3. Else
    1. DELETE_SL_ANY (UGptr[V_i], V_j) //Delete V_i and V_j from the adjacency list //of V_j and V_i, respectively //Delete V_j from the adjacency list of V_i
    2. DELETE_SL_ANY(UGptr[V_j],V_i) //Delete V_i from the adjacency list of V_j
4. EndIf
5. Stop

Now, let us describe the algorithm DELETE_EDGE_LL_DG to delete an edge from a digraph.

Algorithm DELETE_EDGE_LL_DG(V_i, V_j)

Input: DGptr, the pointer to the graph. <V_i, V_j>, the edge to be deleted from vertex V_i to V_j.

Output: The graph without edge from vertex V_i to V_j.

Steps:

1. Let N = number of vertices in the graph
2. If ($V_i > N$) or ($V_j > N$) then
    1. Print "Vertex does not exist: Error in edge removal"
3. Else
    1. DELETE_SL_ANY (UGptr[V_i], V_j) //Delete V_j from the adjacency list of V_i
4. EndIf
5. Stop

Graph traversals

Traversing a graph means visiting all the vertices in the graph exactly once. For the sake of simplicity, we will assume that the graph is connected.

Several methods are known to traverse a graph systematically, out of them two methods are accepted as standard and will be discussed in details in this section. These methods are called *breadth first search* (BFS) and *depth first search* (DFS). With these traversals, starting from a given node we can visit all the nodes which are reachable from that starting node.

Depth first search (DFS) traversal is similar to the inorder traversal of a binary tree. Starting from a given node, this traversal visits all the nodes up to the deepest level and so on.

Figure 8.18(a) shows the DFS traversals on two graphs $G1$ and $G2$ starting from the vertex v_1. The path of traversals are indicated with thick lines. The sequence of visiting of the vertices can be obtained as:

$$\text{DFS}(G1) = v_1\text{-}v_2\text{-}v_5\text{-}v_7\text{-}v_4\text{-}v_8\text{-}v_6\text{-}v_3$$

$$\text{DFS}(G2) = v_1\text{-}v_2\text{-}v_5\text{-}v_7\text{-}v_4\text{-}v_8\text{-}v_3\text{-}v_6$$

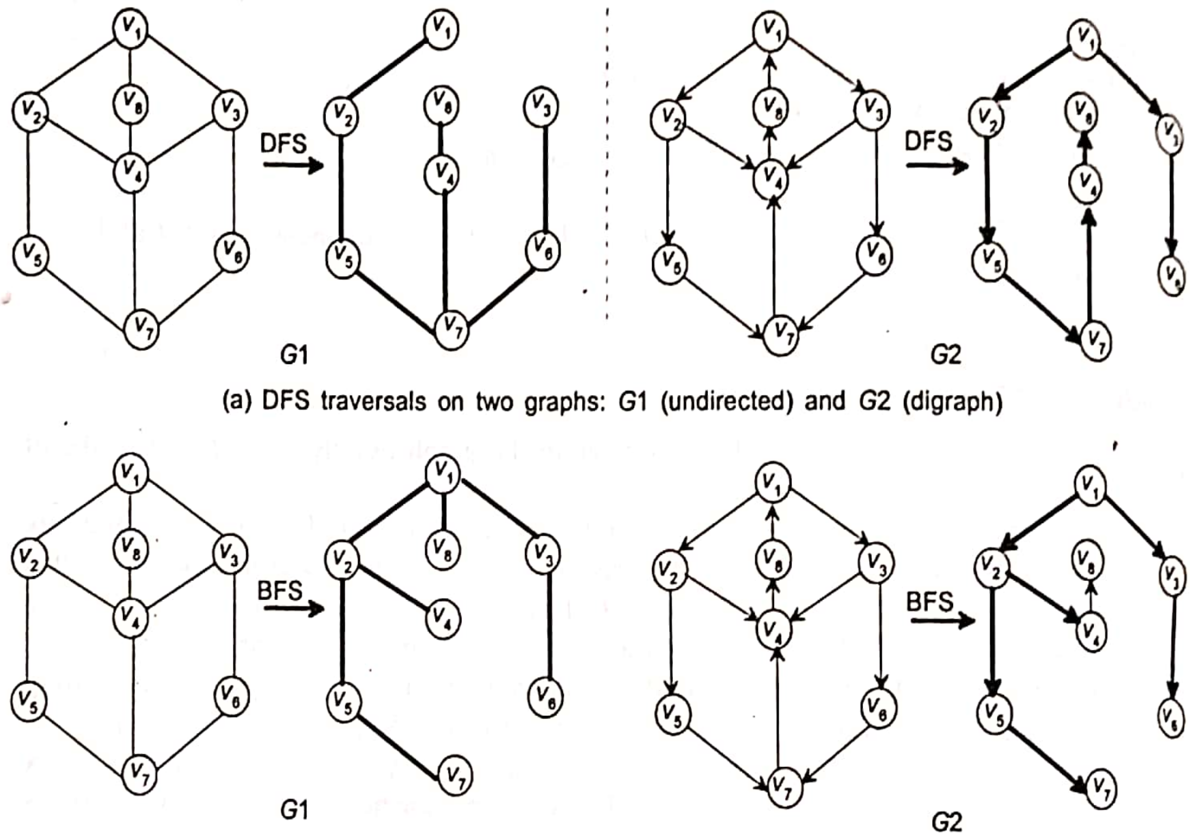
From the above traversals, the traversals take place up to the deepest level, for example, v_1-v_2 -v_5-v_7, then v_4-v_8 and v_6-v_3 (in $G1$), v_3-v_6 (in $G2$). It can be noted that the sequence of visit depends on the depth we choose first and hence the order of visiting the vertices may not be unique.

Another standard graph traversal method is the breadth first search (BFS). This traversal is very similar to the level-by-level traversal of a tree. Here, any vertex in the level i will be visited only after the visit of all the vertices in its preceding level, that is, at $i - 1$. BFS traversal is roughly analogous to the preorder traversal of a tree. Here, suppose we are to visit the vertex

v_i and v_i has $v_{i1}, v_{i2}, \ldots, v_{in}$ as the adjacent vertices of it. Then the BFS traversal can be defined recursively as stated below:

Traverse(v_i)
  Process(v_i)
  Traverse(v_{i1})
  Traverse(v_{i2})
  . . .
  Traverse(v_{in})
End of traversal (v_i)

The BFS traversals on two graphs $G1$ and $G2$ starting from the vertex v_1 are shown as in Figure 8.18(b).



(a) DFS traversals on two graphs: G1 (undirected) and G2 (digraph)

(b) BFS traversals on two graphs: G1 (undirected) and G2 (digraph)

Fig. 8.18 DFS and BFS traversals.

From Figure 8.18(b) the following order of visit of vertices with the BFS traversals can easily be revealed:

$$\text{BFS}(G1) = v_1\text{-}v_2\text{-}v_8\text{-}v_3\text{-}v_5\text{-}v_4\text{-}v_6\text{-}v_7$$

$$\text{BFS}(G2) = v_1\text{-}v_2\text{-}v_3\text{-}v_5\text{-}v_4\text{-}v_6\text{-}v_7\text{-}v_8$$

It can be noted that, in $G1$ (undirected graph), v_1 is in the first level and visited first, then v_2, v_3 and v_8 are visited which are in the same level and next to v_1; similarly v_4, v_5 and v_6 and so on. In $G2$ (digraph), v_2 and v_3 are in same level, v_4, v_5 and v_6 are in one level; again v_7, v_8 are in one level.

Note: 1. DFS and BFS traversals result an acyclic graph.

2. DFS traversal and BFS traversal on the same graph do not give the same order of visit of vertices, or more precisely they generally result in different acyclic graphs.

3. DFS or BFS traversals can be employed to decide whether there is a path from a given vertex v_i to another vertex v_j and if it exists then traces the path from v_i to v_j.

4. DFS and BFS traversals cannot visit a vertex v_i say, if there is no path from starting vertex to that v_i.

In the next two sections, we will describe the details of the algorithms to implement the above mentioned traversals on any graph.

DFS traversal

It is already pointed that the DFS traversal is similar to the inorder traversal of a tree. The general approach behind DFS traversal beginning at vertex v is that visit the vertex v first. Then visit all the vertices along a path which begins at v. This means that, visit the vertex v then the vertex immediate adjacent to v, let it be v_x. Next, if v_x has an immediate adjacent v_y say, then visit it and so on, till there is a 'dead end'. This results a path P, y-v_x-v_y ... *Dead end means* a vertex which do not have immediate adjacent or its immediate adjacent already been visited. After coming to a 'dead end' we backtrack along P to v to see if it has another adjacent vertex other than v_x and then continue the same from it else from the adjacent of the adjacent (which is not visited earlier), and so on.

A stack can be used to maintain the track of all paths from any vertex so as to help backtracking. Initially the starting vertex will be PUSHed onto the stack (let the name of the stack be OPEN). To visit a vertex, we are to POP a vertex from OPEN, and then PUSH all the adjacent vertices onto it. A list, VISIT can be maintained to store the vertices already visited. When a vertex is popped, whether it is already visited or not that can be known by searching the list VISIT; if the vertex is already visited, we will simply ignore it and we will POP the stack for the next vertex to be visited. This procedure will be continued till the stack is not empty. The above idea is expressed more precisely as below:

Algorithm DFS (informal description)

1. Push the starting vertex into the stack OPEN
2. While OPEN is not empty do
    1. POP a vertex V
    2. If V is not in VISIT
        1. Visit the vertex V
        2. Store V in VISIT
        3. Push all the adjacent vertex of V onto OPEN
    3. EndIf
3. EndWhile
4. Stop

The detailed version of the above algorithm, when the given graph is represented using linked lists, is stated as in algorithm DFS_LL.

Algorithm DFS_LL(V)

Input: V, the starting vertex

Output: A list VISIT giving the order of visit of vertices during traversal.

Data structure: Linked structure of graph. GPTR is the pointer to a graph.

Steps:

1. If Gptr = NULL then
   1. Print "Graph is empty"
   2. Exit
2. EndIf
3. $u = V$ //Start from V
4. OPEN.PUSH(u) //Push the starting vertex into OPEN
5. While (OPEN.TOP ≠ NULL) do //Till the stack is not empty
   1. u = OPEN.POP( ) //Pop the top element from OPEN
   2. If (SEARCH_SL(VISIT, u) = FALSE) then //If u is not in VISIT
      1. INSERT_SL_END(VISIT, u) //Store u in VISIT
      2. ptr = GPTR[u] //To push all the adjacent vertices of u //into OPEN
      3. While (ptr.LINK ≠ NULL) do
         1. vptr = ptr.LINK
         2. OPEN.PUSH (vptr.LABEL)
      4. EndWhile
   3. EndIf
6. EndWhile
7. Return(VISIT)
8. Stop

Assignment 8.6

1. For the graphs as shown in Figure 8.19 trace the algorithm DFS_LL to obtain the DFS traversal starting from vertex labelled 1.
   Notice that the vertices will be pushed onto the stack from left to right order, that is, left-most vertex will be pushed first, and so on.
2. Repeat Problem 1, but starting with vertex labelled 9.

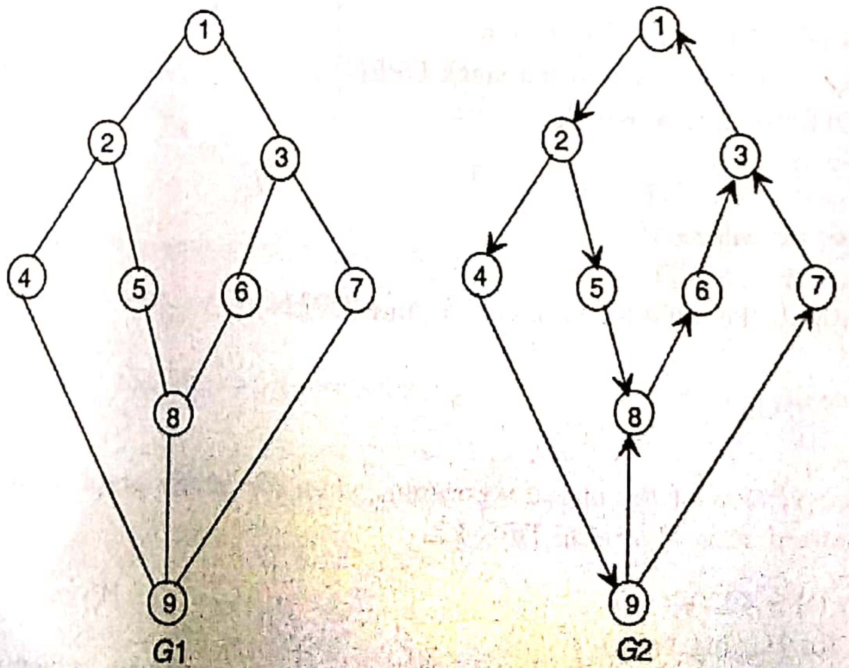


Fig. 8.19 Graphs for Assignment 8.6.

BFS traversal

Now, let us describe the breadth first search (BFS) on a graph. The implementation idea of the BFS traversal is almost the same as the DFS traversal except that in BFS we will use a queue structure instead of a stack structure as in DFS. Let us denote the queue as OPENQ to use it in BFS method. VISIT is the list to store the ordering of vertices during the BFS traversal. The detailed version of the BFS traversal is stated in the algorithm BFS_LL as stated below:

Algorithm BFS_LL(V)

Input: V is the starting vertex.

Output: A list VISIT giving the order of visit of vertices during the traversal.

Data structure: Linked structure of graph. Gptr is the pointer to a graph.

Steps:

```
1. If (GPTR = NULL) then
   1. Print "Graph is empty"
   2. Exit
2. EndIf
3. u = V
4. OPEN.ENQUEUE(u)                          //Enter the starting vertex into OPENQ
5. While (OPENQ.STATUS( ) ≠ EMPTY) do       //Till the OPENQ is not empty
   1. u = OPENQ.DEQUEUE ( )                 //Delete the item from OPENQ
   2. If (SEARCH_SL(VISIT, u) = FALSE) then //If u is not in VISIT then visit it
      1. INSERT_SL_END(VISIT, u)            //Store u in VISIT
      2. ptr = Gptr[u]
      3. While (ptr.LINK ≠ NULL) do         //To enter all the adjacent vertex of v into
                                            //OPENQ
         1. vptr = ptr.LINK
         2. OPENQ.ENQUEUE(vptr.LABEL)
      4. EndWhile
   3. EndIf
6. EndWhile
7. Return (VISIT)
8. Stop
```

In the algorithm DFS_LL, we have used stack and all operations related to stack can be referred form Chapter 4. Similarly, queue is used in the algorithm BFS_LL, and its associated operations can be referred from Chapter 5. Usual operations on single linked list are discussed in Chapter 3.

Assignment 8.7

1. For the graph as shown in Figure 8.19 trace the Algorithm BFS_LL to obtain the BFS traversal starting from vertex labelled as 1.
2. Repeat Problem 1 but starting with the vertex labelled 9.
3. Modify the Algorithms DFS_LL and BFS_LL so that they can return the acyclic graph depicting the path of traversals.

2. EndFor
7. EndFor
8. $N = N - 1$ //The vertex V_x is deleted
9. Stop

The deletion of an edge from a graph is also straightforward. To delete an edge $<V_i, V_j>$ from the undirected graph, we must reset the entries at (i, j) and (j, i) both, in the adjacency matrix by 0's. In case of directed graph, if there is the edge from V_i to V_j then we have to reset the entry at (i, j) of the matrix only. The algorithm DELETE_EDGE_AM is described as below to delete an edge from any graph.

Algorithm DELETE_EDGE_AM(V_i, V_j)

Input: $<V_i, V_j>$, the edge to be deleted between the vertices V_i, V_j.
Output: The graph without edge between V_i and V_j.
Data structure: Matrix representation of graph. Gptr, the pointer to graph.

Steps:

1. Let n = number of vertices in the graph.
2. If ($V_i > n$) or ($V_j > n$) then
    1. Print "Vertex does not exist: Error in edge removal"
3. Else
    1. Case: Undirected graph
        1. Gptr[V_i][V_j] = 0
        2. Gptr[V_j][V_i] = 0
    2. Case: Directed graph
        1. Gptr[V_i][V_j] = 0
4. EndIf
5. Stop

Traversals

In Section 8.4.1, we have discussed two graph traversal methods: DFS and BFS, and we have learnt how these two methods can be implemented on graphs which are represented using linked lists. In this section, we are to describe the realization of same methods when graphs are stored in adjacency matrices.

Let us first consider the DFS traversal on a graph which is represented with an adjacency matrix. Following is the algorithm DFS_AM for the purpose.

Algorithm DFS_AM(V)

Input: V = the starting vertex. Let N be the number of vertices in the graph
Output: An array VISIT giving the order of visit of vertices during traversal.
Data structure: A stack OPEN to hold the vertices which is initially empty. Matrix representation of graph with Gptr is the pointer to it.

Steps:

1. If Gptr = NULL then
    1. Print "Graph is empty"
    2. Exit

2. EndIf
3. $u = V$
4. OPEN.PUSH(u) //Push the starting vertex into OPEN
5. While (OPEN.TOP ≠ NULL) do //Till the stack is not empty
   1. u = OPEN.POP( ) //POP the top element from OPEN
   2. If (SEQ_SEARCH(VISIT, u) = FALSE) then //If v is not in the array VISIT
      1. INSERT(VISIT, u) //Store v in VISIT
      2. For i = 1 to N do //Push all the adjacent vertex of v into OPEN
         1. If (Gptr[v][i] = 1) then
            1. OPEN.PUSH(i)
         2. EndIf
      3. EndFor
   3. EndIf
6. EndWhile
7. Return(VISIT)
8. Stop

Now let us consider the BFS traversal on a graph represented with adjacency matrix. The algorithm as stated below is an attempt to implement BFS traversal.

Algorithm BFS_AM(V)

Input: V, the starting vertex. Let N be the number of vertices in the graph.

Output: An array to store the order of visit of vertices during traversal.

Data structure: A queue OPENQ to hold the vertices which is initially empty. Matrix representation of graph with Gptr is the pointer to it.

Steps:

1. If (Gptr = NULL) then
   1. Print "Graph is empty"
   2. Exit
2. EndIf
3. $u = V$
4. OPENQ.ENQUEUE(u) //Enter the starting vertex into the queue OPENQ
5. While (OPENQ.STATUS( ) ≠ EMPTY) do //Till the OPENQ is not empty
   1. v = OPENQ.DEQUEUE( ) //Delete the item from OPENQ
   2. If (SEQ_SEARCH(VISIT, u) = FALSE) then //If v is not in the array VISIT
      1. INSERT(VISIT, u) //Store visited node u in VISIT
      2. For i = 1 to N do //To enter all the adjacency vertices of v //into OPENQ
         1. If (Gptr[u][i] = 1) then
            1. OPENQ.ENQUEUE(i).
         2. EndIf
      3. EndFor
   3. EndIf
6. EndWhile
7. Return(VISIT)
8. Stop