7.12 Traversal

Traversal in graph is different from traversal in tree or list because of the following reasons-

(a) There is no first vertex or root vertex in a graph, hence the traversal can start from any vertex. We can choose any arbitrary vertex as the starting vertex. A traversal algorithm will produce different sequences for different starting vertices.

(b) In tree or list, when we start traversing from the first vertex, all the elements are visited but in graph only those vertices will be visited which are reachable from the starting vertex. So if we want to visit all the vertices of the graph, we have to select another starting vertex from the remaining vertices in order to visit all the vertices left.

(c) In tree or list while traversing, we never encounter a vertex more than once while in graph we may reach a vertex more than once. This is because in graph a vertex may have cycles and there may be more than one path to reach a vertex. So to ensure that each vertex is visited only once, we have to keep the status of each vertex whether it has been visited or not.

(d) In tree or list we have unique traversals. For example if we are traversing a binary tree in inorder there can be only one sequence in which vertices are visited. But in graph, for the same technique of traversal there can be different sequences in which vertices can be visited. This is because there is no natural order among the successors of a vertex, and thus the successors may be visited in different orders producing different sequences. The order in which successors are visited may depend on the implementation.

Like binary trees, in graph also there can be many methods by which a graph can be traversed but two of them are standard and are known as breadth first search and depth first search.

7.12.1 Breadth First Search

In this technique, first we visit the starting vertex and then visit all the vertices adjacent to the starting vertex. After this we pick these adjacent vertices one by one and visit their adjacent vertices and this process goes on. This traversal is equivalent to level order traversal of trees.Let us take a graph and traverse it using breadth first traversal.

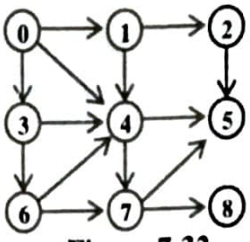
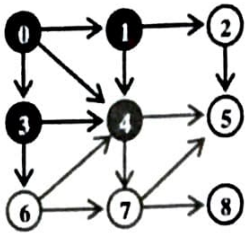


Figure 7.32

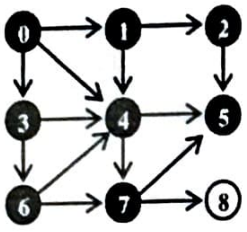
Let us take vertex 0 as the starting vertex. First we will visit the vertex 0. Then we will visit all vertices adjacent to vertex 0 i.e. 1, 4, 3. Here we can visit these three vertices in any order. Suppose we visit the vertices in order 1, 3, 4. Now the traversal is-

0 1 3 4



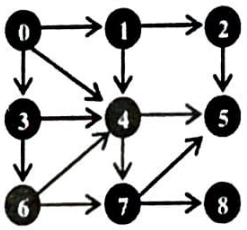
Now first we visit all the vertices adjacent to 1, then all the vertices adjacent to 3 and then all the vertices adjacent to 4. So first we will visit 2, then 6 and then 5, 7. Note that vertex 4 is adjacent to vertices 1 and 3, but it has already been visited so we've ignored it. Now the traversal is -

0 1 3 4 2 6 5 7



Now we will visit one by one all the vertices adjacent to vertices 2,6,5,7. We can see that vertex 5 is adjacent to vertex 2, but it has already been visited so we will just ignore it and proceed further. Now vertices adjacent to vertex 6 are vertices 4 and 7 which have already been visited so ignore them also. Vertex 5 has no adjacent vertices. Vertex 7 has vertices 5 and 8 adjacent to it out of which vertex 8 has not been visited, so visit vertex 8. Now the traversal is-

0 1 3 4 2 6 5 7 8



Now we have to visit vertices adjacent to vertex 8 but there is no vertex adjacent to vertex 8 so our procedure stops.

This was the traversal when we take vertex 0 as the starting vertex. Suppose we take vertex 1 as the starting vertex. Then applying above technique, we will get the following traversal-

1 2 4 5 7 8

Here are different traversals when we take different starting vertices.

Start Vertex	Traversal
0	0 1 3 4 2 6 5 7 8
1	1 2 4 5 7 8
2	2 5
3	3 4 6 5 7 8
4	4 5 7 8
5	5
6	6 4 7 5 8
7	7 5 8
8	8

Note that these traversals are not unique, there can be different traversals depending on the order in which we visit the successors.

We can see that all the vertices are not visited in some cases. The vertices which are visited are those vertices which are reachable from starting vertex. So to make sure that all the vertices are visited we need to repeat the same procedure for each unvisited vertex in the graph. Breadth first search is implemented through queue.

7.12.1.1 Implementation of Breadth First Search using queue

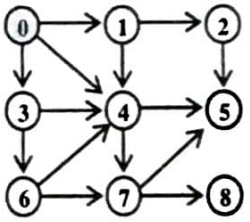
During the algorithm, any vertex will be in one of the three states - initial, waiting, visited. At the start of the algorithm all vertices will be in initial state, when a vertex will be inserted in the queue its state will change from initial to waiting. When a vertex will be deleted from queue and visited, its state will change from waiting to visited. The procedure is as-

Initially queue is empty, and all vertices are in initial state.

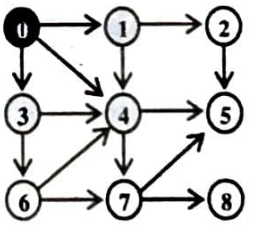
1. Insert the starting vertex into the queue, change its state to waiting.
2. Delete front element from the queue and visit it, change its state to visited.
3. Look for the adjacent vertices of the deleted element, and from these insert only those vertices into the queue which are in the initial state. Change the state of all these inserted vertices from initial to waiting.
4. Repeat steps 2, 3 until the queue is empty.

Let us take vertex 0 as the starting vertex for traversal in the graph of figure 7.32. In each step we will show the traversal and the contents of queue. In the figure, the different states of the vertices are shown by different colors. White color indicates initial state, grey indicates waiting state and black indicates visited state.

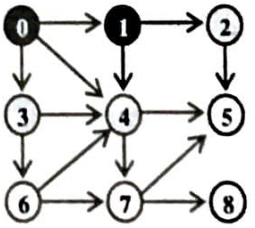
(i) Insert the vertex 0 into the queue.
Queue : 0



(ii) Delete vertex 0 from queue, and visit it.
Traversal : 0
Vertices adjacent to vertex 0 are vertices 1, 3, 4 and all of these are in initial state, so insert them into the queue.
Queue : 1, 3, 4

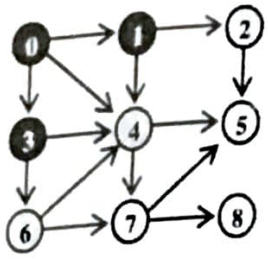


(iii) Delete vertex 1 from queue, and visit it.
Traversal : 0, 1
Vertices adjacent to vertex 1 are vertices 2 and 4. Vertex 4 is in waiting state because it is in the queue, so it is not inserted in the queue. Vertex 2 is in initial state, so insert it into the queue.
Queue : 3, 4, 2

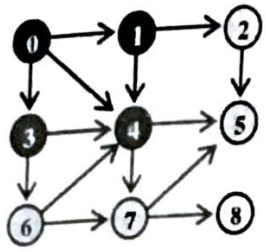


Here we can see why we have taken the concept of waiting state. Vertex 4 is in waiting state i.e. it is already present in the queue so it is not inserted into the queue. The concept of waiting state helps us avoid insertion of duplicate vertices in the queue.

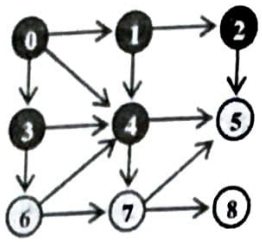
(iv) Delete vertex 3 from queue, and visit it.
Traversal : 0, 1, 3
Vertices adjacent to vertex 3 are vertices 4 and 6.
Vertex 4 is in the waiting state and vertex 6 is in initial state, so insert only vertex 6 into the queue.
Queue : 4, 2, 6



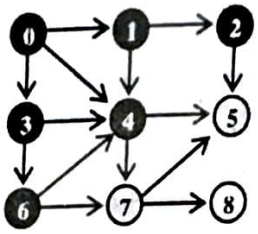
(v) Delete vertex 4 from the queue, and visit it.
Traversal : 0, 1, 3, 4
Vertices adjacent to vertex 4 are vertices 5 and 7, and both are in initial state so insert them into the queue.
Queue : 2, 6, 5, 7



(vi) Delete vertex 2 from the queue, and visit it.
Traversal : 0, 1, 3, 4, 2
Vertex adjacent to vertex 2 is vertex 5.
Vertex 5 is in waiting state because it is already present in the queue, so it is not inserted into the queue.
Queue : 6, 5, 7

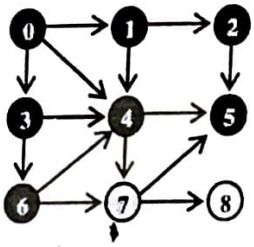


(vii) Delete vertex 6 from the queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6,
Vertices adjacent to vertex 6 are vertices 4 and 7.
Vertex 4 is in visited state and vertex 7 is in waiting state so nothing is inserted into the queue.
Queue : 5, 7

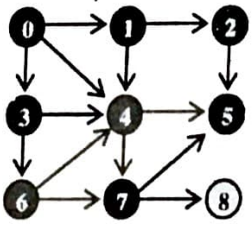


Here we can see why we have taken the concept of visited state. Vertex 4 is in visited state i.e. it has been included in the traversal, so there is no need of its insertion into the queue. The concept of visited state helps us avoid visiting a vertex more than once.

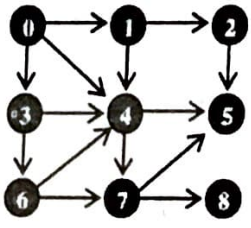
(viii) Delete vertex 5 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5
Vertex 5 has no adjacent vertices.
Queue : 7



(ix) Delete 7 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5, 7
Vertices adjacent to vertex 7 are vertices 5 and 8.
Vertex 5 is in visited state and vertex 8 is in initial state, so insert only vertex 8 into the queue.
Queue : 8



(x) Delete vertex 8 from queue, and visit it.
Traversal : 0, 1, 3, 4, 2, 6, 5, 7, 8
There is no vertex adjacent to vertex 8
Queue : EMPTY



Now the queue is empty so we will stop our process. This way we get the breadth first traversal when vertex 0 is taken as the starting vertex.

```
/*P7.6 Program for traversing a directed graph through BFS, visiting only those vertices that
are reachable from start vertex*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define waiting 2
#define visited 3
int n;      /*Number of vertices in the graph*/
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*can be initial, waiting or visited*/
void create_graph();
void BF_Traversal();
void BFS(int v);
int queue[MAX],front = -1,rear = -1;
void insert_queue(int vertex);
int delete_queue();
int isEmpty_queue();

main()
{
        create_graph();
        BF_Traversal();
}/*End of main()*/
void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting vertex for Breadth First Search : ");
        scanf("%d",&v);
        BFS(v);
}/*End of BF_Traversal()*/
void BFS(int v)
{
        int i;
        insert_queue(v);
        state[v]=waiting;
        while(!isEmpty_queue())
        {
                v = delete_queue();
                printf("%d ",v);
                state[v]=visited;
                for(i=0; i<n; i++)
                {
                        /*Check for adjacent unvisited vertices*/
                        if(adj[v][i]==1 && state[i]==initial)
                        {
                                insert_queue(i);
                                state[i] = waiting;
                        }
                }
        }
        printf("\n");
}/*End of BFS()*/
void insert_queue(int vertex)
{
        if(rear==MAX-1)
                printf("Queue Overflow\n");
        else
```

```
        {
                if(front==-1)  /*If queue is initially empty*/
                        front = 0;
                rear = rear+1;
                queue[rear] = vertex ;
        }
}/*End of insert_queue()*/
int isEmpty_queue()
{
        if(front==-1 || front>rear)
                return 1;
        else
                return 0;
}/*End of isEmpty_queue()*/
int delete_queue()
{
        int del_item;
        if(front==-1 || front>rear)
        {
                printf("Queue Underflow\n");
                exit(1);
        }
        del_item = queue[front];
        front = front+1;
        return del_item;
}/*End of delete_queue()*/
```

The function `create_graph()` is same as in program P7.2.

This process can visit only those vertices which are reachable from the starting vertex. For example if we start traversing from vertex 4 instead of vertex 0, then all the vertices will not be visited. The traversal would be – 4 5 7 8.

If we want to visit all the vertices, then we can take any unvisited vertex as starting vertex and again start breadth first search from there. This process will continue until all the vertices are visited. So if want to visit all the vertices when 4 is the start vertex, then we have to select another start vertex after visiting 5, 7 and 8. Suppose we take 0 as the next start vertex, so now the traversal would be – 4 5 7 8 0 1 3 2 6. Now all the vertices are visited so there is no need to choose any other start vertex.

In the program, we have to make a small addition in the `BF_Traversal()` function. After the call to `BFS()` with start vertex, we will check all vertices one by one in a loop, and if we get any vertex that is in initial state we will call `BFS()` with that vertex.

```
void BF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting vertex for Breadth First Search : ");
        scanf("%d", &v);
        BFS(v);
        for(v=0; v<n; v++)
                if(state[v]==initial)
                        BFS(v);
}/*End of BF_Traversal()*/
```

Now suppose that while

7.12.2 Depth First Search

Traversal using depth first search is like traveling a maze. We travel along a path in the graph and when a dead end comes we backtrack. This technique is named so because search proceeds deeper in the graph i.e. we traverse along a path as deep as we can.

First the starting vertex is visited and then we will pick up any path that starts from the starting vertex and visit all the vertices in this path till we reach a dead end. This means visit the starting vertex(say v1) and then any vertex adjacent to it(say v2). Now if v2 has any vertex adjacent to it which has not been visited then visit it. and so on till we come to a dead end. Dead end means that we reach a vertex which does not have any adjacent vertex or all of its adjacent vertices have been visited. After reaching the dead end we will backtrack along the path that we have visited till now. Suppose the path that we've traversed is v1-v2-v3-v4-v5. After traversing v5 we reached a dead end. Now we will move backwards till we reach a vertex that has any unvisited adjacent vertex. We move back and reach v4 but see that it has no unvisited adjacent vertices so we will reach vertex v3. Now if v3 has an unvisited vertex adjacent to it, we will pick up a path that starts from v3 and visit it until we reach a dead end. Then again we will backtrack. This procedure finishes when we reach the starting vertex and there are no vertices adjacent to it which have to be visited.

Let us take a graph and perform a depth first search taking vertex 0 as the start vertex. The successors of a vertex can be visited in any order.

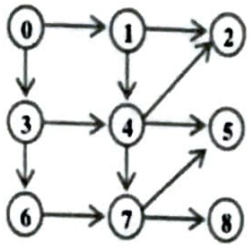
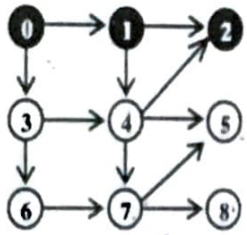


Figure 7.39

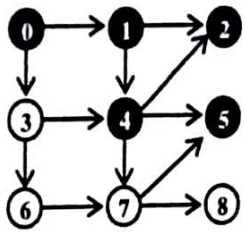
First, we will visit vertex 0. Vertices adjacent to vertex 0 are 1 and 3. Suppose we visit vertex 1. Now we look at the adjacent vertices of 1; from the two adjacent vertices 2 and 4 we choose to visit 2. Till now the traversal is -

0 1 2



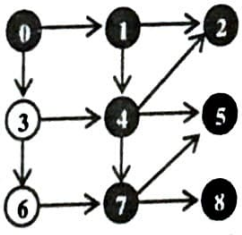
There is no vertex adjacent to vertex 2, means we have reached the end of the path or a dead end from where we can't go forward. So we will move backward. We reach vertex 1 and see if there is any vertex adjacent to it, and not visited yet. Vertex 4 is such a vertex and therefore we visit it. Now vertices 5 and 7 are adjacent to 4 and unvisited, and from these we choose to visit vertex 5. Till now the traversal is -

0 1 2 4 5



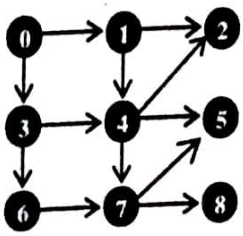
There is no vertex adjacent to vertex 5 so we will backtrack. We reach vertex 4, and its unvisited adjacent vertex is 7 so we visit it. Now vertex 8 is the only unvisited vertex adjacent to 7 so we visit it. Till now the traversal is -

0 1 2 4 5 7 8



Vertex 8 has no unvisited adjacent vertex so we backtrack and reach vertex 7. Now vertex 7 also has no unvisited adjacent vertex so we backtrack and reach vertex 4. Vertex 4 also has no unvisited adjacent vertex so we backtrack and reach vertex 1. Vertex 1 also has no unvisited adjacent vertex so we backtrack and reach vertex 0. Vertex 3 is adjacent to vertex 0 and is unvisited so we visit vertex 3. Vertex 6 is adjacent to vertex 3 and is unvisited so we visit vertex 6. Till now the traversal is -

0 1 2 4 5 7 8 3 6



Now vertex 6 has no unvisited adjacent vertex so we backtrack and reach vertex 3. Vertex 3 also has no unvisited adjacent vertex so we backtrack and reach vertex 0. Vertex 0 also has no unvisited adjacent vertex left and it is the start vertex so now we can't backtrack and hence our traversal finishes. The traversal is- 0 1 2 4 5 7 8 3 6.

Depth first search can be implemented through stack or recursively.

7.12.2.1 Implementation of Depth First Search using stack

During the algorithm any vertex will be in one of the two states – initial or visited. At the start of the algorithm, all vertices will be in initial state, and when a vertex will be popped from stack its state will change to visited. The procedure is as-

Initially stack is empty, and all vertices are in initial state.

1. Push starting vertex on the stack.
2. Pop a vertex from the stack.
3. If popped vertex is in initial state, visit it and change its state to visited. Push all unvisited vertices adjacent to the popped vertex.
4. Repeat steps 2 and 3 until stack is empty.

There is no restriction on the order in which the successors of a vertex are visited and so we can push the successors of a vertex in any order. Here we are pushing the successors in descending order of their numbers. For example if the successors are 2, 4, 6 then we will push 6 first and then 4 and then 2. Let us find the depth first traversal of the following graph using stack.

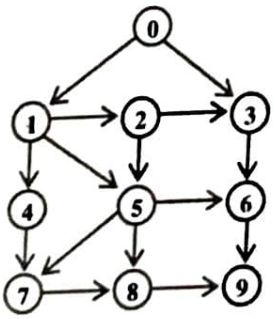


Figure 7.40

Start vertex is 0, so initially push vertex 0 on the stack.

Pop 0: Visit 0	Push 3, 1	Stack : 3 1
Pop 1: Visit 1	Push 5, 4, 2	Stack : 3 5 4 2
Pop 2: Visit 2	Push 5, 3	Stack : 3 5 4 5 3
Pop 3: Visit 3	Push 6	Stack : 3 5 4 5 6
Pop 6: Visit 6	Push 9	Stack : 3 5 4 5 9
Pop 9: Visit 9		Stack : 3 5 4 5
Pop 5: Visit 5	Push 8, 7	Stack : 3 5 4 8 7
Pop 7: Visit 7	Push 8	Stack : 3 5 4 8 8
Pop 8: Visit 8		Stack : 3 5 4 8
Pop 8:		Stack : 3 5 4
Pop 4: Visit 4		Stack : 3 5
Pop 5:		Stack : 3
Pop 3:		Stack : Empty

Depth first traversal is : 0 1 2 3 6 9 5 7 8 4

In BFS, if a vertex was already present in the queue then it was not inserted again in the queue. So there we changed the state of vertex from initial to waiting as soon as it was inserted in the queue, and never inserted a

vertex in the queue that was in waiting state. In DFS, we don't have the concept of waiting state and so there may be multiple copies of a vertex in the stack.

If we don't insert a vertex already present in the stack, then we will not be able to visit the vertices in depth first search order. For example if we traverse the graph of figure 7.40 in this manner then we get the traversal as 0 1 2 4 7 8 9 5 6 3, which is clearly not in depth first order.

```
/*P7.7 Program for traversing a directed graph through DFS, visiting only vertices reachable
from start vertex*/
#include<stdio.h>
#include<stdlib.h>
#define MAX 100
#define initial 1
#define visited 2
int n;    /* Number of nodes in the graph */
int adj[MAX][MAX]; /*Adjacency Matrix*/
int state[MAX]; /*Can be initial or visited */
void DF_Traversal();
void DFS(int v);
void create_graph();
int stack[MAX];
int top = -1;
void push(int v);
int pop();
int isEmpty_stack();

main()
{
        create_graph();
        DF_Traversal();
}/*End of main()*/

void DF_Traversal()
{
        int v;
        for(v=0; v<n; v++)
                state[v]=initial;
        printf("Enter starting node for Depth First Search : ");
        scanf("%d",&v);
        DFS(v);
}/*End of DF_Traversal( )*/

void DFS(int v)
{
        int i;
        push(v);
        while(!isEmpty_stack())
        {
                v = pop();
                if(state[v]==initial)
                {
                        printf("%d ",v);
                        state[v]=visited;
                }
                for(i=n-1; i>=0; i--)
                {
                        if(adj[v][i]==1 && state[i]==initial)
                                push(i);
                }
        }
}/*End of DFS( )*/

void push(int v)
{
        if(top==(MAX-1))
        {
```

```
                printf("Stack Overflow\n");
                return;
        }
        top=top+1;
        stack[top] = v;
}/*End of push()*/
int pop()
{
        int v;
        if(top==-1)
        {
                printf("Stack Underflow\n");
                exit(1);
        }
        else
        {
                v = stack[top];
                top=top-1;
                return v;
        }
}/*End of pop()*/
int isEmpty_stack( )
{
  if(top==-1)
          return 1;
  else
          return 0;
}/*End if isEmpty_stack()*/
void create_graph()
{
        int i,max_edges,origin,destin;
        printf("Enter number of nodes : ");
        scanf("%d",&n);
        max_edges = n*(n-1);
        for(i=1; i<=max_edges; i++)
        {
                printf("Enter edge %d( -1 -1 to quit ) : ",i);
                scanf("%d %d",&origin,&destin);
                if((origin==-1) && (destin==-1))
                        break;
                if(origin>=n || destin>=n || origin<0 || destin<0)
                {
                        printf("Invalid edge!\n");
                        i--;
                }
                else
                        adj[origin][destin] = 1;
        }
}/*End of create_graph()*/
```

If all vertices are not reachable from the start vertex then we need to repeat the procedure taking some other start vertex. This is similar to the process we had done in breadth first search. In the function `DF_Traversal()`, we will add a loop which will check the state of all vertices after the first DFS.

As in BFS, here also we can assign a predecessor to each vertex and get the predecessor subgraph which would be a spanning tree or spanning forest of the given graph depending on the reachability of all vertices from the start vertex.