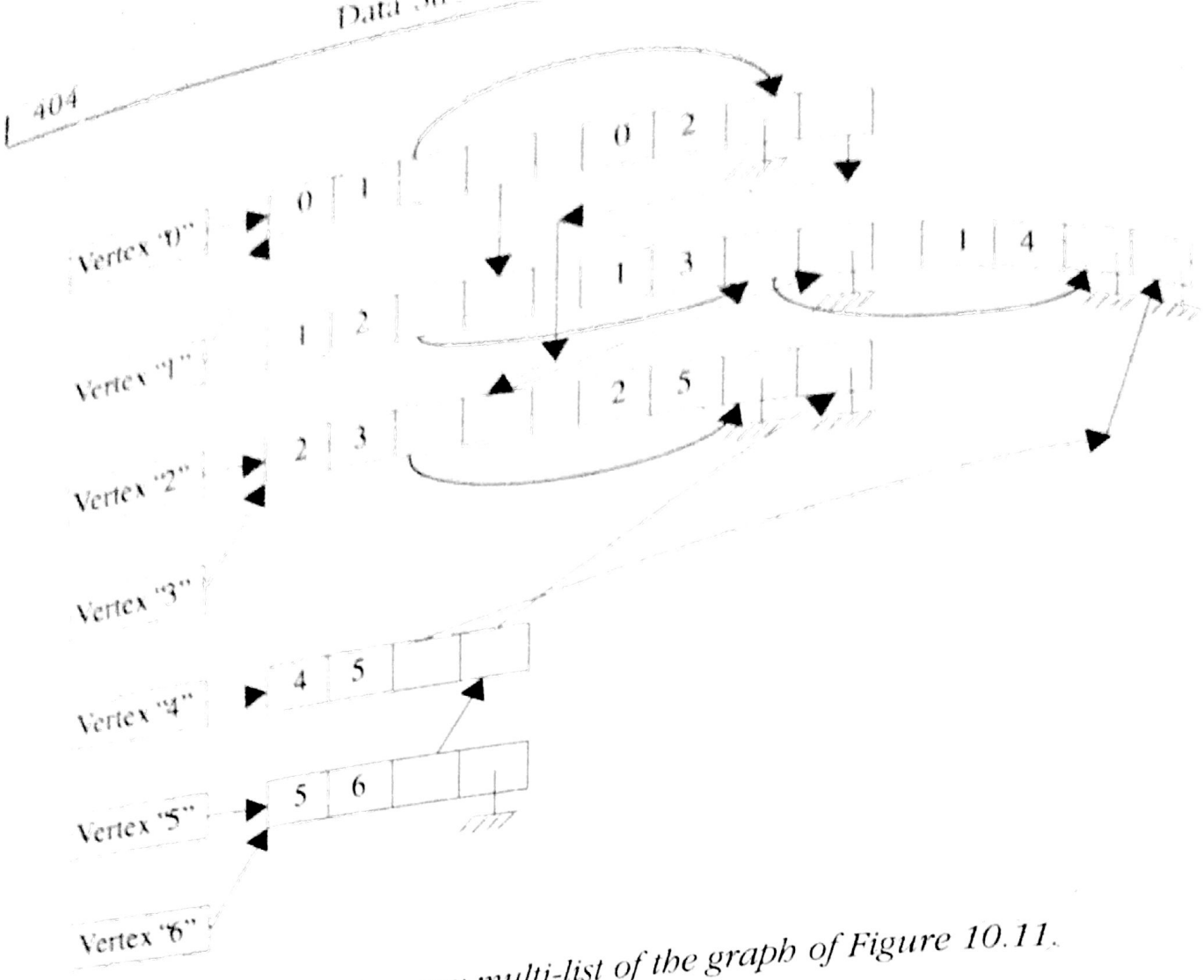


Figure 10.17 Adjacency multi-list of the graph of Figure 10.11.

Since adjacency lists are maintained in a multi-list, the representation is called adjacency multi-lists. The adjacency multi-list representation of the graph of Figure 10.11 is shown in Figure 10.17. See that the edges incident on a vertex "i" may be obtained by traversing the appropriate links of the edges pointed to by vertex "i".

10.4 Traversal of Graphs

A number of graph problems involves traversal of a graph. Traversal of a graph means visiting each of its vertices exactly once. This is accomplished by visiting the vertices in a systematic way. Two commonly used techniques of graph traversal are known as Depth-first search (DFS) and breadth-first search (BFS). These algorithms are discussed for undirected graphs unless otherwise mentioned. However, these algorithms do work for directed graphs also.

10.4.1 Depth First Search

This is one popular technique for systematically traversing the vertices of a graph. The method starts traversing the graph from a given vertex "v_0". That is "v_0" is the first vertex to be visited. The next vertex to visit is an unvisited vertex adjacent to "v_0". If "v_0" has a number of unvisited adjacent vertices then any one of them may

be selected for visiting next. Once a vertex is visited, it is marked as "visited" to facilitate the task of finding unvisited adjacent vertices. Thus, from any vertex "v_i", a new adjacent unvisited vertex "v_j" is explored. Next, another new vertex adjacent to "v_j" is explored without traversing along other edges incident to "v_i". When a vertex "v_p" is reached from "v_q" and "v_p" does not have any unvisited adjacent vertex then "v_q" is re-examined to check whether it has any more unvisited adjacent vertex "v_m". Then, depth first search is initiated from the vertex "v_m". The process continues until all adjacent vertices of the starting vertex have been visited. This leads to a recursive description of the traversal technique as explained in the following.

Depth first search (DFS) from a vertex "i", would mean marking the vertex "i" as visited and initiating DFS from a vertex "j" where "j" is an unvisited adjacent vertex of the vertex "i". DFS from the vertex "i" would be complete when there is no unvisited adjacent vertex of the vertex "i". In order to write an algorithm, the information about already visited vertices must be remembered. For this purpose, an array "visited" of zeroes and ones may be kept where visited[i] = 0 means that the vertex "i" has not been visited and visited[i] = 1 means that the vertex "i" has already been visited. The array "visited" and a variable "g" of type "pointer to graph" are assumed to be global to the function "DFSearch". Moreover, the array "visited" must be initialized so that visited[i] is set to "0" for i = 0 to (g->noOfVertices-1) before calling "DFSearch". For simplicity, "g" is assumed to be a connected graph.

```
void DFSearch (int start)
{
   int v;
   adjVert *adj;
   visited[start] = 1;
   printf("%d ", start);
   adj = g->adjList[start];
   while(adj != NULL)
   {
         v = adj->vertex;
         if (!visited[v])
               DFSearch(v);
         adj = adj->next;
   }
}
```

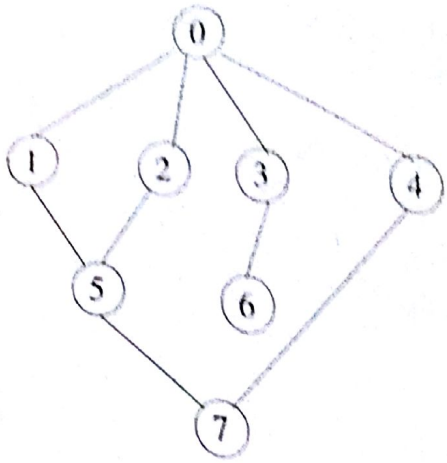


Figure 10.18(a) A sample graph

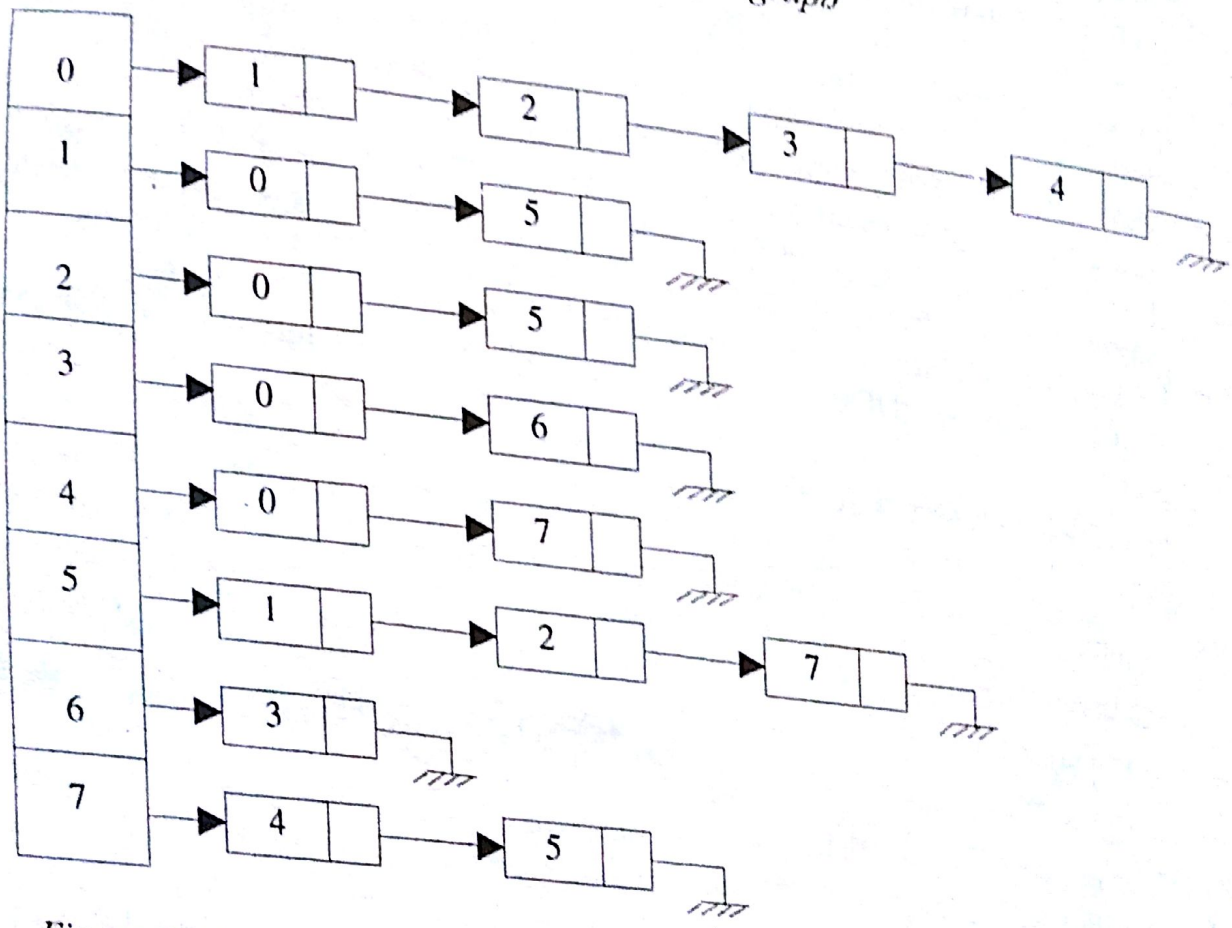


Figure 10.18(b) Adjacency list representation of graph of Figure 10.18(a)

The recursion tree obtained by applying the function "DFSearch" on the graph shown in Figure 10.18(a) is shown in Figure 10.19. The adjacency list representation as shown in Figure 10.18(b) of the graph of Figure 10.18(a) is used to find out adjacent vertices from a given vertex. Traversal starts from the vertex numbered "0". Each node in the recursion tree is labelled with the vertex number on which the function is recursively called. Along with every node in the recursion tree, the content of the array "visited" is also mentioned in Figure 10.19. The content of the array is mentioned at the time when a call to the function is made.

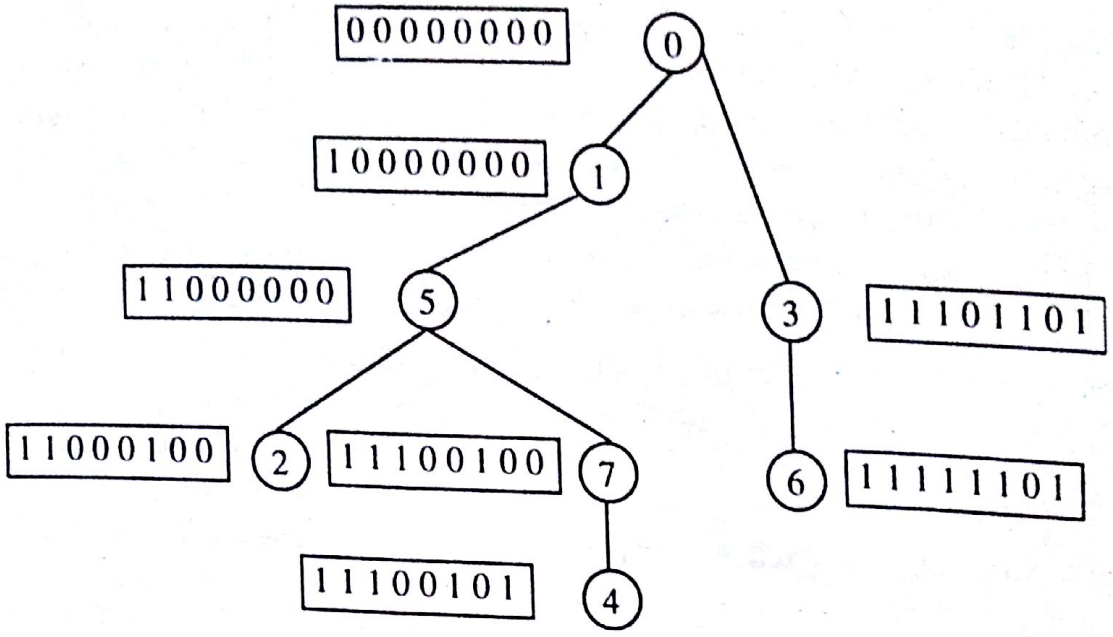


Figure 10.19 Recursion tree obtained by applying DFSearch(0) on the graph of Figure 10.18(a).

The following output will be produced as a result of the call "DFSearch(0)"

0 1 5 2 7 4 3 6

It must be mentioned that the adjacency list or adjacent matrix representation of a graph is not unique. A different numbering scheme of the vertices may yield different adjacency matrices for the same graph. Similarly, the order in which the adjacent vertices of a given vertex are stored in the list is not fixed and consequently there could be different adjacency lists representing the same graph. Due to non-uniqueness of adjacency list or adjacency matrix representation of a graph, the order in which the vertices would be traversed in a DFS is also not unique. Also note that the vertices which are traversed by the DFS are connected by edges. Thus, depth first traversal produces a spanning tree of a given connected graph. For example, the recursion tree shown in Figure 10.19 also represents a spanning tree produced by depth first traversal of the graph of Figure 10.18(a). Such spanning trees are often referred to as depth first search spanning trees.

If the graph is not connected then the resultant depth first spanning tree does not contain all vertices of the graph. In fact the tree produced by depth first search is a spanning tree for one component only. In other words, all vertices connected to the "start" vertex would be traversed by the function "DFSearch". Thus, the component of the graph of which a spanning tree will be generated depends on the "start" vertex (on which "DFSearch" is applied). To generate the forest of all spanning trees for all components, depth first search is to be called repeatedly on all vertices which are yet unvisited. The algorithm to print all connected components is discussed in section 10.4.3.

Observe that the function "DFSearch(0)" traverses adjacency list for vertex "i" only once. Thus, the total complexity of that function is O(2e) because there are (2e) number of nodes in the adjacency list of a graph. Therefore, the complexity of the function "DFSearch" is O(e) if the graph having "e" edges is represented by adjacency list. If the graph having "n" vertices is represented by adjacency matrix then finding all adjacent vertices of a vertex implies scanning a row of "n" elements. So, in that case, depth first traversal would require $O(n^2)$ time.

Removal of recursion from the algorithm "DFSearch" implies that the vertices not visited so far must be maintained in a list in such a way that the vertices encountered last are explored first. A natural option to store the unexplored vertices is to use a stack. The stack should contain only the start vertex at the beginning. At every iteration, the stack is popped to obtain the current vertex. Traversal restarts from the current vertex. All vertices adjacent to the current vertex which are not visited and which are not already stored in the stack are pushed into the stack. The information regarding whether a vertex is already visited or is already existent inside the stack is maintained in the array "visited". Since the vertices are identified by integers only, a stack of integers would serve the purpose.

```
void DFSNonRecursive(int start)
{
  int v;
  stackOfIntegers s;
  adjVert *adj;

  initialize (&s);
  push (&s, start);
  while (!isEmpty(s))
  {
        v = pop (&s);
        visited[v] = 1;
        printf ("%d  ", v);
        adj = g->adjList[v];
        while (adj != NULL)
        {
              if (!visited[adj->vertex])
              {
                    visited[adj->vertex] = 1;
```

```
                push (&s, adj->vertex);
            }
            adj = adj->next;
        }
    }
}
```

As it was the case with "DFSearch", all elements of the array "visited" must be set to zero before calling "DFSNonRecursive". The functions "initialize", "push", "pop" and the type definition of "stackOfIntegers" have been described in Chapter 6.

One can easily note that the order in which the function traverses the vertices of the graph of Figure 10.18(a) (represented by the adjacency list of Figure 10.18(b)) is not the same as the order in which the vertices are visited by the recursive algorithm. First, trace the execution of the non-recursive algorithm as explained in the Figure 10.20. Note that the adjacent vertices are pushed from left to right and thus will be popped in the reverse manner. The content of the stack is shown from left to right in an open-ended rectangle where the open end corresponds to the top of the stack.

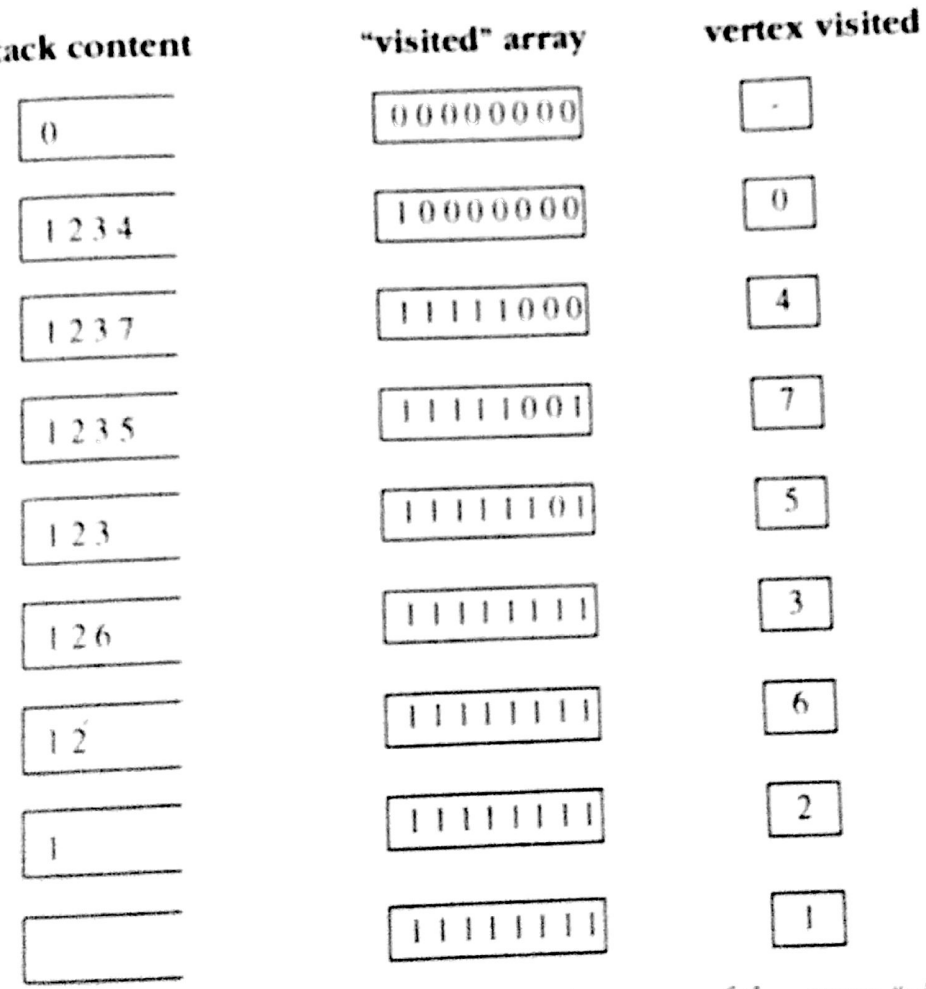


Figure 10.20 The content of the stack and the status of the array "visited" at the beginning of each iteration in the while loop. The vertex visited in each iteration of the while loop is also shown.

The algorithm for depth first traversal also works for directed graphs. However, depth first traversal may produce a forest even when applied to a connected directed graph.

10.4.2 Breadth First Search

This is another popular way to visit the vertices of a graph systematically. This method starts from a given vertex "v_0". The vertex "v_0" is marked visited. All adjacent vertices of "v_0" are visited next. Then, one of the adjacent vertices of "v_0" is taken up and its unvisited adjacent vertices are visited next. This process continues until all vertices reachable from "v_0" are visited. A breadth first search (BFS) initiated from "v_i" visits all vertices in "v_p" where "v_p" denotes the set of all unvisited adjacent vertices of "v_i". Next, the process continues from any vertex in "v_p". The method continues until all the vertices adjacent to "v_i" are fully explored. The algorithm for breadth first traversal has to maintain a list of vertices which have already been visited but whose adjacent vertices have not been explored still. The vertices whose neighbours are yet to be visited can be stored in a queue. Initially the queue contains the start vertex. In every iteration, a vertex is removed from the queue and all of its adjacent vertices, which are not encountered yet, are marked visited and then added to the queue. The algorithm terminates when the queue becomes empty.

The algorithm for BFS is described in the following. As usual, it assumes a global array "visited" to keep track of vertices already explored. Also the graph is assumed to be represented by an adjacency list and the graph is globally available as a variable "g". The algorithm depends on the availability of an implementation of a circular queue of integers. So, a type "queue" and three functions "queueInit", "addToCircq" and "deleteCircq" are to be properly defined before the function for breadth first search is defined. They can be easily defined by replacing "dataType" by "int" in the definition of "queue" and in the functions "addToCircq" and "deleteCircq" as described in Chapter 4. A function "queueInit" is to be written to set "rear" and "front" indices of the queue to zero. The function "deleteCircq" returns (-1) when the queue is empty.

```
void BFSearch (int start)
{
   int v, res;
   queue q;
   adjVert *adj;

   queueInit (&q);
   visited[start] = 1;
addToCircq(start, &q);
```

```
while ((res = deleteCircq (&v, &q)) != -1)
{
    printf("%d  ",v);
    adj = g->adjList[v];
    while (adj != NULL)
    {
        if (!visited[adj->vertex])
        {
            visited[adj->vertex] = 1;
            addToCircq(adj->vertex, &q);
        }
        adj = adj->next;
    }
}
}
```

The working of the algorithm is illustrated in Figure 10.21 by applying the algorithm on the graph of Figure 10.18(a). The graph is represented by the adjacency list of Figure 10.18(b). The vertex numbered "0" is assumed as the start vertex. The status of the queue and the array "visited" in the beginning of each iteration of the while loop are clearly mentioned in the figure. In addition, the vertex number, which is printed in each iteration of the loop is also mentioned. The queue is shown as a linear one where the "front" is represented by the left end and the "rear" is represented by the right end.

Each unvisited vertex is added to the queue. Thus, every vertex is added to the queue exactly once in the inner while loop. Also, a vertex is removed from the queue at the beginning of the outer while loop. If the graph of "n" vertices is represented by adjacency list then the outer while loop has a complexity of $(d_0 + d_1 + \ldots + d_{n-1})$ where "d_i" is the degree of a vertex "i". Thus, the complexity of the

Queue Content	Visited array	Vertex visited
0	10000000	-
1 2 3 4	11111000	0
2 3 4 5	11111100	1
3 4 5	11111100	2
4 5 6	11111110	3
5 6 7	11111111	4
6 7	11111111	5
7	11111111	6
	11111111	7

Figure 10.21 Illustration of breadth first search on the sample graph of Figure 10.18(a).

function "BFSearch" is O(e) as

$\Sigma_{i=0, n-1} d_i = 2e$.

If the graph is represented by an adjacency matrix then the complexity of breadth first traversal is $O(n^2)$ for a graph of "n" vertices.

Like depth first traversal, breadth first traversal generates a spanning tree of a connected graph. If the graph is not connected, breadth first traversal produces a forest.

10.4.3 Connected components

As mentioned in the context of depth first traversal of a graph, the traversal algorithms can be easily adapted to find out the components of a graph. This can be easily achieved by calling "DFSearch" or "BFSearch" for all vertices yet unvisited as shown in the function "printConnectComponents".

```
void printConnectedComponents()
{
```

```
    int i, count = 0;
    for (i = 0; i < g->noOfVertices; i++)
            visited[i] = 0;
    for (i = 0; i < g->noOfVertices; i++)
    {
            if (!visited[i])
            {
                    printf ("The vertices in the %d-th component\n", count);
                    count++;
                    DFSearch(i);
                    printf ("\n");
            }
    }
}
```

The array "visited" and the graph "g" are assumed to be globally defined and appropriately initialized. In the function "printConnectedComponents", one can also repeatedly call "BFSearch" in place of "DFSearch".

10.5 Shortest Path Problems

One of the fundamental problems encountered in many graph-theoretic applications is to find the shortest path (i.e. the path having the least cost) between two vertices in a weighted graph. Length of a path in a weighted graph is defined to be the sum of costs or weights of all edges in that path. In general, there could be more than one path between a pair of specified vertices, say "v_i" and "v_j", and a path with the minimum cost or weight is called a shortest path from "v_i" to "v_j". Note that the shortest path between two vertices may not be unique.

Consider the weighted undirected graph of Figure 10.6. It can be easily seen that there are more than three paths 0-1, 0-2-1, 0-2-3-1, from the vertex "0" to the vertex "1". The path with the shortest length is 0-2-1. The length of the shortest path is "4". It is apparent that the problem of finding the shortest path from "v_i" to "v_j" in a graph "G" is not well-defined if "G" contains a cycle whose length is negative. Because, in that case, each traversal of the cycle reduces the length of the traversed path by a constant amount. Hence, the length of the shortest path can be made arbitrarily low. The presence of negative edges does not ill-define the problem as long as the graph does not contain a cycle of negative length. However, negative edges are rare in practical applications. For simplicity, edges having positive weights have been considered. But, there are algorithms which successfully find the short-

est path in a graph containing negative edges as well.

There are many different variations of the shortest path problem. They vary with respect to the specification of the start vertex (referred to as source) and the end vertex (referred to as destination). Some of the commonly known variants are listed in the following.

- The shortest path from a specified source vertex to a specified destination vertex.
- The shortest paths from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
- The shortest paths between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.

In this section, the single-source shortest path problem is considered. Assume that the graph G = (V, E) contains "n" vertices 0, 1, 2, ..., (n-1). Let "c_{ij}" denote the cost of the edge "e_{ij}" which connects vertex "i" to vertex "j". Assume $c_{ij} > 0$ for all i, j and $i \neq j$. "c_{ij}" is assumed to be infinity if "e_{ij}" does not exist. Without any loss of generality, assume that the problem is to find the shortest paths from the vertex "0" to all other vertices. Usually, shortest path problems are defined on weighted directed graphs. However, undirected graphs may be regarded as a directed graph where each edge in the undirected graph (u, v, w) has to be interpreted as two edges (u, v, w) and (v, u, w).

The following observations are helpful is ascertaining the correctness of the following algorithms for computing the shortest path of all vertices from a given vertex. Readers are encouraged to prove these properties.

- The length of the shortest path is always longer than the length of any of its sub-paths.
- Any sub-path from a vertex "m" to another vertex "n" of a shortest path from a vertex "i" to the vertex "j" must itself be a shortest path from the vertex "m" to the vertex "n".
- For any pair of vertices "i" and "j", the shortest path from the vertex "i" to the vertex "j" may contain at most (n-1) edges in a graph having "n" vertices.

.1 Dantzig's Algorithm

et V={0, 1, ..., n-1} be the "n" vertices of a graph. The problem is to find out the hortest paths from a particular vertex, say "0", to all other vertices of the graph that an be reached from this vertex. The set "V"

10.5.2 Dijkstra's algorithm

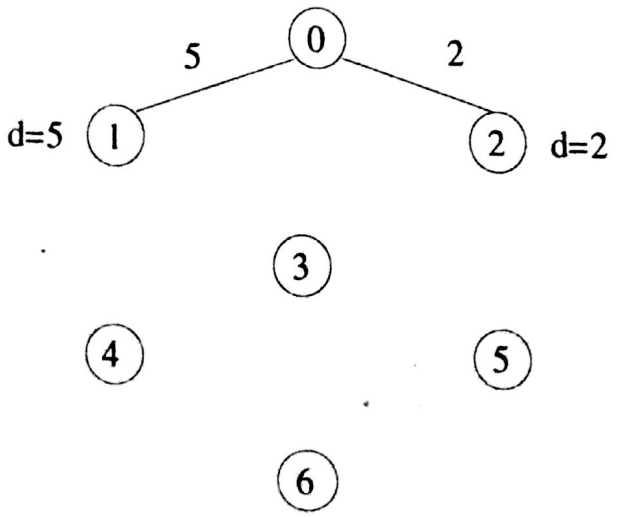
Dijkstra's algorithm also works by maintaining a set "S" of vertices whose shortest paths and shortest distances from the vertex "0" is already known. At each step of the algorithm, one more vertex which does not belong to "S" is included into the set "S". To facilitate such inclusion, an array called "distance" is used to keep track of the current shortest distance of any vertex from the vertex "0". distance[i] is initialized to "c_{0i}" for all "i", $0 \le i \le (n-1)$. "S" is initialized to contain only one vertex "0". Note that, by definition of "S", distance[i] is the shortest distance between the vertices "0" and "i" provided $i \in S$. If $i \notin S$ then distance[i] is calculated as the minimum of c_{ji} + distance[j] over all "j" such that $j \in S$. Once the recalculation of all entries in the array "distance" is done, the vertex "v" is chosen such that distance[v] is minimum among all distance[i], $i \notin S$. At this point, it must be proved that distance[v] actually gives the shortest distance between vertex "0" and vertex "v". This can be proved as follows. Assume that distance[v] is not the shortest distance between vertex "0" and vertex "v" as the shortest path from vertex "0" to "v" consists of a vertex "x" which is not currently included in "S". Then, the shortest path from "0" to "v" must have one vertex "y" which is adjacent to a vertex in the set "S" and $y \notin S$. In that case, distance[y] will be computed as the shortest distance from "0" to "y". But the vertex "y" lies on a shortest path from "0" to "v". Thus, distance[y] < distance[v]. Since "v" was selected for inclusion into "S", distance[v] < distance[y]. These two facts are contradictory. Thus, distance[v] must be the shortest distance between the vertices "0" and "v".

The function "shortestPath" computes the shortest paths of all vertices from the vertex "0" of a graph "g". The graph "g" is assumed to be global. The function fills up the arrays "d" and "t". The shortest path spanning tree is implicitly constructed in the array "t". If t[i] is "j" then, the parent of the vertex "i" is the vertex "j". Whenever a vertex "v" is included to "S" due to the fact that "v" is adjacent to some vertex "w" of "S" and distance[v] is minimum among all vertices currently not belonging to "S" then t[v] is set to "w"

```
void shortestPath (int d[], int t[])
{
   int i, n, min, v, w;
   int shortestDistFound [100];
   n = g->noOfVertices;
   d[0] = 0;
   for (i = 0; i < n; i++)
   {
         shortestDistFound[i] = 0;
         d[i] = g->adjMat[0][i];
```

```
    }
    for (i = 0; i < n; i++)
    {
            min = MAX_INT;
            for (w = 0; w < n; w++)
            {
                    if ((!shortestDistFound[w]) && (d[w] < min))
                    {
                            v = w;
                            min = d[w];
                    }
            }
            shortestDistFound[v] = 1;
            for (w = 0; w < n; w++)
                    if (!shortestDistFound[w])
                            if ((min + g->adjMat[v][w]) < d[w])
                            {
                                    d[w] = min + g->adjMat[v][w];
                                    t[w] = v;
                            }
    }
}
```

The working of this algorithm on the graph of Figure 10.6 to compute the shortest paths from the vertex "0" is illustrated in Figures 10.24(a)-(f). Each figure explains an iteration of the outer "for" loop. The status of "S" before entering the loop is mentioned. The status of "d" and "t" at the end of the iteration is mentioned.

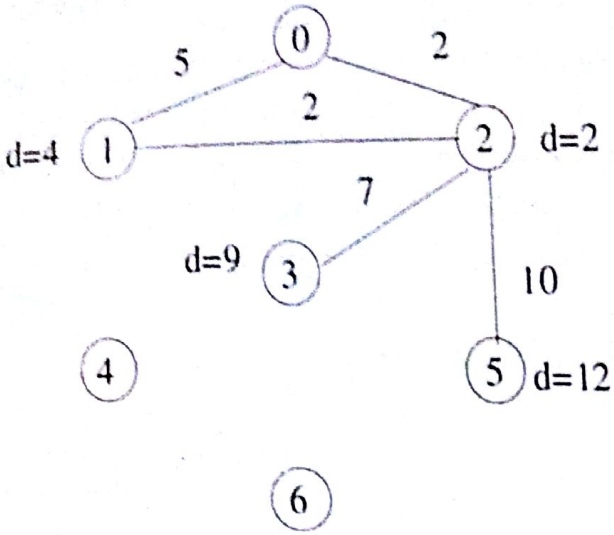


S = {0}

d	0	5	2	∞	∞	∞	∞

t	-	0	0	-	-	-	-

Figure 10.24(a) d[1] and d[2] are calculated. As d[2] is minimum, "2" is added to "S" in the next iteration. t[2] and t[1] are set to "0"

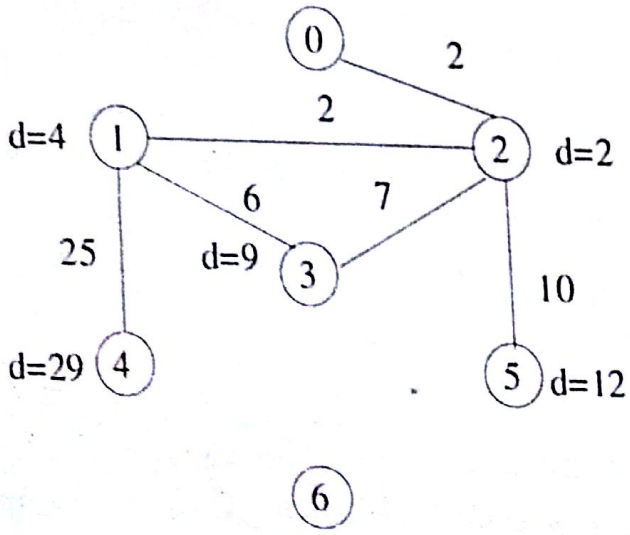


S = {0, 2}

d	0	4	2	9	∞	12	∞

t	-	0	0	2	-	2	-

10.24(b) d[1], d[3], d[5] are recalculated as the vertices "1", "3", "5" do not belong to "S" and they are adjacent to "2", the last vertex added to "S". As d[1] is minimum among all d[i] such that "i" does not belong to "S", "1" is added to "S" and t[1], t[3], t[5] are set to "2".

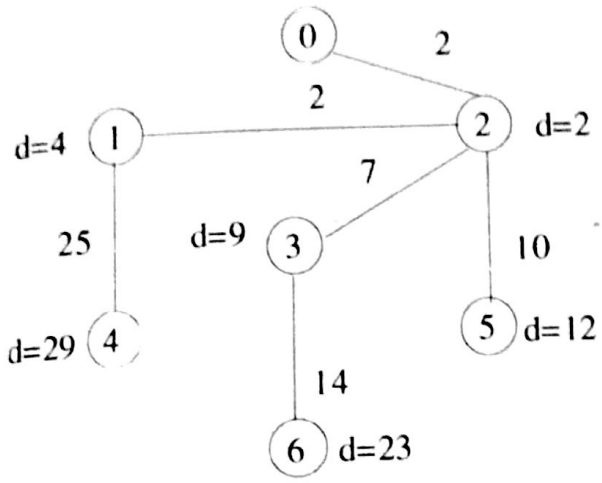


S = {0, 2, 1}

d	0	4	2	9	29	12	∞

t	-	2	0	2	1	2	-

10.24(c) d[3], d[4] are reexamined as "3" and "4" do not belong to "S" and they are adjacent to "1". d[4] is updated. As d[3] is minimum among all d[i] such that "i" does not belong to "S", "3" is added to "S" and t[4] is set to "1".

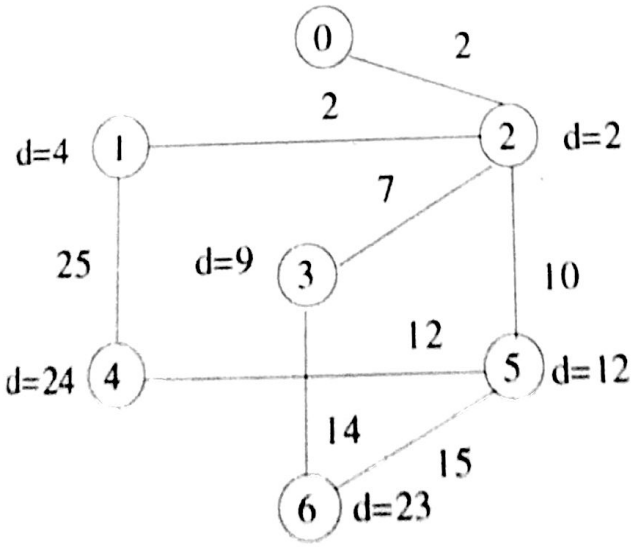


S = {0, 2, 1, 3}

d	0	4	2	9	29	12	23

t	-	2	0	2	1	2	3

10.24(d) d[6] is reexamined as "6" does not belong to "S" and it is adjacent to "3", the last vertex added to "S". d[6] is updated. As, d[5] is minimum among all d[i] such that "i" does not belong to "S", "5" is added to "S" and t[6] is set to "3".

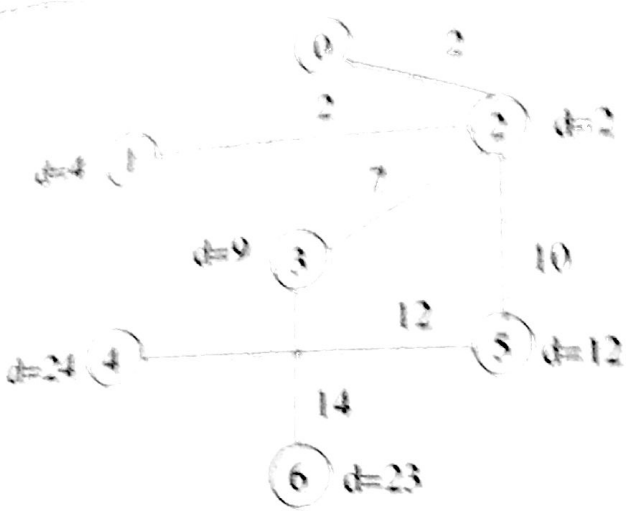


S = {0, 2, 1, 3, 5}

d	0	4	2	9	24	12	23

T	-	2	0	2	5	2	3

10.24(e) d[6], d[4] are reexamined as vertices "6" and "4" do not belong to "S" and they are adjacent to "5", the last vertex added to "S". Only d[4] is updated. As, d[6] is minimum among all d[i] such that "i" does not belong to "S", "6" is added to "S" and t[4] is set to "5".



S = {0, 2, 1, 3, 5, 6}

d	0	4	2	9	24	12	23

t	-	2	0	2	5	2	3

[0.24(f)] d[4] is reexamined as "4" does not belong to "S" and "4" is adjacent to "6", the last vertex added to "S". But, d[4] is not updated. d[4] is the minimum.

The function "shortestPath" contains two nested "for" loops each of which has a complexity of O(n). So, the total complexity of the algorithm is $O(n^2)$ if the graph has "n" vertices.

10.5.3 All pair shortest paths and transitive closure for directed graphs

Consider a more general problem of finding the shortest paths (as well as their costs) between every pair of vertices "i" and "j" of a graph. Recall that Dijkstra's algorithm computes the shortest paths from a specified vertex to all other vertices in $O(n^2)$ time for a graph having "n" vertices. This algorithm can be repeatedly applied "n" times (once for each vertex as the start-vertex) to solve the all pair shortest path problem. The resulting algorithm will have a time complexity of $O(n^3)$. However, the same problem can be solved in $O(n^3)$ time using a much simpler technique for directed graphs. One advantage of the new approach is that the actual computation involved may be less.

Assume that a directed weighted graph "G" having "n" vertices is represented by its adjacency matrix "a". Further, assume that "G" does not have any self-loop. If the vertex "j" is reached from the vertex "i" without passing through any other intermediate vertex then "a_{ij}" is the shortest distance from the vertex "i" to the vertex "j". Then it is attempted to find out a possible shorter path between the vertices "i" and "j" which might include only vertex "0" then a shorter distance between vertices "i" and "j" may be discovered. More specifically, the shortest distance between vertices "i" and "j" passing possibly through the vertex "0" is the minimum of "a_{ij}" and (a_{i0} + a_{0j}). In this way, one can continue allowing more and more vertices from "0" to "k" to be used as intermediate vertices and discover even shorter paths. When "k" is equal to (n-1) then the shortest distance between the vertices "i" and "j" is found out.

where a_{ij} is the (i, j)-th entry of the adjacency matrix.

For $k > 0$

$$T^k(i, j) = T^{k-1}(i, j) \vee (T^{k-1}(i, j) \wedge T^{k-1}(i, j))$$

The symbols "$\vee$" and "$\wedge$" denote OR and AND operations respectively.

"T", the transitive closure matrix = $T^n(i, j)$ if the graph has "n" vertices. The operations involved in computing T^k are logical while those involved in computing A^k are arithmetic. So, the space required to store T^k's is less than that to store A^k's if bit-wise logical operations are performed and each bit represents an entry in the matrix. Similarly, the actual time required to calculate the matrices T^k's is less than that required to calculate A^k.

10.6 Algorithms for computing minimal spanning tree

The notion of minimal spanning tree of a graph has already been introduced. There are two well-known algorithms to compute minimal cost spanning tree of a weighted undirected graph. One of these algorithms was designed by J. Kruskal in 1957 and the other one is due to Prim. In this section, these algorithms will be explained.

10.6.1 Kruskal's Algorithm

This algorithm works on a list of edges of a graph and the list is sorted in order of non-decreasing cost or weight of the edges. It starts with a forest of "n" trees if the graph has "n" vertices. Each tree in the forest contains one distinct vertex of the graph and no edges. In each step of the algorithm, one edge from the sorted list is considered for connecting two forests. The edge is added only if its inclusion does not form a cycle. Once an edge is considered, the edge is deleted from the list of edges. The algorithm continues until (n-1) edges are added or the list of edges is exhausted. Once the algorithm terminates after adding (n-1) edges, a minimal spanning tree is generated. Consider the graph of Figure 10.6. There are eleven edges. An edge connecting the vertices "i" and "j" may be represented by a tuple (i, j). Then, the list of edges of the graph in Figure 10.6 sorted in non-descending order may be given by {(0, 2), (1, 2), (0, 1), (1, 3), (2, 3), (2, 5), (4, 6), (4, 5), (3, 6), (5, 6), (1, 4)}. The initial forest may be depicted by the Figure 10.27(a).

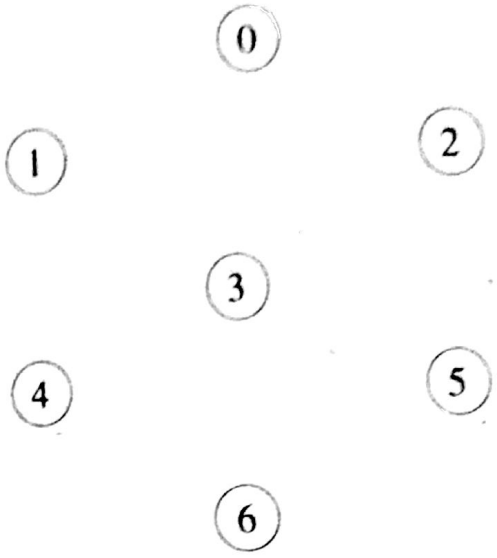


Figure 10.27(a) Initial forest of "n" trees

The first edge (0, 2) from the list of edges is considered. Since the inclusion of this edge does not form a cycle, it can be added to modify the partial minimal spanning tree resulting in the diagram of Figure 10.27(b).

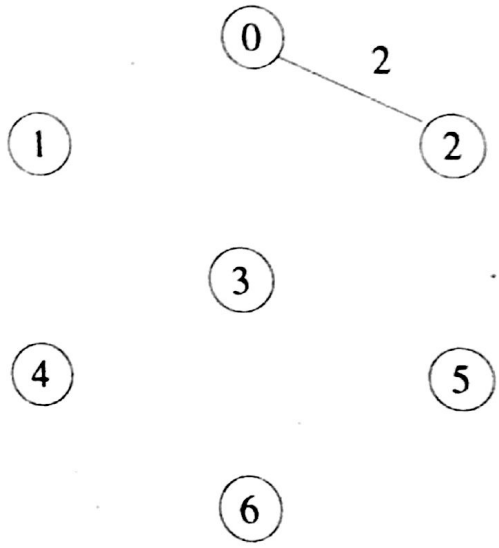


Figure 10.27(b) The edge (0, 2) is added to reduce the number of trees in the forest.

The next edge to be considered is (1, 2). Addition of this edge does not create a cycle too and hence, it will added to the partially formed minimum spanning tree of Figure 10.27(b) resulting a tree as shown in Figure 10.27(c).

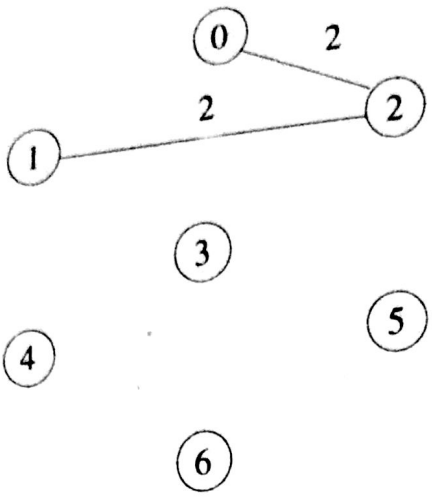


Figure 10.27 (c) The next edge (1, 2) is also added.

If the next edge (0, 1) is considered then a cycle is formed. Therefore, this edge is not added to the partially formed tree of Figure 10.27(c). And the next edge (1, 3) is chosen for consideration. Since inclusion of this edge does not form a cycle, this edge can be safely added and the partial minimal spanning tree of Figure 10.27(d) is created.

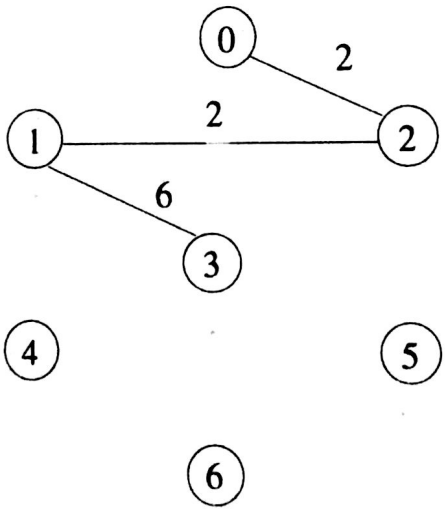


Figure 10.27(d) The edge (1, 3) is added.

If the next edge (2, 3) is added then a cycle 2-1-3-2 is formed. Therefore, this edge is not added to the partially formed tree of Figure 10.27(d). And the next edge (2, 5) is chosen. Since inclusion of this edge does not form a cycle, this edge can be safely added and the partial minimal spanning tree of Figure 10.27(e) is created.

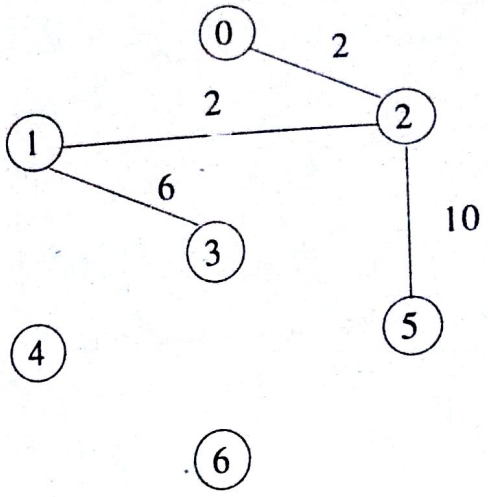


Figure 10.27(e) The edge (2, 5) is added.

Next, the edge (4, 6) is chosen. This edge can be safely added. See that the result of each step of the algorithm is not a tree but a forest. In fact, after a successful addition of an edge the number of forests decreases. See that the number of trees in the forest (shown in Figure 10.27(f)) after addition of the edge (4, 6) is 2.

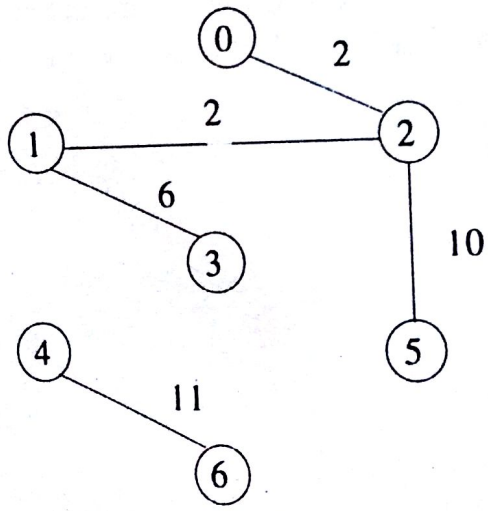


Figure 10.27(f) The edge (4, 5) is added.

The next edge in the list, (4, 5), can be added in the forest merging the two trees creating a minimal cost spanning tree, which is shown in Figure 10.27(g).

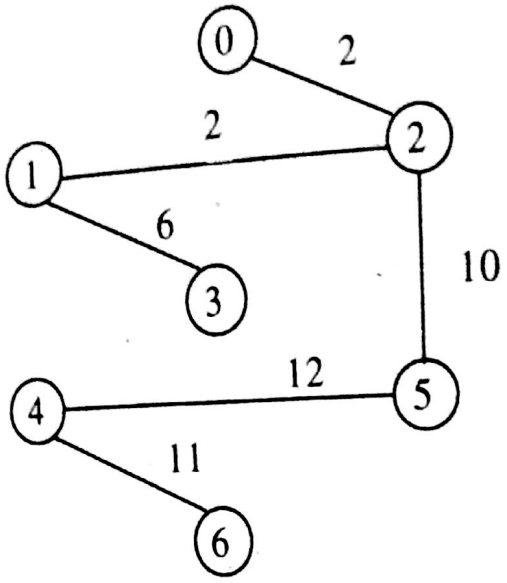


Figure 10.27(g) The edge (4, 6) is added.

Figure 10.27 Illustration of Kruskal's algorithm as applied on the graph of Figure 10.6.

Specifying Kruskal's algorithm in "C" like pseudo-code is now simple. The algorithm in C-like pseudo code is provided in the following. The input parameter of the algorithm is a list of edges sorted in non-decreasing order of their weights. Each node in such a list is a self-referential structure containing two integers representing an edge connecting the vertices identified by the integers. Such a structure is defined in the following.

```
typedef struct edgeList
{
   int u, v;
   float weight;
   struct edgeList * next;
} edgeList;
```

A list of edges can be easily computed from adjacency matrix or adjacency list of a graph. Then the list can be sorted in non-decreasing order of their weights. Once this is done, the pointer to the first node in this list may be used as input to the Kruskal's algorithm. Another input to the algorithm is "n", the number of vertices in the graph. The output of the algorithm is another list of edges which defines a tree. The graph is assumed to be connected for simplicity. If the graph is not connected then there is no spanning tree.

```
edgeList * KruskalMST(edgeList f, int n)
{
   edgeList *T;
   edgeList *e;
   int noOfEdgesAdded = 0;
```

```
    while ( (noOfEdgesAdded < (n-1)) && (f != NULL))
    {
            e = f;
            f = f->next;
            if (e does not form a cycle in T)
            {
                    noOfEdgesAdded++;
                    if (T == NULL)
                    {
                            T = e;
                            r = T;
                            r->next = NULL;
                    }
                    else
                    {
                            r->next = e;
                            r = r->next;
                            r ->next = NULL;
                    }
            }
    }
    return T;
}
```

The following result establishes the fact that Kruskal's algorithm works correctly.

Result

Let a graph be G = (V, E). Suppose that "U" is a proper subset of "V" and "e" be an edge of minimum weight connecting a vertex in "U" with a vertex in (V-U). Then, "e" belongs to a minimal spanning tree "T" of the "G".

Proof:

Suppose that e $\notin$ T where "T" is any minimal spanning tree of "G". Then, add "e" to "T". Consequently, "T" will no longer remain a tree as "e" will introduce a cycle. However, as "T" is a minimal spanning tree, there must be an edge "f" other than "e" from some vertex of "U" to some vertex in (V-U). According to the condition stated, the weight of "f" is greater than or equal to that of "e". Delete "f" from the cycle and the remaining part of the graph i.e. (T+e-f) is once again a tree. The cost

of this tree can not be more than that of "T". Thus, the new tree must be a minimal spanning tree contradicting the assumption that "e" does not belong to any minimal spanning tree. Hence the result is proved.

Kruskal's algorithm always connects some "U" and (V-U) by an edge with the smallest weight without introducing a cycle. Therefore, according to the previous observation, Kruskal's algorithm works properly.

Kruskal's algorithm starts by sorting all edges of a graph. The time complexity of this sorting operation is O(ElogE) if there are "E" number of edges in the graph. The "for" loop in the algorithm makes "E" number of iterations in the worst case. In each iteration, the major task is to find whether the current edge introduces a cycle. As it will be discussed, the complexity of detecting a cycle is O(logn) in the worst case if the graph contains "n" vertices. Thus, the overall time complexity of the algorithm is O(ElogE) + O(Elogn).

The only task remaining in Kruskal's algorithm is to find out whether addition of a new edge introduces a cycle in a forest. This can be done by using the Union-Find algorithms described for sets in Chapter 7. The "n" vertices of a graph may be assumed to be "n" disjoint sets where $S_i = \{i\}$, $0 \le i < n$. Whenever an edge (i, j) is encountered, the set containing "i" and the set containing "j" are found out. If these sets are disjoint then they are united using "Union" operation. Thus, there is a path connecting all vertices in any set. If it is found that vertices "i" and "j" are contained in the same set then addition of the edge (i, j) must introduce a cycle. Thus, finding out whether incorporation of an edge "e" introduces a cycle boils down to do two "find" operations on "e.u" and "e.v". If these "find" operations return the same set then addition of edge "e" would certainly introduce a cycle and hence "e" must be rejected. However, the two sets returned by "find' are disjoint then they are to be joined by a "Union" operation to maintain the fact that now "e.u" and "e.v" belong to the same set. The modified function for Kruskal's algorithm is given in the following.

```
edgeList * KruskalMST(edgeList f, int n)
{
    edgeList *T = NULL;
    edgeList *e, *r;
    int noOfEdgesAdded = 0;
    int i;

    for (i = 0; i < n; i++)
            parent[i] = i;

    for (i = 0; i < n; i++)
            rank[i] = 0;
```

```
    while ( (noOfEdgesAdded < (n-1)) && (f != NULL))
    {
            int p1, p2;
            e = f;
            f = f->next;
            p1 = find(e->u) ;
            p2 = find(e->v);
            if (p1 != p2)
            {
                    unionRank(p1, p2);
                    noOfEdgesAdded++;
                    if (T == NULL)
                    {
                            T = e;
                            r = T;
                            r->next = NULL;
                    }
                    else
                    {
                            r->next = e;
                            r = r->next;
                            r ->next = NULL;
                    }
            }
    }
    return T;
}
```

The arrays "parent" and "rank" are assumed to be global so that the functions "find", "unionRank" and "KruskalMST" can refer to them. The functions "find" and "unionRank" have already been described in Chapter 7. The second version of the function "find" described in Chapter 7 should be used for efficiency. Note that the complexity of the algorithm is dominated by the time to sort the list of edges. To overcome this problem, a heap may be constructed of all edges in linear time. The heap property is to be modified to always put the minimum key at the root of any sub-tree of the heap. The key, in this case, is the weight of an edge. Then, the edge

with minimum weight may be removed and added or rejected for inclusion to th minimal spanning tree. Although this modification does not improve the worst-cas complexity of the algorithm but it improves the average-case time complexity.

10.6.2 Prim's Algorithm

Unlike Kruskal's algorithm, Prim's algorithm does not start with a list of edge sorted in non-decreasing order of their weights. It starts with any arbitrary vertex a the partial minimal spanning tree "T". In each iteration of the algorithm, one edg (u, v) is added to this partial tree "T" so that exactly one end of this edge belong to the set of vertices in "T". Of all such possible edges, the edge having the leas cost is selected. The algorithm continues to add (n-1) edges (if possible). Conside the same graph of Figure 10.6 and apply Prim's algorithm starting with vertex "0" The illustration of the working of Prim's algorithm is shown in Figure 10.28.

The crux in implementing Prim's algorithm is effectively choosing the successive edges. An approach suggested by Kim efficiently solves this problem. For each vertex presently not belonging to partial tree, the algorithm maintains an edge with the minimal cost so that one end of the edge is a vertex belonging to the partial tree.

Specifically, the algorithm maintains three arrays "U", "T" and "Edge". The array "U" maintains the vertices which are already included in the minimal spanning tree. The array "T" maintains the edges which form the minimal spanning tree. The array "Edge" maintains the edge with the minimum cost for each vertex not included yet in the minimal spanning tree so that one end of that edge belongs to the already formed minimal spanning tree. The array "U" initially contains the vertex "0". The array "T" is initially NULL. Type definition of the array "Edge" can be done as follows.

```
edge Edge[MAX_NO_OF_VERTICES];
```

where "edge" is a predefined type containing a tuple, (u, v, weight). "u" denotes the start vertex, "v" denotes the end vertex and "weight" denotes the weight of the edge. The type definition of "edge" is shown in the following.

```
typedef struct edge {
    int u, v;
    float weight;
```