VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Kshitiz Anand (1BM22CS349)

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

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 **Kshitiz Anand(1BM22CS349),** who is 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.

Namratha M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Github Link: https://github.com/KshitizAnandBMS/AI_LAB

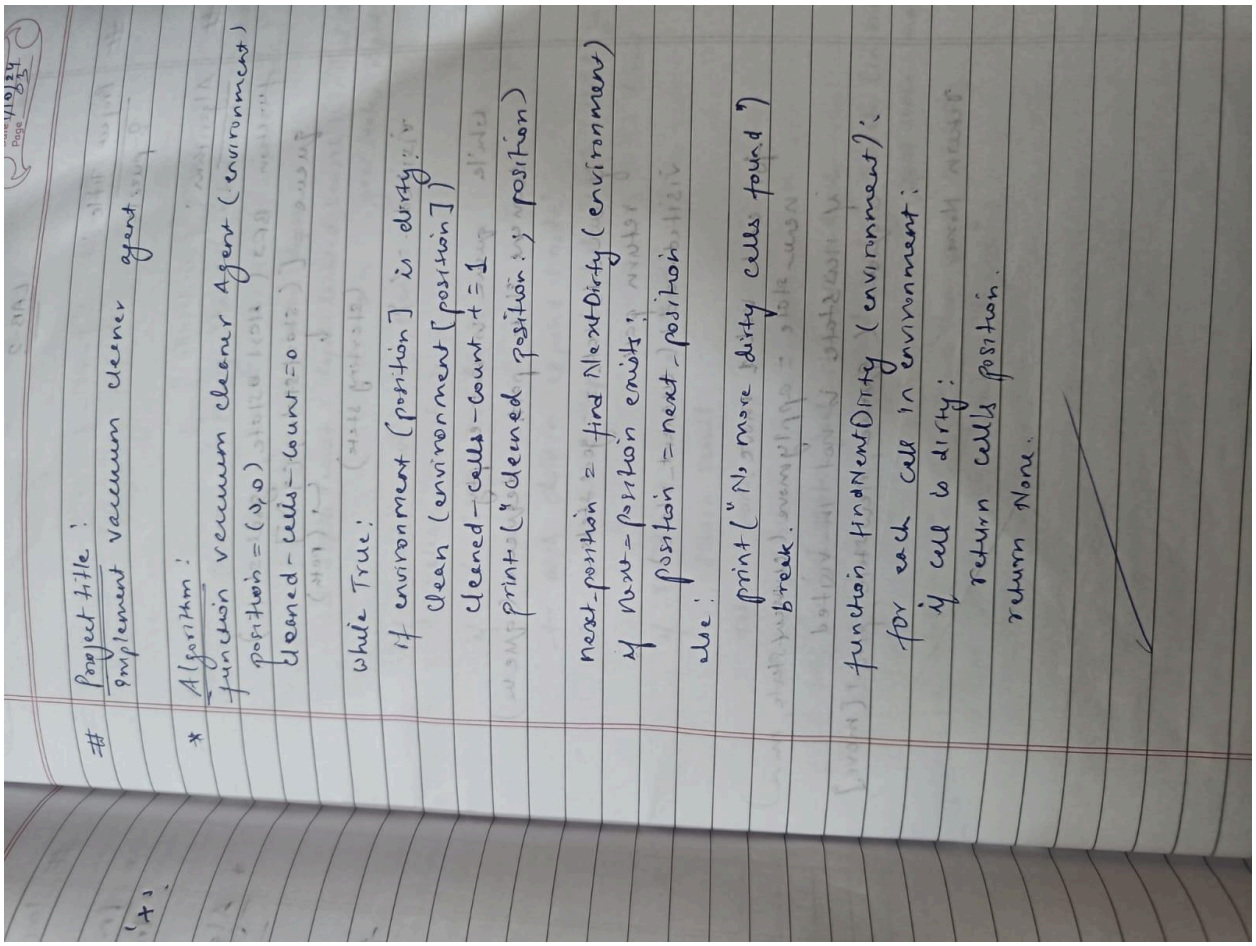
Index

AI OBSERVATION BOOK

LAB-01

Implementation of a Vacuum Cleaner Agent

ALGORITHM:



CODE:

```
def vacuum_world():
    # initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum: ")  # user_input of
location vacuum is placed
```

```
    status_input = input("Enter status of " + location_input + " (0 for
Clean, 1 for Dirty): ")  # user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room: ")

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

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

        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            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 the Location B.")
                cost += 1  # cost for moving right
                print("COST for moving RIGHT: " + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1  # cost for suck
                print("COST for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("No action. " + str(cost))
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean.")

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

```
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location B is already clean.")

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

    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt and mark it as clean
        goal_state['B'] = '0'
        cost += 1  # cost for suck
        print("COST for CLEANING: " + 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 the Location A.")
            cost += 1  # cost for moving left
            print("COST for moving LEFT: " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("COST for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")

    if status_input_complement == '1':  # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A.")
        cost += 1  # cost for moving left
        print("COST for moving LEFT: " + str(cost))
        # suck the dirt and mark it as clean
```

```
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("No action. " + str(cost))
            print("Location A is already clean.")

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


vacuum_world()
```

OUTPUT:

```
Enter Location of Vacuum: B
Enter status of B (0 for Clean, 1 for Dirty): 1
Enter status of other room: 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location B
Location B is Dirty.
COST for CLEANING: 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT: 2
COST for SUCK: 3
Location A has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT: 4
Cost for SUCK: 5
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 5
```

Implementation of Tic Tac Toe Game:

ALGORITHM:

LAB 1

Implement a tic-tac-toe game

* Program title:
Implement a tic-tac-toe game.

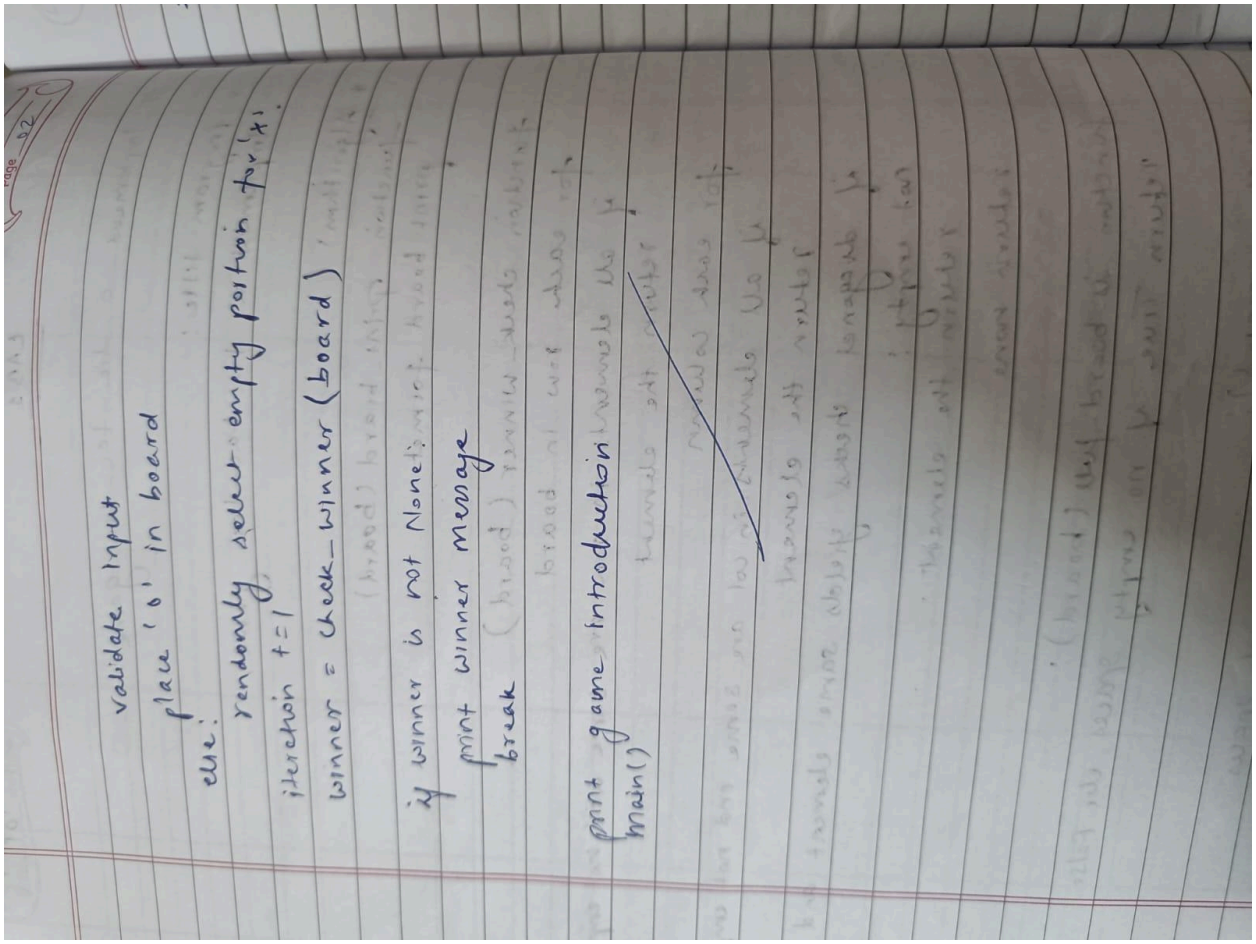
* Algorithm:

function print_board (board)
print board format

function check_winner (board)
for each row in board
if all elements in row are same and not empty:
return the element
for each column
if all elements in col are same and not empty:
return the element
if diagonal checks yields same element and
not empty:
return the element.
return none

function is_board_full (board):
Return True if no empty spaces, else False.

function main():
initialise board with empty spaces
iteration = 0
winner = none
while winner is none:
if iteration is even:
print_board (board)
get user input for row & column.



CODE:

```
board={1:' ',2:' ',3:' ',
      4:' ',5:' ',6:' ',
      7:' ',8:' ',9:' '

}

def printBoard(board):
   print(board[1]+'|'+board[2]+'|'+board[3])
   print('-+-+-')
   print(board[4] + '|' + board[5] + '|' + board[6])
   print('-+-+-')
   print(board[7] + '|' + board[8] + '|' + board[9])
   print('\n')

def spaceFree(pos):
```

```
    if(board[pos]==' '):
        return True
    else:
        return False

def checkWin():
    if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
        return True
    elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
        return True
    elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
        return True
    elif (board[1] == board[5] and board[1] == board[9] and board[1] != '
'):
        return True
    elif (board[3] == board[5] and board[3] == board[7] and board[3] != '
'):
        return True
    elif (board[1] == board[4] and board[1] == board[7] and board[1] != '
'):
        return True
    elif (board[2] == board[5] and board[2] == board[8] and board[2] != '
'):
        return True
    elif (board[3] == board[6] and board[3] == board[9] and board[3] != '
'):
        return True
    else:
        return False

def checkMoveForWin(move):
    if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
```

```
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'
```

```
def playerMove():
   position=int(input('Enter position for O:'))
   insertLetter(player, position)
   return

def compMove():
   bestScore=-1000
   bestMove=0
   for key in board.keys():
       if (board[key]==' '):
           board[key]=bot
           score = minimax(board, False)
           board[key] = ' '
           if (score > bestScore):
               bestScore = score
               bestMove = key

   insertLetter(bot, bestMove)
   return


def minimax(board, isMaximizing):
   if (checkMoveForWin(bot)):
       return 1
   elif (checkMoveForWin(player)):
       return -1
   elif (checkDraw()):
       return 0

   if isMaximizing:
       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:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
                    bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

OUTPUT:

```
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O:5
X| |
-+-+-
 |O|
-+-+-
 | |


X|X|
-+-+-
 |O|
-+-+-
 | |
```

```
Enter position for O:3
X|X|O
-+-+-
 |O|
-+-+-
 | |


X|X|O
-+-+-
 |O|
-+-+-
X| |


Enter position for O:4
X|X|O
-+-+-
O|O|
-+-+-
X| |
```

```
X|X|O
-+-+-
O|O|X
-+-+-
X| |


Enter position for O:8
X|X|O
-+-+-
O|O|X
-+-+-
X|O|


X|X|O
-+-+-
O|O|X
-+-+-
X|O|X


Draw!
```

LAB-02
Implementation of 8 puzzles games using BFS and DFS

ALGORITHM:

LAB-2

Project Title:
8-puzzle

Algorithm:

function BFS (start state, goal state)
queue = [(start state, [ ])]
(starting state) (path)

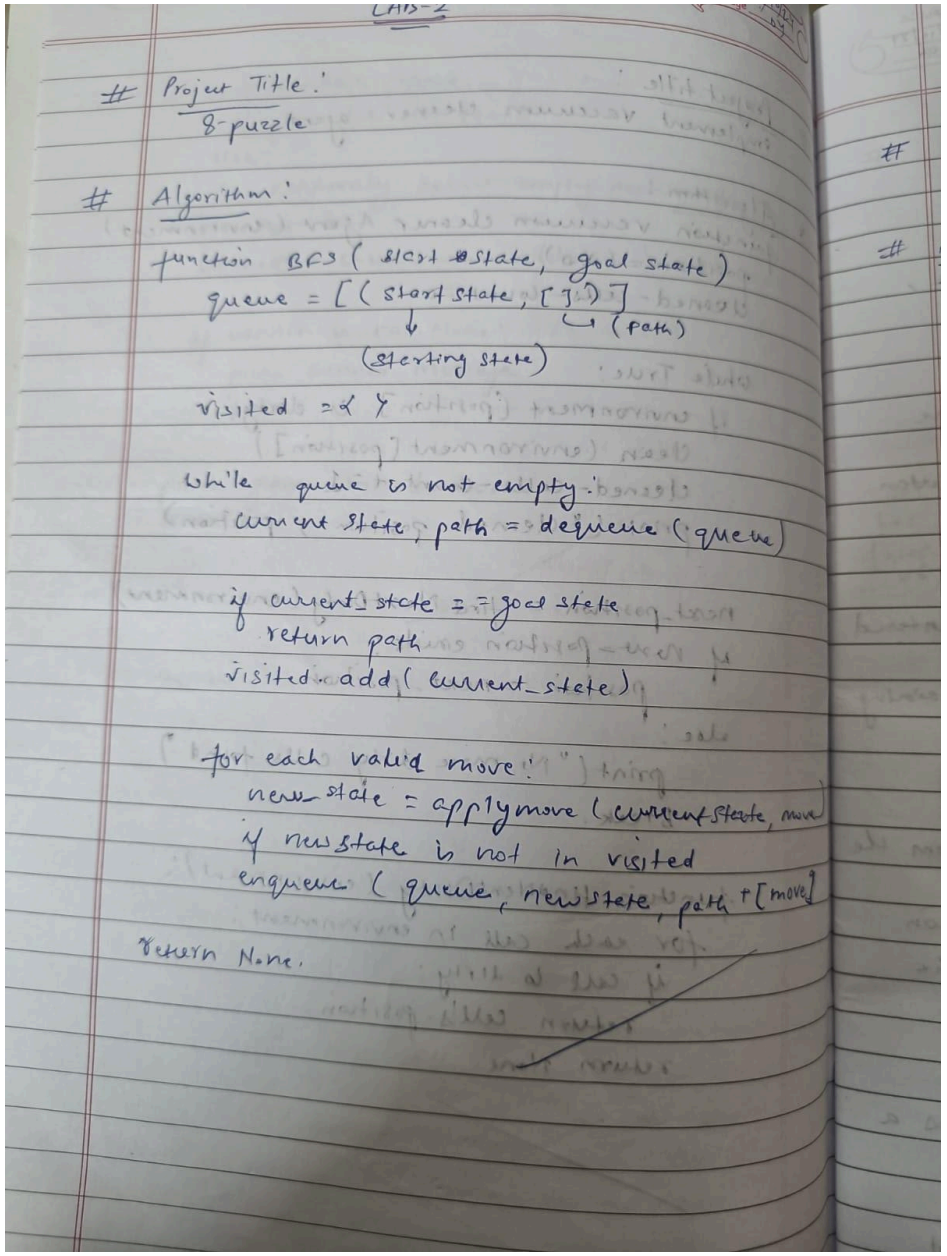
visited = { }

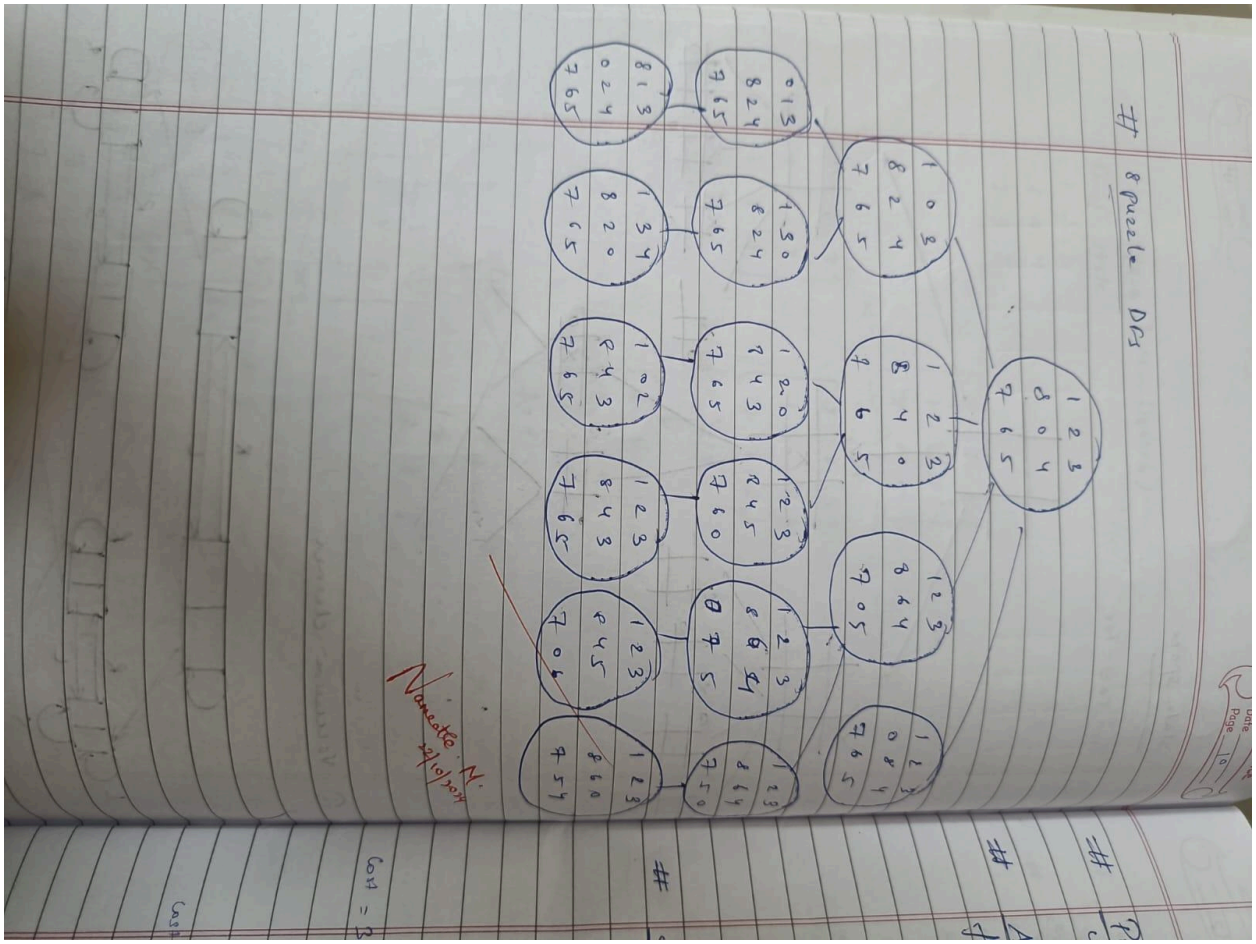
while queue is not empty:
current state, path = dequeue (queue)

if current_state == goal state
return path
visited.add (current_state)

for each valid move:
new_state = applymove (current state, move)
if new state is not in visited
enqueue (queue, new state, path + [move])

return None.





CODE-BFS

```
import numpy as np
from collections import deque

class Node:
    def __init__(self, state, parent, action):
        self.state = state  # The puzzle configuration (as a 2D array)
        self.parent = parent  # Parent node (used to trace back the solution)
        self.action = action  # Action taken to reach this state (e.g., 'up', 'down')

class Puzzle:
    def __init__(self, start, start_index, goal, goal_index):
        self.start = [start, start_index]  # Starting puzzle state and blank tile index
        self.goal = [goal, goal_index]  # Goal puzzle state and blank tile index
```

```
    self.solution = None  # To store the final solution path

def neighbors(self, state):
    mat, (row, col) = state  # Puzzle state and the position of the blank tile (0)
    results = []

    # Move the blank tile up
    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)]))

    # Move the blank tile left
    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)]))

    # Move the blank tile down
    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)]))

    # Move the blank tile right
    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 solve_bfs(self):
    """Breadth-First Search (BFS) implementation for solving the 8-puzzle."""
    # Initialize the frontier with the starting node (queue)
    start = Node(state=self.start, parent=None, action=None)
    frontier = deque([start])  # Using deque for fast popping from the left (FIFO)

    explored = []  # List to keep track of explored states
```

```
    while frontier:
        # Dequeue the node (FIFO)
        node = frontier.popleft()

        # Check if the current state is the goal state
        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.solution

        # Add the current state to the explored set
        explored.append(node.state)

        # Expand the neighbors of the current state
        for action, neighbor in self.neighbors(node.state):
            # If the neighbor state has not been explored and is not in the frontier
            if not any((explored_state[0] == neighbor[0]).all() for explored_state in explored):
                if not any((frontier_node.state[0] == neighbor[0]).all() for frontier_node in frontier):
                    # Add the neighbor to the frontier
                    child_node = Node(state=neighbor, parent=node, action=action)
                    frontier.append(child_node)

    # No solution found
    raise Exception("No solution found.")

def print_solution(self):
    if self.solution is None:
        print("No solution available.")
        return

    # Print the solution path
    print("Initial State:")
    print(self.start[0], "\n")

    for action, state in zip(self.solution[0], self.solution[1]):
        print(f"Action: {action}")
        print(state[0], "\n")
```

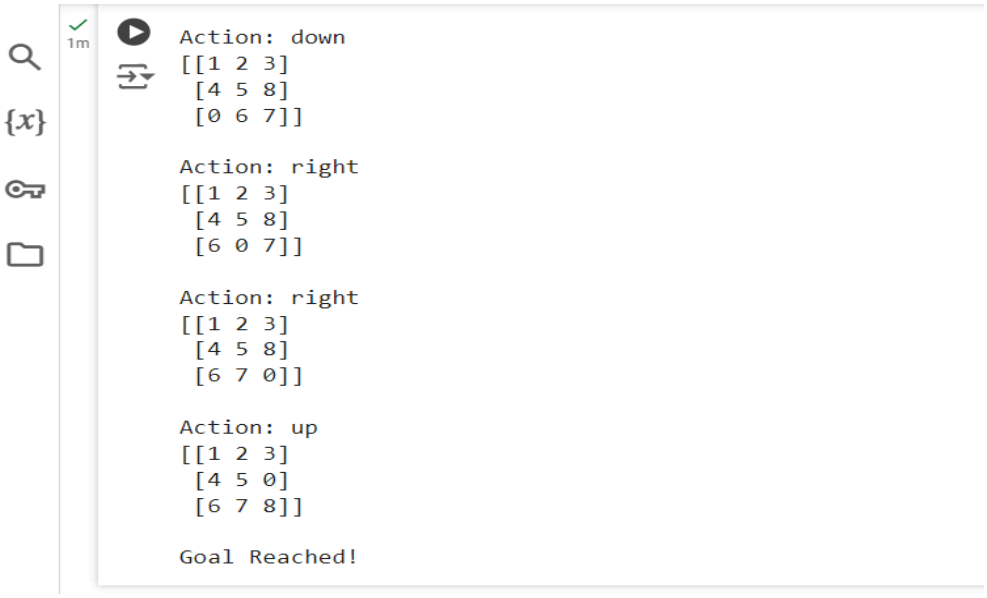
```
        print("Goal Reached!")


# Function to take user input and convert it to a 3x3 matrix
def get_puzzle_input():
    print("Enter the 8-puzzle configuration (use 0 for the blank space):")
    puzzle = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (space-separated numbers): ").split()
        puzzle.append([int(num) for num in row])
    return np.array(puzzle)

# Get user input for start and goal states
start = get_puzzle_input()
goal = get_puzzle_input()

# Find the index of the blank tile (0) in the initial and goal states
start_index = (np.where(start == 0)[0][0], np.where(start == 0)[1][0])
goal_index = (np.where(goal == 0)[0][0], np.where(goal == 0)[1][0])

# Create the puzzle object
puzzle = Puzzle(start, start_index, goal, goal_index)

# Solve the puzzle using BFS
puzzle.solve_bfs()

# Print the solution path
puzzle.print_solution()
```

OUPUT- BFS

```
Enter the 8-puzzle configuration (use 0 for the blank space):
Enter row 1 (space-separated numbers): 1 2 3
Enter row 2 (space-separated numbers): 4 5 6
Enter row 3 (space-separated numbers): 7 8 0
Enter the 8-puzzle configuration (use 0 for the blank space):
Enter row 1 (space-separated numbers): 1 2 3
Enter row 2 (space-separated numbers): 4 5 0
Enter row 3 (space-separated numbers): 6 7 8
Initial State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]

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

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

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

1m

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

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

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

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

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

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



CODE-DFS

```
import numpy as np
from collections import deque

class Node:
    def __init__(self, state, parent, action):
        self.state = state  # The puzzle configuration (as a 2D array)
        self.parent = parent  # Parent node (used to trace back the solution)
        self.action = action  # Action taken to reach this state (e.g., 'up', 'down')

class Puzzle:
    def __init__(self, start, start_index, goal, goal_index):
        self.start = [start, start_index]  # Starting puzzle state and blank tile index
        self.goal = [goal, goal_index]  # Goal puzzle state and blank tile index
        self.solution = None  # To store the final solution path
```

```
def neighbors(self, state):
    mat, (row, col) = state  # Puzzle state and the position of the blank tile (0)
    results = []

    # Move the blank tile up
    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)]))

    # Move the blank tile left
    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)]))

    # Move the blank tile down
    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)]))

    # Move the blank tile right
    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 solve_dfs(self):
    """Depth-First Search (DFS) implementation for solving the 8-puzzle."""
    # Initialize the frontier with the starting node (stack)
    start = Node(state=self.start, parent=None, action=None)
    frontier = [start]  # Using a list as a stack (LIFO)

    explored = []  # List to keep track of explored states

    while frontier:
```

```
        # Pop the node from the stack (LIFO)
        node = frontier.pop()

        # Check if the current state is the goal state
        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.solution

        # Add the current state to the explored set
        explored.append(node.state)

        # Expand the neighbors of the current state
        for action, neighbor in self.neighbors(node.state):
            # If the neighbor state has not been explored and is not in the frontier
            if not any((explored_state[0] == neighbor[0]).all() for explored_state in explored):
                if not any((frontier_node.state[0] == neighbor[0]).all() for frontier_node in frontier):
                    # Add the neighbor to the frontier
                    child_node = Node(state=neighbor, parent=node, action=action)
                    frontier.append(child_node)

    # No solution found
    raise Exception("No solution found.")

def print_solution(self):
    if self.solution is None:
        print("No solution available.")
        return

    # Print the solution path
    print("Initial State:")
    print(self.start[0], "\n")

    for action, state in zip(self.solution[0], self.solution[1]):
        print(f"Action: {action}")
        print(state[0], "\n")
```

```
        print("Goal Reached!")


# Function to take user input and convert it to a 3x3 matrix
def get_puzzle_input():
    print("Enter the 8-puzzle configuration (use 0 for the blank space):")
    puzzle = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (space-separated numbers): ").split()
        puzzle.append([int(num) for num in row])
    return np.array(puzzle)

# Get user input for start and goal states
start = get_puzzle_input()
goal = get_puzzle_input()

# Find the index of the blank tile (0) in the initial and goal states
start_index = (np.where(start == 0)[0][0], np.where(start == 0)[1][0])
goal_index = (np.where(goal == 0)[0][0], np.where(goal == 0)[1][0])

# Create the puzzle object
puzzle = Puzzle(start, start_index, goal, goal_index)

# Print the initial state before solving
print("Initial State:")
print(start, "\n")

# Solve the puzzle using DFS
puzzle.solve_dfs()

# Print the solution path
puzzle.print_solution()
```

OUPUT-DFS

```
Enter the 8-puzzle configuration (use 0 for the blank space):
Enter row 1 (space-separated numbers): 1 2 3
Enter row 2 (space-separated numbers): 4 5 6
Enter row 3 (space-separated numbers): 7 8 0
Enter the 8-puzzle configuration (use 0 for the blank space):
Enter row 1 (space-separated numbers): 1 2 3
Enter row 2 (space-separated numbers): 4 5 6
Enter row 3 (space-separated numbers): 0 7 8
Initial State:
[[1 2 3]
 [4 5 6]
 [7 8 0]]

Initial State:
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)

Action: left
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)

Action: left
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)

Goal Reached!
```

Lab-03
Applying A* algorithm to solve 8-puzzle problem

ALGORITHM:

LAB-3

A* algorithm for 8-puzzle problem:
$f(n) = g(n) + h(n)$

Algorithm & pseudocode:
To implement the A* algorithm for the 8-puzzle, we define a heuristic $h(n)$:
① No of misplaced tiles
② Manhattan Distance
The A* algo will use $f(n) = g(n) + h(n)$, where $g(n)$ is the depth of the node
$h(n)$ is heuristics, whether misplaced tile or manhattan.

- Algo

Initialise
→ start with initial state of the puzzle.
→ Use a priority queue to hold the nodes, entered by $f(n) = g(n) + h(n)$
→ Keep a set of visited nodes to avoid re-exploring nodes.

Expend nodes
→ while the priority queue is not empty.
→ Rename the node with the lowest $f(n)$ from the queue
→ If this node is the goal, return the solution
Generate all possible child node by moving the blank space
→ for each child, calculate $g(n)$
• calculate $h(n)$ by chosen state
If the child has not been visited or if it has a better $f(n)$ than before, add it to the queue

Termination
→ If the goal is found, return the path and total cost
→ If no solution is found, return failure.

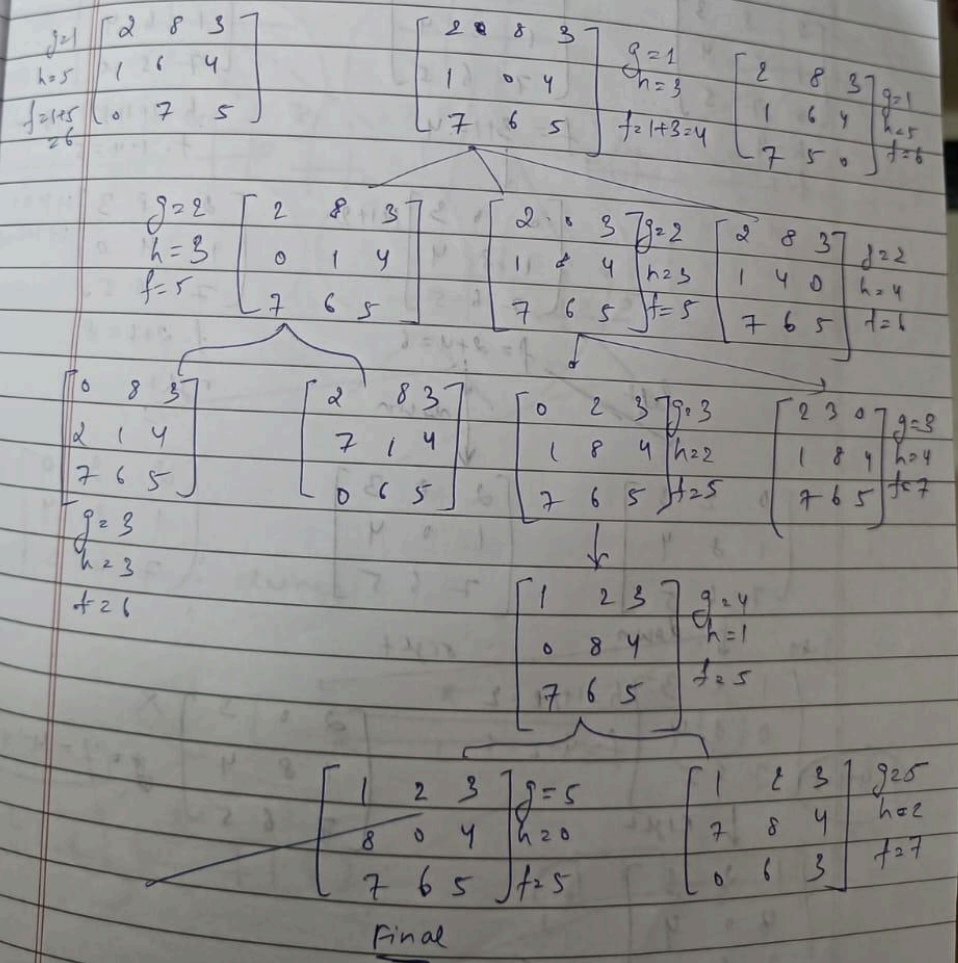
State space tree (misplaced tiles)

Final state:

1	2	3
8		4
7	6	5

Initial state

2	8	3
1	6	4
7	0	5



State space (Manhattan)

Final state

1	2	3
8	0	4
7	6	5

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{bmatrix} \quad h = 1+1+0+0+0+1+0+1 = 4$$

left, up, right

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 6 & 7 & 5 \end{bmatrix} \quad h=4, \; f=5$$

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad f = 3+1 = 4$$

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 5 & 0 \end{bmatrix} \quad h = 1+1+1+1 = 4, \; f = 1+4 = 5$$

$$\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \; 1+1+2$$

$$f = 2+4 = 6$$

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 4 & 0 \\ 7 & 6 & 5 \end{bmatrix} \; 1+1+0+1+1+2$$

$$f = 2+6 = 8$$

left, down, right

$$\begin{bmatrix} 0 & 2 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad g=4$$

$$\begin{bmatrix} 2 & 8 & 3 \\ 1 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 3 & 0 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

down, right

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \quad h = 1+1+1 = 2, \; f = 4+2 = 6, \; f = 4+1 = 5$$

$$\begin{bmatrix} 2 & 0 & 3 \\ 1 & 8 & 4 \\ 7 & 6 & 5 \end{bmatrix} \times \quad f = 4+4 = 8$$

right

$$\begin{bmatrix} 1 & 2 & 3 \\ 8 & 0 & 4 \\ 7 & 6 & 5 \end{bmatrix}$$

final state

Namrata M. 15/10/2024

CODE:

Misplaced Tiles

```
import heapq

def misplaced_tiles(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j] and state[i][j] != 0:
                count += 1
    return count

def get_empty_tile_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def generate_new_states(state):
    i, j = get_empty_tile_position(state)
    possible_moves = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for di, dj in directions:
        new_i, new_j = i + di, j + dj
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state]

            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            possible_moves.append(new_state)
    return possible_moves


def print_puzzle(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()


def a_star(start, goal):
    queue = []
    heapq.heappush(queue, (misplaced_tiles(start, goal), 0, start, []))

    visited = set()
```

```
    visited.add(tuple(tuple(row) for row in start))

    while queue:
        f, g, current_state, path = heapq.heappop(queue)

        print(f"Step {g}:")
        print(f"g(n) = {g}, h(n) = {f - g}")
        print("Current puzzle state:")
        print_puzzle(current_state)

        if current_state == goal:
            print("Goal state reached!")
            print(f"Total moves: {g}")
            print("Solution path:")
            for step, state in enumerate(path + [current_state], 1):
                print(f"Move {step}:")
                print_puzzle(state)
            return

        print("Generated new states (possible moves):")
        trial_states = []
        for new_state in generate_new_states(current_state):
            if tuple(tuple(row) for row in new_state) not in visited:
                h = misplaced_tiles(new_state, goal)
                trial_states.append((g + h + 1, g + 1, new_state, h))
                print(f"g(n) = {g + 1}, h(n) = {h}")
                print_puzzle(new_state)

        print("Evaluating and choosing the best state based on f(n):")
        for f_new, g_new, state, h_new in trial_states:
            if tuple(tuple(row) for row in state) not in visited:
                heapq.heappush(queue, (f_new, g_new, state, path + [current_state]))
                visited.add(tuple(tuple(row) for row in state))
                print(f"Chosen state with f(n) = {f_new} (g(n) = {g_new}, h(n) = {h_new}):")
                print_puzzle(state)

    print("No solution found")
    return

start_state = [[2, 8, 3],
               [1, 6, 4],
               [7,0, 5]]

goal_state = [[1, 2, 3],
```

```
    [8, 0, 4],
    [7, 6, 5]]
```

a_star(start_state, goal_state)

OUTPUT:

```
Goal state reached!
Total moves: 5
Solution path:
Move 1:
2 8 3
1 6 4
7 _ 5

Move 2:
2 8 3
1 _ 4
7 6 5

Move 3:
2 _ 3
1 8 4
7 6 5

Move 4:
_ 2 3
1 8 4
7 6 5

Move 5:
1 2 3
_ 8 4
7 6 5

Move 6:
1 2 3
8 _ 4
7 6 5
```

CODE:

Manhattan distance

```
import heapq


def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                # Find the goal position of the current tile
                goal_x, goal_y = divmod(goal.index(state[i][j]), 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def get_empty_tile_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def generate_new_states(state):
    i, j = get_empty_tile_position(state)
    possible_moves = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for di, dj in directions:
        new_i, new_j = i + di, j + dj
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state]
            # Swap the empty tile with the neighboring tile
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            possible_moves.append(new_state)
    return possible_moves

def print_puzzle(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

def a_star(start, goal):

    goal_flat = [num for row in goal for num in row]


    queue = []
    heapq.heappush(queue, (manhattan_distance(start, goal_flat), 0, start, []))
```

```
    visited = set()
    visited.add(tuple(tuple(row) for row in start))

    while queue:
        f, g, current_state, path = heapq.heappop(queue)

        print(f"Step {g}:")
        print(f"g(n) = {g}, h(n) = {f - g}")
        print("Current puzzle state:")
        print_puzzle(current_state)


        if current_state == goal:
            print("Goal state reached!")
            print(f"Total moves: {g}")
            print("Solution path:")
            for step, state in enumerate(path + [current_state], 1):
                print(f"Move {step}:")
                print_puzzle(state)
            return

        print("Generated new states (possible moves):")
        trial_states = []
        for new_state in generate_new_states(current_state):
            if tuple(tuple(row) for row in new_state) not in visited:
                h = manhattan_distance(new_state, goal_flat)
                trial_states.append((g + h + 1, g + 1, new_state, h))
                print(f"g(n) = {g + 1}, h(n) = {h}")
                print_puzzle(new_state)

        print("Evaluating and choosing the best state based on f(n):")
        for f_new, g_new, state, h_new in trial_states:
            if tuple(tuple(row) for row in state) not in visited:
                heapq.heappush(queue, (f_new, g_new, state, path + [current_state]))
                visited.add(tuple(tuple(row) for row in state))
                print(f"Chosen state with f(n) = {f_new} (g(n) = {g_new}, h(n) = {h_new}):")
                print_puzzle(state)

    print("No solution found")
    return

start_state = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 5, 8]
]

goal_state = [
```

```
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

a_star(start_state, goal_state)
```

OUTPUT:

```
Goal state reached!
Total moves: 2
Solution path:
Move 1:
1 2 3
4 _ 6
7 5 8

Move 2:
1 2 3
4 5 6
7 _ 8

Move 3:
1 2 3
4 5 6
7 8 _
```

LAB-04
Hill Climb algorithm to solve n-queen problem

ALGORITHM:

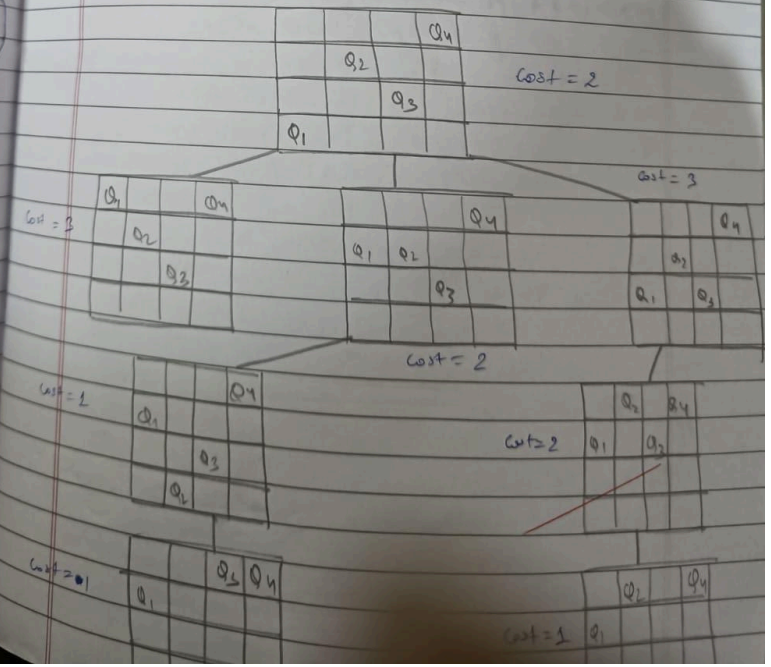
LAB-4

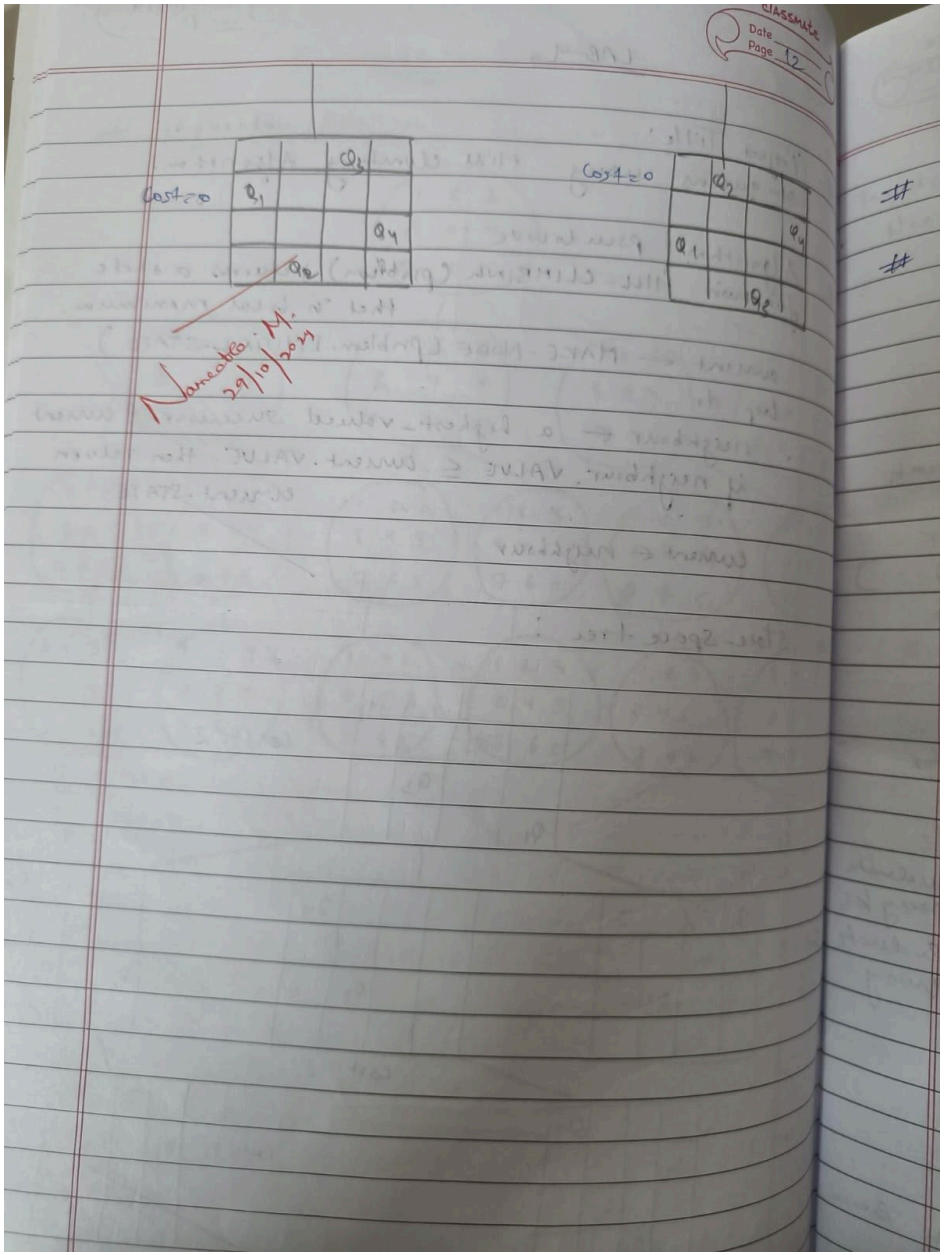
Date 23/10/24
Page 11

Project Title:
4-queens using Hill Climbing Algorithm.

Algorithm / pseudocode:
function HILL-CLIMBING (problem) returns a state that is local maximum.
current ← MAKE-NODE (problem. INITIAL-STATE)
loop do
neighbour ← a highest-valued successor of current
if neighbour. VALUE ≤ current. VALUE then return current. STATE
current ← neighbour

State-Space tree:




Cost=∞
Q1
Q2
Q3
Q4
Cost=0
Q1
Q2
Q3
Q4
Namaste : M.
29/10/2024

CODE:

```
import random

def heuristic(state):
    """Calculate the number of conflicts between queens."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def generate_neighbors(state):
    """Generate all neighboring states by swapping two queens."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):

            new_state = state.copy()
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(new_state)
    return neighbors

def print_board(state):
    """Print the board configuration."""
    n = len(state)
    board = [["." for _ in range(n)] for _ in range(n)]
    for row in range(n):
        board[row][state[row]] = "Q"
    for line in board:
        print(" ".join(line))
    print()

def get_user_input(n):
    """Get initial state from user input."""
    while True:
        try:
            input_state = input(f"Enter the initial positions for {n} queens (0
to {n-1}, space-separated): ")
            positions = list(map(int, input_state.split()))
            if len(positions) != n or any(p < 0 or p >= n for p in positions):
                raise ValueError
            return positions
        except ValueError:
            print("Invalid input. Please enter exactly {} numbers between 0 and
{}.".format(n, n - 1))
```

```
def hill_climbing(n):
    """Perform the Hill-Climbing algorithm using the swapping technique."""
    current_state = get_user_input(n)
    current_cost = heuristic(current_state)

    print("Initial State:")
    print_board(current_state)
    print(f"Initial Heuristic (Conflicts): {current_cost}\n")

    while current_cost > 0:
        neighbors = generate_neighbors(current_state)
        next_state = None
        next_cost = current_cost

        for neighbor in neighbors:
            cost = heuristic(neighbor)
            print(f"Evaluating Neighbor:")
            print_board(neighbor)
            print(f"Heuristic (Conflicts): {cost}")

            if cost < next_cost:
                next_cost = cost
                next_state = neighbor

        if next_state is None:
            print("Local maximum reached. No better neighbors found.")
            break


        print("Moving to Next State:")
        current_state = next_state
        current_cost = next_cost
        print_board(current_state)
        print(f"Heuristic (Conflicts): {current_cost}\n")

    return current_state, current_cost

n = 4
solution, solution_cost = hill_climbing(n)
print("Best Solution Found:")
print_board(solution)
print(f"Final Heuristic (Conflicts): {solution_cost}")
```

OUTPUT:

Enter the initial positions for 4 queens (0 to 3, space-separated): 3 1 2 0
Initial State:

. . . Q
. Q . .
. . Q .
Q . . .

Initial Heuristic (Conflicts): 2

Evaluating Neighbor:
. Q . .
. . . Q
. . Q .
Q . . .

Heuristic (Conflicts): 1
Evaluating Neighbor:
. . Q .
. Q . .
. . . Q
Q . . .

Heuristic (Conflicts): 1
Evaluating Neighbor:
Q . . .
. Q . .
. . Q .
. . . Q

Heuristic (Conflicts): 6
Evaluating Neighbor:
. . . Q
. . Q .
. Q . .
Q . . .

Heuristic (Conflicts): 6
Evaluating Neighbor:
. . . Q
Q . . .
. . Q .
. Q . .

Heuristic (Conflicts): 1
Evaluating Neighbor:
. . . Q
. Q . .
Q . . .
. . Q .

Heuristic (Conflicts): 1

Moving to Next State:
. Q . .
. . . Q
. . Q .
Q . . .

Heuristic (Conflicts): 1

Evaluating Neighbor:
. . . Q
. Q . .
. . Q .
Q . . .

Heuristic (Conflicts): 2
Evaluating Neighbor:
. . Q .
. . . Q
. Q . .
Q . . .

Heuristic (Conflicts): 2
Evaluating Neighbor:
Q . . .
. . . Q
. . Q .
. Q . .

Heuristic (Conflicts): 4
Evaluating Neighbor:
. Q . .
. . Q .
. . . Q
Q . . .

Heuristic (Conflicts): 4
Evaluating Neighbor:
. Q . .
Q . . .
. . Q .
. . . Q

Heuristic (Conflicts): 2
Evaluating Neighbor:
. Q . .
. . . Q
Q . . .
. . Q .

Heuristic (Conflicts): 0
Moving to Next State:
. Q . .
. . . Q
Q . . .
. . Q .

Heuristic (Conflicts): 0

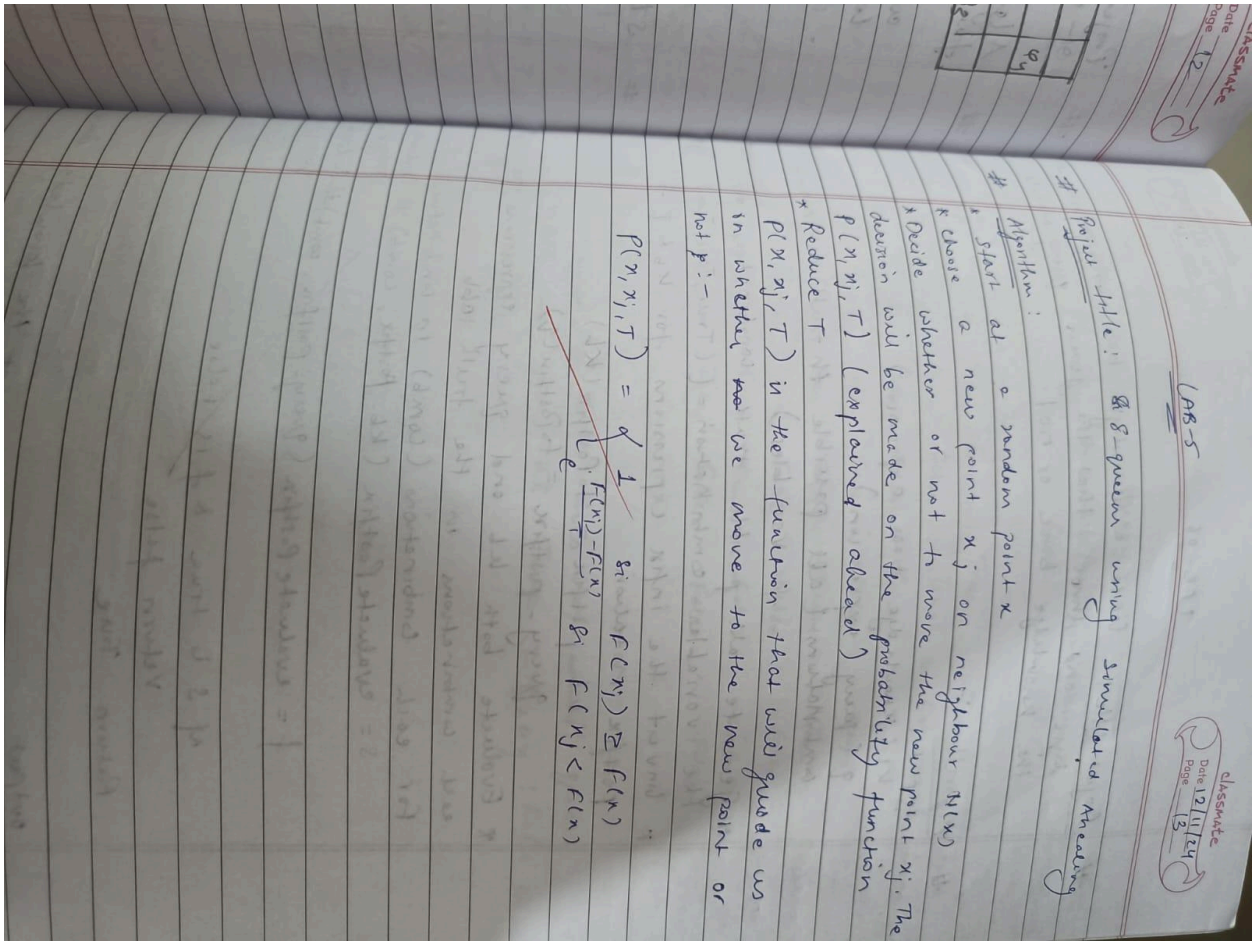
Best Solution Found:
. Q . .
. . . Q
Q . . .
. . Q .

Final Heuristic (Conflicts): 0

LAB-05

Implementing simulated annealing to solve N-queens problem

ALGORITHM:



CODE:

```
import random
import math

def calculate_cost(state):
    attacking_pairs = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) ==
abs(i - j):
                attacking_pairs += 1
    return attacking_pairs

def generate_neighbor(state):
    neighbor = state[:]
    column = random.randint(0, len(state) - 1)
    new_row = random.randint(0, len(state) - 1)
    while new_row == state[column]:
        new_row = random.randint(0, len(state) - 1)
    neighbor[column] = new_row
    return neighbor

def simulated_annealing(n, initial_temperature, cooling_factor,
max_iterations):
    current_state = [random.randint(0, n-1) for _ in range(n)]
    current_cost = calculate_cost(current_state)
    best_state = current_state
    best_cost = current_cost
    temperature = initial_temperature

    print(f"Initial state: {current_state} with cost:
{current_cost}")

    iteration = 0
    while iteration < max_iterations and temperature > 0:
        neighbor_state = generate_neighbor(current_state)
        neighbor_cost = calculate_cost(neighbor_state)

        delta_cost = neighbor_cost - current_cost
```

```
        if delta_cost < 0:
            current_state = neighbor_state
            current_cost = neighbor_cost
        else:
            acceptance_probability = math.exp(-delta_cost /
temperature)
            if random.random() < acceptance_probability:
                current_state = neighbor_state
                current_cost = neighbor_cost

        if current_cost < best_cost:
            best_state = current_state
            best_cost = current_cost

        print(f"Iteration {iteration+1}: {current_state} with cost:
{current_cost}")

        temperature *= cooling_factor
        iteration += 1

    return best_state, best_cost

if __name__ == "__main__":
    N = 4
    initial_temperature = 100
    cooling_factor = 0.95
    max_iterations = 1000

    best_state, best_cost = simulated_annealing(N,
initial_temperature, cooling_factor, max_iterations)

    print(f"\nFinal solution: {best_state} with cost: {best_cost}")
```

OUTPUT:

Initial state: [3, 3, 1, 0] with cost: 4
Iteration 1: [3, 0, 1, 0] with cost: 5
Iteration 2: [3, 0, 1, 1] with cost: 3
Iteration 3: [3, 3, 1, 1] with cost: 4
Iteration 4: [3, 3, 1, 2] with cost: 3
Iteration 5: [3, 0, 1, 2] with cost: 4
Iteration 6: [3, 0, 1, 3] with cost: 3
Iteration 7: [3, 0, 3, 3] with cost: 3
Iteration 8: [2, 0, 3, 3] with cost: 1

Iteration 9: [2, 0, 2, 3] with cost: 2
Iteration 10: [3, 0, 2, 3] with cost: 2
Iteration 11: [1, 0, 2, 3] with cost: 2
Iteration 12: [1, 0, 2, 1] with cost: 3
Iteration 13: [1, 0, 2, 2] with cost: 3
Iteration 14: [1, 0, 3, 2] with cost: 4
Iteration 15: [1, 2, 3, 2] with cost: 5
Iteration 16: [3, 2, 3, 2] with cost: 5
Iteration 17: [3, 2, 3, 1] with cost: 3
Iteration 18: [3, 2, 3, 1] with cost: 3
Iteration 19: [3, 2, 3, 3] with cost: 5
Iteration 20: [3, 2, 3, 1] with cost: 3
Iteration 21: [3, 2, 3, 0] with cost: 5
Iteration 22: [3, 2, 1, 0] with cost: 6
Iteration 23: [3, 3, 1, 0] with cost: 4
Iteration 24: [3, 0, 1, 0] with cost: 5
Iteration 25: [3, 0, 1, 2] with cost: 4
Iteration 26: [1, 0, 1, 2] with cost: 5
Iteration 27: [1, 3, 1, 2] with cost: 2
Iteration 28: [0, 3, 1, 2] with cost: 1
Iteration 29: [0, 3, 1, 0] with cost: 2
Iteration 30: [0, 3, 3, 0] with cost: 2
Iteration 31: [0, 2, 3, 0] with cost: 3
Iteration 32: [0, 2, 1, 0] with cost: 4
Iteration 33: [0, 3, 1, 0] with cost: 2
Iteration 34: [0, 1, 1, 0] with cost: 4
Iteration 35: [0, 3, 1, 0] with cost: 2
Iteration 36: [0, 3, 3, 0] with cost: 2
Iteration 37: [0, 3, 3, 3] with cost: 4
Iteration 38: [0, 3, 2, 3] with cost: 5
Iteration 39: [0, 0, 2, 3] with cost: 4
Iteration 40: [3, 0, 2, 3] with cost: 2
Iteration 41: [3, 0, 0, 3] with cost: 2
Iteration 42: [3, 0, 0, 1] with cost: 2
Iteration 43: [3, 1, 0, 1] with cost: 3
Iteration 44: [3, 0, 0, 1] with cost: 2
Iteration 45: [3, 1, 0, 1] with cost: 3
Iteration 46: [3, 1, 0, 3] with cost: 3
Iteration 47: [3, 1, 2, 3] with cost: 4
Iteration 48: [3, 0, 2, 3] with cost: 2
Iteration 49: [3, 0, 2, 1] with cost: 1
Iteration 50: [3, 0, 3, 1] with cost: 1
Iteration 51: [3, 1, 3, 1] with cost: 2
Iteration 52: [3, 3, 3, 1] with cost: 4

Iteration 53: [3, 0, 3, 1] with cost: 1
Iteration 54: [0, 0, 3, 1] with cost: 1
Iteration 55: [0, 0, 3, 3] with cost: 3
Iteration 56: [3, 0, 3, 3] with cost: 3
Iteration 57: [3, 0, 3, 1] with cost: 1
Iteration 58: [2, 0, 3, 1] with cost: 0
Iteration 59: [2, 0, 3, 1] with cost: 0
Iteration 60: [2, 0, 3, 1] with cost: 0
Iteration 61: [2, 0, 0, 1] with cost: 3
Iteration 62: [1, 0, 0, 1] with cost: 4
Iteration 63: [1, 0, 3, 1] with cost: 3
Iteration 64: [1, 0, 3, 0] with cost: 3
Iteration 65: [1, 0, 3, 1] with cost: 3
Iteration 66: [1, 1, 3, 1] with cost: 4
Iteration 67: [1, 3, 3, 1] with cost: 4
Iteration 68: [1, 3, 0, 1] with cost: 3
Iteration 69: [0, 3, 0, 1] with cost: 3
Iteration 70: [0, 3, 2, 1] with cost: 4
Iteration 71: [0, 0, 2, 1] with cost: 3
Iteration 72: [0, 0, 2, 2] with cost: 4
Iteration 73: [0, 0, 0, 2] with cost: 4
Iteration 74: [0, 0, 0, 0] with cost: 6
Iteration 75: [0, 3, 0, 0] with cost: 3
Iteration 76: [0, 3, 0, 0] with cost: 3
Iteration 77: [0, 3, 2, 0] with cost: 3
Iteration 78: [0, 3, 2, 0] with cost: 3
Iteration 79: [3, 3, 2, 0] with cost: 3
Iteration 80: [3, 3, 2, 1] with cost: 4
Iteration 81: [3, 3, 3, 1] with cost: 4
Iteration 82: [3, 1, 3, 1] with cost: 2
Iteration 83: [3, 1, 3, 1] with cost: 2
Iteration 84: [3, 1, 3, 2] with cost: 2
Iteration 85: [0, 1, 3, 2] with cost: 2
Iteration 86: [1, 1, 3, 2] with cost: 3
Iteration 87: [3, 1, 3, 2] with cost: 2
Iteration 88: [3, 1, 0, 2] with cost: 1
Iteration 89: [3, 1, 0, 2] with cost: 1
Iteration 90: [3, 1, 0, 2] with cost: 1
Iteration 91: [3, 1, 0, 2] with cost: 1
Iteration 92: [1, 1, 0, 2] with cost: 2
Iteration 93: [1, 1, 3, 2] with cost: 3
Iteration 94: [1, 1, 3, 0] with cost: 2
Iteration 95: [1, 1, 3, 0] with cost: 2
Iteration 96: [1, 1, 3, 0] with cost: 2

Iteration 97: [2, 1, 3, 0] with cost: 1
Iteration 98: [2, 1, 3, 0] with cost: 1
Iteration 99: [2, 1, 3, 0] with cost: 1
Iteration 100: [2, 0, 3, 0] with cost: 1
Iteration 101: [1, 0, 3, 0] with cost: 3
Iteration 102: [1, 2, 3, 0] with cost: 4
Iteration 103: [1, 2, 3, 0] with cost: 4
Iteration 104: [1, 2, 3, 3] with cost: 4
Iteration 105: [1, 2, 2, 3] with cost: 3
Iteration 106: [1, 2, 2, 3] with cost: 3
Iteration 107: [1, 2, 2, 0] with cost: 3
Iteration 108: [1, 2, 2, 3] with cost: 3
Iteration 109: [1, 2, 3, 3] with cost: 4
Iteration 110: [1, 0, 3, 3] with cost: 3
Iteration 111: [1, 0, 0, 3] with cost: 2
Iteration 112: [1, 0, 0, 3] with cost: 2
Iteration 113: [1, 0, 0, 3] with cost: 2
Iteration 114: [1, 0, 0, 3] with cost: 2
Iteration 115: [3, 0, 0, 3] with cost: 2
Iteration 116: [3, 0, 2, 3] with cost: 2
Iteration 117: [3, 0, 2, 3] with cost: 2
Iteration 118: [3, 0, 2, 3] with cost: 2
Iteration 119: [3, 0, 2, 0] with cost: 2
Iteration 120: [3, 0, 2, 0] with cost: 2
Iteration 121: [3, 0, 2, 0] with cost: 2
Iteration 122: [3, 1, 2, 0] with cost: 2
Iteration 123: [3, 1, 2, 0] with cost: 2
Iteration 124: [3, 1, 2, 0] with cost: 2
Iteration 125: [3, 1, 2, 0] with cost: 2
Iteration 126: [3, 1, 2, 0] with cost: 2
Iteration 127: [3, 0, 2, 0] with cost: 2
Iteration 128: [3, 1, 2, 0] with cost: 2
Iteration 129: [3, 1, 2, 0] with cost: 2
Iteration 130: [3, 1, 2, 0] with cost: 2
Iteration 131: [3, 1, 2, 0] with cost: 2
Iteration 132: [3, 1, 3, 0] with cost: 2
Iteration 133: [2, 1, 3, 0] with cost: 1
Iteration 134: [2, 1, 3, 0] with cost: 1
Iteration 135: [2, 1, 3, 0] with cost: 1
Iteration 136: [2, 1, 3, 0] with cost: 1
Iteration 137: [2, 0, 3, 0] with cost: 1
Iteration 138: [2, 0, 3, 0] with cost: 1
Iteration 139: [2, 0, 3, 0] with cost: 1
Iteration 140: [2, 1, 3, 0] with cost: 1

Iteration 141: [2, 1, 3, 0] with cost: 1
Iteration 142: [2, 1, 3, 0] with cost: 1
Iteration 143: [2, 1, 3, 0] with cost: 1
Iteration 144: [2, 1, 3, 0] with cost: 1
Iteration 145: [2, 1, 3, 0] with cost: 1
Iteration 146: [2, 1, 3, 0] with cost: 1
Iteration 147: [2, 1, 3, 0] with cost: 1
Iteration 148: [2, 1, 3, 0] with cost: 1
Iteration 149: [2, 1, 3, 0] with cost: 1
Iteration 150: [2, 1, 3, 0] with cost: 1
Iteration 151: [2, 1, 3, 0] with cost: 1
Iteration 152: [2, 1, 3, 0] with cost: 1
Iteration 153: [2, 1, 3, 0] with cost: 1
Iteration 154: [2, 1, 3, 0] with cost: 1
Iteration 155: [2, 1, 3, 0] with cost: 1
Iteration 156: [2, 1, 3, 0] with cost: 1
Iteration 157: [2, 1, 3, 0] with cost: 1
Iteration 158: [2, 1, 3, 0] with cost: 1
Iteration 159: [2, 1, 3, 0] with cost: 1
Iteration 160: [2, 1, 3, 0] with cost: 1
Iteration 161: [2, 1, 3, 0] with cost: 1
Iteration 162: [2, 1, 3, 0] with cost: 1
Iteration 163: [2, 1, 3, 0] with cost: 1
Iteration 164: [2, 1, 3, 0] with cost: 1
Iteration 165: [2, 1, 3, 0] with cost: 1
Iteration 166: [2, 1, 3, 0] with cost: 1
Iteration 167: [2, 1, 3, 0] with cost: 1
Iteration 168: [2, 1, 3, 0] with cost: 1
Iteration 169: [2, 1, 3, 0] with cost: 1
Iteration 170: [2, 1, 3, 0] with cost: 1
Iteration 171: [2, 1, 3, 0] with cost: 1
Iteration 172: [2, 1, 3, 0] with cost: 1
Iteration 173: [2, 1, 3, 0] with cost: 1
Iteration 174: [2, 1, 3, 0] with cost: 1
Iteration 175: [2, 1, 3, 0] with cost: 1
Iteration 176: [2, 1, 3, 0] with cost: 1
Iteration 177: [2, 1, 3, 0] with cost: 1
Iteration 178: [2, 0, 3, 0] with cost: 1
Iteration 179: [2, 0, 3, 0] with cost: 1
Iteration 180: [2, 0, 3, 0] with cost: 1
Iteration 181: [2, 0, 3, 0] with cost: 1
Iteration 182: [2, 0, 3, 0] with cost: 1
Iteration 183: [2, 0, 3, 0] with cost: 1
Iteration 184: [2, 0, 3, 0] with cost: 1

Iteration 185: [2, 0, 3, 0] with cost: 1
Iteration 186: [2, 0, 3, 0] with cost: 1
Iteration 187: [2, 0, 3, 1] with cost: 0
Iteration 188: [2, 0, 3, 1] with cost: 0
Iteration 189: [2, 0, 3, 1] with cost: 0
Iteration 190: [2, 0, 3, 1] with cost: 0
Iteration 191: [2, 0, 3, 1] with cost: 0
Iteration 192: [2, 0, 3, 1] with cost: 0
Iteration 193: [2, 0, 3, 1] with cost: 0
Iteration 194: [2, 0, 3, 1] with cost: 0
Iteration 195: [2, 0, 3, 1] with cost: 0
Iteration 196: [2, 0, 3, 1] with cost: 0
Iteration 197: [2, 0, 3, 1] with cost: 0
Iteration 198: [2, 0, 3, 1] with cost: 0
Iteration 199: [2, 0, 3, 1] with cost: 0
Iteration 200: [2, 0, 3, 1] with cost: 0
Iteration 201: [2, 0, 3, 1] with cost: 0
Iteration 202: [2, 0, 3, 1] with cost: 0
Iteration 203: [2, 0, 3, 1] with cost: 0
Iteration 204: [2, 0, 3, 1] with cost: 0
Iteration 205: [2, 0, 3, 1] with cost: 0
Iteration 206: [2, 0, 3, 1] with cost: 0
Iteration 207: [2, 0, 3, 1] with cost: 0
Iteration 208: [2, 0, 3, 1] with cost: 0
Iteration 209: [2, 0, 3, 1] with cost: 0
Iteration 210: [2, 0, 3, 1] with cost: 0
Iteration 211: [2, 0, 3, 1] with cost: 0
Iteration 212: [2, 0, 3, 1] with cost: 0
Iteration 213: [2, 0, 3, 1] with cost: 0
Iteration 214: [2, 0, 3, 1] with cost: 0
Iteration 215: [2, 0, 3, 1] with cost: 0
Iteration 216: [2, 0, 3, 1] with cost: 0
Iteration 217: [2, 0, 3, 1] with cost: 0
Iteration 218: [2, 0, 3, 1] with cost: 0
Iteration 219: [2, 0, 3, 1] with cost: 0
Iteration 220: [2, 0, 3, 1] with cost: 0
Iteration 221: [2, 0, 3, 1] with cost: 0
Iteration 222: [2, 0, 3, 1] with cost: 0
Iteration 223: [2, 0, 3, 1] with cost: 0
Iteration 224: [2, 0, 3, 1] with cost: 0
Iteration 225: [2, 0, 3, 1] with cost: 0
Iteration 226: [2, 0, 3, 1] with cost: 0
Iteration 227: [2, 0, 3, 1] with cost: 0
Iteration 228: [2, 0, 3, 1] with cost: 0

Iteration 229: [2, 0, 3, 1] with cost: 0
Iteration 230: [2, 0, 3, 1] with cost: 0
Iteration 231: [2, 0, 3, 1] with cost: 0
Iteration 232: [2, 0, 3, 1] with cost: 0
Iteration 233: [2, 0, 3, 1] with cost: 0
Iteration 234: [2, 0, 3, 1] with cost: 0
Iteration 235: [2, 0, 3, 1] with cost: 0
Iteration 236: [2, 0, 3, 1] with cost: 0
Iteration 237: [2, 0, 3, 1] with cost: 0
Iteration 238: [2, 0, 3, 1] with cost: 0
Iteration 239: [2, 0, 3, 1] with cost: 0
Iteration 240: [2, 0, 3, 1] with cost: 0
Iteration 241: [2, 0, 3, 1] with cost: 0
Iteration 242: [2, 0, 3, 1] with cost: 0
Iteration 243: [2, 0, 3, 1] with cost: 0
Iteration 244: [2, 0, 3, 1] with cost: 0
Iteration 245: [2, 0, 3, 1] with cost: 0
Iteration 246: [2, 0, 3, 1] with cost: 0
Iteration 247: [2, 0, 3, 1] with cost: 0
Iteration 248: [2, 0, 3, 1] with cost: 0
Iteration 249: [2, 0, 3, 1] with cost: 0
Iteration 250: [2, 0, 3, 1] with cost: 0
Iteration 251: [2, 0, 3, 1] with cost: 0
Iteration 252: [2, 0, 3, 1] with cost: 0
Iteration 253: [2, 0, 3, 1] with cost: 0
Iteration 254: [2, 0, 3, 1] with cost: 0
Iteration 255: [2, 0, 3, 1] with cost: 0
Iteration 256: [2, 0, 3, 1] with cost: 0
Iteration 257: [2, 0, 3, 1] with cost: 0
Iteration 258: [2, 0, 3, 1] with cost: 0
Iteration 259: [2, 0, 3, 1] with cost: 0
Iteration 260: [2, 0, 3, 1] with cost: 0
Iteration 261: [2, 0, 3, 1] with cost: 0
Iteration 262: [2, 0, 3, 1] with cost: 0
Iteration 263: [2, 0, 3, 1] with cost: 0
Iteration 264: [2, 0, 3, 1] with cost: 0
Iteration 265: [2, 0, 3, 1] with cost: 0
Iteration 266: [2, 0, 3, 1] with cost: 0
Iteration 267: [2, 0, 3, 1] with cost: 0
Iteration 268: [2, 0, 3, 1] with cost: 0
Iteration 269: [2, 0, 3, 1] with cost: 0
Iteration 270: [2, 0, 3, 1] with cost: 0
Iteration 271: [2, 0, 3, 1] with cost: 0
Iteration 272: [2, 0, 3, 1] with cost: 0

Iteration 273: [2, 0, 3, 1] with cost: 0
Iteration 274: [2, 0, 3, 1] with cost: 0
Iteration 275: [2, 0, 3, 1] with cost: 0
Iteration 276: [2, 0, 3, 1] with cost: 0
Iteration 277: [2, 0, 3, 1] with cost: 0
Iteration 278: [2, 0, 3, 1] with cost: 0
Iteration 279: [2, 0, 3, 1] with cost: 0
Iteration 280: [2, 0, 3, 1] with cost: 0
Iteration 281: [2, 0, 3, 1] with cost: 0
Iteration 282: [2, 0, 3, 1] with cost: 0
Iteration 283: [2, 0, 3, 1] with cost: 0
Iteration 284: [2, 0, 3, 1] with cost: 0
Iteration 285: [2, 0, 3, 1] with cost: 0
Iteration 286: [2, 0, 3, 1] with cost: 0
Iteration 287: [2, 0, 3, 1] with cost: 0
Iteration 288: [2, 0, 3, 1] with cost: 0
Iteration 289: [2, 0, 3, 1] with cost: 0
Iteration 290: [2, 0, 3, 1] with cost: 0
Iteration 291: [2, 0, 3, 1] with cost: 0
Iteration 292: [2, 0, 3, 1] with cost: 0
Iteration 293: [2, 0, 3, 1] with cost: 0
Iteration 294: [2, 0, 3, 1] with cost: 0
Iteration 295: [2, 0, 3, 1] with cost: 0
Iteration 296: [2, 0, 3, 1] with cost: 0
Iteration 297: [2, 0, 3, 1] with cost: 0
Iteration 298: [2, 0, 3, 1] with cost: 0
Iteration 299: [2, 0, 3, 1] with cost: 0
Iteration 300: [2, 0, 3, 1] with cost: 0
Iteration 301: [2, 0, 3, 1] with cost: 0
Iteration 302: [2, 0, 3, 1] with cost: 0
Iteration 303: [2, 0, 3, 1] with cost: 0
Iteration 304: [2, 0, 3, 1] with cost: 0
Iteration 305: [2, 0, 3, 1] with cost: 0
Iteration 306: [2, 0, 3, 1] with cost: 0
Iteration 307: [2, 0, 3, 1] with cost: 0
Iteration 308: [2, 0, 3, 1] with cost: 0
Iteration 309: [2, 0, 3, 1] with cost: 0
Iteration 310: [2, 0, 3, 1] with cost: 0
Iteration 311: [2, 0, 3, 1] with cost: 0
Iteration 312: [2, 0, 3, 1] with cost: 0
Iteration 313: [2, 0, 3, 1] with cost: 0
Iteration 314: [2, 0, 3, 1] with cost: 0
Iteration 315: [2, 0, 3, 1] with cost: 0
Iteration 316: [2, 0, 3, 1] with cost: 0

Iteration 317: [2, 0, 3, 1] with cost: 0
Iteration 318: [2, 0, 3, 1] with cost: 0
Iteration 319: [2, 0, 3, 1] with cost: 0
Iteration 320: [2, 0, 3, 1] with cost: 0
Iteration 321: [2, 0, 3, 1] with cost: 0
Iteration 322: [2, 0, 3, 1] with cost: 0
Iteration 323: [2, 0, 3, 1] with cost: 0
Iteration 324: [2, 0, 3, 1] with cost: 0
Iteration 325: [2, 0, 3, 1] with cost: 0
Iteration 326: [2, 0, 3, 1] with cost: 0
Iteration 327: [2, 0, 3, 1] with cost: 0
Iteration 328: [2, 0, 3, 1] with cost: 0
Iteration 329: [2, 0, 3, 1] with cost: 0
Iteration 330: [2, 0, 3, 1] with cost: 0
Iteration 331: [2, 0, 3, 1] with cost: 0
Iteration 332: [2, 0, 3, 1] with cost: 0
Iteration 333: [2, 0, 3, 1] with cost: 0
Iteration 334: [2, 0, 3, 1] with cost: 0
Iteration 335: [2, 0, 3, 1] with cost: 0
Iteration 336: [2, 0, 3, 1] with cost: 0
Iteration 337: [2, 0, 3, 1] with cost: 0
Iteration 338: [2, 0, 3, 1] with cost: 0
Iteration 339: [2, 0, 3, 1] with cost: 0
Iteration 340: [2, 0, 3, 1] with cost: 0
Iteration 341: [2, 0, 3, 1] with cost: 0
Iteration 342: [2, 0, 3, 1] with cost: 0
Iteration 343: [2, 0, 3, 1] with cost: 0
Iteration 344: [2, 0, 3, 1] with cost: 0
Iteration 345: [2, 0, 3, 1] with cost: 0
Iteration 346: [2, 0, 3, 1] with cost: 0
Iteration 347: [2, 0, 3, 1] with cost: 0
Iteration 348: [2, 0, 3, 1] with cost: 0
Iteration 349: [2, 0, 3, 1] with cost: 0
Iteration 350: [2, 0, 3, 1] with cost: 0
Iteration 351: [2, 0, 3, 1] with cost: 0
Iteration 352: [2, 0, 3, 1] with cost: 0
Iteration 353: [2, 0, 3, 1] with cost: 0
Iteration 354: [2, 0, 3, 1] with cost: 0
Iteration 355: [2, 0, 3, 1] with cost: 0
Iteration 356: [2, 0, 3, 1] with cost: 0
Iteration 357: [2, 0, 3, 1] with cost: 0
Iteration 358: [2, 0, 3, 1] with cost: 0
Iteration 359: [2, 0, 3, 1] with cost: 0
Iteration 360: [2, 0, 3, 1] with cost: 0

Iteration 361: [2, 0, 3, 1] with cost: 0
Iteration 362: [2, 0, 3, 1] with cost: 0
Iteration 363: [2, 0, 3, 1] with cost: 0
Iteration 364: [2, 0, 3, 1] with cost: 0
Iteration 365: [2, 0, 3, 1] with cost: 0
Iteration 366: [2, 0, 3, 1] with cost: 0
Iteration 367: [2, 0, 3, 1] with cost: 0
Iteration 368: [2, 0, 3, 1] with cost: 0
Iteration 369: [2, 0, 3, 1] with cost: 0
Iteration 370: [2, 0, 3, 1] with cost: 0
Iteration 371: [2, 0, 3, 1] with cost: 0
Iteration 372: [2, 0, 3, 1] with cost: 0
Iteration 373: [2, 0, 3, 1] with cost: 0
Iteration 374: [2, 0, 3, 1] with cost: 0
Iteration 375: [2, 0, 3, 1] with cost: 0
Iteration 376: [2, 0, 3, 1] with cost: 0
Iteration 377: [2, 0, 3, 1] with cost: 0
Iteration 378: [2, 0, 3, 1] with cost: 0
Iteration 379: [2, 0, 3, 1] with cost: 0
Iteration 380: [2, 0, 3, 1] with cost: 0
Iteration 381: [2, 0, 3, 1] with cost: 0
Iteration 382: [2, 0, 3, 1] with cost: 0
Iteration 383: [2, 0, 3, 1] with cost: 0
Iteration 384: [2, 0, 3, 1] with cost: 0
Iteration 385: [2, 0, 3, 1] with cost: 0
Iteration 386: [2, 0, 3, 1] with cost: 0
Iteration 387: [2, 0, 3, 1] with cost: 0
Iteration 388: [2, 0, 3, 1] with cost: 0
Iteration 389: [2, 0, 3, 1] with cost: 0
Iteration 390: [2, 0, 3, 1] with cost: 0
Iteration 391: [2, 0, 3, 1] with cost: 0
Iteration 392: [2, 0, 3, 1] with cost: 0
Iteration 393: [2, 0, 3, 1] with cost: 0
Iteration 394: [2, 0, 3, 1] with cost: 0
Iteration 395: [2, 0, 3, 1] with cost: 0
Iteration 396: [2, 0, 3, 1] with cost: 0
Iteration 397: [2, 0, 3, 1] with cost: 0
Iteration 398: [2, 0, 3, 1] with cost: 0
Iteration 399: [2, 0, 3, 1] with cost: 0
Iteration 400: [2, 0, 3, 1] with cost: 0
Iteration 401: [2, 0, 3, 1] with cost: 0
Iteration 402: [2, 0, 3, 1] with cost: 0
Iteration 403: [2, 0, 3, 1] with cost: 0
Iteration 404: [2, 0, 3, 1] with cost: 0

Iteration 405: [2, 0, 3, 1] with cost: 0
Iteration 406: [2, 0, 3, 1] with cost: 0
Iteration 407: [2, 0, 3, 1] with cost: 0
Iteration 408: [2, 0, 3, 1] with cost: 0
Iteration 409: [2, 0, 3, 1] with cost: 0
Iteration 410: [2, 0, 3, 1] with cost: 0
Iteration 411: [2, 0, 3, 1] with cost: 0
Iteration 412: [2, 0, 3, 1] with cost: 0
Iteration 413: [2, 0, 3, 1] with cost: 0
Iteration 414: [2, 0, 3, 1] with cost: 0
Iteration 415: [2, 0, 3, 1] with cost: 0
Iteration 416: [2, 0, 3, 1] with cost: 0
Iteration 417: [2, 0, 3, 1] with cost: 0
Iteration 418: [2, 0, 3, 1] with cost: 0
Iteration 419: [2, 0, 3, 1] with cost: 0
Iteration 420: [2, 0, 3, 1] with cost: 0
Iteration 421: [2, 0, 3, 1] with cost: 0
Iteration 422: [2, 0, 3, 1] with cost: 0
Iteration 423: [2, 0, 3, 1] with cost: 0
Iteration 424: [2, 0, 3, 1] with cost: 0
Iteration 425: [2, 0, 3, 1] with cost: 0
Iteration 426: [2, 0, 3, 1] with cost: 0
Iteration 427: [2, 0, 3, 1] with cost: 0
Iteration 428: [2, 0, 3, 1] with cost: 0
Iteration 429: [2, 0, 3, 1] with cost: 0
Iteration 430: [2, 0, 3, 1] with cost: 0
Iteration 431: [2, 0, 3, 1] with cost: 0
Iteration 432: [2, 0, 3, 1] with cost: 0
Iteration 433: [2, 0, 3, 1] with cost: 0
Iteration 434: [2, 0, 3, 1] with cost: 0
Iteration 435: [2, 0, 3, 1] with cost: 0
Iteration 436: [2, 0, 3, 1] with cost: 0
Iteration 437: [2, 0, 3, 1] with cost: 0
Iteration 438: [2, 0, 3, 1] with cost: 0
Iteration 439: [2, 0, 3, 1] with cost: 0
Iteration 440: [2, 0, 3, 1] with cost: 0
Iteration 441: [2, 0, 3, 1] with cost: 0
Iteration 442: [2, 0, 3, 1] with cost: 0
Iteration 443: [2, 0, 3, 1] with cost: 0
Iteration 444: [2, 0, 3, 1] with cost: 0
Iteration 445: [2, 0, 3, 1] with cost: 0
Iteration 446: [2, 0, 3, 1] with cost: 0
Iteration 447: [2, 0, 3, 1] with cost: 0
Iteration 448: [2, 0, 3, 1] with cost: 0

Iteration 449: [2, 0, 3, 1] with cost: 0
Iteration 450: [2, 0, 3, 1] with cost: 0
Iteration 451: [2, 0, 3, 1] with cost: 0
Iteration 452: [2, 0, 3, 1] with cost: 0
Iteration 453: [2, 0, 3, 1] with cost: 0
Iteration 454: [2, 0, 3, 1] with cost: 0
Iteration 455: [2, 0, 3, 1] with cost: 0
Iteration 456: [2, 0, 3, 1] with cost: 0
Iteration 457: [2, 0, 3, 1] with cost: 0
Iteration 458: [2, 0, 3, 1] with cost: 0
Iteration 459: [2, 0, 3, 1] with cost: 0
Iteration 460: [2, 0, 3, 1] with cost: 0
Iteration 461: [2, 0, 3, 1] with cost: 0
Iteration 462: [2, 0, 3, 1] with cost: 0
Iteration 463: [2, 0, 3, 1] with cost: 0
Iteration 464: [2, 0, 3, 1] with cost: 0
Iteration 465: [2, 0, 3, 1] with cost: 0
Iteration 466: [2, 0, 3, 1] with cost: 0
Iteration 467: [2, 0, 3, 1] with cost: 0
Iteration 468: [2, 0, 3, 1] with cost: 0
Iteration 469: [2, 0, 3, 1] with cost: 0
Iteration 470: [2, 0, 3, 1] with cost: 0
Iteration 471: [2, 0, 3, 1] with cost: 0
Iteration 472: [2, 0, 3, 1] with cost: 0
Iteration 473: [2, 0, 3, 1] with cost: 0
Iteration 474: [2, 0, 3, 1] with cost: 0
Iteration 475: [2, 0, 3, 1] with cost: 0
Iteration 476: [2, 0, 3, 1] with cost: 0
Iteration 477: [2, 0, 3, 1] with cost: 0
Iteration 478: [2, 0, 3, 1] with cost: 0
Iteration 479: [2, 0, 3, 1] with cost: 0
Iteration 480: [2, 0, 3, 1] with cost: 0
Iteration 481: [2, 0, 3, 1] with cost: 0
Iteration 482: [2, 0, 3, 1] with cost: 0
Iteration 483: [2, 0, 3, 1] with cost: 0
Iteration 484: [2, 0, 3, 1] with cost: 0
Iteration 485: [2, 0, 3, 1] with cost: 0
Iteration 486: [2, 0, 3, 1] with cost: 0
Iteration 487: [2, 0, 3, 1] with cost: 0
Iteration 488: [2, 0, 3, 1] with cost: 0
Iteration 489: [2, 0, 3, 1] with cost: 0
Iteration 490: [2, 0, 3, 1] with cost: 0
Iteration 491: [2, 0, 3, 1] with cost: 0
Iteration 492: [2, 0, 3, 1] with cost: 0

Iteration 493: [2, 0, 3, 1] with cost: 0
Iteration 494: [2, 0, 3, 1] with cost: 0
Iteration 495: [2, 0, 3, 1] with cost: 0
Iteration 496: [2, 0, 3, 1] with cost: 0
Iteration 497: [2, 0, 3, 1] with cost: 0
Iteration 498: [2, 0, 3, 1] with cost: 0
Iteration 499: [2, 0, 3, 1] with cost: 0
Iteration 500: [2, 0, 3, 1] with cost: 0
Iteration 501: [2, 0, 3, 1] with cost: 0
Iteration 502: [2, 0, 3, 1] with cost: 0
Iteration 503: [2, 0, 3, 1] with cost: 0
Iteration 504: [2, 0, 3, 1] with cost: 0
Iteration 505: [2, 0, 3, 1] with cost: 0
Iteration 506: [2, 0, 3, 1] with cost: 0
Iteration 507: [2, 0, 3, 1] with cost: 0
Iteration 508: [2, 0, 3, 1] with cost: 0
Iteration 509: [2, 0, 3, 1] with cost: 0
Iteration 510: [2, 0, 3, 1] with cost: 0
Iteration 511: [2, 0, 3, 1] with cost: 0
Iteration 512: [2, 0, 3, 1] with cost: 0
Iteration 513: [2, 0, 3, 1] with cost: 0
Iteration 514: [2, 0, 3, 1] with cost: 0
Iteration 515: [2, 0, 3, 1] with cost: 0
Iteration 516: [2, 0, 3, 1] with cost: 0
Iteration 517: [2, 0, 3, 1] with cost: 0
Iteration 518: [2, 0, 3, 1] with cost: 0
Iteration 519: [2, 0, 3, 1] with cost: 0
Iteration 520: [2, 0, 3, 1] with cost: 0
Iteration 521: [2, 0, 3, 1] with cost: 0
Iteration 522: [2, 0, 3, 1] with cost: 0
Iteration 523: [2, 0, 3, 1] with cost: 0
Iteration 524: [2, 0, 3, 1] with cost: 0
Iteration 525: [2, 0, 3, 1] with cost: 0
Iteration 526: [2, 0, 3, 1] with cost: 0
Iteration 527: [2, 0, 3, 1] with cost: 0
Iteration 528: [2, 0, 3, 1] with cost: 0
Iteration 529: [2, 0, 3, 1] with cost: 0
Iteration 530: [2, 0, 3, 1] with cost: 0
Iteration 531: [2, 0, 3, 1] with cost: 0
Iteration 532: [2, 0, 3, 1] with cost: 0
Iteration 533: [2, 0, 3, 1] with cost: 0
Iteration 534: [2, 0, 3, 1] with cost: 0
Iteration 535: [2, 0, 3, 1] with cost: 0
Iteration 536: [2, 0, 3, 1] with cost: 0

Iteration 537: [2, 0, 3, 1] with cost: 0
Iteration 538: [2, 0, 3, 1] with cost: 0
Iteration 539: [2, 0, 3, 1] with cost: 0
Iteration 540: [2, 0, 3, 1] with cost: 0
Iteration 541: [2, 0, 3, 1] with cost: 0
Iteration 542: [2, 0, 3, 1] with cost: 0
Iteration 543: [2, 0, 3, 1] with cost: 0
Iteration 544: [2, 0, 3, 1] with cost: 0
Iteration 545: [2, 0, 3, 1] with cost: 0
Iteration 546: [2, 0, 3, 1] with cost: 0
Iteration 547: [2, 0, 3, 1] with cost: 0
Iteration 548: [2, 0, 3, 1] with cost: 0
Iteration 549: [2, 0, 3, 1] with cost: 0
Iteration 550: [2, 0, 3, 1] with cost: 0
Iteration 551: [2, 0, 3, 1] with cost: 0
Iteration 552: [2, 0, 3, 1] with cost: 0
Iteration 553: [2, 0, 3, 1] with cost: 0
Iteration 554: [2, 0, 3, 1] with cost: 0
Iteration 555: [2, 0, 3, 1] with cost: 0
Iteration 556: [2, 0, 3, 1] with cost: 0
Iteration 557: [2, 0, 3, 1] with cost: 0
Iteration 558: [2, 0, 3, 1] with cost: 0
Iteration 559: [2, 0, 3, 1] with cost: 0
Iteration 560: [2, 0, 3, 1] with cost: 0
Iteration 561: [2, 0, 3, 1] with cost: 0
Iteration 562: [2, 0, 3, 1] with cost: 0
Iteration 563: [2, 0, 3, 1] with cost: 0
Iteration 564: [2, 0, 3, 1] with cost: 0
Iteration 565: [2, 0, 3, 1] with cost: 0
Iteration 566: [2, 0, 3, 1] with cost: 0
Iteration 567: [2, 0, 3, 1] with cost: 0
Iteration 568: [2, 0, 3, 1] with cost: 0
Iteration 569: [2, 0, 3, 1] with cost: 0
Iteration 570: [2, 0, 3, 1] with cost: 0
Iteration 571: [2, 0, 3, 1] with cost: 0
Iteration 572: [2, 0, 3, 1] with cost: 0
Iteration 573: [2, 0, 3, 1] with cost: 0
Iteration 574: [2, 0, 3, 1] with cost: 0
Iteration 575: [2, 0, 3, 1] with cost: 0
Iteration 576: [2, 0, 3, 1] with cost: 0
Iteration 577: [2, 0, 3, 1] with cost: 0
Iteration 578: [2, 0, 3, 1] with cost: 0
Iteration 579: [2, 0, 3, 1] with cost: 0
Iteration 580: [2, 0, 3, 1] with cost: 0

Iteration 581: [2, 0, 3, 1] with cost: 0
Iteration 582: [2, 0, 3, 1] with cost: 0
Iteration 583: [2, 0, 3, 1] with cost: 0
Iteration 584: [2, 0, 3, 1] with cost: 0
Iteration 585: [2, 0, 3, 1] with cost: 0
Iteration 586: [2, 0, 3, 1] with cost: 0
Iteration 587: [2, 0, 3, 1] with cost: 0
Iteration 588: [2, 0, 3, 1] with cost: 0
Iteration 589: [2, 0, 3, 1] with cost: 0
Iteration 590: [2, 0, 3, 1] with cost: 0
Iteration 591: [2, 0, 3, 1] with cost: 0
Iteration 592: [2, 0, 3, 1] with cost: 0
Iteration 593: [2, 0, 3, 1] with cost: 0
Iteration 594: [2, 0, 3, 1] with cost: 0
Iteration 595: [2, 0, 3, 1] with cost: 0
Iteration 596: [2, 0, 3, 1] with cost: 0
Iteration 597: [2, 0, 3, 1] with cost: 0
Iteration 598: [2, 0, 3, 1] with cost: 0
Iteration 599: [2, 0, 3, 1] with cost: 0
Iteration 600: [2, 0, 3, 1] with cost: 0
Iteration 601: [2, 0, 3, 1] with cost: 0
Iteration 602: [2, 0, 3, 1] with cost: 0
Iteration 603: [2, 0, 3, 1] with cost: 0
Iteration 604: [2, 0, 3, 1] with cost: 0
Iteration 605: [2, 0, 3, 1] with cost: 0
Iteration 606: [2, 0, 3, 1] with cost: 0
Iteration 607: [2, 0, 3, 1] with cost: 0
Iteration 608: [2, 0, 3, 1] with cost: 0
Iteration 609: [2, 0, 3, 1] with cost: 0
Iteration 610: [2, 0, 3, 1] with cost: 0
Iteration 611: [2, 0, 3, 1] with cost: 0
Iteration 612: [2, 0, 3, 1] with cost: 0
Iteration 613: [2, 0, 3, 1] with cost: 0
Iteration 614: [2, 0, 3, 1] with cost: 0
Iteration 615: [2, 0, 3, 1] with cost: 0
Iteration 616: [2, 0, 3, 1] with cost: 0
Iteration 617: [2, 0, 3, 1] with cost: 0
Iteration 618: [2, 0, 3, 1] with cost: 0
Iteration 619: [2, 0, 3, 1] with cost: 0
Iteration 620: [2, 0, 3, 1] with cost: 0
Iteration 621: [2, 0, 3, 1] with cost: 0
Iteration 622: [2, 0, 3, 1] with cost: 0
Iteration 623: [2, 0, 3, 1] with cost: 0
Iteration 624: [2, 0, 3, 1] with cost: 0

Iteration 625: [2, 0, 3, 1] with cost: 0
Iteration 626: [2, 0, 3, 1] with cost: 0
Iteration 627: [2, 0, 3, 1] with cost: 0
Iteration 628: [2, 0, 3, 1] with cost: 0
Iteration 629: [2, 0, 3, 1] with cost: 0
Iteration 630: [2, 0, 3, 1] with cost: 0
Iteration 631: [2, 0, 3, 1] with cost: 0
Iteration 632: [2, 0, 3, 1] with cost: 0
Iteration 633: [2, 0, 3, 1] with cost: 0
Iteration 634: [2, 0, 3, 1] with cost: 0
Iteration 635: [2, 0, 3, 1] with cost: 0
Iteration 636: [2, 0, 3, 1] with cost: 0
Iteration 637: [2, 0, 3, 1] with cost: 0
Iteration 638: [2, 0, 3, 1] with cost: 0
Iteration 639: [2, 0, 3, 1] with cost: 0
Iteration 640: [2, 0, 3, 1] with cost: 0
Iteration 641: [2, 0, 3, 1] with cost: 0
Iteration 642: [2, 0, 3, 1] with cost: 0
Iteration 643: [2, 0, 3, 1] with cost: 0
Iteration 644: [2, 0, 3, 1] with cost: 0
Iteration 645: [2, 0, 3, 1] with cost: 0
Iteration 646: [2, 0, 3, 1] with cost: 0
Iteration 647: [2, 0, 3, 1] with cost: 0
Iteration 648: [2, 0, 3, 1] with cost: 0
Iteration 649: [2, 0, 3, 1] with cost: 0
Iteration 650: [2, 0, 3, 1] with cost: 0
Iteration 651: [2, 0, 3, 1] with cost: 0
Iteration 652: [2, 0, 3, 1] with cost: 0
Iteration 653: [2, 0, 3, 1] with cost: 0
Iteration 654: [2, 0, 3, 1] with cost: 0
Iteration 655: [2, 0, 3, 1] with cost: 0
Iteration 656: [2, 0, 3, 1] with cost: 0
Iteration 657: [2, 0, 3, 1] with cost: 0
Iteration 658: [2, 0, 3, 1] with cost: 0
Iteration 659: [2, 0, 3, 1] with cost: 0
Iteration 660: [2, 0, 3, 1] with cost: 0
Iteration 661: [2, 0, 3, 1] with cost: 0
Iteration 662: [2, 0, 3, 1] with cost: 0
Iteration 663: [2, 0, 3, 1] with cost: 0
Iteration 664: [2, 0, 3, 1] with cost: 0
Iteration 665: [2, 0, 3, 1] with cost: 0
Iteration 666: [2, 0, 3, 1] with cost: 0
Iteration 667: [2, 0, 3, 1] with cost: 0
Iteration 668: [2, 0, 3, 1] with cost: 0

Iteration 669: [2, 0, 3, 1] with cost: 0
Iteration 670: [2, 0, 3, 1] with cost: 0
Iteration 671: [2, 0, 3, 1] with cost: 0
Iteration 672: [2, 0, 3, 1] with cost: 0
Iteration 673: [2, 0, 3, 1] with cost: 0
Iteration 674: [2, 0, 3, 1] with cost: 0
Iteration 675: [2, 0, 3, 1] with cost: 0
Iteration 676: [2, 0, 3, 1] with cost: 0
Iteration 677: [2, 0, 3, 1] with cost: 0
Iteration 678: [2, 0, 3, 1] with cost: 0
Iteration 679: [2, 0, 3, 1] with cost: 0
Iteration 680: [2, 0, 3, 1] with cost: 0
Iteration 681: [2, 0, 3, 1] with cost: 0
Iteration 682: [2, 0, 3, 1] with cost: 0
Iteration 683: [2, 0, 3, 1] with cost: 0
Iteration 684: [2, 0, 3, 1] with cost: 0
Iteration 685: [2, 0, 3, 1] with cost: 0
Iteration 686: [2, 0, 3, 1] with cost: 0
Iteration 687: [2, 0, 3, 1] with cost: 0
Iteration 688: [2, 0, 3, 1] with cost: 0
Iteration 689: [2, 0, 3, 1] with cost: 0
Iteration 690: [2, 0, 3, 1] with cost: 0
Iteration 691: [2, 0, 3, 1] with cost: 0
Iteration 692: [2, 0, 3, 1] with cost: 0
Iteration 693: [2, 0, 3, 1] with cost: 0
Iteration 694: [2, 0, 3, 1] with cost: 0
Iteration 695: [2, 0, 3, 1] with cost: 0
Iteration 696: [2, 0, 3, 1] with cost: 0
Iteration 697: [2, 0, 3, 1] with cost: 0
Iteration 698: [2, 0, 3, 1] with cost: 0
Iteration 699: [2, 0, 3, 1] with cost: 0
Iteration 700: [2, 0, 3, 1] with cost: 0
Iteration 701: [2, 0, 3, 1] with cost: 0
Iteration 702: [2, 0, 3, 1] with cost: 0
Iteration 703: [2, 0, 3, 1] with cost: 0
Iteration 704: [2, 0, 3, 1] with cost: 0
Iteration 705: [2, 0, 3, 1] with cost: 0
Iteration 706: [2, 0, 3, 1] with cost: 0
Iteration 707: [2, 0, 3, 1] with cost: 0
Iteration 708: [2, 0, 3, 1] with cost: 0
Iteration 709: [2, 0, 3, 1] with cost: 0
Iteration 710: [2, 0, 3, 1] with cost: 0
Iteration 711: [2, 0, 3, 1] with cost: 0
Iteration 712: [2, 0, 3, 1] with cost: 0

Iteration 713: [2, 0, 3, 1] with cost: 0
Iteration 714: [2, 0, 3, 1] with cost: 0
Iteration 715: [2, 0, 3, 1] with cost: 0
Iteration 716: [2, 0, 3, 1] with cost: 0
Iteration 717: [2, 0, 3, 1] with cost: 0
Iteration 718: [2, 0, 3, 1] with cost: 0
Iteration 719: [2, 0, 3, 1] with cost: 0
Iteration 720: [2, 0, 3, 1] with cost: 0
Iteration 721: [2, 0, 3, 1] with cost: 0
Iteration 722: [2, 0, 3, 1] with cost: 0
Iteration 723: [2, 0, 3, 1] with cost: 0
Iteration 724: [2, 0, 3, 1] with cost: 0
Iteration 725: [2, 0, 3, 1] with cost: 0
Iteration 726: [2, 0, 3, 1] with cost: 0
Iteration 727: [2, 0, 3, 1] with cost: 0
Iteration 728: [2, 0, 3, 1] with cost: 0
Iteration 729: [2, 0, 3, 1] with cost: 0
Iteration 730: [2, 0, 3, 1] with cost: 0
Iteration 731: [2, 0, 3, 1] with cost: 0
Iteration 732: [2, 0, 3, 1] with cost: 0
Iteration 733: [2, 0, 3, 1] with cost: 0
Iteration 734: [2, 0, 3, 1] with cost: 0
Iteration 735: [2, 0, 3, 1] with cost: 0
Iteration 736: [2, 0, 3, 1] with cost: 0
Iteration 737: [2, 0, 3, 1] with cost: 0
Iteration 738: [2, 0, 3, 1] with cost: 0
Iteration 739: [2, 0, 3, 1] with cost: 0
Iteration 740: [2, 0, 3, 1] with cost: 0
Iteration 741: [2, 0, 3, 1] with cost: 0
Iteration 742: [2, 0, 3, 1] with cost: 0
Iteration 743: [2, 0, 3, 1] with cost: 0
Iteration 744: [2, 0, 3, 1] with cost: 0
Iteration 745: [2, 0, 3, 1] with cost: 0
Iteration 746: [2, 0, 3, 1] with cost: 0
Iteration 747: [2, 0, 3, 1] with cost: 0
Iteration 748: [2, 0, 3, 1] with cost: 0
Iteration 749: [2, 0, 3, 1] with cost: 0
Iteration 750: [2, 0, 3, 1] with cost: 0
Iteration 751: [2, 0, 3, 1] with cost: 0
Iteration 752: [2, 0, 3, 1] with cost: 0
Iteration 753: [2, 0, 3, 1] with cost: 0
Iteration 754: [2, 0, 3, 1] with cost: 0
Iteration 755: [2, 0, 3, 1] with cost: 0
Iteration 756: [2, 0, 3, 1] with cost: 0

Iteration 757: [2, 0, 3, 1] with cost: 0
Iteration 758: [2, 0, 3, 1] with cost: 0
Iteration 759: [2, 0, 3, 1] with cost: 0
Iteration 760: [2, 0, 3, 1] with cost: 0
Iteration 761: [2, 0, 3, 1] with cost: 0
Iteration 762: [2, 0, 3, 1] with cost: 0
Iteration 763: [2, 0, 3, 1] with cost: 0
Iteration 764: [2, 0, 3, 1] with cost: 0
Iteration 765: [2, 0, 3, 1] with cost: 0
Iteration 766: [2, 0, 3, 1] with cost: 0
Iteration 767: [2, 0, 3, 1] with cost: 0
Iteration 768: [2, 0, 3, 1] with cost: 0
Iteration 769: [2, 0, 3, 1] with cost: 0
Iteration 770: [2, 0, 3, 1] with cost: 0
Iteration 771: [2, 0, 3, 1] with cost: 0
Iteration 772: [2, 0, 3, 1] with cost: 0
Iteration 773: [2, 0, 3, 1] with cost: 0
Iteration 774: [2, 0, 3, 1] with cost: 0
Iteration 775: [2, 0, 3, 1] with cost: 0
Iteration 776: [2, 0, 3, 1] with cost: 0
Iteration 777: [2, 0, 3, 1] with cost: 0
Iteration 778: [2, 0, 3, 1] with cost: 0
Iteration 779: [2, 0, 3, 1] with cost: 0
Iteration 780: [2, 0, 3, 1] with cost: 0
Iteration 781: [2, 0, 3, 1] with cost: 0
Iteration 782: [2, 0, 3, 1] with cost: 0
Iteration 783: [2, 0, 3, 1] with cost: 0
Iteration 784: [2, 0, 3, 1] with cost: 0
Iteration 785: [2, 0, 3, 1] with cost: 0
Iteration 786: [2, 0, 3, 1] with cost: 0
Iteration 787: [2, 0, 3, 1] with cost: 0
Iteration 788: [2, 0, 3, 1] with cost: 0
Iteration 789: [2, 0, 3, 1] with cost: 0
Iteration 790: [2, 0, 3, 1] with cost: 0
Iteration 791: [2, 0, 3, 1] with cost: 0
Iteration 792: [2, 0, 3, 1] with cost: 0
Iteration 793: [2, 0, 3, 1] with cost: 0
Iteration 794: [2, 0, 3, 1] with cost: 0
Iteration 795: [2, 0, 3, 1] with cost: 0
Iteration 796: [2, 0, 3, 1] with cost: 0
Iteration 797: [2, 0, 3, 1] with cost: 0
Iteration 798: [2, 0, 3, 1] with cost: 0
Iteration 799: [2, 0, 3, 1] with cost: 0
Iteration 800: [2, 0, 3, 1] with cost: 0

Iteration 801: [2, 0, 3, 1] with cost: 0
Iteration 802: [2, 0, 3, 1] with cost: 0
Iteration 803: [2, 0, 3, 1] with cost: 0
Iteration 804: [2, 0, 3, 1] with cost: 0
Iteration 805: [2, 0, 3, 1] with cost: 0
Iteration 806: [2, 0, 3, 1] with cost: 0
Iteration 807: [2, 0, 3, 1] with cost: 0
Iteration 808: [2, 0, 3, 1] with cost: 0
Iteration 809: [2, 0, 3, 1] with cost: 0
Iteration 810: [2, 0, 3, 1] with cost: 0
Iteration 811: [2, 0, 3, 1] with cost: 0
Iteration 812: [2, 0, 3, 1] with cost: 0
Iteration 813: [2, 0, 3, 1] with cost: 0
Iteration 814: [2, 0, 3, 1] with cost: 0
Iteration 815: [2, 0, 3, 1] with cost: 0
Iteration 816: [2, 0, 3, 1] with cost: 0
Iteration 817: [2, 0, 3, 1] with cost: 0
Iteration 818: [2, 0, 3, 1] with cost: 0
Iteration 819: [2, 0, 3, 1] with cost: 0
Iteration 820: [2, 0, 3, 1] with cost: 0
Iteration 821: [2, 0, 3, 1] with cost: 0
Iteration 822: [2, 0, 3, 1] with cost: 0
Iteration 823: [2, 0, 3, 1] with cost: 0
Iteration 824: [2, 0, 3, 1] with cost: 0
Iteration 825: [2, 0, 3, 1] with cost: 0
Iteration 826: [2, 0, 3, 1] with cost: 0
Iteration 827: [2, 0, 3, 1] with cost: 0
Iteration 828: [2, 0, 3, 1] with cost: 0
Iteration 829: [2, 0, 3, 1] with cost: 0
Iteration 830: [2, 0, 3, 1] with cost: 0
Iteration 831: [2, 0, 3, 1] with cost: 0
Iteration 832: [2, 0, 3, 1] with cost: 0
Iteration 833: [2, 0, 3, 1] with cost: 0
Iteration 834: [2, 0, 3, 1] with cost: 0
Iteration 835: [2, 0, 3, 1] with cost: 0
Iteration 836: [2, 0, 3, 1] with cost: 0
Iteration 837: [2, 0, 3, 1] with cost: 0
Iteration 838: [2, 0, 3, 1] with cost: 0
Iteration 839: [2, 0, 3, 1] with cost: 0
Iteration 840: [2, 0, 3, 1] with cost: 0
Iteration 841: [2, 0, 3, 1] with cost: 0
Iteration 842: [2, 0, 3, 1] with cost: 0
Iteration 843: [2, 0, 3, 1] with cost: 0
Iteration 844: [2, 0, 3, 1] with cost: 0

Iteration 845: [2, 0, 3, 1] with cost: 0
Iteration 846: [2, 0, 3, 1] with cost: 0
Iteration 847: [2, 0, 3, 1] with cost: 0
Iteration 848: [2, 0, 3, 1] with cost: 0
Iteration 849: [2, 0, 3, 1] with cost: 0
Iteration 850: [2, 0, 3, 1] with cost: 0
Iteration 851: [2, 0, 3, 1] with cost: 0
Iteration 852: [2, 0, 3, 1] with cost: 0
Iteration 853: [2, 0, 3, 1] with cost: 0
Iteration 854: [2, 0, 3, 1] with cost: 0
Iteration 855: [2, 0, 3, 1] with cost: 0
Iteration 856: [2, 0, 3, 1] with cost: 0
Iteration 857: [2, 0, 3, 1] with cost: 0
Iteration 858: [2, 0, 3, 1] with cost: 0
Iteration 859: [2, 0, 3, 1] with cost: 0
Iteration 860: [2, 0, 3, 1] with cost: 0
Iteration 861: [2, 0, 3, 1] with cost: 0
Iteration 862: [2, 0, 3, 1] with cost: 0
Iteration 863: [2, 0, 3, 1] with cost: 0
Iteration 864: [2, 0, 3, 1] with cost: 0
Iteration 865: [2, 0, 3, 1] with cost: 0
Iteration 866: [2, 0, 3, 1] with cost: 0
Iteration 867: [2, 0, 3, 1] with cost: 0
Iteration 868: [2, 0, 3, 1] with cost: 0
Iteration 869: [2, 0, 3, 1] with cost: 0
Iteration 870: [2, 0, 3, 1] with cost: 0
Iteration 871: [2, 0, 3, 1] with cost: 0
Iteration 872: [2, 0, 3, 1] with cost: 0
Iteration 873: [2, 0, 3, 1] with cost: 0
Iteration 874: [2, 0, 3, 1] with cost: 0
Iteration 875: [2, 0, 3, 1] with cost: 0
Iteration 876: [2, 0, 3, 1] with cost: 0
Iteration 877: [2, 0, 3, 1] with cost: 0
Iteration 878: [2, 0, 3, 1] with cost: 0
Iteration 879: [2, 0, 3, 1] with cost: 0
Iteration 880: [2, 0, 3, 1] with cost: 0
Iteration 881: [2, 0, 3, 1] with cost: 0
Iteration 882: [2, 0, 3, 1] with cost: 0
Iteration 883: [2, 0, 3, 1] with cost: 0
Iteration 884: [2, 0, 3, 1] with cost: 0
Iteration 885: [2, 0, 3, 1] with cost: 0
Iteration 886: [2, 0, 3, 1] with cost: 0
Iteration 887: [2, 0, 3, 1] with cost: 0
Iteration 888: [2, 0, 3, 1] with cost: 0

Iteration 889: [2, 0, 3, 1] with cost: 0
Iteration 890: [2, 0, 3, 1] with cost: 0
Iteration 891: [2, 0, 3, 1] with cost: 0
Iteration 892: [2, 0, 3, 1] with cost: 0
Iteration 893: [2, 0, 3, 1] with cost: 0
Iteration 894: [2, 0, 3, 1] with cost: 0
Iteration 895: [2, 0, 3, 1] with cost: 0
Iteration 896: [2, 0, 3, 1] with cost: 0
Iteration 897: [2, 0, 3, 1] with cost: 0
Iteration 898: [2, 0, 3, 1] with cost: 0
Iteration 899: [2, 0, 3, 1] with cost: 0
Iteration 900: [2, 0, 3, 1] with cost: 0
Iteration 901: [2, 0, 3, 1] with cost: 0
Iteration 902: [2, 0, 3, 1] with cost: 0
Iteration 903: [2, 0, 3, 1] with cost: 0
Iteration 904: [2, 0, 3, 1] with cost: 0
Iteration 905: [2, 0, 3, 1] with cost: 0
Iteration 906: [2, 0, 3, 1] with cost: 0
Iteration 907: [2, 0, 3, 1] with cost: 0
Iteration 908: [2, 0, 3, 1] with cost: 0
Iteration 909: [2, 0, 3, 1] with cost: 0
Iteration 910: [2, 0, 3, 1] with cost: 0
Iteration 911: [2, 0, 3, 1] with cost: 0
Iteration 912: [2, 0, 3, 1] with cost: 0
Iteration 913: [2, 0, 3, 1] with cost: 0
Iteration 914: [2, 0, 3, 1] with cost: 0
Iteration 915: [2, 0, 3, 1] with cost: 0
Iteration 916: [2, 0, 3, 1] with cost: 0
Iteration 917: [2, 0, 3, 1] with cost: 0
Iteration 918: [2, 0, 3, 1] with cost: 0
Iteration 919: [2, 0, 3, 1] with cost: 0
Iteration 920: [2, 0, 3, 1] with cost: 0
Iteration 921: [2, 0, 3, 1] with cost: 0
Iteration 922: [2, 0, 3, 1] with cost: 0
Iteration 923: [2, 0, 3, 1] with cost: 0
Iteration 924: [2, 0, 3, 1] with cost: 0
Iteration 925: [2, 0, 3, 1] with cost: 0
Iteration 926: [2, 0, 3, 1] with cost: 0
Iteration 927: [2, 0, 3, 1] with cost: 0
Iteration 928: [2, 0, 3, 1] with cost: 0
Iteration 929: [2, 0, 3, 1] with cost: 0
Iteration 930: [2, 0, 3, 1] with cost: 0
Iteration 931: [2, 0, 3, 1] with cost: 0
Iteration 932: [2, 0, 3, 1] with cost: 0

Iteration 933: [2, 0, 3, 1] with cost: 0
Iteration 934: [2, 0, 3, 1] with cost: 0
Iteration 935: [2, 0, 3, 1] with cost: 0
Iteration 936: [2, 0, 3, 1] with cost: 0
Iteration 937: [2, 0, 3, 1] with cost: 0
Iteration 938: [2, 0, 3, 1] with cost: 0
Iteration 939: [2, 0, 3, 1] with cost: 0
Iteration 940: [2, 0, 3, 1] with cost: 0
Iteration 941: [2, 0, 3, 1] with cost: 0
Iteration 942: [2, 0, 3, 1] with cost: 0
Iteration 943: [2, 0, 3, 1] with cost: 0
Iteration 944: [2, 0, 3, 1] with cost: 0
Iteration 945: [2, 0, 3, 1] with cost: 0
Iteration 946: [2, 0, 3, 1] with cost: 0
Iteration 947: [2, 0, 3, 1] with cost: 0
Iteration 948: [2, 0, 3, 1] with cost: 0
Iteration 949: [2, 0, 3, 1] with cost: 0
Iteration 950: [2, 0, 3, 1] with cost: 0
Iteration 951: [2, 0, 3, 1] with cost: 0
Iteration 952: [2, 0, 3, 1] with cost: 0
Iteration 953: [2, 0, 3, 1] with cost: 0
Iteration 954: [2, 0, 3, 1] with cost: 0
Iteration 955: [2, 0, 3, 1] with cost: 0
Iteration 956: [2, 0, 3, 1] with cost: 0
Iteration 957: [2, 0, 3, 1] with cost: 0
Iteration 958: [2, 0, 3, 1] with cost: 0
Iteration 959: [2, 0, 3, 1] with cost: 0
Iteration 960: [2, 0, 3, 1] with cost: 0
Iteration 961: [2, 0, 3, 1] with cost: 0
Iteration 962: [2, 0, 3, 1] with cost: 0
Iteration 963: [2, 0, 3, 1] with cost: 0
Iteration 964: [2, 0, 3, 1] with cost: 0
Iteration 965: [2, 0, 3, 1] with cost: 0
Iteration 966: [2, 0, 3, 1] with cost: 0
Iteration 967: [2, 0, 3, 1] with cost: 0
Iteration 968: [2, 0, 3, 1] with cost: 0
Iteration 969: [2, 0, 3, 1] with cost: 0
Iteration 970: [2, 0, 3, 1] with cost: 0
Iteration 971: [2, 0, 3, 1] with cost: 0
Iteration 972: [2, 0, 3, 1] with cost: 0
Iteration 973: [2, 0, 3, 1] with cost: 0
Iteration 974: [2, 0, 3, 1] with cost: 0
Iteration 975: [2, 0, 3, 1] with cost: 0
Iteration 976: [2, 0, 3, 1] with cost: 0

Iteration 977: [2, 0, 3, 1] with cost: 0
Iteration 978: [2, 0, 3, 1] with cost: 0
Iteration 979: [2, 0, 3, 1] with cost: 0
Iteration 980: [2, 0, 3, 1] with cost: 0
Iteration 981: [2, 0, 3, 1] with cost: 0
Iteration 982: [2, 0, 3, 1] with cost: 0
Iteration 983: [2, 0, 3, 1] with cost: 0
Iteration 984: [2, 0, 3, 1] with cost: 0
Iteration 985: [2, 0, 3, 1] with cost: 0
Iteration 986: [2, 0, 3, 1] with cost: 0
Iteration 987: [2, 0, 3, 1] with cost: 0
Iteration 988: [2, 0, 3, 1] with cost: 0
Iteration 989: [2, 0, 3, 1] with cost: 0
Iteration 990: [2, 0, 3, 1] with cost: 0
Iteration 991: [2, 0, 3, 1] with cost: 0
Iteration 992: [2, 0, 3, 1] with cost: 0
Iteration 993: [2, 0, 3, 1] with cost: 0
Iteration 994: [2, 0, 3, 1] with cost: 0
Iteration 995: [2, 0, 3, 1] with cost: 0
Iteration 996: [2, 0, 3, 1] with cost: 0
Iteration 997: [2, 0, 3, 1] with cost: 0
Iteration 998: [2, 0, 3, 1] with cost: 0
Iteration 999: [2, 0, 3, 1] with cost: 0
Iteration 1000: [2, 0, 3, 1] with cost: 0

Final solution: [2, 0, 3, 1] with cost: 0

LAB-06
Creating Knowledge base using propositional logic

ALGORITHM:

LAB-06

Project title :- Create a knowledge base using propositional logic and throw that the given query entails the knowledge base or not

Algorithm :-

Input :

kb (knowledge base expression)

q (query expression)

combinations (all possible truth assignments for variables)

* Generate all possible truth combinations for the variables combination = [(True, True, True)...
* Convert the infix expressions for kb & q to postfix notation

kb_postfix = toPostfix(kb)

query_postfix = toPostfix(q)

* Evaluate both kb and query expressions for each combinations in the truth table

For each combinations (comb) in combinations:

S = evaluatePostfix (kb_postfix, comb) # Evaluate query kb

f = evaluatePostfix (query_postfix, comb) # Evaluate query

if S is true & f is false:

Return false

Return True

output

if the result is True, print the knowledge Base entails query"

Else, print "the knowledge Base does not entail query".

State S
Expressio
State

1 T
2 T
3 T
4 T
5 F
6 F
7
8

Result

The k
trees
state

Narrate. M
19/11/2024

State Space

Expression : $p \wedge q$, Query : r

State	p	q	r	KB Eval	Query Eval	kb → Query entails
1	True	True	True	True	True	True
2	True	True	False	True	False	False
3	True	False	True	False	True	True
4	True	False	False	False	False	True
5	False	True	True	False	True	True
6	False	True	False	False	False	True
7	False	False	True	False	True	True
8	False	False	False	False	False	True

Result :-

The knowledge base does not entail query since we found KB : True & Query : false in state 2

Namrata M.
19/11/2024

CODE:

```
combinations = [
    (True, True, True), (True, True, False), (True, False, True),
(True, False, False),
    (False, True, True), (False, True, False), (False, False, True),
(False, False, False)
]

variable = {'p': 0, 'q': 1, 'r': 2}
kb = ''
q = ''
priority = {'~': 3, 'v': 1, '^': 2}

def input_rules():
    global kb, q
    kb = input("Enter rule: ")
    q = input("Enter the Query: ")

def entailment():
    global kb, q
    print('*' * 10 + " Truth Table Reference " + '*' * 10)
    print('kb', 'alpha')
    print('*' * 10)

    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(s, f)
        print('-' * 10)

        if s and not f:
            return False
    return True

def isOperand(c):
    return c.isalpha() and c != 'v'

def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'
```

```
def isEmpty(stack):
    return len(stack) == 0

def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False

def toPostfix(infix):
    stack = []
    postfix = ''

    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and
hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)

    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix

def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
```

```
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i, val2, val1))

    return stack.pop()

def _eval(i, val1, val2):
    if i == '^':
        return val2 and val1
    return val2 or val1

input_rules()
ans = entailment()

if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

OUTPUT:

Enter rule: ~p
Enter the Query: p
********** Truth Table Reference **********
kb alpha
****
False True

False True

False True

False True

True False

The Knowledge Base does not entail query

LAB-07
Implementing Unification in First Order Logic

ALGORITHM:

LAB-07

Project title :- Unification Algorithm for first order logic
Implementation - Logic

Algorithm :-
Input : Two expressions E_1 & E_2
Output : A substitution θ or failure E_1 and E_2 cannot be unified.

Step1 Initialize Solution :
Start with an empty substitution $\theta = \{\}$

Step2 Compare Expression :
Case 1 : E_1 & E_2 are constants
- If $E_1 = E_2$, return θ (no substitution needed)
- If $E_1 \neq E_2$, return failure

Case 2 : Either E_1 or E_2 is a variable
Let X be the variable
If $x \in \theta$ (not yet substituted) :
- Add $\{X \mapsto E\}$ to θ, where E is the other term
- Ensure no cyclic substitution (e.g. $X \neq E$).

If $X \in \theta$, recursively unify $\theta[X]$ with other term.

Case 3 : E_1 & E_2 have their functors (function names) and arguments.
Example $f(a,b) \rightarrow$ Functor : f, Arguments (a,b)
If their functors differs, return failure
Otherwise, recursively unify each pair of arguments and accumulate substitutions

Case 4 : E_1 or E_2 is neither a variable nor a term (unexpected case)
Return failure.

Step3 Apply substitution.
After obtaining θ, apply it consistently to all variables E_1 & E_2

Step 4 Repeat until fully unified or failure.

- If O is complete, return 1 as the result.
- If unification fails at any recursive step, return failure.

CODE:

```
def unify(psi1, psi2):
    if isinstance(psi1, str) and psi1.islower():
        if psi1 == psi2:
            return []
        elif occurs_check(psi1, psi2):
            return "FAILURE"
        else:
            return [(psi1, psi2)]
    elif isinstance(psi2, str) and psi2.islower():
        if occurs_check(psi2, psi1):
            return "FAILURE"
        else:
            return [(psi2, psi1)]
    elif psi1 == psi2:
        return []
    elif isinstance(psi1, list) and isinstance(psi2, list):
        if psi1[0] != psi2[0]:
            return "FAILURE"
        if len(psi1) != len(psi2):
            return "FAILURE"
        subst = []
        for i in range(len(psi1)):
            s = unify(apply_substitution(subst, psi1[i]),
apply_substitution(subst, psi2[i]))
            if s == "FAILURE":
                return "FAILURE"
            subst.extend(s)
        return subst
    else:
        return "FAILURE"


def occurs_check(var, term):
    """
    Checks if variable `var` occurs in term `term`.
    """
    if var == term:
        return True
    elif isinstance(term, list):
        return any(occurs_check(var, subterm) for subterm in term)
    return False
```

```
def apply_substitution(subst, term):
    """
    Applies a set of substitutions to a term.
    """
    if isinstance(term, str):
        for var, replacement in subst:
            if term == var:
                return apply_substitution(subst, replacement)
        return term
    elif isinstance(term, list):
        return [apply_substitution(subst, subterm) for subterm in
term]
    return term


psi1 = ["P", "x", "y"]
psi2 = ["P", "a", "y"]

result = unify(psi1, psi2)
print("Unification Result:", result)

psi1 = ["P", "x", "y"]
psi2 = ["P", "x", "y"]

result = unify(psi1, psi2)
print("Unification Result:", result)

psi1 = ["P", "x", "y"]
psi2 = ["B", "x", "y"]

result = unify(psi1, psi2)
print("Unification Result:", result)

psi1 = ["P", "x", "y"]
psi2 = ["P", "a", "b"]

result = unify(psi1, psi2)
print("Unification Result:", result)

psi1 = ["P", "x", "y"]
psi2 = ["P", "x", "b"]

result = unify(psi1, psi2)
```

```
print("Unification Result:", result)
```

OUTPUT:

```
Unification Result: [('x', 'a')]
Unification Result: []
Unification Result: FAILURE
Unification Result: [('x', 'a'), ('y', 'b')]
Unification Result: [('y', 'b')]
```

LAB-08
Creating knowledge base of FOL and proving query using forward reasoning

ALGORITHM:

failure.
the result.
step, return

LAB-08

Project title :- Create a knowledge base consisting of FOL logic statements and prove the given query using forward reasoning.

Forward Reasoning Algorithm :-

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

inputs: KB, knowledge base, a set of first or definite clauses, α, the query, an atomic sentences.

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

repeat until new is empty
  new ← { }
  for each rule in KB do
    (p1 ∧ ... pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
    for each θ such that SUBST (θ, p1 ∧ ... ∧ pn)
        = SUBST (θ, p'1 ∧ ... ∧ p'n)
    for some p'1, ... 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.

PTO

Given case study :-

As per the law, it is a crime for an American to sell weapon to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert who is an American citizen. Prove that 'Robert is criminal'.

Representation in FOL

1. It is a crime for an american to sell weapon to hostile nations.
Let's say p, q, and r are variables.
American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

2. Country A has some missile.
∃x Owns(A, x) ∧ Missile(x).

Existential instantiation, introducing a new variable T1
Owns(A, T1)
Missile(T1)

3. All of the missiles was were sold to country A by Robert.
∀x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

4. Missiles are weapons
Missile(x) ⇒ Weapon(x)

5. Enemy of America is known as hostile.
∀x Enemy(x, America) ⇒ Hostile(x)

6. Robert is an American
American(Robert)

7. The country A, an enemy of America
Enemy(A, America).

output :

9> 'Criminal

forward

American

Nameeta. M.
3/12/24

r an American to
country A, an enemy of
all the missiles were
rica citizen.

to sell weapon to

Hostile(x) ⇒
riminal(p)

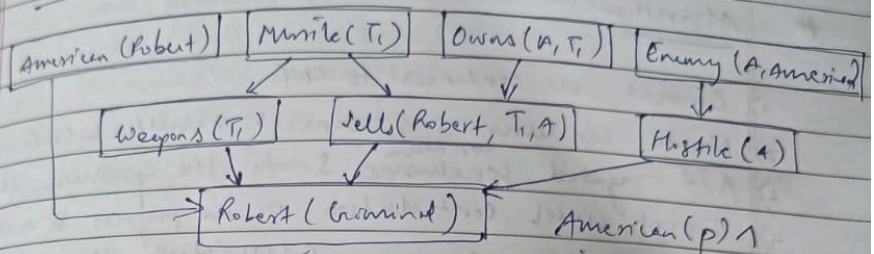
ew variable T_1

ntry A by Robert
bert, T_1, A)

output:

Is 'Criminal(Robert)' provable? True.

forward chaining proof



American(p) ∧
Weapon(q) ∧ Sells(p,q
∧ Hostiles ⇒ Criminal(p)

Namrata M.
3/12/2024

CODE:

```
class ForwardChainingFOL:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def unify(self, fact1, fact2):

        if fact1 == fact2:
            return {}
        if "(" in fact1 and "(" in fact2:

            pred1, args1 = fact1.split("(", 1)
            pred2, args2 = fact2.split("(", 1)
            args1 = args1[:-1].split(",")
            args2 = args2[:-1].split(",")
            if pred1 != pred2 or len(args1) != len(args2):
                return None

            substitution = {}
            for a1, a2 in zip(args1, args2):
                if a1 != a2:
                    if a1.islower():
                        substitution[a1] = a2
                    elif a2.islower():
                        substitution[a2] = a1
                    else:
                        return None
            return substitution
        return None

    def apply_substitution(self, fact, substitution):

        if "(" in fact:
            pred, args = fact.split("(", 1)
```

```
            args = args[:-1].split(",")
            substituted_args = [substitution.get(arg, arg) for arg in
args]
            return f"{pred}({','.join(substituted_args)})"
        return fact

    def forward_chain(self, goal):
        iteration = 1
        while True:
            new_facts = set()
            print(f"\n=== Iteration {iteration} ===")
            print("Known Facts:")
            for fact in self.facts:
                print(f"  - {fact}")

            print("\nApplying rules...")
            rule_triggered = False

            for premises, conclusion in self.rules:
                substitutions = [{}]
                for premise in premises:
                    new_substitutions = []
                    for fact in self.facts:
                        for sub in substitutions:
                            unified =
self.unify(self.apply_substitution(premise, sub), fact)
                            if unified is not None:
                                new_substitutions.append({**sub,
**unified})
                    substitutions = new_substitutions
                for sub in substitutions:
                    inferred_fact =
self.apply_substitution(conclusion, sub)
                    if inferred_fact not in self.facts:
                        rule_triggered = True
                        print(f"Rule triggered: {premises} →
{conclusion}")
                        print(f"  New fact inferred:
{inferred_fact}")
                        new_facts.add(inferred_fact)

            if not new_facts:
                if not rule_triggered:
                    print("No rules triggered in this iteration.")
                print("No new facts inferred in this iteration.")
```

```
                break

            self.facts.update(new_facts)
            if goal in self.facts:
                print(f"\nGoal {goal} reached!")
                return True
            iteration += 1

        print("\nGoal not reached.")
        return False


fc = ForwardChainingFOL()

fc.add_fact("American(Robert)")
fc.add_fact("Enemy(A,America)")
fc.add_fact("Owns(A,T1)")
fc.add_fact("Missile(T1)")

fc.add_rule(["Missile(T1)"], "Weapon(T1)")
fc.add_rule(["Enemy(A,America)"], "Hostile(A)")
fc.add_rule(["Missile(p)", "Owns(A,p)"], "Sells(Robert,p,A)")
fc.add_rule(["American(p)", "Weapon(q)", "Sells(p,q,r)",
"Hostile(r)"], "Criminal(p)")

goal = "Criminal(Robert)"

if fc.forward_chain(goal):
    print(f"\nFinal result: Goal achieved: {goal}")
else:
    print("\nFinal result: Goal not achieved.")
```

OUTPUT:

=== Iteration 1 ===
Known Facts:
- Owns(A,T1)
- Enemy(A,America)
- Missile(T1)
- American(Robert)

Applying rules...
Rule triggered: ['Missile(T1)'] → Weapon(T1)
New fact inferred: Weapon(T1)

Rule triggered: ['Enemy(A,America)'] → Hostile(A)
  New fact inferred: Hostile(A)
Rule triggered: ['Missile(p)', 'Owns(A,p)'] → Sells(Robert,p,A)
  New fact inferred: Sells(Robert,T1,A)

=== Iteration 2 ===
Known Facts:
  - Missile(T1)
  - Weapon(T1)
  - American(Robert)
  - Owns(A,T1)
  - Hostile(A)
  - Sells(Robert,T1,A)
  - Enemy(A,America)

Applying rules...
Rule triggered: ['American(p)', 'Weapon(q)', 'Sells(p,q,r)', 'Hostile(r)'] → Criminal(p)
  New fact inferred: Criminal(Robert)

Goal Criminal(Robert) reached!

Final result: Goal achieved: Criminal(Robert)

LAB-09
Creating knowledge base of FOL and proving query using Resolution

ALGORITHM:

LAB-08

Project title :- Create a knowledge base of first order logic and solve query using resolution

Algorithm :-

1) Convert all sentences to CNF
2) Negate conclusion S & Convert result to CNF
3) Add negated conclusion S to the premise clauses
4) Repeat until contradiction or no progress is made
   a) Select 2 clauses (call them parent clauses)
   b) Release them together, performing all required unification
   c) If resolvent is the empty clause, a contradiction has been found.
   d) If not, add resolvent to the premises. If we succeed in step 4, we have proved the ~~conclusion~~ condition

Proof by Resolution :

Given the KB or Premise

a) John likes all kind of food
b) Apple & vegetables are food
c) Anything anyone eats and not killed is food
d) Anil eats peanuts and still Alive
e) Harry eats anything that Anil eats.
f) Anyone who is alive implies not killed
g) Anyone who is alive implies not killed
h) John likes peanuts

⇒ Representation in FOL

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(Vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$
e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f) $\forall x : \neg killed(x) \rightarrow alive(x)$
g) $\forall x : alive(x) \rightarrow \neg killed(x)$
h) $likes(John, Peanuts)$

Eliminate Implications

a) $\forall x \neg food(x) \vee likes(John, x)$
c) $\forall x \forall y \neg [eats(x, y) \wedge \neg killed(x)] \vee food(y)$
e) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f) $\forall x \neg [\neg killed(x)] \vee alive(x)$
g) $\forall x \neg alive(x) \vee \neg killed(x)$

→ Move negation ($\neg$) inwards

c) $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
f) $\forall x killed(x) \vee alive(x)$

→ Standerize variables

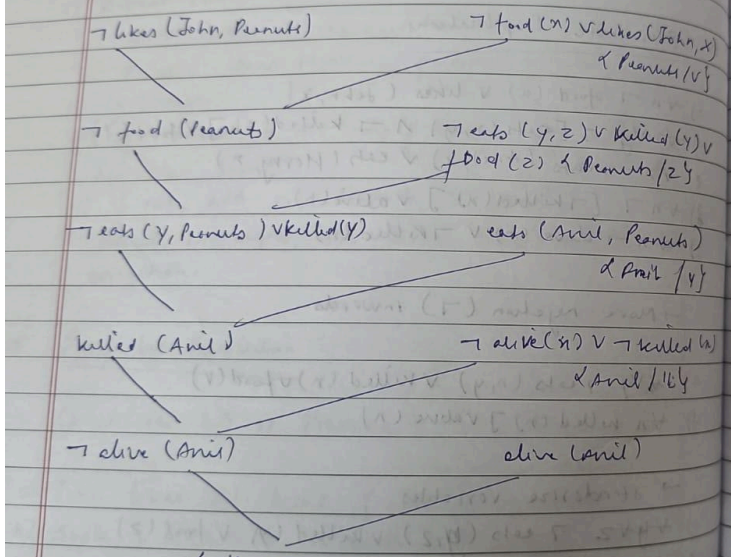
c) $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$
e) $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
f) $\forall g [killed(g)] \vee alive(g)$
g) $\forall k \neg alive(k) \vee \neg killed(k)$

→ Drop universal

a) $\neg food(x) \vee likes(John, x)$
b) $food(Apple)$
c) $food(Vegetables)$

c) food (vegetables)
d) ¬ eats (x z) ∨ killed (y) ∨ food (z)
e) eats (Anil, Peanuts)
f) alive (Anil)
g) ¬ eats (Anil, w) ∨ eats (Harry, w)
h) killed (y) ∨ alive (y)
i) ¬ alive (k) ∨ ¬ killed (k)

Prof :-



Project

Algorithm

↳ Alpha (α
path with
↳ Max wo
killed d
↳ In Both
which co
↳ Both n
same p
↳ Alpha(α
less tha

Problem.
Apply
node (H

CODE:

```
def negate(literal):
    """Return the negation of a literal."""
    if isinstance(literal, tuple) and literal[0] == "not":
        return literal[1]
    else:
        return ("not", literal)

def resolve(clause1, clause2):
    """Return the resolvent of two clauses."""
    resolvents = set()
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negate(literal2):
                resolvent = (clause1 - {literal1}) | (clause2 -
{literal2})
                print(f"    Resolving literal: {literal1} with
{literal2}")
                print(f"    Resulting Resolvent: {resolvent}")
                resolvents.add(frozenset(resolvent))
    return resolvents

def resolution_algorithm(KB, query):
    """Perform the resolution algorithm to check if the query can be
proven."""
    print("\n--- Step-by-Step Resolution Process ---")
    negated_query = negate(query)
    KB.append(frozenset([negated_query]))
    print(f"Negated Query Added to KB: {negated_query}")

    clauses = set(KB)

    step = 1
    while True:
        new_clauses = set()
        print(f"\nStep {step}: Resolving Clauses")
        for c1 in clauses:
            for c2 in clauses:
                if c1 != c2:
                    print(f"  Resolving clauses: {c1} and {c2}")
                    resolvent = resolve(c1, c2)
```

```
                    for res in resolvent:
                        if frozenset([]) in resolvent:
                            print("\nEmpty clause derived! The query
is provable.")
                            return True
                        new_clauses.add(res)

        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived. The query is not
provable.")
            return False

        clauses.update(new_clauses)
        step += 1

KB = [
    frozenset([("not", "food(x)"), ("likes", "John", "x")]),
    frozenset([("food", "Apple")]),
    frozenset([("food", "vegetables")]),
    frozenset([("not", "eats(y, z)"), ("killed", "y"), ("food",
"z")]),
    frozenset([("eats", "Anil", "Peanuts")]),
    frozenset([("alive", "Anil")]),
    frozenset([("not", "eats(Anil, w)"), ("eats", "Harry", "w")]),
    frozenset([("killed", "g"), ("alive", "g")]),
    frozenset([("not", "alive(k)"), ("not", "killed(k)")]),
    frozenset([("likes", "John", "Peanuts")])
]


query = ("likes", "John", "Peanuts")

result = resolution_algorithm(KB, query)
if result:
    print("\nQuery is provable.")
else:
    print("\nQuery is not provable.")
```

OUTPUT:

--- Step-by-Step Resolution Process ---
Negated Query Added to KB: ('not', ('likes', 'John', 'Peanuts'))

Step 1: Resolving Clauses

Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('alive', 'Anil')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})

Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('food', 'vegetables')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('food', 'Apple')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('likes', 'John', 'Peanuts')})

Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('likes', 'John', 'Peanuts')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('likes', 'John', 'x'), ('not', 'food(x)')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('not', 'eats(y, z)'), ('food', 'z'), ('killed', 'y')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('alive', 'Anil')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('food', 'vegetables')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('food', 'Apple')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('eats', 'Anil', 'Peanuts')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('alive', 'g'), ('killed', 'g')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('not', 'killed(k)'), ('not', 'alive(k)')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('not', 'eats(Anil, w)'), ('eats', 'Harry', 'w')})
Resolving clauses: frozenset({('likes', 'John', 'Peanuts')}) and frozenset({('not', ('likes', 'John', 'Peanuts'))})
Resolving literal: ('likes', 'John', 'Peanuts') with ('not', ('likes', 'John', 'Peanuts'))
Resulting Resolvent: frozenset()

Empty clause derived! The query is provable. Query is provable.

LAB-10
Alpha-Beta pruning algorithm

ALGORITHM:

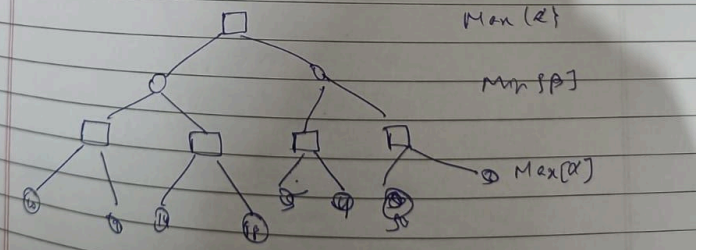
Project title :- Implement Alpha-Beta pruning

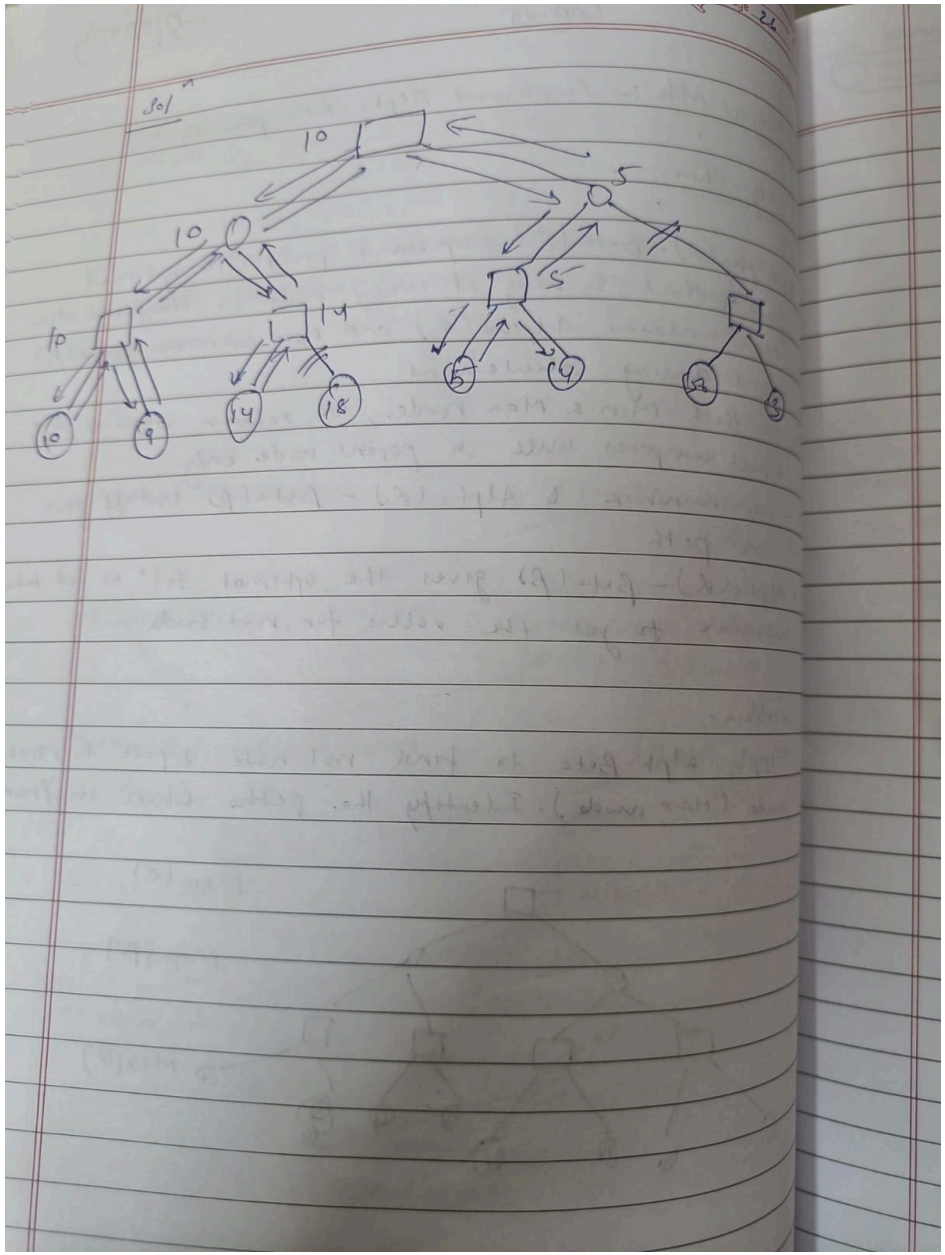
Algorithm :-

- Alpha(α) - Beta(β) proposes to find the optimal path without looking at every node in the game tree
- Max contains Alpha(α) and Min contains Beta(β) bound during calculations
- In Both Min & Max node, we return when α ≥ β which compares with its parent node only.
- Both minimax & Alpha(α) - Beta(β) cut-off give same path
- Alpha(α) - Beta(β) gives the optimal solution as it takes less time to get the value for root node.

Problem.

Apply alph-Beta to find root node & path to root node (MAX node). Identify the paths which are pruned




Sol
10
10
5
5
10
14
10
9
14
18
5
4
17
3

CODE:

```
def alpha_beta_search(state):
    def max_value(state, alpha, beta, path):
        print(f"MAX: Visiting state {state}, alpha={alpha},
beta={beta}")
        if terminal_test(state):
            print(f"MAX: Terminal state {state} has utility
{utility(state)}")
            return utility(state), path
        v = float('-inf')
        best_path = []
        for action in actions(state):
            result_state = result(state, action)
            value, new_path = min_value(result_state, alpha, beta,
path + [action])
            print(f"MAX: From state {state}, action {action} →
value={value}")
            if value > v:
                v = value
                best_path = new_path
            if v >= beta:
                print(f"MAX: Pruning at state {state} with value={v}
≥ beta={beta}")
                return v, best_path
            alpha = max(alpha, v)
        print(f"MAX: Returning value={v} for state {state}")
        return v, best_path

    def min_value(state, alpha, beta, path):
        print(f"MIN: Visiting state {state}, alpha={alpha},
beta={beta}")
        if terminal_test(state):
            print(f"MIN: Terminal state {state} has utility
{utility(state)}")
            return utility(state), path
        v = float('inf')
        best_path = []
        for action in actions(state):
            result_state = result(state, action)
            value, new_path = max_value(result_state, alpha, beta,
path + [action])
```

```
            print(f"MIN: From state {state}, action {action} →
value={value}")
            if value < v:
                v = value
                best_path = new_path
            if v <= alpha:
                print(f"MIN: Pruning at state {state} with value={v}
≤ alpha={alpha}")
                return v, best_path
            beta = min(beta, v)
        print(f"MIN: Returning value={v} for state {state}")
        return v, best_path

    print("Starting Alpha-Beta Search...\n")
    final_value, final_path = max_value(state, float('-inf'),
float('inf'), [state])
    return final_value, final_path


def terminal_test(state):
    return state in terminal_states

def utility(state):
    return terminal_states[state]

def actions(state):
    return game_tree.get(state, [])

def result(state, action):
    return action

game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H','I'],
    'E': ['J','K'],
    'F': ['L','M'],
    'G': ['N','O'],
    'H':[],
    'I':[],
    'J':[],
    'K':[],
    'L':[],
    'M':[],
```

```
    'N':[],
    'O':[]
}

terminal_states = {
    'H': 10,
    'I': 9,
    'J': 14,
    'K': 18,
    'L': 5,
    'M': 4,
    'N': 50,
    'O': 3
}

final_value, final_path = alpha_beta_search('A')
print("\n--- Final Results ---")
print(f"The final utility value is: {final_value}")
print(f"The final path taken is: {final_path}")
```

OUTPUT:

Starting Alpha-Beta Search...

MAX: Visiting state A, alpha=-inf, beta=inf
MIN: Visiting state B, alpha=-inf, beta=inf
MAX: Visiting state D, alpha=-inf, beta=inf
MIN: Visiting state H, alpha=-inf, beta=inf
MIN: Terminal state H has utility 10
MAX: From state D, action H → value=10
MIN: Visiting state I, alpha=10, beta=inf
MIN: Terminal state I has utility 9
MAX: From state D, action I → value=9
MAX: Returning value=10 for state D
MIN: From state B, action D → value=10
MAX: Visiting state E, alpha=-inf, beta=10
MIN: Visiting state J, alpha=-inf, beta=10
MIN: Terminal state J has utility 14
MAX: From state E, action J → value=14
MAX: Pruning at state E with value=14 ≥ beta=10
MIN: From state B, action E → value=14
MIN: Returning value=10 for state B
MAX: From state A, action B → value=10
MIN: Visiting state C, alpha=10, beta=inf

MAX: Visiting state F, alpha=10, beta=inf
MIN: Visiting state L, alpha=10, beta=inf
MIN: Terminal state L has utility 5
MAX: From state F, action L → value=5
MIN: Visiting state M, alpha=10, beta=inf
MIN: Terminal state M has utility 4
MAX: From state F, action M → value=4
MAX: Returning value=5 for state F
MIN: From state C, action F → value=5
MIN: Pruning at state C with value=5 ≤ alpha=10
MAX: From state A, action C → value=5
MAX: Returning value=10 for state A

--- Final Results ---
The final utility value is: 10

The final path taken is: ['A', 'B', 'D', 'H']