UNIT 5 - GRAPHS

The Graph ADT Introduction

Definition

Graph representation

Elementary graph operations BFS, DFS

Introduction to Graphs

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads. Euler used graph theory to solve Seven Bridges of Königsberg problem. Is there a possible way to traverse every bridge exactly once – Euler Tour

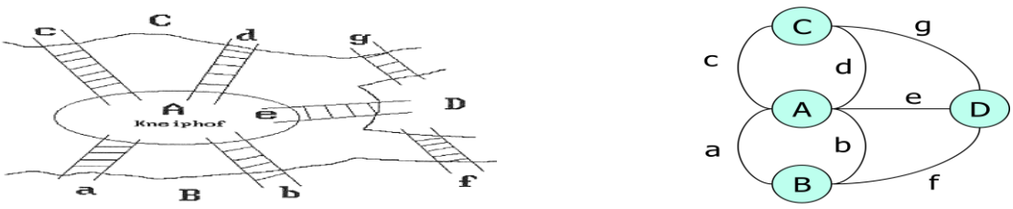


Figure: Section of the river Pregal in Koenigsberg and Euler's graph.

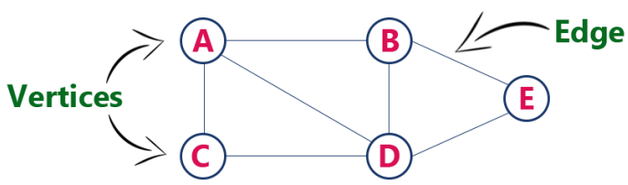
Defining the degree of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.**Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2.**Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1.Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

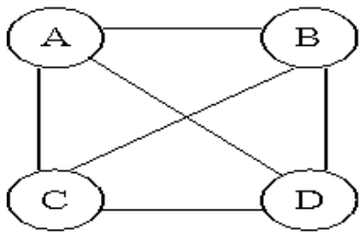
2.Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3.Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1.Undirected Graph

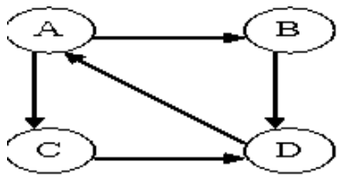
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2.Directed Graph

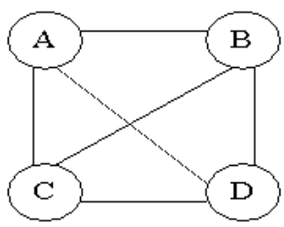
A graph with only directed edges is said to be directed graph.



Directed Graph.

3.Complete Graph

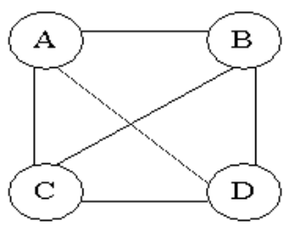
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4.Regular Graph

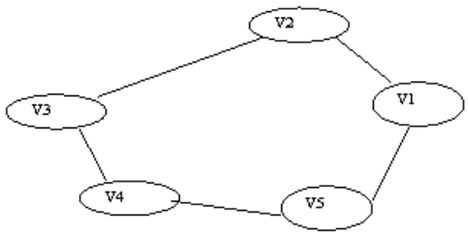
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5.Cycle Graph

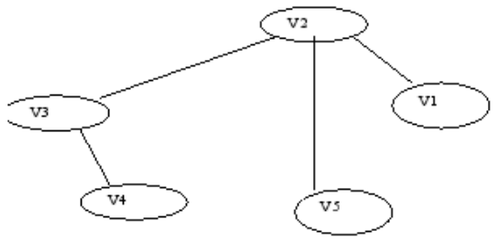
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

6.Acyclic Graph

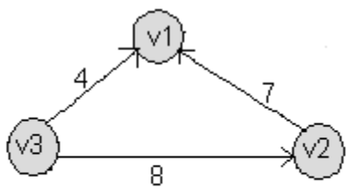
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

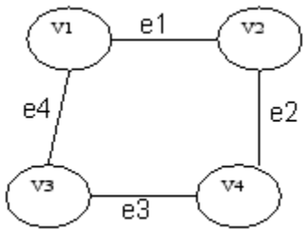
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk

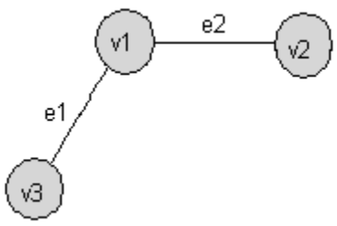
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

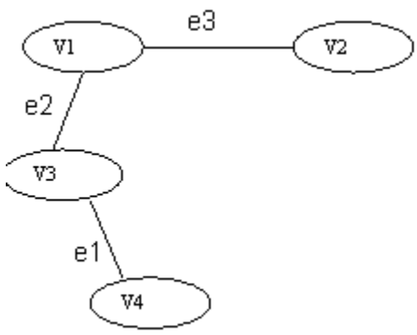
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

The number of edges in a path is called the length of that path. In the following, the length of the path is 3.

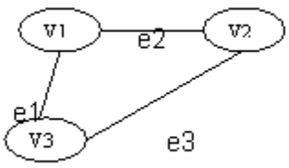


An open walk Graph

Circuit

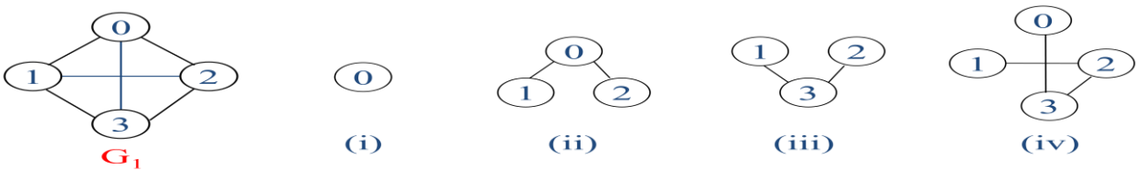
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

A circuit having three vertices and three edges.



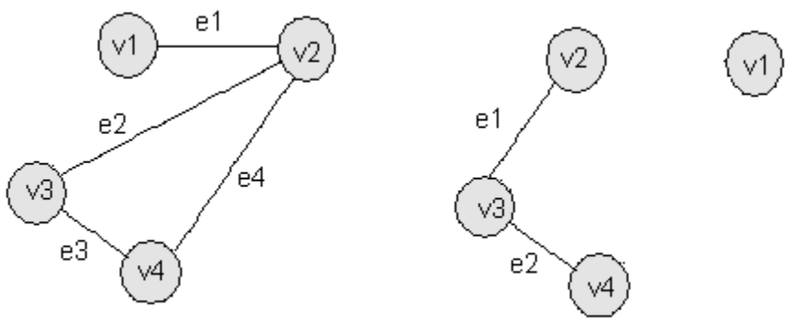
Sub Graph

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

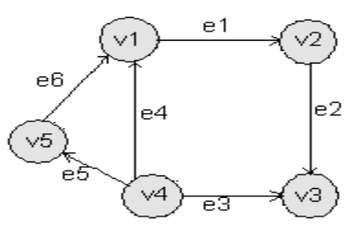
Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree

The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

ADT of Graph:

Structure Graph is

objects: a nonempty set of vertices and a set of undirected edges, where each edge is a pair of vertices

functions: for all *graph* ∈ *Graph*, *v*, v_1 and v_2 ∈ *Vertices*

Graph Create()::=return an empty graph

Graph InsertVertex(*graph*, *v*)::= return a graph with *v* inserted. *v* has no edge.

Graph InsertEdge(*graph*, *v1*,*v2*)::= return a graph with new edge between *v1* and *v2*

Graph DeleteVertex(*graph*, *v*)::= return a graph in which *v* and all edges incident to it are removed

Graph DeleteEdge(*graph*, *v1*, *v2*)::=return a graph in which the edge (*v1*, *v2*) is removed

Boolean IsEmpty(*graph*)::= if (*graph*==*empty graph*) return TRUE else return FALSE

List Adjacent(*graph*,*v*)::= return a list of all vertices that are adjacent to *v*

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**
3. **Adjacency Multilists**

1.Adjacency Matrix

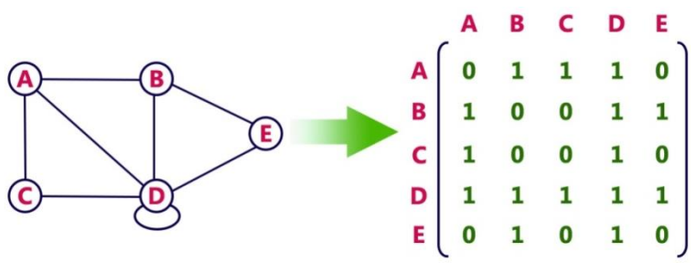
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

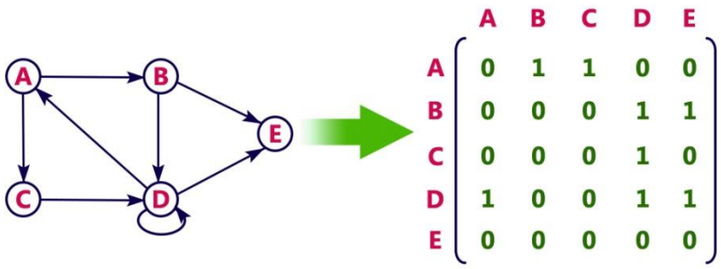
Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff (v_i, v_j) ∈E(G) (⟨v_i, v_j⟩ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



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

For a Directed graph



The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix:

From the adjacency matrix, to determine the connection of vertices is easy

The degree of a vertex is $\sum_{j=0}^{n-1} adj\_mat[i][j]$

For a digraph, the row sum is the out_degree, while the column sum is the in_degree

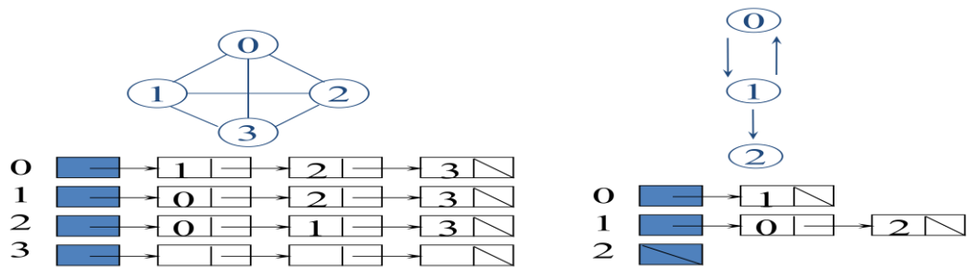
$$ind(vi) = \sum_{j=0}^{n-1} A[j,i] \qquad outd(vi) = \sum_{j=0}^{n-1} A[i,j]$$

The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:

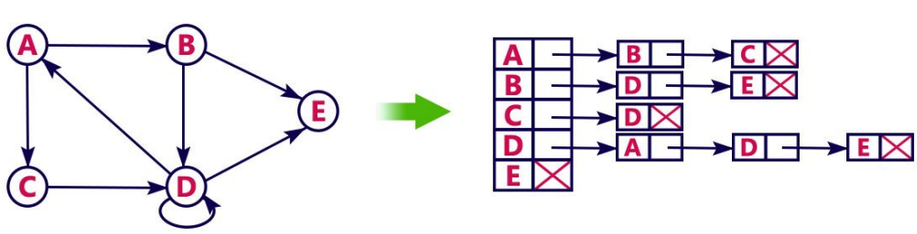


So that we can access the adjacency list for any vertex in O(1) time. Adjlist[i] is a pointer to to first node in the adjacency list for vertex i. Structure is

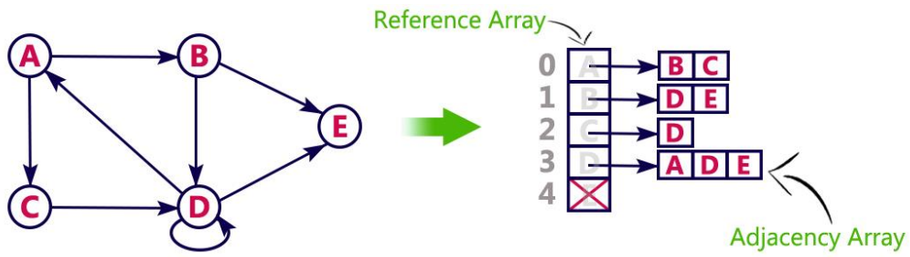
```
#define MAX_VERTICES 50
typedef struct node *node_pointer;
typedef struct node {
   int vertex;
   struct node *link;
};
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */
```

Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

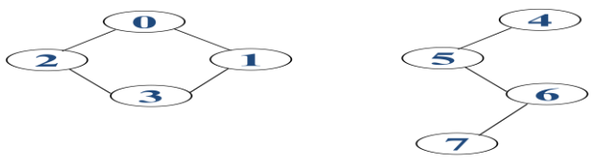


This representation can also be implemented using array



Sequential representation of adjacency list is

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

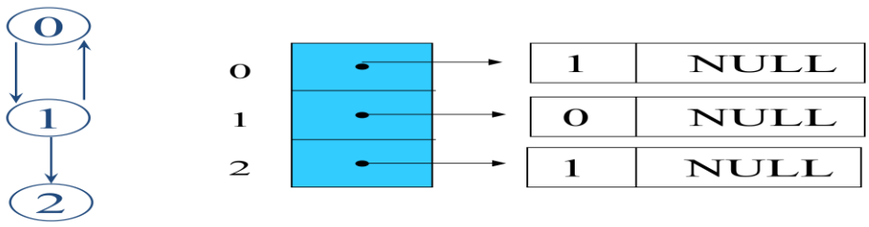


Graph

Instead of chains, we can use sequential representation into an integer array with size n+2e+1. For 0<=i<n, Array[i] gives starting point of the list for vertex I, and array[n] is set to n+2e+1. The adjacent vertices of node I are stored sequentially from array[i].

For an undirected graph with n vertices and e edges, linked adjacency list requires an array of size n and 2e chain nodes. For a directed graph, the number of list nodes is only e. the out degree of any vertex may be determined by counting the number of nodes in its adjacency list. To find in-degree of vertex v, we have to traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.



Determine in-degree of a vertex in a fast way.

3.Adjacency Multilists

In the adjacency-list representation of an undirected graph each edge (u, v) is represented by two entries one on the list for u and the other on tht list for v. As we shall see in some situations it is necessary to be able to determin ie ~ nd enty for a particular edge and mark that edg as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).
For adjacency multilists, node structure is

```
typedef struct edge *edge_pointer;
typedef struct edge {
   short int marked;
   int vertex1, vertex2;
   edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
```

marked	vertex1	vertex2	path1	path2

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4
vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

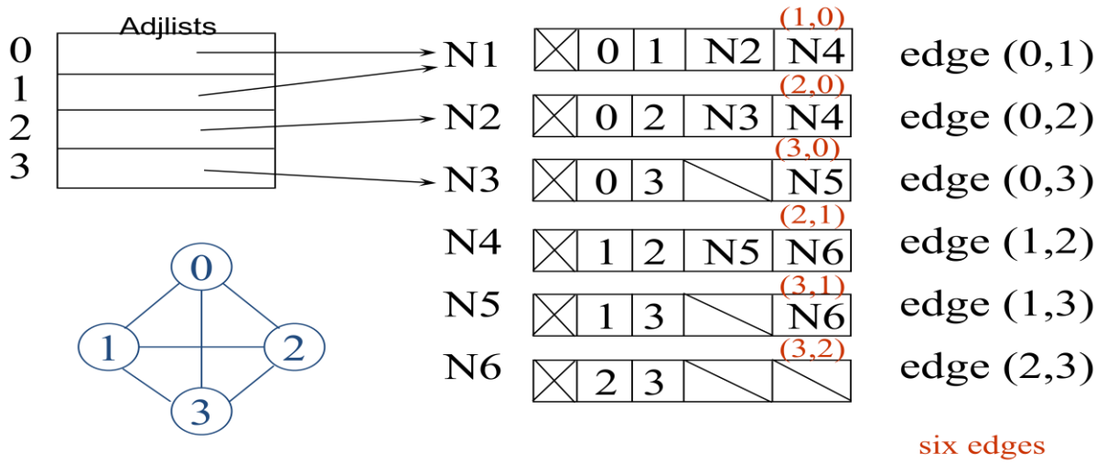
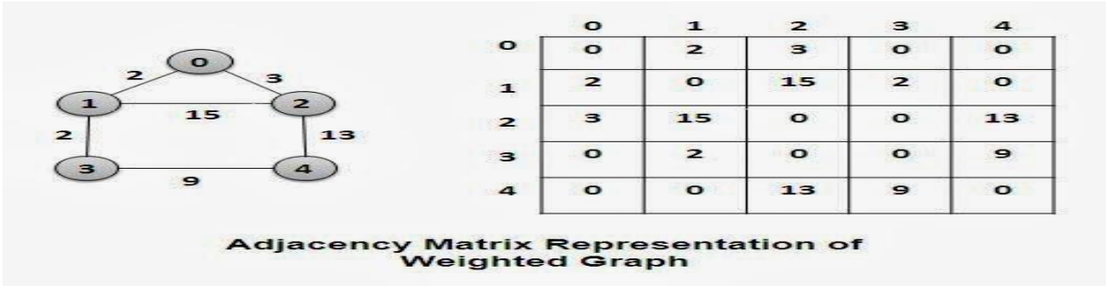


Figure: Adjacency multilists for given graph

4. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- Time complexity
  - Adjacency list: O(|E|)
  - Adjacency matrix: $O(|V|^2)$

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS

traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

This function is best described recursively as in Program.

```
#define FALSE 0
#define TRUE 1
 int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}
```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

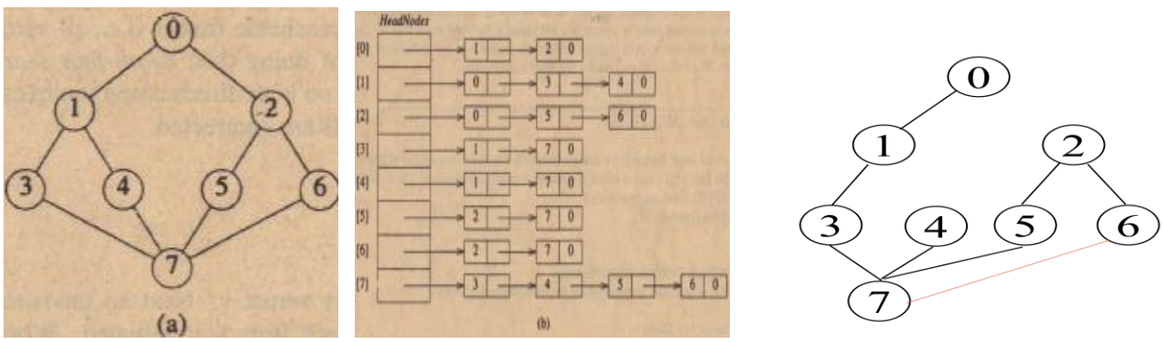


Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is $O(n^2)$.

Breadth-First Search
In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS (Program 6.2) gives the details.

```
typedef struct queue *queue_pointer;
typedef struct queue {
   int vertex;
```

```
   queue_pointer link;
};
void addq(queue_pointer *,
       queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
  node_pointer w;
  queue_pointer front, rear;
  front = rear = NULL;
  printf("%d", v);
  visited[v] = TRUE;
  addq(&front, &rear, v);
while (front) {
    v= deleteq(&front);
    for (w=graph[v]; w; w=w->link)
      if (!visited[w->vertex]) {
        printf("%d", w->vertex);
        addq(&front, &rear, w->vertex);
        visited[w->vertex] = TRUE;
      }
  }
}
```

Steps:

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

Step 1: Define a Queue of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

Step 5: Repeat step 3 and 4 until queue becomes empty.

Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

Analysis Of BFS:

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, $O(n^2)$. If adjacency lists are used the loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

3.Connected Components

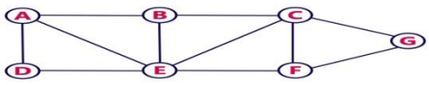
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected(Program 6.3), which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
  for (i=0; i<n; i++) {
     if (!visited[i]) {
        dfs(i);
 printf("\n");       }    } }
```

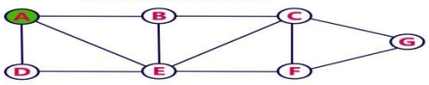
Analysis of Components:

If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used,then the time required is $O(n^2)$

Consider the following example graph to perform DFS traversal

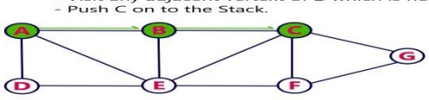


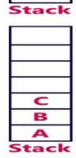
Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



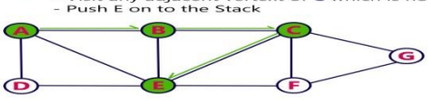
Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

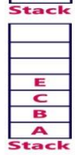
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



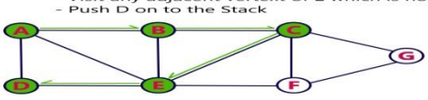


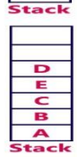
Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack



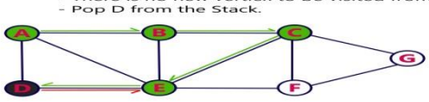


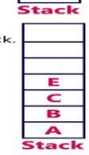
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack



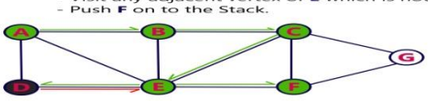


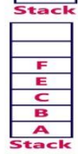
Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.



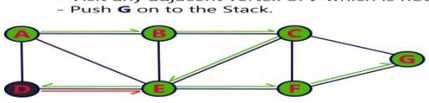


Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

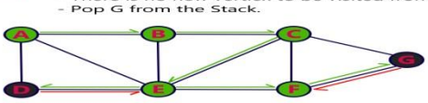


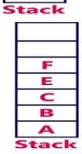


Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

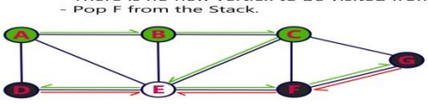


Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

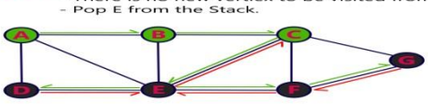


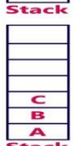


Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

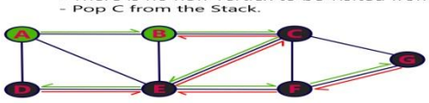


Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.

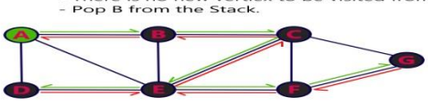




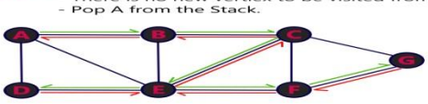
Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



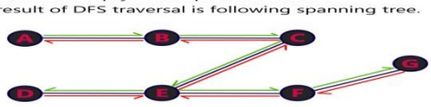
Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



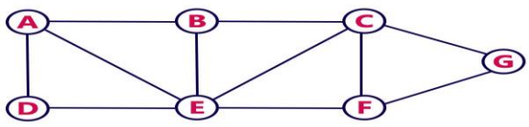
Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

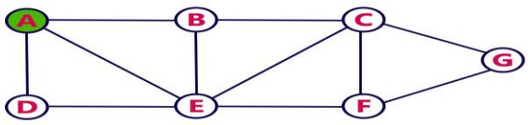


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

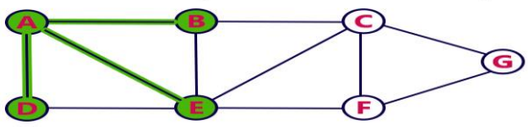


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

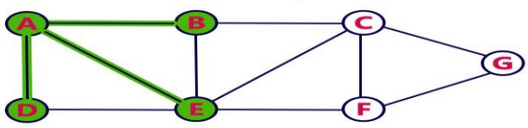


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

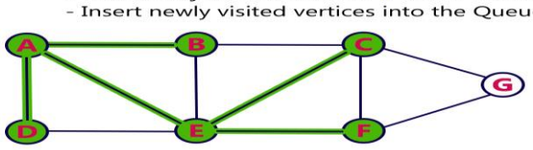


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

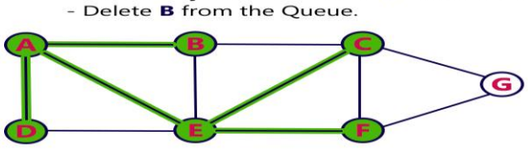


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

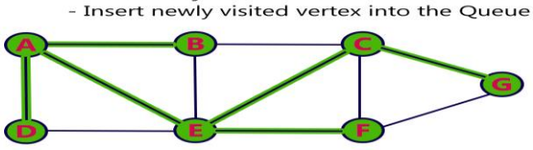


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

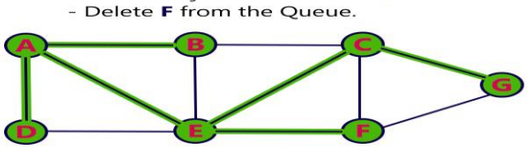


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

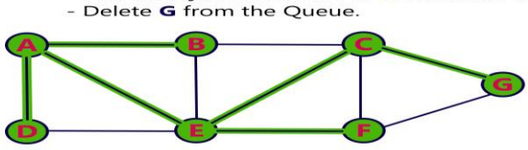


Queue

						G

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...