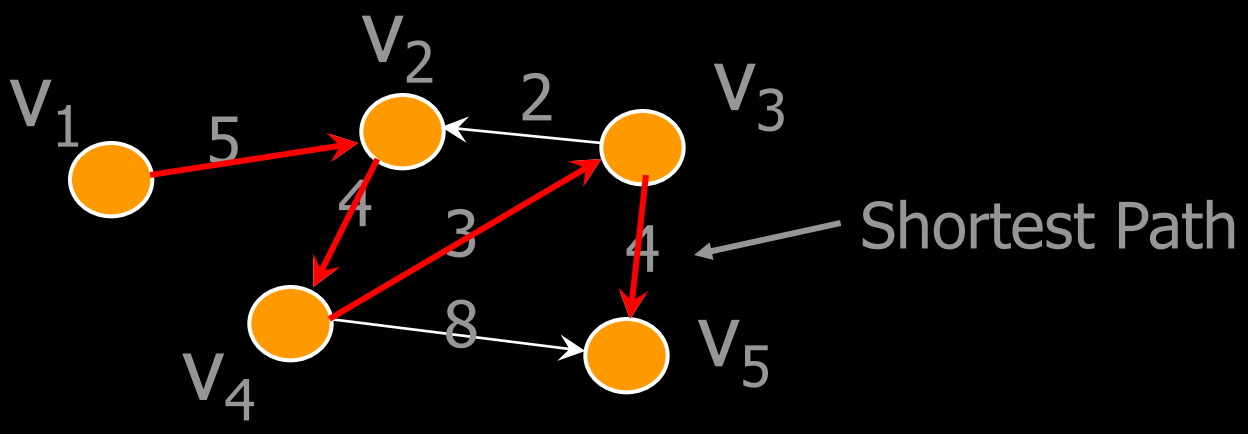
  }

Shortest path

- Consider a weighted directed graph
  - Each node x represents a city x
  - Each edge (x, y) has a number which represent the cost of traveling from city x to city y
- Problem: find the minimum cost to travel from city x to city y
- Solution: find the shortest path from x to y

Formal definition of shortest path

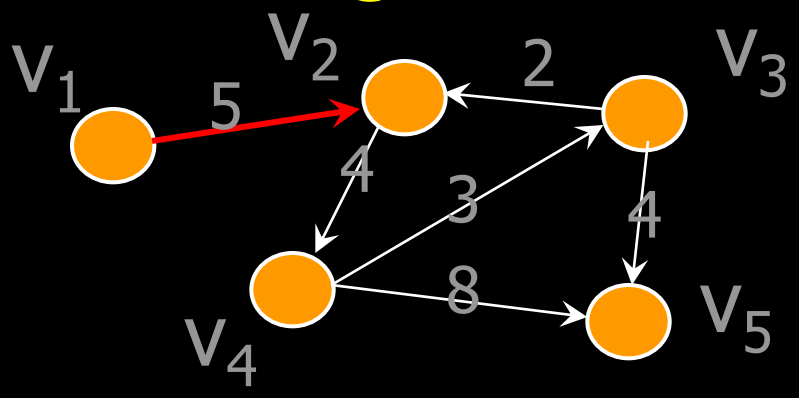
- Given a weighted directed graph G.
- Let P be a path of G from x to y.
- cost(P) = $\sum_{e \in P}$weight(e)
- The shortest path is a path P which minimizes cost(P)



Dijkstra's algorithm

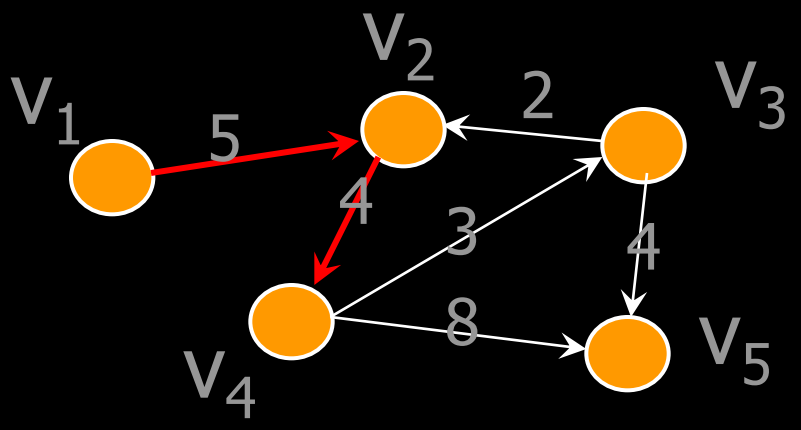
- Consider a graph G, each edge (u, v) has a weight w(u, v) > 0.
- Suppose we want to find the shortest path starting from v_1 to any node v_i
- Let VS be a subset of nodes in G
- Let cost[v_i] be the weight of the shortest path from v_1 to v_i that passes through nodes in VS only.

Example for Dijkstra's algorithm



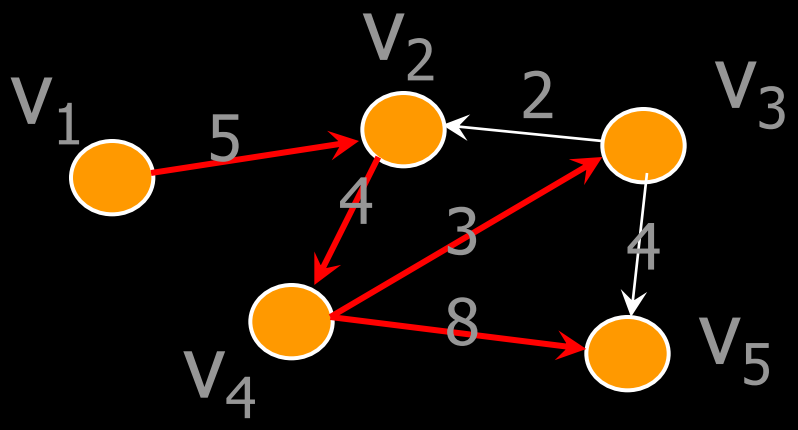
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞

Example for Dijkstra's algorithm



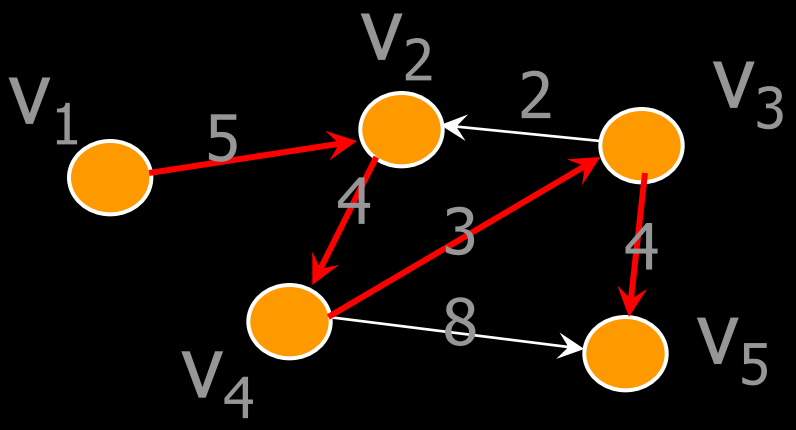
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞

Example for Dijkstra's algorithm



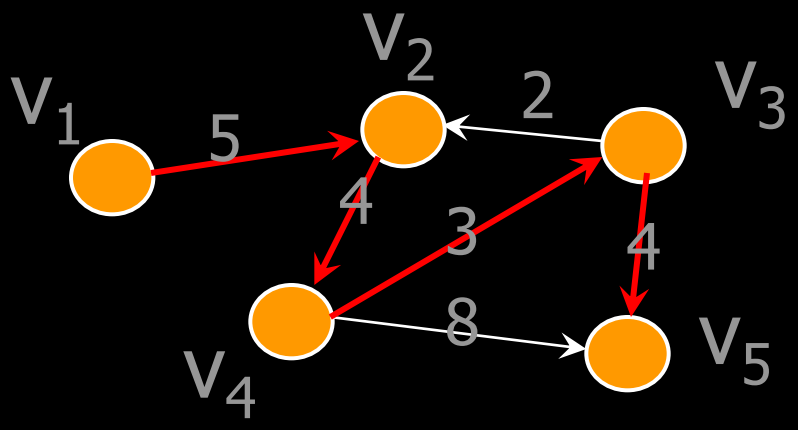
	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17
4	v_3	[v_1, v_2, v_4, v_3]	0	5	12	9	16

Example for Dijkstra's algorithm



	v	VS	cost[v_1]	cost[v_2]	cost[v_3]	cost[v_4]	cost[v_5]
1		[v_1]	0	5	∞	∞	∞
2	v_2	[v_1, v_2]	0	5	∞	9	∞
3	v_4	[v_1, v_2, v_4]	0	5	12	9	17
4	v_3	[v_1, v_2, v_4, v_3]	0	5	12	9	16
5	v_5	[v_1, v_2, v_4, v_3, v_5]	0	5	12	9	16

Dijkstra's algorithm

Algorithm shortestPath()

n = number of nodes in the graph;
for i = 1 to n
    cost[v_i] = w(v_1, v_i);
VS = { v_1 };
for step = 2 to n {
    find the smallest cost[v_i] s.t. v_i is not in VS;
    include v_i to VS;
    for (all nodes v_j not in VS) {
        if (cost[v_j] > cost[v_i] + w(v_i, v_j))
            cost[v_j] = cost[v_i] + w(v_i, v_j);
    }
}

Summary

- Graphs can be used to represent many real-life problems.
- There are numerous important graph algorithms.
- We have studied some basic concepts and algorithms.
  - Graph Traversal
  - Topological Sort
  - Spanning Tree
  - Minimum Spanning Tree
  - Shortest Path