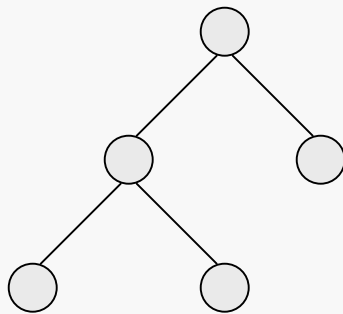
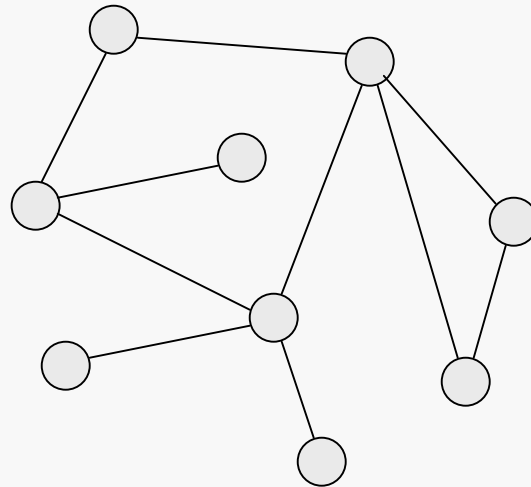


A graph G consists of a set V of vertices and a set E of pairs of distinct vertices from V . These pairs of vertices are called edges.

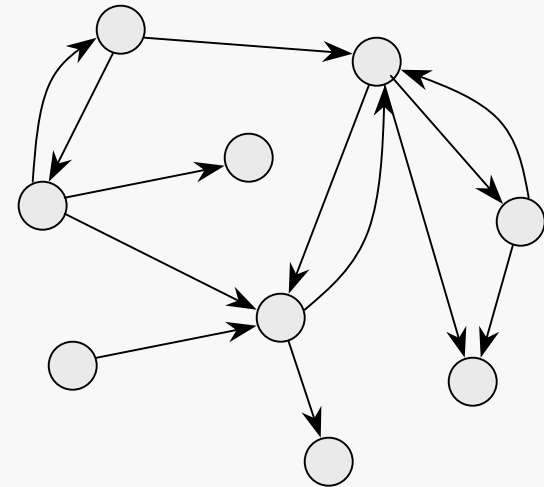
If the pairs of vertices are unordered, G is an undirected graph. If the pairs of vertices are ordered, G is a directed graph or digraph.



A tree is a graph.



An undirected graph.



A directed graph.

An undirected graph G , where:

$V = \{a, b, c, d, e, f, g, h, i\}$

$E = \{ \{a, b\}, \{a, c\}, \{b, e\}, \{b, h\}, \{b, i\},$
 $\{c, d\}, \{c, e\}, \{e, f\}, \{e, g\}, \{h, i\} \}$

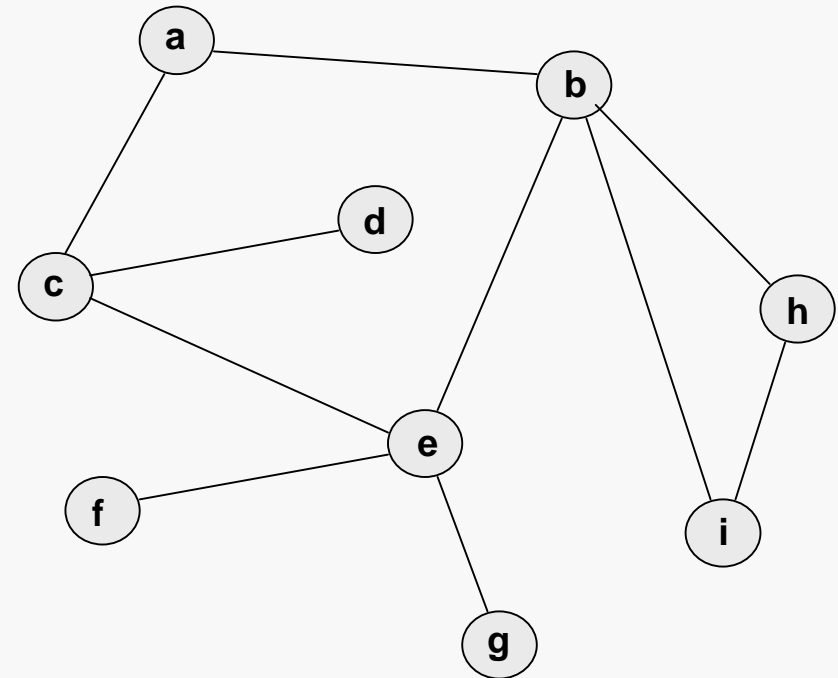
$e = \{c, d\}$ is an edge, incident upon the vertices c and d

Two vertices, x and y , are adjacent if $\{x, y\}$ is an edge (in E).

A path in G is a sequence of distinct vertices, each adjacent to the next.

A path is simple if no vertex occurs twice in the path.

A cycle in G is a path in G , containing at least three vertices, such that the last vertex in the sequence is adjacent to the first vertex in the sequence.

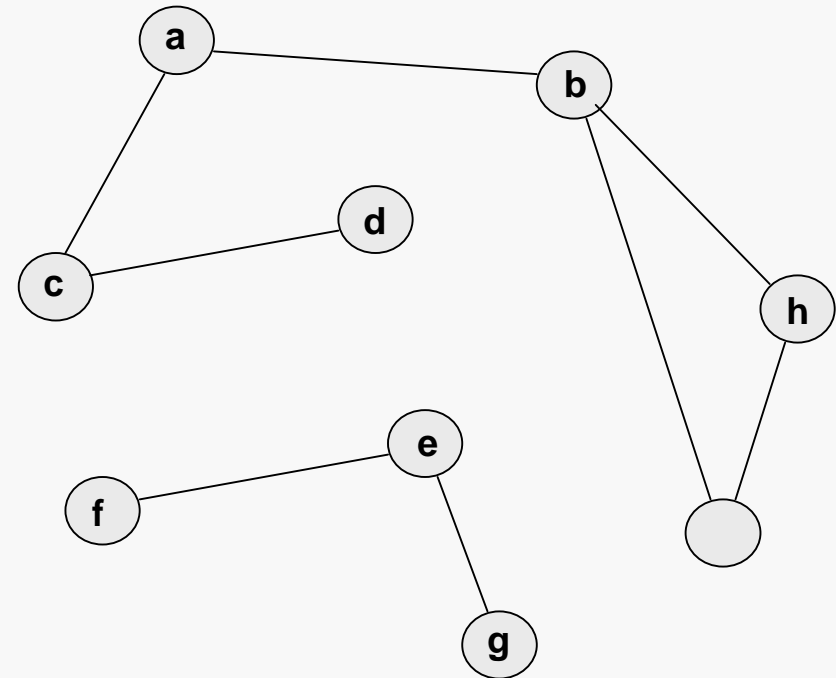


Undirected Graph Terminology

A graph G is connected if, given any two vertices x and y in G , there is a path in G with first vertex x and last vertex y .

The graph on the previous slide is connected.

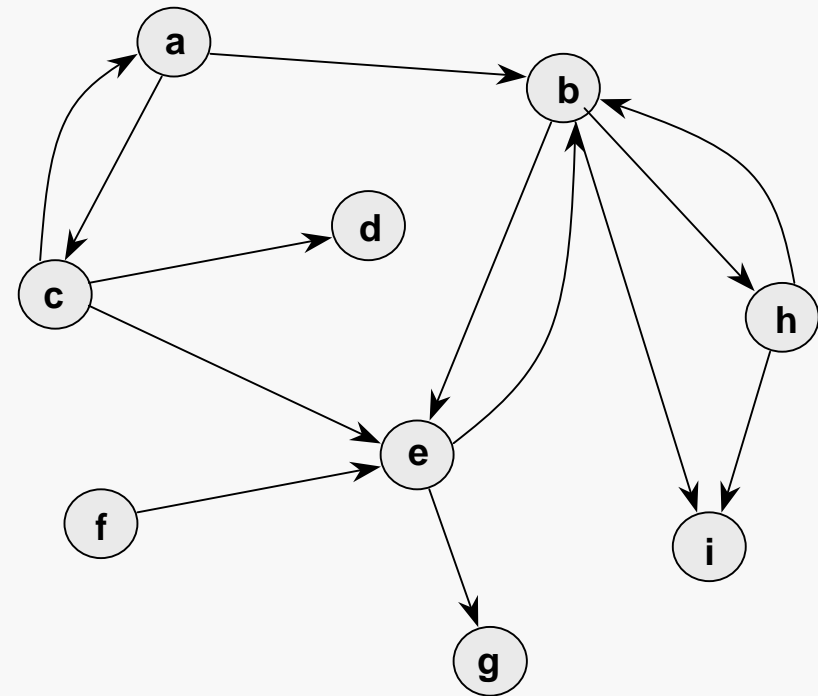
If a graph G is not connected, then we say that a maximal connected set of vertices is a component of G .



The terminology for directed graphs is only slightly different...

$e = (c, d)$ is an edge, from c to d

A directed path in a directed graph G is a sequence of distinct vertices, such that there is an edge from each vertex in the sequence to the next.



A directed graph G is weakly connected if, the undirected graph obtained by suppressing the directions on the edges of G is connected (according to the previous definition).

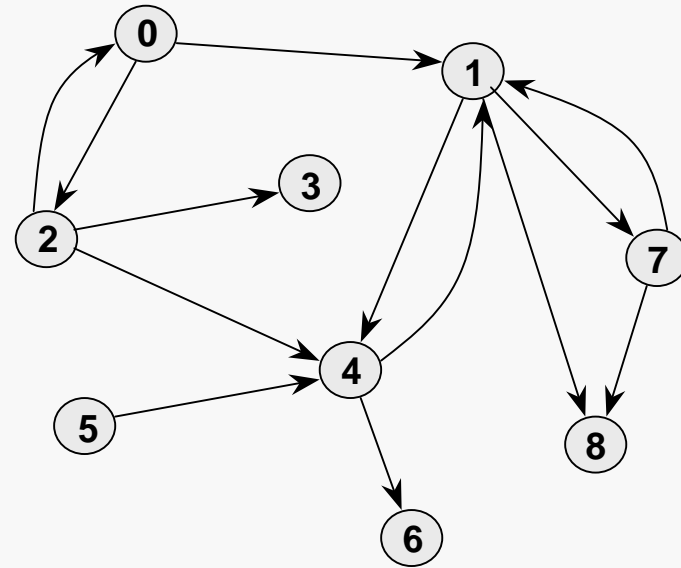
A directed graph G is strongly connected if, given any two vertices x and y in G , there is a directed path in G from x to y .

Adjacency Table Representation

A graph may be represented by a two-dimensional adjacency table:

If G has $n = |V|$ vertices, let M be an n by n matrix whose entries are defined by

$$m_{ij} = \begin{cases} 1 & \text{if } (i, j) \text{ is an edge} \\ 0 & \text{otherwise} \end{cases}$$



$$M(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The adjacency table:

- $\Theta(1)$ to determine existence of a specific edge
- $\Theta(|V|^2)$ storage cost (cut cost by 75% by using `bool` instead of `int`)
- $\Theta(|V|)$ for finding all neighbors of a specific vertex
- $\Theta(1)$ to add or delete an edge
- Not easy to add or delete a vertex; better for static graph structure.
- Symmetric matrix for undirected graph; so half is redundant then.

$$M(G) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

An Adjacency Table Class

```
class AdjacencyTable {
protected:
    string      Name;
    int         numVertices;
    Array2DT<bool> Table;
    bool*      Marker;

public:
    AdjacencyTable(int V = 0, string N = "Unknown");
    AdjacencyTable(const AdjacencyTable& Source);
    AdjacencyTable& operator=(const AdjacencyTable& Source);

    string  getName() const;
    int     getnumVertices() const;

    bool    addEdge(int Source, int Terminus);
    bool    deleteEdge(int Source, int Terminus);
    bool    isEdge(int Source, int Terminus) const;

    // . . . continued . . .
};
```

Array2DT is a template that encapsulates a virtual two-dimensional array.

Marker is a dynamically allocated bool array, used for vertex marking algorithms.

An Adjacency Table Class

```
// . . . continued . . .
```

```
int firstNeighbor(int Source) const;  
int nextNeighbor(int Source, int prevNeighbor) const;
```

```
bool isMarked(int Vertex) const;  
bool Mark(int Vertex);  
bool unMark(int Vertex);
```

```
void Clear();
```

```
virtual void Display(ostream& Out);  
~AdjacencyTable();
```

```
};
```

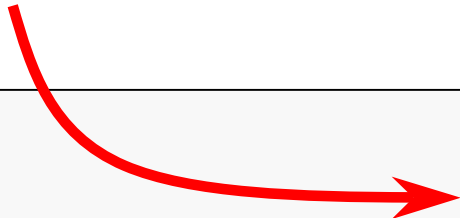
firstNeighbor() returns the first vertex adjacent to Source.

nextNeighbor() returns the next vertex, after prevNeighbor, which is adjacent to Source.

Clear() deletes all the edges from the graph.

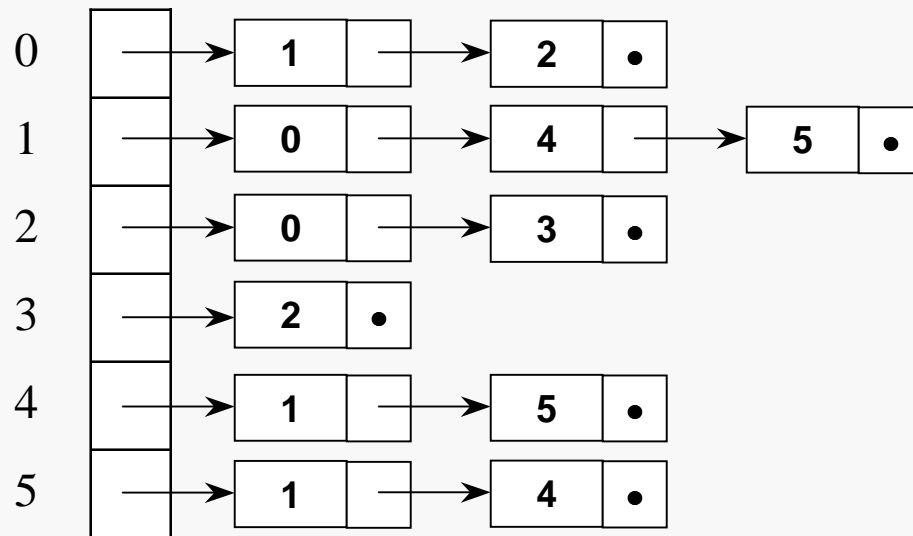
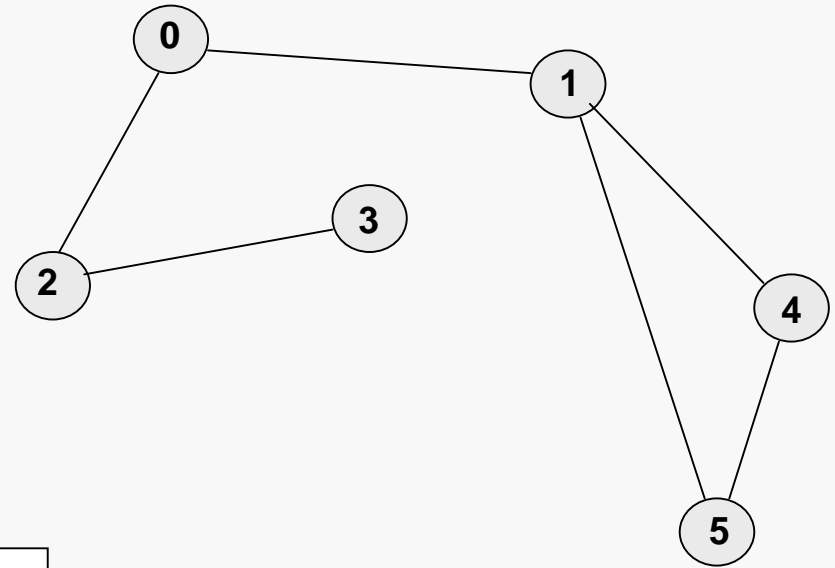
Graph: Sample07

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



Adjacency List Representation

A graph may also be represented by an adjacency list structure:



Array of linked lists, where list nodes store node labels for neighbors.

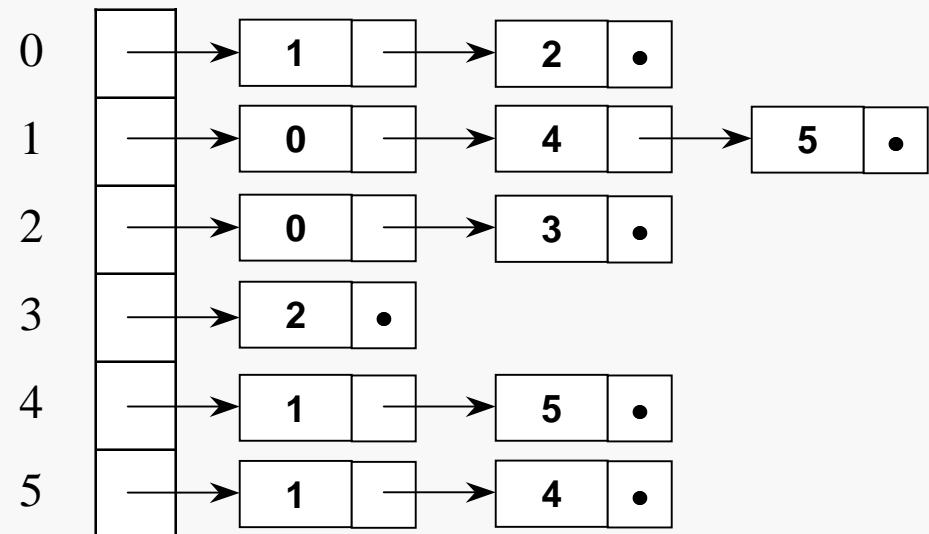
The adjacency list structure:

- Worst case: $\Theta(|V|)$ to determine existence of a specific edge
- $\Theta(|V| + |E|)$ storage cost
- Worst case: $\Theta(|V|)$ for finding all neighbors of a specific vertex
- Worst case: $\Theta(|V|)$ to add or delete an edge
- Still not easy to add or delete a vertex; however, we can use a linked list in place of the array.

Note, for an undirected graph, the upper bound on the number of edges is:

$$|E| \leq |V| * (|V|-1)$$

So, the space comparison with the adjacency table scheme is not trivial.



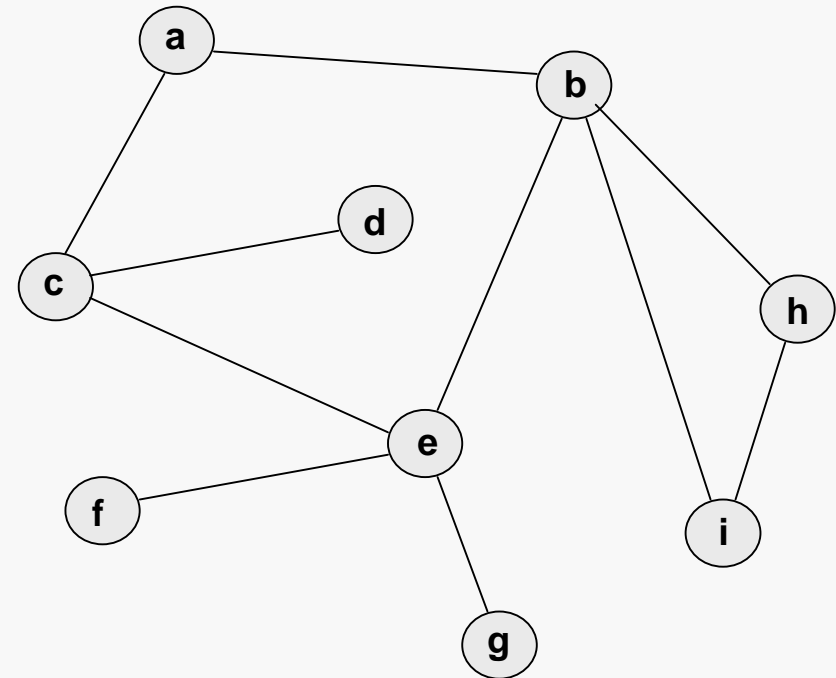
Some algorithms require that every vertex of a graph be visited exactly once.

The order in which the vertices are visited may be important, and may depend upon the particular algorithm.

The two common traversals:

- depth-first
- breadth-first
- topological sort

During a traversal we must keep track of which vertices have been visited; the most common approach is to provide some sort of “marking” support.

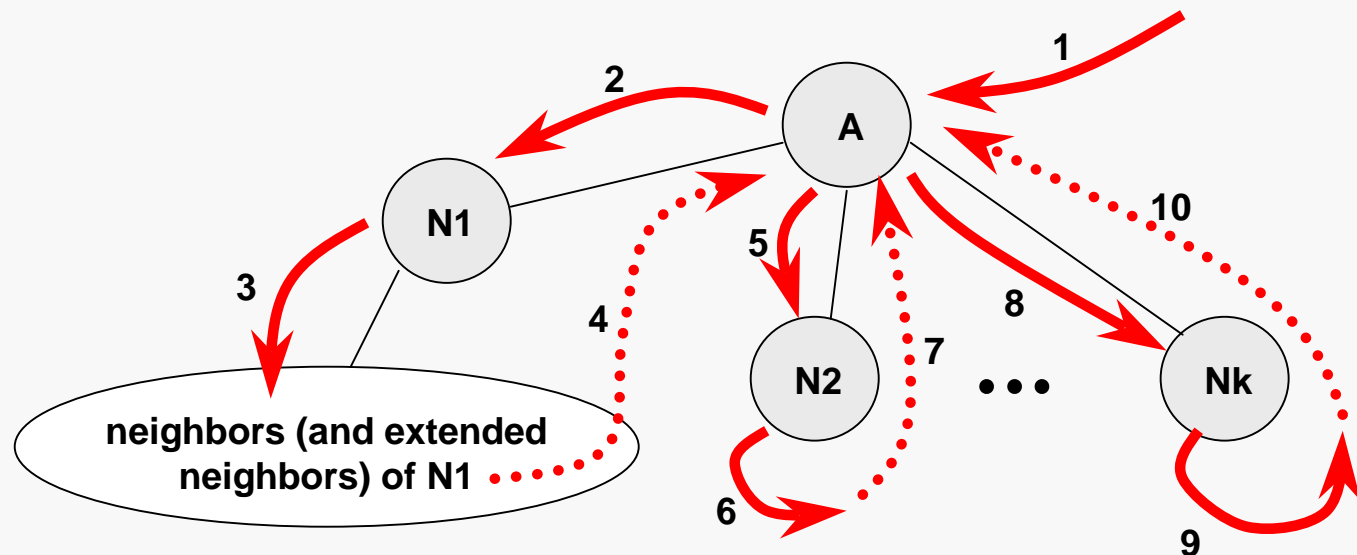


Assume a particular node has been designated as the starting point.

Let A be the last node visited and suppose A has neighbors $N1, N2, \dots, Nk$.

A depth-first traversal will:

- visit $N1$, then
- proceed to traverse all the unvisited neighbors of $N1$, then
- proceed to traverse the remaining neighbors of A in similar fashion.



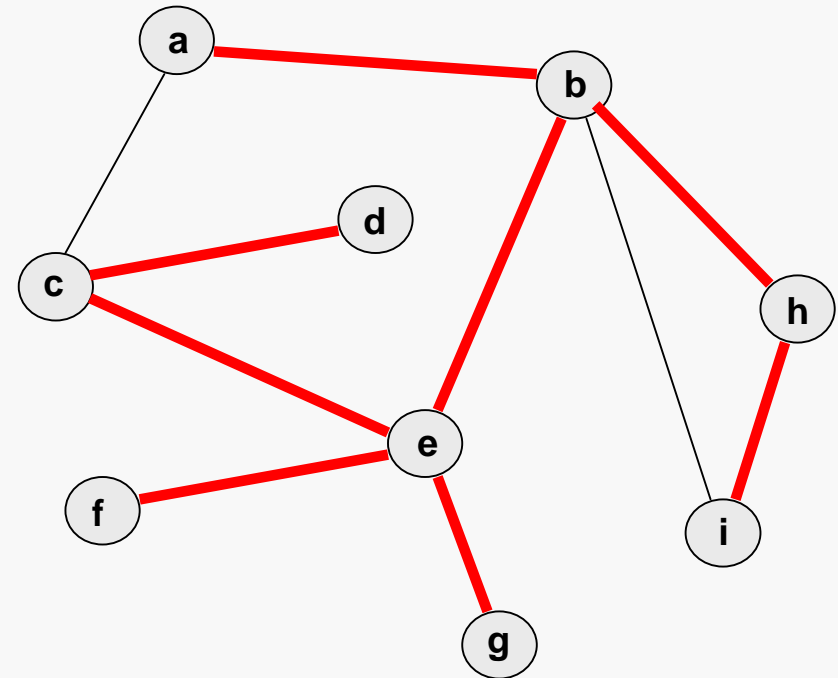
Graph Traversals: Depth-First

Assuming the node labeled **a** has been designated as the starting point, a depth-first traversal would visit the graph nodes in the order:

a b e c d f g h i

Note that if the edges taken during the depth-first traversal are marked, they define a tree (not necessarily binary) which includes all the nodes of the graph.

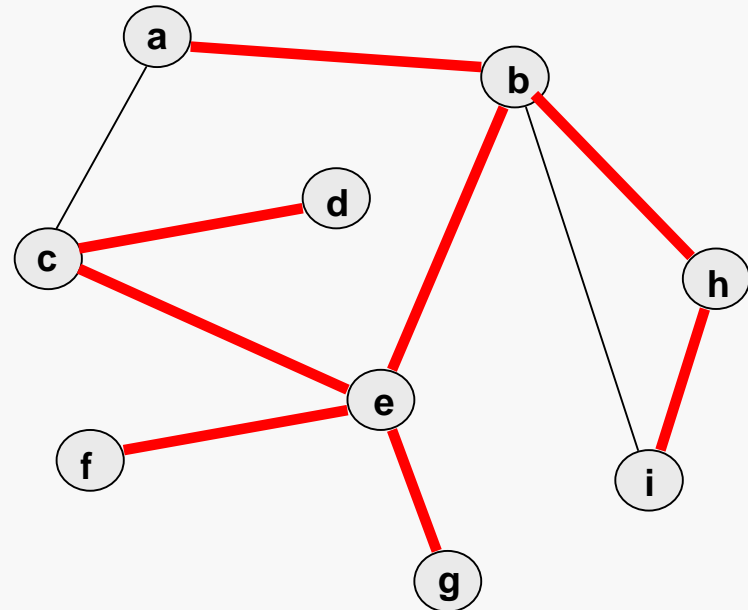
Such a tree is called a spanning tree for the graph.



Implementing a Depth-First Traversal

```
void DFS(AdjacencyTable& G, int Source) {  
    G.Mark(Source);  
    for (int w = G.firstNeighbor(Source);  
         G.isEdge(Source, w); w = G.nextNeighbor(Source, w) ) {  
        if ( !G.isMarked(w) )  
            DFS(G, DFSTree, w);  
    }  
}
```

If we modify DFS () to take another AdjacencyTable object as a parameter, it is relatively trivial to have DFS () build a copy of the spanning tree.

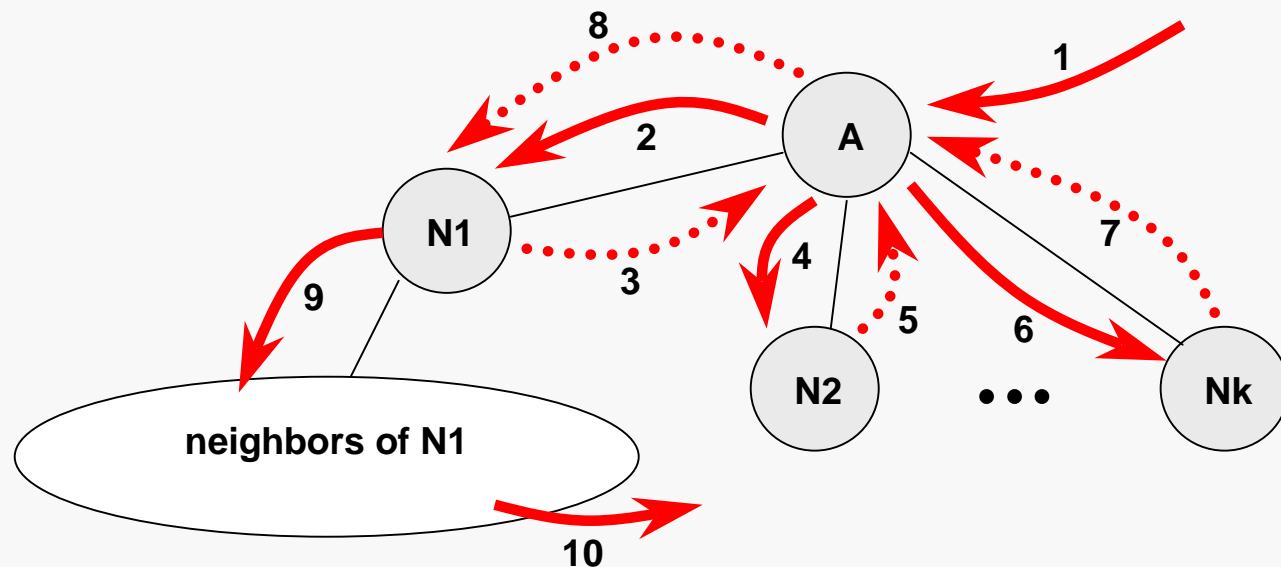


Assume a particular node has been designated as the starting point.

Let A be the last node visited and suppose A has neighbors $N1, N2, \dots, Nk$.

A breadth-first traversal will:

- visit $N1$, then $N2$, and so forth through Nk , then
- proceed to traverse all the unvisited immediate neighbors of $N1$, then
- traverse the immediate neighbors of $N2, \dots, Nk$ in similar fashion.



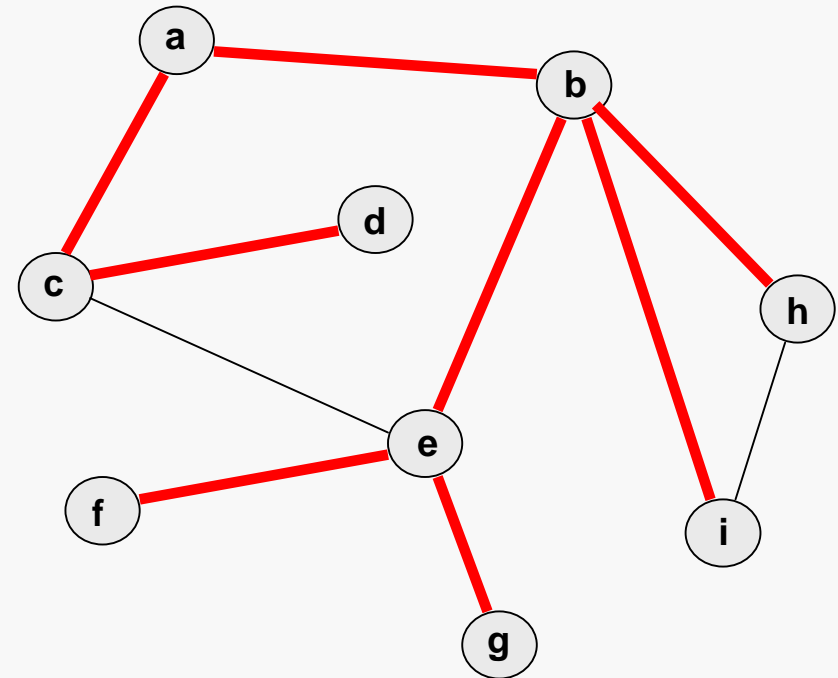
Graph Traversals: Breadth-First

Assuming the node labeled **a** has been designated as the starting point, a breadth-first traversal would visit the graph nodes in the order:

a b c e h i d f g

Note the edges taken during the breadth-first traversal also define a spanning tree for the given graph.

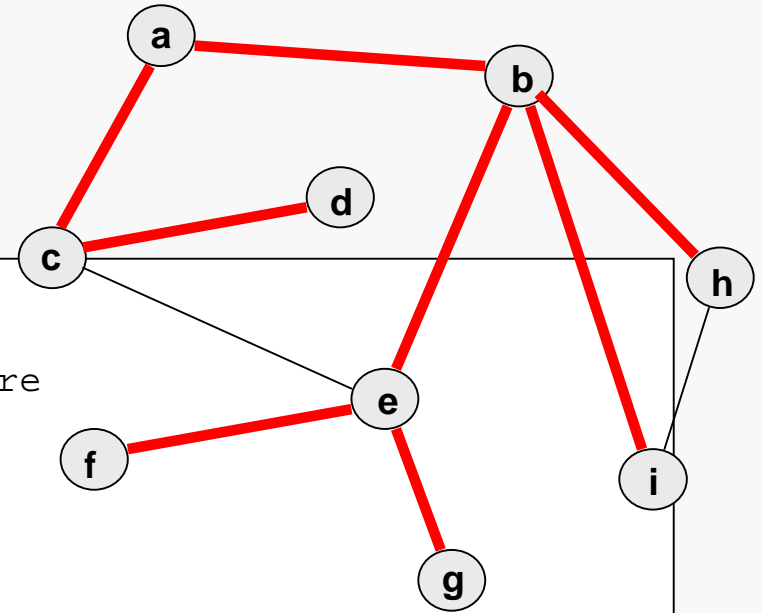
As is the case here, the breadth-first spanning tree is usually different from the depth-first spanning tree.



Implementing a Breadth-First Traversal

The breadth-first traversal uses a local queue to organize the graph nodes into the proper order:

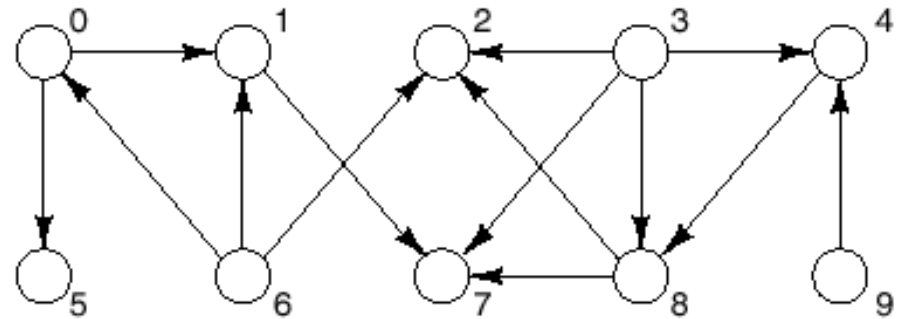
```
void BFS(AdjacencyTable& G, int Source) {  
  
    QueueT<int> toVisit; // schedule nodes here  
    toVisit.Enqueue(Source);  
    G.Mark(Source);  
  
    while ( !toVisit.isEmpty() ) {  
        int VisitNow = toVisit.Dequeue();  
  
        for (int w = G.firstNeighbor(VisitNow);  
            G.isEdge(VisitNow, w); w = G.nextNeighbor(VisitNow, w) ) {  
            if ( !G.isMarked(w) ) {  
                toVisit.Enqueue(w);  
                G.Mark(w);  
            }  
        }  
    }  
}
```



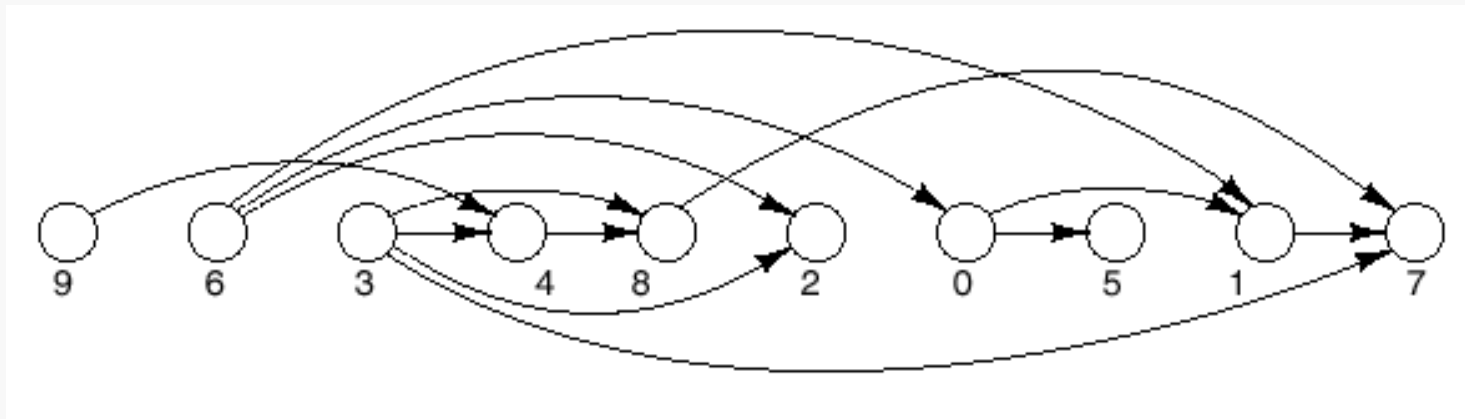
The for loop schedules all the unvisited neighbors of the current node for future visits.

Topological Ordering

Suppose that G is a directed graph which contains no directed cycles:



Then, a topological ordering of the vertices in G is a sequential listing of the vertices such that for any pair of vertices, v and w in G , if (v,w) is an edge in G then v precedes w in the sequential listing.



Computing a Topological Ordering

A topological ordering of the vertices of G may be obtained by performing a generalized depth-first traversal.

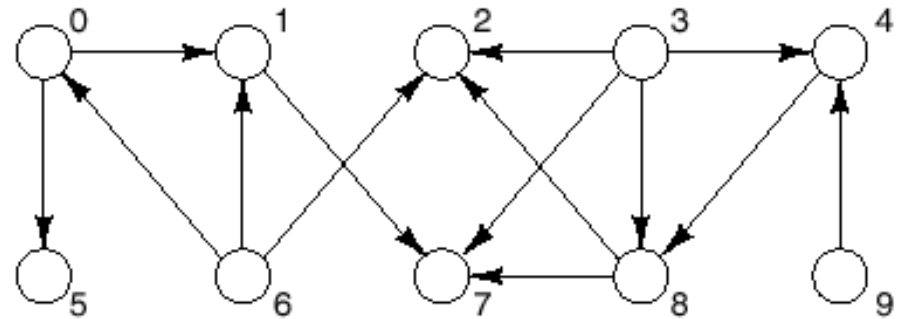
We build a list L of vertices of G .

Begin with vertex 0.

Do a depth-first traversal, marking each visited vertex and adding a vertex to L only if it has no unmarked neighbors.

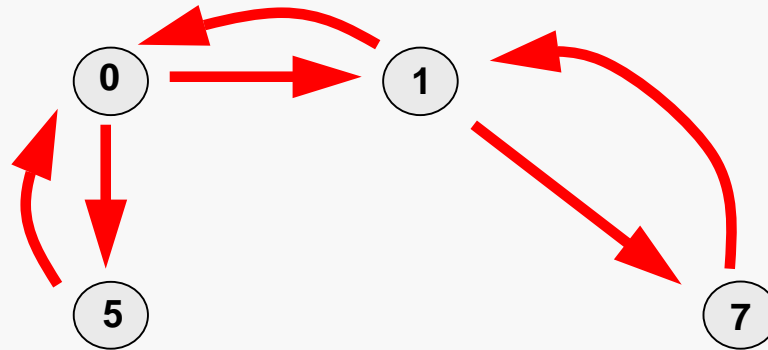
When the traversal adds its starting vertex to L , pick the first unmarked vertex as a new starting point and perform another depth-first traversal.

Stop when all vertices have been marked.



Depth-First Traversal Trace

Initially we probe from 0 to 1 to 7, which has no successors.



L: 7

Next the recursion backs out to 1, which has no unmarked successors.

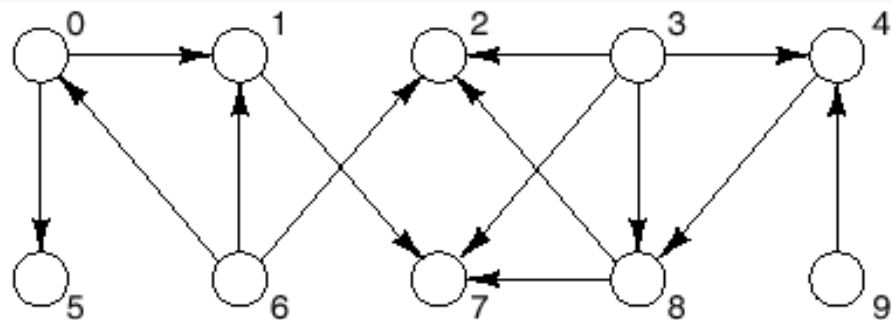
L: 1 7

Next the recursion backs out to 0, and probes to 5, which has no successors.

L: 5 1 7

Next the recursion backs out to 0 again, which now has no unmarked successors.

L: 0 5 1 7



Depth-First Traversal Trace

Now we pick the first unmarked vertex, 2, and continue the process. 2 has no successors.

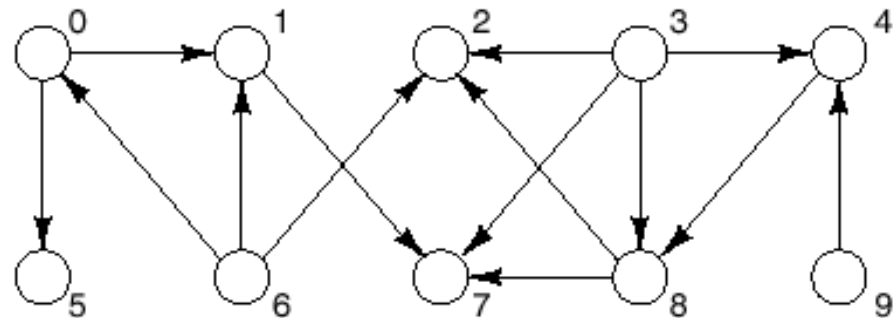
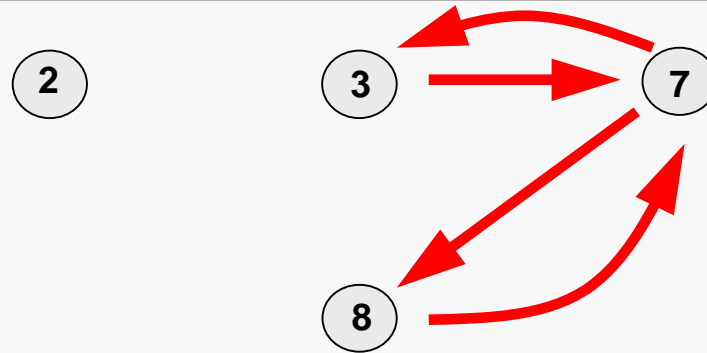
L: 2 0 5 1 7

Next start with vertex 3, and probe to 4 and then to 8, which has no unmarked successors.

L: 8 2 0 5 1 7

The recursion backs out, adding vertices 4 and 3.

L: 3 4 8 2 0 5 1 7



Depth-First Traversal Trace

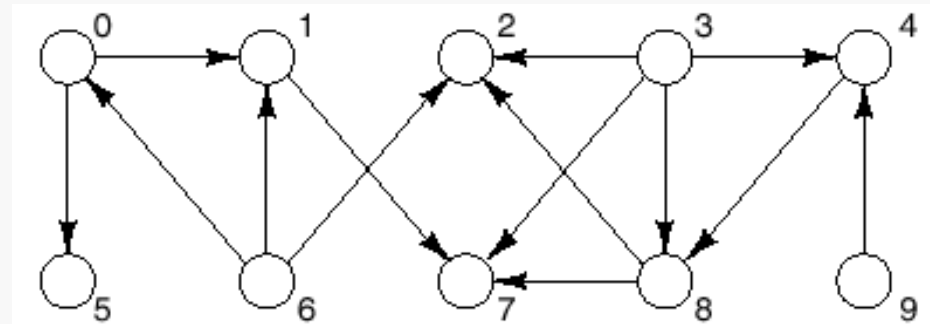
Next we pick the unmarked vertex, 6, which has no unmarked successors.

L: 6 3 4 8 2 0 5 1 7

Next we pick the unmarked vertex, 9, which has no unmarked successors.

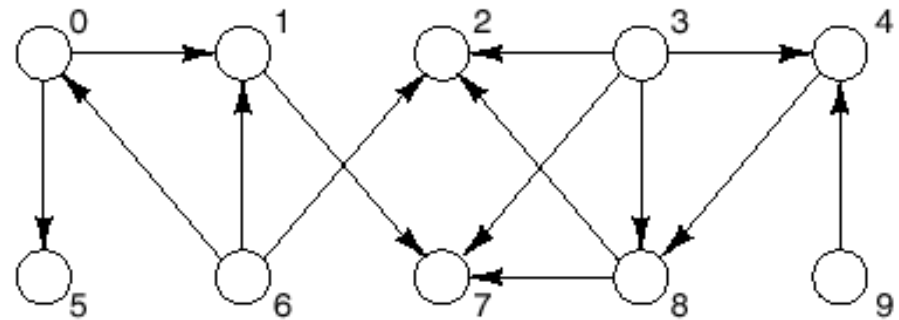
L: 9 6 3 4 8 2 0 5 1 7

At this point, all the vertices have been marked and the algorithm terminates.

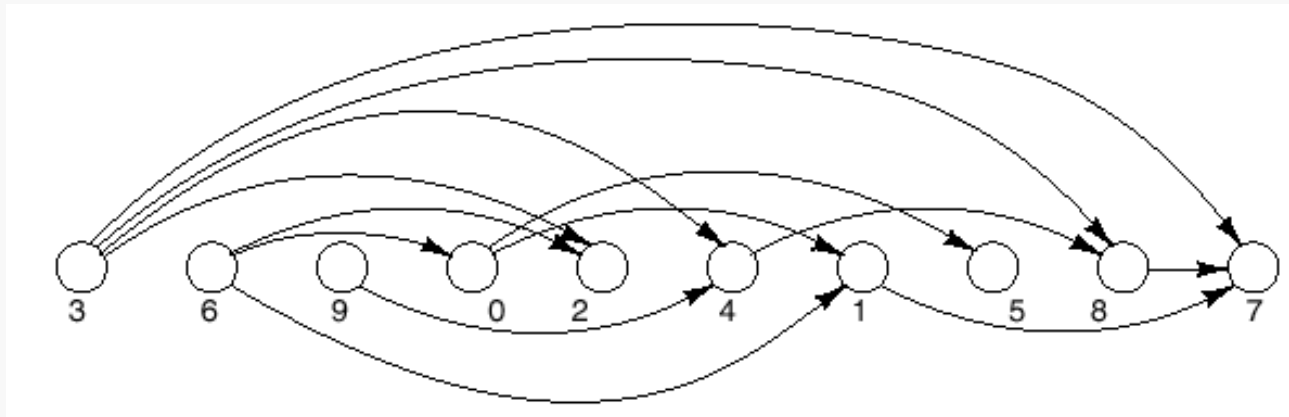


Multiple Topological Orderings

There is usually more than one topological ordering for a graph. Choosing a different rule for picking the starting vertices may yield a different ordering.



Also, a generalized breadth-first traversal can be used instead. For the graph above, a breadth-first traversal yields the ordering:



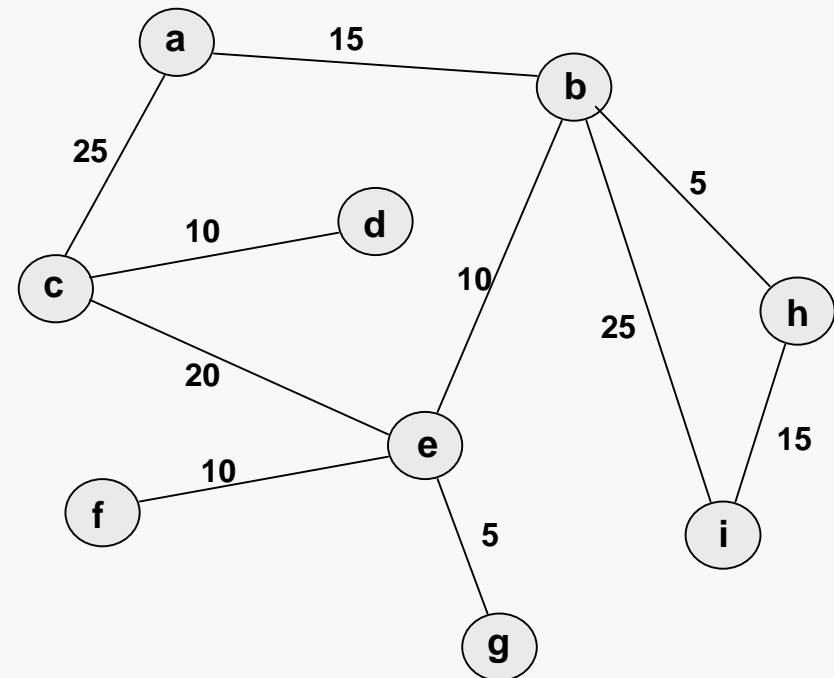
Applications of topological orderings are relatively common...

- prerequisite relationships among courses
- glossary of technical terms whose definitions involve dependencies
- organization of topics in a book or a course

In many applications, each edge of a graph has an associated numerical value, called a weight.

Usually, the edge weights are non-negative integers.

Weighted graphs may be either directed or undirected.



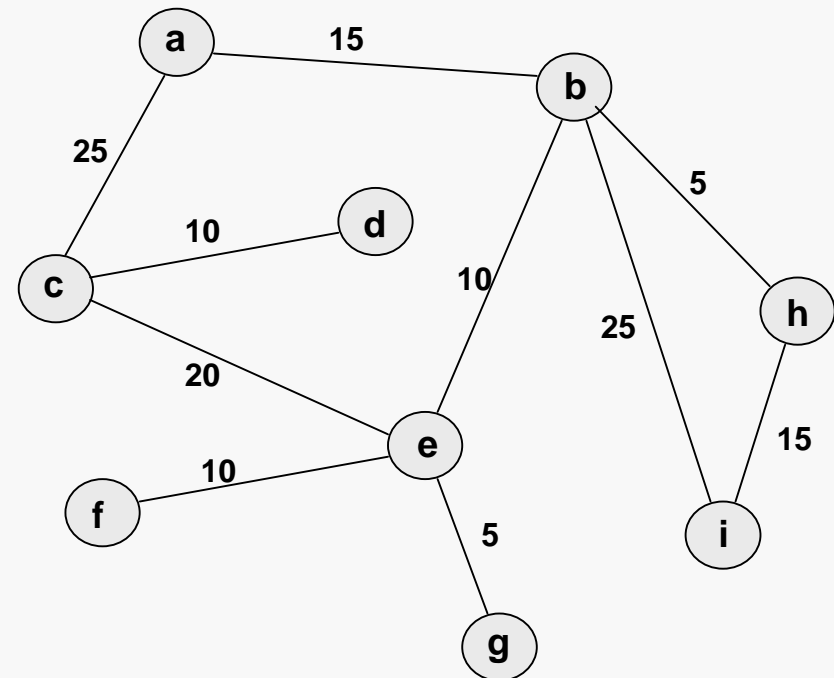
The weight of an edge is often referred to as the "cost" of the edge.

In applications, the weight may be a measure of the length of a route, the capacity of a line, the energy required to move between locations along a route, etc.

Shortest Path Problem*

Given a weighted graph, and a designated node S , we would like to find a path of least total weight from S to each of the other vertices in the graph.

The total weight of a path is the sum of the weights of its edges.



We have seen that performing a DFS or BFS on the graph will produce a spanning tree, but neither of those algorithms takes edge weights into account.

There is a simple, greedy algorithm that will solve this problem.

*single source, all destinations version

Dijkstra's Algorithm*

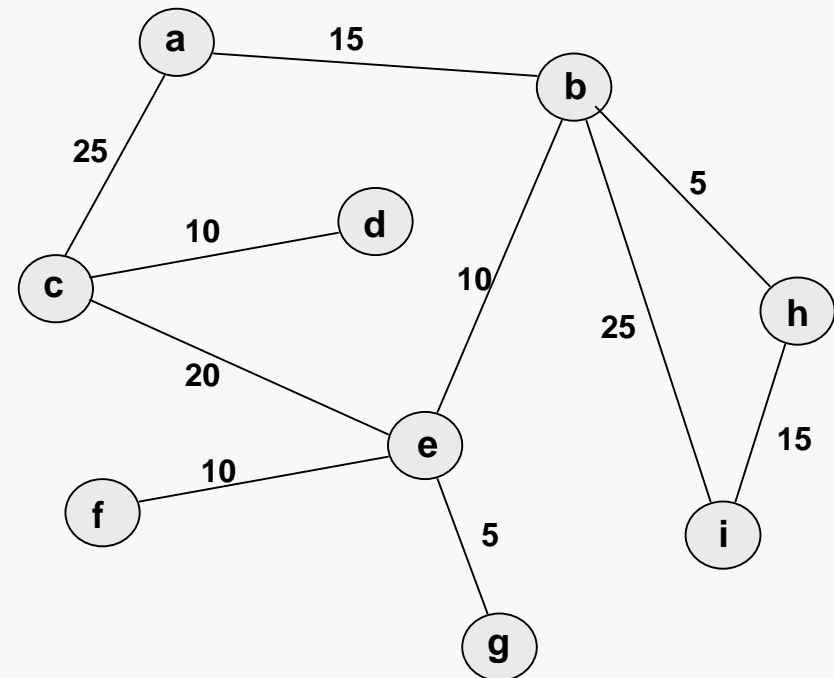
We assume that there is a path from the source vertex S to every other vertex in the graph.

Let S be the set of vertices whose minimum distance from the source vertex has been found. Initially S contains only the source vertex.

The algorithm is iterative, adding one vertex to S on each pass.

We maintain a table D such that for each vertex v , $D(v)$ is the minimum distance from the source vertex to v via vertices that are already in S (aside possibly from v itself).

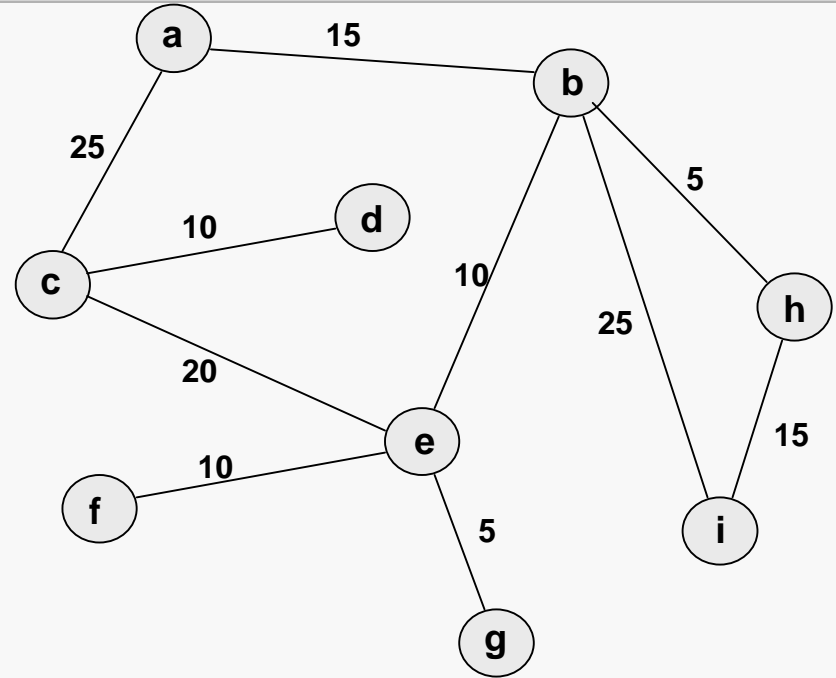
Greed: on each iteration, add to S the vertex v not already in S for which $D(v)$ is minimal.



*1959

Dijkstra's Algorithm Trace

Let the source vertex be a.



$S = \{a\}$

D	a	b	c	d	e	f	g	h	i
	0	15	25	∞	∞	∞	∞	∞	∞

$S = \{a,b\}$

D	a	b	c	d	e	f	g	h	i
	0	0	25	∞	10	∞	∞	5	25

Dijkstra's Algorithm Trace

Continuing:

$S = \{a, b, h\}$

D	a	b	c	d	e	f	g	h	i
	0	0	25	∞	10	∞	∞	0	15

$S = \{a, b, h, e\}$

D	a	b	c	d	e	f	g	h	i
	0	0	20	∞	0	10	5	0	15

$S = \{a, b, h, e, g\}$

D	a	b	c	d	e	f	g	h	i
	0	0	20	∞	0	10	0	0	15

$S = \{a, b, h, e, g, f\}$

D	a	b	c	d	e	f	g	h	i
	0	0	20	∞	0	0	0	0	15

$S = \{a, b, h, e, g, f, i\}$

D	a	b	c	d	e	f	g	h	i
	0	0	20	∞	0	0	0	0	0

Dijkstra's Algorithm Trace

Continuing:

$S = \{a, b, h, e, g, f, i, c\}$

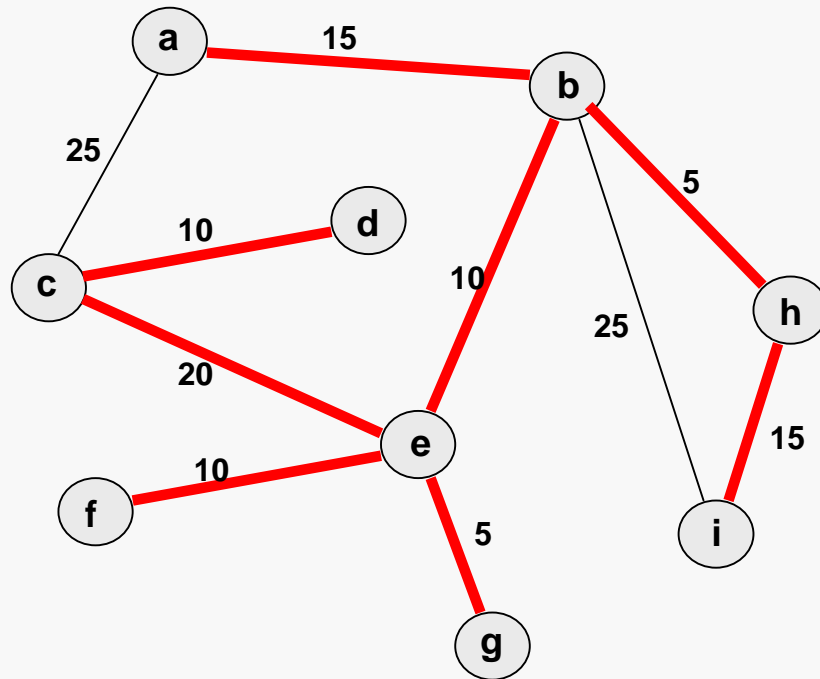
D

a	b	c	d	e	f	g	h	i
0	0	0	10	0	0	0	0	0

$S = \{a, b, h, e, g, f, i, c, d\}$

D

a	b	c	d	e	f	g	h	i
0	0	0	0	0	0	0	0	0



The corresponding tree is shown at left. As described, the algorithm does not maintain a record of the edges that were used, nor does it produce a table of minimum distances for the shortest paths found.

The algorithm can be modified to record the tree and a table of minimum distances by:

- list the edges used as nodes are added
- adding a mark field for each vertex to the distance table, or using one provided by the graph structure and retaining the distance values instead of setting them to zero

$S = \{a\}$

$E = \{\}$

D	a	b	c	d	e	f	g	h	i
	0	15	25	∞	∞	∞	∞	∞	∞
	Y	N	N	N	N	N	N	N	N

$S = \{a, b\}$

$E = \{\{a,b\}\}$

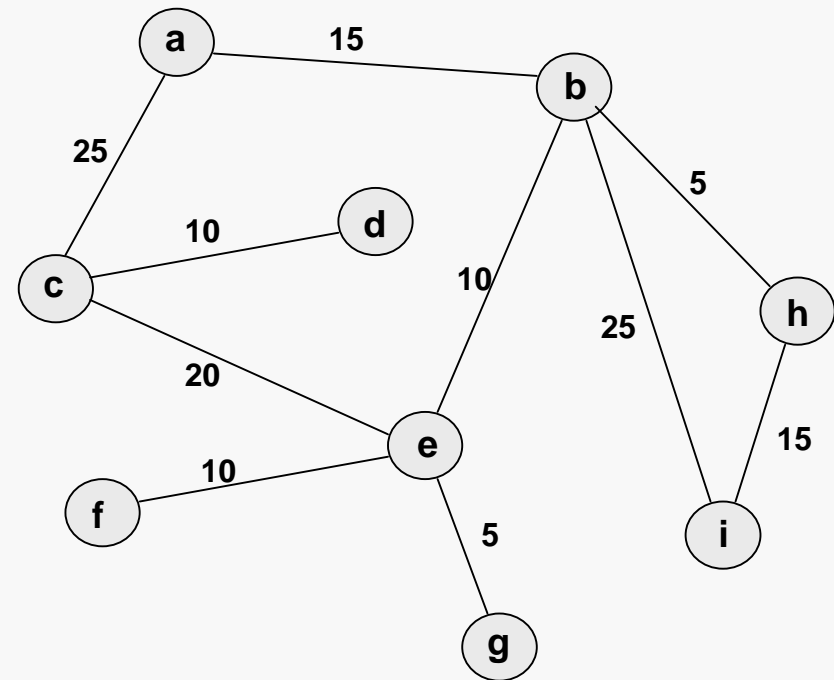
D	a	b	c	d	e	f	g	h	i
	0	15	25	∞	10	∞	∞	5	25
	Y	Y	N	N	N	N	N	N	N

Minimal Spanning Tree

Given a weighted graph, we would like to find a spanning tree for the graph that has minimal total weight.

The total weight of a spanning tree is the sum of the weights of its edges.

We want to find a spanning tree T , such that if T' is any other spanning tree for the graph then the total weight of T is less than or equal to that of T' .



By modifying Dijkstra's Algorithm to build a list of the edges that are used as vertices are added, we obtain Prim's Algorithm (R C Prim, 1957).

It turns out that this algorithm does, in fact, create a spanning tree of minimal weight if the graph to which it is applied is connected.

Since the complex steps in Prim's algorithm are the same as Dijkstra's, no detailed example is traced here.