Output :

The shortest path from node :
0 to 0 is 0
0 to 1 is 8
0 to 2 is 6
0 to 3 is 5
0 to 4 is 3

EXPT. NO. | NAME | Page No.: 1 | Date: 2/8/2023 | YOUVA

M T W T F S S

EXPERIMENT - 1

DIJKSTRA'S ALGORITHM

Code :

```
import java.util.*;
public class Dij {
  private int dist[];
  private Set<Integer> settled;
  private PriorityQueue<Node> pq;
  private int V;
  List<List<Node>> adj;
  public Dij(int V){
  this.V=V;
   dist = new int[V];
   settled = new HashSet<Integer>();
   pq=new PriorityQueue<Node>(V, new Node());
  } //A utility function to print the constructed dist array
  public void dijkstra(List<List<Node>> adj,int src){
   this.adj = adj;
   for(int i=0; i<V; i++) dist[i] = Integer.MAX_VALUE;
   pq.add(new Node(src,0));
   dist[src]= 0; //Find shortest path for all vertices. Distance from
   while(settled.size()!= V){ //source to itself is always 0
    if(pq.isEmpty()) return;
    int u = pq.remove.node;
    if(settled.contains(u)) continue;
```

Teacher's Signature: _____

```
        settled.add(u);
        e_Neighbours(u);
      }// Update dist[v] only if it is not set, there is an edge from u to v
    } // and the total weight is smaller than the current value
    private void e_Neighbours(int u){
      int edgeDistance = -1
      int newDistance = -1
      for(int i=0; i<adj.get(u).size(); i++){
        Node v = adj.get(u).get(i);
        if(!settled.contains(v.node)){
          edgeDistance = v.cost
          newDistance = dist[u]+edgeDistance;
          if(newDistance<dist[v.node]) dist[v.node]=newDistance;
          pq.add(new Node(v.node, dist[v.node]));
        }
      }
    }
    public static void main(String arg[]){
      int V=5;
      int source=0;
      List<List<Node>>adj=new ArrayList<List<Node>>();
      for(int i=0; i<V; i++)
        List<Node>item=new ArrayList<Node>();
        adj.add(item);
      } //Define edges and their costs in the adjacency list
      adj.get(0).add(new Node(1,9));
      adj.get(0).add(new Node(2,6));
```

Teacher's Signature:

```
ad
adj.get(0).add(new Node(3,5));
adj.get(0).add(new Node(4,3));
adj.get(2).add(new Node(1,2));
adj.get(2).add(new Node(3,4));
Dij dpq = new Dij(V); // Create a Dij object and run Dijkstra's
dpq.dijkstra(adj, source); // algorithm
System.out.println("The shortest path from node :");
for(int i=0; i<dpq.dist.length; i++) System.out.println(
    source+" to "+i+" is "+dpq.dist[i]);
}
}
// Comparator to compare nodes based on their cost
class Node implement Comparator<Node>{
public int node;
public int cost;
public Node(){}
public Node(int node, int cost){
this.node = node;
this.cost = cost;
} // Node class to represent vertices and costs
@Override
public int compare(Node node1, Node node2){
if(node1.cost < node2.cost) return -1;
if(node1.cost > node2.cost) return 1;
return 0;
}
}
```

Teacher's Signature:

Output:

Path: [Shimla, Delhi, Indore, Chennai, Kanyakumari]

EXPT. NO. | NAME | Page No.: 4 | Date: 9/8/2023 | YOUVA

EXPERIMENT - 2

A* ALGORITHM

Code:

```
import java.util.*;

public class Astar{
    public static void main (String[] args){
// Create nodes for various cities with their heuristic distances (h scores)
    Node n1 = new Node ("Shimla", 366);
    Node n2 = new Node ("Ahmedabad", 374);
    Node n3 = new Node ("Pune", 380);
    Node n4 = new Node ("Dehhi", 253);
    Node n5 = new Node ("Kolkata", 178);
    Node n6 = new Node ("Indore", 193);
    Node n7 = new Node ("Chennai", 98);
    Node n8 = new Node ("Kanpur", 329);
    Node n9 = new Node ("Shillong", 244);
    Node n10 = new Node ("Ranchi", 241);
    Node n11 = new Node ("Vishakhapatnam", 242);
    Node n12 = new Node ("Goa", 160);
    Node n13 = new Node ("Kanyakumari", 0);
    Node n14 = new Node ("Port Blair", 77);
// Setup the adjacency relations b/w cities using Edge objects
    n1.adjacencies = new Edge[]{new Edge(n2,75), new Edge(n4,140), new
        Edge(n8,118)};
```

Teacher's Signature:

```
n2.adjacencies = new Edge[] {new Edge(n1,75), new Edge(n3,71)};
n3.adjacencies = new Edge[] {new Edge(n2,71), new Edge(n4,151)};
n4.adjacencies = new Edge[] {new Edge(n1,140), new Edge(n5,99),
    new Edge(n3,151), new Edge(n6,80)};
n5.adjacencies = new Edge[] {new Edge(n4,99), new Edge(n13,211)};
n6.adjacencies = new Edge[] {new Edge(n4,80), new Edge(n7,97),
    new Edge(n12,146)};
n7.adjacencies = new Edge[] {new Edge(n6,97), new Edge(n13,101),
    new Edge(n12,138)};
n8.adjacencies = new Edge[] {new Edge(n1,118), new Edge(n9,111)};
n9.adjacencies = new Edge[] {new Edge(n8,111), new Edge(n10,70)};
n10.adjacencies = new Edge[] {new Edge(n9,70), new Edge(n11,75)};
n11.adjacencies = new Edge[] {new Edge(n10,75), new Edge(n12,120)};
n12.adjacencies = new Edge[] {new Edge(n11,120), new Edge(n7,138), new
    Edge(n7,138)};
n13.adjacencies = new Edge[] {new Edge(n7,101), new Edge(n14,90), new
    Edge(n5,211)};
n14.adjacencies = new Edge[] {new Edge(n13,90)};
AstarSearch(n1,n13); // Perform A* Search from n1 to n13
List<Node> path = printPath(n13); // Print path from source to
System.out.println("Path:" + path); // goal city
}
public static List<Node> printPath(Node target){ // Traceback from the
List<Node> path = new ArrayList<Node>(); // goal to source city
for(Node node = target; node != null; node = node.parent){ // to find
    path.add(node);                                        // path
}
```

Teacher's Signature:

```
Collections.reverse(path);
return path;
} // Create a set to keep track of explored nodes & a priority queue
public static void AstarSearch(Node source, Node goal) {
  Set<Node> explored = new HashSet<Node>();
  PriorityQueue<Node> queue = new PriorityQueue<Node>(20,
  new Comparator<Node>() {
  public int compare(Node i, Node j){
// Define a comparator to compare nodes based on f_scores
      if (i.f_scores > j.f_scores) return 1;
      else if (i.f_scores < j.f_scores) return -1;
      else return 0;
    }
  }
  );
  source.g_scores = 0; // Initialize the g_score for node
  queue.add(source); // Add the source to the queue
  boolean found = false;
  while((!queue.isEmpty()) && (!found)){
    Node current = queue.poll();
    explored.add(current);

    if(current.value.equals(goal.value)) found = true;
// Check if the goal node has been reached
    for(Edge e: current.adjacencies){ // Iterate through the
      Node child = e.target; // adjacent nodes of the current
      double cost = e.cost; // node
```

```
double temp_g_scores = current.g_scores + cost;
double temp_f_scores = temp_g_scores + child.h_scores;
if(explored.contains(child)&&(temp_f_scores>=child.f_scores))
{} // If child node already explored and new f score higher, skip iteration
else if((!queue.contains(child))||(temp_f_scores<child.f_scores
)){ // Update child node's parents and scores
child.parent = current;
child.g_scores = temp_g_scores;
child.f_scores = temp_f_scores;
if(queue.contains(child))queue.remove(child);
queue.add(child); // Remove and add the child node
} // again to update its position in the queue
{
}
}
class Node {
public final String value;
public double g_scores;
public final double h_scores;
public double f_scores = 0;
public Edge[] adjacencies;
public Node parent;
public Node(String val, double hVal){
value = val;
h_scores = hVal;
}
public String toString(){ return value;}
```

```
}
class Edge {
  public final double cost;
  public final Node target;
  public Edge(Node targetNode, double costVal){
    target = targetNode;
    cost = costVal;
  }
}
```

Output (DFS):

Following is Depth First Traversal (starting from vertex 2)
2 0 1 3

Output (BFS):

Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

EXPT. NO. | NAME | Page No.: 9 | Date: 16/8/2023 | YOUVA

M T W T F S S

EXPERIMENT - 3

DFS AND BFS

(a) DEPTH FIRST SEARCH (DFS)

Code :

```
import java.io.*;
import java.util.*;
//This graph represents a directed graph using adjacency list
class GraphD{
 private V;
 private LinkedList<Integer> adj[];
 @SupressWarnings("unchecked")
 GraphD(int v){
   V=v
   adj = new LinkedList[v];
   for(int i=0; i<v; ++i) adj[i] = new LinkedList();
 } //Function to add edge in graph
 void addEdge(int v, int w){
   adj[v].add(w);   //Add w to v's list
 }
 void DFSUtil(int v, boolean visited[]){ //Function used for
   visited[v] = true;                    //DFS traversal using
   System.out.println(v+" ");            //recursion
   Iterator<Integer> i = adj[v].listIterator();
   while(i.hasNext()){
```

Teacher's Signature:

```
        int n = i.next();
        if (!visited[n]) DFSUtil(n, visited);
    }
}
void DFS(int v){
    boolean visited[] = new boolean[V]; //Initially all vertices
    DFSUtil(v, visited);                //are not visited
}
public static void main(String args[]){
    GraphD g = new GraphD(4);
    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(1,2);
    g.addEdge(2,0);
    g.addEdge(2,3);
    g.addEdge(3,3);
    System.out.println("Following is Depth First Traversal"
      +"(starting from vertex 2)");
    g.DFS(2); // Function call for DFS
  }
}
```

Teacher's Signature:

(b) BREADTH FIRST SEARCH (BFS)

Code:

```
import java.io.*;
import java.util.*;
// Directed graph made using adjacency list
class GraphB {
  private V;
  private LinkedList<Integer> adj[]; // Adjacency list
  @SuppressWarnings("unchecked")
  GraphB(int v) {
    V = v;
    adj = new LinkedList[v]
    for(int i=0; i<V; ++i) adj[i] = new LinkedList();
  }
  void addEdge(int v, int w) { // Function to add edge to
    adj[v].add(w); // graph
  }
  void BFS(int s) { // Prints BFS traversal from given source s
    boolean visited[] = new boolean[V]; // Initially all vertices
    LinkedList<Integer> queue = new LinkedList<Integer>(); // marked
    visited[s] = true;                                     // as not visited
    queue.add(s);
    while(queue.size() != 0) {
      s = queue.poll(); // Dequeue vertex from queue and
      System.out.print(s+" "); // print it
```

```
        Iterator<Integer> i = adj[s].listIterator();
        while(i.hasNext()){
            int n = i.next();
            if(!visited[n]){
                visited[n] = true;
                queue.add(n);
            }
        }
    }
} // Driver Code
public static void main(String args[]) {
    GraphB g = new GraphB(4);
    g.addEdge(0,1);
    g.addEdge(0,2);
    g.addEdge(1,2);
    g.addEdge(2,0);
    g.addEdge(2,3);
    g.addEdge(3,3);
    System.out.println("Following is Breadth First
    Traversal "+"(starting from vertex 2)");
    g.BFS(2);
}
}
```

Teacher's Signature:

Output :

Coloring of Graph 1

Vertex 0 ---> 0
Vertex 1 ---> 1
Vertex 2 ---> 2
Vertex 3 ---> 0
Vertex 4 ---> 1

Coloring of Graph 2

Vertex 0 ---> 0
Vertex 1 ---> 1
Vertex 2 ---> 2
Vertex 3 ---> 0
Vertex 4 ---> 3

EXPT. NO. | NAME | Page No: 13 | Date: 23/8/2023 | YOUVA

EXPERIMENT - 4

GRAPH COLOURING PROBLEM

Code:

```
import java.io.*;
import java.util.*;
//This class represents an undirected graph using adjacency list
class Graph {
  private int V;
  private LinkedList<Integer> adj[];
  Graph(int v) {
    V = v;
    adj = new LinkedList[v];
    for(int i=0; i<v; ++i) adj[i] = new LinkedList();
  } //Function to add an edge into the graph
  void addEdge(int v, int w) {
    adj[v].add(w);
    adj[w].add(v);
  }
  void Coloring() {
    int result[] = new int[V]; //Initialize all vertices as unassigned
    Arrays.fill(result, -1); //Assign first color to first vertex
    result[0] = 0;
//A temporary array to store the available colours. False value
//of available[cr] would mean that colour cr is assigned to one
//of its adjacent vertices
```

```
boolean available[] = new boolean[V]; // Initially all colours
Arrays.fill(available, true); // available
for(int u = 1; u < V; u++){ // Assign colours to remaining V-1
    Iterator<Integer> it = adj[u].iterator(); // vertices
    while(it.hasNext()){ // Process all adjacent vertices and
        int i = it.next(); // flag their colours as unavailable
        if(result[i] != -1) available[result[i]] = false;
    }
    int cr; // Find first available colour
    for(cr=0; cr<V; cr++) if(available[cr]) break;
    result[u] = cr; // Assign found colour
    Arrays.fill(available, true); // Reset values back to
} // true for next iteration
for(int u=0; u<V; u++) System.out.println("Vertex "
    +u+ "---> Color " + result[u]);
}
public static void main(String args[]) {
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(0, 2);
    g1.addEdge(1, 2);
    g1.addEdge(1, 3);
    g1.addEdge(2, 3);
    g1.addEdge(3, 4);
    System.out.println("Colouring of Graph 1")
    g1.Coloring();
    Graph g2 = new Graph(5);
```

Teacher's Signature:

```
g2.addEdge(0,1);
g2.addEdge(0,2);
g2.addEdge(1,2);
g2.addEdge(1,4);
g2.addEdge(2,4);
g2.addEdge(3,4);
System.out.println("Coloring of Graph 2");
g2.Coloring();
}
}
```

Teacher's Signature:

Output :

```
1 2 3
5 6 0
7 8 4

1 2 3
5 0 6
7 8 4

1 2 3
5 8 6
7 0 4

1 2 3
5 8 6
0 7 4
```

EXPT. NO.	NAME

8 P

Cod

```
import
import

class
    publ
    publ
        M
        u
        u
        u
        u
    }
// Funct
    publi
    for
        f
        S
    }
}
// Functio
    public
    int
```

EXPT. NO. | NAME | Page No: 16 | Date: 30/8/2023 | YOUVA

EXPERIMENT - 5

8 PUZZLE PROBLEM

Code:

```
import java.io.*;
import java.util.*;

class EightPuz {
  public static int N=3;
  public static class Node {
    Node parent; //Stores parent node of current node. Helps trace path
    int mat[][]=new int[N][N]; //Stores puzzle matrix
    int x,y; //Stores blank tile coordinates
    int cost; //Stores no. of misplaced tiles
    int level; //Stores no. of moves so far
  }
//Function to print NXN matrix
  public static void printMatrix(int mat[][]){
    for(int i=0;i<N;i++){
      for(int j=0;j<N;j++)System.out.print(mat[i][j]+" ");
      System.out.println("");
    }
  }
//Function to allocate new Node
  public static Node newNode(int mat[][],int x,int y,
    int newX,int newY,int level,Node parent){
```

Teacher's Signature:

```
int temp=
Node node = new Node();
node.parent = parent; //Set pointer to the root
node.mat = new int[N][N]; int temp=node.mat[x][y];
for(int i=0; i<N; i++) for(int j=0; j<N; j++) node.mat[i][j]=mat[i][j]
node.mat[x][y] = node.mat[newX][newY];
node.mat[newX][newY] = temp; //Move tile by 1 position
node.cost = Integer.MAX_VALUE; //Set the no. of misplaced tiles
node.level = level;
node.x = newX; //Update new blank tile coordinates
node.y = newY;
return node;
}
public static int row[] = {1,0,-1,0}; //bottom, left, top, right
public static int col[] = {0,-1,0,1};
//Function to calculate no. of misplaced tiles
public static int calculateCost(int initialMat[][], int
finalMat[][]){
int count=0;
for(int i=0; i<N; i++) for(int j=0; j<N; j++)
if(initialMat[i][j]!=0 && initialMat[i][j]!=finalMat[i][j])
count++;
return count;
}
//Function to check if (x,y) is a valid matrix coordinate
public static int isSafe(int x, int y){
return (x>=0 && x<N && y>=0 && y<N)?1:0;
}
```

Teacher's Signature:

```
public static void printPath(Node root){ // Print the path
    if((root) == null){return;} // from the root node to
    printPath(root.parent); // the Destination Node
    printMatrix(root.mat);
    System.out.println("");
}
// Comparison object to be used to order the heap
public static class comp implements Comparator<Node>{
    @Override
    public int compare(Node lhs, Node rhs) {
        return (lhs.cost+lhs.level)>(rhs.cost+rhs.level)?1:-1;
    }
}

public static void solve(int initialMat[][], int x, int y, int finalMat[][])
{
    PriorityQueue<Node> pq = new PriorityQueue<>(new comp());
    Node root = newNode(initialMat, x, y, x, y, 0, null);
    root.cost = calculateCost(initialMat, finalMat);
    pq.add(root);
    while(!pq.isEmpty()){
        Node min = pq.peek();
        pq.poll();
        if(min.cost == 0){
            printPath(min); // Print path from the root to
            return;         // destination
        }
```

```
for(int i=0; i<4; i++){ //Do for each child of min
    if(isSafe(min.x+row[i], min.y+col[i])>0){
        Node child = newNode(min.mat, min.x, min.y,
        min.x+row[i], min.y+col[i], min.level+1, min);
        child.cost = calculateCost(child.mat, finalMat);
        pq.add(child);
    }
}
}
}
// Driver Code
public static void main (String[] args){
    int initialMat[][] = {{1,2,3},{5,6,0},{7,8,4}};
    int finalMat[][] = {{1,2,3},{5,8,6},{0,7,4}};
    int x=1, y=2;
    solve(initialMat, x, y, finalMat);
}
}
```

Mark

Teacher's Signature:

Output :

The optimal value is : 5

EXPERIMENT - 6

ALPHA-BETA PRUNING

Code :

```
import java.io.*;
class Alpbet {
  // Initial values of Alpha and Beta
  static int MAX = 1000;
  static int MIN = -1000;

  // Returns optimal value for current player (Initially called
  // for root and maximizer
  static int minimax(int depth, int nodeIndex, Boolean maximizingPlayer,
    int values[], int alpha, int beta) {
    // terminating condition, i.e., leaf node is reached
    if (depth == 3) return values[nodeIndex];

    if (maximizingPlayer) {
      int best = MIN;

      // Recur for left and right children
      for (int i = 0; i < 2; i++) {
        int val = minimax(depth + 1, nodeIndex * 2 + i, false, values,
                          alpha, beta);
        best = Math.max(best, val);
        alpha = Math.max(alpha, best);
```

Teacher's Signature:

```
            // Alpha-Beta pruning
            if (beta <= alpha) break;
        }
        return best;
    }
}

// Driver Code
public static void main(String[] args){
    int values[] = {3,5,6,9,1,2,0,-1};
    System.out.println("The optimal value is: " +
        minimax(0,0,true,values,MIN,MAX));
}
}
```

Teacher's Signature:

Output :

```
The value of the best Move is : 10
The Optimal Move is :
Row : 2 Col : 2
```

EXPERIMENT - 7

Tic Tac Toe

Code :

```
import java.io.*;

// Next optimal move for a player
class TicTacToe {
  static class Move {
    int row, col;
  }
  static char player = 'x', opponent = 'o';

  // This function returns true if there are moves remaining
  // on the board. It returns false if there are no moves
  // left to play.
  static Boolean isMovesLeft(char board[][]) {
    for (int i = 0; i < 3; i++) for (int j = 0; j < 3; j++) {
      if (board[i][j] == '_') return true;
      return false;
    }
  }

  // This is the evaluation function
  static int evaluate(char b[][]) {
```

```
// Checking for Rows for X or O victory
for (int row = 0; row < 3; row ++){
  if (b[row][0] == b[row][1] && b[row][1] == b[row][2]){
    if (b[row][0] == player) return +10;
    else if (b[row][0] == opponent) return -10;
  }
}
// Checking for Columns for X or O victory
for (int col = 0; col < 3; col ++){
  if (b[0][col] == b[1][col] && b[1][col] == b[2][col]){
    if (b[0][col] == player) return +10;
    else if (b[0][col] == opponent) return -10;
  }
}
// Checking for Diagonals for X or O victory
if (b[0][0] == b[1][1] && b[1][1] == b[2][2]){
  if (b[0][0] == player) return +10;
  else if (b[0][0] == opponent) return -10;
}
if (b[0][2] == b[1][1] && b[1][1] == b[2][0]){
  if (b[0][2] == player) return +10;
  else if (b[0][2] == opponent) return -10;
}
// Else if none of them have won then return 0
return 0;
}
```

```
// This is the minimax function. It considers all the possible
// ways the game can go and returns the value of the board
static int minimax(char board[][], int depth, boolean isMax){
    int score = evaluate(board);
    // If maximizer has won the game, return his/her
    // evaluated score
    if (score == 10) return score;
    // If the minimizer has won the game, return his/her
    // evaluated score
    if (score == -10) return score;
    // If there are no more moves and no winner then it
    // is a tie
    if (isMovesLeft(board) == false) return 0;

    // If this is maximizer's move
    if (isMax) {
        int best = -1000;
        // Traverse all cells
        for (int i=0; i<3; i++){
            for (int j=0; j<3; j++){
                // Check if cell is empty
                if (board[i][j] == '_') {
                    // Make the move
                    board[i][j] = player;
                    // Call minimax recursively and choose the
                    // maximum value
```

```
                best = Math.max(best, minimax(board, depth+1, !isMax));
                // Undo the move
                board[i][j] = '_';
            }
        }
    }
    return best;
}

// If this is minimizer's move
else {
    int best = 1000;
    // Traverse all cells
    for (int i=0; i<3; i++) {
        for (int j=0; j<3; j++) {
            // Check if cell is empty
            if (board[i][j]=='_') {
                // Make the move
                board[i][j] = opponent;
                // Call minimax recursively and choose the
                // minimum value
                best = Math.min(best, minimax(board, depth+1, !isMax));
                // Undo the move
                board[i][j] = '_';
            }
        }
    }
```

Teacher's Signature:

```
        }
        return best;
    }
}

// This will return the best possible move for the player
static Move findBestMove(char board[][]){
    int bestVal = -1000;
    Move bestMove = new Move();
    bestMove.row = -1;
    bestMove.col = -1;
    // Traverse all cells, evaluate minimax function for all
    // empty cells. And return the cell with optimal value.
    for(int i=0; i<3; i++){
        for(int j=0; j<3; j++){
        // Check if cell is empty
        if (board[i][j]=='_'){
            // Make the move
            board[i][j] = player;
            // Compute evaluation function for this move
            int moveVal = minimax(board, 0, false);
            // Undo the move
            board[i][j] = '_';
            // If the value of the current move is more than the
            // best value, then update best
            if(moveVal > bestVal){
```

```
                bestMove.row = i;
                bestMove.col = j;
                bestVal = moveVal;
            }
        }
    }
}

System.out.printf("The value of the best Move" +
    " is : %d\n", bestVal);
return bestMove;
}

// Driver Code
public static void main(String[] args){
    char board[][] = {
        {'x', 'o', 'x'},
        {'o', 'o', 'x'},
        {' ', ' ', ' '},
    };
    Move bestMove = findBestMove(board);
    System.out.printf("The Optimal Move is :\n");
    System.out.printf("Row: %d Col: %d\n", bestMove.row,
                                    bestMove.col);
}
}
```

Teacher's Signature:

Output :

Minimum Solution Coordinates : (1.0000000049504859, 2.0000000055564827)
Minimum function value : 4.5242883329266356 E - 17

EXPERIMENT - 8

HILL CLIMBING ALGORITHM

Code :

```
import java.util.function.BiFunction;

public class HillC {
    public static double findMinimum(BiFunction<Double, Double,
                                     Double> f) {
        // Initialize the current solution and its function value
        double curX = 0;
        double curY = 0;
        double curF = f.apply(curX, curY);

        // Start with a large step size
        for (double step = 1e6; step > 1e-7;) {
            double bestF = curF;
            double bestX = curX;
            double bestY = curY;
            boolean find = false;

            // Explore neighboring solutions in a hexagonal pattern
            for (int i = 0; i < 6; i++) {
                double a = 2 * Math.PI * i / 6;
                double nextX = curX + step * Math.cos(a);
```

```
double nextY = curY + step * Math.sin(a);
double nextF = f.apply(nextX, nextY);

// If a better solution is found, update the best values
if (bestF > nextF) {
    bestF = nextF;
    bestX = nextX;
    bestY = nextY;
    find = true;
  }
}

// If no better solution is found, decrease the step size
if (!find) {
  step /= 2;
} else {
  // Update the current solution with the best values
  curX = bestX;
  curY = bestY;
  curF = bestF;
  }
}
// Print the final minimum solution coordinates
System.out.println("Minimum solution coordinates: (" + curX + ", " + curY + ")");
// Return the minimum function value
return curF;
```

Teacher's Signature:

```
}

//Driver Code
public static void main(String[] args) {
    //Define the function to minimize ((x-2)^2+(y-3)^2)
    BiFunction<Double, Double, Double> function = (x,y) -> Math.pow(x-2,2)
                                                  + Math.pow(y-3,2);
    //Find the minimum of the defined function using the hill
    //climbing algorithm
    double minimumValue = findMinimum(function);
    //Print the minimum function value
    System.out.println("Minimum function value: "+minimumValue);
  }
}
```

Teacher's Signature:

Output :

Minimum cost for the path will be : 80

EXPERIMENT - 9

TRAVELLING SALESMAN PROBLEM

Code :

```
import java.io.*;

class TravSal {
  // Function to find the minimum weight Hamiltonian Cycle
  static int tsp(int[][] graph, boolean[] v, int currPos, int n,
      int count, int cost, int ans) {
  // If last node is reached and it has a link to the
  // starting node, i.e., the source then keep the minimum value
  // out of the total cost of traversal and "ans"
  // Finally return to check for more possible values
    if (count == n && graph[currPos][0] > 0) {
      ans = Math.min(ans, cost + graph[currPos][0]);
      return ans;
    }
  // Backtracking Step
  // Loop to traverse the adjacency list of currPos node
  // and increasing the count by 1 and cost by graph[currPos,i]
  // value
    for (int i = 0; i < n; i++) {
      if (v[i] == false && graph[currPos][i] > 0) {
```

Teacher's Signature:

```
            // Mark as visited
            v[i] = true;
            ans = tsp(graph, v, i, n, count+1, cost+graph[curr][i], ans);
            // Mark ith node as unvisited
            v[i] = false;
        }
    }
    return ans;
}

// Driver Code
public static void main (String[] args) {
    // n is the number of nodes, i.e., V
    int n = 4;
    int graph[][] = {
        {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0},
    };
    // Boolean array to check if a node has been visited or not
    boolean[] v = new boolean[n];
    // Mark 0th node as visited
    v[0] = true;
    int ans = Integer.MAX_VALUE;
```

```
// Find the minimum weight Hamiltonian Cycle
ans = tsp(graph, v, 0, n, 1, 0, ans);
// ans is the minimum weight Hamiltonian Cycle
System.out.println("Minimum Cost for the path will be:"
                   + ans);
    }
}
```

Teacher's Signature:

NAME

Page No.: 34

YOUVA

Date:

EXPERIMENT -10

Aim: To build, edit and run inferences on ontologies using Protégé.

Teacher's Signature: _____

Output →

1. studies(jack, What).

   What = english

2. professor(Lewis, Students)

   Students = jane

3. professor(Harry, Thomas)

   False

EXPERIMENT - 11

Aim: Convert Prolog predicates to Semantic Net and run queries.

Program:

*Facts →

```
% Facts

studies(jane, maths).
studies(jack, english).
studies(Thomas, Science).
studies(Anderson, French).

teaches(Harry, French).
teaches(Olivia, Humanities).
teaches(Brown, english).
teaches(Lewis, maths).
```

*Rules →

```
% Rules
professor(X, Y) :- teaches(X, C), studies(Y, C).
```

Teacher's Signature:

* Queries →

% Queries

? - studies (jack, what). % Query to know jack studies what.

? - professor (Lewis, Students). % Query to know who are the students
% of professor Lewis

? - professor (Harry, Thomas). % Query to check whether Harry is
% the professor of Thomas.