

Algorithm:

Input: A connected, weighted, undirected graph $G = (V, E)$ with weight function $w(u, v)$,

Output: A minimum spanning tree (MST).

1. Initialize:

- Select an arbitrary vertex v_0 as the starting node.
- Set $T = \{v_0\}$ (initial MST with one vertex).
- Initialize a priority queue PQ with edges connected to v_0 , prioritized by weight.

2. Repeat until all vertices are included in T (i.e., $|T| = |V|$):

- Extract the edge (u, v) with the minimum weight from PQ.

- If v is not already in T :
 - Add v to T .
 - Add all edges of v (not already in T) to PQ.

3. Output the edges of the minimum spanning tree.

Time Complexity Analysis:

• Using an Adjacency matrix & simple search: $O(V^2)$

Date:	Exp. Title Implement Prim's algorithm and Analyze its time complexity.	Page No.
Exp. No. 05. a.		11

Program:

```
import java.util.Scanner;
public class primAlgorithm1 {
    static final int INF = 9999;
    static final int MAX = 20;
    static int[][] G = new int[MAX][MAX];
    static int[][] spanning = new int[MAX][MAX];
    static int n;

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the number of vertices: ");
        n = scanner.nextInt();
        System.out.println("Enter the adjacency: ");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                G[i][j] = scanner.nextInt();
            }
        }
        int totalcost = Prims();
        System.out.println("In Spanning tree matrix: ");
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                System.out.print(spanning[i][j] + " ");
            }
        }
    }

    int Prims() {
        int cost = 0;
        int parent[] = new int[n];
        int visited[] = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            visited[i] = 0;
        }
        parent[0] = -1;
        visited[0] = 1;
        for (int i = 0; i < n - 1; i++) {
            int min = INF;
            int u = -1;
            for (int j = 0; j < n; j++) {
                if (visited[j] == 0) {
                    for (int k = 0; k < n; k++) {
                        if (G[j][k] != 0 && G[j][k] < min) {
                            min = G[j][k];
                            u = k;
                        }
                    }
                }
            }
            if (u == -1) {
                break;
            }
            cost += min;
            visited[u] = 1;
            for (int k = 0; k < n; k++) {
                if (G[u][k] != 0 && G[u][k] < INF) {
                    parent[k] = u;
                    G[u][k] = INF;
                }
            }
        }
        return cost;
    }
}
```



Date :	Exp. Title	Page No.
Exp. No.		
	System.out.println();	
	}	
	System.out.println("In Total cost of the spanning tree = " + totalCost);	
	}	
	System static int prims(){	
	int [][] cost = new int [MAX][MAX];	
	int [] distance = new int [MAX];	
	int [] from = new int [MAX];	
	int [] visited = new int [MAX];	
	int mincost = 0;	
	for (int i=0; i<n; i++)	
	{	
	for (int j=0; j<n; j++)	
	{	
	if (e[i][j]==0)	
	{	
	cost[i][j]=INF;	
	}	
	else {	
	cost[i][j]=e[i][j];	
	}	
	spanning[i][j]=0;	
	}	
	}	
	distance[0]=0;	
	visited[0]=1;	
	for (int i=1; i<n; i++)	
	{	
	distance[i]=cost[0][i];	
	from[i]=0;	

1) Find the minimum spanning tree
 using the given weighted graph.
 (Number of vertices = 3)
 (Number of edges = 3)

2) Enter the number of vertices : 3
 Enter the adjacency matrix:
 0 1 1
 1 0 1
 1 1 0

Spanning tree matrix:
 0 1 1
 1 0 0
 1 0 0

Total cost of the spanning tree = 2.6

Output.

Enter the number of vertices : 3
 Enter the adjacency matrix:
 0 1 1
 1 0 1
 1 1 0

Spanning tree matrix:

0 1 1
 1 0 0
 1 0 0

Total cost of the spanning tree = 2.6

Date :	Exp. Title	Page No.
Exp. No.		13
	<pre> visited[0] = 0; int noOfEdges = n-1; // n = no of vertices where (noOfEdges > 0) int minDistance = INF, v = -1; for (int i=1; i<n; i++) if (visited[i]==0 && distance[i] < minDistance) minDistance = distance[i]; v = i; minDistance = distance[v]; </pre>	
	<pre> int u = from[v]; spanning[v][v] = distance[v]; spanning[v][u] = distance[v]; noOfEdges--; if (visited[v] == INF) for (int i=1; i<n; i++) if (visited[i]==0 && cost[i][v] < distance[i]) distance[i] = cost[u][v]; from[i] = v; mincost += cost[u][v]; return mincost; } </pre>	



Algorithm :

Input : A weighted, directed graph $G = (V, E)$ with non-negative edge weight $w(u, v)$, and a source vertex s .

Output : The shortest path distances from s to all vertices in G .

1. Initialize :
 - Set the distance to the source : $d(s) = 0$
 - Set all other distances : $d(v) = \infty$ for $v \neq s$.
 - Create a priority queue (PQ) & insert $(s, 0)$ (source with distance 0).

2. While the priority queue is not empty:
 - Extract the vertex u with the minimum distance from PQ.
 - For each neighbor v of u :
 - If $d(u) + w(u, v) < d(v)$:
 - update $d(v)$ to $d(u) + w(u, v)$.
 - Insert/update $(v, d(v))$ in PQ.

3. Output : The shortest path distances from s to all other vertices.

Time Complexity Analysis.

- Using an adjacency matrix with simple search : $O(V^2)$

Date :	Exp. Title Dijkstra's algorithm and Analyze its Time complexity.	Page No.
Exp. No. 05 b.		14
Program :		
Public class DijkstraAlgorithm		{
Public void dijkstraAlgorithm (int[] graph, int		source)
{		
int nodes = graph.length;		
boolean[] visited_vertex = new boolean[nodes];		
int[] dist = new int[nodes];		
for (int i=0; i<nodes; i++)		{
visited_vertex[i] = false;		
dist[i] = Integer.MAX_VALUE;		}
dist[source] = 0;		
for (int i=0; i<nodes; i++)		{
int u = find_min_distance (dist, visited_vertex);		
visited_vertex[u] = true;		
for (int v=0; v<nodes; v++)		{
if (!visited_vertex[v] && graph[u][v] != 0 && dist[u] + graph[u][v] < dist[v])		
dist[v] = dist[u] + graph[u][v];		}
}		
}		



1. Implementing Dijkstra's algorithm
 [Implementation of Dijkstra's algorithm]

2. Implementing Bellman-Ford algorithm
 [Implementation of Bellman-Ford algorithm]

3. Implementing Floyd-Warshall algorithm
 [Implementation of Floyd-Warshall algorithm]

4. Implementing Depth-First Search (DFS)
 [Implementation of Depth-First Search]

5. Implementing Breadth-First Search (BFS)
 [Implementation of Breadth-First Search]

6. Implementing Dijkstra's algorithm using priority queue
 [Implementation of Dijkstra's algorithm using priority queue]

7. Implementing Bellman-Ford algorithm using priority queue
 [Implementation of Bellman-Ford algorithm using priority queue]

8. Implementing Floyd-Warshall algorithm using priority queue
 [Implementation of Floyd-Warshall algorithm using priority queue]

Date:	Exp. Title	Page No.
Exp. No.		15
	System.out.println("String. format ("distance from vertex i.S to vertex j.S" is l.S, source, i, dist[i]));	
	}	
	}	
	Private static int find_min_distance(int[] dist, boolean[] visited_vertex)	
	{	
	int minimum_distance = Integer.MAX_VALUE;	
	int minimum_distance_vertex = -1;	
	for (int i=0; i< dist.length; i++)	
	{	
	if (!visited_vertex[i] && dist[i] < minimum -distance)	
	{	
	minimum_distance = dist[i];	
	minimum_distance_vertex = i;	
	}	
	}	
	return minimum_distance - vertex;	
	}	
	Public static void main(String[] args) {	
	{	
	int graph[][] = new int[4][4];	
	graph[0][0] = 0;	
	{0, 1, 1, 2, 0, 0, 0},	
	{0, 0, 2, 0, 0, 3, 0},	
	{1, 2, 0, 1, 3, 0, 0},	
	{2, 0, 1, 0, 2, 0, 1},	
	{0, 0, 3, 0, 0, 2, 0},	
	{0, 3, 0, 0, 1, 2, 0},	
	{0, 2, 0, 1, 0, 1, 0},	

Date :

Exp. Title

Exp. No.

3;

```
DijkstraAlgorithm test = new DijkstraAlgorithm();
    test.dijkstraAlgorithm(graph, 0);
```

3
or

```
graph = new Graph(10);
    graph.addEdge(0, 1, 1);
    graph.addEdge(0, 2, 1);
    graph.addEdge(1, 2, 1);
    graph.addEdge(1, 3, 1);
    graph.addEdge(2, 3, 1);
    graph.addEdge(2, 4, 1);
    graph.addEdge(3, 4, 1);
    graph.addEdge(3, 5, 1);
    graph.addEdge(4, 5, 1);
    graph.addEdge(4, 6, 1);
    graph.addEdge(5, 6, 1);
    graph.addEdge(5, 7, 1);
    graph.addEdge(6, 7, 1);
    graph.addEdge(6, 8, 1);
    graph.addEdge(7, 8, 1);
    graph.addEdge(7, 9, 1);
    graph.addEdge(8, 9, 1);
```

```
graph.setDistance(0, 0, 0);
    graph.setDistance(1, 0, 1);
    graph.setDistance(2, 0, 1);
    graph.setDistance(3, 0, 2);
    graph.setDistance(4, 0, 3);
    graph.setDistance(5, 0, 4);
    graph.setDistance(6, 0, 5);
    graph.setDistance(7, 0, 6);
    graph.setDistance(8, 0, 7);
    graph.setDistance(9, 0, 8);
```

```
graph.setDistance(1, 1, 0);
    graph.setDistance(2, 1, 1);
    graph.setDistance(3, 1, 2);
    graph.setDistance(4, 1, 3);
    graph.setDistance(5, 1, 4);
    graph.setDistance(6, 1, 5);
    graph.setDistance(7, 1, 6);
    graph.setDistance(8, 1, 7);
    graph.setDistance(9, 1, 8);
```

```
graph.setDistance(2, 2, 0);
    graph.setDistance(3, 2, 1);
    graph.setDistance(4, 2, 2);
    graph.setDistance(5, 2, 3);
    graph.setDistance(6, 2, 4);
    graph.setDistance(7, 2, 5);
    graph.setDistance(8, 2, 6);
    graph.setDistance(9, 2, 7);
```

```
graph.setDistance(3, 3, 0);
    graph.setDistance(4, 3, 1);
    graph.setDistance(5, 3, 2);
    graph.setDistance(6, 3, 3);
    graph.setDistance(7, 3, 4);
    graph.setDistance(8, 3, 5);
    graph.setDistance(9, 3, 6);
```

```
graph.setDistance(4, 4, 0);
    graph.setDistance(5, 4, 1);
    graph.setDistance(6, 4, 2);
    graph.setDistance(7, 4, 3);
    graph.setDistance(8, 4, 4);
    graph.setDistance(9, 4, 5);
```

```
graph.setDistance(5, 5, 0);
    graph.setDistance(6, 5, 1);
    graph.setDistance(7, 5, 2);
    graph.setDistance(8, 5, 3);
    graph.setDistance(9, 5, 4);
```

```
graph.setDistance(6, 6, 0);
    graph.setDistance(7, 6, 1);
    graph.setDistance(8, 6, 2);
    graph.setDistance(9, 6, 3);
```

```
graph.setDistance(7, 7, 0);
    graph.setDistance(8, 7, 1);
    graph.setDistance(9, 7, 2);
```

```
graph.setDistance(8, 8, 0);
    graph.setDistance(9, 8, 1);
```

```
graph.setDistance(9, 9, 0);
```

Output:

Distance from vertex 0 to vertex 0 is 0
 Distance from vertex 0 to vertex 1 is 1
 Distance from vertex 0 to vertex 2 is 1
 Distance from vertex 0 to vertex 3 is 2
 Distance from vertex 0 to vertex 4 is 3
 Distance from vertex 0 to vertex 5 is 4
 Distance from vertex 0 to vertex 6 is 5
 Distance from vertex 0 to vertex 7 is 6
 Distance from vertex 0 to vertex 8 is 7
 Distance from vertex 0 to vertex 9 is 8



Algorithm : Warshall's Algorithm.

Input: A directed graph represented as an adjacency matrix $R(0)$, where i and j are vertices.

- $R(0)[i][j] = 1$ if there is a direct edge from i to j , else 0.

Output: The transitive closure matrix $R(n)$, where $R(n)[i][j] = 1$ if there exists a path from i to j , otherwise 0.

Steps :

1. Initialize $R(0)$ as the adjacency matrix
2. Iterate over all vertices k as intermediate nodes:
 - For each pair of vertices (i, j) :
 - Update the matrix:
$$R(k)[i][j] = R(k-1)[i][j] \text{ OR } (R(k-1)[i][k] \text{ AND } R(k-1)[k][j])$$
3. Final matrix $R(n)$ gives the transitive close of the graph.

Time Complexity of Warshall's Algorithm.

- The algorithm runs three nested loops over V vertices $O(V^3)$.

Date :

Exp. No. 06.aeb.

Exp. Title Implement Warshall and
Floyd's algorithm.

Page No. 17

Program :

```
import java.lang.*;
```

```
public class AllPairsShortestPath {
```

```
final static int INF = 99999, V = 4;
```

```
void floydwarshall (int dist[][])
```

```
{ int i, j, k;
```

```
for (k=0; k<V; k++)
```

```
{
```

```
for (i=0; i<V; i++)
```

```
{
```

```
for (j=0; j<V; j++)
```

```
{
```

```
if (dist[i][k] + dist[k][j] < dist
```

```
[i][j])
```

```
dist[i][j] =
```

```
dist[i][k] + dist[k][j];
```

```
}
```

```
}
```

```
printSolution (dist);
```

```
void printSolution (int dist[][])
```

```
{
```

```
System.out.println ("The following matrix shows
```

```
the shortest distance between every pair of vertices");
```

```
for (int i=0; i<V; i++)
```

```
{
```

```
for (int j=0; j<V; j++)
```

```
{
```

```
if (dist[i][j] == INF)
```

```
System.out.print ("INF");
```



Algorithm : Floyd - Warshall Algorithm.

Input: A weighted graph represented as a distance matrix $D(0)$, where :

- $D(0)[i][j]$ is the weight of edge from i to j (or ∞ if no direct edge exists).

$$\forall i \forall j \forall k \quad D(0)[i][j] = 0 \text{ for all } i \neq j$$

(i, j are adjacent nodes)

Output: A matrix $D(n)$ where $D(n)[i][j]$ gives the shortest distance between i and j .

Steps:

1. Initialize $D(0)$ (as the weight matrix).

2. Iterate over all vertices k as intermediate nodes:

- for each pair (i, j) :

• update the matrix using :

$$D(k)[i][j] = \min(D(k-1)[i][j], D(k-1)[i][k] + D(k-1)[k][j])$$

(i, j are adjacent nodes)

3. final matrix $D(n)$ gives shortest paths between all pairs.

Time Complexity of Floyd-Warshall Algorithm

• It has three nested loops, iterating over all pairs of vertices $\rightarrow O(V^3)$.

Output: The following matrix shows the shortest distance b/w every pair of vertices.

0	5	8	9	3
INF	0	3	4	7
INF	INF	0	1	6
INF	INF	INF	0	5

Date :

Exp. Title

Page No.

Exp. No.

18.

else

~~Wrote a program to find shortest path between two nodes in a graph using Dijkstra's algorithm.~~

3. ~~Program to find shortest path between two nodes in a graph using Dijkstra's algorithm.~~

System.out.println();

{}

3. ~~Program to find shortest path between two nodes in a graph using Dijkstra's algorithm.~~

public static void main (String [] args) {

{

int graph [][] = {{0, 5, INF, 0},
{INF, 0, 3, INF},
{INF, INF, 0, 1},
{INF, INF, INF, 0}};

AllpairshortestPath a = new AllpairshortestPath();

V

3.

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~

~~graph shows all shortest path between all pairs of vertices.~~



Algorithm :- Backtracking approach for Hamiltonian cycle.

This algorithm checks whether a Hamiltonian cycle exists in a given undirected graph using backtracking.

Input :

- A graph $G = (V, E)$ represented by an adjacency matrix $adj[][]$.
- A starting vertex V_0 .

Output :

- A Hamiltonian cycle (if one exists) or a message indicating that no cycle exists.

Steps:

1. Initialize an array $Path[]$ to store the cycle, setting $Path[0] = V_0$.
2. Recursively try to construct the cycle by adding one vertex at a time:
 - Choose the next vertex V such that:
 - It is adjacent to the previous vertex in $Path[]$.
 - It has not been visited before.
 - If a valid vertex is found, mark it as visited and move to the next position.
3. If all vertices are in $Path[]$ and the last vertex connects back to V_0 , return success.
4. Backtrack if no valid vertex can be found at a position.
5. If all possibilities are exhausted, return that no Hamiltonian cycle exists.

Date:

Exp. No. OT

Exp. Title Implement Hamiltonian cycle
- S algorithm.

Page No. 19

Program.

Public class HamiltonianCycle

{ Constructor for graph and other variables.

int Path[];

for (int i=0; i<pos; i++) { }

if (Path[i] == N) { }

return false;

return true;

}

boolean hamcycleutil (int graph [][][], int Path[],

int pos)

{ }

if (pos == v)

{ }

if (graph [Path [pos-1]] [Path [0]] == 1)

return true;

else

return false;

{ }

for (int v=1; v<V; v++)

{ }

if (issafe (v, graph, Path, pos))

{ }

Path [pos] = v;

if (hamcycleutil (graph, Path, pos+1) == true)

return true;

Path [pos] = -1;

{ }

Time Complexity Analysis.

- The algorithm explores all possible permutations of vertex to form a cycle.
- Since each vertex can be chosen in at most $V!$ ways, the worst-case time complexity is $O(V!)$ (Factorial Time complexity).
- In practice, pruning techniques (backtracking and branch-and-bound) reduce the number of cases checked, but the problem remains exponential in the worst case.

Date :

Exp. No.

Exp. Title

Page No.

20.

{

return false;

{

int hamcycle (int graph[][])

{

Path = new int[V];

for (int i=0; i<V; i++)

path[i] = -1;

path[0] = 0;

if (hamcycleutil(graph, Path, 1) == false)

{

System.out.println ("In solution does not

exist");

return 0;

{

printSolution(Path);

return 1;

{

void printSolution (int Path[]) {

{

System.out.println ("Solution exists: following" +

"is one Hamiltonian cycle");

for (int i=0; i<V; i++)

System.out.print (" " + Path[i] + " ");

System.out.println (" " + Path[0] + " ");

{

public static void main (String args[]) {

{

hamiltonianCycle hamiltonian = new hamiltonianCycle();

}

Q8

Write the Hamiltonian circuit if exists.

Graph 1: 0 1 2 3 0

(0-1) edge (1-2) edge (2-3) edge (3-0)

{(0,1), (1,2), (2,3)}

(0-1) edge (1-2) edge (2-3) edge (3-0)

{(0,1), (1,2), (2,3)}

0-1-2-3-0

(0-1) edge (1-2) edge (2-3) edge (3-0)

{(0,1), (1,2), (2,3)}

0-1-2-3-0

(0-1) edge (1-2) edge (2-3) edge (3-0)

{(0,1), (1,2), (2,3)}

0-1-2-3-0

(0-1) edge (1-2) edge (2-3) edge (3-0)

{(0,1), (1,2), (2,3)}

0-1-2-3-0

Output:

+ Solution exists. 0-1-2-3-0

Solution exists. following is one Hamiltonian cycle

0 1 2 3 0

(0-1) edge (1-2) edge (2-3) edge (3-0)

for Graph 2

Solution does not exist.

Final output state is an invalid spanning tree.

Q9



Scanned with OKEN Scanner

Date :
Exp. No.

Exp. Title

Page No.
21

- int graph1[][] = {{0, 1, 0, 1, 0}, {1, 0, 1, 1, 1}, {0, 1, 0, 0, 1}, {1, 1, 0, 0, 1}, {0, 1, 1, 1, 0}};

(int) System.out.println("Graph 1 : ");

for (int i = 0; i < 5; i++) {

{0, 1, 0, 0, 1},

fixed = (int){0, 1, 1, 1, 0},

(int) System.out.println(fixed);

hamiltonian.hamcycle(graph1);

System.out.println("For Graph 2 : ");

- int graph2[][] = {{0, 1, 0, 1, 0}, {1, 0, 1, 1, 1}, {0, 1, 0, 0, 1}, {1, 1, 0, 0, 1}, {0, 1, 1, 1, 0}};

(int) System.out.println("Graph 2 : ");

for (int i = 0; i < 5; i++) {

{1, 1, 0, 0, 1},

{0, 1, 1, 0, 1},

(int) System.out.println(fixed);

hamiltonian.hamcycle(graph2);

3
3

((int) System.out.println(fixed));

3
3

((int) System.out.println(fixed));

(((int) System.out.println(fixed))))



Scanned with OKEN Scanner

Algorithm:

- Steps:
1. Compute $\text{GCD}(a,b)$ using Euclidean Algorithm.
 2. Compute $\text{LCM}(a,b)$ using the formula:

$$\text{LCM}(a,b) = \frac{|axb|}{\text{GCD}(a,b)}$$

Time Complexity Analysis:

- GCD calculation (Euclidean Algorithm):
 - Runs in $O(\log \min(a,b))$ time.
- LCM calculation:
 - Multiplication takes $O(1)$.
 - Division takes $O(1)$.
- Overall time complexity:
 $O(\log \min(a,b))$.

Output:

LCM of the array is : 300.

Date :

Exp. No. 08.

Exp. Title

Implement lcm algorithm.

Page No.

22.

Program.

Public class LCMCalculator

{

Private static int gcd (int a, int b)

{

If (b == 0)

return a;

return gcd (b, a % b);

}

Private static int lcm (int a, int b)

{

return (a * b) / gcd (a, b);

}

Public static int lcmArray (int[] arr)

int result = arr[0];

for (int i = 1; i < arr.length; i++)

{

result = lcm (result, arr[i]);

}

return result;

}

Public static void main (String[] args)

{

int[] numbers = {12, 15, 20, 25};

int result = lcmArray (numbers);

System.out.println ("lcm of the array is : " + result);

}

}.