VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

M Rajashekar Reddy (1BM22CS138)

in partial fulfilment 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
Academic Year 2024-25 (odd)

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 **M Rajashekar Reddy (1BM22CS138),** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above-mentioned subject, and the work prescribed for the said degree.

Prameetha Pai Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

GitHub Link:
https://github.com/Raja3008/AI-1BM22CS138

Program 1: Implement Tic –Tac –Toe Game.

Algorithm:

01/10/24

Bafna Gold

LAB-1

* Implement Tic-Tac-Toe Game.

Algorithm:

Step-1: Create a 3×3 board initialized with empty strings.

Step-2: Start with player 'x'

Step-3: Print the current board & then prompt the current player to enter their move(row and column).

Step-4: Check if the chosen cell is empty. If empty, place player's symbol in that cell.

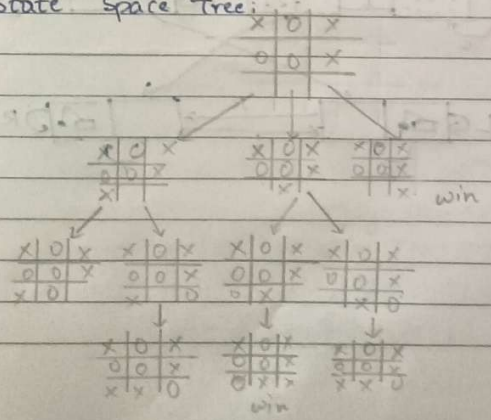
Step-5: Now, check for winner. If there is a winner print the board and declare the winner.

If the board is full and there's no winner, declare a tie.

Else, if the game is still going on, switch to the other player.

If the cell is not empty, prompt for a valid move.

State Space Tree:



Code:

```
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-" * 5)

def check_winner(board):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]

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

    return None

def is_full(board):
    return all(cell != " " for row in board for cell in row)

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]  # Initialize the board with spaces

    current_player = "X"
    while True:
        print_board(board)
        try:
            row = int(input(f"Player {current_player}, Enter the row (0-2): "))
            col = int(input(f"Player {current_player}, Enter the column (0-2): "))

            if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == " ":
                board[row][col] = current_player
                winner = check_winner(board)
                if winner:
                    print_board(board)
                    print(f"Player {winner} wins!")
                    break
                elif is_full(board):
                    print_board(board)
```

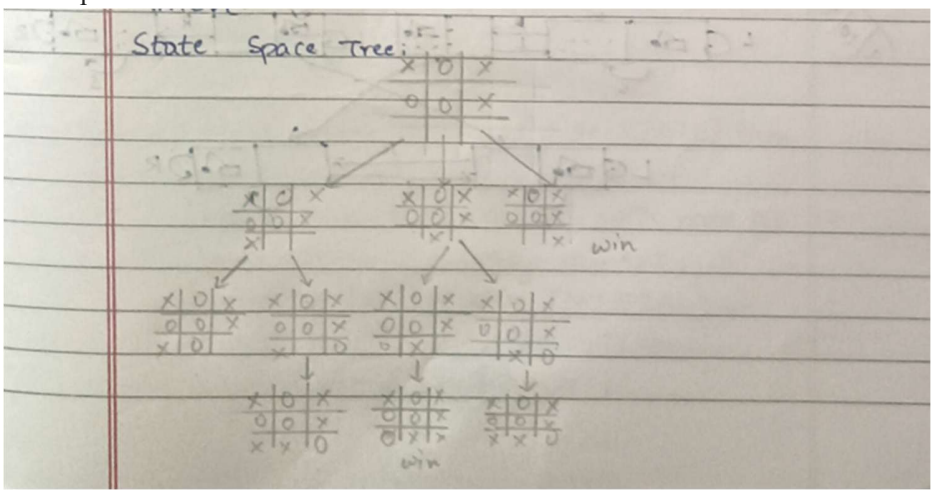
```
                print("It's a tie!")
                break
            current_player = "O" if current_player == "X" else "X"
        else:
            print("Invalid move, try again.")
    except ValueError:
        print("Please enter valid numbers (0-2) for row and column.")

tic_tac_toe()
```

Output:

```
 -----
Player X, Enter the row (0-2): 0
Player X, Enter the column (0-2): 0
X| |
 -----
  | |
 -----
  | |
 -----
Player O, Enter the row (0-2): 1
Player O, Enter the column (0-2): 0
X| |
 -----
O| |
 -----
  | |
 -----
Player X, Enter the row (0-2): 0
Player X, Enter the column (0-2): 1
X|X|
 -----
O| |
 -----
  | |
 -----
Player O, Enter the row (0-2): 0
Player O, Enter the column (0-2): 2
X|X|O
 -----
O| |
 -----
  | |
 -----
```

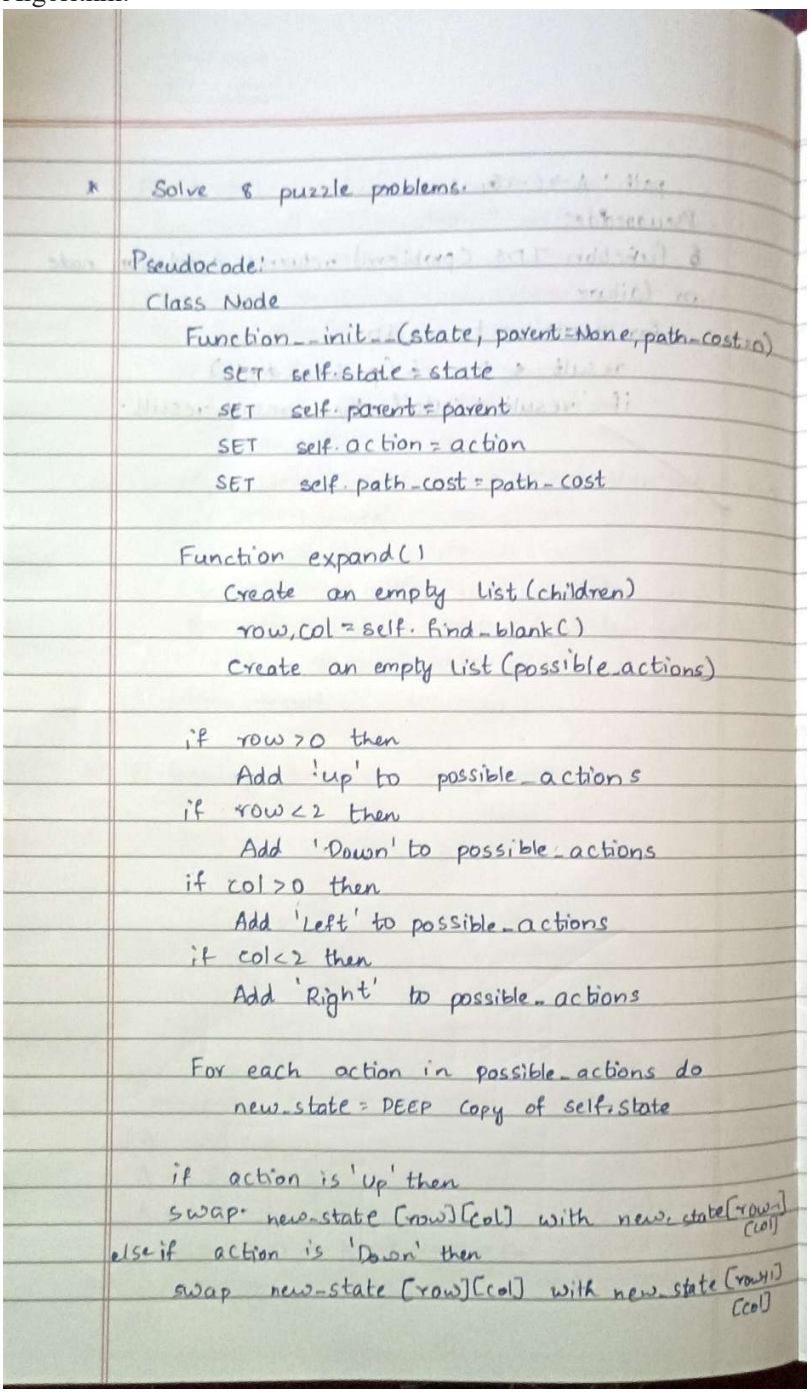
```
Player X, Enter the row (0-2): 1
Player X, Enter the column (0-2): 1
X|X|O
 -----
O|X|
 -----
  | |
 -----
Player O, Enter the row (0-2): 2
Player O, Enter the column (0-2): 1
X|X|O
 -----
O|X|
 -----
  |O|
 -----
Player X, Enter the row (0-2): 2
Player X, Enter the column (0-2): 2
X|X|O
 -----
O|X|
 -----
  |O|X
 -----
Player X wins!
```

State Space Tree:



Program 2: Solve 8 puzzle problems.

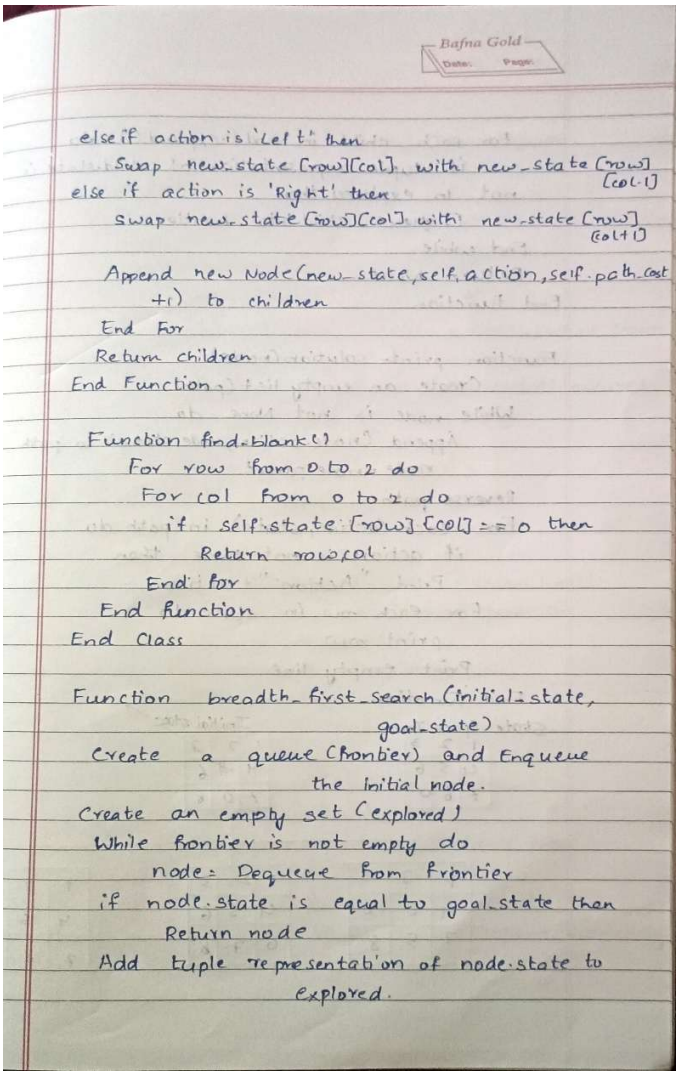
Algorithm:



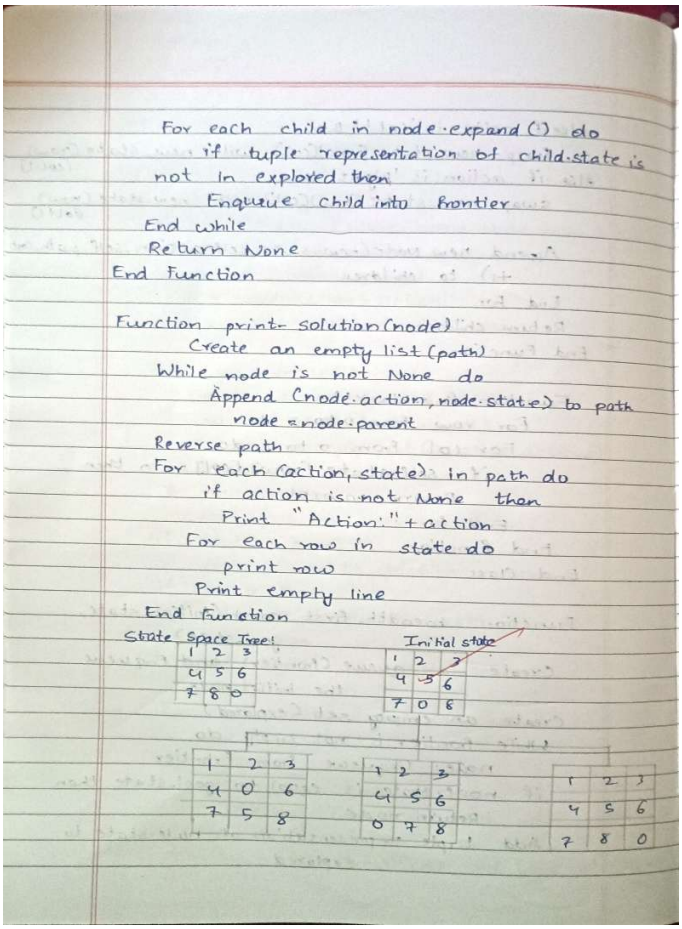
Solve 8 puzzle problems.

Pseudocode:

```
Class Node
    Function __init__(state, parent=None, path_cost=0)
        SET self.state = state
        SET self.parent = parent
        SET self.action = action
        SET self.path_cost = path_cost

    Function expand()
        Create an empty list (children)
        row, col = self.find_blank()
        Create an empty list (possible_actions)

    if row > 0 then
        Add 'Up' to possible_actions
    if row < 2 then
        Add 'Down' to possible_actions
    if col > 0 then
        Add 'Left' to possible_actions
    if col < 2 then
        Add 'Right' to possible_actions

    For each action in possible_actions do
        new_state = DEEP copy of self.state

    if action is 'Up' then
    swap new_state[row][col] with new_state[row-1][col]
else if action is 'Down' then
    swap new_state[row][col] with new_state[row+1][col]
```



```
else if action is 'Left' then
    Swap new_state[row][col] with new_state[row][col-1]
else if action is 'Right' then
    Swap new_state[row][col] with new_state[row][col+1]

  Append new Node(new_state, self.action, self.path_cost+1) to children
 End For
 Return children
End Function

 Function find_blank()
    For row from 0 to 2 do
      For col from 0 to 2 do
        if self.state[row][col] == 0 then
          Return row,col
      End for
  End function
End Class

Function breadth_first_search(initial_state, goal_state)
 Create a queue (frontier) and Enqueue the initial node.
 Create an empty set (explored)
 While frontier is not empty do
     node = Dequeue from frontier
 if node.state is equal to goal_state then
     Return node
 Add tuple representation of node.state to explored.
```



```
    For each child in node.expand() do
      if tuple representation of child.state is
    not in explored then
        Enqueue child into frontier
  End while
  Return None
End Function

Function print_solution(node)
    Create an empty list (path)
  While node is not None do
    Append (node.action, node.state) to path
      node = node.parent
  Reverse path
  For each (action, state) in path do
    if action is not None then
      Print "Action:" + action
    For each row in state do
      print row
    Print empty line
End Function
```

State Space Tree:

1	2	3
4	5	6
7	8	0

Initial state

1	2	3
4	5	6
7	0	8

1	2	3
4	0	6
7	5	8

1	2	3
4	5	6
0	7	8

1	2	3
4	5	6
7	8	0

Code:

```
import copy
from collections import deque

class Node:
    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost

    def expand(self):
        children = []
        row, col = self.find_blank()
```

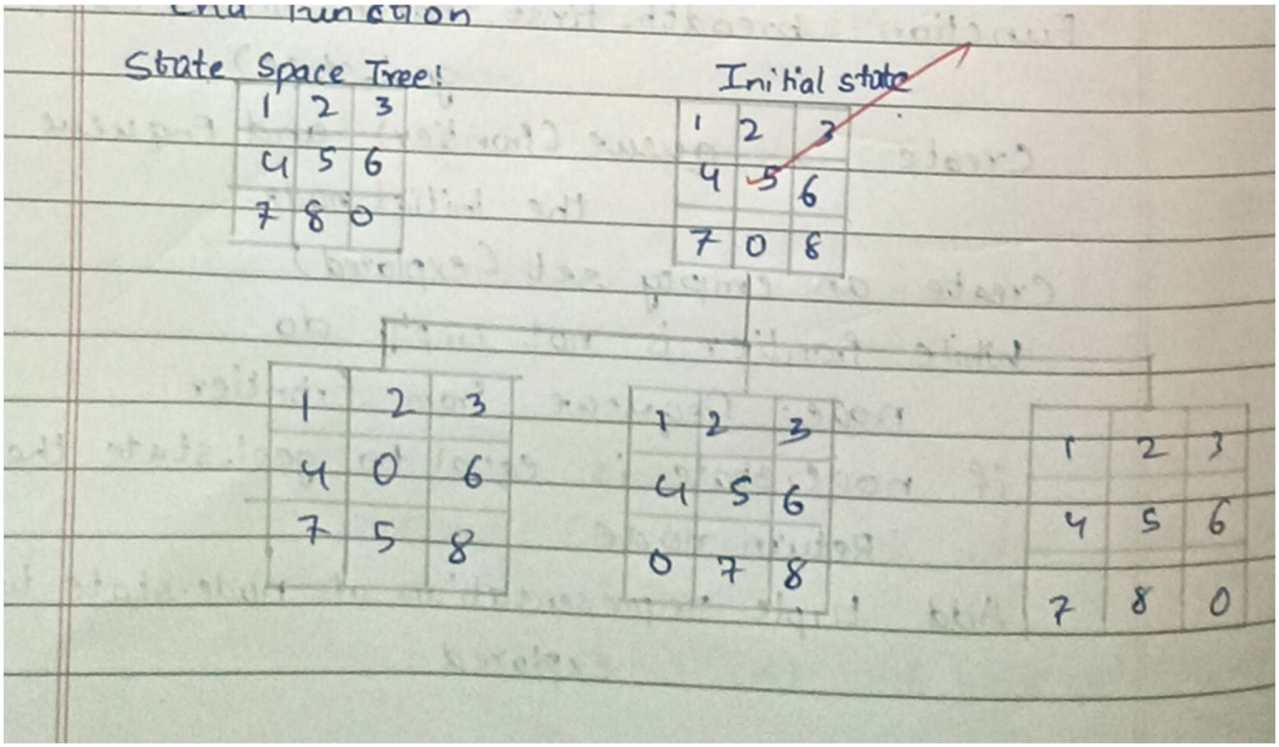
```
        possible_actions = []
        if row > 0:
            possible_actions.append('Up')
        if row < 2:
            possible_actions.append('Down')
        if col > 0:
            possible_actions.append('Left')
        if col < 2:
            possible_actions.append('Right')

        for action in possible_actions:
            new_state = copy.deepcopy(self.state)
            if action == 'Up':
                new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
            elif action == 'Down':
                new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            elif action == 'Left':
                new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
            elif action == 'Right':
                new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            children.append(Node(new_state, self, action, self.path_cost + 1))
        return children

    def find_blank(self):
        for row in range(3):
            for col in range(3):
                if self.state[row][col] == 0:
                    return row, col

def breadth_first_search(initial_state, goal_state):
    frontier = deque([Node(initial_state)])
    explored = set()

    while frontier:
        node = frontier.popleft()
        if node.state == goal_state:
            return node
        explored.add(tuple(map(tuple, node.state)))
        for child in node.expand():
```

```
            if tuple(map(tuple, child.state)) not in explored:
                frontier.append(child)
    return None

def print_solution(node):
    path = []
    while node is not None:
        path.append((node.action, node.state))
        node = node.parent
    path.reverse()
    for action, state in path:
        if action:
            print(f"Action: {action}")
        for row in state:
            print(row)
        print()
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

solution = breadth_first_search(initial_state, goal_state)

if solution:
    print("Solution found:")
    print_solution(solution)
else:
    print("Solution not found.")
```

Output:

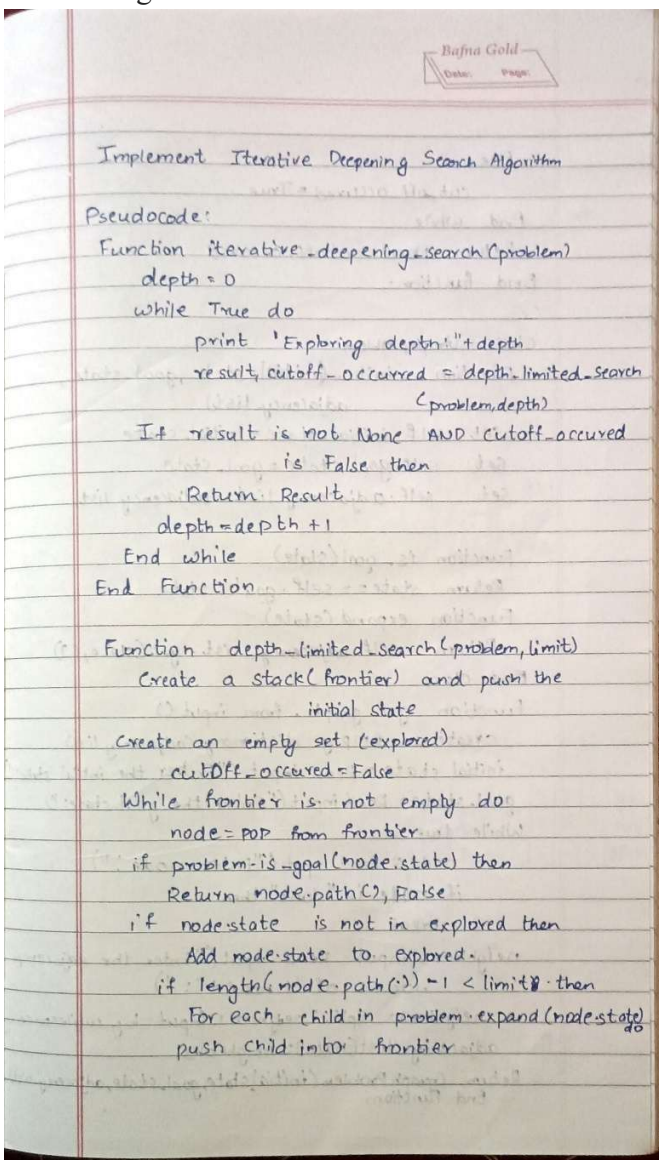
```
Solution found:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

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

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

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

State Space Tree:



Program 3: Implement Iterative deepening search algorithm.

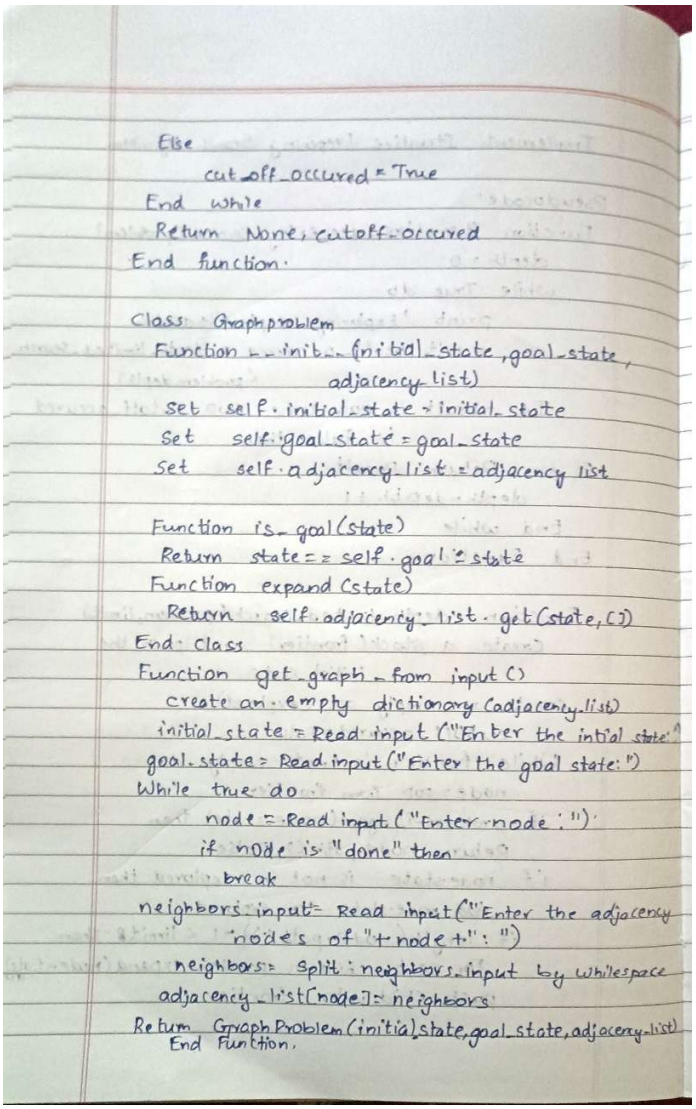
Algorithm:



Implement Iterative Deepening Search Algorithm

Pseudocode:

```
Function iterative_deepening_search(problem)
    depth = 0
    while True do
        print 'Exploring depth:' + depth
        result, cutoff_occurred = depth_limited_search
                                  (problem, depth)
    If result is not None AND cutoff_occured
                is False then
        Return Result
    depth = depth + 1
  End while
End Function

Function depth_limited_search(problem, limit)
    Create a stack(frontier) and push the
                initial state
  Create an empty set (explored)
      cutOff_occured = False
  While frontier is not empty do
      node = POP from frontier
   if problem.is_goal(node.state) then
      Return node.path(), False
   if node.state is not in explored then
      Add node.state to explored.
     if length(node.path()) - 1 < limit then
      For each child in problem.expand(node.state)
      push child into frontier
```



```
    Else
        cut_off_occured = True
  End while
  Return None, cutoff_occured
End function.

Class GraphProblem
  Function __init__(initial_state, goal_state,
                adjacency_list)
    Set self.initial_state = initial_state
    Set self.goal_state = goal_state
    Set self.adjacency_list = adjacency_list

  Function is_goal(state)
    Return state == self.goal_state
  Function expand(state)
    Return self.adjacency_list.get(state,[])
End class
Function get_graph_from_input()
    create an empty dictionary (adjacency_list)
  initial_state = Read input("Enter the initial state:")
  goal_state = Read input("Enter the goal state:")
  While true do
      node = Read input("Enter node:")
      if node is "done" then
        break
  neighbors_input = Read input("Enter the adjacency
          nodes of "+node+": ")
    neighbors = split neighbors_input by whitespace
    adjacency_list[node] = neighbors
  Return GraphProblem(initial_state, goal_state, adjacency_list)
  End Function.
```

Code:

```
class Node:
    def __init__(self, state, parent=None):
        self.state = state
        self.parent = parent

    def path(self):
        result = []
        current_node = self
        while current_node:
```

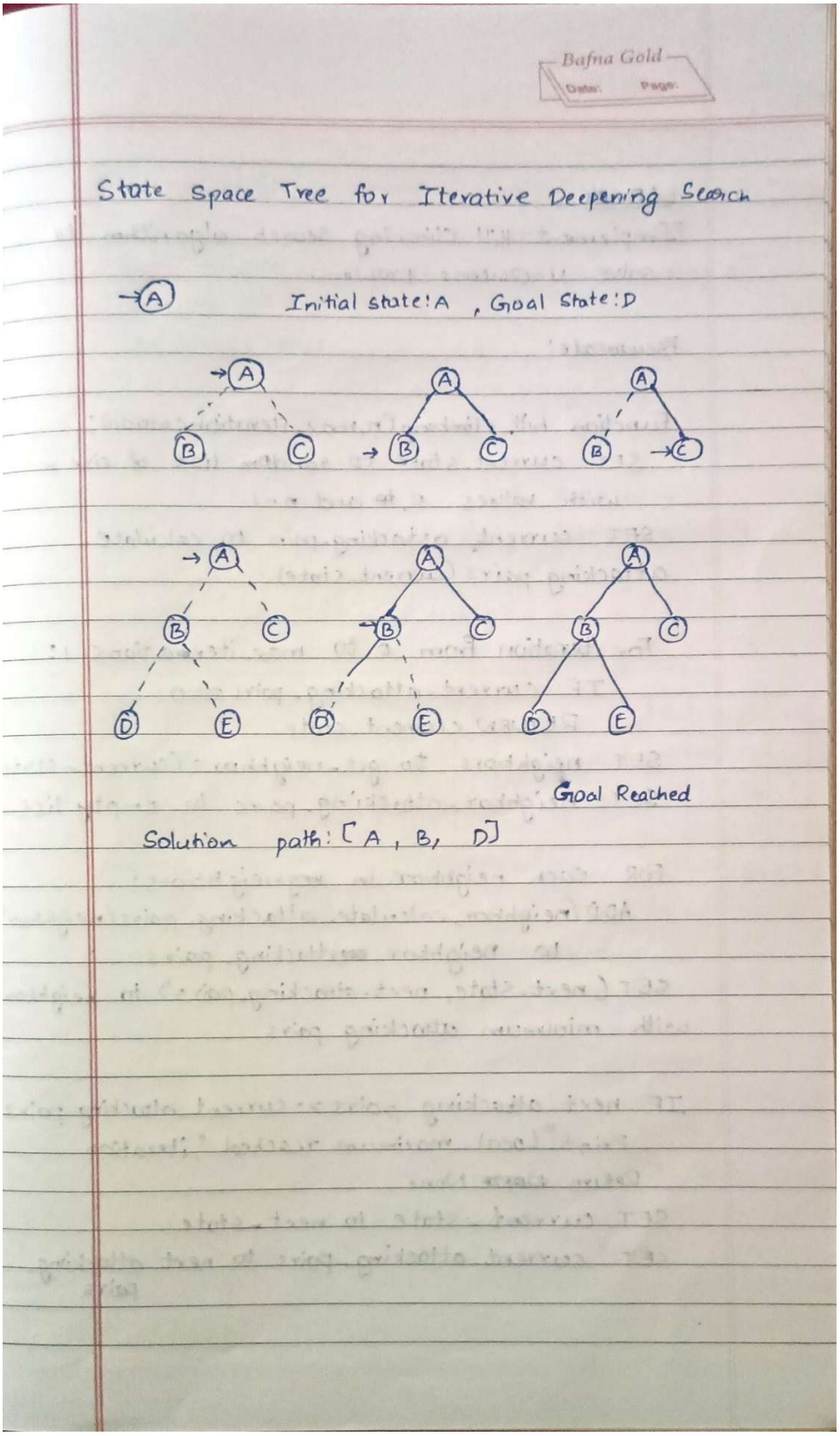
```
            result.append(current_node.state)
            current_node = current_node.parent
        return result[::-1]

def iterative_deepening_search(problem):
    depth = 0
    while True:
        print(f"Exploring depth: {depth}")
        result, cutoff_occurred = depth_limited_search(problem, depth)
        if result is not None and not cutoff_occurred:
            return result
        depth += 1

def depth_limited_search(problem, limit):
    frontier = [Node(problem.initial_state)]
    explored = set()
    cutoff_occurred = False

    while frontier:
        node = frontier.pop()

        if problem.is_goal(node.state):
            return node.path(), False

        if node.state not in explored:
            explored.add(node.state)
            if len(node.path()) - 1 < limit:
                for child in problem.expand(node.state):
                    frontier.append(Node(child, node))
            else:
                cutoff_occurred = True
    return None, cutoff_occurred

class GraphProblem:
    def __init__(self, initial_state, goal_state, adjacency_list):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.adjacency_list = adjacency_list

    def is_goal(self, state):
        return state == self.goal_state

    def expand(self, state):
```

```
        return self.adjacency_list.get(state, [])

def get_graph_from_input():
    adjacency_list = {}
    initial_state = input("Enter the initial state: ").strip()
    goal_state = input("Enter the goal state: ").strip()

    print("Enter the adjacency list for the graph ")
    print("Enter 'done' when finished.")

    while True:
        node = input("Enter node (or 'done' to stop): ").strip()
        if node.lower() == 'done':
            break
        neighbors_input = input(f"Enter the adjacent nodes of {node}: ").strip()
        neighbors = neighbors_input.split()
        adjacency_list[node] = [neighbor.strip() for neighbor in neighbors]

    return GraphProblem(initial_state, goal_state, adjacency_list)

if __name__ == "__main__":
    problem = get_graph_from_input()
    solution = iterative_deepening_search(problem)

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

Output:

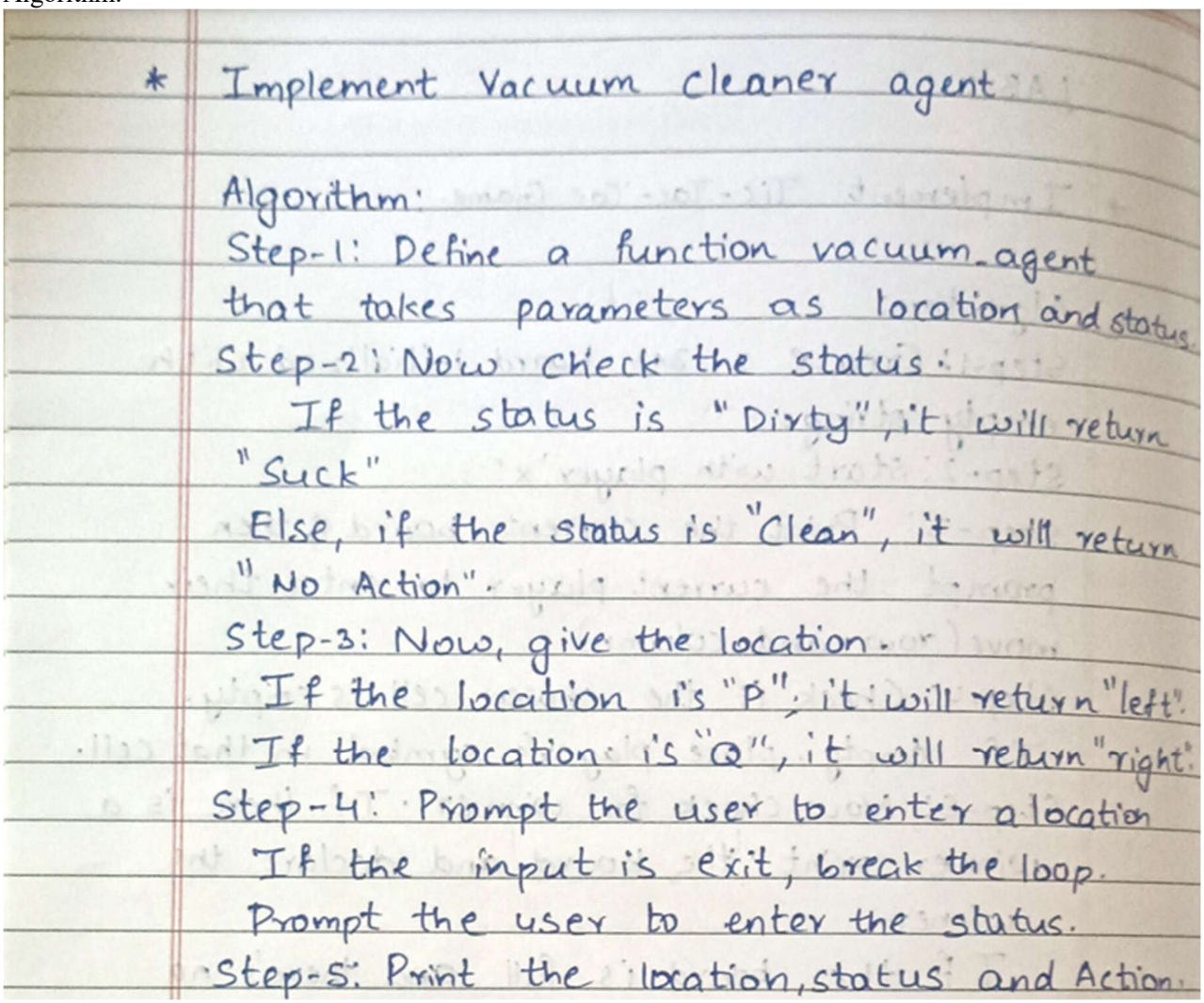
```
Enter the initial state: A
Enter the goal state: F
Enter the adjacency list for the graph
Enter 'done' when finished.
Enter node (or 'done' to stop): A
Enter the adjacent nodes of A: B C
Enter node (or 'done' to stop): B
Enter the adjacent nodes of B: D E
Enter node (or 'done' to stop): C
Enter the adjacent nodes of C: F
Enter node (or 'done' to stop): done
Exploring depth: 0
Exploring depth: 1
Exploring depth: 2
Solution Path: ['A', 'C', 'F']
```

State Stace Tree:



Program 4: Implement vacuum cleaner agent.

Algorithm:



Code:

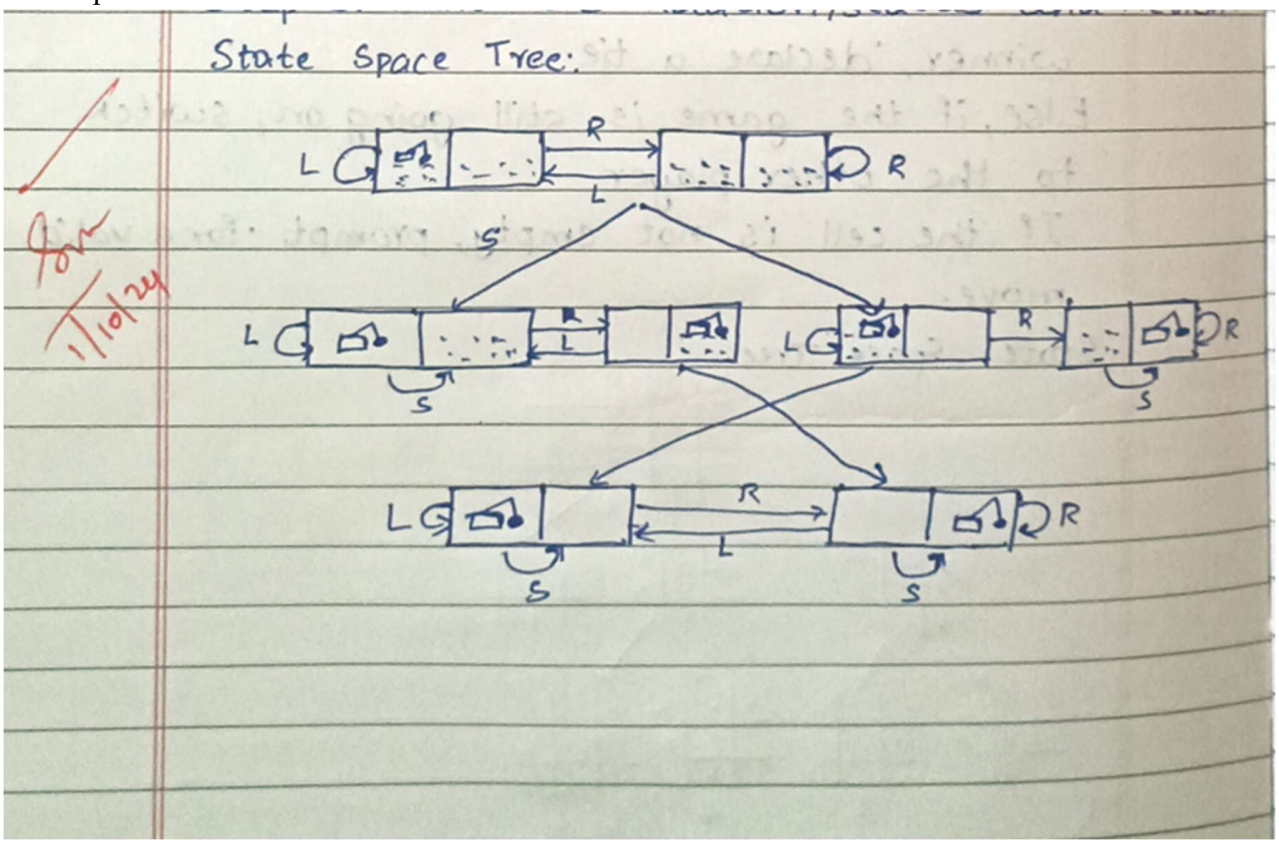
```
def vacuum_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif status == "Clean":
        return "No Action"

    if location == "P":
        return "left"
    elif location == "Q":
        return "right"
    else:
        return "Invalid Location"
```

```
while True:
    location = input("Enter the location (P or Q, or 'exit' to stop): ")
    if location.lower() == 'exit':
        break
    status = input("Enter the status (Dirty or Clean): ")

    action = vacuum_agent(location, status)
    print(f"Location: {location}, Status: {status}, Action: {action}")
```

Output:

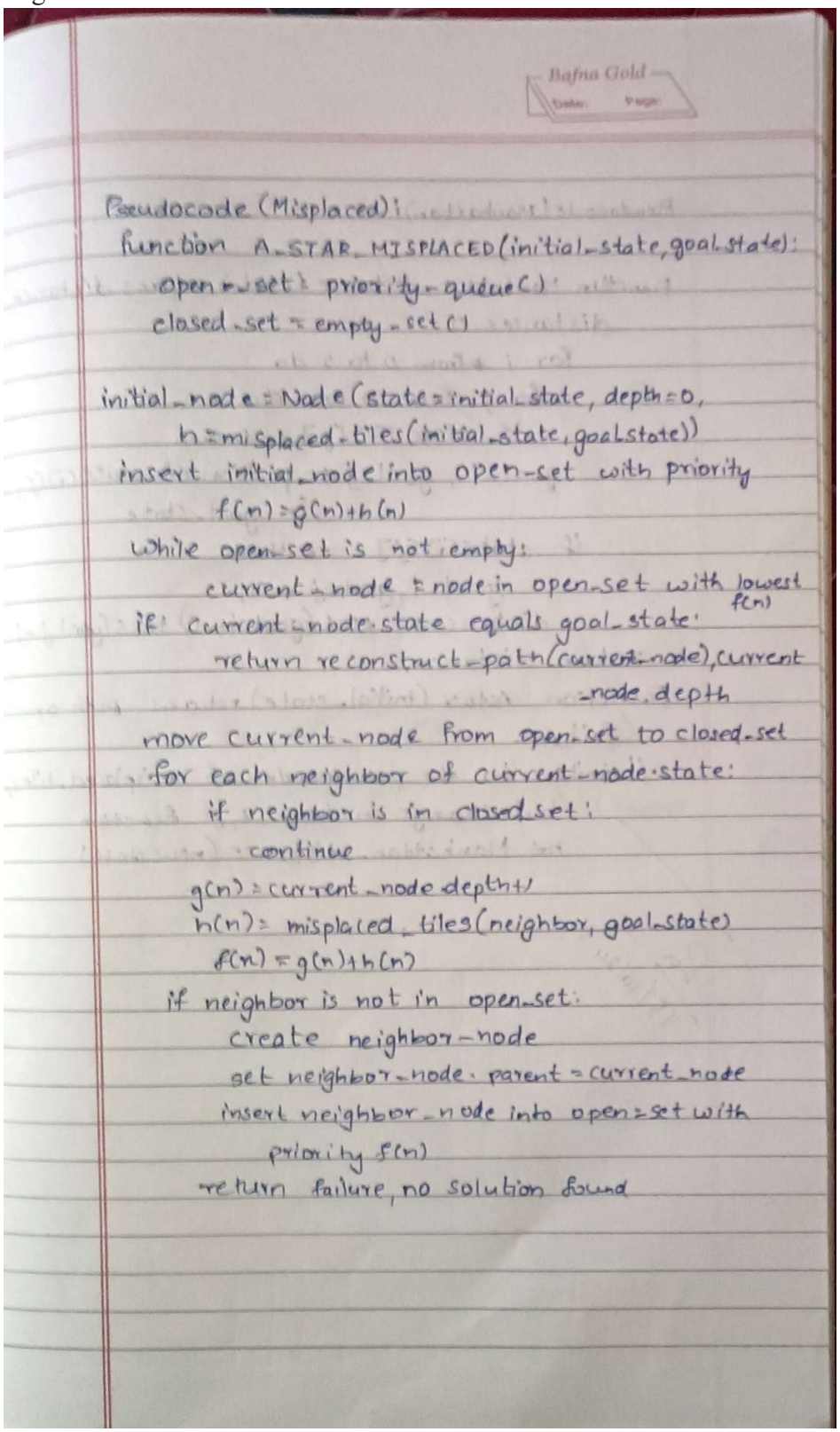
```
Location: P, Status: Dirty, Action: Suck
Location: Q, Status: Dirty, Action: Suck
Location: P, Status: Clean, Action: No Action
Location: Q, Status: Clean, Action: No Action
Location: stop, Status: exit, Action: Invalid Location
```

Stare Space Tree:



Program 5: a. Implement A* search algorithm.
b. Implement Hill Climbing Algorithm.

Algorithm:



Pseudocode (Misplaced):

```
function A_STAR_MISPLACED(initial_state, goal_state):
    open_set = priority_queue()
    closed_set = empty_set()

initial_node = Node(state=initial_state, depth=0,
    h=misplaced_tiles(initial_state, goalstate))
insert initial_node into open_set with priority
    f(n) = g(n)+h(n)
while open_set is not empty:
    current_node = node in open_set with lowest f(n)
if current_node.state equals goal_state:
    return reconstruct_path(current_node), current
                                    _node.depth
move current_node from open_set to closed_set
for each neighbor of current_node.state:
    if neighbor is in closed_set:
        continue
    g(n) = current_node.depth+1
    h(n) = misplaced_tiles(neighbor, goal_state)
    f(n) = g(n)+h(n)
if neighbor is not in open_set:
    create neighbor_node
    set neighbor_node.parent = current_node
    insert neighbor_node into open_set with
        priority f(n)
return failure, no solution found
```

Code:

```
import heapq

def misplaced_tiles(state, goal):
    """Calculate the number of misplaced tiles."""
    return sum(1 for i in range(9) if state[i] != goal[i] and state[i] != 0)

class Node:
    def __init__(self, state, parent=None, depth=0, cost=0, h=0):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.cost = cost
        self.h = h

    def __lt__(self, other):
        return self.cost < other.cost

def astar_misplaced(initial_state, goal_state):
    def get_neighbors(state):
        neighbors = []
        zero_idx = state.index(0)
        x, y = divmod(zero_idx, 3)
        moves = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
        for move, (dx, dy) in moves.items():
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_idx = new_x * 3 + new_y
                new_state = state[:]
                new_state[zero_idx], new_state[new_idx] = new_state[new_idx], new_state[zero_idx]
                neighbors.append(new_state)
        return neighbors

    def reconstruct_path(node):
        path = []
        while node.parent:
            path.append((node.depth, node.h, node.cost, node.state))
            node = node.parent
        path.reverse()
        return path

    open_set = []
    heapq.heappush(open_set, Node(initial_state, cost=misplaced_tiles(initial_state, goal_state),
h=misplaced_tiles(initial_state, goal_state)))
```

```
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if current_node.state == goal_state:
            return reconstruct_path(current_node), current_node.depth

        closed_set.add(tuple(current_node.state))

        for neighbor in get_neighbors(current_node.state):
            if tuple(neighbor) in closed_set:
                continue

            g = current_node.depth + 1
            h_value = misplaced_tiles(neighbor, goal_state)
            f = g + h_value

            neighbor_node = Node(state=neighbor, parent=current_node, depth=g, cost=f, h=h_value)
            heapq.heappush(open_set, neighbor_node)

    return None, -1

def print_puzzle(state):
    print(f"{state[0]} {state[1]} {state[2]}")
    print(f"{state[3]} {state[4]} {state[5]}")
    print(f"{state[6]} {state[7]} {state[8]}")
    print()

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

path, depth = astar_misplaced(initial_state, goal_state)
print(f"Number of moves: {depth}\n")

print("Initial state:")
print_puzzle(initial_state)

for g, h, f, state in path:
    print(f"g(n) = {g}, h(n) = {h}, f(n) = {f}")
    print_puzzle(state)

print("Goal state reached!")
```

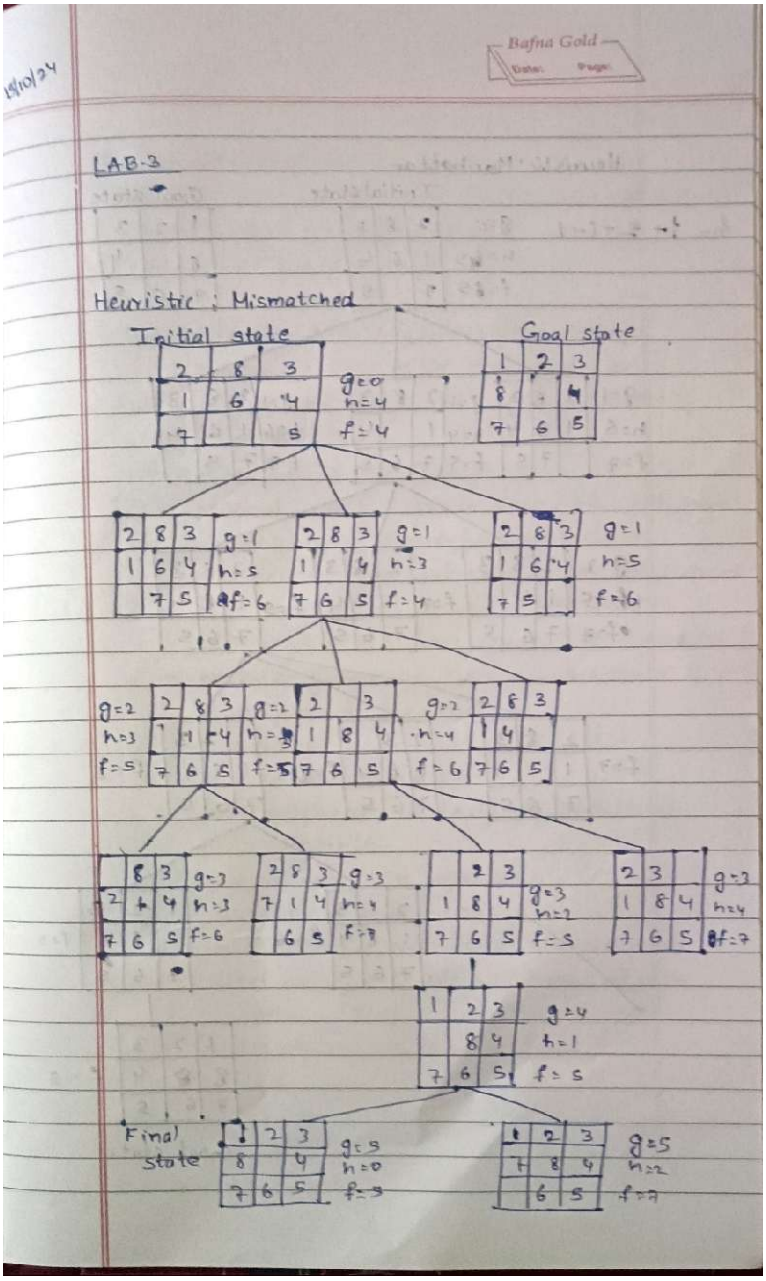
Output:

```
Number of moves: 14

Initial state:
1 2 3
4 0 5
6 7 8

g(n) = 1, h(n) = 3, f(n) = 4
1 2 3
4 5 0
6 7 8

g(n) = 2, h(n) = 3, f(n) = 5
1 2 3
4 5 8
6 7 0

g(n) = 3, h(n) = 3, f(n) = 6
1 2 3
4 5 8
6 0 7

g(n) = 4, h(n) = 3, f(n) = 7
1 2 3
4 5 8
0 6 7

g(n) = 5, h(n) = 4, f(n) = 9
1 2 3
0 5 8
4 6 7
```

```
g(n) = 6, h(n) = 5, f(n) = 11
1 2 3
5 0 8
4 6 7

g(n) = 7, h(n) = 5, f(n) = 12
1 2 3
5 6 8
4 0 7

g(n) = 8, h(n) = 5, f(n) = 13
1 2 3
5 6 8
4 7 0

g(n) = 9, h(n) = 5, f(n) = 14
1 2 3
5 6 0
4 7 8

g(n) = 10, h(n) = 4, f(n) = 14
1 2 3
5 0 6
4 7 8

g(n) = 11, h(n) = 3, f(n) = 14
1 2 3
0 5 6
4 7 8
```

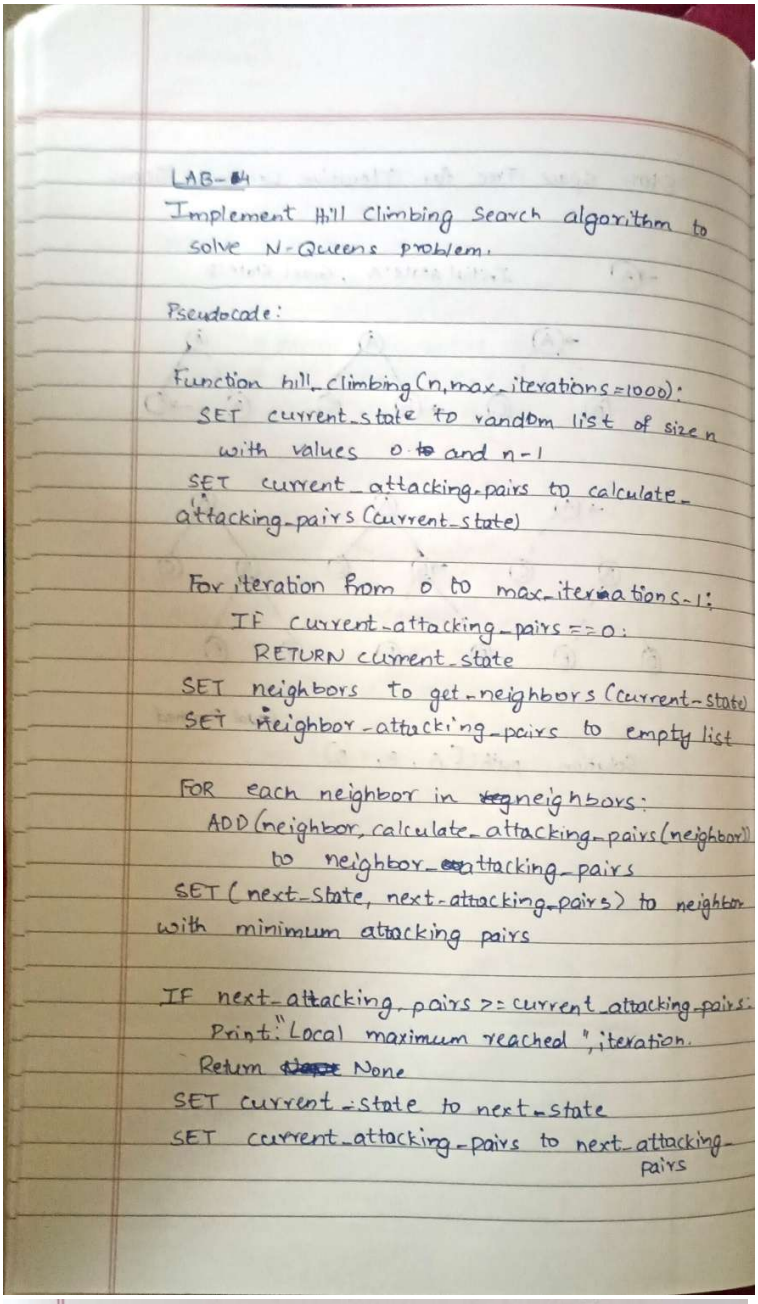
```
g(n) = 12, h(n) = 2, f(n) = 14
1 2 3
4 5 6
0 7 8

g(n) = 13, h(n) = 1, f(n) = 14
1 2 3
4 5 6
7 0 8

g(n) = 14, h(n) = 0, f(n) = 14
1 2 3
4 5 6
7 8 0

Goal state reached!
```

State Space Tree:



b. Implement Hill Climbing Algorithm.

Algorithm:



LAB-04

Implement Hill Climbing Search algorithm to solve N-Queens problem.

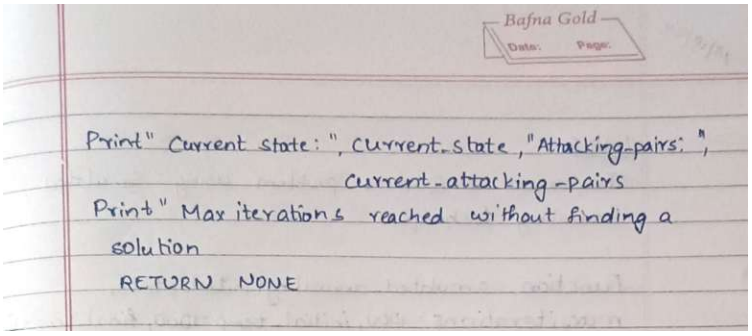
Pseudocode:

Function hill_climbing(n, max_iterations=1000):
SET current_state to random list of size n with values 0 to and n-1
SET current_attacking_pairs to calculate_attacking_pairs(current_state)

For iteration from 0 to max_iterations-1:
IF current_attacking_pairs == 0:
RETURN current_state
SET neighbors to get_neighbors(current_state)
SET neighbor_attacking_pairs to empty list

FOR each neighbor in ~~neg~~neighbors:
ADD(neighbor, calculate_attacking_pairs(neighbor)) to neighbor_attacking_pairs
SET (next_State, next_attacking_pairs) to neighbor with minimum attacking pairs

IF next_attacking_pairs >= current_attacking_pairs:
Print: "Local maximum reached", iteration.
Return None
SET current_state to next_state
SET current_attacking_pairs to next_attacking_pairs



Bafna Gold
Date: Page:

Print" Current state: ", current_state, "Attacking-pairs: ", current_attacking_pairs
Print" Max iterations reached without finding a solution
RETURN NONE

Code:

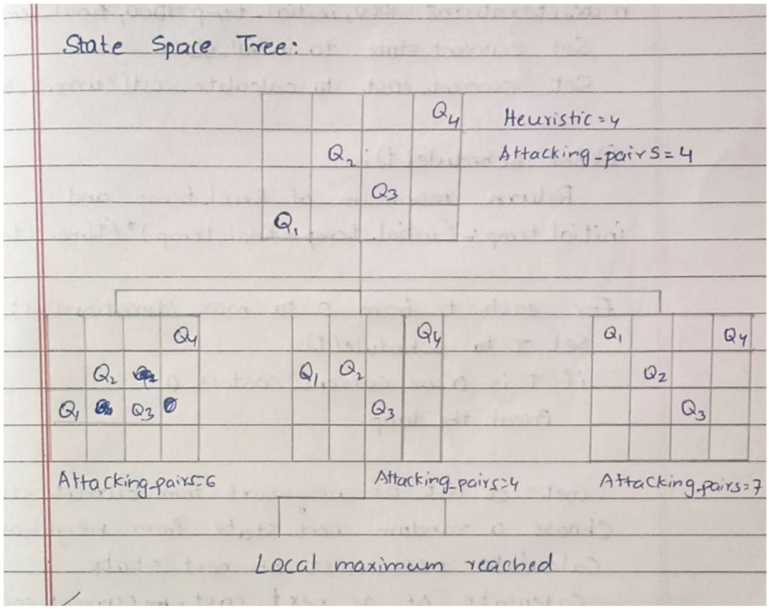
```
import random

def calculate_attacking_pairs(state):
    attacking_pairs = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacking_pairs += 1
    return attacking_pairs

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(n, max_iterations=1000):
    current_state = [random.randint(0, n - 1) for _ in range(n)]
    current_attacking_pairs = calculate_attacking_pairs(current_state)

    for iteration in range(max_iterations):
        if current_attacking_pairs == 0:
            return current_state

        neighbors = get_neighbors(current_state)
        neighbor_attacking_pairs = [(neighbor, calculate_attacking_pairs(neighbor)) for neighbor in
neighbors]
        next_state, next_attacking_pairs = min(neighbor_attacking_pairs, key=lambda x: x[1])

        if next_attacking_pairs >= current_attacking_pairs:
            print(f"Local maximum reached  {iteration}. ")
            return None

        current_state, current_attacking_pairs = next_state, next_attacking_pairs
        print(f" Current state: {current_state}, Attacking pairs: {current_attacking_pairs}")
```

```
    print(f"Max iterations reached without finding a solution.")
    return None

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")
except ValueError as e:
    print(e)
    n = 4

solution = None

while solution is None:
    solution = hill_climbing(n)

print(f"Solution found: {solution}")
```

Output:

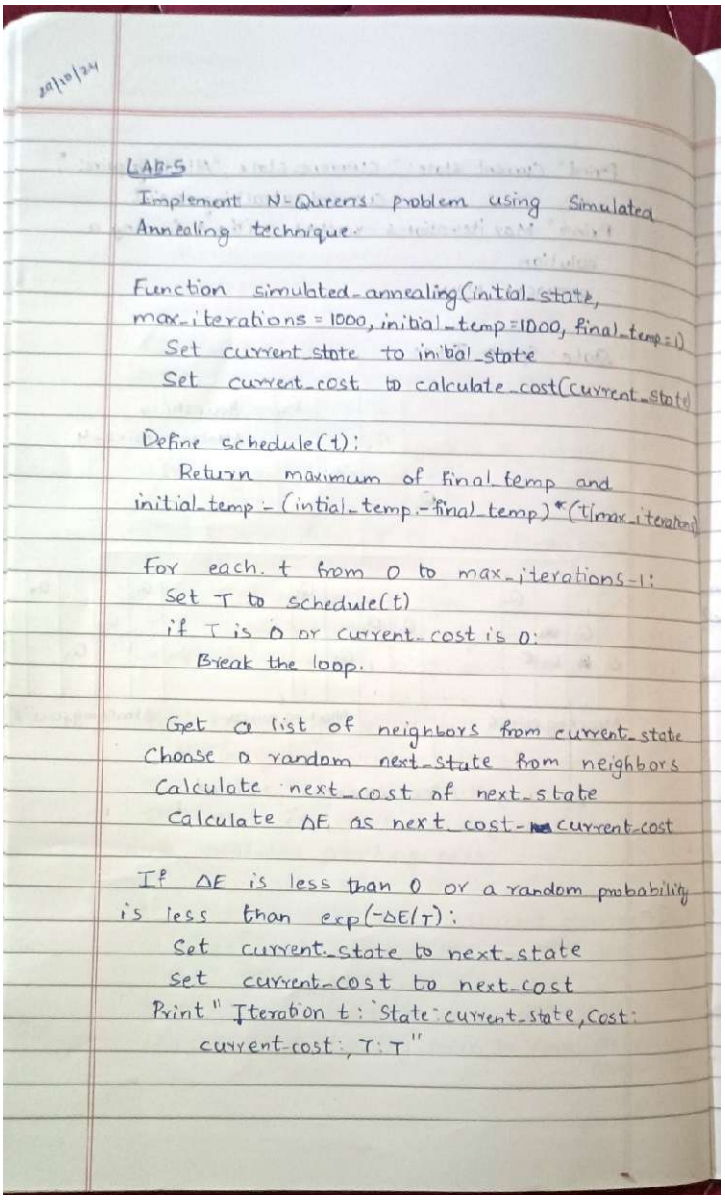
```
Enter the number of queens (N): 4
 Current state: [1, 3, 1, 0], Attacking pairs: 2
 Current state: [1, 3, 0, 0], Attacking pairs: 1
 Current state: [1, 3, 0, 2], Attacking pairs: 0
Solution found: [1, 3, 0, 2]
```

State Space Tree:



Program 6: Write a program to implement Simulated Annealing Algorithm

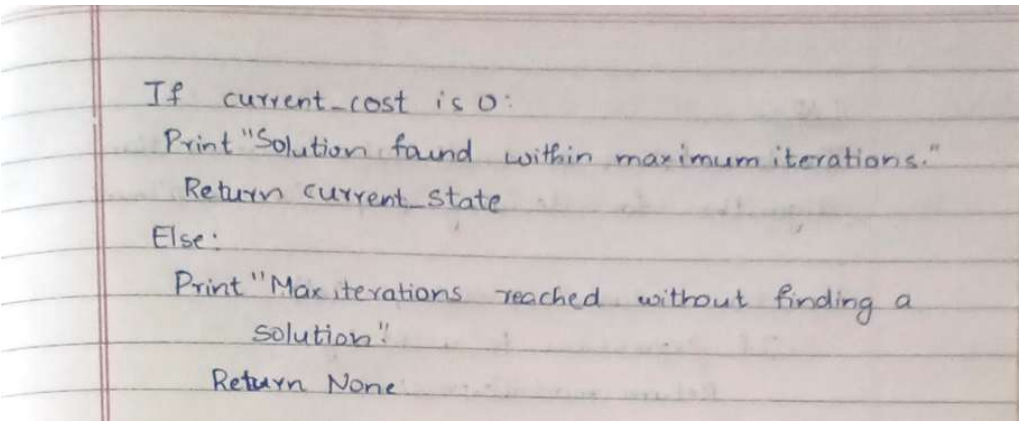
Algorithm:



LAB-5

Implement N-Queens problem using Simulated Annealing technique.

```
Function simulated_annealing(initial_state,
max_iterations = 1000, initial_temp=1000, final_temp=1)
    Set current_state to initial_state
    Set current_cost to calculate_cost(current_state)

Define schedule(t):
    Return maximum of final_temp and
initial_temp - (intial_temp - final_temp) * (t/max_iterations)

For each t from 0 to max_iterations-1:
    Set T to schedule(t)
    if T is 0 or current_cost is 0:
        Break the loop.

    Get a list of neighbors from current_state
    Choose a random next_state from neighbors
    Calculate next_cost of next_state
    Calculate ΔE as next_cost - current_cost

If ΔE is less than 0 or a random probability
is less than exp(-ΔE/T):
    Set current_state to next_state
    set current_cost to next_cost
    Print "Iteration t: State: current_state, Cost:
        current_cost, T: T"
```



```
If current_cost is 0:
    Print "Solution found within maximum iterations."
    Return current_state
Else:
    Print "Max iterations reached without finding a
        solution."
    Return None
```

Code:

```
import random
import math

def calculate_cost(state):
    n = len(state)
    return sum(
        1
        for i in range(n)
        for j in range(i + 1, n)
        if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j)
    )

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = state[:]
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def simulated_annealing(initial_state, max_iterations=1000, initial_temp=1000, final_temp=1):
    current_state = initial_state
    current_cost = calculate_cost(current_state)

    def schedule(t):
        return max(final_temp, initial_temp - (initial_temp - final_temp) * (t / max_iterations))

    for t in range(max_iterations):
        T = schedule(t)
        if T == 0 or current_cost == 0:
            break

        neighbors = get_neighbors(current_state)
        next_state = random.choice(neighbors)
        next_cost = calculate_cost(next_state)
        ΔE = next_cost - current_cost

        if ΔE < 0 or random.random() < math.exp(-ΔE / T):
            current_state, current_cost = next_state, next_cost
```

```
            print(f"Iteration {t}: State: {current_state}, Cost: {current_cost}, T: {T:.2f}")

    if current_cost == 0:
        print("Solution found within maximum iterations.")
        return current_state
    else:
        print("Max iterations reached without finding a solution.")
        return None

try:
    n = int(input("Enter the number of queens (N): "))
    if n <= 0:
        raise ValueError("N must be a positive integer.")

    initial_state_input = input(f"Enter the initial state as {n} integers: ").split()
    initial_state = [int(x) for x in initial_state_input]

    if len(initial_state) != n or any(row < 0 or row >= n for row in initial_state):
        raise ValueError(f"Invalid initial state. Provide {n} integers between 0 and {n-1}.")
except ValueError as e:
    print(e)
    n = 4
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    print(f"Using random initial state: {initial_state}")

solution = None
while solution is None:
    solution = simulated_annealing(initial_state)
    if solution:
        print(f"Solution found: {solution}")
    else:
        print("Restarting with the current initial state...")
```

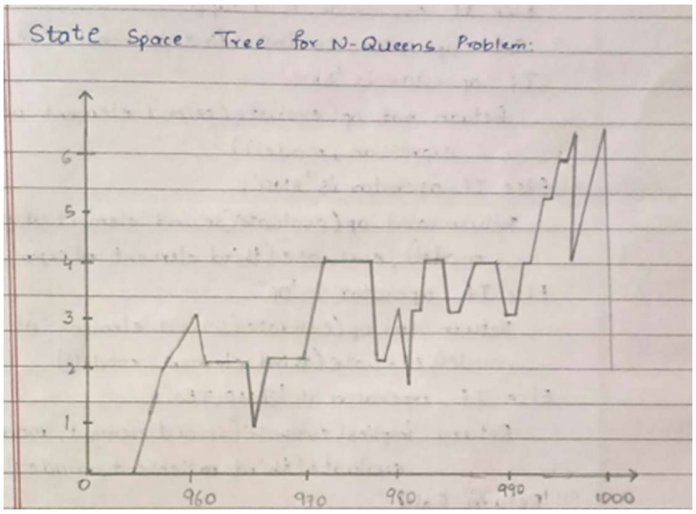
Output:

```
Enter the number of queens (N): 4
Enter the initial state as 4 integers: 3 1 2 0
Iteration 0: State: [3, 2, 2, 0], Cost: 4, T: 1000.00
Iteration 1: State: [3, 2, 0, 0], Cost: 4, T: 999.00
Iteration 2: State: [3, 0, 0, 0], Cost: 4, T: 998.00
Iteration 3: State: [3, 3, 0, 0], Cost: 3, T: 997.00
Iteration 4: State: [3, 3, 0, 2], Cost: 1, T: 996.00
Iteration 5: State: [0, 3, 0, 2], Cost: 1, T: 995.00
```

Iteration 6: State: [0, 3, 2, 2], Cost: 3, T: 994.01
Iteration 7: State: [0, 3, 0, 2], Cost: 1, T: 993.01
Iteration 8: State: [2, 3, 0, 2], Cost: 3, T: 992.01
Iteration 9: State: [2, 2, 0, 2], Cost: 4, T: 991.01
Iteration 10: State: [0, 2, 0, 2], Cost: 2, T: 990.01
Iteration 11: State: [0, 0, 0, 2], Cost: 4, T: 989.01
Iteration 12: State: [0, 0, 0, 3], Cost: 4, T: 988.01
Iteration 13: State: [0, 2, 0, 3], Cost: 2, T: 987.01
Iteration 14: State: [0, 0, 0, 3], Cost: 4, T: 986.01
Iteration 15: State: [1, 0, 0, 3], Cost: 2, T: 985.01
Iteration 16: State: [0, 0, 0, 3], Cost: 4, T: 984.02
Iteration 17: State: [0, 2, 0, 3], Cost: 2, T: 983.02
Iteration 18: State: [0, 2, 0, 0], Cost: 4, T: 982.02
Iteration 19: State: [0, 1, 0, 0], Cost: 5, T: 981.02
Iteration 20: State: [0, 1, 1, 0], Cost: 4, T: 980.02
Iteration 21: State: [0, 3, 1, 0], Cost: 2, T: 979.02
Iteration 22: State: [0, 3, 3, 0], Cost: 2, T: 978.02
Iteration 23: State: [0, 1, 3, 0], Cost: 2, T: 977.02
Iteration 24: State: [0, 3, 3, 0], Cost: 2, T: 976.02
Iteration 25: State: [0, 3, 1, 0], Cost: 2, T: 975.02
Iteration 26: State: [0, 3, 1, 3], Cost: 2, T: 974.03
Iteration 27: State: [0, 3, 1, 1], Cost: 2, T: 973.03
Iteration 28: State: [0, 3, 3, 1], Cost: 2, T: 972.03
Iteration 29: State: [0, 3, 3, 0], Cost: 2, T: 971.03
Iteration 30: State: [1, 3, 3, 0], Cost: 2, T: 970.03
Iteration 31: State: [1, 0, 3, 0], Cost: 3, T: 969.03
Iteration 32: State: [1, 0, 3, 2], Cost: 4, T: 968.03
Iteration 33: State: [1, 2, 3, 2], Cost: 5, T: 967.03
Iteration 34: State: [3, 2, 3, 2], Cost: 5, T: 966.03
Iteration 35: State: [3, 1, 3, 2], Cost: 2, T: 965.03
Iteration 36: State: [3, 3, 3, 2], Cost: 4, T: 964.04
Iteration 37: State: [3, 3, 3, 3], Cost: 6, T: 963.04
Iteration 38: State: [3, 3, 1, 3], Cost: 4, T: 962.04
Iteration 39: State: [3, 3, 1, 1], Cost: 4, T: 961.04
Iteration 40: State: [3, 3, 3, 1], Cost: 4, T: 960.04
Iteration 41: State: [3, 3, 2, 1], Cost: 4, T: 959.04
Iteration 42: State: [3, 2, 2, 1], Cost: 3, T: 958.04
Iteration 43: State: [2, 2, 2, 1], Cost: 4, T: 957.04
Iteration 44: State: [2, 2, 3, 1], Cost: 2, T: 956.04
Iteration 45: State: [2, 3, 3, 1], Cost: 3, T: 955.04
Iteration 46: State: [2, 1, 3, 1], Cost: 2, T: 954.05
Iteration 47: State: [0, 1, 3, 1], Cost: 2, T: 953.05
Iteration 48: State: [0, 1, 3, 2], Cost: 2, T: 952.05

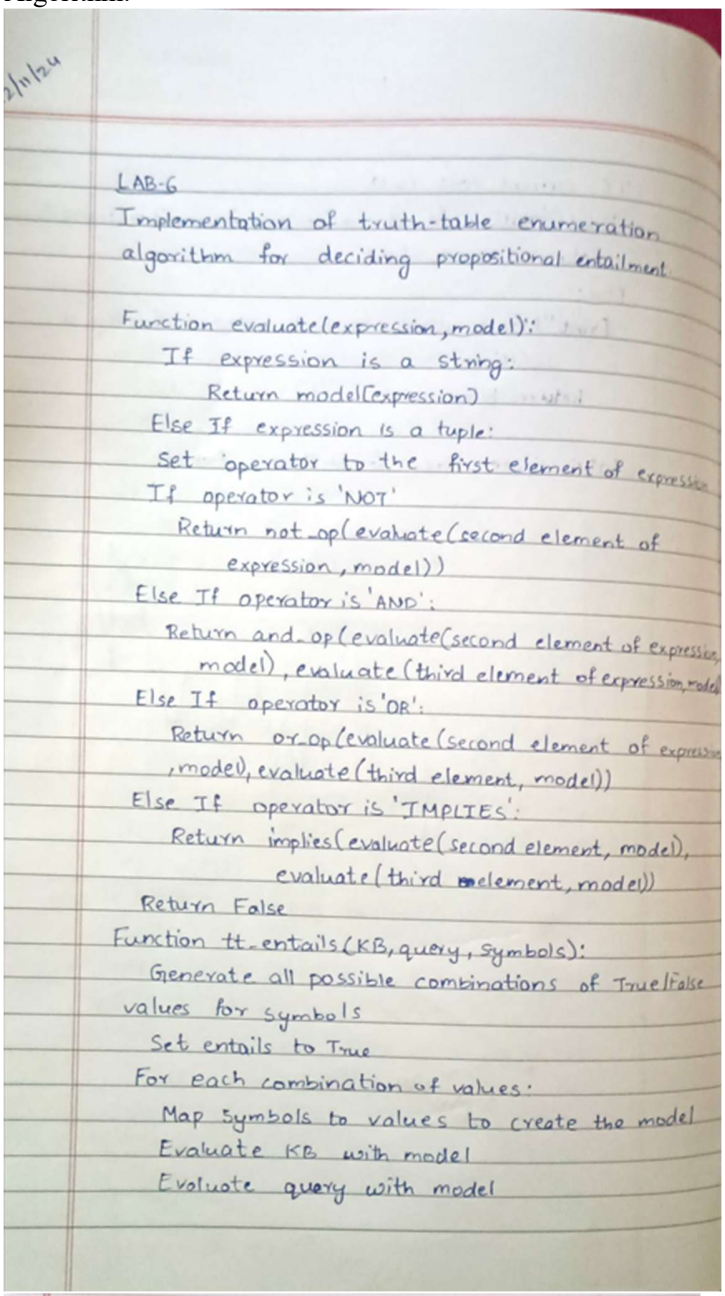
Iteration 49: State: [0, 3, 3, 2], Cost: 2, T: 951.05
Iteration 50: State: [0, 2, 3, 2], Cost: 3, T: 950.05
Iteration 51: State: [1, 2, 3, 2], Cost: 5, T: 949.05
Iteration 52: State: [1, 3, 3, 2], Cost: 3, T: 948.05
Iteration 53: State: [1, 3, 3, 3], Cost: 4, T: 947.05
Iteration 54: State: [1, 3, 0, 3], Cost: 1, T: 946.05
Iteration 55: State: [1, 2, 0, 3], Cost: 1, T: 945.05
Iteration 56: State: [1, 0, 0, 3], Cost: 2, T: 944.06
Iteration 57: State: [1, 0, 3, 3], Cost: 3, T: 943.06
Iteration 58: State: [1, 0, 0, 3], Cost: 2, T: 942.06
Iteration 59: State: [2, 0, 0, 3], Cost: 2, T: 941.06
Iteration 60: State: [0, 0, 0, 3], Cost: 4, T: 940.06
Iteration 61: State: [0, 1, 0, 3], Cost: 5, T: 939.06
Iteration 62: State: [3, 1, 0, 3], Cost: 3, T: 938.06
Iteration 63: State: [3, 1, 2, 3], Cost: 4, T: 937.06
Iteration 64: State: [3, 1, 2, 1], Cost: 3, T: 936.06
Iteration 65: State: [3, 1, 3, 1], Cost: 2, T: 935.07
Iteration 66: State: [3, 1, 3, 0], Cost: 2, T: 934.07
Iteration 67: State: [3, 3, 3, 0], Cost: 4, T: 933.07
Iteration 68: State: [3, 3, 3, 2], Cost: 4, T: 932.07
Iteration 69: State: [3, 3, 0, 2], Cost: 1, T: 931.07
Iteration 70: State: [3, 3, 0, 1], Cost: 3, T: 930.07
Iteration 71: State: [3, 3, 0, 2], Cost: 1, T: 929.07
Iteration 72: State: [3, 3, 1, 2], Cost: 3, T: 928.07
Iteration 73: State: [3, 3, 0, 2], Cost: 1, T: 927.07
Iteration 74: State: [1, 3, 0, 2], Cost: 0, T: 926.07
Solution found within maximum iterations.
Solution found: [1, 3, 0, 2]

State Space Tree:



Program 7: Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

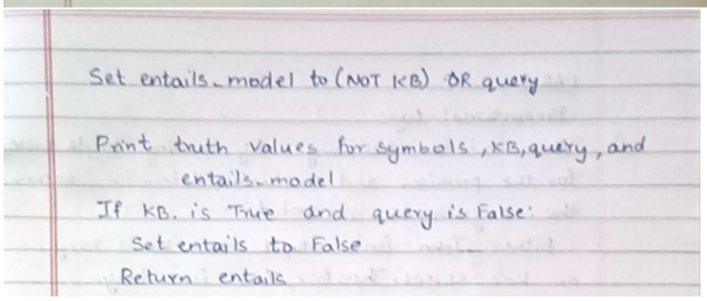
Algorithm:



LAB-6

Implementation of truth-table enumeration algorithm for deciding propositional entailment

```
Function evaluate(expression, model):
    If expression is a string:
        Return model[expression]
    Else If expression is a tuple:
    Set operator to the first element of expression
    If operator is 'NOT'
      Return not_op(evaluate(second element of
          expression, model))
    Else If operator is 'AND':
      Return and_op(evaluate(second element of expression,
        model), evaluate(third element of expression, model))
    Else If operator is 'OR':
      Return or_op(evaluate(second element of expression
      ,model), evaluate(third element, model))
    Else If operator is 'IMPLIES':
      Return implies(evaluate(second element, model),
            evaluate(third element, model))
    Return False
Function tt_entails(KB, query, symbols):
    Generate all possible combinations of True/False
  values for symbols
    Set entails to True
    For each combination of values:
      Map symbols to values to create the model
      Evaluate KB with model
      Evaluate query with model
```



```
    Set entails_model to (NOT KB) OR query

    Print truth values for symbols, KB, query, and
        entails_model
    If KB. is True and query is False:
      Set entails to False
    Return entails
```

Code:

```
from itertools import product

def implies(p, q):
    return not p or q

def and_op(p, q):
    return p and q

def or_op(p, q):
    return p or q

def not_op(p):
    return not p

def evaluate(expr, model):
    if isinstance(expr, str):
        return model[expr]
    elif isinstance(expr, tuple):
        operator = expr[0]
        if operator == 'NOT':
            return not_op(evaluate(expr[1], model))
        elif operator == 'AND':
            return and_op(evaluate(expr[1], model), evaluate(expr[2], model))
        elif operator == 'OR':
            return or_op(evaluate(expr[1], model), evaluate(expr[2], model))
        elif operator == 'IMPLIES':
            return implies(evaluate(expr[1], model), evaluate(expr[2], model))
    return False

def tt_entails_with_truth_table(kb, query, symbols):
    all_models = list(product([True, False], repeat=len(symbols)))
    symbols = list(symbols)
    entails = True

    print(f"{' '.join([f'{s:^5}' for s in symbols])} {'KB':^10} {'Query':^10} {'KB implies Query'}")
    print("-" * (6 * len(symbols) + 30))

    for values in all_models:
        model = dict(zip(symbols, values))

        kb_true = evaluate(kb, model)
        query_true = evaluate(query, model)
```

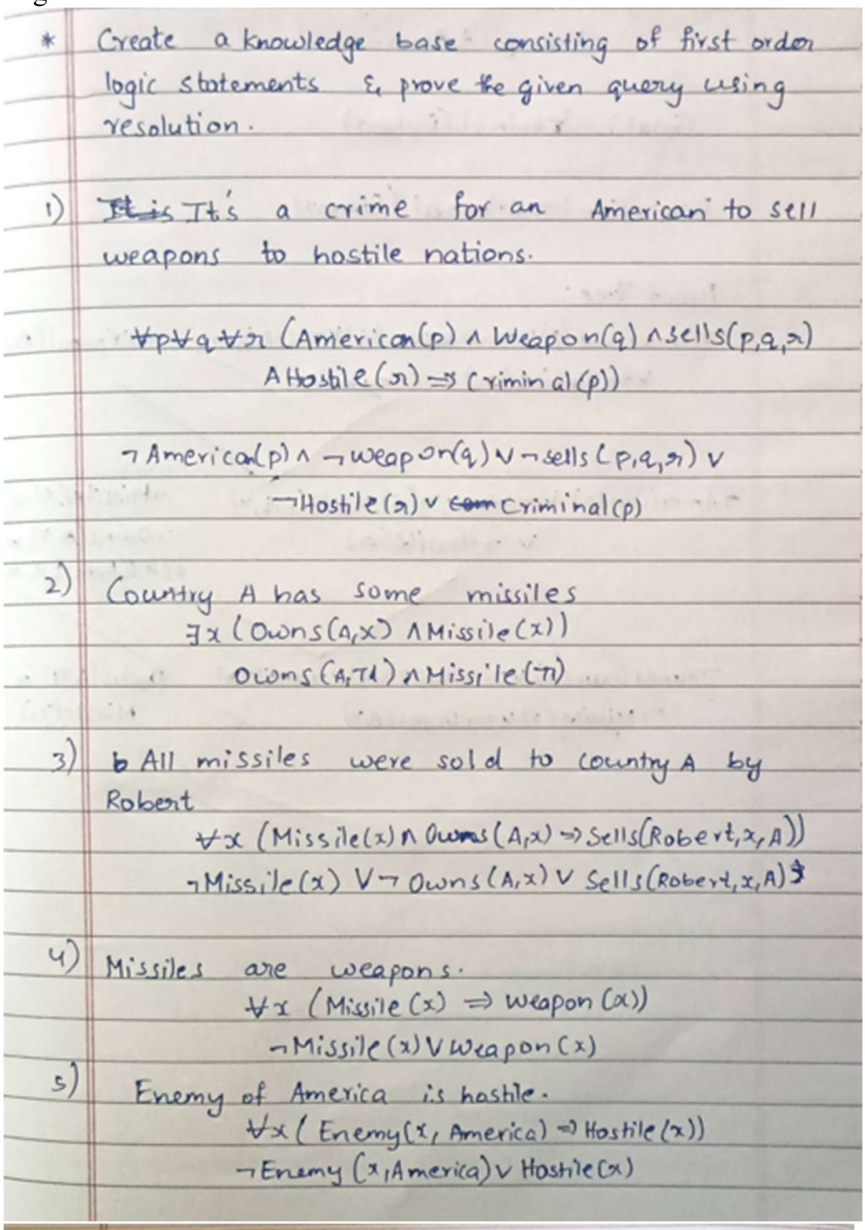
```
        entails_model = not kb_true or query_true

        row = ' '.join([f"{model[s]!s:^5}" for s in symbols])
        print(f"{row} {kb_true!s:^10} {query_true!s:^10} {entails_model!s:^15}")

        if kb_true and not query_true:
            entails = False
    return entails

def parse_expression(input_str):
    """Parse a user input string into a tuple-based logical expression."""
    tokens = input_str.replace('(', '').replace(')', '').split()
    if len(tokens) == 2 and tokens[0] == 'NOT':
        return ('NOT', tokens[1].strip())
    elif len(tokens) == 3:
        op = tokens[1].strip()
        left = tokens[0].strip()
        right = tokens[2].strip()
        if op == 'AND':
            return ('AND', left, right)
        elif op == 'OR':
            return ('OR', left, right)
        elif op == 'IMPLIES':
            return ('IMPLIES', left, right)
    return input_str.strip()

symbols = input("Enter the symbols separated by commas (e.g., A, B, C): ").replace(" ", "").split(",")
kb_input = input("Enter the KB expression (use AND, OR, IMPLIES, NOT): ")
query_input = input("Enter the Query expression (use AND, OR, IMPLIES, NOT): ")
kb = parse_expression(kb_input)
query = parse_expression(query_input)
result = tt_entails_with_truth_table(kb, query, symbols)
print("\nFinal result:")
print(f"Does the KB entail the query? {'Yes' if result else 'No'}")
```

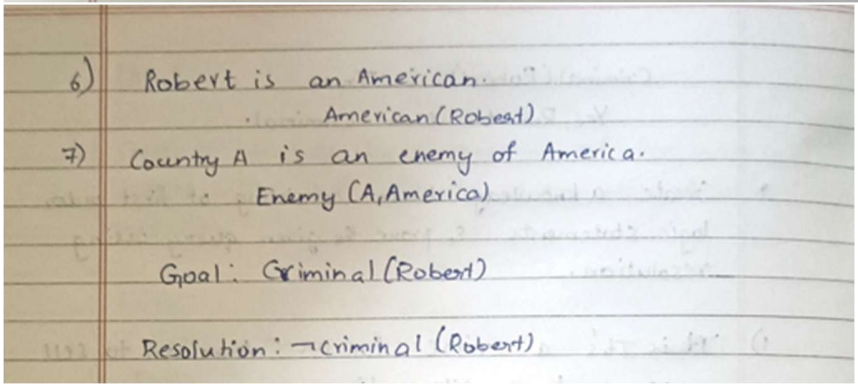
Output:

```
Enter the symbols separated by commas (e.g., A, B, C): A,B
Enter the KB expression (use AND, OR, IMPLIES, NOT): A IMPLIES B
Enter the Query expression (use AND, OR, IMPLIES, NOT): B
  A     B       KB       Query    KB implies Query
-----------------------------------------
True  True     True       True         True
True  False   False      False         True
False True     True       True         True
False False    True      False         False

Final result:
Does the KB entail the query? No
```

Program 8: Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:





Code:

```
class ResolutionKB:
    def __init__(self):
        self.clauses = []

    def add_clause(self, clause):
        self.clauses.append(set(clause))

    def negate_query(self, query):
        return {"-" + literal if not literal.startswith("-") else literal[1:] for literal in query}

    def resolve(self, clause1, clause2):
        resolvent = set()
        for literal in clause1:
            if "-" + literal in clause2 or literal[1:] in clause2:
                resolvent = (clause1 | clause2) - {literal, "-" + literal}
                return resolvent
        return None

    def resolution(self, query):
        clauses = self.clauses[:]
        clauses.append(self.negate_query(query))
        while True:
            new_clauses = []
            for i, clause1 in enumerate(clauses):
                for clause2 in clauses[i + 1:]:
                    resolvent = self.resolve(clause1, clause2)
                    if resolvent is not None:
                        if not resolvent:
                            return True
                        new_clauses.append(resolvent)
            if not new_clauses:
                return False
            clauses.extend(new_clauses)

def input_clauses():
    kb = ResolutionKB()
    kb.add_clause(["American(Robert)"])
    kb.add_clause(["-American(Robert)", "-SellsWeapon(Robert, T1)", "-Enemy(A, America)",
"Criminal(Robert)"])
    kb.add_clause(["SellsWeapon(Robert, T1)"])
    kb.add_clause(["Enemy(A, America)"])
    kb.add_clause(["Missile(T1)"])
    return kb
```

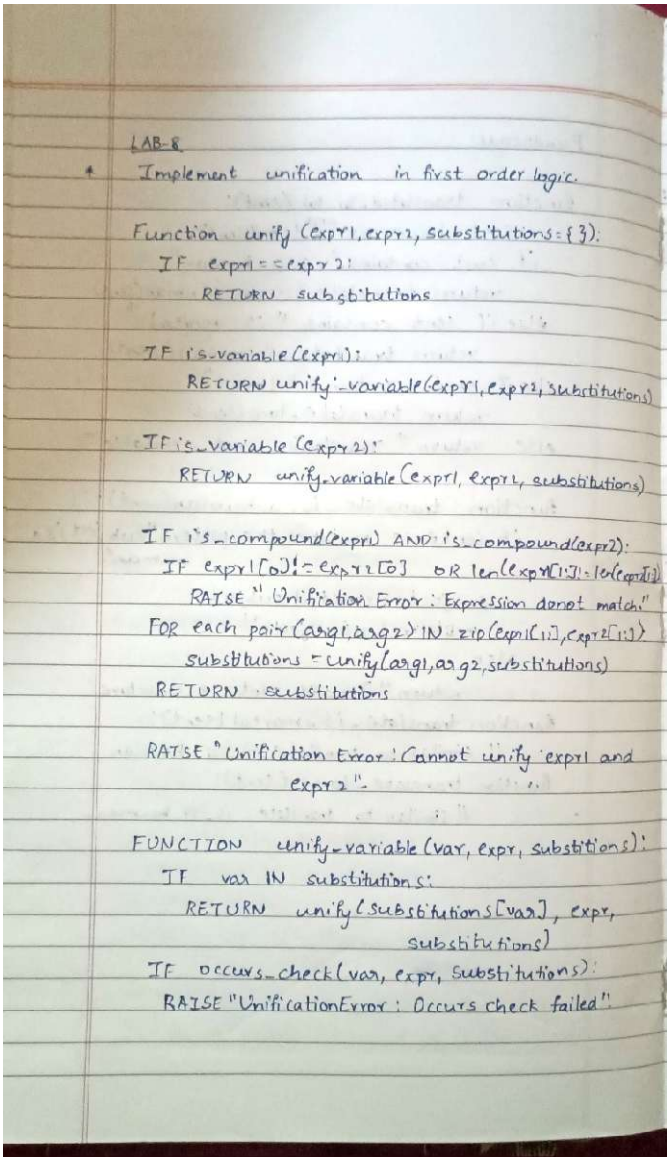
```
def main_resolution():
    kb = input_clauses()
    query = {"Criminal(Robert)"}
    if kb.resolution(query):
        print("\nYes, Robert is a criminal.")
    else:
        print("\nNo, Robert is not a criminal.")

if __name__ == "__main__":
    main_resolution()
```

Output:

Yes, Robert is a criminal.

Program 9: Implement unification in first order logic.

Algorithm:



LAB-8

* Implement unification in first order logic.

Function unify (expr1, expr2, substitutions={}):
IF expr1 == expr2:
RETURN substitutions

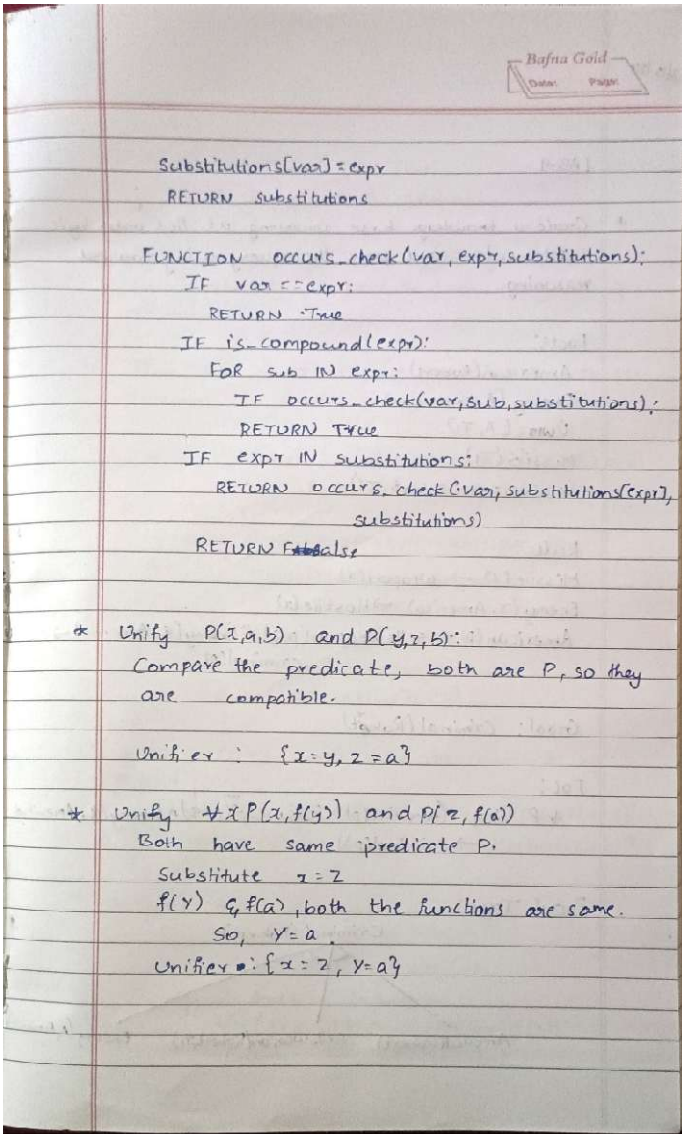
IF is_variable(expr1):
RETURN unify_variable(expr1, expr2, substitutions)

IF is_variable (expr2):
RETURN unify_variable (expr1, expr2, substitutions)

IF is_compound(expr1) AND is_compound(expr2):
IF expr1[0] != expr2[0] OR len(expr1[1:]) != len(expr2[1:])
RAISE "Unification Error: Expression donot match."
FOR each pair (arg1, arg2) IN zip(expr1[1:], expr2[1:])
substitutions = unify(arg1, arg2, substitutions)
RETURN substitutions

RAISE "Unification Error: Cannot unify expr1 and expr2".

FUNCTION unify_variable (var, expr, substitutions):
IF var IN substitutions:
RETURN unify(substitutions[var], expr, substitutions)
IF occurs_check(var, expr, substitutions):
RAISE "UnificationError: Occurs check failed"



Bafna Gold

substitutions[var] = expr
RETURN substitutions

FUNCTION occurs_check(var, expr, substitutions):
IF var == expr:
RETURN True
IF is_compound(expr):
FOR sub IN expr:
IF occurs_check(var, sub, substitutions):
RETURN True
IF expr IN substitutions:
RETURN occurs_check(var, substitutions[expr], substitutions)
RETURN False

* Unify P(x,a,b) and P(y,z,b):
Compare the predicate, both are P, so they are compatible.

Unifier : {x=y, z=a}

* Unify ∀x P(x, f(y)) and P(z, f(a))
Both have same predicate P.
Substitute x = z
f(y) & f(a), both the functions are same.
So, Y = a.
Unifier : {x = z, Y = a}

Code:

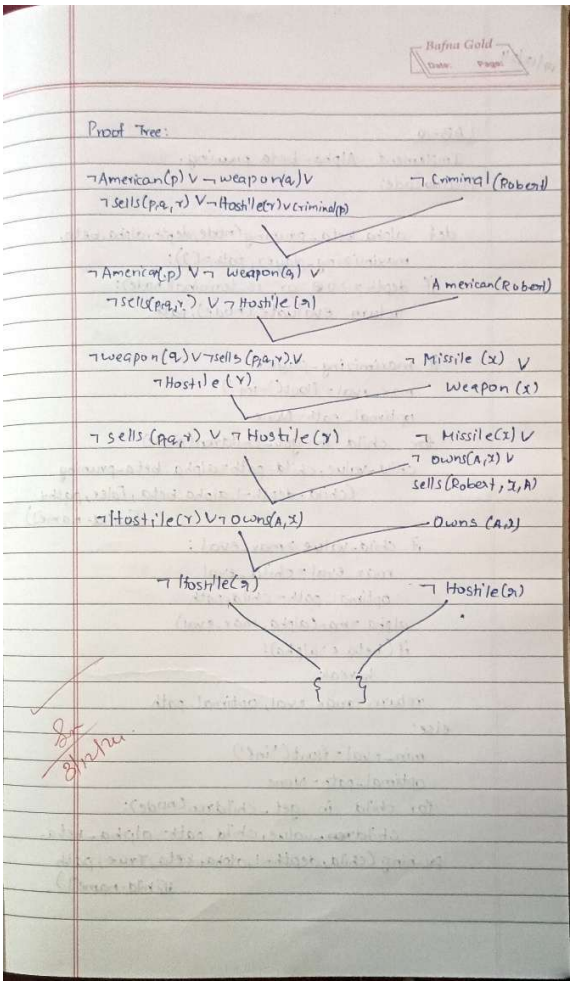
```
class UnificationError(Exception):
    pass

def unify(expr1, expr2, substitutions=None):
    if substitutions is None:
        substitutions = {}
```

```
    if expr1 == expr2:
        return substitutions

    if is_variable(expr1):
        return unify_variable(expr1, expr2, substitutions)

    if is_variable(expr2):
        return unify_variable(expr2, expr1, substitutions)

    if is_compound(expr1) and is_compound(expr2):
        if expr1[0] != expr2[0] or len(expr1[1:]) != len(expr2[1:]):
            raise UnificationError("Expressions do not match.")
        for arg1, arg2 in zip(expr1[1:], expr2[1:]):
            substitutions = unify(arg1, arg2, substitutions)
        return substitutions

    raise UnificationError("Cannot unify expr1 and expr2.")

def unify_variable(var, expr, substitutions):
    if var in substitutions:
        return unify(substitutions[var], expr, substitutions)

    if occurs_check(var, expr, substitutions):
        raise UnificationError("Occurs check failed.")

    substitutions[var] = expr
    return substitutions

def occurs_check(var, expr, substitutions):
    if var == expr:
        return True

    if is_compound(expr):
        for sub in expr[1:]:
            if occurs_check(var, sub, substitutions):
                return True

    if expr in substitutions:
        return occurs_check(var, substitutions[expr], substitutions)

    return False

def is_variable(expr):
```

```
    return isinstance(expr, str) and expr.islower()

def is_compound(expr):
    return isinstance(expr, list) and len(expr) > 0

try:
    expr1 = ['f', 'x', 'y']
    expr2 = ['f', 'a', 'b']
    substitutions = unify(expr1, expr2)
    print("Substitutions:", substitutions)
except UnificationError as e:
    print("Unification failed:", e)
```

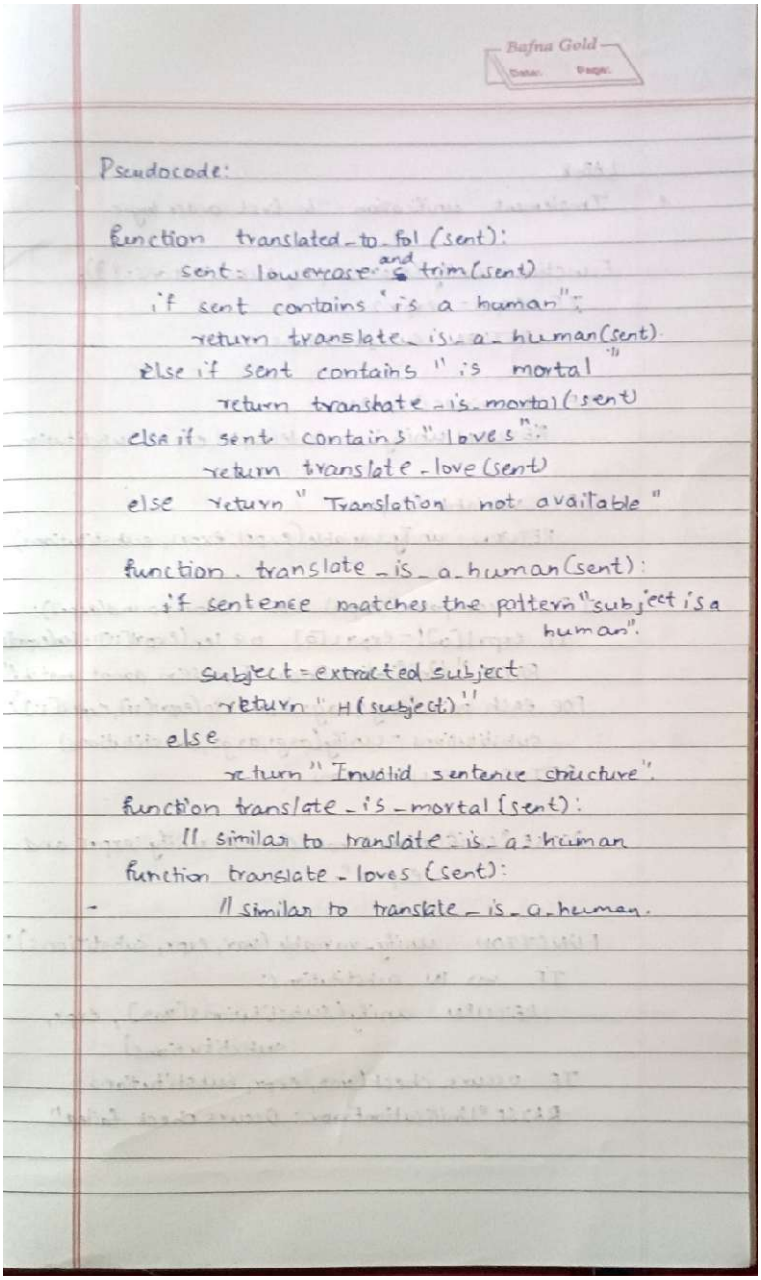
Output:
Substitutions: {'x': 'a', 'y': 'b'}

State Space Tree:



Program 10: Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



Pseudocode:

```
function translated_to_fol(sent):
    sent = lowercase and trim(sent)
    if sent contains 'is a human":
        return translate_is_a_human(sent)
    else if sent contains "is mortal"
        return translate_is_mortal(sent)
    else if sent contains "loves":
        return translate_love(sent)
    else return "Translation not available"

function translate_is_a_human(sent):
    if sentence matches the pattern "subject is a human":
        subject = extracted subject
        return "H(subject)"
    else
        return "Invalid sentence structure"
function translate_is_mortal(sent):
    // similar to translate_is_a_human
function translate_loves(sent):
    // similar to translate_is_a_human.
```

Code:

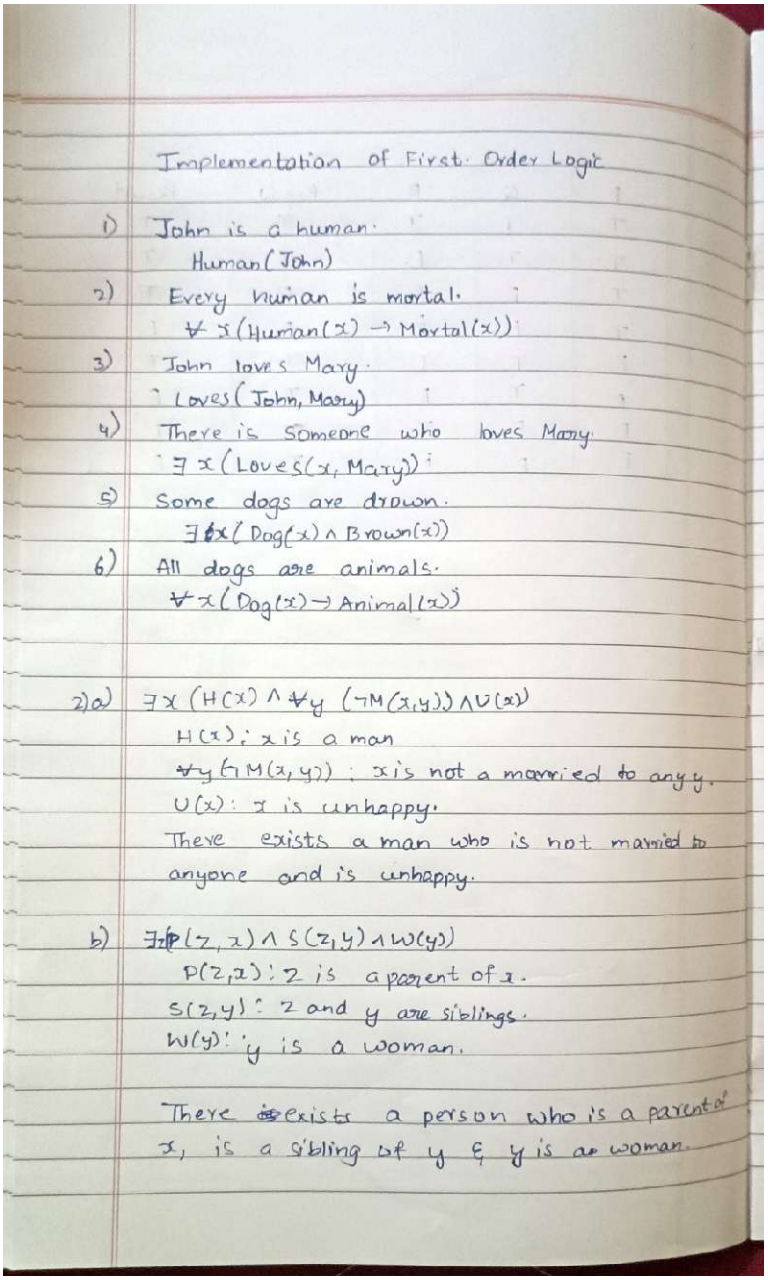
```
import re

def transform_to_fol(sentence):
    # Universal quantifiers: "Every X is Y" or "All X are Y"
    if re.match(r"(Every|All) (\w+) (is|are) (\w+)", sentence, re.IGNORECASE):
        subject, predicate = re.findall(r"(?:Every|All) (\w+) (?:is|are) (\w+)", sentence,
re.IGNORECASE)[0]
        return f"∀x ({subject.capitalize()}(x) → {predicate.capitalize()}(x))"

    # Existential quantifiers: "There is someone who X Y"
    elif re.match(r"There is someone who (\w+) (\w+)", sentence, re.IGNORECASE):
        action, target = re.findall(r"There is someone who (\w+) (\w+)", sentence, re.IGNORECASE)[0]
        return f"∃x ({action.capitalize()}(x, {target.capitalize()}))"

    # Negations: "There is no X who is Y"
    elif re.match(r"There is no (\w+) who is (\w+)", sentence, re.IGNORECASE):
        subject, predicate = re.findall(r"There is no (\w+) who is (\w+)", sentence,
re.IGNORECASE)[0]
        return f"¬∃x ({subject.capitalize()}(x) ∧ {predicate.capitalize()}(x))"

    # Simple relationships: "X loves Y"
    elif re.match(r"(\w+) (\w+) (\w+)", sentence):
        subject, action, target = re.findall(r"(\w+) (\w+) (\w+)", sentence)[0]
        return f"{action.capitalize()}({subject.capitalize()}, {target.capitalize()})"

    # Conditional statements: "If it is X, then the Y is Z"
    elif re.match(r"If it is (\w+), then the (\w+) is (\w+)", sentence, re.IGNORECASE):
        condition, subject, predicate = re.findall(r"If it is (\w+), then the (\w+) is (\w+)", sentence,
re.IGNORECASE)[0]
        return f"({condition.capitalize()} → {predicate.capitalize()}({subject.capitalize()}))"

    # Complex relationships: "John and Mary are both students."
    elif re.match(r"(\w+) and (\w+) are both (\w+)", sentence, re.IGNORECASE):
        person1, person2, role = re.findall(r"(\w+) and (\w+) are both (\w+)", sentence,
re.IGNORECASE)[0]
        return f"{role.capitalize()}({person1.capitalize()}) ∧ {role.capitalize()}({person2.capitalize()})"

    # Nested quantifiers: "There is a person who knows every other person."
    elif re.match(r"There is a (\w+) who (\w+) every (\w+)", sentence, re.IGNORECASE):
        subject, action, target = re.findall(r"There is a (\w+) who (\w+) every (\w+)", sentence,
re.IGNORECASE)[0]
        return f"∃x ∀y (x ≠ y → {action.capitalize()}(x, y))"
```

```
    # Reflexive statements: "Nobody is taller than themselves."
    elif re.match(r"Nobody is (\w+) than themselves", sentence, re.IGNORECASE):
        predicate = re.findall(r"Nobody is (\w+) than themselves", sentence, re.IGNORECASE)[0]
        return f"∀x ¬{predicate.capitalize()}(x, x)"

    # Fallback for unsupported patterns
    return "Unable to transform sentence into FOL. Please enter a valid sentence pattern."

def main():
    print("Enter a sentence to convert into First-Order Logic (FOL).")
    print("Type 'exit' to quit.\n")

    while True:
        user_sentence = input("Enter your sentence: ").strip()

        if user_sentence.lower() == "exit":
            print("Exiting the program. Goodbye!")
            break

        fol_representation = transform_to_fol(user_sentence)
        print(f"FOL Representation:\n{fol_representation}\n")

if __name__ == "__main__":
    main()
```

Output:

```
Enter a sentence to convert into First-Order Logic (FOL).
Type 'exit' to quit.

Enter your sentence: there is someone who loves mary
FOL Representation:
∃x (Loves(x, Mary))

Enter your sentence: John loves Mary
FOL Representation:
Loves(John, Mary)
```

State Space Tree:



Implementation of First Order Logic

1) John is a human.
   Human(John)

2) Every human is mortal.
   $\forall x (Human(x) \rightarrow Mortal(x))$

3) John loves Mary.
   Loves(John, Mary)

4) There is someone who loves Mary.
   $\exists x (Loves(x, Mary))$

5) Some dogs are brown.
   $\exists x (Dog(x) \wedge Brown(x))$

6) All dogs are animals.
   $\forall x (Dog(x) \rightarrow Animal(x))$

2)a) $\exists x (H(x) \wedge \forall y (\neg M(x,y)) \wedge U(x)$
   $H(x)$: x is a man
   $\forall y \neg M(x,y)$: x is not a married to any y.
   $U(x)$: x is unhappy.
   There exists a man who is not married to anyone and is unhappy.

b) $\exists z (P(z,x) \wedge S(z,y) \wedge W(y))$
   $P(z,x)$: z is a parent of x.
   $S(z,y)$: z and y are siblings.
   $W(y)$: y is a woman.

   There exists a person who is a parent of x, is a sibling of y & y is a woman.

Program 11: Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

3/12/24

LAB-9

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

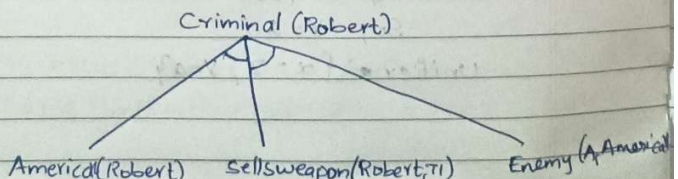
Facts:
American(Robert)
Enemy(A, American)
Owns(A, T1)
Missiles(T1)
SellsWeapon(Robert, T1)

Rules:
Missile(x) → weapon(x)
Enemy(x, America) → Hostile(x)
American(p) ∧ sellsweapon(p,q) ∧ Enemy(q, America) → Criminal(p)

Goal: Criminal(Robert)

FoL:
∀p ∀q (American(p) ∧ sellsweapon(p, q) ∧ Enemy(q, America) → criminal(p)).

Proof Tree:



Code:

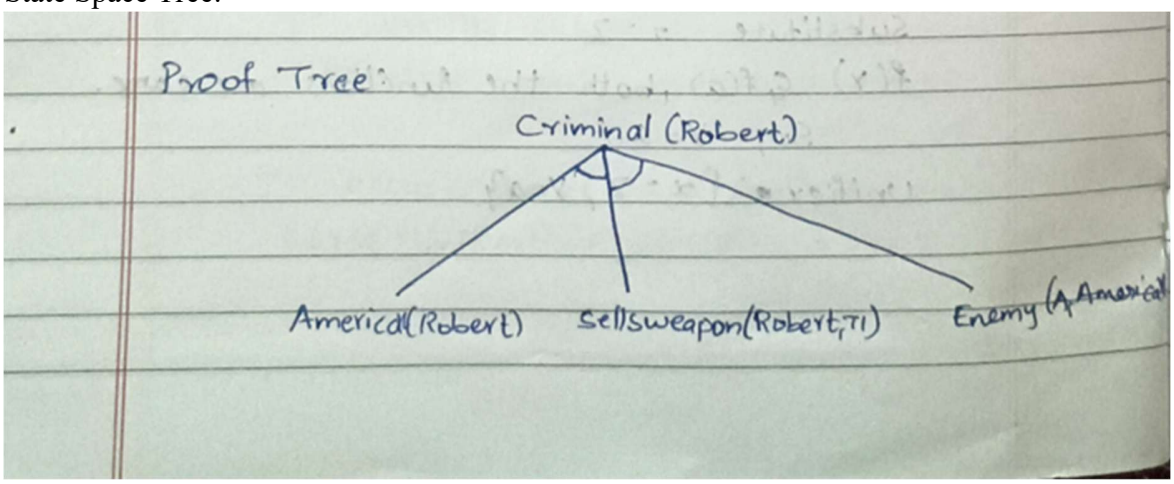
```
class FirstOrderLogicKB:
    def __init__(self):
        self.facts = set()
        self.rules = []

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, premise, conclusion):
        self.rules.append((premise, conclusion))

    def forward_reasoning(self):
        new_facts = set(self.facts)
        inferred = True

        while inferred:
            inferred = False
            for premise, conclusion in self.rules:
                if premise.issubset(new_facts) and conclusion not in new_facts:
                    new_facts.add(conclusion)
                    inferred = True
        self.facts = new_facts

    def query(self, fact):
        return fact in self.facts

    def display_facts(self):
        print("Known Facts:")
        for fact in self.facts:
            print(fact)

def input_facts_and_rules():
    kb = FirstOrderLogicKB()

    kb.add_fact("American(Robert)")
    kb.add_fact("Enemy(A, America)")
    kb.add_fact("Owns(A, T1)")
    kb.add_fact("Missile(T1)")
    kb.add_fact("SellsWeapon(Robert, T1)")

    rule_premise = {"American(Robert)", "SellsWeapon(Robert, T1)", "Enemy(A, America)"}
    rule_conclusion = "Criminal(Robert)"
    kb.add_rule(rule_premise, rule_conclusion)
```

```
    kb.forward_reasoning()

    return kb

def main():
    kb = input_facts_and_rules()

    kb.display_facts()

    if kb.query("Criminal(Robert)"):
        print("\nYes, Robert is a criminal.")
    else:
        print("\nNo, Robert is not a criminal.")

if __name__ == "__main__":
    main()
```

Output:

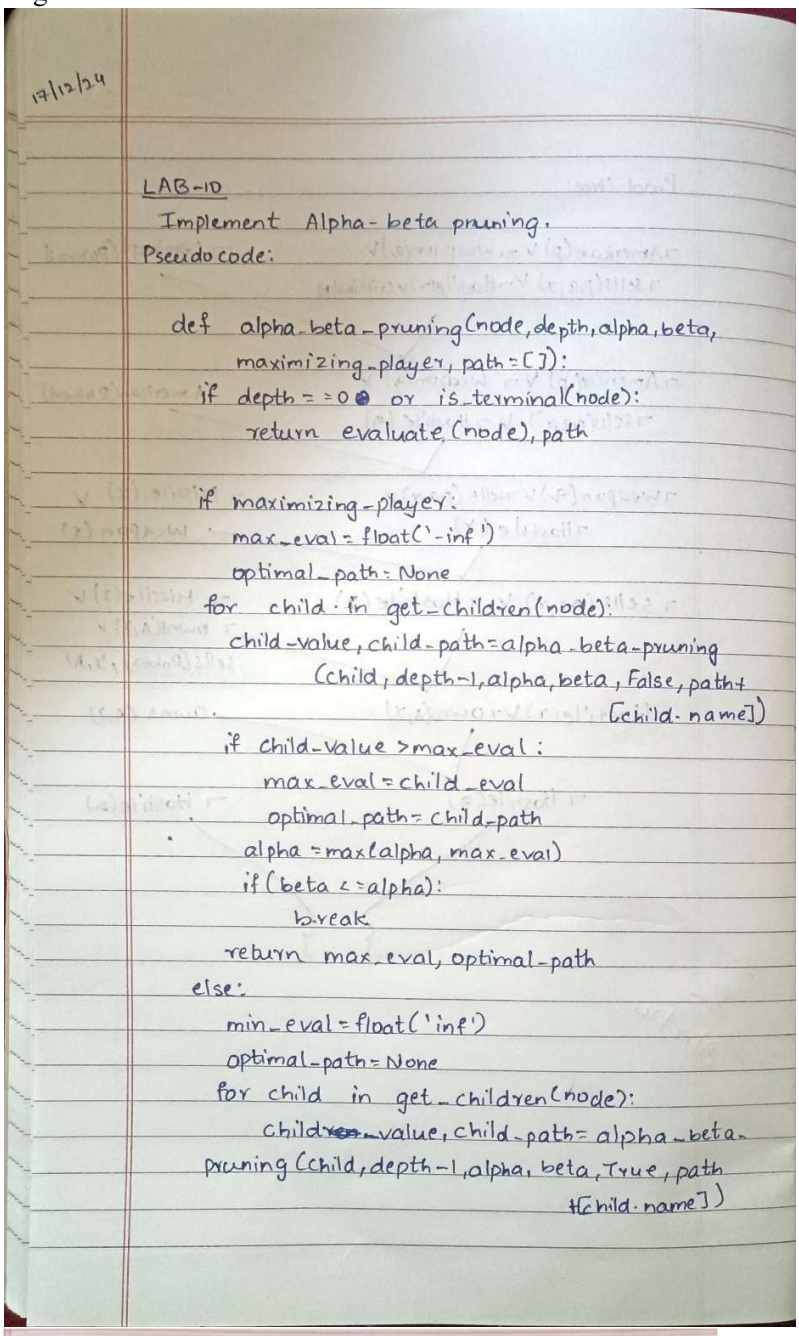
```
Known Facts:
Enemy(A, America)
Missile(T1)
SellsWeapon(Robert, T1)
Owns(A, T1)
American(Robert)
Criminal(Robert)

Yes, Robert is a criminal.
```

State Space Tree:



Program 12: Implement Alpha-Beta Pruning.

Algorithm:



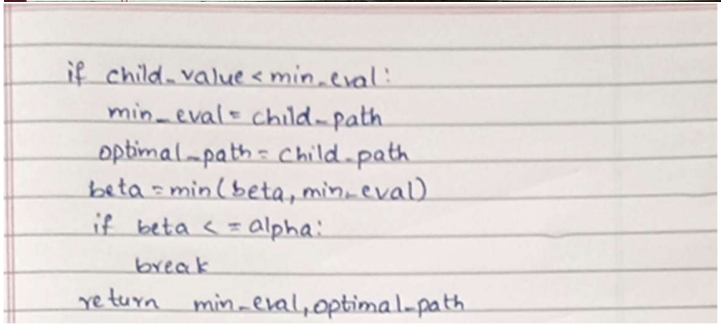
17/12/24

LAB-10

Implement Alpha-beta pruning.

Pseudo code:

```
def alpha_beta_pruning(node, depth, alpha, beta,
    maximizing_player, path=[]):
  if depth == 0 or is_terminal(node):
    return evaluate(node), path

  if maximizing_player:
    max_eval = float('-inf')
    optimal_path = None
    for child in get_children(node):
      child_value, child_path = alpha_beta_pruning
          (child, depth-1, alpha, beta, False, path+
                                    [child.name])
      if child_value > max_eval:
        max_eval = child_eval
        optimal_path = child_path
      alpha = max(alpha, max_eval)
      if (beta <= alpha):
        break
    return max_eval, optimal_path
  else:
    min_eval = float('inf')
    optimal_path = None
    for child in get_children(node):
      child_value, child_path = alpha_beta_
    pruning (child, depth-1, alpha, beta, True, path
                                    +[child.name])
```



```
if child_value < min_eval:
  min_eval = child_path
  optimal_path = child_path
 beta = min(beta, min_eval)
 if beta <= alpha:
   break
return min_eval, optimal_path
```

Code:

```
0 class Node:
    def __init__(self, name, children=None, value=None):
        self.name = name
        self.children = children if children is not None else []
        self.value = value

def evaluate(node):
    return node.value

def is_terminal(node):
    return node.value is not None

def get_children(node):
    return node.children

def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player, path=[]):
    # Terminal condition: leaf node or depth is 0
    if depth == 0 or is_terminal(node):
        return evaluate(node), path

    if maximizing_player:
        max_eval = float('-inf')
        optimal_path = None
        for child in get_children(node):
            child_value, child_path = alpha_beta_pruning(child, depth - 1, alpha, beta, False, path +
[child.name])
            if child_value > max_eval:
                max_eval = child_value
                optimal_path = child_path
            alpha = max(alpha, max_eval)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval, optimal_path
    else:
        min_eval = float('inf')
        optimal_path = None
        for child in get_children(node):
            child_value, child_path = alpha_beta_pruning(child, depth - 1, alpha, beta, True, path +
[child.name])
            if child_value < min_eval:
                min_eval = child_value
                optimal_path = child_path
            beta = min(beta, min_eval)
            if beta <= alpha:
```

```
            break  # Alpha cut-off
        return min_eval, optimal_path

# Create the game tree
H = Node('H', value=10)
I = Node('I', value=9)
J = Node('J', value=14)
K = Node('K', value=18)
L = Node('L', value=5)
M = Node('M', value=4)
N = Node('N', value=50)
O = Node('O', value=3)

D = Node('D', children=[H, I])
E = Node('E', children=[J, K])
F = Node('F', children=[L, M])
G = Node('G', children=[N, O])

B = Node('B', children=[D, E])
C = Node('C', children=[F, G])

A = Node('A', children=[B, C])

# Run the alpha-beta pruning algorithm
maximizing_player = True
initial_alpha = float('-inf')
initial_beta = float('inf')
depth = 3  # Maximum depth of the tree

optimal_value, optimal_path = alpha_beta_pruning(A, depth, initial_alpha, initial_beta,
maximizing_player)
print(f"The optimal value is: {optimal_value}")
print(f"The optimal path is: A -> {' -> '.join(optimal_path)}")
```

Output:

```
The optimal value is: 10
The optimal path is: A -> B -> D -> H
```