

# ARTIFICIAL INTELLIGENCE

## PRACTICAL FILE



NAME: Tarun Adhikari

ROLL NUMBER: 2020UCB6057

BATCH: CSDA

SUBMITTED TO: Prakash Rao Ragiri

## INDEX

S No.	Experiment	Page	Signature
1	Dijkstra Algorithm	3-7	
2	A* Algorithm	8-13	
3	Depth and Breadth First Search	14-18	
4	Constraint Satisfaction Problem-Graph Coloring	19-22	
5	Alpha – Beta Pruning	23-25	
6	Tic-Tac-Toe	26-30	
7	Hill Climbing Algorithm	31-32	
8	8 Puzzle Problem	33-37	
9	Travelling Salesman Problem	38-39	
10	Convert the following Prolong predicates to Sematic Net	40-41	

# 1.Dijkstra Algorithm

*Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a SPT (shortest path tree) with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.*

Source: <https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7>

## 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());

    }
```

```
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;
```

```
    while (settled.size() != V) {
```

```
        if (pq.isEmpty())
```

```
            return;
```

```
        int u = pq.remove().node;
```

```
        if (settled.contains(u))
```

```
            continue;
```

```
        settled.add(u);
```

```
        e_Neighbours(u);
```

```
    }
```

```
}
```

```
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);

```

```

    }

```

```

    adj.get(0).add(new Node(1, 9));

```

```

    adj.get(0).add(new Node(2, 6));

```

```

    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);
dpq.dijkstra(adj, source);

System.out.println("The shorted path from node :");
for (int i = 0; i < dpq.dist.length; i++)

    System.out.println(source + " to " + i + " is "
                        + dpq.dist[i]);
}
}

class Node implements Comparator<Node> {

    public int node;

    public int cost;

    public Node() {}

    public Node(int node, int cost)

    {

        this.node = node;

        this.cost = cost;

    }

    @Override public int compare(Node node1, Node node2)

    {

        if (node1.cost < node2.cost)

```

```
        return -1;

    if (node1.cost > node2.cost)

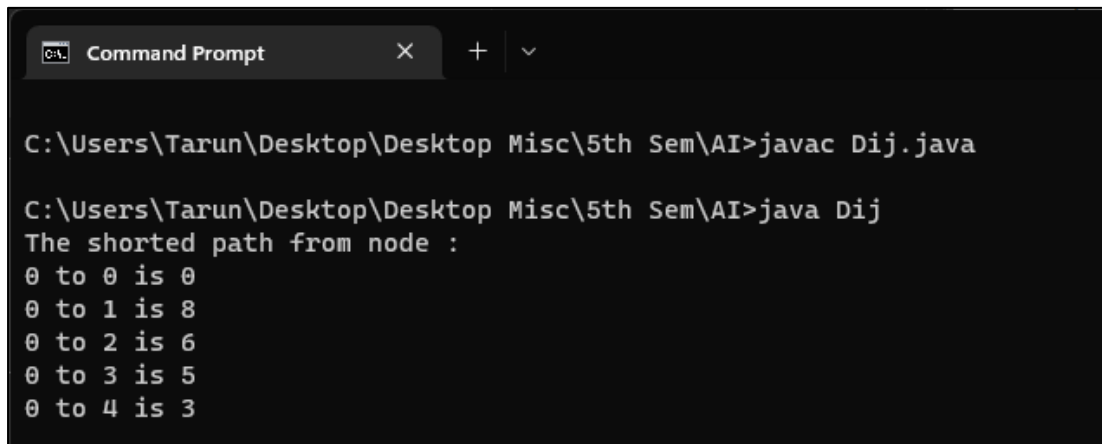
        return 1;

    return 0;

    }

}
```

## OUTPUT:



```
Command Prompt

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac Dij.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java Dij
The shorted 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
```

## 2. A\* Algorithm

*A\* (pronounced as "A star") is a computer algorithm that is widely used in pathfinding and graph traversal. The algorithm efficiently plots a walkable path between multiple nodes, or points, on the graph. However, the A\* algorithm introduces a heuristic into a regular graph-searching algorithm, essentially planning ahead at each step so a more optimal decision is made. With A\*, a robot would instead find a path in a way similar to the diagram on the right below.*

*Source: <https://brilliant.org/wiki/a-star-search>*

### CODE:

```
import java.util.PriorityQueue;

import java.util.HashSet;

import java.util.Set;

import java.util.List;

import java.util.Comparator;

import java.util.ArrayList;

import java.util.Collections;

public class Astar

{

    public static void main(String[] args)

    {

        Node n1 = new Node("Shimla", 366);

        Node n2 = new Node("Ahmedabad", 374);

        Node n3 = new Node("Pune", 380);

        Node n4 = new Node("Delhi", 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);

n1.adjacencies = new Edge[]{
    new Edge(n2, 75), new Edge(n4, 140), new Edge(n8, 118)
};
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(n6, 146), 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);

List<Node> path = printPath(n13);

System.out.println("Path: " + path);
}

```

```

    public static List<Node> printPath(Node target)
    {
        List<Node> path = new ArrayList<Node>();

        for (Node node = target; node != null; node = node.parent) {
            path.add(node);
        }
    }

```

```

    Collections.reverse(path);

    return path;
}

    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) {
            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;
    queue.add(source);
    boolean found = false;
    while ((!queue.isEmpty()) && (!found)) {
        Node current = queue.poll();
        explored.add(current);
        if (current.value.equals(goal.value)) {
            found = true;
        }
    }

```

```

//check every child of current node
for (Edge e : current.adjacencies) {
    Node child = e.target;

    double cost = e.cost;

    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)) {
    } else if ((!queue.contains(child)) ||
        (temp_f_scores < child.f_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);
    }
}
}
}
}

```

```

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;
}

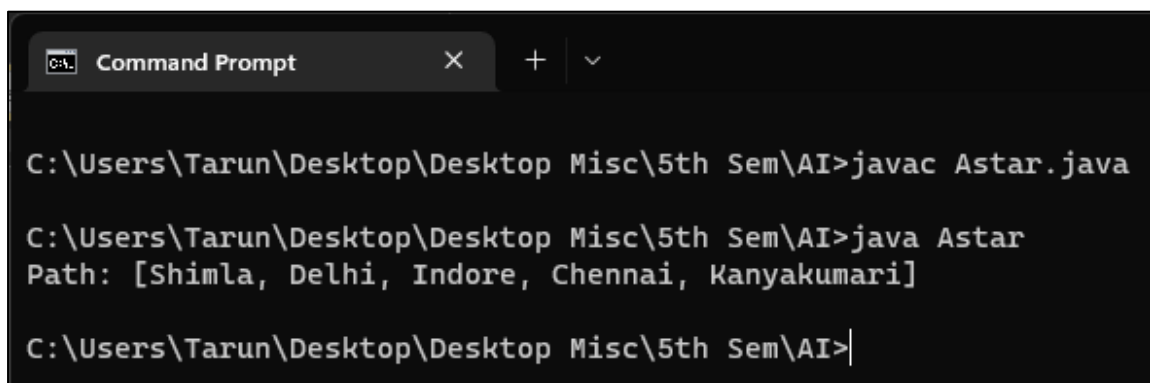
}

class Edge {
    public final double cost;
    public final Node target;

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

```

## OUTPUT:



```

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac Astar.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java Astar
Path: [Shimla, Delhi, Indore, Chennai, Kanyakumari]

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>

```

### 3.Depth First and Breadth First Search

#### a. Depth First Search(DFS)

*Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.*

*Source: <https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph>*

#### CODE:

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

class GraphD {
    private int V;
    private LinkedList<Integer> adj[];
    @SuppressWarnings("unchecked") GraphD(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)
    {
        adj[v].add(w);
    }

    void DFSUtil(int v, boolean visited[])
```

```

{
    visited[v] = true;
    System.out.print(v + " ");
    Iterator<Integer> i = adj[v].listIterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n])
            DFSUtil(n, visited);
    }
}

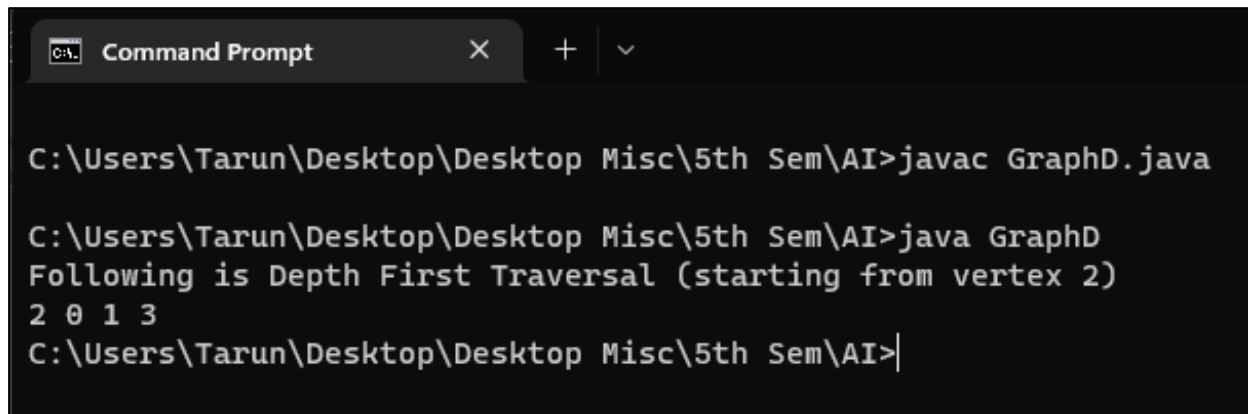
void DFS(int v)
{
    boolean visited[] = new boolean[V];
    DFSUtil(v, visited);
}

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);
}
}

```

## OUTPUT:



```
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac GraphD.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java GraphD
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>|
```

### b. Breadth First Search(BFS)

*BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layer wise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.*

*Source: <https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial>*

## CODE:

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

class GraphB
{
    private int V;
    private LinkedList<Integer> adj[];

    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)
    {
        adj[v].add(w);
    }

    void BFS(int s)
    {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<Integer>();

        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            s = queue.poll();
            System.out.print(s+" ");
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}

```

```

    }

    public static void main(String args[])
    {
        GraphB g = new GraphB(4);

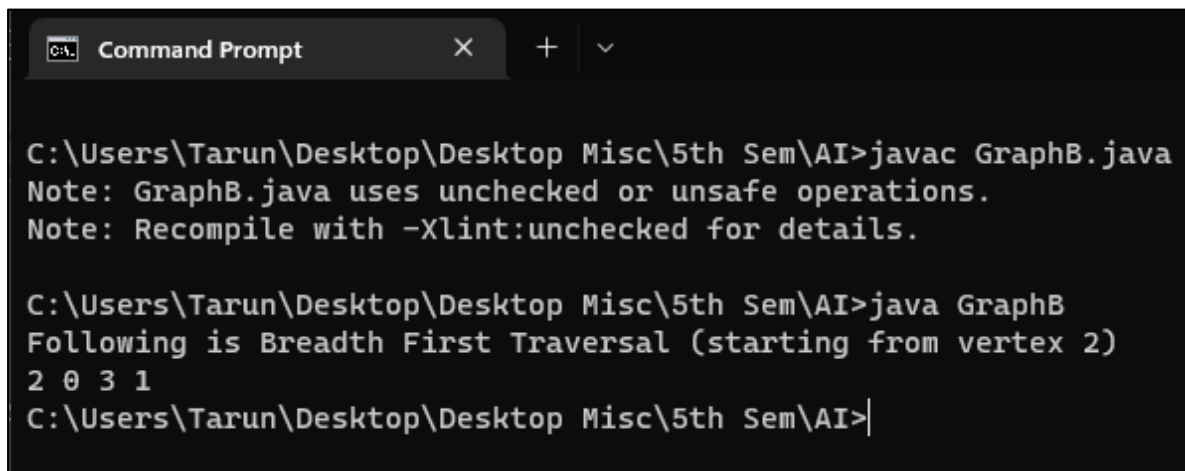
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println("Following is Breadth First Traversal "+"(starting from vertex 2)");

        g.BFS(2);
    }
}

```

## OUTPUT:



```

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac GraphB.java
Note: GraphB.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java GraphB
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>

```

## 4.Constraint Satisfaction Problem – Graph Coloring

*Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such that no two adjacent vertexes have the same color is called Graph Colouring. This is also known as vertex coloring.*

*Source: <https://www.interviewbit.com/blog/graph-coloring-problem>*

### CODE:

```
import java.io.*;

import java.util.*;

import java.util.LinkedList;

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();

    }

}
```

```

void addEdge(int v,int w)

{

    adj[v].add(w);

    adj[w].add(v); //Graph is undirected

}


void Coloring()

{

    int result[] = new int[V];

    Arrays.fill(result, -1);

    result[0] = 0;


    boolean available[] = new boolean[V];

    Arrays.fill(available, true);

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

    {

        Iterator<Integer> it = adj[u].iterator() ;

        while (it.hasNext())

        {

            int i = it.next();

            if (result[i] != -1)

                available[result[i]] = false;

        }

        int cr;

```

```

        for (cr = 0; cr < V; cr++){

            if (available[cr])

                break;

        }

        result[u] = cr;

        Arrays.fill(available, true);

    }

    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("Coloring of graph 1");

    g1.Coloring();

}

}

```

## OUTPUT:

```
Command Prompt
Microsoft Windows [Version 10.0.22621.819]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Tarun>cd C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac Graph.java
Note: Graph.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java Graph
Coloring of graph 1
Vertex 0 ---> Color 0
Vertex 1 ---> Color 1
Vertex 2 ---> Color 2
Vertex 3 ---> Color 0
Vertex 4 ---> Color 1

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>
```

## 5. Alpha – Beta Pruning

*Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Connect 4, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.*

Source: [https://en.wikipedia.org/wiki/Alpha-beta\\_pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning)

### CODE:

```
import java.io.*;

class AlpBet {

    static int MAX = 1000;
    static int MIN = -1000;

    static int minimax(int depth, int nodeIndex, Boolean maximizingPlayer, int values[], int alpha, int beta)
    {
        if (depth == 3)
            return values[nodeIndex];
        if (maximizingPlayer)
        {
            int best = MIN;
            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);
        if (beta <= alpha)
            break;
    }
    return best;
}
else
{
    int best = MAX;
    for (int i = 0; i < 2; i++)
    {

        int val = minimax(depth + 1, nodeIndex * 2 + i, true, values, alpha, beta);
        best = Math.min(best, val);
        beta = Math.min(beta, best);
        if (beta <= alpha)
            break;
    }
    return best;
}
}

public static void main (String[] args)
{

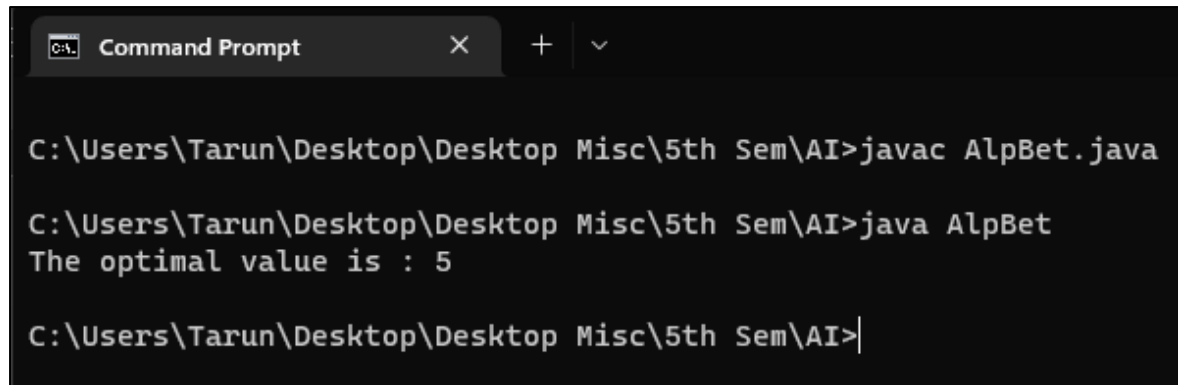
    int values[] = {3, 5, 6, 9, 1, 2, 0, -1};
    System.out.println("The optimal value is : " + minimax(0, 0, true, values, MIN, MAX));

}
}

```



## OUTPUT:



```
Command Prompt
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac AlpBet.java
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java AlpBet
The optimal value is : 5
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>|
```

## 6.Tic Tac Toe

*Tic-tac-toe (American English), noughts and crosses (Commonwealth English), or Xs and Os (Canadian or Irish English) is a paper-and-pencil game for two players who take turns marking the spaces in a three-by-three grid with X or O. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game, with a forced draw assuming best play from both players.*

Source: <https://en.wikipedia.org/wiki/Tic-tac-toe>

### CODE:

```
public class TicTacToe {  
    static class Move {  
        int row, col;  
    }  
  
    static char player = 'x', opponent = 'o';  
  
    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;  
    }  
  
    static int evaluate(char[][] b) {  
        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;
    }
}

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;
    }
}

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;
}

return 0;
}

```

```

static int minimax(char[][] board, int depth, Boolean isMax) {
    int score = evaluate(board);
    if (score == 10)
        return score;

    if (score == -10)
        return score;

    if (!isMovesLeft(board))
        return 0;

    if (isMax) {
        int best = -1000;

        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = player;
                    best = Math.max(best, minimax(board, depth + 1, false));
                    board[i][j] = '_';
                }
            }
        }
        return best;
    } else {
        int best = 1000;
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i][j] == '_') {
                    board[i][j] = opponent;

```

```

        best = Math.min(best, minimax(board, depth + 1, true));
        board[i][j] = '_';
    }
}
}
return best;
}
}

```

```

static Move findBestMove(char[][] board) {
    int bestVal = -1000;
    Move bestMove = new Move();
    bestMove.row = -1;
    bestMove.col = -1;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[i][j] == '_') {
                board[i][j] = player;
                int moveVal = minimax(board, 0, false);
                board[i][j] = '_';
                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;
    }

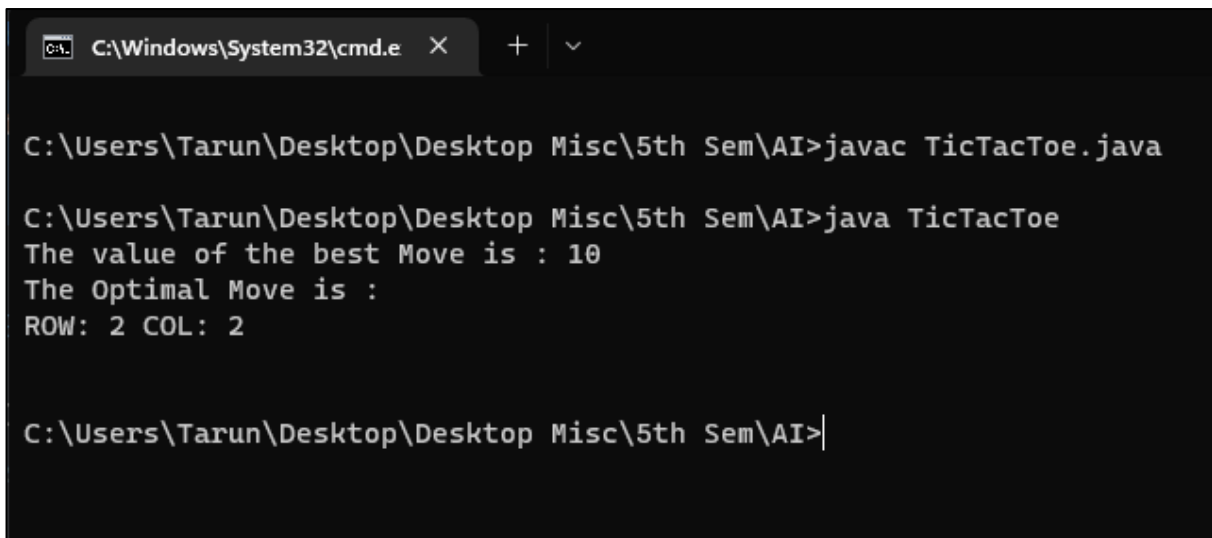
    public static void main(String[] args) {
        char[][] board = {{'x', 'o', 'x'},
                           {'o', 'o', 'x'},
                           {'_', '_', '_'}};

        Move bestMove = findBestMove(board);

        System.out.print("The Optimal Move is :\n");
        System.out.printf("ROW: %d COL: %d\n", bestMove.row, bestMove.col);
    }
}

```

## OUTPUT:



```

C:\Windows\System32\cmd.e
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac TicTacToe.java
C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java TicTacToe
The value of the best Move is : 10
The Optimal Move is :
ROW: 2 COL: 2

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>

```

## 7.Hill Climbing Algorithm

*Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbour has a higher value.*

*Source: <https://www.javatpoint.com/hill-climbing-algorithm-in-ai>*

### CODE:

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

public class HillC {

    public static double findMinimum(BiFunction<Double, Double, Double> f){

        double curX = 0;

        double curY = 0;

        double curF = f.apply(curX, curY);

        for (double step = 1e6; step > 1e-7; ) {

            double bestF = curF;

            double bestX = curX;

            double bestY = curY;

            boolean find = false;

            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 (bestF > nextF) {

                    bestF = nextF;

                    bestX = nextX;

                    bestY = nextY;

                    find = true;

                }

            }

            step /= 2;

        }

        return bestF;

    }

}
```

```

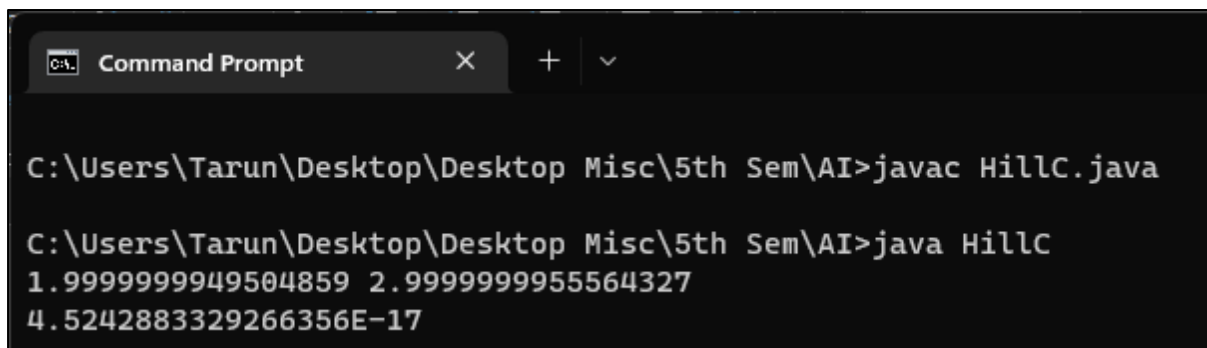
    }
    }
    if (!find) {
        step /= 2;
    }
    else {
        curX = bestX;
        curY = bestY;
        curF = bestF;
    }
}

System.out.println(curX + " " + curY);
return curF;
}

public static void main(String[] args)
{
    System.out.println(findMinimum((x, y) -> (x - 2) * (x - 2) + (y - 3) * (y - 3)));
}
}

```

## OUTPUT:



```

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac HillC.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java HillC
1.9999999949504859 2.9999999955564327
4.5242883329266356E-17

```



## 8. 8 Puzzle Problem

*The 8-puzzle problem belongs to the category of “sliding block puzzle” type of problem. The 8-puzzle is a square tray in which eight square tiles are placed. The remaining ninth square is uncovered. Each tile in the tray has a number on it. A tile that is adjacent to blank space can be slide into that space. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.*

*Source: <https://benchpartner.com/8-puzzle-problem-in-artificial-intelligence>*

### CODE:

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

class EightPuz
{
    public static int N = 3;
    public static class Node
    {
        Node parent;
        int mat[][] = new int[N][N];
        int x, y;
        int cost;
        int level;
    }

    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("");
}

}

public static Node newNode(int mat[][], int x, int y, int newX, int newY, int level, Node parent){
    Node node = new Node();
    node.parent = parent;
    node.mat = new int[N][N];
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            node.mat[i][j] = mat[i][j];
        }
    }

    int temp = node.mat[x][y];
    node.mat[x][y] = node.mat[newX][newY];
    node.mat[newX][newY] = temp;

    node.cost = Integer.MAX_VALUE;
    node.level = level;

    node.x = newX;
    node.y = newY;
    return node;
}

public static int row[] = { 1, 0, -1, 0 };
public static int col[] = { 0, -1, 0, 1 };

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;
}

public static int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N)?1:0;
}

public static void printPath(Node root){
    if(root == null){
        return;
    }
    printPath(root.parent);
    printMatrix(root.mat);
    System.out.println("");
}

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);
        return;
    }
    for (int i = 0; i < 4; i++)
    {
        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);
        }
    }
}

public static void main (String[] args)
{
    int initialMat[][] =
    {
        {1, 2, 3},
        {5, 6, 0},
        {7, 8, 4}
    };

```

```

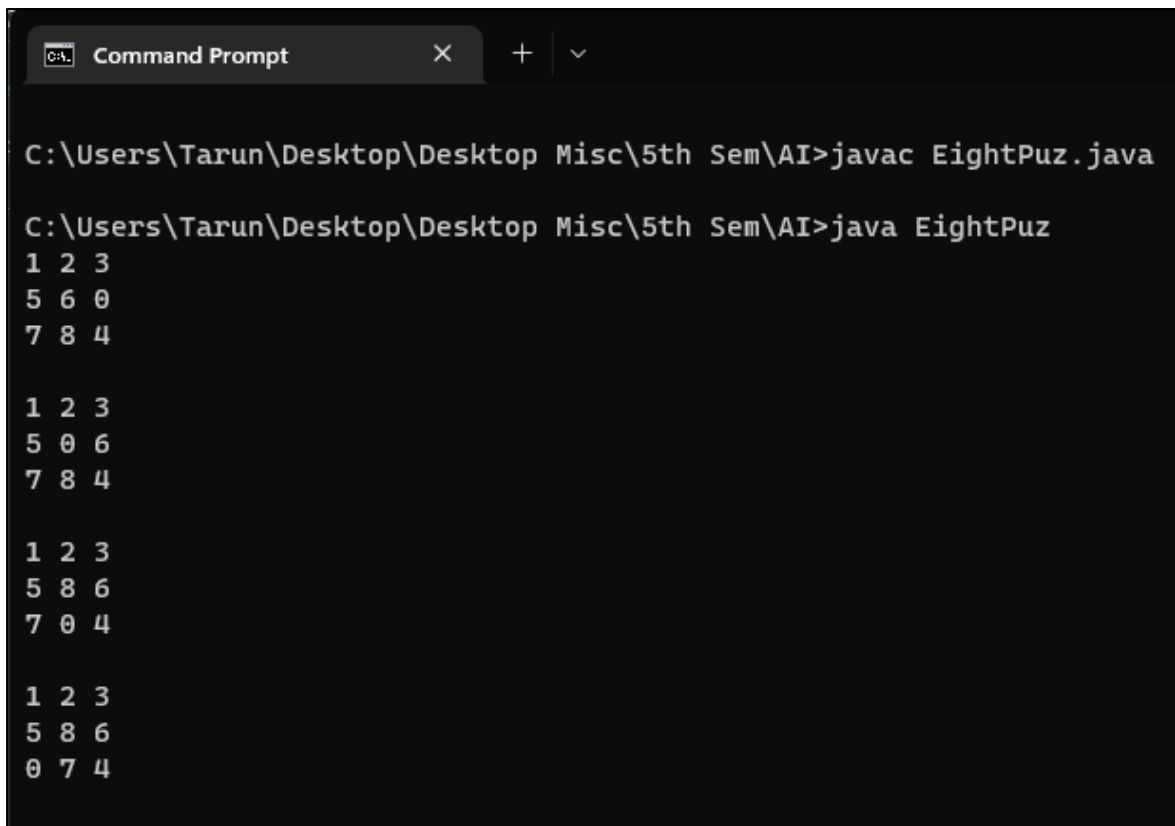
int finalMat[][] =
{
    {1, 2, 3},
    {5, 8, 6},
    {0, 7, 4}
};

int x = 1, y = 2;

solve(initialMat, x, y, finalMat);
}
}

```

## OUTPUT:



```

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac EightPuz.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java EightPuz
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

```

## 9.Travelling Salesman Problem

*The travelling salesman problem (also called the travelling salesperson problem or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.*

*Source: [https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem)*

### CODE:

```
class TravSal
{

    static int tsp(int[][] graph, boolean[] v, int currPos, int n, int count, int cost, int ans)
    {
        if (count == n && graph[currPos][0] > 0)
        {
            ans = Math.min(ans, cost + graph[currPos][0]);
            return ans;
        }

        for (int i = 0; i < n; i++)
        {
            if (v[i] == false && graph[currPos][i] > 0)
            {
                v[i] = true;
                ans = tsp(graph, v, i, n, count + 1, cost + graph[currPos][i], ans);
                v[i] = false;
            }
        }
    }
}
```

```

        return ans;
    }

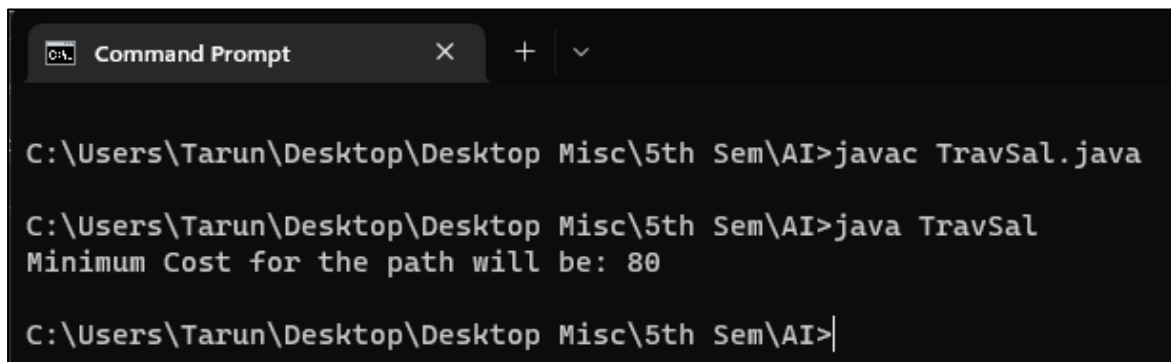
    public static void main(String[] args)
    {
        int nodes = 4;
        int[][] graph = {{0, 10, 15, 20},
                        {10, 0, 35, 25},
                        {15, 35, 0, 30},
                        {20, 25, 30, 0}};

        boolean[] v = new boolean[nodes];

        v[0] = true;
        int ans = Integer.MAX_VALUE;
        ans = tsp(graph, v, 0, nodes, 1, 0, ans);
        System.out.println("Minimum Cost for the path will be: " + ans);
    }
}

```

## OUTPUT:



```

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>javac TravSal.java

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>java TravSal
Minimum Cost for the path will be: 80

C:\Users\Tarun\Desktop\Desktop Misc\5th Sem\AI>

```

## 10. Convert following Prolong predicates into Semantic Net:

**cat ( tom ) .**

**cat ( cat ) .**

**mat ( mat 1 ) .**

**sat on ( cat1 , mat 1 ) .**

**bird ( bird 1 ) .**

**like ( X , cream ) : -cat ( X ) .**

**mammal ( X ) : -cat ( X ) .**

**has ( X , fur ) -mammal ( X ) .**

**animal ( X ) -mammal ( X ) .**

**animal ( X ) -bird( X ) .**

**owns john , tom ) .**

**is coloured ( tom , ginger ) .**

*Semantic networks are an alternative to predicate logic as a form of knowledge representation. The idea is that we can store our knowledge in the form of a graph, with nodes representing objects in the world, and arcs representing relationships between those objects.*

*Source: <http://www.eecs.qmul.ac.uk/~mmh/AINotes/AINotes4.pdf>*



## OUTPUT:

