VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Navya Billalar (1BM22CS175)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

Prof. Swathi Sridharan
Assistant Professor
Department of Computer Science and Engineering

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried outby Navya Billalar **(1BM22CS175),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Prof. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Tic-Tac-Toe (Lab 1: 24-09-2024)

Observation Book:

Code:

```
~~Function~~ def print_board (board):
    print (f"{board [0]} | {board [1]} | {board [2]}")
    print ("--+--+--")
    print (f"{board [3]} | {board [4]} | {board [5]}")
    print ("--+--+--")
    print (f"{board [6]} | {board [7]} | {board [8]}")

def check_winner (board, player):
    win_conditions = [(0,1,2), (3,4,5), (6,7,8),
        (0,3,6), (1,4,7), (2,5,8),
        (0,4,8), (2,4,6)]
    return any (board [a] == board [b] ==
      board [c] == player for a,b,c in
        win_conditions)

def computer_move (board):
    for i in range (9):
        if board [i] == '':
            board [i] = 'O'
            if check_winner (board, 'O'):
                return i
            board [i] = ''
~~for~~
for i in range (9):
    if board [i] == '':
        board [i] = 'X'
        if check_winner (board, 'X'):
            board [i] = 'O'
            return i
        board [i] = ''
```

```
# // any move
    for i in range (9):
        if board [i] == ' ':
            return i

def play game():
    board = [ " for _ in range (9)]
    current _ player = 'x'
    while true :
        print _ board (board)
    if current _ player == 'x':
    try:
        move = int (input (" Choose a position")) -1

        except ValueError:
        print ("Invalid input")
        continue.
    if move < 0 or move > 8 :
        print ("invalid move")
    if board [move]! = '':
        print ("position taken")
        continue.
    else:
        move = computer _ move (board)
    board [move] = current _ player
    if check _ winner (board, current - player)
    print_ board (board)
    print (f "player {current_ player} win!")
    break.

if is_ board_ full (board)
    print_ board (board)
    print (" Tie! ").
    break
play-game()
```

o/p ?

Output Screenshots:

Player X winning:

```
  |   |  
--+---+--
  |   |  
--+---+--
  |   |  
Player X, choose a position (1-9): 9
  |   |  
--+---+--
  |   |  
--+---+--
  |   | X
Computer's turn (O)...
  |   |  
--+---+--
  | O |  
--+---+--
  |   | X
Player X, choose a position (1-9): 1
X |   |  
--+---+--
  | O |  
--+---+--
  |   | X
Computer's turn (O)...
X |   | O
--+---+--
  | O |  
--+---+--
  |   | X
```

```
Player X, choose a position (1-9): 7
X |   | O
--+---+--
  | O |  
--+---+--
X |   | X
Computer's turn (O)...
X |   | O
--+---+--
O | O |  
--+---+--
X |   | X
Player X, choose a position (1-9): 8
X |   | O
--+---+--
O | O |  
--+---+--
X | X | X
Player X wins!
```

Player O winning (Computer winning):

```
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, choose a position (1-9): 2
  | X |
--+---+--
  |   |
--+---+--
  |   |
Computer's turn (O)...
  | X |
--+---+--
  | O |
--+---+--
  |   |
Player X, choose a position (1-9): 3
  | X | X
--+---+--
  | O |
--+---+--
  |   |
```

```
Computer's turn (O)...
O | X | X
--+---+--
  | O |
--+---+--
  |   |
Player X, choose a position (1-9): 6
O | X | X
--+---+--
  | O | X
--+---+--
  |   |
Computer's turn (O)...
O | X | X
--+---+--
  | O | X
--+---+--
  |   | O
Player O wins!
```

Tie:

```
  |   |
--+---+--
  |   |
--+---+--
  |   |
Player X, choose a position (1-9): 7
  |   |
--+---+--
  |   |
--+---+--
X |   |
Computer's turn (O)...
  |   |
--+---+--
  | O |
--+---+--
X |   |
Player X, choose a position (1-9): 1
X |   |
--+---+--
  | O |
--+---+--
X |   |
Computer's turn (O)...
X |   |
--+---+--
O | O |
--+---+--
X |   |
```

```
Player X, choose a position (1-9): 6
X |   |
--+---+--
O | O | X
--+---+--
X |   |
Computer's turn (O)...
X |   | O
--+---+--
O | O | X
--+---+--
X |   |
Player X, choose a position (1-9): 9
X |   | O
--+---+--
O | O | X
--+---+--
X |   | X
Computer's turn (O)...
X |   | O
--+---+--
O | O | X
--+---+--
X | O | X
Player X, choose a position (1-9): 2
X | X | O
--+---+--
O | O | X
--+---+--
X | O | X
It's a tie!
```

LAB:2 Vacuum cleaner

Observation book:

Date 1-10-24

LAB-02

Vacuum Cleaner

Algorithm:

```
Function ON (state, location):
    Print ("Suction turned ON")

Function OFF (state, location):
    Print ("Suction turned OFF")
            Turn
Function ~~Right~~ (location):
    if location == 1:   ## Room 1
        location = 2    # move to room 2
        Print ("Turning right to enter room 2")
    else:  # Room 2
~~Function left (location):~~
        ~~if location == 2: # Room 2~~
        location = 1    # move to room 1
        Print ("Turning left to enter room 1")

Function state (state):
    return state # return current state.

Function Vacuum ():
    State = "Dirty"
    Location = 1
    for i in range (2):
        if State == "Dirty":
            ON()
            State = "Clean"
            OFF()
            Turn (location)
        else:
```

Print ("Room already cleaned")
Turn (location).

Percept Sequence.

1, Clean					
1, clean	1, right				
1, clean	1, right	2, ~~clean~~ dirty			
1, clean	1, right	2, dirty	2, ~~left~~ clean		
1, clean	1, right	2, dirty	2, clean	2, left	
1, clean	1, right	2, dirty	2, clean	2, left	1, Clea

Code:

```
def ON():
    print("Suction turned ON").

def OFF():
    print ("Suction turned OFF ")

def Turn (location , direction):
    if direction == "forward":
        if location ==1 :
            location = 2.
            print("Turning right to enter
             room 2")
        elif location == 2 :
            location = 3
            print ("Turning right to enter
             room 3")
```

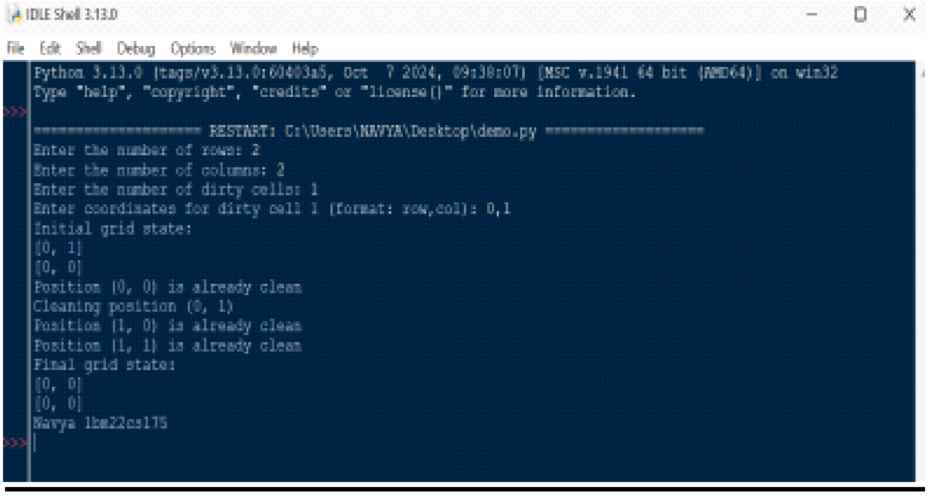
```
        elif location == 3 :
            location = 4
            print ("Turning right to enter
              room 4")
        else :
            pass
    else:
        if location == 2:
            location == 1
            print ("Turning left to enter
              room 1").
        elif location == 3 :
            location = 2
            print ("Turning left to enter
              room 2")
        elif location == 4 :
            location = 3
            print ("Turning left to enter room
                  3")
        elif location = 4
            location
            print ("Turning right to enter
             room 4")
    return location
state = "Dirty"
location = 1
print (" Starting at room 1, Dirty")
pass
four i in range (4) :
    if state == "Dirty":
        ON()
        state = "clean"
        print ("Room is clean")
```

```
        location = Turn (Location , "Forward")
    else :
        print ("Room is already cleaned")

for i in range (3) :
    if state == "Dirty" :
        ON ()
        State = "Clean"
        print (f"Room is clean now")
        OFF()
        location = Turn (Location , "reverse")
    else :
        print ("Room is clean")
        location = Turn (Location , "reverse
```

Output:

Starting room 1 , Dirty
Suction turned ON
Room is clean
Suction turned OFF
Turning right to enter room 2
room is already clean
" room 3
room is already clean
" room 4
room is already clean
Turning left to enter room 3
room is already cleaned

Output:


IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct 7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: C:\Users\NAVYA\Desktop\demo.py ====================
Enter the number of rows: 2
Enter the number of columns: 2
Enter the number of dirty cells: 1
Enter coordinates for dirty cell 1 (format: row,col): 0,1
Initial grid state:
[0, 1]
[0, 0]
Position (0, 0) is already clean
Cleaning position (0, 1)
Position (1, 0) is already clean
Position (1, 1) is already clean
Final grid state:
[0, 0]
[0, 0]
Navya 1bm22cs175
>>>

LAB 3 : 8 puzzle problems using DFS and Manhattan distance.

Observation book:

LAB-03

Manhattan Distance:

Algorithm:

1. ~~Initial state~~ Represent the initial state as ~~the~~ stack
2. Push the initial state onto a stack.
3. Pop the top state off the stack and check if it's the goal state.
4. If it's not the goal, generate all possible next states by moving the blank tile i.e up, down, left and right
5. Push all new states onto the stack, except already visited states.
6. Repeat this process until the stack is empty or the goal state is found.

MD - 9   DFS - 9

MD = 0

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

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_x, goal_y = divmod(state[i][j] - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def is_goal(state):
    return state == goal

def dfs(state, depth_limit, moves):
    blank_x, blank_y = find_blank(state)
    if is_goal(state):
        return True, state, moves
    if depth_limit == 0:
        return False, None, moves
    possible_moves = []
    for dx, dy in directions:
        new_x, new_y = blank_x + dx,
        blank_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
```

```
new_state = copy.deepcopy(state)
new_state[blank_x][blank_y], new_state[
    new_x][new_y] = new_state[new_x]
    [new_y], new_state[blank_x][blank_y]
md = manhattan_distance(new_state)
possible_moves.append((md, new_state))
possible_moves.sort(key = lambda x: x[0])
for _, next_state in possible_moves:
    moves.append(next_state)
  print("move made:")
  purint_board(next_state)
 found, vresult, moves = dfs(next_state, depth_
 limit-1, moves)
  if found:
      vreturn True, vresult, moves
  moves.pop()
 vreturn False, None, moves

def solve_puzzle(initial_state, depth_limit = 30):
moves = [initial_state]
purint("initial state:")
purint_board(initial_state)
found, final_state, moves = dfs(initial_state,
depth_limit, moves)
if found:
    purint("Solution found!")
    purint("Final
    purint_board(final_state)
else:
    purint("no solution")
Initial_state = [[1,2,3], [4,0,6], [7,5,8]]

solve-puzzle(initial_state)
```

Output:

```
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
1 0 3
4 2 6
7 5 8

1 2 3
4 0 6
7 5 8

1 2 3
4 5 6
7 0 8

1 2 3
4 5 6
7 8 0

Navya Billalar 1BM22CS175
>>
```

Lab 4 : 8 puzzle with A* and IDFS

Observation book:

10.24

LAB-4

Iterating deepening Search Algorithm

A* Algorithm:

Initial state	Goal state
[1 2 3]	[2 8 1]
[8 0 4]	[0 4 3]
[7 6 5]	[7 6 5]

* ~~Open list~~
* Initialize : initial state of the puzzle
  - Set the goal state
* Use a priority queue to store states of the puzzle, prioritized by $f(n) = g(n) + h(n)$,
  - $g(n)$ is the no. of moves
  - $h(n)$ counts how many tiles are not in their position.
* Remove the state with the smallest $f(n)$ from the queue.
* If ~~it~~ the state is the goal state, then stop and return solution.
* For each new state calculate $f(n) = g(n) + h$ and add to priority queue.

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

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

IDFS

```
function IDDFS (root, goal)
    for d = 0 to INT_MAX
      root = DFS (root , goal, depth)
       if result
           return result
    return NULL

function DLS (root, goal, depth)
 if depth = 0:
    if root - goal
           return root
    return NULL
 else
    result = DLS (root , goal, depth-1)
    if result :
         return result
 return NULL
```

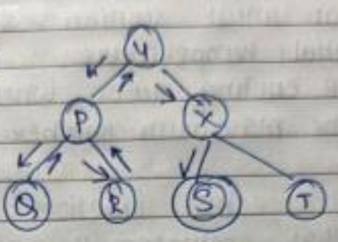
At depth = 0   4   4 return NULL

At depth 1

Y → P → X

return NULL

At depth 2



Y P Q R X S

return S

15/10

Code:

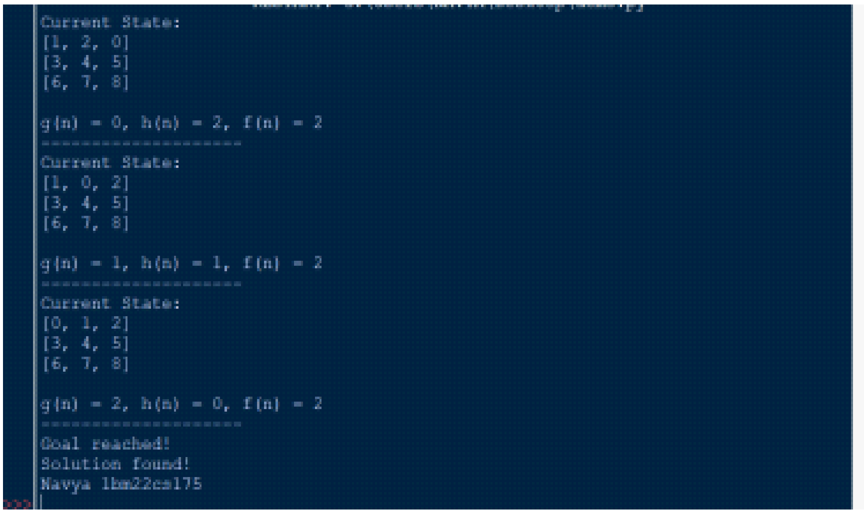
A* algorithm:

```
import heapq
goal_state = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
]
def flatten(puzzle):
    return [item for row in puzzle for item in row]
def find_blank(puzzle):
    for i in range(3):
        for j in range(3):
            if puzzle[i][j] == 0:
                return i, j
def misplaced_tiles(puzzle):
    flat_puzzle = flatten(puzzle)
    flat_goal = flatten(goal_state)
    return sum([1 for i in range(9) if flat_puzzle[i] != flat_goal[i] and flat_puzzle[i]
!= 0])
def generate_neighbors(puzzle):
    x, y = find_blank(puzzle)
    neighbors = []
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
```

```
            new_puzzle = [row[:] for row in puzzle]
            new_puzzle[x][y], new_puzzle[nx][ny] = new_puzzle[nx][ny],
new_puzzle[x][y]
            neighbors.append(new_puzzle)
    return neighbors


def is_goal(puzzle):
    return puzzle == goal_state


def print_puzzle(puzzle):
    for row in puzzle:
        print(row)
    print()


def a_star_misplaced_tiles(initial_state):
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial_state), 0, initial_state, []))
    visited = set()
    while frontier:
        f, g, current_state, path = heapq.heappop(frontier)
        print("Current State:")
        print_puzzle(current_state)
        h = misplaced_tiles(current_state)
        print(f"g(n) = {g}, h(n) = {h}, f(n) = {g + h}")
        print("-" * 20)

        if is_goal(current_state):
```

```
            print("Goal reached!")
            return path
        visited.add(tuple(flatten(current_state)))
        for neighbor in generate_neighbors(current_state):
            if tuple(flatten(neighbor)) not in visited:
                h = misplaced_tiles(neighbor)
                heapq.heappush(frontier, (g + 1 + h, g + 1, neighbor, path +
[neighbor]))
    return None
initial_state = [
    [1, 2, 0],
    [3, 4, 5],
    [6, 7, 8]
]
solution = a_star_misplaced_tiles(initial_state)
if solution:
    print("Solution found!")
else:
    print("No solution found.")
print("Navya 1bm22cs175")
```

Output:



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

g(n) = 0, h(n) = 2, f(n) = 2
====================
Current State:
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 1, h(n) = 1, f(n) = 2
====================
Current State:
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]

g(n) = 2, h(n) = 0, f(n) = 2
====================
Goal reached!
Solution found!
Navya 1bm22cs175
```

IDFS:

Code:

```
class Graph:
    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v):
        if u not in self.adjacency_list:
            self.adjacency_list[u] = []
        self.adjacency_list[u].append(v)

    def depth_limited_dfs(self, node, goal, limit, visited):
        if limit < 0:
            return False
        if node == goal:
```

```
            return True
        visited.add(node)
        for neighbor in self.adjacency_list.get(node, []):
            if neighbor not in visited:
                if self.depth_limited_dfs(neighbor, goal, limit - 1, visited):
                    return True
        visited.remove(node)  # Allow revisiting for the next iteration
        return False

    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth + 1):
            visited = set()
            if self.depth_limited_dfs(start, goal, depth, visited):
                return True
        return False


def main():
    graph = Graph()

    # Input number of edges
    num_edges = int(input("Enter the number of edges: "))

    # Input edges
    for _ in range(num_edges):
        edge = input("Enter an edge (format: A B): ").split()
```

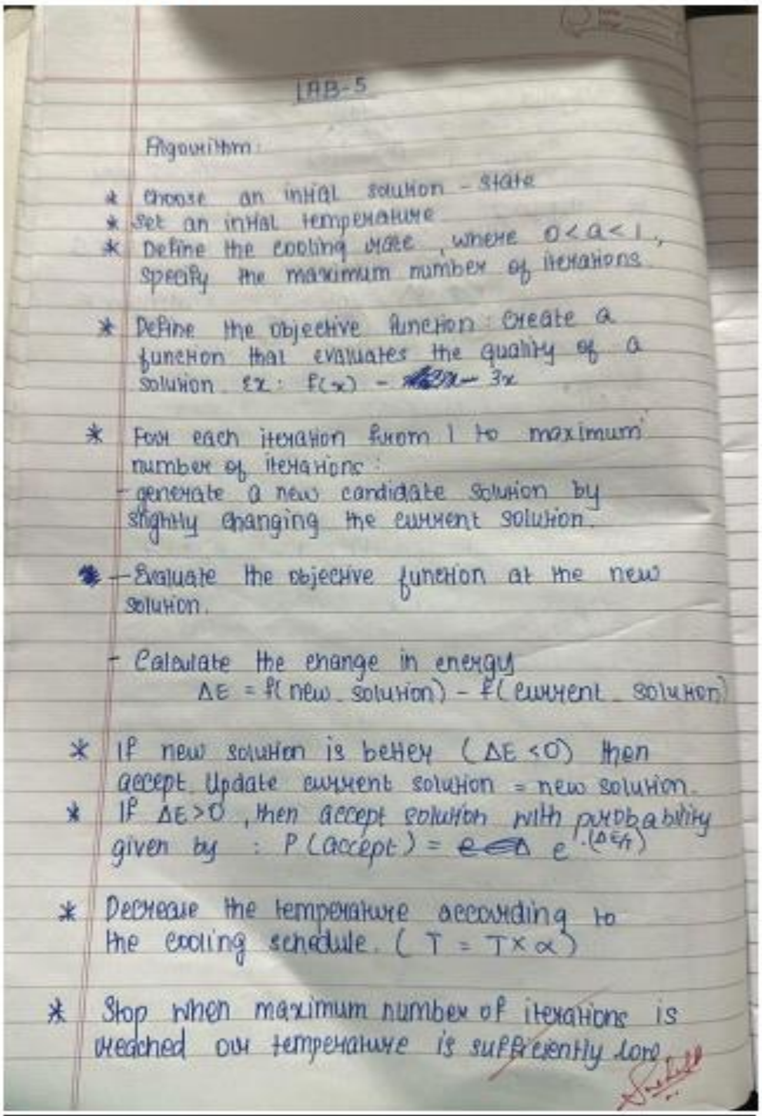
```
        graph.add_edge(edge[0], edge[1])

    start_node = input("Enter the start node: ")
    goal_node = input("Enter the goal node: ")
    max_depth = int(input("Enter the maximum depth for IDDFS: "))

    if graph.iddfs(start_node, goal_node, max_depth):
        print(f"Goal node {goal_node} found!")
    else:
        print(f"Goal node {goal_node} not found within depth {max_depth}.")


if __name__ == "__main__":
    main()
print("Navya 1bm22cs175")
```

Output:

```
==================== RESTART: C:\Users\NAVYA\Desktop\
Enter the number of edges: 5
Enter an edge (format: A B): A B
Enter an edge (format: A B): B C
Enter an edge (format: A B): C D
Enter an edge (format: A B): D E
Enter an edge (format: A B): E F
Enter the start node: A
Enter the goal node: F
Enter the maximum depth for IDDFS: 3
Goal node F not found within depth 3.
Navya 1bm22cs175
```

LAB 5: Simulated Annealing Algorithm

Observation book:



LAB-5

Algorithm:

* Choose an initial solution - state
* Set an initial temperature
* Define the cooling rate, where $0 < a < 1$, Specify the maximum number of iterations.

* Define the objective function: Create a function that evaluates the quality of a solution. Ex: $f(x) = x - 3x$

* For each iteration from 1 to maximum number of iterations:
  - generate a new candidate solution by slightly changing the current solution.

* - Evaluate the objective function at the new solution.

  - Calculate the change in energy
    $\Delta E = f(\text{new\_solution}) - f(\text{current\_solution})$

* If new solution is better ($\Delta E < 0$) then accept. Update current solution = new solution.
* If $\Delta E > 0$, then accept solution with probability given by: $P(\text{accept}) = e^{-(\Delta E/T)}$

* Decrease the temperature according to the cooling schedule. ($T = T \times \alpha$)

* Stop when maximum number of iterations is reached or temperature is sufficiently low.

Code:

```
import math
import random

def func(x):
  return 3x

def annealing (initial_s, initial_t, cool_rate,
                max_itns):
  curr_state = initial_s
  curr_val = func(curr_state)
  best_state = curr_state
  best_val = curr_val
  temp = initial_t

for iteration in range(max_itns):
  new_s = curr_state + random uniform (-1,1)
  new_val = func (new_s)
  delta_val = new_val - curr_val

if delta_value < 0 :
  curr_state = new_s
  curr_val = new_val
else:
  acc_prob = math.exp(-delta_val /temp)
  if random.random() < acc_prob :
     curr_state = new_s
     curr_val = new_val

if curr_val < best_val :
  best_state = curr_state
  best_val = curr_val
```

```
temp *= cool_rate

print(f"Iteration {iteration+1}: Current
State = {curr_state:.4f},
Current value = {curr_val:.4f},
Best state = {best_state:.4f},
Best value = {best_val:.4f}")

return best_state, best_val

if _name_ == "_main_":
    initial_s = random.uniform(-10,10)
    initial_t = 100
    cool_rate = 0.95
    max_itns = 100

best_state, best_val = annealing(initial_s,
initial_t, cool_rate, max_itns)

print(f"\n Best State Found: {best_state:.4f}
      Best Value: {best_val:.4f}")
```

Output:

Iteration 1: Current state = -1.6967, Current Value = 2.8789, Best State = -1.3766, Best Value = 1.8950

Iteration 2: Current state = -2.1096, Current value = 4.4503, Best State = -1.3766, Current value = 1.8950.

27/10/24

Output:

```
==================== RESTART: C:\Users\NAVYA\Desktop\demo.py ====================
Enter the initial state (starting point): 10
Enter the initial temperature: 12
Enter the cooling rate (between 0 and 1): 0.3
Enter the number of iterations: 5
Iteration 1: Current State = 9.2863, Current Energy = 86.2355, Temperature = 3.6
000
Iteration 2: Current State = 9.0532, Current Energy = 81.9601, Temperature = 1.0
800
Iteration 3: Current State = 8.8327, Current Energy = 78.0164, Temperature = 0.3
240
Iteration 4: Current State = 8.8327, Current Energy = 78.0164, Temperature = 0.0
972
Iteration 5: Current State = 8.8327, Current Energy = 78.0164, Temperature = 0.0
292
Best State: 8.8327, Best Energy: 78.0164
NAvya 1BM22CS175
```

LAB-6 - Implementing A* and Hill Climbing Algorithm on 8 Queens.

Observation book:

LAB-6

~~A* search Algorithm~~ Hill climbing for 8-Queen

1. Initial state : random configuration, one queen per row in random column.
2. h → number of pairs of queens attacking each other.
3. Goal : h = 0.
4. Check for conflict : same column, same row or same diagonal.
5. For each queen, count how many other queens it conflicts and add this to h.
6. Divide the final count by 2, each conflict is counted twice i.e once for each queen in the pair.

Output :

```
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . Q . . . . .
. . . . . Q . .
```

A* algorithm for 8 Queens:

1. Inital state : ~~Empty board~~ Random ~~configuration of~~, one queen every ~~row in random column~~ Empty Board and place quee

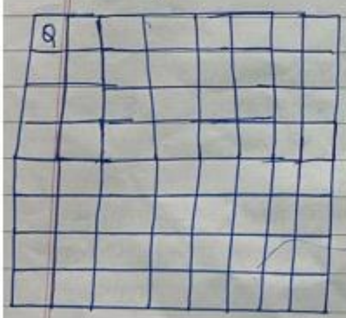
2. Cost for each ~~queen~~ setup:
   - steps taken so far (g) : This is the number of queen placed.
   - Conflicts (h) : how many Queens are conflicting each other.

3. Calculate $f(n) = g(n) + h(n)$

4. Choose the setup with lowest score.

5. Goal: If setup has no conflicts i.e $h=0$ and all Queens are placed.
   If not keep repeating until final solution.

Example:



$g = 1$

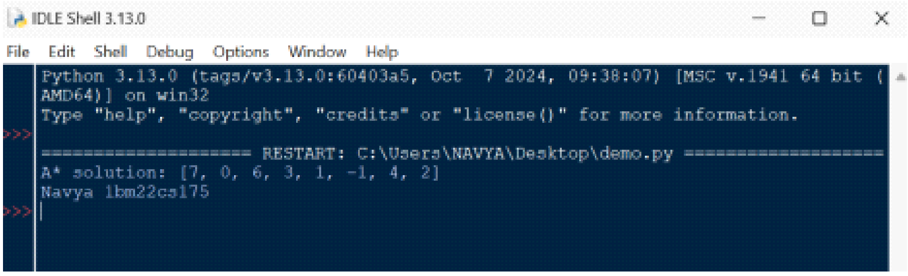
A * Code:

```
import numpy as np
import heapq

class Node:
    def __init__(self, state, g, h):
        self.state = state  # current state of the board
        self.g = g  # cost to reach this state
        self.h = h  # heuristic cost to reach goal
        self.f = g + h  # total cost

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

def heuristic(state):
    # Count pairs of queens that can attack each other
    attacks = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or abs(state[i] - state[j]) == j - i:
                attacks += 1
    return attacks

def a_star_8_queens():
    initial_state = [-1] * 8  # -1 means no queen placed
    open_list = []
```

```
    closed_set = set()
    initial_h = heuristic(initial_state)
    heapq.heappush(open_list, Node(initial_state, 0, initial_h))

    while open_list:
        current_node = heapq.heappop(open_list)
        current_state = current_node.state
        closed_set.add(tuple(current_state))

        # Check if we reached the goal
        if current_node.h == 0:
            return current_state

        for col in range(8):
            for row in range(8):
                if current_state[col] == -1:  # Only place a queen if none is present in
this column
                    new_state = current_state.copy()
                    new_state[col] = row
                    if tuple(new_state) not in closed_set:
                        g_cost = current_node.g + 1
                        h_cost = heuristic(new_state)
                        heapq.heappush(open_list, Node(new_state, g_cost, h_cost))

    return None

solution = a_star_8_queens()
```

```
print("A* solution:", solution)
print("Navya 1bm22cs175")
```

output:



Hill climbing :

Code:

```
import random

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

def hill_climbing_8_queens():
    state = [random.randint(0, 7) for _ in range(8)]  # Random initial state

    while True:
```

```
    current_h = heuristic(state)
    if current_h == 0:  # Found a solution
        return state

    next_state = None
    next_h = float('inf')

    for col in range(8):
        for row in range(8):
            if state[col] != row:  # Only consider moving the queen
                new_state = state.copy()
                new_state[col] = row
                h = heuristic(new_state)
                if h < next_h:
                    next_h = h
                    next_state = new_state

    if next_h >= current_h:  # No better neighbor found
        return None  # Stuck at local maximum

    state = next_state

solution = hill_climbing_8_queens()
print("Hill Climbing solution:", solution)
print("Navya 1bm22cs175")
```

Output :

```
Hill Climbing solution: None
navya 1bm22cs175
>>>
```

LAB-7 - Entailment Using Literals

Observation book:

LAB-7

Propositional logic:

Knowledge Base:

1. Alice is mother of Bob
2. Bob is the father of Charlie
3. A father is a parent
4. A mother is a parent
5. All parents have children
6. If someone is a parent, their children are siblings
7. Alice is married to David.

Hypothesis:

- Charlie is sibling of Bob = Q

Premises	Logical form	From knowledge base
1.	$P_1 : A \to B$	1. $A \to B$
2.	$P_2 : C \to D$	2. $B \to C$
3.	$P_3 : F \to P$	3. $F \to P$
4.	$P_4 : M \to P$	4. $M \to P$
5.	$P_5 : P \to C$	5. $P \to S$
6.	$P_6 : P \to (S)$	6. $A \wedge B \to Q$
7.	$P_7 : A \to D$	

1. $A \to B$ (if alice is mother of Bob and Bob is father of Charlie)

2. $A \wedge \to B \to S$ (if Alice and Bob are parents their children are siblings).

Entailment

1. if A (Alice is mother of Bob) is true
   → B must be true (since A → B)

2. if B is true
   → P must be true (F → P)
   → N must be true (M → P)

3. If Both Alice and Bob are parents (M & F are true)
   → S must be true

4. Since S is true
   → Q is true

Conclusion:
The hypothesis "Charlie is a sibling of Bob" is True. Therefore the hypothesis is entailed by the knowledge base.

Code:

```
import re


# Helper function to parse user input into logical predicates
def parse_input(input_sentence, knowledge_base):
    # Convert the sentence to lowercase for consistency
    input_sentence = input_sentence.lower()


    # Match patterns for predicates and facts (e.g., 'X is the mother of Y' or 'X is
married to Y')


    # Fact or Rule: "X is the mother of Y"
    mother_match = re.match(r"(\w+) is the mother of (\w+)", input_sentence)


    # Fact or Rule: "X is the father of Y"
    father_match = re.match(r"(\w+) is the father of (\w+)", input_sentence)


    # General rule: "All X have children"
    parent_match = re.match(r"all (\w+) have children", input_sentence)


    # Rule for parent-child relation and siblings
    parent_rule_match = re.match(r"if someone is a parent, their children are
siblings", input_sentence)


    # General fact: "X is married to Y"
    married_match = re.match(r"(\w+) is married to (\w+)", input_sentence)
```

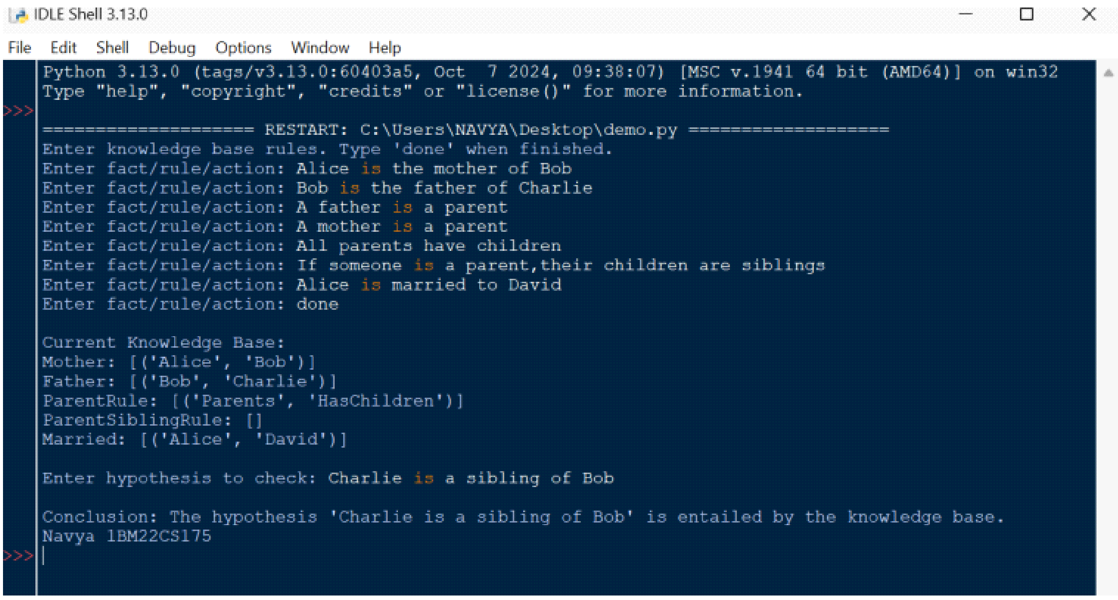
```
    # Parsing rules and facts
    if mother_match:
      mother, child = mother_match.groups()
      # Add the mother-child relationship to knowledge base
      knowledge_base["Mother"].append((mother.capitalize(),
child.capitalize()))

    elif father_match:
      father, child = father_match.groups()
      # Add the father-child relationship to knowledge base
      knowledge_base["Father"].append((father.capitalize(), child.capitalize()))

    elif parent_match:
      parent = parent_match.group(1)
      # Rule: All X are parents with children
      knowledge_base["ParentRule"].append((parent.capitalize(),
"HasChildren"))

    elif parent_rule_match:
      # General rule: If someone is a parent, their children are siblings
      knowledge_base["ParentSiblingRule"].append(("Parent", "Siblings"))

    elif married_match:
      spouse1, spouse2 = married_match.groups()
      # Add the married relationship to knowledge base
      knowledge_base["Married"].append((spouse1.capitalize(),
spouse2.capitalize()))
```

```
# Function to check if two children are siblings
def are_siblings(child1, child2, knowledge_base):
    # Check if both children share the same parent
    parents = set()
    for mother, child in knowledge_base["Mother"]:
        if child == child1:
            parents.add(mother)
        if child == child2:
            parents.add(mother)

    for father, child in knowledge_base["Father"]:
        if child == child1:
            parents.add(father)
        if child == child2:
            parents.add(father)

    return len(parents) > 1  # If both children share a parent, they are siblings

# Function to check the hypothesis "Charlie is a sibling of Bob"
def check_hypothesis(hypothesis, knowledge_base):
    # Parse the hypothesis
    hyp_match = re.match(r"(\w+) is a sibling of (\w+)", hypothesis.lower())
    if hyp_match:
        child1, child2 = hyp_match.groups()
        # Check if the children are siblings
```

```
        if are_siblings(child1.capitalize(), child2.capitalize(), knowledge_base):
            return True
    return False

# Main function for user input and entailment reasoning
def main():
    # Create an empty knowledge base
    knowledge_base = {
        "Mother": [],
        "Father": [],
        "ParentRule": [],
        "ParentSiblingRule": [],
        "Married": []
    }

    print("Enter knowledge base rules. Type 'done' when finished.")

    # Allow the user to input knowledge base facts, rules, or actions
    while True:
        user_input = input("Enter fact/rule/action: ").strip()
        if user_input.lower() == "done":
            break
        parse_input(user_input, knowledge_base)

    # Print the current knowledge base
    print("\nCurrent Knowledge Base:")
```

```
    for category, items in knowledge_base.items():
        print(f"{category}: {items}")

    # Ask for the hypothesis (the statement to check)
    hypothesis = input("\nEnter hypothesis to check: ").strip()

    # Check if the hypothesis is entailed
    if check_hypothesis(hypothesis, knowledge_base):
        print(f"\nConclusion: The hypothesis '{hypothesis}' is entailed by the 
knowledge base.")
    else:
        print(f"\nConclusion: The hypothesis '{hypothesis}' is NOT entailed by the 
knowledge base.")

# Run the program
main()
print("Navya 1BM22CS175")
```

Output:



```
IDLE Shell 3.13.0
File Edit Shell Debug Options Window Help
Python 3.13.0 (tags/v3.13.0:60403a5, Oct  7 2024, 09:38:07) [MSC v.1941 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
==================== RESTART: C:\Users\NAVYA\Desktop\demo.py ===================
Enter knowledge base rules. Type 'done' when finished.
Enter fact/rule/action: Alice is the mother of Bob
Enter fact/rule/action: Bob is the father of Charlie
Enter fact/rule/action: A father is a parent
Enter fact/rule/action: A mother is a parent
Enter fact/rule/action: All parents have children
Enter fact/rule/action: If someone is a parent,their children are siblings
Enter fact/rule/action: Alice is married to David
Enter fact/rule/action: done

Current Knowledge Base:
Mother: [('Alice', 'Bob')]
Father: [('Bob', 'Charlie')]
ParentRule: [('Parents', 'HasChildren')]
ParentSiblingRule: []
Married: [('Alice', 'David')]

Enter hypothesis to check: Charlie is a sibling of Bob

Conclusion: The hypothesis 'Charlie is a sibling of Bob' is entailed by the knowledge base.
Navya 1BM22CS175
>>>
```

LAB-8 - FOL using Unification.

Observation book:

LAB-8

First order logic

"If all dogs are mammals, and all mammals are animals, and Rex is a dog, then Rex is an animal"

FOL representation

1. All dogs are mammals

   $\forall x\,(Dog(x) \rightarrow Mammals(x))$

2. All mammals are animals

   $\forall x\,(Mammal(x) \rightarrow Animal(x))$

3. Rex is a dog:

   Dog (Rex)

4. Rex is an animal → conclusion

   Animal (Rex)

↳ Universal instantiation

1. From $\forall x\,(Dog(x) \rightarrow Mammal(x))$, instantiate for R
   Dog (Rex) → Mammal (Rex)

2. From $\forall x\,(Mammal(x) \rightarrow Animal(x))$, instantiate for
   Mammal (Rex) → Animal (Rex)

→ Modus Ponens (First step)

Dog(Rex) and Dog(Rex) → Mammal(Rex) imply Mammal(Rex)

So, we conclude : Mammal(Rex)

→ Modus Ponens (second step)

conclude that : Animal(Rex)

Conclusion:

$\forall x(Dog(x) \rightarrow Mammal(x))$
$\forall x (Mammal(x) \rightarrow Animal(x))$
Dog(Rex)
Rex is an animal

19/11

Code:

```
import re


# Define a simple function for extracting predicates from sentences
def extract_predicate(sentence):
    # Regular expression to find patterns like Predicate(Argument)
    pattern = r"([A-Za-z]+)\((\w+)\)"
    match = re.search(pattern, sentence)
    if match:
        predicate = match.group(1)
        subject = match.group(2)
        return predicate, subject
    return None, None


# Function for unification
def unify(fact, query):
    # Check if the fact and query are the same
    if fact == query:
        return True

    # Extract predicate and subject from fact and query
    fact_predicate, fact_subject = extract_predicate(fact)
    query_predicate, query_subject = extract_predicate(query)

    # If predicates match, unify the subjects
```

```
        if fact_subject == query_subject:
            return True
        else:
            # Here, we could handle variable substitution (unification)
            return False
    return False

# Function to deduce the goal using given rules
def deduct(rules, goal):
    # Try to find unification for the goal from the rules
    for rule in rules:
        if unify(rule, goal):
            print(f"Unification successful: {rule} matches with {goal}.")
            return True
    return False

# Main function to handle user input
def main():
    # Step 1: Get the rules (facts/implications) from the user
    print("Enter the rules (facts/implications). Type 'done' to finish entering
rules.")
    rules = []
    while True:
        rule_input = input("Enter rule: ")
        if rule_input.lower() == 'done':
            break
        else:
```

```
        rules.append(rule_input.strip())

    # Step 2: Get the goal (query) from the user
    goal_input = input("Enter the goal (query) to prove: ").strip()

    # Step 3: Try to deduce the goal using the given rules
    print("\nAttempting to deduce the goal...")
    if deduct(rules, goal_input):
        print(f"Conclusion: The goal '{goal_input}' is true based on the rules.")
    else:
        print(f"Conclusion: The goal '{goal_input}' cannot be proven with the
provided rules.")

# Run the program
main()
print("Navya 1bm22cs175")
```

Output:

```
Enter the rules (facts/implications). Type 'done' to finish entering rules.
Enter rule: all dogs are mammals
Enter rule: all mammals are animals
Enter rule: rex is a dog
Enter rule: done
Enter the goal (query) to prove: rex is an animal

Attempting to deduce the goal...
Unification successful: all dogs are mammals matches with rex is an animal.
Conclusion: The goal 'rex is an animal' is true based on the rules.
Navya 1bm22cs175
>>>
```

LAB-9 – FOL(forward chaining),minimax(tic-tac-toe),Alpha-Beta(8-Queens)

Observation book:

LAB-9

First Order Logic [Forward chaining]

"As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen"

Prove that "Robert is criminal".

↳ Predicates:

* American(x) : x is American citizen
* Hostile (x,y) : x is a hostile nation to y
* Sold (x,m,y): x sold missile m to y
* Criminal (x): x is a criminal

↳ logical axioms:

1. law of crime: It is a crime for an American to sell weapons to hostile nations.:
   * $\forall x \forall m \forall y$ (American (x) $\land$ Hostile (y, America) sold (x, m, y) $\rightarrow$ Criminal (x))

2. Robert is an American:
   * American (Robert)

3. Country A is hostile to America:
   * Hostile (A, America)

4. Robert sold missiles to Country A:
   * $\exists m$ Sold (Robert, m, A)

↳ Forward Chaining

* We know that for any American citizen x, if x sells missiles to hostile nation, x is a criminal. In logical form, the law is:

$\forall x \forall m \forall y$ (American(x) $\wedge$ Hostile$(y$, America$)$ $\wedge$ Sold(x, m, y) $\longrightarrow$ Criminal(x))

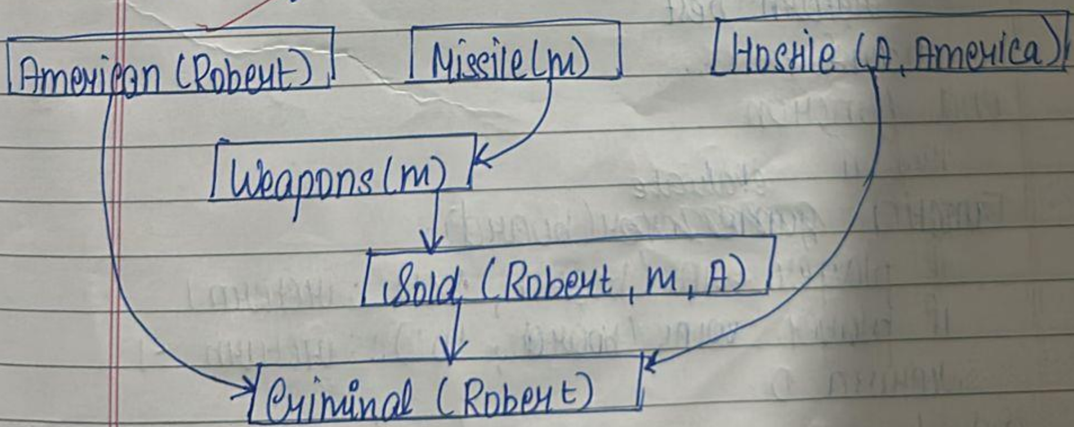
* From given information:

America (Robert)
Hostile (A, America)
$\exists m$ Sold (Robert, m, A)

* Using forward chaining, combine the fact with law.

American (Robert)
Hostile (A, America)
Sold (Robert, m, A)

⇒ Since all these facts are true, appling the rule from law: Criminal (Robert)



MinMax (Tic-Tac-Toe)

Algorithm:

```
Function minimax (board, player, MaxPlayer):
  if game_over(board):
     return eval(board);

  if MaxPlayer:
     best = - infinity
     for each moves in avail_moves(board):
        make_move (board, move, 'X')
        score = minmax (board, depth+1, false)
        undo_move (board, move)
        best = max(best, score)
     end for
     return best
  else:
     best = INFINITY
     for each move in available_moves(board):
     make_move (board, move, 'O')
     best = min (best, minimax (board, depth+1,
                  True))
     undo_move (board, move)
     end for
     return best
  end if
end function

Function evaluate(board):
  if player_wins (board, 'X') : return 1
  if player_wins (board, 'O') : return -1
  return 0
end function
```

Function game-over(board):
Return player_wins (board, 'X') OR player_wins (board, 'O') OR no_move _moves (board)
END FUNCTION
Function available_moves(board):
moves = [ ]
FOR row = 0 to 2:
Four col = 0 to 2:
IF board[row][col] == empty : moves.append((row, col))
END for
END for
Return moves
END FUNCTION
FUNCTION player_wins (board, player):
FOR each row, col, diagonal : IF player has 3 in a row, column, or diagonal : RETURN TRUE
RETURN FALSE
END FUNCTION
Function main():
board = [[empty, empty, empty][empty, empty, empty], [empty, empty, empty]]
current_player = 'X'
best_move = NONE
IF current_player == 'X':
best_score = - INFINITY
FOR each move in available_moves(board):
make_move(board, move, 'X')
score = minimax (board, 0, False)
undo_move (board, move)
IF score >best_score : best_score = score ;
best_move = move
END FOR
make_move(board, best_move, 'X')
END IF
Print board
END FUNCTION

Pruning

Alpha- Beta (8-Queens)

Algorithm:

```
function is_safe (board , row, col):
    for i =0 to row-1:
        if board [i] == col or abs(board[i]-col) == abs
                (i-row):
            return false
    end for
    return true
end function

function alpha_beta_queens(board , row, alpha, beta):
   if row == 8 :
      if is_solution (board):
         print board
         return true
   end if
   return false
end if
for col =0 to 7 :
   if is_safe (board, row, col):
      board [row] = col
      if alpha_beta_queens (board , row+1 , alpha,
         beta):
      return true
   end if
   board [row] = -1
end if
if row % 2 == 0 :
   alpha = max (alpha , value)
else :
      beta = min(beta , value)
```

```
    end if
    if alpha >= beta:
        break
    end if
end for
return false
end function

Function is_solution (board):
    return (distinct values in board == 8)
end function

Function main():
    board = [-1, -1, -1, -1, -1, -1, -1, -1]
    alpha = - infinity
    beta = infinity
    alpha_beta_queens (board, 0, alpha, beta)
end function.
```

30/12

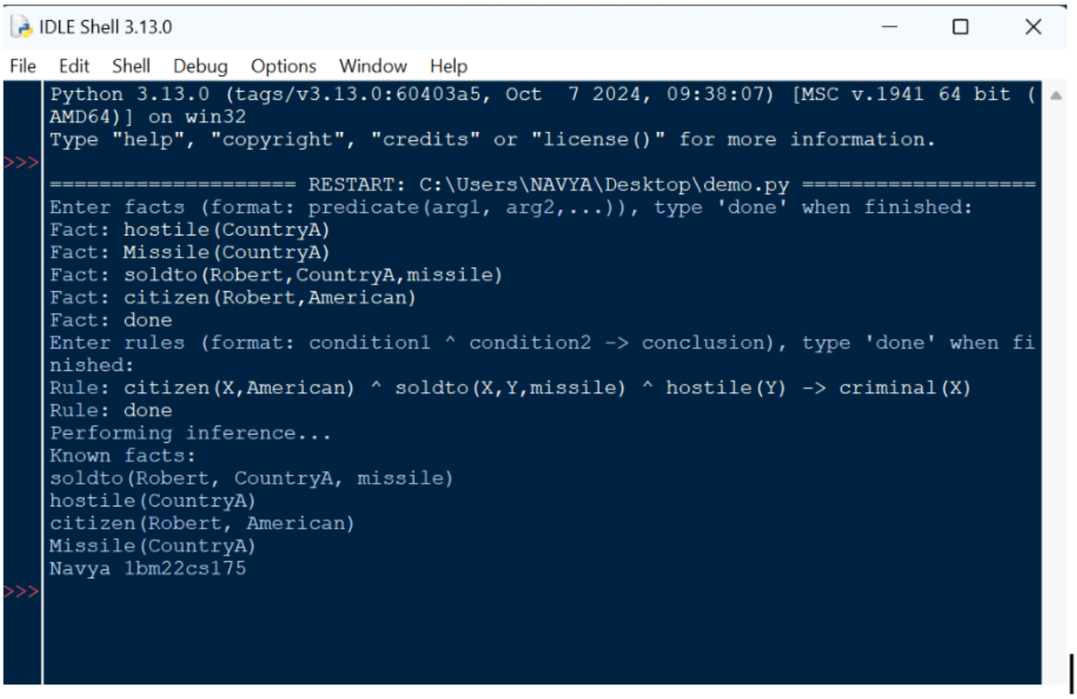
FOL forward chaining:

Code:

```
class Fact:
    def __init__(self, predicate, *args):
        self.predicate = predicate
        self.args = args

    def __eq__(self, other):
        return self.predicate == other.predicate and self.args == other.args

    def __hash__(self):
        return hash((self.predicate, self.args))

    def __str__(self):
        return f"{self.predicate}({', '.join(self.args)})"

class Rule:
    def __init__(self, conditions, conclusion):
        self.conditions = conditions
        self.conclusion = conclusion

    def is_satisfied(self, known_facts):
        return all(condition in known_facts for condition in self.conditions)

    def __str__(self):
   conditions_str = " ^ ".join(str(c) for c in self.conditions)
        return f"{conditions_str} -> {self.conclusion}"

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

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

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        new_facts = True
        while new_facts:
            new_facts = False
            for rule in self.rules:
                if rule.is_satisfied(self.facts) and rule.conclusion not in self.facts:
                    print(f"Applying rule: {rule}")
                    self.facts.add(rule.conclusion)
                    new_facts = True
```

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

if __name__ == "__main__":
    fc = ForwardChaining()

    print("Enter facts (format: predicate(arg1, arg2,...)), type 'done' when finished:")
    while True:
        user_input = input("Fact: ").strip()
        if user_input.lower() == "done":
            break
        try:
            predicate, args = user_input.split("(")
            args = args.strip(")").split(",")
            fc.add_fact(Fact(predicate.strip(), *[arg.strip() for arg in args]))
        except ValueError:
            print("Invalid format. Try again.")

    print("Enter rules (format: condition1 ^ condition2 -> conclusion), type 'done' when 
finished:")
    while True:
        user_input = input("Rule: ").strip()
        if user_input.lower() == "done":
            break
        try:
            conditions_part, conclusion_part = user_input.split("->")
            conditions = []
            for condition in conditions_part.split("^"):
                predicate, args = condition.strip().split("(")
                args = args.strip(")").split(",")
                conditions.append(Fact(predicate.strip(), *[arg.strip() for arg in args]))
            predicate, args = conclusion_part.strip().split("(")
            args = args.strip(")").split(",")
            conclusion = Fact(predicate.strip(), *[arg.strip() for arg in args])
            fc.add_rule(Rule(conditions, conclusion))
        except ValueError:
            print("Invalid format. Try again.")

    print("Performing inference...")
    fc.infer()
    fc.display_facts()
print("Navya 1bm22cs175")
```

Output:



Minimax (Tic-tac-toe):

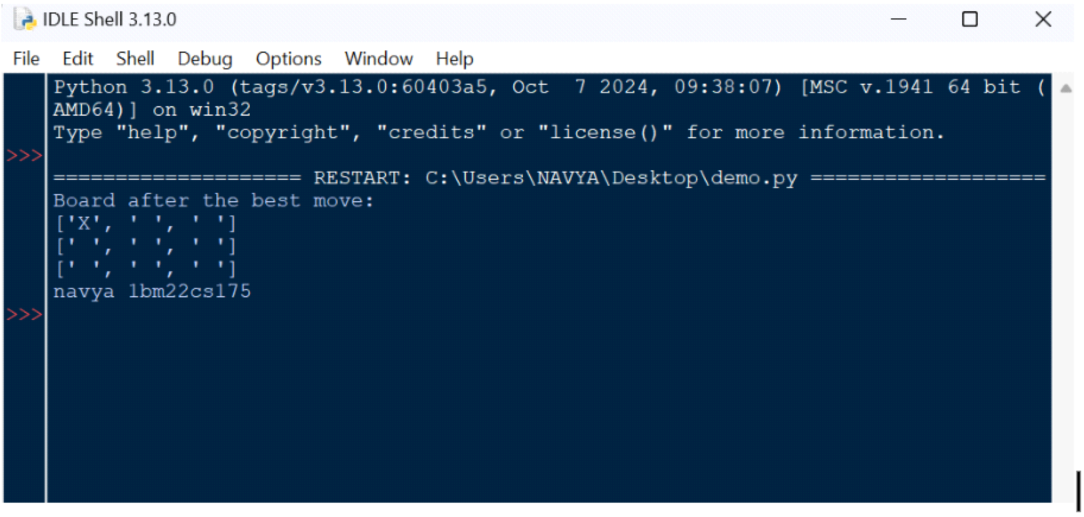
Code:

```
import math

def minimax(board, depth, is_maximizing_player):
    if game_over(board):
        return evaluate(board)

    if is_maximizing_player:
        best = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
            best = max(best, minimax(board, depth + 1, False))
            undo_move(board, move)
        return best
    else:
        best = math.inf
        for move in available_moves(board):
            make_move(board, move, 'O')
            best = min(best, minimax(board, depth + 1, True))
            undo_move(board, move)
        return best

def evaluate(board):
    if player_wins(board, 'X'):
        return 1
    if player_wins(board, 'O'):
        return -1
    return 0
```

```
def game_over(board):
    return player_wins(board, 'X') or player_wins(board, 'O') or no_more_moves(board)

def available_moves(board):
    moves = []
    for row in range(3):
        for col in range(3):
            if board[row][col] == " ":
                moves.append((row, col))
    return moves

def make_move(board, move, player):
    row, col = move
    board[row][col] = player

def undo_move(board, move):
    row, col = move
    board[row][col] = " "

def player_wins(board, player):
    # Check rows and columns
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or all(board[j][i] == player for j in range(3)):
            return True
    # Check diagonals
    if all(board[i][i] == player for i in range(3)) or all(board[i][2 - i] == player for i in
range(3)):
        return True
    return False

def no_more_moves(board):
    return all(board[row][col] != " " for row in range(3) for col in range(3))

def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = 'X'
    best_move = None

    if current_player == 'X':
        best_score = -math.inf
        for move in available_moves(board):
            make_move(board, move, 'X')
            score = minimax(board, 0, False)
            undo_move(board, move)
            if score > best_score:
                best_score = score
                best_move = move
        make_move(board, best_move, 'X')
```

```
    print("Board after the best move:")
    for row in board:
        print(row)

if __name__ == "__main__":
    main()
print("navya 1bm22cs175")
```

Output:



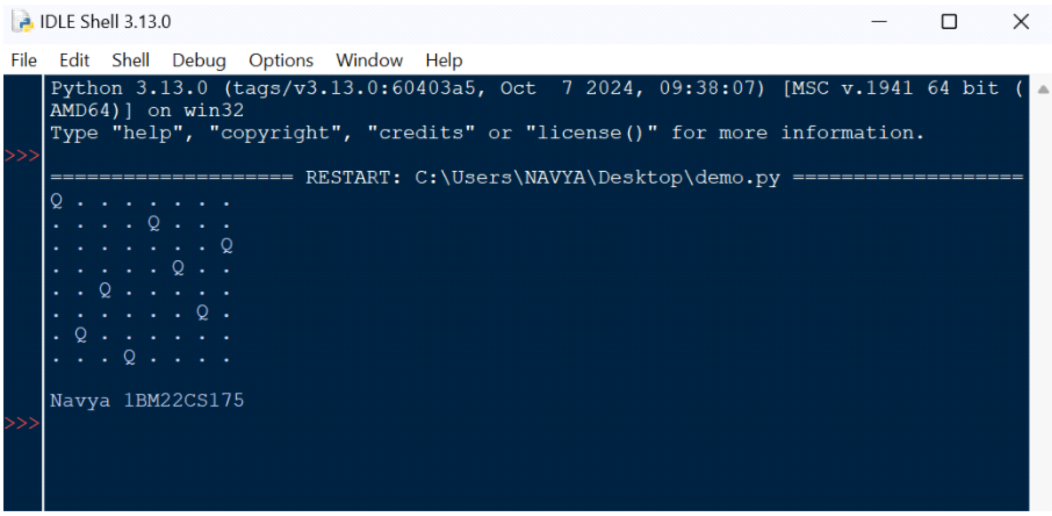
Alpha-beta(8Queens):

Code:

```
# Function to check if placing a queen at (row, col) is safe
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):  # Check for column and
diagonal conflicts
            return False
    return True

# Backtracking function for N-Queens
def solve_n_queens(board, row):
    if row == 8:  # All queens have been placed
        print_board(board)  # Print the board if solution is found
        return True

    for col in range(8):  # Try placing a queen in each column of the current row
        if is_safe(board, row, col):  # Check if placing a queen at (row, col) is safe
            board[row] = col  # Place the queen in the current column

            # Recursively attempt to place the next queen in the next row
            if solve_n_queens(board, row + 1):
```

```
                return True  # Solution found, propagate up

            board[row] = -1  # Backtrack: Remove the queen from the current position

    return False  # No solution found in the current row and column configurations

# Function to print the board in a readable format
def print_board(board):
    for row in range(8):
        line = ['Q' if board[row] == col else '.' for col in range(8)]
        print(" ".join(line))
    print()

# Main function to start solving the N-Queens problem
def main():
    board = [-1] * 8  # Initialize the board (no queens placed)
    if not solve_n_queens(board, 0):  # Start solving from the first row
        print("No solution found.")

# Call the main function
main()

print("Navya 1BM22CS175")
```

Output:



CIE_3

3A)
Code:

```
# 3a: Defining facts and rules in a simple way

# Facts
Cat = {"Tom"}  # Set of cats
Mary_allergic_to_cats = True  # Mary is allergic to cats
LivesWith_Mary_and_Cat = True  # We assume Mary lives with a cat (as per the context)
Allergic = {"Mary"}  # Set of people who suffer from allergies (we'll start with Mary)

# Rule: If someone suffers from allergies, they sneeze
def sneeze(x):
    return x in Allergic

# Rule: If someone lives with a cat and is allergic to it, then they suffer from allergies
def suffer_allergies(x):
    if LivesWith_Mary_and_Cat and Mary_allergic_to_cats:
        Allergic.add("Mary")

# Apply the rule to see if Mary sneezes
suffer_allergies("Mary")

# Now, check if Mary sneezes
if sneeze("Mary"):
    print("Mary sneezes: True")
else:
    print("Mary sneezes: False")
```

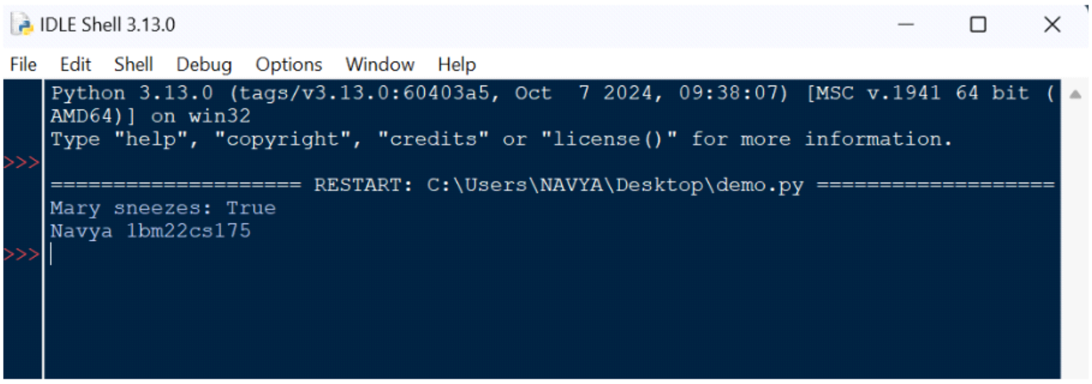
output:

CIE_3
3A)
Code:

```
# 3a: Defining facts and rules in a simple way

# Facts
Cat = {"Tom"}  # Set of cats
Mary_allergic_to_cats = True  # Mary is allergic to cats
LivesWith_Mary_and_Cat = True  # We assume Mary lives with a cat (as per the context)
Allergic = {"Mary"}  # Set of people who suffer from allergies (we'll start with Mary)

# Rule: If someone suffers from allergies, they sneeze
def sneeze(x):
    return x in Allergic
```

```
# Rule: If someone lives with a cat and is allergic to it, then they suffer from allergies
def suffer_allergies(x):
    if LivesWith_Mary_and_Cat and Mary_allergic_to_cats:
        Allergic.add("Mary")

# Apply the rule to see if Mary sneezes
suffer_allergies("Mary")

# Now, check if Mary sneezes
if sneeze("Mary"):
    print("Mary sneezes: True")
else:
    print("Mary sneezes: False")
```

output:



3B)
Code:

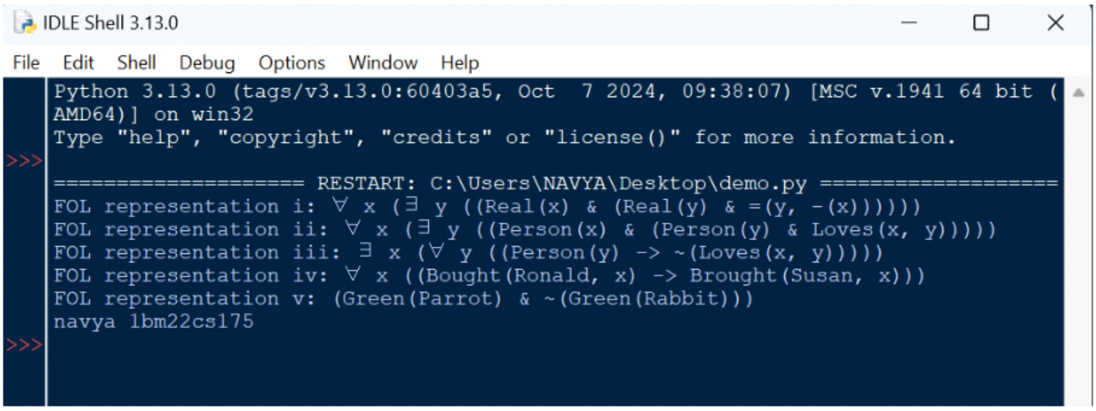
```
# Define basic classes for FOL terms, predicates, and quantifiers
class Term:
    pass

class Variable(Term):
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return self.name

class Function(Term):
    def __init__(self, func_name, *args):
        self.func_name = func_name
        self.args = args

    def __repr__(self):
```

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

class Predicate:
    def __init__(self, pred_name, *args):
        self.pred_name = pred_name
        self.args = args

    def __repr__(self):
        return f"{self.pred_name}({', '.join(map(str, self.args))})"

# Define logical operations for Predicate
def __and__(self, other):
    if isinstance(other, Predicate):
        return Conjunction(self, other)
    return NotImplemented

def __or__(self, other):
    if isinstance(other, Predicate):
        return Disjunction(self, other)
    return NotImplemented

def __invert__(self):
    return Negation(self)

def __rshift__(self, other):
    if isinstance(other, Predicate):
        return Implication(self, other)
    return NotImplemented

class Quantifier:
    def __init__(self, quantifier, variable, expression):
        self.quantifier = quantifier  # 'forall' or 'exists'
        self.variable = variable
        self.expression = expression

    def __repr__(self):
        return f"{self.quantifier} {self.variable} ({self.expression})"

# Logical Connective Classes
class Conjunction:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __repr__(self):
        return f"({self.left} & {self.right})"

class Disjunction:
    def __init__(self, left, right):
```

```
        self.left = left
        self.right = right

    def __repr__(self):
        return f"({self.left} | {self.right})"

class Negation:
    def __init__(self, expression):
        self.expression = expression

    def __repr__(self):
        return f"~({self.expression})"

class Implication:
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def __repr__(self):
        return f"({self.left} -> {self.right})"

# Helper function to create FOL statements
def forall(variable, expression):
    return Quantifier('∀', variable, expression)

def exists(variable, expression):
    return Quantifier('∃', variable, expression)

# FOL Representation for all the examples
# i. Every real number has its corresponding negative.
x = Variable('x')
y = Variable('y')
Real = Predicate('Real', x)
negative = Function('-', x)
# Real(x) -> exists y (Real(y) & (y = -(x)))
expression_i = forall(x, exists(y, Conjunction(Real, Conjunction(Predicate('Real', y),
    Predicate('=', y, negative)))))
print("FOL representation i:", expression_i)

# ii. Everybody loves somebody.
Loves = Predicate('Loves', x, y)
expression_ii = forall(x, exists(y, Conjunction(Predicate('Person', x),
    Conjunction(Predicate('Person', y), Loves))))
print("FOL representation ii:", expression_ii)

# iii. There is somebody whom no one loves.
expression_iii = exists(x, forall(y, Implication(Predicate('Person', y), Negation(Loves))))
print("FOL representation iii:", expression_iii)
```

```
# iv. Susan brought everything that Ronald bought.
Bought = Predicate('Bought', 'Ronald', x)
Brought = Predicate('Brought', 'Susan', x)
expression_iv = forall(x, Implication(Bought, Brought))
print("FOL representation iv:", expression_iv)

# v. Parrot is green while rabbit is not.
Green = Predicate('Green', 'Parrot')
Green_Rabbit = Predicate('Green', 'Rabbit')
expression_v = Conjunction(Green, Negation(Green_Rabbit))
print("FOL representation v:", expression_v)
print("navya 1bm22cs175")
```

Output:



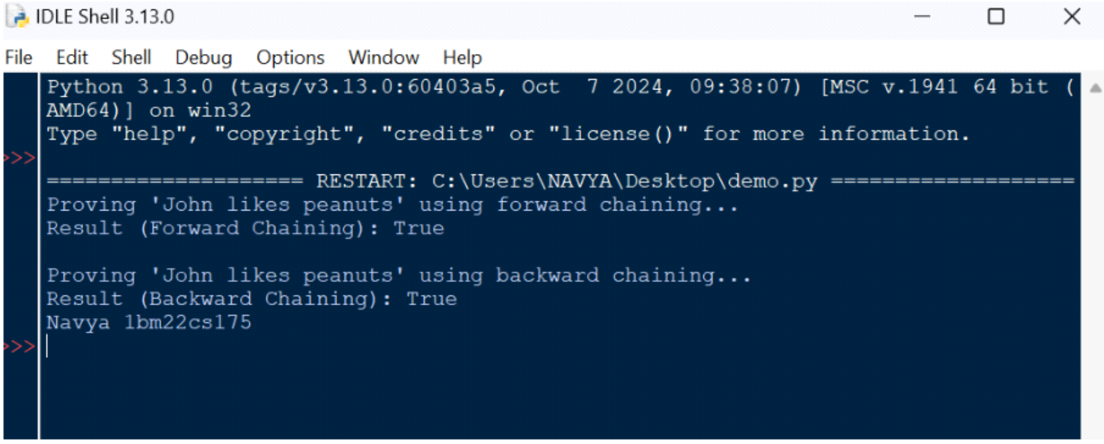
4a)

Code:

```
# 4a: Facts

facts = {
    "Food": {"Apples", "Chicken", "Peanuts"},  # Initial known food items
    "Eats": {"Bill": {"Peanuts"}},  # Bill eats peanuts
    "Alive": {"Bill": True},  # Bill is alive
}

# Rules
def john_likes_food(x):
    """John likes all food."""
    return x in facts["Food"]

def food_from_eating(y, x):
    """Anything anyone eats and isn't killed by is food."""
    return x in facts["Eats"].get(y, set()) and facts["Alive"].get(y, False)

# Function to perform forward chaining
def forward_chaining():
```

```
    # Start with the known facts about food
    inferred_facts = set(facts["Food"])

    # Step 1: Apply "Anything anyone eats and isn't killed by is food"
    for person in facts["Eats"]:
        for food in facts["Eats"][person]:
            if food_from_eating(person, food):  # If food is safe to eat
                inferred_facts.add(food)  # Add it to food

    # Step 2: Apply "John likes food" to all food items
    for food in list(inferred_facts):  # We convert to list to avoid modifying while iterating
        if john_likes_food(food):
            inferred_facts.add(f"Likes_John_{food}")  # Add the fact that John likes the food

    # Check if John likes peanuts
    return "Likes_John_Peanuts" in inferred_facts

# Add Peanuts as a food item if Bill eats peanuts and survives
facts["Eats"]["Bill"].add("Peanuts")
facts["Alive"]["Bill"] = True

# 1. Forward chaining to prove "John likes Peanuts"
print("Proving 'John likes peanuts' using forward chaining...")
result_forward = forward_chaining()
print("Result (Forward Chaining):", result_forward)  # Expected output: True

# Function to perform backward chaining
def backward_chaining(goal):
    # The goal is "Likes_John_Peanuts"
    if goal == "Likes_John_Peanuts":
        # To prove John likes peanuts, we need to show that Peanuts are food
        if "Peanuts" in facts["Food"]:
            return True
        else:
            # Check if Peanuts can be derived as food using the "Food_from_eating" rule
            if food_from_eating("Bill", "Peanuts"):
                facts["Food"].add("Peanuts")  # Add Peanuts to the food set
            return True
    return False

print("\nProving 'John likes peanuts' using backward chaining...")
result_backward = backward_chaining("Likes_John_Peanuts")
print("Result (Backward Chaining):", result_backward)  # Expected output: True
print("Navya 1bm22cs175")
```

Output:



4b)

Code:

```
# 4b: Minimax with Alpha-Beta Pruning

def minimax(node, depth, is_maximizing_player, values, alpha=float('-inf'), beta=float('inf')):
    # Base case: If we reach a leaf node or exceed the depth
    if depth == 0 or 2 * node + 1 >= len(values):
        return values[node] if node < len(values) else 0  # Return leaf node value or 0 if out of
bounds

    # If this is a MAX node
    if is_maximizing_player:
        best = float('-inf')
        for i in range(2):  # Two child nodes
            child_index = 2 * node + 1 + i  # Left and Right children
            if child_index < len(values):  # Ensure child_index is within bounds
                child_value = minimax(child_index, depth - 1, False, values, alpha, beta)
                best = max(best, child_value)
                alpha = max(alpha, best)
                if beta <= alpha:
                    break  # Beta cut-off
        return best
    # If this is a MIN node
    else:
        best = float('inf')
        for i in range(2):  # Two child nodes
            child_index = 2 * node + 1 + i  # Left and Right children
            if child_index < len(values):  # Ensure child_index is within bounds
                child_value = minimax(child_index, depth - 1, True, values, alpha, beta)
                best = min(best, child_value)
                beta = min(beta, best)
                if beta <= alpha:
                    break  # Alpha cut-off
        return best
```

```
# Function to call minimax and simulate the game tree
def solve_game_tree():
    # Leaf node values (given in the game tree)
    values = [8, 9, 11, 10, 13, 12, 4, 6, 9, 6, 12, 14, 20, 2, 2, 2]
    depth = 4  # Depth of the tree
    root_node = 0  # Start from the root node

    # Start the minimax algorithm
    result = minimax(root_node, depth, True, values)
    print(f"The optimal value for the root node is: {result}")

# Run the solution
solve_game_tree()
print("Navya 1bm22cs175")
```

Output: