Artificial Intelligence Lab Report

Submitted by

Venugopala C S(1BM22CS425)
Batch: C4

Course: Artificial Intelligence
Course Code: 22CS5PCAIP
Sem & Section: 5F

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of contents

Program 1 - Tic Tac toe

Algorithm

Code

import random
class TicTacToe:

```
def minimax(board, is moxing):
    if(Checkmovesforwin(bot)):
        return 1
    elif (Checkmoveforwin(player)):
        return -1
    elif (CheckDraw()):
        return 0

    if is maximizing:
        bestScore=-1000

    for key in board.keys():
        if board(key)==' ':
            board(key)=bot
            Score = minimax(board, False
            board(key = ' '
            if(Score> bestScore):
                bestScore = Score
    return bestScore
    else:
        best Score=1000

while not Checkwin():
    compmove()
    playermove()
```

04.10

```
def __init__(self):
    self.board = []
def create_board(self):
    for i in range(3):
        row = []
        for j in range(3):
            row.append('-')
        self.board.append(row)
def get_random_first_player(self):
    return random.randint(0, 1)
def fix_spot(self, row, col, player):
    self.board[row][col] = player
def is_player_win(self, player):
    win = None
    n = len(self.board)
    for i in range(n):
        win = True
        for j in range(n):
            if self.board[i][j] != player:
                win = False
                break



        if win:
            return win
    for i in range(n):
        win = True
        for j in range(n):
            if self.board[j][i] != player:
                win = False
                break
        if win:
            return win
    win = True
    for i in range(n):
        if self.board[i][i] != player:
            win = False
            break
    if win:
        return win
    win = True
```

```
        for i in range(n):
            if self.board[i][n - 1 - i] != player:
                win = False
                break
        if win:
            return win
        return False
        for row in self.board:
            for item in row:
                if item == '-':
                    return False
        return True
    def is_board_filled(self):
        for row in self.board:
            for item in row:
                if item == '-':
                    return False
        return True
    def swap_player_turn(self, player):
        return 'X' if player == 'O' else 'O'
    def show_board(self):
        for row in self.board:
            for item in row:
                print(item, end=" ")
            print()
    def start(self):
        self.create_board()
        player = 'X' if self.get_random_first_player() == 1 else 'O'
        while True:
            print(f"Player {player} turn")
            self.show_board()
            row, col = list(
                map(int, input("Enter row and column numbers to fix spot: ").split()))
            print()
            self.fix_spot(row - 1, col - 1, player)
            if self.is_player_win(player):
                print(f"Player {player} wins the game!")
                break
            if self.is_board_filled():
```

```
                print("Match Draw!")
                break
            player = self.swap_player_turn(player)
        print()
        self.show_board()
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Output Snapshot

```
Player O turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 0 3
Player X turn
- - -
- - -
- - O
Enter row and column numbers to fix spot: 1 2
Player O turn
- X - - - -
- - O
Enter row and column numbers to fix spot: 3 0
Player X turn
- X -
- - -
- - O
Enter row and column numbers to fix spot: 3 2
Player O turn
- X -
- - -
- X O
Enter row and column numbers to fix spot: 2 1
Player X turn
- X -
O - -
- X O
Enter row and column numbers to fix spot: 2 2
Player X wins the game!
```

Program 2 - 8 Puzzle Using BFS

Algorithm

Lab-2

18-10-24

P1:- 8 puzzle problem using DFS.

Algorithm:-

Let fringe be a list containing the initial state

loop
 if fringe is empty return failure.
 node ← remove first (fring)
 if node is a goal
  Than return the path from initial state to node
 else generate all successor node & add generated node to the front of fringe
end loop

P2:- 8 puzzle problem using BFS

Algorithm:-

Let fring be a list containing the initial state

loop
 if fringe is empty return failure
 node ← remove-first (fringe)
 if node is a goal
  then return the path from initial state to node
 else generate all successor node & add generated node to the back of fring
end loop.

Code

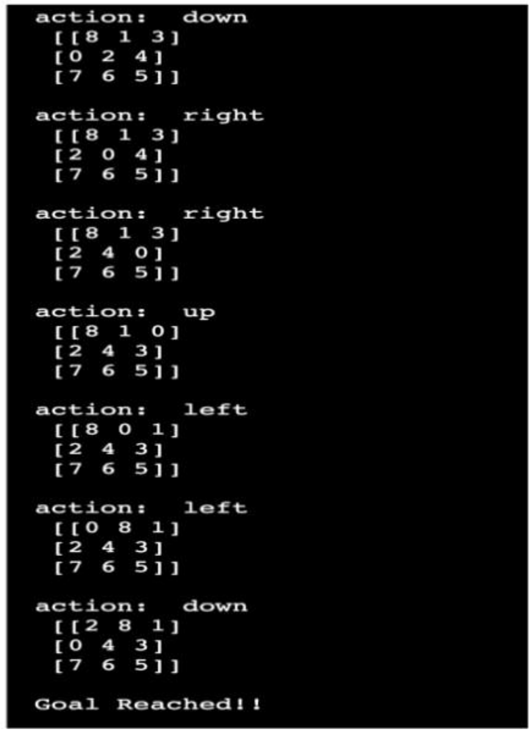
```
import sys
import numpy as np
class Node:
	def __init__(self, state, parent, action):
		self.state = state
		self.parent = parent
		self.action = action
class StackFrontier:
	def __init__(self):
		self.frontier = []
	def add(self, node):
		self.frontier.append(node)
	def contains_state(self, state):
		return any((node.state[0] == state[0]).all() for node in self.frontier)
	def empty(self):
		return len(self.frontier) == 0
	def remove(self):
		if self.empty():
			raise Exception("Empty Frontier")
		else:
			node = self.frontier[-1]
			self.frontier = self.frontier[:-1]
			return node


class QueueFrontier(StackFrontier):
	def remove(self):
		if self.empty():
			raise Exception("Empty Frontier")
		else:
			node = self.frontier[0]
			self.frontier = self.frontier[1:]
			return node
class Puzzle:
	def __init__(self, start, startIndex, goal, goalIndex):
		self.start = [start, startIndex]
		self.goal = [goal, goalIndex]
		self.solution = None

	def neighbors(self, state):
		mat, (row, col) = state
```

```
        results = []
        if row > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))
        return results
    def print(self):
        solution = self.solution if self.solution is not None else None



        print("Start State:\n", self.start[0], "\n")
        print("Goal State:\n", self.goal[0], "\n")
        print("\nStates Explored: ", self.num_explored, "\n")
        print("Solution:\n ")
        for action, cell in zip(solution[0], solution[1]):
            print("action: ", action, "\n", cell[0], "\n")
        print("Goal Reached!!")
    def does_not_contain_state(self, state):
        for st in self.explored:
            if (st[0] == state[0]).all():
            return False
        return True
    def solve(self):
        self.num_explored = 0
        start = Node(state=self.start, parent=None, action=None)
```

```
            frontier = QueueFrontier()
            frontier.add(start)
            self.explored = []
            while True:
                if frontier.empty():
                    raise Exception("No solution")
                node = frontier.remove()
                self.num_explored += 1
                if (node.state[0] == self.goal[0]).all():
                    actions = []
                    cells = []
                    while node.parent is not None:
                        actions.append(node.action)
                        cells.append(node.state)
                        node = node.parent
                    actions.reverse()
                    cells.reverse()
                    self.solution = (actions, cells)
                    return
                self.explored.append(node.state)
                for action, state in self.neighbors(node.state):
        if not frontier.contains_state(state) and self.does_not_contain_state(state): child =
                        Node(state=state, parent=node, action=action)
                        frontier.add(child)
start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])



goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])
startIndex = (1, 1)
goalIndex = (1, 0)
p = Puzzle(start, startIndex, goal, goalIndex)
p.solve() p.print()
```

Output Snapshot

```
Start State:
 [[1 2 3]
 [8 0 4]
 [7 6 5]]

Goal State:
 [[2 8 1]
 [0 4 3]
 [7 6 5]]


States Explored:  358

Solution:

action:  up
 [[1 0 3]
 [8 2 4]
 [7 6 5]]

action:  left
 [[0 1 3]
 [8 2 4]
 [7 6 5]]

action:  down
 [[8 1 3]
 [0 2 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 0 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 4 0]
 [7 6 5]]
```



```
action:  down
 [[8 1 3]
 [0 2 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 0 4]
 [7 6 5]]

action:  right
 [[8 1 3]
 [2 4 0]
 [7 6 5]]

action:  up
 [[8 1 0]
 [2 4 3]
 [7 6 5]]

action:  left
 [[8 0 1]
 [2 4 3]
 [7 6 5]]

action:  left
 [[0 8 1]
 [2 4 3]
 [7 6 5]]

action:  down
 [[2 8 1]
 [0 4 3]
 [7 6 5]]

Goal Reached!!
```

Program 3 – VACUUM

CLEANER

Algorithm

classmate
Date
Page 6

Vacume cleaner algo

States:- The state is determined by both the agent location and the dirt location. The agent is in one of ~~two~~ two location. each ~~of~~ which might or might not contain dirt.

initial state: any state can be designated as the initial state.

Actions:- in this simple environment each state has just three actions: Left, right & suck. Larger environment might also include up & down

Transition model:- The action have their expected effects. ~~exp~~ except that moving left in the leftmost square, moving Right in the right-most sqare & sucking in a clean square have no effect

goal test:- This check whether all the squares are clean.

Path cost: each step costs 1, so the path cost is the no. of steps in the path.

18.10

```
def vacuum_world():
    # Initialize goal state: 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    # User input for vacuum location and status
    location_input = input("Enter location of vacuum (A/B): ").strip().upper()
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty): ").strip()
    status_input_complement = input(f"Enter status of other room ({'B' if location_input == 'A' else 'A'}): ").strip()

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        print("Vacuum is placed in Location A")

        if status_input == '1':  # Location A is Dirty
            print("Location A is Dirty.")
            # Clean A
            goal_state['A'] = '0'
            cost += 1  # Cost for cleaning
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':  # If B is Dirty
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print("Cost for moving RIGHT: " + str(cost))
```

```
            # Clean B
            goal_state['B'] = '0'
            cost += 1  # Cost for cleaning
            print("Cost for CLEANING B: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")
    else:
        print("Location A is already clean.")

        if status_input_complement == '1':  # If B is Dirty
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1  # Cost for moving right
            print("Cost for moving RIGHT: " + str(cost))
            # Clean B
            goal_state['B'] = '0'
            cost += 1  # Cost for cleaning
            print("Cost for CLEANING B: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")

else:  # Vacuum is placed in Location B
    print("Vacuum is placed in Location B")

    if status_input == '1':  # Location B is Dirty
        print("Location B is Dirty.")
        # Clean B
        goal_state['B'] = '0'
        cost += 1  # Cost for cleaning
```

```
        print("Cost for CLEANING B: " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':  # If A is Dirty
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print("Cost for moving LEFT: " + str(cost))
            # Clean A
            goal_state['A'] = '0'
            cost += 1  # Cost for cleaning
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")

        if status_input_complement == '1':  # If A is Dirty
            print("Location A is Dirty.")
            print("Moving left to Location A.")
            cost += 1  # Cost for moving left
            print("Cost for moving LEFT: " + str(cost))
            # Clean A
            goal_state['A'] = '0'
            cost += 1  # Cost for cleaning
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
```

```
    # Done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

# Call the function
vacuum_world()
```

Lab-3

25-10-24

A* Algorithm:-

function A* search(problem) returns a solution or failure

node ← a node n with n.state = problem.initial state n.g=0

frontier ← a priority queue ordered by ascending g+h only element

loop do

if empty?(frontier) then return failure.

n ← pop(frontier)

if problem.goalTest(n.state) then return solution(n)

for each action a in problem.action(n.state) do

n' ← child node(problem.n.a)

insert(n', g(n')+h(n'), frontier)

Program 04 - 8 Puzzle Using A*

Algorithm

Code

```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
f"""
 {state[0]} {state[1]} {state[2]}
 {state[3]} {state[4]} {state[5]}
 {state[6]} {state[7]} {state[8]}
 “””
    )
def h(state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target[i]:
            count = count+1
    return count
def astar(state, target):
    states = [src]
    g = 0
    visited_states = []
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.append(state)
            print_b(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(
                state, visited_states) if move not in moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i]
                  for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")
def possible_moves(state, visited_state):
```

```
    b = state.index(-1)
    d = []
    if b - 3 in range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if move not in visited_state]
def gen(state, m, b):
    temp = state.copy()
    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]
    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    return temp
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5,6, 7, 8,-1]
astar(src, target)
```

Output Snapshot

```
Enter the start state matrix

1 0 1 0
1 0 0 1
1 1 1 1
Enter the goal state matrix

1 1 0 1
1 0 0 1
1 1 1 0
|
  |
 \'/

1 0 1 0
1 0 0 1
1 1 1 1
```

Program-07

MisplaceTiles

Algorithm

TITEL: MANHATTAN DISTANCE

```
import heapq

class PuzzleState:
    def __init__(self, board, g, h):
        self.board = board  # The current state of the board
        self.g = g  # Cost to reach this node (depth)
        self.h = h  # Heuristic cost (Manhattan distance)
        self.f = g + h  # Total cost (f(n) = g(n) + h(n))

    def __lt__(self, other):
        return self.f < other.f  # For priority queue to sort by f(n)

def print_board(board):
    """Print the current board state."""
    for row in board:
        print(" ".join(str(num) for num in row))
    print()  # Empty line for better readability

def get_blank_position(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:  # Find the blank space (0)
                return (i, j)

def get_successors(state):
    successors = []
    x, y = get_blank_position(state.board)  # Get position of blank tile
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Possible moves
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
```

```
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Valid move
            new_board = [row[:] for row in state.board]  # Copy the current board
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
# Swap
            successors.append(PuzzleState(new_board, state.g + 1, 0))  # Create new state
    return successors

def heuristic_manhattan_distance(board):
    distance = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0:
                target_x = (board[i][j] - 1) // 3
                target_y = (board[i][j] - 1) % 3
                distance += abs(i - target_x) + abs(j - target_y)
    return distance

def is_goal_state(board):
    return board == [[1, 2, 3],
                     [8, 0, 4],
                     [7, 6, 5]]  # Check if the board is in the goal state

def a_star_search_manhattan_distance(start_board):
    start_state = PuzzleState(start_board, 0,
heuristic_manhattan_distance(start_board))
    open_set = []
    heapq.heappush(open_set, start_state)
    closed_set = set()

    while open_set:
        current_state = heapq.heappop(open_set)

        # Print current board state and details
        print("Current board state:")
        print_board(current_state.board)
        print(f"g(n): {current_state.g}, h(n): {current_state.h}, f(n): {current_state.f}\n")

        # Check if we've reached the goal
        if is_goal_state(current_state.board):
            print("Goal state reached!")
            return current_state.g  # Return the cost to reach the goal

        closed_set.add(tuple(map(tuple, current_state.board)))
```

```
        for successor in get_successors(current_state):
            successor.h = heuristic_manhattan_distance(successor.board)
            successor.f = successor.g + successor.h

            if tuple(map(tuple, successor.board)) in closed_set:
                continue

            heapq.heappush(open_set, successor)

    return None  # No solution found

def get_user_input():
    board = []
    for i in range(3):
        while True:
            row = input(f"Enter row {i + 1} (3 numbers separated by space): ")
            nums = list(map(int, row.split()))
            if len(nums) == 3 and all(0 <= num <= 8 for num in nums):
                board.append(nums)
                break
            else:
                print("Invalid input. Please enter 3 numbers between 0 and 8.")
    return board

if __name__ == "__main__":
    start_board = get_user_input()
    steps = a_star_search_manhattan_distance(start_board)
    print(f"Steps to solve with Manhattan Distance heuristic: {steps}")
```

output:

```
Enter row 1 (3 numbers separated by space): 2 8 3
Enter row 2 (3 numbers separated by space): 1 6 4
Enter row 3 (3 numbers separated by space): 7 0 5
Current board state:
2 8 3
1 6 4
7 0 5

g(n): 0, h(n): 9, f(n): 9

Current board state:
2 8 3
1 6 4
7 5 0
```

g(n): 1, h(n): 8, f(n): 9

Current board state:
2 8 3
1 6 4
0 7 5

g(n): 1, h(n): 10, f(n): 11

Current board state:
2 8 3
1 0 4
7 6 5

g(n): 1, h(n): 10, f(n): 11

Current board state:
2 8 3
1 6 0
7 5 4

g(n): 2, h(n): 9, f(n): 11

Current board state:
2 0 3
1 8 4
7 6 5

g(n): 2, h(n): 9, f(n): 11

Current board state:
2 8 3
1 0 6
7 5 4

g(n): 3, h(n): 8, f(n): 11

Current board state:
0 2 3
1 8 4
7 6 5

g(n): 3, h(n): 8, f(n): 11

Current board state:
2 0 3
1 8 6
7 5 4

g(n): 4, h(n): 7, f(n): 11

Current board state:
2 8 3
1 4 0
7 6 5

g(n): 2, h(n): 9, f(n): 11

Current board state:
2 8 3
1 5 6
7 0 4

g(n): 4, h(n): 7, f(n): 11

Current board state:
1 2 3
0 8 4
7 6 5

g(n): 4, h(n): 7, f(n): 11

Current board state:
2 8 3
1 5 6
7 4 0

g(n): 5, h(n): 6, f(n): 11

Current board state:
2 8 3
1 4 5
7 6 0

g(n): 3, h(n): 8, f(n): 11

Current board state:
0 2 3
1 8 6

7 5 4

g(n): 5, h(n): 6, f(n): 11

Current board state:
2 8 3
1 4 5
7 0 6

g(n): 4, h(n): 7, f(n): 11

Current board state:
1 2 3
0 8 6
7 5 4

g(n): 6, h(n): 5, f(n): 11

Current board state:
2 8 3
0 6 4
1 7 5

g(n): 2, h(n): 11, f(n): 13

Current board state:
2 8 3
0 1 6
7 5 4

g(n): 4, h(n): 9, f(n): 13

Current board state:
2 8 3
1 4 5
0 7 6

g(n): 5, h(n): 8, f(n): 13

Current board state:
1 2 3
7 8 6
0 5 4

g(n): 7, h(n): 6, f(n): 13

Current board state:
1 2 3
8 0 6
7 5 4

g(n): 7, h(n): 6, f(n): 13

Current board state:
2 3 0
1 8 4
7 6 5

g(n): 3, h(n): 10, f(n): 13

Current board state:
2 8 3
1 0 5
7 4 6

g(n): 5, h(n): 8, f(n): 13

Current board state:
1 2 3
7 8 4
0 6 5

g(n): 5, h(n): 8, f(n): 13

Current board state:
2 8 0
1 4 3
7 6 5

g(n): 3, h(n): 10, f(n): 13

Current board state:
2 8 3
1 5 6
0 7 4

g(n): 5, h(n): 8, f(n): 13

Current board state:
2 8 3

1 5 0
7 4 6

g(n): 6, h(n): 7, f(n): 13

Current board state:
2 8 3
1 5 0
7 4 6

g(n): 6, h(n): 7, f(n): 13

Current board state:
2 8 0
1 6 3
7 5 4

g(n): 3, h(n): 10, f(n): 13

Current board state:
2 3 0
1 8 6
7 5 4

g(n): 5, h(n): 8, f(n): 13

Current board state:
1 2 3
8 0 4
7 6 5

g(n): 5, h(n): 8, f(n): 13

Goal state reached!

Title: A* MISPLACED TILES

```
import heapq


class PuzzleState:
    def __init__(self, board, g, h):
```

```
        self.board = board  # The current state of the board
        self.g = g  # Cost to reach this node (depth)
        self.h = h  # Heuristic cost (misplaced tiles)
        self.f = g + h  # Total cost (f(n) = g(n) + h(n))

    def __lt__(self, other):
        return self.f < other.f  # For priority queue to sort by f(n)

def print_board(board):
    """Print the current board state."""
    for row in board:
        print(" ".join(str(num) for num in row))
    print()  # Empty line for better readability

def get_blank_position(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:  # Find the blank space (0)
                return (i, j)

def get_successors(state):
    successors = []
    x, y = get_blank_position(state.board)  # Get position of blank tile
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Possible moves
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Valid move
            new_board = [row[:] for row in state.board]  # Copy the current board
            new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
```

```
# Swap
            successors.append(PuzzleState(new_board, state.g + 1, 0))  # Create new
state
    return successors


def heuristic_misplaced_tiles(board):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if board[i][j] != 0 and board[i][j] != i * 3 + j + 1:  # Check for misplaced tiles
                misplaced += 1
    return misplaced


def is_goal_state(board):
    return board == [[1, 2, 3],
                     [8, 0, 4],
                     [7, 6, 5]]  # Check if the board is in the goal state


def a_star_search_misplaced_tiles(start_board):
    start_state = PuzzleState(start_board, 0, heuristic_misplaced_tiles(start_board))
    open_set = []
    heapq.heappush(open_set, start_state)
    closed_set = set()


    while open_set:
        current_state = heapq.heappop(open_set)


        # Print current board state and details
        print("Current board state:")
        print_board(current_state.board)
```

```
        print(f"g(n): {current_state.g}, h(n): {current_state.h}, f(n): {current_state.f}\n")

        # Check if we've reached the goal
        if is_goal_state(current_state.board):
            print("Goal state reached!")
            return current_state.g  # Return the cost to reach the goal

        closed_set.add(tuple(map(tuple, current_state.board)))

        for successor in get_successors(current_state):
            successor.h = heuristic_misplaced_tiles(successor.board)
            successor.f = successor.g + successor.h

            if tuple(map(tuple, successor.board)) in closed_set:
                continue

            heapq.heappush(open_set, successor)

    return None  # No solution found

def get_user_input():
    board = []
    for i in range(3):
        while True:
            row = input(f"Enter row {i + 1} (3 numbers separated by space): ")
            nums = list(map(int, row.split()))
            if len(nums) == 3 and all(0 <= num <= 8 for num in nums):
                board.append(nums)
                break
```

```
        else:
            print("Invalid input. Please enter 3 numbers between 0 and 8.")
    return board

if __name__ == "__main__":
    start_board = get_user_input()
    steps = a_star_search_misplaced_tiles(start_board)
    print(f"Steps to solve with Misplaced Tiles heuristic: {steps}")
```

OUTPUT:

```
Enter row 1 (3 numbers separated by space): 2 8 3
Enter row 2 (3 numbers separated by space): 1 6 4
Enter row 3 (3 numbers separated by space): 0 7 5
Current board state:
2 8 3
1 6 4
0 7 5
g(n): 0, h(n): 7, f(n): 7
Current board state:
2 8 3
1 6 4
7 0 5
g(n): 1, h(n): 6, f(n): 7
Current board state:
2 8 3
0 6 4
1 7 5

g(n): 1, h(n): 7, f(n): 8
```

Current board state:

2 8 3

1 0 4

7 6 5

g(n): 2, h(n): 6, f(n): 8

Current board state:

2 8 3

1 6 4

7 5 0

g(n): 2, h(n): 6, f(n): 8

Current board state:

0 8 3

2 6 4

1 7 5

g(n): 2, h(n): 7, f(n): 9

Current board state:

2 8 3

1 4 0

7 6 5

g(n): 3, h(n): 6, f(n): 9

Current board state:

2 8 3

1 6 0

7 5 4

g(n): 3, h(n): 6, f(n): 9

Current board state:

2 8 3

6 0 4

1 7 5

g(n): 2, h(n): 7, f(n): 9

Current board state:

2 8 3

0 1 4

7 6 5

g(n): 3, h(n): 6, f(n): 9

Current board state:

2 0 3

1 8 4

7 6 5

g(n): 3, h(n): 6, f(n): 9

Current board state:

2 8 3

1 0 6

7 5 4

g(n): 4, h(n): 5, f(n): 9

Current board state:

0 2 3

1 8 4

7 6 5

g(n): 4, h(n): 5, f(n): 9

Current board state:

2 8 3

1 5 6

7 0 4

g(n): 5, h(n): 4, f(n): 9

Current board state:

1 2 3

0 8 4

7 6 5

g(n): 5, h(n): 4, f(n): 9

Current board state:

2 8 3

0 1 6

7 5 4

g(n): 5, h(n): 5, f(n): 10

Current board state:

2 8 3

1 5 6

7 4 0

g(n): 6, h(n): 4, f(n): 10

Current board state:

8 0 3

2 6 4

1 7 5

g(n): 3, h(n): 7, f(n): 10

Current board state:

2 0 3

1 8 6

7 5 4

g(n): 5, h(n): 5, f(n): 10

Current board state:

1 2 3

8 0 4

7 6 5

g(n): 6, h(n): 4, f(n): 10

Goal state reached!

Steps to solve with Misplaced Tiles heuristic: 6

Program-08 KnowledgeBase - Resolution

Algorithm

Lab-8

30-11-24
16

Knowledge base with resolution

function resolution(kb, query)

clause ← convert to CNF (kb)

add ¬ query to clause.

loop
new_clause ← {}

for each pair of clause:
resolvent ← ~~set~~ resolve (pair)

if resolvent == empty-clause:
return True

end if
add resolvent to new-clause
end for

if new-clause ⊆ clauses
return false
end if
add new-clauses to clauses
end repeat
end function

Code

```
def disjunctify(clauses):
    disjuncts = []
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
    return disjuncts

def getResolvant(ci, cj, di, dj):
    resolvant = list(ci) + list(cj)
    resolvant.remove(di)
    resolvant.remove(dj)
    return tuple(resolvant)

def resolve(ci, cj):
    for di in ci:
        for dj in cj:
            if di == '~' + dj or dj == '~' + di:
                return getResolvant(ci, cj, di, dj)

def checkResolution(clauses, query):
    clauses += [query if query.startswith('~') else '~' + query]
    proposition = '^'.join(['(' + clause + ')' for clause in clauses])
    print(f'Trying to prove {proposition} by contradiction. ..')

    clauses = disjunctify(clauses)
    resolved = False
    new = set()

    while not resolved:
        n = len(clauses)



pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]
for (ci, cj) in pairs:
resolvant = resolve(ci, cj)
if not resolvant:
resolved = True
break
new = new.union(set(resolvents))
if new.issubset(set(clauses)):
break
```

```
for clause in new:
if clause not in clauses:
clauses.append(clause)

if resolved:
print('Knowledge Base entails the query, proved by resolution')else:
print("Knowledge Base doesn't entail the query, no empty set produced after resolution") clauses
= input('Enter the clauses ').split()
query = input('Enter the query: ')
checkResolution(clauses, query)
```

Output Snapshot

```
Enter the clauses (~qv~pvr)^(~q^p)^q
Enter the query: r
Trying to prove ((~qv~pvr)^(~q^p)^q)^(~r) by contradiction....
Knowledge Base entails the query, proved by resolution
```

Program-09 Unification in first order logic

Algorithm

lab-6

Date 22-11-24
Page 10

unification algorithm:-

Algorithm: unify(Ψ_1, Ψ_2)

Step1: if Ψ_1 or Ψ_2 is a variable or constant than:
    if Ψ_1 or Ψ_2 are identical, than:
        return NIL
    else if Ψ_1 is a variable
        than
            if Ψ_1 occurs in Ψ_2 than
                return Failure
        else
            return {(Ψ_2/Ψ_1)}.
    else if Ψ_2 is a variable?
        if Ψ_2 occurs in Ψ_1 than
            return Failure
        else:
            return{(Ψ_1/Ψ_2)}

    else:
        return failure

Step2: if the initial predicate symbal in Ψ_1 & Ψ_2 are not same, than
    return Failure

Step3: if Ψ_1 & Ψ_2 have a diffrent no. of argument, than
    return failure
Step4: set substitution set(SUBST) to NIL

Code

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):



    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True
def getFirstPart(expression):
```

```
    attributes = getAttributes(expression)
    return attributes[0]
def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):




        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot
be unified")
        return []

    head1 = getFirstPart(exp1)
```

```
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    if attributeCount1 == 1:
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return []

    return initialSubstitution + remainingSubstitution

if __name__ == "__main__":
    print("Enter the first expression")
    e1 = input()



print("Enter the second expression")
e2 = input()
substitutions = unify(e1, e2)
print("The substitutions are:")
print([' / '.join(substitution) for substitution in substitutions])
```

Output Snapshot

```
Enter the first expression
king(x)
Enter the second expression
king(john)
The substitutions are:
['john / x']
```

Program-10 First Order Logic to Conjunctive Normal Form

Algorithm

Lab-8

Date 20-12-24
Page 13

convert FOL to CNF

function convert to CNF(stam):

Eliminate implication
move negation inward
Standardize Variable.
convert to Pernex form
remove existential quantifier via
Skolemization
Distribute ∧ over V
return CNF-form.
end function

Code

```
import re

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[[^]]+\]]', statement)
        for s in statements:
```

```
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement

def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+ statement[i+1:] + '=>' +
statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)



    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
```

```
            s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
            statement = ''.join(s)
    statement = statement.replace('~[∀','[~∀')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[∀V∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement

def main():
    print("Enter FOL:")
    fol = input()
    print("The CNF form of the given FOL is: ")
    print(Skolemization(fol_to_cnf(fol)))
    main()
```

Output Snapshot

```
Enter FOL:
∀x ∃(y)[P (x, y, z)]
The CNF form of the given FOL is:
y)[P (x, y, z)]
```

Program-10 Forward Reasoning

Algorithm

Lab-7

classmate
Date 29/11/24
Page 17

Forward Reasoning algo

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of
first-order logic clauses
α, the query, an atomic sentence

local variables: new, the new sentences
inferred on each iteration

repeat until new is empty
new ← { }
for each rule in KB do
$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow$ S-V(rule)
for each θ such that subst(θ, $p_1 \wedge \dots \wedge p_n$)
= subst(θ, $p_1' \wedge \dots \wedge p_n'$)
for some $p_1' \dots p_n'$ in KB
q' ← subst(θ, q)
if q' does not unify with some
sentence already in KB or new
then
add q' to new
φ ← unify(q', α)
if φ is not fail then
return φ
add new to KB
return false

29.11.24

Code

```
import re
def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:

    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')



        return [predicate, params]


    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in
self.params])})"
```

```
        return Fact(f)

class Implication:

    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None

class KB:

    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)
```

```
    def ask(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')

def main():
    kb = KB()
    print("Enter the number of FOL expressions present in KB:")
    n = int(input())
    print("Enter the expressions:")
    for i in range(n):
        fact = input()
kb.tell(fact)
print("Enter the query:")
query = input()
kb.ask(query)
kb.display()

main()
```

Output Snapshot

```
Querying criminal(x):
1. criminal(West)
All facts:
	1. american(West)
	2. sells(West,M1,Nono)
	3. owns(Nono,M1)
	4. missile(M1)
	5. enemy(Nono,America)
	6. weapon(M1)
	7. hostile(Nono)
	8. criminal(West)
Querying evil(x):
	1. evil(John)
```

Alpha-Beta Pruning:

Lab-8

classmate
Date 20-12-24
Page 24

Alpha, beta pruning.

Function alpha.beta(node, depth, alpha
beta. maxPlayer)

if depth==0 or node is terminal
return evaluate(node)
end if

if max player:

value ← -∞

for each child of node

value ← max(value, alpha.beta(
child. depth-1. alpha, beta, fals

alpha ← max(alpha, value

if beta<=alph
Break
return value

else
value ← ∞

for each child in node :
value ← min(value, alphabeta(child.
depth-1, alpha. beta, True)

beta ← min(beta, value)

Code:

```
# Define the possible moves for the game
def minimax(node, depth, alpha, beta, maximizingPlayer):
    if depth == 0 or is_terminal(node):
        return evaluate(node)

    if maximizingPlayer:
        maxEval = float('-inf')
        for child in get_children(node):
            eval = minimax(child, depth - 1, alpha, beta, False)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return maxEval
    else:
        minEval = float('inf')
        for child in get_children(node):
            eval = minimax(child, depth - 1, alpha, beta, True)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return minEval

# Dummy functions for evaluation, checking terminal nodes, and getting children nodes
# Replace these with actual implementations
def evaluate(node):
    # Evaluate the utility of the node
    return node.value

def is_terminal(node):
    # Check if the node is a terminal node
    return node.is_terminal

def get_children(node):
    # Get the children of the node
```

```
    return node.children

# Define a simple game tree node class for demonstration purposes
class Node:
    def __init__(self, value, is_terminal=False):
        self.value = value
        self.is_terminal = is_terminal
        self.children = []

# Example usage
if __name__ == "__main__":
    # Creating a simple game tree for demonstration
    root = Node(0)
    child1 = Node(3)
    child2 = Node(5)
    child3 = Node(6)
    child4 = Node(9, True)
    child5 = Node(1, True)
    child6 = Node(2, True)
    child7 = Node(0, True)

    root.children = [child1, child2, child3]
    child1.children = [child4, child5]
    child2.children = [child6]
    child3.children = [child7]

    # Run the alpha-beta pruning algorithm
    optimal_value = minimax(root, 3, float('-inf'), float('inf'), True)
    print("The optimal value is:", optimal_value)
```

program:
simulated anneling:

Lab-5

classmate
Date 15-11-2024
Page 9

Simulated anneling - Basic algorithm.

```
function simulated anneling()
  current ← randomly generated intial state
  current.cost ← cost(current) # conflicts
  T ← a large positive value
  while T>0 and current-cost>0
        neighbour ← generated neighbore of
        ~~cost-diff ← current.cost neighbor cost current~~ . state
        if ~~cost-diff>0~~ : current cost < nighbore cost
            current ← nighbore
            current cost ← nighbore_cost
     T=T-1
     end while
  return current, current_state.

function cost(state)
      conflicts = 0
      n ← len(state)
      for i range(n)
          for j range(i+1,n)
          if state(i)==state(j)
              conflict +=1
return conflicts
```

Code:

```
import random
import math

# Function to generate an initial random state (placement of queens)
def initial_state(N):
   return [random.randint(0, N-1) for _ in range(N)]  # Random row positions for each
column

# Function to compute the cost (number of conflicts) of a given state
def cost(state):
   conflicts = 0
   N = len(state)
   for i in range(N):
      for j in range(i + 1, N):
         # Check if queens share the same row or diagonal
         if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
            conflicts += 1
   return conflicts

# Function to generate a neighbouring state by randomly moving one queen
def generate_neighbour(state):
   new_state = state[:]
   col = random.randint(0, len(state) - 1)
   new_row = random.randint(0, len(state) - 1)
   new_state[col] = new_row
   return new_state

# Simulated Annealing function
def simulated_annealing(N, initial_temp=1000, alpha=0.95, max_iter=10000):
   current_state = initial_state(N)
   current_cost = cost(current_state)
   temp = initial_temp
   iteration = 0

   while current_cost > 0 and iteration < max_iter:
      neighbour = generate_neighbour(current_state)
      neighbour_cost = cost(neighbour)

      delta_cost = neighbour_cost - current_cost

      # Accept the neighbour with probability depending on temperature
      if delta_cost < 0 or random.random() < math.exp(-delta_cost / temp):
```

```
            current_state = neighbour
            current_cost = neighbour_cost

        # Decrease temperature
        temp *= alpha
        iteration += 1

    return current_state, current_cost

# Function to print the solution as a matrix (chessboard representation)
def print_solution(state):
    N = len(state)
    board = [['.' for _ in range(N)] for _ in range(N)]

    # Place queens on the board (represented as 'Q')
    for col, row in enumerate(state):
        board[row][col] = 'Q'

    # Print the board
    for row in board:
        print(' '.join(row))

# Example usage
N = int(input("Enter the number: "))
solution, cost_value = simulated_annealing(N)

if cost_value == 0:
    print(f"Solution found: {solution}")
    print_solution(solution)  # Print the solution as a matrix if found
else:
    print(f"No solution found. Final cost: {cost_value}")
```

Program :
Hill-Climbing

Lab-4

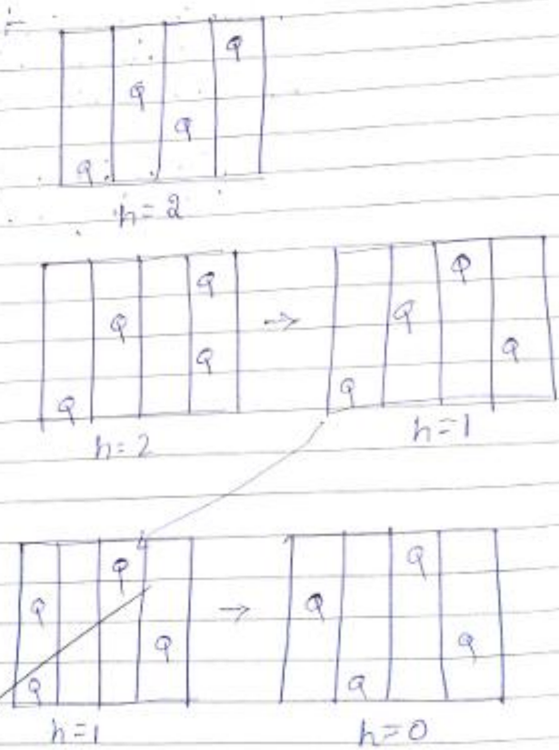
Date 8/11/24
Page 8

Hill-climbing search algorithm :-

function Hill-climbing (problem) return a state that is a local maximum
current ← make-node (problem, initial-state)

loop do
neighbor ← a highest-valued successor of current
if neighbor.value ≤ current.value
then return current.state
current ← neighbor

problem :-



Code:

```
import random

def get_user_board(n):
   board = []
   print(f"Enter the initial row positions for each column (0 to {n-1}):")
   for col in range(n):
      row = int(input(f"Column {col + 1}: "))
      if 0 <= row < n:
         board.append(row)
      else:
         print("Invalid input. Row must be between 0 and", n - 1)
         return None
   return board

def heuristic(board):
   n = len(board)
   attacks = 0
   for i in range(n):
      for j in range(i + 1, n):
         if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
            attacks += 1
   return attacks

def get_neighbors(board):
   neighbors = []
   n = len(board)
   for col in range(n):
      for row in range(n):
         if board[col] != row:
            neighbor = board[:]
            neighbor[col] = row
            neighbors.append(neighbor)
   return neighbors

def print_board(board):
   """Prints the board visually, showing 'Q' for queens and '.' for empty spaces."""
   n = len(board)
   for row in range(n):
      line = ""
      for col in range(n):
         if board[col] == row:
            line += "Q "
```

```
            else:
                line += ". "
        print(line)
    print("\n")

def hill_climbing_with_restarts(n, initial_board):
    current = initial_board
    restarts = 0
    heuristic_evaluations = 0
    iteration = 1

    while True:
        print(f"\nRestart #{restarts + 1}")
        while True:
            current_heuristic = heuristic(current)
            heuristic_evaluations += 1
            print(f"Iteration {iteration}: Heuristic = {current_heuristic}")
            print_board(current)  # Print the current board visually
            iteration += 1

            if current_heuristic == 0:
                print(f"Solution found!\nTotal restarts: {restarts}\nTotal heuristic
evaluations: {heuristic_evaluations}")
                return current

            neighbors = get_neighbors(current)
            best_neighbor = min(neighbors, key=heuristic)
            best_neighbor_heuristic = heuristic(best_neighbor)
            heuristic_evaluations += len(neighbors)

            if best_neighbor_heuristic >= current_heuristic:
                print("Stuck in a local minimum, restarting...\n")
                break  # Local minimum reached, so we restart

            current = best_neighbor

        # Restart with a new random board if stuck
        current = generate_board(n)
        restarts += 1

def generate_board(n):
    return [random.randint(0, n - 1) for _ in range(n)]

# Main execution
```

```
n = int(input("Enter the number of queens (e.g., 4 for 4-Queens): "))
initial_board = get_user_board(n)

if initial_board:
    solution = hill_climbing_with_restarts(n, initial_board)
    print("Final Solution:")
    print_board(solution)
    print("Attacking pairs:", heuristic(solution))
else:
    print("Invalid initial board configuration.")
```