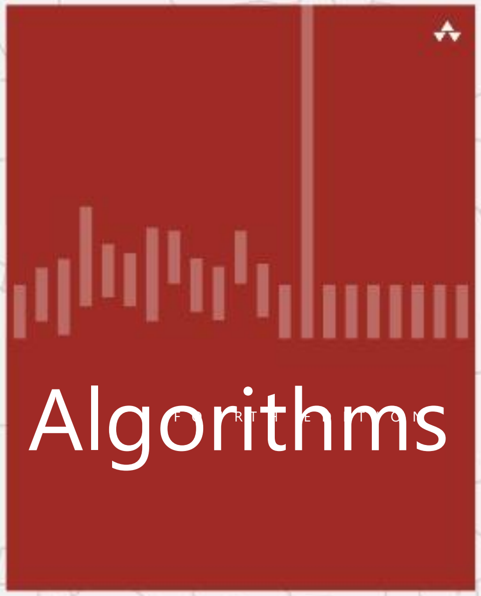
Algorithms: Theory, Design and Implementation



Categories of algorithms

- *Sorting algorithms*
- *Search algorithms*
- ***Graph algorithms***
- *Path-finding in AI algorithms*
- *Network Flow algorithms*
- *Computational Geometry*

Algorithms

GRAPHS

i. Definitions

ii. Graph representation

iii. Depth-first, Breadth-first search algorithms

iv. Single source shortest path

v. All pairs shortest path (Dijkstra's algorithm)

vi. Minimum spanning tree algorithms

(kruskal and Prim algorithms)

Graph Definitions

Introduction to Graphs

Graphs are cool and vital ways of representing information and relationships in the world around us. We can use graphs to do amazing stuff with computers, and graph algorithms offer a lot of tools to understand complex networks and relationships.

At its most basic, a graph is a group of dots connected by lines.

We use graphs to model relationships in the world. For example:

- Google Maps uses a series of dots and lines to model the road network and give you directions to your final destination
- Facebook friend networks are a graph where each person is a dot, and the friendships between people are lines
- The Internet is a giant graph, where web pages are dots and the links between pages are lines

Introduction to Graphs

We can model objects in physical space, relationships between people, and document structures all using graphs, simple dots and lines!

Once we have the relationships modeled, we can:

- Find the shortest path between two points
- Identify groups of relationships
- Store data and create links between it in almost any context (think linked lists and trees)

Graphs are everywhere, all around you! But chances are you don't really understand them.

Definitions

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge.

- G = (V,E)
- V is the vertex set.

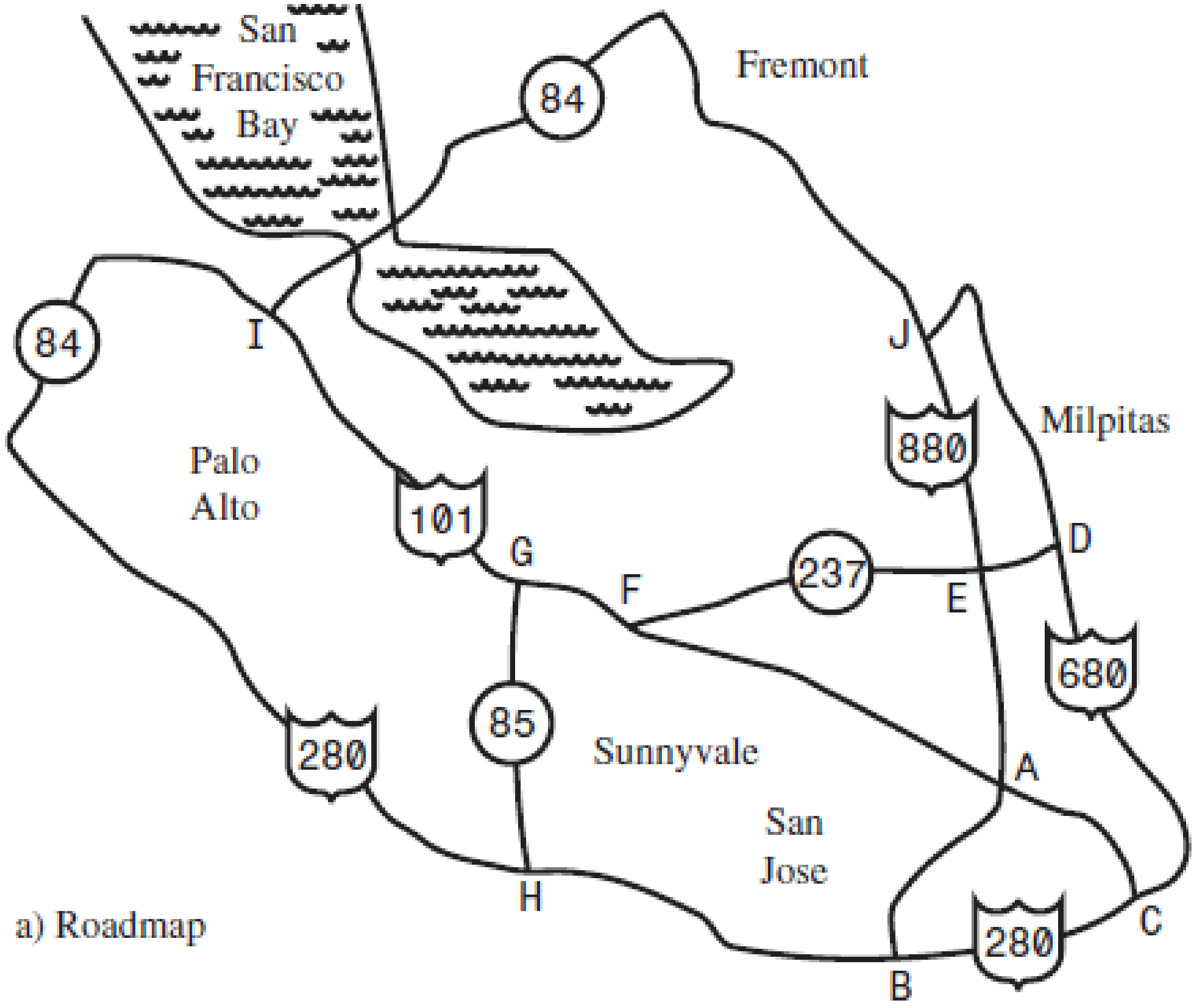
  Vertices are also called nodes and points.
- E is the edge set.

  Each edge connects two different vertices.

  Edges are also called arcs and lines.
- Directed edge has an orientation (u,v).

$$\begin{tikzcd} u \arrow[r] & v \end{tikzcd}$$

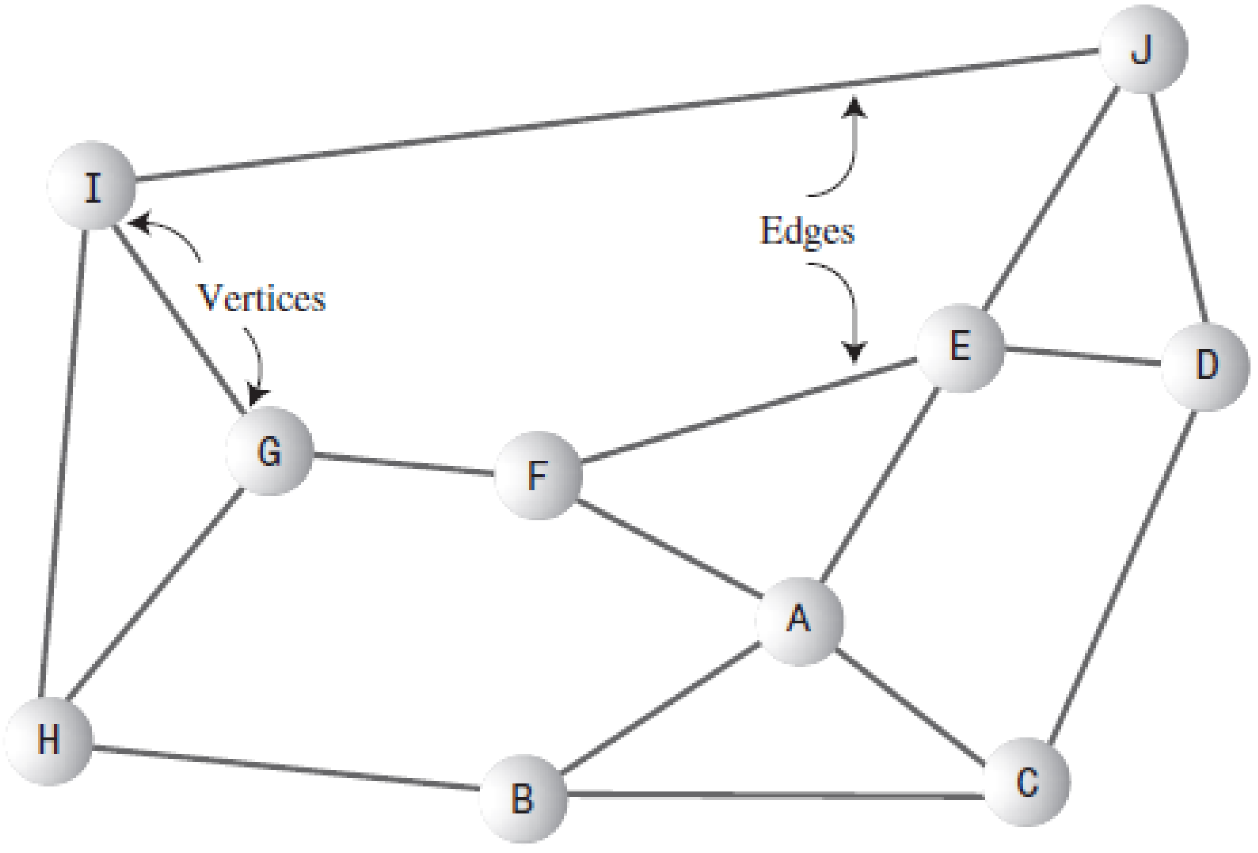
A Road Map of Silicon Valley



a) Roadmap

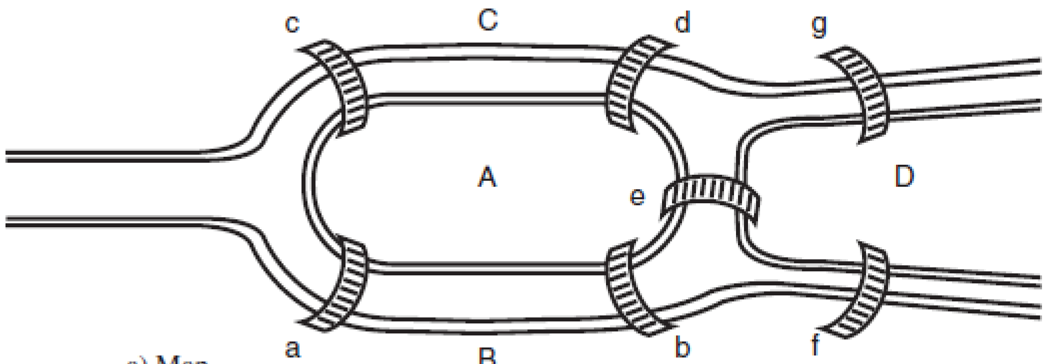
@Sudharshan Welihinda - 2021

...and its corresponding Graph

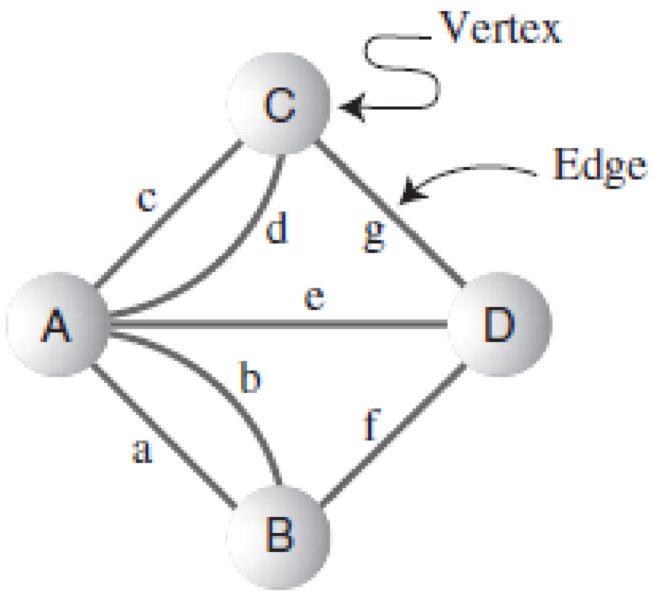


b) Graph

Historical Note: Euler and the Bridges of Königsberg
First real-world graph application

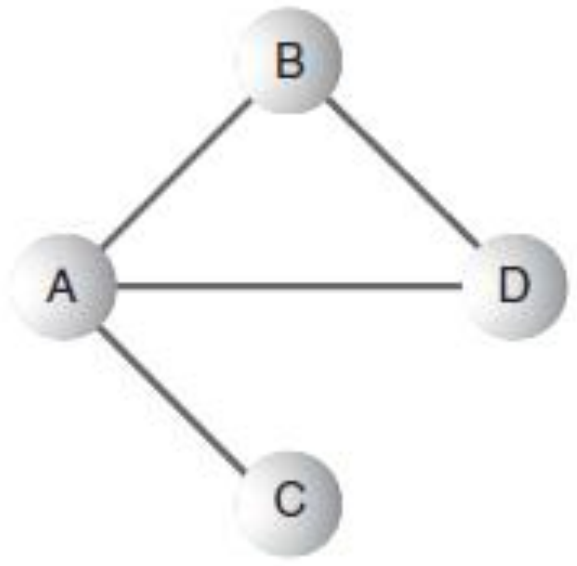


a) Map

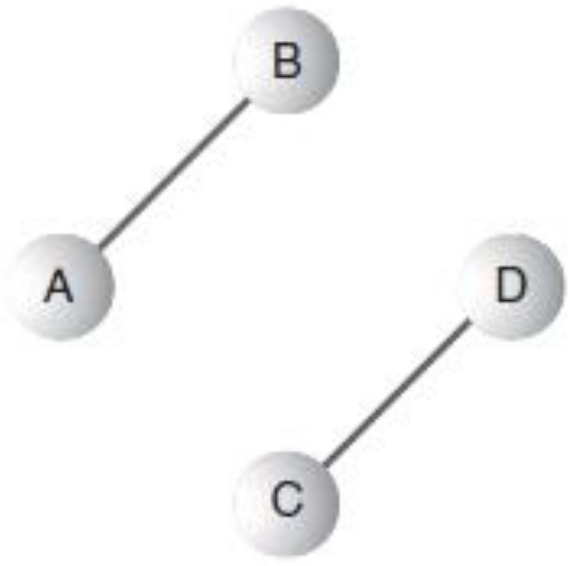


b) Graph

Connected vs. **Non-connected** Graphs



a) Connected Graph



b) Non-connected Graph

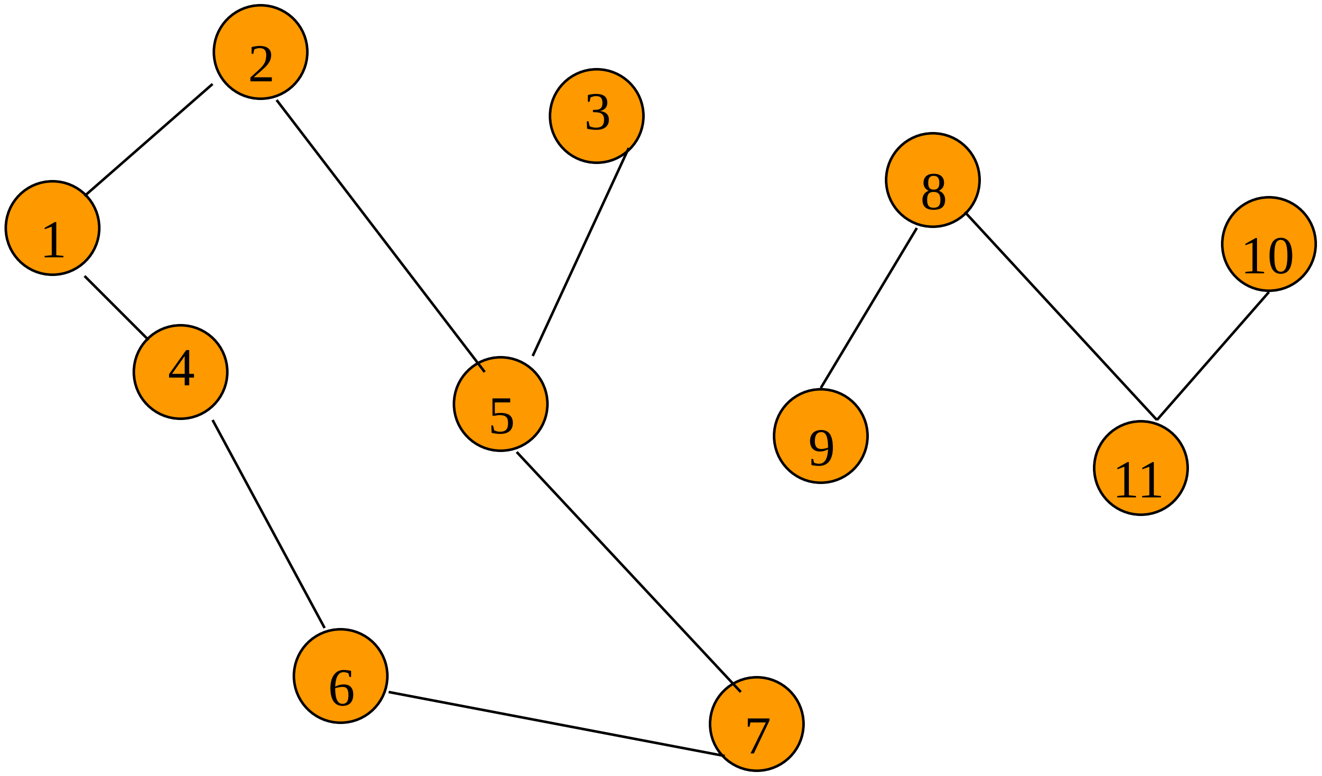
Directed vs. **Undirected** Graphs

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

$$\begin{tikzcd} u \arrow[r, dash] & v \end{tikzcd}$$

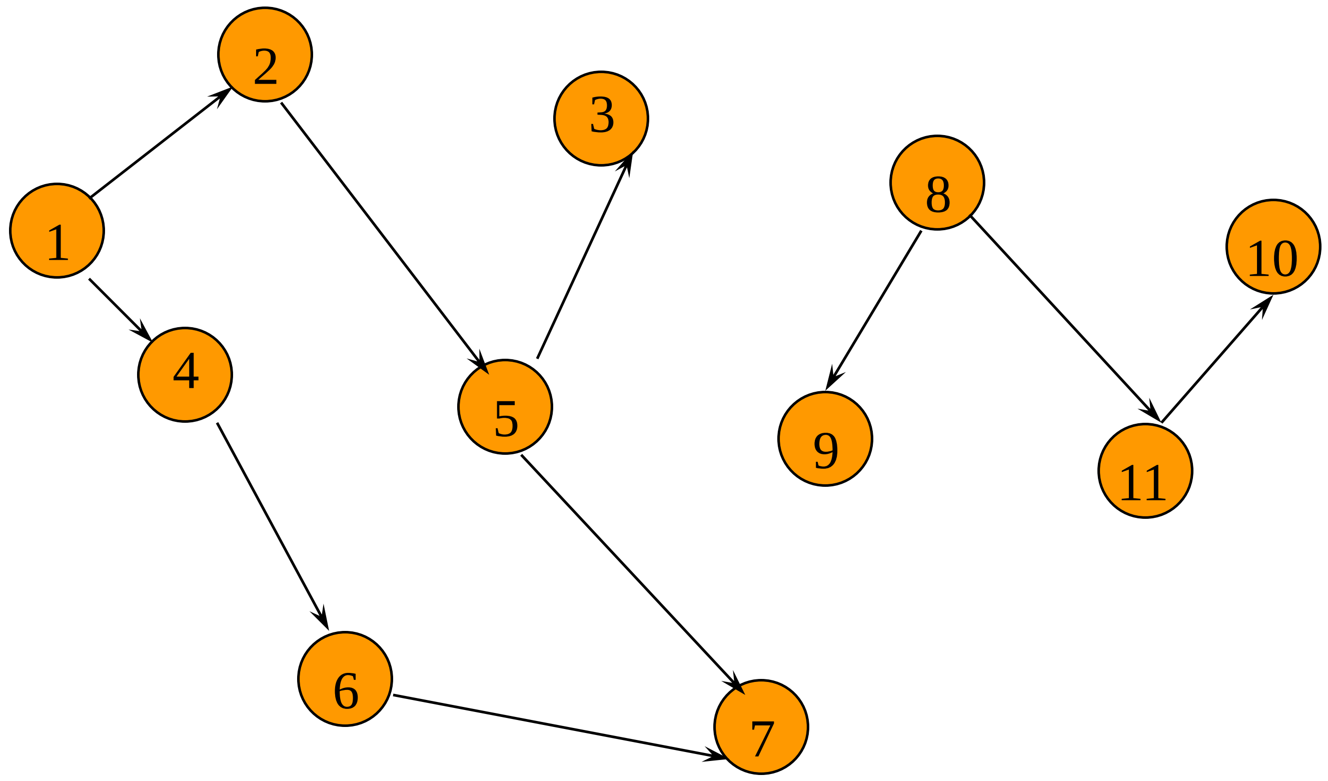
- Undirected graph => no oriented edge.
- Directed graph => every edge has an orientation.

Undirected Graph

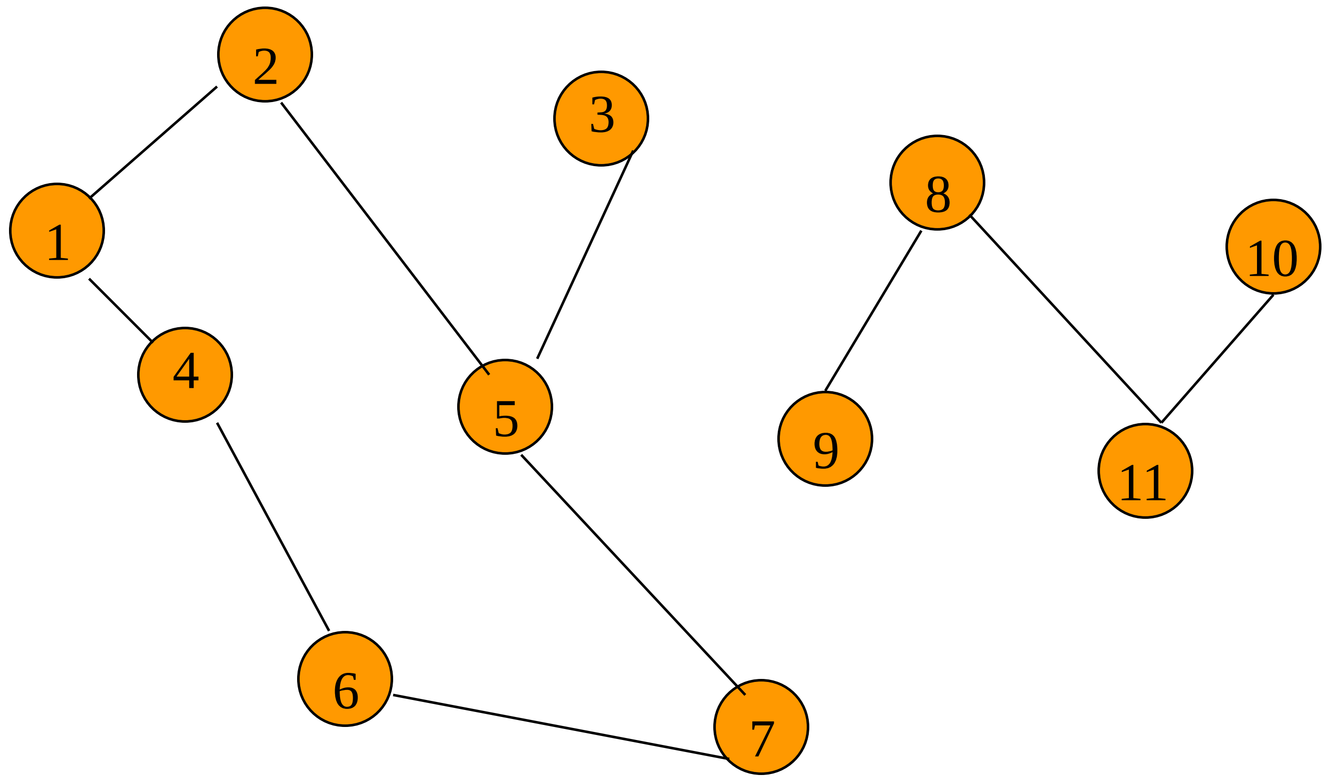


@Sudharshan Welihinda - 2021

Directed Graph (Digraph)

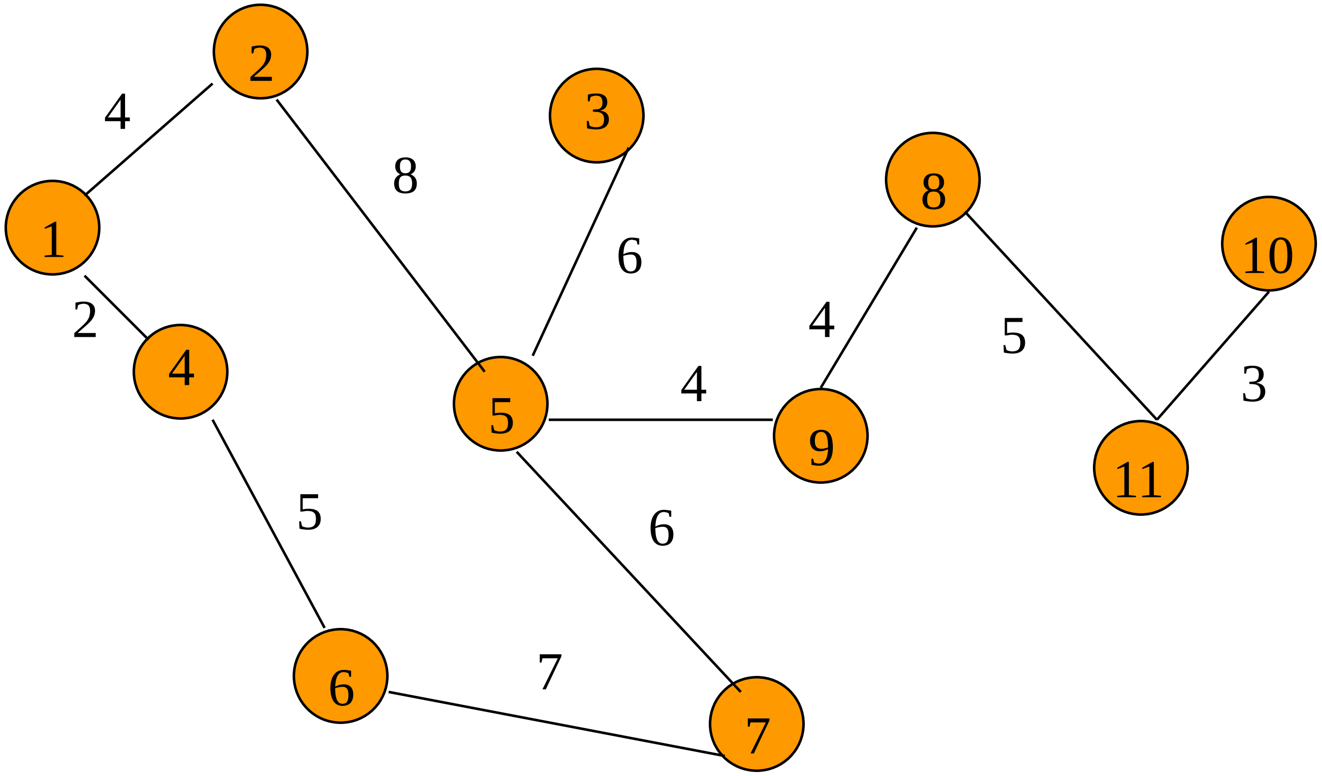


Applications—Communication Network



- Vertex = city, edge = communication link.

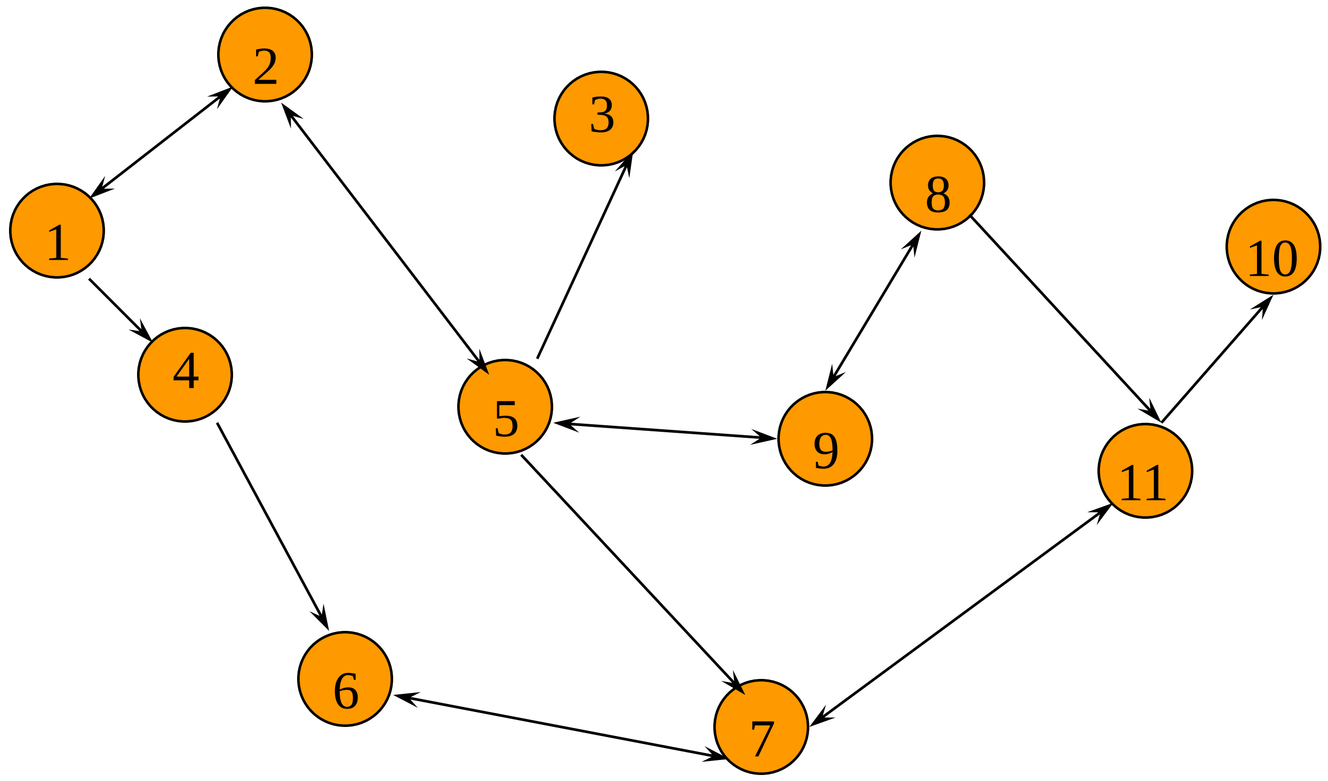
Driving Distance/Time Map
A weighted graph



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

Street Map

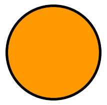
A hybrid (un)directed graph

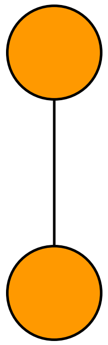


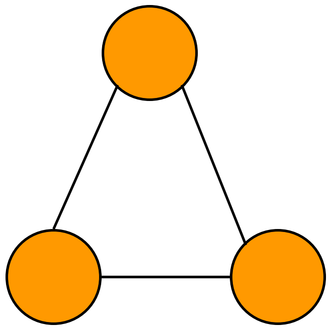
- Some streets are one way.

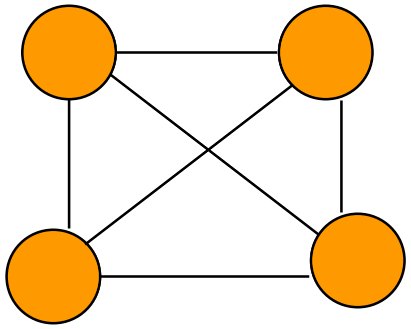
Complete Undirected Graph

Has all possible edges.


n = 1


n = 2


n = 3


n = 4

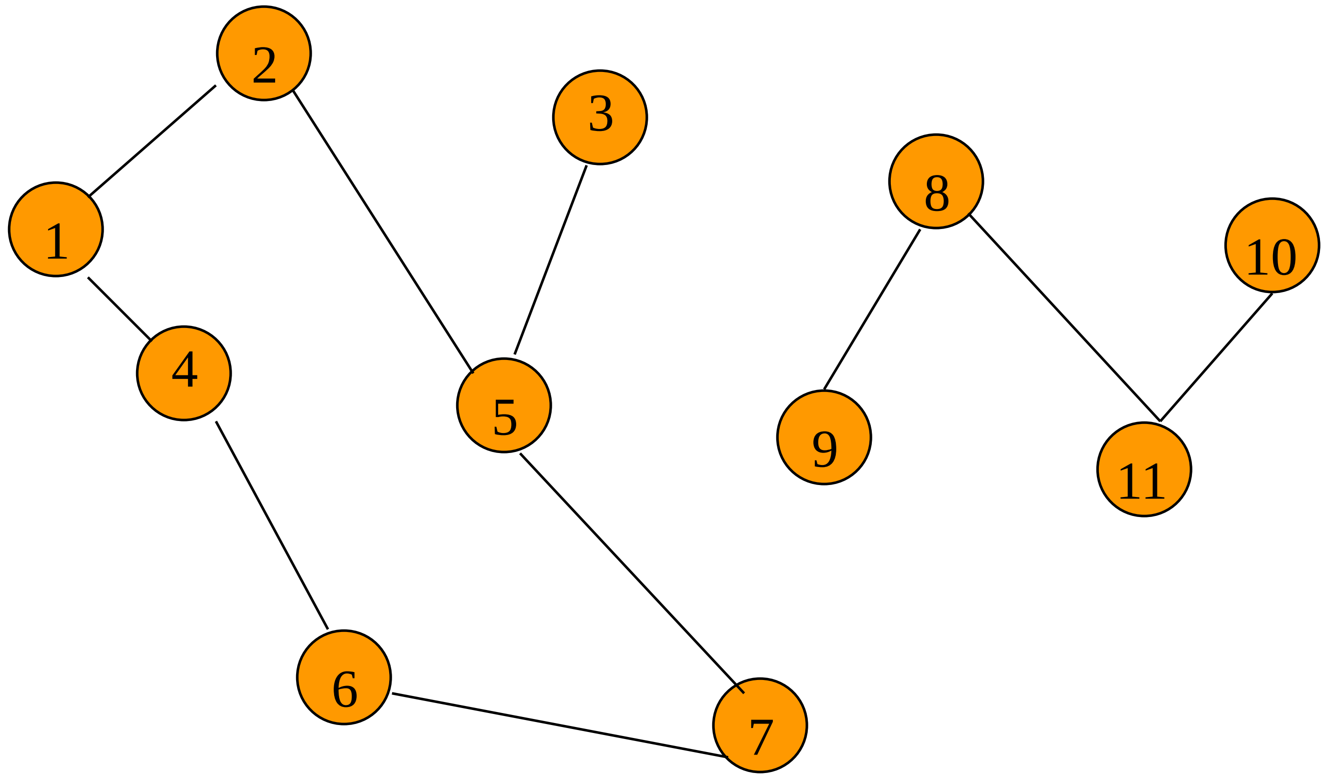
Number of Edges—Undirected Graph

- Each edge is of the form (u,v), u != v.
- Number of such pairs in an n vertex graph is n(n-1).
- Since edge (u,v) is the same as edge (v,u), the number of edges in a complete undirected graph is n(n-1)/2.
- Number of edges in an undirected graph is $\leq$ 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 $\leq$ n(n-1).

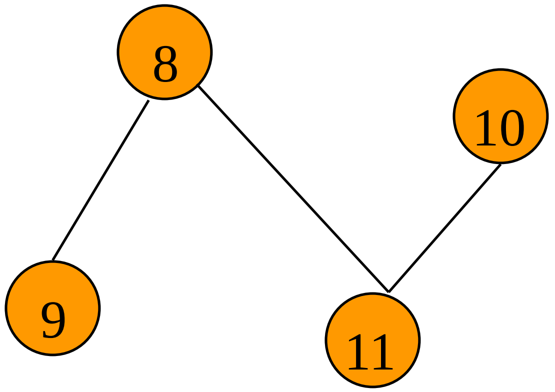
Vertex Degree



Number of edges incident to vertex.

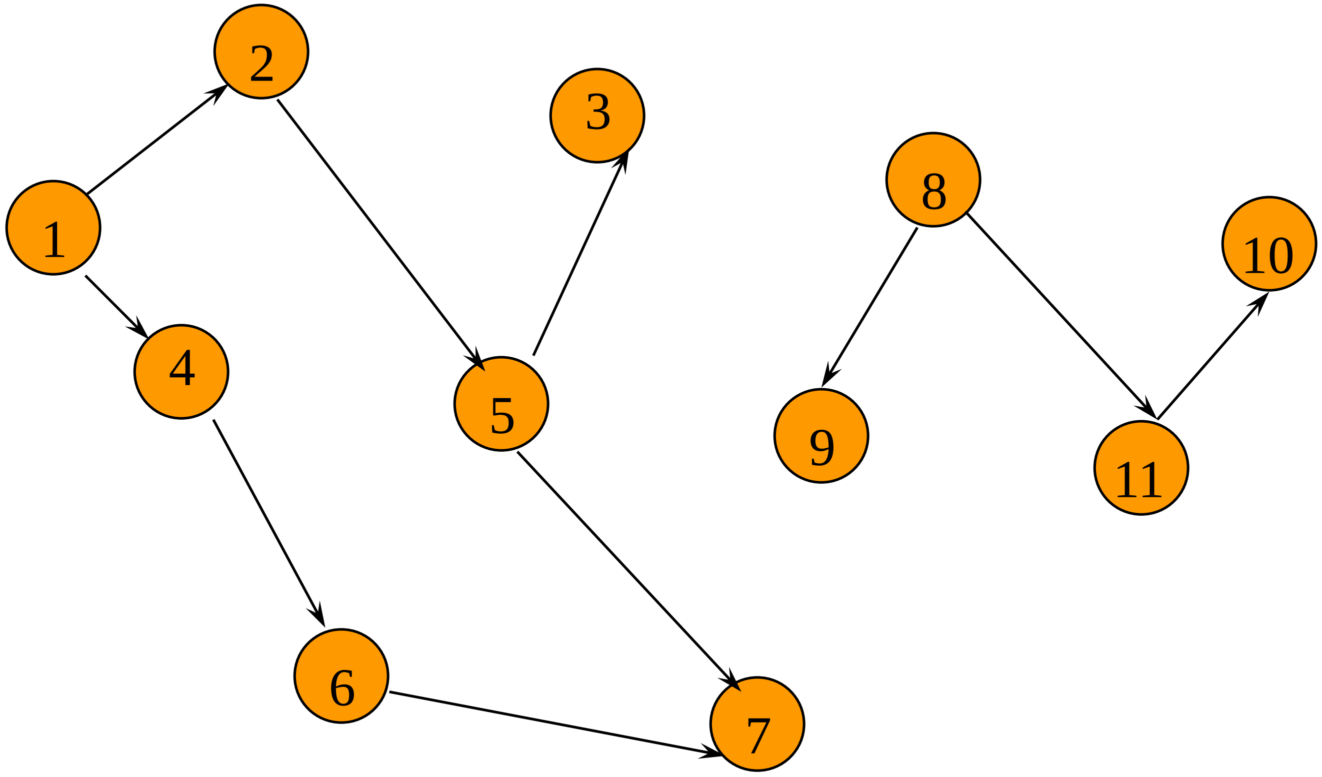
degree(2) = 2, degree(5) = 3, degree(3) = 1

Sum of Vertex Degrees



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

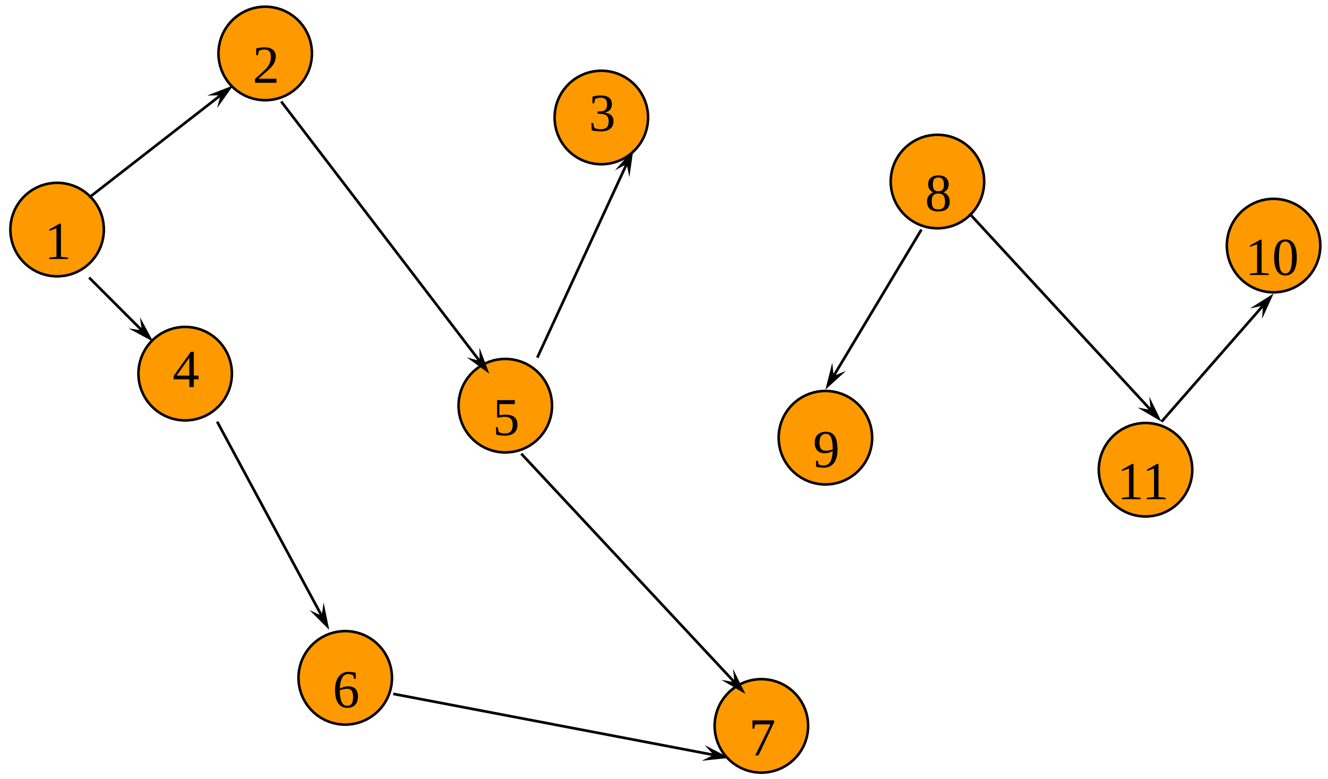
In-Degree of a Vertex



In-Degree is number of incoming edges

In-Degree(2) = 1, In-Degree(8) = 0

Out-Degree of a Vertex



Out-Degree is number of outbound edges

Out-Degree(2) = 1, Out-Degree(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

Some important graphs:

- A *planar* graph - vertices and edges can be drawn in a plane such that no two of the edges intersect
- A *cycle* graph - length $n \geq 3$, vertices can be named v_1, ..., v_n so that the edges are $v_{i-1}v_i$ for each $i = 2,\ldots,n$ in addition to v_nv_1
- A *bipartite* graph - the vertex set can be partitioned into two sets, *W* and *X*, so that no two vertices in *W* are adjacent and no two vertices in *X* are adjacent
- A *tree* graph

Graph Representations

How to represent a *Vertex* in Java

```
class Vertex
   {
   public char label;         // label (e.g. 'A')
   public boolean wasVisited;

   public Vertex(char lab)    // constructor
      {
      label = lab;
      wasVisited = false;
      }
   }  // end class Vertex
```

Fine with Vertices, but how about *Edge Representation*?

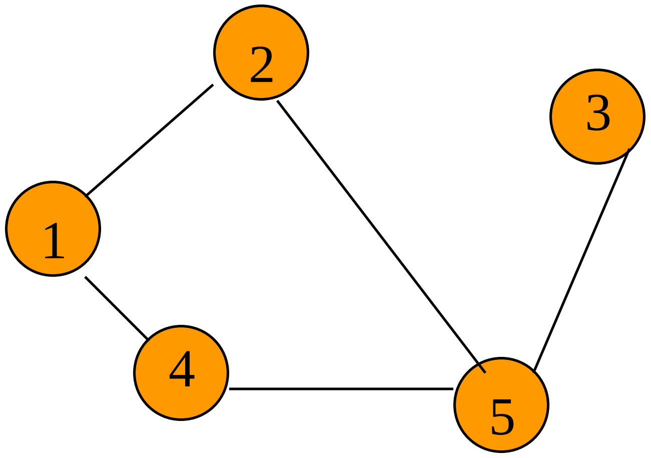
- *Unlike trees, free-form since any vertex can be connected with any number of other vertices…*

  Therefore,

  - **Adjacency Matrix**
  - **Adjacency Lists**

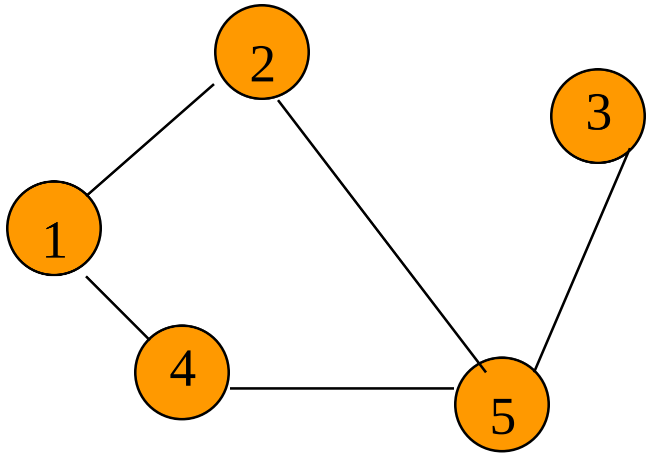
Adjacency Matrix

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



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

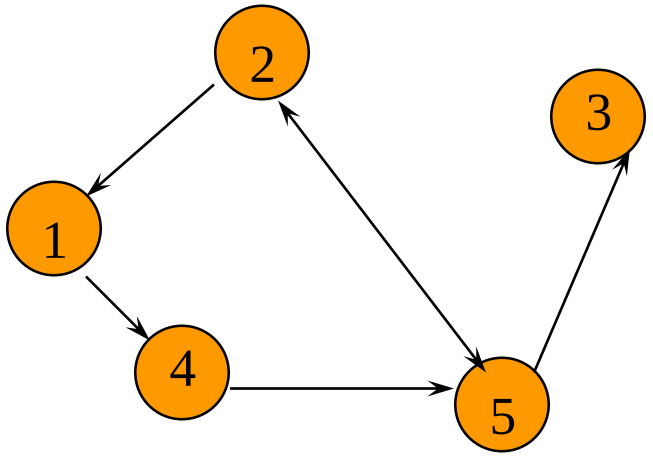
Adjacency Matrix Properties



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

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

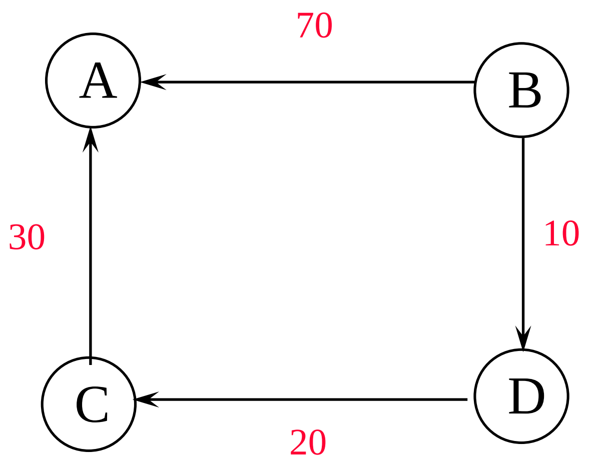
Adjacency Matrix (Digraph)



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

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

Representation of Weighted Graph



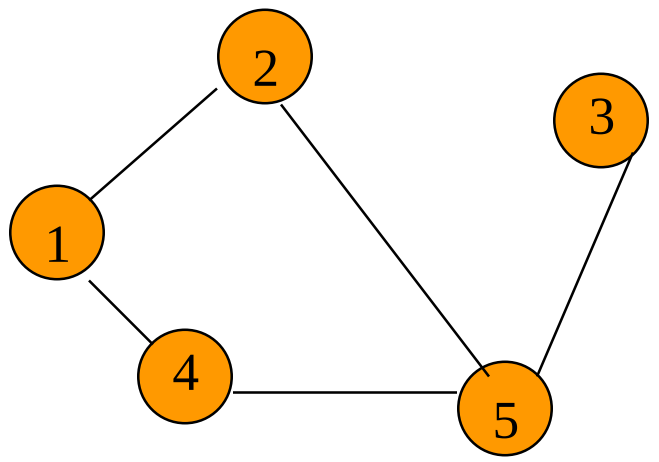
	A	B	C	D
A				
B	70			10
C	30			
D			20	

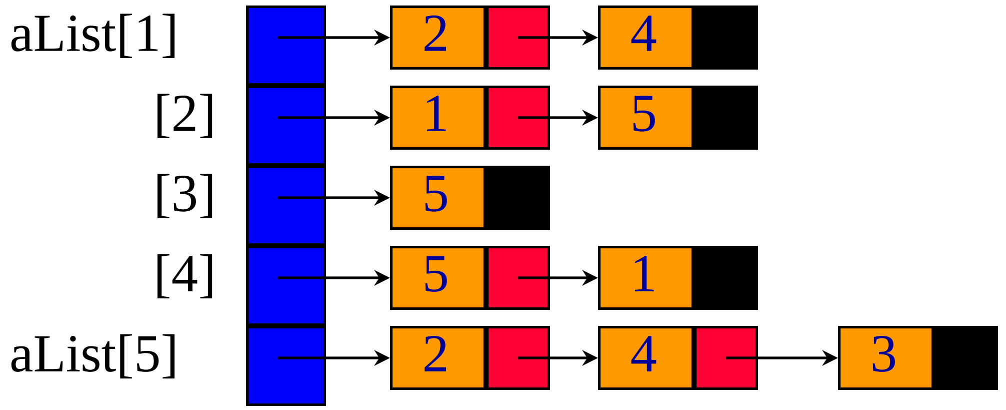
Adjacency Matrix

- 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.

Linked Adjacency Lists

- Each adjacency list is a chain.





Array Length = n

# of chain nodes = 2e (undirected graph)

# of chain nodes = e (digraph)

Question 1

Let us assume you are given a graph with vertices {A,B,C,D} and the relationships and the edges {A leads to B}, {B leads to A,C,D} , {C leads to B}, {D leads to B}

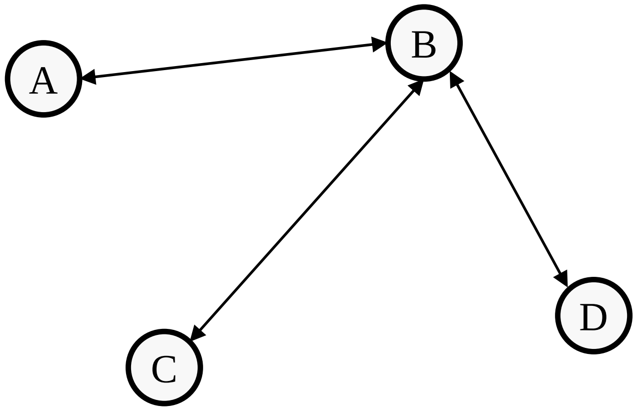
a) Provide the corresponding adjacency matrix in terms of {0,1} elements, which reflects the graph above, and justify whether the feature of symmetry for the matrix hold or not.

b) Subsequently, provide an alternative representation of the graph based to be implemented with the help of an array.

Question 2

Let us assume you are given four friends, {Soniya, Anna, Matilda, Dora}, and the relationships {*Soniya likes Anna*}, {*Anna likes Soniya, Matilda, Dora*}, {*Matilda likes Anna*}, {*Dora likes Anna*}. Suggest a data structure and its two alternative implementations capable of representing the knowledge about the four friends above.

Question 1 [12 Marks]



Adjacency Matrix

	A	B	C	D
A	0	1	0	0
B	1	0	1	0
C	0	1	0	0
D	0	1	0	0

Each correct row will get a 1. mark. i.e. 1 * 4 rows = **[4 Marks]**

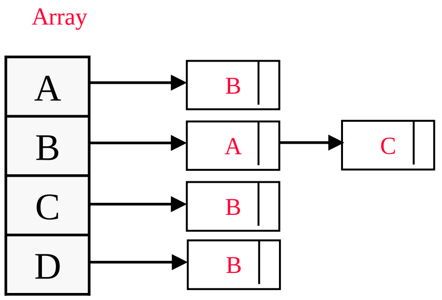
The matrix is symmetrical.

Justification:

Since the given edges are not directed. A careful consideration of all given relationships proves that they all lead to each other, though the meaning of “leads to” relationship is directed.

[3 Marks]

Adjacency List



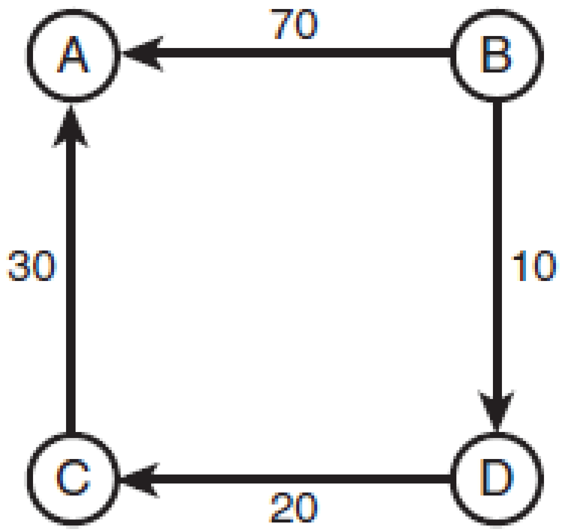
Each correct array node list entry will get 1 mark. i.e 1 * 4 (node) = 4 marks + 1 mark for mentioning the alternative method as “Adjacency List” **[5 Marks]**

Note : The marking scheme can get changed depending on the question. This is just to give an idea to you based on the past years marking schemes.

Some Abstract Methods of Graph

```
// ADT methods
public abstract int vertices();
public abstract int edges();
public abstract boolean existsEdge(int i, int j);
public abstract void putEdge(Object theEdge);
public abstract void removeEdge(int i, int j);
public abstract int degree(int i);
public abstract int inDegree(int i);
public abstract int outDegree(int i);
```

Can you represent this WEIGHTED graph with an *adjacency list*?



	A	B	C	D
A				
B	70			10
C	30			
D			20	

A weighted graph and its adjacency matrix.

How do I walk through the graph?
SEARCH

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 (BFS)

- The **breadth-first traversal**, or **search**, algorithm visits all vertices adjacent to the starting vertex, then visits all vertices adjacent to those vertices , and so on.

  Since all adjacent vertices are visited before probing further, the search is broad rather than deep, hence the term "**breadth-first**".

- The following gives a breadth-first traversal, or search, of a graph stored in matrix 'm' (starting from vertex 1):

```
BFSearch:
   FOR v<--1 TO maxVertex DO
        visited[v] = false;
   ENDFOR
   FOR v<--1 TO maxVertex DO
        IF (!visited[v]) THEN
             bfs(v);
        ENDIF
   ENDFOR
```

```
bfs(vertex v):
   Q <-- empty_queue;    /* FIFO queue structure */
   visited[v] = true;
   add v to rear of Q;
   WHILE Q is not empty DO
         u <-- remove front item in Q;
         FOR w<--1 TO maxVertex DO
                 IF ((m[u][w] == true) AND (!visited[w])) THEN
                      visited[w] = true;
                      add w to rear of Q;
                 ENDIF
         ENDFOR
   ENDWHILE
```

- Note that this algorithm is not recursive. It employs a **queue** data structure to store the neighbours of a given vertex.

Breadth-First Search

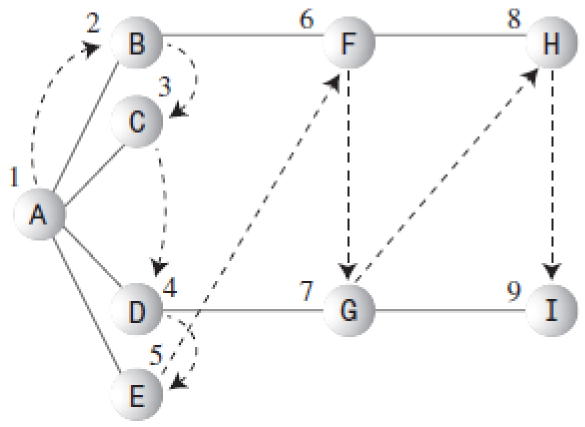
- Visit start vertex and put into a FIFO queue.
- Repeatedly remove a vertex from the queue, visit its unvisited adjacent vertices, put newly visited vertices into the queue.

Pseudocode of BFS

- put starting node in the queue
- while queue is not empty
  - get first node from the queue, name it v
  - (process v)

```
        for each edge e going from v to other nodes
            put the other end of edge e at the end of the queue
```

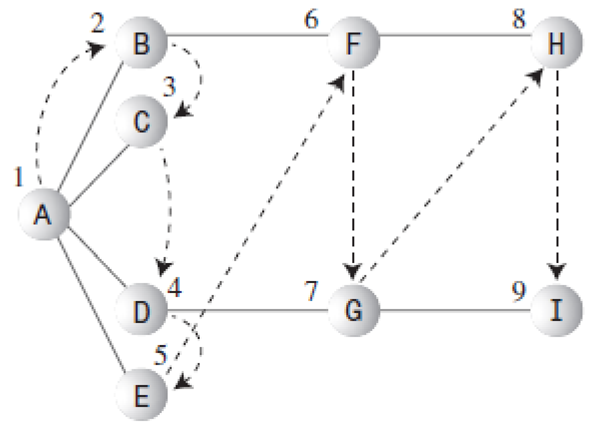
Breadth-first Search with a Queue



Breadth-first search.

Event	Queue(Front & Rear)
Visit A	
Visit B	B
Visit C	B C
Visit D	B C D
Visit E	B C D E
Remove B	C D E
Visit F	C D E F
Remove C	D E F
Remove D	E F
Visit G	E F G
Remove E	F G
Remove F	G
Visit H	G H
Remove G	H
Visit I	H I
Remove H	I
Remove I	

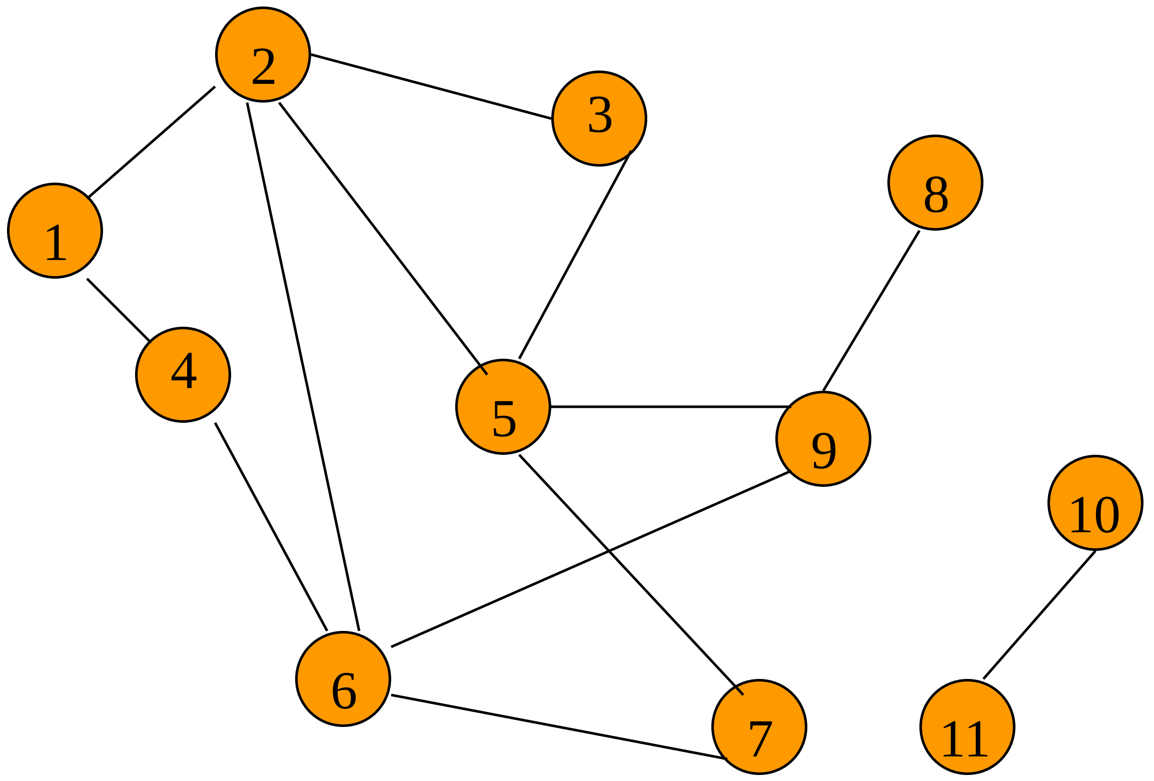
Breadth-first Search with a Queue



Breadth-first search.

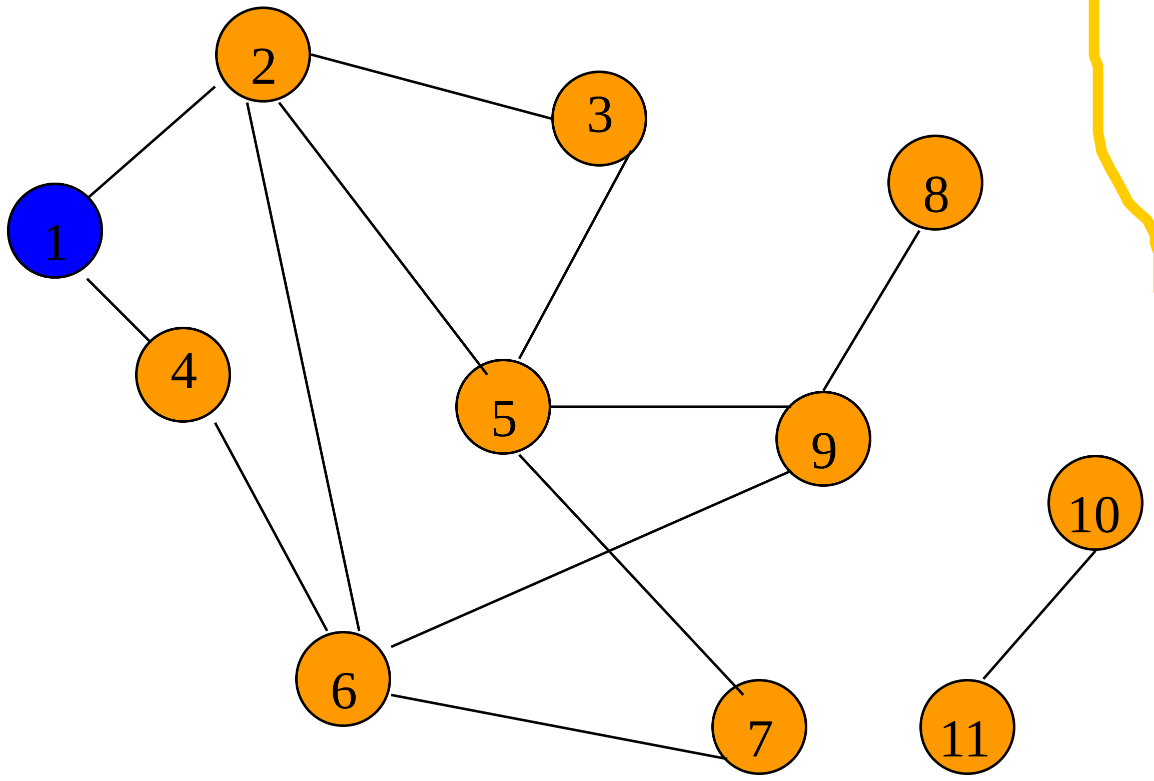
Event	Queue (Front to Rear)
Visit A	
Visit B	B
Visit C	BC
Visit D	BCD
Visit E	BCDE
Remove B	CDE
Visit F	CDEF
Remove C	DEF
Remove D	EF
Visit G	EFG
Remove E	FG
Remove F	G
Visit H	GH
Remove G	H
Visit I	HI
Remove H	I
Remove I	
Done	

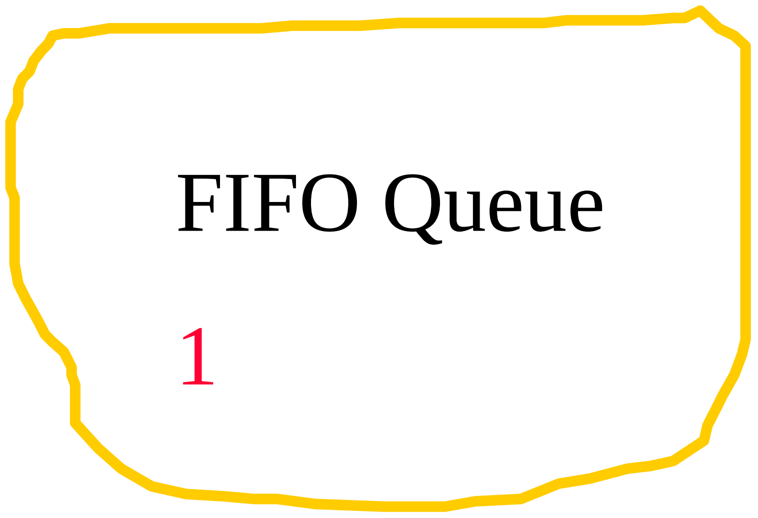
Breadth-First Search Example



Start search at vertex 1.

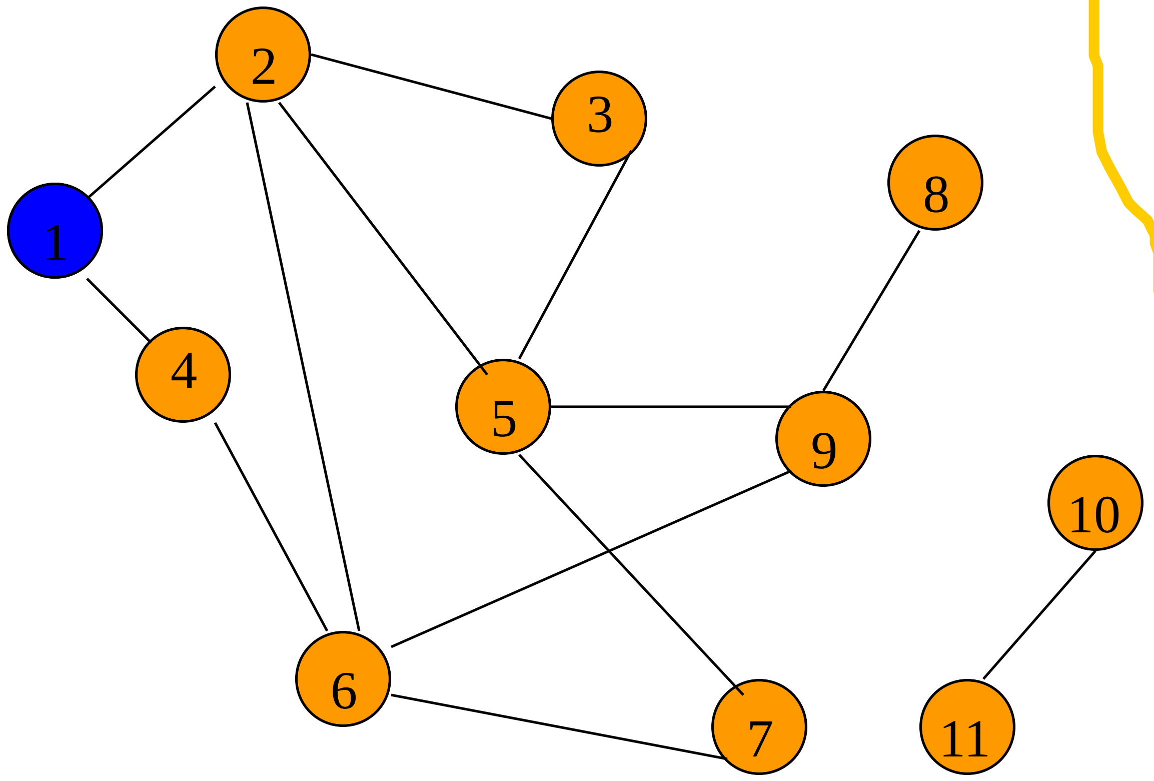
Breadth-First Search Example

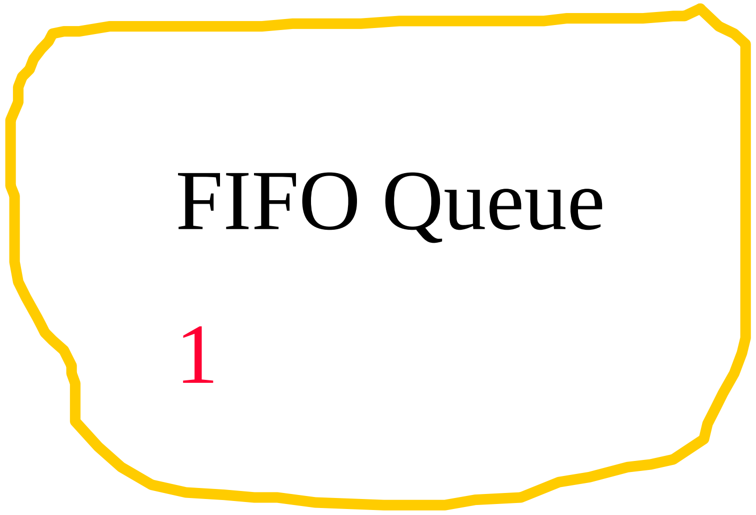




Visit/mark/label start vertex and put in a FIFO queue.

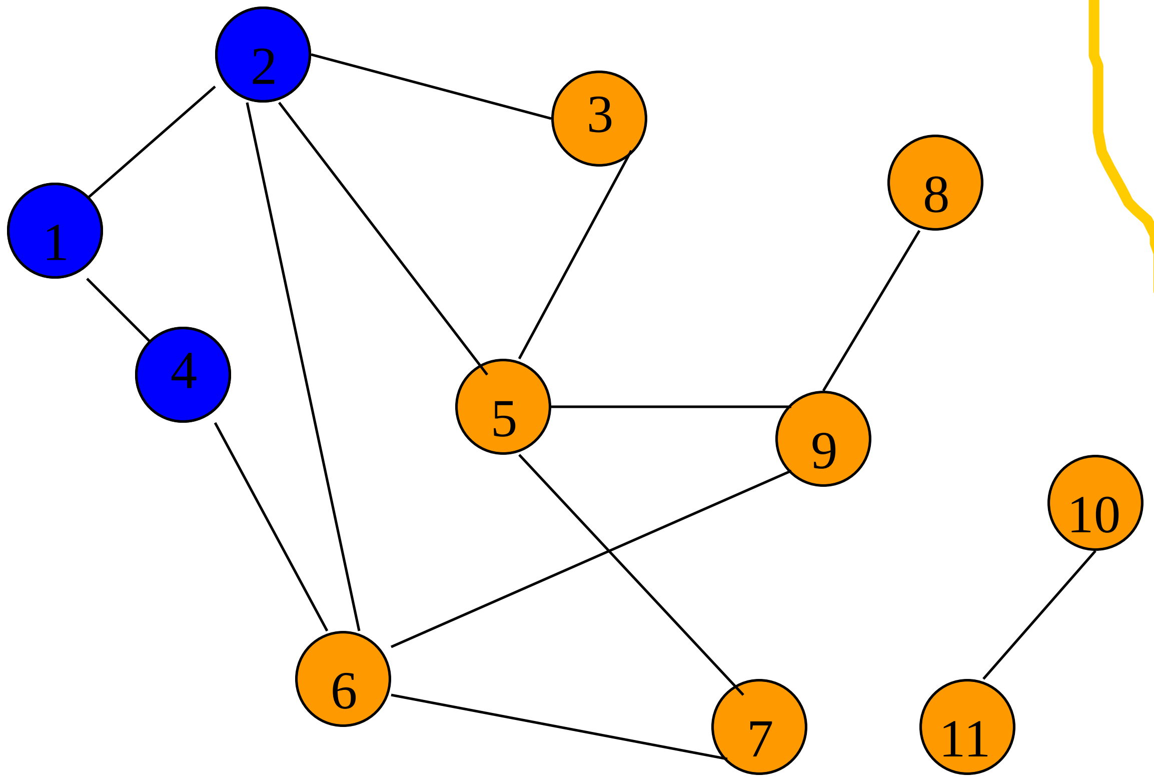
Breadth-First Search Example





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

Breadth-First Search Example

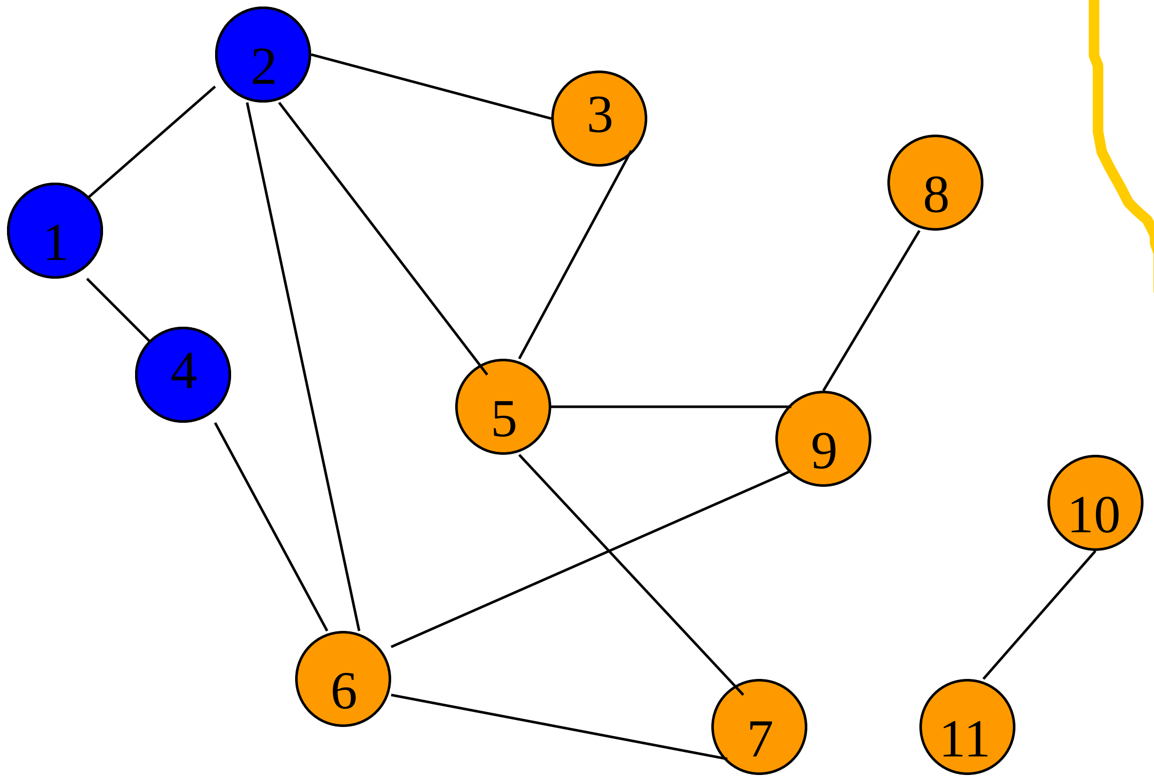


FIFO Queue

2 4

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

Breadth-First Search Example

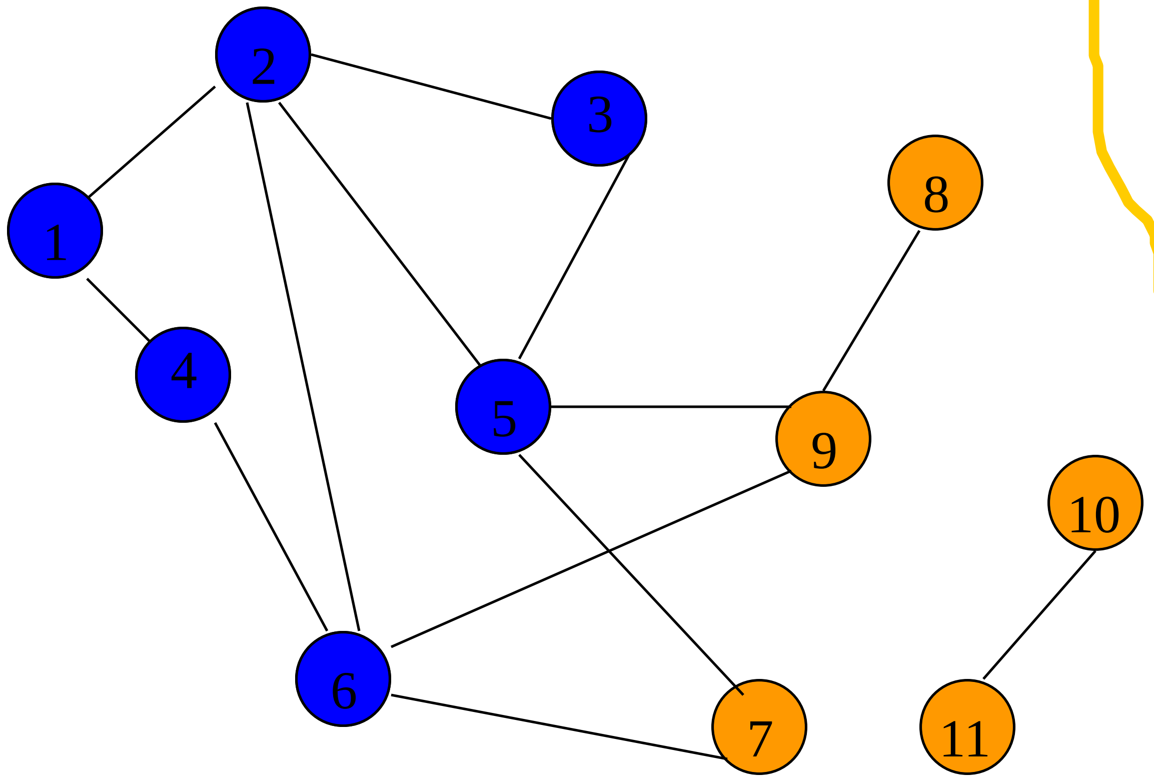


FIFO Queue

2 4

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

Breadth-First Search Example

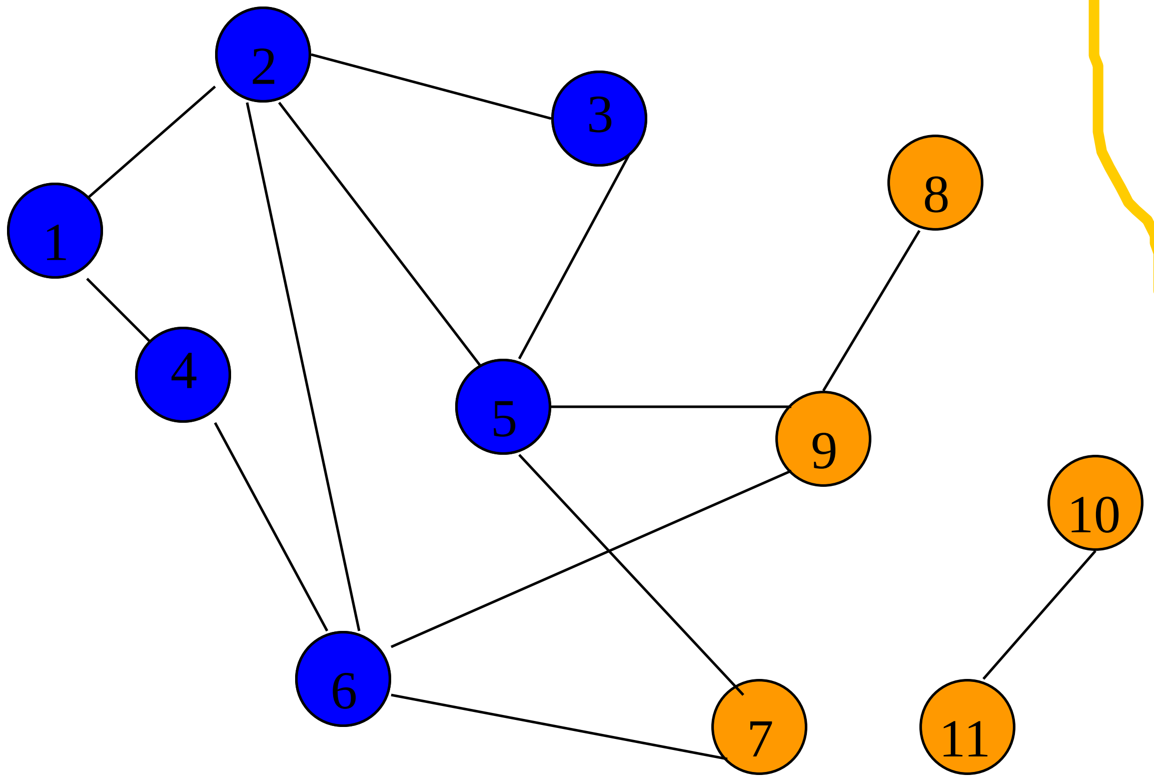


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

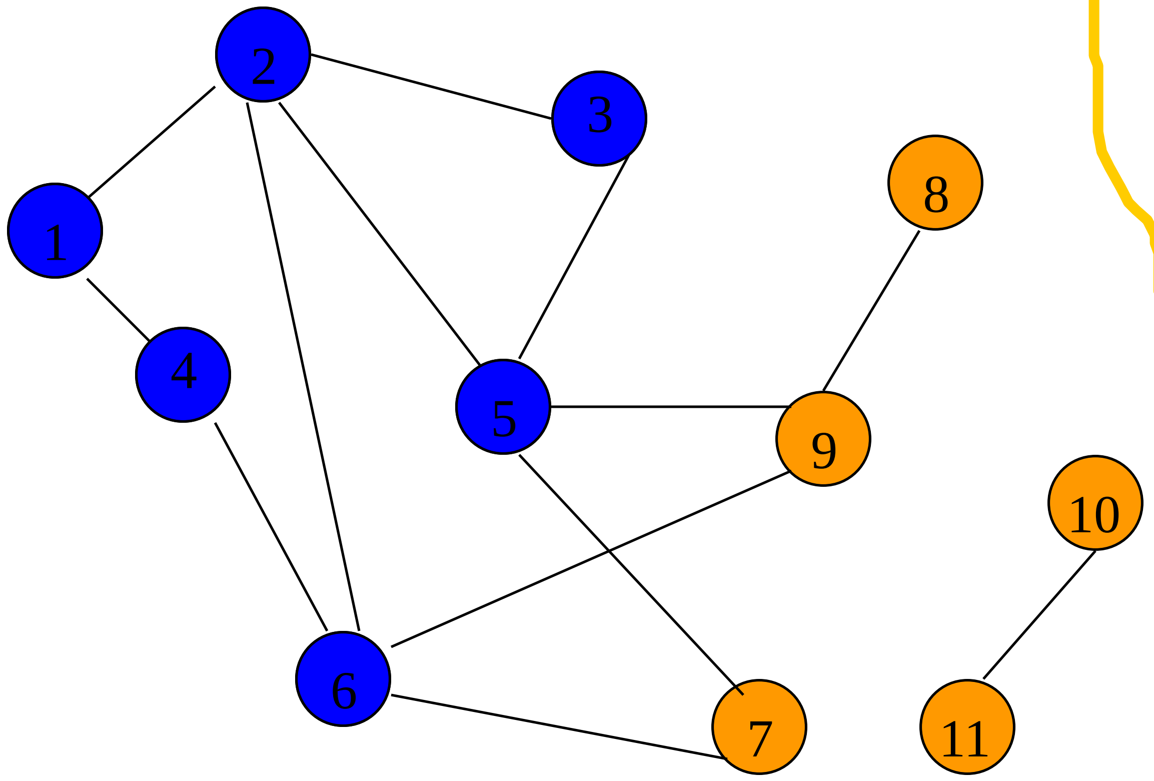


FIFO Queue

4 5 3 6

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

Breadth-First Search Example

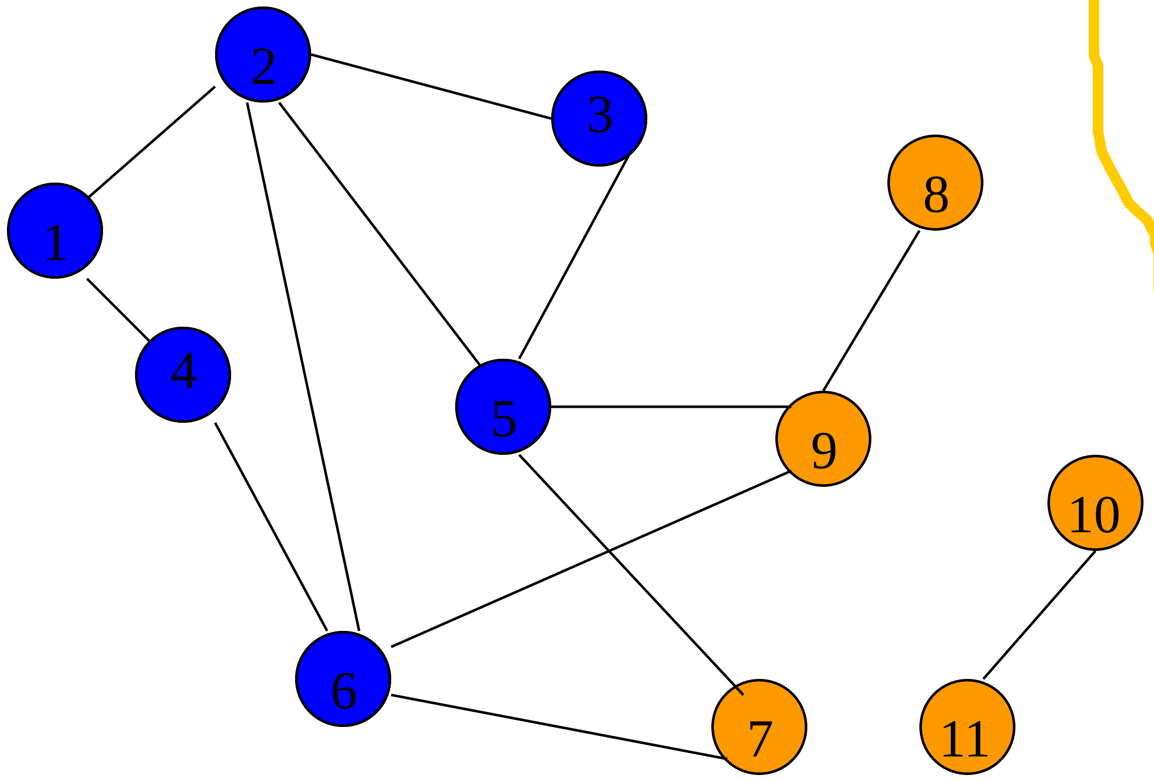


FIFO Queue

5 3 6

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

Breadth-First Search Example

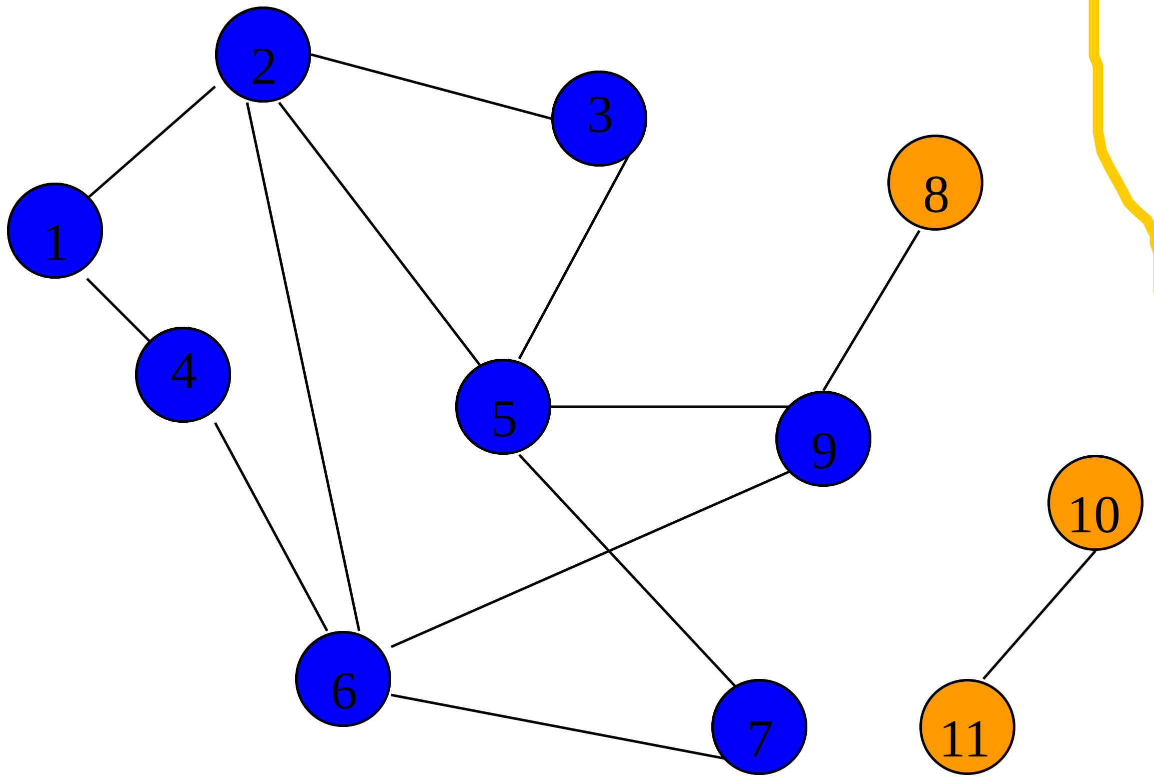


FIFO Queue

5 3 6

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

Breadth-First Search Example

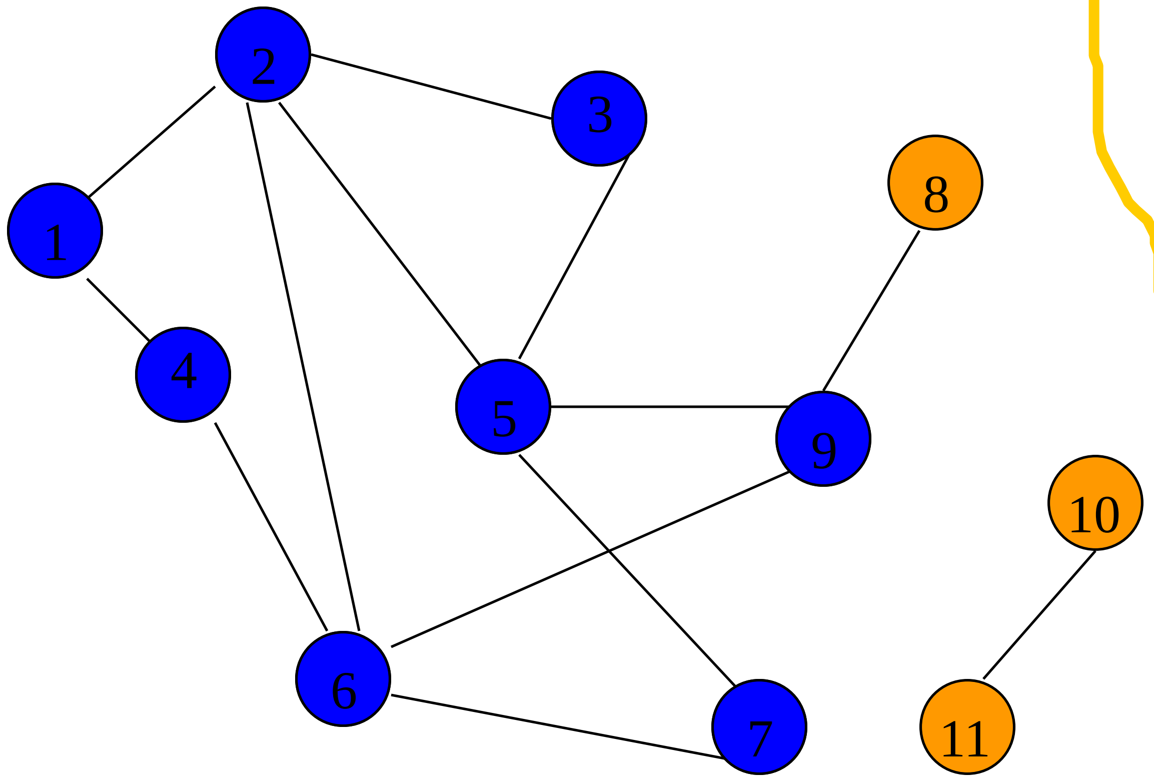


FIFO Queue

3 6 9 7

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

Breadth-First Search Example

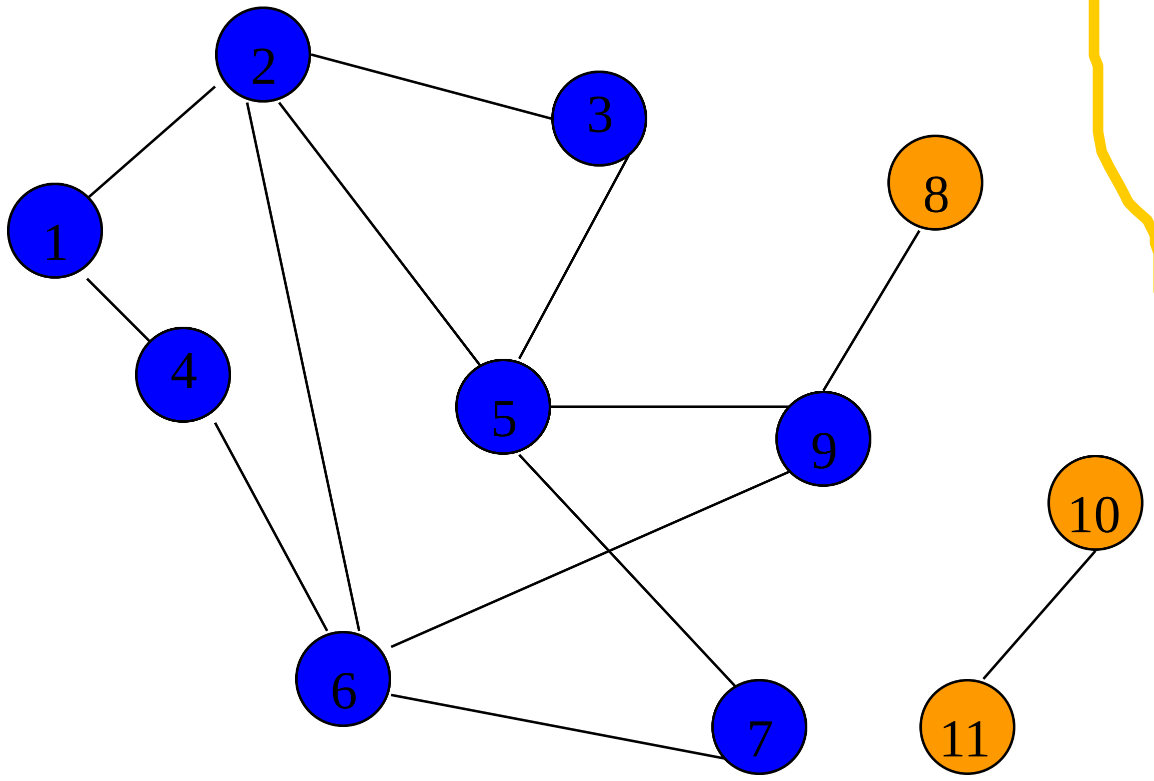


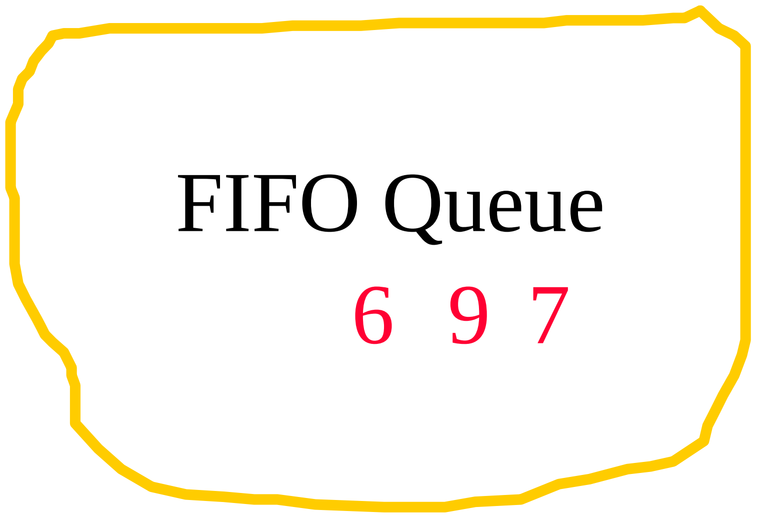
FIFO Queue

3 6 9 7

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

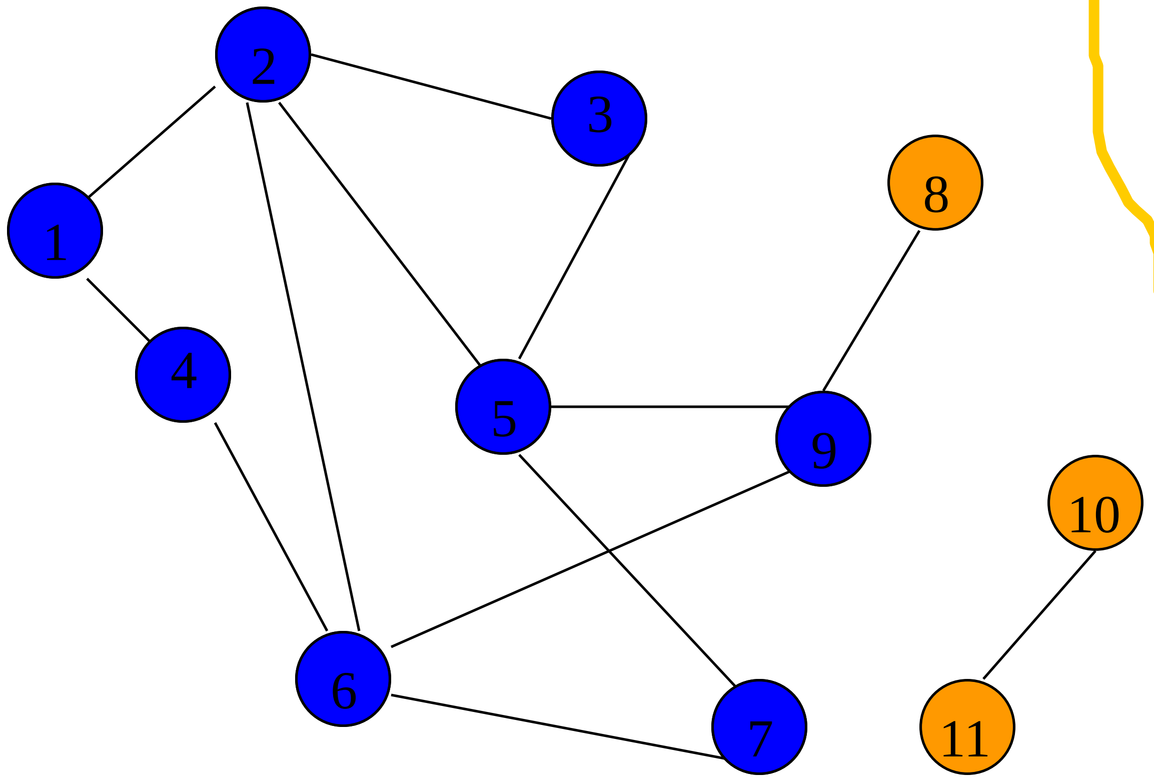
Breadth-First Search Example





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

Breadth-First Search Example

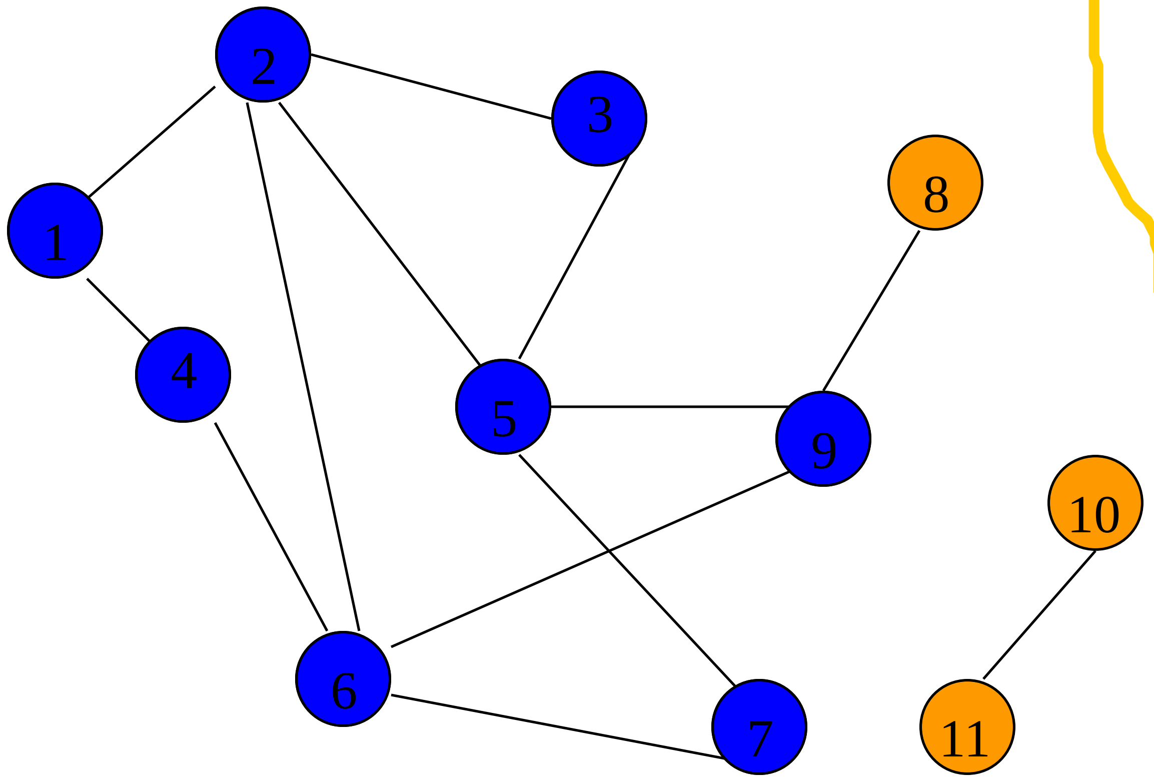


FIFO Queue

6 9 7

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

Breadth-First Search Example

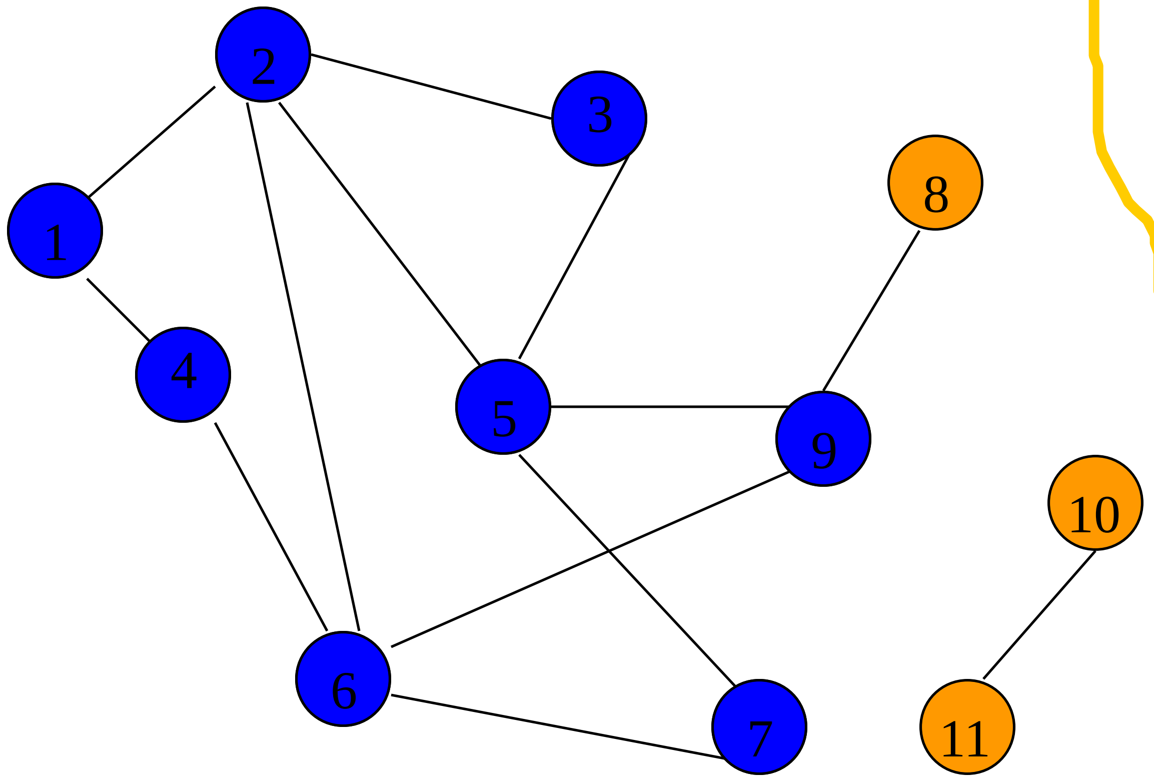


FIFO Queue

9 7

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

Breadth-First Search Example

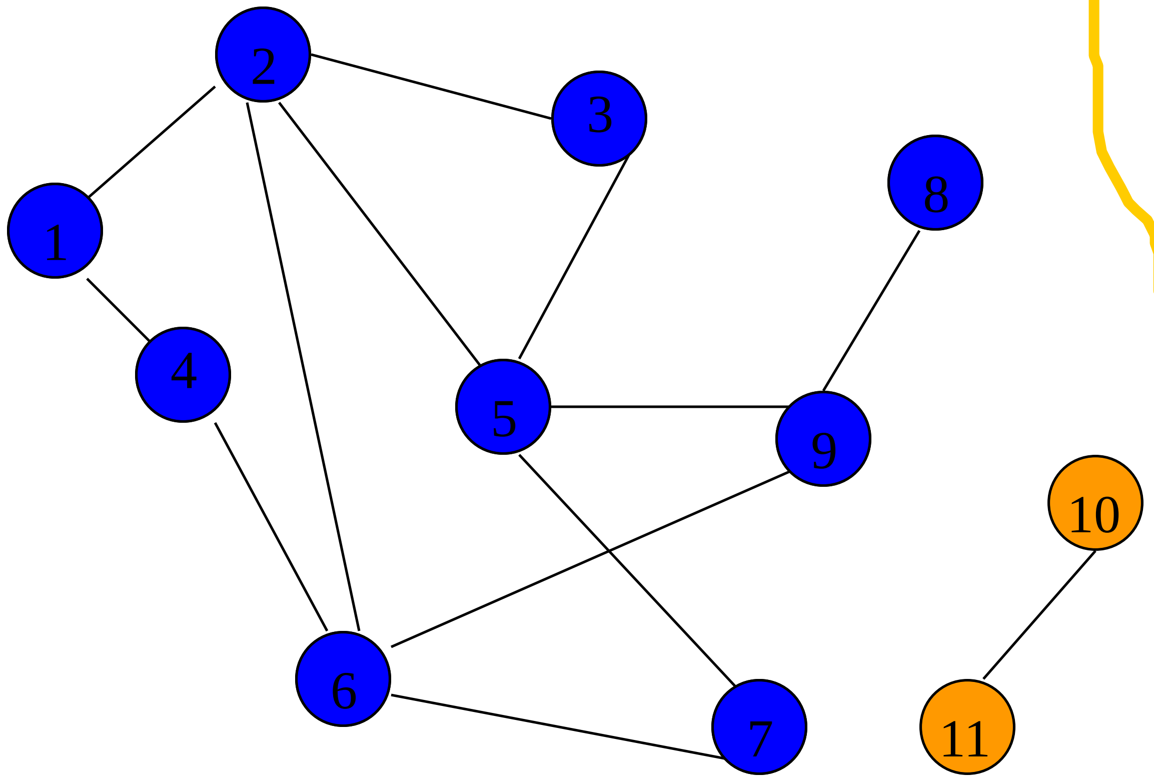


FIFO Queue

9 7

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

Breadth-First Search Example

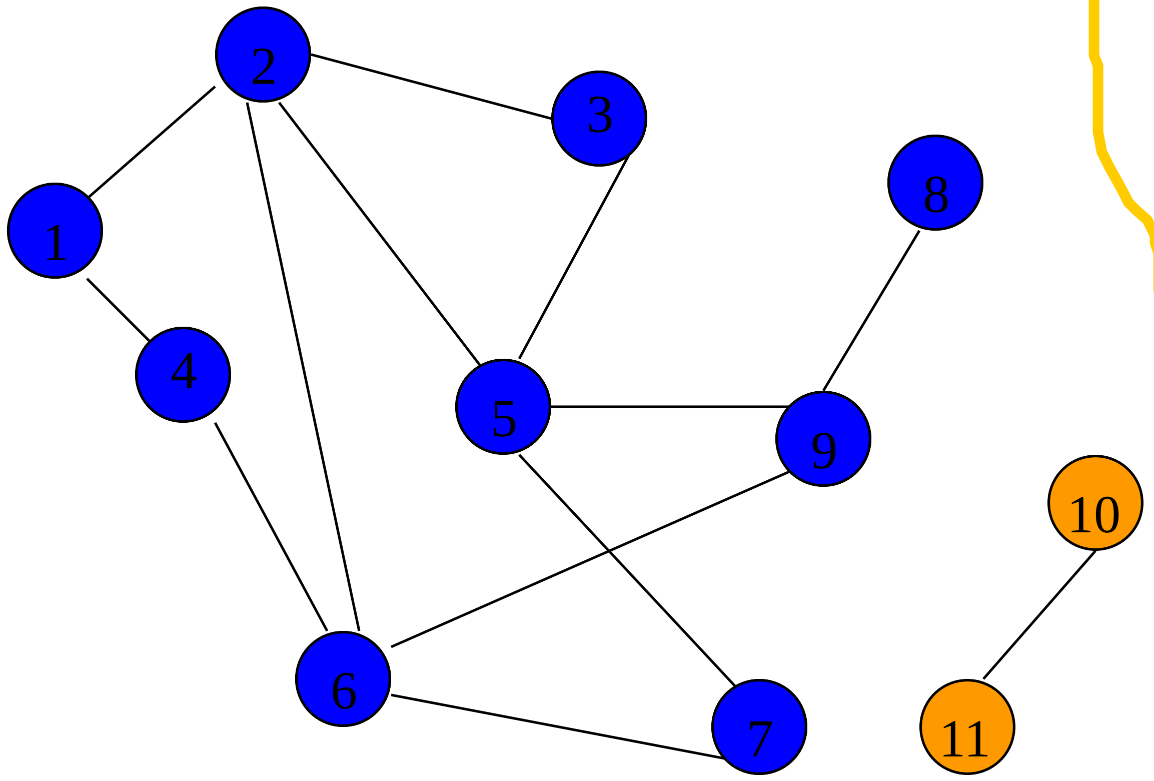


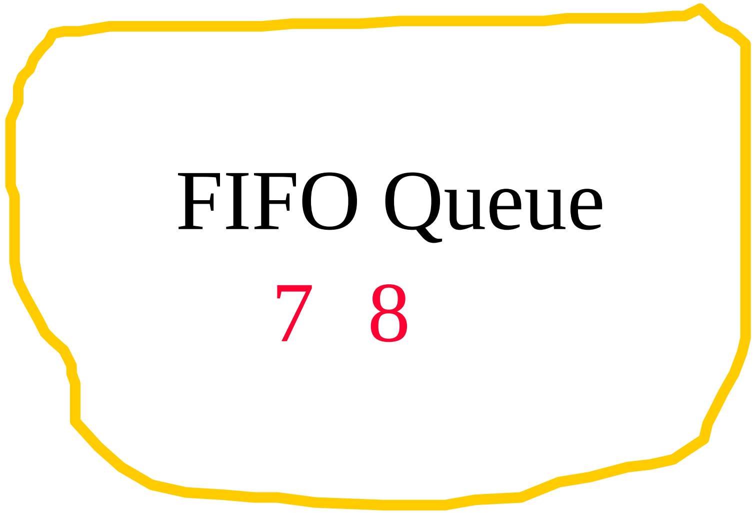
FIFO Queue

7 8

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

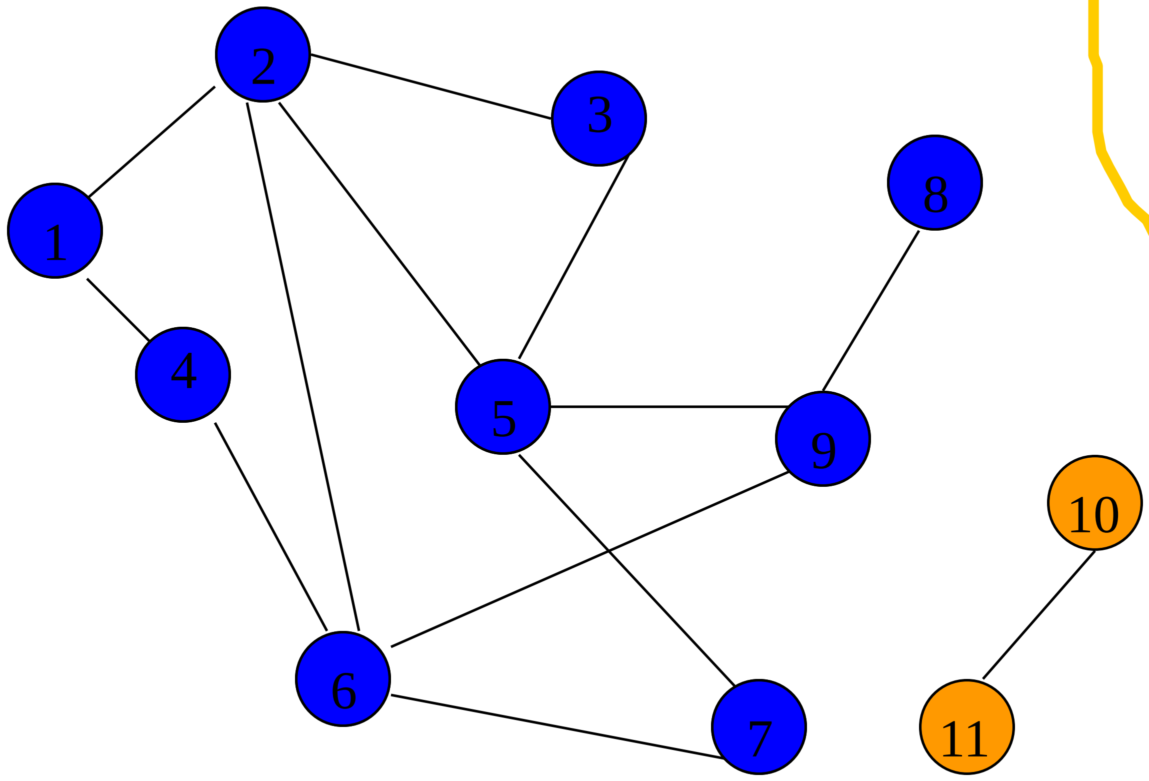
Breadth-First Search Example

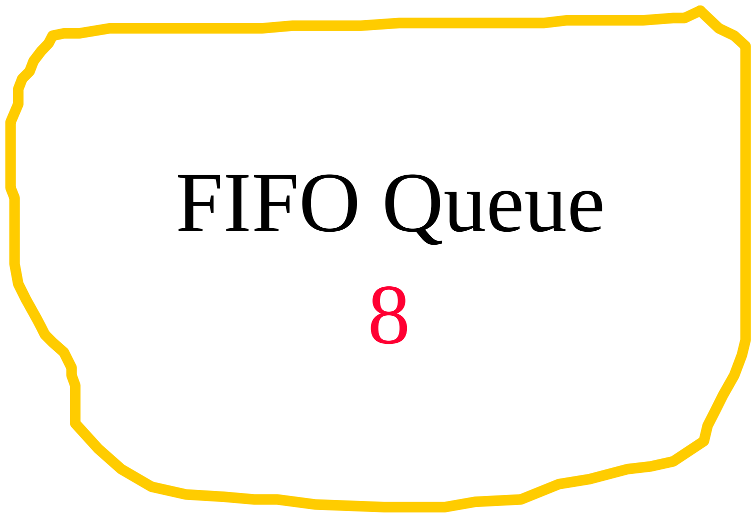




Remove 7 from Q; visit adjacent unvisited vertices; put in Q.

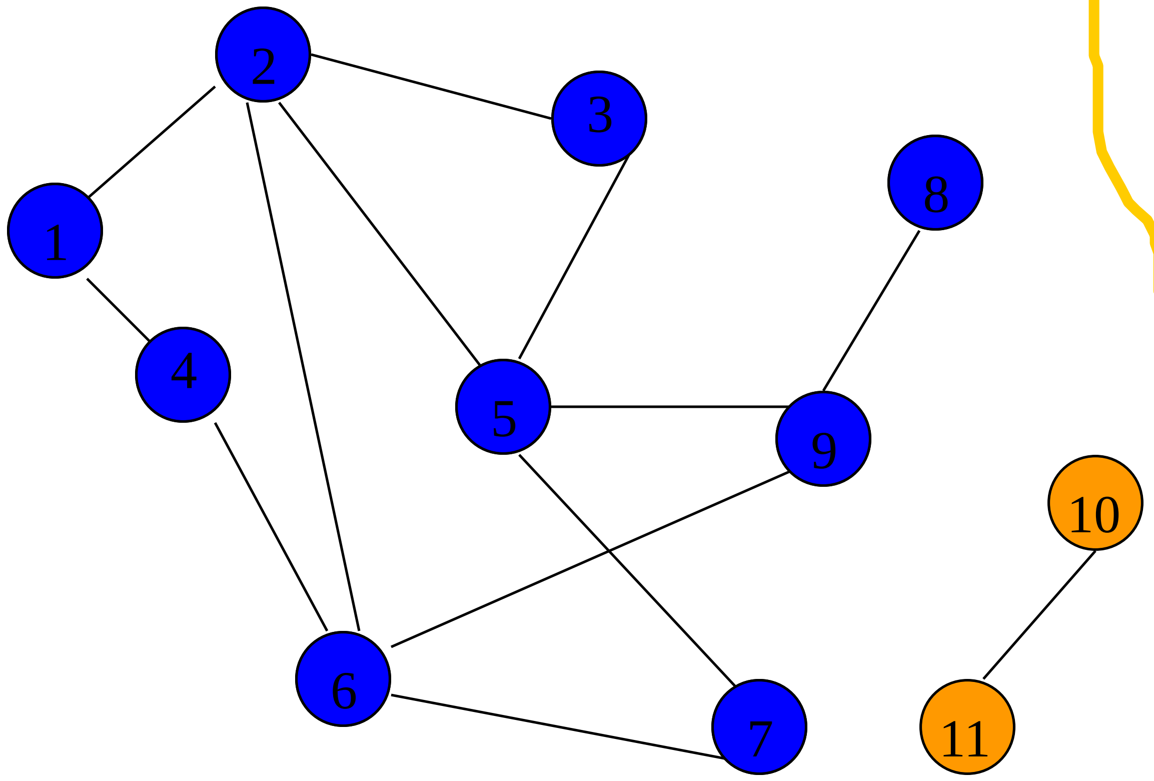
Breadth-First Search Example





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

Breadth-First Search Example

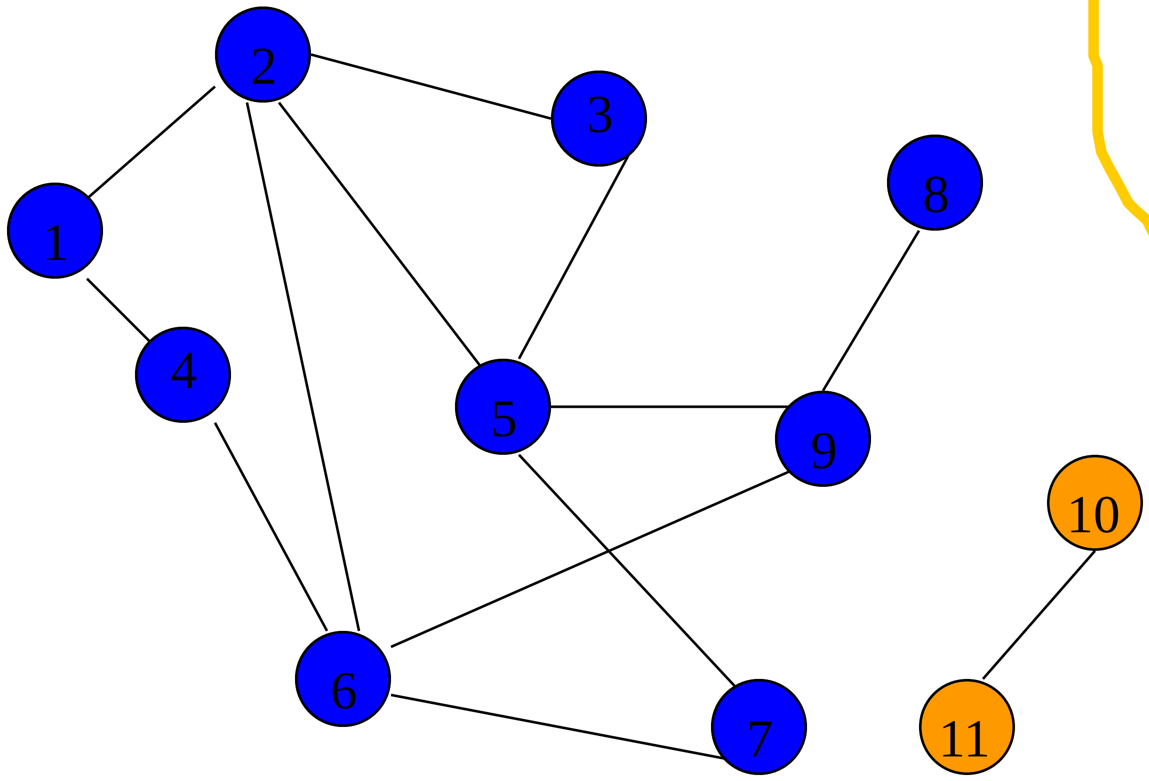


FIFO Queue

8

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

Breadth-First Search Example



Queue is empty. Search terminates.

Breadth-First Search Property

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

Depth-First Search (DFS)

- The **depth-first traversal**, or **search**, finds all graph vertices reachable from a particular starting vertex in a way that explores a given path from the starting vertex before starting another path.
- The search strategy is to probe deeper and deeper along a path, before **backtracking** hence the name "**depth-first**".
- Goes on until it visits a node with no successors or a node with all successors already visited.
- A **depth-first search** algorithm requires that each vertex is visited (and processed) exactly once.
- Since graphs can have cycles, we need a way of keeping track of the vertices that have already been visited.
- In the algorithm that follows this is achieved via the array '**visited**' which can be declared as, **bool visited[maxVertex]**, and **maxVertex** represents the maximum number of vertices in a graph.

DFS - algorithm

- The algorithm first marks vertex 1 as having been visited.

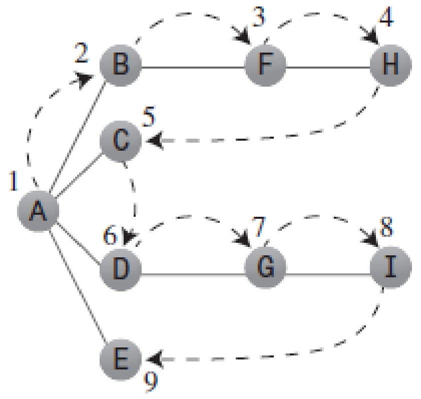
  Then if there is a vertex adjacent to vertex 1 (i.e. a destination vertex for which vertex 1 is a source vertex) that has not yet been visited, this vertex is chosen as the next starting point and the depth-first search function (dfs) is called recursively, etc.

  The recursion stops only when exploration of the graph is blocked and can go no further.

  At this point the recursion "unwinds" so alternative possibilities at higher levels can be explored.

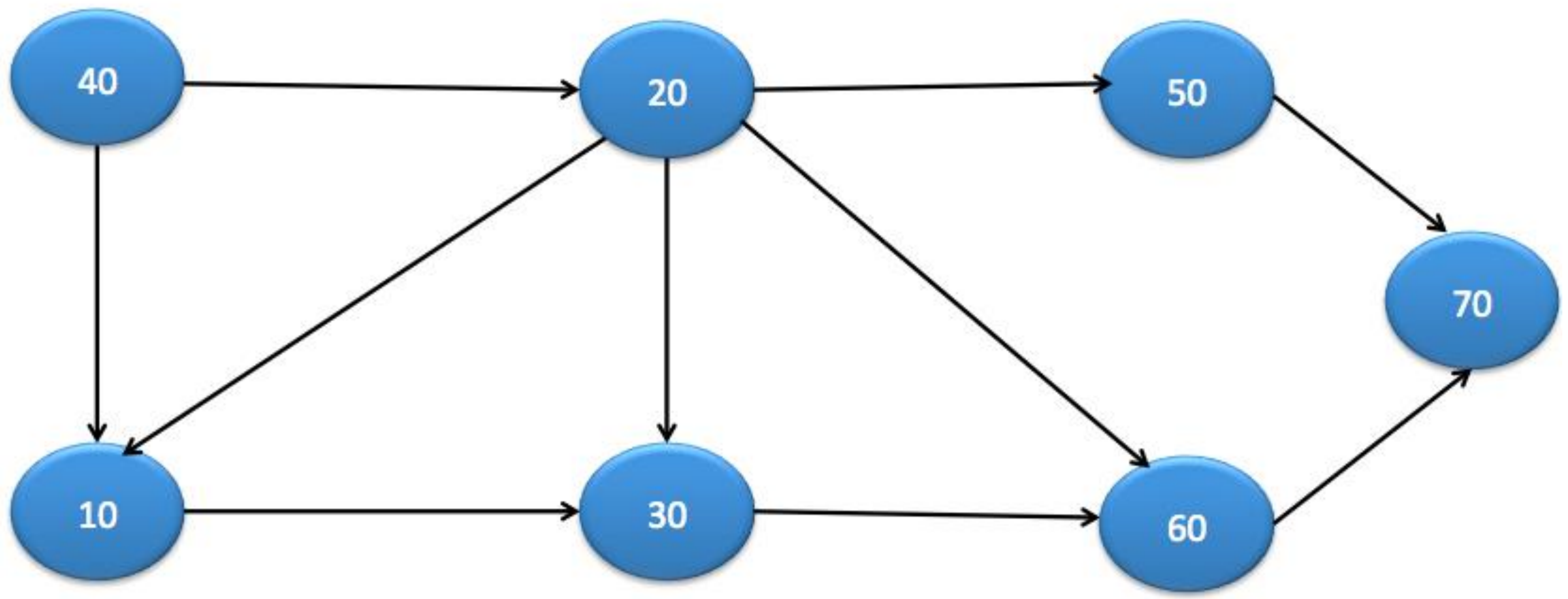
- The depth-first search algorithm is normally implemented using a **stack**.

Depth-first Search with a Stack



Depth-first search.

Event	Stack
Visit A	A
Visit B	AB
Visit F	ABF
Visit H	ABFH
Pop H	ABF
Pop F	AB
Pop B	A
Visit C	AC
Pop C	A
Visit D	AD
Visit G	ADG
Visit I	ADGI
Pop I	ADG
Pop G	AD
Pop D	A
Visit E	AE
Pop E	A
Pop A	
Done	



Depth first traversal of above graph can be :