Data Structures and Algorithms

BSc. In Computing

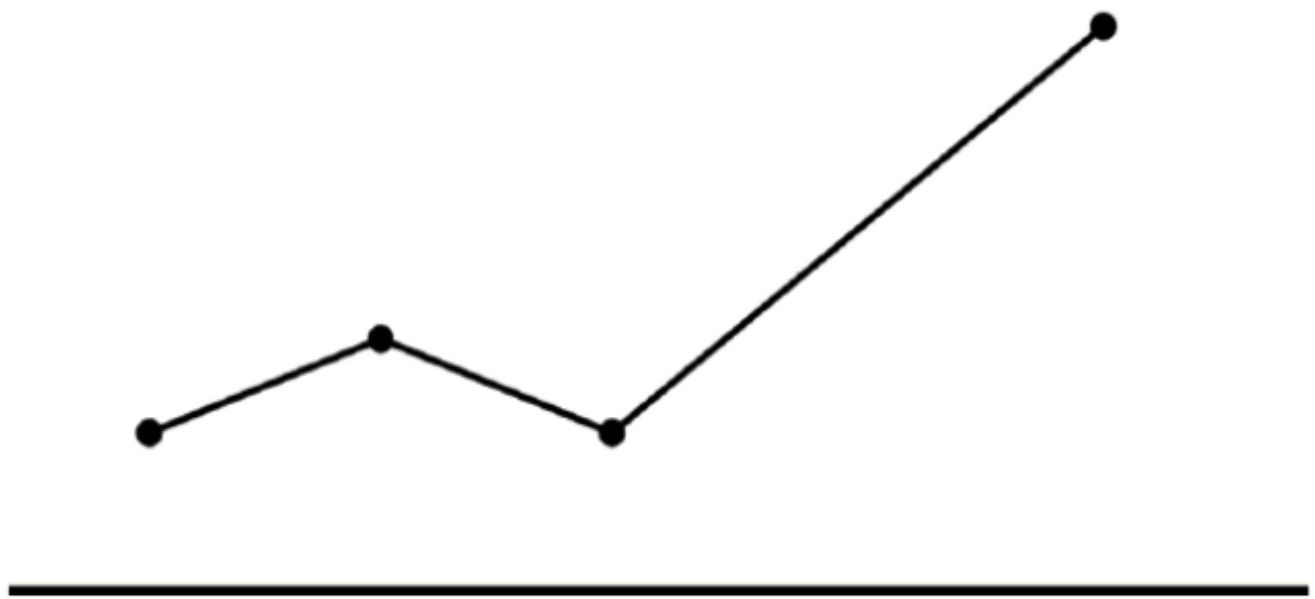
Semester 5 Lecture 6

Lecturer: Dr. Simon McLoughlin

This Week:

- Introduction to Graphs
  - Applications
  - Terminology
- Representing a Graph
  - as a Set
  - as a Diagram
- Storage of a Graph
  - Adjacency Matrix
  - Adjacency List
- Operations on a Graph considered as an ADT
- Searching a Graph
- Depth first search
- Breadth first search
- Traversal Analysis
- Efficiency of a traversal

An ordinary line graph



AIRLINK + Express Bus Stops

Airlink Prepaid Tickets

These are available at the CIE Information Desk in the Arrivals Hall, Dublin Airport

Airlink

Price Adult £3.50 €4.44 single. £6.00 €7.62 return

Price Child £2.00 €2.54 single

valid for a single/return trip on the Airlink service

Airlink - DART

Price £3.50 €4.44 single

valid for a single journey on the Airlink and the DART Rail Network

Airlink - DART tickets are available at all DART Stations

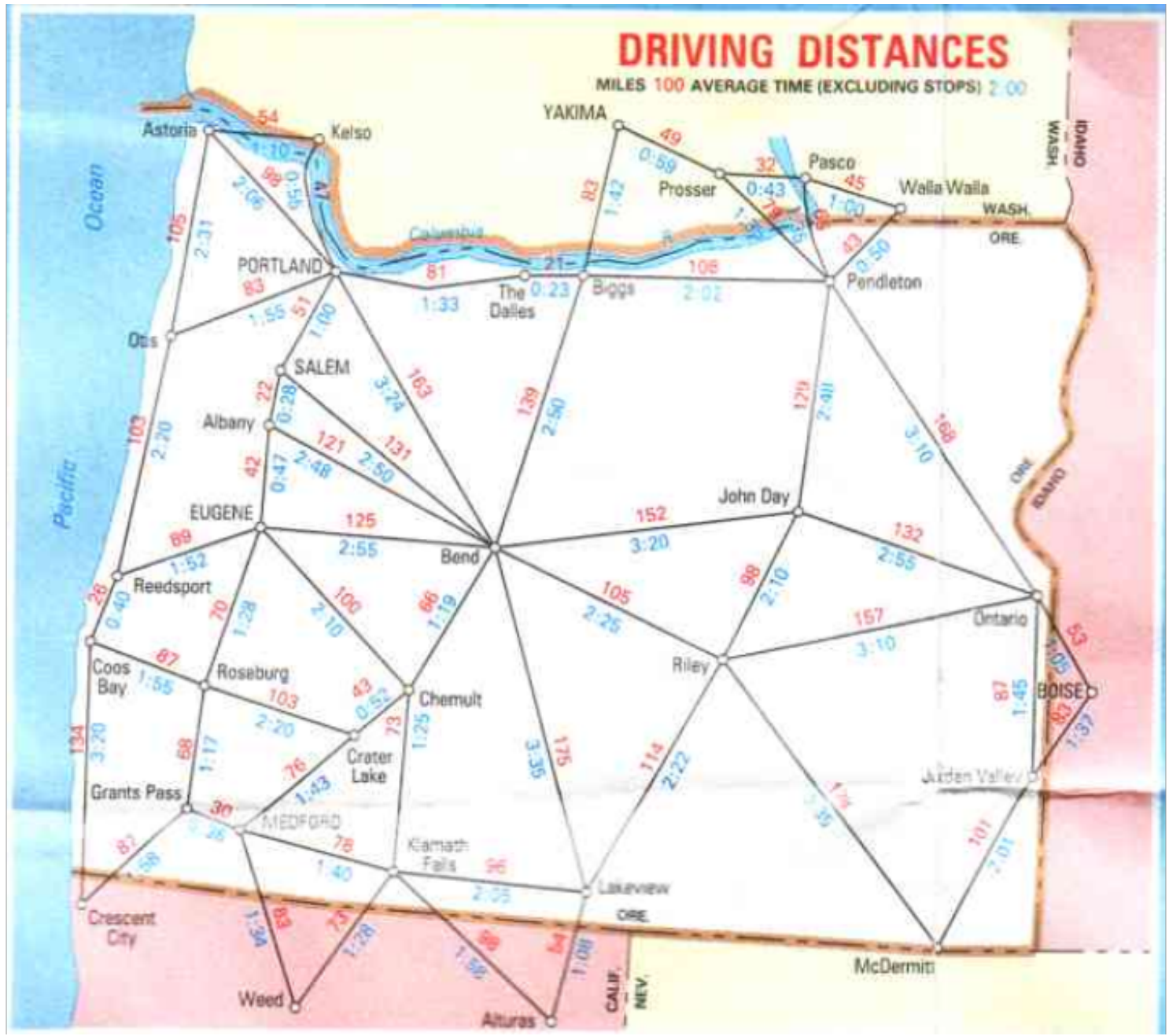
Airlink is a fully accessible service

INFORMATION BUREAU AND CUSTOMER SERVICE

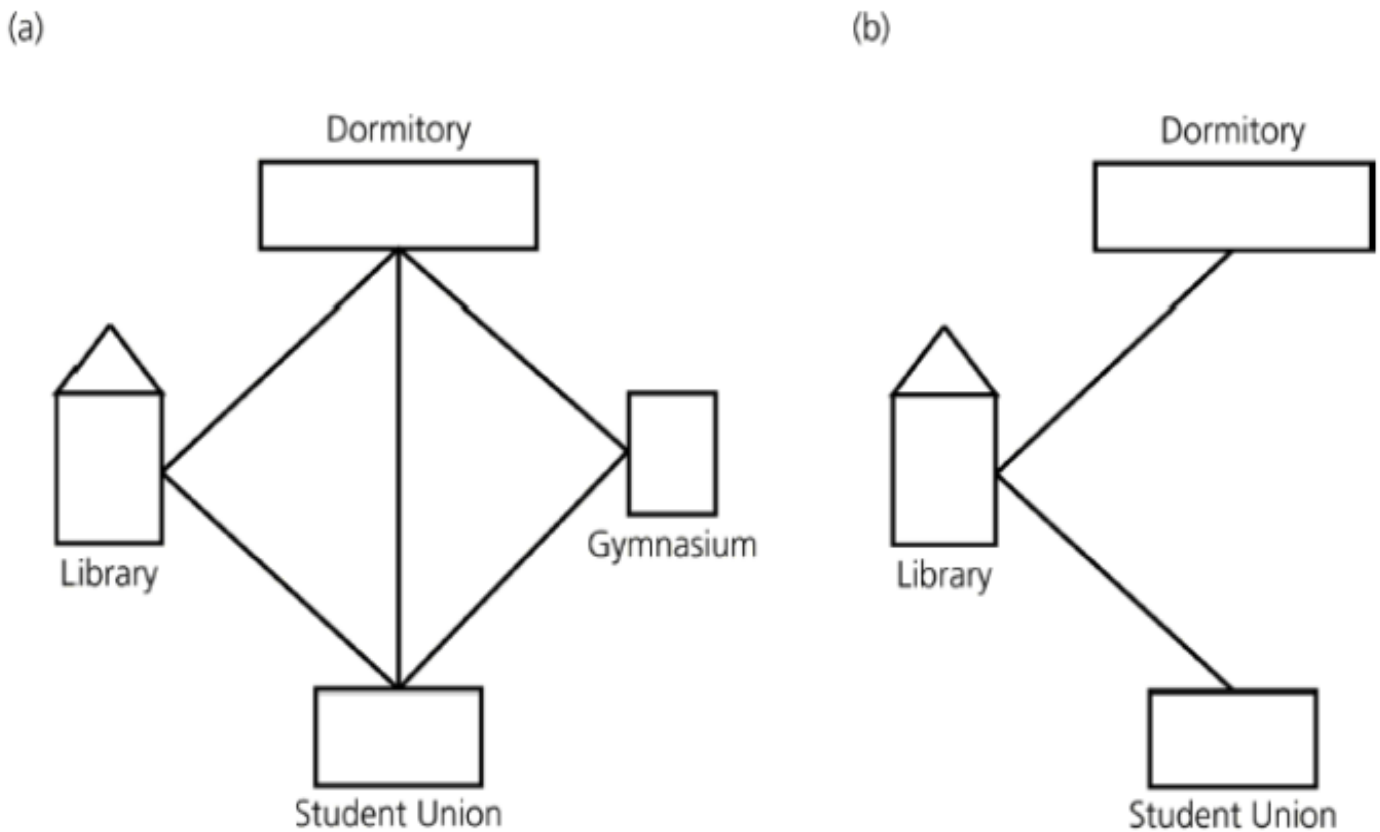
Tel. (01) 873 4222 9am to 7pm (Mon - Sat)

Dublin Bus, 59 Upr. O'Connell Street, Dublin 1.
Website: www.dublinbus.ie e-mail: info@dublinbus.ie

Buses serving the Airport


DRIVING DISTANCES
MILES 100 AVERAGE TIME (EXCLUDING STOPS) 2:00
Pacific Ocean
Astoria
Kelso
YAKIMA
Prosser
Pasco
Walla Walla
WASH.
IDAHO
ORE.
PORTLAND
Columbia R.
The Dalles
Biggs
Pendleton
Otis
SALEM
Albany
EUGENE
John Day
Bend
Reedsport
Coos Bay
Roseburg
Chemult
Riley
Ontario
BOISE
Crater Lake
Grants Pass
MEDFORD
Klamath Falls
Lakeview
Jordan Valley
Crescent City
Weed
Alturas
McDermitt
CALIF.
NEV.

a) A campus map as a graph; b) a subgraph

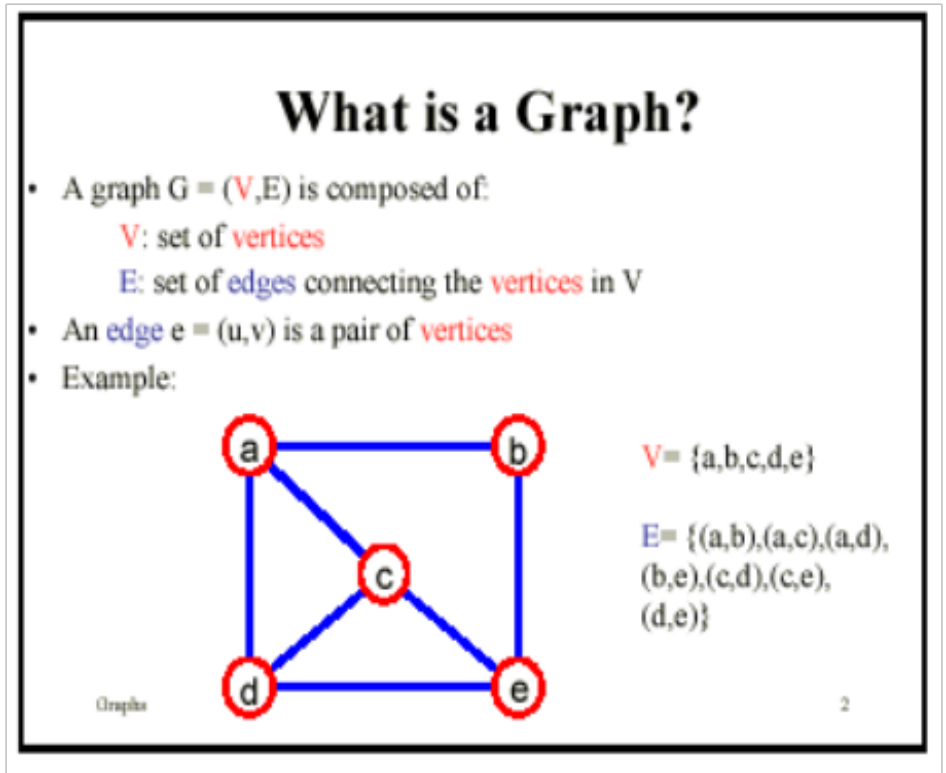


Graph Terminology and Background

A **graph** is an ordered pair, G = (V, E) of two sets representing the **nodes** or **vertices** of the graph and the**edges** of the graph.

An **edge** specifies which nodes have a connection between them.

When working with graphs, we ar e interested in how these edges are put togetherso we can move through the graph.



We will often talk about travelling an edge.

- Travelling an edge means we are changing our node of interest by following an edge connected to it.

If our graph has nodes A and B that are connected by an edge, to represent the fact that our interest has changed from A to B, we can talk about:

- Moving from A to B or
- Travelling from A to B or
- Traversing from A to B

Usually, we will simply just write the two node labels as shorthand for the edge that connects them.

AB will therefore represent the edge between node A and B.

We will say that B is adjacent to A.

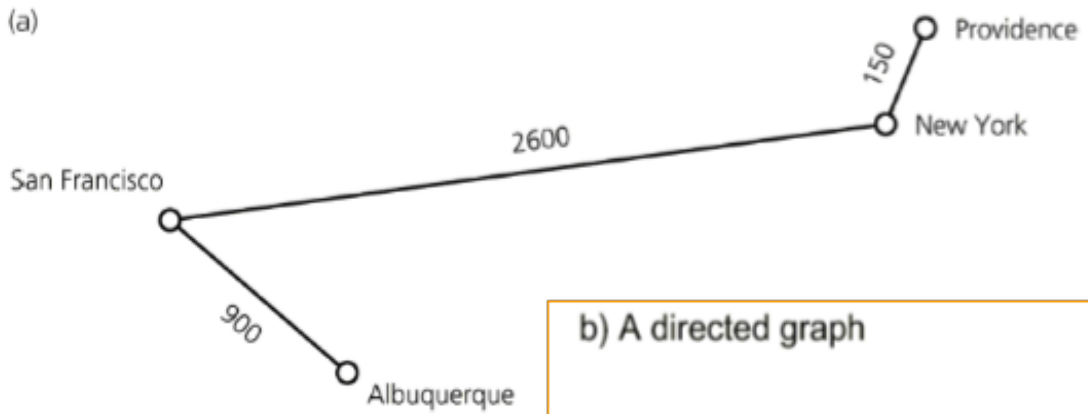
A graph can be undirected or directed.

An **undirected graph**, or simply a graph, has edges that can be traversed in either direction.

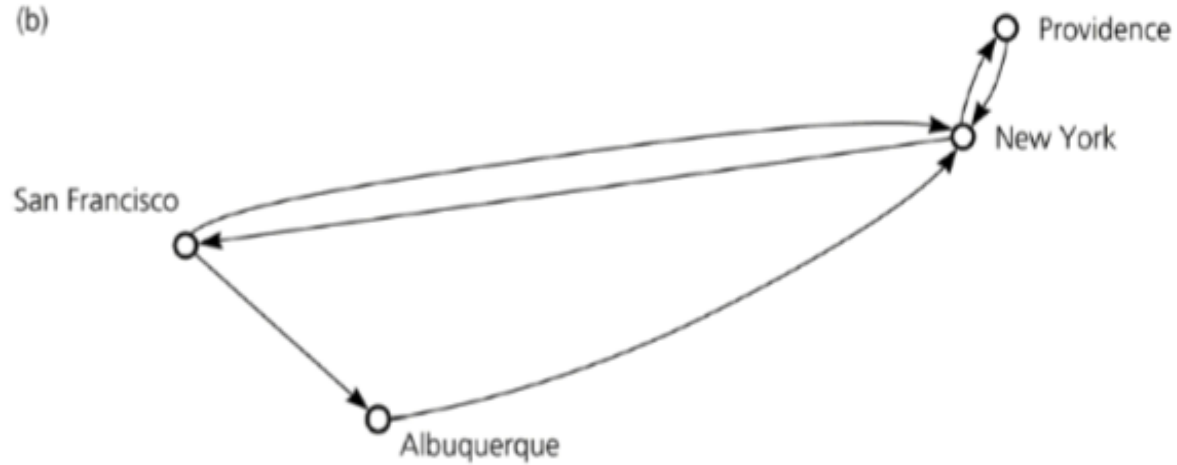
In this case, an edge contains the labels of the nodes that are at the two ends of the edge.

If an edge has just one node , that represents an edge that "loops", i.e. starts and ends at the same node.

a) A weighted graph



b) A directed graph



A **directed graph, a digraph** , has edges that can only be traversed in one direction.

- For a digraph, our set of edges will have ordered pairs in which the first item is where the edge starts and the second is where the edge ends.

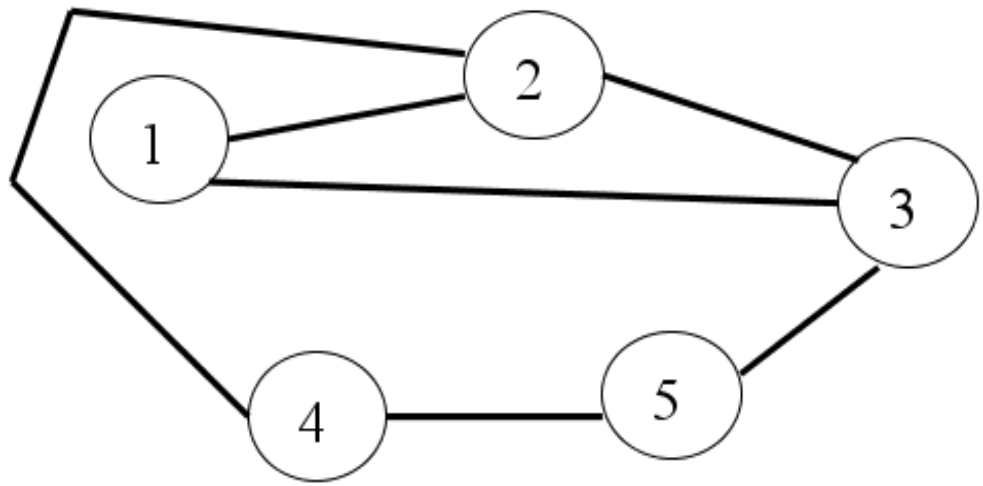
- In a digraph, an ordered p air that has the same label for both components represents an edge that starts and ends on the same node.

Therefore, …

We will usually draw graphs rather than specifying them as sets.

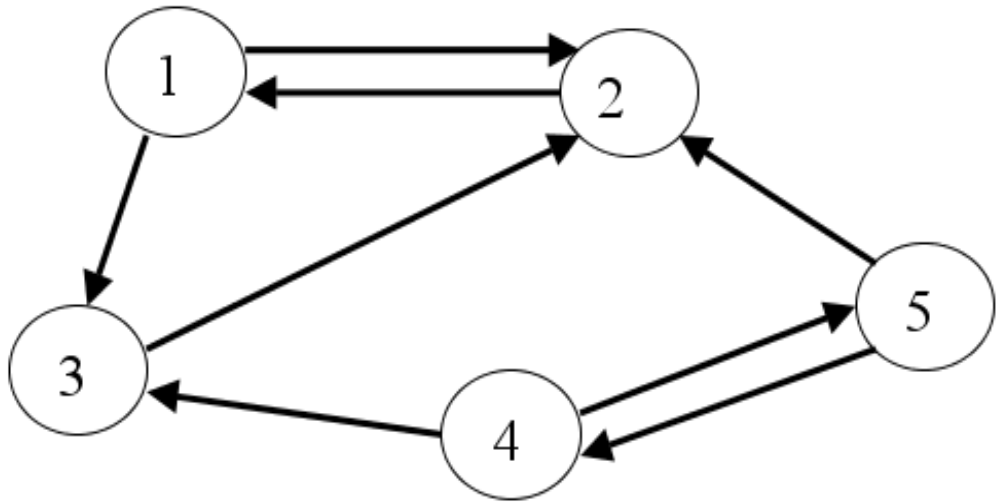
We will use circles to represent nodes and lines connecting the nodes to represent the edges.

We will put the node labels inside the circles.



The graph:

G = ({1, 2, 3, 4, 5}, { {1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5})



The digraph:

G = ({1, 2, 3, 4, 5}, { (1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5, 4)})

Terminology

A **complete graph** is a graph with an edge between every pair.

If there are N nodes, there will be $(N^2 - N)/2$ edges in a complete graph without loop edges.

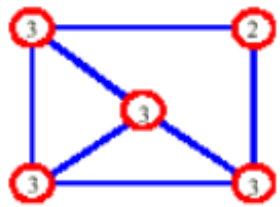
A **complete digraph** is a digraph with an edge allowing traversal between every pair of nodes.

Because the edges of a graph allow travel in two directions, whereas a digraph's edges allow travel in only one, a digraph with N nodes will have twice as many edges, specifically $N^2 - N$ edges.

Graph Terminology

- adjacent vertices: vertices connected by an edge
- degree (of a vertex): # of adjacent vertices

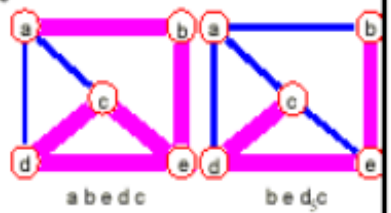
NOTE: The sum of the degrees of all vertices is twice the number of edges. ***Why?***



Since adjacent vertices each count the adjoining edge, it will be counted twice

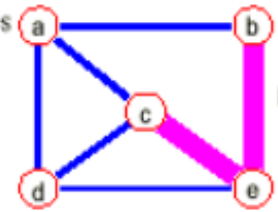
- path: sequence of vertices

$v_1, v_2, \ldots v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.

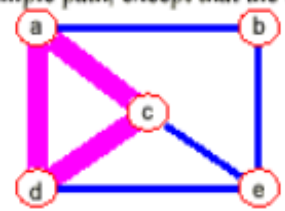


More Graph Terminology

- simple path: no repeated vertices



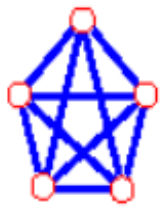
- cycle: simple path, except that the last vertex is the same as the first vertex



a c d a

Connectivity

- Let **n** = #vertices, and **m** = #edges
- **A complete graph**: one in which all pairs of vertices are adjacent
- *How many total edges in a complete graph?*
  - Each of the n vertices is incident to **n**-1 edges, however, we would have counted each edge twice!!! Therefore, intuitively, m = **n**(**n** -1)/2.
- Therefore, if a graph is not complete, m < **n**(**n** -1)/2



n = 5
m = (5 * 4)/2 = 10

More Connectivity

n = #vertices
m = #edges

- For a tree **m** = **n** - 1

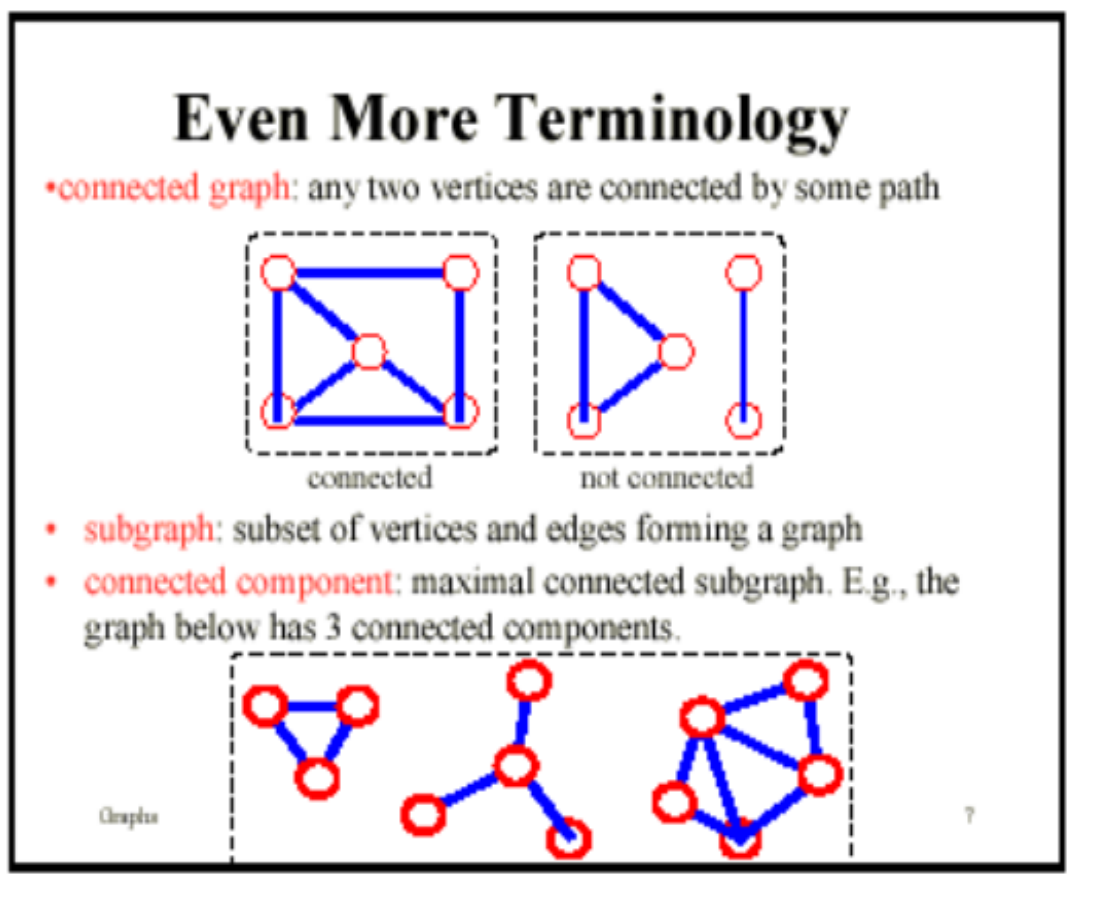
n = 5
m = 4

If **m** < **n** - 1, G is not connected

n = 5
m = 3

A **subgraph** (V_s, E_s) of a graph or digraph (V, E) is one that has a subset of the vertices ($V_s \subseteq V$) and edges ($E_s \subseteq E$) of the full graph.

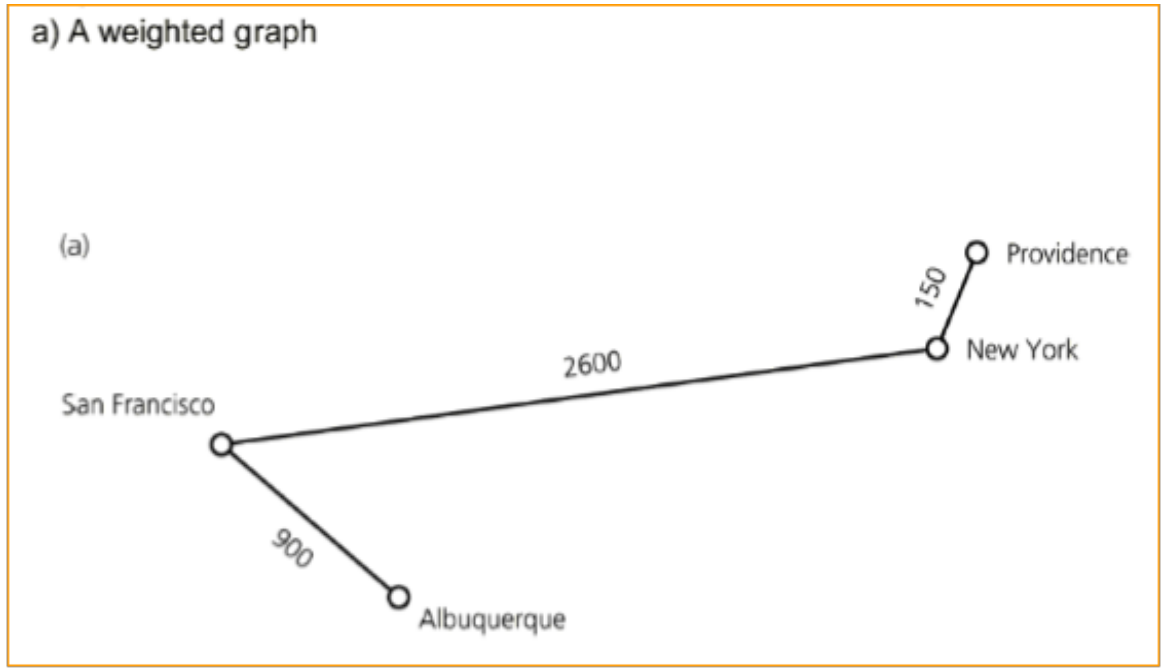
A **path** between two nodes of a graph or digraph is a sequence of edges that can be travelled consecutively.



A **weighted graph** is one where each edge has a value, called a **weight**, associated with it.

In graph drawings, the weight will be written near the edge.

In formal definitions the weight will be an extra component in the set of an edge or the ordered "pair" – more correctly, now an **ordered triplet**.

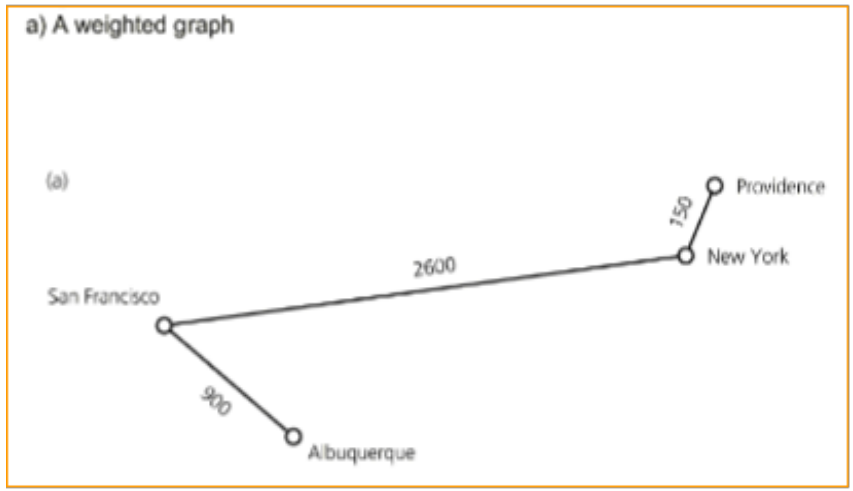


When working with weighted graphs, we consider the **weight as the "cost" of traversing the edge**

A path through a weighted graph has a cost that is the sum of the weights of an edge in the path.

In a weighted graph, the shortest path between two nodes is the path with the smallest cost, even if it does not have the fewest edges!!

If path P1 has five edges with a total cost of 24, and
path P2 has three edges with a total cost of 36,
then path P1 will be considered the shortest path
because its cost is less.



A **graph or digraph is called connected** if there is at least one path between every pair of nodes.

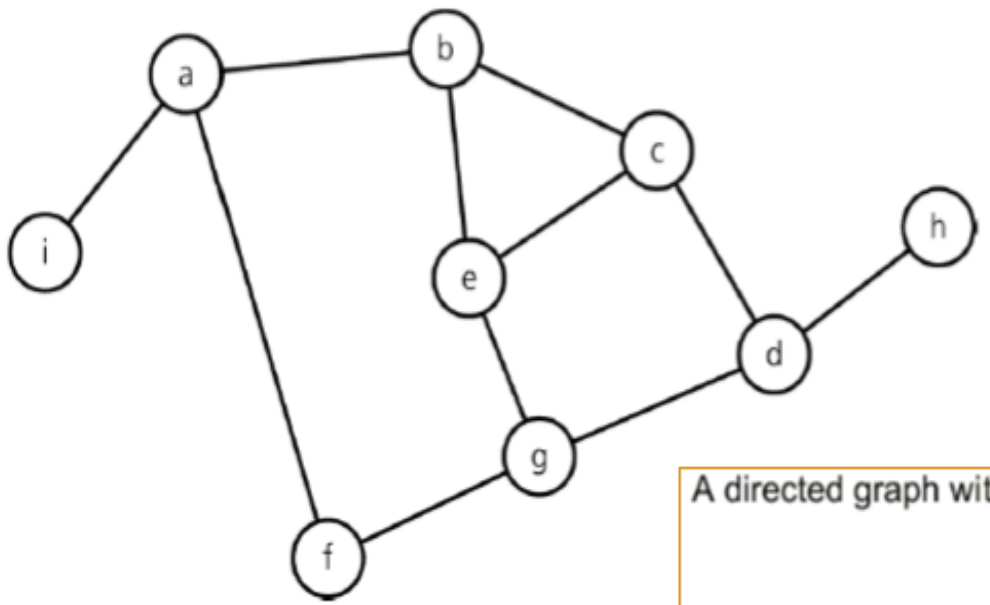
A **cycle** is a path that begins and ends at the same node.

An **acyclic graph or digraph** is one that has no cycles.

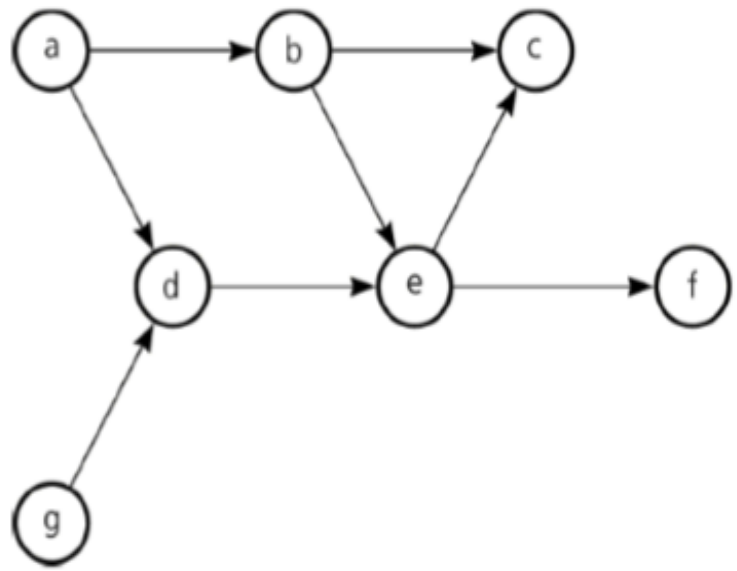
A graph that is connected and acyclic is called an **unrooted tree**.

An unrooted tree has the structure of a tree except that no node has been specified as the root … but every node could serve as the root.

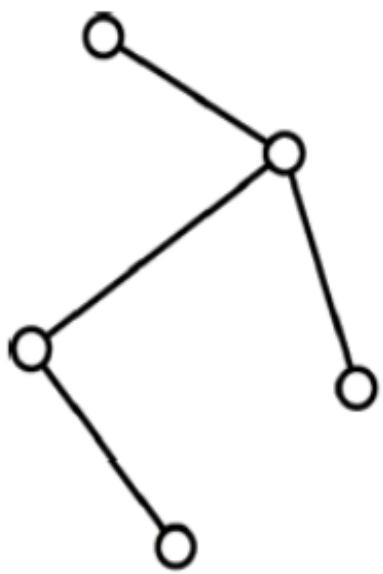
A connected graph with cycles



A directed graph without cycles

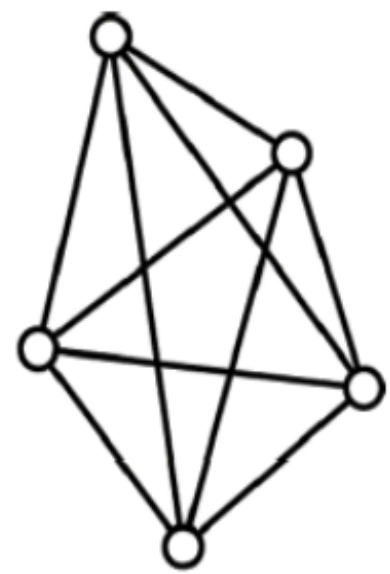


Graphs that are a) connected; b) disconnected; and c) complete



(a)

(b)



(c)

Data Structures and Methods for Graphs

There are two ways that we can store the graph or digraph information:

- An adjacency matrix
- An adjacency list.

An **adjacency matrix** gives us the ability to quickly access edge information, but …

if the graph is far from being a complete graph,
then there will be many more empty elements in the array than there are full elements.

An **adjacency list** uses space that is proportional to the number of edges in the graph, but the time to access edge information may be greater.

The Adjacency Matrix

An adjacency matrix for a graph $G = (V, E)$, with $|V| = N$, will be stored as a two dimensional array of size $N \times N$.

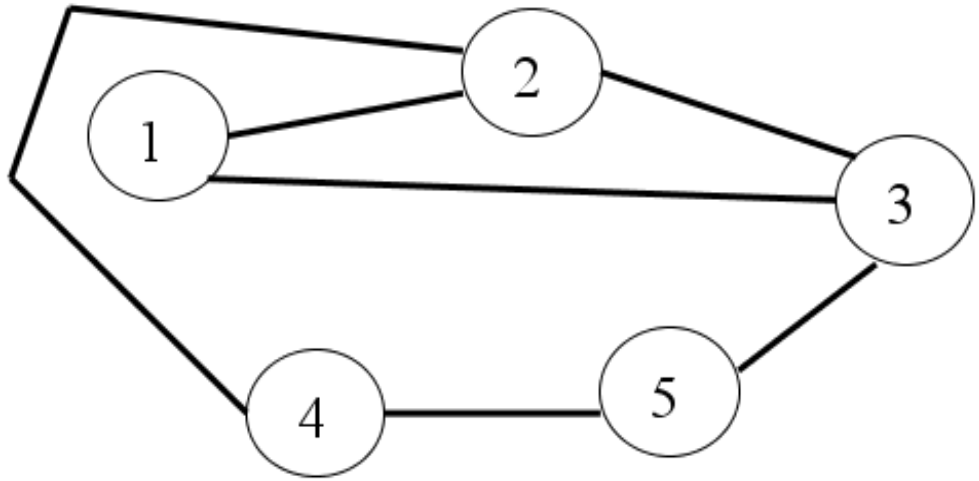
Each location $[j, k]$ of this array will store a 0 (zero), except if there is an edge from node $\mathbf{v_j}$ to node $\mathbf{v_k}$, the location will store a 1.

Formally:

$$\text{AdjacencyMatrix}[j, k] = \begin{cases} 1 & \text{if } v_j\, v_k \in E \\ 0 & \text{if } v_j\, v_k \notin E \end{cases}$$

for all j and k in the range 1 to N

The adjacency matrix for the graph below is given next.



The graph G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 5}, {4, 5}})

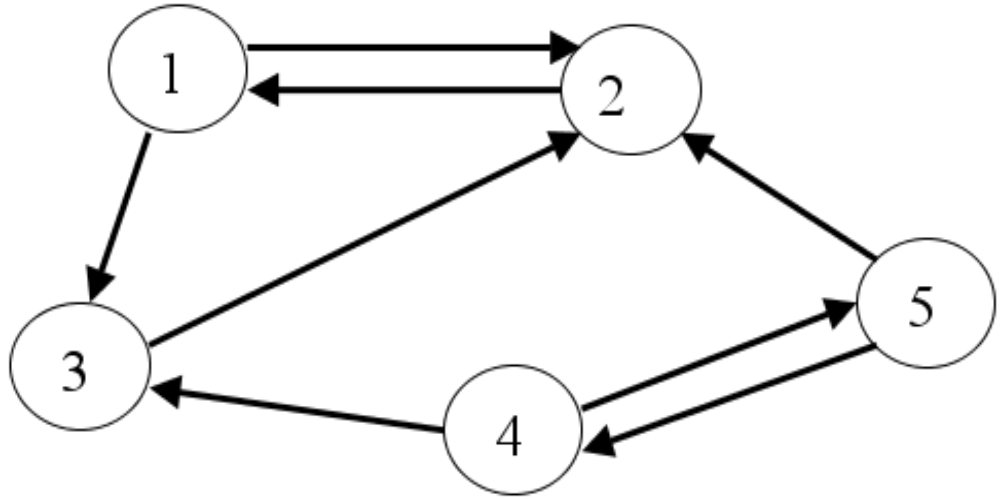
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	1
5	0	0	1	1	0

The adjacency matrix for the graph

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

The adjacency matrix for the graph

The adjacency matrix for the digraph below is given next.



The digraph:

G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5, 4)})

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

The adjacency matrix for the digraph

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

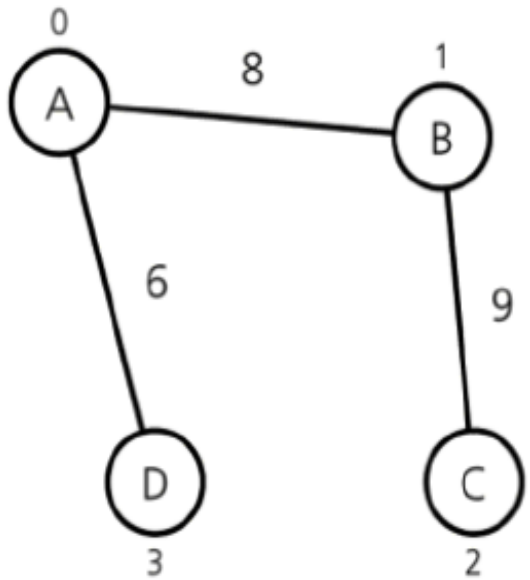
The adjacency matrix for the digraph

For weighed graphs and digraphs, the adjacency matrix entries would be ∞ (infinity) if there is no edge and the weight for the edge in all other cases.

We can if we wish set the diagonal elements to be 0, because there is no cost to travel from a node to itself.

a) A weighted undirected graph and b) its adjacency matrix

(a)



(b)

		0 A	1 B	2 C	3 D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

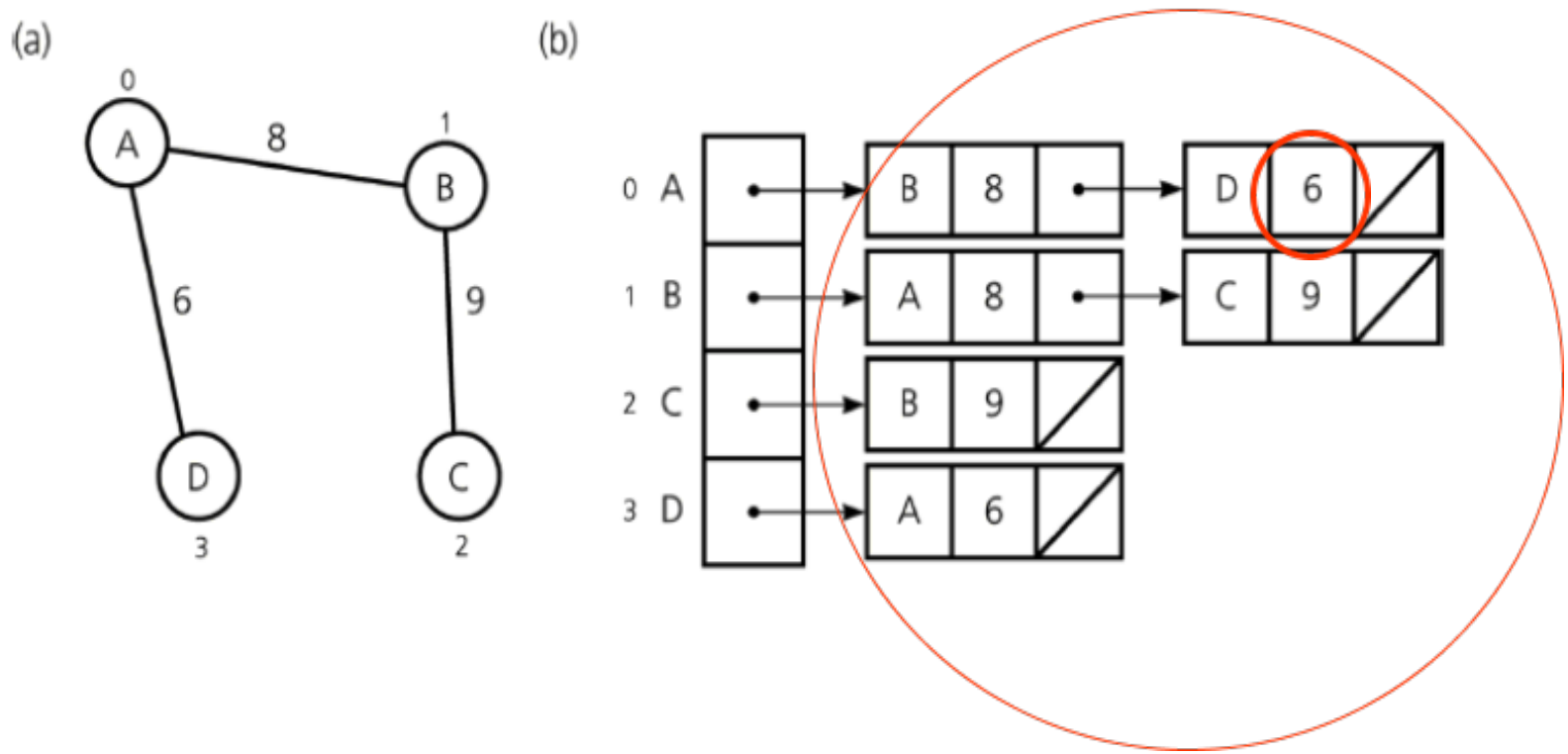
The Adjacency List

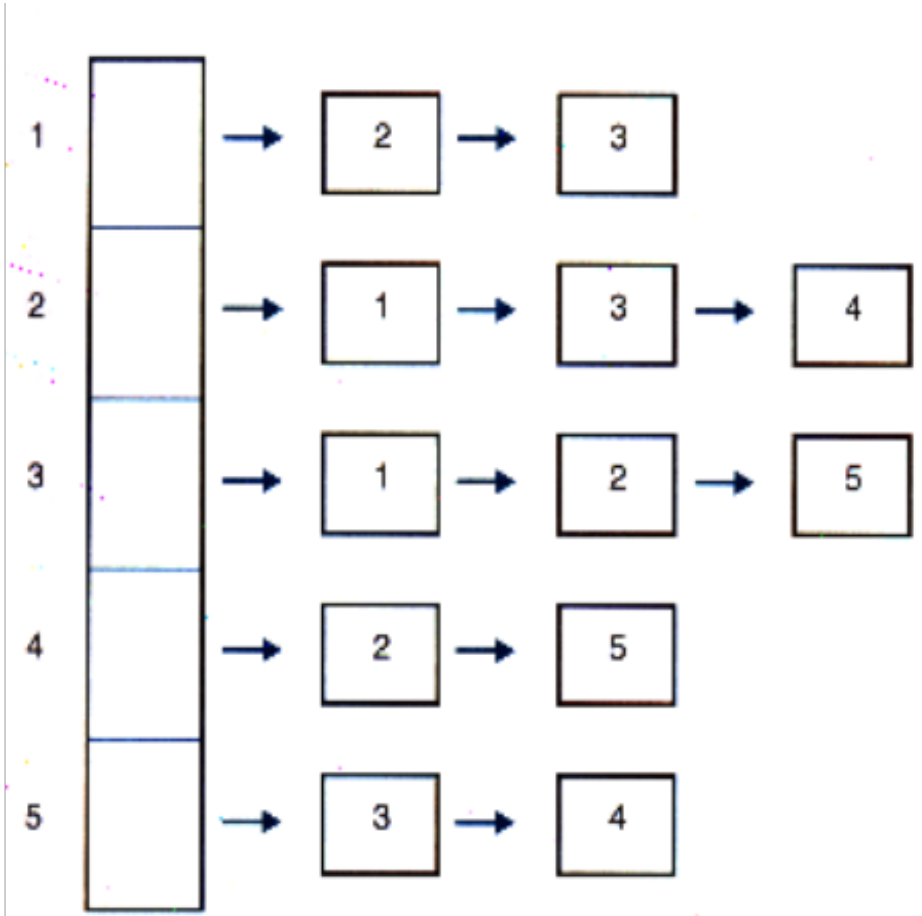
An adjacency list for a graph $G = (V, E)$, with $|V| = N$, will be stored as a one-dimensional array of size N, with each location being a reference to a linked list.

- There will be one list for each node, and that list will have one entry for each adjacent node.

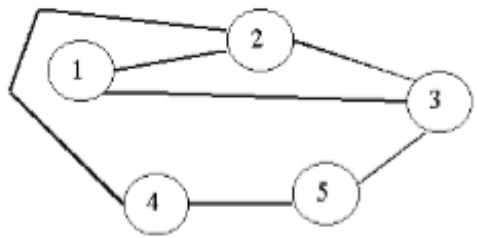
- For weighed graphs and weighed digraphs , the adjacency list entries would have an additional field to hold the weight for that edge.

a) A weighted undirected graph and b) its adjacency list

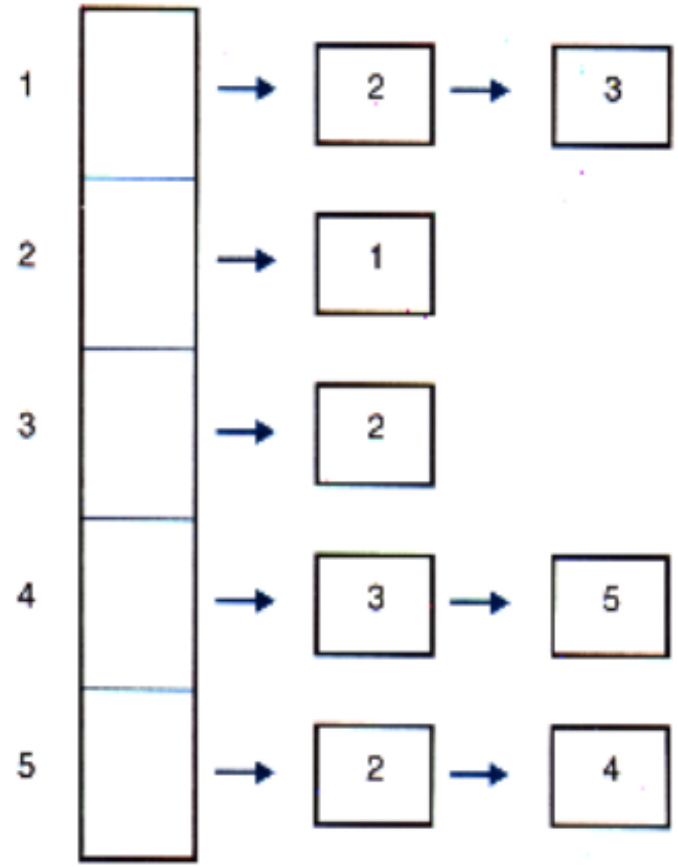




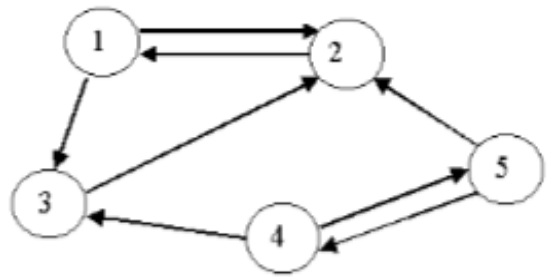
The adjacency list



The graph:
G = ({1, 2, 3, 4, 5}, {{1, 2}, {1, 3}, 2, 3}, {2, 4}{ {3, 5}, 4, 5}})



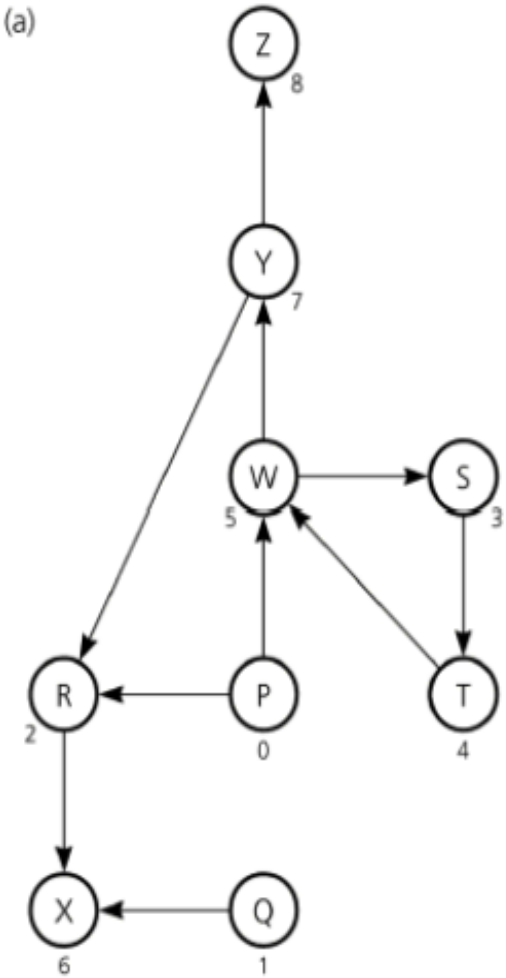
The adjacency list



The digraph:
G = ({1, 2, 3, 4, 5), {(1, 2), (1, 3), (2, 1), (3, 2), (4, 3), (4, 5), (5, 2), (5, 4)})

a) A directed graph and b) its adjacency matrix

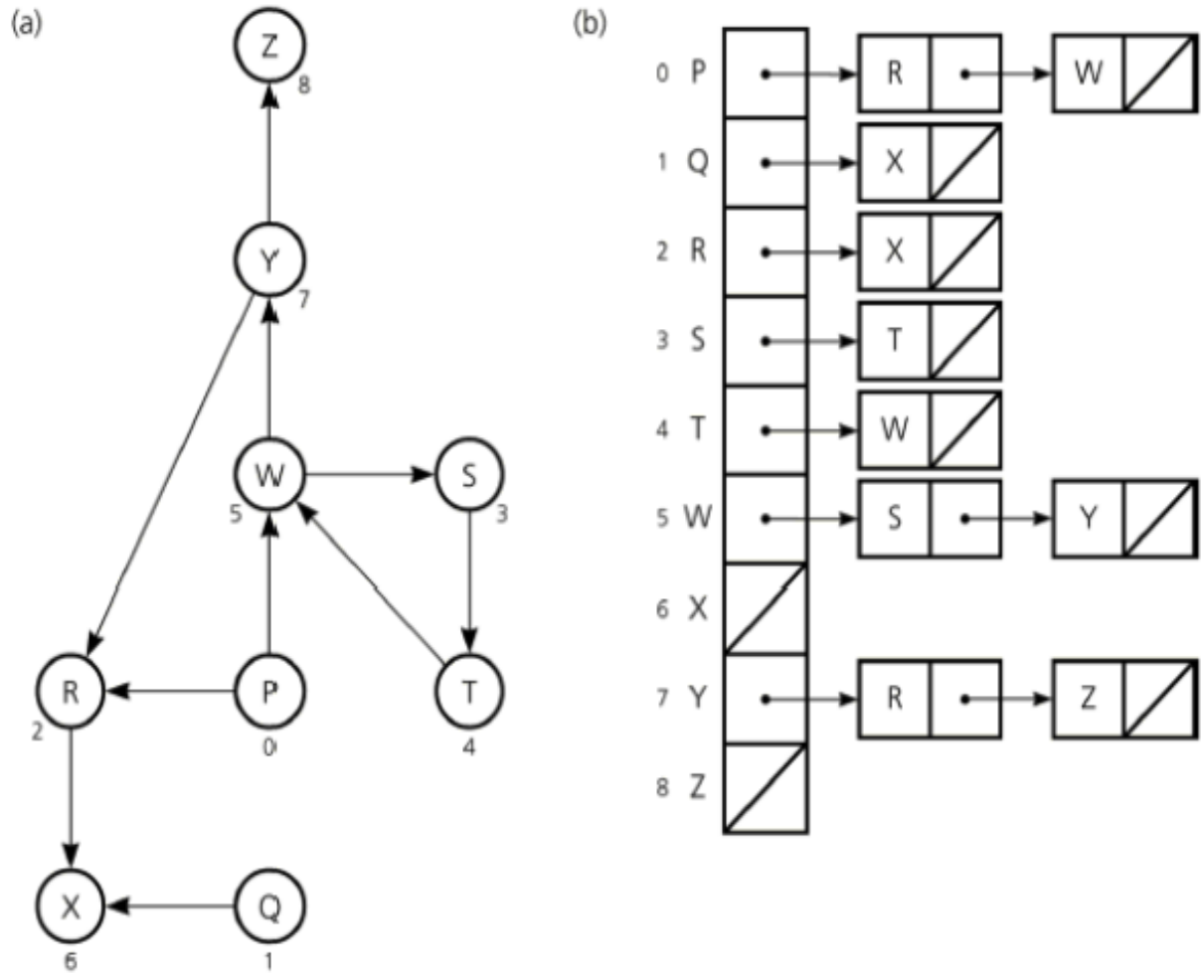
(a)



(b)

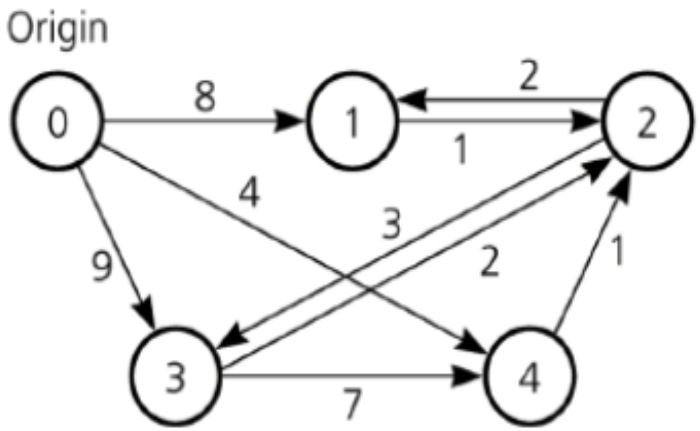
		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

a) A directed graph and b) its adjacency list



a) A weighted directed graph and b) its adjacency matrix

(a)



(b)

	0	1	2	3	4
0	∞	8	∞	9	4
1	∞	∞	1	∞	∞
2	∞	2	∞	3	∞
3	∞	∞	2	∞	7
4	∞	∞	1	∞	∞

The choice by the programmer as to either approach will be closely linked to knowledge of the graphs that will be input to the algorithm.

- In situations where the graph has many nodes , but they are connected to only a few nodes, an **adjacency list**would be best suited because it uses less space, and there will not be long edge lists to traverse.

- In situations where the graph has few nodes , an **adjacency matrix** would be best because it would not be very large, so even a sparse graph would not waste many entries.

- In situations where the graph has many edges and begins to approach a complete graph , an **adjacency matrix** would be best because memory requirements are similar but checking if an edge exists between two vertices takes constant time with an adjacency matrix

Operations of the ADT Graph

1. Create an empty graph.
2. Determine whether a graph is empty.
3. Determine the number of vertices in a graph.
4. Determine the number of edges in a graph.
5. Determine whether an edge exists between two given vertices. For weighted graphs, return weight value.
6. Insert a vertex in a graph whose vertices have distinct search keys that differ from the new vertex's search key.
7. Insert an edge between two given vertices in a graph.
8. Delete a particular vertex from a graph and any edges between the vertex and other vertices.
9. Delete the edge between two given vertices in a graph.
10. Retrieve from a graph the vertex that contains a given search key.

Programs to do this week:

TO DO this week

1. Draw the following graph:

   G = ({1, 2, 3,4, 5, 6}, {{1, 2}, {1, 4}, {2, 5}, {2, 6}, {3, 4}, {3, 5}, {3, 6}, {4, 5}, {4, 6}, {5, 6}}).

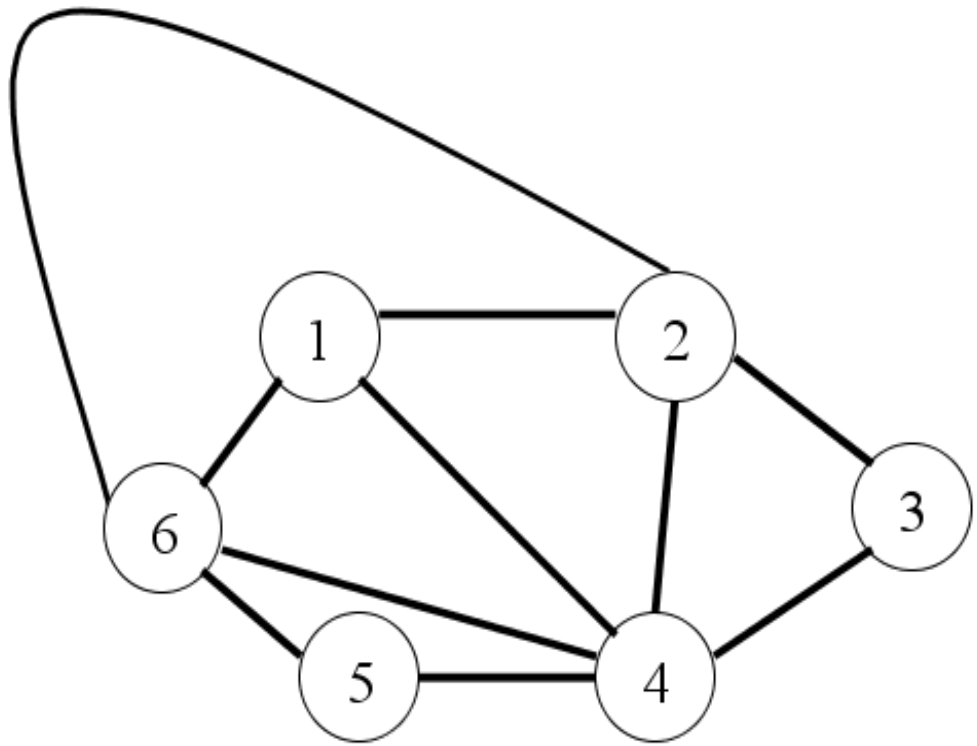
- Give the adjacency matrix for this graph.
- Give the adjacency list for this graph.

2. Draw the following digraph:

   G = ({1, 2, 3, 4, 5}, {(1, 2), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 2), (3, 4), (3, 5), (4, 1), (4, 2), (4, 5), (5, 2), (5, 3), (5, 4)}).

- Give the adjacency matrix for this digraph.
- Give the adjacency list for this digraph.

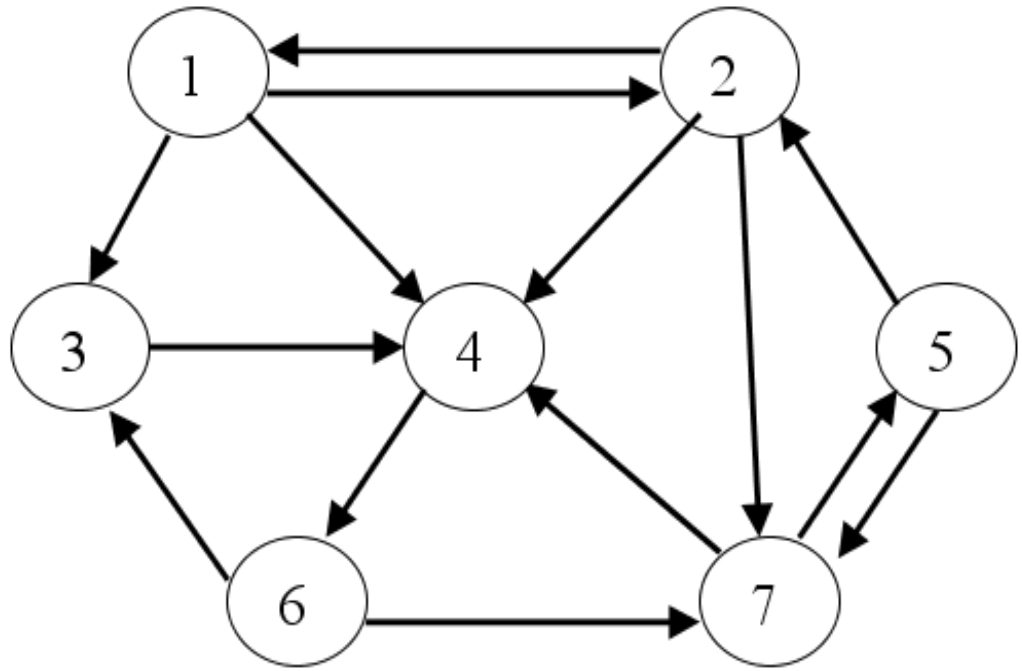
3. Give the set description for the following graph:



- List all the paths between node 1 and node 5 in the above graph.
- List all the cycles that start at node 3 in the above graph.

- Give the adjacency matrix for this graph.
- Give the adjacency list for this graph.

4. Give the set description for the following digraph:



- List all of the paths between node 1 and node 4 in the above digraph.
- List all of the cycles that start at node 7 in the above digraph.

- Give the adjacency matrix for this digraph.
- Give the adjacency list for this digraph.

Traversal of Graphs

When we work with graphs, there will be times when we need to do something to each node in the graph exactly once.

For example, we may need to update all network nodes with some piece of information.

There are two techniques that we can use to achieve this.

These are:

- Depth-first traversal
- Breadth-first traversal

In **depth first traversal**, our traversal will go as far as possible down a path before considering another.

In **breadth first traversal**, our traversal will go evenly in many directions.

For these two techniques, we need to **choose one node as the starting point** from which we navigate the graph.

As we discussed last lecture, we will use the term " **visit the node** " to represent the action that needs to be done at each node.

If, for example, we were searching for a node, then visiting each node would entail checking the node for the information we are looking for.

- These techniques work with both directed graphs and undirected graphs.

Either of these traversal techniques can be used to determine if a graph is connected.

If we create a list of the nodes we visit during each traversal,
this list can be compared to the set of nodes in the graph.

If the set of nodes is the same as those visited,
the graph is **connected**.

Depth First Traversal

In depth first traversal, we visit the starting node and then proceed to follow the links through the graph until we reach a dead end.

- In an **undirected graph**, a node is a dead end if all of the nodes adjacent to it have already been visited.

- In a **directed graph**, if a node has no outgoing edges, then we have reached a dead end

When we reach a dead end, we back up along our path until we find an unvisited adjacent node and then continue in that new direction.

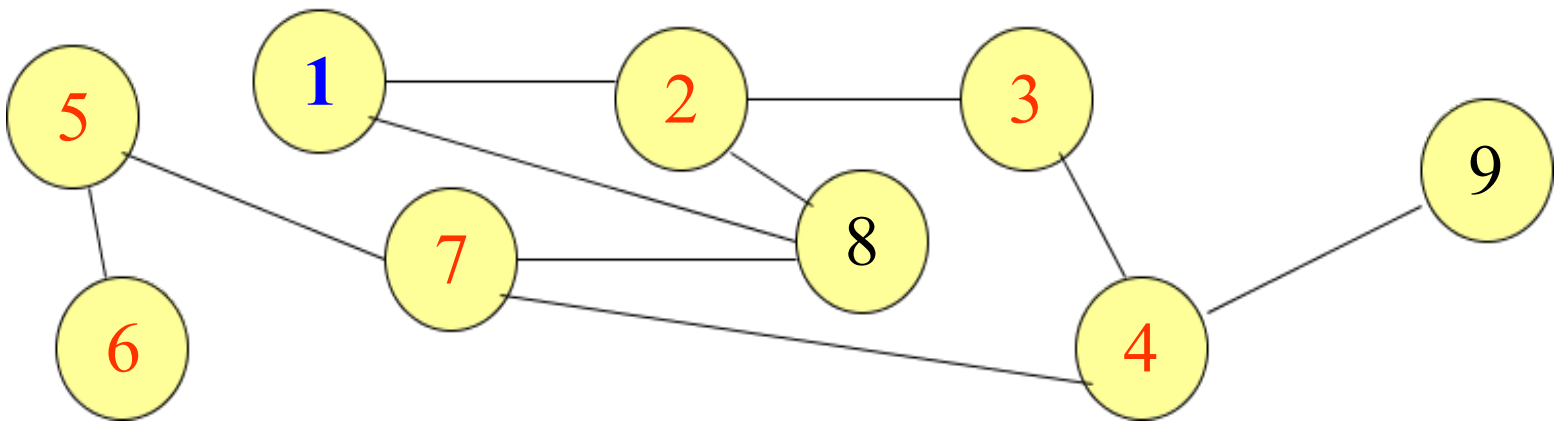
The process will have completed when we back up to the starting node and all of the nodes adjacent to it have been visited.

Where do we start in a graph?

If presented with a choice of two nodes, we will choose the node with the smaller alphabetic or numeric label or search key.

If we begin the **DF traversal at node 1** in the following graph, we then visit, in order, the **nodes 2, 3, 4, 7, 5 and 6** before we reach a dead end.

- We would then back up to **node 7** to find that **node 8** has not yet had a visit, but that immediately leads to a dead end. [note that nodes linked to 8 have already been visited].
- We next back up to **node 4** and find that **node 9** has not yet been visited, but again we have a dead end.
- We continue to back up, but still reach dead ends.
- We then back up to the starting node, and because all of the nodes adjacent have been visited, we have completed the task at hand.



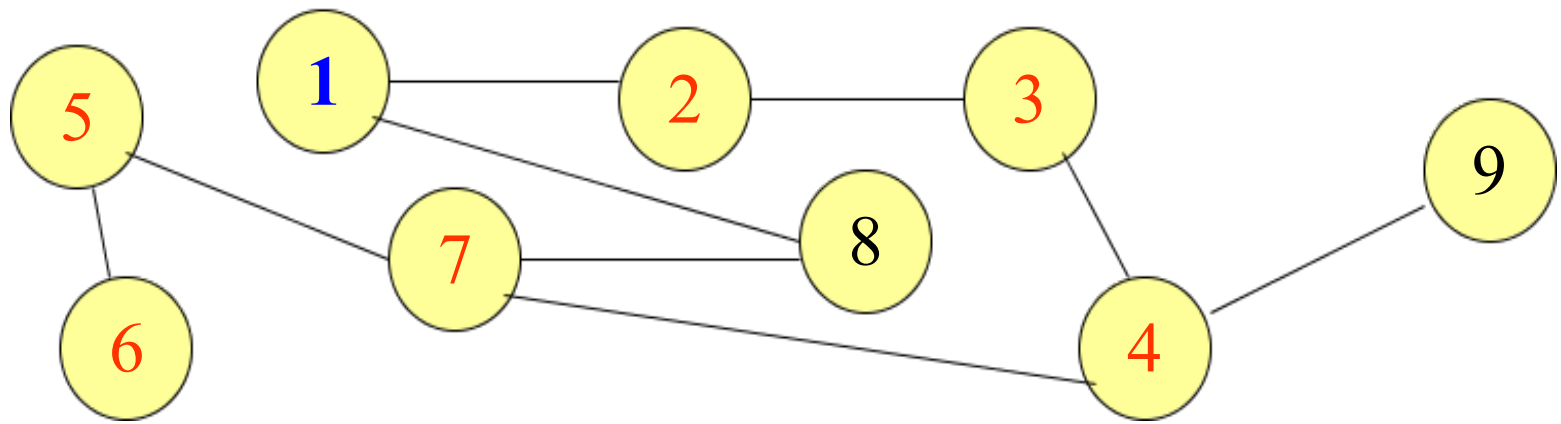
DepthFirstTraversal(G, V)

G the graph;

V the current node/vector;

A **stack** is a good data structure to manage where we have been in the graph with DF traversal.

```
Visit(v );
for (each vertex w adjacent to v)
{
    if (w has not been visited)
    {
        DepthFirstTraversal(G, w)
    }
}
```



Breadth First Traversal

In a breadth first traversal, …

- We visit the **starting node** and then …
- On the **first pass,** visit all of the nodes directly connected to **starting node**
- In the **second pass**, we visit nodes that are two edges away from the starting node.

With each pass, we visit nodes that are one more edge away.

- Because there might be **cycles** in the graph, it is possible for a node to be on two paths of different lengths from the starting node.
- Because we will visit that node for the first time along the shortest path from the starting node, we will not need to consider it again.

We will, therefore, either need to

- keep a list of the nodes visited or
- use a variable in the node to mark it as visited to prevent multiple visits.

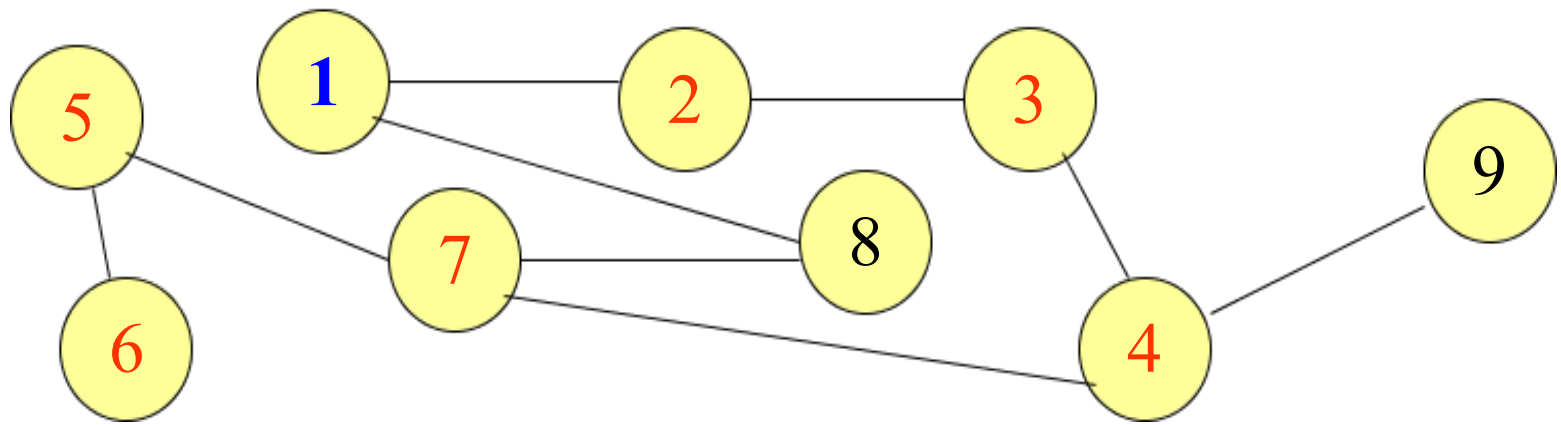
Using the same graph as previous, if we begin our traversal at **node 1**, we will visit nodes 2 and 8 on the first pass.

On the **second pass**, we will visit **nodes 3 and 7**.

- Even thought nodes 2 and 8 are on the same path, we will not return to them, as they were already visited.

On the **third pass**, we visit **nodes 4 and 5**.

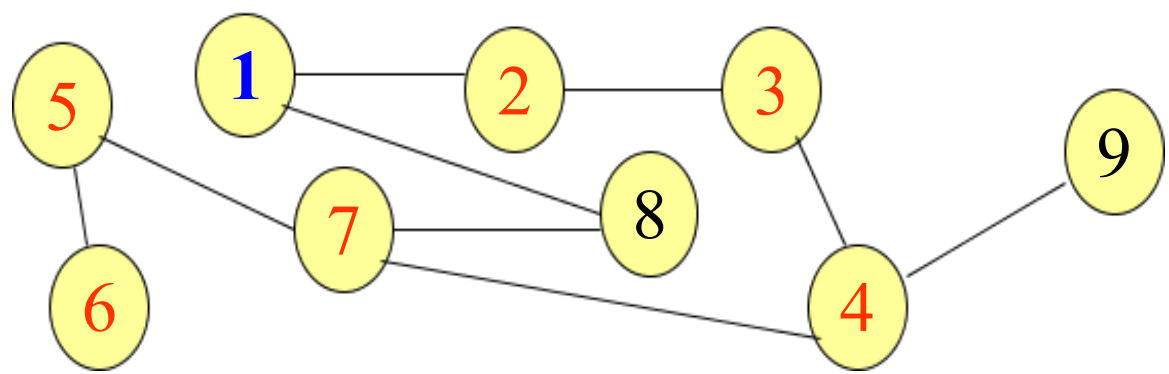
On the **last pass** we visit **nodes 6 and 9**.



The pseudocode for the breadth first traversal is:

```
BreadthFirstTraversal(G, V)
G    the graph;
V    the current node/vector;

visit( v);
enqueue(v);
while (the queue is not empty)
{
      w =  dequeue();
      for (every node u adjacent to w in G)
      {
            if (u has not been visited)
            {
                  visit( u);
                  enqueue(u);
            } //end if
      } //end for
} //end while
```

A **queue** is a good data structure to manage where we have been in a graph with BF traversal



This algorithm will add the starting node of graph to the queue but then immediately remove it.

As it looks at the nodes that are adjacent to the starting node, they will be added to the end of the queue.

Once all of the nodes adjacent to thestarting node have been visited, we will return to the queue and get the first of those nodes.

- We should notice that because nodes are added to the end of the queue, no node that is two edges away from the root will be considered again until all of the nodes one edge away ha ve been taken off the queue and processed.

Remember the attributes of queues, stacks and lists, etc….!

Traversal Analysis

With these two traversal algorithms we have a process that would visit each node of a connected graph exactly once.

We must determine if this has been accomplished.

Is it possible to have a node that was not BF visited in a connected graph?

- Because we move from our initial node to those nodes one edge away, then two edges, three edges etc, the only way we could not visit a node would be if there was no path from it to our initial node.
- Such a condition would mean that the graph is not connected, which contradicts out starting precondition: The graph is connected
- Therefore, all nodes are visited

In **DF**, we travel deeply into the graph until we reach a dead end.

- Each time that we reach a dead end, we back up to the first node that has an unvisited adjacent node that we find.

- There has to be a path from a node to every other node in a connected graph. So when we backtrack from a node we must encounter a path to an unvisited node (if there is any left) unless no path to that unvisited node exists

- Again, this would imply that the graph is not connected, which contradicts our precondition that the graph is connected.

Therefore, all the nodes must be visited in DF traversal of a connected graph.

Efficiency of the Traversals

Our assumption is that the work done as we visit each node is the most complex part of this process.

- The work done to check if an adjacent node has been visited, and the work to traverse the edges is not significant.

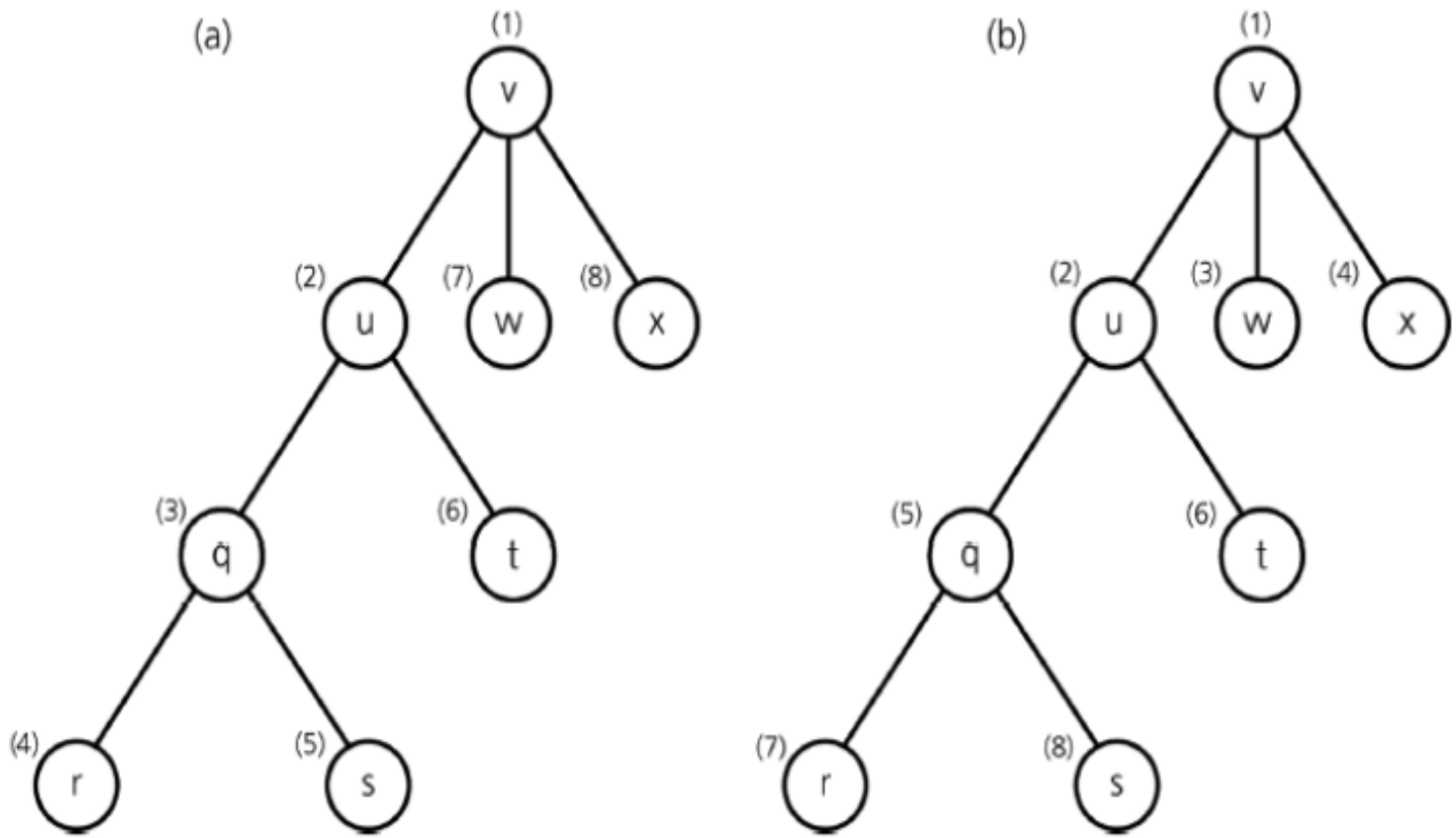
The order of the algorithms is the number of times a node is visited.

- Because we have said that these algorithms visit each node exactly one time, for a graph with N nodes, the visit process will be done N times.

- These traversals are therefore of order **O(N)**.

We will look at some more examples….

Visitation order for a) a depth-first search; b) a breadth-first search

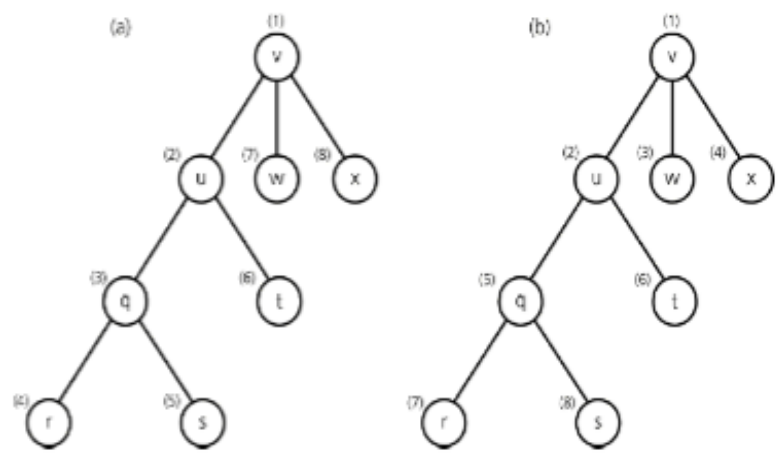


An iterative version of the DF algorithm called DFS for depth first search.

An iterative DFS traversal algorithm uses a stack

```
dfs(v)
// Traverses a graph beginning at vertex v by using a
// depth-first search: Iterative version.

  s.createStack()

  // push v onto the stack and mark it
  s.push(v)
  Mark v as visited

  // loop invariant: there is a path from vertex v at the
  // bottom of the stack s to the vertex at the top of s
  while (!s.isEmpty()) {
    if (no unvisited vertices are adjacent to
          the vertex on the top of the stack)
      s.pop()  // backtrack
    }
    else {
      Select an unvisited vertex u adjacent to
           the vertex on the top of the stack
      s.push(u)
      Mark u as visited
    }  // end if
  }  // end while
```

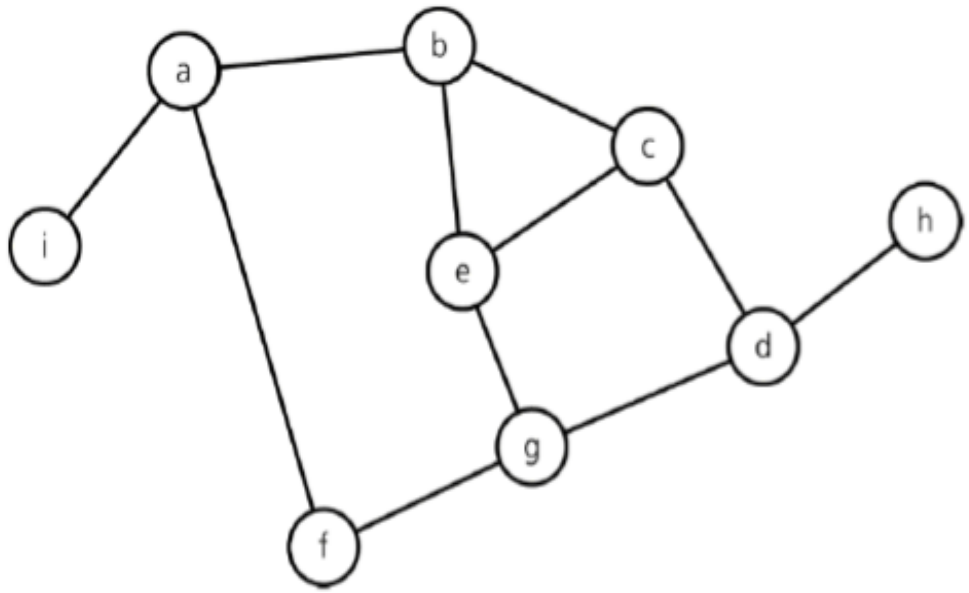
Visitation order for a) a depth-first search; b) a breadth-first search



Trace of DFS progress using a stack

The results of a depth-first traversal, beginning at vertex *a*, of the graph

A connected graph with cycles

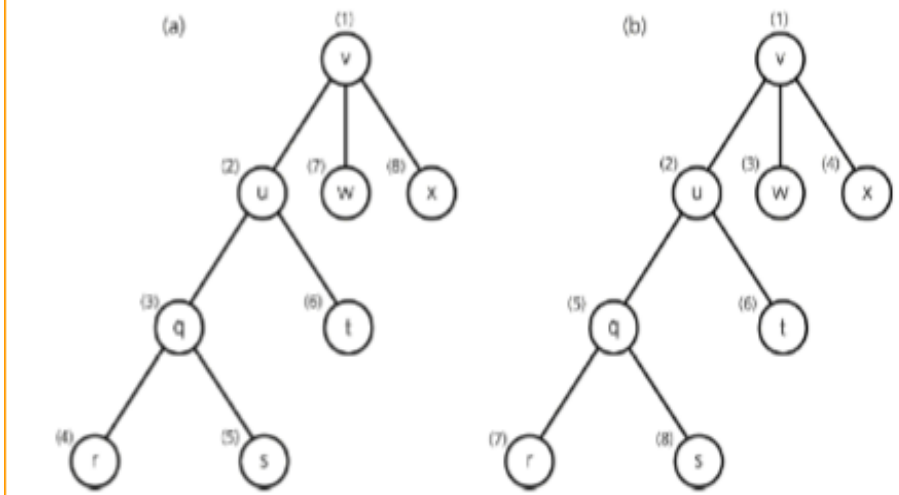


Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	*(empty)*

An iterative version of BFS

```
bfs(v)
// Traverses a graph beginning at vertex v by using a
// breadth-first search: Iterative version.

  q.createQueue()

  // add v to queue and mark it
  q.enqueue(v)
  Mark v as visited

  while (!q.isEmpty()) {
      w = q.dequeue()

      // loop invariant: there is a path from vertex w to
      // every vertex in the queue q
      for (each unvisited vertex u adjacent to w) {
         Mark u as visited
         q.enqueue(u)
      } // end for
   } // end while
```

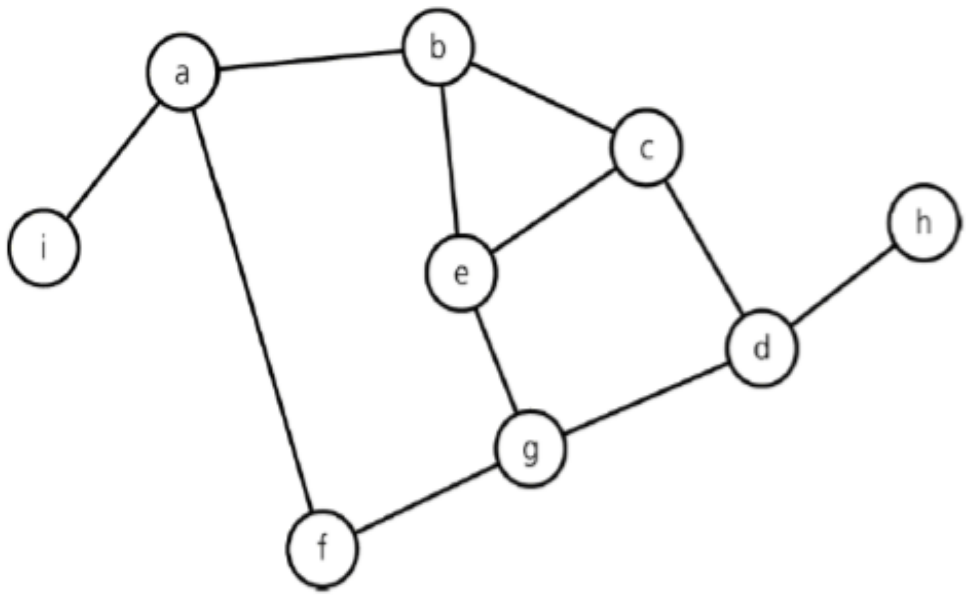
Visitation order for a) a depth-first search; b) a breadth-first search



Trace of progress of BF using a queue

The results of a breadth-first traversal, beginning at vertex *a*, of the graph

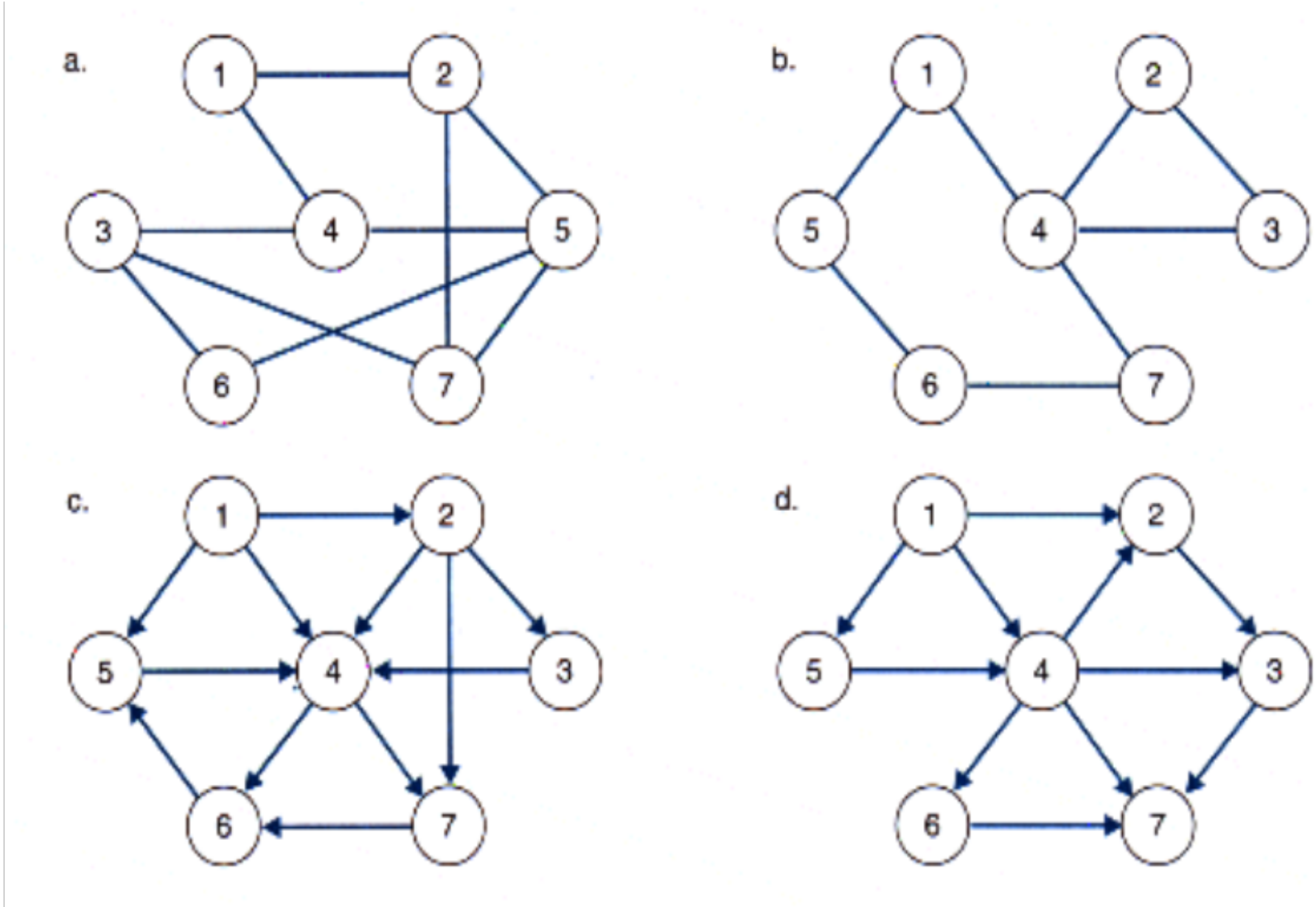
A connected graph with cycles



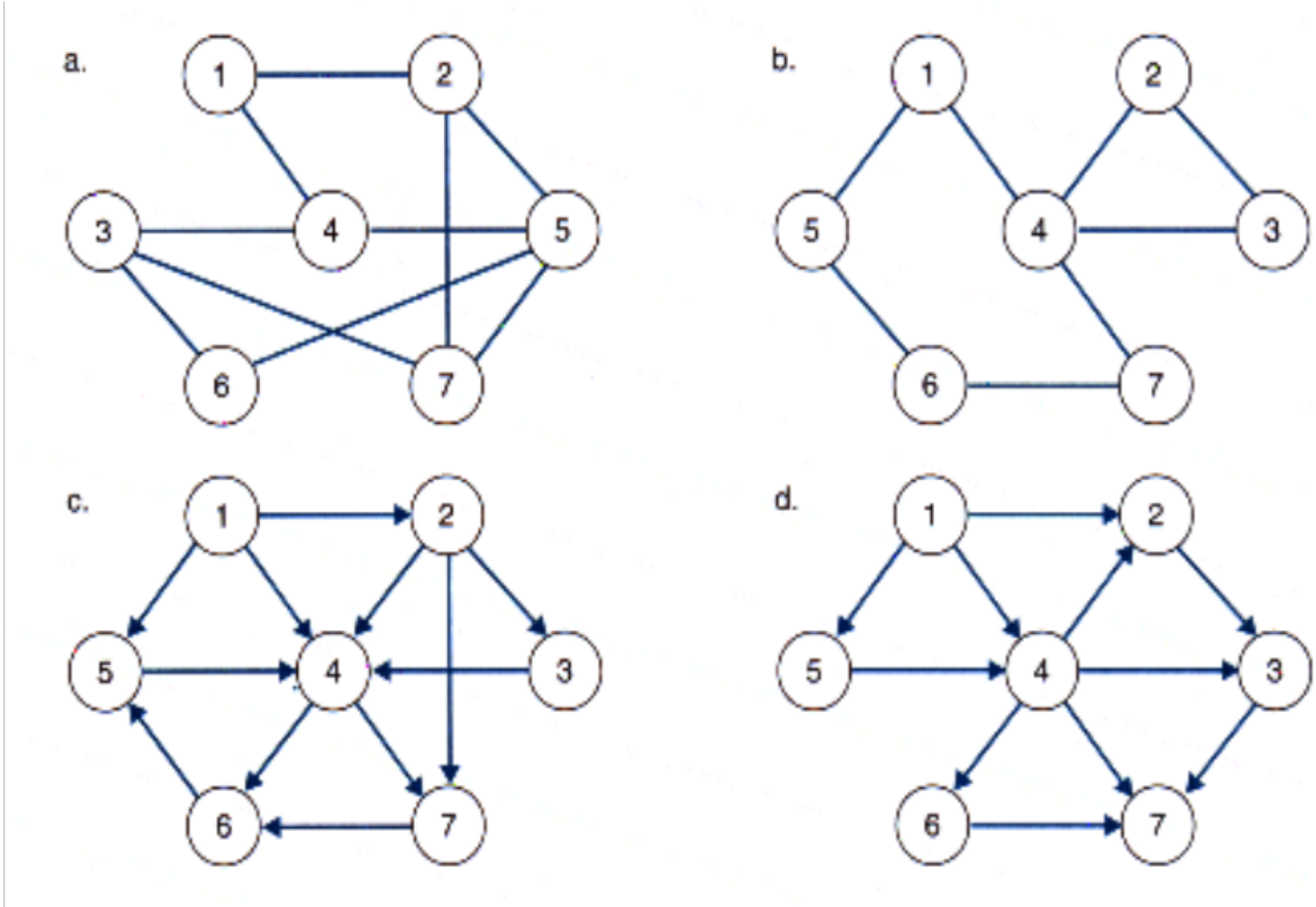
Node visited	Queue (front to back)
a	a
	(empty)
b	b
f	b f
i	b f i
	f i
c	f i c
e	f i c e
	i c e
g	i c e g
	c e g
	e g
d	e g d
	g d
	d
	(empty)
h	h
	(empty)

Programs to do this week:

1. For the following graphs, give the order that the nodes will be first visited
when doing a **breadth-first traversal** starting at the node labelled with a 1.



2. For the following graphs, give the order that the nodes will be first visited when doing a **depth-first traversal** starting at the node labelled with a 1.



3. You are required to verify your DF Traversal answer for graph (a) (back one slide) by implementing the recursive DFS algorithm in a Java application/applet.

Use an adjacency matrix to represent the graph, but include an extra column in the graph to keep track of which nodes have been visited.

You must use a recursive function that performs a Depth First traversal of the graph and shows the order of this traversal by printing out the current node or vertex (See pseudo code overleaf)

Create a graph class that has methods to add an edge between two nodes and also to do a DF traversal of the graph

You should also create a test class that creates a graph object, adds the required edges and invokes it's traversal

DepthFirstTraversal(G, V)

G the graph;

V the current node/vector;

```
Visit(v );   Print(v);
for (each vertex w adjacent to v)
{
    if (w has not been visited)
    {
        DepthFirstTraversal(G, w)
    }
}
```