REPRESENTATION OF GRAPH

Graph is a mathematical structure and finds its application in many areas, where the problem is to be solved by computers. The problems related to graph G must be represented in computer memory using any suitable data structure to solve the same. There are two standard ways of maintaining (representation) a graph G in the memory of a computer.

1. Sequential representation of a graph using adjacent (adjacency matrix)
2. Linked representation of a graph using linked list (adjacency list)

1- ADJACENCY MATRIX REPRESENTATION

The S A of a graph G = (V, E) with n vertices, is an n × n matrix. In this section let us see how a directed graph can be represented using adjacency matrix. Considered a directed graph in Fig. 9.12 where all the vertices are numbered, (1, 2, 3, 4..... etc.)

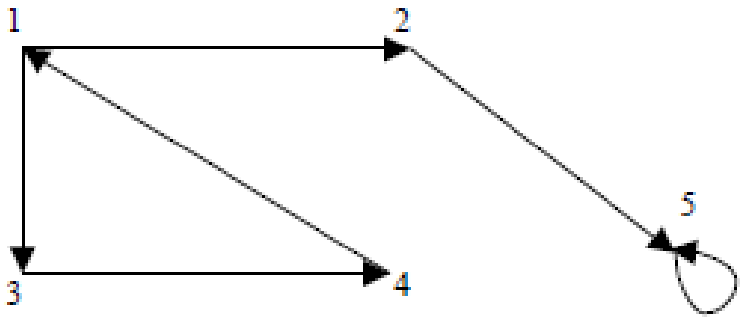


Fig. 9.12

The adjacency matrix A of a directed graph G = (V, E) can be represented (in Fig 9.13) with the following conditions

A_{ij} = 1 {if there is an edge from V_i to V_j or if the edge (i, j) is member of E.}

A_{ij} = 0 {if there is no edge from V_i to V_j}

i \ j	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	1	0	0	0	0
5	0	0	0	0	1

Fig. 9.13

We have seen how a directed graph can be represented in adjacency matrix. Now let us discuss how an undirected graph can be represented using adjacency matrix. Considered an undirected graph in Fig. 9.14

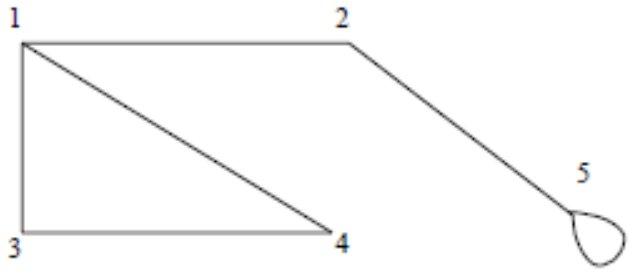


Fig. 9.14

The adjacency matrix A of an undirected graph G = (V, E) can be represented (in Fig 9.15) with the following conditions

$A_{ij} = 1$ {if there is an edge from V_i to V_j or if the edge (i, j) is member of E}

$A_{ij} = 0$ {if there is no edge from V_i to V_j or the edge i, j, is not a member of E}

i \ j	1	2	3	4	5
1	0	1	1	1	0
2	1	0	0	0	1
3	1	0	0	1	0
4	1	0	1	0	0
5	0	1	0	0	1

Fig. 9.15

To represent a weighted graph using adjacency matrix, weight of the edge (i, j) is simply stored as the entry in i th row and j th column of the adjacency matrix. There

are some cases where zero can also be the possible weight of the edge, then we have to store some sentinel value for non-existent edge, which can be a negative value; since the weight of the edge is always a positive number. Consider a weighted graph, Fig. 9.16

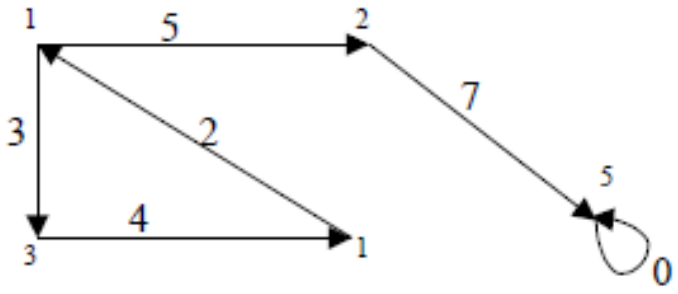


Fig. 9.16

The adjacency matrix A for a directed weighted graph G = (V, E, W_e) can be represented (in Fig. 9.17) as

$A_{ij} = W_{ij}$ { if there is an edge from V_i to V_j then represent its weight W_{ij}.}

$A_{ij} = -1$ { if there is no edge from V_i to V_j}

i \ j	1	2	3	4	5
1	– 1	5	3	– 1	– 1
2	– 1	– 1	– 1	– 1	7
3	– 1	– 1	– 1	4	– 1
4	2	– 1	– 1	– 1	– 1
5	– 1	– 1	– 1	– 1	0

Fig. 9.17

In this representation, n^2 memory location is required to represent a graph with n vertices. The adjacency matrix is a simple way to represent a graph, but it has two disadvantages.

1. It takes n^2 space to represent a graph with n vertices, even for a sparse graph.

2. It takes O (n^2) time to solve the graph problem

2- LINKED LIST REPRESENTATION

In this representation (also called adjacency list representation), we store a graph as a linked structure. First we store all the vertices of the graph in a list and then each adjacent vertices will be represented using linked list node. Here terminal vertex of an edge is stored in a structure node and linked to a corresponding initial vertex in the list. Consider a directed graph in Fig. 9.12, it can be represented using linked list as Fig. 9.18.

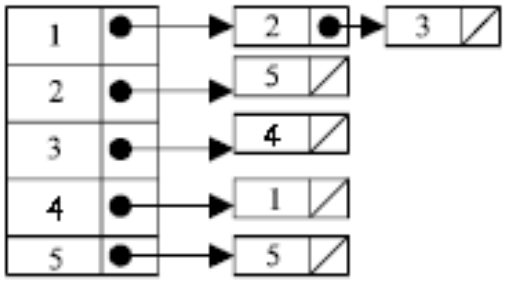


Fig. 9.18

Weighted graph can be represented using linked list by storing the corresponding weight along with the terminal vertex of the edge. Consider a weighted graph in Fig. 9.16, it can be represented using linked list as in Fig. 9.19.

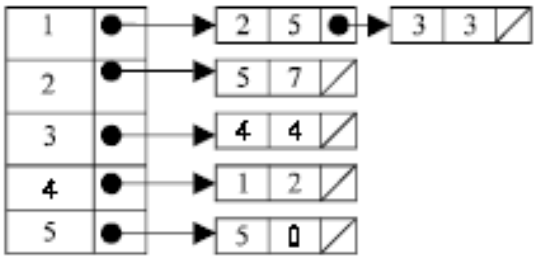


Fig. 9.19

Although the linked list representation requires very less memory as compared to the adjacency matrix, the simplicity of adjacency matrix makes it preferable when graph are reasonably small.

TRAVERSING A GRAPH

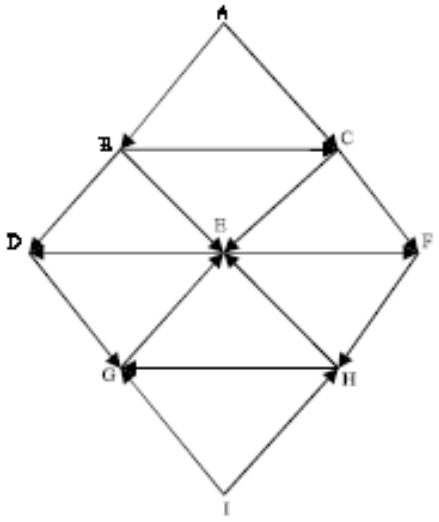
Many application of graph requires a structured system to examine the vertices and edges of a graph G. That is a graph traversal, which means visiting all the nodes of the graph. There are two graph traversal methods.

(a) Breadth First Search (BFS)
(b) Depth First Search (DFS)

(a) BREADTH FIRST SEARCH

Given an input graph G = (V, E) and a source vertex S, from where the searching starts. The breadth first search systematically traverse the edges of G to explore every vertex that is reachable from S. Then we examine all the vertices neighbor to source vertex S. Then we traverse all the neighbors of the neighbors of source vertex S and so on. A queue is used to keep track of the progress of traversing the neighbor nodes.

BFS can be further discussed with an example. Considering the graph G in Fig. 9.20

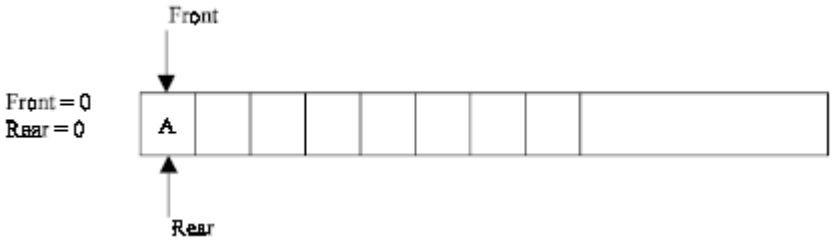


Vertex	*Adjacency list*
A	B, C
B	C, D, E
C	E, F
D	G
E	D, F
F	H
G	E
H	E, G
I	G, H

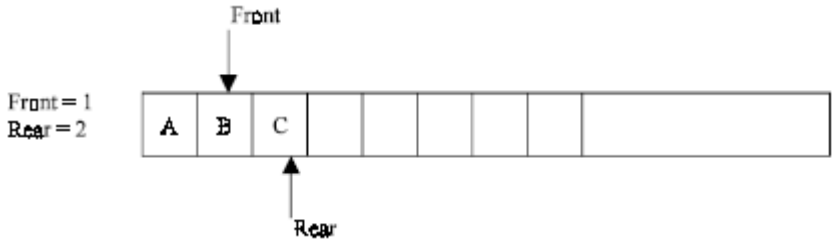
Fig. 9.20

The linked list (or adjacency list) representation of the graph Fig. 9.20 is also shown. Suppose the source vertex is A. Then following steps will illustrate the BFS.

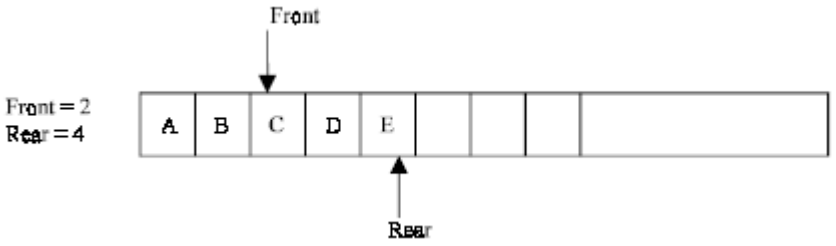
Step 1: Initially push A (the source vertex) to the queue.



Step 2: Pop (or remove) the front element A from the queue (by incrementing front = front +1) and display it. Then push (or add) the neighboring vertices of A to the queue, (by incrementing Rear = Rear +1) if it is not in queue.

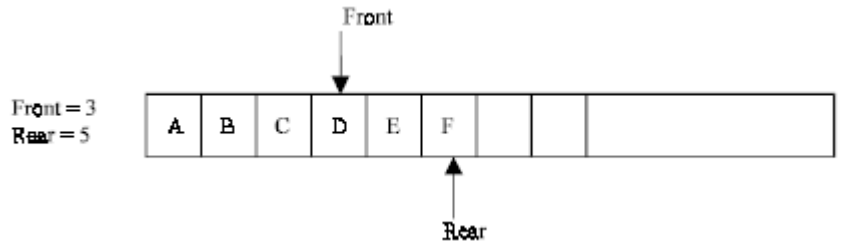


Step 3: Pop the front element B from the queue and display it. Then add the neighboring vertices of B to the queue, if it is not in queue.



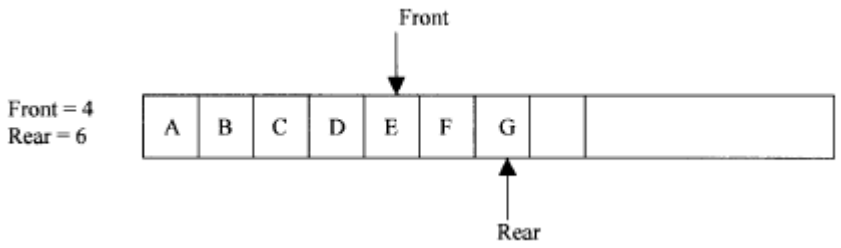
One of the neighboring element C of B is preset in the queue, So C is not added to queue.

Step 4: Remove the front element C and display it. Add the neighboring vertices of C, if it is not present in queue.

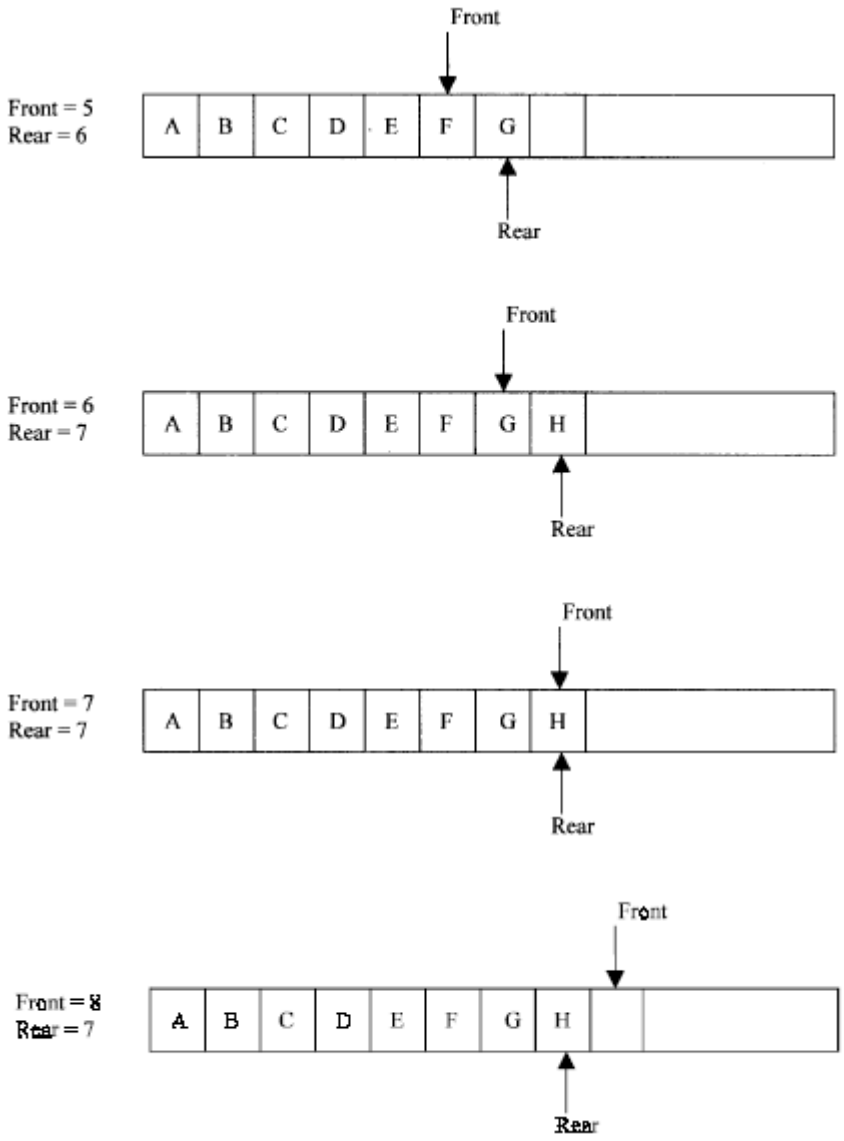


One of the neighboring elements E of C is present in the queue. So E is not added.

Step 5: Remove the front element D, and add the neighboring vertex if it is not present in queue.



Step 6: again the process is repeated (until front > rear). That is remove the front element E of the queue and add the neighboring vertex if it is not present in queue.



So A, B, C, D, E, F, G, H is the BFS traversal of the graph in Fig. 9.20

ALGORITHM

1. Input the vertices of the graph and its edges G = (V, E)
2. Input the source vertex and assign it to the variable S.
3. Add or push the source vertex to the queue.
4. Repeat the steps 5 and 6 until the queue is empty (i.e., front > rear)
5. Pop the front element of the queue and display it as visited.
6. Push the vertices, which is neighbor to just, popped element, if it is not in the queue and displayed (i.e., not visited).
7. Exit.

(b) DEPTH FIRST SEARCH

The depth first search (DFS), as its name suggest, is to search deeper in the graph, whenever possible. Given an input graph G = (V, E) and a source vertex S, from where the searching starts. First we visit the starting node. Then we travel through each node along a path, which begins at S. That is we visit a neighbor vertex of S and again a neighbor of a neighbor of S, and so on. The implementation of BFS is almost same except a stack is used instead of the queue. DFS can be further discussed with an example. Consider the graph in Fig. 9.20 and its linked list representation. Suppose the source vertex is I. The following steps will illustrate the DFS

Step 1: Initially push I on to the stack.

STACK: I

DISPLAY:

Step 2: Pop and display the top element, and then push all the neighbors of popped element (i.e., I) onto the stack, if it is not visited (or displayed or not in the stack).

STACK: G, H

DISPLAY: I

Step 3: Pop and display the top element and then push all the neighbors of popped the element (i.e., H) onto top of the stack, if it is not visited.

STACK: G, E

DISPLAY: I, H

The popped element H has two neighbors E and G. G is already visited, means G is either in the stack or displayed. Here G is in the stack. So only E is pushed onto the top of the stack.

Step 4: Pop and display the top element of the stack. Push all the neighbors of the popped element on to the stack, if it is not visited.

STACK: G, D, F

DISPLAY: I, H, E

Step 5: Pop and display the top element of the stack. Push all the neighbors of the popped element onto the stack, if it is not visited.

STACK: G, D

DISPLAY: I, H, E, F

The popped element (or vertex) F has neighbor(s) H, which is already visited. Then H is displayed, and will not be pushed again on to the stack.

Step 6: The process is repeated as follows.

STACK: G

DISPLAY: I, H, E, F, D

STACK: //now the stack is empty

DISPLAY: I, H, E, F, D, G

So I, H, E, F, D, G is the DFS traversal of graph Fig 9:20 from the source vertex I.

ALGORITHM

1. Input the vertices and edges of the graph G = (V, E).
2. Input the source vertex and assign it to the variable S.
3. Push the source vertex to the stack.
4. Repeat the steps 5 and 6 until the stack is empty.
5. Pop the top element of the stack and display it.
6. Push the vertices which is neighbor to just popped element, if it is not in the queue and displayed (i.e.; not visited).
7. Exit.