Practical: 1

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

```
def create_board():
    return [[' ' for _ in range(3)] for _ in range(3)]
def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-' * 5)
def check_win(board, player):
    # Check rows
    for row in board:
        if all([cell == player for cell in row]):
            return True
    # Check columns
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True

    # Check diagonals
    if all([board[i][i] == player for i in range(3)]) or \
       all([board[i][2 - i] == player for i in range(3)]):
        return True

    return False

def check_draw(board):
    return all([cell != ' ' for row in board for cell in row])

def player_move(board, player):
    while True:
        try:
            row = int(input(f"Player {player}, enter the row (0, 1, 2): "))
            col = int(input(f"Player {player}, enter the column (0, 1, 2): "))
```

```
            if board[row][col] == ' ':
                board[row][col] = player
                break
            else:
                print("Cell is already occupied, try again.")
        except (ValueError, IndexError):
            print("Invalid input, please enter row and column as numbers between 0 and 2.")

def tic_tac_toe():
    board = create_board()
    current_player = 'X'

    while True:
        display_board(board)
        player_move(board, current_player)

        if check_win(board, current_player):
            display_board(board)
            print(f"Player {current_player} wins!")
            break

        if check_draw(board):
            display_board(board)
            print("It's a draw!")
            break

        current_player = 'O' if current_player == 'X' else 'X'

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

Output:

```
 | |
-----
 | |
-----
 | |
-----
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 0
X| |
-----
 | |
-----
 | |
-----
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 1
X| |
-----
 |O|
-----
 | |
-----
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 1
X|X|
-----
 |O|
-----
 | |
-----
Player O, enter the row (0, 1, 2): 1
Player O, enter the column (0, 1, 2): 2
X|X|
-----
 |O|O
-----
 | |
-----
Player X, enter the row (0, 1, 2): 0
Player X, enter the column (0, 1, 2): 2
X|X|X
-----
 |O|O
-----
 | |
-----
Player X wins!
2
```

Practical: 2

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

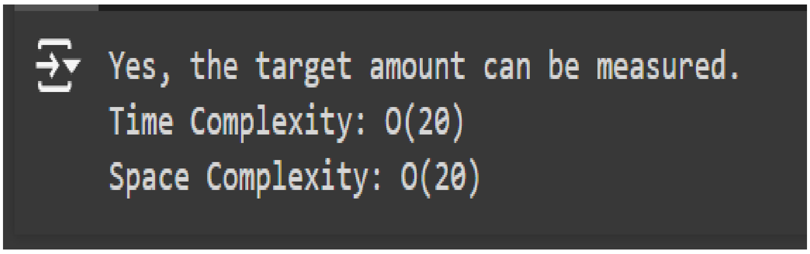
Water Jug Problem Using BFS Algorithm: -

```
from collections import deque

def waterjug(jug1, jug2, target):
    """
    Determines if it's possible to measure exactly `target` liters using two jugs with capacities
`jug1` and `jug2`.
    """
    visited = set()  # Set to store visited states (to avoid cycles)
    queue = deque([(0, 0)])  # Initialize queue with starting state (both jugs empty)

    def pour(j1, j2, from_jug, to_jug):
        """
        Simulates pouring water from one jug to another and returns the resulting state.
        """
        amount = min(from_jug, to_jug - j2)
        return (j1 - amount, j2 + amount)

    while queue:  # Loop until queue is empty (all reachable states explored)
        j1, j2 = queue.popleft()  # Get the next state from the front of the queue (BFS)
        if (j1, j2) in visited:  # If state already visited, skip it
            continue

        visited.add((j1, j2))  # Mark current state as visited

        if j1 == target or j2 == target:  # Check if target amount is reached
            return True  # If yes, return True (target reachable)
```

```
        # Possible operations from current state:
        # 1. Fill jugs completely
        queue.append((jug1, j2))  # Fill jug1 completely
        queue.append((j1, jug2))  # Fill jug2 completely

        # 2. Empty jugs
        queue.append((0, j2))  # Empty jug1
        queue.append((j1, 0))  # Empty jug2

        # 3. Pour water from one jug to another
        queue.append(pour(j1, j2, j1, jug2))  # Pour from jug1 to jug2
        queue.append(pour(j2, j1, j2, jug1))  # Pour from jug2 to jug1

    return False  # If target not found after exploring all states, return False
def print_complexity(jug1, jug2):
    """
    Prints the time and space complexity based on the capacities of the jugs.
    """
    num_states = (jug1 + 1) * (jug2 + 1)
    print(f"Time Complexity: O({num_states})")
    print(f"Space Complexity: O({num_states})")

# Example usage (you can change these values):
jug1_capacity = 3
jug2_capacity = 4
target_amount = 2

if waterjug(jug1_capacity, jug2_capacity, target_amount):
    print("Yes, the target amount can be measured.")
else:
    print("No, the target amount cannot be measured.")

# Print complexity information
```

```
print_complexity(jug1_capacity, jug2_capacity)
```

Output:



Water Jug Problem Using A* Algorithm: -

```
import heapq


def waterjug_astar(jug1, jug2, target): """
    Solves the water jug problem using A* search algorithm.
    """
    def heuristic(state): ""
Estimates the cost to reach the target from the current state. """
        j1, j2 = state

        return abs(j1 - target) + abs(j2 - target)
    start_state = (0, 0)

    open_set = [(heuristic(start_state), start_state)]  # Priority queue (heap)
    came_from = {}  # To reconstruct the path
    cost_so_far = {start_state: 0}


    def pour(j1, j2, from_jug, to_jug): """
        Simulates pouring water from one jug to another.
        """
        amount = min(from_jug, to_jug - j2) return (j1 - amount, j2 + amount)
    while open_set:

        _, current = heapq.heappop(open_set)
```

```
        if current[0] == target or current[1] == target:

            return True, reconstruct_path(came_from, current)
        for neighbor in [

            (jug1, current[1]),  # Fill jug1 (current[0],
            jug2),  # Fill jug2


            (0, current[1]),  # Empty jug1 (current[0], 0),  # Empty jug2
            pour(current[0], current[1], current[0], jug2),  # Pour from jug1 to jug2
            pour(current[1], current[0], current[1], jug1),  # Pour from jug2 to jug1
        ]:

            new_cost = cost_so_far[current] + 1

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + heuristic(neighbor)
                heapq.heappush(open_set, (priority, neighbor))
                came_from[neighbor] = current
    return False, None  # Target not reachable
def reconstruct_path(came_from, current): """
    Reconstructs the path from the start state to the target state.
    """
    total_path = [current]

    while current in came_from: current = came_from[current]
        total_path.insert(0, current)
    return total_path
# Example usage
 jug1_capacity =3
jug2_capacity = 4

target_amount = 2
result, path = waterjug_astar(jug1_capacity, jug2_capacity, target_amount)
```

```
  if result:
      print("Yes, the target amount can be measured.")
      print("Path:", path)
  else:

      print("No, the target amount cannot be measured.")
  # Time and Space Complexity
      print(f"Time Complexity: O({jug1_capacity} *
         {jug2_capacity})") print(f"Space Complexity:
         O({jug1_capacity} * {jug2_capacity})
```

```
Yes, the target amount can be measured.
Path: [(0, 0), (0, 4), (1, 3), (1, 4), (2, 3)]
Time Complexity: O(3 * 4)
Space Complexity: O(3 * 4)
```

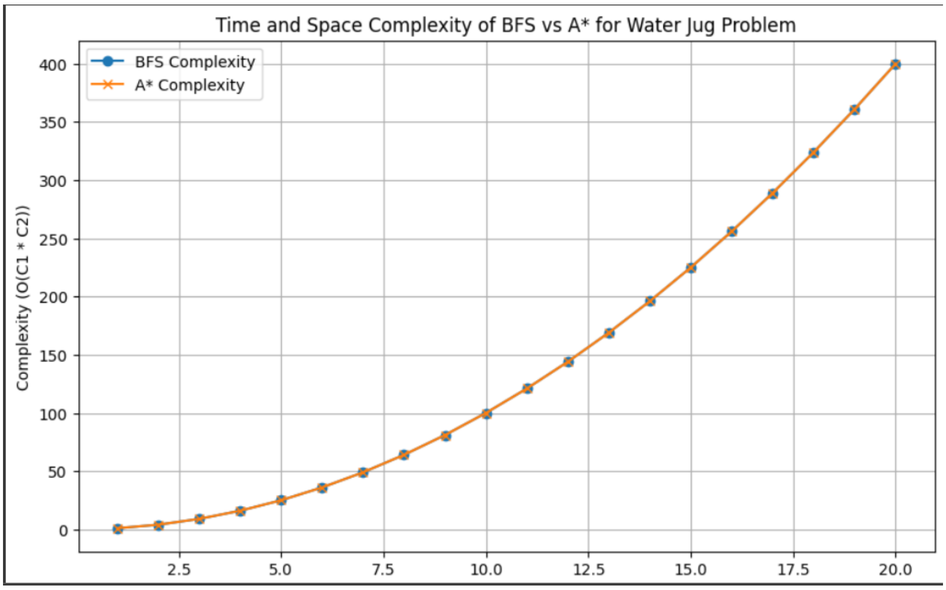
Output:

<u>Compare the BFS Vs A* Algorithm</u>

```
import matplotlib.
pyplot as plt
import numpy as np

# Define the range for jug capacities
capacities = np.arange(1, 21)  # Jug capacities from 1 to 20

# Compute complexities
bfs_complexities = capacities * capacities  # O(C1 * C2) for BFS
astar_complexities = capacities * capacities  # O(C1 * C2) for A*

# Plotting

plt.figure(figsize=(10, 6))

plt.plot(capacities, bfs_complexities, label='BFS Complexity', marker='o')
```

```
plt.plot(capacities, astar_complexities, label='A* Complexity', marker='x')

plt.xlabel('Jug Capacity (C1 or C2)') plt.ylabel('Complexity (O(C1 * C2))')
plt.title('Time and Space Complexity of BFS vs A* for Water Jug Problem')
plt.legend()
plt.grid(True) plt.show()
```

Output:



Practical: 3

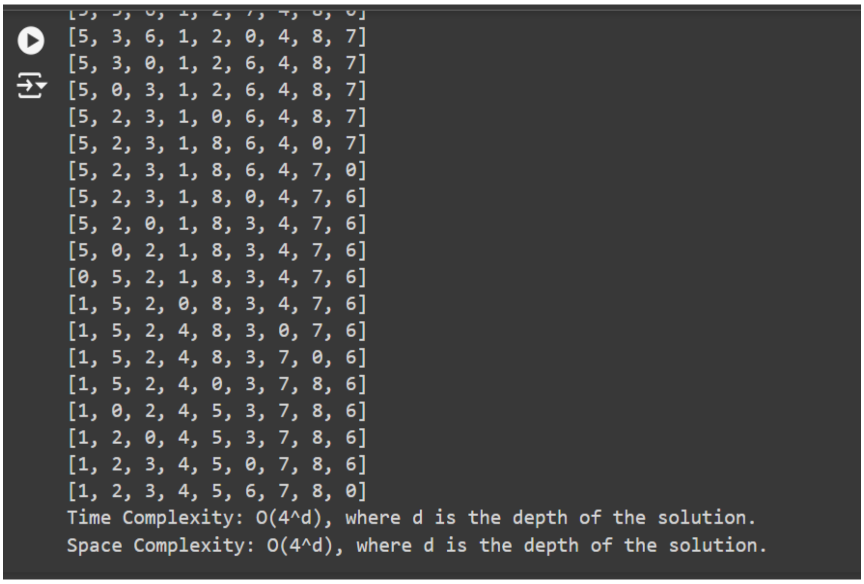
Aim: Write a program to implement DFS (for 8 puzzle problem or Water Jug problem or any AI search problem).

8 Puzzle Problem Using DFS Algorithm: -

```
def dfs(initial_state, goal_state):
    stack = [(initial_state, [])]  # Stack to store states and their paths
    visited = set()

    while stack:
        current_state, path = stack.pop()
        visited.add(tuple(current_state))  # Convert to tuple for hashability


        if current_state == goal_state:
            return path

        empty_tile_index = current_state.index(0)
        row, col = divmod(empty_tile_index, 3)

        # Possible moves (Up, Down, Left, Right)
        moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for dr, dc in moves:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_index = new_row * 3 + new_col
                new_state = current_state[:]
                new_state[empty_tile_index], new_state[new_index] = (
                    new_state[new_index],
                    new_state[empty_tile_index],
                )

                if tuple(new_state) not in visited:
                    stack.append((new_state, path + [new_state]))
    return None  # No solution found
```

```
# Example usage
initial_state = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
solution_path = dfs(initial_state, goal_state)
if solution_path:
    print("Solution found!")
    for state in solution_path:
        print(state)

else:
    print("No solution found.")


# Print time and space complexity
from math import factorial

num_states = factorial(9)  # Total number of possible states (9!)
print(f"Time Complexity: O(4^d), where d is the depth of the solution.")
print(f"Space Complexity: O(4^d), where d is the depth of the solution.")
```

Output:



```
[5, 3, 6, 1, 2, 0, 4, 8, 7]
[5, 3, 0, 1, 2, 6, 4, 8, 7]
[5, 0, 3, 1, 2, 6, 4, 8, 7]
[5, 2, 3, 1, 0, 6, 4, 8, 7]
[5, 2, 3, 1, 8, 6, 4, 0, 7]
[5, 2, 3, 1, 8, 6, 4, 7, 0]
[5, 2, 3, 1, 8, 0, 4, 7, 6]
[5, 2, 0, 1, 8, 3, 4, 7, 6]
[5, 0, 2, 1, 8, 3, 4, 7, 6]
[0, 5, 2, 1, 8, 3, 4, 7, 6]
[1, 5, 2, 0, 8, 3, 4, 7, 6]
[1, 5, 2, 4, 8, 3, 0, 7, 6]
[1, 5, 2, 4, 8, 3, 7, 0, 6]
[1, 5, 2, 4, 0, 3, 7, 8, 6]
[1, 0, 2, 4, 5, 3, 7, 8, 6]
[1, 2, 0, 4, 5, 3, 7, 8, 6]
[1, 2, 3, 4, 5, 0, 7, 8, 6]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
Time Complexity: O(4^d), where d is the depth of the solution.
Space Complexity: O(4^d), where d is the depth of the solution.
```

8 Puzzle Problem Using A* Algorithm: -

```
import heapq

def manhattan_distance(state, goal_state):
    """Calculates the Manhattan distance heuristic."""
    distance = 0
    for i in range(9):
        if state[i] != 0:
            goal_row, goal_col = divmod(goal_state.index(state[i]), 3)
            current_row, current_col = divmod(i, 3)
            distance += abs(goal_row - current_row) + abs(goal_col - current_col)
    return distance

def astar(initial_state, goal_state):
    """Solves the 8-puzzle problem using A* search."""
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(initial_state, goal_state), initial_state, []))
    closed_set = set()

    while open_set:
        _, current_state, path = heapq.heappop(open_set)
        if tuple(current_state) in closed_set:
            continue
        closed_set.add(tuple(current_state))
        if current_state == goal_state:
            return path


        empty_tile_index = current_state.index(0)
        row, col = divmod(empty_tile_index, 3)

        moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for dr, dc in moves:
```

```
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_index = new_row * 3 + new_col
                new_state = current_state[:]
                new_state[empty_tile_index], new_state[new_index] = new_state[new_index],
new_state[empty_tile_index]

                if tuple(new_state) not in closed_set:
                    cost = len(path) + 1 + manhattan_distance(new_state, goal_state)
                    heapq.heappush(open_set, (cost, new_state, path + [new_state]))

    return None

# Example usage
initial_state = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
solution_path = astar(initial_state, goal_state)
if solution_path:
    print("Solution found!")
    for state in solution_path:
        print(state)
else:
    print("No solution found.")
# Print time and space complexity
print("Time Complexity: O(b^d), where b is the branching factor and d is the depth of the
solution.")

print("Space Complexity: O(b^d), where b is the branching factor and d is the depth
ofthesolution."
```

Output

```
Solution found!
[1, 2, 3, 4, 0, 6, 7, 5, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
Time Complexity: O(b^d), where b is the branching factor and d is the depth of the solution.
Space Complexity: O(b^d), where b is the branching factor and d is the depth of the solution.
```

Compare the BFS Vs A* Algorithm

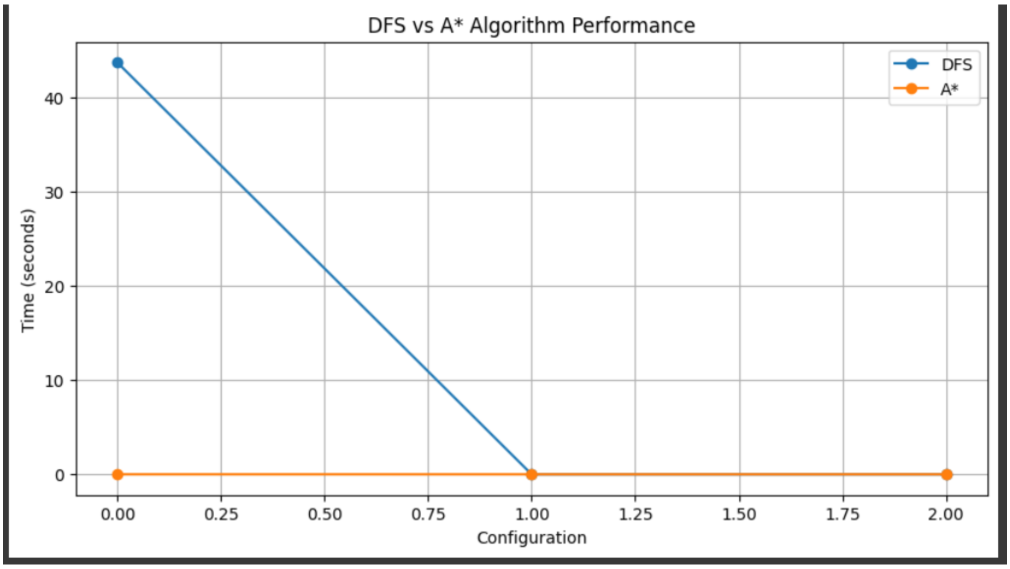
```
import matplotlib.pyplot as plt
import time
import heapq

def dfs(initial_state, goal_state):
    stack = [(initial_state, [])]  # Stack to store states and their paths
    visited = set()
    start_time = time.time()

    while stack:
        current_state, path = stack.pop()
        visited.add(tuple(current_state))  # Convert to tuple for hashability

        if current_state == goal_state:
            end_time = time.time()
            return path, end_time - start_time

        empty_tile_index = current_state.index(0)
        row, col = divmod(empty_tile_index, 3)

        # Possible moves (Up, Down, Left, Right)
        moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for dr, dc in moves:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_index = new_row * 3 + new_col
                new_state = current_state[:]
                new_state[empty_tile_index], new_state[new_index] = (
                    new_state[new_index],
                    new_state[empty_tile_index],)
                if tuple(new_state) not in visited:
```

```
                stack.append((new_state, path + [new_state]))

    end_time = time.time()
    return None, end_time - start_time

 def manhattan_distance(state, goal_state):
    """Calculates the Manhattan distance
    heuristic.""" distance = 0
    for i in
        range(9)
        : if
        state[i]
        != 0:
            goal_row, goal_col = divmod(goal_state.index(state[i]), 3)
            current_row, current_col = divmod(i, 3)
            distance += abs(goal_row - current_row) + abs(goal_col -
    current_col) return distance


 def astar(initial_state, goal_state):
    """Solves the 8-puzzle problem using A*
    search.""" open_set = []
    heapq.heappush(open_set, (manhattan_distance(initial_state, goal_state), initial_state, []))
    closed_set = set()
    start_time = time.time()

    while open_set:
        _, current_state, path =
        heapq.heappop(open_set) if
        tuple(current_state) in closed_set:
            continue
        closed_set.add(tuple(current_state)
        )
        if current_state ==
            goal_state: end_time =
```

```
            time.time()
            return path, end_time - start_time

        empty_tile_index =
        current_state.index(0) row, col =
        divmod(empty_tile_index, 3)

        moves = [(0, -1), (0, 1), (-1, 0), (1, 0)]

        for dr, dc in moves:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col <
                3: new_index = new_row * 3 +
                new_col new_state = current_state[:]
                new_state[empty_tile_index], new_state[new_index] = new_state[new_index],
new_state[empty_tile_index]

                if tuple(new_state) not in closed_set:
                    cost = len(path) + 1 + manhattan_distance(new_state, goal_state)
                    heapq.heappush(open_set, (cost, new_state, path + [new_state]))

    end_time = time.time()
    return None, end_time - start_time

# Define different puzzle configurations for
comparison configurations = [
    ([1, 2, 3, 0, 4, 5, 6, 7, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]),
    ([1, 2, 3, 4, 5, 6, 7, 8, 0], [1, 2, 3, 4, 5, 6, 7, 8, 0]),
    ([1, 2, 0, 4, 5, 3, 7, 8, 6], [1, 2, 3, 4, 5, 6, 7, 8, 0])]
dfs_times = []
astar_times = []

    for initial, goal in configurations:
    _, dfs_time = dfs(initial, goal)
    _, astar_time = astar(initial, goal)
    dfs_times.append(dfs_time)
```

```
    astar_times.append(astar_time)
#                Plotting
plt.figure(figsize=(
10, 5))
plt.plot(range(len(configurations)), dfs_times, label='DFS', marker='o')
plt.plot(range(len(configurations)), astar_times, label='A*', marker='o')
plt.xlabel('Configuration')
plt.ylabel('Time (seconds)')
plt.title('DFS vs A* Algorithm Performance')
plt.legend()
plt.grid(True)
plt.show()
```

Output:



Practical: 4

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

```
import random
print("#------------------ #")

print("| GUESS THE NUMBER |")

print("#------------------ #")

print('\n')

print('Range of Random Numbers.') start = int(input('Enter Starting Index:'))
end = int(input('Enter Ending Index:')) number = random.randint(start, end)
print('\n')
while True:

    guess = int(input('Guess the number: ')) if guess > number:
        print('\nHmmm, try a lower
    number...\n') elif guess < number:
        print('\nGo a little
    higher\n') else:
        print("Right on! Well
        done!") break
```

Output:

```
#-----------------#
| GUESS THE NUMBER |
#-----------------#


Range of Random Numbers.
Enter Starting Index:20
Enter Ending Index:25


Guess the number: 21

Go a little higher

Guess the number: 24
Right on! Well done!
```

Practical: 5

Aim: Write a program to Implement A* Algorithm.

```
Import heapq

 # Define the heuristic function: Manhattan Distance for

 grids def heuristic(a, b):

     return abs(a[0] - b[0]) + abs(a[1] - b[1])

 # A* Algorithm

 implementation def

 astar(grid, start, goal):

     rows, cols = len(grid), len(grid[0])

     # Priority queue: stores (priority, node) open_set =

     [] heapq.heappush(open_set, (0, start))

     # Cost from start to current

     node g_score = {start: 0}

     # Estimated cost from current node to goal (f = g + h)

     f_score = {start: heuristic(start, goal)}

     # Keep track of the path came_from = {}

     while open_set:

         # Get the node with the lowest f_score
         current = heapq.heappop(open_set)[1]


         # If we have reached the goal, reconstruct and return

         the path if current == goal:

             path = []

             while current in

                 came_from:
```

```
            path.append(current)

            current = came_from[current]

          return path[::-1]  # Return reversed path

        # Explore neighbors

        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:

          neighbor = (current[0] + dx, current[1] + dy)

          # Ensure the neighbor is within bounds and walkable

          if 0 <= neighbor[0] < rows and 0 <= neighbor[1] < cols and
    grid[neighbor[0]][neighbor[1]] == 0:

            tentative_g_score = g_score[current] + 1  # Assuming each move has a cost of 1
            if neighbor not in g_score or tentative_g_score <

              g_score[neighbor]: # Record this path as the best so far

              came_from[neighbor] = current g_score[neighbor] = tentative_g_score

              f_score[neighbor] = tentative_g_score + heuristic(neighbor,

              goal) # Add the neighbor to the open set

              heapq.heappush(open_set, (f_score[neighbor], neighbor))
      # If there's no

      path return

      None

    # Example

    usage: grid =

    [

      [0, 1, 0, 0, 0],

      [0, 1, 0, 1, 0],

      [0, 0, 0, 1, 0],

      [0, 1, 1, 1, 0],

      [0, 0, 0, 0, 0]
```

```
]

start = (0, 0)  #
Starting point goal =
(4, 4)  # Goal point
path = astar(grid, start, goal)

if path:
    print("Path found:",
path) else:
    print("No path found.")
```

Output:

```
Path found: [(1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
```

Practical: 6

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

```
import math

# Function implementing the Minimax algorithm

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):

    # Base case: if the target depth is reached, return the score at this

    node if curDepth == targetDepth:

        return scores[nodeIndex]

    # If it's the maximizing

    player's turn if maxTurn:

        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),

                   minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))

    else:

        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),

                   minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))

scores = [3, 5, 2, 9, 12, 5, 23, 23]  # The leaf node values (outcomes of the

game) treeDepth = int(math.log(len(scores), 2))   # Calculate the depth of

the tree Output the result

print("The optimal value is : ", end="")

print(minimax(0, 0, True, scores, treeDepth))
```

Output:

```
The optimal value is : 12
```

Practical: 7

Aim: Assume given a set of facts of the form father (name1, name2) (name1 is the father of name2).

```
% Facts
father(ram, ajay).
father(ajay, rahul).
father(ajay, deepa).
father(ram, sita).
father(gopal, ram).

mother(sita, rahul).
mother(sita, deepa).
mother(hema, ajay).
mother(rahini, ram).

% Define gender
male(ram).
male(ajay).
male(rahul).
male(gopal).
female(deepa).
female(sita).
female(hema).
female(rahini).

% Rules

% A parent can be either a father or a mother
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).

% Siblings share at least one parent
sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
```

```
% Brother and sister relationships
brother(X, Y) :- sibling(X, Y), male(X).
sister(X, Y) :- sibling(X, Y), female(X).

% Uncle and aunt relationships
uncle(X, Y) :- brother(X, Z), parent(Z, Y).
aunt(X, Y) :- sister(X, Z), parent(Z, Y).

% Grandparent relationships
grandfather(X, Y) :- father(X, Z), parent(Z, Y).
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

Output:

```
?-
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/practical7.pl compiled 0.00 sec, 26 clauses
?- father(Father, ajay).
Father = ram.

?- parent(ram, Child).
Child = ajay ,

?- sibling(Sibling, rahul).
Sibling = deepa ,

?- brother(Brother, deepa).
Brother = rahul ,

?- uncle(Uncle, rahul).
Uncle = ajay ,

?- grandfather(Grandfather, rahul).
Grandfather = ram ,

?- grandmother(Grandmother, rahul).
Grandmother = hema ,

?- mother(Mother, rahul).
Mother = sita.

?- █
```

Practical: 8

Aim: Define a predicate brother(X,Y) which holds iff X and Y are brothers. Define a predicate cousin(X,Y) which holds iff X and Y are cousins. Define a predicate grandson(X,Y) which holds iff X is a grandson of Y. Define a predicate descendent(X,Y) which holds iff X is a descendent of Y. Consider the following genealogical tree: father(a,b). father(a,c). father(b,d). father(b,e). father(c,f). Say which answers, and in which order, are generated by your definitions for the following queries in Prolog: ?- brother(X,Y). ?- cousin(X,Y). ?- grandson(X,Y). ?- descendent(X,Y).

```
father(a,b).
father(a,c).
father(b,d).
father(b,e).
father(c,f).
brother(X,Y) :- father(Z,X), father(Z,Y), not(X=Y).
cousin(X,Y) :- father(Z,X), father(W,Y), brother(Z,W).
grandson(X,Y) :- father(Z,X), father(Y,Z).
descendent(X,Y) :- father(Y,X).
descendent(X,Y) :- father(Z,X), descendent(Z,Y).
```

Output:

```
?-
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/Pr8.pl compiled 0.00 sec, 10 clauses
?- brother(X,Y).
X = b,
Y = c ,

?- cousin(X,Y).
X = d,
Y = f ,

?- grandson(X,Y).
X = d,
Y = a ,

?- decendant(X,Y).
ERROR: Unknown procedure: decendant/2 (DWIM could not correct goal)
?- descendent(X,Y).
X = b,
Y = a
```

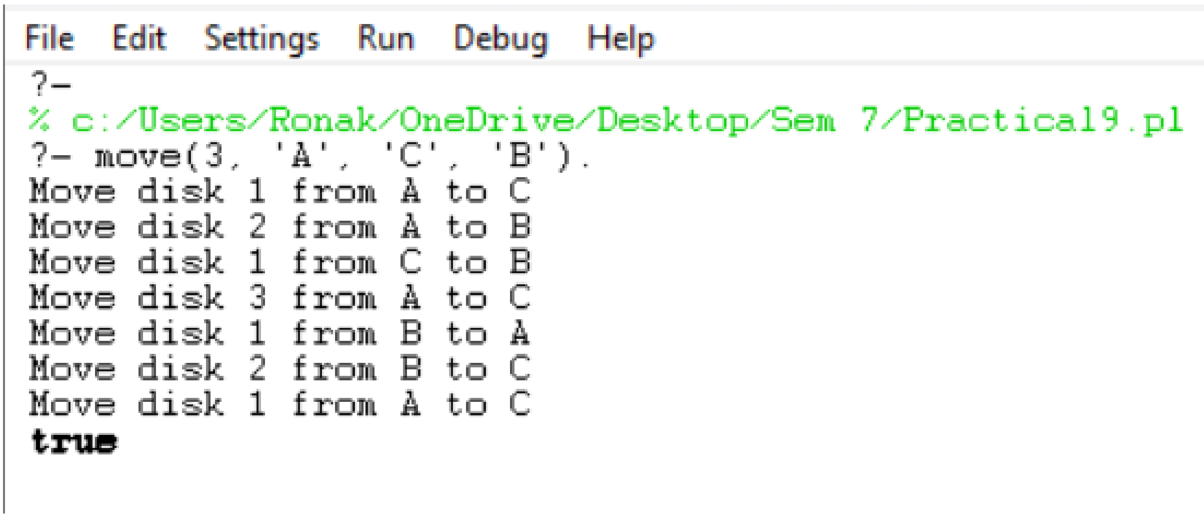
Practical: 9

Aim: Write a program to solve Tower of Hanoi problem using Prolog.

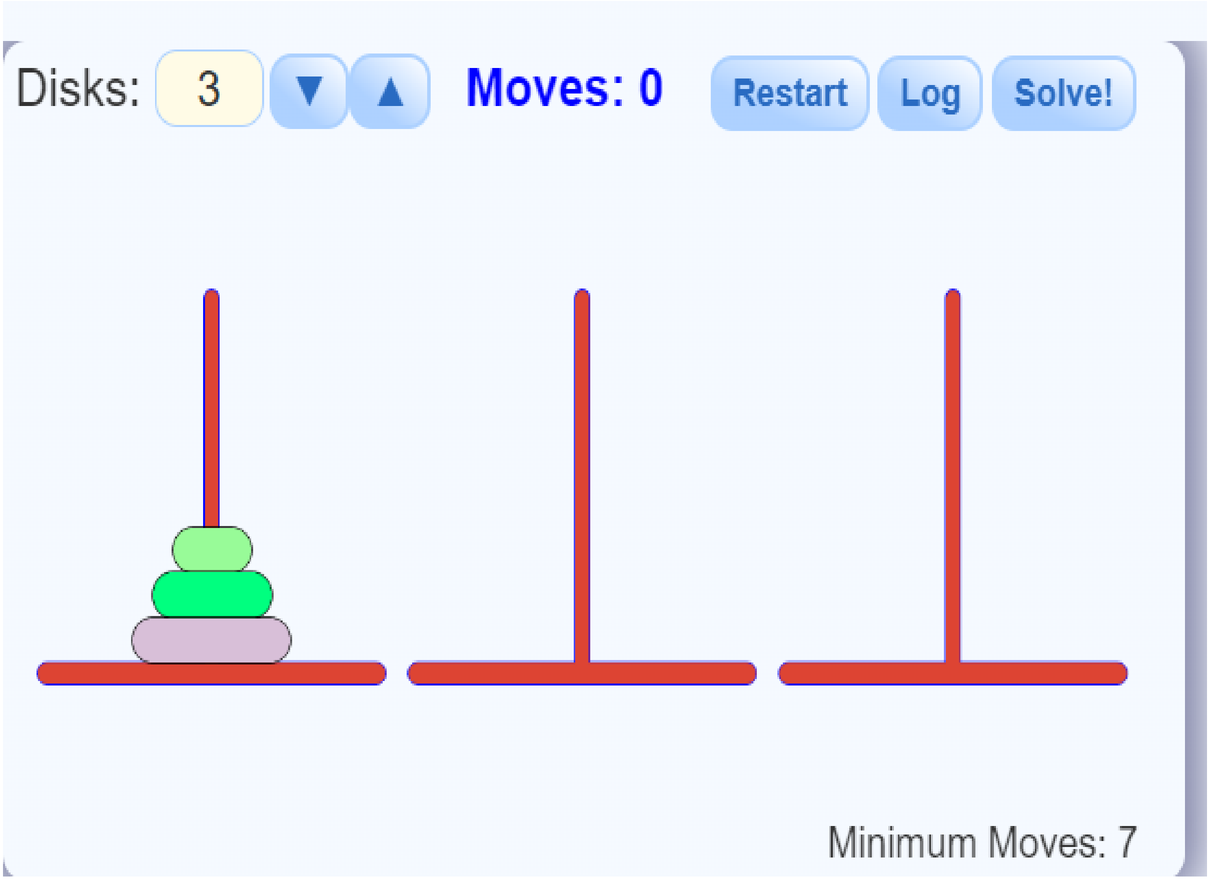
```
% Tower of Hanoi solver in Prolog with disk size names


% move/4: Solves the Tower of Hanoi puzzle
move(1, Source, Destination, _) :-
    format('Move smallest disk from ~w to ~w~n', [Source, Destination]).
move(N, Source, Destination, Aux) :-
    N > 1,
    M is N - 1,
    move(M, Source, Aux, Destination),      % Move smaller disks from Source to Aux
    format('Move disk ~w from ~w to ~w~n', [N, Source, Destination]),  % Move the largest
        disk
    move(M, Aux, Destination, Source).      % Move smaller disks from Aux to Destination


% To solve the Tower of Hanoi for N disks:
% ?- move(N, 'Source', 'Destination', 'Auxiliary').
```

Output:

- **3 Disk Problem**



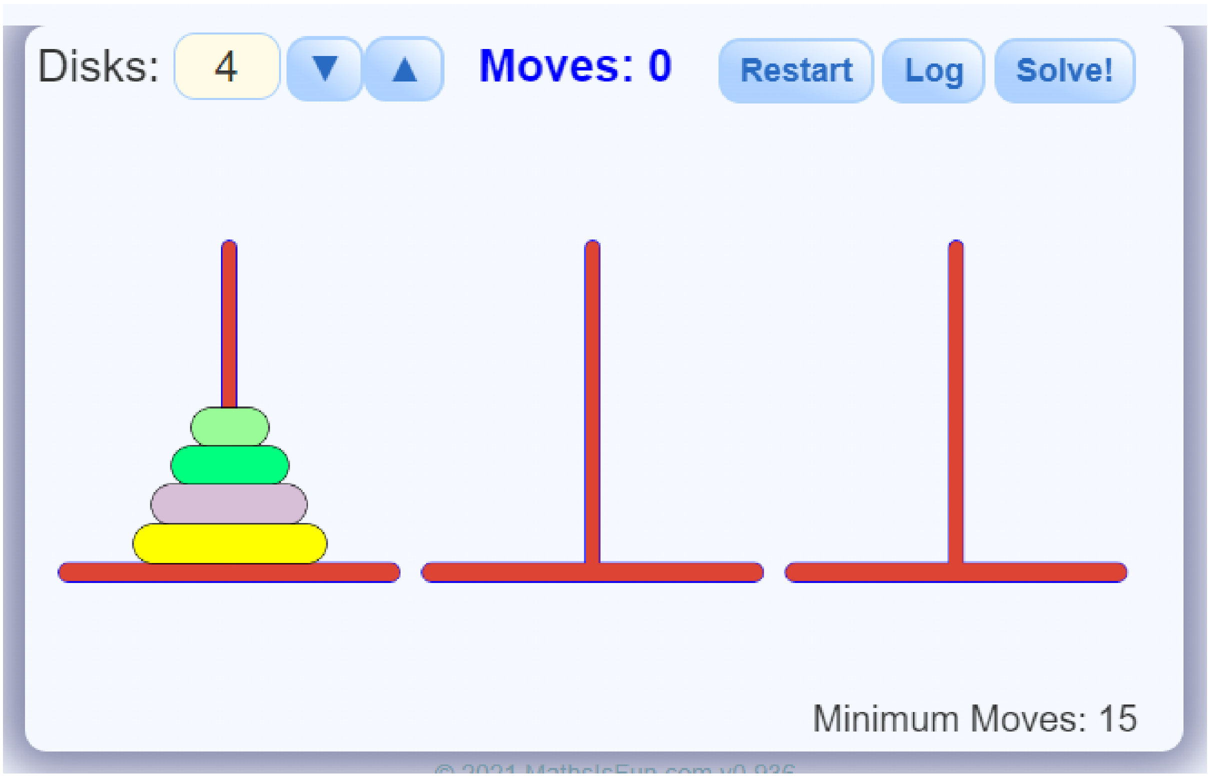
```
File   Edit   Settings   Run   Debug   Help
?-
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/Practical9.pl
?- move(3, 'A', 'C', 'B').
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
true
```


Disks: 3
Moves: 0
Restart
Log
Solve!
Minimum Moves: 7


Disks: 3
Moves: 11
Restart
Log
Solve!
Well Done !
Minimum Moves: 7
© 2021 MathsIsFun.com v0.936

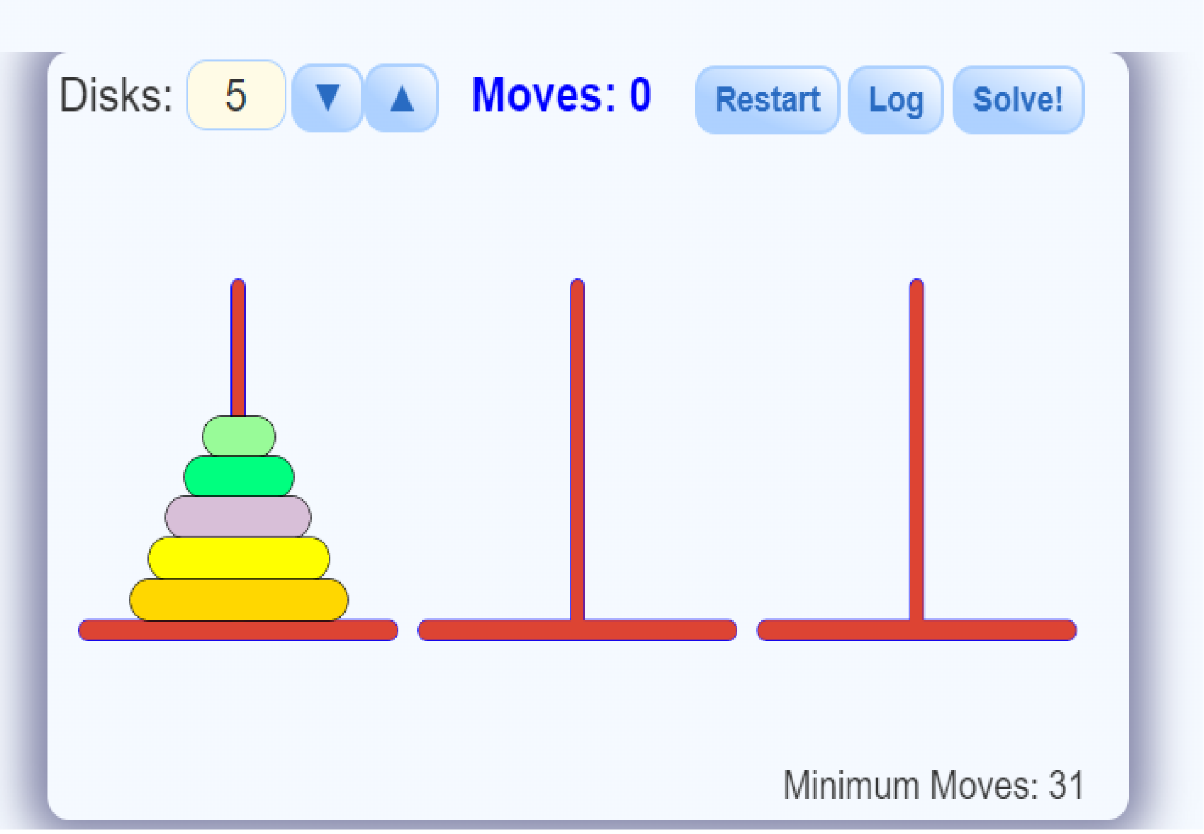
- **4 Disk Problem**

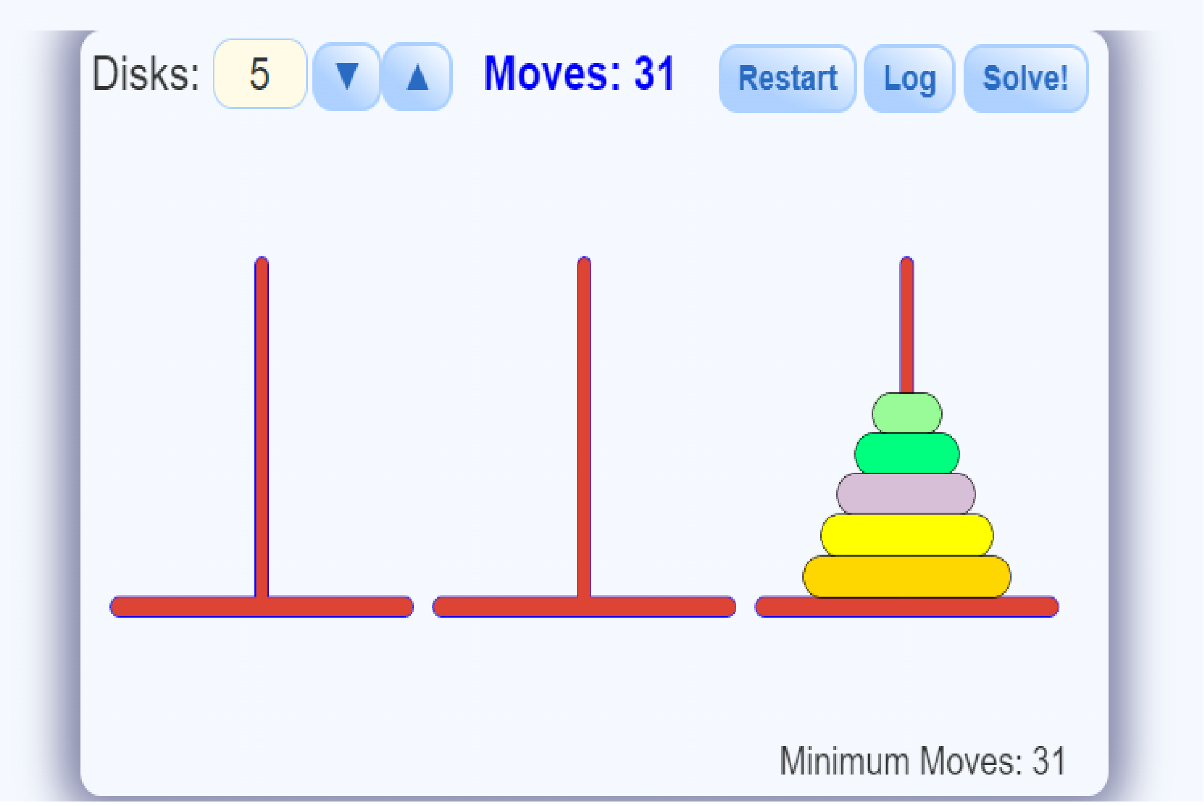
```
?-
% c:/Users/Ronak/OneDrive/Desktop
?- move(4, 'A', 'C', 'B').
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
Move disk 3 from A to B
Move disk 1 from C to A
Move disk 2 from C to B
Move disk 1 from A to B
Move disk 4 from A to C
Move disk 1 from B to C
Move disk 2 from B to A
Move disk 1 from C to A
Move disk 3 from B to C
Move disk 1 from A to B
Move disk 2 from A to C
Move disk 1 from B to C
true
```



- **5 Disc Problem**

```
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/Practical9.pl

Unknown action: A (h for help)
Action? .

?- move(5, 'A', 'C', 'B').
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 3 from C to B
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 5 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 3 from B to A
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A
Move disk 4 from B to C
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
true
```


Disks: 5
Moves: 0
Restart
Log
Solve!
Minimum Moves: 31


Disks: 5
Moves: 31
Restart
Log
Solve!
Minimum Moves: 31

Practical: 10

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

solutions nicely.

```
:- use_rendering(chess).

%  queens(+N, -Queens) is nondet.

%	@param	Queens is a list of column numbers for placing the queens.

%	@author Richard A. O'Keefe (The Craft of Prolog)

queens(N, Queens) :-

   length(Queens, N),

	board(Queens, Board, 0, N, _, _),

	queens(Board, 0, Queens).


board([], [], N, N, _, _).

board([_|Queens], [Col-Vars|Board], Col0, N, [_|VR], VC) :-

	Col is Col0+1,

	functor(Vars, f, N),

	constraints(N, Vars, VR, VC),

	board(Queens, Board, Col, N, VR, [_|VC]).


constraints(0, _, _, _) :- !.

constraints(N, Row, [R|Rs], [C|Cs]) :-

	arg(N, Row, R-C),

	M is N-1,

	constraints(M, Row, Rs, Cs).
```

```
queens([], _, []).

queens([C|Cs], Row0, [Col|Solution]) :-

        Row is Row0+1,

        select(Col-Vars, [C|Cs], Board),

        arg(Row, Vars, Row-Row),

        queens(Board, Row, Solution).
```

Output:

```
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/AI/p10.pl compiled
?- queens(8,Queens).
Queens = [1, 5, 8, 6, 3, 7, 2, 4]
```

Practical: 11

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

```
test(Plan):-
    write('Initial state:'), nl,
    Init = [at(tile4,1), at(tile3,2), at(tile8,3), at(empty,4), at(tile2,5), at(tile6,6), at(tile5,7),
            at(tile1,8), at(tile7,9)],
    write_sol(Init),
    Goal = [at(tile1,1), at(tile2,2), at(tile3,3), at(tile4,4), at(empty,5), at(tile5,6), at(tile6,7),
            at(tile7,8), at(tile8,9)],
    nl, write('Goal state:'), nl,
    write(Goal), nl, nl,
    solve(Init, Goal, Plan).

solve(State, Goal, Plan):-
    solve(State, Goal, [], Plan).

% Determines whether Current and Destination tiles are a valid move.
is_movable(X1,Y1) :-
    (1 is X1 - Y1) ; (-1 is X1 - Y1) ; (3 is X1 - Y1) ; (-3 is X1 - Y1).

/* This predicate produces the plan. Once the Goal list is a subset of the current State the plan
is complete and it is written to the screen using write_sol */
solve(State, Goal, Plan, Plan):-
    is_subset(Goal, State), nl,
    write_sol(Plan).

solve(State, Goal, Sofar, Plan):-
    act(Action, Preconditions, Delete, Add),
    is_subset(Preconditions, State),
    \+ member(Action, Sofar),
    delete_list(Delete, State, Remainder),
    append(Add, Remainder, NewState),
    solve(NewState, Goal, [Action|Sofar], Plan).
```

```
/* The problem has three operators.
 1st arg = name
 2nd arg = preconditions
 3rd arg = delete list
 4th arg = add list. */

% Tile can move to new position only if the destination tile is empty & Manhattan distance =
1
act(move(X,Y,Z),
    [at(X,Y), at(empty,Z), is_movable(Y,Z)],
    [at(X,Y), at(empty,Z)],
    [at(X,Z), at(empty,Y)]).

% Utility predicates.

% Check if the first list is a subset of the second
is_subset([H|T], Set):-
    member(H, Set),
    is_subset(T, Set).

is_subset([], _).

% Remove all elements of the 1st list from the second to create the third.
delete_list([H|T], Curstate, Newstate):-
    remove(H, Curstate, Remainder),
    delete_list(T, Remainder, Newstate).

delete_list([], Curstate, Curstate).

remove(X, [X|T], T).
remove(X, [H|T], [H|R]):-
    remove(X, T, R).

write_sol([]).
```

```
write_sol([H|T]):-
    write_sol(T),
    write(H), nl.

append([H|T], L1, [H|L2]):-
    append(T, L1, L2).

append([], L, L).

member(X, [X|_]).
member(X, [_|T]):-
    member(X, T).
```

Output:

```
?-
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/AI/p11.pl compiled 0.00 sec, 18 clauses
?- test(Plan).
Initial state:
at(tile7,9)
at(tile1,8)
at(tile5,7)
at(tile6,6)
at(tile2,5)
at(empty,4)
at(tile8,3)
at(tile3,2)
at(tile4,1)

Goal state:
[at(tile1,1),at(tile2,2),at(tile3,3),at(tile4,4),at(empty,5),at(tile5,6),at(tile6,7),at(tile7,8),at(tile8,9)]

false.

?-
```

Practical: 12

Aim: Write a program to solve travelling salesman problem using Prolog.

Code:

```
% Define the distances between the cities
distance(a, b, 10).
distance(a, c, 15).
distance(a, d, 20).
distance(b, c, 35).
distance(b, d, 25).
distance(c, d, 30).
distance(b, a, 10). % assuming bidirectional
distance(c, a, 15).
distance(d, a, 20).
distance(c, b, 35).
distance(d, b, 25).
distance(d, c, 30).

% Calculate the total distance for a given route
total_distance([_], 0). % Base case for single city
total_distance([City1, City2 | Rest], Distance) :-
    distance(City1, City2, D1),
    total_distance([City2 | Rest], DRest),
    Distance is D1 + DRest.

% Generate all permutations of the cities
permutation([], []).
permutation(L, [H|P]) :-
    select(H, L, R),
    permutation(R, P).

% Find the optimal route
tsp([Start|Cities], OptimalRoute, MinDistance) :-
    permutation(Cities, Route),
```

```
    append([Start|Route], [Start], FullRoute), % to return to the starting city
    total_distance(FullRoute, Distance),
    (   var(MinDistance) -> % Initialize if unbound
        MinDistance = Distance,
        OptimalRoute = Route
    ;   Distance < MinDistance ->
        MinDistance = Distance,
        OptimalRoute = Route
    ).

% Main predicate to find TSP solution
solve_tsp(OptimalRoute, MinDistance) :-
    findall(City, distance(City, _, _), Cities),
    list_to_set(Cities, UniqueCities),
    UniqueCities = [Start|Rest],
    tsp([Start|Rest], OptimalRoute, MinDistance).
```

Output:

```
?-
% c:/Users/Ronak/OneDrive/Desktop/Sem 7/AI/p12.pl compiled 0.00
?- solve_tsp(Route,Distance).
Route = [b, c, d],
Distance = 95
```