
UNIT 8 GRAPHS

Structure	Page Nos.
8.0 Introduction	20
8.1 Objectives	20
8.2 Definitions	20
8.3 Shortest Path Algorithms	23
8.3.1 Dijkstra's Algorithm	
8.3.2 Graphs with Negative Edge costs	
8.3.3 Acyclic Graphs	
8.3.4 All Pairs Shortest Paths Algorithm	
8.4 Minimum cost Spanning Trees	30
8.4.1 Kruskal's Algorithm	
8.4.2 Prim's Algorithm	
8.4.3 Applications	
8.5 Breadth First Search	34
8.6 Depth First Search	34
8.7 Finding Strongly Connected Components	36
8.8 Summary	38
8.9 Solutions/Answers	39
8.10 Further Readings	39

8.0 INTRODUCTION

In this unit, we will discuss a data structure called Graph. In fact, graph is a general tree with no parent-child relationship. Graphs have many applications in computer science and other fields of science. In general, graphs represent a relatively less restrictive relationship between the data items. We shall discuss about both undirected graphs and directed graphs. The unit also includes information on different algorithms which are based on graphs.

8.1 OBJECTIVES

After going through this unit, you should be able to

- know about graphs and related terminologies;
- know about directed and undirected graphs along with their representations;
- know different shortest path algorithms;
- construct minimum cost spanning trees;
- apply depth first search and breadth first search algorithms, and
- finding strongly connected components of a graph.

8.2 DEFINITIONS

A graph G may be defined as a finite set V of vertices and a set E of edges (pair of connected vertices). The notation used is as follows:

Graph $G = (V, E)$

Consider the graph of *Figure 8.1*.

The set of vertices for the graph is $V = \{1, 2, 3, 4, 5\}$.

The set of edges for the graph is $E = \{(1,2), (1,5), (1,3), (5,4), (4,3), (2,3)\}$.

The elements of E are always a pair of elements.

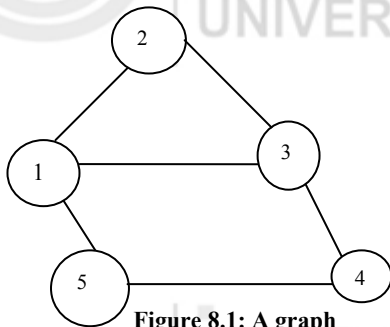


Figure 8.1: A graph

It may be noted that unlike nodes of a tree, graph has a very limited relationship between the nodes (vertices). There is no direct relationship between the vertices 1 and 4 although they are connected through 3.

Directed graph and Undirected graph: If every edge (a,b) in a graph is marked by a direction from a to b , then we call it a Directed graph (digraph). On the other hand, if directions are not marked on the edges, then the graph is called an Undirected graph.

In a Directed graph, the edges $(1,5)$ and $(5,1)$ represent two different edges whereas in an Undirected graph, $(1,5)$ and $(5,1)$ represent the same edge. Graphs are used in various types of modeling. For example, graphs can be used to represent connecting roads between cities.

Graph terminologies :

Adjacent vertices: Two vertices a and b are said to be adjacent if there is an edge connecting a and b . For example, in Figure 8.1, vertices 5 and 4 are adjacent.

Path: A path is defined as a sequence of distinct vertices, in which each vertex is adjacent to the next. For example, the path from 1 to 4 can be defined as a sequence of adjacent vertices $(1,5), (5,4)$.

A path, p , of length, k , through a graph is a sequence of connected vertices:

$$p = \langle v_0, v_1, \dots, v_k \rangle$$

Cycle : A graph contains cycles if there is a path of non-zero length through the graph, $p = \langle v_0, v_1, \dots, v_k \rangle$ such that $v_0 = v_k$.

Edge weight : It is the cost associated with edge.

Loop: It is an edge of the form (v,v) .

Path length : It is the number of edges on the path.

Simple path : It is the set of all distinct vertices on a path (except possibly first and last).

Spanning Trees: A spanning tree of a graph, G , is a set of $|V|-1$ edges that connect all vertices of the graph.

There are different representations of a graph. They are:

- Adjacency list representation
- Adjacency matrix representation

Adjacency list representation

An Adjacency list representation of a Graph $G = \{V, E\}$ consists of an array of adjacency lists denoted by *adj of V* list. For each vertex $u \in V$, $adj[u]$ consists of all vertices adjacent to u in the graph G .

Consider the graph of *Figure 8.2*.

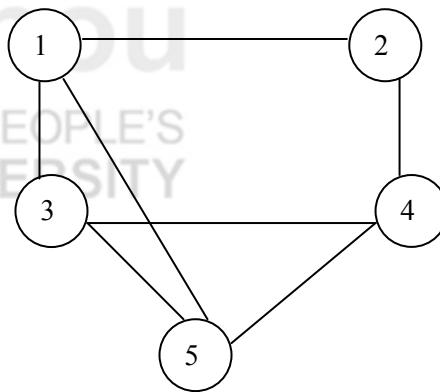


Figure 8.2: A Graph

The following is the adjacency list representation of graph of *Figure 8.2*:

$adj[1] = \{2, 3, 5\}$
 $adj[2] = \{1, 4\}$
 $adj[3] = \{1, 4, 5\}$
 $adj[4] = \{2, 3, 5\}$
 $adj[5] = \{1, 3, 4\}$

An adjacency matrix representation of a Graph $G=(V, E)$ is a matrix $A(a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \text{ belongs to } E \\ 0 & \text{otherwise} \end{cases}$$

The adjacency matrix for the graph of *Figure 8.2* is given below:

	1	2	3	4	5
1	0	1	1	0	1
2	1	0	0	1	1
3	1	0	0	1	1
4	0	1	1	0	1
5	1	0	1	1	0

Observe that the matrix is symmetric along the main diagonal. If we define the adjacency matrix as A and the transpose as A^T , then for an undirected graph G as above, $A = A^T$.

Graph connectivity :

A connected graph is a graph in which path exists between every pair of vertices.

A strongly connected graph is a directed graph in which every pair of distinct vertices are connected with each other.

A weakly connected graph is a directed graph whose underlying graph is connected, but not strongly connected.

A complete graph is a graph in which there exists edge between every pair of vertices.

☞ Check Your Progress 1

- 1) A graph with no cycle is called _____ graph.
- 2) Adjacency matrix of an undirected graph is _____ on main diagonal.
- 3) Represent the following graphs (Figure 8.3 and Figure 8.4) by adjacency matrix:

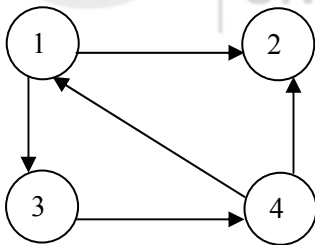


Figure 8.3: A Directed Graph

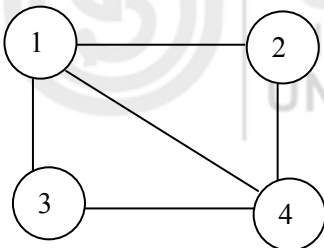


Figure 8.4: A Graph

8.3 SHORTEST PATH ALGORITHMS

A driver takes shortest possible route to reach destination. The problem that we will discuss here is similar to this kind of finding shortest route in a graph. The graphs are weighted directed graphs. The weight could be time, cost, losses other than distance designated by numerical values.

Single source shortest path problem : To find a shortest path from a single source to every vertex of the Graph.

Consider a Graph $G = (V, E)$. We wish to find out the shortest path from a single source vertex $s \in V$, to every vertex $v \in V$. The single source shortest path algorithm (Dijkstra's Algorithm) is based on assumption that no edges have negative weights.

The procedure followed to find shortest path are based on a concept called relaxation. This method repeatedly decreases the upper bound of actual shortest path of each vertex from the source till it equals the shortest-path weight. Please note that shortest path between two vertices contains other shortest path within it.

8.3.1 Dijkstra's Algorithm

Dijkstra's algorithm (named after its discover, Dutch computer scientist E.W. Dijkstra) solves the problem of finding the shortest path from a point in a graph (the *source*) to a destination with non-negative weight edge.

It turns out that one can find the shortest paths from a given source to *all* vertices (points) in a graph in the same time. Hence, this problem is sometimes called the *single-source shortest paths* problem. Dijkstra's algorithm is a greedy algorithm, which finds shortest path between all pairs of vertices in the graph. Before describing the algorithms formally, let us study the method through an example.

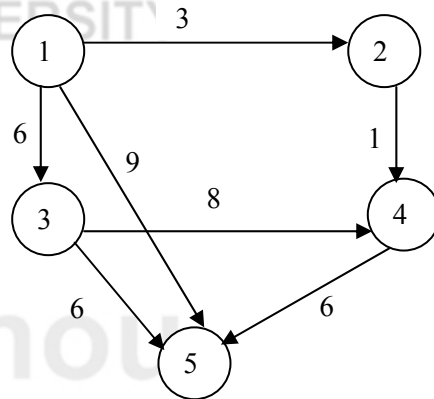


Figure 8.5: A Directed Graph with no negative edge(s)

Dijkstra's algorithm keeps two sets of vertices:

S is the set of vertices whose shortest paths from the source have already been determined

Q = V-S is the set of remaining vertices .

The other data structures needed are:

d array of best estimates of shortest path to each vertex from the source

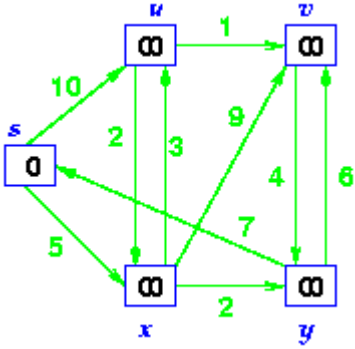
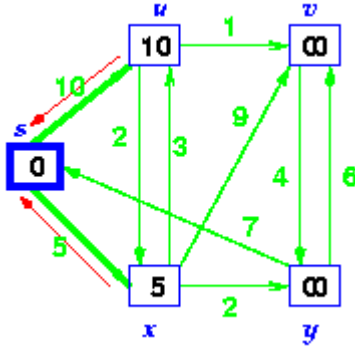
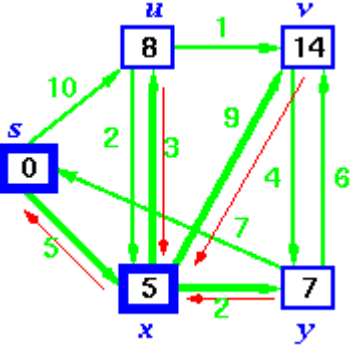
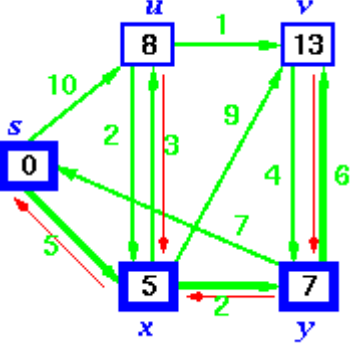
pi an array of predecessors for each vertex. *predecessor* is an array of vertices to which shortest path has already been determined.

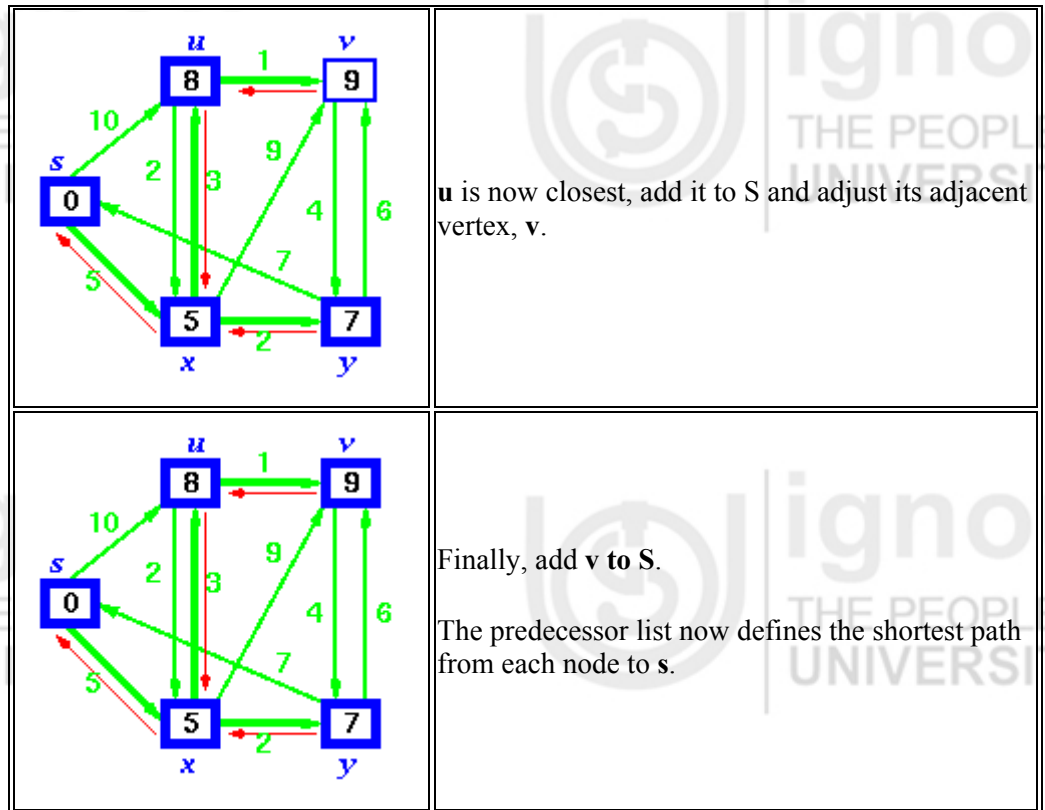
The basic operation of Dijkstra's algorithm is edge relaxation. If there is an edge from *u* to *v*, then the shortest known path from *s* to *u* can be extended to a path from *s* to *v* by adding edge (*u,v*) at the end. This path will have length $d[u]+w(u,v)$. If this is less than $d[v]$, we can replace the current value of $d[v]$ with the new value.

The predecessor list is an array of indices, one for each vertex of a graph. Each vertex entry contains the index of its predecessor in a path through the graph.

Operation of Algorithm

The following sequence of diagrams illustrate the operation of Dijkstra's Algorithm. The bold vertices indicate the vertex to which shortest path has been determined.

 <p>The diagram shows a graph with five vertices: s (source), u, v, x, and y. The source vertex s is labeled with 0. All other vertices (u, v, x, y) are labeled with infinity (∞). Edges connect s to u (weight 10), s to x (weight 5), u to v (weight 1), u to x (weight 2), u to y (weight 3), x to v (weight 9), x to y (weight 2), v to y (weight 4), and y to v (weight 6).</p>	<p>Initialize the graph, all the vertices have infinite costs except the source vertex which has zero cost</p>
 <p>The source vertex s is now bolded. Its neighbors u and x have their costs updated to 10 and 5 respectively. The other vertices (v, y) remain at infinity. Red arrows point from s to u and s to x, indicating relaxation.</p>	<p>From all the adjacent vertices, choose the closest vertex to the source s.</p> <p>As we initialized $d[s]$ to 0, it's s. (shown in bold circle)</p> <p>Add it to S</p> <p>Relax all vertices adjacent to s, i.e u and x</p> <p>Update vertices u and x by 10 and 5 as the distance from s.</p>
 <p>Vertex x is now bolded. Its neighbors u, v, and y have their costs updated to 8, 14, and 7 respectively. Red arrows point from x to u, x to v, and x to y, indicating relaxation.</p>	<p>Choose the nearest vertex, x.</p> <p>Relax all vertices adjacent to x</p> <p>Update predecessors for u, v and y.</p> <p>Predecessor of x = s</p> <p>Predecessor of v = x, s</p> <p>Predecessor of y = x, s</p> <p>add x to S</p>
 <p>Vertex y is now bolded. Its neighbor v has its cost updated to 13. Red arrows point from y to v, indicating relaxation.</p>	<p>Now y is the closest vertex. Add it to S.</p> <p>Relax v and adjust its predecessor.</p>



Dijkstra's algorithm

* Initialise d and pi *
for each vertex v in $V(g)$
 $g.d[v] := \text{infinity}$
 $g.pi[v] := \text{nil}$
 $g.d[s] := 0;$
* Set S to empty *
 $S := \{0\}$
 $Q := V(g)$
* While $(V-S)$ is not null*
while not Empty(Q)

- Sort the vertices in $V-S$ according to the current best estimate of their distance from the source
 $u := \text{Extract-Min}(Q);$
- Add vertex u , the closest vertex in $V-S$, to S ,
AddNode(S, u);
- Relax all the vertices still in $V-S$ connected to u
relax(Node u , Node v , double $w[u][v]$)
if $d[v] > d[u] + w[u][v]$ then
 $d[v] := d[u] + w[u][v]$
 $pi[v] := u$

In summary, this algorithm starts by assigning a weight of infinity to all vertices, and then selecting a source and assigning a weight of zero to it. Vertices are added to the set for which shortest paths are known. When a vertex is selected, the weights of its adjacent vertices are relaxed. Once all vertices are relaxed, their predecessor's vertices

are updated (pi). The cycle of selection, weight relaxation and predecessor update is repeated until the shortest path to all vertices has been found.

Complexity of Algorithm

The simplest implementation of the Dijkstra’s algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract-Min(Q) is simply a linear search through all vertices in Q . In this case, the running time is $\Theta(n^2)$.

8.3.2 Graphs with Negative Edge costs

We have seen that the above Dijkstra’s single source shortest-path algorithm works for graphs with non-negative edges (like road networks). The following two scenarios can emerge out of negative cost edges in a graph:

- Negative edge with non-negative weight cycle reachable from the source.
- Negative edge with non-negative weight cycle reachable from source.

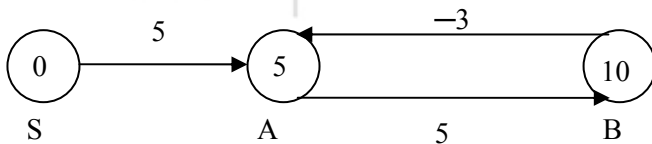


Figure 8.6 : A Graph with negative edge and non-negative weight cycle

The net weight of the cycle is 2(non-negative)(refer to Figure 8.6).

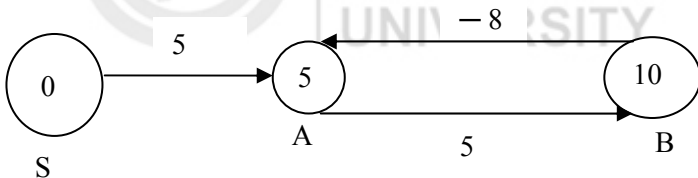


Figure 8.7: A graph with negative edge and negative weight cycle

The net weight of the cycle is -3 (negative) (refer to Figure 8.7). The shortest path from A to B is not well defined as the shortest path to this vertex are infinite, i.e., by traveling each cycle we can decrease the cost of the shortest path by 3, like (S, A, B) is path (S, A, B, A, B) is a path with less cost and so on.

Dijkstra’s Algorithm works only for directed graphs with non-negative weights (cost).

8.3.3 Acyclic Graphs

A path in a directed graph is said to form a cycle if there exists a path (A,B,C,.....P) such that A = P. A graph is called acyclic if there is no cycle in the graph.

8.3.4 All Pairs Shortest Paths Algorithm

In the last section, we discussed about shortest path algorithm which starts with a single source and finds shortest path to all vertices in the graph. In this section, we shall discuss the problem of finding shortest path between all pairs of vertices in a graph. This problem is helpful in finding distance between all pairs of cities in a road atlas. All pairs shortest paths problem is mother of all shortest paths problems.

In this algorithm, we will represent the graph by adjacency matrix.

The weight of an edge C_{ij} in an adjacency matrix representation of a directed graph is represented as follows

$$C_{ij} = \begin{cases} 0 & \text{if } i = j \\ \text{weight of the directed edge from } i \text{ to } j \text{ i.e. } (i,j) & \text{if } i \neq j \text{ and } (i,j) \text{ belongs to } E \\ \infty & \text{if } i \neq j \text{ and } (i,j) \text{ does not belong to } E \end{cases}$$

Given a directed graph $G = (V, E)$, where each edge (v, w) has a non-negative cost $C(v, w)$, for all pairs of vertices (v, w) to find the lowest cost path from v to w .

The All pairs shortest paths problem can be considered as a generalisation of single-source-shortest-path problem, by using Dijkstra's algorithm by varying the source node among all the nodes in the graph. If negative edge(s) is allowed, then we can't use Dijkstra's algorithm.

In this section we shall use a recursive solution to all pair shortest paths problem known as Floyd-Warshall algorithm, which runs in $O(n^3)$ time.

This algorithm is based on the following principle. For graph G let $V = \{1, 2, 3, \dots, n\}$. Let us consider a sub set of the vertices $\{1, 2, 3, \dots, k\}$. For any pair of vertices that belong to V , consider all paths from i to j whose intermediate vertices are from $\{1, 2, 3, \dots, k\}$. This algorithm will exploit the relationship between path p and shortest path from i to j whose intermediate vertices are from $\{1, 2, 3, \dots, k-1\}$ with the following two possibilities:

1. If k is not an intermediate vertex in the path p , then all the intermediate vertices of the path p are in $\{1, 2, 3, \dots, k-1\}$. Thus, shortest path from i to j with intermediate vertices in $\{1, 2, 3, \dots, k-1\}$ is also the shortest path from i to j with vertices in $\{1, 2, 3, \dots, k\}$.
2. If k is an intermediate vertex of the path p , we break down the path p into path p_1 from vertex i to k and path p_2 from vertex k to j . So, path p_1 is the shortest path from i to k with intermediate vertices in $\{1, 2, 3, \dots, k-1\}$.

During iteration process we find the shortest path from i to j using only vertices $\{1, 2, 3, \dots, k-1\}$ and in the next step, we find the cost of using the k^{th} vertex as an intermediate step. If this results in lower cost, then we store it.

After n iterations (all possible iterations), we find the lowest cost path from i to j using all vertices (if necessary).

Note the following:

Initialize the matrix

$C[i][j] = \infty$ if (i, j) does not belong to E for graph $G = (V, E)$

Initially, $D[i][j] = C[i][j]$

We also define a path matrix P where $P[i][j]$ holds intermediate vertex k on the least cost path from i to j that leads to the shortest path from i to j .

Algorithm (All Pairs Shortest Paths)

N = number of rows of the graph

$D[i][j] = C[i][j]$

For k from 1 to n

 Do for $i = 1$ to n

 Do for $j = 1$ to n

$D[i][j] = \text{minimum}(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

 Enddo

 Enddo

Enddo

where $d_{ij}^{(k-1)}$ = minimum path from i to j using $k-1$ intermediate vertices

where $d_{ik}^{(k-1)}$ = minimum path from j to k using $k-1$ intermediate vertices

where $d_{kj}^{(k-1)}$ = minimum path from k to j using $k-1$ intermediate vertices

Program 8.1 gives the program segment for the All pairs shortest paths algorithm.

AllPairsShortestPaths(int N, Matrix C, Matrix P, Matrix D)

```
{
    int i, j, k

    if i = j then C[i][j] = 0
    for ( i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            D[i][j] = C[i][j];
            P[i][j] = -1;
        }
        D[i][j] = 0;
    }

    for (k=0; k<N; k++)
    {
        for (i=0; i<N; i++)
        {
            for (j=0; J<N; J++)
            {
                if (D[i][k] + D[k][j] < D[i][j])
                {
                    D[i][j] = D[i][k] + D[k][j];
                    P[i][j] = k;
                }
            }
        }
    }
}
```

/****** End *****/

Program 8.1 : Program segment for All pairs shortest paths algorithm

From the above algorithm, it is evident that it has $O(N^3)$ time complexity.

Shortest path algorithms had numerous applications in the areas of Operations Research, Computer Science, Electrical Engineering and other related areas.

☞ Check Your Progress 2

- 1) _____ is the basis of Dijkstra’s algorithm
 - 2) What is the complexity of All pairs shortest paths algorithm?
-

8.4 MINIMUM COST SPANNING TREES

A *spanning tree* of a graph is just a subgraph that contains all the vertices and is a tree (with no cycle). A graph may have many spanning trees.

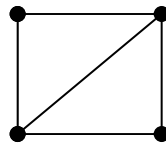


Figure 8.8: A Graph

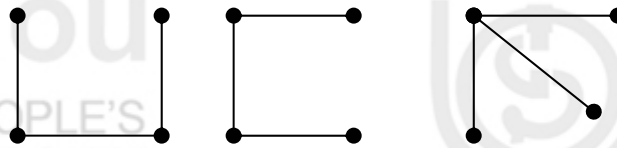


Figure 8.9 : Spanning trees of the Graph of Figure 8.8

Consider the graph of *Figure 8.8*. It’s spanning trees are shown in *Figure 8.9*. Now, if the graph is a weighted graph (length associated with each edge). The weight of the tree is just the sum of weights of its edges. Obviously, different spanning trees have different weights or lengths. Our objective is to find the minimum length (weight) spanning tree.

Suppose, we have a group of islands that we wish to link with bridges so that it is possible to travel from one island to any other in the group. The set of bridges which will enable one to travel from any island to any other at minimum capital cost to the government is the minimum cost spanning tree.

8.4.1 Kruskal’s Algorithm

Kruskal’s algorithm uses the concept of *forest* of trees. Initially the forest consists of **n** single node trees (and no edges). At each step, we add one (the cheapest one) edge so that it links two trees together. If it forms a cycle, it would simply mean that it links two nodes that were already connected. So, we reject it.

The steps in Kruskal's Algorithm are as follows:

1. The forest is constructed from the graph G - with each node as a separate tree in the forest.
2. The edges are placed in a priority queue.
3. Do until we have added $n-1$ edges to the graph,
 1. Extract the cheapest edge from the queue.
 2. If it forms a cycle, then a link already exists between the concerned nodes. Hence reject it.
 3. Else add it to the forest. Adding it to the forest will join two trees together.

The forest of trees is a partition of the original set of nodes. Initially all the trees have exactly one node in them. As the algorithm progresses, we form a union of two of the trees (sub-sets), until eventually the partition has only one sub-set containing all the nodes.

Let us see the sequence of operations to find the Minimum Cost Spanning Tree(MST) in a graph using Kruskal's algorithm. Consider the graph of Figure 8.10., Figure 8.11 shows the construction of MST of graph of Figure 8.10.

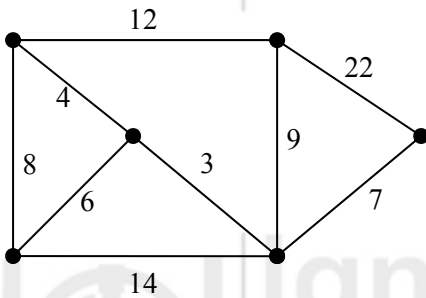
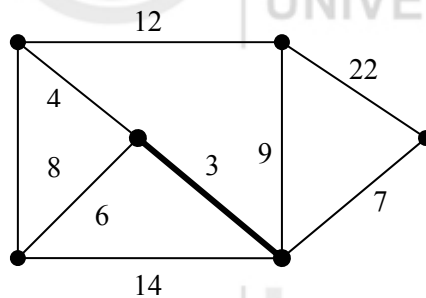
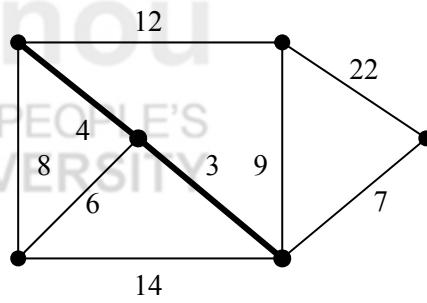


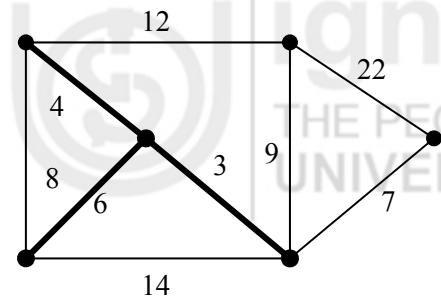
Figure 8.10 : A Graph



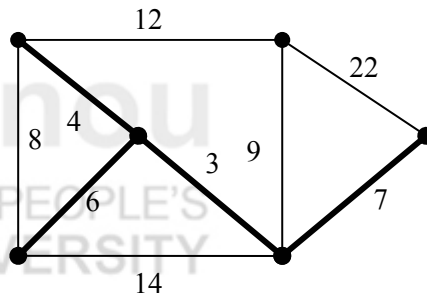
Step 1



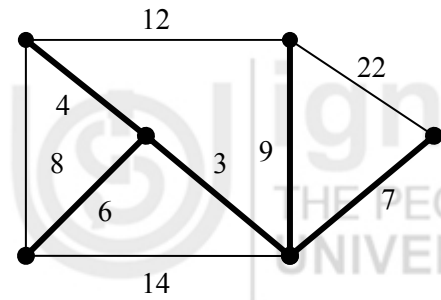
Step 2



Step 3



Step 4



Step 5

Figure 8.11 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Kruskal's algorithm

The following are various steps in the construction of MST for the graph of Figure 8.10 using Kruskal's algorithm.

Step 1 : The lowest cost edge is selected from the graph which is not in MST (initially MST is empty). The lowest cost edge is 3 which is added to the MST (shown in bold edges)

Step 2: The next lowest cost edge which is not in MST is added (edge with cost 4).

Step 3 : The next lowest cost edge which is not in MST is added (edge with cost 6).

Step 4 : The next lowest cost edge which is not in MST is added (edge with cost 7).

Step 5 : The next lowest cost edge which is not in MST is 8 but will form a cycle. So, it is discarded . The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

8.4.2 Prim's Algorithm

Prim's algorithm uses the concept of sets. Instead of processing the graph by sorted order of edges, this algorithm processes the edges in the graph randomly by building up disjoint sets.

It uses two disjoint sets A and \bar{A} . Prim's algorithm works by iterating through the nodes and then finding the shortest edge from the set A to that of set \bar{A} (i.e. outside A), followed by the addition of the node to the new graph. When all the nodes are processed, we have a minimum cost spanning tree.

Rather than building a sub-graph by adding one edge at a time, Prim's algorithm builds a tree one vertex at a time.

The steps in Prim's algorithm are as follows:

Let G be the graph with n vertices for which minimum cost spanning tree is to be generated.

Let T be the minimum spanning tree.

Let T be a single vertex x .

while (T has fewer than n vertices)

```
{
    find the smallest edge connecting  $T$  to  $G-T$ 
    add it to  $T$ 
}
```

Consider the graph of Figure 8.10. Figure 8.12 shows the various steps involved in the construction of Minimum Cost Spanning Tree of graph of Figure 8.10.

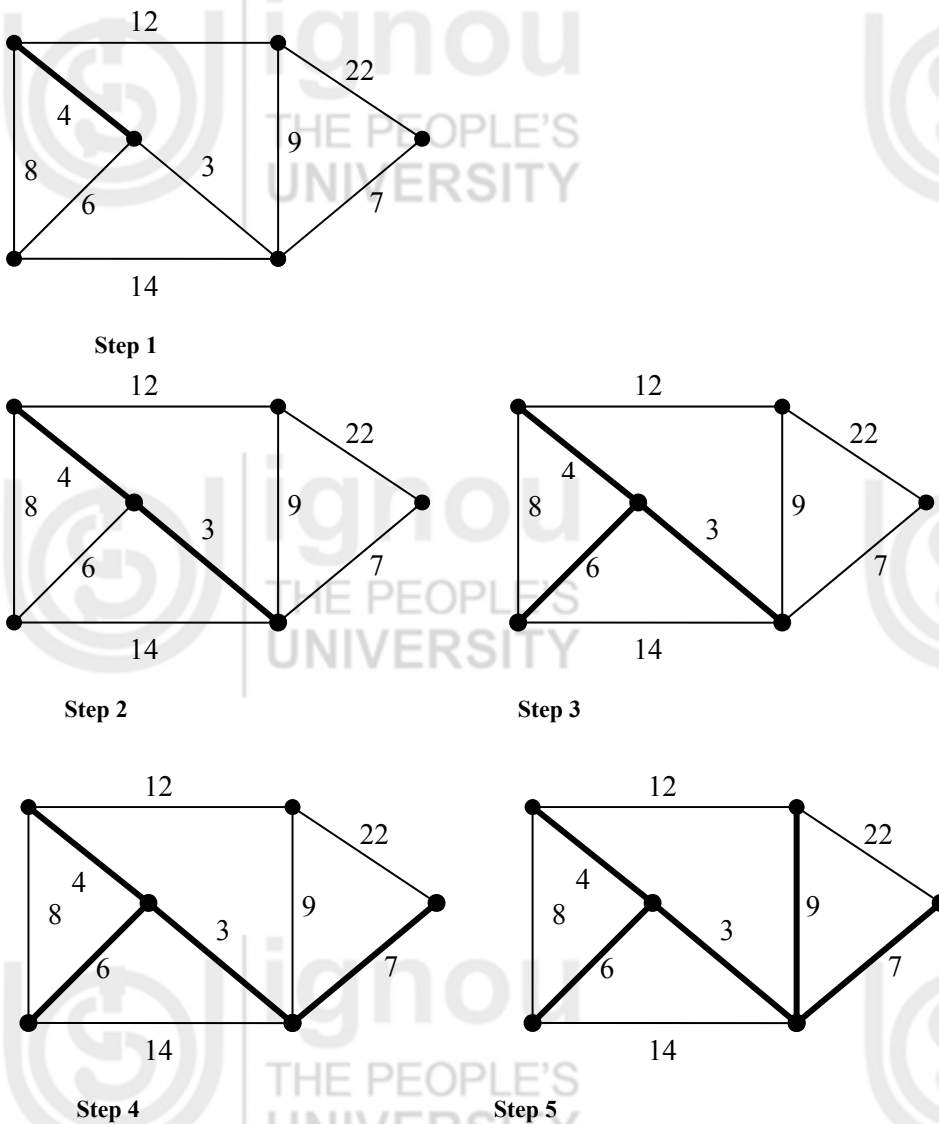


Figure 8.12 : Construction of Minimum Cost Spanning Tree for the Graph of Figure 8.10 by application of Prim's algorithm

The following are various steps in the construction of MST for the graph of Figure 8.10 using Prim's algorithm.

Step 1 : We start with a single vertex (node). Now the set A contains this single node and set A' contains rest of the nodes. Add the edge with the lowest cost from A to A' . The edge with cost 4 is added.

Step 2: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 3) is selected and added to MST.

Step 3: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 6) is selected and added to MST.

Step 4: Lowest cost path from shaded portion of the graph to the rest of the graph (edge with cost 7) is selected and added to MST.

Step 5: The next lowest cost edge to the set not in MST is 8 but forms a cycle. So, it is discarded. The next lowest cost edge 9 is added. Now the MST contains all the vertices of the graph. This results in the MST of the original graph.

Comparison of Kruskal's algorithm and Prim's algorithm

	Kruskal's algorithm	Prim's algorithm
Principle	Based on generic minimum cost spanning tree algorithms	A special case of generic minimum cost spanning tree algorithm. Operates like Dijkstra's algorithm for finding shortest path in a graph.
Operation	Operates on a single set of edges in the graph	Operates on two disjoint sets of edges in the graph
Running time	$O(E \log E)$ where E is the number of edges in the graph	$O(E \log V)$, which is asymptotically same as Kruskal's algorithm

For the above comparison, it may be observed that for dense graphs having more number of edges for a given number of vertices, Prim's algorithm is more efficient.

8.4.3 Applications

The minimum cost spanning tree has wide applications in different fields. It represents many complicated real world problems like:

1. Minimum distance for travelling all cities at most one (travelling salesman problem).
2. In electronic circuit design, to connect n pins by using n-1 wires, using least wire.
3. Spanning tree also finds their application in obtaining independent set of circuit equations for an electrical network.

8.5 BREADTH FIRST SEARCH (BFS)

When BFS is applied, the vertices of the graph are divided into two categories. The vertices, which are visited as part of the search and those vertices, which are not visited as part of the search. The strategy adopted in breadth first search is to start search at a vertex (source). Once you started at source, the number of vertices that are visited as part of the search is 1 and all the remaining vertices need to be visited. Then, search the vertices which are adjacent to the visited vertex from left to order. In this way, all the vertices of the graph are searched.

Consider the digraph of *Figure 8.13*. Suppose that the search started from S. Now, the vertices (from left to right) adjacent to S which are not visited as part of the search are B, C, A. Hence, B, C and A are visited after S as part of the BFS. Then, F is the unvisited vertex adjacent to B. Hence, the visit to B, C and A is followed by F. The unvisited vertex adjacent of C is D. So, the visit to F is followed by D. There are no

unvisited vertices adjacent to A. Finally, the unvisited vertex E adjacent to D is visited.

Hence, the sequence of vertices visited as part of BFS is S, B, C, A, F, D and E.

8.6 DEPTH FIRST SEARCH (DFS)

The strategy adopted in depth first search is to search deeper whenever possible. This algorithm repeatedly searches deeper by visiting unvisited vertices and whenever an unvisited vertex is not found, it backtracks to previous vertex to find out whether there are still unvisited vertices.

As seen, the search defined above is inherently recursive. We can find a very simple recursive procedure to visit the vertices in a depth first search. The DFS is more or less similar to pre-order tree traversal. The process can be described as below:

Start from any vertex (source) in the graph and mark it visited. Find vertex that is adjacent to the source and not previously visited using adjacency matrix and mark it visited. Repeat this process for all vertices that is not visited, if a vertex is found visited in this process, then return to the previous step and start the same procedure from there.

If returning back to source is not possible, then DFS from the originally selected source is complete and start DFS using any unvisited vertex.

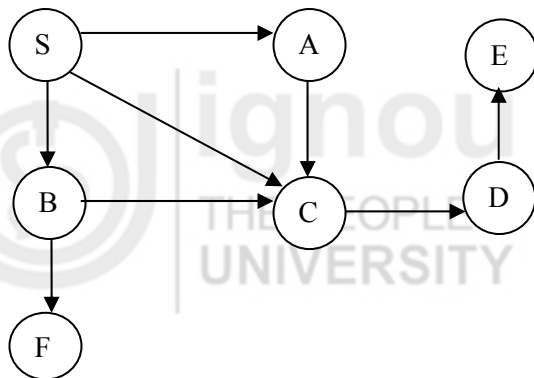


Figure 8.13 : A Digraph

Consider the digraph of *Figure 8.13*. Start with S and mark it visited. Then visit the next vertex A, then C and then D and at last E. Now there are no adjacent vertices of E to be visited next. So, now, backtrack to previous vertex D as it also has no unvisited vertex. Now backtrack to C, then A, at last to S. Now S has an unvisited vertex B. Start DFS with B as a root node and then visit F. Now all the nodes of the graph are visited.

Figure 8.14 shows a DFS tree with a sequence of visits. The first number indicates the time at which the vertex is visited first and the second number indicates the time at which the vertex is visited during back tracking.

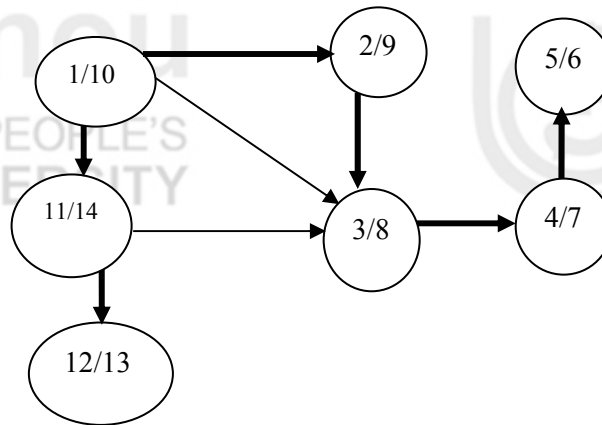


Figure 8.14 : DFS tree of digraph of Figure 8.13

The DFS forest is shown with shaded arrow in Figure 8.14.

Algorithm for DFS

- Step 1: Select a vertex in the graph and make it the source vertex and mark it visited.
- Step 2: Find a vertex that is adjacent to the source vertex and start a new search if it is not already visited.
- Step 3: Repeat step 2 using a new source vertex. When all adjacent vertices are visited, return to previous source vertex and continue search from there.

If n is the number of vertices in the graph and the graph is represented by an adjacency matrix, then the total time taken to perform DFS is $O(n^2)$. If G is represented by an adjacency list and the number of edges of G are e , then the time taken to perform DFS is $O(e)$.

8.7 FINDING STRONGLY CONNECTED COMPONENTS

A beautiful application of DFS is finding a strongly connected component of a graph.

Definition: For graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, we define a strongly connected components as follows:

U is a sub set of V such that u, v belongs to U such that, there is a path from u to v and v to u . That is, all pairs of vertices are reachable from each other.

In this section we will use another concept called transpose of a graph. Given a directed graph G a transpose of G is defined as G^T . G^T is defined as a graph with the same number of vertices and edges with only the direction of the edges being reversed. G^T is obtained by transposing the adjacency matrix of the directed graph G .

The algorithm for finding these strongly connected components uses the transpose of G , G^T .

$$G = (V, E), G^T = (V, E^T), \text{ where } E^T = \{ (u, v) : (v, u) \text{ belongs to } E \}$$

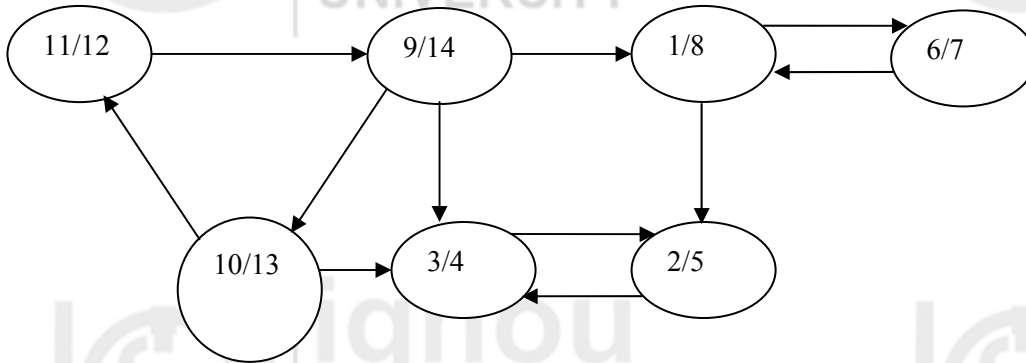


Figure 8.15: A Digraph

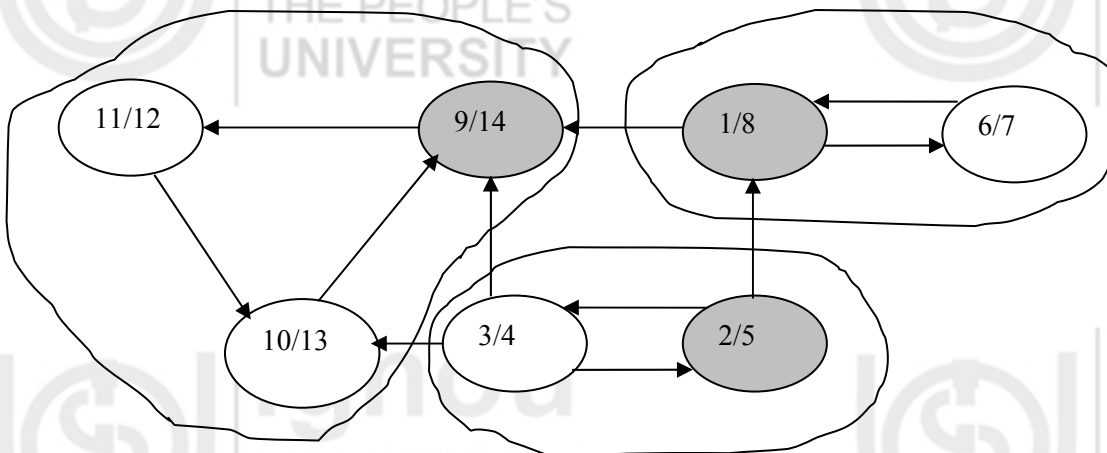


Figure 8.16: Transpose and strongly connected components of digraph of Figure 8.15

Figure 8.15 shows a directed graph with sequence in DFS (first number of the vertex shows the discovery time and second number shows the finishing time of the vertex during DFS. Figure 8.16 shows the transpose of the graph in Figure 8.15 whose edges are reversed. The strongly connected components are shown in zig-zag circle in Figure 8.16.

To find strongly connected component we start with a vertex with the highest finishing time and start DFS in the graph G^T and then in decreasing order of finishing time. DFS with vertex with finishing time 14 as root finds a strongly connected component. Similarly, vertices with finishing times 8 and then 5, when selected as source vertices also lead to strongly connected components.

Algorithm for finding strongly connected components of a Graph:

Strongly Connected Components (G)

where $d[u]$ = discovery time of the vertex u during DFS, $f[u]$ = finishing time of a vertex u during DFS, G^T = Transpose of the adjacency matrix

- Step 1: Use DFS(G) to compute $f[u] \forall u \in V$
- Step 2: Compute G^T
- Step 3: Execute DFS in G^T

Step 4: Output the vertices of each tree in the depth-first forest of Step 3 as a separate strongly connected component.

Check Your Progress 3

- 1) Which graph traversal uses a queue to hold vertices that are to be processed next ?
.....
.....
- 2) Which graph traversal is recursive by nature?
.....
.....
- 3) For a dense graph, Prim’s algorithm is faster than Kruskal’s algorithm
True/False
- 4) Which graph traversal technique is used to find strongly connected component of a graph?
.....
.....

8.8 SUMMARY

Graphs are data structures that consist of a set of vertices and a set of edges that connect the vertices. A graph where the edges are directed is called directed graph. Otherwise, it is called an undirected graph. Graphs are represented by adjacency lists and adjacency matrices. Graphs can be used to represent a road network where the edges are weighted as the distance between the cities. Finding the minimum distance between single source and all other vertices is called single source shortest path problem. Dijkstra’s algorithm is used to find shortest path from a single source to every other vertex in a directed graph. Finding shortest path between every pair of vertices is called all pairs shortest paths problem.

A spanning tree of a graph is a tree consisting of only those edges of the graph that connects all vertices of the graph with minimum cost. Kruskal’s and Prim’s algorithms find minimum cost spanning tree in a graph. Visiting all nodes in a graph systematically in some manner is called traversal. Two most common methods are depth-first and breadth-first searches.

8.9 SOLUTIONS/ANSWERS

Check Your Progress 1

- 1) an acyclic
- 2) symmetric
- 3) The adjacency matrix of the directed graph and undirected graph are as follows:

$$\begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

(Refer to Figure 8.3)

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

(Refer to Figure 8.3)

Check Your Progress 2

- 1) Node relaxation
- 2) $O(N^3)$

Check Your Progress 3

- 1) BFS
- 2) DFS
- 3) True
- 4) DFS

8.10 FURTHER READINGS

1. *Fundamentals of Data Structures in C++* by E.Horowitz, Sahni and D.Mehta; Galgotia Publications.
2. *Data Structures and Program Design in C* by Kruse, C.L.Tonodo and B.Leung; Pearson Education.
3. *Data Structures and Algorithms* by Alfred V.Aho; Addison Wesley.

Reference Websites

- <http://www.onesmartclick.com/engineering/data-structure.html>
<http://msdn.microsoft.com/vcsharp/programming/datastructures/>
http://en.wikipedia.org/wiki/Graph_theory