**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Neelvani Varsha Vittal (1BM23CS412)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

****

**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Sep-2024 to Jan-2025**

**B.M.S. College of Engineering,**

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

****

**CERTIFICATE**

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Shreya Bharamanna Patil (1BM23CS420),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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 |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 24-9-2024  1-10-2024 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent | 1  4 |
| 2 | 8-10-2024  22-10-2024 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 8  10 |
| 3 | 15-10-2024 | Implement A\* search algorithm | 13 |
| 4 | 22-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 21 |
| 5 | 29-10-2024 | Simulated Annealing to Solve 8-Queens problem | 26 |
| 6 | 12-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 28 |
| 7 | 19-11-2024 | Implement unification in first order  logic | 31 |
| 8 | 26-11-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 40 |
| 9 | 26-11-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 43 |
| 10 | 26-11-2024 | Implement Alpha-Beta Pruning. | 46 |

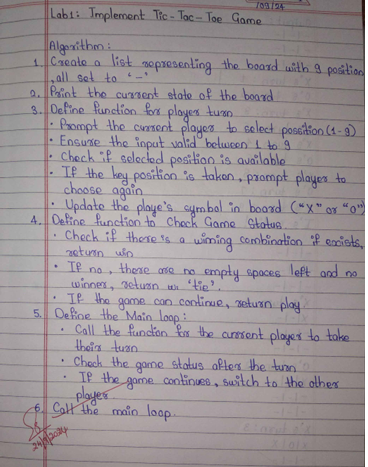
**Github Link:**

<https://github.com/PatilShreya22/AI>

**Program 1**

**Implement Tic –Tac –Toe Game**

**Algorithm:**



**Code:**

# Set up the game board as a 2D list

board = [["-", "-", "-"],

["-", "-", "-"],

["-", "-", "-"]]

# Define a function to print the game board

def print\_board():

for row in board:

print(" | ".join(row))

# Define a function to handle a player's turn

def take\_turn(player):

print(player + "'s turn.")

position = input("Choose a position from 1-9: ")

while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:

position = input("Invalid input. Choose a position from 1-9: ")

position = int(position) - 1

row, col = divmod(position, 3)

while board[row][col] != "-":

position = int(input("Position already taken. Choose a different position: ")) - 1

row, col = divmod(position, 3)

board[row][col] = player

print\_board()

# Define a function to check if the game is over

def check\_game\_over():

# Check for a win

for i in range(3):

if board[i][0] == board[i][1] == board[i][2] != "-":

return "win"

if board[0][i] == board[1][i] == board[2][i] != "-":

return "win"

if board[0][0] == board[1][1] == board[2][2] != "-":

return "win"

if board[0][2] == board[1][1] == board[2][0] != "-":

return "win"

# Check for a tie

elif all(cell != "-" for row in board for cell in row):

return "tie"

# Game is not over

else:

return "play"

# Define the main game loop

def play\_game():

print\_board()

current\_player = "X"

game\_over = False

while not game\_over:

take\_turn(current\_player)

game\_result = check\_game\_over()

if game\_result == "win":

print(current\_player + " wins!")

game\_over = True

elif game\_result == "tie":

print("It's a tie!")

game\_over = True

else:

# Switch to the other player

current\_player = "O" if current\_player == "X" else "X"

# Start the game

play\_game()

**Output:**

- | - | -

- | - | -

- | - | -

X's turn.

Choose a position from 1-9: 1

X | - | -

- | - | -

- | - | -

O's turn.

Choose a position from 1-9: 2

X | O | -

- | - | -

- | - | -

X's turn.

Choose a position from 1-9: 3

X | O | X

- | - | -

- | - | -

O's turn.

Choose a position from 1-9: 5

X | O | X

- | O | -

- | - | -

X's turn.

Choose a position from 1-9: 4

X | O | X

X | O | -

- | - | -

O's turn.

Choose a position from 1-9: 6

X | O | X

X | O | O

- | - | -

X's turn.

Choose a position from 1-9: 8

X | O | X

X | O | O

- | X | -

O's turn.

Choose a position from 1-9: 7

X | O | X

X | O | O

O | X | -

X's turn.

Choose a position from 1-9: 9

X | O | X

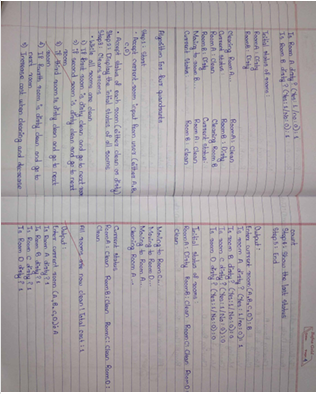
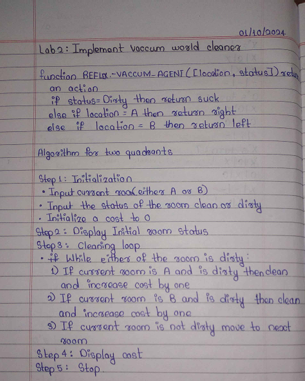
X | O | O

O | X | X

It's a tie!

**Implement vacuum cleaner agent**

**Algorithm:**

****

**Code:**

#For two quadrants

def vacuum\_cleaner\_simulation():

current\_room = input("Enter current room either A or B: ").upper()

room\_A = int(input("Is Room A dirty? (yes:1/no:0): "))

room\_B = int(input("Is Room B dirty? (yes:1/no:0): "))

cost = 0

def display\_rooms():

print(f"Room A: {'Clean' if room\_A == 0 else 'Dirty'}")

print(f"Room B: {'Clean' if room\_B == 0 else 'Dirty'}")

print("\nInitial status of rooms:")

display\_rooms()

print()

while room\_A == 1 or room\_B == 1:

if current\_room == 'A' and room\_A == 1:

print("Cleaning Room A...")

room\_A = 0

cost += 1

elif current\_room == 'B' and room\_B == 1:

print("Cleaning Room B...")

room\_B = 0

cost += 1

else:

current\_room = 'B' if current\_room == 'A' else 'A'

print(f"Moving to Room {current\_room}...")

print("Current status:")

display\_rooms()

print(f"\nBoth rooms are now clean! Total cost: {cost}")

vacuum\_cleaner\_simulation()

#For four quadrants

def vacuum\_cleaner\_simulation():

current\_room = input("Enter current room (A, B, C, or D): ").upper()

room\_A = int(input("Is Room A dirty? (yes:1/no:0): "))

room\_B = int(input("Is Room B dirty? (yes:1/no:0): "))

room\_C = int(input("Is Room C dirty? (yes:1/no:0): "))

room\_D = int(input("Is Room D dirty? (yes:1/no:0): "))

cost = 0

count=2

def display\_rooms():

print(f"Room A: {'Clean' if room\_A == 0 else 'Dirty'}")

print(f"Room B: {'Clean' if room\_B == 0 else 'Dirty'}")

print(f"Room C: {'Clean' if room\_C == 0 else 'Dirty'}")

print(f"Room D: {'Clean' if room\_D == 0 else 'Dirty'}")

print("\nInitial status of rooms:")

display\_rooms()

print()

while room\_A == 1 or room\_B == 1 or room\_C == 1 or room\_D == 1:

if count==0:

print("Vacuum is recharging")

count=2

else:

if current\_room == 'A' and room\_A == 1:

print("Cleaning Room A...")

room\_A = 0

cost += 1

count-=1

elif current\_room == 'B' and room\_B == 1:

print("Cleaning Room B...")

room\_B = 0

cost += 1

count-=1

elif current\_room == 'C' and room\_C == 1:

print("Cleaning Room C...")

room\_C = 0

cost += 1

count-=1

elif current\_room == 'D' and room\_D == 1:

print("Cleaning Room D...")

room\_D = 0

cost += 1

count-=1

else:

if current\_room == 'A':

current\_room = 'B'

elif current\_room == 'B':

current\_room = 'C'

elif current\_room == 'C':

current\_room = 'D'

else:

current\_room = 'A'

print(f"Moving to Room {current\_room}...")

print("\nCurrent status:")

display\_rooms()

print(f"\nAll rooms are now clean! Total cost: {cost}")

vacuum\_cleaner\_simulation()

**Output:**

Enter current room either A or B: A

Is Room A dirty? (yes:1/no:0): 0

Is Room B dirty? (yes:1/no:0): 1

Initial status of rooms:

Room A: Clean

Room B: Dirty

Moving to Room B...

Current status:

Room A: Clean

Room B: Dirty

Cleaning Room B...

Current status:

Room A: Clean

Room B: Clean

Both rooms are now clean! Total cost: 1

Enter current room (A, B, C, or D): A

Is Room A dirty? (yes:1/no:0): 0

Is Room B dirty? (yes:1/no:0): 0

Is Room C dirty? (yes:1/no:0): 1

Is Room D dirty? (yes:1/no:0): 0

Initial status of rooms:

Room A: Clean

Room B: Clean

Room C: Dirty

Room D: Clean

Moving to Room B...

Moving to Room C...

Cleaning Room C...

Current status:

Room A: Clean

Room B: Clean

Room C: Clean

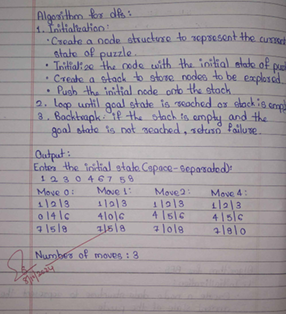
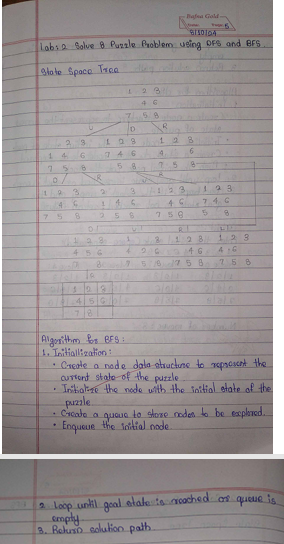
Room D: Clean

All rooms are now clean! Total cost: 1

**Program 2**

**Implement 8 puzzle problems using Depth First Search (DFS)**

**Algorithm:**

** **

**Code:**

from collections import deque

def dfs(start, max\_depth):

stack = deque([(start, [start], 0)]) # (node, path, level)

visited = set([start])

all\_moves = []

while stack:

node, path, level = stack.pop()

all\_moves.append((path, level))

if level < max\_depth:

for next\_node in get\_neighbors(node):

if next\_node not in visited:

visited.add(next\_node)

stack.append((next\_node, path + [next\_node], level + 1))

return all\_moves

def get\_neighbors(node):

neighbors = []

for i in range(9):

if node[i] == 0:

x, y = i // 3, i % 3

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

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

n = list(node)

n[i], n[nx \* 3 + ny] = n[nx \* 3 + ny], n[i]

neighbors.append(tuple(n))

break

return neighbors

def print\_board(board):

board = [board[i:i+3] for i in range(0, 9, 3)]

for row in board:

print(" | ".join(str(x) for x in row))

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

def main():

start = tuple(int(x) for x in input("Enter the initial state (space-separated): ").split())

max\_depth = 10 # maximum depth to search

all\_moves = dfs(start, max\_depth)

if all\_moves:

print("All possible moves:")

for i, (path, level) in enumerate(all\_moves):

print(f"Move {i+1}:")

for j, node in enumerate(path):

print(f"Step {j}:")

print\_board(node)

print()

print(f"Number of moves: {level}")

print()

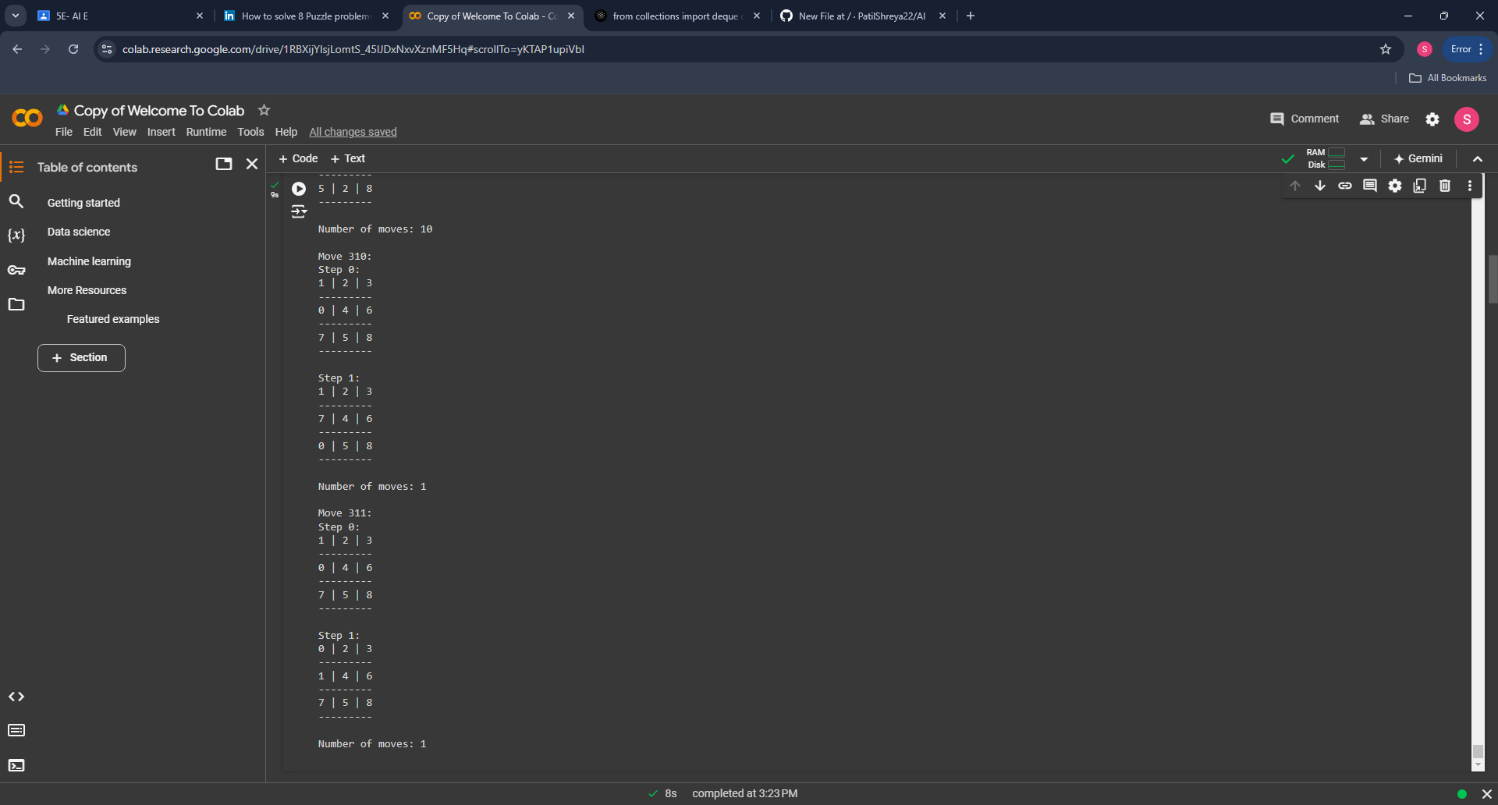
else:

print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_":

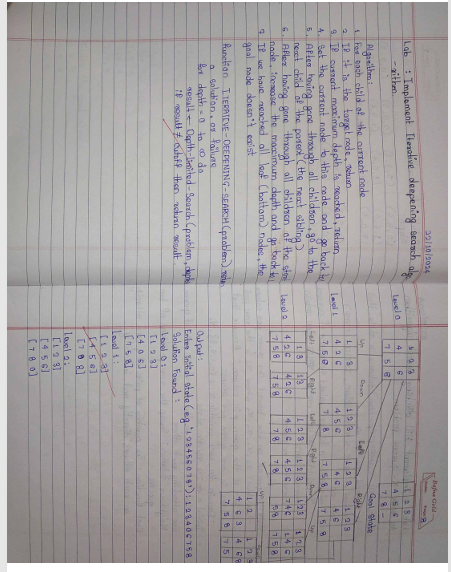
main()

**Output:**



**Implement Iterative deepening search algorithm**

**Algorithm:**

****

**Code:**

from collections import deque

def bfs(start, goal):

queue = deque([(start, [start], 0)]) # (node, path, level)

visited = set([start])

while queue:

node, path, level = queue.popleft()

if node == goal:

return path, level

for next\_node in get\_neighbors(node):

if next\_node not in visited:

visited.add(next\_node)

queue.append((next\_node, path + [next\_node], level + 1))

return None, None

def get\_neighbors(node):

neighbors = []

for i in range(9):

if node[i] == 0:

x, y = i // 3, i % 3

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

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

n = list(node)

n[i], n[nx \* 3 + ny] = n[nx \* 3 + ny], n[i]

neighbors.append(tuple(n))

break

return neighbors

def print\_board(board):

board = [board[i:i+3] for i in range(0, 9, 3)]

for row in board:

print(" | ".join(str(x) for x in row))

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

def main():

start = tuple(int(x) for x in input("Enter the initial state (space-separated): ").split())

goal = (1, 2, 3, 4, 5, 6, 7, 8, 0)

path, level = bfs(start, goal)

if path:

print("Solution found:")

for i, node in enumerate(path):

print(f"Move {i}:")

print\_board(node)

print()

print(f"Number of moves: {level}")

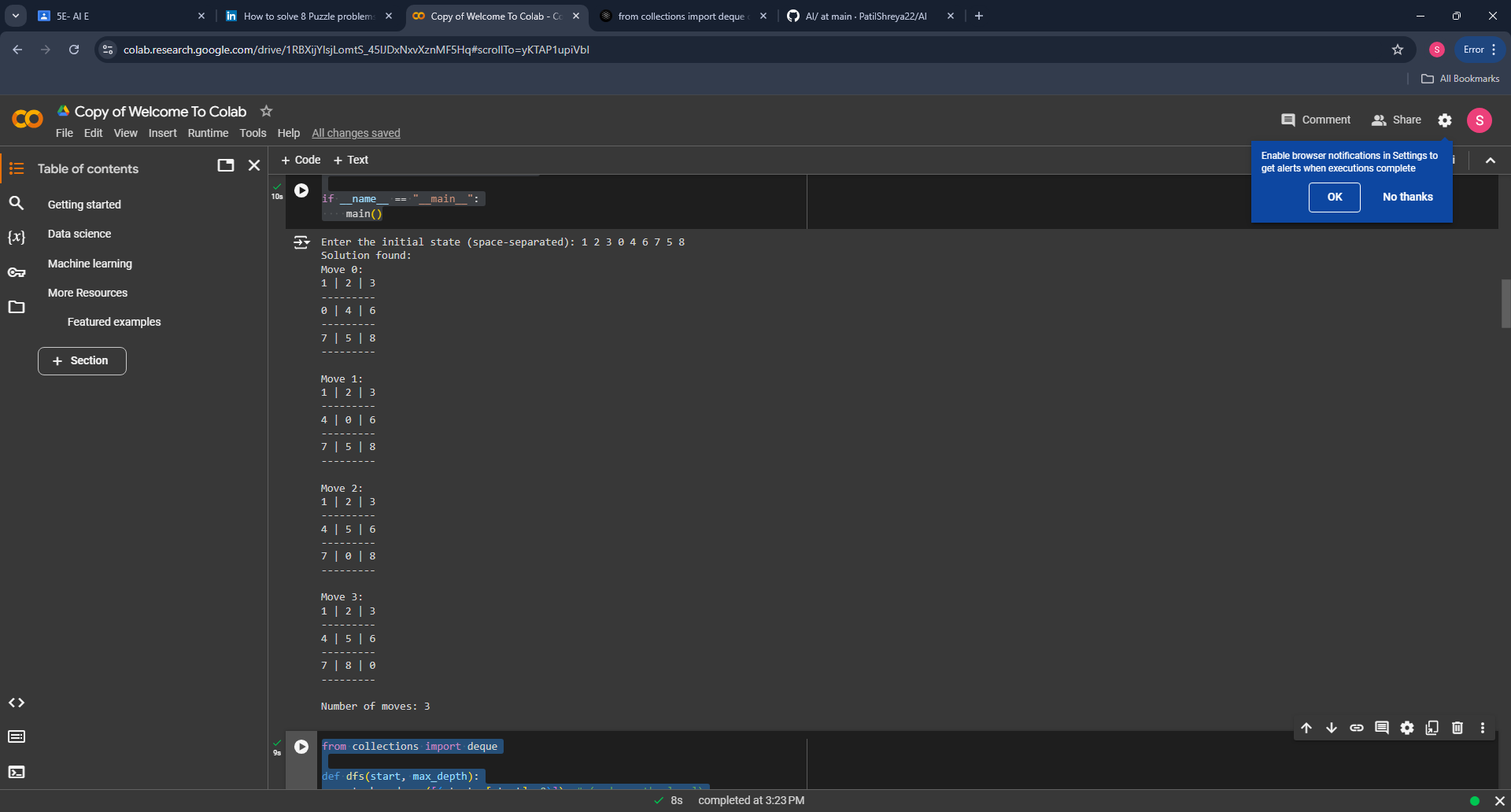
else:

print("No solution found.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

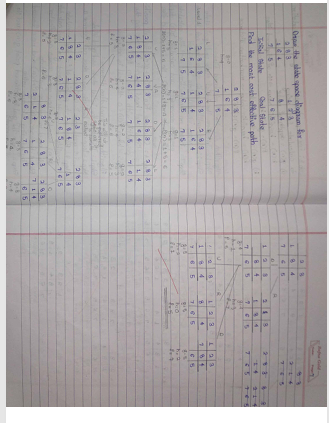
**Output:**

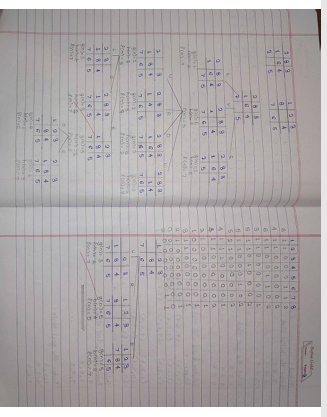


**Program 3**

**Implement A\* search algorithm**

**Algorithm:**

****

****

**Code:**

import heapq

def misplaced\_tile(state, goal\_state):

misplaced = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0 and state[i][j] != goal\_state[i][j]:

misplaced += 1

return misplaced

def find\_blank(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def generate\_neighbors(state):

neighbors = []

x, y = find\_blank(state)

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

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(tuple(tuple(row) for row in new\_state))

return neighbors

def reconstruct\_path(came\_from, current):

path = [current]

while current in came\_from:

current = came\_from[current]

path.append(current)

path.reverse()

return path

def a\_star(start, goal):

open\_list = []

heapq.heappush(open\_list, (0 + misplaced\_tile(start, goal), 0, start))

g\_score = {start: 0}

came\_from = {}

visited = set()

while open\_list:

\_, g, current = heapq.heappop(open\_list)

if current == goal:

path = reconstruct\_path(came\_from, current)

return path, g

visited.add(current)

for neighbor in generate\_neighbors(current):

if neighbor in visited:

continue

tentative\_g = g\_score[current] + 1

if tentative\_g < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g

f\_score = tentative\_g + misplaced\_tile(neighbor, goal) # f(n) = g(n) + h(n)

heapq.heappush(open\_list, (f\_score, tentative\_g, neighbor))

return None, None

def print\_state(state):

for row in state:

print(row)

print()

def get\_state\_from\_user(prompt):

state = []

for i in range(3):

row = input(f"{prompt} row {i+1} (space-separated): ")

state.append(tuple(map(int, row.split())))

return tuple(state)

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter the initial state:")

start\_state = get\_state\_from\_user("Initial state")

print("\nEnter the goal state:")

goal\_state = get\_state\_from\_user("Goal state")

print("\nInitial State:")

print\_state(start\_state)

print("\nGoal State:")

print\_state(goal\_state)

solution, cost = a\_star(start\_state, goal\_state)

if solution:

print(f"\nSolution found with cost: {cost}")

print("Steps:")

for step in solution:

print\_state(step)

else:

print("\nNo solution found.")

**Output:**

Enter the initial state:

Initial state row 1 (space-separated): 2 8 3

Initial state row 2 (space-separated): 1 6 4

Initial state row 3 (space-separated): 7 0 5

Enter the goal state:

Goal state row 1 (space-separated): 1 2 3

Goal state row 2 (space-separated): 8 0 4

Goal state row 3 (space-separated): 7 6 5

Initial State:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Goal State:

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

Solution found with cost: 5

Steps:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

(2, 8, 3)

(1, 0, 4)

(7, 6, 5)

(2, 0, 3)

(1, 8, 4)

(7, 6, 5)

(0, 2, 3)

(1, 8, 4)

(7, 6, 5)

(1, 2, 3)

(0, 8, 4)

(7, 6, 5)

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

import heapq

def manhattan\_distance(state, goal\_state):

distance = 0

for i in range(3):

for j in range(3):

value = state[i][j]

if value != 0:

goal\_i, goal\_j = find\_position(value, goal\_state)

distance += abs(i - goal\_i) + abs(j - goal\_j)

return distance

def find\_position(value, state):

for i in range(3):

for j in range(3):

if state[i][j] == value:

return i, j

def find\_blank(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def generate\_neighbors(state):

neighbors = []

x, y = find\_blank(state)

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

for dx, dy in directions:

nx, ny = x + dx, y + dy

if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(tuple(tuple(row) for row in new\_state))

return neighbors

def reconstruct\_path(came\_from, current):

path = [current]

while current in came\_from:

current = came\_from[current]

path.append(current)

path.reverse()

return path

def a\_star(start, goal):

open\_list = []

heapq.heappush(open\_list, (0 + manhattan\_distance(start, goal), 0, start))

g\_score = {start: 0}

came\_from = {}

visited = set()

while open\_list:

\_, g, current = heapq.heappop(open\_list)

if current == goal:

path = reconstruct\_path(came\_from, current)

return path, g

visited.add(current)

for neighbor in generate\_neighbors(current):

if neighbor in visited:

continue

tentative\_g = g\_score[current] + 1

if tentative\_g < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g

f\_score = tentative\_g + manhattan\_distance(neighbor, goal)

heapq.heappush(open\_list, (f\_score, tentative\_g, neighbor))

return None, None

def print\_state(state):

for row in state:

print(row)

print()

def get\_state\_from\_user(prompt):

state = []

for i in range(3):

row = input(f"{prompt} row {i+1} (space-separated): ")

state.append(tuple(map(int, row.split())))

return tuple(state)

if \_\_name\_\_ == "\_\_main\_\_":

print("Enter the initial state:")

start\_state = get\_state\_from\_user("Initial state")

print("\nEnter the goal state:")

goal\_state = get\_state\_from\_user("Goal state")

print("\nInitial State:")

print\_state(start\_state)

print("\nGoal State:")

print\_state(goal\_state)

solution, cost = a\_star(start\_state, goal\_state)

if solution:

print(f"\nSolution found with cost: {cost}")

print("Steps:")

for step in solution:

print\_state(step)

else:

print("\nNo solution found.")

**Output:**

Enter the initial state:

Initial state row 1 (space-separated): 2 8 3

Initial state row 2 (space-separated): 1 6 4

Initial state row 3 (space-separated): 7 0 5

Enter the goal state:

Goal state row 1 (space-separated): 1 2 3

Goal state row 2 (space-separated): 8 0 4

Goal state row 3 (space-separated): 7 6 5

Initial State:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Goal State:

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

Solution found with cost: 5

Steps:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

(2, 8, 3)

(1, 0, 4)

(7, 6, 5)

(2, 0, 3)

(1, 8, 4)

(7, 6, 5)

(0, 2, 3)

(1, 8, 4)

(7, 6, 5)

(1, 2, 3)

(0, 8, 4)

(7, 6, 5)

(1, 2, 3)

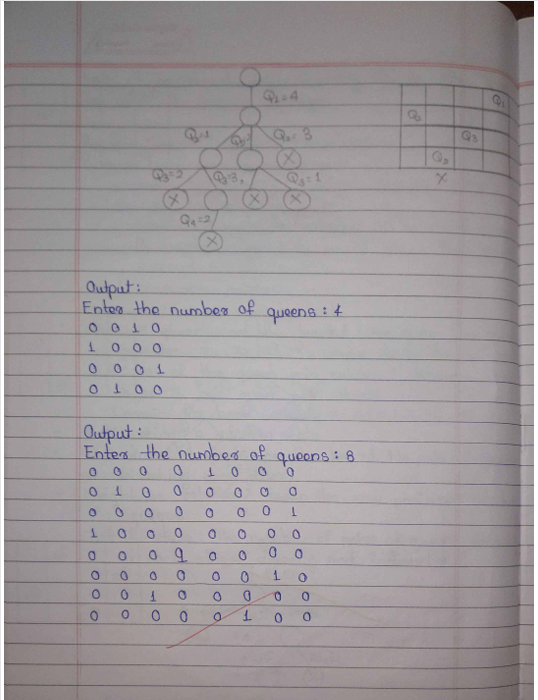
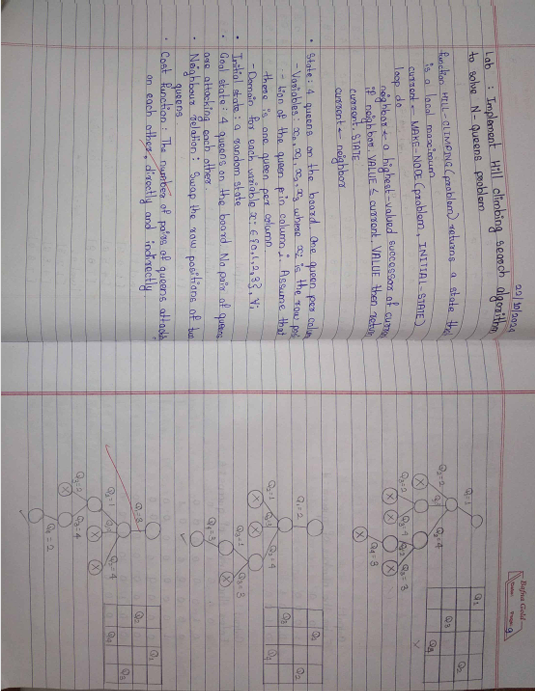
(8, 0, 4)

(7, 6, 5)

**Program 4**

**Implement Hill Climbing search algorithm to solve N-Queens problem**

**Algorithm:**

****

**Code:**

from random import randint

N = int(input("Enter the number of queens:"))

def configureRandomly(board, state):

for i in range(N):

state[i] = randint(0, 100000) % N;

board[state[i]][i] = 1;

def printBoard(board):

for i in range(N):

print(\*board[i])

def printState( state):

print(\*state)

def compareStates(state1, state2):

for i in range(N):

if (state1[i] != state2[i]):

return False;

return True;

def fill(board, value):

for i in range(N):

for j in range(N):

board[i][j] = value;

def calculateObjective( board, state):

attacking = 0;

for i in range(N):

row = state[i]

col = i - 1;

while (col >= 0 and board[row][col] != 1) :

col -= 1

if (col >= 0 and board[row][col] == 1) :

attacking += 1;

row = state[i]

col = i + 1;

while (col < N and board[row][col] != 1):

col += 1;

if (col < N and board[row][col] == 1) :

attacking += 1;

row = state[i] - 1

col = i - 1;

while (col >= 0 and row >= 0 and board[row][col] != 1) :

col-= 1;

row-= 1;

if (col >= 0 and row >= 0 and board[row][col] == 1) :

attacking+= 1;

row = state[i] + 1

col = i + 1;

while (col < N and row < N and board[row][col] != 1) :

col+= 1;

row+= 1;

if (col < N and row < N and board[row][col] == 1) :

attacking += 1;

row = state[i] + 1

col = i - 1;

while (col >= 0 and row < N and board[row][col] != 1) :

col -= 1;

row += 1;

if (col >= 0 and row < N and board[row][col] == 1) :

attacking += 1;

row = state[i] - 1

col = i + 1;

while (col < N and row >= 0 and board[row][col] != 1) :

col += 1;

row -= 1;

if (col < N and row >= 0 and board[row][col] == 1) :

attacking += 1;

return int(attacking / 2);

def generateBoard( board, state):

fill(board, 0);

for i in range(N):

board[state[i]][i] = 1;

def copyState( state1, state2):

for i in range(N):

state1[i] = state2[i];

def getNeighbour(board, state):

opBoard = [[0 for \_ in range(N)] for \_ in range(N)]

opState = [0 for \_ in range(N)]

copyState(opState, state);

generateBoard(opBoard, opState);

opObjective = calculateObjective(opBoard, opState);

NeighbourBoard = [[0 for \_ in range(N)] for \_ in range(N)]

NeighbourState = [0 for \_ in range(N)]

copyState(NeighbourState, state);

generateBoard(NeighbourBoard, NeighbourState);

for i in range(N):

for j in range(N):

if (j != state[i]) :

NeighbourState[i] = j;

NeighbourBoard[NeighbourState[i]][i] = 1;

NeighbourBoard[state[i]][i] = 0;

temp = calculateObjective( NeighbourBoard, NeighbourState);

if (temp <= opObjective) :

opObjective = temp;

copyState(opState, NeighbourState);

generateBoard(opBoard, opState);

NeighbourBoard[NeighbourState[i]][i] = 0;

NeighbourState[i] = state[i];

NeighbourBoard[state[i]][i] = 1;

copyState(state, opState);

fill(board, 0);

generateBoard(board, state);

def hillClimbing(board, state):

neighbourBoard = [[0 for \_ in range(N)] for \_ in range(N)]

neighbourState = [0 for \_ in range(N)]

copyState(neighbourState, state);

generateBoard(neighbourBoard, neighbourState);

while True:

*# Copying the neighbour board and*

*# state to the current board and*

*# state, since a neighbour*

*# becomes current after the jump.*

copyState(state, neighbourState);

generateBoard(board, state);

*# Getting the optimal neighbour*

getNeighbour(neighbourBoard, neighbourState);

if (compareStates(state, neighbourState)) :

printBoard(board);

break;

elif (calculateObjective(board, state) == calculateObjective( neighbourBoard,neighbourState)):

*# Random neighbour*

neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;

generateBoard(neighbourBoard, neighbourState);

*# Driver code*

state = [0] \* N

board = [[0 for \_ in range(N)] for \_ in range(N)]

configureRandomly(board, state);

hillClimbing(board, state);

**Output:**

Enter the number of queens:8

0 0 0 0 1 0 0 0

0 1 0 0 0 0 0 0

0 0 0 0 0 0 0 1

1 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 0 0 1 0

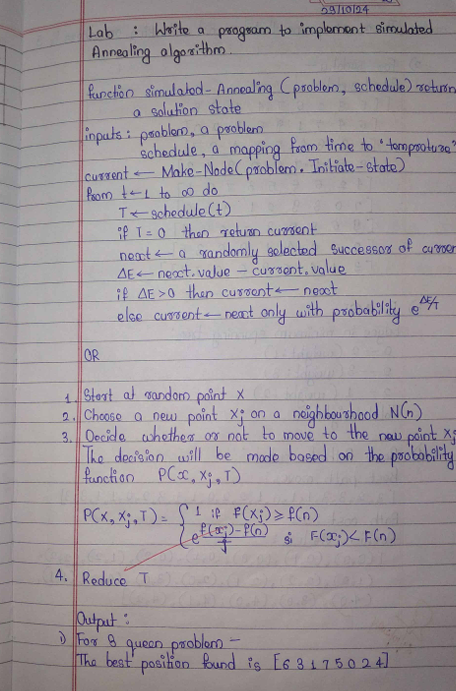
0 0 1 0 0 0 0 0

0 0 0 0 0 1 0 0

**Program 5**

**Simulated Annealing to Solve 8-Queens problem**

**Algorithm:**



**Code:**

import numpy as np

from scipy.optimize import dual\_annealing

def queens\_max(position):

*# This function calculates the number of pairs of queens that are not attacking each other*

position = np.round(position).astype(int) *# Round and convert to integers for queen positions*

n = len(position)

queen\_not\_attacking = 0

for i in range(n - 1):

no\_attack\_on\_j = 0

for j in range(i + 1, n):

*# Check if queens are on the same row or on the same diagonal*

if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):

no\_attack\_on\_j += 1

if no\_attack\_on\_j == n - 1 - i:

queen\_not\_attacking += 1

if queen\_not\_attacking == n - 1:

queen\_not\_attacking += 1

return -queen\_not\_attacking *# Negative because we want to maximize this value*

*# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)*

bounds = [(0, 7) for \_ in range(8)]

*# Use dual\_annealing for simulated annealing optimization*

result = dual\_annealing(queens\_max, bounds)

*# Display the results*

best\_position = np.round(result.x).astype(int)

best\_objective = -result.fun *# Flip sign to get the number of non-attacking queens*

print('The best position found is:', best\_position)

print('The number of queens that are not attacking each other is:', best\_objective)

**Output:**

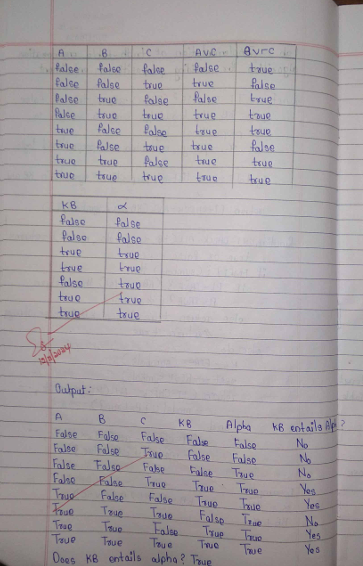
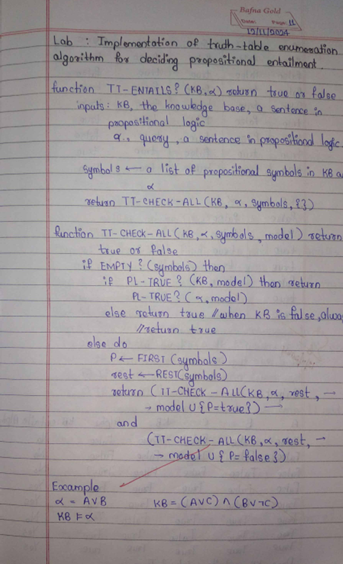
The best position found is: [1 4 6 0 2 7 5 3]

The number of queens that are not attacking each other is: 8

**Program 6**

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

**Algorithm:**

****

**Code:**

import itertools

def evaluate\_formula(formula, valuation):

"""

Evaluate the propositional formula under the given truth assignment (valuation).

The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A', 'B', 'C'.

"""

*# Create a local environment (dictionary) for variable assignments*

env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

*# Replace logical operators with Python equivalents*

formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

*# Replace variables in the formula with their corresponding truth values*

for var in env:

formula = formula.replace(var, str(env[var]))

*# Evaluate the formula and return the result (True or False)*

try:

return eval(formula)

except Exception as e:

raise ValueError(f"Error in evaluating formula: {e}")

def truth\_table(variables):

"""

Generate all possible truth assignments for the given variables.

"""

return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):

"""

Decide if KB entails alpha using a truth-table enumeration algorithm.

KB is a propositional formula (string), and alpha is another propositional formula (string).

"""

*# Generate all possible truth assignments for A, B, and C*

assignments = truth\_table(['A', 'B', 'C'])

print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}") *# Header for the truth table*

print("-" \* 70) *# Separator for readability*

for assignment in assignments:

*# Evaluate KB and alpha under the current assignment*

KB\_value = evaluate\_formula(KB, assignment)

alpha\_value = evaluate\_formula(alpha, assignment)

*# Print the current truth assignment and the results for KB and alpha*

print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB\_value):<15}{str(alpha\_value):<15}{'Yes' if KB\_value and alpha\_value else 'No'}")

*# If KB is true and alpha is false, then KB does not entail alpha*

if KB\_value and not alpha\_value:

return False

*# If no counterexample was found, then KB entails alpha*

return True

*# Define the formulas for KB and alpha*

alpha = 'A OR B'

KB = '(A OR C) AND (B OR NOT C)'

*# Check if KB entails alpha*

result = entails(KB, alpha)

*# Print the final result of entailment*

print(f"\nDoes KB entail alpha? {result}")

**Output:**

A B C KB alpha KB entails alpha?

----------------------------------------------------------------------

False False False False False No

False False True False False No

False True False False True No

False True True True True Yes

True False False True True Yes

True False True False True No

True True False True True Yes

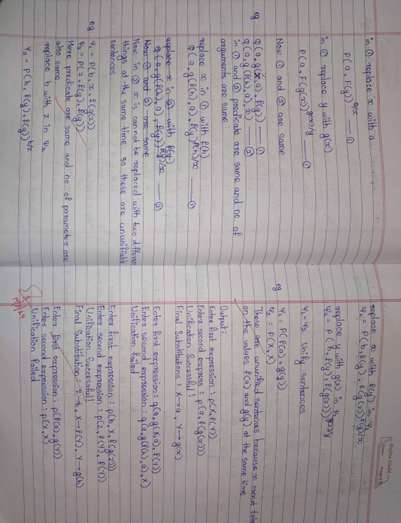
True True True True True Yes

Does KB entail alpha? True

**Program 7**

**Implement unification in first order logic**

**Algorithm:**

** **

**Code:**

class Term:

def \_\_init\_\_(self, symbol, args=None):

self.symbol = symbol

self.args = args if args else []

def \_\_str\_\_(self):

if not self.args:

return str(self.symbol)

return f"{self.symbol}({','.join(str(arg) for arg in self.args)})"

def is\_variable(self):

return isinstance(self.symbol, str) and self.symbol.isupper() and not self.args

def occurs\_check(var, term, substitution):

"""Check if variable occurs in term"""

if term.is\_variable():

if term.symbol in substitution:

return occurs\_check(var, substitution[term.symbol], substitution)

return var.symbol == term.symbol

return any(occurs\_check(var, arg, substitution) for arg in term.args)

def substitute(term, substitution):

"""Apply substitution to term"""

if term.is\_variable() and term.symbol in substitution:

return substitute(substitution[term.symbol], substitution)

if not term.args:

return term

return Term(term.symbol, [substitute(arg, substitution) for arg in term.args])

def unify(term1, term2, substitution=None, iteration=1):

"""Unify two terms with detailed iteration steps"""

if substitution is None:

substitution = {}

print(f"\nIteration {iteration}:")

print(f"Attempting to unify: {term1} and {term2}")

print(f"Current substitution: {', '.join(f'{k}->{v}' for k,v in substitution.items()) or 'empty'}")

term1 = substitute(term1, substitution)

term2 = substitute(term2, substitution)

if term1.symbol == term2.symbol and not term1.args and not term2.args:

print("Terms are identical - no substitution needed")

return substitution

if term1.is\_variable():

if occurs\_check(term1, term2, substitution):

print(f"Occurs check failed: {term1.symbol} occurs in {term2}")

return None

substitution[term1.symbol] = term2

print(f"Added substitution: {term1.symbol} -> {term2}")

return substitution

if term2.is\_variable():

if occurs\_check(term2, term1, substitution):

print(f"Occurs check failed: {term2.symbol} occurs in {term1}")

return None

substitution[term2.symbol] = term1

print(f"Added substitution: {term2.symbol} -> {term1}")

return substitution

if term1.symbol != term2.symbol or len(term1.args) != len(term2.args):

print(f"Unification failed: Different predicates or argument lengths")

return None

for arg1, arg2 in zip(term1.args, term2.args):

result = unify(arg1, arg2, substitution, iteration + 1)

if result is None:

return None

substitution = result

return substitution

def parse\_term(s):

"""Parse terms like P(X,f(Y)) or X"""

s = s.strip()

if '(' not in s:

return Term(s)

pred = s[:s.index('(')]

args\_str = s[s.index('(')+1:s.rindex(')')]

args = []

current = ''

depth = 0

for c in args\_str:

if c == '(' or c == '[':

depth += 1

elif c == ')' or c == ']':

depth -= 1

elif c == ',' and depth == 0:

args.append(parse\_term(current.strip()))

current = ''

continue

current += c

if current:

args.append(parse\_term(current.strip()))

return Term(pred, args)

def print\_examples():

print("\nExample format:")

print("1. Simple terms: P(X,Y)")

print("2. Nested terms: P(f(X),g(Y))")

print("3. Mixed terms: Knows(John,X)")

print("4. Complex nested terms: P(f(g(X)),h(Y,Z))")

print("\nNote: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.")

def validate\_input(expr):

"""Basic validation for input expressions"""

if not expr:

return False

*# Check balanced parentheses*

count = 0

for char in expr:

if char == '(':

count += 1

elif char == ')':

count -= 1

if count < 0:

return False

return count == 0

def main():

while True:

print("\n=== First Order Predicate Logic Unification ===")

print("1. Start Unification")

print("2. Show Examples")

print("3. Exit")

choice = input("\nEnter your choice (1-3): ")

if choice == '1':

print("\nEnter two expressions to unify.")

print\_examples()

while True:

expr1 = input("\nEnter first expression (or 'back' to return): ")

if expr1.lower() == 'back':

break

if not validate\_input(expr1):

print("Invalid expression! Please check the format and try again.")

continue

expr2 = input("Enter second expression: ")

if not validate\_input(expr2):

print("Invalid expression! Please check the format and try again.")

continue

try:

term1 = parse\_term(expr1)

term2 = parse\_term(expr2)

print("\nUnification Process:")

result = unify(term1, term2)

print("\nFinal Result:")

if result is None:

print("Unification failed!")

else:

print("Unification successful!")

print("Final substitutions:", ', '.join(f'{k}->{v}' for k,v in result.items()))

retry = input("\nTry another unification? (y/n): ")

if retry.lower() != 'y':

break

except Exception as e:

print(f"Error processing expressions: {str(e)}")

print("Please check your input format and try again.")

elif choice == '2':

print("\n=== Example Expressions ===")

print("1. P(X,h(Y)) and P(a,f(Z))")

print("2. P(f(a),g(Y)) and P(X,X)")

print("3. Knows(John,X) and Knows(X,Elisabeth)")

print("\nPress Enter to continue...")

input()

elif choice == '3':

print("\nThank you for using the Unification Program!")

break

else:

print("\nInvalid choice! Please enter 1, 2, or 3.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

=== First Order Predicate Logic Unification ===

1. Start Unification

2. Show Examples

3. Exit

Enter your choice (1-3): 1

Enter two expressions to unify.

Example format:

1. Simple terms: P(X,Y)

2. Nested terms: P(f(X),g(Y))

3. Mixed terms: Knows(John,X)

4. Complex nested terms: P(f(g(X)),h(Y,Z))

Note: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.

Enter first expression (or 'back' to return): p(X,f(Y))

Enter second expression: p(a,f(g(x)))

Unification Process:

Iteration 1:

Attempting to unify: p(X,f(Y)) and p(a,f(g(x)))

Current substitution: empty

Iteration 2:

Attempting to unify: X and a

Current substitution: empty

Added substitution: X -> a

Iteration 2:

Attempting to unify: f(Y) and f(g(x))

Current substitution: X->a

Iteration 3:

Attempting to unify: Y and g(x)

Current substitution: X->a

Added substitution: Y -> g(x)

Final Result:

Unification successful!

Final substitutions: X->a, Y->g(x)

Try another unification? (y/n): y

Enter first expression (or 'back' to return): q(a,g(X,a),f(Y))

Enter second expression: q(a,g(f(h),a),X)

Unification Process:

Iteration 1:

Attempting to unify: q(a,g(X,a),f(Y)) and q(a,g(f(h),a),X)

Current substitution: empty

Iteration 2:

Attempting to unify: a and a

Current substitution: empty

Terms are identical - no substitution needed

Iteration 2:

Attempting to unify: g(X,a) and g(f(h),a)

Current substitution: empty

Iteration 3:

Attempting to unify: X and f(h)

Current substitution: empty

Added substitution: X -> f(h)

Iteration 3:

Attempting to unify: a and a

Current substitution: X->f(h)

Terms are identical - no substitution needed

Iteration 2:

Attempting to unify: f(Y) and X

Current substitution: X->f(h)

Iteration 3:

Attempting to unify: Y and h

Current substitution: X->f(h)

Added substitution: Y -> h

Final Result:

Unification successful!

Final substitutions: X->f(h), Y->h

Try another unification? (y/n): y

Enter first expression (or 'back' to return): p(b,X,f(g(Z)))

Enter second expression: p(Z,f(Y),f(Y))

Unification Process:

Iteration 1:

Attempting to unify: p(b,X,f(g(Z))) and p(Z,f(Y),f(Y))

Current substitution: empty

Iteration 2:

Attempting to unify: b and Z

Current substitution: empty

Added substitution: Z -> b

Iteration 2:

Attempting to unify: X and f(Y)

Current substitution: Z->b

Added substitution: X -> f(Y)

Iteration 2:

Attempting to unify: f(g(Z)) and f(Y)

Current substitution: Z->b, X->f(Y)

Iteration 3:

Attempting to unify: g(b) and Y

Current substitution: Z->b, X->f(Y)

Added substitution: Y -> g(b)

Final Result:

Unification successful!

Final substitutions: Z->b, X->f(Y), Y->g(b)

Try another unification? (y/n): y

Enter first expression (or 'back' to return): p(f(a),g(Y))

Enter second expression: p(X,X)

Unification Process:

Iteration 1:

Attempting to unify: p(f(a),g(Y)) and p(X,X)

Current substitution: empty

Iteration 2:

Attempting to unify: f(a) and X

Current substitution: empty

Added substitution: X -> f(a)

Iteration 2:

Attempting to unify: g(Y) and X

Current substitution: X->f(a)

Unification failed: Different predicates or argument lengths

Final Result:

Unification failed!

Try another unification? (y/n): n

=== First Order Predicate Logic Unification ===

1. Start Unification

2. Show Examples

3. Exit

Enter your choice (1-3): 2

=== Example Expressions ===

1. P(X,h(Y)) and P(a,f(Z))

2. P(f(a),g(Y)) and P(X,X)

3. Knows(John,X) and Knows(X,Elisabeth)

Press Enter to continue...

=== First Order Predicate Logic Unification ===

1. Start Unification

2. Show Examples

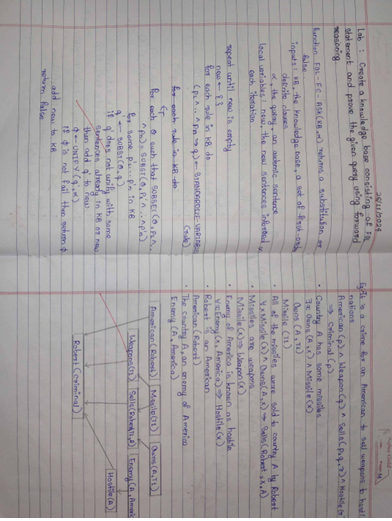
3. Exit

Enter your choice (1-3): 3

**Program 8**

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

**Algorithm:**

****

**Code:**

class ForwardReasoning:

def \_\_init\_\_(self, rules, facts):

"""

Initializes the ForwardReasoning system.

Parameters:

rules (list): List of rules as tuples (condition, result),

where 'condition' is a set of facts.

facts (set): Set of initial known facts.

"""

self.rules = rules *# List of rules (condition -> result)*

self.facts = set(facts) *# Known facts*

def infer(self, query):

"""

Applies forward reasoning to infer new facts based on rules and initial facts.

Parameters:

query (str): The fact to verify if it can be inferred.

Returns:

bool: True if the query can be inferred, False otherwise.

"""

applied\_rules = True

while applied\_rules:

applied\_rules = False

for condition, result in self.rules:

*# Check if all conditions are met in the current facts*

if condition.issubset(self.facts) and result not in self.facts:

self.facts.add(result) *# Add the inferred result*

applied\_rules = True

print(f"Applied rule: {condition} -> {result}")

*# If the query is inferred, return True immediately*

if query in self.facts:

return True

*# Return whether the query can be inferred from the facts*

return query in self.facts

*# Define the Knowledge Base (KB) with rules as (condition, result)*

rules = [

({"American(Robert)", "Missile(m1)", "Owns(CountryA, m1)"}, "Sells(Robert, m1, CountryA)"), *# Sells(Robert, m1, CountryA) based on facts*

({"Sells(Robert, m1, CountryA)", "American(Robert)", "Hostile(CountryA)"}, "Criminal(Robert)"), *# Criminal inference*

]

*# Define initial facts*

facts = {

"American(Robert)",

"Hostile(CountryA)",

"Missile(m1)",

"Owns(CountryA, m1)",

}

*# Query*

query = "Criminal(Robert)"

*# Initialize and run forward reasoning*

reasoner = ForwardReasoning(rules, facts)

result = reasoner.infer(query)

*# Final output*

print("\nFinal facts:")

print(reasoner.facts)

print(f"\nQuery '{query}' inferred: {result}")

Output:

Applied rule: {'Missile(m1)', 'American(Robert)', 'Owns(CountryA, m1)'} -> Sells(Robert, m1, CountryA)

Applied rule: {'American(Robert)', 'Sells(Robert, m1, CountryA)', 'Hostile(CountryA)'} -> Criminal(Robert)

Final facts:

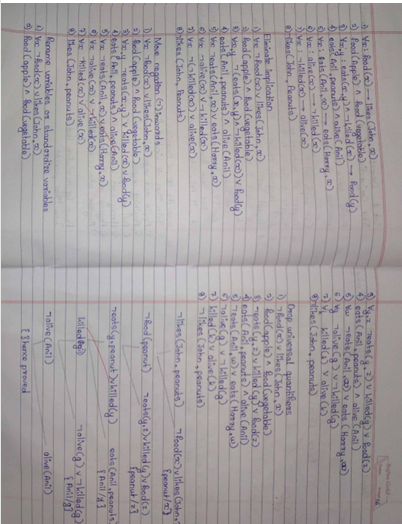
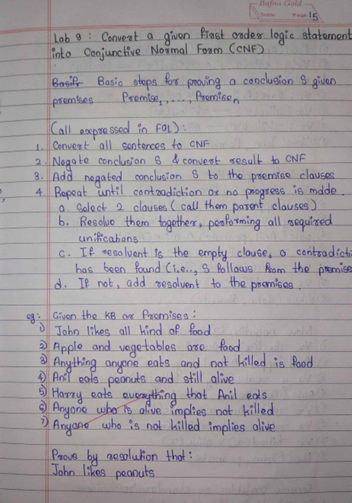
{'Criminal(Robert)', 'Missile(m1)', 'Owns(CountryA, m1)', 'Sells(Robert, m1, CountryA)', 'Hostile(CountryA)', 'American(Robert)'}

Query 'Criminal(Robert)' inferred: True

**Program 9**

**Create a knowledge base consisting of first order logic statements and prove the given query using Resolution**

**Algorithm:**

****

**Code:**

*# Knowledge Base (KB)*

facts = {

"Eats(Anil, Peanuts)": True,

"not Killed(Anil)": True,

"Food(Apple)": True,

"Food(Vegetables)": True,

}

rules = [

*# Rule: Food(X) :- Eats(Y, X) and not Killed(Y)*

{"conditions": ["Eats(Y, X)", "not Killed(Y)"], "conclusion": "Food(X)"},

*# Rule: Likes(John, X) :- Food(X)*

{"conditions": ["Food(X)"], "conclusion": "Likes(John, X)"},

]

*# Query*

query = "Likes(John, Peanuts)"

*# Helper function to substitute variables in a rule*

def substitute(rule\_part, substitutions):

for var, value in substitutions.items():

rule\_part = rule\_part.replace(var, value)

return rule\_part

*# Function to resolve the query*

def resolve\_query(facts, rules, query):

working\_facts = facts.copy()

while True:

new\_facts\_added = False

for rule in rules:

conditions = rule["conditions"]

conclusion = rule["conclusion"]

*# Try all substitutions for variables (X, Y) in the rules*

for entity in ["Apple", "Vegetables", "Peanuts", "Anil", "John"]:

substitutions = {"X": "Peanuts", "Y": "Anil"} *# Fixed for this problem*

resolved\_conditions = [substitute(cond, substitutions) for cond in conditions]

resolved\_conclusion = substitute(conclusion, substitutions)

*# Check if all conditions are true*

if all(working\_facts.get(cond, False) for cond in resolved\_conditions):

if resolved\_conclusion not in working\_facts:

working\_facts[resolved\_conclusion] = True

new\_facts\_added = True

print(f"Derived Fact: {resolved\_conclusion}")

if not new\_facts\_added:

break

*# Check if the query is resolved*

return working\_facts.get(query, False)

*# Run the resolution process*

if resolve\_query(facts, rules, query):

print(f"Proven: {query}")

else:

print(f"Not Proven: {query}")

**Output:**

Derived Fact: Food(Peanuts)

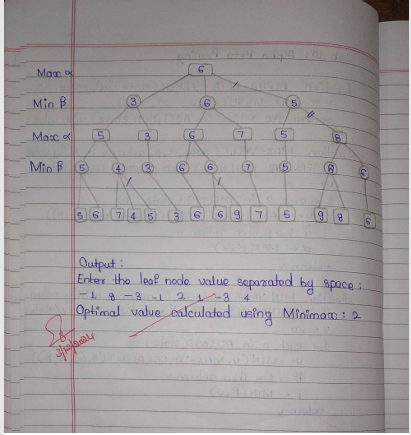
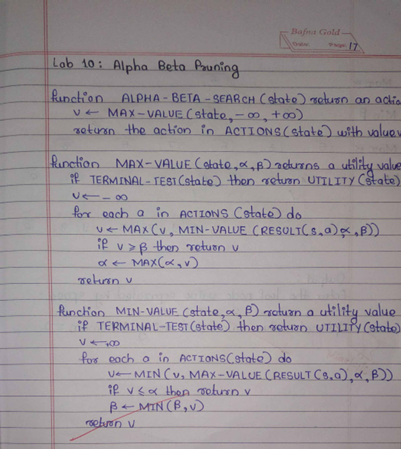
Derived Fact: Likes(John, Peanuts)

Proven: Likes(John, Peanuts)

**Program 10**

**Implement Alpha-Beta Pruning.**

**Algorithm:**

****

**Code:**

import math

def minimax(depth, index, maximizing\_player, values, alpha, beta):

*# Base case: when we've reached the leaf nodes*

if depth == 0:

return values[index]

if maximizing\_player:

max\_eval = float('-inf')

for i in range(2): *# 2 children per node*

eval = minimax(depth - 1, index \* 2 + i, False, values, alpha, beta)

max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval)

if beta <= alpha: *# Beta cutoff*

break

return max\_eval

else:

min\_eval = float('inf')

for i in range(2): *# 2 children per node*

eval = minimax(depth - 1, index \* 2 + i, True, values, alpha, beta)

min\_eval = min(min\_eval, eval)

beta = min(beta, eval)

if beta <= alpha: *# Alpha cutoff*

break

return min\_eval

*# Accept values from the user*

leaf\_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

*# Check if the number of values is a power of 2*

if math.log2(len(leaf\_values)) % 1 != 0:

print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8, 16).")

else:

*# Calculate depth of the tree*

tree\_depth = int(math.log2(len(leaf\_values)))

*# Run Minimax with Alpha-Beta Pruning*

optimal\_value = minimax(depth=tree\_depth, index=0, maximizing\_player=True, values=leaf\_values, alpha=float('-inf'), beta=float('inf'))

print("Optimal value calculated using Minimax:", optimal\_value)

**Output:**

Enter the leaf node values separated by spaces: -1 8 -3 -1 2 1 -3 4

Optimal value calculated using Minimax: 2