Assignment 8.2

1. Obtain the linked list representation and adjacency matrix representation of the graphs as shown in Figures 8.2(a) and 8.2(b).
2. Suppose G is a simple undirected graph with m vertices $v_1, v_2, \ldots, v_m$ and n edges $e_1, e_2, \ldots, e_n$. The *incident matrix* of G is the $m \times n$ matrix $M = [a_{ij}]$, where

$$a_{ij} = 1, \text{ if node } v_i \text{ belongs to edge } e_j$$
$$= 0, \text{ otherwise.}$$

Find the incidence matrix of the graph as shown in Figure 8.10.

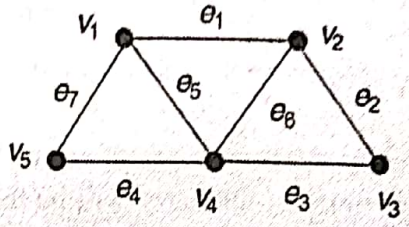


Fig. 8.10

Advantages of adjacency matrix representation of graphs

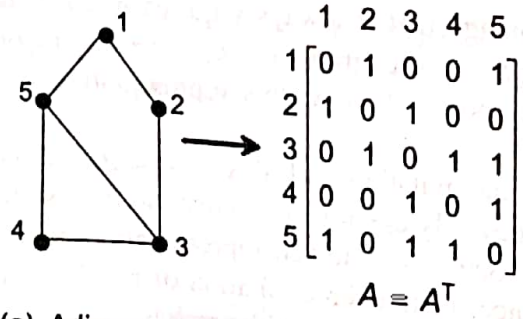
In this section, we are to elaborate how some extra informations can be retrieved easily from a graph if it is represented with an adjacency matrix.

Lemma 8.1 If A be an adjacency matrix of a graph G, and if $A = A^T$, where A^T is the transpose matrix of A, then G is a simple undirected graph.

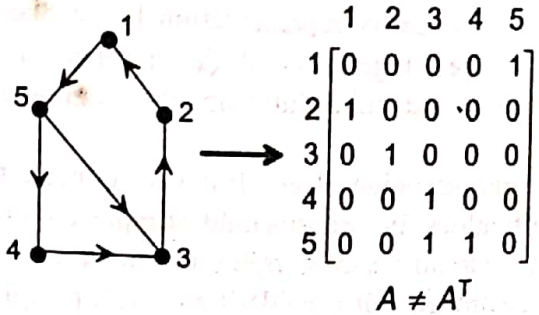
Proof Let a_{ij} denotes an entry in the i-th row and j-th column of the matrix A. Now, in case of a simple (no loop, no parallel edges) undirected graph, $a_{ij} = 1(0)$ implies that there is a path (no path) from v_i to v_j then there is also path (no path) from v_j to v_i, that is, $a_{ji} = 1(0)$. Thus, for a simple undirected graph $a_{ij} = a_{ji}$. Thus, the matrix is symmetric. Since, for any symmetric matrix $A = A^T$, the result follows.

Corollary 1: For a simple digraph, $A \neq A^T$.

Example: Figure 8.11 shows two simple graphs and their adjacency matrices. For a simple undirected graph, $A = A^T$; and for a simple directed graph, $A \neq A^T$.



(a) Adjacency matrix of a simple directed graph is symmetric



(b) Adjacency matrix of a simple digraph is asymmetric

Fig. 8.11 Illustration of Lemma 8.1.

Corollary 2: If the graph is simple then all the diagonal entries are 0.

Lemma 8.2 Suppose G is a digraph and A be its adjacency matrix. Then the diagonal entries of $A \cdot A^T$ gives the outdegrees of all vertices in G, and the diagonal entries of $A^T \cdot A$ gives the indegrees of all vertices in G.

Proof Let us first consider the case of $A \cdot A^T$. Let $B = A \cdot A^T$ and suppose b_{ij} be an element in the i-th row and j-th column of B. In general,

$$b_{ij} = \sum_{k=1}^{n} a_{ik} \cdot a_{jk}$$

where $i, j = 1, 2, ..., n$. The value of $a_{ik} \cdot a_{jk} = 1$ if and only if both a_{ik} and a_{jk} are 1; otherwise $a_{ik} \cdot a_{jk} = 0$. Now, $a_{ik} = 1$ if $(v_i, v_k) \in E$ and $a_{jk} = 1$ if $(v_j, v_k) \in E$. If both (v_i, v_k) and (v_j, v_k) are the edges of the graph for some fixed k, then we get a contribution of 1 in the summation expressing b_{ij}. Therefore, the values of b_{ij} shows the number of vertices which are connected both from v_i and v_j.

Now, if $i = j$, then

$$b_{ii} = \sum_{k=1}^{n} a_{ik}^2$$

and $a_{ik}^2 = 1$ if $a_{ik} = 1$, that is, if $(v_i, v_k) \in E$, or in other words, there is an edge directed from v_i to v_k. Thus, total contribution in summation expression gives the total outgoing edges from v_i. Therefore, the diagonal entries of $A \cdot A^T$ implies the outdegrees of the vertices. Similar conclusion can be drawn for $A^T \cdot A$ but for indegrees and proof of which is left as an exercise.

Example: Figure 8.12 shows a digraph with its adjacency matrix. $A \cdot A^T$ and $A^T \cdot A$ are computed as shown:

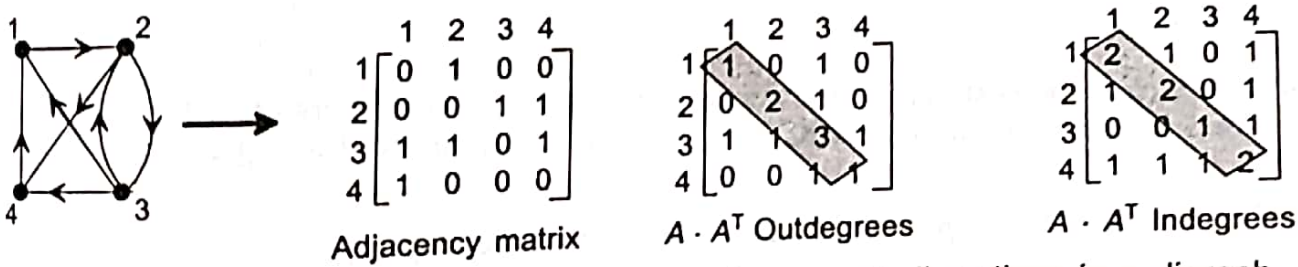


Fig. 8.12 Computation of indegrees and outdegrees of all vertices in a digraph.

Lemma 8.3 Let A be an adjacency matrix of a digraph. $A^L = [a_{ij}^L]$, gives the number of paths of length L from v_i to v_j, where A^L is the L-th *power matirx* of A.

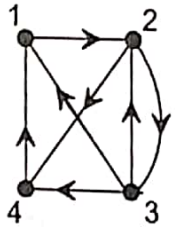
Proof Naturally an entry of 1 in the i-th row and j-th column of A shows the existence of an edge (v_i, v_j), that is, a path of length 1 from v_i to v_j. Let us denote the elements of A^2 by a_{ij}^2. Then

$$a_{ij}^2 = \sum_{k=1}^{n} a_{ik} \cdot a_{kj}$$

For any value of k, $a_{ik} \cdot a_{ij} = 1$ if both a_{ik} and a_{kj} equal 1, that is, iff (v_i, v_k) and (v_k, v_j) are the

edges of the graph. For each such k, we get a contribution of 1 in the summation expression. Now, (v_i, v_k) and (v_k, v_j) imply that there is a path from v_i to v_j of length 2. Therefore, $a^2_{ij} = 1$ means a path of exactly length 2 from v_i to v_j. By a similar argument, one can show that the element in the i-th row and j-th column of A^3 gives the number of paths of exactly length 3 from v_i to v_j. Therefore, it follows the proof.

Example: For the digraph G in Figure 8.13, A^2, A^3 and A^4 are computed as shown:



$$A = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 1 & 1 \\ 2 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

$$A^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 \\ 2 & 2 & 1 & 2 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array} \qquad A^4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 2 & 1 & 1 \\ 2 & 2 & 2 & 3 \\ 3 & 3 & 2 & 3 \\ 2 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Fig. 8.13 Computation of number of paths in a digraph.

As illustrated in the figure, in case of A^3, $a^3_{23} = 1$, that is there is only one path of length 3 from vertex 2 to vertex 3 which is [2-3-2-3], on the other hand, from A^4 it is observed that $a^4_{23} = 2$, that is, there are 2 different paths of length 4, such as [2-4-1-2-3], [2-3-1-2-3].

Note: In the above computation, A^L not necessarily gives the count of *elementary paths* (the path which does not cross one vertex more than once).

Path matrix

Before going to the next important Lemma, let us introduce the concept of *path* matrix. Let G be a simple graph with n vertices $v_1, v_2, \ldots, v_n$. An $n \times n$ matrix $P = [p_{ij}]$ whose entries are defined as:

$$p_{ij} = 1, \quad \text{if there exists a path from } v_i \text{ to } v_j$$
$$= 0, \quad \text{otherwise}$$

then P is called the path matrix or *reachability* matrix of the graph G.

Let us discuss how path matrix P of a given graph G can be obtained from its adjacency matrix A. From the adjacency matrix of A we can immediately determine whether there exists an edge from v_i to v_j. Also from the matrix A^L, where L is some positive integer, we can establish the number of paths of length L from v_i to v_j. Add the matrices $A, A^2, A^3, \ldots, A^L$ so that

$$B^L = A + A^2 + A^3 + \ldots + A^L$$

then from the matrix B^L, we can determine the number of paths of length less than or equal to L from any v_i to v_j. If we wish to determine whether v_j is reachable from v_i, it would be necessary to find whether there exists a path of any length from v_i to v_j.

Lemma 8.4 In a simple digraph, the length of any elementary path is less than or equal to $n - 1$, where n is the number of vertices in the graph. Similarly, the length of any elementary cycle does not exceed n.

Proof The proof is based on the fact that in any elementary path the node appearing in the sequence are distinct. The number of distinct nodes in an elementary path of length l is $l + 1$, since there are only n distinct nodes in the graph, we cannot have an elementary path of length greater than $n - 1$.

For an elementary cycles of length l, the sequence contains l distinct nodes hence the result.

In a graph, all the elementary cycles or path can be determined from the matrix B_n where

$$B_n = A + A^2 + A^3 + \cdots + A^n$$

n being the number of vertices in the graph and A being the adjacency matrix of the graph.

For an example, B_4 of the graph as presented in Figure 8.13, can be obtained as:

$$B_4 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 3 & 4 & 2 & 3 \\ 5 & 5 & 4 & 6 \\ 7 & 7 & 4 & 7 \\ 3 & 2 & 1 & 2 \end{bmatrix} \end{array} \qquad P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The element in the i-th row and j-th column of B_4 shows the number of paths having length 4 or less than those exist between v_i to v_j in the graph.

From the matrix B_n, we can obtain the path matrix of the graph as follows: Let $P = [p_{ij}]$ be a path matrix. Then $p_{ij} = 1$ iff there is a non-zero element in the i, j entry of the matrix B_n else $p_{ij} = 0$. Thus, the path matrix of the running example is shown above as P. [*Note:* The path matrix only shows the presence or absence of at least one path between a pair of points and also the presence or absence of a cycle at any node.]

Lemma 8.5

(a) A vertices v_i contains a cycle if the i, i entry in the path matrix P is 1.

(b) A graph is strongly connected if for all $v_i, v_j \in G$, both the i, j entry and j, i entry in the path matrix are 1.

Proof Proof can be easily followed from the previous discussion, and left as an exercise.

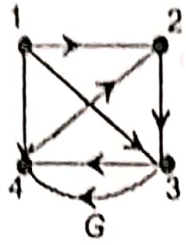
As an example, for the graph as given in Figure 8.13, and whose path matrix P is obtained as above, we can conclude that, it is strongly connected and all the vertices have cycle.

Lemma 8.6 Let P be a path matrix of a graph G. If P^T is the transpose of the matrix P, then the i-th row of the matrix $P * P^T$, [which is obtained by the element-wise product (AND) of the elements] gives the strong component containing v_i.

Proof If v_j is reachable from v_i then clearly $p_{ij} = 1$; also if v_i is reachable from v_j, then $p_{ji} = 1$. Therefore the element in the i-th row and j-th column of $P * P^T$ is 1 iff v_i and v_j are mutually reachable. This is true for all j. Hence the result.

Example: As an example, consider the graph as given in Figure 8.14. Its path matrices P and $P * P^T$ are calculated as shown.

From $P * P^T$, strong component is the sub-graph containing the vertices 2, 3 and 4, that is, any vertex is reachable from all these vertices.

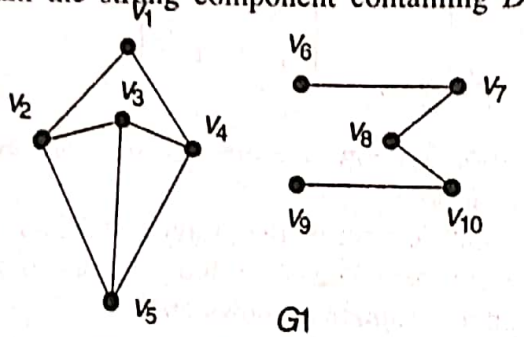


$$P = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array} \qquad P * P^T = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Fig. 8.14 Path matrices and strong component of G.

Assignment 8.3

1. For the graph $G1$ as shown in Figure 8.15(a), obtain the adjacency matrix and incidence matrix.
2. Obtain the adjacency matrix of the digraph $G2$ as given in Figure 8.15(b). Obtain the path matrix of this graph and then decide whether it is strongly connected or not.
   Obtain the strong component containing D.



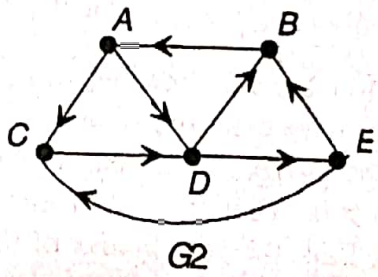


Fig. 8.15 Graphs for Assignment 8.3.

8.4 OPERATIONS ON GRAPHS

The important operations possible on graphs are listed below:

Insertion

(a) To insert a vertex and hence establishing connectivity with other vertices in the existing graph.

(b) To insert an edge between two existing vertices in the graph.

Deletion

(a) To delete a vertex from the graph.

(b) To delete an edge from the graph.

04	Y	–	Y	–
05	–	Y	Y	–
06	–	Y	–	Y
07	–	–	Y	–

(a) Draw a graph indicating people which can communicate directly with each other.
(b) Suppose, 01 generally communicates with 03 with the help of 02. Due to some acute reason 02 is not available. Can 01 communicates with 03? How?
(c) How many minimum interpreters 03 should take if he wants to communicate with 07.
(d) 01 wants to send a message to each officer. Whenever a message comes to an officer he reads it and transmits it to another officers possibly after translation to someone who has not read it. Finally, it returns to 01 so that he can verify that nothing was lost in transition. Using the graph, give a routing of the message.

It is not possible to discuss all the problems where graph structures are involved because is domain is very large; however, we limit our discussions with a few important of them which ay significant roles in various applications.

In order to maintain the generality, we will discuss them as graph theoretic problems. onversion of a particular problem into this general graph theoretic problem will rest on the ader. The major graph theoretic problems are listed as below:

1. Shortest path problem
2. Topological sorting of a graph
3. Spanning trees
4. Connectivity of a graph
5. Euler's path and Hamiltonian path
6. Binary decision diagram.

/e shall discuss all these problems separately in the following sub-sections.

.5.1 Shortest Path Problem

his problem of a graph is about finding a path between two vertices in such a way that this ath will satisfy some criteria of optimization. For example, for a non-weighted graph, the umber of edges will be minimum and for weighted graph, the sum of weights on all edges in ie path will be minimum. This problem has varieties of classical solutions. A few important lgorithms are: Warshall's algorithm, Floyd's algorithm and Dijkstra's algorithm.

Warshall's algorithm

This is a classical algorithm by which we can determine whether there is a path from any vertex v_i to another vertex v_j either directly or through one or more intermediate vertices. In other words, we can test the reachability of all pairs of vertices in a graph. In Section 8.3.3, we have introduced the notion of path matrix for a graph. Also we have seen how the path matrix can

be computed from the adjacency matrix, A, of a graph using Lemma 8.4. This computation involves the computation of A^2, A^3, ..., A^n and then B_n. The method is computationally not efficient at all. To compute the path matrix of a given graph another elegant method is *Warshall's* algorithm. This algorithm treats the entries in the adjacency matrix as bit entries and performs AND ($\wedge$), and OR ($\vee$)—the Boolean operations on them. The heart of the algorithm is a triple of loops, which operates very much like the loops in the classic algorithms for matrix multiplication.

Let us first state this algorithm and then we will elaborate how it works.

Algorithm WARSHALL( )

Input: A graph G whose pointer to its adjacency matrix is Gptr and vertices are labelled as 1, 2, ..., N; N being the number of vertices in the graph.

Output: The path matrix P.

Data structure: Matrix representation of graph with pointer as Gptr.

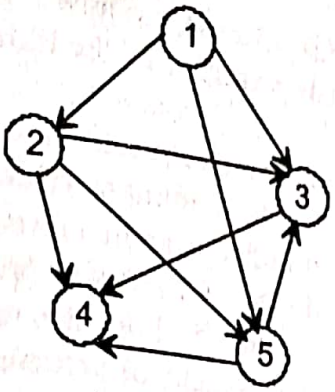
Steps:

/*Initialization of path matrix P with the adjacency matrix Gptr*/

1. For i = 1 to N do
   1. For j = 1 to N do
      1. $P[i][j]$ = Gptr$[i][j]$
   2. EndFor
2. EndFor
3. For k = 1 to N do
   1. For i = 1 to N do
      1. For j = 1 to N do
         1. $P[i][j] = P[i][j] \vee (P[i][k] \wedge P[k][j])$
      2. EndFor
   2. EndFor
4. EndFor
5. Return(P)
6. Stop

The functionality of this algorithm can be understood very easily. In Step 1, we are to initialize the path matrix P with the adjacency matrix Gptr. This signifies that initially the path matrix maintains $P[i][j]$ entries whether there is a direct path from v_i to v_j. In Step 3, the outermost loop will execute N times. For each k, it decides whether there is a path from v_i to v_j (for all i, j = 1, 2, ..., N) either directly or via k, i.e., from v_i to v_k and then from v_k to v_j. It therefore sets the $P[i][j]$ entries to 0 or 1 accordingly.

As an illustration, let us consider a graph as shown in Figure 8.26(a). From the graph it is observed that there is a path from v_1 to v_4 via v_2. During the execution of step 3 in Warshall's algorithm and when k = 2, we see that $P[1][4] = P[1][4] \vee (P[1][2] \wedge P[2][4]) = 0 \vee (1 \wedge 1)$ = 1 thus enumerating the path from v_1 to v_4 via v_2. The final path matrix is obtained as shown in Figure 8.26(b). From this, it is seen that there is no path from the vertex v_4 to any other vertex. Also, we cannot reach to vertex v_1 from any other vertex in the graph. Other possible reachabilities can also be seen from this path matrix. Here, the vertex v_1 acts as a source vertex (one can reach to any vertex from it) and vertex v_4 as a sink vertex (that is, if we reach at this vertex then it cannot be moved to any other vertices). There is no cycle in the graph as all the diagonal entries in the path matrix are zero.



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

(a) A graph and its adjacency matrix

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

After $k = 2$

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

Final P

(b) Path matrices

Fig. 8.26. Illustration of Warshall's algorithm.

Assignment 8.13 Consider a graph G as shown in Figure 8.27.

(a) Find the path matrix P of G using Warshall's algorithm.

(b) From the path matrix, obtain all the elementary paths from any vertex to other.

(c) Can you conclude about any cycle in the graph from the path matrix P as obtained by you.

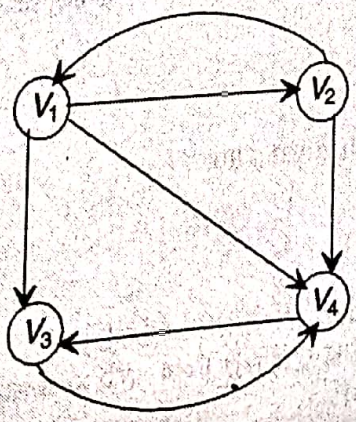


Fig. 8.27

Floyd's algorithm

It can be noted that the path matrix so obtained using Warshall's algorithm shows the presence or absence of any path between a pair of vertices; it also detects the presence or absence of a cycle at any vertex. The Warshall's algorithm does not take into account the weights of edges. If weights are to be taken into account and if we are interested in the length of the shortest path

between any pair of vertices, then another classical solution is known, given by Robert Floyd and is called the *Floyd's* algorithm. In fact, the basic structure of the Floyd's algorithm is same as Warshall's algorithm.

Let us first state the Floyd's algorithm. To describe it, we will use two functions, namely, MIN(x, y) and COMBINE(p_1, p_2). MIN(x, y) returns the minimum value between x and y, and COMBINE(p_1, p_2) returns the concatenation of two paths p_1 and p_2 resulting a single path. For example, say p_1 = 1-2-3 and p_2 = 3-4, then COMBINE(p_1, p_2) will return p = 1-2-3-4. This algorithm produces two matrices: one stores the length of shortest paths between all pair of vertices and another stores the shortest paths between all pair of vertices, if exist. Following is the algorithm in detail.

Algorithm FLOYD( )

Input: A graph G whose pointer to its weighted adjacency matrix is Gptr and vertices are labelled as 1, 2, ..., N; N being the number of vertices in the graph.

Output: Q is a matrix of order $N \times N$ containing the length of the shortest path between all pair of vertices. PATHS is a matrix of order $N \times N$ containing the string of the shortest length of paths between all pair of vertices.

Steps:

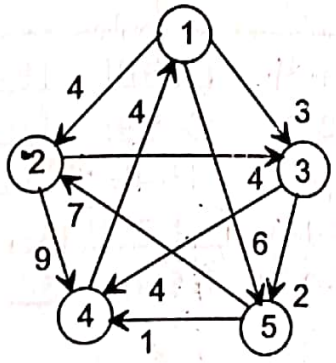
/*To initialize the matrix Q and PATHS*/

1. For i = 1 to N do
   1. For j = 1 to N do
      1. If (Gptr[i][i] = 0) then //No path between v_i and v_j
         1. $Q[i][j] = \infty$ //Initialize with large number say, infinity
         2. PATHS[i][j] = NULL //Path is empty
      2. Else
         1. $Q[i][j]$ = Gptr[i][j]
         2. P = COMBINE(i, j) //Initial path is i-j
         3. PATHS[i][j] = P //Store this path
      3. EndIf
   2. EndFor
2. EndFor

/*To compute all pairs of shortest paths*/

3. For k = 1 to N do
   1. For i = 1 to N do
      1. For j = 1 to N do
         1. $Q[i][j] = \text{MIN}(Q[i][j], Q[i][k] + Q[k][j])$
         2. If ($Q[i][k] + Q[k][j] < Q[i][j]$) then //If alternate path is found
            1. p_1 = PATHS[i][k]
            2. p_2 = PATHS[k][j]
            3. PATHS[i][j] = COMBINE (p_1, p_2) //Store the altered path
         3. EndIf
      2. EndFor
   2. EndFor
4. EndFor
5. Return (Q, PATHS)
6. Stop

One can easily notice the similarities of Warshall's algorithm and Floyd's algorithm. In the second algorithm, the statement in inner-most loop (that is, j-loop), namely, $Q[i][j] = MIN(Q[i][j], Q[i][k] + Q[k][j])$ is in fact, the modified version of $P[i][j] = P[i][j] \vee (P[i][k] \wedge P[k][j])$, which appears in Warshall's algorithm. In addition to this, there are some extra steps in Floyd's algorithm. The Floyd's algorithm has one underlying assumption that if there is no path from v_i to v_j then $Gptr[i][j] = \infty$. (Infinity is the largest possible positive value.) This is incorporated in our algorithm as stated above (see step 1). Floyd's algorithm not only computes the length of the shortest paths, it also enumerates the shortest paths between the vertices and keeps a track of it; this is accomplished as in inner-most j-loop in step 3. It is better, if we illustrate the execution of this algorithm with an example. Let us consider a weighted digraph as shown in Figure 8.28 for the purpose.



	1	2	3	4	5
1	–	4	3	–	6
2	–	–	4	9	–
3	–	–	–	4	2
4	4	–	–	–	–
5	–	7	–	1	–

Fig. 8.28 A weighted digraph and its adjacency matrix.

The algorithm when traced, the cost matrix Q and PATHS as they appear are shown below. The new/altered entries are underlined. The cost matrix Q and its corresponding path matrix are shown side by side.

After initialization of Q and PATHS,

	1	2	3	4	5
1	∞	4	3	∞	6
2	∞	∞	4	9	∞
3	∞	∞	∞	4	2
4	4	∞	∞	∞	∞
5	∞	7	∞	1	∞

	1	2	3	4	5
1	–	[1–2]	[1–3]	–	[1–5]
2	–	–	[2–3]	[2–4]	–
3	–	–	–	[3–4]	[3–5]
4	[4–1]	–	–	–	–
5	–	[5–2]	–	[5–4]	–

After $k = 1$

	1	2	3	4	5
1	∞	4	3	∞	6
2	∞	∞	4	9	∞
3	∞	∞	∞	4	2
4	4	<u>8</u>	<u>7</u>	∞	<u>10</u>
5	∞	7	∞	1	∞

	1	2	3	4	5
1	–	[1–2]	[1–3]	–	[1–5]
2	–	–	[2–3]	[2–4]	–
3	–	–	–	[3–4]	[3–5]
4	[4–1]	<u>[4–1–2]</u>	<u>[4–1–3]</u>	–	<u>[4–1–5]</u>
5	–	[5–2]	–	[5–4]	–

After $k = 2$,

	1	2	3	4	5
1	∞	4	3	13	6
2	∞	∞	4	9	∞
3	∞	∞	∞	4	2
4	4	8	7	17	10
5	∞	7	11	1	∞

	1	2	3	4	5
1	–	[1–2]	[1–3]	[1–2–4]	[1–5]
2	–	–	[2–3]	[2–4]	–
3	–	–	–	[3–4]	[3–5]
4	[4–1]	[4–1–2]	[4–1–3]	[4–1–2–4]	[4–1–5]
5	–	[5–2]	[5–2–3]	[5–4]	–

After $k = 3$,

	1	2	3	4	5
1	∞	4	3	7	5
2	∞	∞	4	8	6
3	∞	∞	∞	4	2
4	4	8	7	11	9
5	∞	7	11	1	13

	1	2	3	4	5
1	–	[1–2]	[1–3]	[1–3–4]	[1–3–5]
2	–	–	[2–3]	[2–3–4]	[2–3–5]
3	–	–	–	[3–4]	[3–5]
4	[4–1]	[4–1–2]	[4–1–3]	[4–1–3–4]	[4–1–3–5]
5	–	[5–2]	[5–2–3]	[5–4]	[5–2–3–5]

After $k = 4$,

	1	2	3	4	5
1	11	4	3	7	5
2	12	16	4	8	6
3	8	12	11	4	2
4	4	8	7	11	9
5	5	7	8	1	10

	1	2	3	4	5
1	[1–3–4–1]	[1–2]	[1–3]	[1–3–4]	[1–3–5]
2	[2–3–4–1]	[2–3–4–1–2]	[2–3]	[2–3–4]	[2–3–5]
3	[3–4–1]	[3–4–1–2]	[3–4–1–3]	[3–4]	[3–5]
4	[4–1]	[4–1–2]	[4–1–3]	[4–1–3–4]	[4–1–3–5]
5	[5–4–1]	[5–2]	[5–4–1–3]	[5–4]	[5–4–1–3–5]

After $k = 5$,

	1	2	3	4	5
1	10	4	3	6	5
2	11	13	4	7	6
3	7	9	10	3	2
4	4	8	7	10	9
5	5	7	8	1	10

	1	2	3	4	5
1	[1–3–5–4–1]	[1–2]	[1–3]	[1–3–5–4]	[1–3–5]
2	[2–3–5–4–1]	[2–3–5–2]	[2–3]	[2–3–5–4]	[2–3–5]
3	[3–5–4–1]	[3–5–2]	[3–5–4–1–3]	[3–5–4]	[3–5]
4	[4–1]	[4–1–2]	[4–1–3]	[4–1–3–5–4]	[4–1–3–5]
5	[5–4–1]	[5–2]	[5–4–1–3]	[5–4]	[5–4–1–3–5]

The outer-most loop (k-loop) in step 3 is to explore the possibility of the shortest path through vertex $k = 1, 2, ..., 5$, if found, it updates the cost and alters the path otherwise retains previous value.

For the given graph, we found that: there exist paths (shortest) between every pair of vertices, and every vertex has their cycle. (*Note:* Floyd's algorithm can also be used for non-weighted graphs. In that case, weight of edge should be taken as 1.)

Assignment 8.14 Sometime it can be seen that between a pair of vertices more than one shortest paths are there. In that case, that shortest path is favoured which contains minimum number of edges. Modify the Floyd's algorithm to impose this selection criteria.

Dijkstra's algorithm

Another kind of the shortest path problem is the *single source* shortest path problem. In this problem, there is a distinct vertex, called source vertex and it requires to find the shortest path from this source vertex to all other vertices. As an example, let us consider a simple graph as shown in Figure 8.29. The different shortest paths, assuming v_1 as the source vertex, are listed as below:

	Shortest path	*Length of the shortest path*
From v_1 to v_2	v_1–v_2	1
From v_1 to v_3	v_1–v_2–v_3	3
From v_1 to v_4	v_1–v_2–v_4	4
From v_1 to v_5	v_1–v_2–v_3–v_5	5

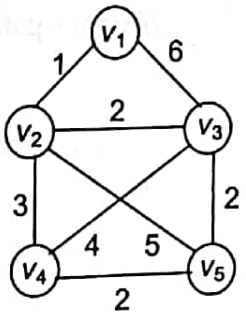


Fig. 8.29 A simple graph representing shortest path.

The algorithm to find such paths was first proposed by E.W. Dijkstra and popularly known as *Dijkstra's* algorithm. In this section, we will describe this algorithm in a simplest way and then will talk about its potential applications.

Assume that all the vertices in the graph are labelled as 1, 2, ..., N and the graph is represented through an adjacency matrix. Dijkstra's algorithm requires three arrays as below:

LENGTH[1 ... N] = array of distances

PATH[1 ... N] = array of vertices

SET[1 ... N] = array of Boolean tags

In conclusion, the shortest distance from the source vertex to any vertex say, i(i = 1, 2, ..., N) is stored in LENGTH[i], while PATH[i] contains the nearest predecessor(s) of the vertex i on the path determining the shortest distance from source vertex to the vertex i. In other words, the array PATH keeps a track of shortest path from any vertex to the source vertex. The Boolean array SET is used during the execution of the algorithm. SET[i] = 1 means the shortest distance, and path from the source vertex to the vertex i is already enumerated.

Suppose the source vertex be S, the algorithm consists of two major parts: an initialization part followed by an iteration part. In initialization part, we are to initialize the above mentioned three arrays. Once the initialization is over, we are to enter into iteration part, where the vertices will be included one by one in the set of vertices for which the shortest distance from S is enumerated. The detail steps of the algorithm is stated as below:

Algorithm DIJKSTRA(S)

Input: Gptr, the pointer to the graph. S, the source vertex. Let N be the number of vertices.

Output: LENGTH, an array of distances from S to all other vertices. PATH, an array of string of vertices giving the track of all shortest paths.

Data structure: Matrix representation of graph with Gptr as the pointer to it.

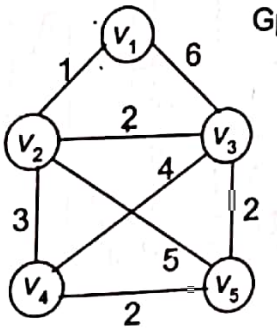
Steps:

```
   /*INITIALIZATION*/
1. For i = 1 to N do
   1. SET[i] = 0                          //Initialization of SET array
2. EndFor
3. For i = 1 to N do
   1. If Gptr[S][i] = 0 then              //There is no direct path from S to vertex i
      1. LENGTH[i] = ∞
      2. PATH[i] = NULL                   //Empty path
   2. Else
      1. LENGTH[i] = Gptr[S][i]
      2. PATH[i] = S                      //Source vertex is the immediate predecessor to
                                          //vertex i
   3. EndIf
4. EndFor
5. SET[S] = 1
6. LENGTH[S] = 0                          //Source vertex is implicitly enumerated with
                                          //length as 0
   /*ITERATION*/
7. complete = FALSE                       //It is a flag for controlling the iteration
8. While (not complete) do
   1. j = SEARCH_MIN(LENGTH, SET)         //Find a vertex j which has minimum distance among
                                          //those vertices yet not enumerated for the shortest path
   2. SET[j] = 1                          //Vertex j is enumerated
   3. For i = 1 to N do                   //For each i not yet enumerated
      1. If SET[i] = 1, then              //If i is enumerated already
         1. i = i + 1                     //then go to next vertex
      2. Else                             //If i is connected to j by an edge
         1. If Gptr[i][j] ≠ 0 then
            1. If ((LENGTH[j] + GPTR[i][j]) < LENGTH[i]) then
               1. LENGTH[i] = LENGTH[j] + GPTR[i][j]
               2. PATH[i] = j             //Vertex j becomes the immediate predecessor of i
            2. EndIf
         2. EndIf
      3. EndIf
```

```
4. EndFor
   /*To test whether all vertices are enumerated or not*/
5. complete = TRUE
6. For i = 1 to N do
   1. If SET[i] = 0 then
      1. complete = FALSE
      2. Break                          //Break the loop
   2. Else
      1. i = i + 1
   3. EndIf
7. EndFor
9. EndWhile
10. Return (LENGTH, PATH)
11. Stop
```

this algorithm, we have assumed a procedure SEARCH_MIN (*see* in step 8.1). This procedure ll return the label of the vertex which has minimum distance (by consulting the array LENGTH) d which is not yet enumerated in the shortest path (by consulting the array SET). The actual de of this procedure is very simple and is left as an exercise.

To understand Dijkstra's algorithm, it is better to trace it with a graph. Let us consider the aph as shown in Figure 8.30.



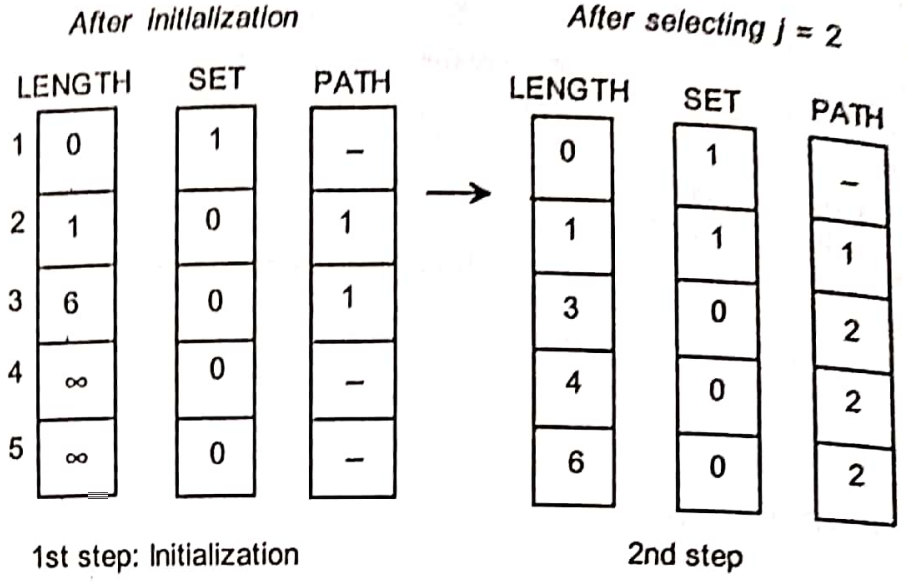
Gptr →

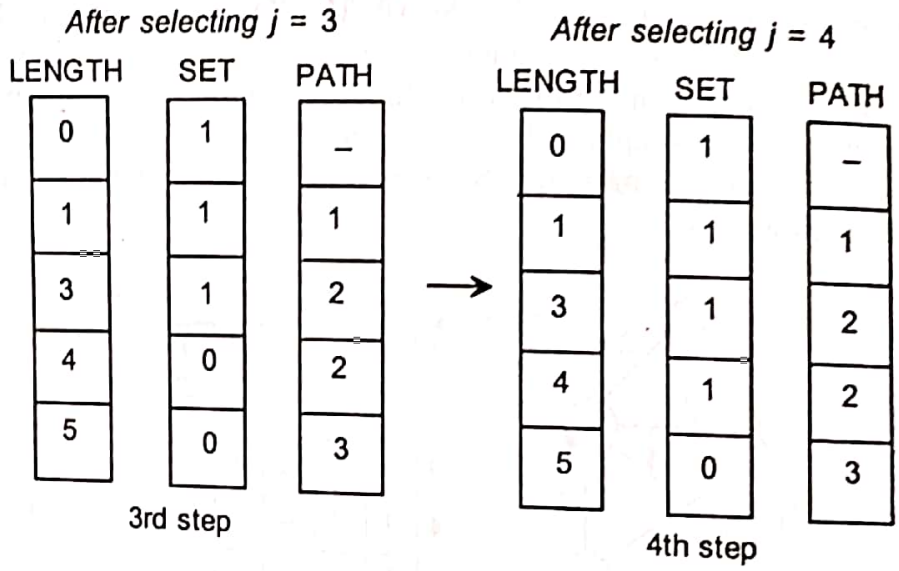
1	0	1	6	0	0
2	1	0	2	3	5
3	6	2	0	4	2
4	0	3	4	0	2
5	0	5	2	2	0

Fig. 8.30 · A graph for Dijkstra's algorithm.

The result of the application of Dijkstra's algorithm on the graph (Figure 8.30) is illustrated shown in Figure 8.31. The entries in the arrays LENGTH and PATH, during the execution various steps of Dijkstra's algorithms, are illustrated step-by-step. Following information can concluded from these two arrays:

Path		*Length*	*Shortest path*
From (Source vertex)	*To*		
1	2	1	1–2
	3	3	1–2–3
	4	4	1–2–4
	5	5	1–2–3–5





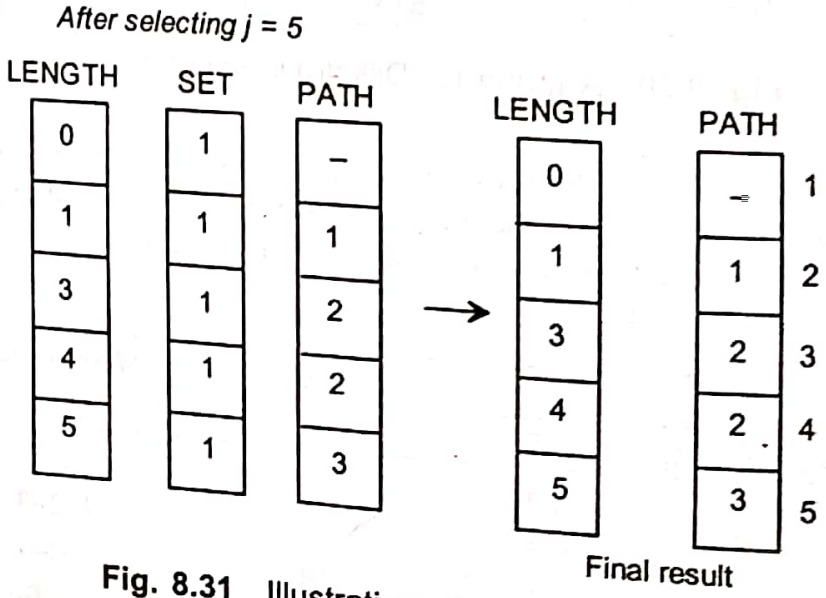


Fig. 8.31 Illustration of Dijkstra's algorithm.