**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



# LAB REPORT on

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Harsh C Kavediya(1BM23CS370)**

***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 **Harsh C Kavediya(1BM23CS370),** 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.

|  |  |
| --- | --- |
| Sandhya A Kulkarni  Assistant Professor  Department of CSE, BMSCE | **Dr. Kavitha Sooda**  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 20-08-2025 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent | 7 |
| 2 | 03-09-2025 | Implement 8 puzzle problems using Depth First  Search (DFS)  Implement Iterative deepening search algorithm | 13 |
| 3 | 10-09-2025 | Implement A\* search algorithm | 21 |
| 4 | 08-10-2025 | Implement Hill Climbing search algorithm to  solve N-Queens problem | 25 |
| 5 | 08-10-2025 | Simulated Annealing to Solve 8Queens problem | 29 |
| 6 | 15-10-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 32 |
| 7 | 29-10-2025 | Implement unification in first order logic | 35 |
| 8 | 12-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 36 |
| 9 | 12-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 39 |
| 10 | 12-11-2025 | Implement Alpha-Beta Pruning. | 44 |

**Github Link:**

**https://github.com/Harshkavediya17/AI**





A certificate with a qr code

AI-generated content may be incorrect.

A certificate with a qr code

AI-generated content may be incorrect.

**A certificate with a qr code

AI-generated content may be incorrect.**

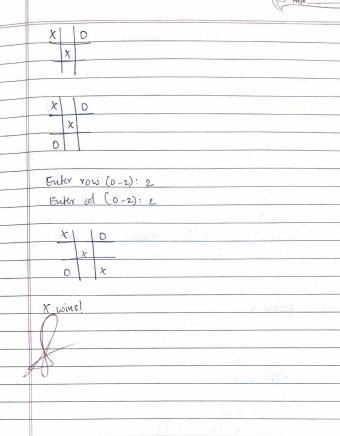
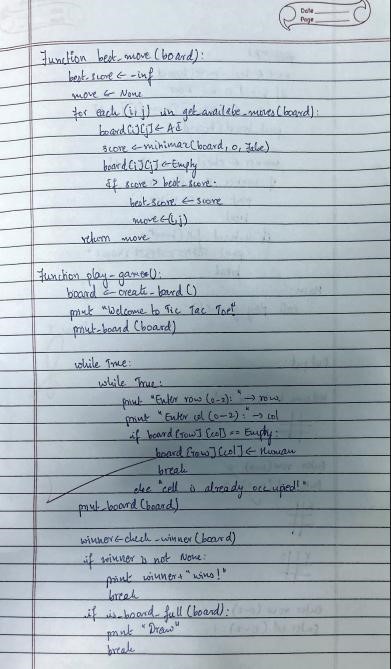
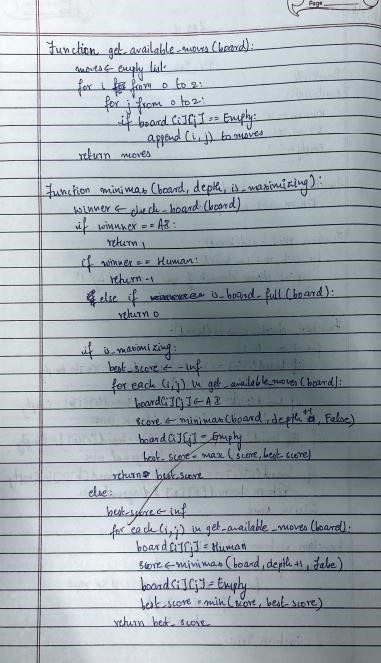
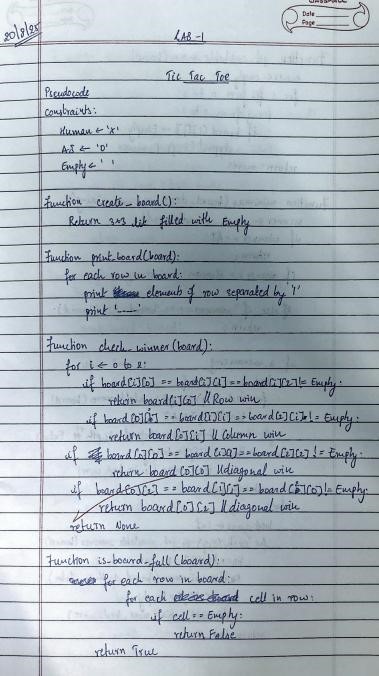
## Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

**Tic-Tac-Toe**

### Algorithm



### Code

def print\_board(board):

print("\n") for i in range(3):

print(" | ".join(board[i\*3:(i+1)\*3])) if i < 2: print("-" \* 10)

print("\n")

def check\_winner(board, player): win\_conditions = [

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

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

[0, 4, 8], [2, 4, 6]

]

for combo in win\_conditions:

count=0 for pos in combo: if board[pos]==player:

count+=1 if count==3: return True

return False

board = [" "] \* 9 current\_player = "X"

print\_board(board)

while True: while True:

pos = int(input(f"Player {current\_player}, enter your move (1-9): ")) - 1 if 0 <= pos <= 8 and board[pos] == " ":

board[pos] = current\_player

break else:

print("Invalid move. Try again.")

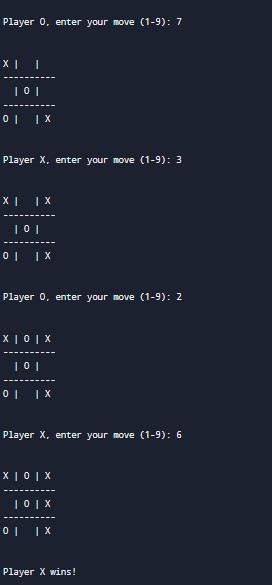
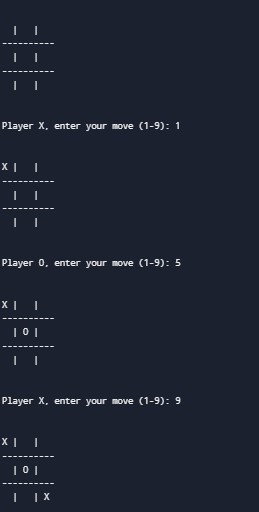
print\_board(board)

if check\_winner(board, current\_player): print(f"Player {current\_player} wins!") break if " " not in board: print("It's a draw!") break

# Switch players

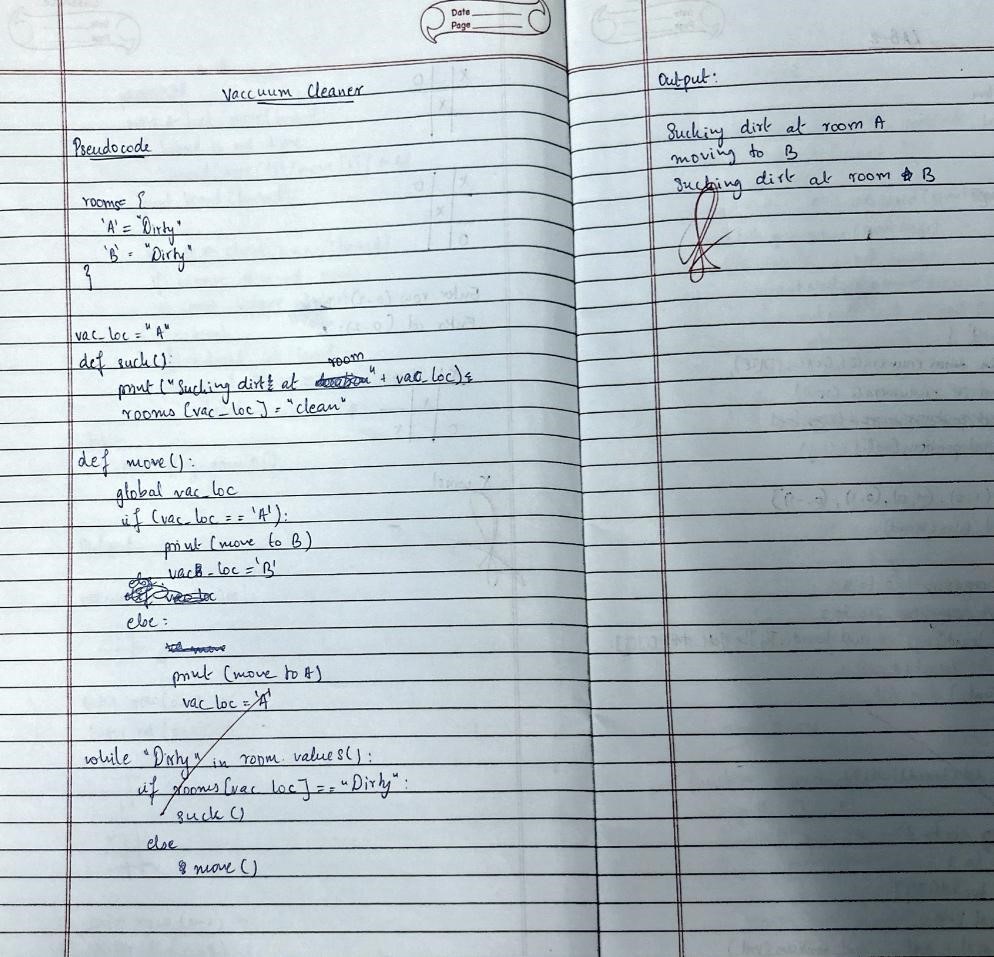
current\_player = "O" if current\_player == "X" else "X"

### Output



**Vacuum Cleaner**

### Algorithm



### Code

import random rooms=[1,1,1,1] botpos =(int(input("Enter Initial Position"))-1) cleanedpos=[] cost=0

def movebot(pos):

while True:

n= random.randint(0,3) if n != pos and n not in cleanedpos:

pos = n break return pos

while True: print(str(rooms))

print(botpos+1)

if rooms[botpos]==1:

rooms[botpos]=0 cleanedpos.append(botpos) cost+=1 if len(cleanedpos) == 4: break

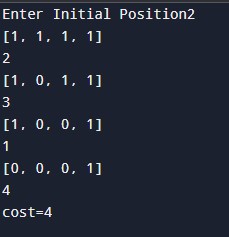
botpos=movebot(botpos)

elif rooms[botpos]==0: cleanedpos.append(botpos) if len(cleanedpos) == 4: break

botpos = movebot(botpos)

print("cost="+str(cost))

### Output



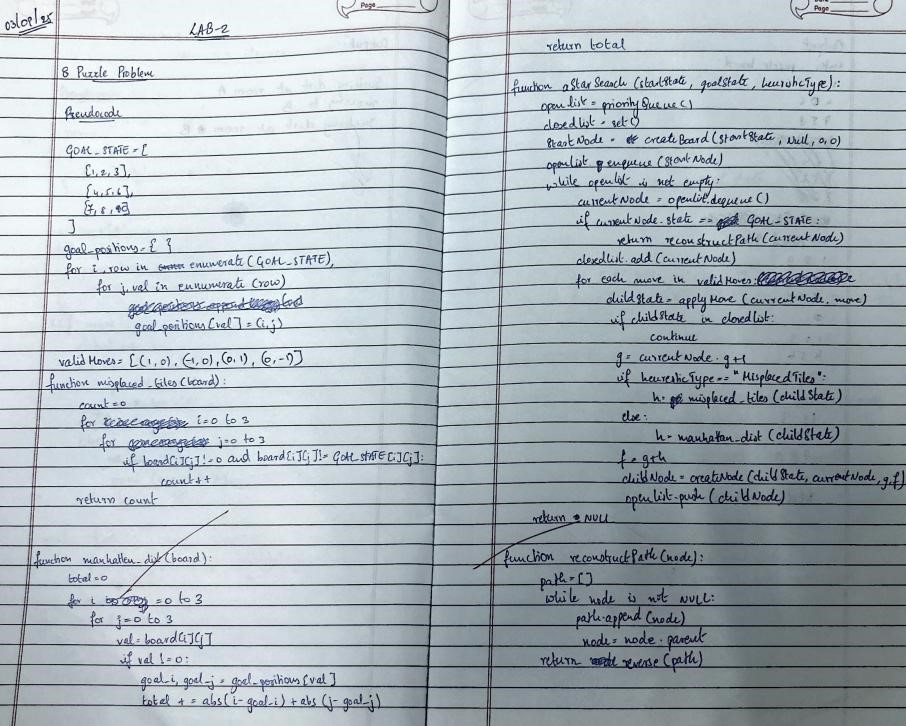
## Program 2

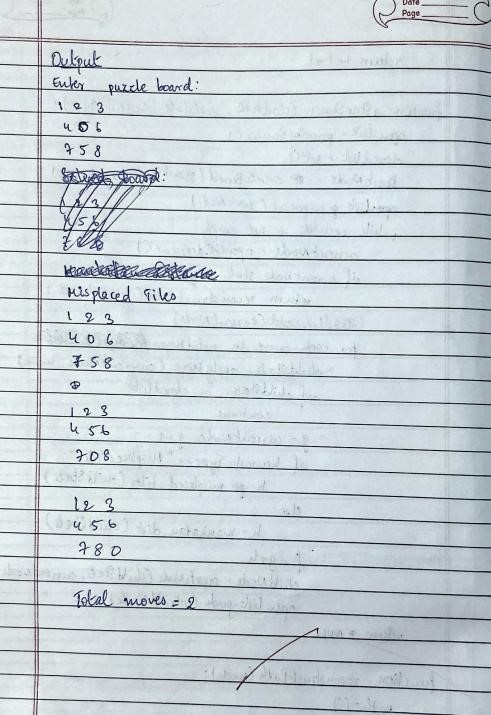
Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

**8 Puzzle Problem**

### Algorithm





### Code

import time

def find\_possible\_moves(state): index = state.index('\_') moves = {

0: [1, 3],

1: [0, 2, 4],

2: [1, 5],

3: [0, 4, 6],

4: [1, 3, 5, 7],

5: [2, 4, 8],

6: [3, 7],

7: [6, 8, 4],

8: [5, 7],

}

return moves.get(index, [])

def dfs(initial\_state, goal\_state, max\_depth=50):

stack = [(initial\_state, [], 0)] visited = {tuple(initial\_state)} states\_explored = 0

printed\_depths = set()

while stack:

current\_state, path, depth = stack.pop()

if depth > max\_depth:

continue

if depth not in printed\_depths: print(f"\n--- Depth {depth} ---")

printed\_depths.add(depth)

states\_explored += 1 print(f"State #{states\_explored}: {current\_state}")

if current\_state == goal\_state: print(f"\n Goal reached at depth {depth} after exploring {states\_explored} states.\n") return path, states\_explored

possible\_moves\_indices = find\_possible\_moves(current\_state)

for move\_index in reversed(possible\_moves\_indices): # Reverse for DFS order next\_state = list(current\_state) blank\_index = next\_state.index('\_') next\_state[blank\_index], next\_state[move\_index] = next\_state[move\_index], next\_state[blank\_index]

if tuple(next\_state) not in visited: visited.add(tuple(next\_state)) stack.append((next\_state, path + [next\_state], depth + 1))

print(f"\n Goal state not reachable within depth {max\_depth}. Explored {states\_explored} states.\n") return None, states\_explored

# ----------------- TEST ----------------- initial\_state = [1, 2, 3,

4, 8, '\_',

7, 6, 5]

goal\_state = [1, 2, 3,

4, 5, 6,

7, 8, '\_']

# Measure execution time start\_time = time.time() solution\_path, explored = dfs(initial\_state, goal\_state, max\_depth=50) end\_time = time.time()

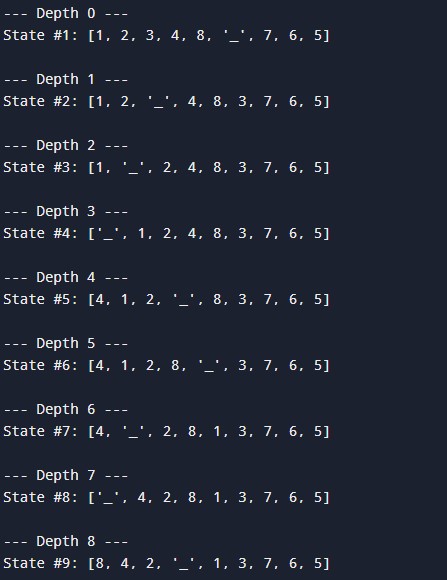
if solution\_path is None: print("No solution found.") else:

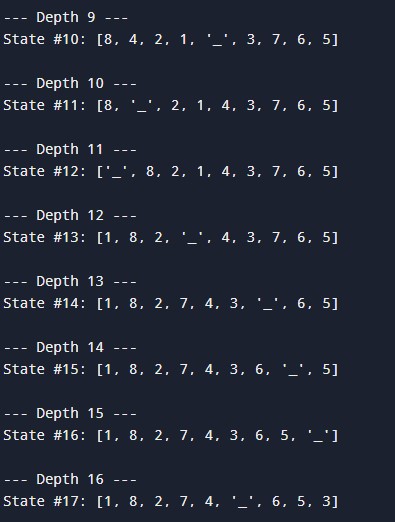
print("Solution path:") for step, state in enumerate(solution\_path, start=1):

print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end\_time - start\_time)) print("Total states explored:", explored)

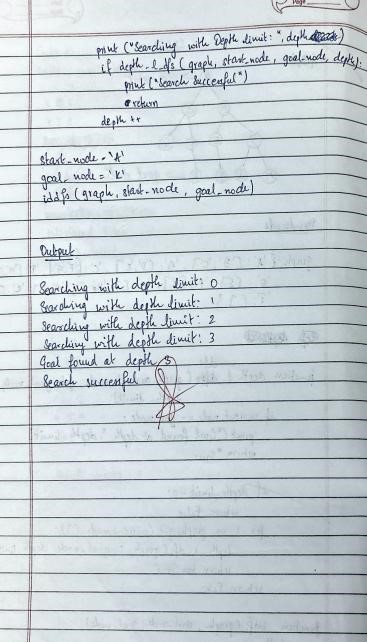
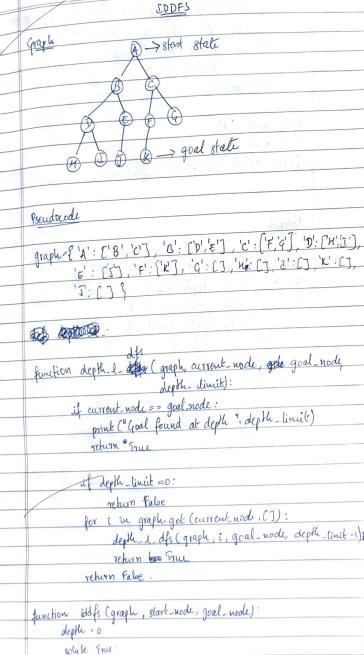
### Output





**IDDFS**

### Algorithm



### Code

import time

def find\_possible\_moves(state):

index = state.index('\_')

if index == 0: return [1, 3] elif index == 1: return [0, 2, 4] elif index == 2: return [1, 5] elif index == 3: return [0, 4, 6] elif index == 4:

return [1, 3, 5, 7] elif index == 5: return [2, 4, 8] elif index == 6: return [3, 7] elif index == 7: return [4, 6, 8] elif index == 8: return [5, 7]

return []

def depth\_limited\_dfs(state, goal\_state, limit, path, visited): if state == goal\_state:

return path

if limit <= 0: return None

visited.add(tuple(state))

for move\_index in find\_possible\_moves(state):

next\_state = list(state) blank\_index = next\_state.index('\_') next\_state[blank\_index], next\_state[move\_index] = next\_state[move\_index], next\_state[blank\_index]

if tuple(next\_state) not in visited:

result = depth\_limited\_dfs(next\_state, goal\_state, limit - 1, path + [next\_state], visited) if result is not None:

return result

return None

def iddfs(initial\_state, goal\_state, max\_depth=30): for depth in range(max\_depth):

print(f"Searching at depth limit = {depth}") visited = set() result = depth\_limited\_dfs(initial\_state, goal\_state, depth, [initial\_state], visited) if result is not None: return result, depth

return None, max\_depth

# ----------------- TEST ----------------- initial\_state = [1, 2, 3,

4, 8, '\_',

7, 6, 5]

goal\_state = [1, 2, 3,

4, 5, 6,

7, 8, '\_']

# Measure execution time start\_time = time.time() solution\_path, depth\_reached = iddfs(initial\_state, goal\_state, max\_depth=30) end\_time = time.time()

if solution\_path is None:

print("Goal state is not reachable within given depth limit.") else:

print("\n\nSolution path found:") for step, state in enumerate(solution\_path, start=0):

print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end\_time - start\_time)) print("Depth reached:", depth\_reached)

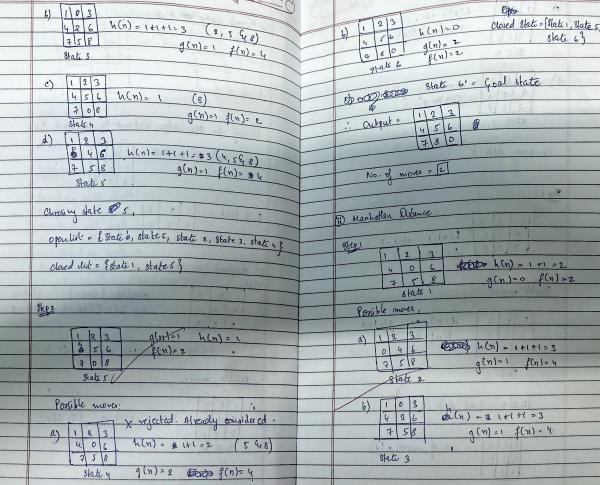
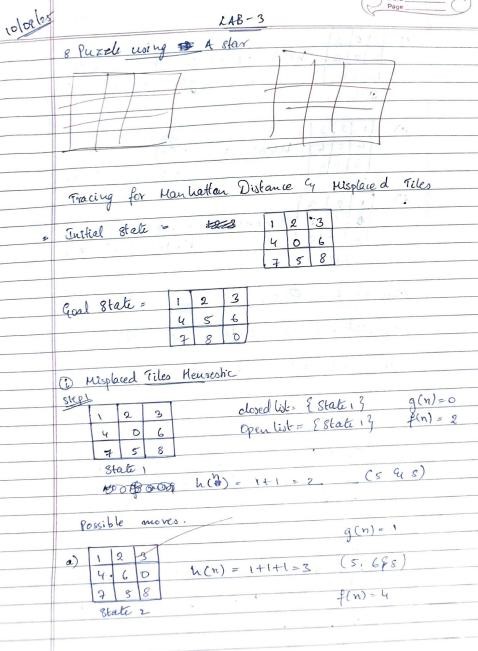
### Output

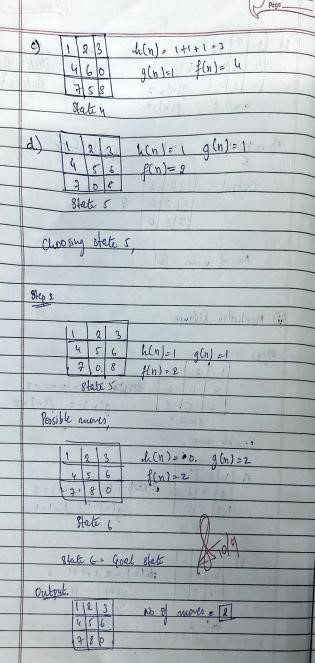


## Program 3

Implement A\* search algorithm

**Algorithm**





### Code

import heapq

import time

# Heuristic: Manhattan Distance def heuristic(state, goal):

distance = 0 for i in range(1, 9): # tile numbers 1 to 8 x1, y1 = divmod(state.index(i), 3) x2, y2 = divmod(goal.index(i), 3) distance += abs(x1 - x2) + abs(y1 - y2) return distance

# Get neighbors by sliding blank (0) up/down/left/right def get\_neighbors(state):

neighbors = [] i = state.index(0) # position of blank x, y = divmod(i, 3)

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

for dx, dy in moves:

new\_x, new\_y = x + dx, y + dy if 0 <= new\_x < 3 and 0 <= new\_y < 3: j = new\_x \* 3 + new\_y new\_state = list(state) new\_state[i], new\_state[j] = new\_state[j], new\_state[i] neighbors.append(tuple(new\_state)) return neighbors

# A\* Search for 8-puzzle def astar(start, goal): open\_set = [] heapq.heappush(open\_set, (heuristic(start, goal), 0, start))

came\_from = {}

g\_score = {start: 0}

while open\_set:

\_, cost, current = heapq.heappop(open\_set)

if current == goal: # Reconstruct path path = [] while current in came\_from: path.append(current) current = came\_from[current]

path.append(start)

return path[::-1]

for neighbor in get\_neighbors(current): tentative\_g = g\_score[current] + 1 if neighbor not in g\_score or tentative\_g < g\_score[neighbor]:

came\_from[neighbor] = current g\_score[neighbor] = tentative\_g f\_score = tentative\_g + heuristic(neighbor, goal)

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

return None # no solution

# ----------------- TEST ----------------- start = (1, 2, 3,

4, 8, 0,

7, 6, 5)

goal = (1, 2, 3,

4, 5, 6,

7, 8, 0)

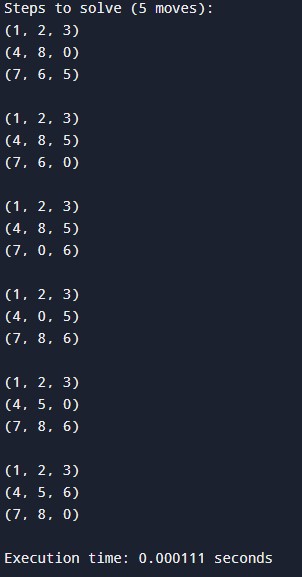
# Measure execution time start\_time = time.time() path = astar(start, goal) end\_time = time.time()

if path:

print("Steps to solve ({} moves):".format(len(path)-1)) for state in path: for i in range(0, 9, 3): print(state[i:i+3]) print() else: print("No solution found")

print("Execution time: {:.6f} seconds".format(end\_time - start\_time))

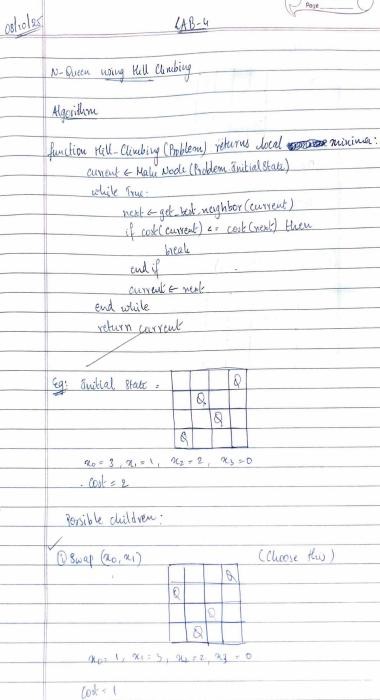
### Output

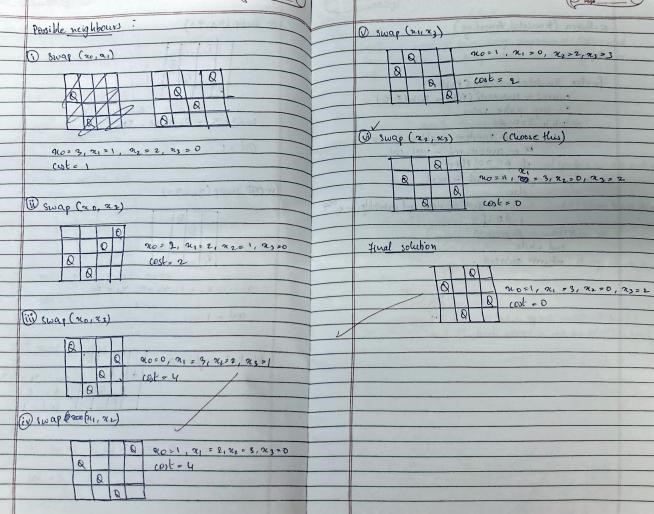
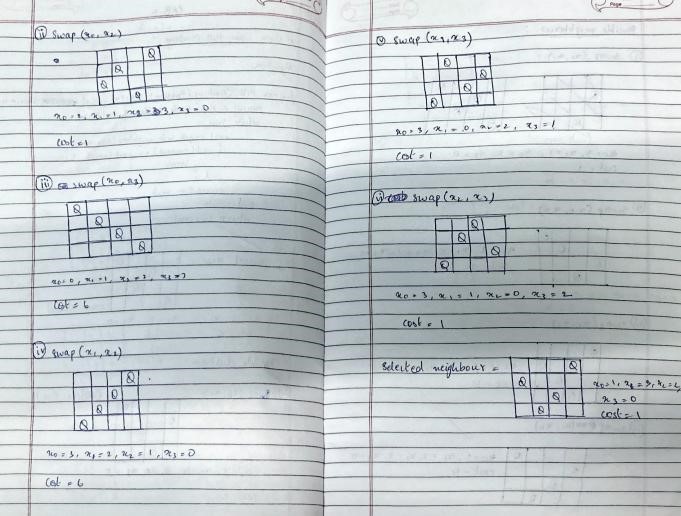


## Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

### Algorithm





### Code

import random

import math

def compute\_cost(state):

"""Count diagonal conflicts for a permutation-state (one queen per row & column).""" conflicts = 0 n = len(state) for i in range(n): for j in range(i + 1, n): if abs(state[i] - state[j]) == abs(i - j):

conflicts += 1

return conflicts

def random\_permutation(n): arr = list(range(n)) random.shuffle(arr)

return arr

def neighbors\_by\_swaps(state):

"""All neighbors obtained by swapping two columns (keeps permutation property).""" n = len(state) for i in range(n - 1): for j in range(i + 1, n): nb = state.copy() nb[i], nb[j] = nb[j], nb[i] yield nb

def hill\_climb\_with\_restarts(n, max\_restarts=None):

"""Hill climbing on permutations with random restart on plateau (no revisits).""" visited = set() total\_states = math.factorial(n)

restarts = 0

while True:

# pick a random unvisited start permutation if len(visited) >= total\_states:

raise RuntimeError("All states visited — giving up (no solution found).")

state = random\_permutation(n) while tuple(state) in visited:

state = random\_permutation(n)

visited.add(tuple(state))

# climb from this start while True:

cost = compute\_cost(state) if cost == 0:

return state, restarts

# find best neighbor (swap-based neighbors) best\_neighbor = None

best\_cost = float("inf") for nb in neighbors\_by\_swaps(state):

c = compute\_cost(nb) if c < best\_cost: best\_cost = c

best\_neighbor = nb

# if strictly better, move; otherwise it's a plateau/local optimum -> restart if best\_cost < cost:

state = best\_neighbor visited.add(tuple(state)) else:

# plateau or local optimum -> restart

restarts += 1 if max\_restarts is not None and restarts >= max\_restarts:

raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).") break # go pick a new unvisited start

def format\_board(state):

n = len(state) lines = [] for r in range(n):

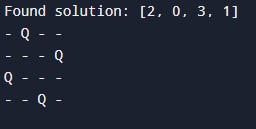
lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n))) return "\n".join(lines)

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

n = 4 solution, restarts = hill\_climb\_with\_restarts(n) print("Found solution:", solution)

print(format\_board(solution))

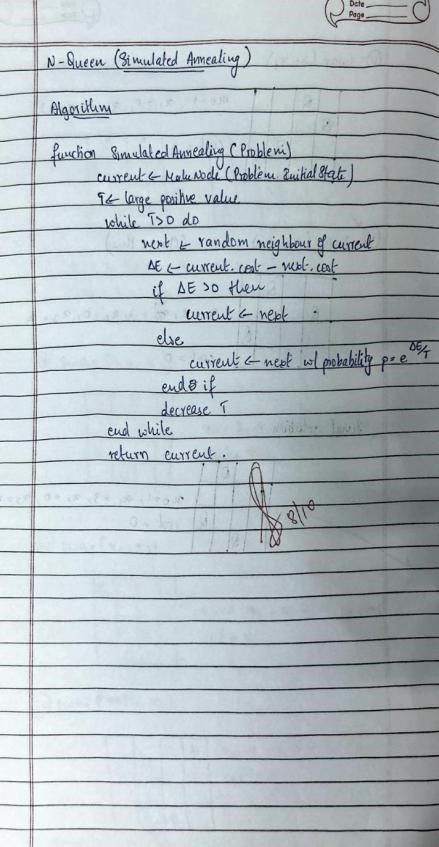
### Output



## Program 5

Simulated Annealing to Solve 8-Queens problem

### Algorithm



### Code

import random

import math

def cost(state):

attacks = 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):

attacks += 1

return attacks

def get\_neighbor(state):

neighbor = state[:] i, j = random.sample(range(len(state)), 2) neighbor[i], neighbor[j] = neighbor[j], neighbor[i] return neighbor

def simulated\_annealing(n=8, max\_iter=10000, temp=100.0, cooling=0.95):

current = list(range(n)) random.shuffle(current)

current\_cost = cost(current)

temperature = temp

cooling\_rate = cooling

best = current[:]

best\_cost = current\_cost

for \_ in range(max\_iter): if temperature <= 0 or best\_cost == 0: break

neighbor = get\_neighbor(current) neighbor\_cost = cost(neighbor)

delta = current\_cost - neighbor\_cost

if delta > 0 or random.random() < math.exp(delta / temperature):

current, current\_cost = neighbor, neighbor\_cost if neighbor\_cost < best\_cost:

best, best\_cost = neighbor[:], neighbor\_cost

temperature \*= cooling\_rate

return best, best\_cost

def print\_board(state):

n = len(state) for row in range(n):

line = " ".join("Q" if state[col] == row else "." for col in range(n)) print(line)

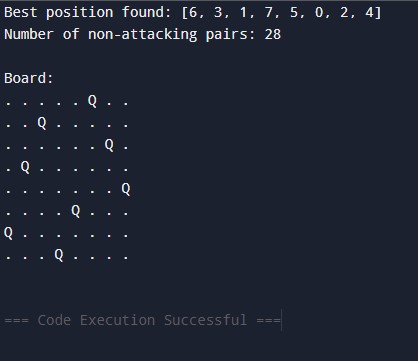
print()

n = 8 solution, cost\_val = simulated\_annealing(n, max\_iter=20000) print("Best position found:", solution)

print(f"Number of non-attacking pairs: {n\*(n-1)//2 - cost\_val}") print("\nBoard:")

print\_board(solution)

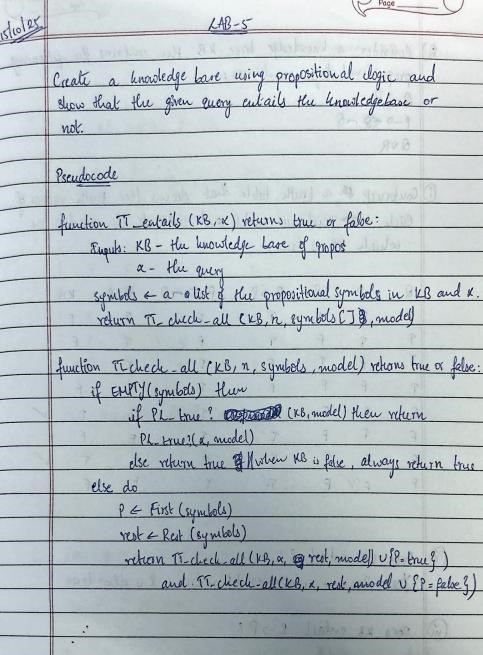
### Output

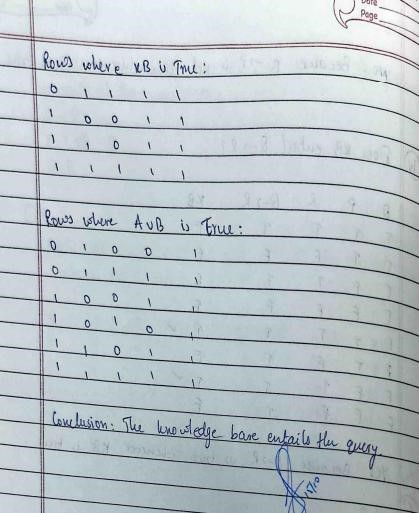


## Program 6

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, truth\_assignment):

eval\_formula = formula for symbol, value in truth\_assignment.items():

eval\_formula = eval\_formula.replace(symbol, str(value)) return eval(eval\_formula)

def generate\_truth\_table(variables):

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

def is\_entailed(KB\_formula, alpha\_formula, variables): truth\_combinations = generate\_truth\_table(variables) print(f"{' '.join(variables)} | KB Result | Alpha Result") print("-" \* (len(variables) \* 2 + 15)) for combination in truth\_combinations:

truth\_assignment = dict(zip(variables, combination)) KB\_value = evaluate\_formula(KB\_formula, truth\_assignment) alpha\_value = evaluate\_formula(alpha\_formula, truth\_assignment) result\_str = " ".join(["T" if value else "F" for value in combination]) print(f"{result\_str} | {'T' if KB\_value else 'F'} | {'T' if alpha\_value else 'F'}") if KB\_value and not alpha\_value:

return False

return True

KB = "(A or C) and (B or not C)"

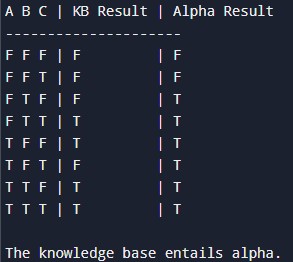
alpha = "A or B"

variables = ['A', 'B', 'C']

if is\_entailed(KB, alpha, variables):

print("\nThe knowledge base entails alpha.") else: print("\nThe knowledge base does not entail alpha.")

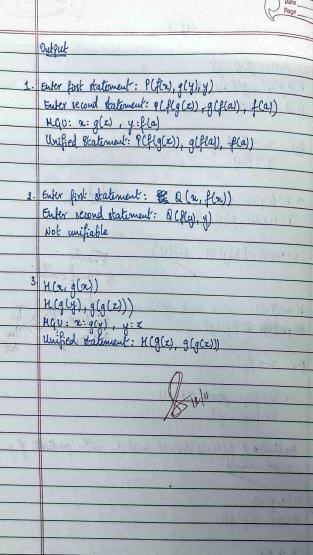
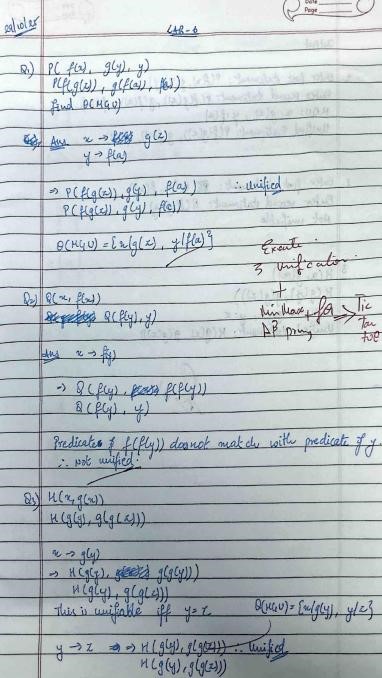
### Output



## Program 7

Implement unification in first order logic

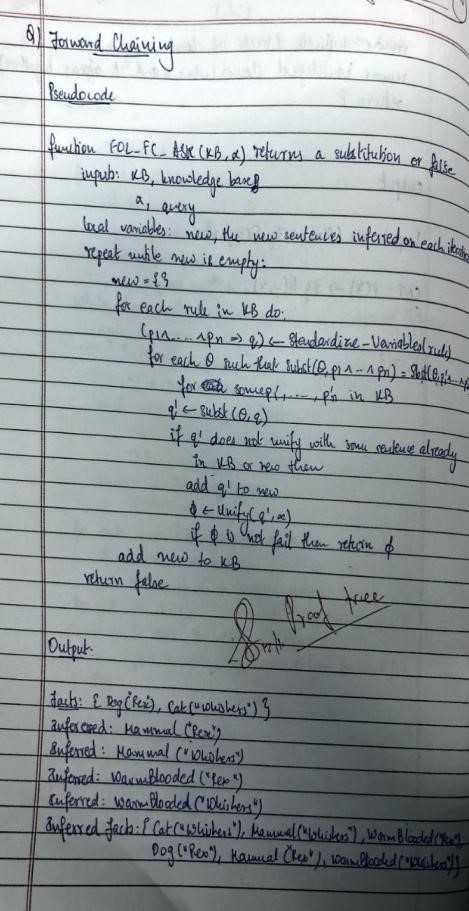
### Algorithm



## Program 8

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

### Algorithm



### Code

import re

def match\_pattern(pattern, fact):

"""

Checks if a fact matches a rule pattern using regex-style variable substitution.

Variables are lowercase words like p, q, x, r etc.

Returns a dict of substitutions or None if not matched.

"""

# Extract predicate name and arguments pattern\_pred, pattern\_args = re.match(r'(\w+)

', pattern).groups()

fact\_pred, fact\_args = re.match(r'(\w+)

', fact).groups()

if pattern\_pred != fact\_pred:

return None # predicate mismatch

pattern\_args = [a.strip() for a in pattern\_args.split(",")] fact\_args = [a.strip() for a in fact\_args.split(",")]

if len(pattern\_args) != len(fact\_args):

return None

subst = {} for p\_arg, f\_arg in zip(pattern\_args, fact\_args): if re.fullmatch(r'[a-z]\w\*', p\_arg): # variable subst[p\_arg] = f\_arg elif p\_arg != f\_arg: # constants mismatch

return None

return subst

def apply\_substitution(expr, subst):

"""Replaces all variable names in expr using the given substitution dict.""" for var, val in subst.items():

expr = re.sub(rf'\b{var}\b', val, expr) return expr

# ---------- Knowledge Base ----------

rules = [

(["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)"),

(["Missile(x)"], "Weapon(x)"),

(["Enemy(x, America)"], "Hostile(x)"),

(["Missile(x)", "Owns(A, x)"], "Sells(Robert, x, A)")

]

facts = {

"American(Robert)",

"Enemy(A, America)",

"Owns(A, T1)",

"Missile(T1)"

}

goal = "Criminal(Robert)"

def forward\_chain(rules, facts, goal):

added = True

while added: added = False for premises, conclusion in rules:

possible\_substs = [] for p in premises: for f in facts:

subst = match\_pattern(p, f) if subst:

possible\_substs.append(subst)

break else:

break else:

combined = {} for s in possible\_substs: combined.update(s)

new\_fact = apply\_substitution(conclusion, combined)

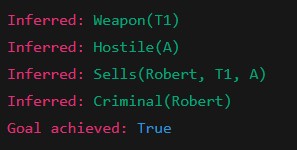
if new\_fact not in facts: facts.add(new\_fact) print(f"Inferred: {new\_fact}") added = True if new\_fact == goal:

return True

return goal in facts

print("Goal achieved:", forward\_chain(rules, facts, goal))

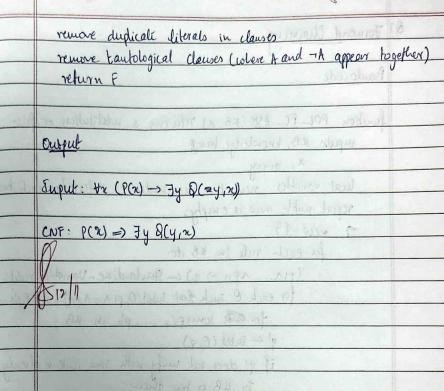
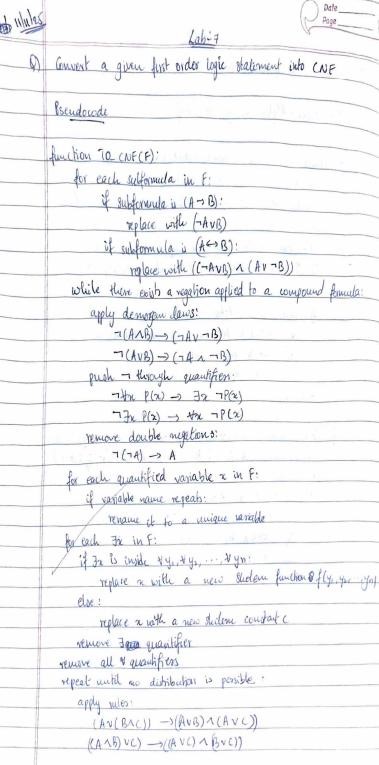
### Output



## Program 9

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

### Algorithm



### Code

from copy import deepcopy

def print\_step(title, content):

print(f"\n{'='\*45}\n{title}\n{'='\*45}") if isinstance(content, list): for i, c in enumerate(content, 1):

print(f"{i}. {c}") else:

print(content)

KB = [

["¬Food(x)", "Likes(John,x)"],

["Food(Apple)"],

["Food(Vegetable)"],

["¬Eats(x,y)", "Killed(x)", "Food(y)"],

["Eats(Anil,Peanuts)"],

["Alive(Anil)"],

["¬Alive(x)", "¬Killed(x)"],

["Killed(x)", "Alive(x)"]

]

QUERY = ["Likes(John,Peanuts)"]

def negate(literal): if literal.startswith("¬"):

return literal[1:] return "¬" + literal

def substitute(clause, subs):

new\_clause = [] for lit in clause: for var, val in subs.items(): lit = lit.replace(var, val) new\_clause.append(lit)

return new\_clause

def unify(lit1, lit2):

"""Small unifier for patterns like Food(x) and Food(Apple).""" if "(" not in lit1 or "(" not in lit2: return None pred1, args1 = lit1.split("(") pred2, args2 = lit2.split("(") args1 = args1[:-1].split(",") args2 = args2[:-1].split(",") if pred1 != pred2 or len(args1) != len(args2):

return None subs = {} for a, b in zip(args1, args2): if a == b: continue if a.islower(): subs[a] = b elif b.islower(): subs[b] = a else:

return None

return subs

def resolve(ci, cj):

"""Return list of (resolvent, substitution, pair).""" resolvents = [] for li in ci: for lj in cj: if li == negate(lj):

new\_clause = [x for x in ci if x != li] + [x for x in cj if x != lj] resolvents.append((list(set(new\_clause)), {}, (li, lj))) else:

# same predicate, opposite sign if li.startswith("¬") and not lj.startswith("¬") and li[1:].split("(")[0] == lj.split("(")[0]:

subs = unify(li[1:], lj) if subs:

new\_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs) resolvents.append((list(set(new\_clause)), subs, (li, lj))) elif lj.startswith("¬") and not li.startswith("¬") and lj[1:].split("(")[0] == li.split("(")[0]:

subs = unify(lj[1:], li) if subs:

new\_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs) resolvents.append((list(set(new\_clause)), subs, (li, lj))) return resolvents

def resolution(kb, query): clauses = deepcopy(kb) negated\_query = [negate(q) for q in query] clauses.append(negated\_query)

print\_step("Initial Clauses", clauses)

steps = [] new = [] while True:

pairs = [(clauses[i], clauses[j]) for i in range(len(clauses))

for j in range(i + 1, len(clauses))] for (ci, cj) in pairs: for r, subs, pair in resolve(ci, cj): if not r:

steps.append({

"parents": (ci, cj),

"resolvent": r,

"subs": subs

})

print\_tree(steps) print("\n✅ Empty clause derived — query proven.") return True if r not in clauses and r not in new:

new.append(r) steps.append({ "parents": (ci, cj),

"resolvent": r,

"subs": subs

})

if all(r in clauses for r in new):

print\_step("No New Clauses", "Query cannot be proven ❌") print\_tree(steps) return False

clauses.extend(new)

def print\_tree(steps): print("\n" + "="\*45) print("Resolution Proof Trace") print("="\*45) for i, s in enumerate(steps, 1):

p1, p2 = s["parents"] r = s["resolvent"] subs = s["subs"] subs\_text = f" Substitution: {subs}" if subs else ""

print(f" Resolve {p1} and {p2}") if subs\_text:

print(subs\_text) if r: print(f" ⇒ {r}") else:

print(" ⇒ {} (empty clause)") print("-"\*45)

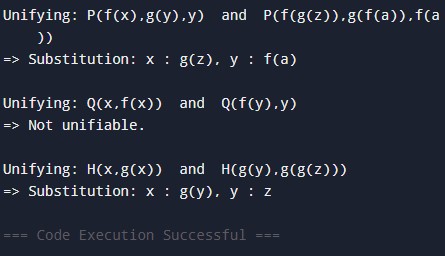
def main():

print\_step("Knowledge Base in CNF", KB) print\_step("Negated Query", [negate(q) for q in QUERY]) proven = resolution(KB, QUERY) if proven: print("\n✅ Query Proven by Resolution: John likes peanuts.") else:

print("\n❌ Query cannot be proven from KB.")

if \_\_name\_\_ == "\_\_main\_\_": main()

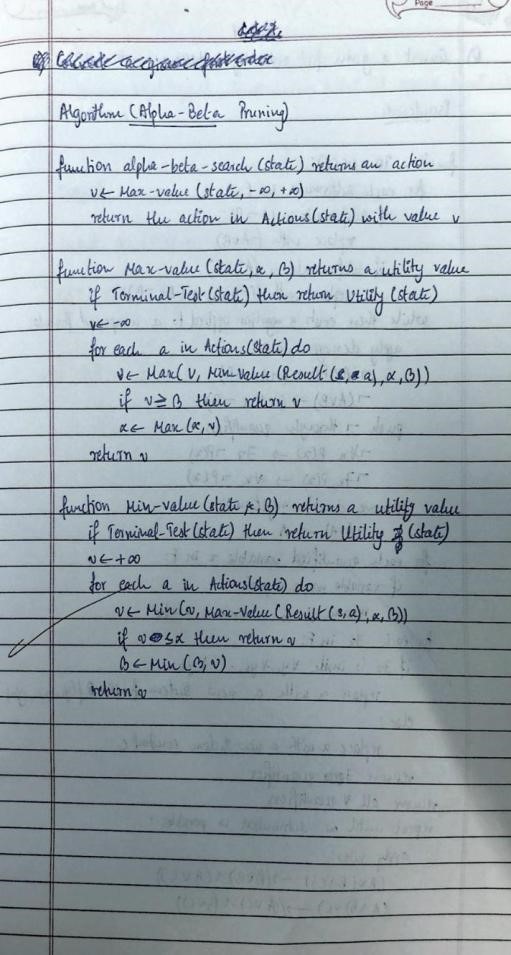
### Output



## Program 10

Implement Alpha-Beta Pruning

### Algorithm



### Code

def unify(a, b):

"""Very simple unification for small terms like ('line', [X,O,X])""" if a == b: return {} if isinstance(a, str) and a.islower(): # variable

return {a: b} if isinstance(b, str) and b.islower():

return {b: a} if isinstance(a, tuple) and isinstance(b, tuple): if a[0] != b[0] or len(a[1]) != len(b[1]):

return None subs = {} for x, y in zip(a[1], b[1]): s = unify(x, y) if s is None: return None subs.update(s) return subs

return None

# Winning triples (rows, cols, diagonals)

WIN\_TRIPLES = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def winner(board):

pattern = ('line', ['X','X','X']) for i,j,k in WIN\_TRIPLES: term = ('line', [board[i], board[j], board[k]]) if unify(term, pattern):

return 'X' if unify(term, ('line',['O','O','O'])):

return 'O'

return None

def is\_full(board): return all(c != '\_' for c in board)

def evaluate(board): w = winner(board) if w == 'X': return 1 if w == 'O': return -1 if is\_full(board): return 0

return None

def alpha\_beta(board, player, alpha=-float('inf'), beta=float('inf')):

val = evaluate(board) if val is not None: return val, None

moves = [i for i,c in enumerate(board) if c == '\_']

best\_move = None if player == 'X':

max\_eval = -float('inf') for m in moves:

new\_board = board[:] new\_board[m] = 'X' eval\_, \_ = alpha\_beta(new\_board, 'O', alpha, beta) if eval\_ > max\_eval:

max\_eval, best\_move = eval\_, m alpha = max(alpha, eval\_) if beta <= alpha: break

return max\_eval, best\_move else:

min\_eval = float('inf') for m in moves:

new\_board = board[:] new\_board[m] = 'O' eval\_, \_ = alpha\_beta(new\_board, 'X', alpha, beta) if eval\_ < min\_eval: min\_eval, best\_move = eval\_, m beta = min(beta, eval\_) if beta <= alpha: break

return min\_eval, best\_move

def print\_board(b): for i in range(0,9,3):

print(' '.join(b[i:i+3])) print()

# --- Example usage --- board = ['\_']\*9 score, move = alpha\_beta(board, 'X') print("Best first move for X:", move) board[move] = 'X'

print\_board(board)

### Output

