practical no 1

import java.util.Stack;

class Main {

public static int total = 0;

static class Node {

int x;

int y;

public Node(int x, int y) {

this.x = x;

this.y = y;

}

}

static Stack<Node> result = new Stack<>();

static boolean[][] visited = new boolean[5][5];

static boolean solveDFS(int curj1, int curj2, int jug1, int jug2, int tx, int ty) {

// If we reach the target state

if (curj1 == tx && curj2 == ty) {

result.add(new Node(curj1, curj2));

return true;

}

total++;

// If the state has already been visited

if (visited[curj1][curj2]) {

return false;

}

// Mark the state as visited

visited[curj1][curj2] = true;

// Fill the first jug completely

if (curj1 < jug1 && solveDFS(jug1, curj2, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

// Fill the second jug completely

if (curj2 < jug2 && solveDFS(curj1, jug2, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

// Empty the first jug

if (curj1 > 0 && solveDFS(0, curj2, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

// Empty the second jug

if (curj2 > 0 && solveDFS(curj1, 0, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

// Pour water from the first jug into the second jug

if (curj1 > 0 && curj2 < jug2) {

int transfer = Math.min(curj1, jug2 - curj2);

if (solveDFS(curj1 - transfer, curj2 + transfer, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

}

// Pour water from the second jug into the first jug

if (curj2 > 0 && curj1 < jug1) {

int transfer = Math.min(curj2, jug1 - curj1);

if (solveDFS(curj1 + transfer, curj2 - transfer, jug1, jug2, tx, ty)) {

result.add(new Node(curj1, curj2));

return true;

}

}

// Unmark the state as visited (backtracking)

visited[curj1][curj2] = false;

return false;

}

// Function to print the stack in reverse order

public static void printReverseStack(Stack<Node> stack) {

// Create a temporary stack to reverse the order

Stack<Node> temporaryStack = new Stack<>();

// Copy all the elements from the original stack to the temporary stack

while (!stack.isEmpty()) {

temporaryStack.push(stack.pop());

}

// Print the reversed stack

for (Node n : temporaryStack) {

System.out.println("(" + n.x + "," + n.y + ")");

}

}

public static void main(String[] args) {

// Solve the problem using DFS

boolean res = solveDFS(0, 0, 4, 3, 2, 0);

// Print the solution path

if (res) {

printReverseStack(result);

} else {

System.out.println("No solution found.");

}

// Print the total nodes explored

System.out.println("Total Nodes: " + total);

}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical 2 aim Write a program to Implement BFS for Missionaries and Cannibal Problem.

import java.util.\*;

class State {

int missionaries;

int cannibals;

int boat;

int missionariesRight;

int cannibalsRight;

public State(int missionaries, int cannibals, int boat, int missionariesRight, int cannibalsRight) {

this.missionaries = missionaries;

this.cannibals = cannibals;

this.boat = boat;

this.missionariesRight = missionariesRight;

this.cannibalsRight = cannibalsRight;

}

public boolean isValid() {

if (missionaries < 0 || cannibals < 0 || missionariesRight < 0 || cannibalsRight < 0) {

return false;

}

if (missionaries > 0 && missionaries < cannibals) {

return false;

}

if (missionariesRight > 0 && missionariesRight < cannibalsRight) {

return false;

}

return true;

}

public boolean isGoal() {

return missionaries == 0 && cannibals == 0 && boat == 0;

}

@Override

public boolean equals(Object obj) {

if (this == obj) return true;

if (obj == null || getClass() != obj.getClass()) return false;

State state = (State) obj;

return missionaries == state.missionaries && cannibals == state.cannibals &&

boat == state.boat && missionariesRight == state.missionariesRight &&

cannibalsRight == state.cannibalsRight;

}

@Override

public int hashCode() {

return Objects.hash(missionaries, cannibals, boat, missionariesRight, cannibalsRight);

}

@Override

public String toString() {

return "(" + missionaries + ", " + cannibals + ", " + boat + ") -> (" +

missionariesRight + ", " + cannibalsRight + ")";

}

}

class Main {

public static List<State> getSuccessors(State state) {

List<State> successors = new ArrayList<>();

int[][] moves = {{1, 0}, {2, 0}, {0, 1}, {0, 2}, {1, 1}};

if (state.boat == 1) { // Boat on the left side

for (int[] move : moves) {

State newState = new State(

state.missionaries - move[0],

state.cannibals - move[1],

0,

state.missionariesRight + move[0],

state.cannibalsRight + move[1]

);

if (newState.isValid()) {

successors.add(newState);

}

}

} else { // Boat on the right side

for (int[] move : moves) {

State newState = new State(

state.missionaries + move[0],

state.cannibals + move[1],

1,

state.missionariesRight - move[0],

state.cannibalsRight - move[1]

);

if (newState.isValid()) {

successors.add(newState);

}

}

}

return successors;

}

public static List<State> bfs(State initialState) {

Queue<State> queue = new LinkedList<>();

Set<State> visited = new HashSet<>();

Map<State, State> parent = new HashMap<>();

queue.add(initialState);

visited.add(initialState);

parent.put(initialState, null);

while (!queue.isEmpty()) {

State state = queue.poll();

if (state.isGoal()) {

List<State> path = new ArrayList<>();

while (state != null) {

path.add(state);

state = parent.get(state);

}

Collections.reverse(path);

return path;

}

for (State successor : getSuccessors(state)) {

if (!visited.contains(successor)) {

queue.add(successor);

visited.add(successor);

parent.put(successor, state);

}

}

}

return null;

}

public static void main(String[] args) {

State initialState = new State(3, 3, 1, 0, 0);

List<State> solution = bfs(initialState);

if (solution != null) {

for (State step : solution) {

System.out.println(step);

}

} else {

System.out.println("No solution found.");

}

}

}

output

(3, 3, 1) -> (0, 0)

(3, 1, 0) -> (0, 2)

(3, 2, 1) -> (0, 1)

(3, 0, 0) -> (0, 3)

(3, 1, 1) -> (0, 2)

(1, 1, 0) -> (2, 2)

(2, 2, 1) -> (1, 1)

(0, 2, 0) -> (3, 1)

(0, 3, 1) -> (3, 0)

(0, 1, 0) -> (3, 2)

(1, 1, 1) -> (2, 2)

(0, 0, 0) -> (3, 3)

---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Practical no 3

Write a program to implement Heuristic(Steepest Ascent)Search for 8 puzzle game

problem.

import java.util.\*;

class PuzzleState {

int[][] puzzle;

int heuristic;

public PuzzleState(int[][] puzzle, int heuristic) {

this.puzzle = puzzle;

this.heuristic = heuristic;

}

}

public class Main {

public static void printPuzzle(int[][] puzzle) {

for (int[] row : puzzle) {

System.out.println(Arrays.toString(row));

}

System.out.println();

}

public static int calculateHeuristic(int[][] puzzle, int[][] goal) {

int misplaced = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (puzzle[i][j] != 0 && puzzle[i][j] != goal[i][j]) {

misplaced++;

}

}

}

return misplaced;

}

public static List<int[][]> getPossibleMoves(int[][] puzzle) {

List<int[][]> moves = new ArrayList<>();

int x = 0, y = 0;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (puzzle[i][j] == 0) {

x = i;

y = j;

break;

}

}

}

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

for (int[] dir : directions) {

int newX = x + dir[0];

int newY = y + dir[1];

if (newX >= 0 && newX < 3 && newY >= 0 && newY < 3) {

int[][] newPuzzle = deepCopy(puzzle);

newPuzzle[x][y] = newPuzzle[newX][newY];

newPuzzle[newX][newY] = 0;

moves.add(newPuzzle);

}

}

return moves;

}

public static int[][] deepCopy(int[][] puzzle) {

int[][] copy = new int[3][3];

for (int i = 0; i < 3; i++) {

System.arraycopy(puzzle[i], 0, copy[i], 0, 3);

}

return copy;

}

public static int[][] steepestAscent(int[][] initialState, int[][] goalState) {

int[][] current = initialState;

int currentHeuristic = calculateHeuristic(current, goalState);

while (true) {

printPuzzle(current);

List<int[][]> neighbors = getPossibleMoves(current);

int[][] bestNeighbor = null;

int bestHeuristic = currentHeuristic;

for (int[][] neighbor : neighbors) {

int heuristic = calculateHeuristic(neighbor, goalState);

if (heuristic < bestHeuristic) {

bestHeuristic = heuristic;

bestNeighbor = neighbor;

}

}

if (bestNeighbor == null || bestHeuristic >= currentHeuristic) {

System.out.println("Reached local maximum or no better neighbor found.");

break;

}

current = bestNeighbor;

currentHeuristic = bestHeuristic;

if (currentHeuristic == 0) {

System.out.println("Goal state reached!");

break;

}

}

return current;

}

public static void main(String[] args) {

int[][] initialState = {

{ 1, 2, 3 },

{ 5, 6, 0 },

{ 7, 8, 4 }

};

int[][] goalState = {

{ 1, 2, 3 },

{ 0, 8, 6 },

{ 5, 7, 4 }

};

int[][] solution = steepestAscent(initialState, goalState);

System.out.println("Goal State is:");

printPuzzle(solution);

}

}

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]

Goal state reached!

Goal State is:

[1, 2, 3]

[0, 8, 6]

[5, 7, 4]

=== Code Execution Successful ===

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical no 4

Aim

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical no 5

write a prgram to solve the maze probelm using the A\* algo

import java.util.\*;

class MazeNode implements Comparable<MazeNode> {

int[] position; // Position in the maze (x, y)

int gCost; // Cost from start to this node

int hCost; // Heuristic cost to the goal

int fCost; // Total cost (g + h)

MazeNode parent; // Parent node for path reconstruction

public MazeNode(int[] position, int gCost, int hCost, MazeNode parent) {

this.position = position;

this.gCost = gCost;

this.hCost = hCost;

this.fCost = gCost + hCost;

this.parent = parent;

}

@Override

public int compareTo(MazeNode other) {

return Integer.compare(this.fCost, other.fCost);

}

}

public class Main {

public static int manhattanDistance(int[] current, int[] end) {

return Math.abs(current[0] - end[0]) + Math.abs(current[1] - end[1]);

}

public static List<int[]> aStarSolveMaze(int[][] maze, int[] start, int[] end) {

PriorityQueue<MazeNode> openList = new PriorityQueue<>();

Set<String> closedList = new HashSet<>();

openList.add(new MazeNode(start, 0, manhattanDistance(start, end), null));

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

while (!openList.isEmpty()) {

MazeNode currentNode = openList.poll();

if (Arrays.equals(currentNode.position, end)) {

// Reconstruct the path

List<int[]> path = new ArrayList<>();

while (currentNode != null) {

path.add(currentNode.position);

currentNode = currentNode.parent;

}

Collections.reverse(path);

return path;

}

closedList.add(Arrays.toString(currentNode.position));

for (int[] direction : directions) {

int newX = currentNode.position[0] + direction[0];

int newY = currentNode.position[1] + direction[1];

if (newX >= 0 && newY >= 0 && newX < maze.length && newY < maze[0].length && maze[newX][newY] == 0) {

int[] neighborPos = { newX, newY };

if (!closedList.contains(Arrays.toString(neighborPos))) {

int gCost = currentNode.gCost + 1;

int hCost = manhattanDistance(neighborPos, end);

MazeNode neighborNode = new MazeNode(neighborPos, gCost, hCost, currentNode);

openList.add(neighborNode);

}

}

}

}

return null; // No path found

}

public static void main(String[] args) {

int[][] maze = {

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

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

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

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

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

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

};

int[] start = { 0, 0 };

int[] end = { 5, 3 };

List<int[]> path = aStarSolveMaze(maze, start, end);

if (path != null) {

System.out.println("Path found:");

for (int[] position : path) {

System.out.println(Arrays.toString(position));

}

} else {

System.out.println("No path found.");

}

}

}

Path found:

[0, 0]

[1, 0]

[2, 0]

[3, 0]

[4, 0]

[4, 1]

[4, 2]

[4, 3]

[5, 3]

=== Code Execution Successful ===

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical no 6

Write a program to Implement Min-Max Algorithm for Tic-Tac-

Toe game problem.

import java.util.\*;

public class Main {

public static void printBoard(char[][] board) {

for (char[] row : board) {

System.out.println(String.join("|", new String(new char[] { row[0], row[1], row[2] })));

System.out.println("-----");

}

}

public static boolean checkWinner(char[][] board, char player) {

for (int i = 0; i < 3; i++) {

if (board[i][0] == player && board[i][1] == player && board[i][2] == player) {

return true;

}

if (board[0][i] == player && board[1][i] == player && board[2][i] == player) {

return true;

}

}

return (board[0][0] == player && board[1][1] == player && board[2][2] == player) ||

(board[0][2] == player && board[1][1] == player && board[2][0] == player);

}

public static boolean isFull(char[][] board) {

for (char[] row : board) {

for (char cell : row) {

if (cell == ' ') {

return false;

}

}

}

return true;

}

public static int minimax(char[][] board, int depth, boolean isMaximizing) {

if (checkWinner(board, 'X')) return 1;

if (checkWinner(board, 'O')) return -1;

if (isFull(board)) return 0;

if (isMaximizing) {

int bestScore = Integer.MIN\_VALUE;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == ' ') {

board[i][j] = 'X';

int score = minimax(board, depth + 1, false);

board[i][j] = ' ';

bestScore = Math.max(bestScore, score);

}

}

}

return bestScore;

} else {

int bestScore = Integer.MAX\_VALUE;

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == ' ') {

board[i][j] = 'O';

int score = minimax(board, depth + 1, true);

board[i][j] = ' ';

bestScore = Math.min(bestScore, score);

}

}

}

return bestScore;

}

}

public static int[] bestMove(char[][] board) {

int bestScore = Integer.MIN\_VALUE;

int[] move = new int[2];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 3; j++) {

if (board[i][j] == ' ') {

board[i][j] = 'X';

int score = minimax(board, 0, false);

board[i][j] = ' ';

if (score > bestScore) {

bestScore = score;

move[0] = i;

move[1] = j;

}

}

}

}

return move;

}

public static void playGame() {

Scanner scanner = new Scanner(System.in);

char[][] board = {

{ ' ', ' ', ' ' },

{ ' ', ' ', ' ' },

{ ' ', ' ', ' ' }

};

System.out.println("Tic-Tac-Toe Game! You are 'O' and the computer is 'X'");

while (true) {

printBoard(board);

System.out.println("Enter your move (row and column, 0-2): ");

int row = scanner.nextInt();

int col = scanner.nextInt();

if (board[row][col] != ' ') {

System.out.println("Invalid move! Try again.");

continue;

}

board[row][col] = 'O';

if (checkWinner(board, 'O')) {

printBoard(board);

System.out.println("You win!");

break;

}

if (isFull(board)) {

printBoard(board);

System.out.println("It's a draw!");

break;

}

int[] move = bestMove(board);

board[move[0]][move[1]] = 'X';

System.out.println("Computer chose position (" + move[0] + ", " + move[1] + ")");

if (checkWinner(board, 'X')) {

printBoard(board);

System.out.println("Computer wins!");

break;

}

if (isFull(board)) {

printBoard(board);

System.out.println("It's a draw!");

break;

}

}

scanner.close();

}

public static void main(String[] args) {

playGame();

}

}

output

Tic-Tac-Toe Game! You are 'O' and the computer is 'X'

-----

-----

-----

Enter your move (row and column, 0-2):

1 1

Computer chose position (0, 0)

X

-----

O

-----

-----

Enter your move (row and column, 0-2):

2 0

Computer chose position (0, 2)

X X

-----

O

-----

O

-----

Enter your move (row and column, 0-2):

2 1

Computer chose position (0, 1)

XXX

-----

O

-----

OO

-----

Computer wins!

=== Code Execution Successful ===

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical no 7

write a program to implement the 8 queen probelm

import java.util.\*;

public class Main{

private static final int N = 8;

private static List<int[]> solutions = new ArrayList<>();

public static void solveNQueens(int row, int[] board) {

if (row == N) {

solutions.add(board.clone());

return;

}

for (int col = 0; col < N; col++) {

if (isSafe(row, col, board)) {

board[row] = col;

solveNQueens(row + 1, board);

}

}

}

public static boolean isSafe(int row, int col, int[] board) {

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

if (board[i] == col || Math.abs(board[i] - col) == Math.abs(i - row)) {

return false;

}

}

return true;

}

public static void printSolution(int[] board) {

for (int row : board) {

for (int col = 0; col < N; col++) {

if (col == row) {

System.out.print("Q ");

} else {

System.out.print(". ");

}

}

System.out.println();

}

System.out.println();

}

public static void main(String[] args) {

int[] board = new int[N];

solveNQueens(0, board);

if (solutions.size() < 2) {

System.out.println("Not enough solutions found.");

return;

}

System.out.println("Second Last Solution:");

printSolution(solutions.get(solutions.size() - 2));

System.out.println("Last Solution:");

printSolution(solutions.get(solutions.size() - 1));

}

}

output for the

Second Last Solution:

. . . . . . . Q

. . Q . . . . .

Q . . . . . . .

. . . . . Q . .

. Q . . . . . .

. . . . Q . . .

. . . . . . Q .

. . . Q . . . .

Last Solution:

. . . . . . . Q

. . . Q . . . .

Q . . . . . . .

. . Q . . . . .

. . . . . Q . .

. Q . . . . . .

. . . . . . Q .

. . . . Q . . .

=== Code Execution Successful ===

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Practical no 8 : prolog program