Artificial Intelligence Lab Report

Submitted by

Tushar Tyagi(1BM22CS311)

Course: Artificial Intelligence
Course Code: 23CS5PCAIN
Sem & Section: 5F

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Table of contents

Program 1 - Tic Tac toe

Algorithm

PAGE NO :
DATE :

4/10/24

Implement TIC TAC TOE Game

Pseudocode.

```
function minmax( nodes, depth, isMaximizingPlayer)
        if node is a terminal state
        return evaluate(node)
if isMaximizingPlayer:
        best Value = -∞
        for each child in node:
                value = minimax(child, depth+1, false)
                bestValue = max(bestValue, value)
        return bestValue
else:
    bestvalue = +∞
    for each child in node:
            value : minimax(child, depth+1, true)
            bestValue : min(bestValue, value)
    return best Value
```

B 4/10/24

Code

```
import random
class TicTacToe:
def __init__(self):
self.board = []
    def create_board(self):
        for i in range(3):
            row = []
            for j in range(3):
                row.append('-')
            self.board.append(row)
    def get_random_first_player(self):
        return random.randint(0, 1)
    def fix_spot(self, row, col, player):
        self.board[row][col] = player
    def is_player_win(self, player):
        win = None
        n = len(self.board)
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[i][j] != player:
                    win = False
                    break



            if win:
                return win
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[j][i] != player:
                    win = False
                    break
            if win:
                return win
        win = True
        for i in range(n):
            if self.board[i][i] != player:
                win = False
                break
        if win:
```

```
        return win
    win = True
        for i in
    range(n):
        if self.board[i][n - 1 - i] != player:
            win = False
            break
    if win:
        return win
    return False
    for row in self.board:
        for item in row:
            if item == '-':
                return False
    return True
def is_board_filled(self):
    for row in self.board:
        for item in row:
            if item == '-':
                return False
    return True
def swap_player_turn(self, player):
    return 'X' if player == 'O' else 'O'
def show_board(self):
    for row in self.board:
        for item in row:
            print(item, end=" ")
        print()
def start(self):
    self.create_board()
    player = 'X' if self.get_random_first_player() == 1 else 'O'
    while True:
        print(f"Player {player} turn")
        self.show_board()
        row, col = list(
            map(int, input("Enter row and column numbers to fix spot: ").split()))
        print()
        self.fix_spot(row - 1, col - 1, player)
        if self.is_player_win(player):
            print(f"Player {player} wins the game!")
            break
```

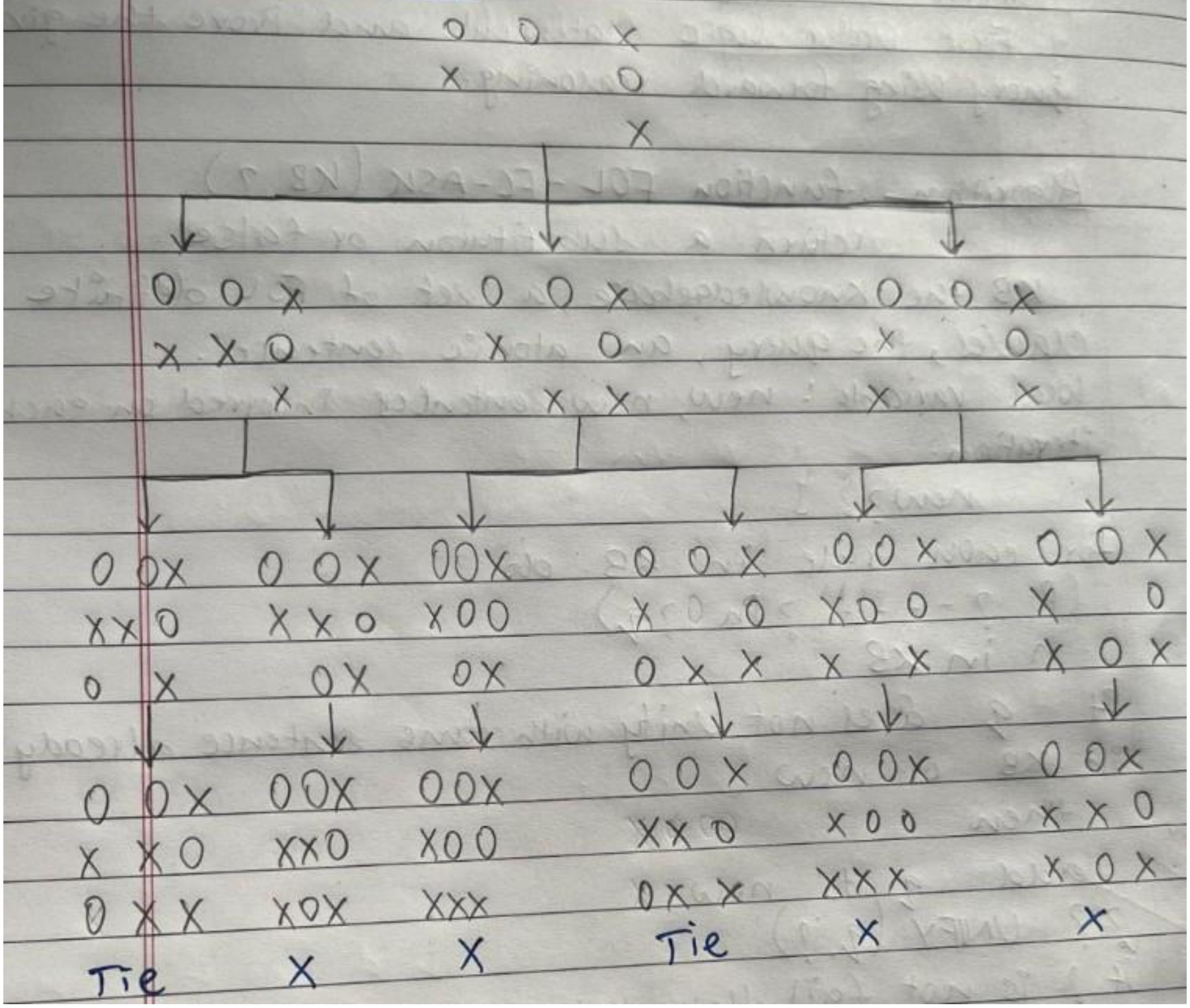
```
            if self.is_board_filled():
                print("Match Draw!")
                break
            player = self.swap_player_turn(player)
        print()
        self.show_board()
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Output Snapshot

```
Player O turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 0 3
Player X turn
- - -
- - -
- - O
Enter row and column numbers to fix spot: 1 2
Player O turn
- X - - - -
- - O
Enter row and column numbers to fix spot: 3 0
Player X turn
- X -
- - -
- - O
Enter row and column numbers to fix spot: 3 2
Player O turn
- X -
- - -
- X O
Enter row and column numbers to fix spot: 2 1
Player X turn
- X -
O - -
- X O
Enter row and column numbers to fix spot: 2 2
Player X wins the game!
```

State Space Tree

1) Tic Tac Toe.



Program 2 - 8 Puzzle Using BFS

Algorithm

PAGE NO :
DATE :

18/10/24

LAB-2

→ Implementing Vacuum cleaner agent

→ Algorithm:

- Initialize the agent's starting (A,B)
- Loop until all cells are clean:
  - Perceive the current cell
  - If the cell is dirty
    - clean the current cell
  - else:
    - check surrounding to check if they are dirty
    - move to the next dirty cell
    - If no dirty cells are perceived, Stop
- End

O/P ?

⇒ Solution to 8-Puzzle Problem

→ BFS:-

Algorithm:-

Let fringe be a list containing the initial state.
Loop
    if fringe is empty return failure
    node <- remove-first(fringe)
    if node is a goal
        then return the path from initial state to node, & add generated nodes to the fringe
end loop.

Code

```
import sys
import numpy as np
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
class StackFrontier:
    def __init__(self):
        self.frontier = []
    def add(self, node):
        self.frontier.append(node)
    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for node in self.frontier)
    def empty(self):
        return len(self.frontier) == 0
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node


class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None

    def neighbors(self, state):
        mat, (row, col) = state
```

```
        results = []
        if row > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row - 1][col]
            mat1[row - 1][col] = 0
            results.append(('up', [mat1, (row - 1, col)]))
        if col > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col - 1]
            mat1[row][col - 1] = 0
            results.append(('left', [mat1, (row, col - 1)]))
        if row < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row + 1][col]
            mat1[row + 1][col] = 0
            results.append(('down', [mat1, (row + 1, col)]))
        if col < 2:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col + 1]
            mat1[row][col + 1] = 0
            results.append(('right', [mat1, (row, col + 1)]))
        return results
    def print(self):
        solution = self.solution if self.solution is not None else None



        print("Start State:\n", self.start[0], "\n")
        print("Goal State:\n", self.goal[0], "\n")
        print("\nStates Explored: ", self.num_explored, "\n")
        print("Solution:\n ")
        for action, cell in zip(solution[0], solution[1]):
            print("action: ", action, "\n", cell[0], "\n")
        print("Goal Reached!!")
    def does_not_contain_state(self, state):
        for st in self.explored:
            if (st[0] == state[0]).all():
            return False
        return True
    def solve(self):
        self.num_explored = 0
        start = Node(state=self.start, parent=None, action=None)
```

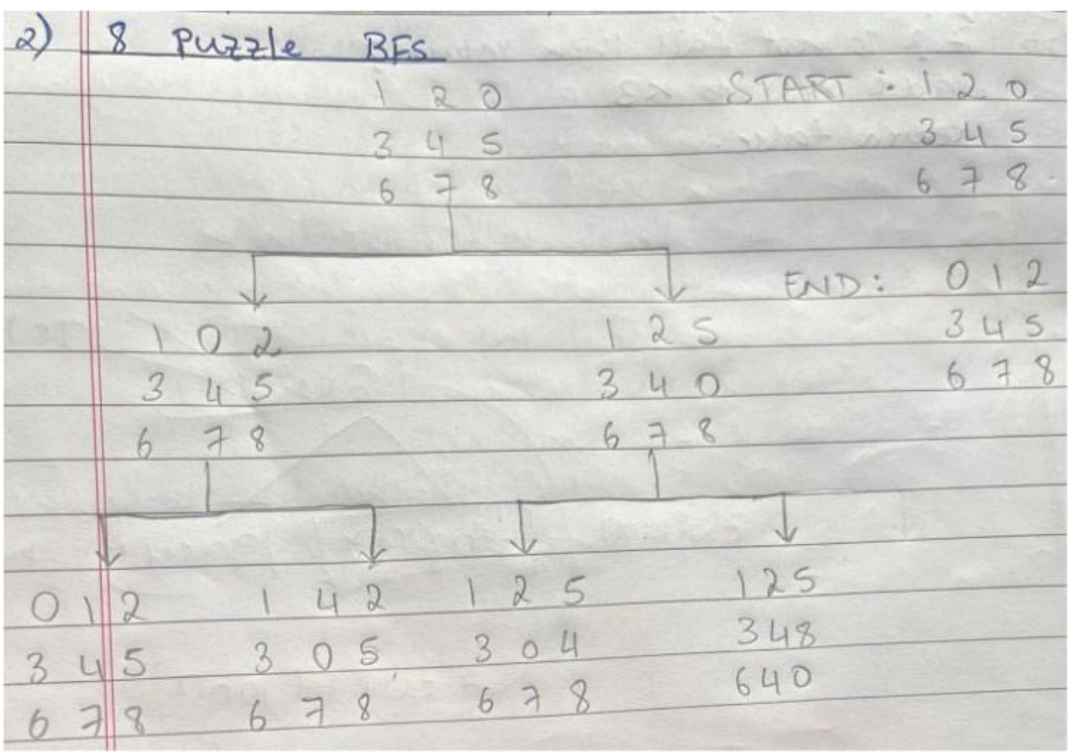
```
        frontier = QueueFrontier()
        frontier.add(start)
        self.explored = []
        while True:
            if frontier.empty():
                raise Exception("No solution")
            node = frontier.remove()
            self.num_explored += 1
            if (node.state[0] == self.goal[0]).all():
                actions = []
                cells = []
                while node.parent is not None:
                    actions.append(node.action)
                    cells.append(node.state)
                    node = node.parent
                actions.reverse()
                cells.reverse()
                self.solution = (actions, cells)
                return
            self.explored.append(node.state)
            for action, state in self.neighbors(node.state):
    if not frontier.contains_state(state) and self.does_not_contain_state(state): child =
                    Node(state=state, parent=node, action=action)
                    frontier.add(child)
start = np.array([[1, 2, 3], [8, 0, 4], [7, 6, 5]])


goal = np.array([[2, 8, 1], [0, 4, 3], [7, 6, 5]])
startIndex = (1, 1)
goalIndex = (1, 0)
p = Puzzle(start, startIndex, goal, goalIndex)
p.solve() p.print()
```

Output Snapshot

```
Start State:
 [[1 2 3]
 [8 0 4]
 [7 6 5]]

Goal State:
 [[2 8 1]
 [0 4 3]
 [7 6 5]]

States Explored:  358

Solution:

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

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

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

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

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

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

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

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

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

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

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

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

Goal Reached!!
```

State Space Tree



Program 3 - 8 puzzle using DFS

Algorithm

PAGE NO :
DATE :

→ DFS:-

Algorithm :-

Let fringe be a set containing the initial state.

loop:

if fringe is empty return failure

node <= remove.first (fringe)

if node is a goal

then return the path from initial state to node

else

generate all successors

→ State space tree

Initial

1 2 3
4 5 6
0 7 8

Final

1 2 3
4 5 6
7 8 0

Initial state

1 2 3
4 5 6
0 7 8

1 2 3
0 5 6
4 7 3

1 2 3
4 5 6
7 0 8

0 2 3
1 5 6
4 7 8

1 2 3
5 0 6
4 7 8

1 2 3
4 5 6
7 8 0

1 2 3
4 0 6
7 5 8

Code

```
from copy import deepcopy

class Puzzle8:
    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.visited_states = set()

    def is_goal(self, state):
        """Check if the current state is the goal state."""
        return state == self.goal_state

    def find_empty(self, state):
        """Find the position of the empty tile (0) in the puzzle."""
        for i, row in enumerate(state):
            for j, val in enumerate(row):
                if val == 0:
                    return i, j

    def generate_moves(self, state):
        """Generate all possible moves from the current state."""
        empty_row, empty_col = self.find_empty(state)
        moves = []
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

        for dr, dc in directions:
            new_row, new_col = empty_row + dr, empty_col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_state = deepcopy(state)
                new_state[empty_row][empty_col], new_state[new_row][new_col] = (
                    new_state[new_row][new_col],
                    new_state[empty_row][empty_col],
                )
                moves.append(new_state)

        return moves

    def dfs(self, state, depth, path):
        """Depth-First Search to solve the 8-puzzle problem."""
        if self.is_goal(state):
            return path

        # Mark the current state as visited
        self.visited_states.add(self.state_to_tuple(state))

        # Explore all possible moves
        for move in self.generate_moves(state):
            move_tuple = self.state_to_tuple(move)
            if move_tuple not in self.visited_states:
                result = self.dfs(move, depth + 1, path + [move])
```

```
                if result:
                    return result

            return None

    def solve(self):
        """Solve the puzzle using DFS."""
        return self.dfs(self.initial_state, 0, [self.initial_state])

    def state_to_tuple(self, state):
        """Convert the state to a tuple for hashable representation."""
        return tuple(tuple(row) for row in state)


# Example Usage
if __name__ == "__main__":
    # Initial State (0 is the empty tile)
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8],
    ]

    # Goal State
    goal_state = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0],
    ]

    # Solve the puzzle
    puzzle = Puzzle8(initial_state, goal_state)
    solution = puzzle.solve()

    if solution:
        print("Solution found!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            for row in state:
                print(row)
            print()
    else:
        print("No solution exists.")
```

Output Snapshot

```
Solution found!
Step 0:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

Program 04 - A* Algorithm

PAGE NO :
DATE :

LAB - 3

18| 25|10|24

* A* algorithm

function A* search (problem) returns a solution / failure.

node ← a node n with n state:
problem. initial state

frontier ← a priority queue ordered by ascending g+h only element

loop do
  if empty > (frontier) then return failure
  n ← pop(frontier)
  if problem. goalTest (n. state) then return solution(n)
  for each action a in problem,
    actions (n. state) do
    n ← childNode (problem, n, a)
    insert (n, g(n) + h(n'), frontier)

op

1
4
7

Ma
→
→ T

Code

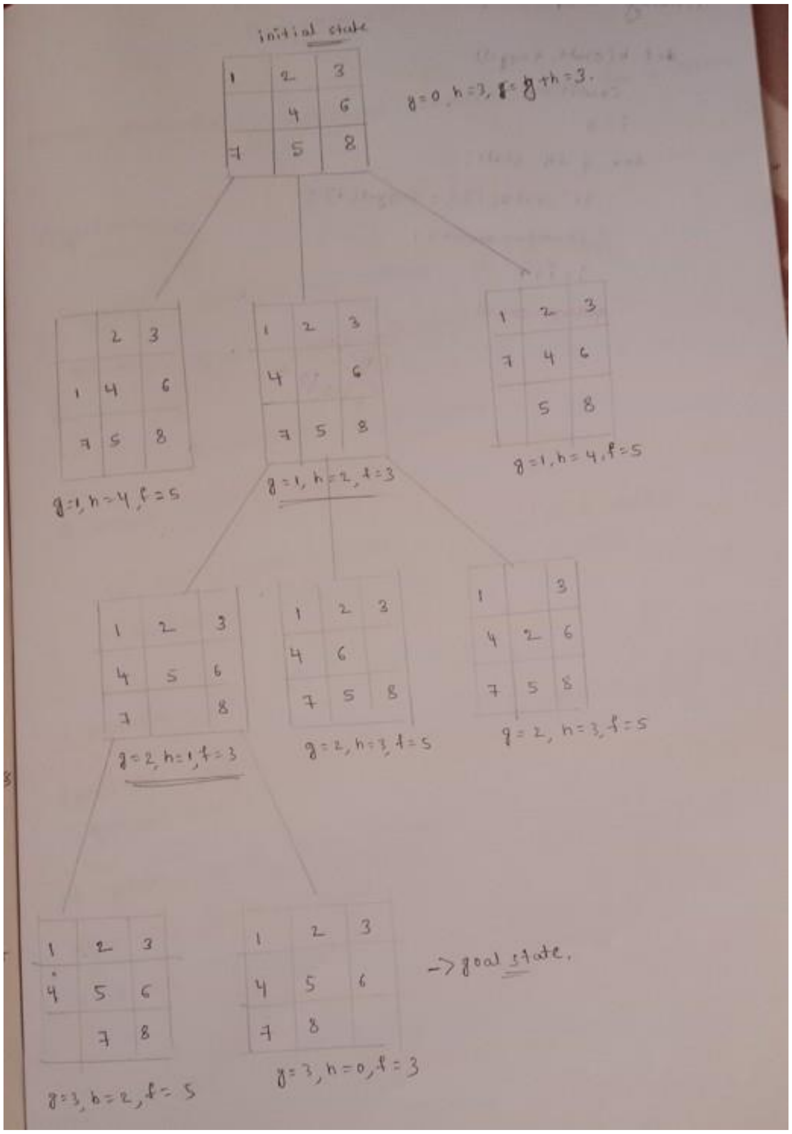
```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
f"""
 {state[0]} {state[1]} {state[2]}
 {state[3]} {state[4]} {state[5]}
 {state[6]} {state[7]} {state[8]}
 “””
    )
def h(state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target[i]:
            count = count+1
    return count
def astar(state, target):
    states = [src]
    g = 0
    visited_states = []
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.append(state)
            print_b(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(
                state, visited_states) if move not in moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i]
                 for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")
def possible_moves(state, visited_state):
```

```
    b = state.index(-1)
    d = []
    if b - 3 in range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if move not in visited_state]
def gen(state, m, b):
    temp = state.copy()
    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]
    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    return temp
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5,6, 7, 8,-1]
astar(src, target)
```

Output Snapshot

```
Enter the start state matrix

1 0 1 0
1 0 0 1
1 1 1 1
Enter the goal state matrix

1 1 0 1
1 0 0 1
1 1 1 0
|
  |
 \'/

1 0 1 0
1 0 0 1
1 1 1 1
```

State Space Tree



Program 5 - Vacuum Cleaner

Algorithm

PAGE NO :
DATE :

18/10/24

LAB-2

→ Implementing Vacuum cleaner agent

→ Algorithm:

- Initialize the agent's starting (A, B)
- Loop until all cells are clean:
  - Perceive the current cell
  - If the cell is dirty
    - clean the current cell

    else:
    - check surrounding to check if they are dirty
    - move to the next dirty cell
    - If no dirty cells are perceived, Stop
- End

O/P ?

⇒ Solution to 8-Puzzle Problem

→ BFS:-

Algorithm :-

Let fringe be a list containing the initial state.
Loop
  if fringe is empty return failure
  Node <= remove - first (fringe)
  if Node is a goal
    then return the path from initial state to node, & add generated nodes to the fringe
end loop.

Code

```
def vacuum_world():

    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input+ " : ")
    status_input_complement = input("Enter status of other room : ")
    print("Initial Location Condition {A : " + str(status_input_complement) + ", B : " +
str(status_input) + " }" )

    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'



            cost += 1 #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))
                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))
```

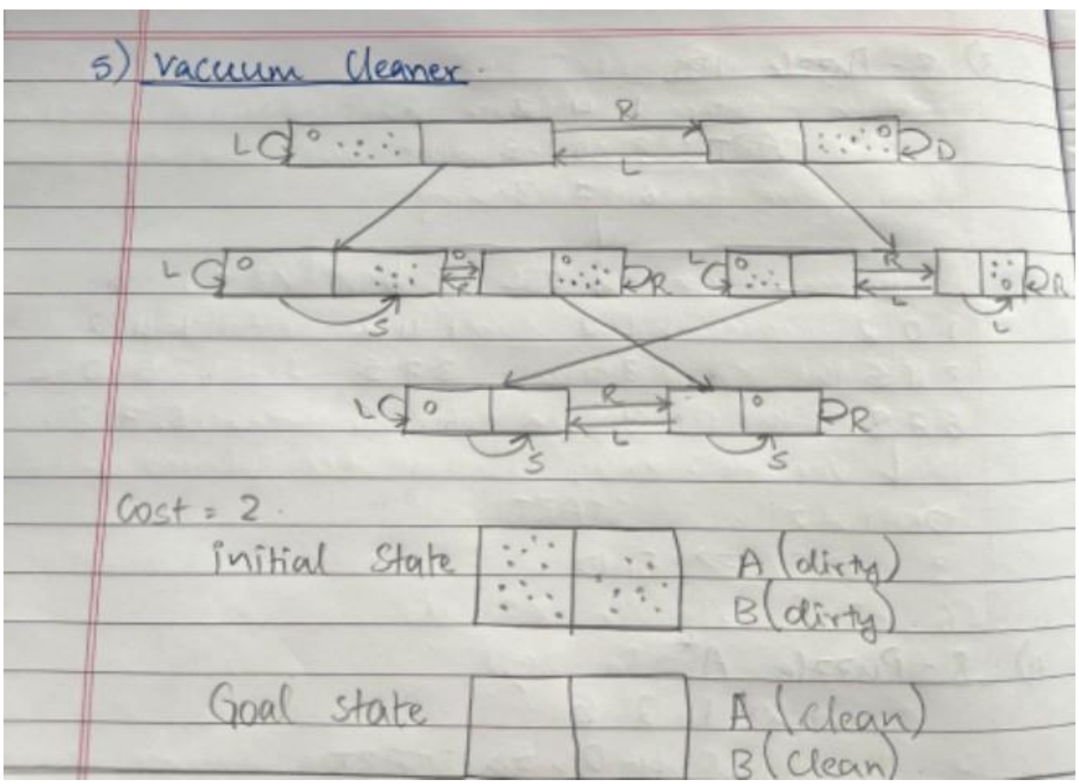
```
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING " + str(cost))



        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
            print("Location A has been Cleaned.")

    else:
        print(cost)
        print("Location B is already clean.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
```

```
    else:
        print("No action " + str(cost))
        print("Location A is already clean.")

print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))
vacuum_world()
```

Output Snapshot

```
Enter Location of Vacuum: A
Enter status of A : 0
Enter status of other room : 1
Initial Location Condition {A : 1, B : 0 }
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

<u>**State Space Tree**</u>



Program-06 Hill Climbing

Algorithm

PAGE NO :
DATE :

LAB-4

18/11 - 8/11/24

→ Implementing Hill climbing search algorithm to solve N queens problem.

Algorithm

* function HILLCLIMBING(problem) returns a state i.e. local maximum

current ← MAKE-NODE (problem. INITIAL-STATE)

loop do

neighbor ← a highest valued successor of current

if neighbour. VALUE ≤ current. VALUE then return current state

Execute

current ← ~~neigh~~ neighbour

→ State space tree

Initial state

			Q
	Q		
		Q	
Q			

Initial state.

		Q	
Q			
			Q
	Q		

Goal State

State:	Score:
3120	2
1320	1 → select
2130	1
0123	6
3210	6
3021	1
3102	1

... ...

Code

```
import random

class NQueensHillClimbing:
    def __init__(self, N):
        self.N = N

    def calculate_heuristic(self, board):
        """Calculate the number of attacking pairs of queens."""
        attacks = 0
        for i in range(self.N):
            for j in range(i + 1, self.N):
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    attacks += 1
        return attacks

    def get_neighbors(self, board):
        """Generate all possible neighbors by moving each queen to a new row."""
        neighbors = []
        for col in range(self.N):
            for row in range(self.N):
                if board[col] != row:
                    new_board = board[:]
                    new_board[col] = row
                    neighbors.append(new_board)
        return neighbors

    def hill_climbing(self, initial_board):
        """Perform the hill climbing algorithm to solve the N-Queens problem."""
        current_board = initial_board
        current_heuristic = self.calculate_heuristic(current_board)

        while True:
            neighbors = self.get_neighbors(current_board)
            neighbors_heuristics = [self.calculate_heuristic(neighbor) for neighbor in neighbors]
            min_heuristic = min(neighbors_heuristics)

            # If the heuristic cannot be improved, stop
            if min_heuristic >= current_heuristic:
                break
```

```
            # Move to the neighbor with the best heuristic
            best_index = neighbors_heuristics.index(min_heuristic)
            current_board = neighbors[best_index]
            current_heuristic = min_heuristic

        return current_board, current_heuristic

    def solve(self, max_restarts=100):
        """Solve the N-Queens problem using Random Restart Hill Climbing."""
        for restart in range(max_restarts):
            # Start with a random initial state
            initial_board = [random.randint(0, self.N - 1) for _ in range(self.N)]
            solution, heuristic = self.hill_climbing(initial_board)

            if heuristic == 0:
                return solution  # Found a solution

        return None  # No solution found after max_restarts


# Example Usage
if __name__ == "__main__":
    N = 8  # Size of the chessboard
    n_queens = NQueensHillClimbing(N)
    solution = n_queens.solve(max_restarts=1000)  # Try up to 1000 random restarts

    if solution:
        print("Solution found:")
        print(solution)

        # Display the board
        for row in range(N):
            line = ""
            for col in range(N):
                if solution[col] == row:
                    line += "Q "
                else:
                    line += ". "
            print(line)
    else:
        print("No solution found, even after random restarts.")
```

Output Snapshot

```
Solution found:
[7, 3, 0, 2, 5, 1, 6, 4]
. . Q . . . . .
. . . . . Q . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
```

Program 7: Simulated Annealing

Algorithm

15/11/24

→ Implement N-Queens using Simulated Annealing

```
current ← initial state
T ← a large positive value
while T > 0 do
    next ← a random neighbour of current
    ΔE ← current.cost − next.cost
    if ΔE > 0 then
        current ← next
    else
        current ← next with probability
    end if
    decrease T
end while
return current
```

O/P

```
Enter the value of n: 4
Final solution: [1,3,0,2]
No. of conflicts: 0
Total iterations: 7
Solution found!
```

15/11/24

Code

```
import random
import math

class NQueensSimulatedAnnealing:
    def __init__(self, N):
        self.N = N

    def calculate_heuristic(self, board):
        """Calculate the number of attacking pairs of queens."""
        attacks = 0
        for i in range(self.N):
            for j in range(i + 1, self.N):
                if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                    attacks += 1
        return attacks

    def get_random_neighbor(self, board):
        """Generate a random neighbor by moving one queen to a different row."""
        neighbor = board[:]
        col = random.randint(0, self.N - 1)  # Pick a random column
        row = random.randint(0, self.N - 1)  # Pick a random row
        while neighbor[col] == row:
            row = random.randint(0, self.N - 1)  # Ensure the new row is different
        neighbor[col] = row
        return neighbor

    def simulated_annealing(self, initial_board, max_steps=1000, initial_temp=100, cooling_rate=0.99):
        """Solve the N-Queens problem using Simulated Annealing."""
        current_board = initial_board
        current_heuristic = self.calculate_heuristic(current_board)
        temperature = initial_temp

        for step in range(max_steps):
            if current_heuristic == 0:
                return current_board  # Solution found

            # Generate a random neighbor
            neighbor = self.get_random_neighbor(current_board)
            neighbor_heuristic = self.calculate_heuristic(neighbor)

            # Calculate the change in heuristic
            delta_heuristic = neighbor_heuristic - current_heuristic

            # Decide whether to accept the neighbor
            if delta_heuristic < 0 or random.uniform(0, 1) < math.exp(-delta_heuristic / temperature):
                current_board = neighbor
                current_heuristic = neighbor_heuristic

            # Cool down the temperature
            temperature *= cooling_rate
```

```
        return None  # No solution found within the maximum steps

    def solve(self):
        """Solve the N-Queens problem using Simulated Annealing."""
        initial_board = [random.randint(0, self.N - 1) for _ in range(self.N)]  # Random initial state
        return self.simulated_annealing(initial_board)


# Example Usage
if __name__ == "__main__":
    N = 8  # Size of the chessboard
    n_queens = NQueensSimulatedAnnealing(N)
    solution = n_queens.solve()

    if solution:
        print("Solution found:")
        print(solution)

        # Display the board
        for row in range(N):
            line = ""
            for col in range(N):
                if solution[col] == row:
                    line += "Q "
                else:
                    line += ". "
            print(line)
    else:
        print("No solution found.")



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

OUTPUT

```
Solution found:
[2, 5, 1, 6, 0, 3, 7, 4]
. . . . Q . . .
. . Q . . . . .
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . . Q .
```

Program-08- Unification in FOL

Algorithm

PAGE NO :
DATE : 22/11/24

LAB-6

⇒ Implement unification in first order logic

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

(a) If Ψ_1 or Ψ_2 are identical, then return NIL.

(b) Else if Ψ_1 is a variable

  (a) then if Ψ_1 occurs in Ψ_2, then return FAILURE

  (b) Else return $\{(\Psi_2 / \Psi_1)\}$

(c) Else if Ψ_2 is a variable,

  (a) If Ψ_2 occurs in Ψ_1 then return FAILURE

  (b) Else return $\{(\Psi_1 / \Psi_2)\}$

(d) Else return FAILURE

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE

Step 3: If Ψ_1 & Ψ_2 have a diff no. of arguments, then return FAILURE

Step 4: Set Substitution Set (SUBST) to NIL

Step 5: For i=1 to the no. of elements in Ψ_1

(a) Call unify function with the i^{th} element of Ψ_1 & i^{th} element of Ψ_2, & put the result into S

(b) If S = failure then return failure

(c) If S ≠ ~~NIL~~ recall then do

  (a) Apply S to the remainder of both L1 & L2

  (b) SUBST = APPEND(S, SUBST).

Step 6: Return SUBST

Code

```
def is_variable(term):
    """Check if a term is a variable."""
    return isinstance(term, str) and term.islower()

def is_constant(term):
    """Check if a term is a constant."""
    return isinstance(term, str) and term.isupper()

def unify(term1, term2, subst=None):
    """
    Unify two terms.
    Args:
        term1: The first term (variable, constant, or function).
        term2: The second term (variable, constant, or function).
        subst: Current set of substitutions (dictionary).
    Returns:
        A substitution dictionary if unification is successful, otherwise None.
    """
    if subst is None:
        subst = {}

    if term1 == term2:  # If terms are identical
        return subst

    if is_variable(term1):  # If term1 is a variable
        return unify_variable(term1, term2, subst)

    if is_variable(term2):  # If term2 is a variable
        return unify_variable(term2, term1, subst)

    if isinstance(term1, tuple) and isinstance(term2, tuple):
        # If terms are functions, unify their name and arguments
        if term1[0] != term2[0] or len(term1[1]) != len(term2[1]):
            return None  # Function names or argument lengths differ
        for arg1, arg2 in zip(term1[1], term2[1]):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None
        return subst

    return None  # Terms cannot be unified


def unify_variable(var, term, subst):
    """
    Unify a variable with a term.
    Args:
```

```
        var: The variable (string).
        term: The term to unify with (variable, constant, or function).
        subst: Current set of substitutions (dictionary).
    Returns:
        Updated substitution dictionary or None.
    """
    if var in subst:  # Variable already substituted
        return unify(subst[var], term, subst)

    if occurs_check(var, term, subst):  # Prevent infinite loops
        return None

    subst[var] = term
    return subst


def occurs_check(var, term, subst):
    """
    Check if a variable occurs in a term (to prevent infinite loops).
    Args:
        var: The variable (string).
        term: The term to check against.
        subst: Current set of substitutions (dictionary).
    Returns:
        True if var occurs in term, False otherwise.
    """
    if var == term:
        return True
    if isinstance(term, tuple):  # If term is a function, check its arguments
        return any(occurs_check(var, arg, subst) for arg in term[1])
    if var in subst and occurs_check(var, subst[var], subst):
        return True
    return False


def apply_substitution(term, subst):
    """
    Apply a substitution to a term.
    Args:
        term: The term to substitute (variable, constant, or function).
        subst: The substitution dictionary.
    Returns:
        The term after applying the substitution.
    """
    if is_variable(term) and term in subst:
        return apply_substitution(subst[term], subst)
    if isinstance(term, tuple):  # If the term is a function, apply substitution to its arguments
        return (term[0], [apply_substitution(arg, subst) for arg in term[1]])
    return term  # Return the term as-is for constants or unbound variables


# Example Usage
```

```
if __name__ == "__main__":
    # Example terms:
    term1 = ("f", ["x", "y"])  # f(x, y)
    term2 = ("f", ["a", "b"])  # f(a, b)

    # Perform unification
    result = unify(term1, term2)
    if result:
        print("Unification successful! Substitution:")
        print(result)

        # Apply substitution to the original terms
        term1_substituted = apply_substitution(term1, result)
        term2_substituted = apply_substitution(term2, result)

        print("\nTerms after substitution:")
        print(f"Term 1: {term1_substituted}")
        print(f"Term 2: {term2_substituted}")
    else:
        print("Unification failed.")
else:
      print("Knowledge Base doesn't entail the query, no empty set produced after resolution") clauses
      = input('Enter the clauses ').split()
      query = input('Enter the query: ')
      checkResolution(clauses, query)
```

Output Snapshot

```
Unification successful! Substitution:
{'x': 'a', 'y': 'b'}

Terms after substitution:
Term 1: ('f', ['a', 'b'])
Term 2: ('f', ['a', 'b'])
```

Program-09 Forward Reasoning

Algorithm

PAGE NO :
DATE :

LAB-7

29/11/24

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

⇒ Algorithm :

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

inputs : KB, the knowledge base, a set of first-order definite clauses

α, the query, an atomic sentence

local variables: new, the new sentences inferred to each iteration

```
repeat until new is empty
    new ← { }
    for each rule in KB do
        (p1 ∧ ... ∧ pn ⇒ q) ← standardized-VARIABLES(rule)
        for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p1' ∧ ... ∧ pn')
            for some p1' ... pn' in KB
        q' ← SUBST (θ, q)
        if q' does not unify with some sentence already in KB or new then
            add q' to new
            φ ← UNIFY(q', α)
            if φ is not fail then return φ
    add new to KB
return false
```

Code

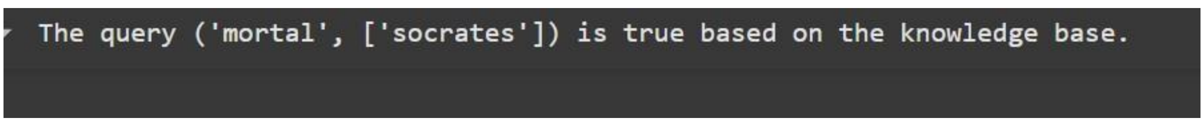
```
def is_variable(term):
    """Check if a term is a variable."""
    return isinstance(term, str) and term.islower()

def apply_substitution(term, subst):
    """Apply a substitution to a term."""
    if is_variable(term) and term in subst:
        return apply_substitution(subst[term], subst)
    if isinstance(term, tuple):  # If term is a function, apply substitution to arguments
        return (term[0], [apply_substitution(arg, subst) for arg in term[1]])
    return term  # Return the term as-is for constants or unbound variables

def unify(term1, term2, subst=None):
    """Unify two terms."""
    if subst is None:
        subst = {}
    if term1 == term2:
        return subst
    if is_variable(term1):
        return unify_variable(term1, term2, subst)
    if is_variable(term2):
        return unify_variable(term2, term1, subst)
    if isinstance(term1, tuple) and isinstance(term2, tuple):
        if term1[0] != term2[0] or len(term1[1]) != len(term2[1]):
            return None
        for arg1, arg2 in zip(term1[1], term2[1]):
            subst = unify(arg1, arg2, subst)
            if subst is None:
                return None
        return subst
    return None

def unify_variable(var, term, subst):
    """Unify a variable with a term."""
    if var in subst:
```

```
        return unify(subst[var], term, subst)
    if occurs_check(var, term, subst):
        return None
    subst[var] = term
    return subst

def occurs_check(var, term, subst):
    """Check if a variable occurs in a term."""
    if var == term:
        return True
    if isinstance(term, tuple):
        return any(occurs_check(var, arg, subst) for arg in term[1])
    if var in subst and occurs_check(var, subst[var], subst):
        return True
    return False

def forward_reasoning(kb, query):
    """
    Perform forward reasoning on the knowledge base (KB) to prove the query.
    Args:
        kb: The knowledge base, a list of first-order logic rules or facts.
        query: The goal to prove.
    Returns:
        True if the query can be proved, otherwise False.
    """
    known_facts = set()
    new_facts = True

    while new_facts:
        new_facts = False

        for rule in kb:
            if isinstance(rule, tuple) and rule[0] == "implies":  # Implication rule
                conditions, conclusion = rule[1], rule[2]

                substitutions = [{}]
```

```
            for condition in conditions:
                next_substitutions = []
                for fact in known_facts:
                    subst = unify(condition, fact)
                    if subst is not None:
                        next_substitutions.append(subst)
                substitutions = [
                    {**s1, **s2} for s1 in substitutions for s2 in next_substitutions
                ]

            for subst in substitutions:
                derived_fact = apply_substitution(conclusion, subst)
                if derived_fact not in known_facts:
                    known_facts.add(derived_fact)
                    new_facts = True

        else:  # It's a fact
            if rule not in known_facts:
                known_facts.add(rule)
                new_facts = True

    # Check if the query is in the known facts
    for fact in known_facts:
        if unify(fact, query) is not None:
            return True

return False



# Example Usage
if __name__ == "__main__":
    # Knowledge Base
    kb = [
        ("implies", [("human", ["x"])], ("mortal", ["x"])),  # human(x) -> mortal(x)
        ("human", ["socrates"]),  # human(socrates)
    ]
```

```
# Query
query = ("mortal", ["socrates"])  # Is Socrates mortal?

# Perform forward reasoning
result = forward_reasoning(kb, query)

if result:
    print(f"The query {query} is true based on the knowledge base.")
else:
    print(f"The query {query} cannot be proved from the knowledge base.")
```

Output Snapshot



Program-10: 8-Puzzle IDS

Algorithm:

PAGE NO :
DATE :

(9) Implement 8 puzzle using IDS.

Algo:-

depth_first_ids (node, soln)
start-path (node, goal-node, soln)
goal (goal-node)
path (node1, Node2, [Node3])
path (firstNode, lastNode, [LastNode [path]])
path (firstNode, lastNode, Path)
path (LastSubone, lastNode)
path (lastNode, Path).

State Space Tree:-

```
              1 4 2
              3 0 5
              6 7 8

→ 1 0 2       1 4 2       1 4 2       1 4 2
  3 4 5       0 3 5       3 7 5       3 5 0
  6 7 8       6 7 8       6 0 8       6 7 8
    ↓

  0 1 2
  3 4 5   => end.
  6 7 8
```

Code:

```
from collections import deque

def iterative_deepening_search(start_state, goal_state):
    def dfs(state, depth, path, visited):
        if state == goal_state:
            return path
        if depth == 0:
            return None

        visited.add(state)

        for next_state, move in get_successors(state):
            if next_state not in visited:
                result = dfs(next_state, depth - 1, path + [move], visited)
                if result is not None:
                    return result

        visited.remove(state)
        return None

    depth = 0
    while True:
        visited = set()
        result = dfs(start_state, depth, [], visited)
        if result is not None:
            return result
        depth += 1

def get_successors(state):
    """
    Generate successors for the given 8-puzzle state.
    Each successor is a tuple (next_state, move), where:
    - next_state: the state after the move
    - move: the move made to reach the next state (e.g., 'up', 'down', 'left', 'right')
    """
    successors = []
    state_list = list(state)
    zero_index = state_list.index(0)  # Find the blank tile (represented by 0)

    # Define possible moves
    moves = {
        'up': -3,     # Move blank tile up
        'down': 3,    # Move blank tile down
        'left': -1,   # Move blank tile left
        'right': 1    # Move blank tile right
    }

    for move, position_change in moves.items():
        new_index = zero_index + position_change

        # Check if the move is valid
```

```
        if 0 <= new_index < 9 and not (
            (zero_index % 3 == 0 and move == 'left') or
            (zero_index % 3 == 2 and move == 'right')
        ):
            new_state = state_list[:]
            # Swap the blank tile with the target tile
            new_state[zero_index], new_state[new_index] = new_state[new_index], new_state[zero_index]
            successors.append((tuple(new_state), move))

    return successors

# Example usage
if __name__ == "__main__":
    # Initial state (represented as a tuple)
    start_state = (1, 2, 3, 4, 0, 5, 6, 7, 8)  # 0 represents the blank tile

    # Goal state
    goal_state = (1, 2, 3, 4, 5, 6, 7, 8, 0)

    # Perform Iterative Deepening Search
    solution = iterative_deepening_search(start_state, goal_state)

    if solution:
        print("Solution found:", solution)
    else:
        print("No solution found.")
```

OUTPUT SNAPSHOT:

Program 11: Resolution

Algorithm

PAGE NO:
DATE:

10) Create a KB using propositional logic & prove that the query using resolution.

function resolution(KB, query): returns query is true/false
input: KB, the knowledge base
query: a statement to prove

clauses = convert to CNF (KB)
negated_query = negate (query)
new_clauses = set()
apply the resolution rules.
- Select two clauses contain complementary clauses
- resolve the two to form a new clause
- add the new_clauses to set
- if the new_clauses is empty {}, contradiction is found.
- if the new_clauses is empty{} == {} return True.

else return False

O/P:- KB: $P \vee Q$ IP -> R is true.

$\neg Q \vee R$ $\neg R$.

$\neg P \vee S$

$R \vee \neg S$

$\neg R$

Resolution ->

$\neg R$ $R \vee \neg S$

$\neg P \vee \cancel{S}$ $\neg S$

$\neg P$ $\cancel{P} \vee Q$

(True) $\neg Q \vee R$ $\cancel{Q}$

$\cancel{R}$ $\neg R$ = {}

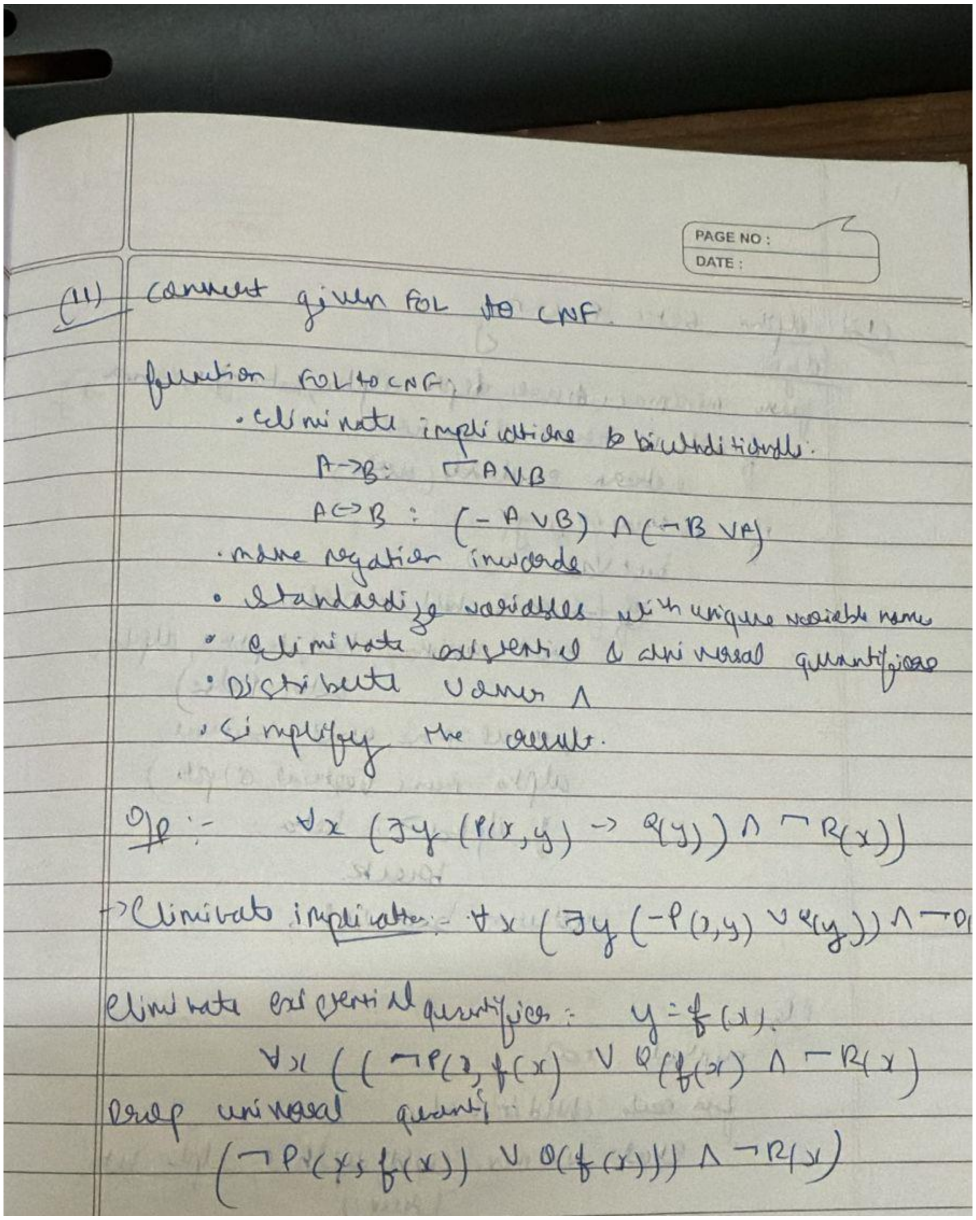
Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies
from sympy import symbols

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)
    else:
        print("The query is proved (contradiction found).")

# Example usage
if __name__ == "__main__":
    knowledge_base_resolution()
```

OUTPUT SNAPSHOT:

```
Knowledge Base in CNF: False
The query is proved (contradiction found).
```

Program 12: FOL to CNF
Algorithm:



(11) Convert given FOL to CNF.

function FOLtoCNF:

- Eliminate implications & biconditionals.
  - $A \to B$: $\neg A \vee B$
  - $A \leftrightarrow B$: $(\neg A \vee B) \wedge (\neg B \vee A)$
- Move negation inwards
- Standardize variables with unique variable name
- Eliminate existential & universal quantifiers
- Distribute $\vee$ over $\wedge$
- Simplify the result.

O/P :- $\forall x\, (\exists y\, (P(x,y) \to Q(y)) \wedge \neg R(x))$

→ Eliminate implications :- $\forall x\, (\exists y\, (\neg P(x,y) \vee Q(y)) \wedge \neg R($

Eliminate existential quantifier : $y = f(x)$.

$\forall x\, ((\neg P(x, f(x)) \vee Q(f(x)) \wedge \neg R(x))$

Drop universal quantifier

$(\neg P(x, f(x)) \vee Q(f(x))) \wedge \neg R(x)$

Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)
```

```
    if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)

    else:
        print("The query is proved (contradiction found).")

# Example usage for converting FOL to CNF
if __name__ == "__main__":
    # Define symbols for FOL example
    A, B, C = symbols('A B C')

    # Example FOL statement: (A -> B) AND (B -> C)
    fol_statement = And(Implies(A, B), Implies(B, C))

    # Convert to CNF
    cnf_statement = convert_to_cnf(fol_statement)
    print("Original FOL Statement:", fol_statement)
    print("Converted CNF Statement:", cnf_statement)

    # Run resolution demonstration
    knowledge_base_resolution()
```

OUTPUT SNAPSHOT:

```
Original FOL Statement: (Implies(A, B)) & (Implies(B, C))
Converted CNF Statement: (B | ~A) & (C | ~B)
Knowledge Base in CNF: False
The query is proved (contradiction found).
```

Program 12: Alpha Beta Pruning

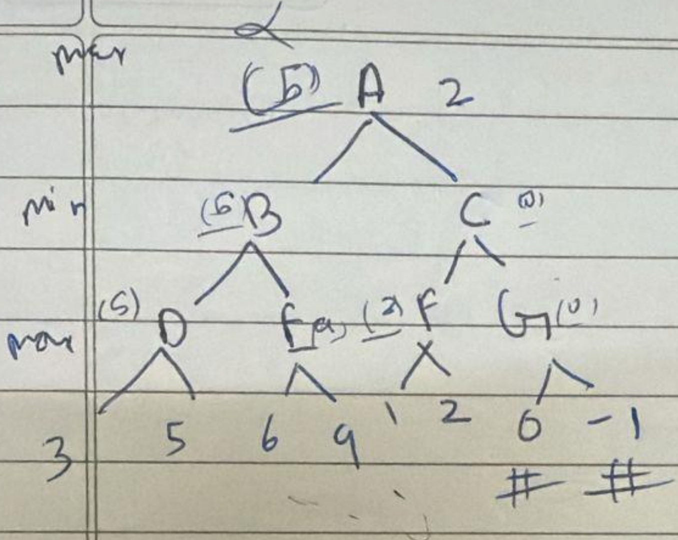
Algorithm:

(12) Alpha Beta Pruning

Algo:-

```
func minimax(node, depth, alpha, beta, isMaximizingPlayer)
    if node is terminal node
        return evaluate(node)
    if isMaximizing()
        bestVal = -∞
        for each child in node:
            eval = minimax(child, depth-1, alpha,
                           beta, False)
            bestVal = max(eval, bestVal)
            alpha = max(bestVal, alpha)
            if alpha >= beta:
                break.
        return bestVal

else:
    minVal = +∞
    for each child in node:
        eval = minimax(child, depth-1, alpha, beta,
                       True)
        minVal = min(eval, minVal)
        beta = min(minVal, beta)
        if alpha >= beta
            break.
return minVal.
```

PAGE N
DATE :



$\alpha \geq \beta$.

$5 \geq 2$ prune

Sn
20/12/24

Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def alpha_beta_pruning(depth, node_index, maximizing_player, values, alpha, beta):
    """
    Implement the Alpha-Beta Pruning algorithm.

    Parameters:
        depth (int): Current depth in the game tree.
        node_index (int): Index of the current node in the game tree.
        maximizing_player (bool): True if the current player is maximizing, False otherwise.
        values (list): Terminal node values (leaf nodes).
        alpha (float): Alpha value for pruning.
        beta (float): Beta value for pruning.

    Returns:
        int: The optimal value for the current player.
    """
    if depth == 0 or node_index >= len(values):
        return values[node_index]

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # Assume binary tree
            eval = alpha_beta_pruning(depth - 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # Assume binary tree
            eval = alpha_beta_pruning(depth - 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval

def knowledge_base_resolution():
    """
    Demonstrate resolution-based proof in propositional logic.
```

```
    """
    # Step 1: Define symbols
    P, Q, R = symbols('P Q R')

    # Step 2: Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )

    # Step 3: Define the query
    query = R

    # Step 4: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 5: Convert KB to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:", kb_cnf)

    # Step 6: Apply Resolution
    # Note: Implementing resolution directly requires symbolic manipulation of CNF clauses.
    # For simplicity, we demonstrate by showing the result from the CNF.

    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        print("The query is NOT proved (no contradiction found).")
        print("Satisfying assignment:", result)
    else:
        print("The query is proved (contradiction found).")

# Example usage for converting FOL to CNF
if __name__ == "__main__":
    # Example usage of Alpha-Beta Pruning
    print("Alpha-Beta Pruning Example:")
    values = [3, 5, 6, 9, 1, 2, 0, -1]  # Leaf nodes of the game tree
    depth = 3  # Depth of the tree
    optimal_value = alpha_beta_pruning(depth, 0, True, values, float('-inf'), float('inf'))
    print("Optimal value:", optimal_value)

    # Define symbols for FOL example
    A, B, C = symbols('A B C')

    # Example FOL statement: (A -> B) AND (B -> C)
    fol_statement = And(Implies(A, B), Implies(B, C))
```

```
# Convert to CNF
cnf_statement = convert_to_cnf(fol_statement)
print("Original FOL Statement:", fol_statement)
print("Converted CNF Statement:", cnf_statement)

# Run resolution demonstration
knowledge_base_resolution()
```

OUTPUT SNAPSHOT:

```
Alpha-Beta Pruning Example:
Optimal value: 5
Original FOL Statement: (Implies(A, B)) & (Implies(B, C))
Converted CNF Statement: (B | ~A) & (C | ~B)
Knowledge Base in CNF: False
The query is proved (contradiction found).
```

Program 14: Query entails Knowledge base or not

Algorithm:

(13) Show that the given query entails the knowledge base or not

→ Input:- (i) A knowledge base consisting of propositional logic statements

(ii) A query Q to verify whether it is entailed KB

Step 1) → Define the KB

· Represent the knowledge base KB as a set of propositional logic statements

Step 2 → Define the query:

· Represent the query Q as a propositional logic st

Step 3) → Negate the query:

· Compute ¬Q (the negation of query)

Step 4) → Combine KB & ¬Q

· Construct KB ∧ ¬Q, which includes the KB & the negated query

Step 5) → Convert to CNF

· Convert KB ∧ ¬Q to CNF

Step 6) → Apply resolution

Step 7) → Check for contradiction

Step 8) O/p results

→ O/P :- KB: (P >> Q) & (Q >> R) & P

Query: R

KB in CNF.

Entailment result: The query is entailed by KB (contradiction fou

Code:

```
from sympy.logic.boolalg import Or, And, Not, Implies, Equivalent
from sympy import symbols

def convert_to_cnf(statement):
    """
    Convert a given first-order logic statement into Conjunctive Normal Form (CNF).
    """
    from sympy.logic.boolalg import to_cnf
    return to_cnf(statement, simplify=True)

def check_entailment(kb, query):
    """
    Check if the given query is entailed by the knowledge base (KB) using resolution.

    Parameters:
        kb (Expr): The knowledge base in propositional logic.
        query (Expr): The query to check for entailment.

    Returns:
        str: Result indicating whether the query is entailed or not.
    """
    # Step 1: Negate the query and add it to the KB
    kb_with_negated_query = And(kb, Not(query))

    # Step 2: Convert KB with negated query to Conjunctive Normal Form (CNF)
    from sympy.logic.boolalg import to_cnf
    kb_cnf = to_cnf(kb_with_negated_query, simplify=True)

    print("Knowledge Base in CNF:\n", kb_cnf)

    # Step 3: Apply Resolution
    # Check satisfiability
    from sympy.logic.inference import satisfiable
    result = satisfiable(kb_cnf, all_models=False)

    if result:
        return "The query is NOT entailed by the knowledge base (no contradiction found)."
    else:
        return "The query is entailed by the knowledge base (contradiction found)."

if __name__ == "__main__":
    # Define symbols for the knowledge base and query
    P, Q, R = symbols('P Q R')

    # Define the Knowledge Base (KB)
    kb = And(
        Implies(P, Q),  # If P, then Q
        Implies(Q, R),  # If Q, then R
        P               # P is true
    )
```

```
# Define the query
query = R

# Check entailment
print("Knowledge Base:", kb)
print("Query:", query)
result = check_entailment(kb, query)
print("Entailment Result:", result)
```

OUTPUT SNAPSHOT: