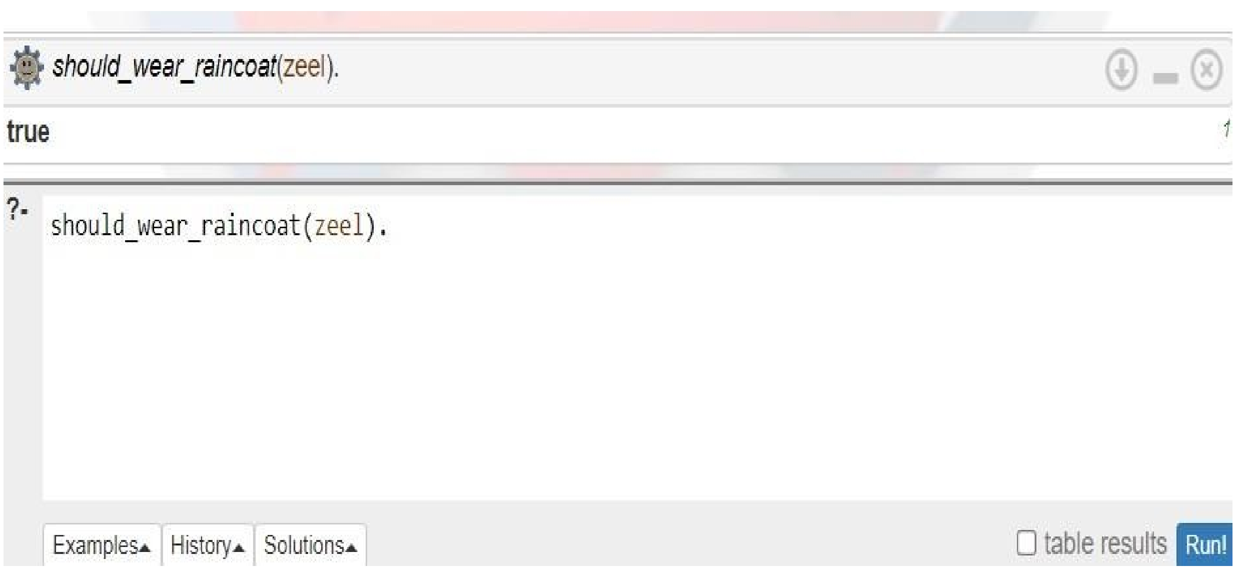
Practical No.1

Aim: Write the following in form of Facts and rules and solve the query
% Today is rainy
% Zeel is a person
% Every person should wear raincoat if it is rainy today
% Query: Should zeel wear raincoat today ?

Program Code:

```
todayIsRainy.
person(zeel).
should_wear_raincoat(Person) :-
    todayIsRainy,
    person(Person).
```

Output Screenshot:



Practical No.2

AIM: Load the following facts into familytree.pl which is shared on classroom, consult the Prolog file and answer the given questions Use SWI – Prolog for answering the following questions (load the rules in the file familytree.pl)

1. Is Albert a parent of Peter?
2. Who is the child of Jim?
3. Who are the parents of Brian?
4. Is Irene a grandparent of Brian?
5. Find all the grandchildren of Irene
6. Now add the following rule to familytree.pl and re-consult:

older(Person1, Person2) :-
yearOfBirth(Person1, Year1),
yearOfBirth(Person2, Year2), Year2 > Year1.

7. Who is older than Pat?
8. Who is younger than Darren?
9. List the siblings of Sandra.
10. Who is the older brother of Sandra?
11. Find the predecessors of Kyle.
12. Does Kate have a sister?
13. How many females and males are there in the knowledge base?

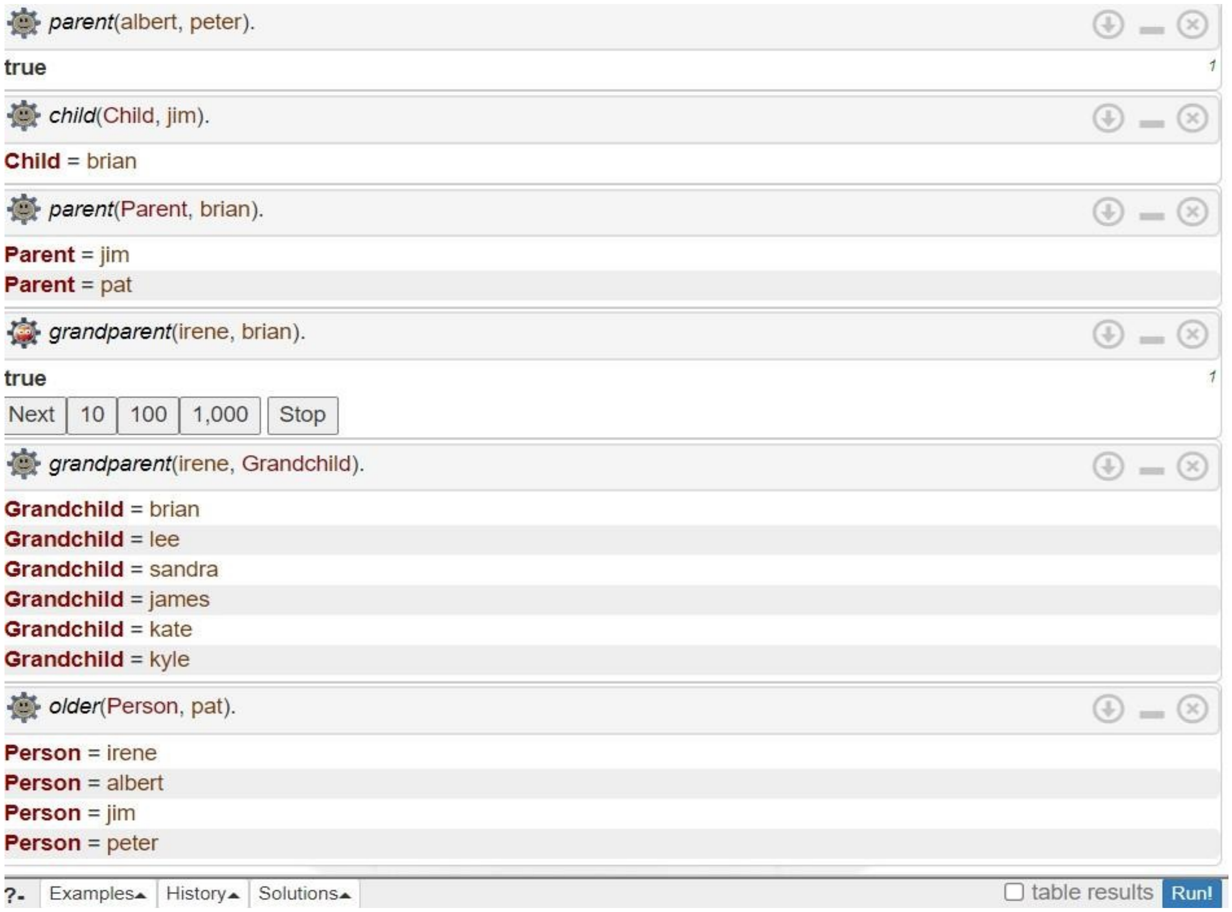
Program Code:

```
parent(albert, jim).
parent(albert, peter).
parent(jim, brian).
parent(john, darren).
parent(peter, lee).
parent(peter, sandra).
parent(peter, james).
parent(peter, kate).
parent(peter, kyle).
parent(brian, jenny).
```

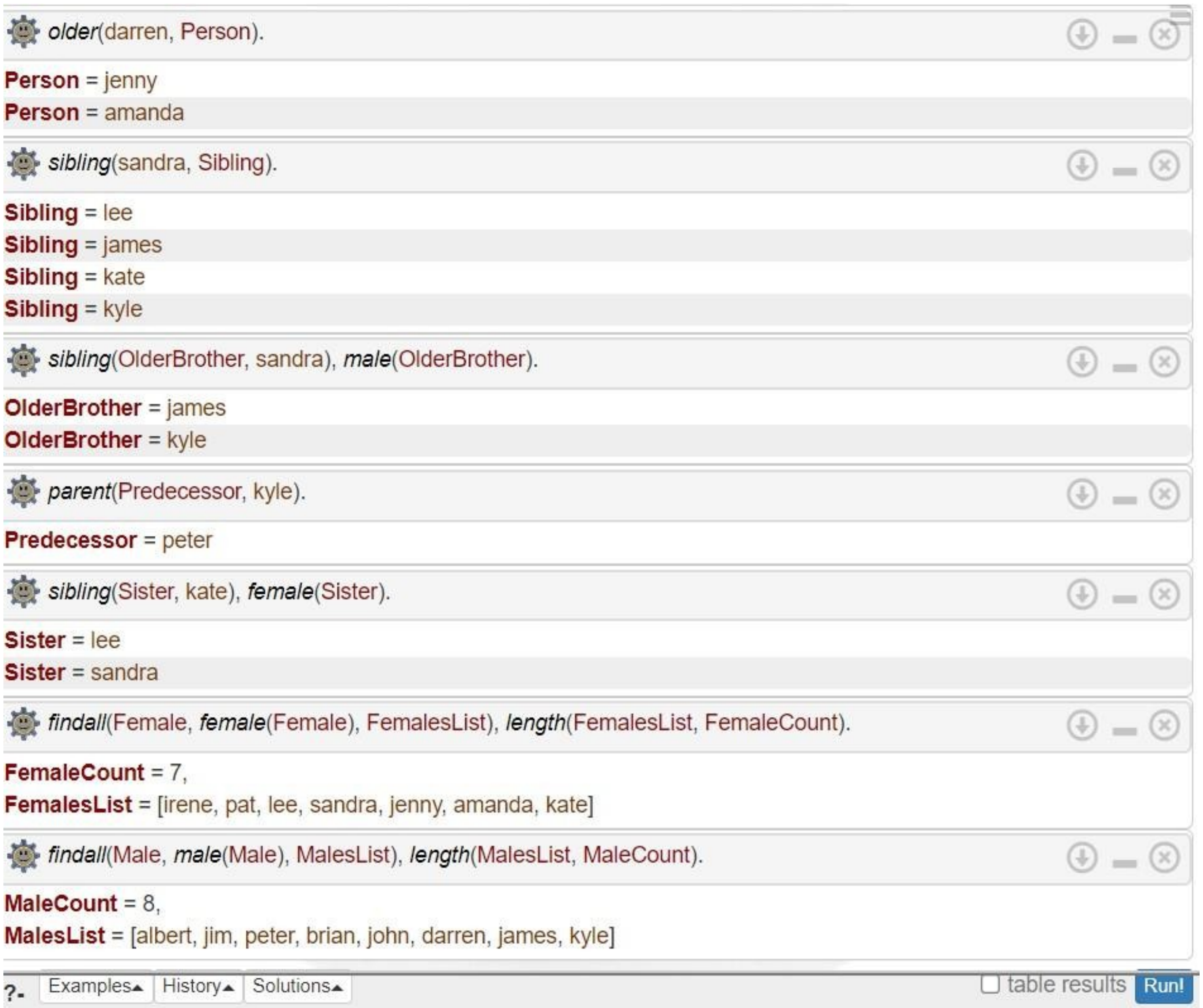
```
parent(irene, jim).
parent(irene, peter).
parent(pat, brian).
parent(pat, darren).
parent(amanda, jenny).

% female(Person)

female(irene).
female(pat).
female(lee).
female(sandra).
female(jenny).
female(amanda).
female(kate).

% male(Person)

male(albert).
male(jim).
male(peter).
male(brian).
male(john).
male(darren).
male(james).
male(kyle).

% yearOfBirth(Person, Year).

yearOfBirth(irene, 1923).
yearOfBirth(pat, 1954).
yearOfBirth(lee, 1970).
yearOfBirth(sandra, 1973).
yearOfBirth(jenny, 2004).
yearOfBirth(amanda, 1979).
yearOfBirth(albert, 1926).
yearOfBirth(jim, 1949).
yearOfBirth(peter, 1945).
yearOfBirth(brian, 1974).
yearOfBirth(john, 1955).
```

```
yearOfBirth(darren, 1976).
yearOfBirth(james, 1969).
yearOfBirth(kate, 1975).
yearOfBirth(kyle, 1976).

% Rules

child(Child, Parent) :- parent(Parent, Child).

% Grandparent predicate
grandparent(Grandparent, Grandchild) :- parent(Grandparent, Parent), child(Grandchild, Parent).

% Sibling predicate
sibling(Sibling1, Sibling2) :- parent(Parent, Sibling1), parent(Parent, Sibling2), Sibling1 \=
Sibling2.

% Older predicate
older(Person1, Person2) :- yearOfBirth(Person1, Year1), yearOfBirth(Person2, Year2), Year2 >
Year1.
```

Output Screenshot: 1 to 7



Output Screenshot: 8 to 13



```
older(darren, Person).
Person = jenny
Person = amanda

sibling(sandra, Sibling).
Sibling = lee
Sibling = james
Sibling = kate
Sibling = kyle

sibling(OlderBrother, sandra), male(OlderBrother).
OlderBrother = james
OlderBrother = kyle

parent(Predecessor, kyle).
Predecessor = peter

sibling(Sister, kate), female(Sister).
Sister = lee
Sister = sandra

findall(Female, female(Female), FemalesList), length(FemalesList, FemaleCount).
FemaleCount = 7,
FemalesList = [irene, pat, lee, sandra, jenny, amanda, kate]

findall(Male, male(Male), MalesList), length(MalesList, MaleCount).
MaleCount = 8,
MalesList = [albert, jim, peter, brian, john, darren, james, kyle]

?- Examples History Solutions   table results  Run!
```

Practical No.3

AIM: 1. Write a prolog program to implement a Menu Driven Calculator.

Program Code:

```
menu :-
    write('--- Menu Driven Calculator ---'), nl,
    write('1. Addition'), nl,
    write('2. Subtraction'), nl,
    write('3. Multiplication'), nl,
    write('4. Division'), nl,
    write('5. Exit'), nl,
    write('Enter your choice (1-5): '), nl,
    read(Choice),
    handle_choice(Choice).

% Handle user choices
handle_choice(1) :-
    get_numbers(X, Y),
    Result is X + Y,
    write('Result: '), write(Result), nl,
    menu.
handle_choice(2) :-
    get_numbers(X, Y),
    Result is X - Y,
    write('Result: '), write(Result), nl,
    menu.
handle_choice(3) :-
    get_numbers(X, Y),
    Result is X * Y,
    write('Result: '), write(Result), nl,
    menu.
handle_choice(4) :-
    get_numbers(X, Y),
    ( Y =\= 0 ->
        Result is X / Y,
        write('Result: '), write(Result), nl;
        write('Error: Division by zero is not allowed.'), nl
```

```
    ),
    menu.
handle_choice(5) :-
    write('Exiting the calculator. Goodbye!'), nl.
handle_choice(_) :-
    write('Invalid choice, please try again.'), nl,
    menu.

% Get two numbers from the user
get_numbers(X, Y) :-
    write('Enter the first number: '),
    read(X),
    write('Enter the second number: '),
    read(Y).
```

Output Screenshot:

```
menu
--- Menu Driven Calculator ---
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Exit
Enter your choice (1-5):
3
Enter the first number:
12
Enter the second number:
21
Result: 252
--- Menu Driven Calculator ---
1. Addition
```

AIM: 2. Write a prolog program to find maximum and minimum salaries

Program Code:

```
% Employee facts: employee(Name, Salary)
employee(jacob, 34000).
employee(jeremy, 12000).
employee(kisanlal, 5000).
employee(ramlal, 90000).
employee(dharampal, 8000).

% Find the maximum salary
max_salary(Max) :-
    findall(Salary, employee(_, Salary), Salaries),
    max_list(Salaries, Max).

% Find the minimum salary
min_salary(Min) :-
    findall(Salary, employee(_, Salary), Salaries),
    min_list(Salaries, Min).
```

Output Screenshot:

AIM: 3. Write a prolog program to check whether a given number is odd or even.

Program Code:

```
is_even(Number) :-
    Number mod 2 =:= 0.
is_odd(Number) :-
    Number mod 2 =:= 1.
```

Output Screenshot:

Practical No.4

AIM: Write a program to implement Tic-Tac-Toe game problem

Program Code:

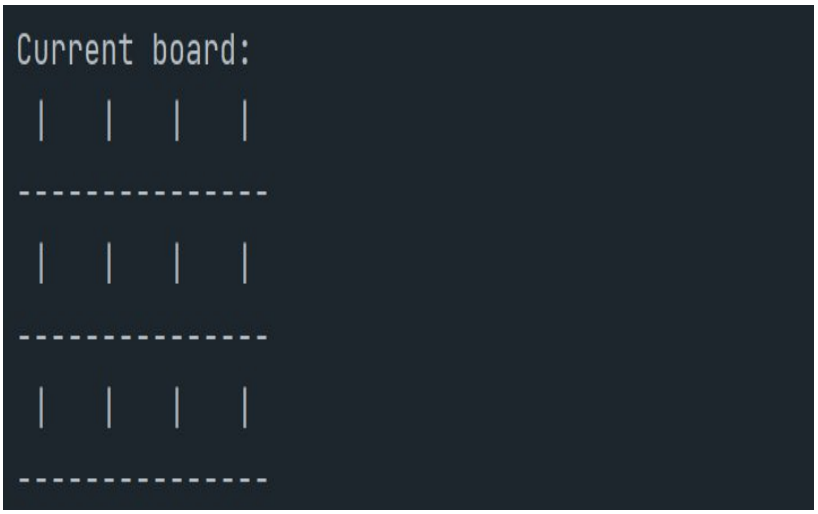
```
import java.util.Scanner;

public class TicTacToe
  { private char[][] board;
  private char currentPlayer;

  public TicTacToe()
    { board = new
    char[3][3];
    currentPlayer = 'X'; // X always starts first
    initializeBoard();
  }

  private void initializeBoard()
    {for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        {board[i][j] = ' ';
      }
    }
  }

  public void printBoard()
    { System.out.println("Current
    board:");for (int i = 0; i < 3; i++) {
      System.out.print(" | ");
      for (int j = 0; j < 3; j++) {
        System.out.print(board[i][j] + " | ");
      }
      System.out.println();
      System.out.println(" --------- ");
    }
  }

  public void changePlayer() {
```

```
        currentPlayer = (currentPlayer == 'X') ? 'O' : 'X';
    }

    public boolean placeMark(int row, int col) {
        if (row >= 0 && row < 3 && col >= 0 && col < 3 && board[row][col] == ' ')
            {board[row][col] = currentPlayer;
            return true;
        }
        return false;
    }

    public boolean checkForWin() {
        // Check rows
        for (int i = 0; i < 3; i++) {
            if (board[i][0] == currentPlayer && board[i][1] == currentPlayer && board[i][2] ==
currentPlayer) {
                return true;
            }
        }
        // Check columns
        for (int j = 0; j < 3; j++) {
            if (board[0][j] == currentPlayer && board[1][j] == currentPlayer && board[2][j] ==
currentPlayer) {
                return true;
            }
        }
        // Check diagonals
        if (board[0][0] == currentPlayer && board[1][1] == currentPlayer && board[2][2] ==
currentPlayer) {
            return true;
        }
        if (board[0][2] == currentPlayer && board[1][1] == currentPlayer && board[2][0] ==
currentPlayer) {
            return true;
        }
        return false;
    }

    public boolean isBoardFull()
        {for (int i = 0; i < 3; i++) {
```

```
            for (int j = 0; j < 3; j++)
                {if (board[i][j] == ' ')
                {
                    return false;
                }
            }
        }
        return true;
    }

    public static void main(String[] args)
        { TicTacToe game = new TicTacToe();
        Scanner scanner = new Scanner(System.in);
        boolean gameWon = false;

        while (!gameWon && !game.isBoardFull())
            {game.printBoard();
            System.out.println("Player " + game.currentPlayer + ", enter your move (row and
column): ");
            int row = scanner.nextInt() - 1;
            int col = scanner.nextInt() - 1;

            if (game.placeMark(row, col))
                { gameWon =
                game.checkForWin();if
                (!gameWon) {
                    game.changePlayer();
                }
            } else {
                System.out.println("This move is invalid. Try again.");
            }
        }

        game.printBoard();

        if (gameWon) {
            System.out.println("Player " + game.currentPlayer + " wins!");
        } else {
            System.out.println("It's a draw!");
        }

        scanner.close();
```

```
    }
}
```

Output Screenshot:



```
Player O, enter your move (row and column):
2
2
Current board:
 | X |   |   |
---------------
 |   | O |   |
---------------
 |   |   |   |
---------------
```

```
layer X, enter your move (row and column):


urrent board:
| X | X |   |
-------------
|   | O |   |
-------------
|   |   |   |
-------------
```

```
Player X, enter your move (row and column):
1
3
Current board:
 | X | X | X |
---------------
 |   | O | O |
---------------
 |   |   |   |
---------------
Player X wins!
```

Practical No.5

AIM: Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem)

Program Code:

```
import java.util.*;

class PuzzleState {
  int[][] board; // 3x3 board
  String path; // To store the path to reach this state
  int emptyRow, emptyCol; // Position of the empty space

  PuzzleState(int[][] board, int emptyRow, int emptyCol, String path)
    {this.board = board;
    this.emptyRow = emptyRow;
    this.emptyCol = emptyCol;
    this.path = path;
  }

  // Generate possible moves from current state
  List<PuzzleState> generateMoves() {
    List<PuzzleState> moves = new ArrayList<>();
    int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}; // Down, Up, Right, Left
    String[] moveNames = {"D", "U", "R", "L"}; // Move names for path

    for (int i = 0; i < directions.length; i++) {
      int newRow = emptyRow + directions[i][0];
      int newCol = emptyCol + directions[i][1];

      if (isValid(newRow, newCol)) {
        int[][] newBoard = copyBoard(board);
        // Swap empty space with the adjacent tile
        newBoard[emptyRow][emptyCol] = newBoard[newRow][newCol];
        newBoard[newRow][newCol] = 0; // Update empty space

        moves.add(new PuzzleState(newBoard, newRow, newCol, path + moveNames[i]));
      }
```

```
        }

        return moves;
    }

    // Check if the new position is valid
    boolean isValid(int row, int col) {
        return row >= 0 && row < 3 && col >= 0 && col < 3;
    }

    // Create a copy of the board
    int[][] copyBoard(int[][] original) {
        int[][] newBoard = new int[3][3];
        for (int i = 0; i < 3; i++) {
            System.arraycopy(original[i], 0, newBoard[i], 0, 3);
        }
        return newBoard;
    }

    // Check if the current state is the goal state
    boolean isGoalState() {
        int[][] goal = {
                {1, 2, 3},
                {4, 5, 6},
                {7, 8, 0}
        };

        for (int i = 0; i < 3; i++)
            { for (int j = 0; j < 3; j++)
            {
                if (board[i][j] != goal[i][j])
                    {return false;
                }
            }
        }
        return true;
    }
}

public class EightPuzzleBFS {
    public static void main(String[] args) {
```

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

        PuzzleState initialPuzzleState = new PuzzleState(initialState, 1, 1, "");
        String solution = bfs(initialPuzzleState);

        if (solution != null) {
            System.out.println("Solution found! Moves: " + solution);
        } else {
            System.out.println("No solution found.");
        }
    }

    // Perform BFS to find the solution
    public static String bfs(PuzzleState initialState)
        { Queue<PuzzleState> queue = new
        LinkedList<>();
        Set<String> visited = new HashSet<>(); // To avoid revisiting states

        queue.add(initialState);
        visited.add(arrayToString(initialState.board));

        while (!queue.isEmpty()) {
            PuzzleState currentState = queue.poll();

            if (currentState.isGoalState()) {
                return currentState.path; // Return the path if goal state is reached
            }

            for (PuzzleState nextState : currentState.generateMoves())
                {String stateString = arrayToString(nextState.board);
                if (!visited.contains(stateString))
                    {visited.add(stateString);
                    queue.add(nextState);
                }
            }
        }
        return null; // No solution found
```

```
    }

    // Convert board to string for storing visited states
    public static String arrayToString(int[][] board) {
        StringBuilder sb = new StringBuilder();
        for (int[] row : board) {
            for (int num : row)
                { sb.append(num).append(",
                ");
            }
        }
        return sb.toString();
    }
}
```

Output Screenshot:

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Editi
Solution found! Moves: RD
```

Practical No.6

AIM: Write a program to implement BFS (for 8 puzzle problem or Water Jug problem or any AI search problem)

Program Code:

```
class PuzzleState:
    def init (self, board, empty_row, empty_col, path):
        self.board = board  # 3x3 board
        self.empty_row = empty_row # Position of the empty space
        self.empty_col = empty_col
        self.path = path  # To store the path to reach this state

    def generate_moves(self):
        moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # Down, Up, Right, Left
        move_names = ["D", "U", "R", "L"]  # Move names for path

        for i, (dr, dc) in enumerate(directions):
            new_row = self.empty_row + dr
            new_col = self.empty_col + dc

            if self.is_valid(new_row, new_col):
                new_board = self.copy_board(self.board)
                # Swap empty space with the adjacent tile
                new_board[self.empty_row][self.empty_col] = new_board[new_row][new_col]
                new_board[new_row][new_col] = 0  # Update empty space

                moves.append(PuzzleState(new_board, new_row, new_col, self.path +
move_names[i]))

        return moves

    def is_valid(self, row, col):
        return 0 <= row < 3 and 0 <= col < 3

    def copy_board(self, original):
        return [row[:] for row in original] # Deep copy of the board
```

```
    def is_goal_state(self):
        goal = [
            [1, 2, 3],
            [4, 5, 6],
            [7, 8, 0]
        ]
        return self.board == goal  # Check if current board matches the goal


def dfs(current_state, visited):
    if current_state.is_goal_state():
        print(f"Solution found! Moves: {current_state.path}")
        return True  # Solution found

    state_string = array_to_string(current_state.board)
    if state_string in visited:
        return False # Already visited this state
    visited.add(state_string)

    for next_state in current_state.generate_moves():
        if dfs(next_state, visited):
            return True  # If the solution is found in the next state

    return False  # No solution found in this path


def array_to_string(board):
    return ''.join(str(num) for row in board for num in row)  # Convert board to string


if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    empty_row, empty_col = 1, 1 # Initial position of the empty space
    initial_puzzle_state = PuzzleState(initial_state, empty_row, empty_col, "")
    visited = set()  # To avoid revisiting states
```

```
if not dfs(initial_puzzle_state, visited):
    print("No solution found.")
```

Output Screenshot:

```
Solution found! Moves:
    DRUULDDRUULDDRUULDDRUULDDRUULLDDRUURDDLUURDDLUURDDLUURDDLUURDLDRUULDDRUULDDRUULDDR
    UULDDRUULLDDRUURDDLUURDDLUURDDLUURDDLUURDLDRUULDDRUULDDRUULDDRUULDDRUULLDDRUURDDLU
    URDDLUURDDLUURDDLUURDLDRUULDDRUULDDRUULDDRUULDDRUULLDDRUURDDLUURDDLUURDDLUURDDLUUR
    DLDRUULDDRUULDDRUULDDRUULDDRUULLDDRUURDDLUURDDLUURDDLUURDDLUURDLDRUULDDRUULDDRUULD
    DRUULDDRUULLDDRUURDDLUURDDLUURDDLUURDDLUURDLDRUULDDRUULDDRUULDDRUULDDRUULLDDRUURDD
    LUURDDLUURDDLUURDDLUURDD

=== Code Execution Successful ===
```

Practical No.7

Aim: Write a program to implement Single Player Game (Using Heuristic Function)

Program code:

```
import random

class Puzzle:
    def_init_(self, board):
        self.board = board
        self.empty_tile = self.find_empty_tile()

    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def display(self):
        for row in self.board:
            print(' '.join(str(tile) for tile in row))
        print()

    def move(self, direction):
        row, col = self.empty_tile

        if direction == 'U' and row > 0:
            self.swap_tiles(row, col, row - 1, col)
        elif direction == 'D' and row < 2:
            self.swap_tiles(row, col, row + 1, col)
        elif direction == 'L' and col > 0:
            self.swap_tiles(row, col, row, col - 1)
        elif direction == 'R' and col < 2:
            self.swap_tiles(row, col, row, col + 1)
        else:
            print("Invalid Move")

    def swap_tiles(self, r1, c1, r2, c2):
```

```
        self.board[r1][c1], self.board[r2][c2] = self.board[r2][c2], self.board[r1][c1]
        self.empty_tile = (r2, c2)

    def is_solved(self):
        return self.board == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

    def heuristic(self):
        distance = 0
        goal_positions = {1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (1, 0),
                          5: (1, 1), 6: (1, 2), 7: (2, 0), 8: (2, 1)}

        for i in range(3):
            for j in range(3):
                if self.board[i][j] != 0:
                    target_pos = goal_positions[self.board[i][j]]
                    distance += abs(i - target_pos[0]) + abs(j - target_pos[1])

        return distance

def main():
    # Create a solvable initial state of the puzzle
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]
    puzzle = Puzzle(initial_state)

    print("Initial Puzzle State:")
    puzzle.display()

    while not puzzle.is_solved():
        print("Heuristic (Manhattan Distance):", puzzle.heuristic())
        move = input("Enter move (U/D/L/R): ").strip().upper()
        puzzle.move(move)
        puzzle.display()

    print("Congratulations! You've solved the puzzle!")

if __name__ == "__main__":
    main()
```

Output Screenshot:

```
Initial Puzzle State:
1 2 3
4 0 5
7 8 6

Heuristic (Manhattan Distance): 4
Enter move (U/D/L/R): D
1 2 3
4 5 0
7 8 6

Heuristic (Manhattan Distance): 2
Enter move (U/D/L/R): R
1 2 3
4 5 6
7 8 0

Congratulations! You've solved the puzzle!
```

Practical No.8

Aim: Write a program to implement A* algorithm

Program code:

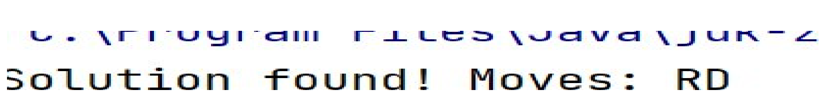
```
import java.util.*;

class PuzzleState {
    int[][] board;       // 3x3 board
    int emptyRow;       // Row of the empty space
    int emptyCol;       // Column of the empty space
    String path;        // Path to reach this state
    int g;              // Cost to reach this state
    int h;              // Heuristic cost to reach goal state
    int f;              // Total cost (g + h)

    public PuzzleState(int[][] board, int emptyRow, int emptyCol, String path, int g, int h)
        {this.board = board;
        this.emptyRow = emptyRow;
        this.emptyCol = emptyCol;
        this.path = path;
        this.g = g;
        this.h = h;
        this.f = g + h;
    }

    // Generate possible moves from the current state
    public List<PuzzleState> generateMoves() {
        List<PuzzleState> moves = new ArrayList<>();
        int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}}; // Down, Up, Right, Left
        String[] moveNames = {"D", "U", "R", "L"}; // Move names for path

        for (int i = 0; i < directions.length; i++) {
            int newRow = this.emptyRow + directions[i][0];
            int newCol = this.emptyCol + directions[i][1];

            if (isValid(newRow, newCol)) {
                int[][] newBoard = copyBoard(this.board);
```

```
            // Swap empty space with the adjacent tile
            newBoard[this.emptyRow][this.emptyCol] = newBoard[newRow][newCol];
            newBoard[newRow][newCol] = 0; // Update empty space

            // Calculate the new costs
            int newG = this.g + 1;
            int newH = heuristic(newBoard);
            moves.add(new PuzzleState(newBoard, newRow, newCol, this.path + moveNames[i],
newG, newH));
        }
    }

    return moves;
}

// Check if the position is valid
private boolean isValid(int row, int col) {
    return row >= 0 && row < 3 && col >= 0 && col < 3;
}

// Create a deep copy of the board
private int[][] copyBoard(int[][] original)
    {int[][] newBoard = new int[3][3];
    for (int i = 0; i < 3; i++)
        { System.arraycopy(original[i], 0, newBoard[i], 0,
        3);
    }
    return newBoard;
}

// Check if the current board is the goal state
public boolean isGoalState() {
    int[][] goal = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 0}
    };
    return Arrays.deepEquals(this.board, goal); // Check if current board matches the goal
}
public int heuristic(int[][] board)
    {int distance = 0;
```

```
        for (int i = 0; i < 3; i++)
            { for (int j = 0; j < 3; j++)
            {
                if (board[i][j] != 0) {
                    int targetRow = (board[i][j] - 1) / 3;
                    int targetCol = (board[i][j] - 1) % 3;
                    distance += Math.abs(targetRow - i) + Math.abs(targetCol - j);
                }
            }
        }
        return distance;
    }
}

// A* algorithm implementation
public class EightPuzzleAStar {
    public static void aStar(PuzzleState initialState) {
        Set<String> visited = new HashSet<>(); // To avoid revisiting states
        PriorityQueue<PuzzleState> priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(state -> state.f));
        priorityQueue.add(initialState); // Add initial state to the queue

        while (!priorityQueue.isEmpty()) {
            PuzzleState currentState = priorityQueue.poll(); // Get state with lowest f

            if (currentState.isGoalState()) {
                System.out.println("Solution found! Moves: " + currentState.path);
                return; // Solution found
            }

            String stateString = arrayToString(currentState.board);
            if (visited.contains(stateString)) {
                continue; // Already visited this state
            }
            visited.add(stateString);

            for (PuzzleState nextState : currentState.generateMoves())
                { priorityQueue.add(nextState); // Add new states to the
                queue
            }
        }
```

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

    // Convert the board to a string format for easy comparison
    private static String arrayToString(int[][] board) {
        StringBuilder sb = new StringBuilder();
        for (int[] row : board) {
            for (int num : row)
                {sb.append(num);
            }
        }
        return sb.toString();
    }

    public static void main(String[] args)
        {int[][] initialState = {
            {1, 2, 3},
            {4, 0, 5},
            {7, 8, 6}
        };
        int emptyRow = 1, emptyCol = 1; // Initial position of the empty space
        PuzzleState initialPuzzleState = new PuzzleState(initialState, emptyRow, emptyCol, "", 0,
0);

        // Calculate the heuristic for the initial state
        initialPuzzleState.h = initialPuzzleState.heuristic(initialState);
        initialPuzzleState.f = initialPuzzleState.g + initialPuzzleState.h;

        aStar(initialPuzzleState);
    }
}
```

Output Screenshot:



Practical No.9

Aim: Write a program to implement mini-max algorithm for any game development.

Program code:

```
import math

# Initial board setup
board = [' ' for _ in range(9)]

# Print the board
def print_board():
    for i in range(3):
        print('|'.join(board[i*3:(i+1)*3]))
        print('-' * 5)

# Check if there's a winner
def check_winner():
    win_conditions = [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                      (0, 3, 6), (1, 4, 7), (2, 5, 8),
                      (0, 4, 8), (2, 4, 6)]
    for condition in win_conditions:
        if board[condition[0]] == board[condition[1]] == board[condition[2]] != ' ':
            return board[condition[0]]
    return None

# Check if the board is full (draw)
def is_draw():
    return ' ' not in board

# Mini-Max algorithm
def minimax(is_maximizing):
    winner  = check_winner()
    if winner == 'X':
        return -1  # Player wins
    elif winner == 'O':
        return 1  # AI wins
    elif is_draw():
```

```
        return 0  # Draw

    if is_maximizing:
        best_score = -math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'O' # AI move
                score = minimax(False)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for i in range(9):
            if board[i] == ' ':
                board[i] = 'X' # Player move
                score = minimax(True)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score

# Get the best move for AI
def best_move():
    move = -1
    best_score = -math.inf
    for i in range(9):
        if board[i] == ' ':
            board[i] = 'O' # AI move
            score = minimax(False)
            board[i] = ' '
            if score > best_score:
                best_score = score
                move = i
    return move

# Main game loop
def main():
    print("Welcome to Tic-Tac-Toe!")
    print_board()
```

```
    while True:
        # Player move
        player_move = int(input("Enter your move (1-9): ")) - 1
        if board[player_move] == ' ':
            board[player_move] = 'X'
        else:
            print("Invalid move. Try again.")
            continue

        if check_winner() == 'X':
            print_board()
            print("You win!")
            break
        if is_draw():
            print_board()
            print("It's a draw!")
            break

        # AI move
        ai_move = best_move()
        board[ai_move] = 'O'
        print("AI's move:")
        print_board()

        if check_winner() == 'O':
            print("AI wins!")
            break
        if is_draw():
            print("It's a draw!")
            break

if __name__ == '__main__':
    main()
```

Output Screenshot:

```
Enter your move (1-9): 3
AI's move:
 | |X
-----
 |O|
-----
 | |
-----
Enter your move (1-9): 1
AI's move:
X|O|X
-----
 |O|
-----
 | |
-----
Enter your move (1-9): 4
AI's move:
X|O|X
-----
X|O|
-----
 |O|
-----
AI wins!
```

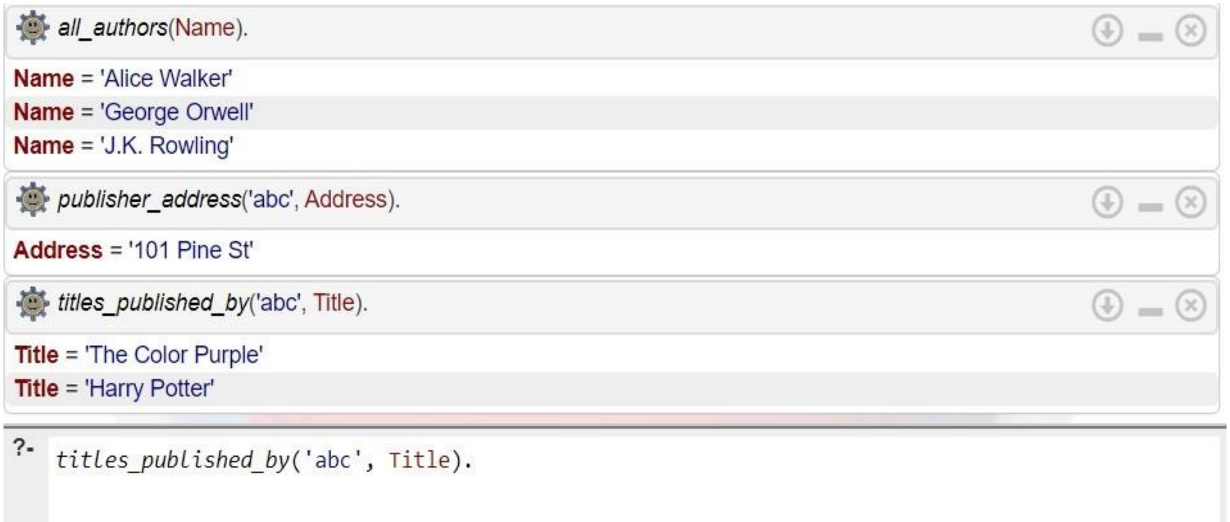
Practical No.10

Aim: Write a program in Prolog that will answer the question for the following facts.
Author(name,address,age)
Publisher(name,address)
Book(title,author,publisher)
a. What are the names of all authors?
b. What is the address of publisher abc?
c. What are the titles published by abc?

Program code:

```
% Facts

% Author(name, address, age)
author('Alice Walker', '123 Elm St', 75).
author('George Orwell', '456 Oak St', 72).
author('J.K. Rowling', '789 Maple St', 58).

% Publisher(name, address)
publisher('abc', '101 Pine St').
publisher('xyz', '202 Cedar St').

% Book(title, author, publisher)
book('The Color Purple', 'Alice Walker', 'abc').
book('1984', 'George Orwell', 'xyz').
book('Harry Potter', 'J.K. Rowling', 'abc').
book('Animal Farm', 'George Orwell', 'xyz').

% Rules
all_authors(Name) :- author(Name, _, _).

publisher_address(Name, Address) :- publisher(Name, Address).

titles_published_by(Publisher, Title) :- book(Title, _, Publisher).
```

Output Screenshot:



Practical No.11

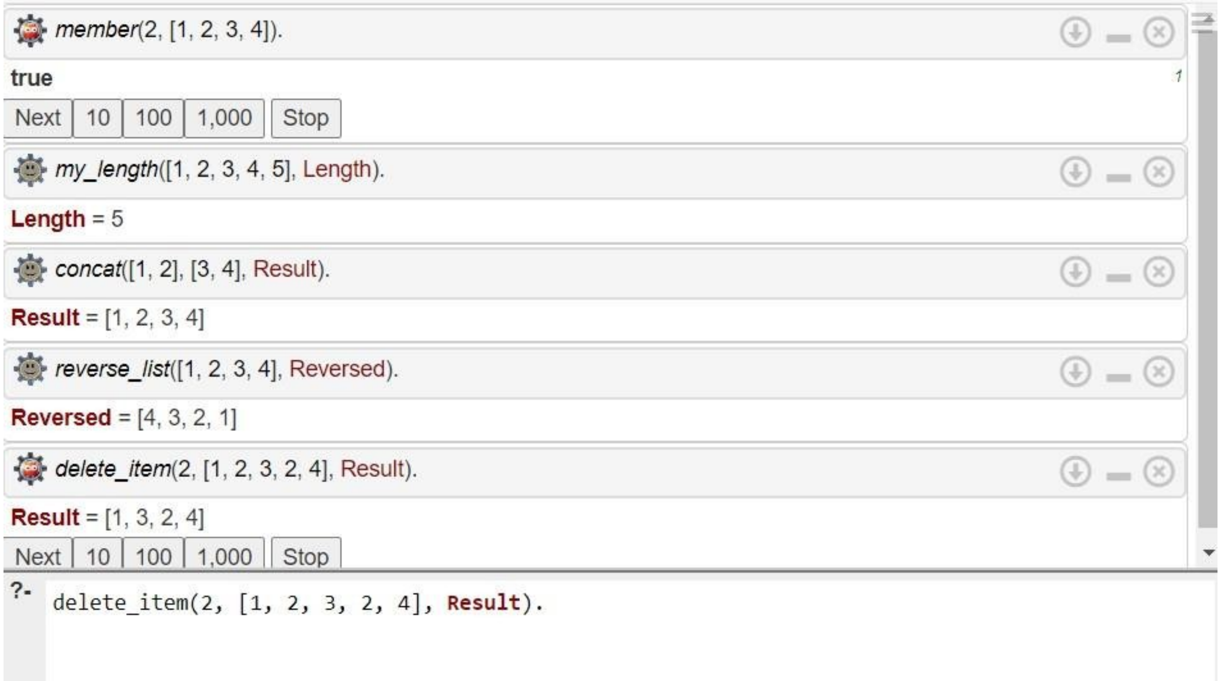
Aim: Write a program in Prolog to find,

- Member of a list
- The length of an input list.
- Concatenation of two
- Reverse of a list.
- ○Delete an item from list

Program Code:

```
% Check if an element is a member of a list
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).

% Find the length of a list
my_length([], 0).
my_length([_|Tail], Length) :- my_length(Tail, LengthTail), Length is LengthTail + 1.

% Concatenate two lists
concat([], L, L).
concat([Head|Tail1], L2, [Head|Tail3]) :- concat(Tail1, L2, Tail3).

% Reverse a list
reverse_list([], []).
reverse_list([Head|Tail], Reversed) :- reverse_list(Tail, ReversedTail), append(ReversedTail,
[Head], Reversed).

% Delete an item from a list
delete_item(_, [], []).
delete_item(X, [X|Tail], Tail).
delete_item(X, [Head|Tail], [Head|ResultTail]) :- delete_item(X, Tail, ResultTail).
```

Output Screenshot:


member(2, [1, 2, 3, 4]).
true
Next 10 100 1,000 Stop
my_length([1, 2, 3, 4, 5], Length).
Length = 5
concat([1, 2], [3, 4], Result).
Result = [1, 2, 3, 4]
reverse_list([1, 2, 3, 4], Reversed).
Reversed = [4, 3, 2, 1]
delete_item(2, [1, 2, 3, 2, 4], Result).
Result = [1, 3, 2, 4]
Next 10 100 1,000 Stop
?- delete_item(2, [1, 2, 3, 2, 4], Result).

Practical No.12

Aim: Write a Program in Prolog for reading in a character and decide whether it is a digit or an alphanumeric character

Program code:

```
check_character :-
    write('Enter a character: '),
    read(Character), % Read the character input
    ( char_type(Character, digit) -> % Check if it's a digit
        write(Character), write(' is a digit.')
    ; char_type(Character, alnum) -> % Check if it's alphanumeric
        write(Character), write(' is an alphanumeric character.')
    ;  write(Character), write(' is neither a digit nor an alphanumeric character.')
    ).

start :-
    check_character.
```

Output Screenshot:

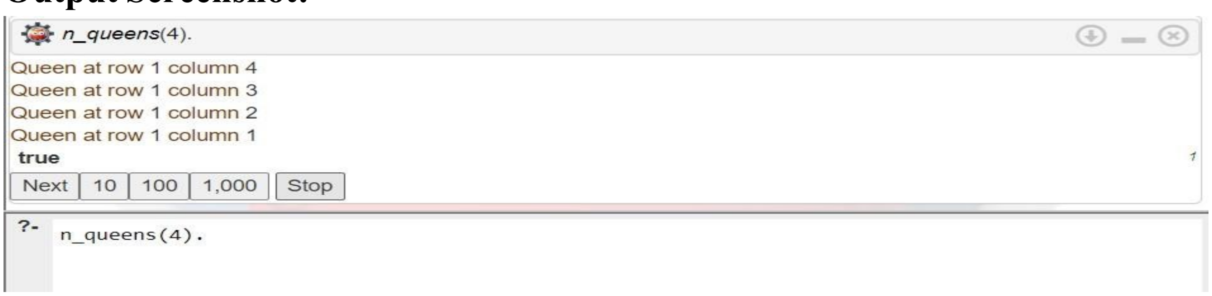
Practical No.13

Aim: Write a program to solve N-Queens problems using Prolog.

Program code:

```
% Place Queen in column Col on row Row
place_queen(Row, Col, [], []) :-
    !,
    write('Queen at row '), write(Row), write(' column '), write(Col), nl.
place_queen(Row, Col, [H|T], [C|Cols]) :-
    (   H =:= Row -> fail ;   % Same row
        C =:= Col -> fail ; % Same column
        abs(H-Row) =:= abs(C-Col) -> fail  % Same diagonal
    ),
    place_queen(Row, Col, T, Cols).

% Solve N-Queens problem for N queens
solve_n_queens(0, _) :- !.
solve_n_queens(N, Cols) :-
    N > 0,
    N1 is N - 1,
    between(1, N, Row),
    place_queen(Row, N, Cols, Cols1),
    solve_n_queens(N1, Cols1).

% Initialize and solve N-Queens problem
n_queens(N) :-
    solve_n_queens(N, []).
```

Output Screenshot:



Practical No.14

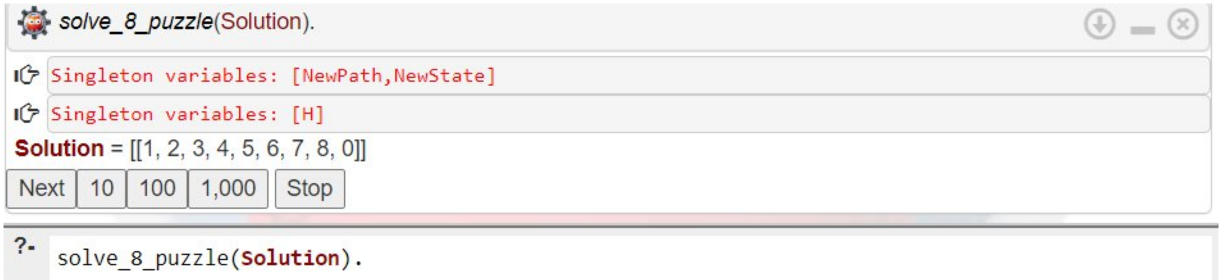
Aim: Write a program to solve 8 puzzle problem using Prolog

Program code:

```
:- use_module(library(lists)).

% Define the initial state and the goal state.
initial_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).
goal_state([1, 2, 3, 4, 5, 6, 7, 8, 0]).

% Find the solution using breadth-first search.
solve_8_puzzle(Solution) :-
    initial_state(Start),
    bfs([[Start]], Solution).

% Perform breadth-first search.
bfs([[State|Path]|_], [State|Path]) :-
    goal_state(State).
bfs([Path|Paths], Solution) :-
    extend(Path, NewPaths),
    append(Paths, NewPaths, UpdatedPaths),
    bfs(UpdatedPaths, Solution).

% Generate new paths by making valid moves.
extend(Path, NewPaths) :-
    path_last(Path, State),
    findall(NewPath, move(State, NewState), NewPathsList),
    maplist(append(Path), NewPathsList, NewPaths).

% Get the last element of the path.
path_last([H|T], Last) :- path_last(T, Last).
path_last([Last], Last).

% Define valid moves.
move(State, NewState) :-
    blank_position(State, BlankIndex),
    adjacent(BlankIndex, AdjIndex),
```

```
    swap(State, BlankIndex, AdjIndex, NewState).

% Find the position of the blank (0).
blank_position(State, Index) :-
    nth0(Index, State, 0).

% Define adjacent positions for the blank.
adjacent(Index, AdjIndex) :-
    (Index = 0, AdjIndex = 1);    % Right
    (Index = 1, AdjIndex = 0);    % Left
    (Index = 1, AdjIndex = 2);    % Right
    (Index = 2, AdjIndex = 1);    % Left
    (Index = 3, AdjIndex = 4);    % Right
    (Index = 4, AdjIndex = 3);    % Left
    (Index = 4, AdjIndex = 5);    % Right
    (Index = 5, AdjIndex = 4);    % Left
    (Index = 6, AdjIndex = 7);    % Right
    (Index = 7, AdjIndex = 6);    % Left
    (Index = 7, AdjIndex = 8);    % Right
    (Index = 8, AdjIndex = 7).    % Left

% Swap the blank position with the adjacent position.
swap(State, BlankIndex, AdjIndex, NewState) :-
    nth0(BlankIndex, State, Blank),
    nth0(AdjIndex, State, Adj),
    Blank \= 0,
    replace(State, BlankIndex, Adj, TempState),
    replace(TempState, AdjIndex, Blank, NewState).

% Replace an element in the list at a specific index.
replace([_|T], 0, X, [X|T]).
replace([H|T], Index, X, [H|R]) :-
    Index > 0,
    NewIndex is Index - 1,
    replace(T, NewIndex, X, R).
```

Output Screenshot:



Practical No.15

Aim: Write a program to solve traveling salesman problems using Prolog.

Program Code:

```
:- discontiguous tsp/3.

% Define distances between cities
distance(new_york, chicago, 790).
distance(new_york, boston, 215).
distance(chicago, boston, 860).
distance(chicago, san_francisco, 1850).
distance(boston, san_francisco, 2700).
distance(san_francisco, new_york, 2500).

% Nearest neighbor algorithm
tsp(Cities, Path, TotalDistance) :-
    Cities = [Start|_],
    nearest_neighbor(Start, Cities, [Start], Path, TotalDistance).

% Base case: if there is only one city left, return it with distance 0
nearest_neighbor(_, [], Path, Path, 0).
nearest_neighbor(CurrentCity, Cities, Visited, Path, TotalDistance) :-
    Cities \= [],
    find_nearest_city(CurrentCity, Cities, NearestCity, Distance),
    select(NearestCity, Cities, RemainingCities), % Remove the nearest city from remaining cities
    nearest_neighbor(NearestCity, RemainingCities, [NearestCity|Visited], Path,
RemainingDistance),
    TotalDistance is Distance + RemainingDistance.

% Find the nearest city to the current city
find_nearest_city(City, Cities, NearestCity, Distance) :-
    member(NearestCity, Cities),
    distance(City, NearestCity, Distance),
    \+ (member(OtherCity, Cities),
        distance(City, OtherCity, OtherDistance),
        OtherDistance < Distance).
```

```
% Example usage
solve_tsp :-
    Cities = [new_york, chicago, boston, san_francisco],
    tsp(Cities, Path, TotalDistance),
    write('Optimal Path: '), write(Path), nl,
    write('Total Distance: '), write(TotalDistance), nl.
```

OutputScreenshot:

Practical No.16

Aim: Write a program to implement perceptron for AND gate

Program Code:

```
import numpy as np

class Perceptron:
    def_init_(self, input_size, learning_rate=0.1):
        # Initialize weights and bias
        self.weights = np.zeros(input_size)
        self.bias = 0
        self.learning_rate = learning_rate

    def activation_function(self, x):
        # Activation function (Step function)
        return 1 if x >= 0 else 0

    def predict(self, inputs):
        # Calculate the weighted sum
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return self.activation_function(weighted_sum)

    def train(self, training_inputs, labels, epochs):
        for _ in range(epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs)
                # Update weights and bias based on the error
                error = label - prediction
                self.weights += self.learning_rate * error * inputs
                self.bias += self.learning_rate * error

# Define the training data for AND gate
training_inputs = np.array([
    [0, 0],
    [0, 1],
    [1, 0],
    [1, 1]
```

```
])

labels = np.array([0, 0, 0, 1]) # Corresponding outputs for AND gate

# Create a perceptron and train it
perceptron = Perceptron(input_size=2)
perceptron.train(training_inputs, labels, epochs=10)

# Test the perceptron
print("AND Gate Predictions:")
for inputs in training_inputs:
    prediction = perceptron.predict(inputs)
    print(f"Input: {inputs}, Prediction: {prediction}")
```

Output Screenshot:

```
Output

AND Gate Predictions:
Input: [0 0], Prediction: 0
Input: [0 1], Prediction: 0
Input: [1 0], Prediction: 0
Input: [1 1], Prediction: 1

=== Code Execution Successful ===
```