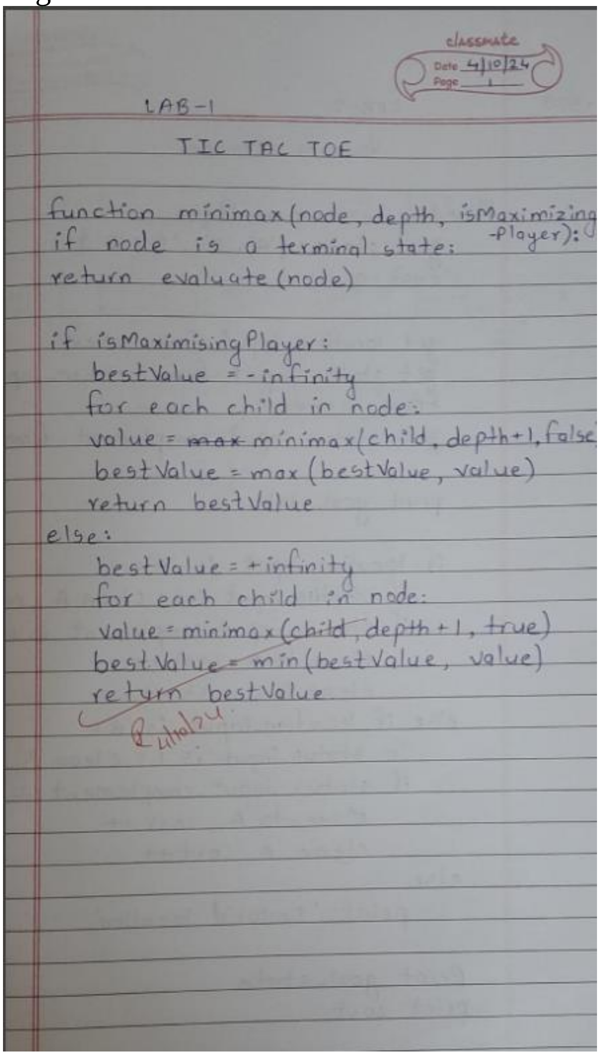
Program 1

Implement Tic –Tac –Toe Game

Tic Tac Toe:

Algorithm:


classmate

Date 4/10/24

Page 1

LAB-1

TIC TAC TOE

```
function minimax(node, depth, isMaximizing-Player):
if node is a terminal state:
return evaluate(node)

if isMaximisingPlayer:
   bestValue = -infinity
   for each child in node:
   value = ~~max~~ minimax(child, depth+1, false)
   bestValue = max(bestValue, value)
   return bestValue
else:
   bestValue = +infinity
   for each child in node:
   value = minimax(child, depth+1, true)
   bestValue = min(bestValue, value)
   return bestValue
```

Code:

```
board=([['1','1','1'],['1','1','1'],['1','1','1']])

def check(board,user):
  for i in range(3):
    if(board[0][i]==user and board[1][i]==user and board[2][i]==user):
      return True
    if(board[i][0]==user and board[i][1]==user and board[i][2]==user):
      return True
    if(board[0][0]==user and board[1][1]==user and board[2][2]==user):
      return True
    if(board[0][2]==user and board[1][1]==user and board[2][0]==user):
      return True
  return False
```

```
def show(board):
  for b in board:
    print(b)
```

```
def full(board):
```

```
  for i in range(3):
    for j in range(3):
      if(board[i][j] == '1'):
        return False
  return True

user=0

user1=input("Enter user name:")
user2=input("Enter user name:")

while True :

  if (full(board)) :
    print("Draw")
    break

  if(user==0):

    show(board)
    print(user1 + " play")
    row=int(input("Enter row:"))
    col=int(input("Enter col:"))

    if(board[row][col]=='1'):
      board[row][col]='X'
    else:
      print("Wrong!")
      continue

    if(check(board,'X')):
      print(user1 + " won!")
      break
    else:
      user=1

  if(full(board)):
      print("Draw")
      break

  if(user==1):

    show(board)

    print(user2 + " play")
    row=int(input("Enter row:"))
    col=int(input("Enter col:"))

    if(board[row][col]=='1'):
      board[row][col]='O'
    else:
      print("Wrong!")
      continue

    if(check(board,'O')):
      print(user2 + " won!")
      break
```

```
    else:
      user=0
```

```
  if full(board):
        print("Draw")
        break
```

```
X| |
-+-+-
 | |
-+-+-
 | |


Enter position for O: 5
X| |
-+-+-
 |O|
-+-+-
 | |


X|X|
-+-+-
 |O|
-+-+-
 | |


Enter position for O: 3
X|X|O
-+-+-
 |O|
-+-+-
 | |
```

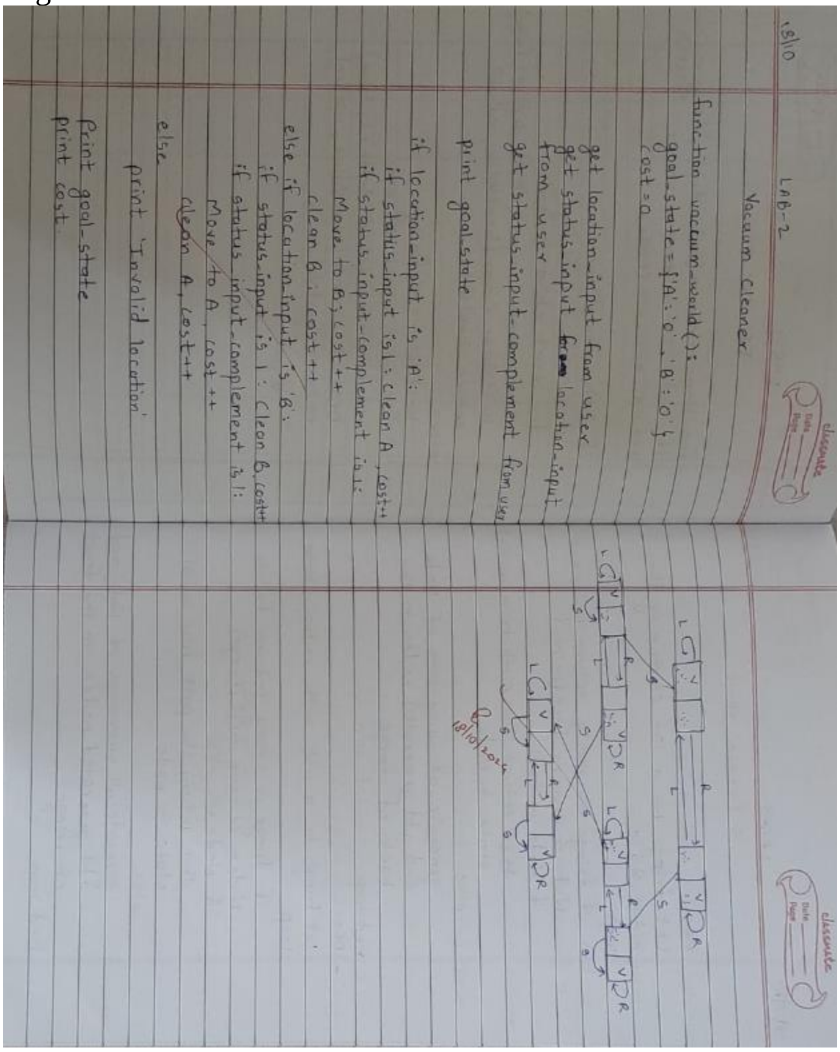
```
X|X|O
-+-+-
 |O|
-+-+-
X| |


Enter position for O: 4
X|X|O
-+-+-
O|O|
-+-+-
X| |


X|X|O
-+-+-
O|O|X
-+-+-
X| |


Enter position for O: 8
X|X|O
-+-+-
O|O|X
-+-+-
X|O|
```

```
X|X|O
-+-+-
O|O|X
-+-+-
X|O|X


Draw!
```

Program 2:

Vacuum World:

Algorithm:


18/10
LAB-2
Vacuum Cleaner

function vacuum_world():
goal_state = {'A':'0', 'B':'0'}
cost = 0

get location_input from user
get status_input from location_input from user
get status_input_complement from user

print goal_state

if location_input is 'A':
if status_input is 1: clean A, cost++
if status_input_complement is 1:
Move to B; cost++
clean B, cost++
else if location_input is 'B':
if status_input is 1: Clean B, cost++
if status_input_complement is 1:
Move to A, cost++
clean A, cost++
else
print 'Invalid location'

print goal_state
print cost

Code:

```
cost=0

def vacuum(state1, state2, loc):
    global cost
    if state1 == "0" and state2 == "0":
        print("All done")
        return

    if loc == "A":
        if state1 == "1" and state2=="1":
            print("Cleaned A")
            cost= cost+1
            state1 = "0"
            state1 = input("Is A dirty again?:  ")
            vacuum(state1,state2,"A")
        elif state1=="1" and state2=="0":
            print("Cleaned A")
            cost= cost+1
            state1 = "0"
            state1 = input("Is A dirty again?:  ")
            state2 = input("Is B dirty again?:  ")
```

```
            vacuum(state1,state2,"A")
        elif state1=="0" and state2=="1":
            print("Moving to B")
            loc="B"
            vacuum(state1,state2,loc)
```

```
    elif loc == "B":
        if state1 == "1" and state2=="1":
            print("Cleaned B")
            cost= cost+1
            state2 = "0"
            state2 = input("Is A dirty again?:  ")
            vacuum(state1,state2,"B")
        elif state1=="0" and state2=="1":
            print("Cleaned B")
            cost= cost+1
            state1 = "0"
            state1 = input("Is A dirty again?:  ")
            state2 = input("Is B dirty again?:  ")
            vacuum(state1,state2,"A")
        elif state1=="1" and state2=="0":
            print("Moving to B")
            loc="B"
            vacuum(state1,state2,loc)
```

```
print("Enter both states and location of vacuum")
state1 = input("Enter state 1 (0 or 1): ")
state2 = input("Enter state 2 (0 or 1): ")
loc = input("Enter loc (A or B): ")
vacuum(state1, state2, loc)
print("Total cost " + str(cost))
```

```
Enter both states and location of vacuum
Enter state 1 (0 or 1): 1
Enter state 2 (0 or 1): 1
Enter loc (A or B): A
Cleaned A
Is A dirty again?:  1
Cleaned A
Is A dirty again?:  0
Moving to B
Cleaned B
Is A dirty again?:  0
Is B dirty again?:  1
Moving to B
Cleaned B
Is A dirty again?:  0
Is B dirty again?:  0
All done
Total cost 4
```

Program 3
Implement 8 puzzle problems using DFS and BFS

8 puzzle using DFS and BFS:
Algorithm:

18/10/24

LAB-3

8 Puzzle game

BFS:
Let fringe be a list containing the initial state
loop
- if fringe is empty return failure
- Node ← remove first (fringe)
- if Node is goal
  - then return the path from initial state to Node
- else
  - generate all successors of Node, and add generated nodes to the back of fringe

End loop

- DFS:

let fringe be a list with initial state
loop
- if fringe is empty return failure
- Node ← remove first (fringe)
- if Node is goal
  - then return the path from initial state to node
- else
  - generate all successors of Node, and add generated nodes to the front of fringe.

End loop.

Code:
```
count=0;
def print_state(in_array):
    global count
    count+=1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()
```

```
def helper(goal, in_array, row, col, vis):
    # Marking current position as visited
    vis[row][col] = 1
    drow = [-1, 0, 1, 0]  # Dir for row : up, right, down, left
    dcol = [0, 1, 0, -1]  # Dir for column
```

```
    dchange = ['Up', 'Right', 'Down', 'Left']

    # Print current state
    print("Current state:")
    print_state(in_array)

    # Check if the current state is the goal state
    if in_array == goal:
        print_state(in_array)
        print(f"Number of states:{cnt}")
        return True

    # Explore all possible directions
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]

        # Check if the new position is within bounds and not visited
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not
vis[nrow][ncol]:
            # Make the move (swap the empty space with the adjacent tile)
            print(f"Took a {dchange[i]} move")
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

            # Recursive call
            if helper(goal, in_array, nrow, ncol, vis):
                return True

            # Backtrack (undo the move)
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol],
in_array[row][col]

    # Mark the position as unvisited before returning
    vis[row][col] = 0
    return False

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]  # 0 represents the empty
space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]  # 3x3 visited matrix
empty_row, empty_col = 1, 0  # Initial position of the empty space

found_solution = helper(goal_state, initial_state, empty_row, empty_col,
visited)
print("Solution found:", found_solution)
```

```
Current state:
1 2 3
4 6 8
7 5 0

Took a Left move
Current state:
1 2 3
4 6 8
7 0 5

Took a Left move
Current state:
1 2 3
4 6 8
0 7 5

Took a Down move
Current state:
1 2 3
4 5 6
7 0 8

Took a Right move
Current state:
1 2 3
4 5 6
7 8 0

1 2 3
4 5 6
7 8 0

Number of states:42
Solution found: True
```

Iterative deepening search algorithm:
Code:

```
#iterative-deepening
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_pos, moves=0, previous=None):
        self.board = board
        self.zero_pos = zero_pos  # Position of the zero tile
        self.moves = moves          # Number of moves taken to reach this state
        self.previous = previous    # For tracking the path
```

```python
    def is_goal(self, goal_state):
        return self.board == goal_state
```

```python
    def get_possible_moves(self):
        moves = []
        x, y = self.zero_pos
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = [row[:] for row in self.board]
                # Swap the zero tile with the adjacent tile
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y], new_board[x][y]
                moves.append((new_board, (new_x, new_y)))
        return moves
```

```python
def ids(initial_state, goal_state, max_depth):
    for depth in range(max_depth):
        visited = set()
        result = dls(initial_state, goal_state, depth, visited)
        if result:
            return result
    return None
```

```python
def dls(state, goal_state, depth, visited):
    if state.is_goal(goal_state):
        return state
    if depth == 0:
        return None
```

```python
    visited.add(tuple(map(tuple, state.board)))  # Mark this state as visited
    for new_board, new_zero_pos in state.get_possible_moves():
        new_state = PuzzleState(new_board, new_zero_pos, state.moves + 1, state)
        if tuple(map(tuple, new_board)) not in visited:
            result = dls(new_state, goal_state, depth - 1, visited)
            if result:
                return result
    visited.remove(tuple(map(tuple, state.board)))  # Unmark this state
    return None
```

```python
def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for board in reversed(path):
        for row in board:
            print(row)
        print()
```

```python
# Define the initial state and goal state
initial_state = PuzzleState(
    board=[[1, 2, 3],
           [4, 0, 5],
           [7, 8, 6]],
    zero_pos=(1, 1)
)
```

```python
goal_state = [
```

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

```
# Perform Iterative Deepening Search
max_depth = 20  # You can adjust this value
solution = ids(initial_state, goal_state, max_depth)
```

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

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

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

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

Program 4

Implement A* search algorithm

Algorithm:

classmate

Date

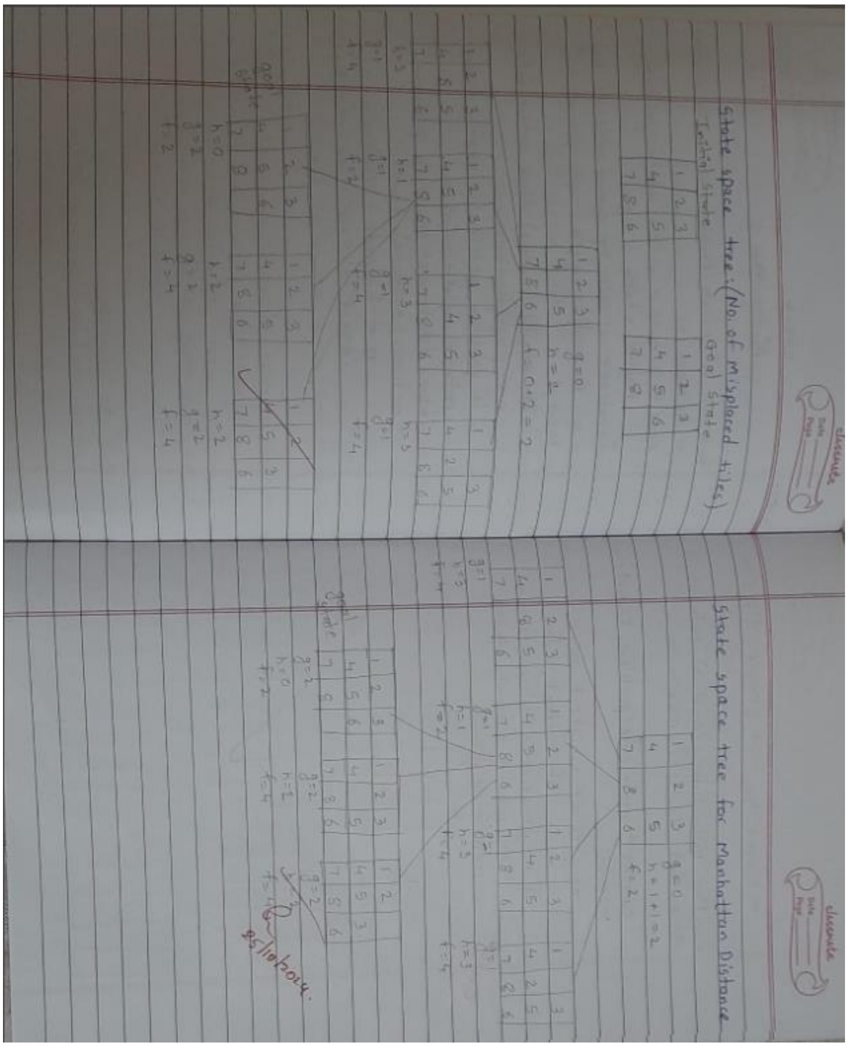
Page

25/10/24

LAB - 4

A* Algorithm

function A* search (Problem) returns a solution or failure
node ← a node n with n. state = problem. initial state, n.g = 0
frontier ← a priority queue ordered by ascending g+h, only element n.
loop do
if empty(frontier) then return failure
n ← pop (frontier)
if problem.goalTest(n.state) then return solution(n)
for each action a in problem. actions(n.state) do
n' ← childnode (problem.n, a)
insert (n', g(n') + h(n'), frontier)



Code:

Misplaced Tiles

```
def mistil(state, goal):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != goal[i][j]:
                count += 1
    return count
def findmin(open_list, goal):
    minv = float('inf')
    best_state = None
    for state in open_list:
        h = mistil(state['state'], goal)
        f = state['g'] + h
        if f < minv:
            minv = f
            best_state = state
    open_list.remove(best_state)
    return best_state

def operation(state):
```

```
    next_states = []
    blank_pos = find_blank_position(state['state'])
    for move in ['up', 'down', 'left', 'right']:
        new_state = apply_move(state['state'], blank_pos, move)
        if new_state:
            next_states.append({
                'state': new_state,
                'parent': state,
                'move': move,
                'g': state['g'] + 1
            })
    return next_states

def find_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def apply_move(state, blank_pos, move):
    i, j = blank_pos
    new_state = [row[:] for row in state]
    if move == 'up' and i > 0:
        new_state[i][j], new_state[i - 1][j] = new_state[i - 1][j], new_state[i][j]
    elif move == 'down' and i < 2:
        new_state[i][j], new_state[i + 1][j] = new_state[i + 1][j], new_state[i][j]
    elif move == 'left' and j > 0:
        new_state[i][j], new_state[i][j - 1] = new_state[i][j - 1], new_state[i][j]
    elif move == 'right' and j < 2:
        new_state[i][j], new_state[i][j + 1] = new_state[i][j + 1], new_state[i][j]
    else:
        return None
    return new_state

def print_state(state):
    for row in state:
        print(' '.join(map(str, row)))

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
goal_state = [[1,2,3], [8,0,4], [7,6,5]]
open_list = [{'state': initial_state, 'parent': None, 'move': None, 'g': 0}]
visited_states = []

while open_list:
    best_state = findmin(open_list, goal_state)
    print("Current state:")
    print_state(best_state['state'])
    h = mistil(best_state['state'], goal_state)
    f = best_state['g'] + h
```

```
    print(f"g(n): {best_state['g']}, h(n): {h}, f(n): {f}")
    if best_state['move'] is not None:
        print(f"Move: {best_state['move']}")
    print()
    if mistil(best_state['state'], goal_state) == 0:
        goal_state_reached = best_state
        break
    visited_states.append(best_state['state'])
    next_states = operation(best_state)
    for state in next_states:
        if state['state'] not in visited_states:
            open_list.append(state)

moves = []
while goal_state_reached['move'] is not None:
    moves.append(goal_state_reached['move'])
    goal_state_reached = goal_state_reached['parent']
moves.reverse()

print("\nMoves to reach the goal state:", moves)
print("\nGoal state reached:")
print_state(goal_state)
```

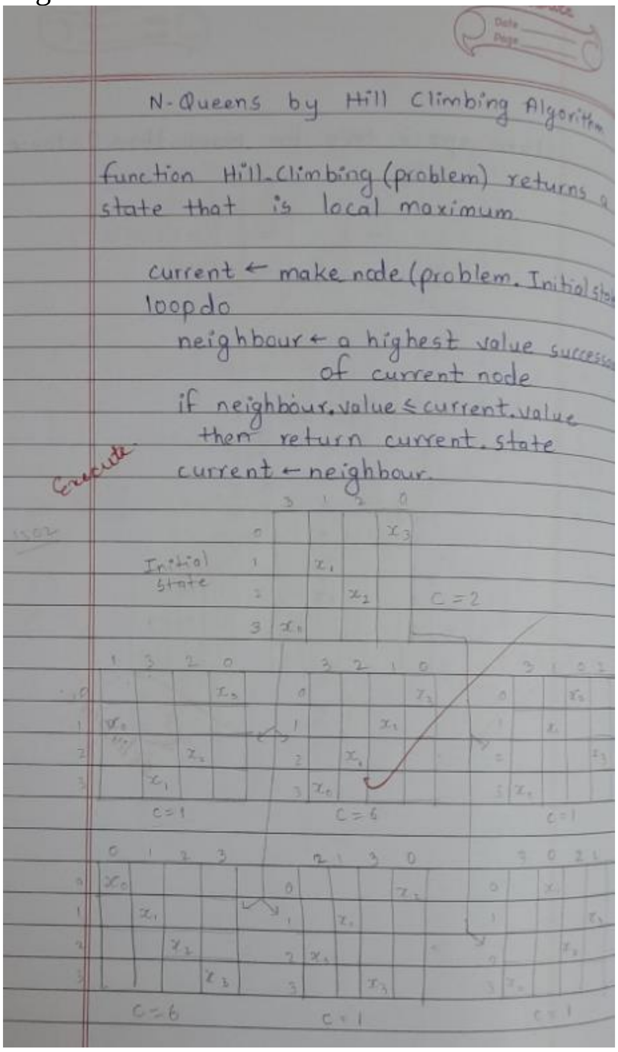
```
Current state:
2 8 3
1 6 4
7 0 5
g(n): 0, h(n): 5, f(n): 5

Current state:
2 8 3
1 0 4
7 6 5
g(n): 1, h(n): 3, f(n): 4
Move: up

Current state:
2 0 3
1 8 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: up

Current state:
2 8 3
0 1 4
7 6 5
g(n): 2, h(n): 4, f(n): 6
Move: left

Current state:
0 2 3
1 8 4
7 6 5
g(n): 3, h(n): 3, f(n): 6
Move: left

Current state:
1 2 3
0 8 4
7 6 5
g(n): 4, h(n): 2, f(n): 6
Move: down
```

```
Current state:
1 2 3
8 0 4
7 6 5
g(n): 5, h(n): 0, f(n): 5
Move: right


Moves to reach the goal state: ['up', 'up', 'left', 'down', 'right']

Goal state reached:
1 2 3
8 0 4
7 6 5
```

Program 5

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



N-Queens by Hill Climbing Algorithm

function Hill_Climbing (problem) returns a state that is local maximum

current ← make node (problem. Initial state)
loop do
  neighbour ← a highest value successor of current node
  if neighbour.value ≤ current.value
    then return current.state
  current ← neighbour

Execute

Initial state

C = 2

C = 1 C = 6 C = 1

C = 6 C = 1 C = 1

Code:

```
import random

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts


def hill_climbing(n):
    cost=0
    while True:
        # Initialize a random board
        current_board = list(range(n))
        random.shuffle(current_board)
        current_conflicts = calculate_conflicts(current_board)


        while True:
            # Generate neighbors by moving each queen to a different position
```

```
            found_better = False
            for i in range(n):
                for j in range(n):
                    if j != current_board[i]:  # Only consider different positions
                        neighbor_board = list(current_board)
                        neighbor_board[i] = j
                        neighbor_conflicts = calculate_conflicts(neighbor_board)
```

```
                        if neighbor_conflicts < current_conflicts:
                            current_board = neighbor_board
                            current_conflicts = neighbor_conflicts
                            cost+=1
                            found_better = True
                            break
                if found_better:
                    break
```

```
            # If no better neighbor found, stop searching
            if not found_better:
                break
```

```
        # If a solution is found (zero conflicts), return the board
        if current_conflicts == 0:
            return current_board, current_conflicts, cost
```

```
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['.'] * n
        row[board[i]] = 'Q'  # Place a queen
        print(' '.join(row))
    print()
```

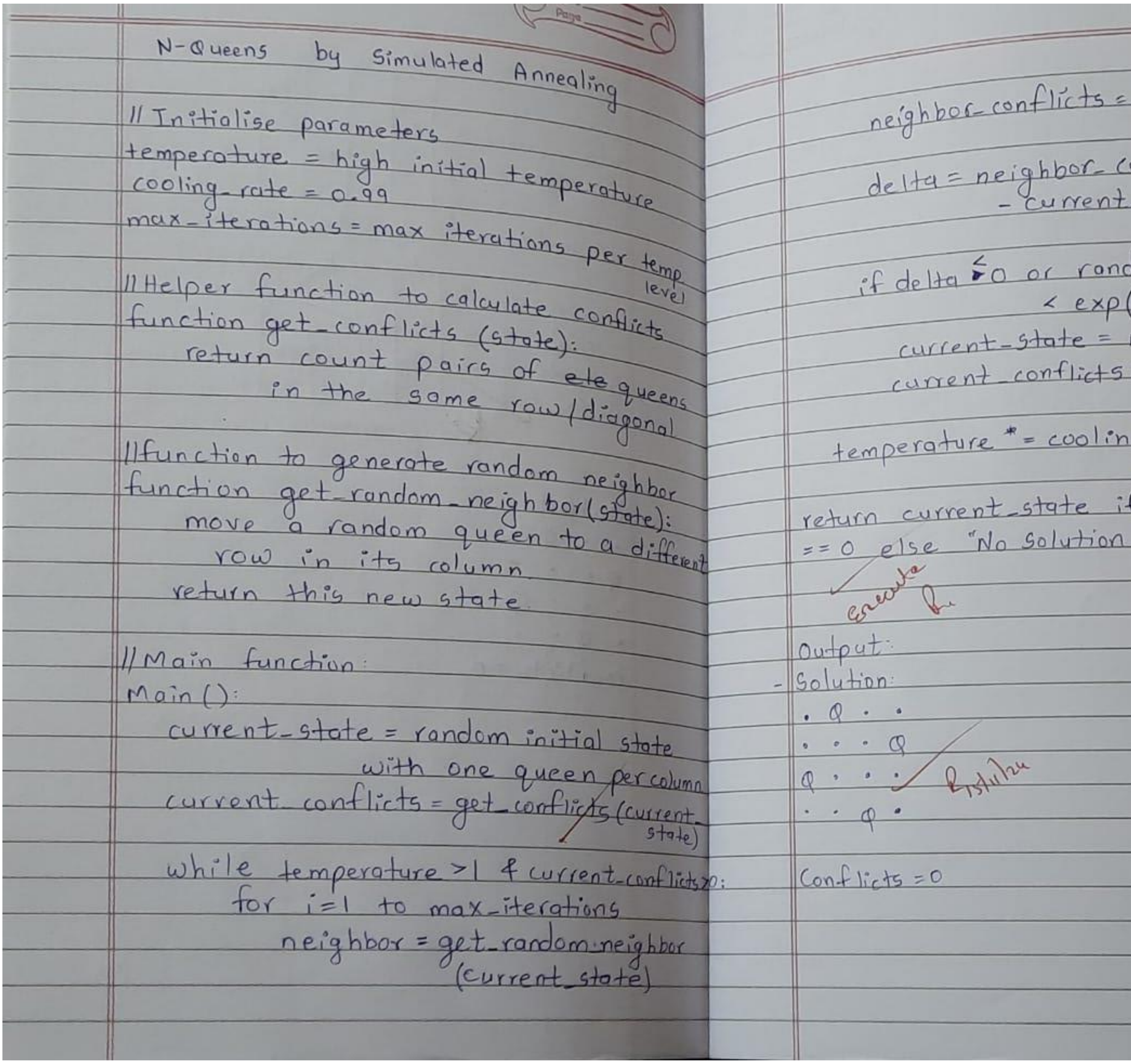
```
# Example Usage
n = 4
solution, conflicts, cost = hill_climbing(n)
print("Final Board Configuration:")
print_board(solution)
print("Number of Cost:", cost)
```

```
Final Board Configuration:
. Q . .
. . . Q
Q . . .
. . Q .

Number of Cost: 32
```

Program 6

Simulated Annealing to Solve 8-Queens problem

Algorithm:



N-Queens by Simulated Annealing

// Initialise parameters
temperature = high initial temperature
cooling_rate = 0.99
max_iterations = max iterations per temp level

// Helper function to calculate conflicts
function get_conflicts (state):
    return count pairs of ~~ele~~ queens in the same row/diagonal

// function to generate random neighbor
function get_random_neighbor(state):
    move a random queen to a different row in its column
    return this new state

// Main function:
Main():
    current_state = random initial state with one queen per column
    current_conflicts = get_conflicts(current_state)

    while temperature > 1 & current_conflicts > 0:
        for i=1 to max_iterations
            neighbor = get_random_neighbor (current_state)

neighbor_conflicts =

delta = neighbor_c
    - current

if delta ≤ 0 or ranc
    < exp(

current_state =
current_conflicts

temperature *= coolin

return current_state i
== 0 else "No solution

Output:
Solution:
. Q . .
. . . Q
Q . . .
. . Q .

Conflicts = 0

Code:

```
import numpy as np
from scipy.optimize import dual_annealing

def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
```

```
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1
    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

```
The best position found is: [2 4 1 7 0 6 3 5]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Code:

```
import itertools

# Function to evaluate an expression
def evaluate_expression(a, b, c, expression):
    # Use eval() to evaluate the logical expression
    return eval(expression)

# Function to generate the truth table and evaluate a logical expression
def truth_table_and_evaluation(kb, query):
    # All possible combinations of truth values for a, b, and c
    truth_values = [True, False]
    combinations = list(itertools.product(truth_values, repeat=3))

    # Reverse the combinations to start from the bottom (False -> True)
    combinations.reverse()

    # Header for the full truth table
    print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Evaluate the expressions for each combination
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)
```

```
        # Replace True/False with string "True"/"False"
        kb_result_str = "True" if kb_result else "False"
        query_result_str = "True" if query_result else "False"

        # Convert boolean values of a, b, c to "True"/"False"
        a_str = "True" if a else "False"
        b_str = "True" if b else "False"
        c_str = "True" if c else "False"

        # Print the results for the knowledge base and the query
        print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

    # Additional output for combinations where both KB and query are true
    print("\nCombinations where both KB and Query are True:")
    print(f"{'a':<5}   {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

    # Print only the rows where both KB and Query are True
    for combination in combinations:
        a, b, c = combination

        # Evaluate the knowledge base (KB) and query expressions
        kb_result = evaluate_expression(a, b, c, kb)
        query_result = evaluate_expression(a, b, c, query)

        # If both KB and query are True, print the combination
        if kb_result and query_result:
            a_str = "True" if a else "False"
            b_str = "True" if b else "False"
            c_str = "True" if c else "False"
            kb_result_str = "True" if kb_result else "False"
            query_result_str = "True" if query_result else "False"
            print(f"{a_str:<5} {b_str:<5} {c_str:<5} {kb_result_str:<20} {query_result_str:<20}")

# Define the logical expressions as strings
kb = "(a or c) and (b or not c)"  # Knowledge Base
query = "a or b"  # Query to evaluate

# Generate the truth table and evaluate the knowledge base and query
truth_table_and_evaluation(kb, query)
```

```
a        b      c      KB                   Query
False False False False                    False
False False True  False                    False
False True  False False                    True
False True  True  True                     True
True  False False True                     True
True  False True  False                    True
True  True  False True                     True
True  True  True  True                     True

Combinations where both KB and Query are True:
a        b      c      KB                   Query
False True  True  True                     True
True  False False True                     True
True  True  False True                     True
True  True  True  True                     True
```

Program 7

Implement unification in first order logic

Algorithm:

Unification

- Unification Algorithm:

Unify (Ψ_1, Ψ_2)

I If Ψ_1 or Ψ_2 is a variable / constant, then:

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

b) Else if Ψ_1 is a variable,

- if Ψ_1 occurs in Ψ_2, then return failure
- Else return $\{(\Psi_2/\Psi_1)\}$

c) Else if Ψ_2 is a variable,

- If Ψ_2 occurs in Ψ_1 then return failure
- Else return $\{(\Psi_1/\Psi_2)\}$

d) Else return failure.

II If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return failure

III If Ψ_1 & Ψ_2 have different no. of arguments return failure.

IV Set substitution set(SUBST) to NIL.

V for i=1 to the no. of elements in Ψ_1.

a) Call unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S=failure then returns failure.

c) If S≠NIL then do,

- Apply S to the remainder of both L1 & L2
- SUBST = APPEND (S, SUBST)

VI Return SUBST.

Code:

```
import re
```

```python
def occurs_check(var, x):
    """Checks if var occurs in x (to prevent circular substitutions)."""
    if var == x:
        return True
    elif isinstance(x, list):  # If x is a compound expression (like a function or predicate)
        return any(occurs_check(var, xi) for xi in x)
    return False
```

```python
def unify_var(var, x, subst):
    """Handles unification of a variable with another term."""
    if var in subst:  # If var is already substituted
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:  # Handle compound expressions
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):  # Check for circular references
        return "FAILURE"
    else:
        # Add the substitution to the set (convert list to tuple for hashability)
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst
```

```python
def unify(x, y, subst=None):
    """
    Unifies two expressions x and y and returns the substitution set if they can be unified.
    Returns 'FAILURE' if unification is not possible.
    """
    if subst is None:
        subst = {}  # Initialize an empty substitution set
```

```python
    # Step 1: Handle cases where x or y is a variable or constant
    if x == y:  # If x and y are identical
        return subst
    elif isinstance(x, str) and x.islower():  # If x is a variable
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():  # If y is a variable
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):  # If x and y are compound expressions (lists)
        if len(x) != len(y):  # Step 3: Different number of arguments
            return "FAILURE"
```

```python
        # Step 2: Check if the predicate symbols (the first element) match
        if x[0] != y[0]:  # If the predicates/functions are different
            return "FAILURE"
```

```python
        # Step 5: Recursively unify each argument
        for xi, yi in zip(x[1:], y[1:]):  # Skip the predicate (first element)
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:  # If x and y are different constants or non-unifiable structures
        return "FAILURE"
```

```python
def unify_and_check(expr1, expr2):
    """
    Attempts to unify two expressions and returns a tuple:
```

```
    (is_unified: bool, substitutions: dict or None)
    """
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result
```

```
def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})
```

```
def parse_input(input_str):
    """Parses a string input into a structure that can be processed by the unification algorithm."""
    # Remove spaces and handle parentheses
    input_str = input_str.replace(" ", "")
```

```
    # Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']])
    def parse_term(term):
        # Handle the compound term
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)\((.*)\)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term
```

```
    return parse_term(input_str)
```

```
# Main function to interact with the user
def main():
    while True:
        # Get the first and second terms from the user
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")
```

```
        # Parse the input strings into the appropriate structures
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)
```

```
        # Perform unification
        is_unified, result = unify_and_check(expr1, expr2)
```

```
        # Display the results
        display_result(expr1, expr2, is_unified, result)
```

```
        # Ask the user if they want to run another test
        another_test = input("Do you want to test another pair of expressions? (yes/no):
").strip().lower()
```

```
        if another_test != 'yes':
            break
```

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

```
Enter the first expression (e.g., p(x, f(y))): q(a,g(x,a),f(y))
Enter the second expression (e.g., p(a, f(z))): q(a,g(f(b),a),x)
Expression 1: ['q', 'a', 'g(x', 'a)', ['f', 'y']]
Expression 2: ['q', 'a', ['g', 'f(b'], 'a)', 'x']
Result: Unification Successful
Substitutions: {'g(x': ['g', 'f(b'], 'x': ['f', 'y']}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(z,x,f(g(z))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', 'z', 'x', ['f', 'g(z']]
Expression 2: ['p', 'z', ['f', 'y'], ['f', 'y']]
Result: Unification Successful
Substitutions: {'x': ['f', 'y'], 'g(z': 'y'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(x))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', ['f', 'a'], ['g', 'x']]
Expression 2: ['p', 'x', 'x']
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): no
```

Program 8
Create a knowledge base consisting of first order logic statements and prove the given query using
forward reasoning
Algorithm:

FOL [Forward Reasoning]

function FOL-FC-Ask (KB, α)
returns a substitution or false
inputs: KB, the knowledge based agent, a set of first-order definite clauses α, the query, an atomic sentence.
local variables: new, the new sentences inferred on each iteration
repeat until new is empty
new ← { }
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ Standardise-variables (rule)
for each θ such that subst(θ, $p_1 \wedge \ldots \wedge p_n$)
= subst (θ, $p_1' \wedge \ldots \wedge p_n'$)
for some $p_1' \ldots p_n'$ in KB
$q' \leftarrow$ subst (θ, q)
if q' doesn't unify with some sentence already in KB or new then
add q' to new
$\phi \leftarrow$ Unify (q', α)
if ϕ is not fail then return ϕ
add new to KB
return false.

* Output:
New facts inferred: {'Weapon(T1)', 'Sells(Robert, T1, A)', 'Hostile(A)'}
New facts inferred: {'Criminal(Robert)'}
Robert is a criminal

Code:

```
# Define the knowledge base (KB) as a set of facts
KB = set()

# Premises based on the provided FOL problem
KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')
```

```
# Define inference rules
def modus_ponens(fact1, fact2, conclusion):
    """ Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")
```

```
def forward_chaining():
    """ Perform forward chaining to infer new facts until no more inferences can be made """
    # 1. Apply: Missile(x) → Weapon(x)
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")
```

```
    # 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1)
    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")
```

```
    # 3. Apply: Hostile(A) from Enemy(A, America)
    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")
```

```
    # 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and
'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")
```

```
    # Check if we've reached our goal
    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")
```

```
# Run forward chaining to attempt to derive the conclusion
forward_chaining()
```

```
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)
Robert is a criminal!
```

Program 9

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

Algorithm:

20/12/24

Conjuctive Normal form (CNF)

Step I:
Eliminate implications and bidirectionals
$\alpha \rightarrow \beta$ is $\neg\alpha \vee \beta$
$\alpha \leftrightarrow \beta$ is $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$

Step II:
Move negation inside
$\neg(\forall p) \Rightarrow \exists p$
$\neg(\exists p) \Rightarrow \forall p$

Step III: Standardize variables
$(\forall x)(\exists y) \quad (\forall x)(\exists y)$
$(\forall x \ \exists y) \quad (\forall A \ \exists B)$

Step IV: Drop universal quantifiers
$\forall y \ f(y) \Rightarrow f(y)$

Step V: Skolemize variables
$\exists y(f(y)) \Rightarrow f(G_1)$

Step VI: Distribute $\vee$ over $\wedge$
$(\alpha \wedge \beta) \vee (A \wedge B)$
$(\alpha \vee A) \wedge (\alpha \vee B) \wedge (\beta \vee A) \wedge (\beta \vee B)$

Resolution

Step 1: Convert the statements to CNF

Step 2: Negate the statement that needs to be proved

Step 3: Repeat until there is contradiction or it is a failure
→ Take 2 clause known as parent clauses & resolve
→ Unify the variables if needed

Step 4: If unification is possible, it is proved by resolution.

Code:

```
# Define the knowledge base (KB)
KB = {
    "food(Apple)": True,
    "food(vegetables)": True,
    "eats(Anil, Peanuts)": True,
    "alive(Anil)": True,
    "likes(John, X)": "food(X)",  # Rule: John likes all food
    "food(X)": "eats(Y, X) and not killed(Y)",  # Rule: Anything eaten and not killed is food
    "eats(Harry, X)": "eats(Anil, X)",  # Rule: Harry eats what Anil eats
    "alive(X)": "not killed(X)",  # Rule: Alive implies not killed
    "not killed(X)": "alive(X)",  # Rule: Not killed implies alive
}

# Function to evaluate if a predicate is true based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]
```

```
    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]
        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())
```

```
    # If the predicate is a specific query (e.g., likes(John, Peanuts))
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        if func == "food" and args[0] == "Peanuts":
            return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)")
        if func == "likes" and args[0] == "John" and args[1] == "Peanuts":
            return resolve("food(Peanuts)")
```

```
    # Default to False if no rule or fact applies
    return False
```

```
# Query to prove: John likes Peanuts
query = "likes(John, Peanuts)"
result = resolve(query)
```

```
# Print the result
print(f"Does John like peanuts? {'Yes' if result else 'No'}")
```

```
Does John like peanuts? Yes
```

Program 10

Implement Alpha-Beta Pruning

Algorithm:

Alpha-Beta Pruning

```
function AlphaBetaSearch(state):
 · return action with the highest value
   from Maxvalue (state, -∞, +∞)

function MaxValue (state, α, β):
    if TerminalTest (state):
        return Utility (state)
    v = -∞
    for each action in Actions(state):
        v = max(v, MinValue (Result (state, action
                               α, β)
        if v ≥ β:
            return v
        α = max(α, v)
    return v

function MinValue (state, α, β):
    if Terminal(state)
        return Utility (state)
    v = ∞
    for each action in Actions(state)
        v = min(v, MaxValue (Result (state,
                                action), α, β)
        if v ≤ α:
            return v
        β = min(β, v)
    return
```

Code:

Alpha-Beta Pruning Implementation

def alpha_beta_pruning(node, alpha, beta, maxmizing_player):

```
    # Base case: If it's a leaf node, return its value (simulating evaluation of the node)
    if type(node) is int:
        return node

    # If not a leaf node, explore the children
    if maximizing_player:
        max_eval = -float('inf')
        for child in node:  # Iterate over children of the maximizer node
            eval = alpha_beta_pruning(child, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)  # Maximize alpha
            if beta <= alpha:  # Prune the branch
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in node:  # Iterate over children of the minimizer node
            eval = alpha_beta_pruning(child, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)  # Minimize beta
            if beta <= alpha:  # Prune the branch
                break
        return min_eval

# Function to build the tree from a list of numbers
def build_tree(numbers):
    # We need to build a tree with alternating levels of maximizers and minimizers
    # Start from the leaf nodes and work up
    current_level = [[n] for n in numbers]

    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
                next_level.append(current_level[i] + current_level[i + 1])  # Combine two nodes
            else:
                next_level.append(current_level[i])  # Odd number of elements, just carry forward
        current_level = next_level

    return current_level[0]  # Return the root node, which is a maximizer

# Main function to run alpha-beta pruning
def main():
    # Input: User provides a list of numbers
numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

    # Build the tree with the given numbers
    tree = build_tree(numbers)

    # Parameters: Tree, initial alpha, beta, and the root node is a maximizing player
```

```python
    alpha = -float('inf')
    beta = float('inf')
    maximizing_player = True  # The root node is a maximizing player

    # Perform alpha-beta pruning and get the final result
    result = alpha_beta_pruning(tree, alpha, beta, maximizing_player)

    print("Final Result of Alpha-Beta Pruning:", result)

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

```
Enter numbers for the game tree (space-separated): 10 9 14 18 5 4 50 3
Final Result of Alpha-Beta Pruning: 50
```