**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**Nishanth K S (1BM22CS183)**

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

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

**Prof. Swathi Sridharan**

**Assistant Professor**

**Department of 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 Nishanth K S **(1BM22CS183),** 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.

Dr. Kavitha Sooda

Professor & HOD

Department of CSE, BMSCE

Prof. Swathi Sridharan

Assistant Professor

Department of CSE, BMSCE

**Index**

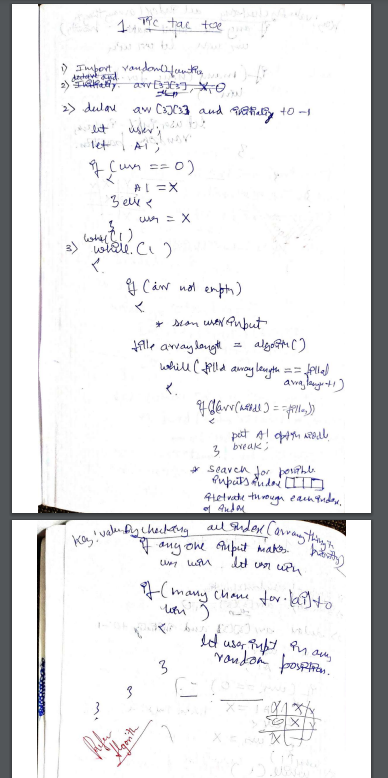
|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 30-9-2024 | Implement Tic –Tac –Toe Game Implement vacuum cleaner agent | 1-6 |
| 2 | 7-10-2024 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 7-12 |
| 3 | 14-10-2024 | Implement A\* search algorithm | 13-17 |
| 4 | 21-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 18-20 |
| 5 | 28-10-2024 | Simulated Annealing to Solve 8-Queens problem | 21-22 |
| 6 | 11-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 23-25 |
| 7 | 2-12-2024 | Implement unification in first order logic | 27-28 |
| 8 | 2-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 29-31 |
| 9 | 16-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 32-34 |
| 10 | 16-12-2024 | Implement Alpha-Beta Pruning. | 35-36 |

Github Link: <https://github.com/nishanthhks/Artificial-Intelligence.git>

# Program 1

Implement Tic –Tac –Toe Game Implement vacuum cleaner agent Tic-Tac-Toe

Algorithm:



Code:

def check\_win(board, r, c):

if board[r - 1][c - 1] == 'X': ch = "O"

else:

ch = "X"

if ch not in board[r - 1] and '-' not in board[r - 1]: return True

elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]):

return True

elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1], board[2][2]):

return True

elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1], board[2][0]):

return True return False

def displayb(board):

print(board[0]) print(board[1]) print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]

displayb(board) xo=1

flag=0

while '-' in board[0] or '-' in board[1] or '-' in board[2]:

if xo==1:

print("enter position to place X:") x=int(input())

y=int(input()) if(x>3 or y>3):

print("invalid position") continue

if(board[x-1][y-1]=='-'): board[x-1][y-1]='X' xo=0

displayb(board) else:

print("invalid position") continue if(check\_win(board,x,y)):

print("X wins") flag=1

break else :

print("enter position to place O:") x=int(input())

y=int(input()) if(x>3 or y>3):

print("invalid position") continue

if(board[x-1][y-1]=='-'): board[x-1][y-1]='O' xo=1

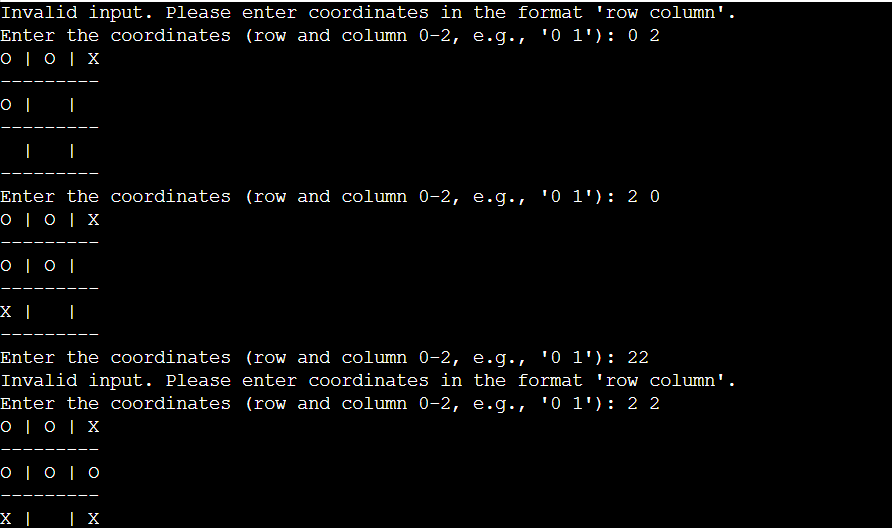
displayb(board) else:

print("invalid position") continue if(check\_win(board,x,y)):

print("0 wins") flag=1

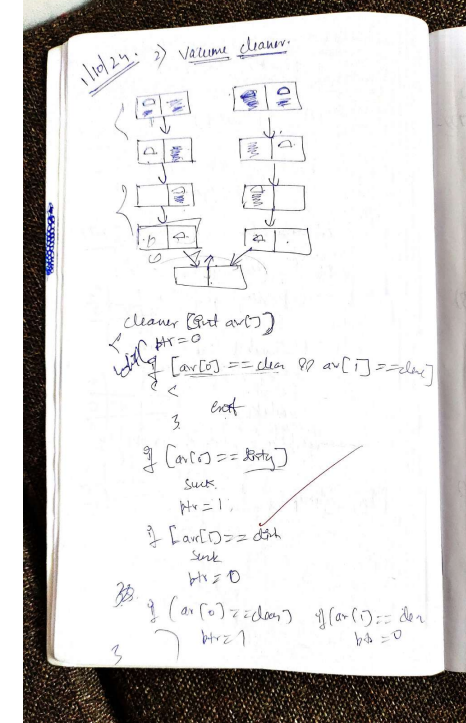
break if flag==0:

print("Draw") print("Game Over")



Vacuum Cleaner

Algorithm:



Code:

*class* VacuumCleanerAgent:

*def* \_\_init\_\_(*self*, *environment*):

*self*.environment = *environment*

*self*.cleaned\_cells = 0

*self*.position = (0, 0)

*def* clean(*self*):

*while* True:

            x, y = *self*.position

*# Clean the current position if dirty*

*if* *self*.environment[x][y] == 'D':

*self*.environment[x][y] = 'C'  *# Clean the cell*

*self*.cleaned\_cells += 1

                print(*f*"Cleaned position {*self*.position}")

*# Find next dirty cell*

            next\_position = *self*.find\_next\_dirty()

*if* next\_position:

                print(*f*"Moving to next dirty position {next\_position}")

*self*.position = next\_position

*else*:

                print("No more dirty cells found. Cleaning complete.")

*break*

*def* find\_next\_dirty(*self*):

*# Check all positions in the environment for dirty cells*

*for* i *in* range(len(*self*.environment)):

*for* j *in* range(len(*self*.environment[i])):

*if* *self*.environment[i][j] == 'D':

*return* (i, j)  *# Return the first found dirty cell*

*return* None  *# No dirty cells found*

*def* display\_environment(*self*):

*for* row *in* *self*.environment:

            print(" ".join(row))

        print(*f*"Total cleaned cells: {*self*.cleaned\_cells}")

*# Example usage:*

initial\_environment = [

    ['D', 'C', 'D', 'D'],

    ['C', 'D', 'C', 'C'],

    ['D', 'C', 'D', 'C'],

    ['C', 'C', 'C', 'D']

]

agent = VacuumCleanerAgent(initial\_environment)

print("Initial environment:")

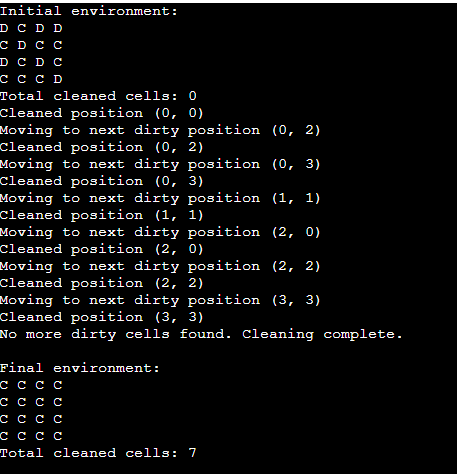
agent.display\_environment()

agent.clean()

print("\nFinal environment:")

agent.display\_environment()

Output:

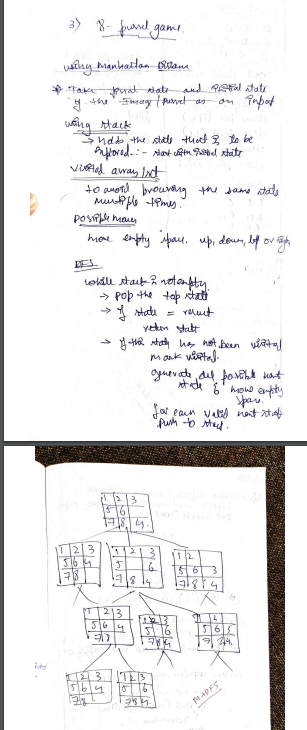


# Program 2

Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm

8 puzzle using DFS

Algorithm:



Code:

*import* heapq

*import* numpy *as* np

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

vis = set()

q = []

path = []

*def* manhattan(*curr*):

    ans = 0

*for* i *in* range(3):

*for* j *in* range(3):

*for* k *in* range(3):

*for* l *in* range(3):

*if* goal[i][j] == *curr*[k][l]:

                        ans += abs(i - k) + abs(l - j)

*return* ans

*def* moves(*curr*):

    x = 0

    y = 0

*for* i *in* range(3):

*for* j *in* range(3):

*if* *curr*[i][j] == 0:

                x = i

                y = j

*break*

    poss = [[0, -1], [-1, 0], [1, 0], [0, 1]]

    x1 = x

    y1 = y

*for* pos *in* poss:

        x += pos[0]

        y += pos[1]

*if* 0 <= x < 3 and 0 <= y < 3:

            curr1 = [row.copy() *for* row *in* *curr*]

            curr1[x1][y1], curr1[x][y] = curr1[x][y], curr1[x1][y1]

            tuple\_curr1 = tuple(map(tuple, curr1))

*if* tuple\_curr1 not in vis:

                heapq.heappush(q, (manhattan(curr1), curr1))

                vis.add(tuple\_curr1)

        x = x1

        y = y1

*def* dfs(*curr*):

*if* *curr* == goal:

        path.append(*curr*)

*return* True

    moves(*curr*)

*if* q:

*curr* = heapq.heappop(q)[1]

*if* dfs(*curr*):

            path.append(*curr*)

*return* True

*return* False

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

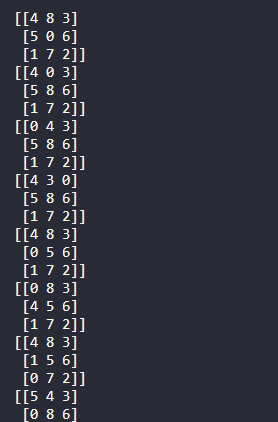
dfs(c)

print(np.array(c))

*for* state *in* reversed(path):

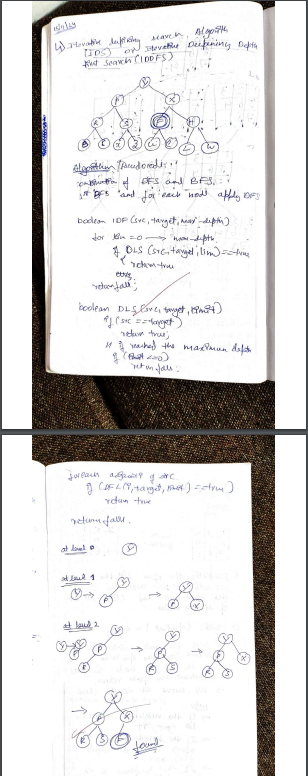
    print(np.array(state))

Output:



Implement Iterative deepening search algorithm

Algorithm:



Code:

*# Define a simple Node class for the tree structure*

*class* Node:

*def* \_\_init\_\_(*self*, *value*):

*self*.value = *value*

*self*.children = []

*# Iterative Deepening Depth-First Search (IDDFS)*

*def* iddfs(*root*, *goal*):

    depth = 0

*while* True:

        print(*f*"Searching at depth {depth}...")

        result = dls(*root*, *goal*, depth)

*if* result == "found":

*return* "Goal State Found"

        depth += 1

*# Depth-Limited Search (DLS)*

*def* dls(*node*, *goal*, *depth*):

*if* *depth* == 0:

*if* *node*.value == *goal*:

*return* "found"

*else*:

*return* "not found"

*elif* *depth* > 0:

*for* child *in* *node*.children:

            result = dls(child, *goal*, *depth* - 1)

*if* result == "found":

*return* "found"

*return* "not found"

*# Build the tree (graph)*

root = Node('Y')

p = Node('P')

X = Node('X')

r = Node('R')

S = Node('S')

F = Node('F')

H = Node('H')

B = Node('B')

C = Node('C')

Z = Node('Z')

U = Node('U')

E = Node('E')

L = Node('L')

W = Node('W')

x = Node('x')

*# Connect the nodes (edges)*

root.children = [p, X]

p.children = [r, S]

X.children = [F, H]

r.children = [B, C]

S.children = [x, Z]

p.children = [r, S]

X.children = [F, H]

F.children = [U, E]

H.children = [L, W]

*# Define the goal state*

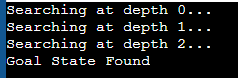
goal = 'F'

*# Run IDDFS*

result = iddfs(root, goal)

print(result)

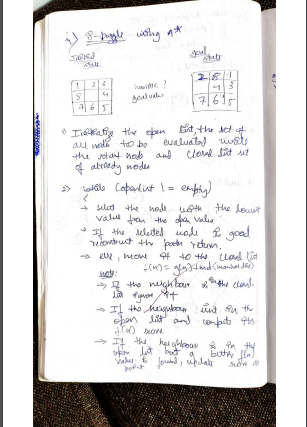
Output:

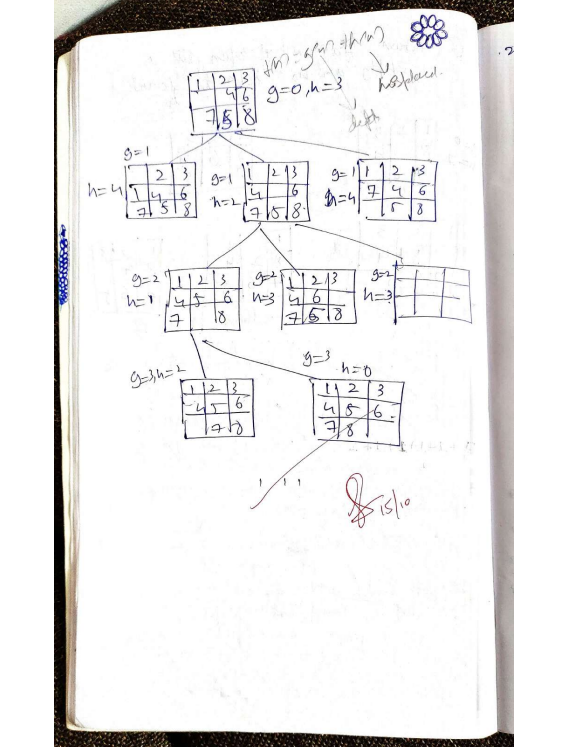


# Program 3

Implement A\* search algorithm

Algorithm:





Code:

*import* heapq

*from* copy *import* deepcopy

*# Define the goal state*

goal\_state = [

    [2, 8, 1],

    [0, 4, 3],

    [7, 6, 5]

]

*# Function to find the position of the blank tile (0)*

*def* find\_blank\_tile(*state*):

*for* i *in* range(3):

*for* j *in* range(3):

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

*return* i, j

*return* None

*# Function to calculate the Manhattan distance*

*def* manhattan\_distance(*state*):

    distance = 0

*for* i *in* range(3):

*for* j *in* range(3):

            tile = *state*[i][j]

*if* tile != 0:  *# Ignore the blank tile*

                goal\_x, goal\_y = divmod(tile - 1, 3)

                distance += abs(goal\_x - i) + abs(goal\_y - j)

*return* distance

*# Function to make a move by sliding the blank tile (0)*

*def* make\_move(*state*, *move*):

    new\_state = deepcopy(*state*)

    blank\_x, blank\_y = find\_blank\_tile(*state*)

*if* *move* == "up" and blank\_x > 0:

        new\_state[blank\_x][blank\_y], new\_state[blank\_x - 1][blank\_y] = new\_state[blank\_x - 1][blank\_y], new\_state[blank\_x][blank\_y]

*elif* *move* == "down" and blank\_x < 2:

        new\_state[blank\_x][blank\_y], new\_state[blank\_x + 1][blank\_y] = new\_state[blank\_x + 1][blank\_y], new\_state[blank\_x][blank\_y]

*elif* *move* == "left" and blank\_y > 0:

        new\_state[blank\_x][blank\_y], new\_state[blank\_x][blank\_y - 1] = new\_state[blank\_x][blank\_y - 1], new\_state[blank\_x][blank\_y]

*elif* *move* == "right" and blank\_y < 2:

        new\_state[blank\_x][blank\_y], new\_state[blank\_x][blank\_y + 1] = new\_state[blank\_x][blank\_y + 1], new\_state[blank\_x][blank\_y]

*return* new\_state

*# Function to get valid moves for the blank tile (0)*

*def* get\_valid\_moves(*state*):

    blank\_x, blank\_y = find\_blank\_tile(*state*)

    moves = []

*if* blank\_x > 0:

        moves.append("up")

*if* blank\_x < 2:

        moves.append("down")

*if* blank\_y > 0:

        moves.append("left")

*if* blank\_y < 2:

        moves.append("right")

*return* moves

*def* astar\_solve\_puzzle(*initial\_state*):

    """Solves the 8-puzzle using A\* search with Manhattan distance heuristic."""

    open\_list = [(manhattan\_distance(*initial\_state*), *initial\_state*, [])]  *# (f, state, path)*

    closed\_list = set()

*while* open\_list:

*# Extracting last tuple and destructuring into three variables*

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

*# tuple of tuples*

        state\_tuple = tuple(map(tuple, current\_state))

*# checking*

*if* current\_state == goal\_state:

*return* path

*# set of tuples*

*if* state\_tuple in closed\_list:

*continue*

        closed\_list.add(state\_tuple)

*for* move *in* get\_valid\_moves(current\_state):

*# making moves to all possible places*

            new\_state = make\_move(current\_state, move)

*# appending to new path to list*

            new\_path = path + [move]

*# calculating new distance*

            f\_value = manhattan\_distance(new\_state) + len(new\_path)  *# f = g + h*

*# by doing this we get min distance state,  which will be used for next state.*

            heapq.heappush(open\_list, (f\_value, new\_state, new\_path))

*return* None  *# No solution found*

*# Main function*

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

*# ... (your initial state definition) ...*

*# Define the initial state*

    initial\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]

*# Run A\* search using Manhattan distance*

    solution\_moves = astar\_solve\_puzzle(initial\_state)

*if* solution\_moves:

        print("Solution found!")

        print(*f*"Moves: {' -> '.join(solution\_moves)}")

*else*:

*# if len(solution\_moves) == 0:*

*#   print("ALready Won!")*

*# else:*

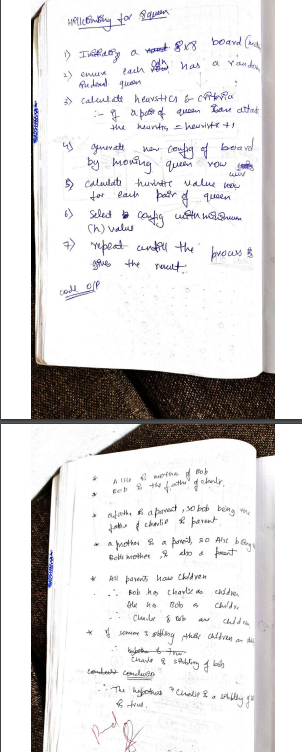
          print("No solution exists for this puzzle.")

Output:



# Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Code:

*import* random

*def* h(*s*):

    h = 0

    n = len(*s*)

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

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

*if* *s*[i] == *s*[j] or abs(*s*[i] - *s*[j]) == abs(i - j):

                h += 1

*return* h

*def* new(*s*):

    best=*s*

*for* i *in* range(len(*s*)):

*for* j *in* range(1,9):

*if* j!=*s*[i]:

                n=*s*[:i]+[j]+*s*[i+1:]

*if* h(n)<h(best):

                    best=n

*return* best

*def* hc():

    curr=[random.randint(1,8) *for* i *in* range(8)]

*while* True:

        ch=h(curr)

        curr=new(curr)

*if* h(curr)==0:

*return* curr

*if* h(curr)>=ch:

            curr=[random.randint(1,8) *for* i *in* range(8)]

*def* print\_board(*solution*):

    print("Solution for 8 Queens Hill climbing is: ",*solution*)

*if* *solution* is None:

        print("No solution found.")

*return*

    board = [['.' *for* \_ *in* range(8)] *for* \_ *in* range(8)]

*for* row *in* range(len(*solution*)):

        col = *solution*[row] - 1

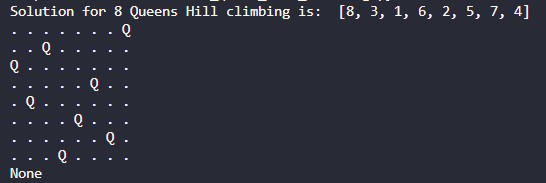
        board[row][col] = 'Q'

*for* row *in* board:

        print(' '.join(row))

print(print\_board(hc()))

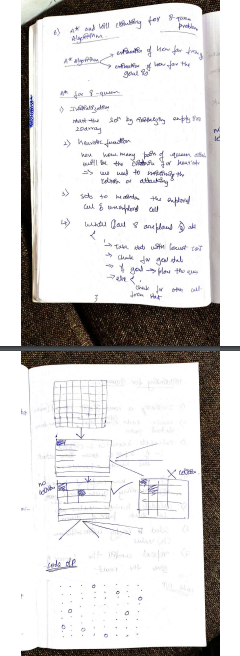
Output:



# 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, 8) 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

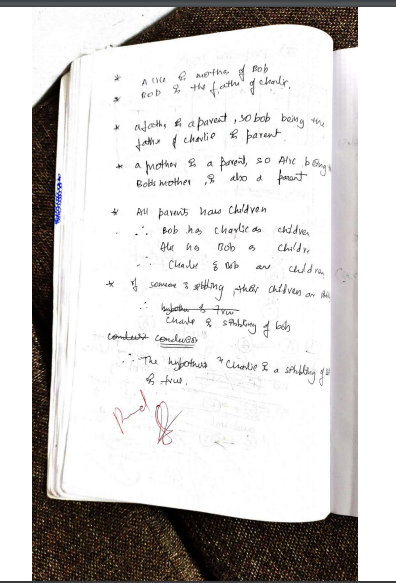
print('The best position found is:', best\_position)

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

# Program 6

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

Algorithm:



Code:

*# Define the knowledge base (KB)*

KB = {

*# Rules and facts*

    "philosopher(X)": "human(X)",  *# Rule 1: All philosophers are humans*

    "human(Socrates)": True,  *# Socrates is human (deduced from philosopher)*

    "teachesAtUniversity(X)": "philosopher(X) or scientist(X)",  *# Rule 2*

    "some(philosopher, not scientist)": True,  *# Rule 3: Some philosophers are not scientists*

    "writesBooks(X)": "teachesAtUniversity(X) and philosopher(X)",  *# Rule 4*

    "philosopher(Socrates)": True,  *# Fact: Socrates is a philosopher*

    "teachesAtUniversity(Socrates)": True,  *# Fact: Socrates teaches at university*

}

*# Function to evaluate a predicate based on the KB*

*def* resolve(*predicate*):

*# If it's a direct fact in KB*

*if* *predicate* in KB and isinstance(KB[*predicate*], bool):

*return* KB[*predicate*]

*# If it's a derived rule*

*if* *predicate* in KB:

        rule = KB[*predicate*]

*if* " and " in rule:  *# Handle conjunction*

            sub\_preds = rule.split(" and ")

*return* all(resolve(sub.strip()) *for* sub *in* sub\_preds)

*elif* " or " in rule:  *# Handle disjunction*

            sub\_preds = rule.split(" or ")

*return* any(resolve(sub.strip()) *for* sub *in* sub\_preds)

*elif* "not " in rule:  *# Handle negation*

            sub\_pred = rule[4:]  *# Remove "not "*

*return* not resolve(sub\_pred.strip())

*else*:  *# Handle single predicate*

*return* resolve(rule.strip())

*# If the predicate contains variables*

*if* "(" in *predicate*:

        func, args = *predicate*.split("(")

        args = args.strip(")").split(", ")

*# Handle philosopher and human link*

*if* func == "philosopher":

*return* resolve(*f*"human({args[0]})")

*# Handle writesBooks rule explicitly*

*if* func == "writesBooks":

*return* resolve(*f*"teachesAtUniversity({args[0]})") and resolve(*f*"philosopher({args[0]})")

*# Default to False if no rule or fact applies*

*return* False

*# Query to check if Socrates writes books*

query = "writesBooks(Socrates)"

result = resolve(query)

*# Print the result*

print(*f*"Does Socrates write books? {'Yes' *if* result *else* 'No'}")

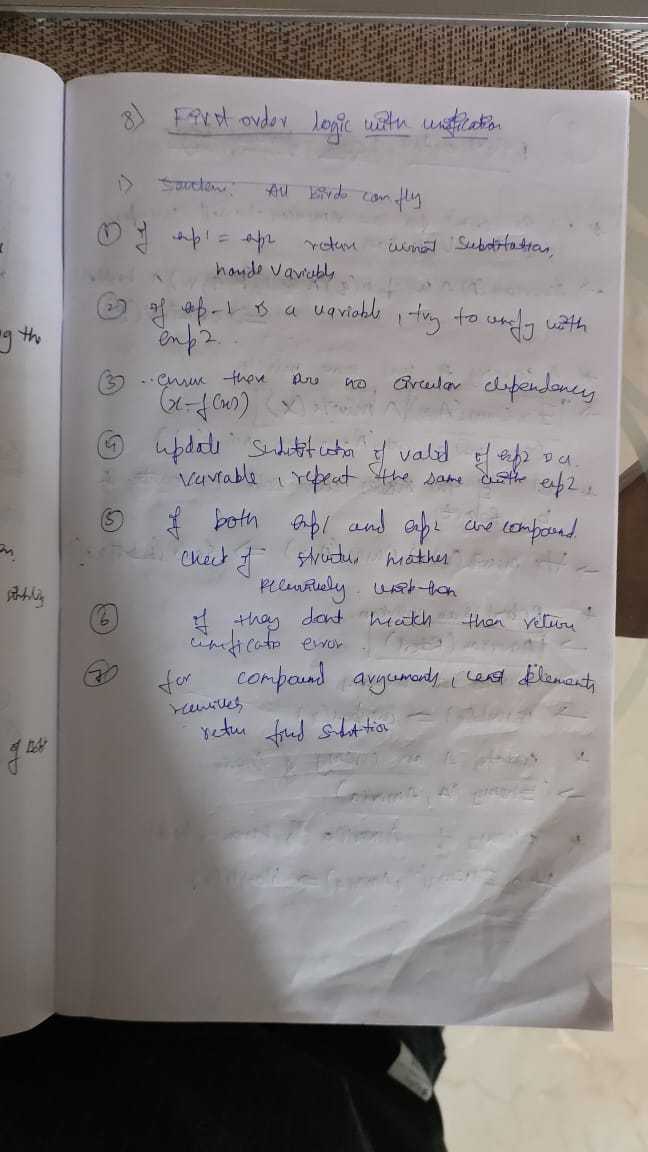
)



# Program 7

Implement unification in first order logic

Algorithm:



Code:

*import* re

*def* occurs\_check(*var*, *x*):

*if* *var* == *x*:

*return* True

*elif* isinstance(*x*, list):

*return* any(occurs\_check(*var*, xi) *for* xi *in* *x*)

*return* False

*def* unify\_var(*var*, *x*, *subst*):

*if* *var* in *subst*:

*return* unify(*subst*[*var*], *x*, *subst*)

*elif* isinstance(*x*, (list, tuple)) and tuple(*x*) in *subst*:

*return* unify(*var*, *subst*[tuple(*x*)], *subst*)

*elif* occurs\_check(*var*, *x*):

*return* "FAILURE"

*else*:

*subst*[*var*] = tuple(*x*) *if* isinstance(*x*, list) *else* *x*

*return* *subst*

*def* unify(*x*, *y*, *subst*=None):

*if* *subst* is None:

*subst* = {}

*if* *x* == *y*:

*return* *subst*

*elif* isinstance(*x*, str) and *x*.islower():

*return* unify\_var(*x*, *y*, *subst*)

*elif* isinstance(*y*, str) and *y*.islower():

*return* unify\_var(*y*, *x*, *subst*)

*elif* isinstance(*x*, list) and isinstance(*y*, list):

*if* len(*x*) != len(*y*):

*return* "FAILURE"

*if* *x*[0] != *y*[0]:

*return* "FAILURE"

*for* xi, yi *in* zip(*x*[1:], *y*[1:]):

*subst* = unify(xi, yi, *subst*)

*if* *subst* == "FAILURE":

*return* "FAILURE"

*return* *subst*

*else*:

*return* "FAILURE"

*def* unify\_and\_check(*expr1*, *expr2*):

    result = unify(*expr1*, *expr2*)

*if* result == "FAILURE":

*return* False, None

*return* True, result

*def* display\_result(*expr1*, *expr2*, *is\_unified*, *subst*):

    print("Expression 1:", *expr1*)

    print("Expression 2:", *expr2*)

*if* not *is\_unified*:

        print("Result: Unification Failed")

*else*:

        print("Result: Unification Successful")

        print("Substitutions:", {k: list(v) *if* isinstance(v, tuple) *else* v *for* k, v *in* *subst*.items()})

*def* parse\_input(*input\_str*):

*input\_str* = *input\_str*.replace(" ", "")

*def* parse\_term(*term*):

*if* '(' in *term*:

            match = re.match(*r*'*(*[a-zA-Z0-9\_]+*)(.*\**)*', *term*)

*if* match:

                predicate = match.group(1)

                arguments\_str = match.group(2)

                arguments = [parse\_term(arg.strip()) *for* arg *in* arguments\_str.split(',')]

*return* [predicate] + arguments

*return* *term*

*return* parse\_term(*input\_str*)

*def* main():

*while* True:

        expr1\_input = input("Enter the first expression (e.g., p(x, f(y))): ")

        expr2\_input = input("Enter the second expression (e.g., p(a, f(z))): ")

        expr1 = parse\_input(expr1\_input)

        expr2 = parse\_input(expr2\_input)

        is\_unified, result = unify\_and\_check(expr1, expr2)

        display\_result(expr1, expr2, is\_unified, result)

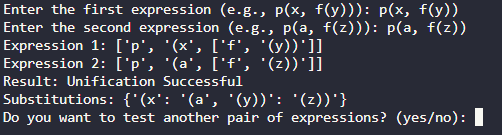
        another\_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()

*if* another\_test != 'yes':

*break*

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

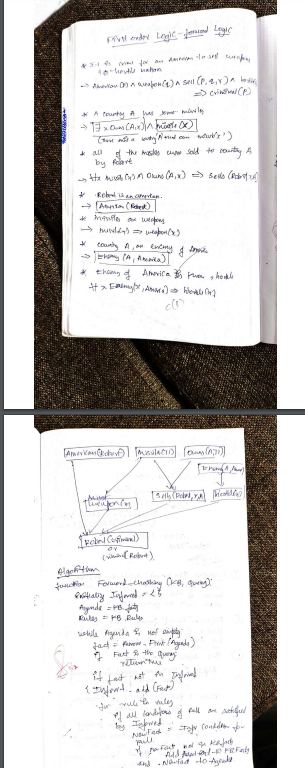
    main()



# Program 8

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

Algorithm:



Code:

*# Define initial facts and rules*

facts = {"InAmerica(West)", "SoldWeapons(West, Nono)", "Enemy(Nono, America)"}

rules = [

    {

        "conditions": ["InAmerica(x)", "SoldWeapons(x, y)", "Enemy(y, America)"],

        "conclusion": "Criminal(x)",

    },

    {

        "conditions": ["Enemy(y, America)"],

        "conclusion": "Dangerous(y)",

    },

]

*# Forward chaining function*

*def* forward\_chaining(*facts*, *rules*):

    derived\_facts = set(*facts*)  *# Initialize derived facts*

*while* True:

        new\_fact\_found = False

*for* rule *in* *rules*:

*# Substitute variables and check if conditions are met*

*for* fact *in* derived\_facts:

*if* "x" in rule["conditions"][0]:

*# Substitute variables (x, y) with specific instances*

*for* condition *in* rule["conditions"]:

*if* "x" in condition or "y" in condition:

                            x = "West"  *# Hardcoded substitution for simplicity*

                            y = "Nono"

                            conditions = [

                                cond.replace("x", x).replace("y", y)

*for* cond *in* rule["conditions"]

                            ]

                            conclusion = (

                                rule["conclusion"].replace("x", x).replace("y", y)

                            )

*# Check if all conditions are satisfied*

*if* all(cond *in* derived\_facts *for* cond *in* conditions) and conclusion not in derived\_facts:

                                derived\_facts.add(conclusion)

                                print(*f*"New fact derived: {conclusion}")

                                new\_fact\_found = True

*# Exit loop if no new fact is found*

*if* not new\_fact\_found:

*break*

*return* derived\_facts

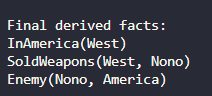
*# Run forward chaining*

final\_facts = forward\_chaining(facts, rules)

print("\nFinal derived facts:")

*for* fact *in* final\_facts:

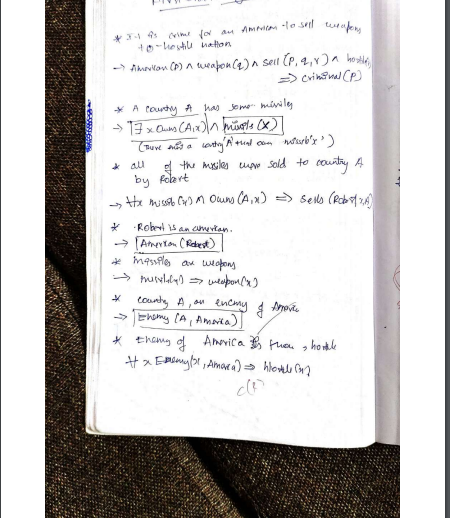
    print(fact)



# Program 9

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

Algorithm:



Code:

# Define the knowledge base (KB) KB = {

"food(Apple)": True, "food(vegetables)": True, "eats(Anil, Peanuts)": True, "alive(Anil)": True,

"likes(John, X)": "food(X)", # Rule: John likes all food

"food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats

"alive(X)": "not killed(X)", # Rule: Alive implies not killed "not killed(X)": "alive(X)", # Rule: Not killed implies alive

}

# Function to evaluate if a predicate is true based on the KB def resolve(predicate):

# If it's a direct fact in KB

if predicate in KB and isinstance(KB[predicate], bool): return KB[predicate]

# If it's a derived rule if predicate in KB:

rule = KB[predicate]

if " and " in rule: # Handle conjunction sub\_preds = rule.split(" and ")

return all(resolve(sub.strip()) for sub in sub\_preds) elif " or " in rule: # Handle disjunction

sub\_preds = rule.split(" or ")

return any(resolve(sub.strip()) for sub in sub\_preds) elif "not " in rule: # Handle negation

sub\_pred = rule[4:] # Remove "not " return not resolve(sub\_pred.strip())

else: # Handle single predicate return resolve(rule.strip())

# If the predicate is a specific query (e.g., likes(John, Peanuts)) if "(" in predicate:

func, args = predicate.split("(") args = args.strip(")").split(", ")

if func == "food" and args[0] == "Peanuts":

return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)") if func == "likes" and args[0] == "John" and args[1] == "Peanuts":

return resolve("food(Peanuts)")

# Default to False if no rule or fact applies return False

# Query to prove: John likes Peanuts query = "likes(John, Peanuts)"

result = resolve(query)

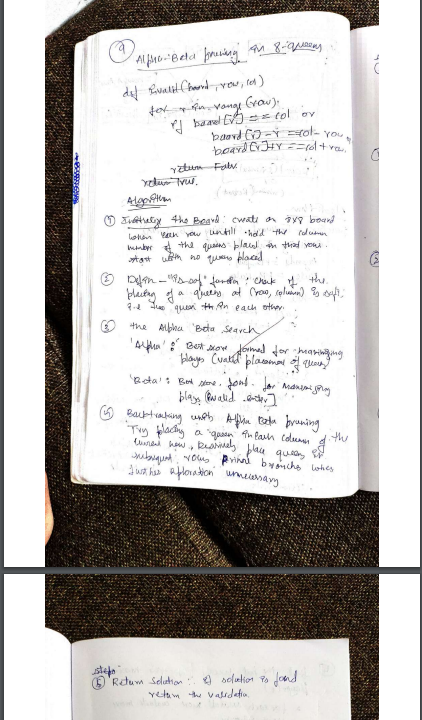
# Print the result

print(f"Does John like peanuts? {'Yes' if result else 'No'}")

# Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Code:

*def* is\_valid(*board*, *row*, *col*):

*for* i *in* range(*row*):

*if* *board*[i] == *col* or \

           abs(*board*[i] - *col*) == abs(i - *row*):

*return* False

*return* True

*def* alpha\_beta(*board*, *row*, *alpha*, *beta*, *isMaximizing*):

*if* *row* == len(*board*):

*return* 1

*if* *isMaximizing*:

        max\_score = 0

*for* col *in* range(len(*board*)):

*if* is\_valid(*board*, *row*, col):

*board*[*row*] = col

                max\_score += alpha\_beta(*board*, *row* + 1, *alpha*, *beta*, False)

*board*[*row*] = -1

*alpha* = max(*alpha*, max\_score)

*if* *beta* <= *alpha*:

*break*

*return* max\_score

*else*:

        min\_score = float('inf')

*for* col *in* range(len(*board*)):

*if* is\_valid(*board*, *row*, col):

*board*[*row*] = col

                min\_score = min(min\_score, alpha\_beta(*board*, *row* + 1, *alpha*, *beta*, True))

*board*[*row*] = -1

*beta* = min(*beta*, min\_score)

*if* *beta* <= *alpha*:

*break*

*return* min\_score

*def* solve\_8\_queens():

    board = [-1] \* 8

    alpha = -float('inf')

    beta = float('inf')

*return* alpha\_beta(board, 0, alpha, beta, True)

solutions = solve\_8\_queens()

print(*f*"Number of solutions for the 8 Queens problem: {solutions}")