**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Tanmay Agarwal (1WA23CS010)**

***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**

**Aug-2025 to Dec-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 **Tanmay Agarwal (1WA23CS010),** 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. K.R. Mamatha  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 12-8-2025 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent |  |
| 2 | 19-8-2025 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 9 |
| 3 | 26-8-2025 | Implement Hill Climbing search algorithm to solve N-Queens problem | 14 |
| 4 | 2-9-2025 | Implement A\* search algorithm | 18 |
| 5 | 16-9-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 21 |
| 6 | 23-9-2025 | Implement unification in first order  logic | 24 |
| 7 | 30-09-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 36 |
| 8 | 21-10-2025 | Implement Alpha-Beta Pruning. | 45 |

Github Link:

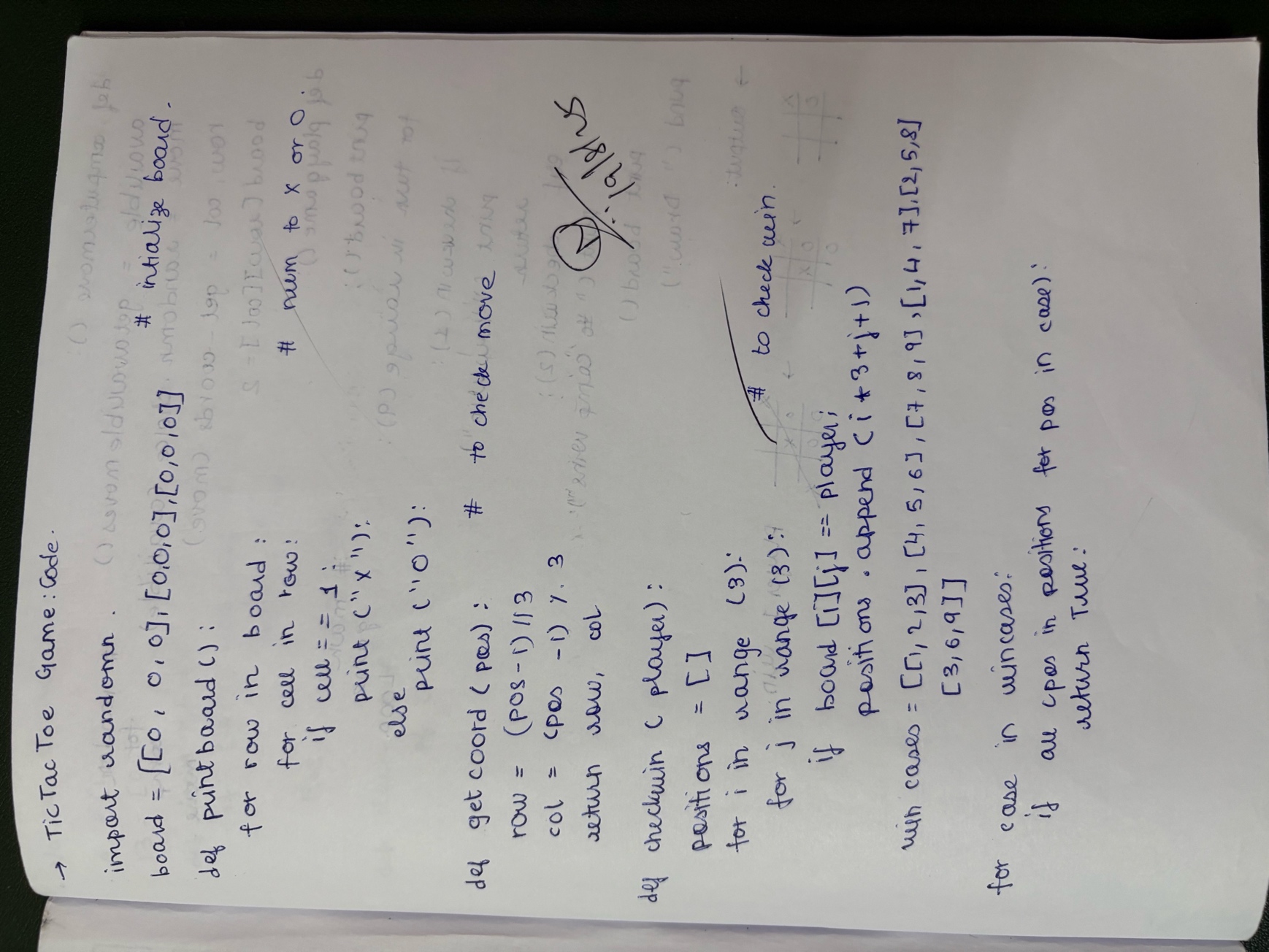
*https://github.com/Tanmay1WA23CS010/ArtificialIntelligence*

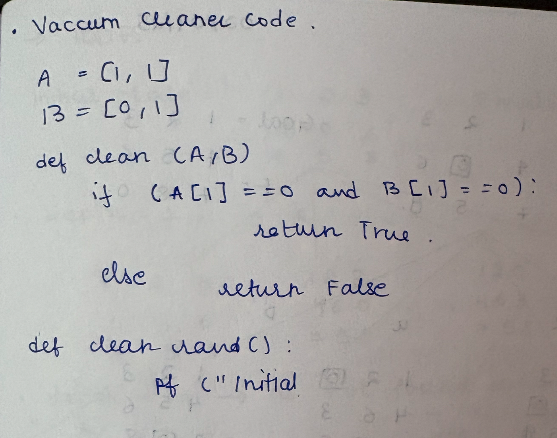
**Program 1**

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:





Code:

TIC TAC TOE

import random # Import random module to allow computer to pick random moves

# Initialize the board as a 3x3 2D list filled with zeros (0 means empty cell)

board = [[0, 0, 0],

[0, 0, 0],

[0, 0, 0]]

# List of all winning position combinations based on numbered board positions

win\_combinations = [

[1, 2, 3], # Top row

[4, 5, 6], # Middle row

[7, 8, 9], # Bottom row

[1, 4, 7], # Left column

[2, 5, 8], # Middle column

[3, 6, 9], # Right column

[1, 5, 9], # Left diagonal

[3, 5, 7] # Right diagonal

]

# Function to print the current state of the board in a user-friendly way

def print\_board():

for row in board:

display = [] # Temporary list to hold string representations of cells

for cell in row:

if cell == 1: # If cell contains 1, print 'X' (user)

display.append('X')

elif cell == 2: # If cell contains 2, print 'O' (computer)

display.append('O')

else: # If cell is 0 (empty), print a blank space

display.append(' ')

print(" | ".join(display)) # Join the cells with vertical bars for board format

print("-" \* 9) # Print a line separator between rows

# Convert a board position number (1-9) into row and column indices for 2D list

def get\_position\_coords(pos):

row = (pos - 1) // 3 # Calculate row index (0 to 2)

col = (pos - 1) % 3 # Calculate column index (0 to 2)

return row, col

# Check if a given player (1 or 2) has won the game by matching any winning combo

def check\_win(player):

positions = [] # List to store all board positions occupied by the player

for i in range(3): # Loop through each row

for j in range(3): # Loop through each column

if board[i][j] == player: # If player's mark found at this cell

pos = i \* 3 + j + 1 # Convert coordinates to position number (1-9)

positions.append(pos) # Add position to player's list

for combo in win\_combinations: # Check all winning combinations

if all(pos in positions for pos in combo): # If player has all positions in a combo

return True # Player has won

return False # No winning combination found

# Get a list of all empty positions (available moves) on the board

def get\_available\_moves():

moves = [] # List to store available position numbers

for i in range(3): # Loop through rows

for j in range(3): # Loop through columns

if board[i][j] == 0: # If cell is empty

moves.append(i \* 3 + j + 1) # Add its position number to available moves

return moves # Return list of free spots

# Handle the user's move by asking for input and placing an 'X' if valid

def user\_move():

while True: # Keep looping until a valid move is made

try:

move = int(input("Enter your move (1-9): ")) # Ask user for move input

if move < 1 or move > 9: # Check if input is out of range

print("Invalid input. Choose a number between 1 and 9.")

continue # Ask again

row, col = get\_position\_coords(move) # Convert move to row,col

if board[row][col] == 0: # If the spot is empty

board[row][col] = 1 # Place user's mark (1) on the board

break # Exit loop since move is valid

else:

print("That spot is already taken.") # Spot occupied, ask again

except ValueError: # Catch non-integer inputs

print("Please enter a valid number.") # Prompt again

# Handle the computer's move by randomly choosing one of the available spots

def computer\_move():

available = get\_available\_moves() # Get list of free spots

move = random.choice(available) # Pick one random move from available ones

row, col = get\_position\_coords(move) # Convert position to row,col

board[row][col] = 2 # Place computer's mark (2) on the board

print(f"Computer chose position {move}") # Show computer's choice

# Main function to play the Tic Tac Toe game

def play\_game():

print("Tic Tac Toe - You (X) vs Computer (O)") # Game start message

print\_board() # Display empty board initially

for turn in range(9): # Maximum of 9 moves possible on the board

if turn % 2 == 0: # Even turns: user's move

user\_move()

else: # Odd turns: computer's move

computer\_move()

print\_board() # Show updated board after each move

if check\_win(1): # Check if user has won

print("You win!")

return # End the game

elif check\_win(2): # Check if computer has won

print("Computer wins!")

return # End the game

print("It's a draw!") # If loop ends with no winner, it's a draw

# Start the game by calling the main function

play\_game()

# i need to make a tic tac toe game using Ai, where there are 2 players, one is the computer and one

# is the user, i want to use 2d array in order to do this,

# use 1 for X and 2 for Y, it shouol be a siple code where either

# the computer wins or the user wins, all win or loose cases must be cinsidered

#the board is numbered :

# 1 2 3

# 4 5 6

# 7 8 9

#and the winning cases are if the player has [1,2,3]. [4,5,6], [7,8,9], [1,4,7], [2,5,8], [3,6,9]

VACCUM CLEANER

print("Vacuum Cleaner")

A=[1,1] # Room A: Vacuum present, Dirt present

B=[0,1] # Room B: Vacuum absent, Dirt present

def cleanCheck(A,B):

if(A[1] == 0 and B[1] == 0):

return True

else:

return False

def clean\_room():

print("Initial State: Room A",A,", Room B",B)

while not cleanCheck(A,B):

if(A[0] == 1):

A[1] = 0

A[0] = 0

B[0] = 1

print("Vacuum cleaner moved to room B. Room A cleaned.")

elif(B[0] == 1):

B[1] = 0

B[0] = 0

A[0] = 1

print("Vacuum cleaner moved to room A. Room B cleaned.")

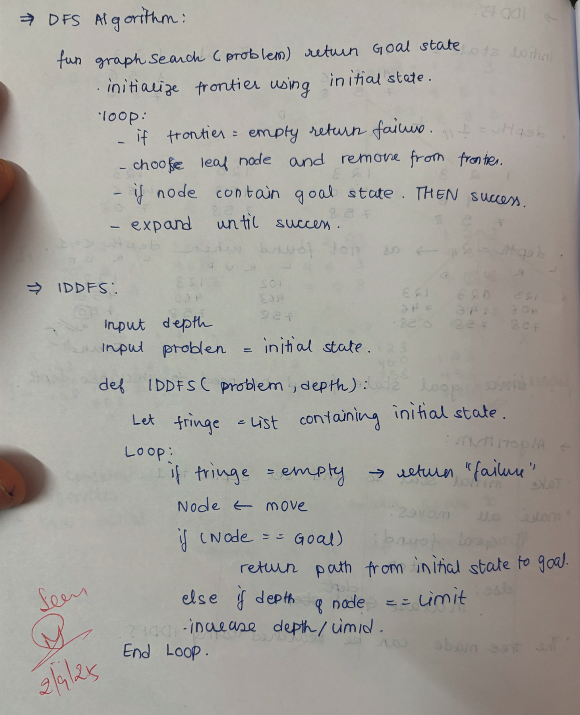
print("Final State: Room A",A,", Room B",B)

clean\_room()

Programme 2:

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:  
  


Code:

# 8-Puzzle using Iterative Deepening DFS (IDDFS)

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]] # 0 is the blank tile

# Directions: (dx, dy, move\_name)

moves = [(-1, 0, "Up"),

(1, 0, "Down"),

(0, -1, "Left"),

(0, 1, "Right")]

def find\_blank(state):

"""Find the blank (0) position."""

for i in range(3):

for j in range(3):

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

return i, j

def is\_goal(state):

return state == goal\_state

def print\_state(state):

for row in state:

print(row)

print()

def dls(state, depth\_limit, path, visited):

"""Depth-Limited Search (helper for IDDFS)."""

if is\_goal(state):

print("✅ Goal found with depth limit:", depth\_limit)

print("Moves:", " -> ".join(path))

print()

return True

if depth\_limit == 0:

return False

visited.add(tuple(map(tuple, state)))

x, y = find\_blank(state)

for dx, dy, move in moves:

nx, ny = x + dx, y + dy

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

# Copy state

new\_state = [row[:] for row in state]

# Swap blank

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

if tuple(map(tuple, new\_state)) not in visited:

if dls(new\_state, depth\_limit - 1, path + [move], visited):

return True

return False

def iddfs(initial\_state, max\_depth=20):

"""Iterative Deepening DFS."""

for depth in range(max\_depth + 1):

visited = set()

print(f"🔎 Searching with depth limit = {depth}")

if dls(initial\_state, depth, [], visited):

return True

return False

# Example

initial\_state = [[1, 2, 3],

[4, 0, 6],

[7, 5, 8]]

print("Initial State:")

print\_state(initial\_state)

if not iddfs(initial\_state, max\_depth=20):

print("❌ Goal not found within depth limit.")

# 8-Puzzle using DFS with move directions

goal\_state = [[1, 2, 3],

[4, 5, 6],

[7, 8, 0]] # 0 is the blank tile

# Directions: (dx, dy, move\_name)

moves = [(-1, 0, "Up"),

(1, 0, "Down"),

(0, -1, "Left"),

(0, 1, "Right")]

def find\_blank(state):

"""Find the blank (0) position."""

for i in range(3):

for j in range(3):

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

return i, j

def is\_goal(state):

return state == goal\_state

def print\_state(state):

for row in state:

print(row)

print()

def dfs(state, visited, path, depth=0, max\_depth=20):

"""DFS recursive search with moves."""

if is\_goal(state):

print("✅ Goal reached in", depth, "moves")

print("Moves:", " -> ".join(path))

print()

return True

if depth >= max\_depth:

return False

visited.add(tuple(map(tuple, state)))

x, y = find\_blank(state)

for dx, dy, move in moves:

nx, ny = x + dx, y + dy

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

# Copy state

new\_state = [row[:] for row in state]

# Swap blank

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

if tuple(map(tuple, new\_state)) not in visited:

if dfs(new\_state, visited, path + [move], depth + 1, max\_depth):

return True

return False

# Example

initial\_state = [[1, 2, 3],

[4, 0, 6],

[7, 5, 8]]

print("Initial State:")

print\_state(initial\_state)

visited = set()

if not dfs(initial\_state, visited, []):

print("❌ Goal not found within depth limit.")

Programme 3:  
  
Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:  
  
A piece of paper with writing on it

AI-generated content may be incorrect.

Code:

import copy

goal\_state = [[1, 2, 3],

[8, 0, 4],

[7, 6, 5]]

def heuristic(state):

distance = 0

for i in range(3):

for j in range(3):

value = state[i][j]

if value != 0:

for x in range(3):

for y in range(3):

if goal\_state[x][y] == value:

distance += abs(i - x) + abs(j - y)

return distance

def find\_blank(state):

for i in range(3):

for j in range(3):

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

return i, j

def get\_neighbors(state):

neighbors = []

x, y = find\_blank(state)

moves = {

"Down": (1, 0),

"Up": (-1, 0),

"Right": (0, 1),

"Left": (0, -1)

}

for move, (dx, dy) in moves.items():

nx, ny = x + dx, y + dy

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

new\_state = copy.deepcopy(state)

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

neighbors.append((move, new\_state))

return neighbors

def hill\_climbing(initial\_state):

current = initial\_state

current\_h = heuristic(current)

steps = 0

path\_moves = ["Start"]

print("\nInitial State (h={}):".format(current\_h))

for row in current:

print(row)

while True:

neighbors = get\_neighbors(current)

if not neighbors:

print("\nNo more neighbors. Stopping.")

return current, steps, current\_h, path\_moves

print("\nStep {}: Generating neighbor states...".format(steps+1))

for move, child in neighbors:

print(f"\nMove: {move} (h={heuristic(child)})")

for row in child:

print(row)

neighbor\_move, neighbor = min(neighbors, key=lambda x: heuristic(x[1]))

neighbor\_h = heuristic(neighbor)

print("\nChosen move: {} → h={}".format(neighbor\_move, neighbor\_h))

for row in neighbor:

print(row)

if neighbor\_h >= current\_h:

print("\nNo better neighbor found. Stopping.")

return current, steps, current\_h, path\_moves

current, current\_h = neighbor, neighbor\_h

steps += 1

path\_moves.append(neighbor\_move)

if current\_h == 0:

print("\nGoal Reached!")

return current, steps, current\_h, path\_moves

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

initial\_state = [[2, 8, 3],

[1, 0, 4],

[7, 6, 5]]

solution, steps, h\_val, path\_moves = hill\_climbing(initial\_state)

print("\nFinal State:")

for row in solution:

print(row)

print(f"Steps taken: {steps}")

print(f"Final Heuristic value: {h\_val}")

print("\nFinal Path of Moves:")

print(" → ".join(path\_moves))

Programme 4:

Implement A\* Search Algorithm

Algorithm:  
  
A piece of paper with writing on it

AI-generated content may be incorrect.

Code:  
  
import heapq

GOAL\_STATE = ((1, 2, 3), (4, 5, 6), (7, 8, 0))

MOVES = [(-1, 0, "Up"), (1, 0, "Down"), (0, -1, "Left"), (0, 1, "Right")]

def manhattan\_distance(state):

distance = 0

for r in range(3):

for c in range(3):

value = state[r][c]

if value != 0:

goal\_r, goal\_c = (value - 1) // 3, (value - 1) % 3

distance += abs(r - goal\_r) + abs(c - goal\_c)

return distance

def get\_neighbors(state):

neighbors = []

for r in range(3):

for c in range(3):

if state[r][c] == 0:

blank\_pos = (r, c)

break

for move in MOVES:

new\_r, new\_c = blank\_pos[0] + move[0], blank\_pos[1] + move[1]

if 0 <= new\_r < 3 and 0 <= new\_c < 3:

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

new\_state[blank\_pos[0]][blank\_pos[1]], new\_state[new\_r][new\_c] = new\_state[new\_r][new\_c], new\_state[blank\_pos[0]][blank\_pos[1]]

neighbors.append((tuple(tuple(row) for row in new\_state), move[2]))

return neighbors

def a\_star(start\_state):

open\_list = []

heapq.heappush(open\_list, (0 + manhattan\_distance(start\_state), 0, start\_state, []))

visited = set()

visited.add(start\_state)

while open\_list:

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

if current\_state == GOAL\_STATE:

return path

for neighbor, move\_direction in get\_neighbors(current\_state):

if neighbor not in visited:

visited.add(neighbor)

new\_g = g + 1

new\_h = manhattan\_distance(neighbor)

new\_f = new\_g + new\_h

new\_path = path + [(neighbor, move\_direction)]

heapq.heappush(open\_list, (new\_f, new\_g, neighbor, new\_path))

return None

def print\_state(state):

for row in state:

print(row)

print()

start\_state = (

(1, 2, 3),

(4, 0, 6),

(7, 5, 8)

)

print("Initial State:")

print\_state(start\_state)

solution\_path = a\_star(start\_state)

if solution\_path:

print("Solution found! Path to goal:")

for step, move in solution\_path:

print(f"Move: {move}")

print\_state(step)

else:

print("No solution found.")

print("Final Goal State:")

print\_state(GOAL\_STATE)

Programme 5:

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

Algorithm:  
  
A hand holding a piece of paper

AI-generated content may be incorrect.

Code:  
  
# simple\_resolution.py

# Simple propositional entailment checker using resolution (KB |= query iff KB ∧ ¬query unsat)

# Supports operators: ~ (NOT), & (AND), | (OR), -> (IMPLIES), <-> (IFF)

# Clauses are frozensets of literal strings like "A" or "~A".

import re

import itertools

from typing import Set, FrozenSet, List, Tuple, Optional

# --------- Parser to AST ---------

TOKEN\_RE = r"\s\*([A-Za-z\_][A-Za-z\_0-9]\*|->|<->|[~\(\)&\|])\s\*"

def tokenize(s: str):

return [t for t in re.findall(TOKEN\_RE, s) if t.strip()]

class AST: pass

class Var(AST):

def \_\_init\_\_(self, name): self.name = name

def \_\_repr\_\_(self): return f"Var({self.name})"

class Not(AST):

def \_\_init\_\_(self,a): self.a=a

def \_\_repr\_\_(self): return f"Not({self.a})"

class And(AST):

def \_\_init\_\_(self,l,r): self.l=l; self.r=r

def \_\_repr\_\_(self): return f"And({self.l},{self.r})"

class Or(AST):

def \_\_init\_\_(self,l,r): self.l=l; self.r=r

def \_\_repr\_\_(self): return f"Or({self.l},{self.r})"

class Implies(AST):

def \_\_init\_\_(self,l,r): self.l=l; self.r=r

def \_\_repr\_\_(self): return f"Implies({self.l},{self.r})"

class Iff(AST):

def \_\_init\_\_(self,l,r): self.l=l; self.r=r

def \_\_repr\_\_(self): return f"Iff({self.l},{self.r})"

def parse\_formula(s: str) -> AST:

tokens = tokenize(s)

pos = 0

def parse\_expr():

return parse\_iff()

def parse\_iff():

nonlocal pos

left = parse\_imp()

while pos < len(tokens) and tokens[pos] == "<->":

pos += 1

right = parse\_imp()

left = Iff(left, right)

return left

def parse\_imp():

nonlocal pos

left = parse\_or()

if pos < len(tokens) and tokens[pos] == "->":

pos += 1

right = parse\_imp() # right-associative

return Implies(left, right)

return left

def parse\_or():

nonlocal pos

left = parse\_and()

while pos < len(tokens) and tokens[pos] == "|":

pos += 1

right = parse\_and()

left = Or(left, right)

return left

def parse\_and():

nonlocal pos

left = parse\_unary()

while pos < len(tokens) and tokens[pos] == "&":

pos += 1

right = parse\_unary()

left = And(left, right)

return left

def parse\_unary():

nonlocal pos

if pos < len(tokens) and tokens[pos] == "~":

pos += 1

return Not(parse\_unary())

return parse\_atom()

def parse\_atom():

nonlocal pos

if pos >= len(tokens):

raise ValueError("Unexpected end")

tok = tokens[pos]

if tok == "(":

pos += 1

node = parse\_expr()

if pos >= len(tokens) or tokens[pos] != ")":

raise ValueError("Missing )")

pos += 1

return node

if re.match(r"[A-Za-z\_][A-Za-z\_0-9]\*", tok):

pos += 1

return Var(tok)

raise ValueError("Unexpected token: "+tok)

ast = parse\_expr()

if pos != len(tokens):

raise ValueError("Extra tokens: " + " ".join(tokens[pos:]))

return ast

# --------- Convert to CNF (simple pipeline) ---------

def eliminate\_iff(ast):

if isinstance(ast, Iff):

a = eliminate\_iff(ast.l); b = eliminate\_iff(ast.r)

return And(Implies(a,b), Implies(b,a))

if isinstance(ast, Implies):

return Implies(eliminate\_iff(ast.l), eliminate\_iff(ast.r))

if isinstance(ast, And):

return And(eliminate\_iff(ast.l), eliminate\_iff(ast.r))

if isinstance(ast, Or):

return Or(eliminate\_iff(ast.l), eliminate\_iff(ast.r))

if isinstance(ast, Not):

return Not(eliminate\_iff(ast.a))

return ast

def eliminate\_implies(ast):

if isinstance(ast, Implies):

# A -> B == ~A | B

return Or(eliminate\_implies(Not(ast.l)), eliminate\_implies(ast.r))

if isinstance(ast, And):

return And(eliminate\_implies(ast.l), eliminate\_implies(ast.r))

if isinstance(ast, Or):

return Or(eliminate\_implies(ast.l), eliminate\_implies(ast.r))

if isinstance(ast, Not):

return Not(eliminate\_implies(ast.a))

return ast

def push\_not(ast):

# produce negation normal form: push NOTs to variables

if isinstance(ast, Not):

a = ast.a

if isinstance(a, Not):

return push\_not(a.a)

if isinstance(a, And):

return Or(push\_not(Not(a.l)), push\_not(Not(a.r)))

if isinstance(a, Or):

return And(push\_not(Not(a.l)), push\_not(Not(a.r)))

if isinstance(a, Var):

return ast

# should not happen for other forms if previous steps ok

return Not(push\_not(a))

if isinstance(ast, And):

return And(push\_not(ast.l), push\_not(ast.r))

if isinstance(ast, Or):

return Or(push\_not(ast.l), push\_not(ast.r))

return ast

def distribute\_or(ast):

# distribute OR over AND to get CNF

if isinstance(ast, Or):

A = distribute\_or(ast.l); B = distribute\_or(ast.r)

if isinstance(A, And):

return And(distribute\_or(Or(A.l,B)), distribute\_or(Or(A.r,B)))

if isinstance(B, And):

return And(distribute\_or(Or(A,B.l)), distribute\_or(Or(A,B.r)))

return Or(A,B)

if isinstance(ast, And):

return And(distribute\_or(ast.l), distribute\_or(ast.r))

return ast

def to\_cnf(ast):

return distribute\_or(push\_not(eliminate\_implies(eliminate\_iff(ast))))

# --------- CNF AST -> set of clauses (frozenset of literal strings) ---------

def literal\_to\_str(node) -> str:

if isinstance(node, Var):

return node.name

if isinstance(node, Not) and isinstance(node.a, Var):

return "~" + node.a.name

raise ValueError("Unexpected literal: "+repr(node))

def cnf\_to\_clauses(ast) -> Set[FrozenSet[str]]:

# ast is And of Ors of literals / or single literal / or Or

if isinstance(ast, And):

left = cnf\_to\_clauses(ast.l)

right = cnf\_to\_clauses(ast.r)

return left | right

# single clause: gather disjunction literals

def gather(node) -> Set[str]:

if isinstance(node, Or):

return gather(node.l) | gather(node.r)

return {literal\_to\_str(node)}

return {frozenset(gather(ast))}

# --------- Resolution ---------

def is\_complement(a: str, b: str) -> bool:

return a == ("~" + b) or b == ("~" + a)

def resolve(c1: FrozenSet[str], c2: FrozenSet[str]) -> Set[FrozenSet[str]]:

new = set()

for l1 in c1:

for l2 in c2:

if is\_complement(l1, l2):

merged = set(c1 | c2)

merged.discard(l1); merged.discard(l2)

# drop tautologies (A and ~A in same clause)

skip = any(is\_complement(x,y) for x in merged for y in merged if x!=y)

if not skip:

new.add(frozenset(merged))

return new

def resolution(kb\_clauses: Set[FrozenSet[str]], neg\_query\_clauses: Set[FrozenSet[str]], verbose=False) -> bool:

clauses = set(kb\_clauses) | set(neg\_query\_clauses)

if verbose:

print("Initial clauses:")

for c in clauses: print(" ", set(c))

new = set()

pairs\_checked = set()

while True:

pairs = [(ci,cj) for ci in clauses for cj in clauses if ci != cj]

added = False

for (ci,cj) in pairs:

if (ci,cj) in pairs\_checked: continue

pairs\_checked.add((ci,cj))

resolvents = resolve(ci,cj)

if frozenset() in resolvents:

if verbose: print("Derived empty clause from", set(ci), "and", set(cj))

return True

for r in resolvents:

if r not in clauses and r not in new:

new.add(r)

if verbose: print("Derived", set(r), "from", set(ci), "and", set(cj))

added = True

if not new:

return False

clauses |= new

new = set()

# --------- Main entailment function ---------

def entails(kb: List[str], query: str, verbose=False) -> bool:

# convert KB formulas to clauses

kb\_clauses = set()

for f in kb:

ast = parse\_formula(f)

cnf = to\_cnf(ast)

kb\_clauses |= cnf\_to\_clauses(cnf)

# convert negated query to clauses

neg\_query\_ast = Not(parse\_formula(query))

neg\_cnf = to\_cnf(neg\_query\_ast)

neg\_clauses = cnf\_to\_clauses(neg\_cnf)

return resolution(kb\_clauses, neg\_clauses, verbose=verbose)

# --------- Examples / usage ---------

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

examples = [

(["A -> B", "A"], "B"),

(["A -> B"], "A"),

(["(P & Q) -> R", "P", "Q"], "R"),

(["A", "~A"], "B"),

]

for kb, q in examples:

print("KB:", kb, "Query:", q)

res = entails(kb, q, verbose=True)

print("Entails?", res)

print("-"\*40)

Programme 7:  
  
Implement unification in first order logic

Algorithm:



Code:  
  
# fol\_unify.py

# A simple First-Order Logic unifier with parser and occurs-check.

import re

from typing import List, Dict, Optional, Union, Tuple

# ---------------------------

# Term classes

# ---------------------------

class Term:

def apply\_subst(self, subst: Dict["Variable", "Term"]) -> "Term":

raise NotImplementedError()

def vars(self) -> set:

"""Return set of Variable objects occurring in this term."""

raise NotImplementedError()

def \_\_repr\_\_(self):

return self.\_\_str\_\_()

class Variable(Term):

def \_\_init\_\_(self, name: str):

self.name = name

def apply\_subst(self, subst: Dict["Variable", Term]) -> Term:

# If variable is in substitution map, apply substitution (and then apply recursively)

for v in subst:

# use name equality for lookup convenience

if v.name == self.name:

return subst[v].apply\_subst(subst)

return self

def vars(self) -> set:

return {self}

def \_\_eq\_\_(self, other):

return isinstance(other, Variable) and self.name == other.name

def \_\_hash\_\_(self):

return hash(("Var", self.name))

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

return self.name

class Constant(Term):

def \_\_init\_\_(self, name: str):

self.name = name

def apply\_subst(self, subst: Dict[Variable, Term]) -> Term:

return self # constants unaffected

def vars(self) -> set:

return set()

def \_\_eq\_\_(self, other):

return isinstance(other, Constant) and self.name == other.name

def \_\_hash\_\_(self):

return hash(("Const", self.name))

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

return self.name

class Function(Term):

def \_\_init\_\_(self, name: str, args: List[Term]):

self.name = name

self.args = args

def apply\_subst(self, subst: Dict[Variable, Term]) -> Term:

return Function(self.name, [arg.apply\_subst(subst) for arg in self.args])

def vars(self) -> set:

s = set()

for a in self.args:

s |= a.vars()

return s

def \_\_eq\_\_(self, other):

return isinstance(other, Function) and self.name == other.name and self.args == other.args

def \_\_hash\_\_(self):

return hash(("Func", self.name, tuple(self.args)))

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

if len(self.args) == 0:

return self.name

return f"{self.name}({', '.join(map(str, self.args))})"

# ---------------------------

# Substitution utilities

# ---------------------------

Subst = Dict[Variable, Term]

def compose\_subst(s1: Subst, s2: Subst) -> Subst:

"""

Return composition s = s1 ∘ s2 meaning apply s2 then s1.

Implemented so that each term in s1 has s2 applied, and we keep s2 entries that aren't overridden.

"""

new = {}

# Apply s2 to all terms in s1

for v, t in s1.items():

new\_v = v

new\_t = t.apply\_subst(s2)

new[new\_v] = new\_t

# Add entries from s2 that are not in s1

for v, t in s2.items():

if v not in new:

new[v] = t

return new

def apply\_subst\_to\_term(term: Term, subst: Subst) -> Term:

return term.apply\_subst(subst)

# ---------------------------

# Occurs-check

# ---------------------------

def occurs\_check(var: Variable, term: Term, subst: Subst) -> bool:

"""

Check whether variable `var` occurs in `term` after applying current substitution `subst`.

Returns True if occurs -> then cannot bind var to term (would create a cyclic substitution).

"""

term\_applied = term.apply\_subst(subst)

return var in term\_applied.vars()

# ---------------------------

# Unification algorithm (Robinson's algorithm with occurs-check)

# ---------------------------

def unify(t1: Term, t2: Term, subst: Optional[Subst] = None) -> Optional[Subst]:

"""

Attempt to unify terms t1 and t2 given initial substitution subst.

Returns the most general unifier (a substitution dict Variable -> Term) or None on failure.

"""

if subst is None:

subst = {}

# Worklist of pairs

pairs: List[Tuple[Term, Term]] = [(t1, t2)]

current\_subst: Subst = dict(subst) # copy

while pairs:

s, t = pairs.pop(0)

# apply current substitution

s = s.apply\_subst(current\_subst)

t = t.apply\_subst(current\_subst)

# print("Debug pair:", s, t) # uncomment for step debugging

if s == t:

continue

if isinstance(s, Variable):

if occurs\_check(s, t, current\_subst):

return None # failure due to occurs-check

# add substitution s -> t

current\_subst = compose\_subst({s: t}, current\_subst)

continue

if isinstance(t, Variable):

if occurs\_check(t, s, current\_subst):

return None

current\_subst = compose\_subst({t: s}, current\_subst)

continue

# Both are functions or constants

if isinstance(s, Constant) and isinstance(t, Constant):

# different constants can't be unified

if s.name != t.name:

return None

else:

continue

if isinstance(s, Function) and isinstance(t, Function):

if s.name != t.name or len(s.args) != len(t.args):

return None

# push pairwise arguments

pairs = [(sa, ta) for sa, ta in zip(s.args, t.args)] + pairs

continue

# other cases not unifyable

return None

return current\_subst

# ---------------------------

# Simple parser for terms and predicates

# ---------------------------

TOKEN\_REGEX = r"\s\*([A-Za-z\_][A-Za-z\_0-9]\*|\(|\)|,)\s\*"

def tokenize(s: str) -> List[str]:

tokens = re.findall(TOKEN\_REGEX, s)

return [t for t in tokens if t.strip() != ""]

def parse\_term\_from\_tokens(tokens: List[str], pos: int = 0) -> Tuple[Term, int]:

"""

Parse a term starting at tokens[pos].

Returns (term, new\_pos)

Grammar (simple):

term ::= ID | ID '(' term (',' term)\* ')'

We decide variable vs constant/function name by identifier's first character:

- If starts with lowercase letter -> Variable

- Else -> Constant or Function (if followed by '(')

"""

if pos >= len(tokens):

raise ValueError("Unexpected end of tokens")

token = tokens[pos]

if re.match(r"[A-Za-z\_][A-Za-z\_0-9]\*", token) is None:

raise ValueError(f"Expected identifier at pos {pos}, found {token}")

name = token

pos += 1

# function / predicate with args?

if pos < len(tokens) and tokens[pos] == "(":

pos += 1 # skip '('

args = []

# handle empty-arg functions (rare)

if pos < len(tokens) and tokens[pos] == ")":

pos += 1

term = Function(name, [])

return term, pos

while True:

arg, pos = parse\_term\_from\_tokens(tokens, pos)

args.append(arg)

if pos >= len(tokens):

raise ValueError("Missing closing ')' in function")

if tokens[pos] == ",":

pos += 1

continue

elif tokens[pos] == ")":

pos += 1

break

else:

raise ValueError(f"Unexpected token {tokens[pos]} in args")

term = Function(name, args)

return term, pos

else:

# no args -> variable or constant

if name[0].islower():

return Variable(name), pos

else:

return Constant(name), pos

def parse\_term(s: str) -> Term:

tokens = tokenize(s)

term, pos = parse\_term\_from\_tokens(tokens, 0)

if pos != len(tokens):

raise ValueError("Extra tokens after parsing: " + " ".join(tokens[pos:]))

return term

# ---------------------------

# Utility to pretty-print substitution

# ---------------------------

def subst\_to\_str(subst: Optional[Subst]) -> str:

if subst is None:

return "Fail"

if not subst:

return "{}"

items = []

for v, t in subst.items():

items.append(f"{v} -> {t}")

return "{ " + ", ".join(items) + " }"

# ---------------------------

# Demonstration / Tests

# ---------------------------

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

examples = [

("Eats(x, Apple)", "Eats(Riya, y)"),

("p(f(a), g(Y))", "p(X, X)"), # should fail (example in prompt)

("Knows(John, x)", "Knows(x, Elisabeth)"), # should fail

("P(x, h(y))", "P(a, f(z))"), # should fail because h != f

("f(x, x)", "f(a, b)"), # fail: x must be both a and b

("parent(John, Mary)", "parent(John, Mary)"), # trivial unify

("q(X, g(Y))", "q(f(a), g(b))"), # Y->b, X->f(a)

("r(X)", "r(f(X))"), # occurs-check failure

]

for a, b in examples:

t1 = parse\_term(a)

t2 = parse\_term(b)

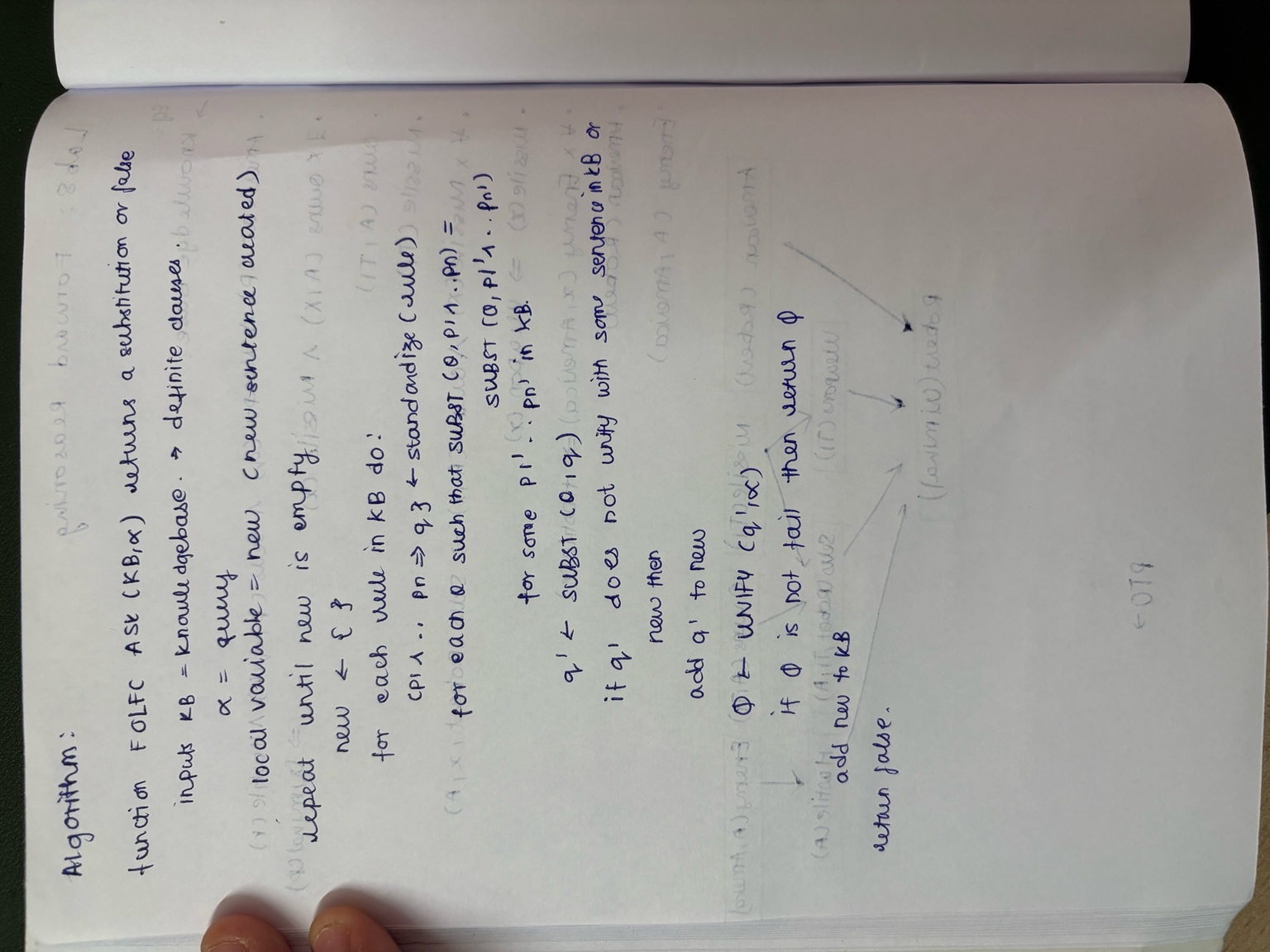
print("------------------------------------------------")

print("Unify:", a, " WITH ", b)

result = unify(t1, t2)

print("Result:", subst\_to\_str(result))

Programme 7:  
  
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:  
  


Code:  
  
# fol\_forward\_chaining.py

# Forward chaining for FOL definite clauses with unification and standardizing variables

import re

import itertools

import math

from typing import List, Dict, Tuple, Optional, Set

# -----------------------

# Term classes + parser

# -----------------------

class Term:

def apply(self, subst: Dict[str, "Term"]) -> "Term":

raise NotImplementedError()

def vars(self) -> Set[str]:

raise NotImplementedError()

class Var(Term):

def \_\_init\_\_(self, name: str):

self.name = name

def apply(self, subst):

if self.name in subst:

return subst[self.name].apply(subst)

return self

def vars(self):

return {self.name}

def \_\_eq\_\_(self, other):

return isinstance(other, Var) and self.name == other.name

def \_\_hash\_\_(self):

return hash(("Var", self.name))

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

return self.name

class Const(Term):

def \_\_init\_\_(self, name: str):

self.name = name

def apply(self, subst):

return self

def vars(self):

return set()

def \_\_eq\_\_(self, other):

return isinstance(other, Const) and self.name == other.name

def \_\_hash\_\_(self):

return hash(("Const", self.name))

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

return self.name

class Func(Term):

def \_\_init\_\_(self, name: str, args: List[Term]):

self.name = name

self.args = args

def apply(self, subst):

return Func(self.name, [a.apply(subst) for a in self.args])

def vars(self):

s = set()

for a in self.args:

s |= a.vars()

return s

def \_\_eq\_\_(self, other):

return isinstance(other, Func) and self.name == other.name and self.args == other.args

def \_\_hash\_\_(self):

return hash(("Func", self.name, tuple(self.args)))

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

if not self.args:

return self.name

return f"{self.name}({', '.join(map(str, self.args))})"

TOKEN\_RE = r"\s\*([A-Za-z\_][A-Za-z\_0-9]\*|\(|\)|,|\->)\s\*"

def tokenize(s: str):

toks = re.findall(TOKEN\_RE, s)

return [t for t in toks if t.strip()]

def parse\_term\_from\_tokens(tokens, pos=0) -> Tuple[Term,int]:

if pos >= len(tokens):

raise ValueError("unexpected end")

tok = tokens[pos]

if not re.match(r"[A-Za-z\_][A-Za-z\_0-9]\*", tok):

raise ValueError("expected identifier")

name = tok

pos += 1

if pos < len(tokens) and tokens[pos] == "(":

pos += 1

args = []

if pos < len(tokens) and tokens[pos] == ")":

pos += 1

return Func(name, []), pos

while True:

arg, pos = parse\_term\_from\_tokens(tokens, pos)

args.append(arg)

if pos >= len(tokens):

raise ValueError("missing ')'")

if tokens[pos] == ",":

pos += 1

continue

if tokens[pos] == ")":

pos += 1

break

raise ValueError("unexpected token " + tokens[pos])

return Func(name, args), pos

else:

# variable if starts with lowercase, constant/function symbol if uppercase

if name[0].islower():

return Var(name), pos

else:

return Const(name), pos

def parse\_atom(s: str) -> Func:

tokens = tokenize(s)

term, pos = parse\_term\_from\_tokens(tokens, 0)

if pos != len(tokens):

raise ValueError("extra tokens")

if not isinstance(term, Func):

# treat zero-arg predicate as Func(name, [])

return Func(str(term), [])

return term

# -----------------------

# Unification (with occurs-check)

# -----------------------

def occurs\_check(varname: str, term: Term, subst: Dict[str, Term]) -> bool:

t = term.apply(subst)

return varname in t.vars()

def unify\_terms(t1: Term, t2: Term, subst: Dict[str, Term]) -> Optional[Dict[str, Term]]:

# returns updated subst or None if failure

# apply current substitution first

t1 = t1.apply(subst)

t2 = t2.apply(subst)

if isinstance(t1, Var):

if t1.name == t2.\_\_dict\_\_.get("name", None) and isinstance(t2, Var):

return subst

if occurs\_check(t1.name, t2, subst):

return None

new = dict(subst)

new[t1.name] = t2

return new

if isinstance(t2, Var):

return unify\_terms(t2, t1, subst)

if isinstance(t1, Const) and isinstance(t2, Const):

if t1.name == t2.name:

return subst

return None

if isinstance(t1, Func) and isinstance(t2, Func):

if t1.name != t2.name or len(t1.args) != len(t2.args):

return None

s = dict(subst)

for a,b in zip(t1.args, t2.args):

s = unify\_terms(a, b, s)

if s is None:

return None

return s

return None

def unify\_atoms(a: Func, b: Func, subst: Dict[str, Term]) -> Optional[Dict[str, Term]]:

# predicate names and arity must match

if a.name != b.name or len(a.args) != len(b.args):

return None

s = dict(subst)

for x,y in zip(a.args, b.args):

s = unify\_terms(x, y, s)

if s is None:

return None

return s

# -----------------------

# Clauses / KB

# -----------------------

class Rule:

def \_\_init\_\_(self, antecedents: List[Func], consequent: Func):

self.antecedents = antecedents

self.consequent = consequent

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

if self.antecedents:

return f"{' & '.join(map(str,self.antecedents))} -> {self.consequent}"

else:

return str(self.consequent)

# Standardize variables apart by renaming variables in rule to fresh names

\_counter = 0

def fresh\_var\_name(base: str) -> str:

global \_counter

\_counter += 1

return f"{base}\_{\_counter}"

def standardize\_apart(rule: Rule) -> Rule:

# collect variables in the rule

varset = set()

for a in rule.antecedents + [rule.consequent]:

varset |= a.vars()

mapping = {}

for v in varset:

mapping[v] = Var(fresh\_var\_name(v))

# apply mapping to terms

def remap\_term(t: Term):

if isinstance(t, Var):

return mapping[t.name]

if isinstance(t, Const):

return t

return Func(t.name, [remap\_term(arg) for arg in t.args])

antecedents = [remap\_term(a) for a in rule.antecedents]

consequent = remap\_term(rule.consequent)

return Rule(antecedents, consequent)

# -----------------------

# Forward chaining algorithm (FOL-FC-ASK)

# -----------------------

def fol\_fc\_ask(kb\_facts: List[Func], kb\_rules: List[Rule], query: Func) -> Optional[Dict[str, Term]]:

"""

kb\_facts: list of ground atoms (Func) -- constants only expected

kb\_rules: list of Rule objects (with Vars)

query: atom to prove (may contain constants/vars)

Returns substitution (dict var->Term) that proves query or None.

"""

# KB as set for fast membership (use string repr for simplicity)

known = list(kb\_facts) # allow duplicates semantics via list

known\_set = set(str(f) for f in known)

print("Initial facts:")

for f in known:

print(" ", f)

iteration = 0

while True:

iteration += 1

new\_facts = []

# For each rule, standardized apart

for rule in kb\_rules:

std\_rule = standardize\_apart(rule)

n = len(std\_rule.antecedents)

# if no antecedents (a fact rule), just try consequent

if n == 0:

g = std\_rule.consequent

ground\_g = g.apply({}) # no subst

if str(ground\_g) not in known\_set:

new\_facts.append(ground\_g)

continue

# try to find substitutions that make each antecedent unify with some known fact

# we will try all combinations of known facts of length n (with repetition allowed)

for facts\_combo in itertools.product(known, repeat=n):

s = {}

failed = False

for pat, fact in zip(std\_rule.antecedents, facts\_combo):

s = unify\_atoms(pat, fact, s)

if s is None:

failed = True

break

if failed:

continue

# s is a substitution making all antecedents match these facts

# produce the instantiated consequent

instantiated\_consequent = std\_rule.consequent.apply(s)

if str(instantiated\_consequent) not in known\_set:

new\_facts.append(instantiated\_consequent)

# deduplicate new\_facts

added\_any = False

for nf in new\_facts:

if str(nf) not in known\_set:

known.append(nf)

known\_set.add(str(nf))

added\_any = True

print(f"[Iter {iteration}] Inferred: {nf}")

# check whether query is satisfied by this new fact (allow query with variables or constants)

s = {}

unify\_result = unify\_atoms(query, nf, s)

if unify\_result is not None:

print("Query unified with inferred fact:", nf)

return unify\_result

# Also check existing facts for query (in case present at start)

for f in known:

res = unify\_atoms(query, f, {})

if res is not None:

print("Query matches known fact:", f)

return res

if not added\_any:

print("No new facts inferred; stopping.")

return None

# -----------------------

# Build the KB for the Robert example

# -----------------------

def build\_robert\_kb():

facts = []

rules = []

# Given facts:

# Existential instantiation: we assume T1 is a fresh constant for the missile that A owns

# Owns(A, T1)

# Missile(T1)

facts.append(parse\_atom("Owns(A, T1)"))

facts.append(parse\_atom("Missile(T1)"))

# American(Robert)

facts.append(parse\_atom("American(Robert)"))

# Enemy(A, America)

facts.append(parse\_atom("Enemy(A, America)"))

# Rules:

# Missile(x) => Weapon(x)

r1 = Rule([parse\_atom("Missile(x)")], parse\_atom("Weapon(x)"))

rules.append(r1)

# All missiles were sold to country A by Robert:

# For all x: Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

r2 = Rule([parse\_atom("Missile(x)"), parse\_atom("Owns(A, x)")], parse\_atom("Sells(Robert, x, A)"))

rules.append(r2)

# Enemy(x, America) => Hostile(x)

r3 = Rule([parse\_atom("Enemy(x, America)")], parse\_atom("Hostile(x)"))

rules.append(r3)

# American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

r4 = Rule([

parse\_atom("American(p)"),

parse\_atom("Weapon(q)"),

parse\_atom("Sells(p, q, r)"),

parse\_atom("Hostile(r)")

], parse\_atom("Criminal(p)"))

rules.append(r4)

return facts, rules

# -----------------------

# Demo run

# -----------------------

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

kb\_facts, kb\_rules = build\_robert\_kb()

query = parse\_atom("Criminal(Robert)")

print("\nProving query:", query)

subst = fol\_fc\_ask(kb\_facts, kb\_rules, query)

print("\nFinal result:")

if subst is None:

print("Could NOT prove", query)

else:

print("Proved", query, "with substitution:", subst)

Programme 7:  
  
Implement Alpha-Beta Pruning

Algorithm:  
  
A paper with writing on it

AI-generated content may be incorrect.

Code:  
  
# auto\_tic\_tac\_toe\_alphabeta.py

# Automatic Tic-Tac-Toe demonstrations using Minimax + Alpha-Beta pruning

# No user input required: the script runs games automatically.

import math

import random

import time

EMPTY = " "

X = "X" # we'll treat X as +1 (maximizer)

O = "O" # O is -1 (minimizer)

WIN\_LINES = [

(0,1,2), (3,4,5), (6,7,8), # rows

(0,3,6), (1,4,7), (2,5,8), # cols

(0,4,8), (2,4,6) # diags

]

def new\_board():

return [EMPTY] \* 9

def print\_board(board):

for r in range(3):

row = board[3\*r:3\*r+3]

print(" " + " | ".join(row))

if r < 2:

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

print()

def available\_moves(board):

return [i for i, v in enumerate(board) if v == EMPTY]

def is\_winner(board, player):

return any(all(board[i] == player for i in line) for line in WIN\_LINES)

def is\_full(board):

return all(cell != EMPTY for cell in board)

def evaluate(board):

"""Terminal evaluation: +1 if X wins, -1 if O wins, 0 otherwise."""

if is\_winner(board, X):

return 1

if is\_winner(board, O):

return -1

return 0

def minimax\_alpha\_beta(board, player, alpha, beta, depth=0):

"""

General minimax with alpha-beta pruning for Tic-Tac-Toe.

player: the player to move now (X or O).

Returns (best\_score, best\_move\_index).

Depth is used only to prefer faster wins / slower losses.

"""

score = evaluate(board)

if score != 0 or is\_full(board):

# If terminal, prefer faster win / slower loss by factoring depth

if score == 1:

return 10 - depth, None # X wins -> positive

if score == -1:

return depth - 10, None # O wins -> negative

return 0, None # draw

if player == X:

max\_eval = -math.inf

best\_move = None

for move in available\_moves(board):

board[move] = X

eval\_score, \_ = minimax\_alpha\_beta(board, O, alpha, beta, depth+1)

board[move] = EMPTY

if eval\_score > max\_eval:

max\_eval = eval\_score

best\_move = move

alpha = max(alpha, eval\_score)

if beta <= alpha:

break # beta cut-off

return max\_eval, best\_move

else: # player == O (minimizer)

min\_eval = math.inf

best\_move = None

for move in available\_moves(board):

board[move] = O

eval\_score, \_ = minimax\_alpha\_beta(board, X, alpha, beta, depth+1)

board[move] = EMPTY

if eval\_score < min\_eval:

min\_eval = eval\_score

best\_move = move

beta = min(beta, eval\_score)

if beta <= alpha:

break # alpha cut-off

return min\_eval, best\_move

def best\_move\_alphabeta(board, player):

"""Return best move for player using alpha-beta pruning."""

\_, move = minimax\_alpha\_beta(board, player, -math.inf, math.inf, depth=0)

return move

def best\_move\_random(board):

"""Random move for weak opponent."""

moves = available\_moves(board)

return random.choice(moves) if moves else None

def play\_auto\_game(starting\_player, mode="AIvRandom", verbose=True, delay=0.3):

"""

Play a single automatic game.

starting\_player: X or O

mode: "AIvRandom" or "AIvAI"

verbose: print board and moves if True

delay: seconds to wait between moves (for readability)

Returns final board and result string.

"""

board = new\_board()

current = starting\_player

if verbose:

print("Starting automatic game. Mode:", mode)

print("Starting player:", current)

print\_board(board)

time.sleep(delay)

while True:

if mode == "AIvRandom":

if current == X:

move = best\_move\_alphabeta(board, X)

else: # O is random

move = best\_move\_random(board)

else: # "AIvAI"

move = best\_move\_alphabeta(board, current)

if move is None:

# No moves left (should be handled by terminal checks)

break

board[move] = current

if verbose:

print(f"{current} -> {move+1}")

print\_board(board)

time.sleep(delay)

if is\_winner(board, current):

if current == X:

result = "X wins"

else:

result = "O wins"

if verbose:

print("Result:", result)

return board, result

if is\_full(board):

if verbose:

print("Result: Draw")

return board, "Draw"

current = O if current == X else X

def demo():

random.seed(1) # deterministic randomness for repeatability

print("\n--- Demo 1: AI (X, alpha-beta) vs Random (O) ---\n")

board, result = play\_auto\_game(starting\_player=X, mode="AIvRandom", verbose=True, delay=0.15)

print("\n--- Demo 2: AI (X) vs AI (O) (both alpha-beta) ---\n")

board2, result2 = play\_auto\_game(starting\_player=X, mode="AIvAI", verbose=True, delay=0.15)

print("\nSummary:")

print("Demo 1 result:", result)

print("Demo 2 result:", result2)

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

demo()