Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 13: Graphs

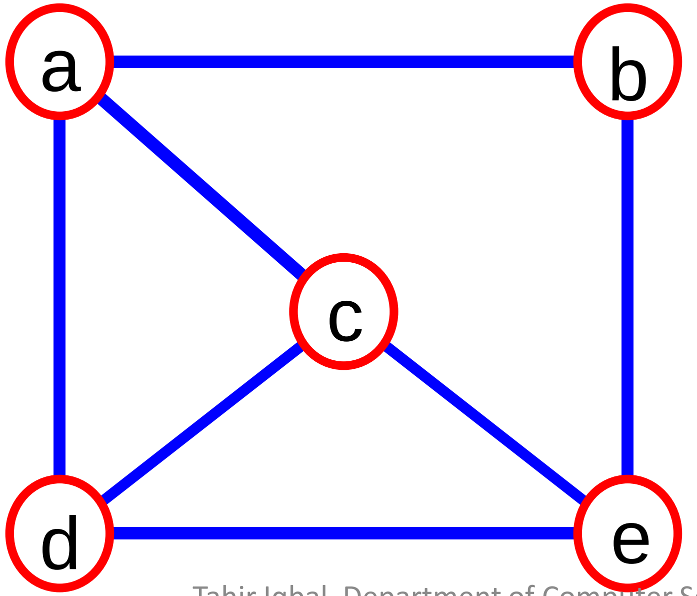
What is a Graph?

- **A graph consists of a number of data items, each of which is called a vertex. Any vertex may be connected to any other, these connections are called edges.**
- **A graph G = (V,E) is composed of:**

  V: set of vertices

  E: set of edges connecting the vertices in V

- **An edge e = (u,v) is a pair of vertices**
- **Example:**

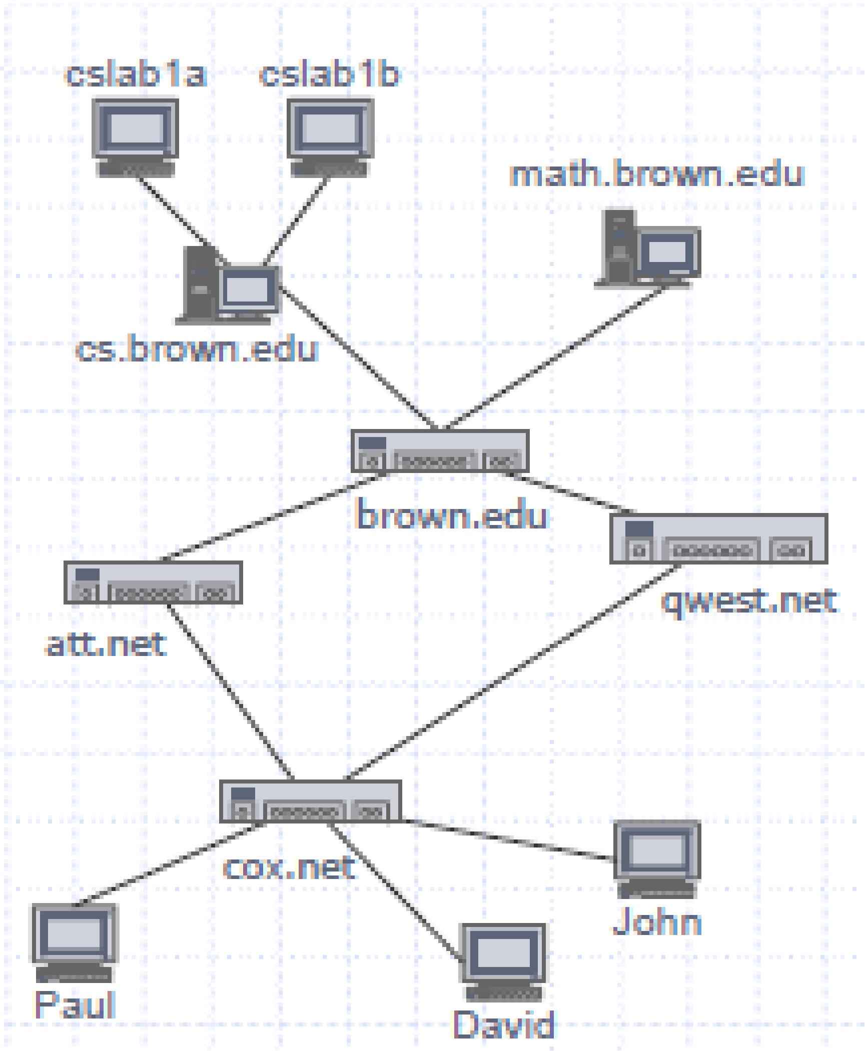


V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)}

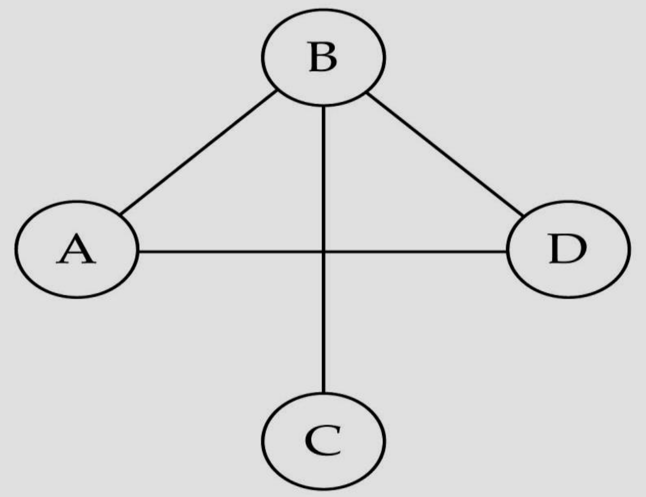
Applications

- **Electronic circuits**
  - **Printed circuit board**
  - **Integrated circuit**
- **Transportation networks**
  - **Highway network**
  - **Flight network**
- **Computer networks**
  - **Local area network**
  - **Wide area network**
  - **Internet**
- **Databases**
  - **Entity-relationship diagram**



Directed and Undirected Graph

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

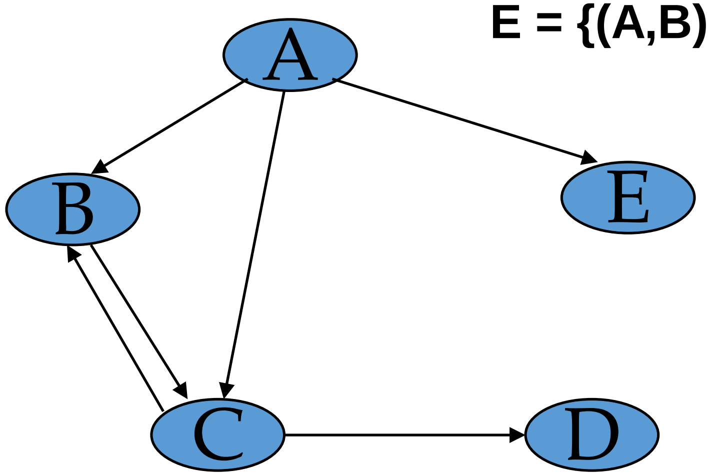


$V(\text{Graph1}) = \{ A, B, C, D \}$
$E(\text{Graph1}) = \{ (A, B), (A, D), (B, C), (B, D) \}$

Directed and Undirected Graph

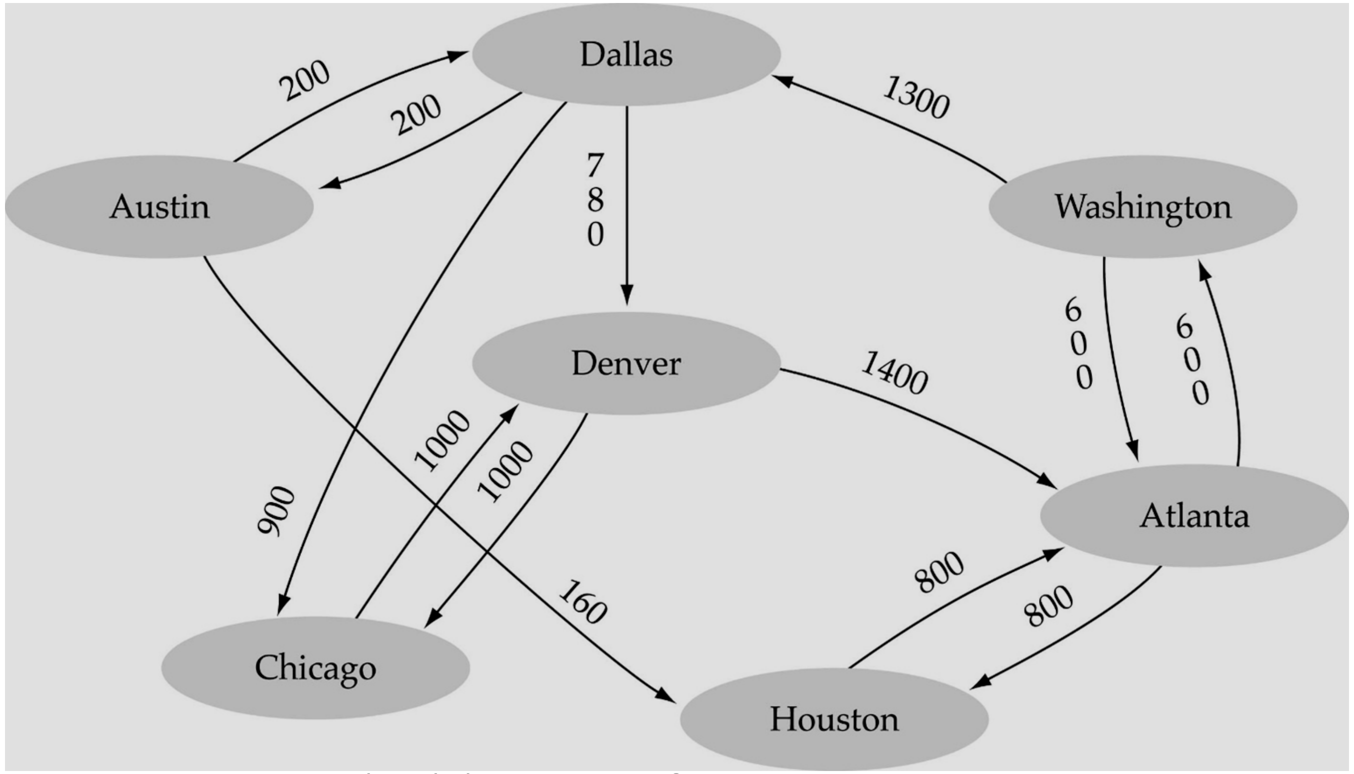
- **When the edges in a graph have a direction, the graph is called *directed graph* .**
- **Also known as *digraph.***
- **This kind of graph contains ordered pair of vertices i.e. if the graph is directed, the order of the vertices in each edge is important.**

V = {A, B, C, D, E}
E = {(A,B), (B,C), (C,B), (A,C), (A,E), (C,D)}



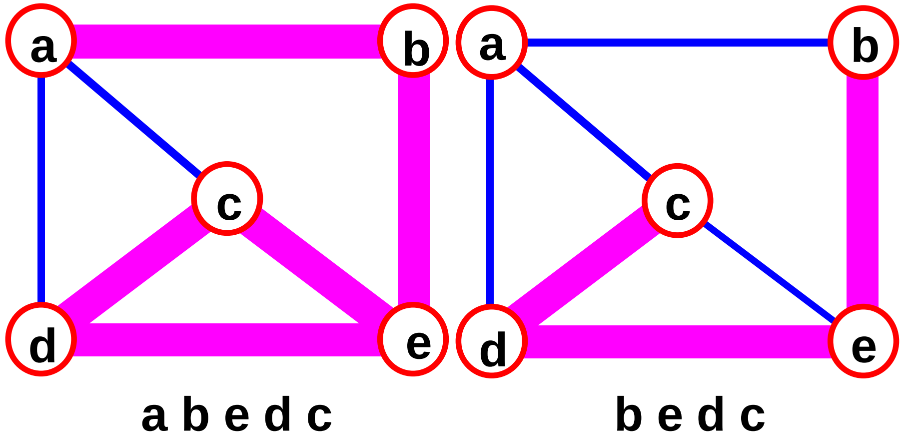
Graph Terminologies

- **Weighted Graph**
  - **A graph is suppose to be weighted if its every edge is assigned some value which is greater than or equal to zero.**



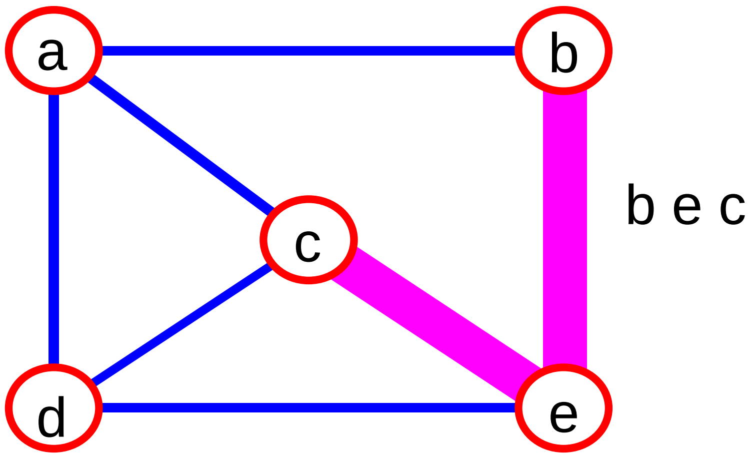
Graph Terminologies

- **Adjacent Nodes**
  - **When there is an edge from one node to another then these nodes are called adjacent nodes.**
- **Path**
  - **sequence of vertices $v_1, v_2, . . . v_k$ such that consecutive vertices v_i and v_{i+1} are adjacent.**



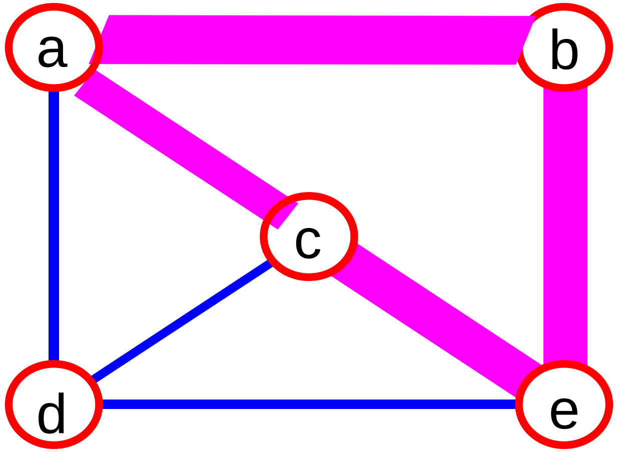
Graph Terminologies

- Length of a Path
  - Length of a path is nothing but the total number of edges included in the path from source to destination node.
- Simple Path
  - path such that all its vertices and edges are distinct.



Graph Terminologies

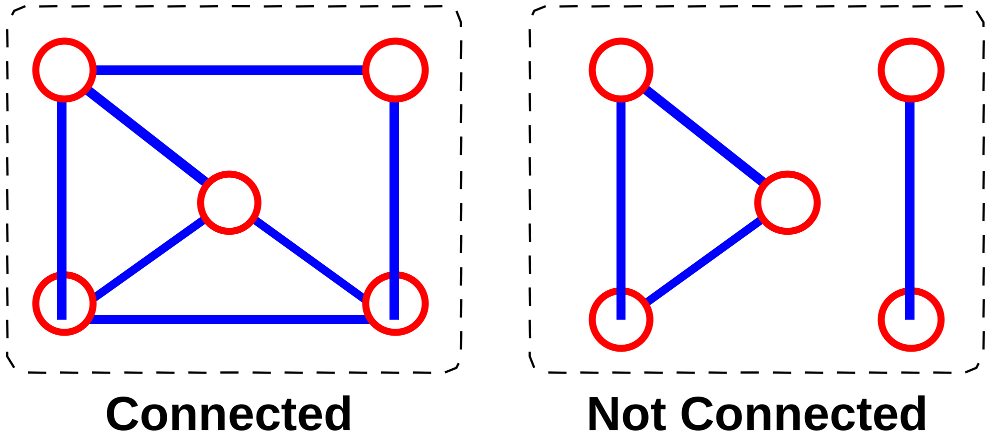
- Cycle
  - simple path, except that the last vertex is the same as the first vertex
- Acyclic Graph
  - A graph without cycle is called acyclic Graph. A tree is a good example of acyclic graph.



a b e c a

Graph Terminologies

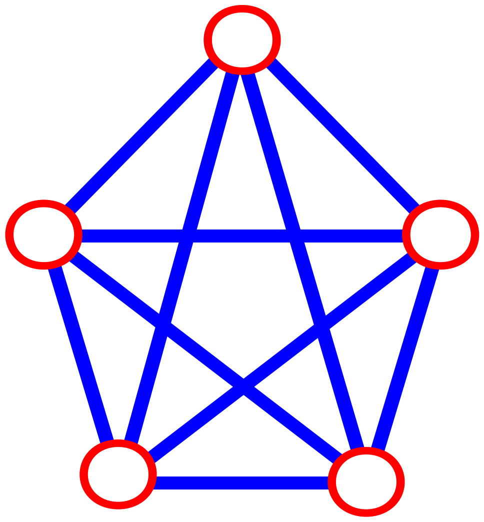
- Connected Graph
  - An undirected graph is connected if, for any pair of vertices, there is a path between them.



  - A directed graph is connected if, for any pair of vertices, there is a path between them.

Graph Terminologies

- **Complete Graph**
  - **A graph in which all pairs of vertices are adjacent OR**
  - **A graph in which every vertex is directly connected to every other vertex**
  - Let **n** = Number of vertices, and
    m = Number of edges
  - **For a complete graph with *n* vertices, the number of edges is *n(n – 1)/2*. A graph with 6 vertices needs 15 edges to be complete.**

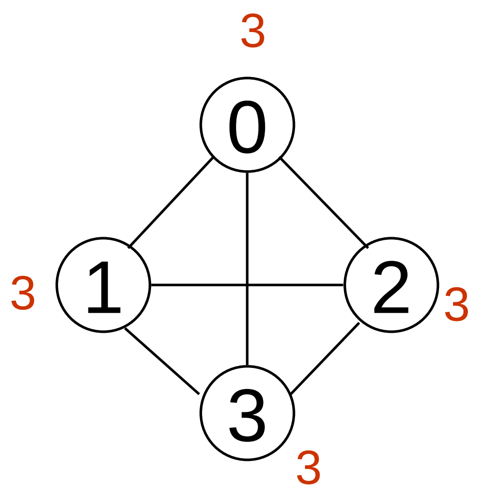


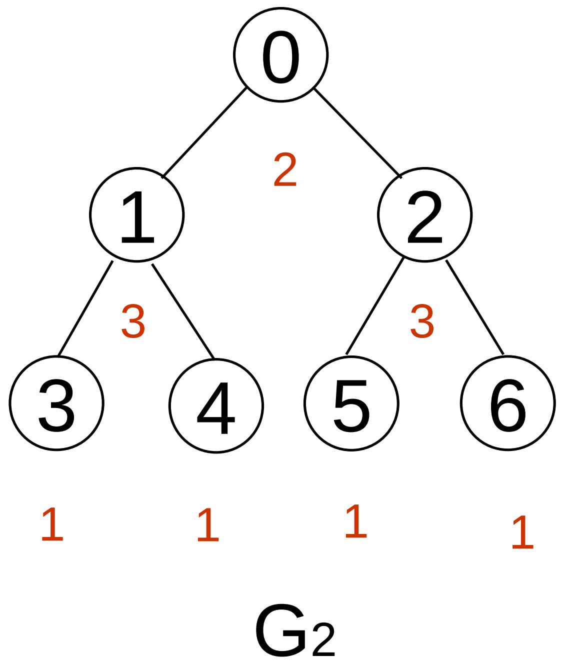
$$\mathbf{n} = 5$$
$$\mathbf{m} = (5 * 4)/2 = 10$$

Graph Terminologies

- Degree of a node
  - In an Undirected graph, the total number of edges linked to a node is called a degree of that node.
  - For directed graph there are two degree for every node
    - Indegree
      - The indegree of a node is the total number of edges coming to that node.
    - Outdegree
      - The outdegree of a node is the total number of edges going out from that node

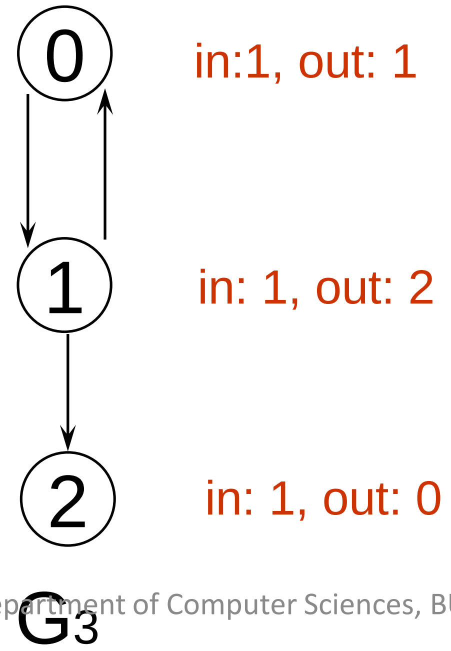
Graph Terminologies





Directed graph

in-degree
out-degree



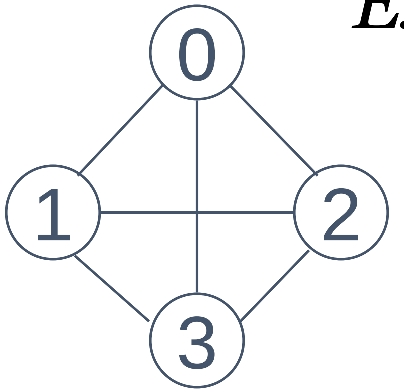
Graph Representation

- There are two methods to represent a Graph
  - Adjacency Matrix (Array Implementation)
  - Adjacency List (Linked List Implementation)

Adjacency Matrix

- Let G=(V,E) be a graph with n vertices.
- The adjacency matrix of G is a two-dimensional n by n array, say adj_mat
- If the edge (vi, vj) is in E(G), adj_mat[i][j]=1
- If there is no such edge in E(G), adj_mat[i][j]=0
- The adjacency matrix for an undirected graph is symmetric; since adj_mat[i][j]=adj_mat[j][i]
- The adjacency matrix for a digraph may not be symmetric

Examples for Adjacency Matrix

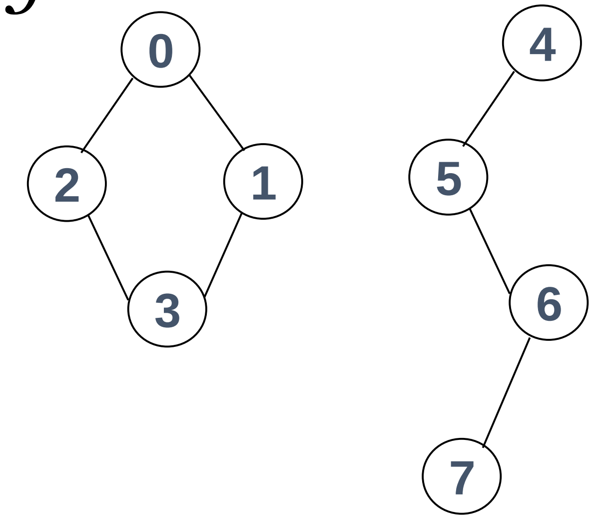


$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

G_1

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

G_2



$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

G_4

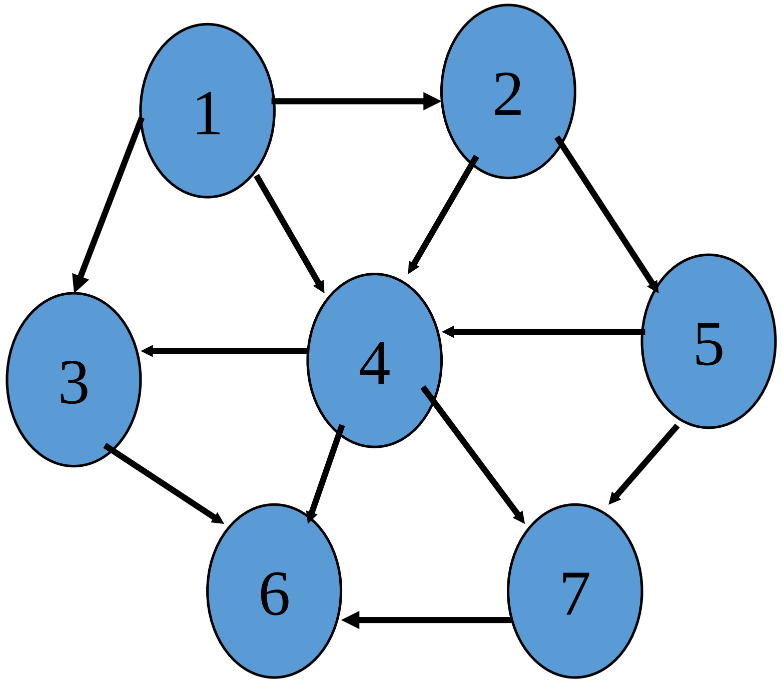
Adjacency Matrix

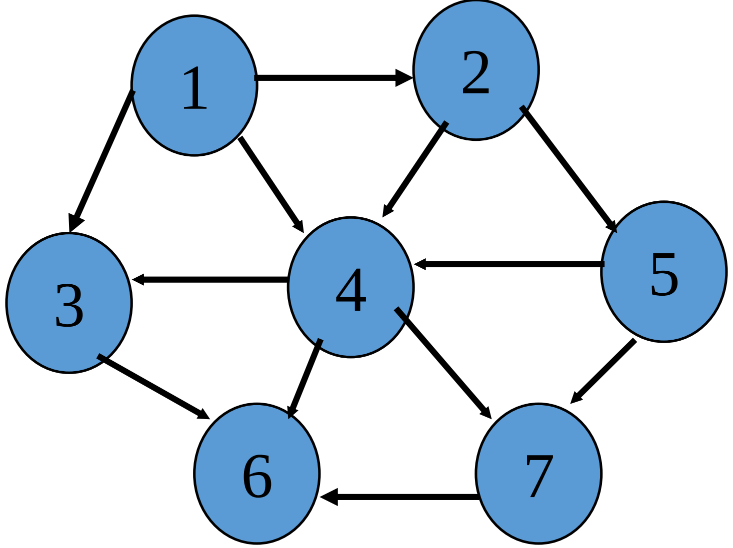
- From the adjacency matrix, to determine the connection of vertices is easy
- The degree of a vertex is

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

- We can also represent weighted graph with Adjacency Matrix. Now the contents of the martix will not be 0 and 1 but the value is substituted with the corresponding weight.

Another Example
Adjacency Matrix?



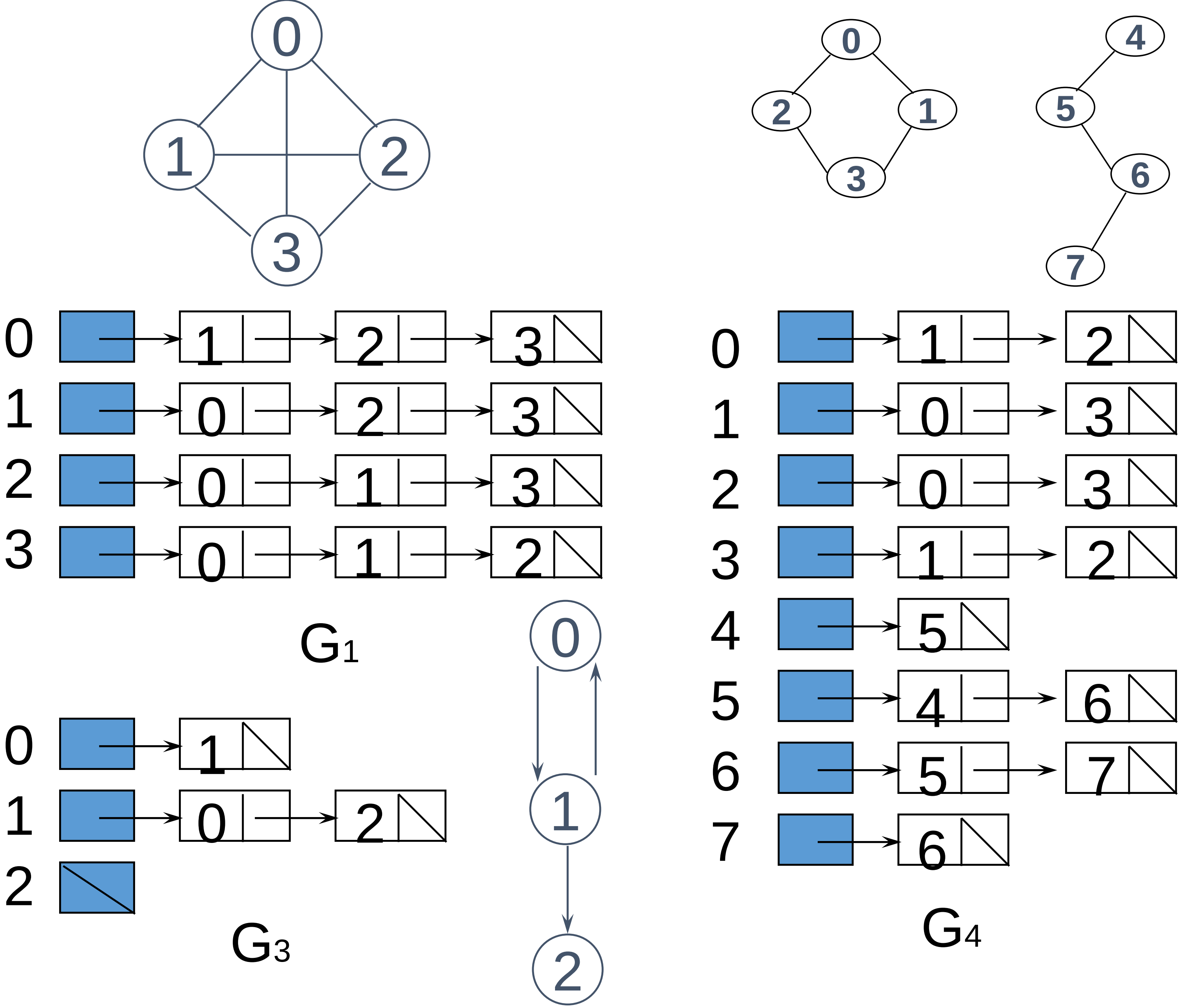


	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	1
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

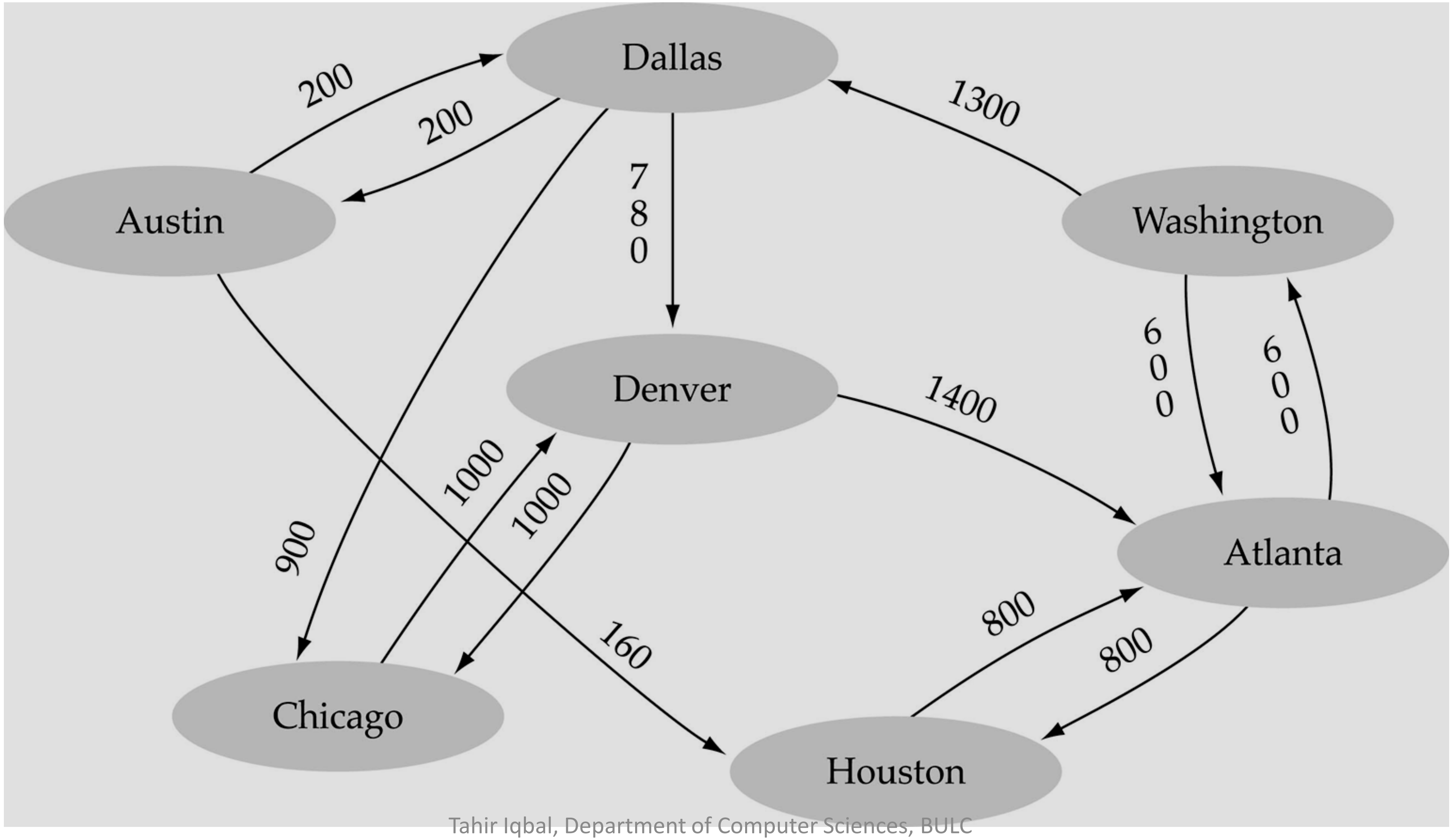
Adjacency List

- **A Single Dimension array of Structure/List is used to represent the vertices**

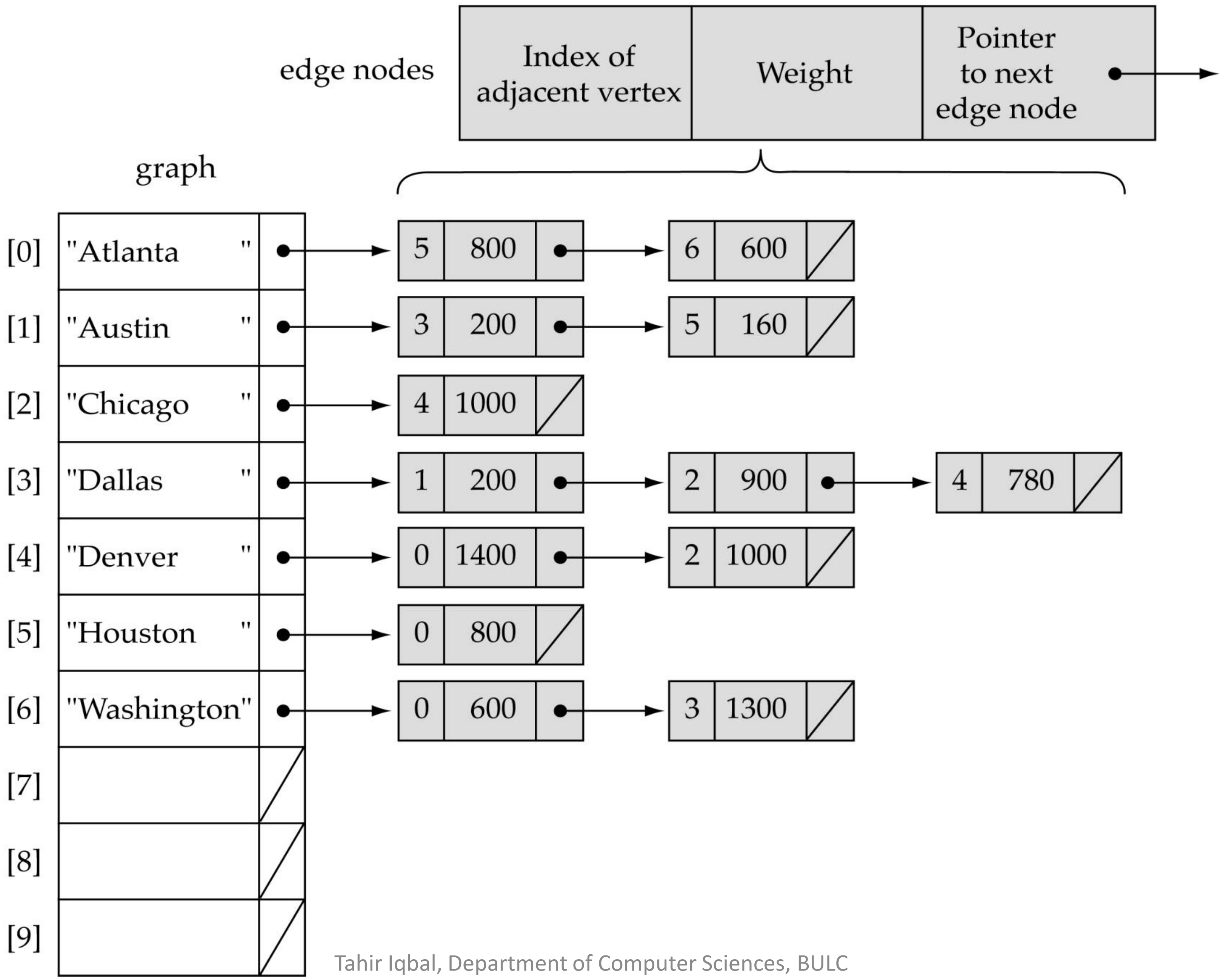
- **A Linked list is used for each vertex V which contains the vertices which are adjacent from V (adjacency list)**


G_1
G_3
G_4

Another Example of Adjacency List

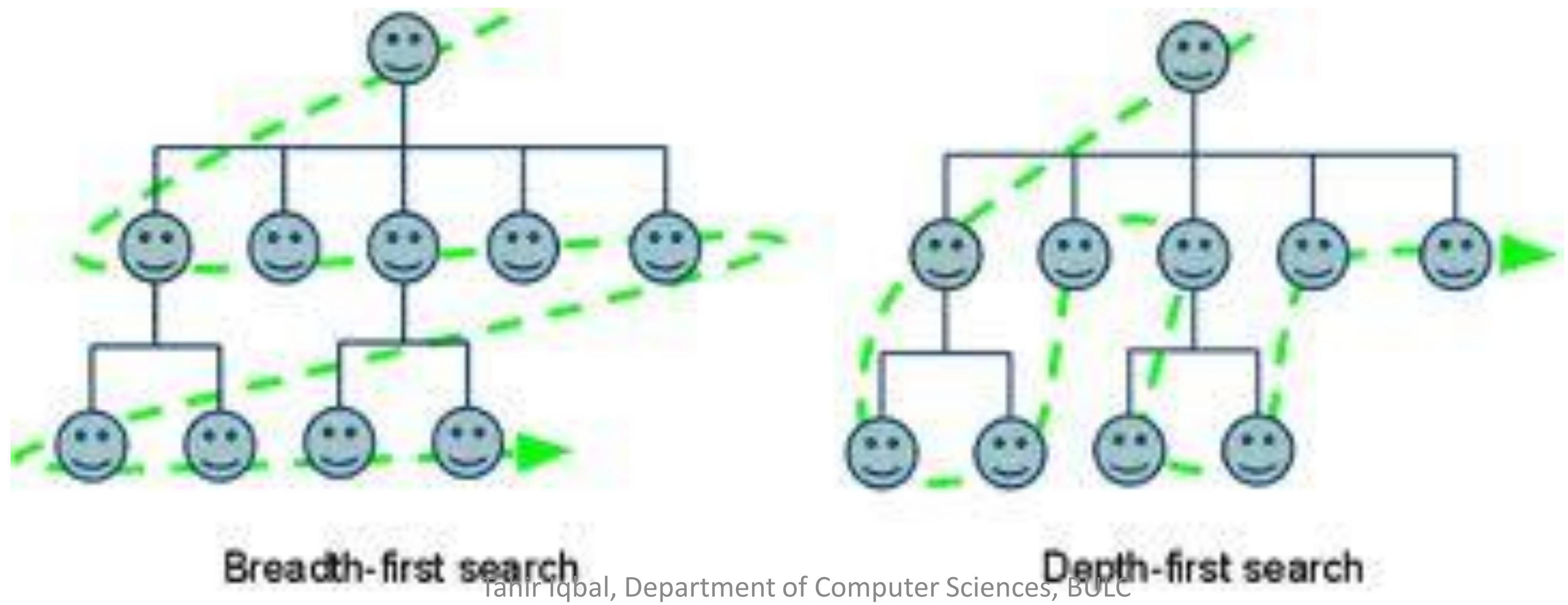


(a)


edge nodes
Index of adjacent vertex
Weight
Pointer to next edge node
graph
[0] "Atlanta "
[1] "Austin "
[2] "Chicago "
[3] "Dallas "
[4] "Denver "
[5] "Houston "
[6] "Washington"
[7]
[8]
[9]
5 800
6 600
3 200
5 160
4 1000
1 200
2 900
4 780
0 1400
2 1000
0 800
0 600
3 1300

Graph Traversal

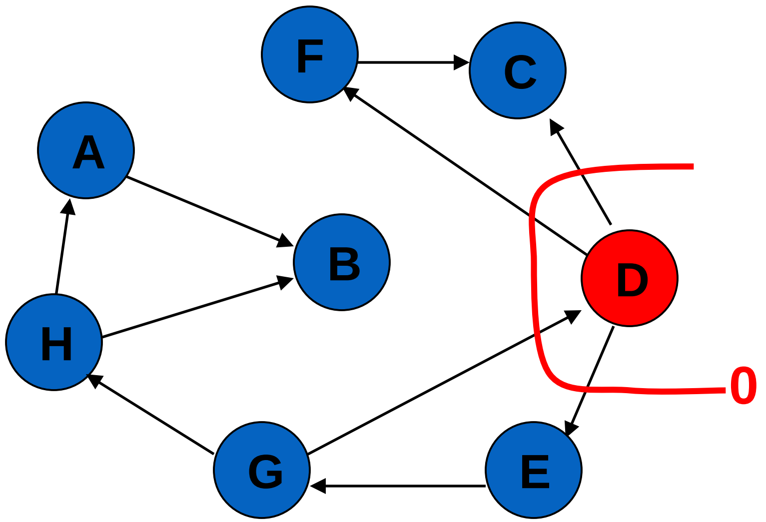
- **Traversal is the facility to move through a structure visiting each of the vertices once.**
- **We looked previously at the ways in which a binary tree can be traversed. Two of the traversal methods for a graph are breadth-first and depth-first.**



Breadth-First Graph Traversal

- This method visits all the vertices, beginning with a specified **start vertex**. It can be described roughly as “neighbours-first”.
- No vertex is visited more than once, and **vertices are visited only if they can be reached** – that is, if there is a path from the start vertex.
- Breadth-first traversal makes use of a **queue data structure**. The queue holds a list of vertices which have not been visited yet but which should be visited soon.
- Since a queue is a first-in first-out structure, vertices are visited in the order in which they are added to the queue.
- Visiting a vertex involves, for example, outputting the data stored in that vertex, and also **adding its neighbours to the queue**.
- Neighbours are not added to the queue if they are already in the queue, or have already been visited.

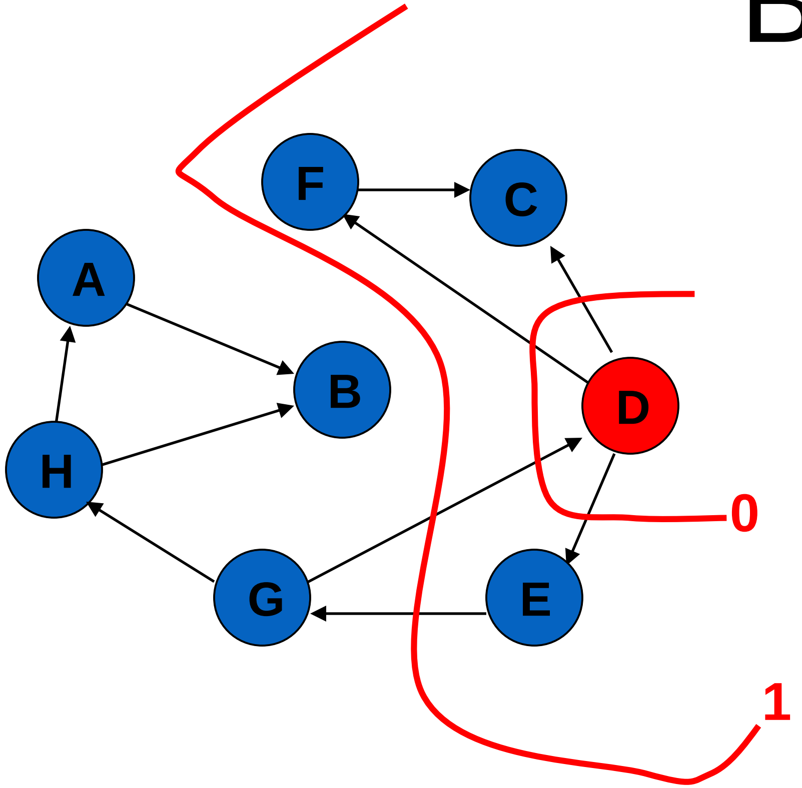
BFS



Breadth-first search starts with given node

Task: Conduct a breadth-first search of the graph starting with node D

BFS



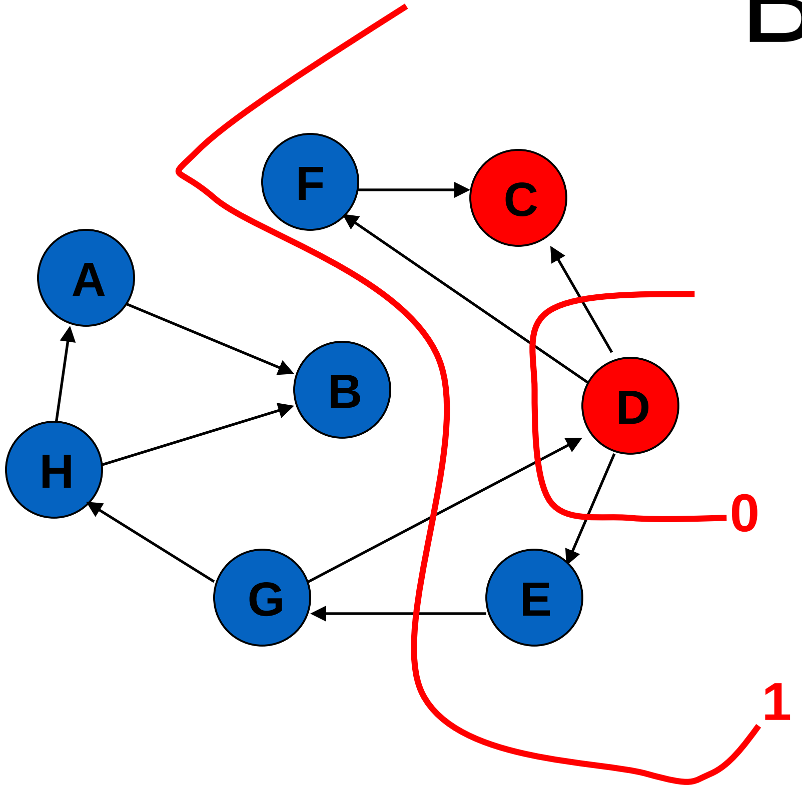
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D

BFS



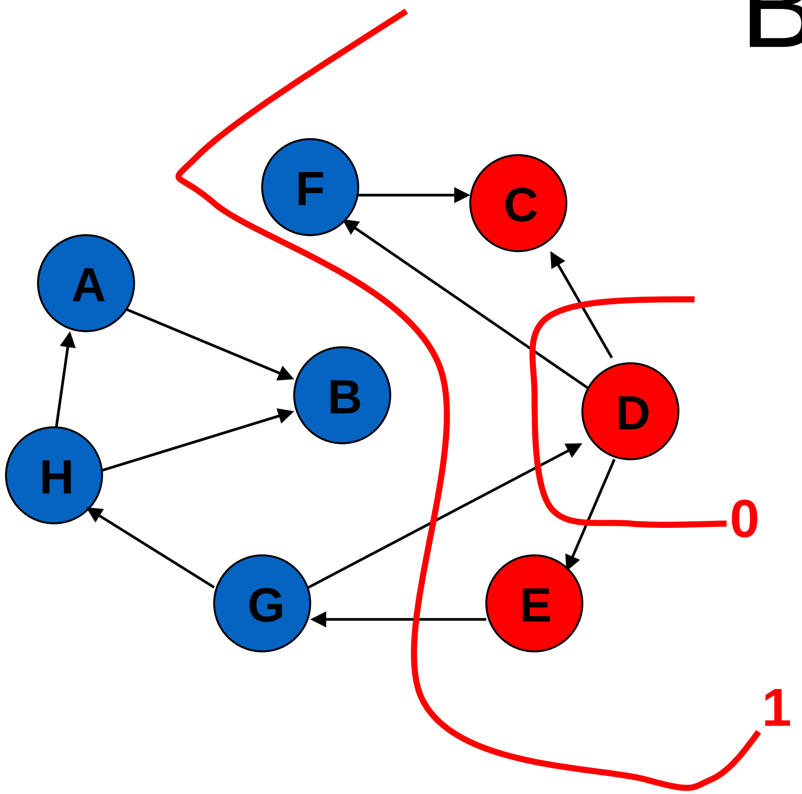
Nodes visited: D, C

Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

BFS



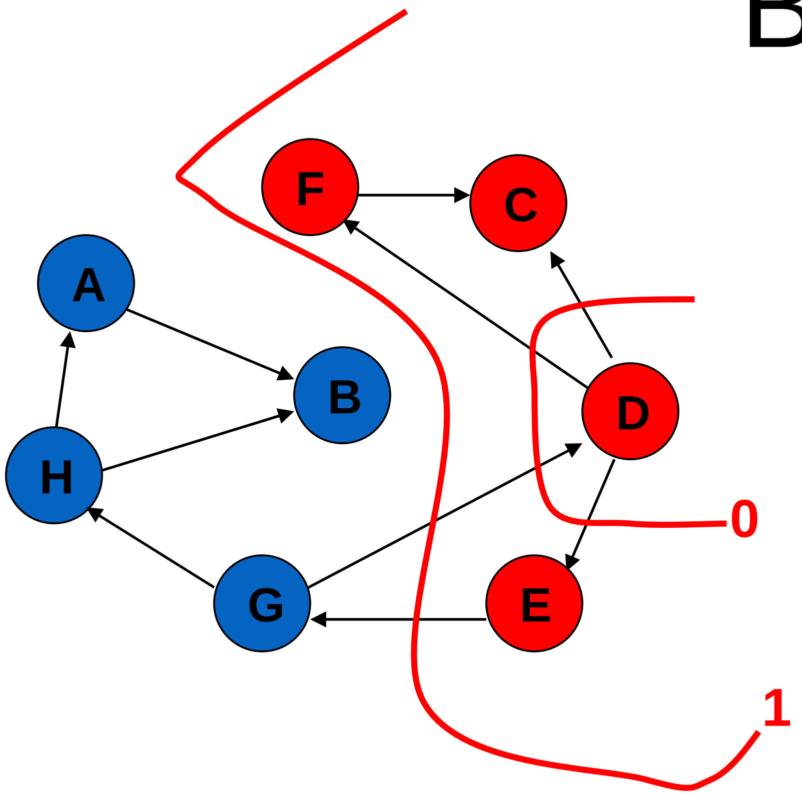
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E

BFS



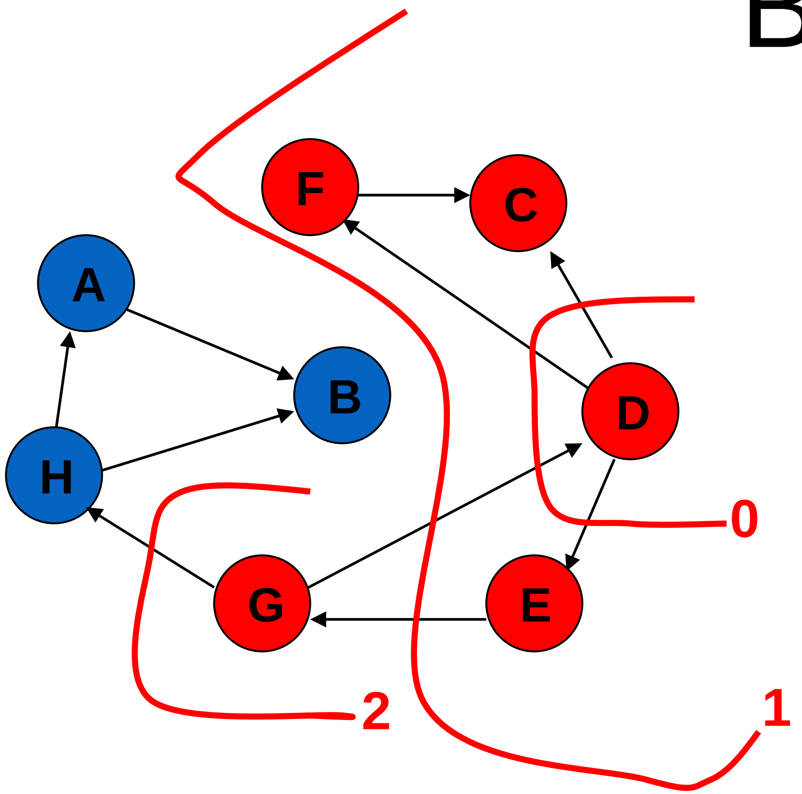
Breadth-first search starts with given node

Then visits nodes adjacent in some specified order (e.g., alphabetical)

Like ripples in a pond

Nodes visited: D, C, E, F

BFS

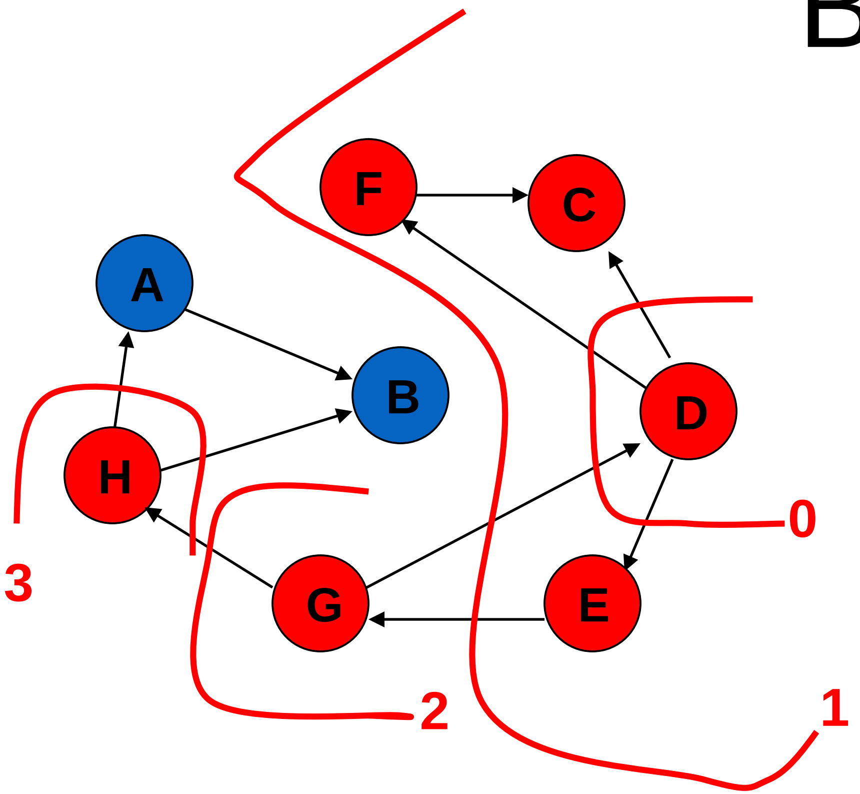


When all nodes in ripple are visited,
visit nodes in next ripples

Nodes visited: D, C, E, F, G

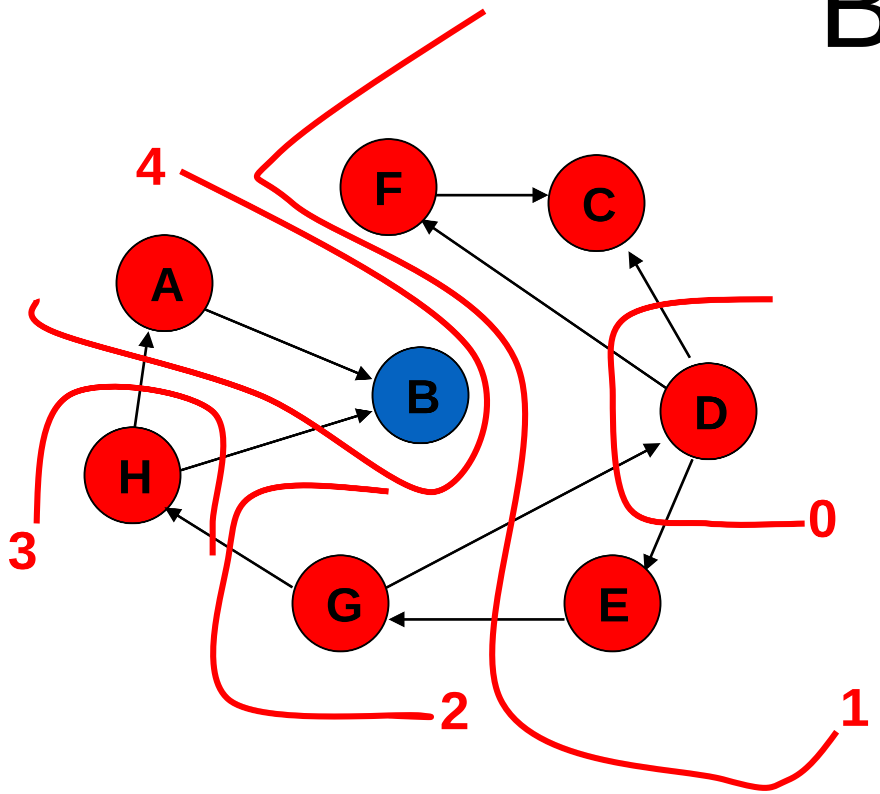
BFS

When all nodes in ripple are visited, visit nodes in next ripples



Nodes visited: D, C, E, F, G, H

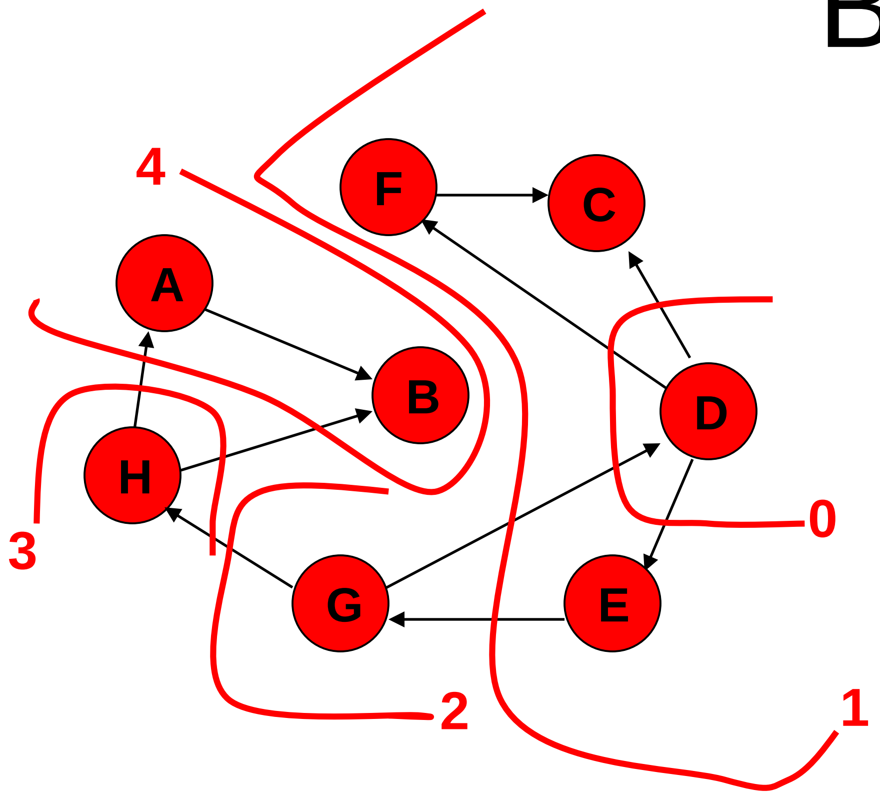
BFS



When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H, A

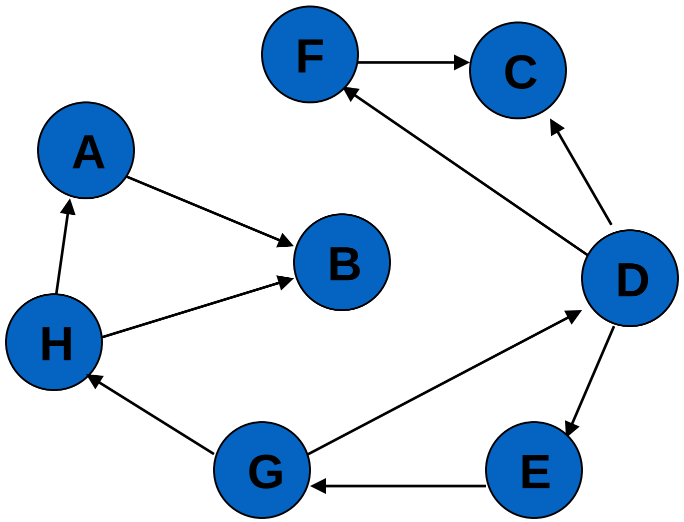
BFS



When all nodes in ripple are visited, visit nodes in next ripples

Nodes visited: D, C, E, F, G, H, A, B

BFS

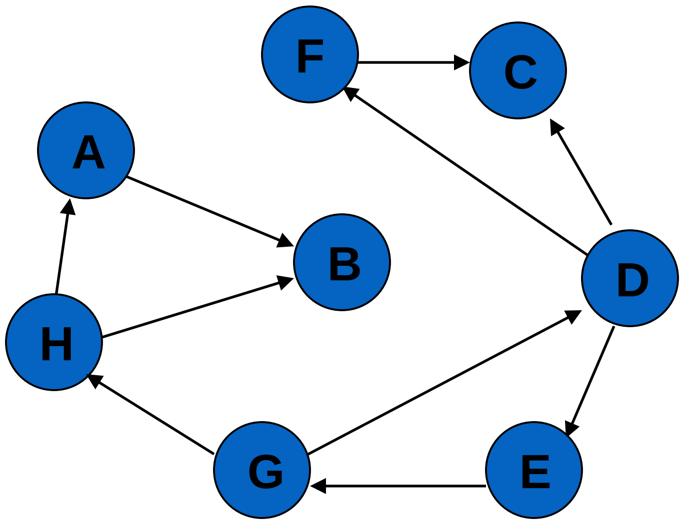


A	
B	
C	
D	
E	
F	
G	
H	

Q →

How is this accomplished? Simply replace the stack with a queue! Rules: (1) Maintain an *enqueued* array. (2) Visit node when *dequeued*.

BFS



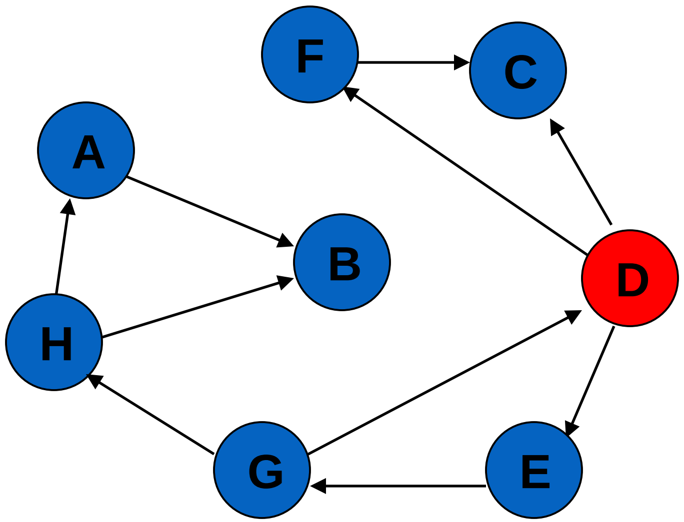
A	
B	
C	
D	√
E	
F	
G	
H	

Q → D

Nodes visited:

Enqueue D. Notice, D not yet visited.

BFS



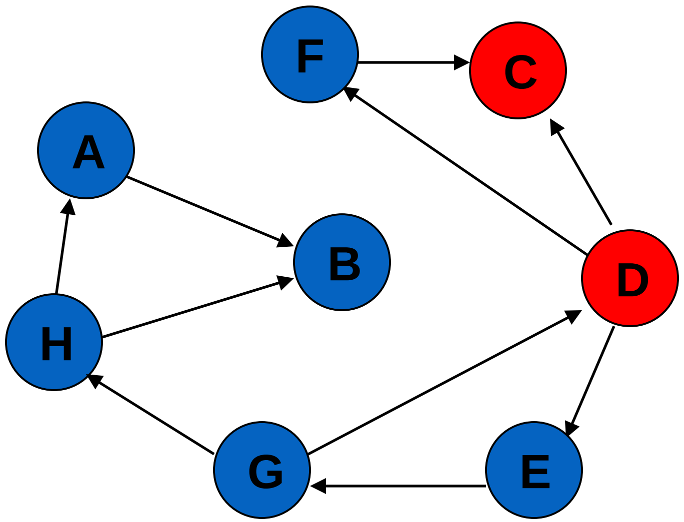
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → C → E → F

Nodes visited: D

Dequeue D. Visit D. Enqueue unenqueued nodes adjacent to D.

BFS



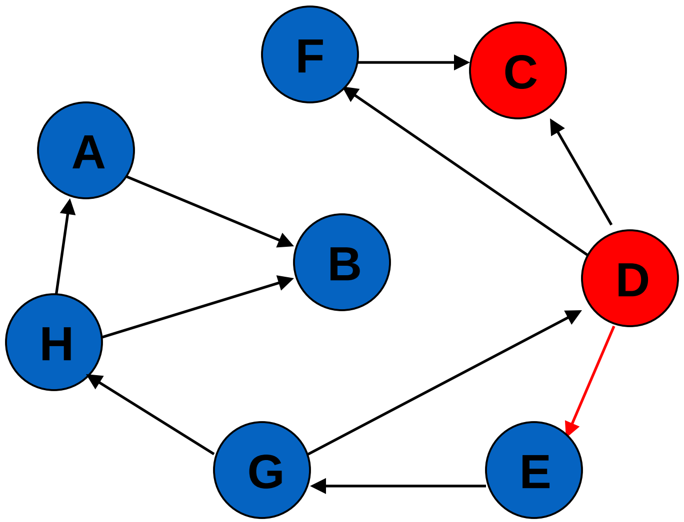
A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → E → F

Nodes visited: D, C

Dequeue C. Visit C. Enqueue unenqueued nodes adjacent to C.

BFS



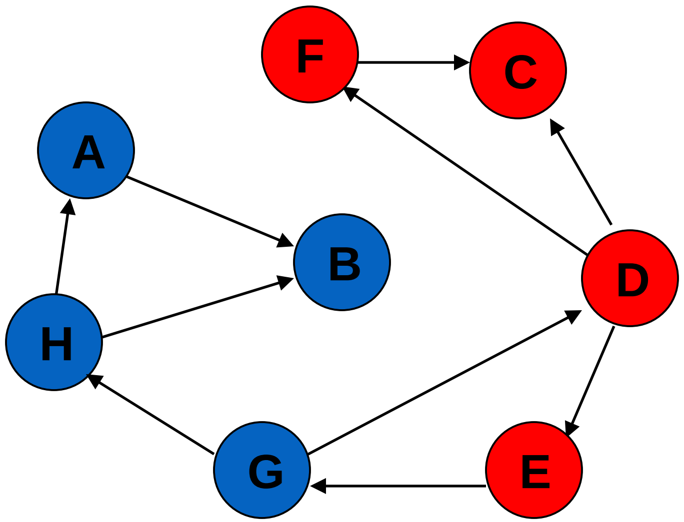
Nodes visited: D, C, E

A	
B	
C	√
D	√
E	√
F	√
G	
H	

Q → F → G

Dequeue E. Visit E. Enqueue unenqueued nodes adjacent to E.

BFS



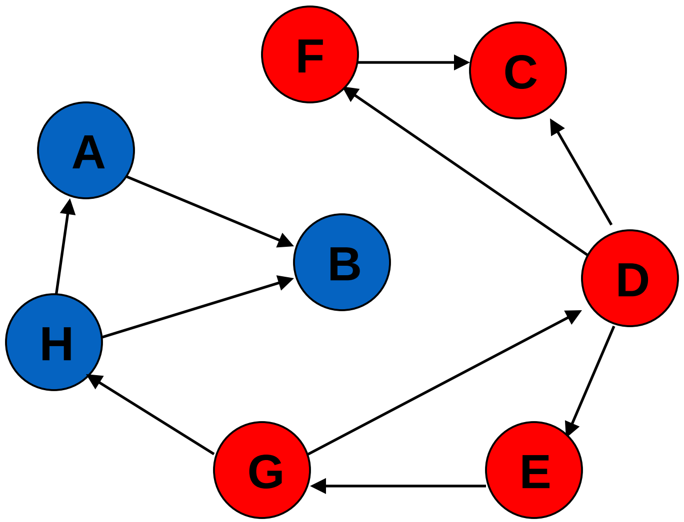
A	
B	
C	√
D	√
E	√
F	√
G	√
H	

Q → G

Nodes visited: D, C, E, F

Dequeue F. Visit F. Enqueue unenqueued nodes adjacent to F.

BFS



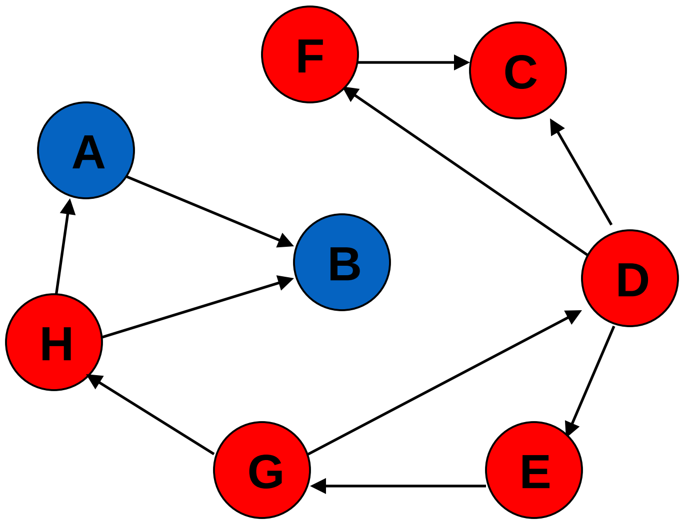
A	
B	
C	√
D	√
E	√
F	√
G	√
H	√

Q → H

Nodes visited: D, C, E, F, G

Dequeue G. Visit G. Enqueue unenqueued nodes adjacent to G.

BFS



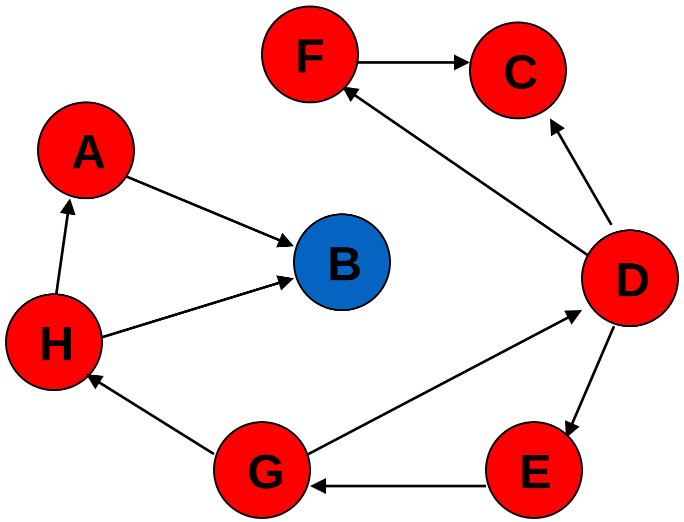
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q → A → B

Nodes visited: D, C, E, F, G, H

Dequeue H. Visit H. Enqueue unenqueued nodes adjacent to H.

BFS



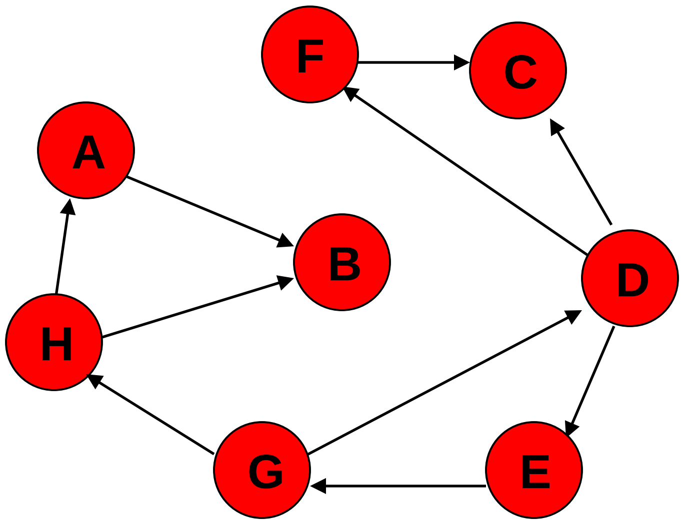
Nodes visited: D, C, E, F, G, H, A

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q → B

Dequeue A. Visit A. Enqueue unenqueued nodes adjacent to A.

BFS



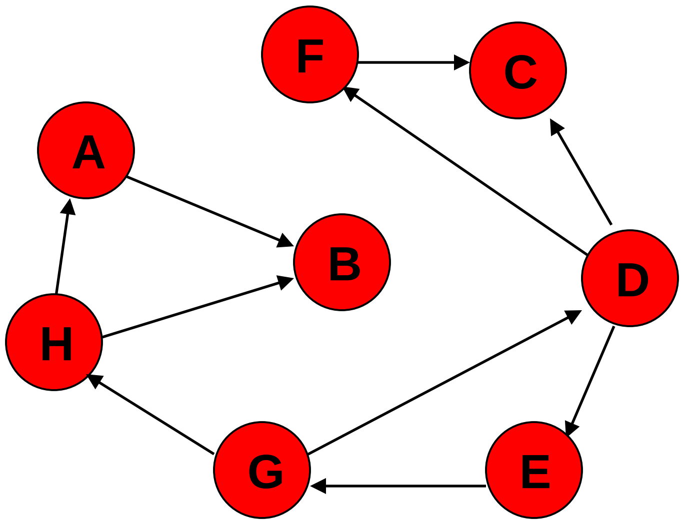
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

Q empty

Nodes visited: D, C, E, F, G, H, A, B

Dequeue B. Visit B. Enqueue unenqueued nodes adjacent to B.

BFS



A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

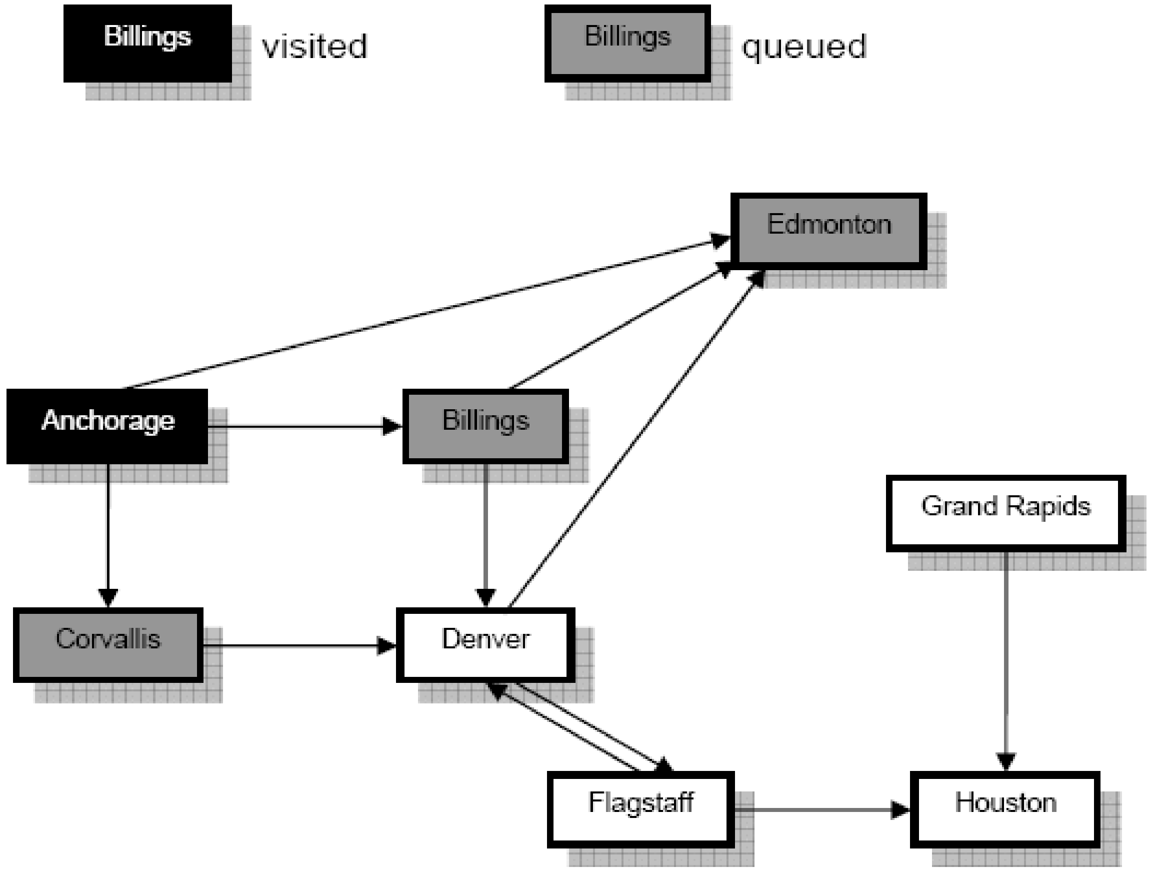
Q empty

Nodes visited: D, C, E, F, G, H, A, B

Q empty. Algorithm done.

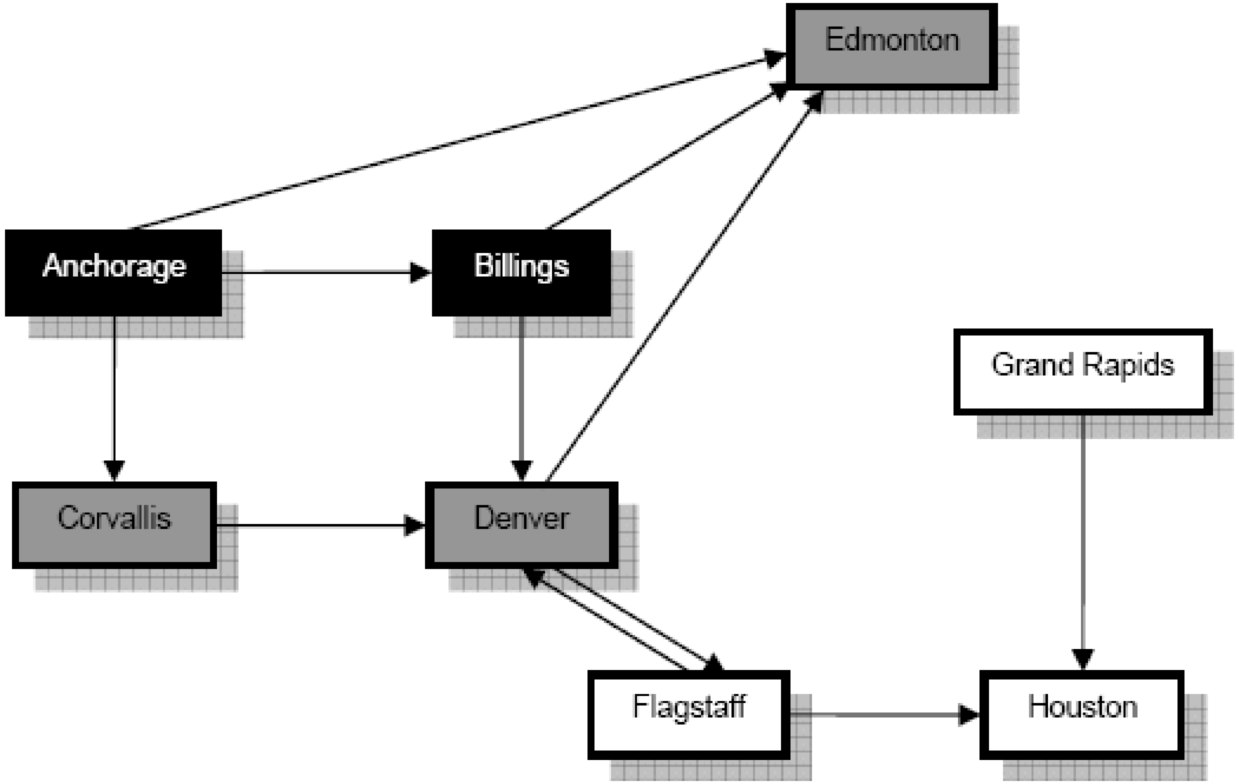
Breadth-First Traversal Another Example

- Neighbours are added to the queue in alphabetical order. Visited and queued vertices are shown as follows:



Visited: *Anchorage*
Queue: *Billings, Corvallis, Edmonton* Visit *Billings* next

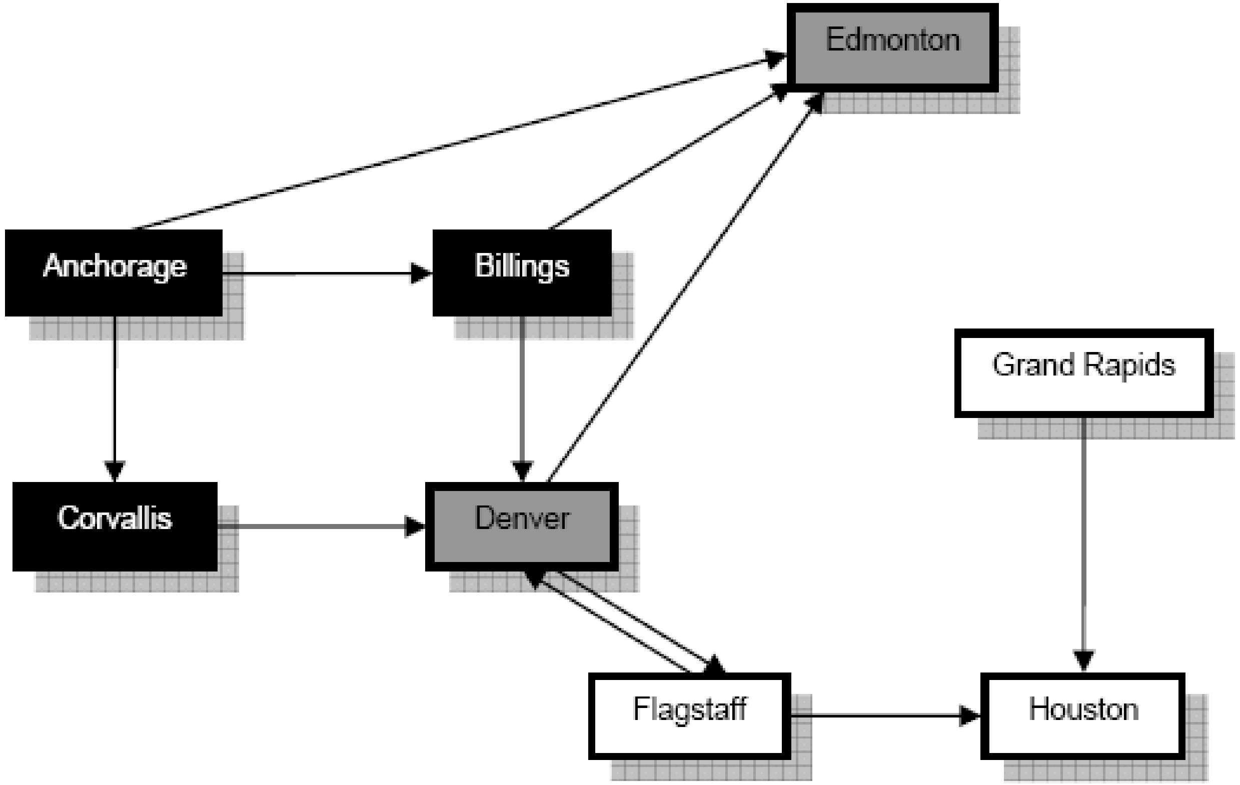
Breadth-First Graph Traversal



Visited: *Anchorage, Billings*
Queue: *Corvallis, Edmonton, Denver* visit *Corvallis* next

- **Note** that we only add Denver to the queue as the other neighbours of Billings are already in the queue.

Breadth-First Graph Traversal

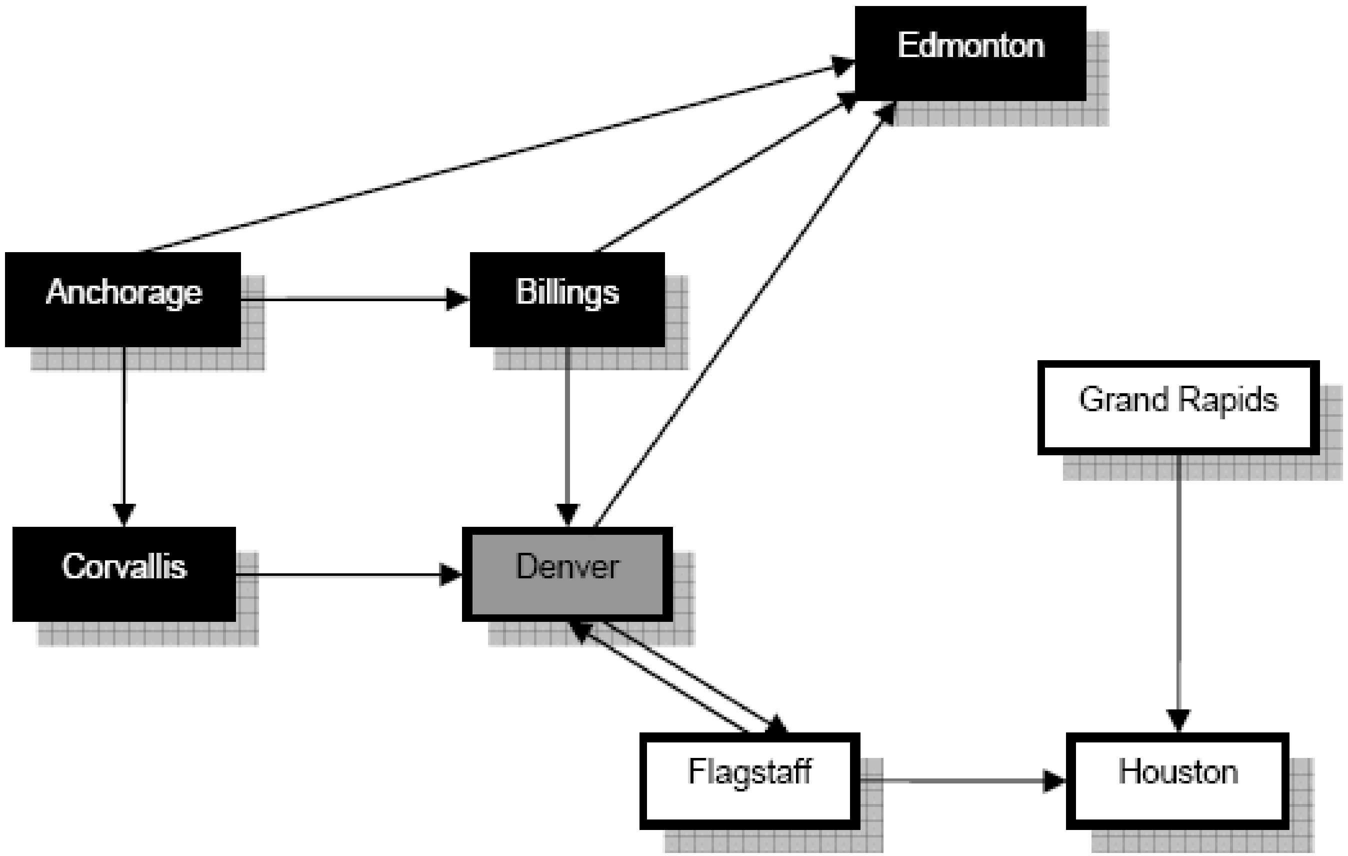


Visited: *Anchorage, Billings, Corvallis*
Queue: *Edmonton, Denver*

visit *Edmonton* next

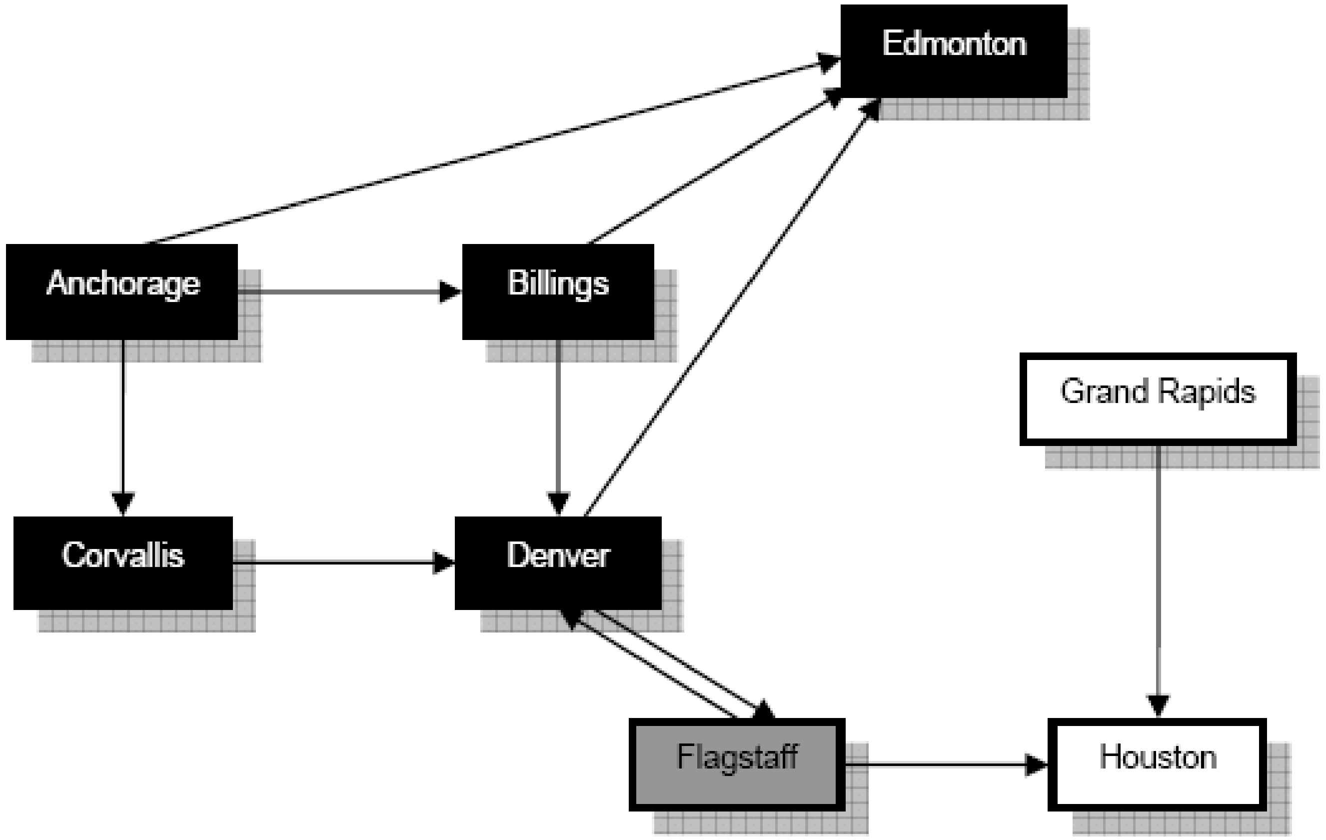
- **Note** that nothing is added to the queue as Denver, the only neighbour of Corvallis, is already in the queue.

Breadth-First Graph Traversal



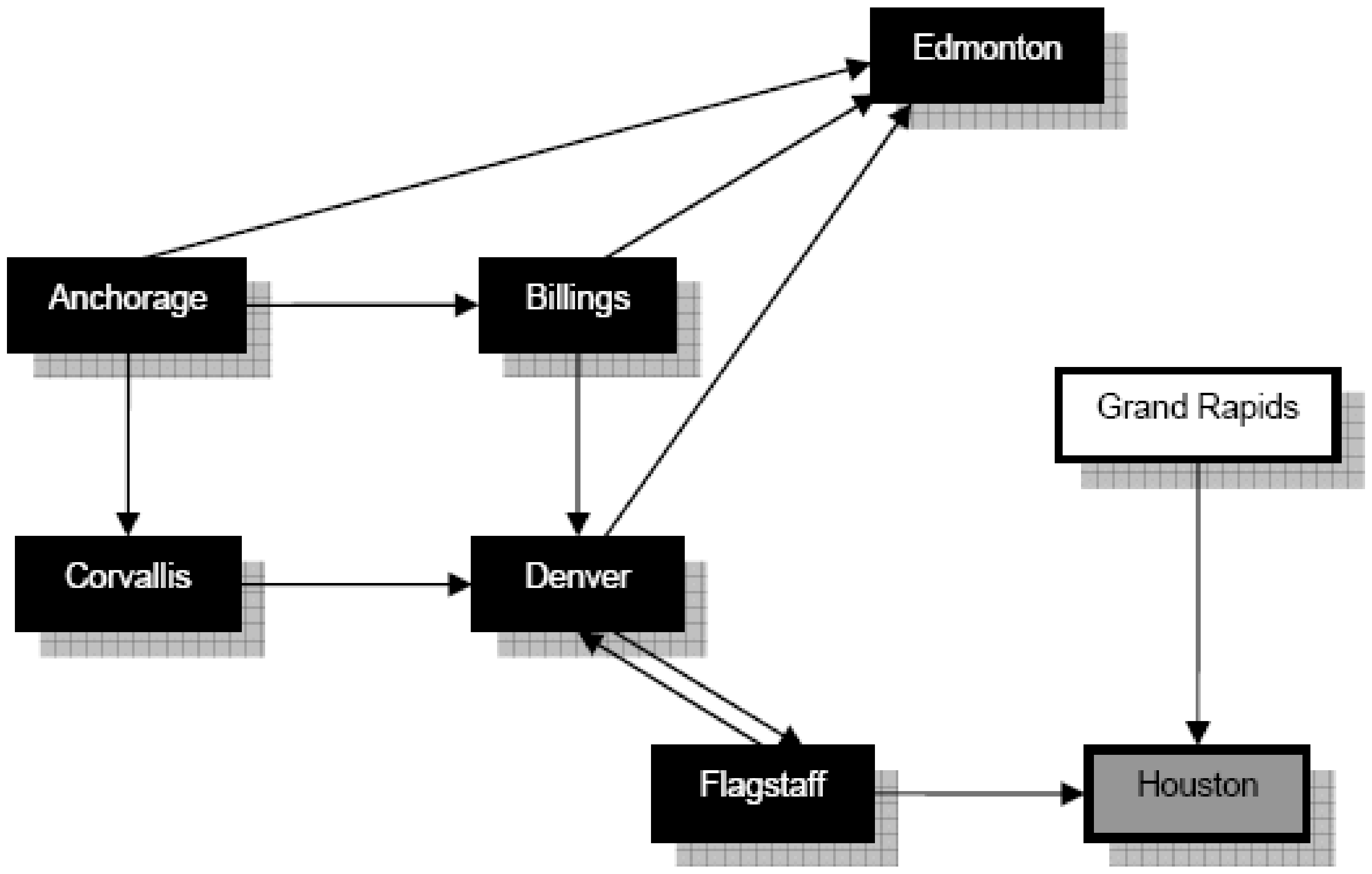
Visited: *Anchorage, Billings, Corvallis, Edmonton*
Queue: *Denver* visit *Denver* next

Breadth-First Graph Traversal



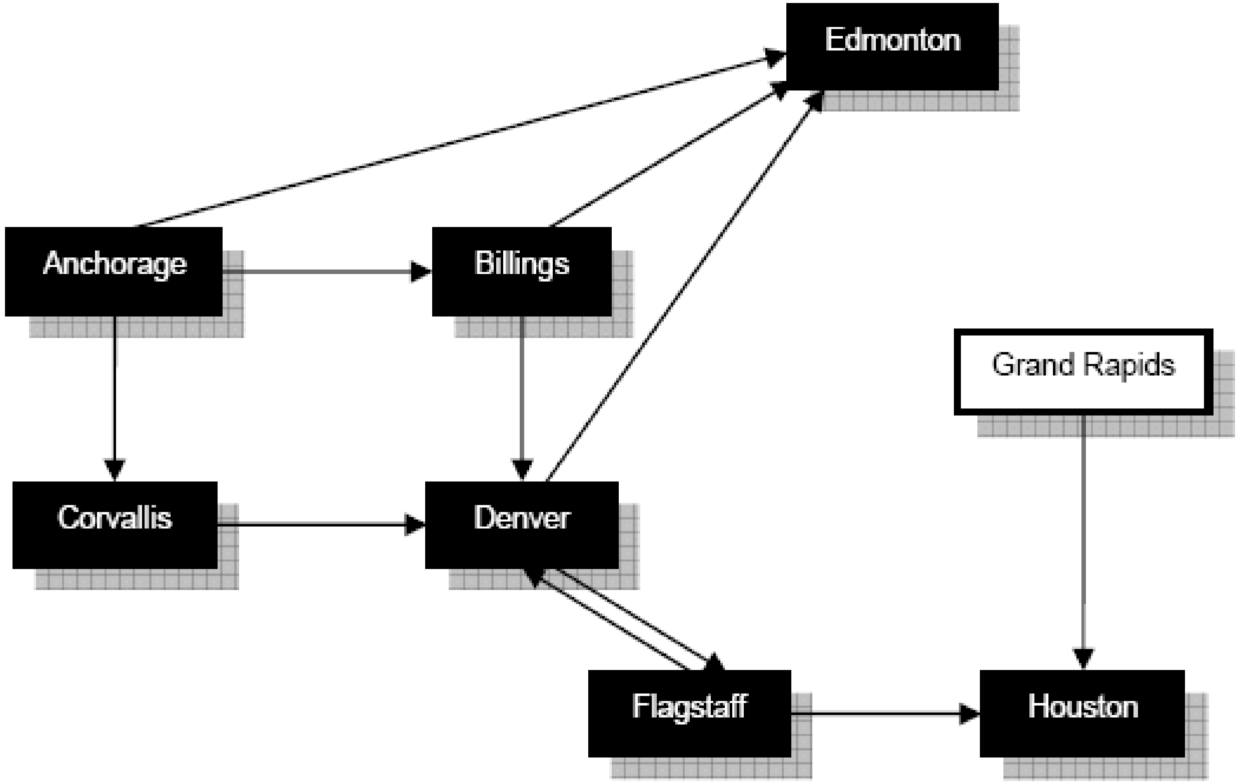
Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver*
Queue: *Flagstaff* visit *Flagstaff* next

Breadth-First Graph Traversal



Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff*
Queue: *Houston* visit *Houston* next

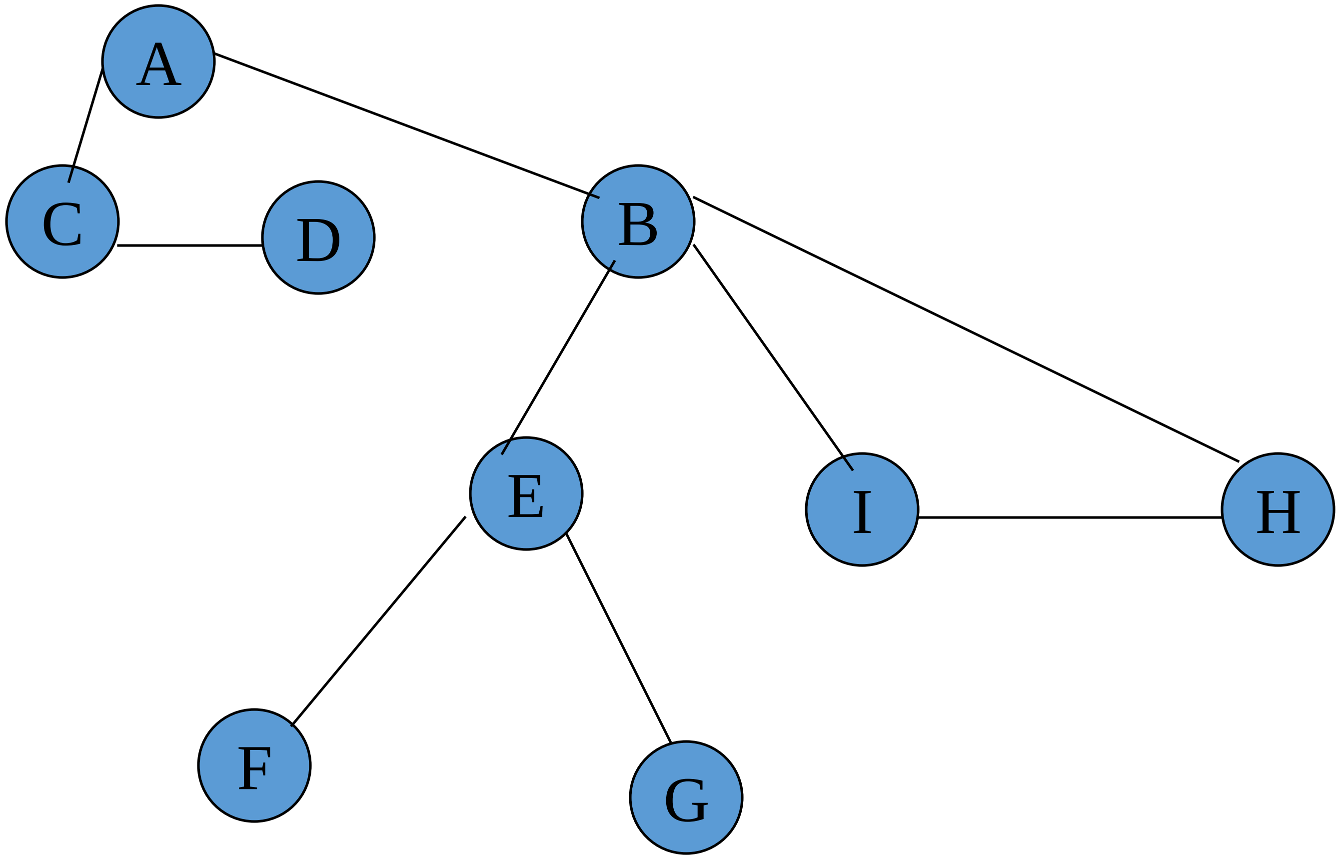
Breadth-First Graph Traversal



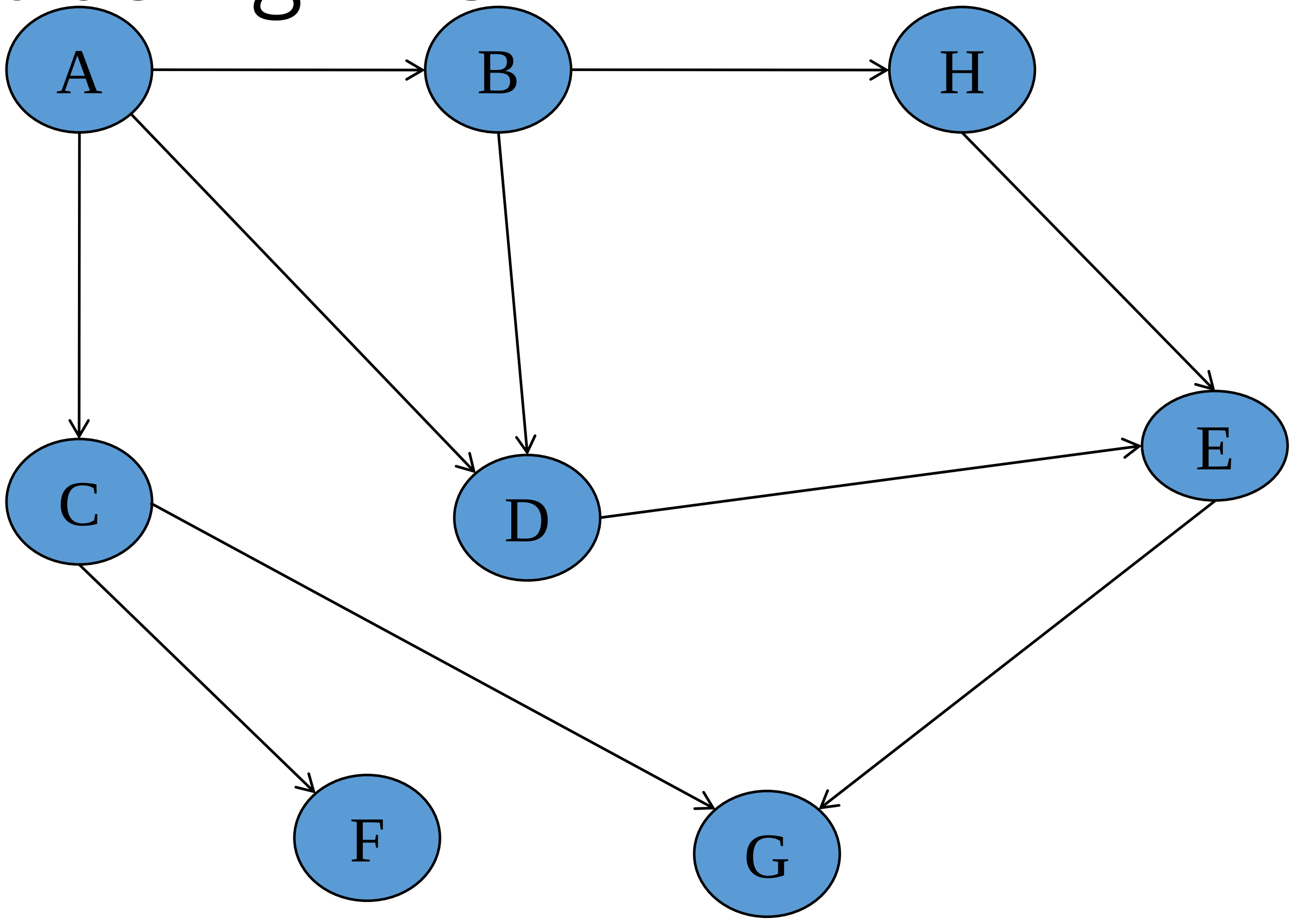
Visited: *Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff, Houston*
Queue: *empty*

- **Note** that Grand Rapids was not added to the queue as there is no path from Houston because of the edge direction.
- Since the queue is empty, we must stop, so the traversal is complete.
- The order of traversal was:*Anchorage, Billings, Corvallis, Edmonton, Denver, Flagstaff, Houston*

Exercise
Solve it using BFS



Exercise 2
Solve it using BFS



Depth-First Search (DFS)

Depth-First Search

1. From the given vertex, visit one of its adjacent vertices and leave others;
2. Then visit one of the adjacent vertices of the previous vertex;
3. Continue the process, visit the graph as deep as possible until:
    - A visited vertex is reached;
    - An end vertex is reached.

Depth-First Traversal

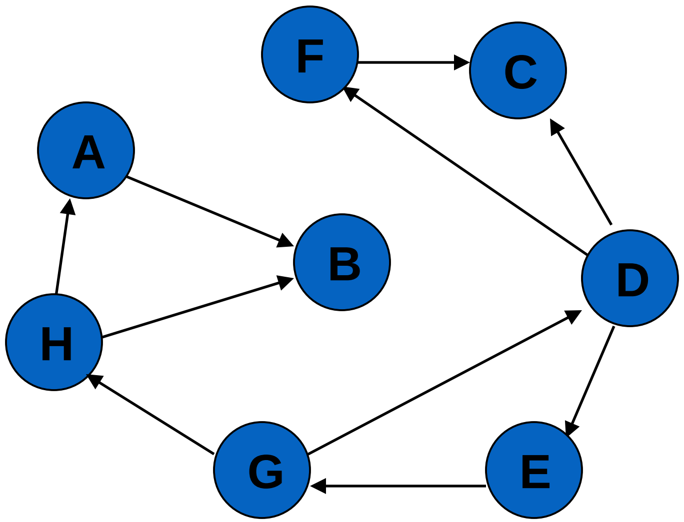
1. Depth-first traversal of a graph:
2. Start the traversal from an arbitrary vertex;
3. Apply depth-first search;
4. When the search terminates, backtrack to the previous vertex of the finishing point,
5. Repeat depth-first search on other adjacent vertices, then backtrack to one level up.
6. Continue the process until all the vertices that are reachable from the starting vertex are visited.
7. Repeat above processes until all vertices are visited.

Depth First Search Traversal

- This method visits all the vertices, beginning with a specified **start vertex**.
- This strategy proceeds along a path from vertex V as deeply into the graph as possible.
- This means that after visiting V, the algorithm tries to visit any unvisited vertex adjacent to V.
- When the traversal reaches a vertex which has no adjacent vertex, it back tracks and visits an unvisited adjacent vertex.

- Depth-first traversal makes use of a **Stack data structure**.

DFS

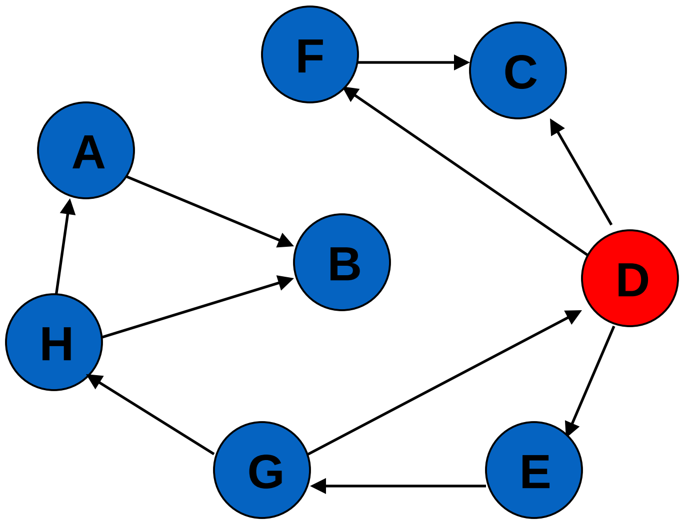


Visited Array

A	
B	
C	
D	
E	
F	
G	
H	

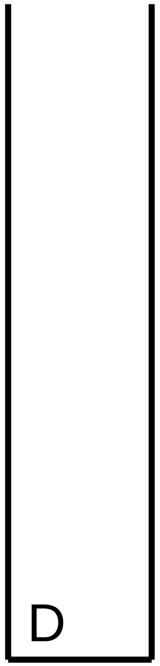
Task: Conduct a depth-first search of the graph starting with node D

DFS



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

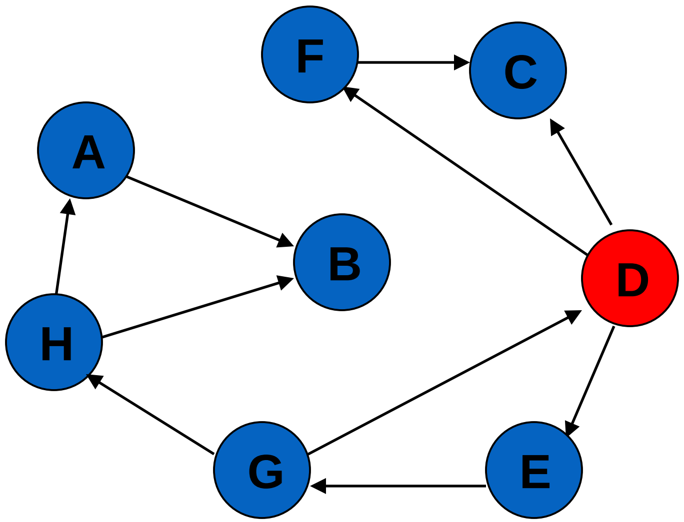


The order nodes are visited:

D

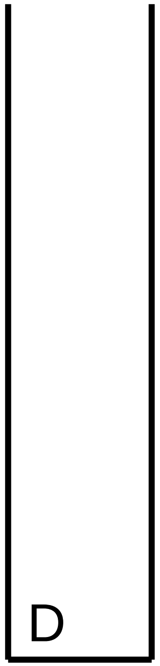
Visit D

DFS



Visited Array

A	
B	
C	
D	√
E	
F	
G	
H	

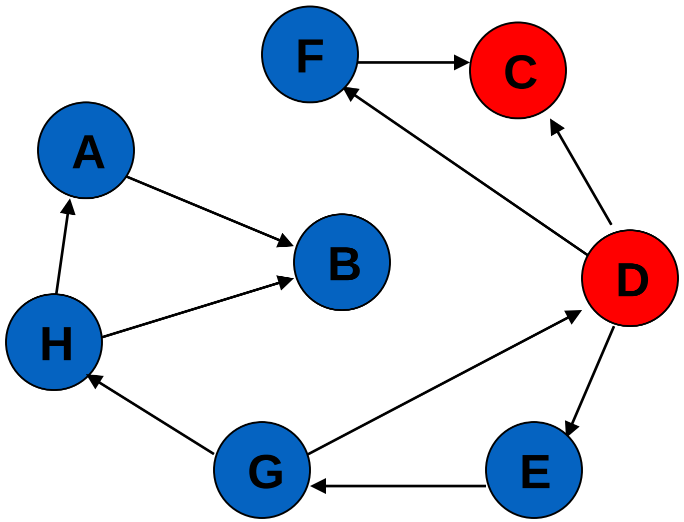


The order nodes are visited:

D

Consider nodes adjacent to D, decide to visit C first (Rule: visit adjacent nodes in alphabetical order)

DFS



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

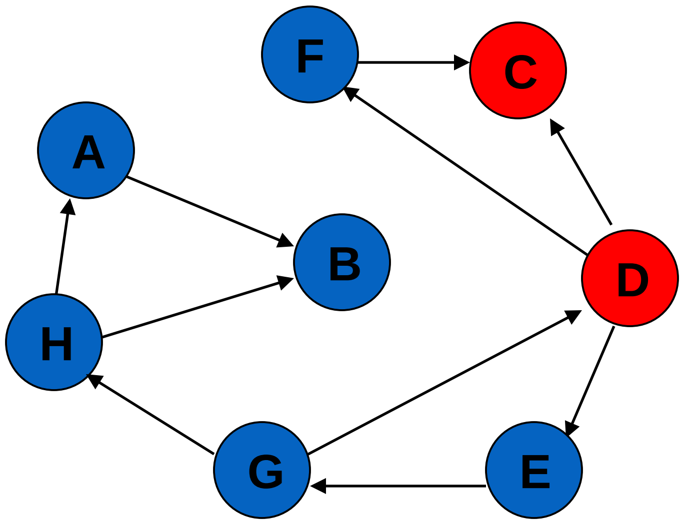
C
D

The order nodes are visited:

D, C

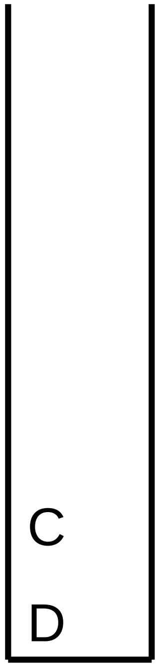
Visit C

DFS



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

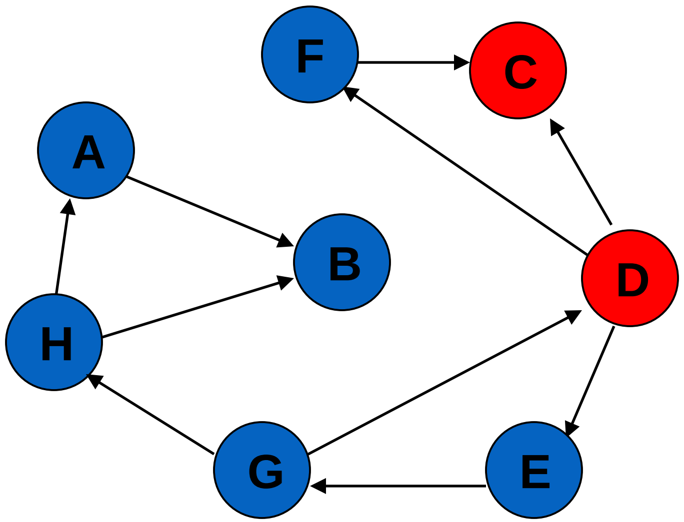


The order nodes are visited:

D, C

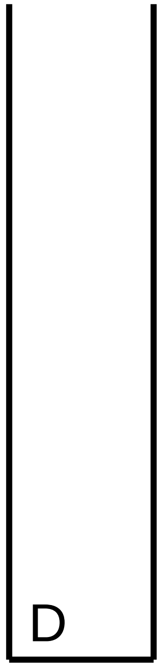
No nodes adjacent to C; cannot continue ➔ *backtrack*, i.e., pop stack and restore previous state

DFS



Visited Array

A	
B	
C	√
D	√
E	
F	
G	
H	

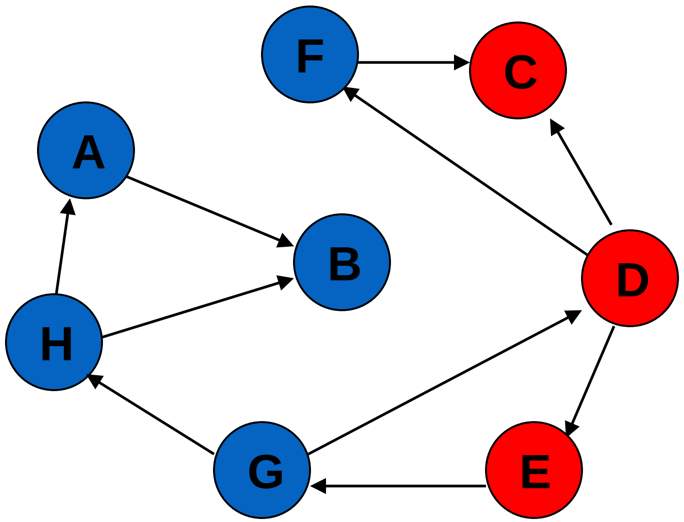


The order nodes are visited:

D, C

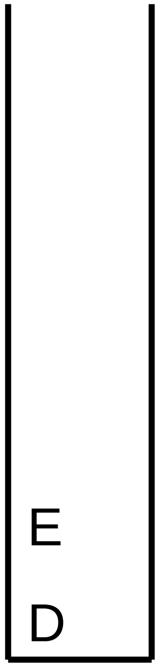
Back to D – C has been visited, decide to visit E next

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

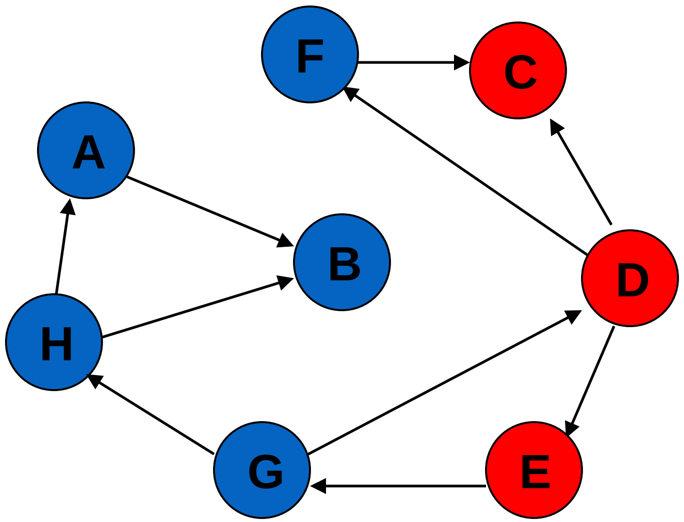


The order nodes are visited:

D, C, E

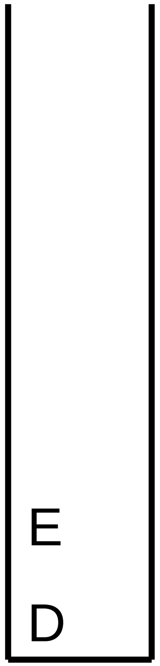
Back to D – C has been visited, decide to visit E next

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	
H	

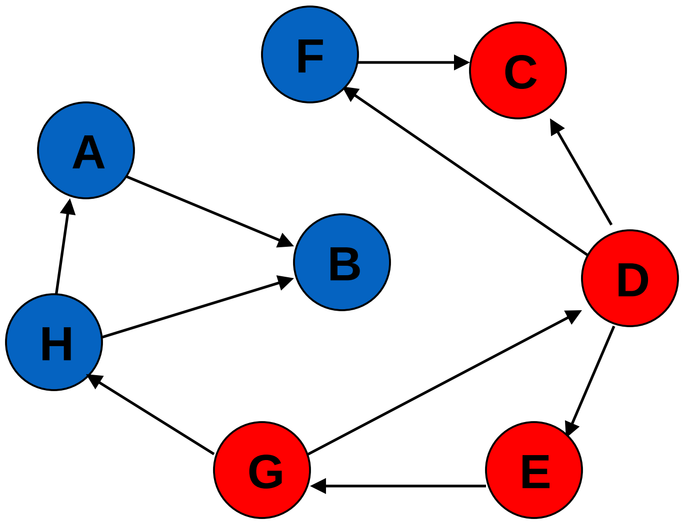


The order nodes are visited:

D, C, E

Only G is adjacent to E

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

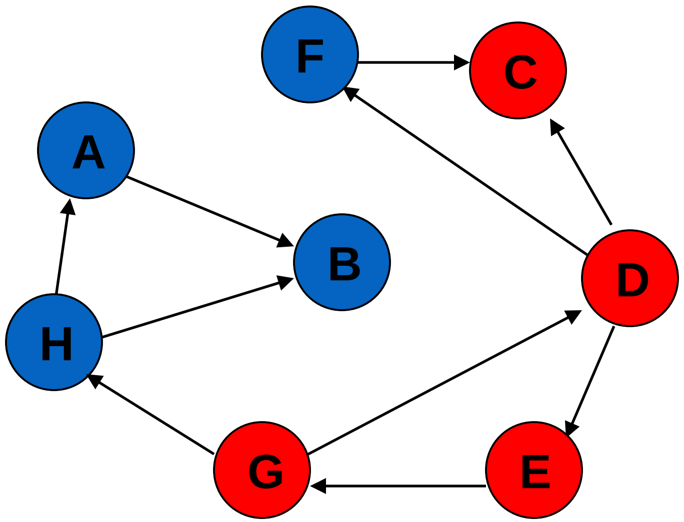
G
E
D

The order nodes are visited:

D, C, E, G

Visit G

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	

G

E

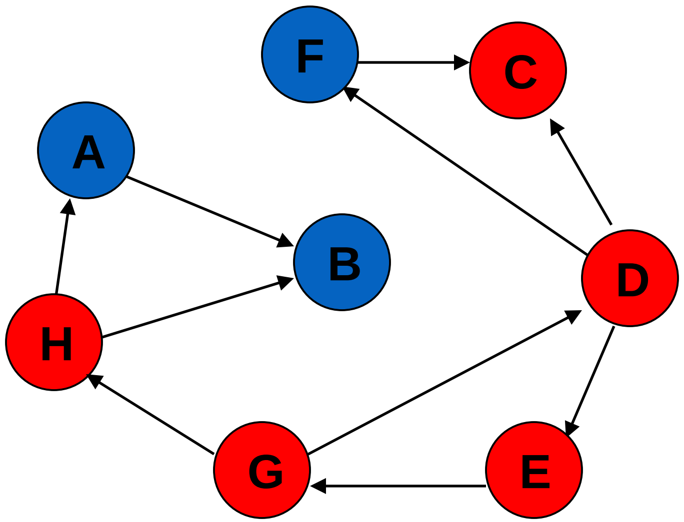
D

The order nodes are visited:

D, C, E, G

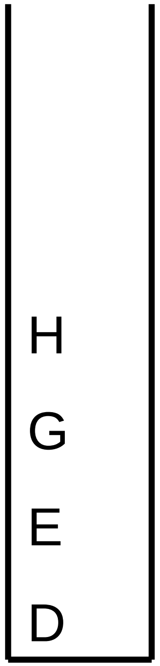
Nodes D and H are adjacent to G. D has already been visited. Decide to visit H.

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

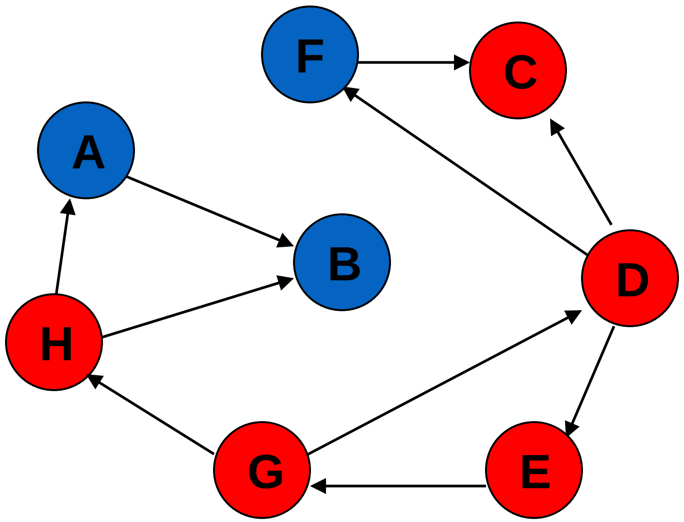


The order nodes are visited:

D, C, E, G, H

Visit H

DFS



Visited Array

A	
B	
C	√
D	√
E	√
F	
G	√
H	√

H

G

E

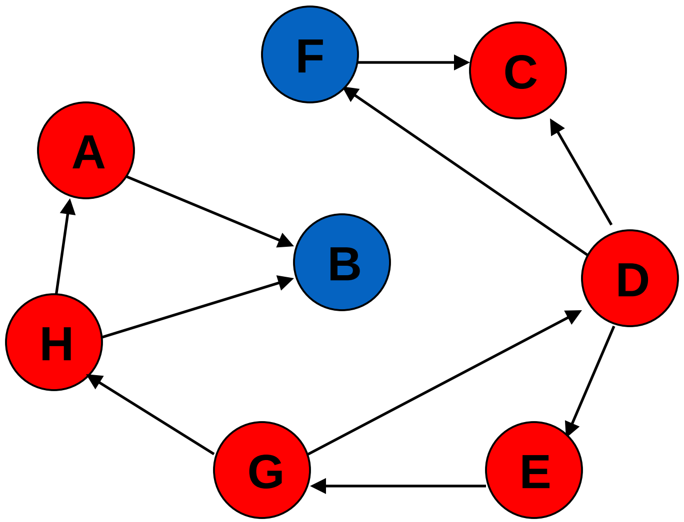
D

The order nodes are visited:

D, C, E, G, H

Nodes A and B are adjacent to F. Decide to visit A next.

DFS



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

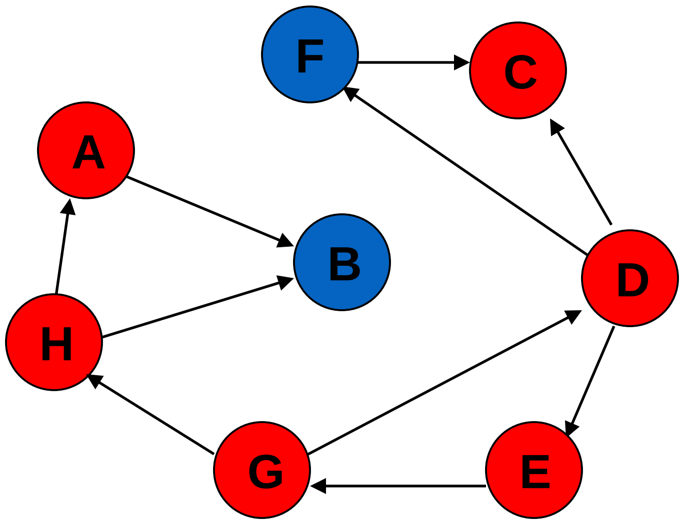
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A

Visit A

DFS



Visited Array

A	√
B	
C	√
D	√
E	√
F	
G	√
H	√

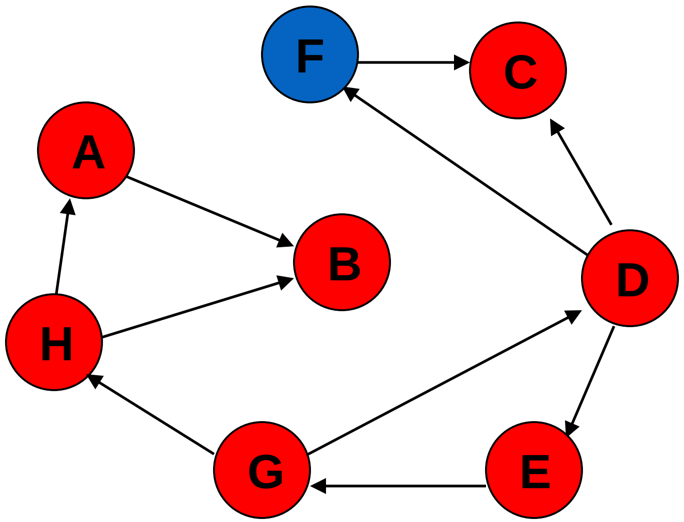
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A

Only Node B is adjacent to A. Decide to visit B next.

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

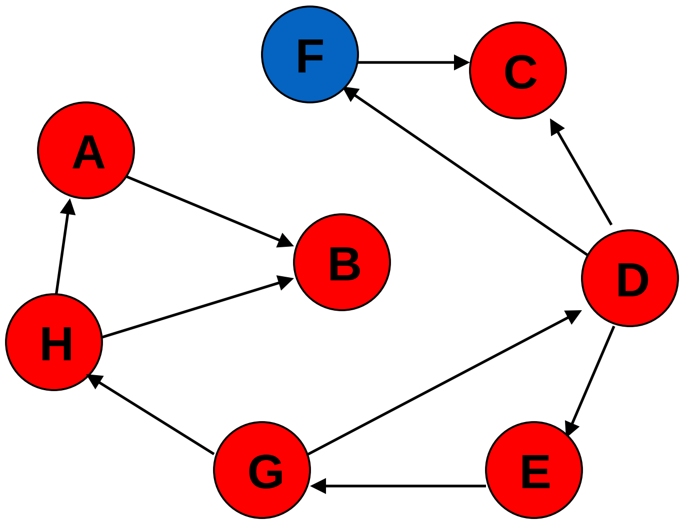
B
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A, B

Visit B

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

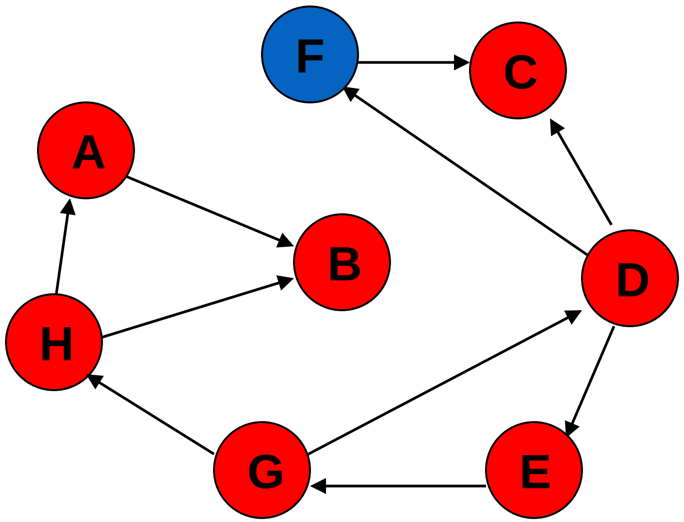
A
H
G
E
D

The order nodes are visited:

D, C, E, G, H, A, B

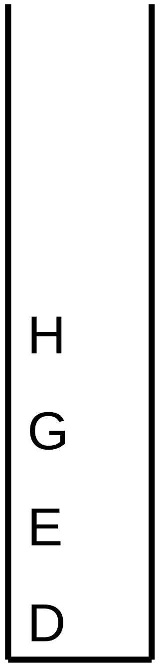
No unvisited nodes adjacent to B.
Backtrack (pop the stack).

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

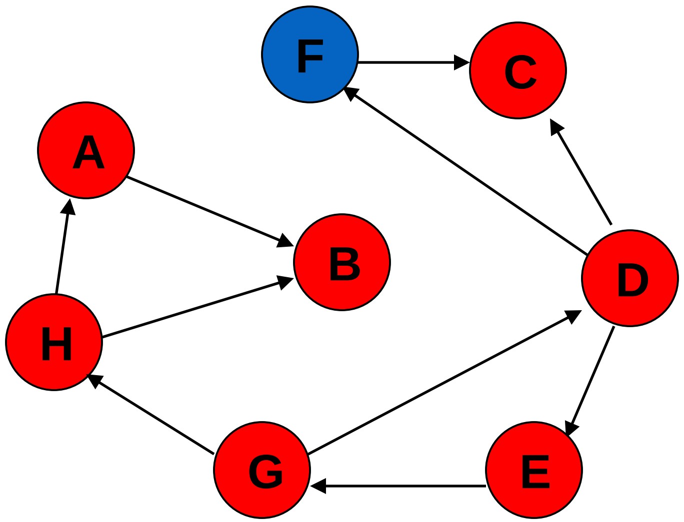


The order nodes are visited:

D, C, E, G, H, A, B

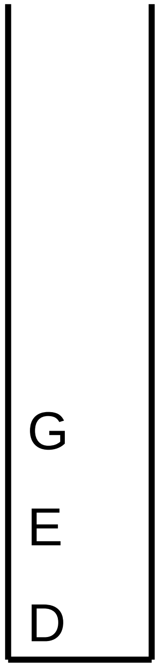
No unvisited nodes adjacent to A.
Backtrack (pop the stack).

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

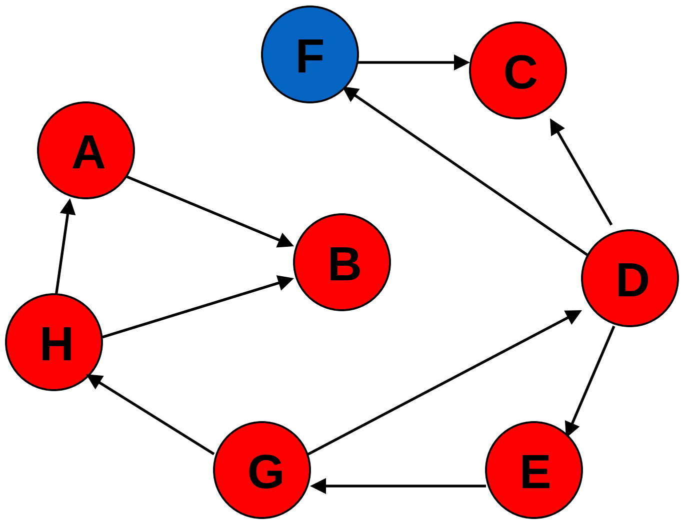


The order nodes are visited:

D, C, E, G, H, A, B

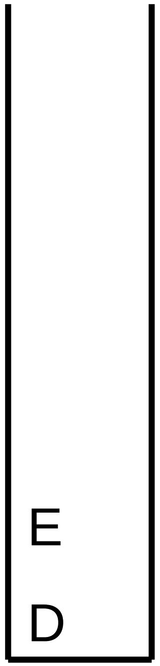
No unvisited nodes adjacent to H.
Backtrack (pop the stack).

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

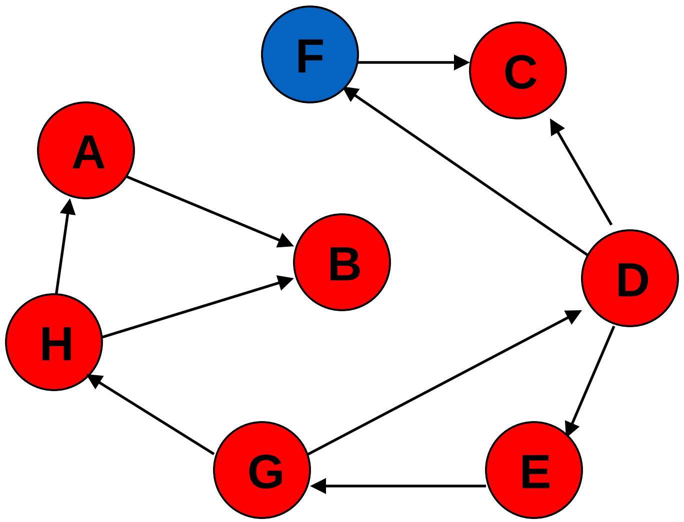


The order nodes are visited:

D, C, E, G, H, A, B

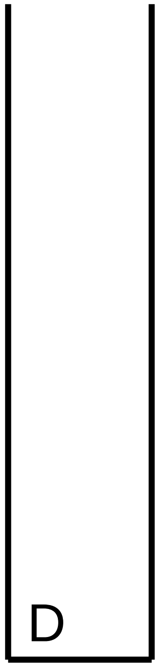
No unvisited nodes adjacent to G. Backtrack (pop the stack).

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

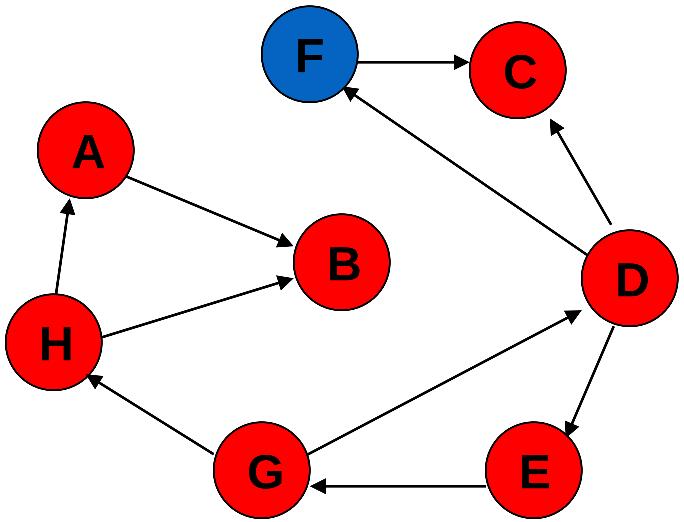


The order nodes are visited:

D, C, E, G, H, A, B

No unvisited nodes adjacent to E.
Backtrack (pop the stack).

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	
G	√
H	√

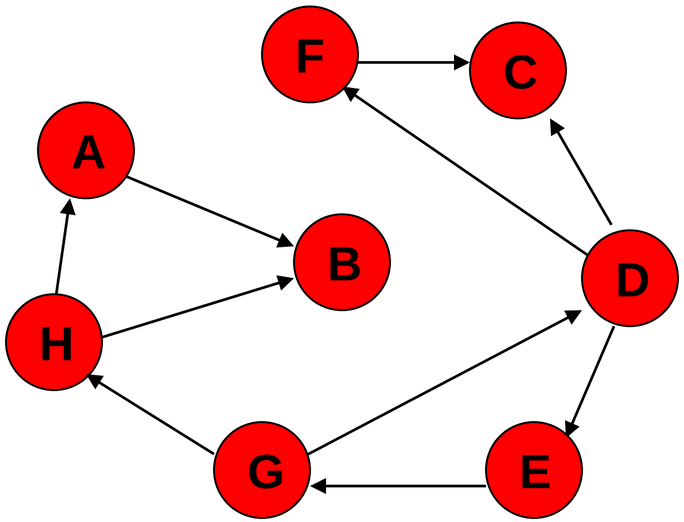
D

The order nodes are visited:

D, C, E, G, H, A, B

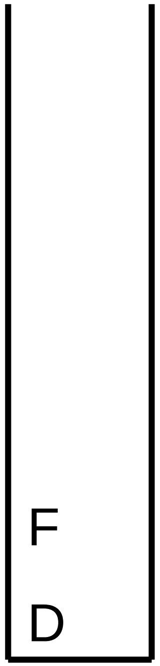
F is unvisited and is adjacent to D.
Decide to visit F next.

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

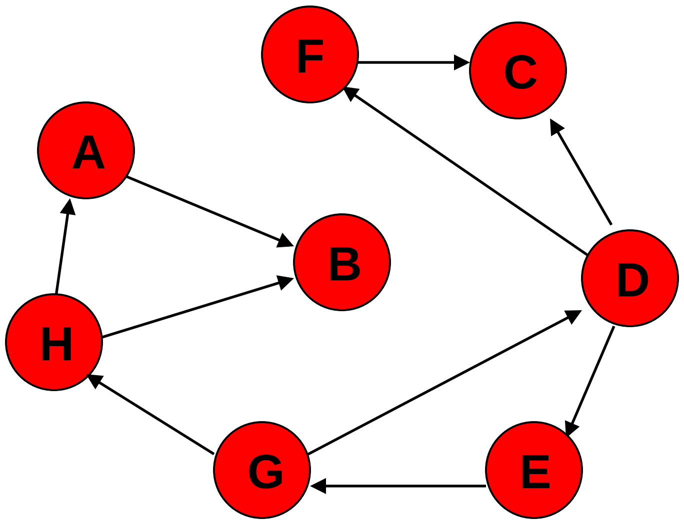


Visit F

The order nodes are visited:

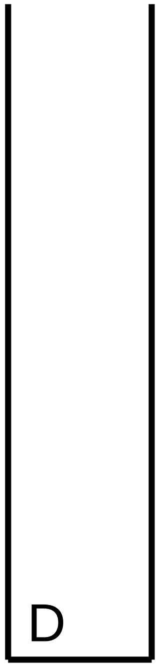
D, C, E, G, H, A, B, F

DFS



Visited Array

A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

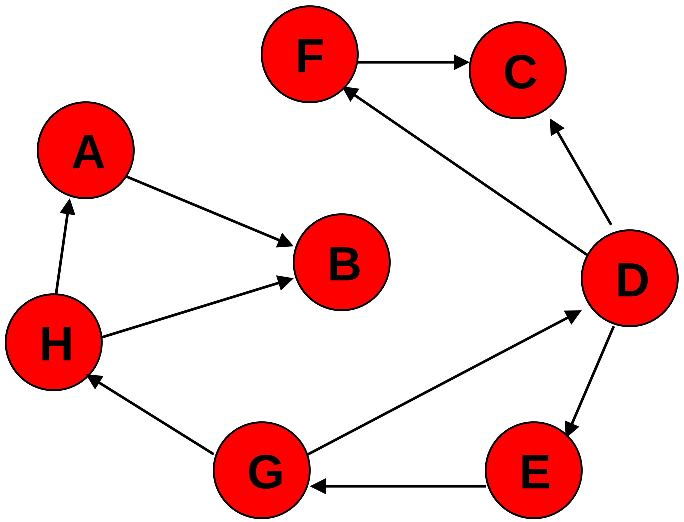


The order nodes are visited:

D, C, E, G, H, A, B, F

No unvisited nodes adjacent to F. Backtrack.

DFS



Visited Array

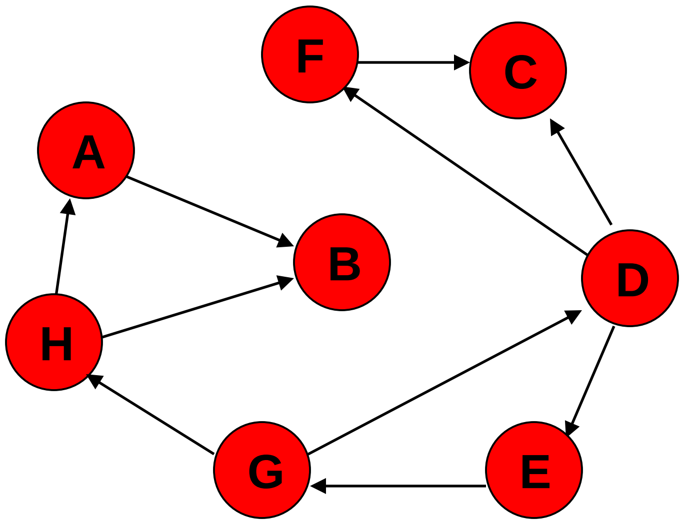
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

D, C, E, G, H, A, B, F

No unvisited nodes adjacent to D. Backtrack.

DFS



Visited Array

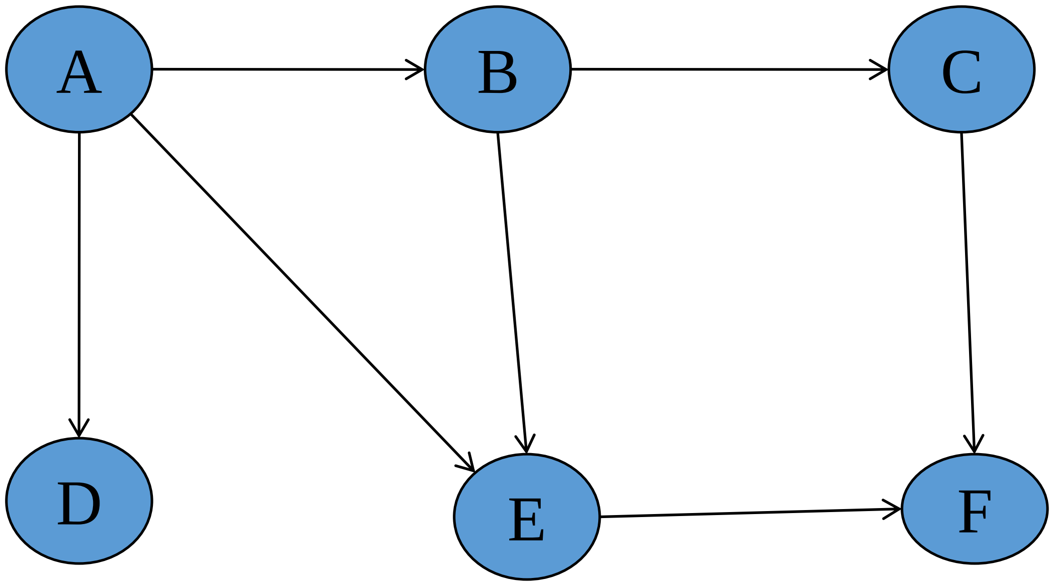
A	√
B	√
C	√
D	√
E	√
F	√
G	√
H	√

The order nodes are visited:

D, C, E, G, H, A, B, F

Stack is empty. Depth-first traversal is done.

Another DFS Example



A B C F E D

Adjacency List Implementation

```
struct GNode{
        int data;
        GNode *ptr;
};
GNode *GList;

void Create();
void InsertVertex(int);
```

```
    void InsertEdge();
    void DisplayVertices();
    void DeleteVertex();
    void DeleteEdges();
    bool SearchVertex();
    bool SearchEdge();
    void BFSTraversal();
    void DFSTraversal();

};
```

create( )

```
void Create()
{
   GList = new GNode[10];
   for(int i=0; i<10; i++)
   {
            GList[i].data=-1;
         GList[i].ptr=NULL;
   }
}
```

InsertVertex( )

```
void InsertVertex ( int VCount )
{

    int num;
   cout<<"Enter Vertex/Node Number =";
   cin>>num;

   GList[VCount].data=num;

}
```

InsertEdge( )

```
void InsertEdge()
{   int i, source,dest;
    cout<<"Enter Source Node=";
    cin>>source;
    cout<<"Enter Destinition Node=";
    cin>>dest;
    for(i=0;i<10;i++)
            if(source==GList[i].data)
                    break;

   GNode *check,*temp =new GNode;
   temp->data=dest;
   temp->ptr=NULL;
   check = GList[i].ptr;
```

```
  if(check!=NULL)    {
        while(check->ptr!=NULL)   {
                check=check->ptr;
        }
        check->ptr=temp;
  }
  else
        GList[i].ptr=temp;
}
```

SearchVertex( )

```
bool SearchVertex()
{
    int vertex;
    cout<<"Enter Vertex To Be Searched=";
   cin>>vertex;
   for(int i=0;i<10;i++)
    if(GList[i].data ==vertex)
            return true;
   return false;
}
```

SearchEdge( )

```
bool SearchEdge()
{
    int source, dest;
   GNode *temp;
   cout<<"Enter Edge to be searched"<<endl;
   cout<<"Enter Source Vertex=";
   cin>>source;
   cout<<"Enter Destination Vertex=";
   cin>>dest;
```

```
for( int i=0;i<10;i++)   {
    if(source==GList[i].data)      {
                temp=GList[i].ptr;
                while(temp!=NULL)        {
                if(temp->data==dest)
                                return true;
            else
                                temp=temp->ptr;
        }
      }
   }
   return false;
}
```

```
struct node{
    int vertex;
    node *next;
};
node *adj[MAX_NODE]; //For storing Adjacency list of nodes.
int totNodes; //No. of Nodes in Graph.//
int top=-1;
int stack[MAX_NODE];
void DFS_traversal(){
    node *tmp;
    int N,v,start_node, status[MAX_NODE];//status arr for maintaing status.
constant int ready=1,wait=2,processed=3; //status of node.
    cout<<"Enter starting node : ";
    cin>>start_node;
    //step 1 : Initialize all nodes to ready state.
for(int i=1;i<=totNodes;i++)
        status[i]=ready;
    //step 2 : push the start node in stack and change status.
    push(start_node); //Push starting node into stack.
    status[start_node]=wait; //change it status to wait state.
//Step 3 : Repeat until stack is empty.
while(is_stk_empty()!=1){
        //step 4 : pop the node N of stack.
//process N and change the status of N to
//be processed state.

 N = pop(); //pop the node from stack.
        status[N]=processed; //status of N to processed.
        cout<<"   "<<N; //displaying processed node.

//step 5 : push onto stack all the neighbours of N,
//that are in ready state and change their status to
//wait state.

 tmp = adj[N];
while(tmp!=NULL){
            v = tmp->vertex;
            if(status[v]==ready){//check status of N's neighbour.
                push(v); //push N's neighbour who are in ready state.
                status[v]=wait; //and make their status to wait state.
            }
            tmp=tmp->next;
        }
    }
}
```

DFS

```
    #define MAX_NODE 50
    struct node{
        int vertex;
        node *next;
    };
    node *adj[MAX_NODE]; //For storing Adjacency list of
nodes.
    int totNodes; //No. of Nodes in Graph.//
    int top=-1;
    int stack[MAX_NODE];

    void DFS_traversal(){
        node *tmp;
        int N,v,start_node, status[MAX_NODE];//status
arr for maintaing status.
    constant int ready=1,wait=2,processed=3; //status of
node.
        cout<<"Enter starting node : ";
        cin>>start_node;
```

DFS

```
        //step 1 : Initialize all nodes to ready state.
    for(int i=1;i<=totNodes;i++)
            status[i]=ready;
    //step 2 : push the start node in stack and change
status.
        push(start_node); //Push starting node into
stack.
        status[start_node]=wait; //change it status to
wait state.
```

DFS

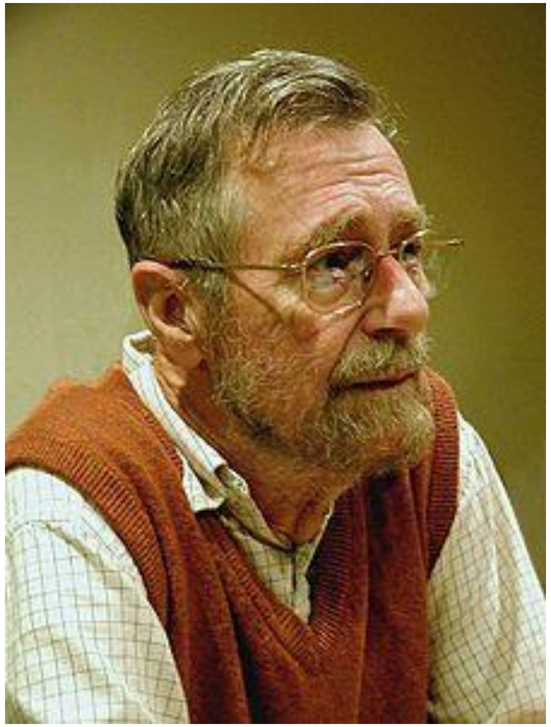
```
        #//Step 3 : Repeat until stack is empty.
    while(is_stk_empty()!=1){

    //step 4 : pop the node N of stack.
    //process N and change the status of N to
    //be processed state.

     N = pop(); //pop the node from stack.
            status[N]=processed; //status of N to
processed.
            cout<<"   "<<N; //displaying processed node.

    //step 5 : push onto stack all the neighbours of N,
    //that are in ready state and change their status to
    //wait state.
```

DFS

```
     tmp = adj[N];
    while(tmp!=NULL){
                v = tmp->vertex;
                if(status[v]==ready){//check status of N's neighbour.
                    push(v); //push N's neighbour who are in ready
state.
                    status[v]=wait; //and make their status to wait
state.
                }
                tmp=tmp->next;
            }
        }
    }
```

Dijkstra's algorithm

The author: Edsger Wybe Dijkstra

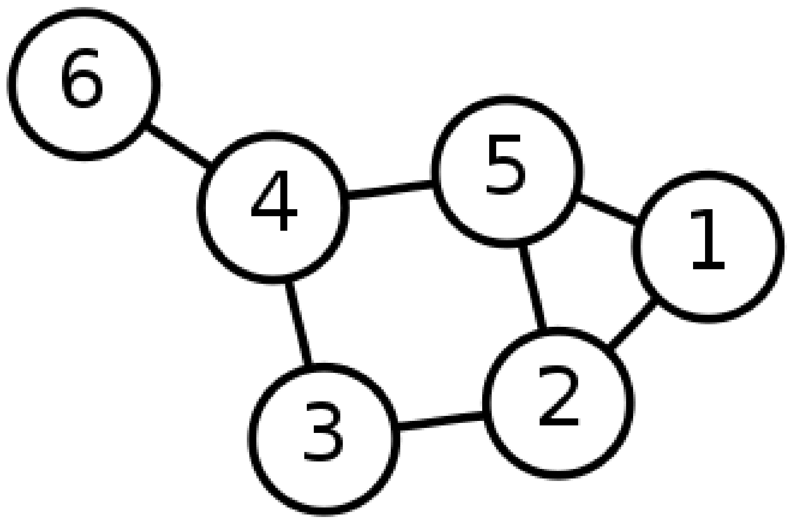


"Computer Science is no more about computers than astronomy is about telescopes."

http://www.cs.utexas.edu/~EWD/

Single-Source Shortest Path Problem

<u>Single-Source Shortest Path Problem</u> - The problem of finding shortest paths from a source vertex *v* to all other vertices in the graph.



Dijkstra's algorithm

Dijkstra's algorithm - is a solution to the single-source shortest path problem in graph theory.

Works on both directed and undirected graphs. However, all edges must have nonnegative weights.

Approach: Greedy

Input: Weighted graph G={E,V} and source vertex $v \in V$, such that all edge weights are nonnegative

Output: Lengths of shortest paths (or the shortest paths themselves) from a given source vertex $v \in V$ to all other vertices

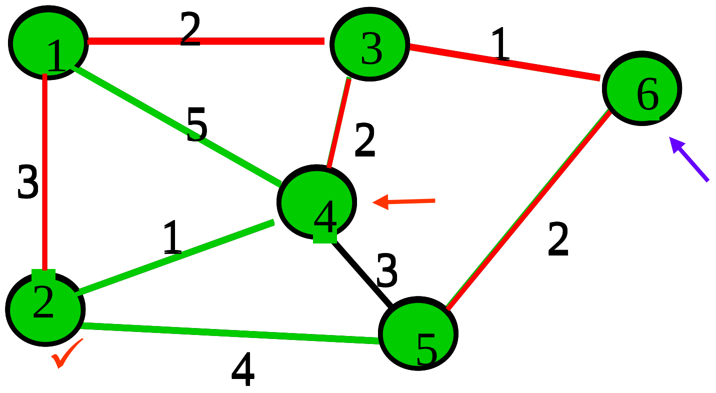
Dijkstra's algorithm

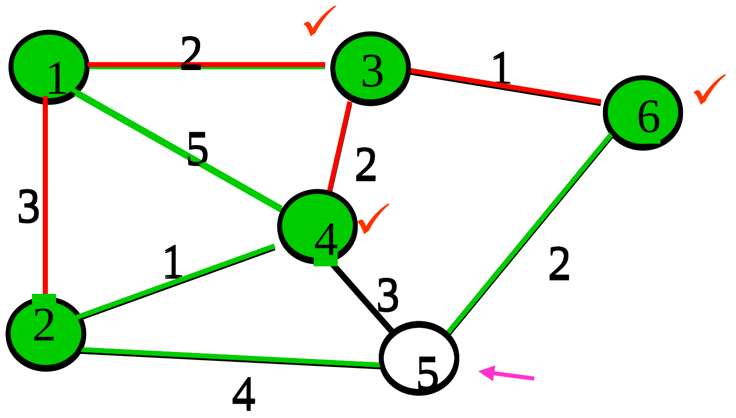
- N: set of nodes for which shortest path already found
- Initialization: (*Start with source node s)*
  - $N = \{s\}$, $D_s = 0$, "*s* is distance zero from itself"
  - $D_j = C_{sj}$ for all $j \neq s$, distances of directly-connected neighbors
- Step A: (*Find next closest node i*)
  - Find $i \notin N$ such that
  - $D_i = \min Dj$ for $j \notin N$
  - Add i to N
  - If N contains all the nodes, stop
- Step B: (*update minimum costs)*
  - For each node $j \notin N$
  - $D_j = \min (D_j, D_i + C_{ij})$
  - Go to Step A

Dijkstra's algorithm - Pseudocode

dist[s] ← 0	(distance to source vertex is zero)
for all v ∈ V–{s}	
do dist[v] ←∞	(set all other distances to infinity)
S←∅	(S, the set of visited vertices is initially empty)
Q←V	(Q, the queue initially contains all vertices)
while Q ≠∅	(while the queue is not empty)
do u ← mindistance(Q,dist)	(select the element of Q with the min. distance)
S←S∪{u}	(add u to list of visited vertices)
for all v ∈ neighbors[u]	
do if dist[v] > dist[u] + w(u, v)	(if new shortest path found)
then d[v] ←d[u] + w(u, v)	(set new value of shortest path)
(if desired, add traceback code)	
return dist	

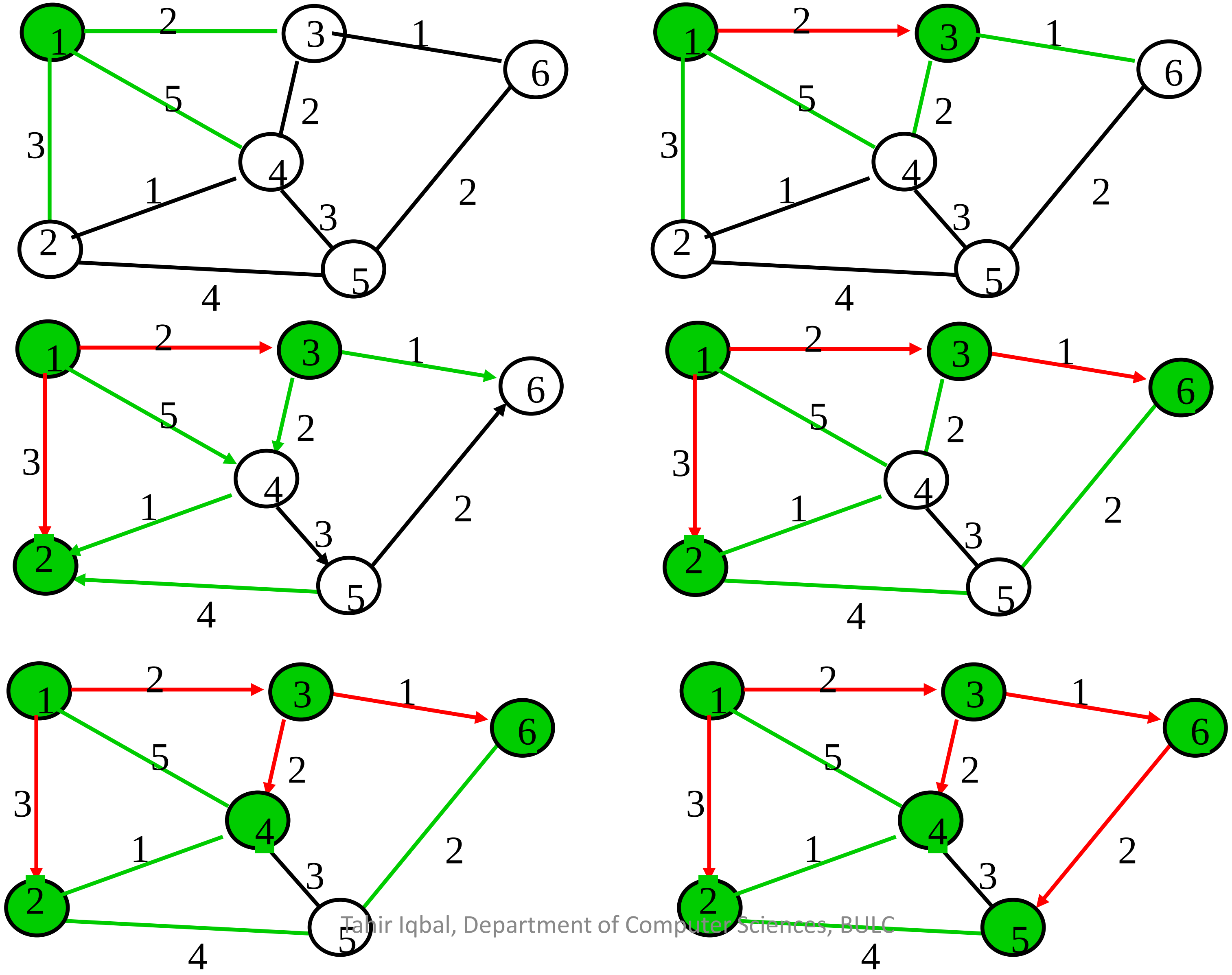
Execution of Dijkstra’s algorithm





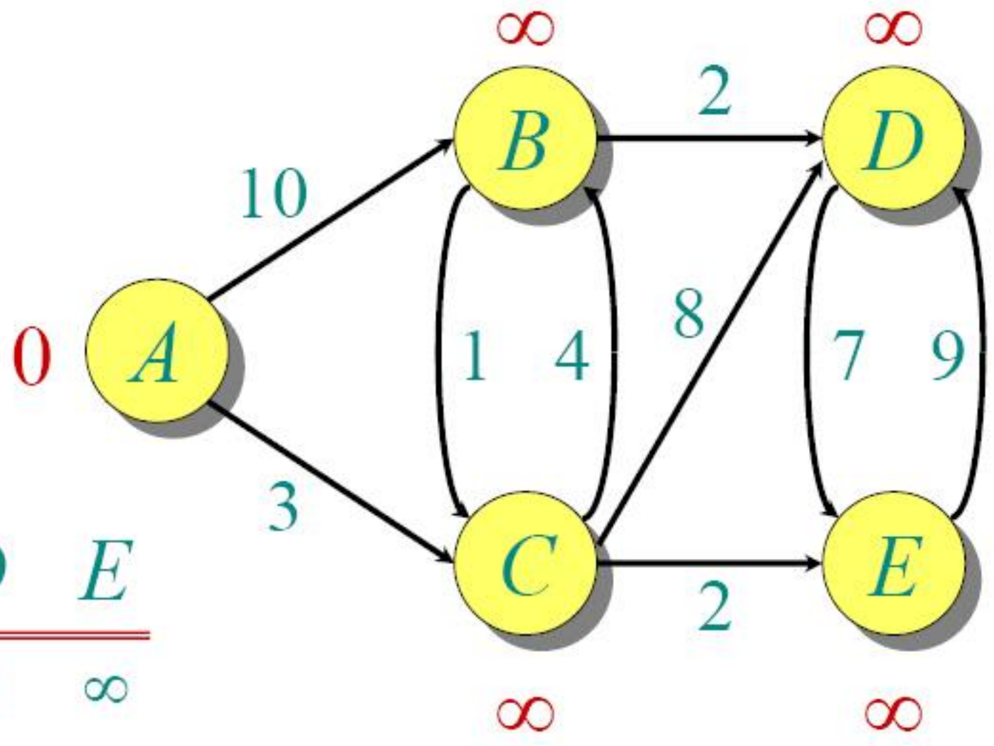
Iteration	**N**	D_2	D_3	D_4	D_5	D_6
Initial	{1}	3	2 ✓	5	∝	∝
1	{1,3}	3 ✓	**2**	4	∝	3
2	{1,2,3}	**3**	**2**	4	7	3 ✓
3	{1,2,3,6}	**3**	**2**	4 ✓	5	**3**
4	{1,2,3,4,6}	**3**	**2**	**4**	5 ✓	**3**
5	{1,2,3,4,5,6}	**3**	**2**	**4**	**5**	**3**

Shortest Paths in Dijkstra’s Algorithm



Dijkstra Animated Example

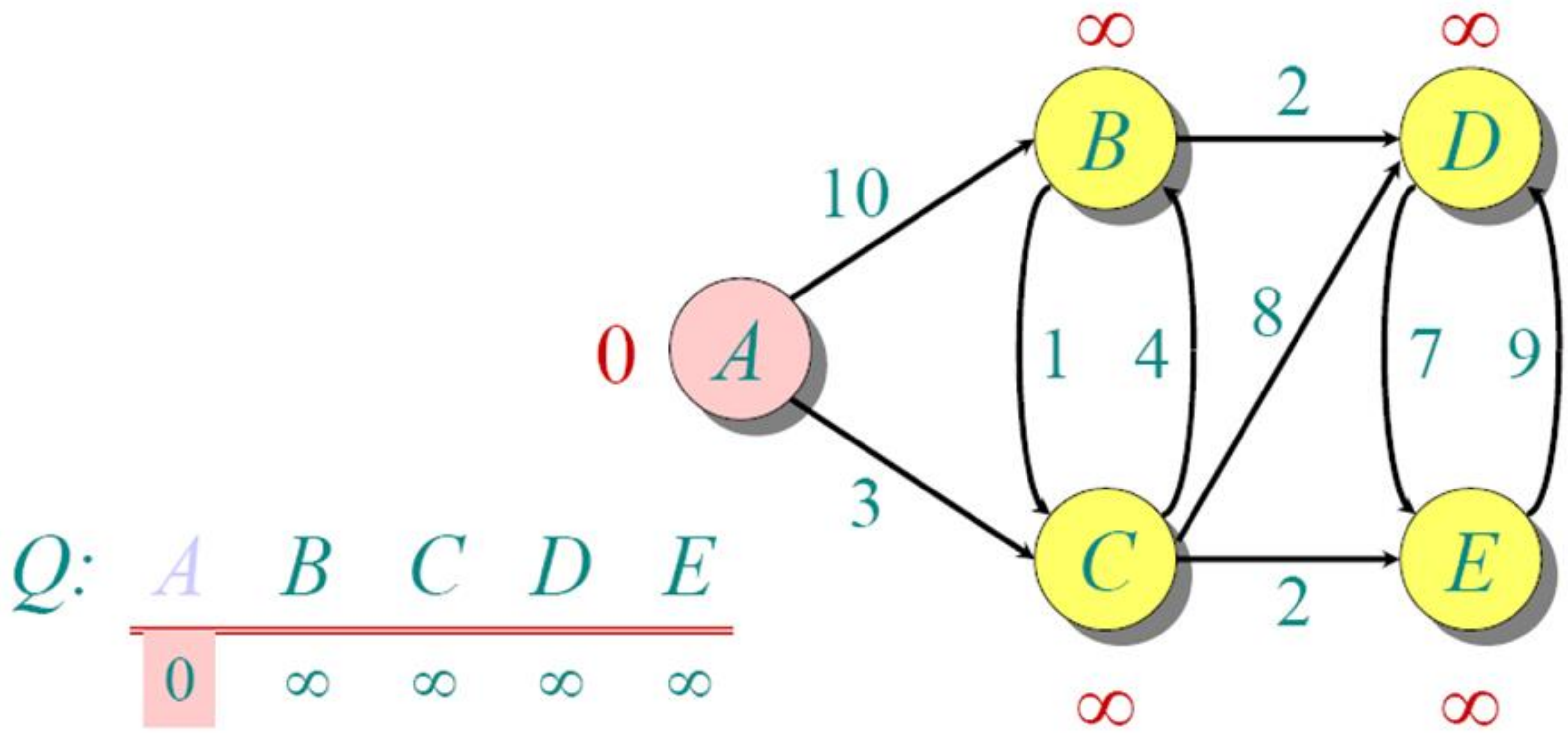
Initialize:



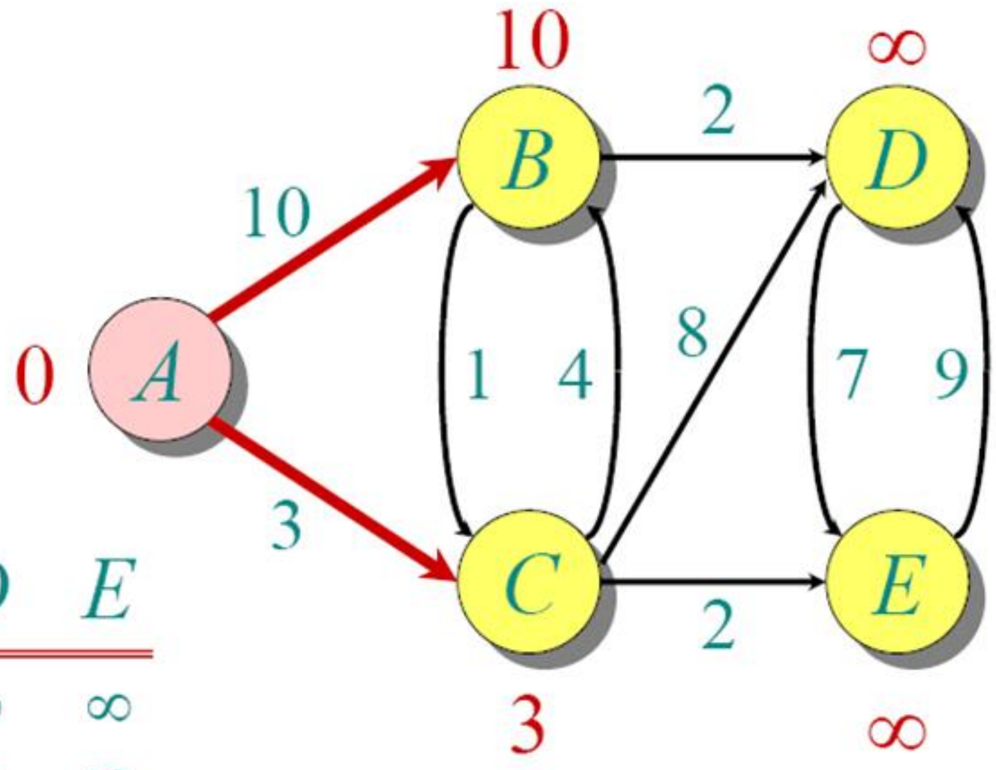
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Dijkstra Animated Example



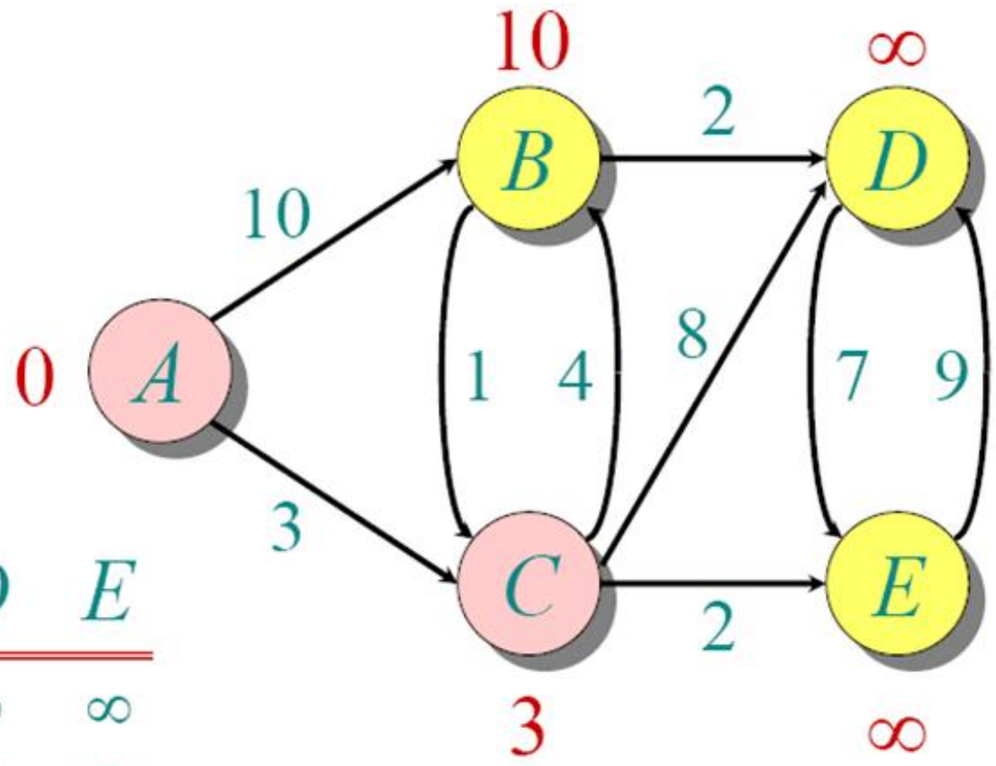
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Dijkstra Animated Example

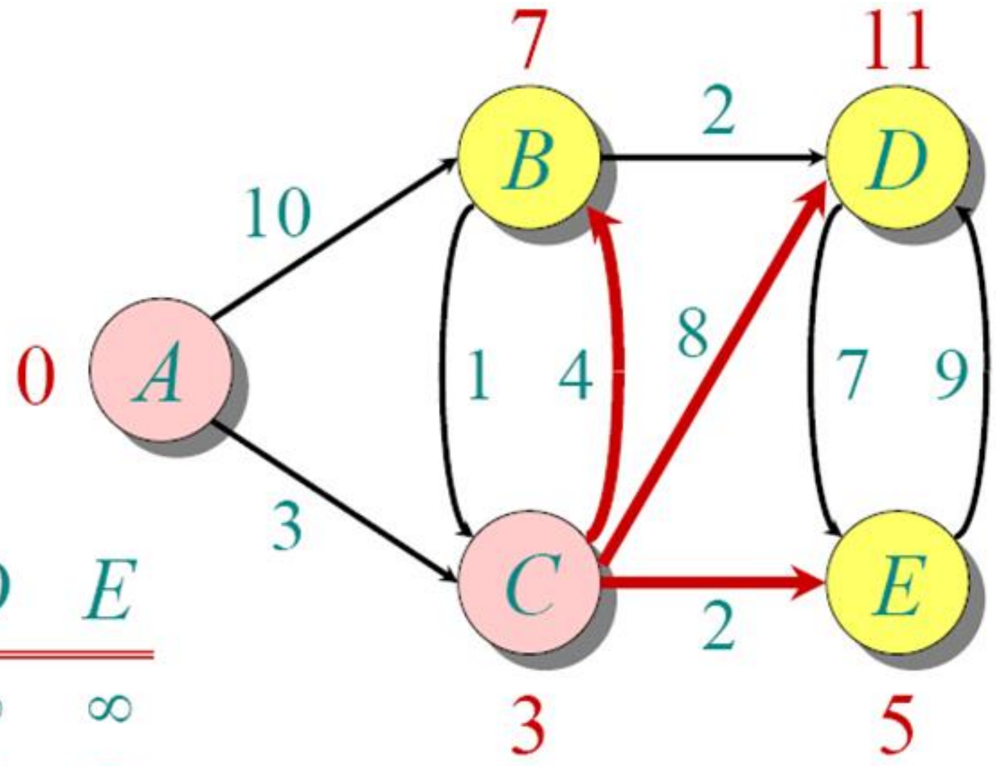


Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

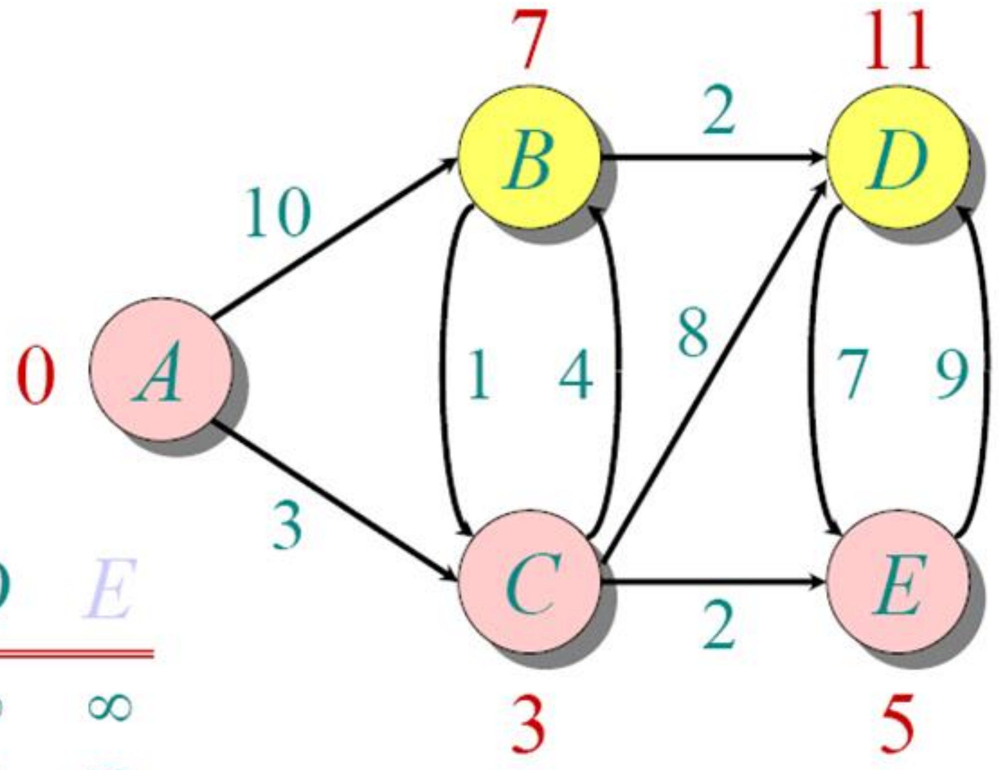
Dijkstra Animated Example

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5



S: { A, C }

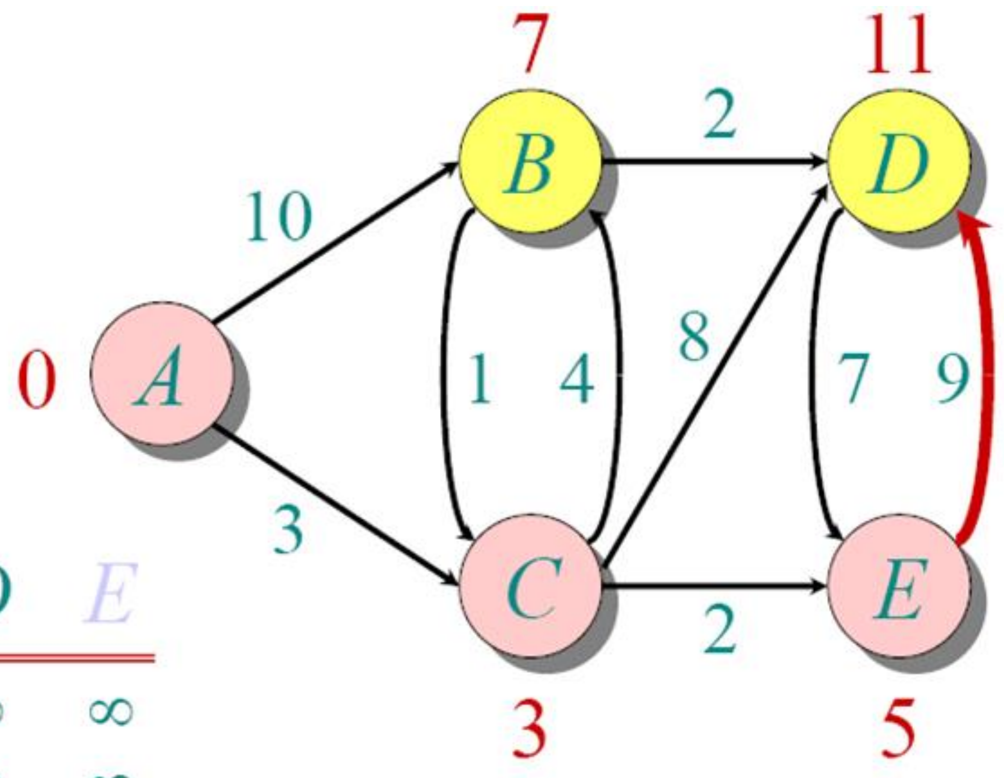
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

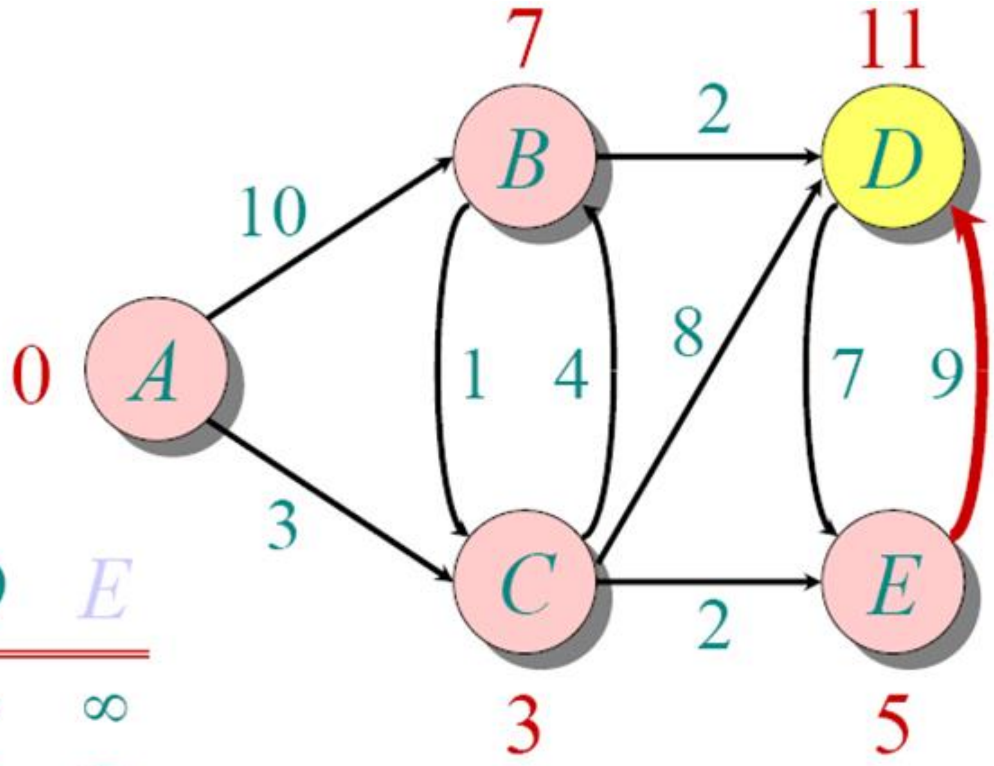
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

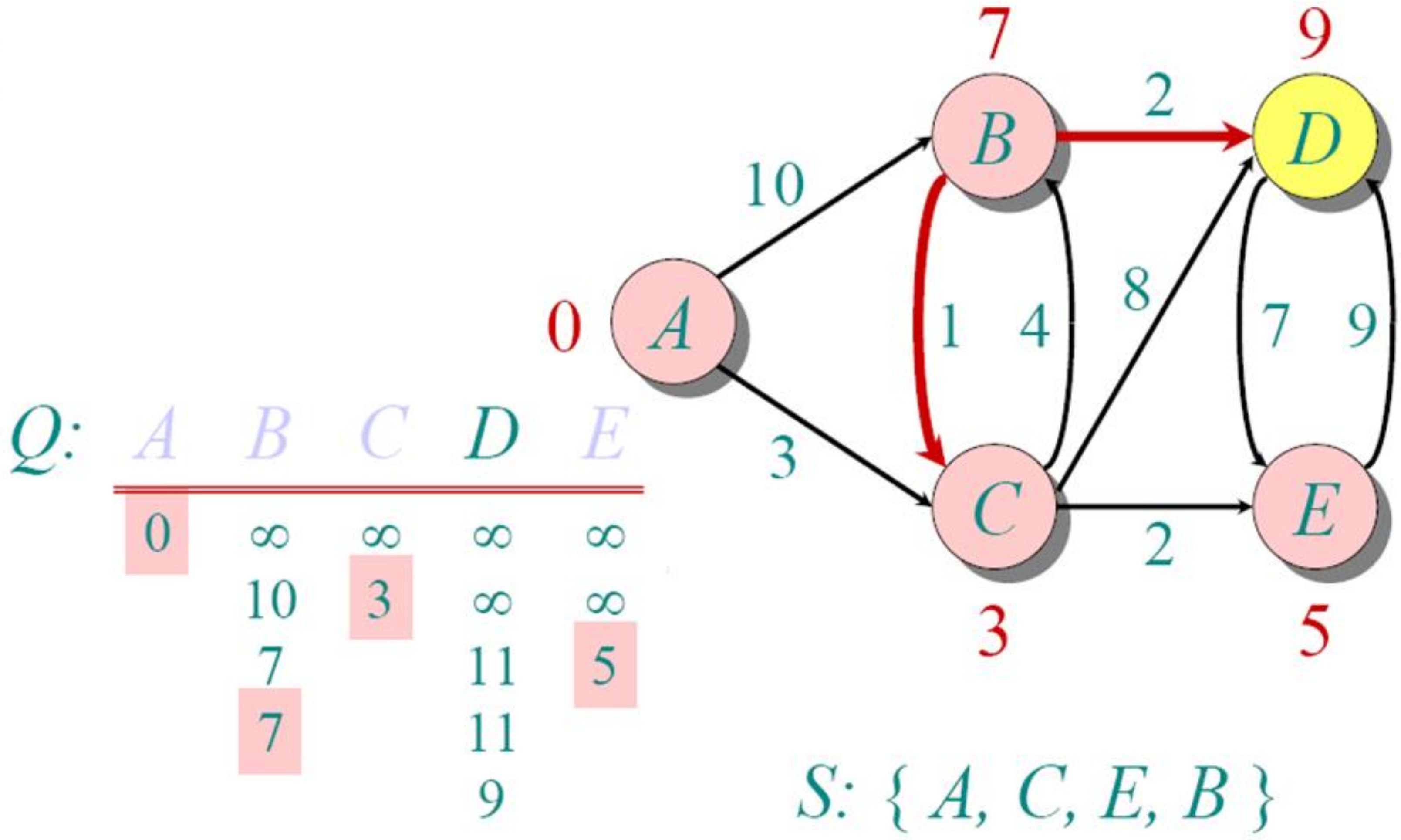
Dijkstra Animated Example



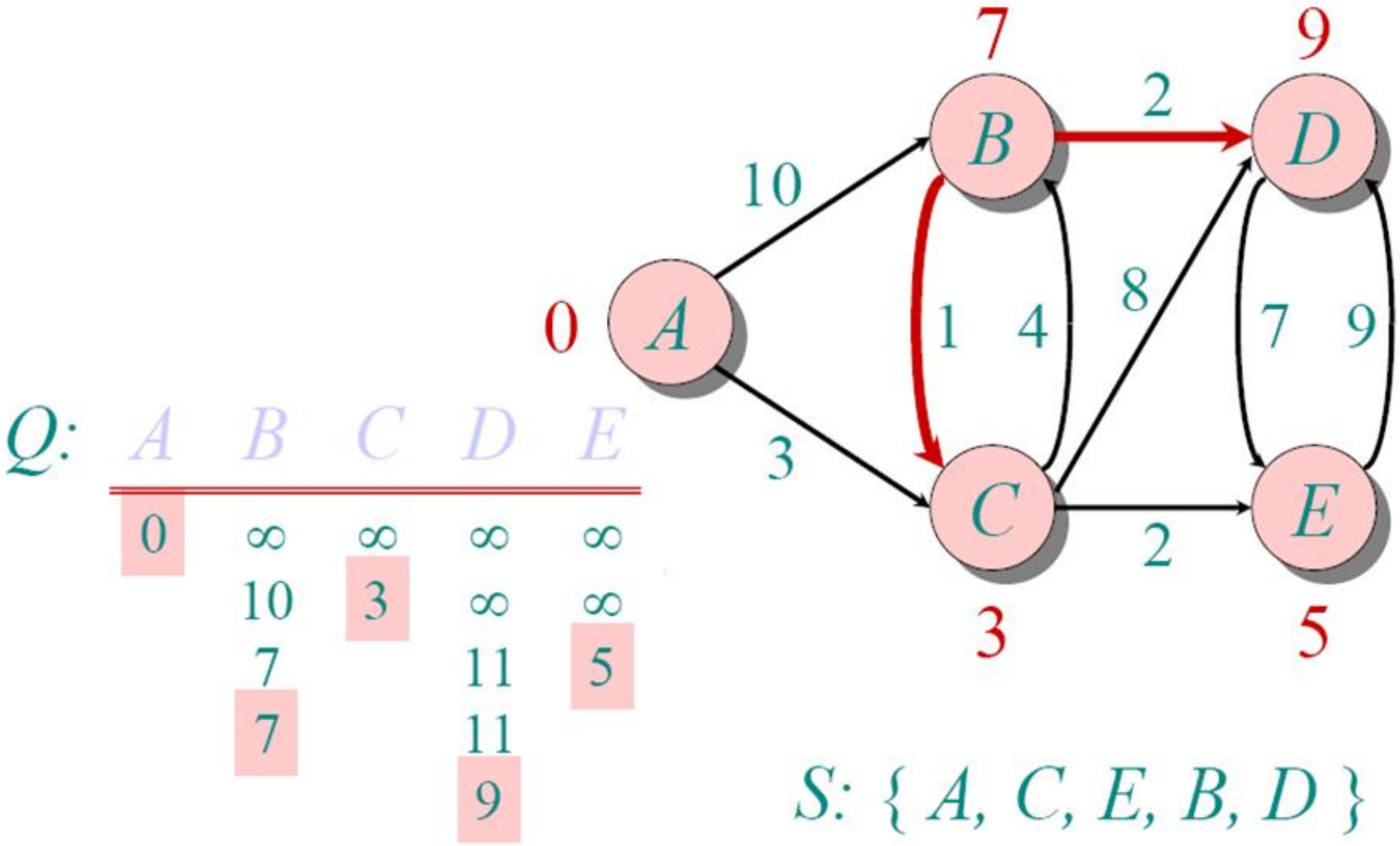
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

$S: \{A, C, E, B\}$

Dijkstra Animated Example

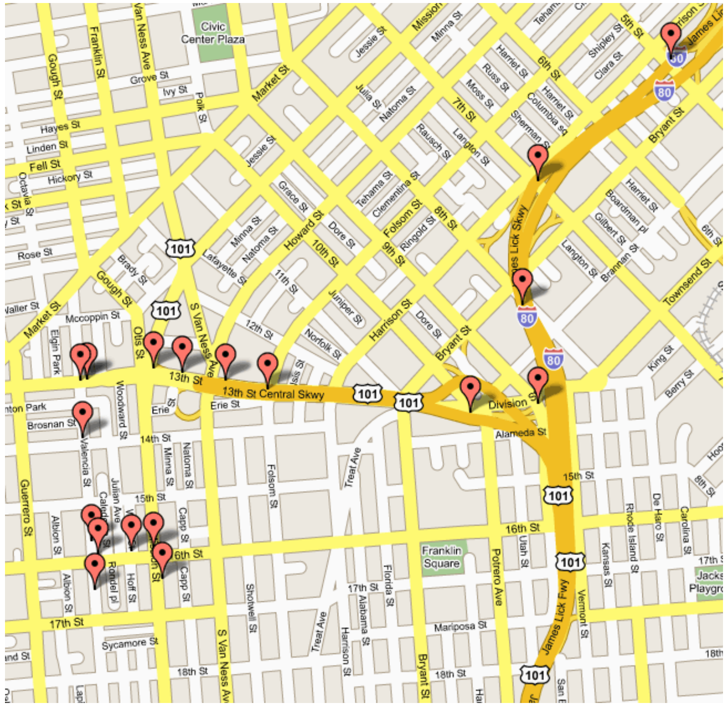


Dijkstra Animated Example

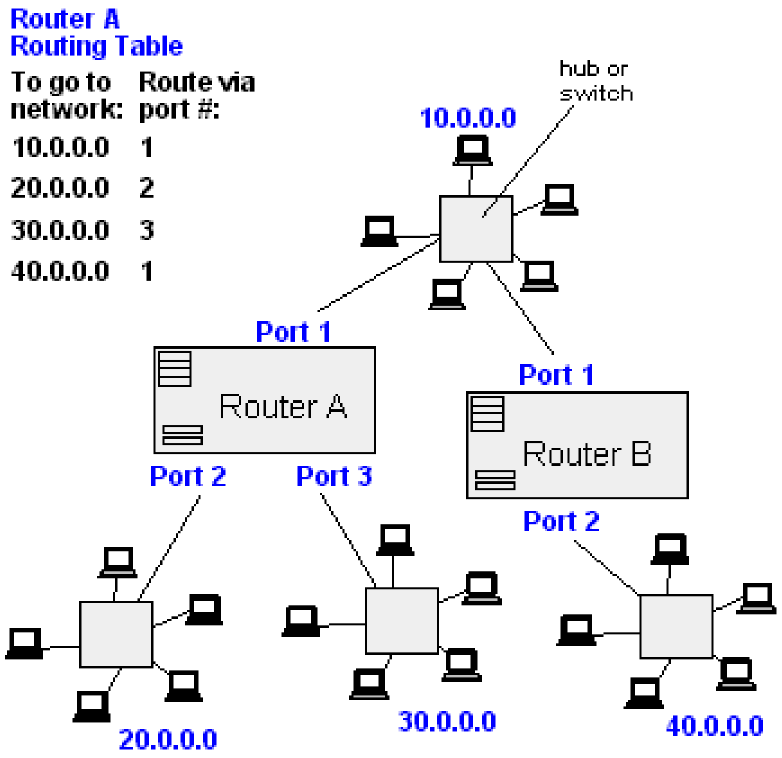


Applications of Dijkstra's Algorithm

- Traffic Information Systems are most prominent use
- Mapping (Map Quest, Google Maps)
- Routing Systems



From Computer Desktop Encyclopedia
© 1998 The Computer Language Co. Inc.



Tokyo Subway Map