VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Disha H Jain (1BM23CS095)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

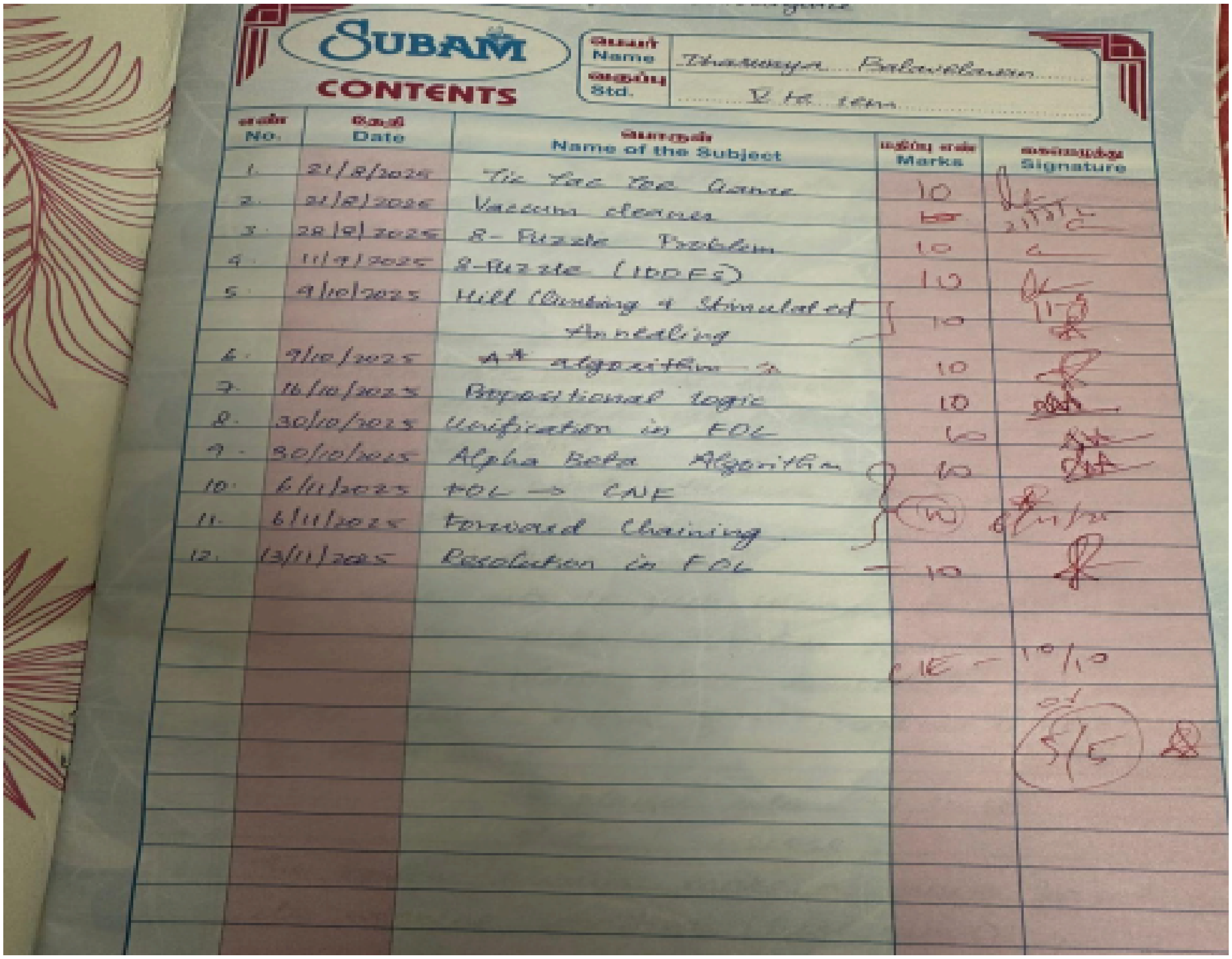
CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by Disha H Jain **(1BM23CS095),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Sheetal V A
Assistant Professor
Department of CSE, BMSCE
Dr. Joythi S Nayak
Professor & HOD
Department of CSE, BMSCE

Index

Github Link:
https://github.com/dishahjain/AI



SUBAM CONTENTS

Name: Dharunya Balachandran
Std.: V th sem

No.	Date	Name of the Subject	Marks	Signature
1.	21/8/2025	Tic Tac Toe Game	10	
2.	21/8/2025	Vaccum cleaner	10	
3.	28/8/2025	8-Puzzle Problem	10	
4.	11/9/2025	8-Puzzle (IDDFS)	10	
5.	9/10/2025	Hill Climbing & Stimulated Annealing	10	
6.	9/10/2025	A* algorithm	10	
7.	16/10/2025	Propositional logic	10	
8.	30/10/2025	Unification in FOL	10	
9.	30/10/2025	Alpha Beta Algorithm	10	
10.	6/11/2025	FOL -> CNF	10	
11.	6/11/2025	Forward Chaining		6/11/25
12.	13/11/2025	Resolution in FOL	10	

CIE - 10/10

5/5

<u>Program 1</u>

Implement Tic –Tac –Toe Game

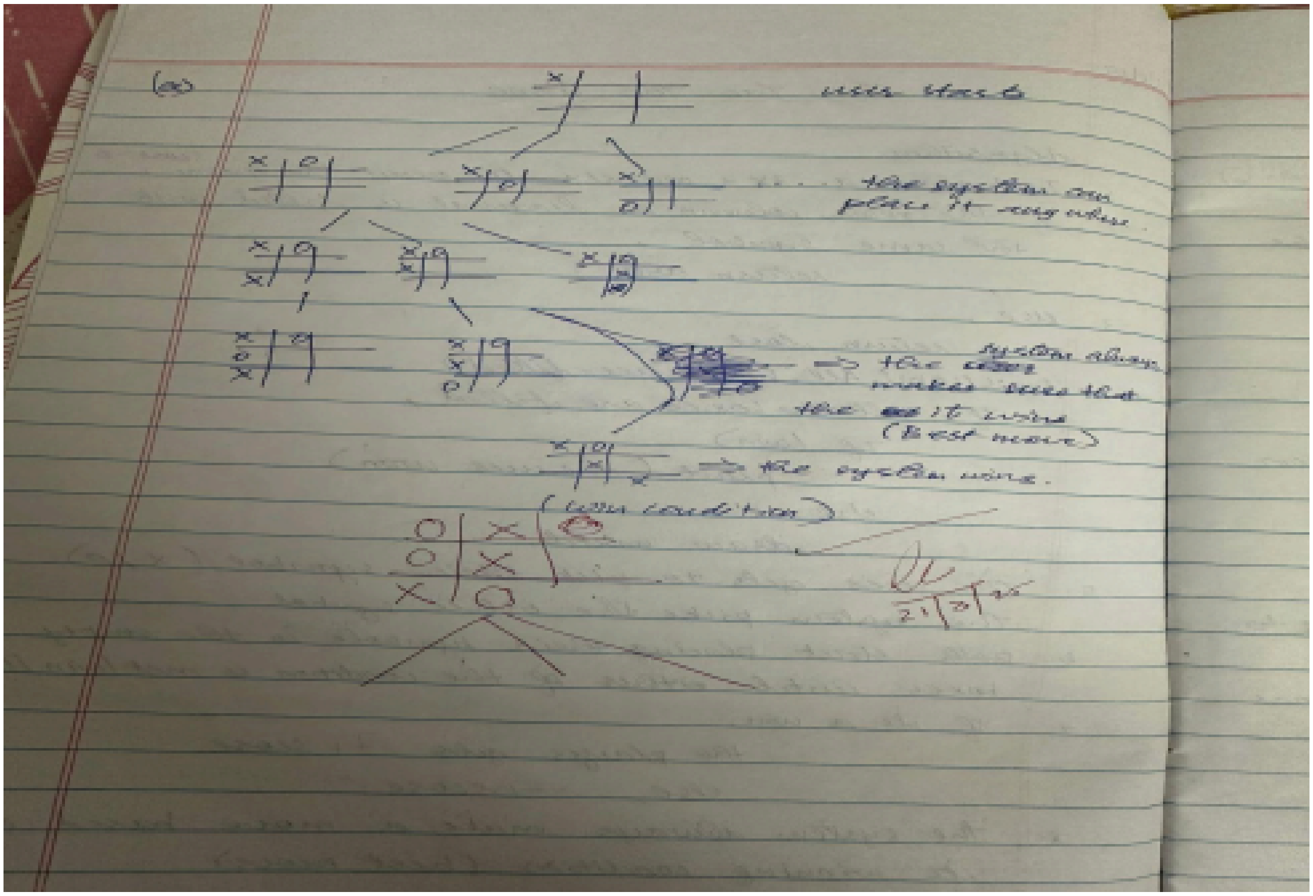
Algorithm:

21/8/25

Tic Tac Toe Game

Algorithm:

1. Initialize a 3x3 matrix to play the game, score = 0
2. If the row, column or diagonal is filled with the same symbol,
   return win
3. else,
   return lost
4. FUNCTION(All boxes are filled)
   If, All the boxes are filled,
   If (win)
   print (the user won)
   else
   draw match
5. The user gets to pick their symbol (X/O). The system picks the other symbol.
6. Both start placing their symbols in the empty boxes until either of the condition is met (win/lose)
7. If its a win,
   the player gets +1 score
   else, -1 score.
8. The system always make a move based on its winning condition (best move)



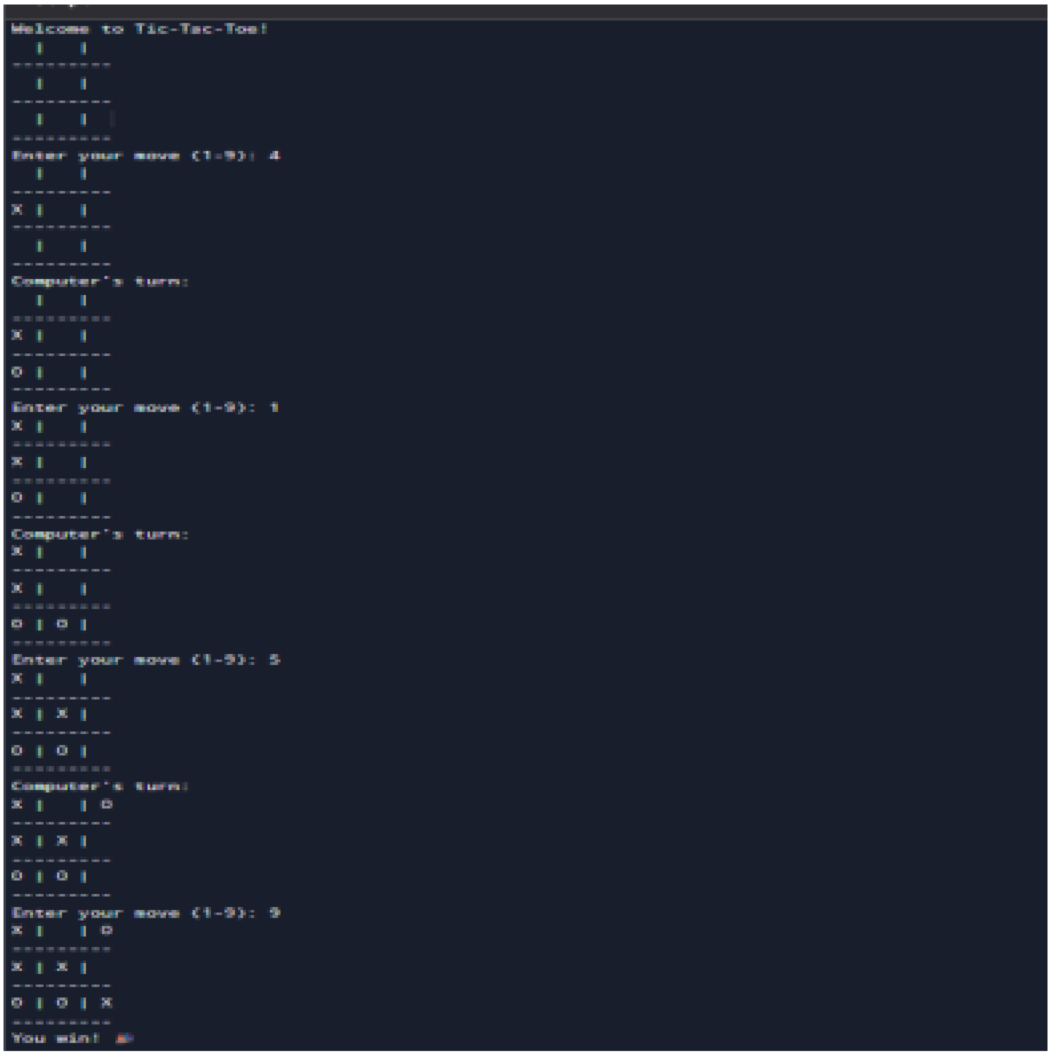
Code:

```
import random
board= [' ' for_ in range(9)]

def print_board():
 print()
 fori in range(3):
 print(""+" | ".join(board[i*3:(i+1)*3]))
 ifi< 2:
 print("---+---+---")
 print()
def check_winner(player):
 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 cond in win_conditions:
 if all(board[i] == player fori in cond):
 returnTrue
 return False
def is_full():
```

```
return all(cell != ' ' for cell in board)
def player_move():
whileTrue:
try:
move= int(input("Enter your move (1-9): "))-1
if move<0 or move >= 9:
print("Invalid move. Choose between 1-9.")
elif board[move] != ' ':
print("That spot is taken.")
else:
board[move]= 'X'
break
exceptValueError:
print("Please entera valid number.")
def ai_move():
empty_spots= [i for i, val in enumerate(board) if val == ' ']
move= random.choice(empty_spots)
board[move]= 'O'
print(f"System placed 'O' in position {move+1}")
def play_game():
print("Welcome toTicTacToe!")
print_board()
whileTrue:
player_move()
print_board()
if check_winner('X'):
print("Congratulations!You win!")
break
if is_full():
print("It'sa tie!")
break
ai_move()
print_board()
if check_winner('O'):
print("System wins. Better luck next time!")
break
if is_full():
print("It'sa tie!")
break
if __name__ == "__main__":
play_game()
```

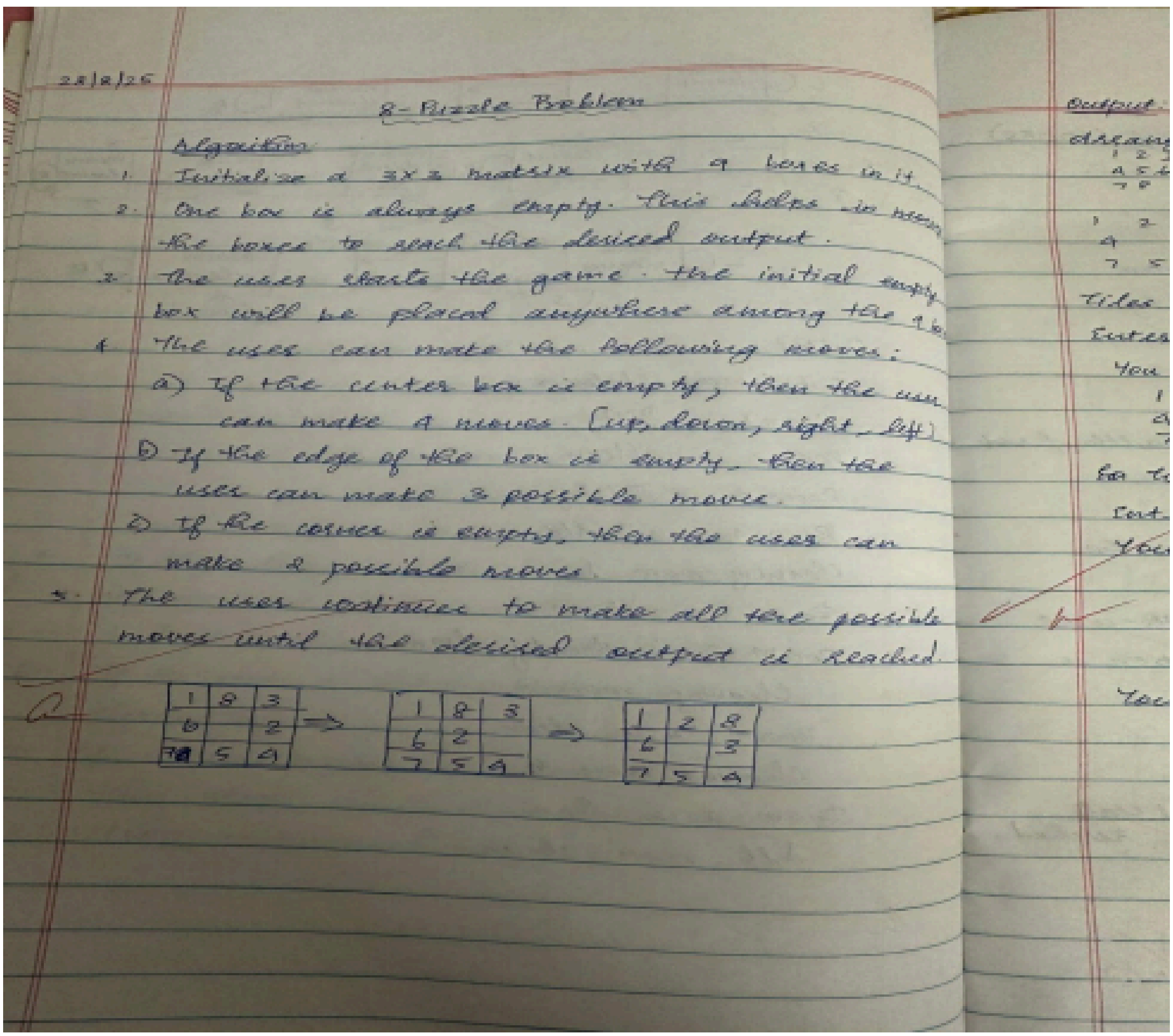
ScreenShots:



```
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (1-9): 4
  |   |
---------
X |   |
---------
  |   |
---------
Computer's turn:
  |   |
---------
X |   |
---------
O |   |
---------
Enter your move (1-9): 1
X |   |
---------
X |   |
---------
O |   |
---------
Computer's turn:
X |   |
---------
X |   |
---------
O | O |
---------
Enter your move (1-9): 5
X |   |
---------
X | X |
---------
O | O |
---------
Computer's turn:
X |   | O
---------
X | X |
---------
O | O |
---------
Enter your move (1-9): 9
X |   | O
---------
X | X |
---------
O | O | X
---------
You win!
```

<u>Program 2:</u>

Solve 8 puzzle problems.
Algorithm:


22/2/25

8-Puzzle Problem

Algorithm

1. Initialise a 3x3 matrix with 9 boxes in it.
2. One box is always empty. This helps in moving the boxes to reach the desired output.
3. The user starts the game. The initial empty box will be placed anywhere among the 9 boxes.
4. The user can make the following moves:
   a) If the center box is empty, then the user can make 4 moves. (up, down, right, left)
   b) If the edge of the box is empty, then the user can make 3 possible moves.
   c) If the corner is empty, then the user can make 2 possible moves.
5. The user continues to make all the possible moves until the desired output is reached.

1	8	3
6		2
7	5	4

→

1	8	3
6	2	
7	5	4

→

1	2	8
6		3
7	5	4

Code:

```
import copy
```

```
def print_board(board):
for row in board:
print(' '.join(str(x) ifx !=0 else ' ' forx in row))
print()
def find_zero(board):
fori in range(3):
forj in range(3):
if board[i][j] == 0:
return i,j
def is_solved(board):
solved= [1,2,3,4,5,6,7,8,0]
flat= [num for row in board for num in row]
```



```
return flat == solved
def valid_moves(zero_pos):
i, j = zero_pos
moves = []
if i > 0: moves.append((i-1, j))
if i < 2: moves.append((i+1, j))
if j > 0: moves.append((i, j-1))
if j < 2: moves.append((i, j+1))
return moves
def correct_tiles_count(board):
"""Count how many tiles are in their correct position."""
count = 0
goal = [1,2,3,4,5,6,7,8,0]
flat = [num for row in board for num in row]
for i in range(9):
if flat[i] != 0 and flat[i] == goal[i]:
count += 1
return count
def get_user_move(board):
zero_pos = find_zero(board)
moves = valid_moves(zero_pos)
movable_tiles = [board[i][j] for (i,j) in moves]
print(f"Tiles you can move: {movable_tiles}")
while True:
try:
move = int(input("Enter the tile number to move (or 0 to quit): "))
if move == 0:
return None
if move in movable_tiles:
return move
else:
print("Invalid tile. Please choose a tile adjacent to the empty space.")  except
ValueError:
```

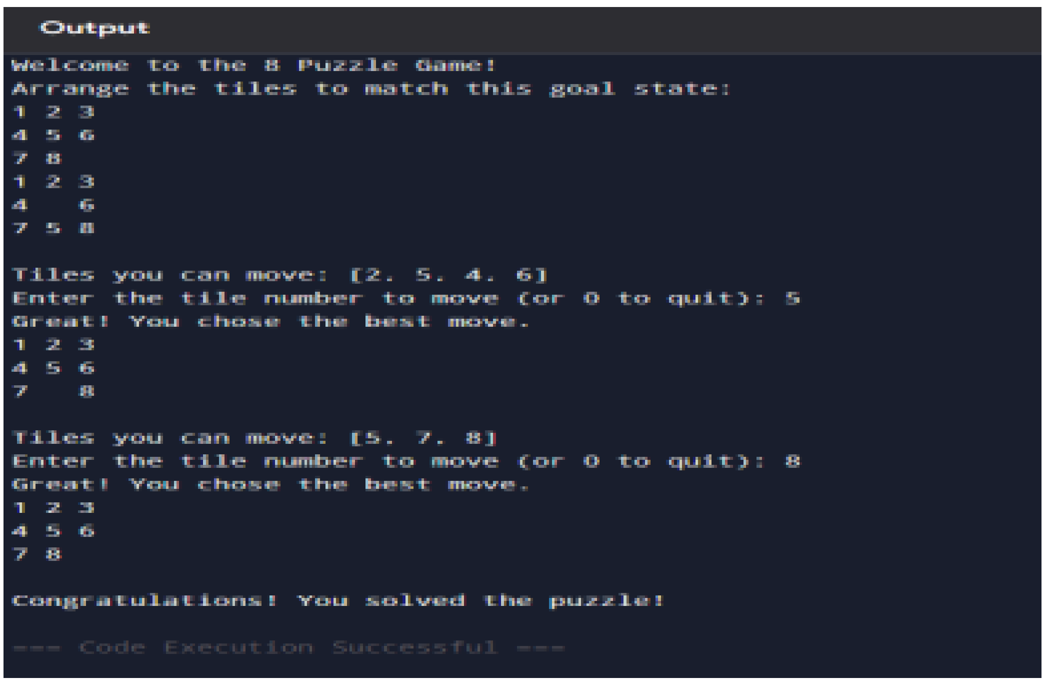
```
print("Please enter a valid number.")
def evaluate_move(board, tile):
"""Compare user move to all possible moves and tell if it's best/worst."""
zero_pos = find_zero(board)
moves = valid_moves(zero_pos)
movable_tiles = [board[i][j] for (i,j) in moves]
scores = {}
for t in movable_tiles:
temp_board = copy.deepcopy(board)
make_move(temp_board, t)
scores[t] = correct_tiles_count(temp_board)
user_score = scores[tile]
best_score = max(scores.values())
worst_score = min(scores.values())
```



```
if user_score == best_score and user_score == worst_score:
return "Your move is the only possible move."
elif user_score == best_score:
return "Great! You chose the best move."
elif user_score == worst_score:
return "Oops! You chose the worst move."
else:
return "Your move is neither the best nor the worst."
def make_move(board, tile):
zero_i, zero_j = find_zero(board)
for i, j in valid_moves((zero_i, zero_j)):
if board[i][j] == tile:
board[zero_i][zero_j], board[i][j] = board[i][j], board[zero_i][zero_j]
return
def main():
board = [
[1, 2, 3],
[4, 0, 6],
[7, 5, 8]
]
print("Welcome to the 8 Puzzle Game!")
print("Arrange the tiles to match this goal state:")
print("1 2 3\n4 5 6\n7 8 ")
while True:
print_board(board)
if is_solved(board):
print("Congratulations! You solved the puzzle!")
break
move = get_user_move(board)
if move is None:
print("Game exited. Goodbye!")
```

```
 break
 feedback = evaluate_move(board, move)
 print(feedback)
 make_move(board, move)
if __name__ == "__main__":
 main()
```

ScreenShot:



<u>Program 3:</u>

Implement Iterative deepening search algorithm.
Algorithm:

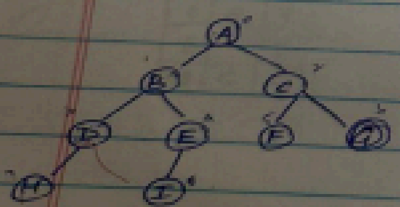
11/9/2025

Iterative deeping depth first search (IDDFS)

Algorithm:

1. Construct a 8-puzzle problem.
2. Set the depth limit to 0 for each row.
3. Our goal is reach the goal state by solving using IDDFS.

1.→ Set the depth limit to 0.
2. For each depth limit, solve using breadth first search (BFS).
3. If the solution is found, return it.
4. If solution is not found, then increase the depth limit.
5. Continue searching the solution breadth-wise. Continue these steps, until the solution is reached.



→ A

→ ABC

→ ABDECFG goal state reached.

→ ABDHEICFG

Code:

```
import copy
def get_puzzle(name):
 print(f"\nEnter the {name} puzzle (3x3, use -1 for blank):")
 puzzle = []
 for i in range(3):
 row = list(map(int, input(f"Row {i+1} (space-separated 3 numbers): ").split()))
puzzle.append(row)
 return puzzle
def move(temp, movement):
 for i in range(3):
 for j in range(3):
 if temp[i][j] == -1:
```

```
if movement == "up" and i > 0:
temp[i][j], temp[i-1][j] = temp[i-1][j], temp[i][j]
elif movement == "down" and i < 2:
temp[i][j], temp[i+1][j] = temp[i+1][j], temp[i][j]
elif movement == "left" and j > 0:
temp[i][j], temp[i][j-1] = temp[i][j-1], temp[i][j]
elif movement == "right" and j < 2:
temp[i][j], temp[i][j+1] = temp[i][j+1], temp[i][j]
return temp
return temp
def dls(puzzle, depth, limit, last_move, goal):
if puzzle == goal:
return True, [puzzle], []
if depth >= limit:
return False, [], []
for move_dir, opposite in [("up","down"), ("left","right"), ("down","up"), ("right","left")]:
if last_move == opposite: # avoid direct backtracking
continue
temp = copy.deepcopy(puzzle)
new_state = move(temp, move_dir)
if new_state != puzzle: # valid move
found, path, moves = dls(new_state, depth+1, limit, move_dir, goal)
if found:
return True, [puzzle] + path, [move_dir] + moves
return False, [], []
def ids(start, goal):
for limit in range(1, 50): # reasonable max depth
print(f"\nTrying depth limit = {limit}")
found, path, moves = dls(start, 0, limit, None, goal)
if found:
print("Solution found!")
for step in path:
print(step)
print("Moves:", moves)
print("Path cost =", len(path)-1)
return
print(" Solution not found within depth limit.")
start_puzzle = get_puzzle("start")
goal_puzzle = get_puzzle("goal")
print("\n~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~")
ids(start_puzzle, goal_puzzle)
```

ScreenShot:

Output

```
Enter the start puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 7 8
Row 3 (space-separated 3 numbers): 5 6 -1

Enter the goal puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 5 6
Row 3 (space-separated 3 numbers): 7 8 -1

~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~

Trying depth limit = 1

Trying depth limit = 2

Trying depth limit = 3

Trying depth limit = 4

Trying depth limit = 5

Trying depth limit = 6

Trying depth limit = 7

Trying depth limit = 8

Trying depth limit = 9

Trying depth limit = 10
```

Program 4:

Implement a vacuum cleaner agent.

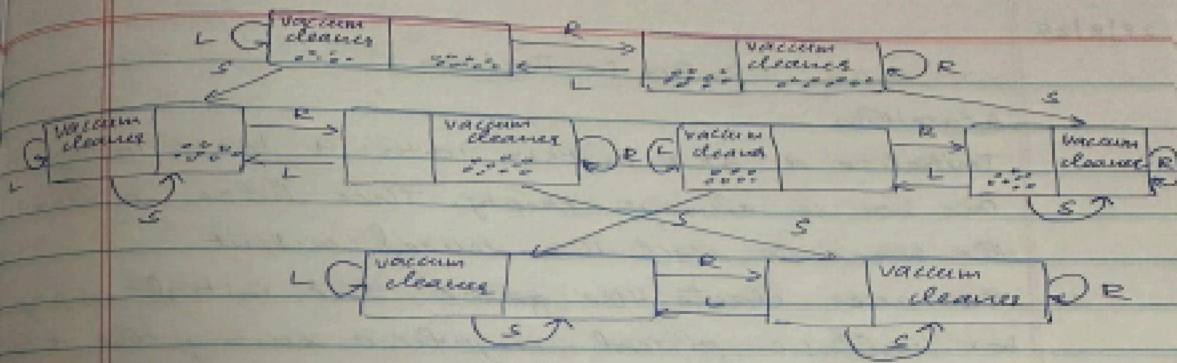
Algorithm:

21/8/25

2) Vaccum cleaner

Algorithm:

1. Consider four rooms (A, B, C, D) that has to be cleaned by the vaccum cleaner.
2. FUNCTION CLEAN.
   If the room is clean,
   move to the next room.
3. FUNCTION DIRTY
   If the room is dirty
   , it has to apply suck function and clean the room.
4. If the room is clean, then move to the next room in clock wise direction.
5. Initialise cleaned_room ~~=0~~ = status
6. If the room is clean, increment the status ~~value of~~ to cleaned rooms.
7. If cleaned_rooms = status ~~4~~, ~~after~~ for all the rooms then stop.
8. Else, continue cleaning.



Output:

Initial room statuses:

Room 1 : Dirty

Room 2 : Clean

Room 3 : Dirty

Room 4 : Dirty

Cleaning room 1 .....

Room 1 is clean

Room 2 is already clean

Cleaning room 3

Room 3 is clean

Cleaning room 4 ..

Room 4 is clean

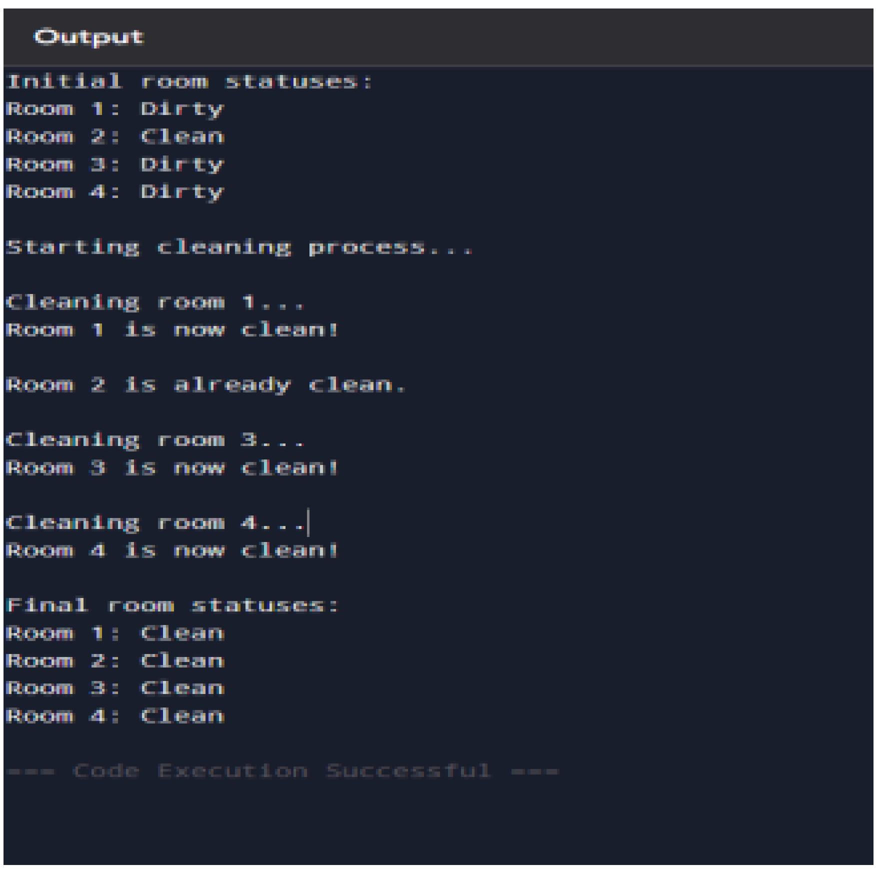
All room clean!

Code:

```
def show_rooms_status(rooms):
 for room_number, status in rooms.items():
 print(f"Room {room_number}: {'Clean' if status else 'Dirty'}")
def clean_room(rooms, room_number):
 if rooms[room_number]:
 print(f"Room {room_number} is already clean.")
 else:
 print(f"Cleaning room {room_number}...")
 rooms[room_number]=True
 print(f"Room {room_number} is now clean!")
def clean_all_rooms(rooms):
 print("Initial room statuses:")
 show_rooms_status(rooms)
```

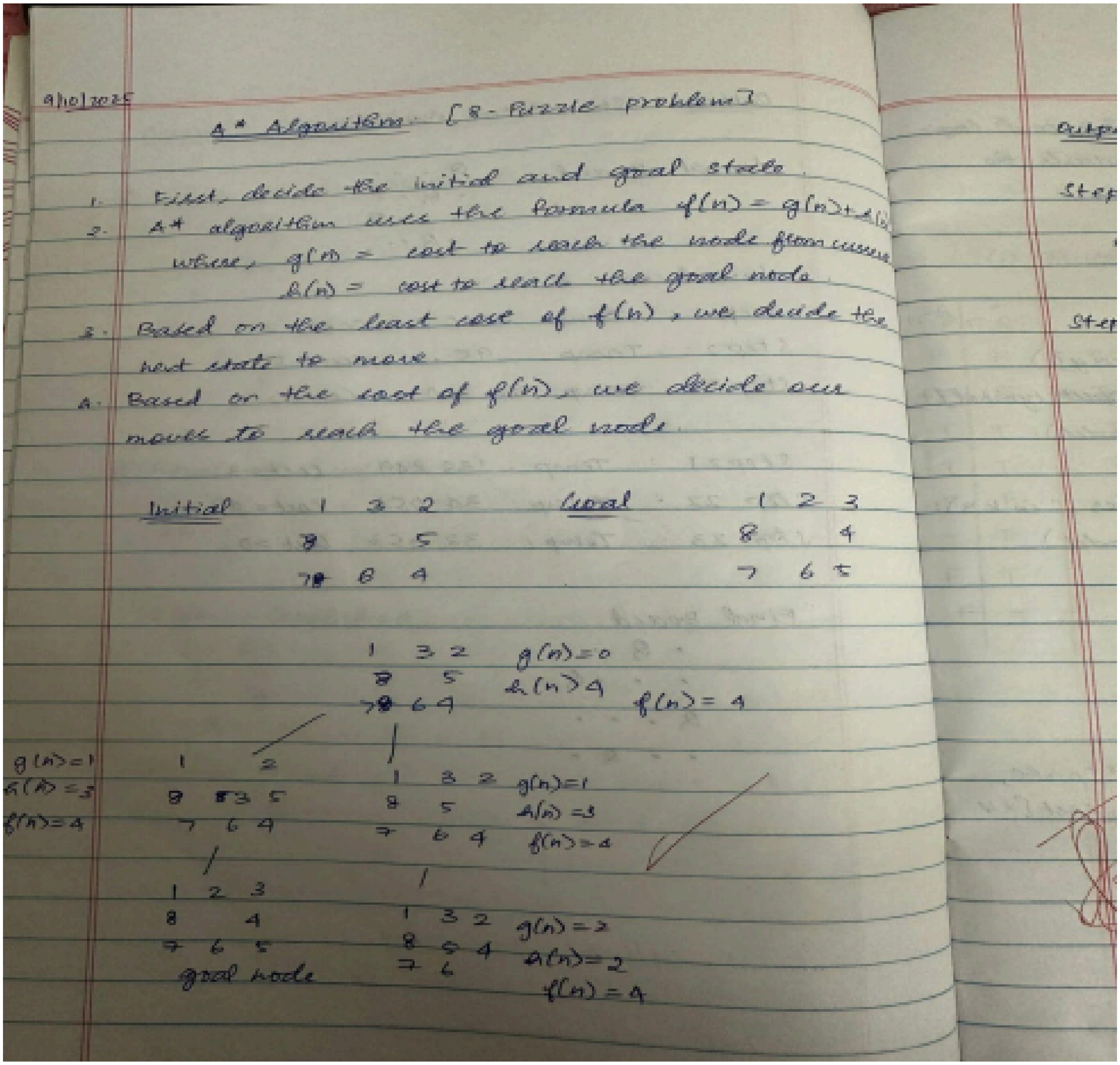
```
print("\nStarting cleaning process...\n")
for room_number in rooms:
clean_room(rooms, room_number)
print()
print("Final room statuses:")
show_rooms_status(rooms)
if __name__ == "__main__":
rooms = {
1: False,
2: True,
3: False,
4: False
}
clean_all_rooms(rooms)
```

ScreenShot:



<u>Program 5:</u>

Implement A* search algorithm.
Algorithm:


9/10/2025

A* Algorithm [8-Puzzle problem]

1. First decide the initial and goal state.
2. A* algorithm uses the formula f(n) = g(n)+h(n)
   where, g(n) = cost to reach the node from curren
   h(n) = cost to reach the goal node.
3. Based on the least cost of f(n), we decide the next state to move.
4. Based on the cost of f(n), we decide our moves to reach the goal node.

Code:

```
from heapq import heappush, heappop
goal_state=[
 [1, 2, 3],
 [8, 0, 4],
 [7, 6, 5]
]
directions= [(-1, 0), (1, 0), (0, -1), (0, 1)]
direction_names= ["UP", "DOWN", "LEFT", "RIGHT"]
def misplaced_tiles(state):
```

```
count=0
fori in range(3):
forj in range(3):
```



```
if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
count += 1
return count
def manhattan_distance(state):
distance = 0
for i in range(3):
for j in range(3):
tile = state[i][j]
if tile != 0:
goal_x, goal_y = divmod(tile - 1, 3)
distance += abs(i - goal_x) + abs(j - goal_y)
return distance
def get_neighbors_with_actions(state):
neighbors = []
for i in range(3):
for j in range(3):
if state[i][j] == 0:
x, y = i, j
break
for (dx, dy), action in zip(directions, direction_names):
nx, ny = x + dx, y + dy
if 0 <= nx < 3 and 0 <= ny < 3:
new_state = [list(row) for row in state]
new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
neighbors.append((new_state, action))
return neighbors
def state_to_tuple(state):
return tuple(tuple(row) for row in state)
def reconstruct_path(came_from, current):
actions = []
states = []
while current in came_from:
prev_state, action = came_from[current]
actions.append(action)
states.append(current)
current = prev_state
states.append(current)
actions.reverse()
states.reverse()
return actions, states
def a_star_search_with_steps(initial_state, heuristic_func):
open_list = []
```

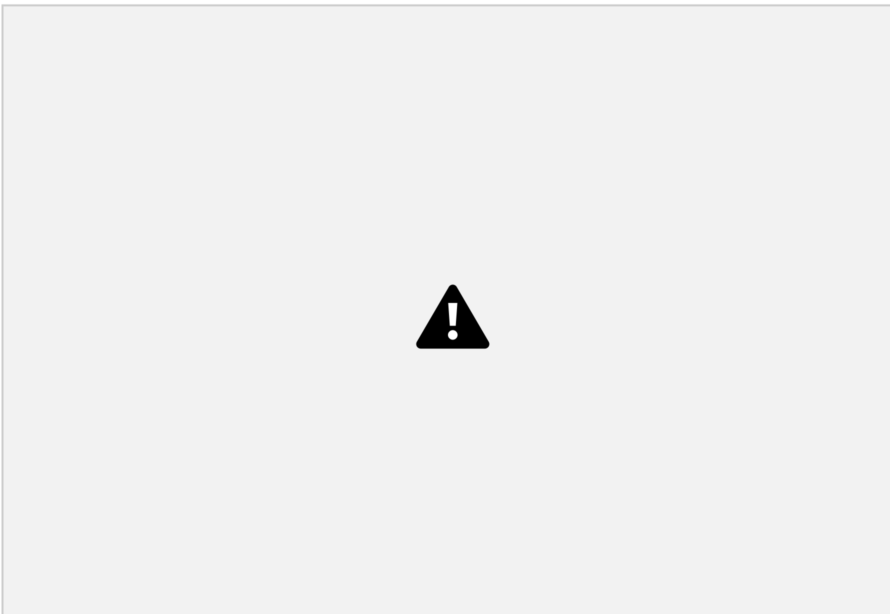
```
 closed_set = set()
 g_score = {state_to_tuple(initial_state): 0}
 f_score = {state_to_tuple(initial_state): heuristic_func(initial_state)}
 came_from = {}
 heappush(open_list, (f_score[state_to_tuple(initial_state)], initial_state))
```



```
 while open_list:
 _, current_state = heappop(open_list)
 current_t = state_to_tuple(current_state)
 if current_state == goal_state:
 return reconstruct_path(came_from, current_t)
 closed_set.add(current_t)
 for neighbor, action in get_neighbors_with_actions(current_state):
 neighbor_t = state_to_tuple(neighbor)
 if neighbor_t in closed_set:
 continue
 tentative_g = g_score[current_t] + 1
 if neighbor_t not in g_score or tentative_g < g_score[neighbor_t]:
 came_from[neighbor_t] = (current_t, action)
 g_score[neighbor_t] = tentative_g
 f_score[neighbor_t] = tentative_g + heuristic_func(neighbor)
 heappush(open_list, (f_score[neighbor_t], neighbor))
 return None, None
def print_path(actions, states):
 for i, (action, state) in enumerate(zip(actions, states[1:]), 1):
 print(f"Step {i}: {action}")
 for row in state:
 print(row)
 print()
initial_state = [
 [1, 2, 3],
 [8, 0, 5],
 [7, 4, 6]
]
print("Using Misplaced Tiles heuristic:")
actions, states = a_star_search_with_steps(initial_state, misplaced_tiles)
if actions:
 print_path(actions, states)
 print("Total steps:", len(actions))
else:
 print("No solution found.")
print("\nUsing Manhattan Distance heuristic:")
actions, states = a_star_search_with_steps(initial_state, manhattan_distance)
if actions:
 print_path(actions, states)
 print("Total steps:", len(actions))
```

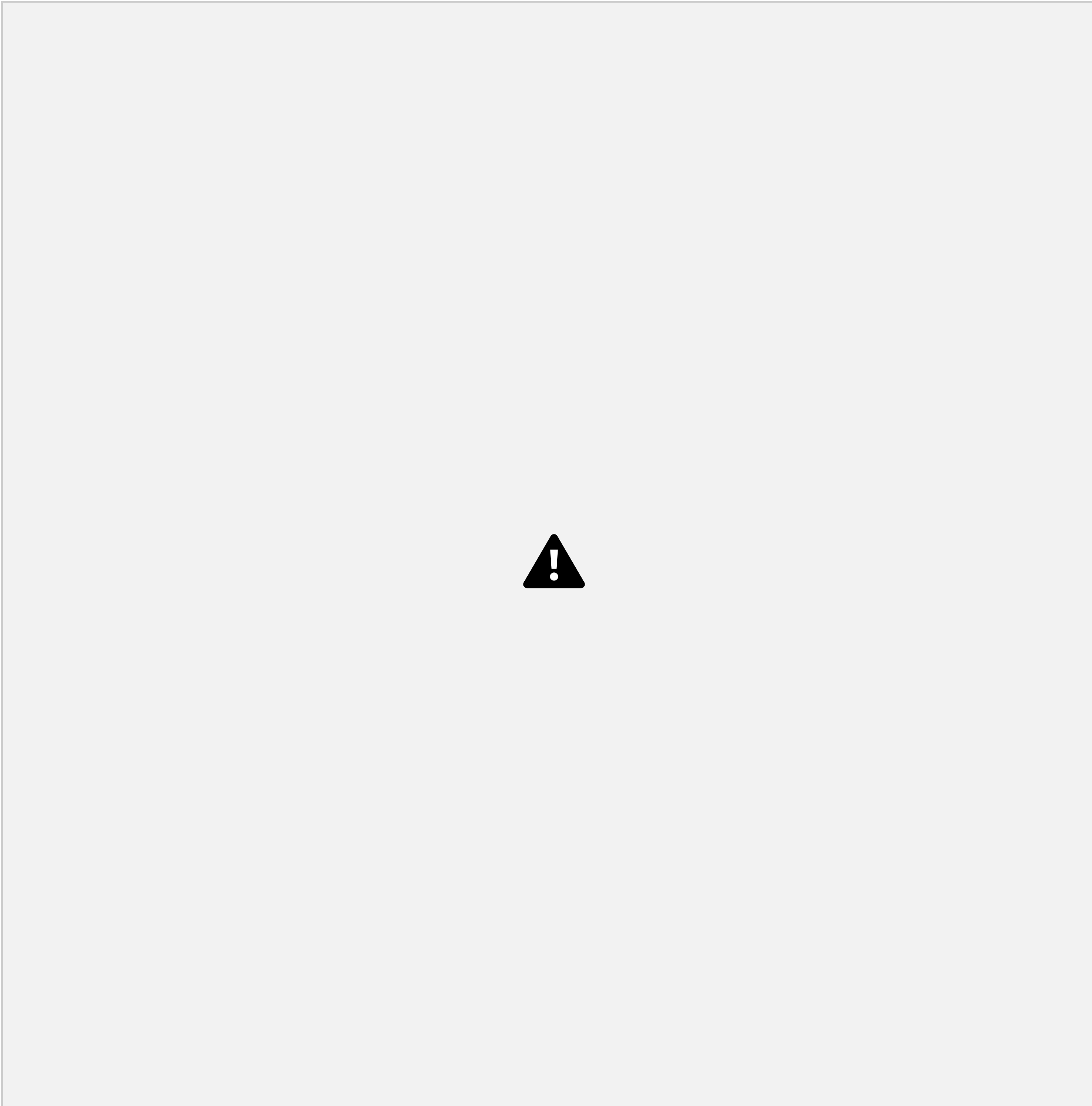
```
else:
 print("No solution found.")
```

ScreenShot:



b. Implement Hill Climbing Algorithm

Algorithm:



Code:

```
import random
import time
def generate_initial_state(n=4):
 return [random.randint(0,n- 1) for_ in range(n)]
def calculate_conflicts(state):
 conflicts=0
 n= len(state)
 fori in range(n):
 forj in range(i+ 1, n):
```

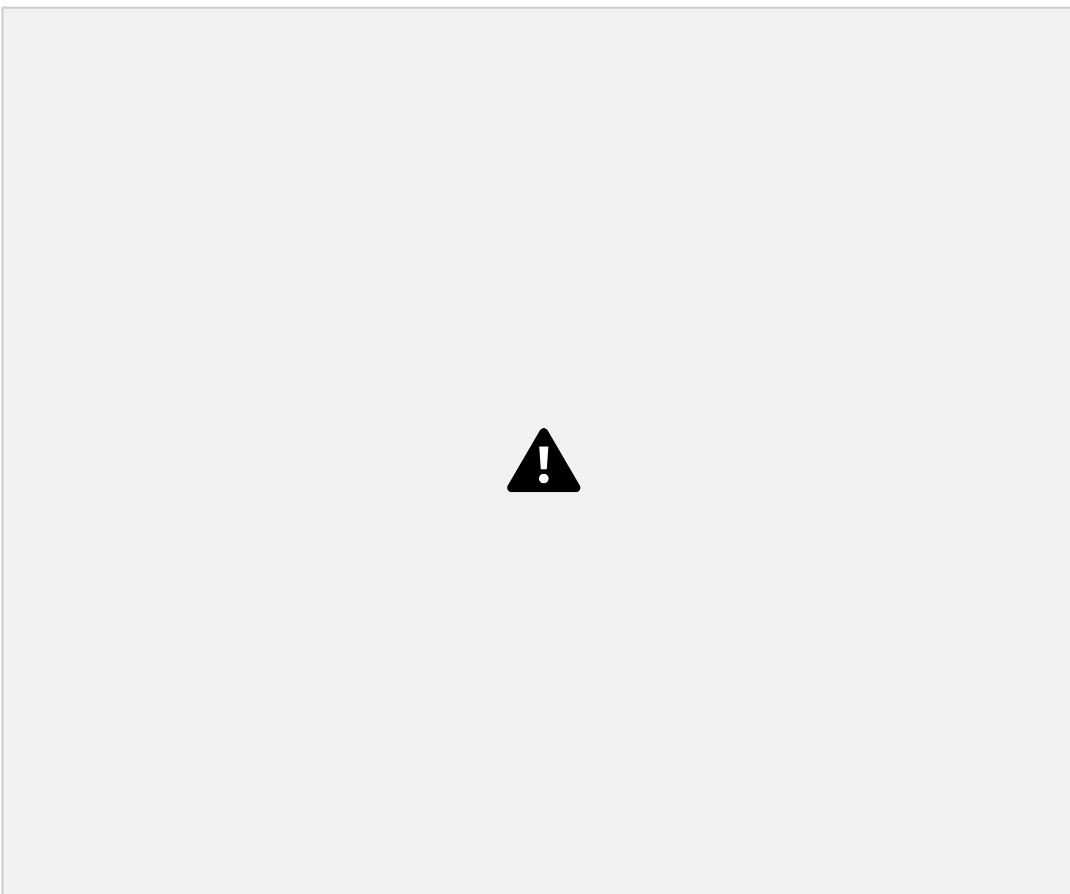
```
if state[i] == state[j]:
conflicts += 1
if abs(state[i] - state[j]) == abs(i - j):
conflicts += 1
return conflicts
def get_neighbors(state):
neighbors = []
n = len(state)
for col in range(n):
for row in range(n):
if state[col] != row:
neighbor = state.copy()
neighbor[col] = row
neighbors.append(neighbor)
return neighbors
def print_board(state):
n = len(state)
for row in range(n):
line = ""
for col in range(n):
if state[col] == row:
line += "Q "
else:
line += ". "
print(line)
print()
def hill_climbing_with_steps(n=4, max_restarts=100):
for restart in range(max_restarts):
current = generate_initial_state(n)
step = 0
print(f"Restart #{restart+1}: Initial state (Conflicts = {calculate_conflicts(current)})")
print_board(current)
while True:
current_conflicts = calculate_conflicts(current)
if current_conflicts == 0:
print(f"Solution found in {step} steps!")
return current
neighbors = get_neighbors(current)
neighbor_conflicts = [calculate_conflicts(nbr) for nbr in neighbors]
min_conflict = min(neighbor_conflicts)
if min_conflict >= current_conflicts:
print("Reached local minimum, restarting...\n")
break
best_neighbor = neighbors[neighbor_conflicts.index(min_conflict)]
step += 1
```

```
 print(f"Step {step}: Conflicts = {min_conflict}")
 print_board(best_neighbor)
```



```
 current = best_neighbor
 return None
solution = hill_climbing_with_steps()
if solution:
 print("Final Solution:")
 print_board(solution)
else:
 print("No solution found.")
```

ScreenShot:



<u>**Program 6:**</u>

Write a program to implement Simulated Annealing Algorithm

Code:

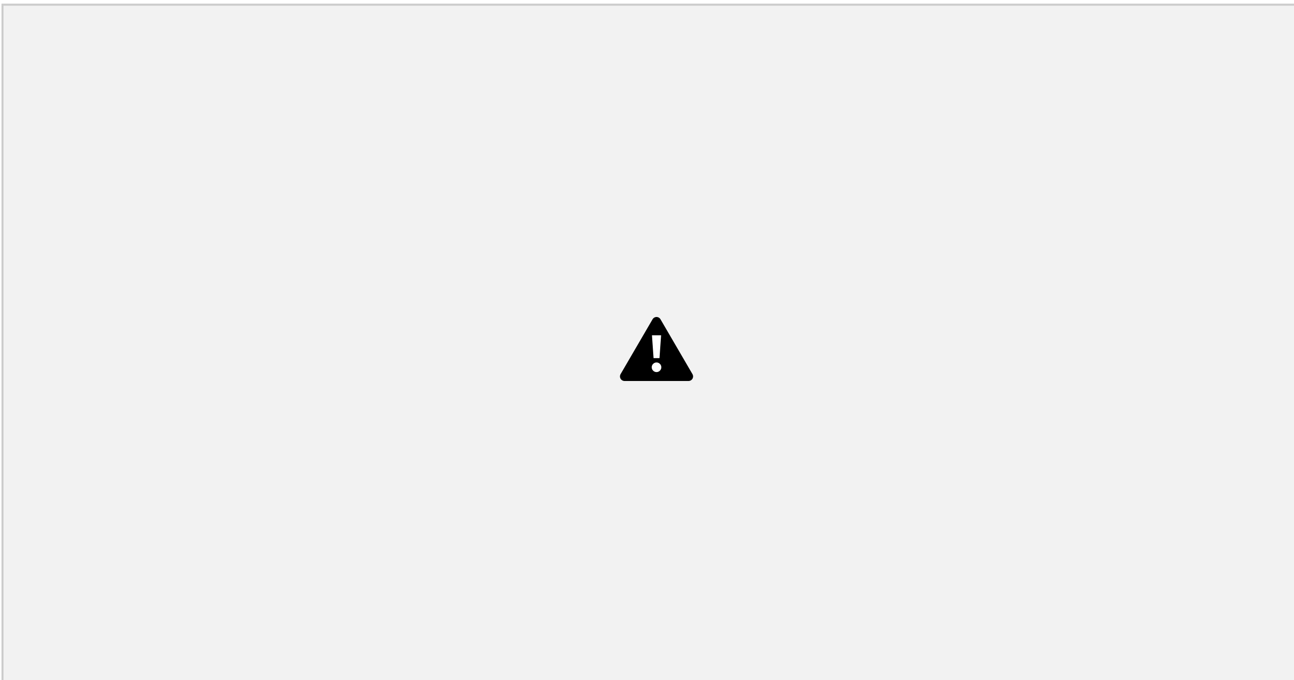
```
import random
import math
def print_board(board):
 n = len(board)
```

```
for i in range(n):
```


```
row = ["Q" if board[i] == j else "." for j in range(n)]
print(" ".join(row))
print()
def calculate_cost(board):
"""Heuristic: number of pairs of queens attacking each other"""
n = len(board)
cost = 0
for i in range(n):
for j in range(i + 1, n):
if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
cost += 1
return cost
def random_neighbor(board):
"""Generate a random neighboring board by moving one queen"""
n = len(board)
neighbor = list(board)
row = random.randint(0, n - 1)
col = random.randint(0, n - 1)
neighbor[row] = col
return neighbor
def simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
current_board = [random.randint(0, n - 1) for _ in range(n)]
current_cost = calculate_cost(current_board)
temperature = initial_temp
step = 1
print("Initial Board:")
print_board(current_board)
print(f"Initial Cost: {current_cost}\n")
while temperature > stopping_temp and current_cost > 0:
neighbor = random_neighbor(current_board)
neighbor_cost = calculate_cost(neighbor)
delta = neighbor_cost - current_cost
if delta < 0 or random.random() < math.exp(-delta / temperature):
current_board = neighbor
current_cost = neighbor_cost
print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1
temperature *= cooling_rate
print("\nFinal Board:")
print_board(current_board)
print(f"Final Cost: {current_cost}")
if current_cost == 0:
print("Goal State Reached!")
else:
```

```
 print("Terminated before reaching goal.")
simulated_annealing(4)
```

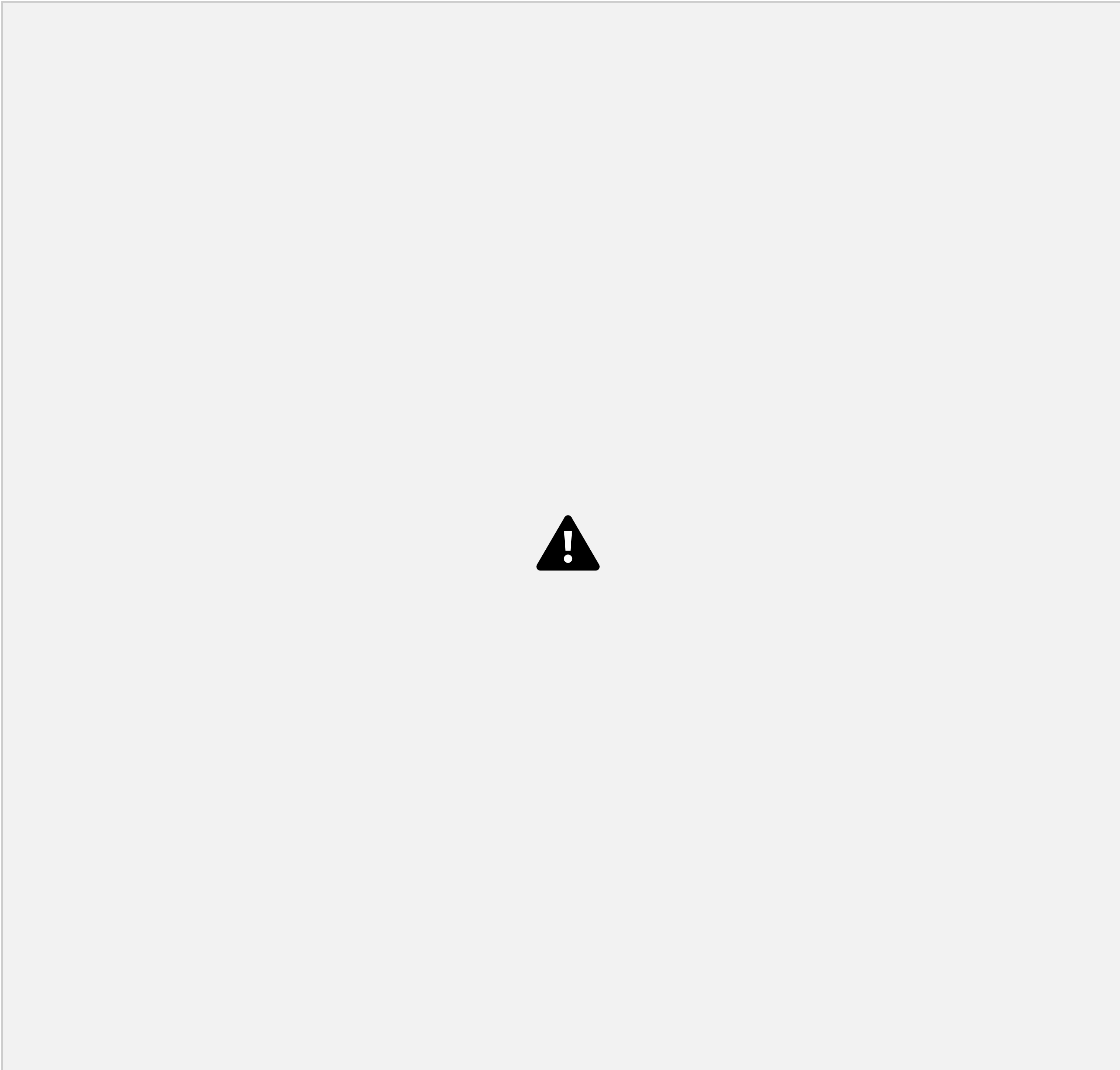

ScreenShot:



Program 7:

Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

Algorithm:

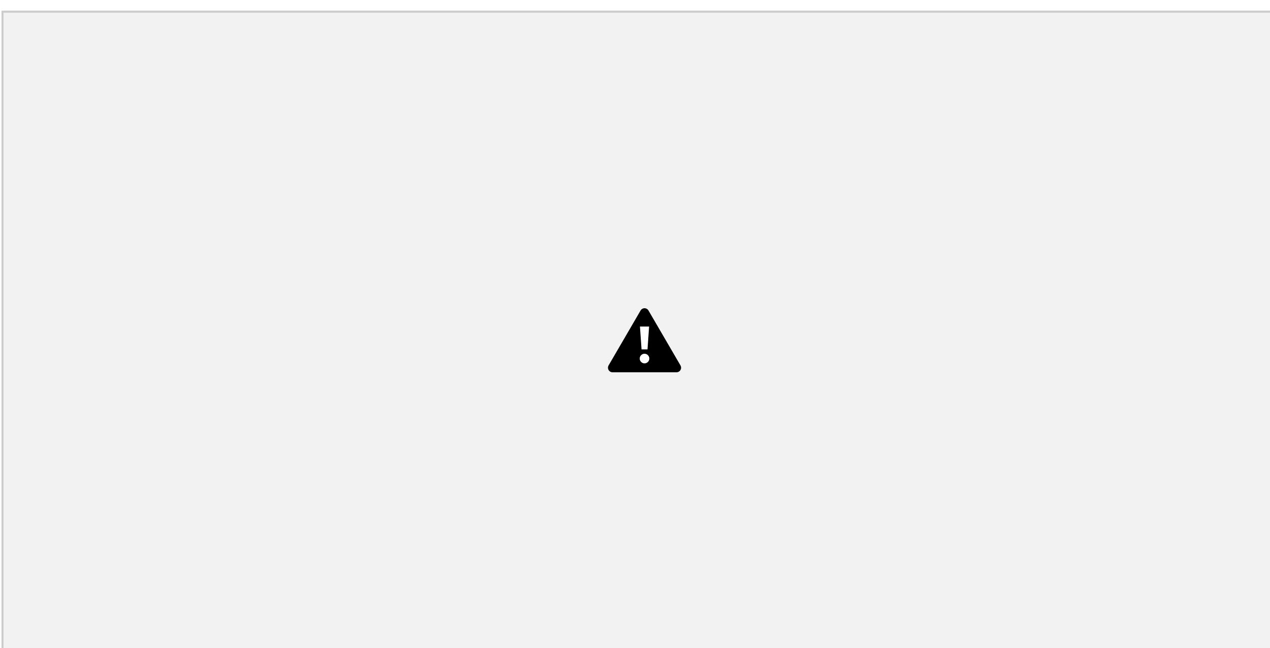


Code:

```
import itertools
import pandas as pd
variables= ['P', 'Q', 'R']
combinations= list(itertools.product([False,True], repeat=3))
rows= []
for (P, Q, R) in combinations:
 s1= (not Q) orP
 s2= (not P) or (not Q) #P→ ¬Q
 s3=Q orR #Q∨R
 KB= s1 and s2 and s3
 entail_R=R
```

```
entail_R_imp_P = (not R) or P
entail_Q_imp_R = (not Q) or R
rows.append({
'P': P, 'Q': Q, 'R': R,
'Q → P': s1,
'P → ¬Q': s2,
'Q ∨ R': s3,
'KB True?': KB,
'R': entail_R,
'R → P': entail_R_imp_P,
'Q → R': entail_Q_imp_R
})
df = pd.DataFrame(rows)
print("Truth Table for Knowledge Base:\n")
print(df.to_string(index=False))
models_true = df[df['KB True?'] == True]
print("\nModels where KB is True:\n")
print(models_true[['P', 'Q', 'R']])
def entails(column):
 """Check if KB entails the given statement."""
 return all(models_true[column])
print("\nEntailment Results:")
print(f"KB ⊨ R ? {'Yes' if entails('R') else 'No'}")
print(f"KB ⊨ R → P ? {'Yes' if entails('R → P') else 'No'}")
print(f"KB ⊨ Q → R ? {'Yes' if entails('Q → R') else 'No'}")
```

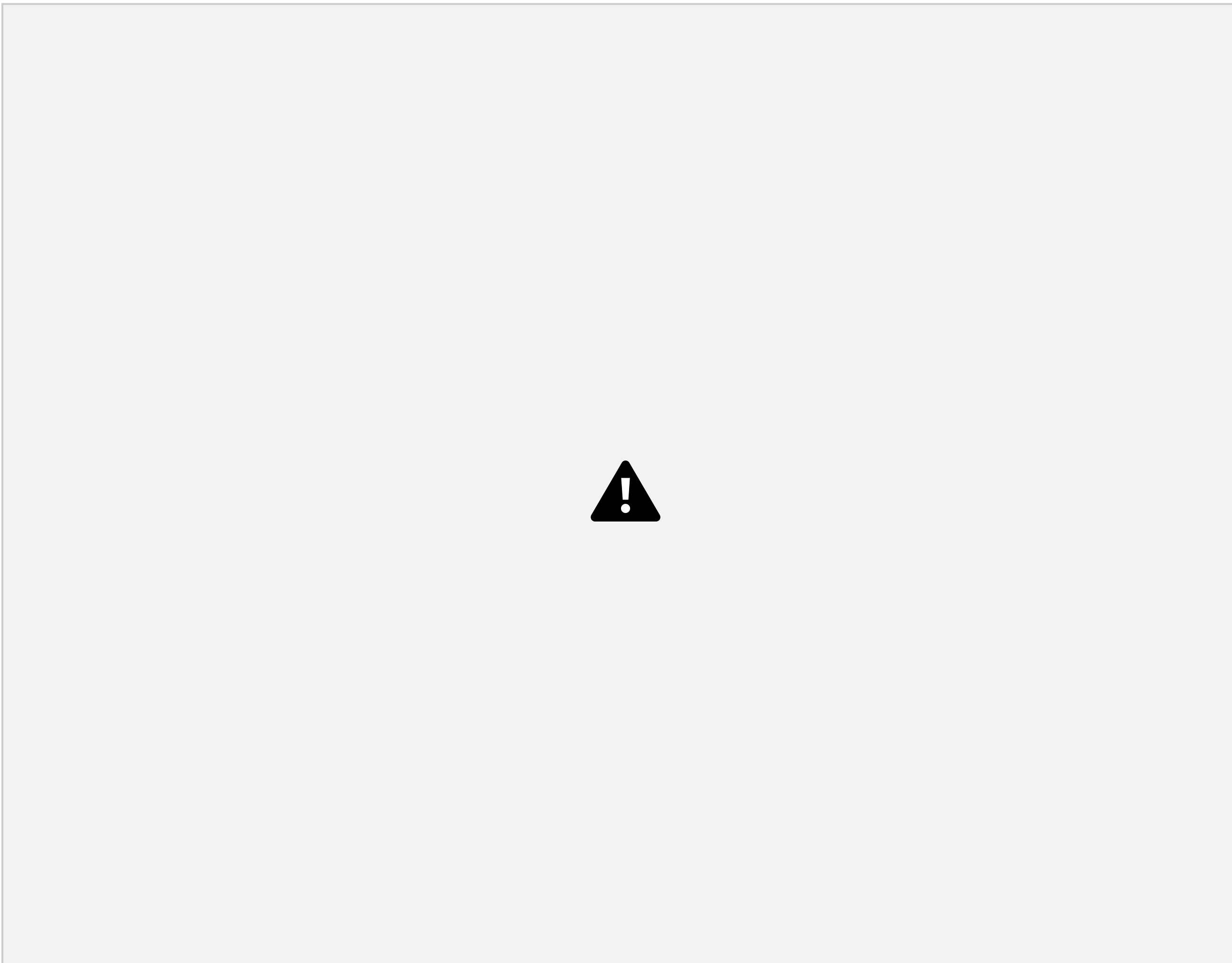
ScreenShot:

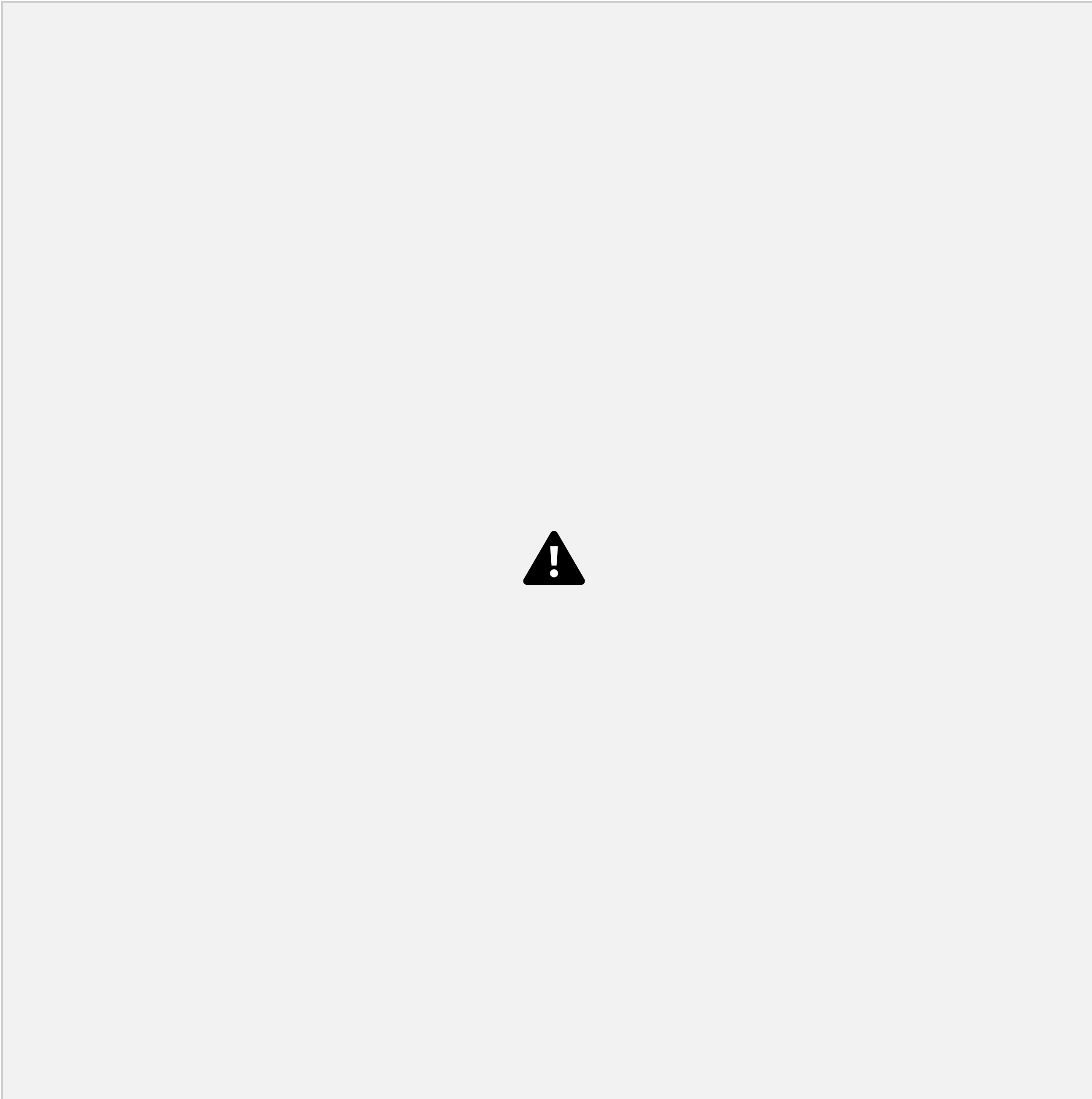


Program 8:

Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:

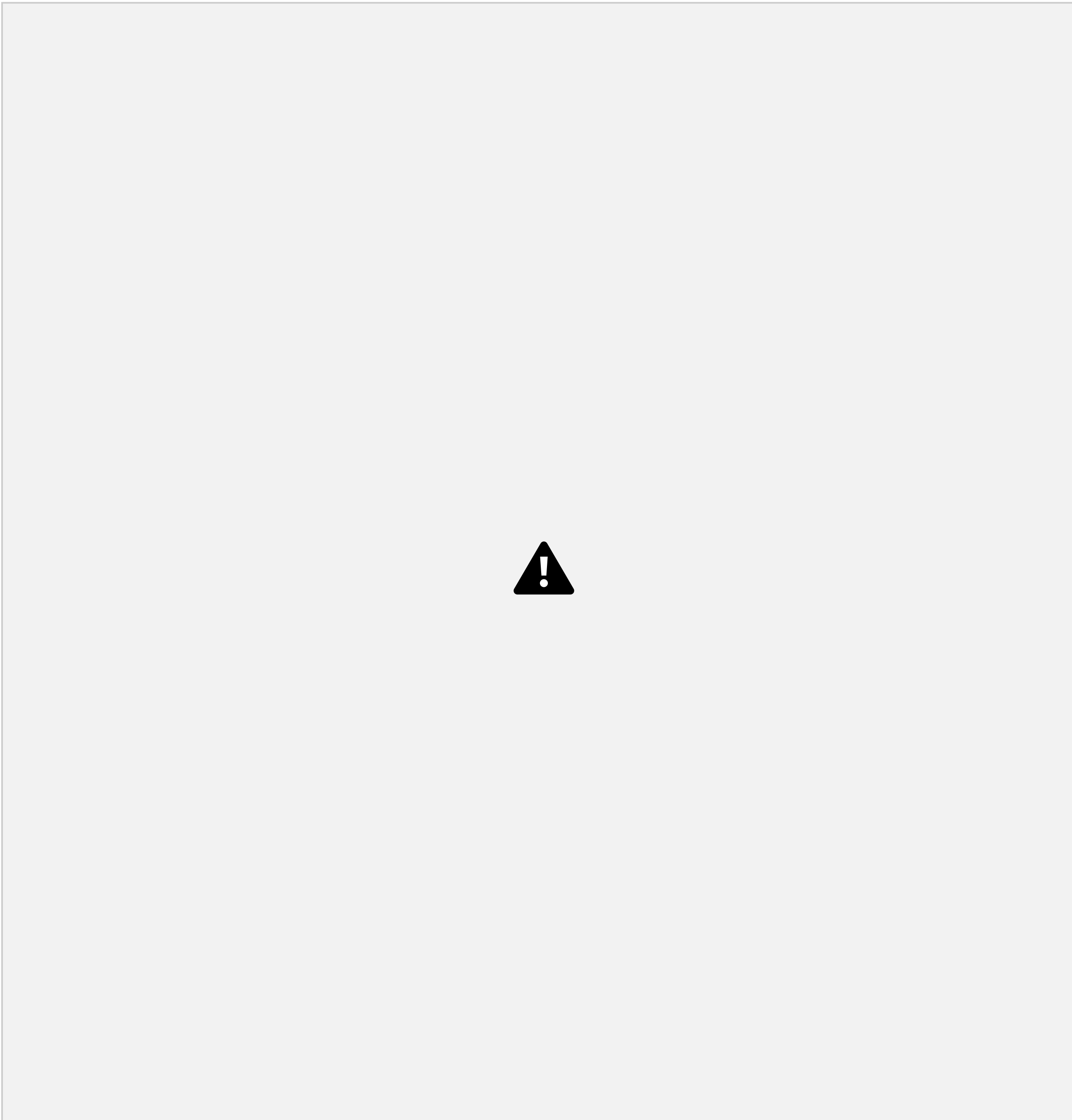




Program 9:

Implement unification in first order logic.

Algorithm:



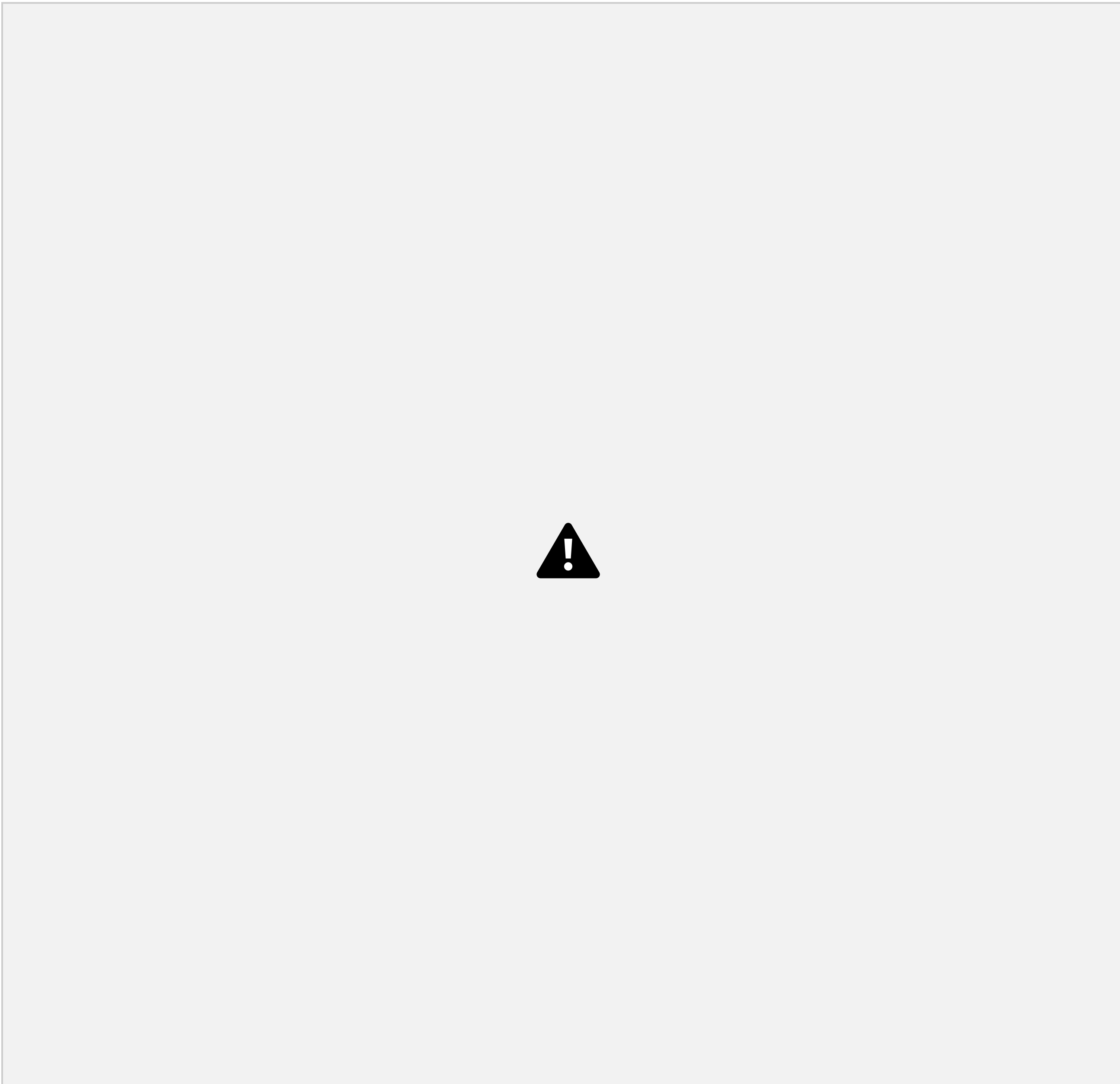
<u>Program 10:</u>

Convert a given first order logic statement into Conjunctive Normal Form (CNF).
Algorithm:

<u>Program 11:</u>

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



<u>Program 12:</u>

Implement Alpha-Beta Pruning.

Algorithm:

Code:

```
import math
PLAYER= "X" # Human
AI= "O" # Computer
def print_board(board):
 for row in board:
 print(" | ".join(row))
 print("-"* 9)
def available_moves(board):
 """Return list of available (row, col) moves."""
 moves= []
```

```
fori in range(3):
forj in range(3):
if board[i][j] ==" ":
moves.append((i, j))
return moves
def check_winner(board):
"""Return 'X' ifX wins, 'O' ifO wins, or None otherwise."""
fori in range(3):
if board[i][0] == board[i][1] == board[i][2] !=" ":
return board[i][0]
if board[0][i] == board[1][i] == board[2][i] !=" ":
return board[0][i]
if board[0][0] == board[1][1] == board[2][2] !=" ":
return board[0][0]
if board[0][2] == board[1][1] == board[2][0] !=" ":
return board[0][2]
return None
def is_full(board):
return all(cell !="" for row in board for cell in row)
def minimax(board, depth, is_maximizing):
winner= check_winner(board)
if winner == AI:
return1
elif winner == PLAYER:
return -1
elif is_full(board):
return0
if is_maximizing:
best_score= -math.inf
for (i, j) in available_moves(board):
board[i][j]= AI
score= minimax(board, depth+ 1, False)
board[i][j]=""
best_score= max(score, best_score)
return best_score
else:
best_score= math.inf
for (i, j) in available_moves(board):
board[i][j]= PLAYER
score= minimax(board, depth+ 1,True)
board[i][j]=""
best_score= min(score, best_score)
return best_score
def best_move(board):
"""Find the best move for the AI."""
```

```
best_score= -math.inf
move= None

for (i, j) in available_moves(board):
board[i][j]= AI
score= minimax(board, 0, False)
board[i][j]=""
if score> best_score:
best_score= score
move= (i, j)
return move
def play_game():
board= [["" for_ in range(3)] for_ in range(3)]
print("TicTacToe-You are X, AI is O")
print_board(board)
whileTrue:
row= int(input("Enter row (0-2): "))
col= int(input("Enter col (0-2): "))
if board[row][col] !=" ":
print("Cell taken, try again.")
continue
board[row][col]= PLAYER
if check_winner(board) == PLAYER:
print_board(board)
print("You win!")
break
elif is_full(board):
print_board(board)
print("It'sa draw!")
break
print("AI is makinga move...")
move= best_move(board)
if move:
board[move[0]][move[1]]= AI
print_board(board)
if check_winner(board) == AI:
print("AI wins!")
break
elif is_full(board):
print("It'sa draw!")
break
if __name__ == "__main__":
play_game()
```

ScreenShot: