VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Akshat Basra (1BM22CS020)

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

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
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 out by **Akshat Basra (1BM22CS020),** 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.

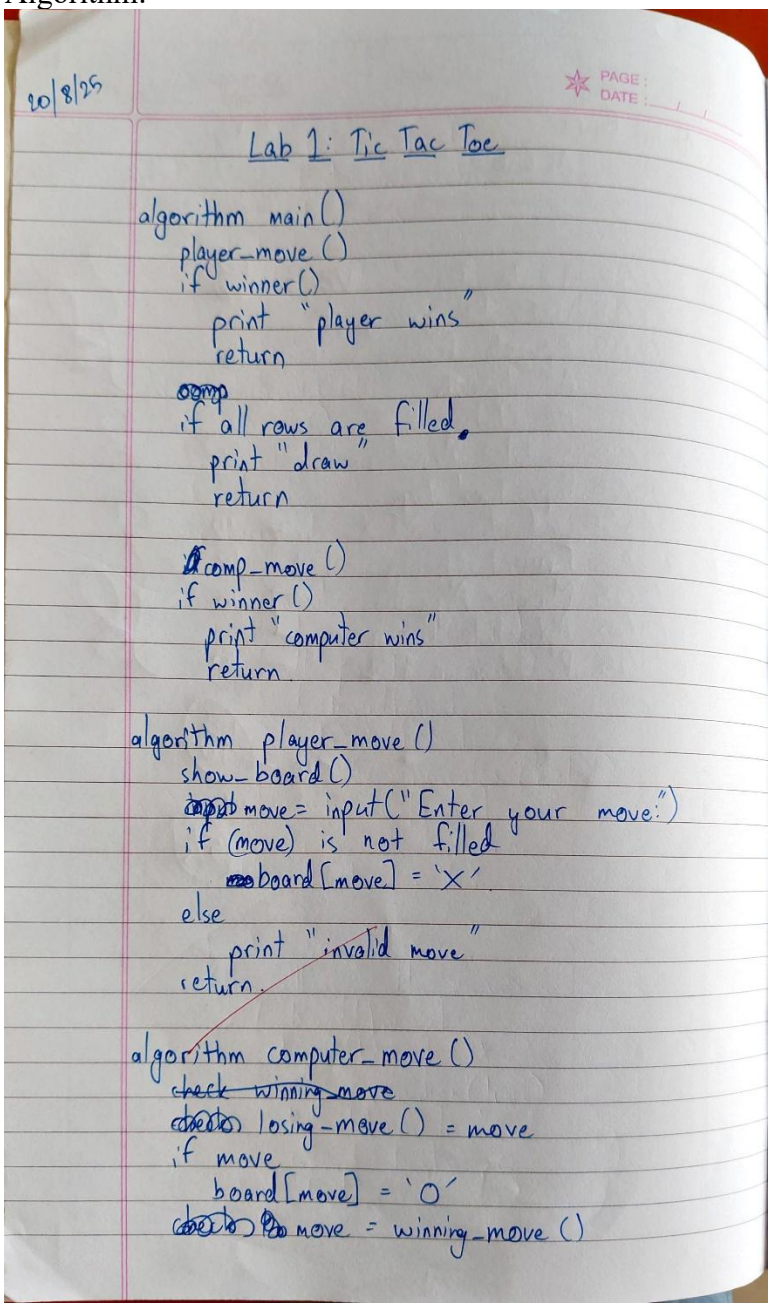
Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/AkshatBasra/AILab.git

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



20/8/25

PAGE :
DATE :

Lab 1: Tic Tac Toe

```
algorithm main()
    player_move()
    if winner()
        print "player wins"
        return
    if all rows are filled.
        print "draw"
        return

    comp_move()
    if winner()
        print "computer wins"
        return

algorithm player_move()
    show_board()
    move = input("Enter your move:")
    if (move) is not filled
        board[move] = 'X'
    else
        print "invalid move"
    return

algorithm computer_move()
    losing_move() = move
    if move
        board[move] = 'O'
    move = winning_move()
```

if move
    board[move] = 'O'
else
    move = random valid move.
    board[move] = 'O'

winning_move()
    if there are 2 Os in a row & 1 empty
        return missing_O_pos.

algorithm losing_move()
    if there are 2 Xs and 1 empty in a row
        return missing_X_pos

algorithm winner()
    if there are 3 Os or 3 Xs
        // check all winning arrangements
        return True
    return False.

Backtracking

Modify computer_move() and add minimax.

algorithm computer_move()
    best_score = - float('inf')
    best_move = None
    for all positions
        if board = '-'
            board = 'O'
            score = minimax(False)
            board = '-'   // backtrack.

```
        if score > best_score
            best_score = score
            best_move = move.
    if best_move
        board[move] = 'O'
```

```
algorithm minimax(& is_computer)
    if winner()
        return winner.
    if draw
        return 'draw'

    if is_computer:
        best_score = -float('inf')
        for all possible moves
            if board[move] = '-'
                board[move] = 'O'
                score = minimax(False)
                board[move] = '-'          //backtrack
                record best_score.
        return best_score
    else
        best_score = -∞.
        for all possible moves
            if board[move] = '-'
                board[move] = 'X'
                score = minimax(True)
                board[move] = '-'
                record best_score.
        return best_score.
```

o/p - ?

Vaccum Cleaner Robot.

```
algorithm Robot (state)
  if state.status = dirty
    return clean.
  if state.position[0] = 0
    return down
  if state.position[0] = 1
    if state.position[1] = 0
      return left
    return right
  return up.

algorithm main()
  initial_state = (0, 0)
  while room is not clean
    action = robot(state)
    change_state (action)
```

Code:

```
import random

board = [['-', '-', '-'],
        ['-', '-', '-'],
        ['-', '-', '-']]

playcount = 0

def print_board():
    for row in board:
        print(' '.join(row))

def check_winner():
    # Check rows
    for row in board:
        if row[0] == row[1] == row[2] != '-':
            return row[0]

    # Check columns
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != '-':
            return board[0][col]

    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != '-':
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != '-':
        return board[0][2]

    return None

def minimax(is_computer):
    winner = check_winner()
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1

    # Check for draw
    if all(cell != '-' for row in board for cell in row):
        return 0

    if is_computer:
        best_score = -float('inf')
        for row in range(3):
            for col in range(3):
```

```
                if board[row][col] == '-':
                    board[row][col] = 'O'
                    score = minimax(False)
                    board[row][col] = '-'
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for row in range(3):
            for col in range(3):
                if board[row][col] == '-':
                    board[row][col] = 'X'
                    score = minimax(True)
                    board[row][col] = '-'
                    best_score = min(score, best_score)
        return best_score

def computer_move():
    best_score = -float('inf')
    best_move = None
    for row in range(3):
        for col in range(3):
            if board[row][col] == '-':
                board[row][col] = 'O'
                score = minimax(False)
                board[row][col] = '-'
                if score > best_score:
                    best_score = score
                    best_move = (row, col)

    if best_move:
        row, col = best_move
        board[row][col] = 'O'

def player_move():
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            if move < 0 or move > 8:
                print("Invalid move. Try again.")
                continue
            row = move // 3
            col = move % 3
            if board[row][col] != '-':
                print("Cell already taken. Try again.")
                continue
            board[row][col] = 'X'
```

```
            break
        except ValueError:
            print("Invalid input. Please enter a number between 1 and 9.")

while True:
    print_board()
    player_move()
    playcount += 1
    winner = check_winner()
    if winner:
        print_board()
        print("You win!")
        break
    if playcount == 9:
        print_board()
        print("It's a draw!")
        break

    computer_move()
    playcount += 1
    winner = check_winner()
    if winner:
        print_board()
        print("Computer wins!")
        break
    if playcount == 9:
        print_board()
        print("It's a draw!")
        break
```

Output:

```
Tic Tac Toe — You (O) vs Computer (X)
Positions:
1 | 2 | 3
4 | 5 | 6
7 | 8 | 9

Do you want to play first? (y/n): y

  |   |
  |   |
  |   |

Computer chooses position 1

X |   |
  |   |
  |   |

Enter your move (1–9): 5

X |   |
  | O |
  |   |

Computer chooses position 2

X | X |
  | O |
  |   |

Enter your move (1–9): 3
```

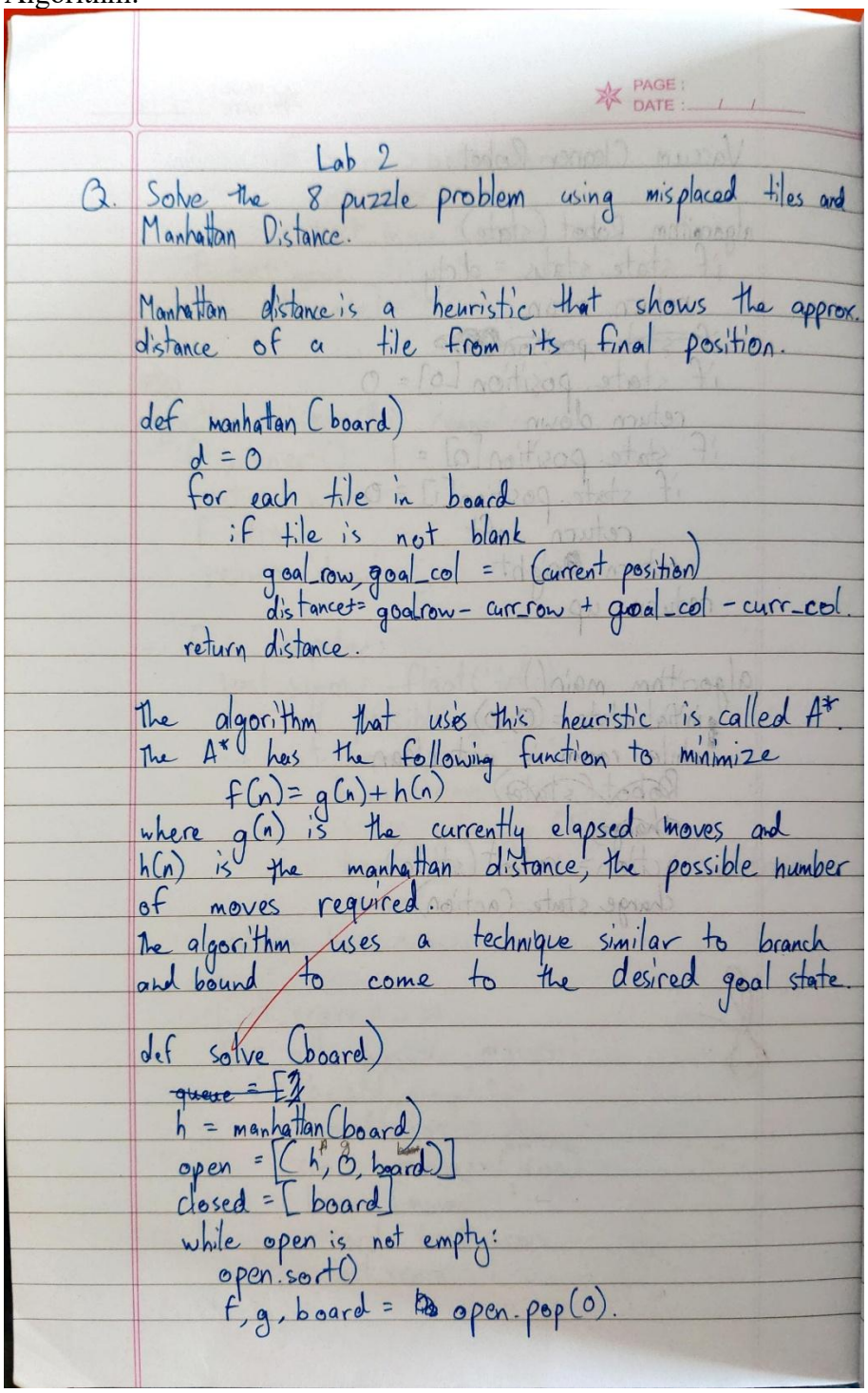
```
X | X | O
O | O | X
X | O | X

It's a draw!
```

```
X | X | O
  | O |
  |   |

Computer chooses position 7

X | X | O
  | O |
X |   |

Enter your move (1–9): 4

X | X | O
O | O |
X |   |

Computer chooses position 6

X | X | O
O | O | X
X |   |

Enter your move (1–9): 7
Invalid move. Try again.

X | X | O
O | O | X
X |   |

Enter your move (1–9): 8

X | X | O
O | O | X
X | O |

Computer chooses position 9
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



Lab 2

Q. Solve the 8 puzzle problem using misplaced tiles and Manhattan Distance.

Manhattan distance is a heuristic that shows the approx. distance of a tile from its final position.

```
def manhattan(board)
    d = 0
    for each tile in board
        if tile is not blank
            goal_row, goal_col = (current position)
            distance = goalrow - curr_row + goal_col - curr_col.
    return distance.
```

The algorithm that uses this heuristic is called A*. The A* has the following function to minimize

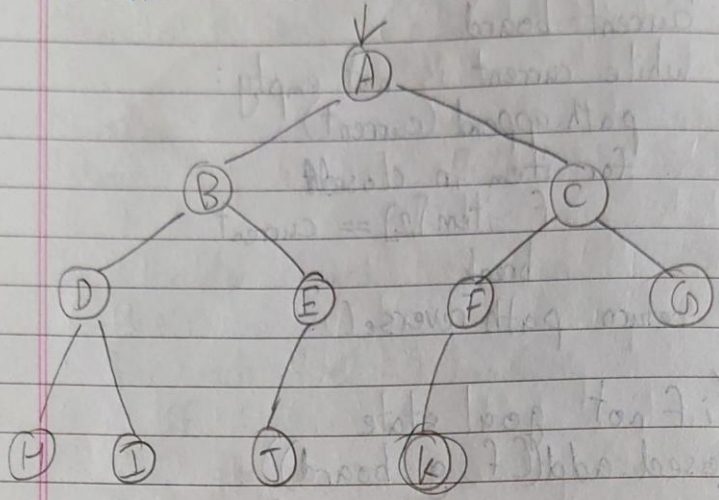
$$f(n) = g(n) + h(n)$$

where g(n) is the currently elapsed moves and h(n) is the manhattan distance, the possible number of moves required.
The algorithm uses a technique similar to branch and bound to come to the desired goal state.

```
def solve (board)
    ~~queue = []~~
    h = manhattan(board)
    open = [(h, 0, board)]
    closed = [board]
    while open is not empty:
        open.sort()
        f, g, board = open.pop(0).
```

```
if board == goal          // reverse track all moves
    path = []
    current = board
    while current is not empty:
        path.append(current)
        for item in closed:
            if item[2] == current
                break
    return path.reverse().

// if not goal state.
closed.add((f, g, board))
for every neighbour_board
    if neighbour_board is in closed: // dont revisit.
        continue
    new_g = g + 1
    new_h = manhattan(neighbour_board)
    f = new_g + new_h.
    if neighbour_board in open:
        update open.
    else:
        insert into open.
```

Q. Solve this using Iterative Deepening Depth first Search.



IDDFS is a hybrid of DFS and BFS.
It requires a DLS (Depth limit search).

```
def DLS (graph, limit, node)
  if node == goal
    return node  //as solution.
  else if limit = 0
    return cutoff.
  else
    for each action/children in moves:
      child = move(board)
      res = DLS (graph, child, limit - 1)

def IDDFS (graph)
  for depth = 0 to height (graph)
    res = DLS (graph, root, depth).
```

The shortest path for this graph is ['A', 'C', 'F', 'K']

Code:

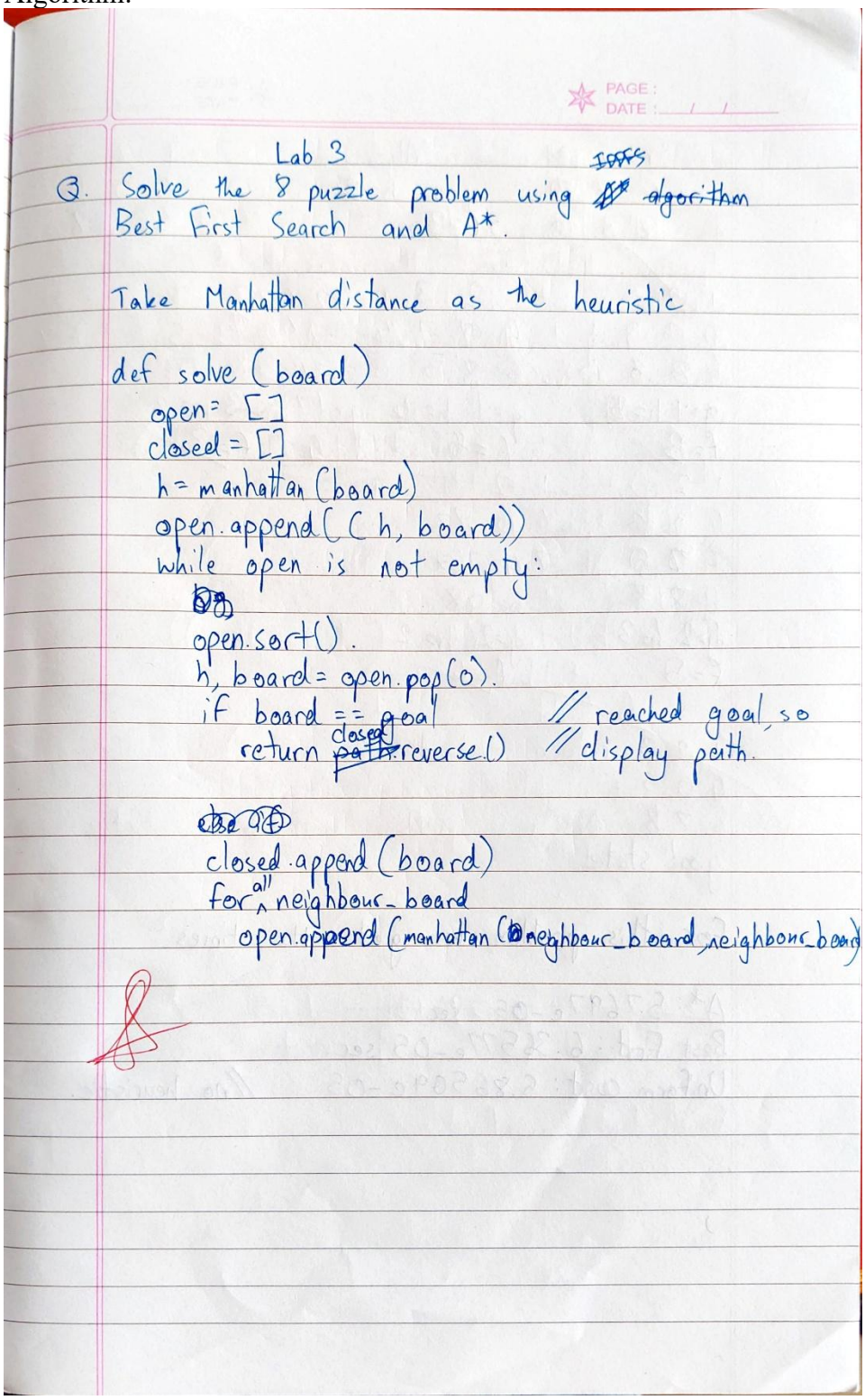
```
def DLS(graph, node, limit, goal, path):
    if node == goal:
        return path
    if limit <= 0:
        return None
    if node in graph:
        for neighbour in graph[node]:
            new_path = path.copy()
            new_path.append(neighbour)
            result = DLS(graph, neighbour, limit - 1, goal, new_path)
            if result is not None:
                return result


def IDDFS(graph, start, goal):
    depth = 0
    while True:
        path = DLS(graph, start, depth, goal, [start])
        if path is not None:
            return path
        depth += 1

my_graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F', 'G'],
        'D': ['H', 'I'],
        'E': ['J'],
        'F': ['K']
    }

start_node = 'A'
goal_node = 'K'

shortest_path = IDDFS(my_graph, start_node, goal_node)

if shortest_path:
    print("Shortest path found:", shortest_path)
else:
    print("No path found.")
```

Output:

```
Shortest path found: ['A', 'C', 'F', 'K']
```

Program 3
Implement A* search algorithm

Algorithm:

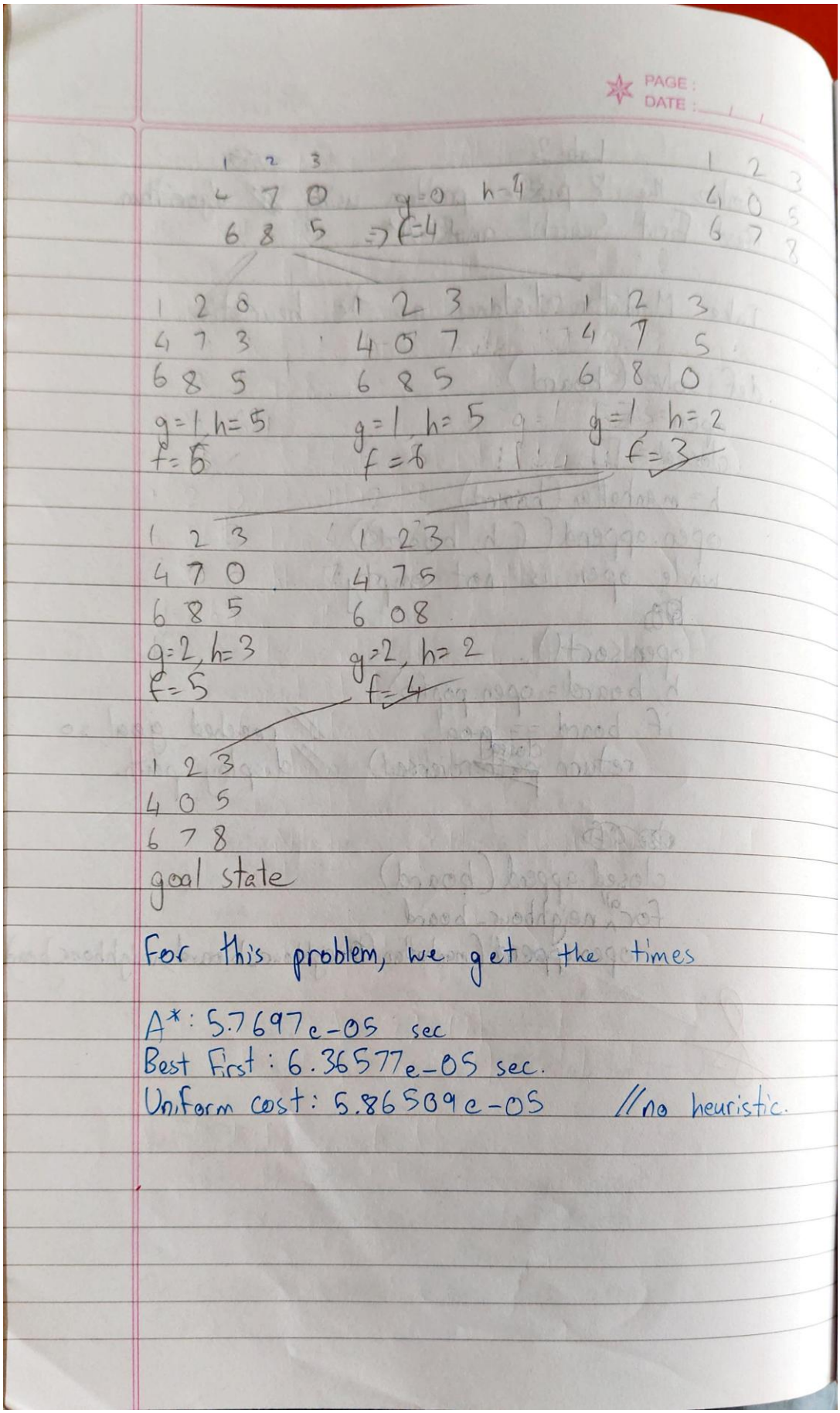


Lab 3

Q. Solve the 8 puzzle problem using ~~algorithm~~ Best First Search and A*.

Take Manhattan distance as the heuristic

```
def solve (board)
    open = []
    closed = []
    h = manhattan (board)
    open.append ( (h, board))
    while open is not empty:

        open.sort().
        h, board = open.pop(0).
        if board == goal                    // reached goal, so
            return closed.reverse()         // display path.

        closed.append (board)
        for all neighbour_board
            open.append (manhattan (neighbour_board, neighbour_board)
```


PAGE :
DATE :
1 2 3
4 7 0
6 8 5
g=0 h=4
=>f=4
1 2 0
4 7 3
6 8 5
g=1 h=5
f=6
1 2 3
4 0 7
6 8 5
g=1 h=5
f=6
1 2 3
4 7 5
6 8 0
g=1 h=2
f=3
1 2 3
4 7 0
6 8 5
g=2 h=3
f=5
1 2 3
4 7 5
6 0 8
g=2 h=2
f=4
1 2 3
4 0 5
6 7 8
goal state
For this problem, we get the times
A*: 5.7697e-05 sec
Best First: 6.3657e-05 sec.
Uniform cost: 5.86509e-05 //no heuristic.

PAGE :
DATE : / /

Problem 2:

3 2 8
4 0 7
1 5 6

Solution in 20 steps.

A*: 0.06017
Best first: 0.01033
Uniform cost: 0.5289

10/9/25

Code:

```
import heapq
import time

# Goal state
GOAL = [[1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]]

moves = [(-1,0), (1,0), (0,-1), (0,1)]

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

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

def neighbors(state):
    x, y = find_blank(state)
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            yield new_state

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def astar(start):
    open = []
    heapq.heappush(open, (manhattan(start), 0, start, []))  # (f, g, state, path)
    closed = set()

    while open:
        f, g, state, path = heapq.heappop(open)
        if state == GOAL:
```

```
            return path + [state]

        state_id = state_to_tuple(state)
        if state_id in closed:
            continue
        closed.add(state_id)

        for next in neighbors(state):
            heapq.heappush(open, (g+1 + manhattan(next), g+1, next, path + [state]))

    return None

def print_board(board):
    for row in board:
        print(row)
    print()

start_state = [[3, 2, 8],
               [4, 0, 7],
               [1, 5, 6]]
print("Random Start State:")
print_board(start_state)

start = time.time()
solution = astar(start_state)
end = time.time()

if solution:
    print("Solution found in", len(solution)-1, "moves:")
    for step in solution:
        print_board(step)
    print("Time taken: ", end - start)
else:
    print("No solution exists")
```

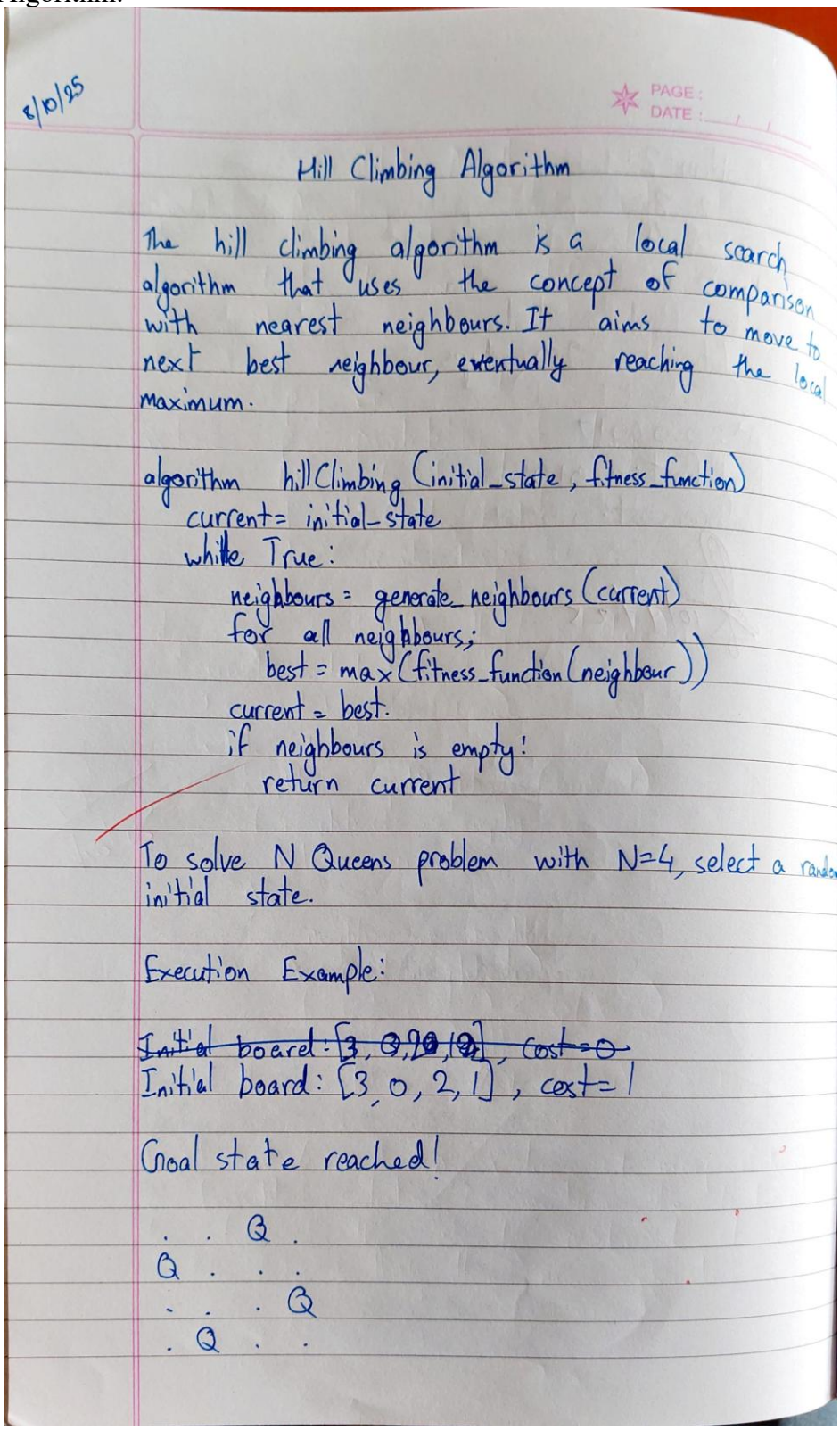
Output:

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

Solution found in 20 moves:
[3, 2, 8]
[4, 0, 7]
[1, 5, 6]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Time taken:  0.28453493118286133
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



8/10/25

PAGE :
DATE : / /

Hill Climbing Algorithm

The hill climbing algorithm is a local search algorithm that uses the concept of comparison with nearest neighbours. It aims to move to next best neighbour, eventually reaching the local maximum.

```
algorithm hillClimbing (initial_state, fitness_function)
    current = initial_state
    while True:
        neighbours = generate_neighbours (current)
        for all neighbours;
            best = max (fitness_function (neighbour))
        current = best.
        if neighbours is empty:
            return current
```

To solve N Queens problem with N=4, select a random initial state.

Execution Example:

~~Initial board: [3, 0, 2, 1], cost = 0~~
Initial board: [3, 0, 2, 1], cost = 1

Goal state reached!

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

n value can be changed to higher values to solve them. By allowing sideways movement in plateaus, we can decrease the chance of no solution found.

Code:

```
import random

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if board[i] == j else '.' for j in range(n)]
        print(' '.join(row))
    print()

def compute_conflicts(board):
    n = len(board)
    conflicts = 0
    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 get_best_neighbor(board):
    n = len(board)
    current_conflicts = compute_conflicts(board)
    best_board = board[:]
    best_conflicts = current_conflicts

    for i in range(n):
        for j in range(i + 1, n):
            new_board = board[:]
            new_board[i], new_board[j] = new_board[j], new_board[i]
            new_conflicts = compute_conflicts(new_board)
            if new_conflicts < best_conflicts:
                best_conflicts = new_conflicts
                best_board = new_board[:]

    return best_board, best_conflicts

def hill_climbing(n):
    board = random.sample(range(n), n)
    print(f"Initial board: {board}, Cost = {compute_conflicts(board)}\n")
    sideways_moves = 0
    max_sides = 2

    while True:
        print(f"{board} -> Cost = {compute_conflicts(board)}")
        neighbor, neighbor_conflicts = get_best_neighbor(board)

        current_conflicts = compute_conflicts(board)
```

```
        if neighbor_conflicts < current_conflicts:
            board = neighbor
            sideways_moves = 0
        elif neighbor_conflicts == current_conflicts and sideways_moves < max_sides:
            print(f"Sideways here: {sideways_moves}")
            board = neighbor
            sideways_moves += 1
        else:
            break
        if neighbor_conflicts == 0:
            print("\nGoal state reached!")
            print_board(board)
            return
    print("\nNo solution found from this start.")

hill_climbing(4)
```

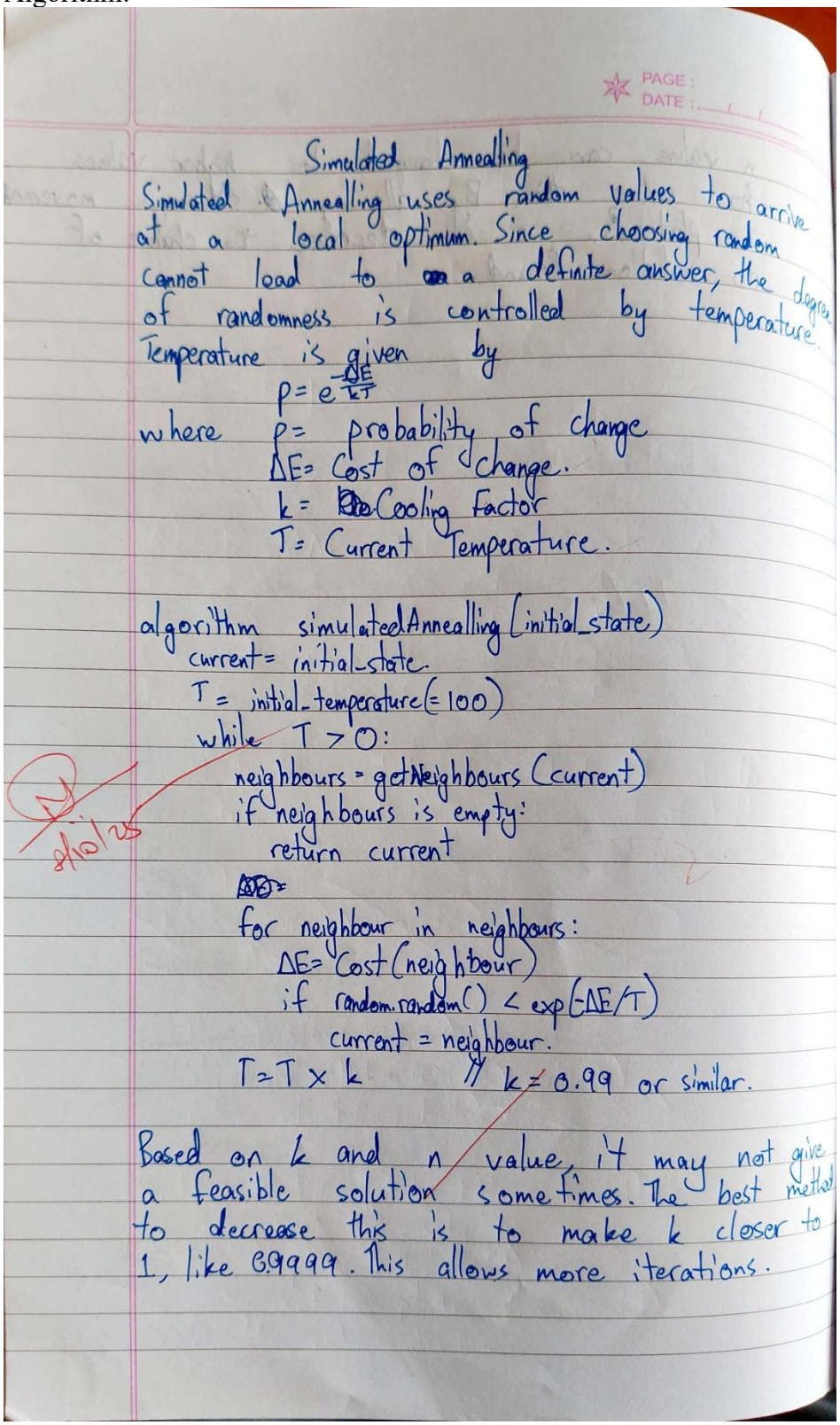
Output:

```
Initial board: [0, 1, 3, 2], Cost = 2

[0, 1, 3, 2] -> Cost = 2
[3, 1, 0, 2] -> Cost = 1

Goal state reached!
. Q . .
. . . Q
Q . . .
. . Q .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Annealling

Simulated Annealling uses random values to arrive at a local optimum. Since choosing random cannot lead to a definite answer, the degree of randomness is controlled by temperature. Temperature is given by

$$p = e^{\frac{-\Delta E}{kT}}$$

where p = probability of change
ΔE = Cost of change.
k = Cooling Factor
T = Current Temperature.

```
algorithm simulatedAnnealling(initial_state)
    current = initial_state
    T = initial_temperature(=100)
    while T > 0:
        neighbours = getNeighbours(current)
        if neighbours is empty:
            return current
        for neighbour in neighbours:
            ΔE = Cost(neighbour)
            if random.random() < exp(-ΔE/T)
                current = neighbour.
        T = T x k        // k = 0.99 or similar.
```

Based on k and n value, it may not give a feasible solution sometimes. The best method to decrease this is to make k closer to 1, like 0.9999. This allows more iterations.

PAGE :
DATE : / /

Example:

Initial State:

. Q . .
Q . . .
Q . . .
. Q . .

Solution Reached.

. . Q .
Q . . .
. . . Q
. Q . .

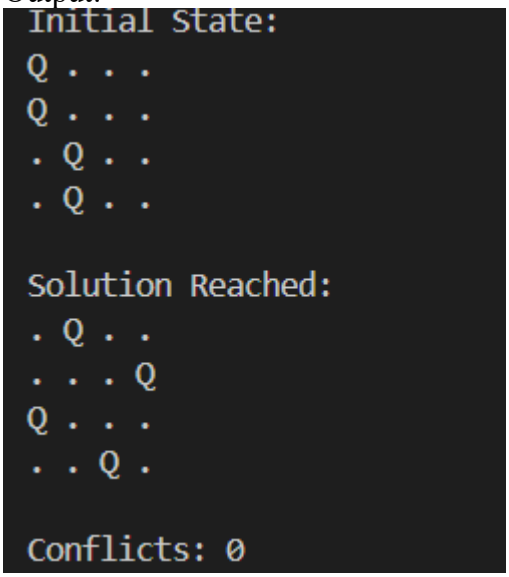
8/10

Code:

```
import random, math

def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if board[i] == j else '.' for j in range(n)]
        print(' '.join(row))
    print()

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

def random_neighbor(state):
    n = len(state)
    new_state = state[:]
    col = random.randint(0, n-1)
    new_row = random.randint(0, n-1)
    while new_row == new_state[col]:
        new_row = random.randint(0, n-1)
    new_state[col] = new_row
    return new_state

def simulated_annealing(n, T=1.0, k=0.999):
    state = [random.randint(0, n-1) for _ in range(n)]
    print("Initial State:")
    print_board(state)
    while True:
        if cost(state) == 0:
            return state
        neighbor = random_neighbor(state)
        delta = cost(neighbor) - cost(state)

        if delta < 0 or random.random() < math.exp(-delta / T):
            state = neighbor

        T *= k
        if T < 1e-6:
            break
    return state
```

```
solution = simulated_annealing(4)
print("Solution Reached:")
print_board(solution)
print(f"Conflicts: {cost(solution)}")
```

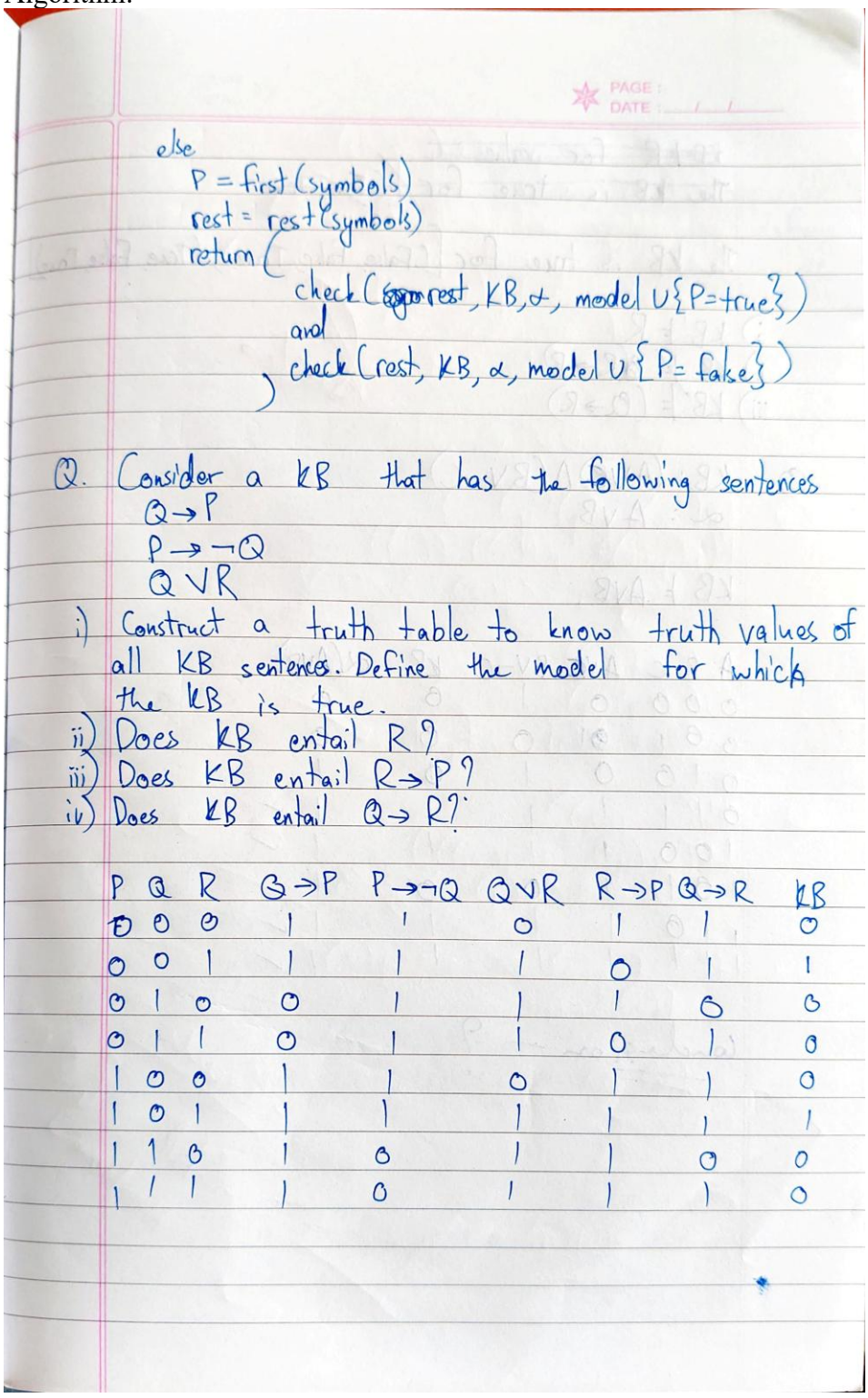
Output:



Program 6

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

Algorithm:



else
P = first(symbols)
rest = rest(symbols)
return (
check(rest, KB, α, model ∪ {P=true})
and
check(rest, KB, α, model ∪ {P=false})
)

Q. Consider a KB that has the following sentences
Q → P
P → ¬Q
Q ∨ R

i) Construct a truth table to know truth values of all KB sentences. Define the model for which the KB is true.
ii) Does KB entail R?
iii) Does KB entail R → P?
iv) Does KB entail Q → R?

P	Q	R	Q→P	P→¬Q	Q∨R	R→P	Q→R	KB
0	0	0	1	1	0	1	1	0
0	0	1	1	1	1	0	1	1
0	1	0	0	1	1	1	0	0
0	1	1	0	1	1	0	1	0
1	0	0	1	1	0	1	1	0
1	0	1	1	1	1	1	1	1
1	1	0	1	0	1	1	0	0
1	1	1	1	0	1	1	1	0

Propositional Logic

Propositional logic aims to determine if a statement is true, given a knowledge base that has statements known to be true. In other words, we must know if the knowledge base entails the statement

$KB \models \alpha$.

The notations used for sentences are

A	B	$A \wedge B$	$A \vee B$	$\neg A$	$A \Rightarrow B$	$A \Leftrightarrow B$
T	T	T	T	F	T	T
T	F	F	T	F	F	F
F	T	F	T	T	T	F
F	F	F	F	T	T	T

To find the entails, we check if both statements are true. For this, we make a truth table.

Example:

~~$KB = (A \vee C) \wedge (B \vee \neg C)$~~

~~$\alpha = A \vee B$~~

```
function Entails (KB, α)
    symbols = list ((KB+α).split())
    return check(symbols, KB, α, {})

function check(symbols, KB, α, model)
    if symbols is empty,
        if pl-true (KB, model)
            return pl-true (α, model)
        else return true.
```

~~KB ⊨ R for values t~~
~~The KB is true for [(True, True~~

The KB is true for [(False, False, True), (True, False, True)]

i) KB ⊨ R
ii) KB ⊭ (R → P)
iii) KB ⊨ (Q → R)

Q. KB : (A ∨ B) ∧ (B ∨ ¬C)
α : A ∨ B

KB ⊨ A ∨ B.

A	B	C	A ∨ B	B ∨ ¬C	KB	α (A ∨ B)
0	0	0	0	1	0	0
0	0	1	0	0	0	0
0	1	0	1	1	1	1 ✓
0	1	1	1	1	1	1 ✓
1	0	0	1	1	1	1 ✓
1	0	1	1	0	0	1
1	1	0	1	1	1	1 ✓
1	1	1	1	1	1	1 ✓

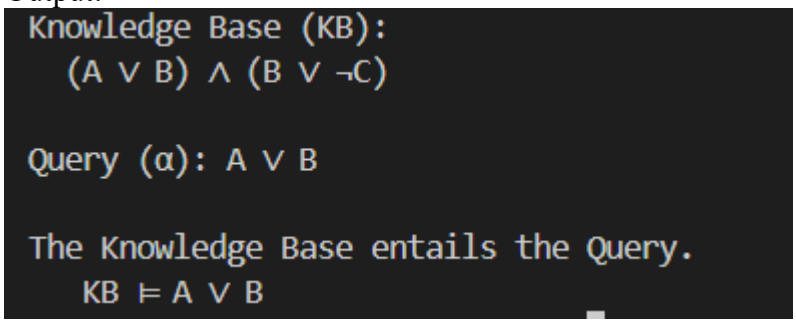
Conclusion — ?

Code:

```
import re
from itertools import product

def pl_true(sentence, model):
    try:
        return eval(sentence, {}, model)
    except NameError:
        return False

def translate(sentence):
    sentence = sentence.replace('↔', '==')
    sentence = sentence.replace('¬', ' not ')
    sentence = sentence.replace('∧', ' and ')
    sentence = sentence.replace('∨', ' or ')

    sentence = re.sub(r'([A-Z])\s*→\s*(not [A-Z]|[A-Z])', r'(not \1 or \2)', sentence)
    return sentence

def tt_entails(kb_list, alpha):
    kb = " and ".join(f"({translate(stmt)})" for stmt in kb_list)
    alpha = translate(alpha)

    symbols = sorted(list(set(re.findall(r'[A-Z]', kb + alpha))))

    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if pl_true(kb, model):
            if not pl_true(alpha, model):
                return False
    return True

if __name__ == "__main__":
    kb_list = [
        "(A ∨ B) ∧ (B ∨ ¬C)"
    ]
    alpha_formula = "A ∨ B"

    print("Knowledge Base (KB):")
    for stmt in kb_list:
        print(" ", stmt)
    print(f"\nQuery (α): {alpha_formula}\n")

    result = tt_entails(kb_list, alpha_formula)

    if result:
        print(f"The Knowledge Base entails the Query.")
```

```
    print(f"   KB ⊨ {alpha_formula}")
else:
    print(f"The Knowledge Base does NOT entail the Query.")
    print(f"   KB ⊭ {alpha_formula}")
```

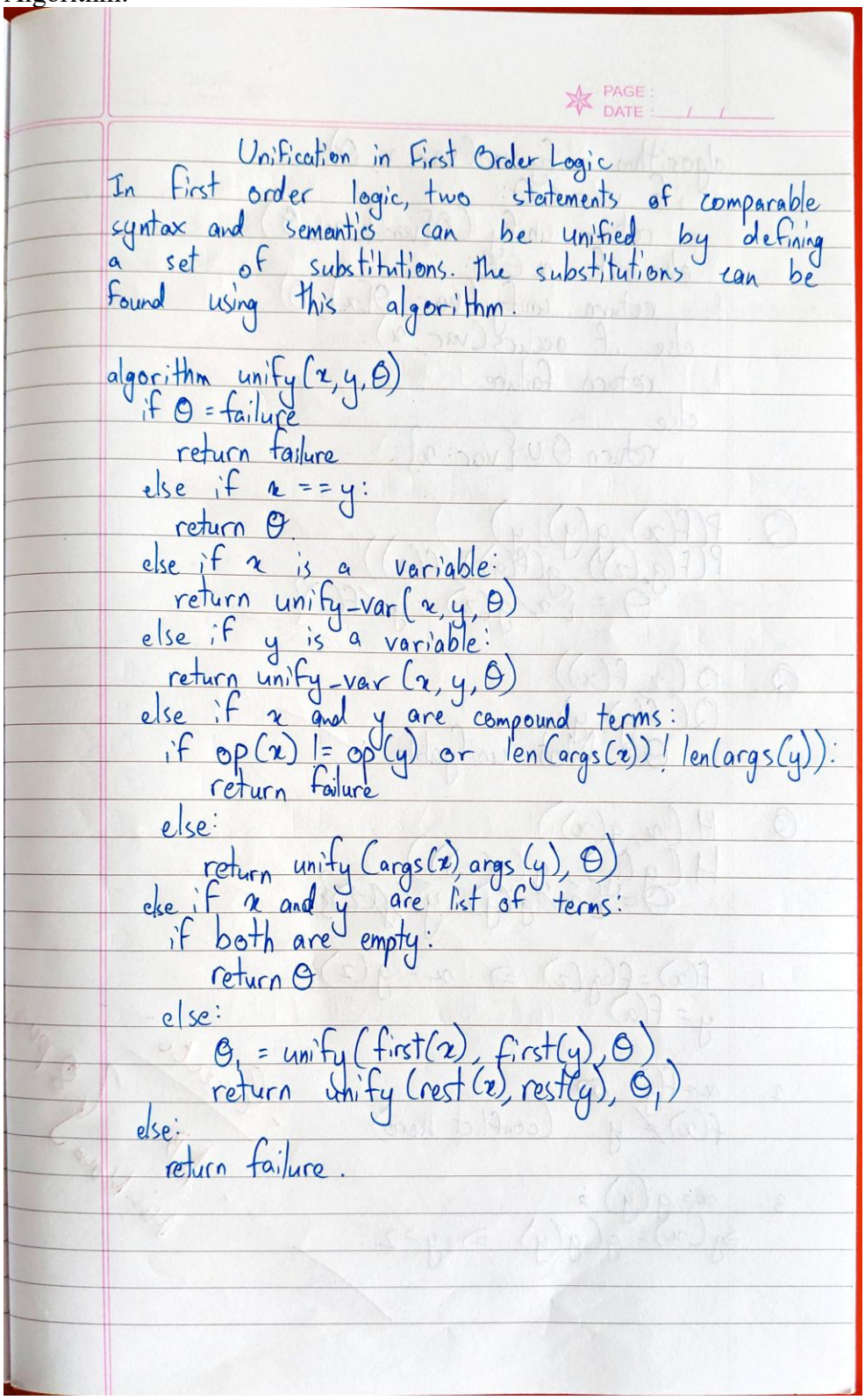
Output:



Program 7

Implement unification in first order logic

Algorithm:



Unification in First Order Logic

In first order logic, two statements of comparable syntax and semantics can be unified by defining a set of substitutions. The substitutions can be found using this algorithm.

```
algorithm unify(x, y, Θ)
  if Θ = failure
    return failure
  else if x == y:
    return Θ
  else if x is a variable:
    return unify_var(x, y, Θ)
  else if y is a variable:
    return unify_var(x, y, Θ)
  else if x and y are compound terms:
    if op(x) != op(y) or len(args(x)) != len(args(y)):
      return failure
    else:
      return unify(args(x), args(y), Θ)
  else if x and y are list of terms:
    if both are empty:
      return Θ
    else:
      Θ1 = unify(first(x), first(y), Θ)
      return unify(rest(x), rest(y), Θ1)
  else:
    return failure.
```

PAGE :
DATE : / /

```
algorithm unify_var(var, x, θ):
  if var ∈ θ:
    return unify(θ[var], x, θ)
  else if x ∈ θ:
    return unify(var, θ[x], θ)
  else if occurs(var, x):
    return failure
  else
    return θ ∪ {var: x}.
```

Q. P(f(x), g(y), y)
P(f(g(z)), g(f(a)), f(a))
θ = {x/g(z), y/f(a)}.

Q. Q(x, f(x))
Q(f(y), y)
θ = Not unifyable.

Q. H(x, g(x))
H(g(y), g(g(z)))
θ = {x/g(y), y/g(z)}

1. f(x) = f(g(z)) ⇒ x = g(z)
y = f(a)

2. x = f(y)
f(x) ≠ y (conflict here)

3. x = g(y)
⇒ g(x) = g(g(y)) ⇒ y = z.

Resolution
AI
Min Max

Code:

```
import re

def parse(expr):
    tokens = re.findall(r"[A-Za-z_][A-Za-z0-9_]*|[(),]", expr)

    def parse_rec():
        token = tokens.pop(0)
        if tokens and tokens[0] == '(':
            tokens.pop(0)
            args = []
            while tokens[0] != ')':
                args.append(parse_rec())
                if tokens[0] == ',':
                    tokens.pop(0)
            tokens.pop(0)
            return (token, args)
        return token

    return parse_rec()

isvar = lambda x: isinstance(x, str) and x[0].islower()

def occurs(v, x, s):
    if v == x: return True
    if isvar(x) and x in s: return occurs(v, s[x], s)
    if isinstance(x, tuple): return any(occurs(v, a, s) for a in x[1])
    return False

def unify(x, y, s=None):
    s = {} if s is None else s

    if s == "fail": return "fail"
    if x == y: return s

    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for a, b in zip(x, y):
            s = unify(a, b, s)
            if s == "fail": return "fail"
        return s

    if isvar(x): return unify_var(x, y, s)
    if isvar(y): return unify_var(y, x, s)

    if isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]): return "fail"
        return unify(x[1], y[1], s)
```

```
    return "fail"

def unify_var(v, x, s):
    if v in s: return unify(s[v], x, s)
    if isvar(x) and x in s: return unify(v, s[x], s)
    if occurs(v, x, s): return "fail"
    s[v] = x
    return s

def to_str(e):
    return e if isinstance(e, str) else f"{e[0]}({', '.join(to_str(a) for a in e[1])})"

def show(s):
    if s == "fail":
        print("Unification failed.")
    else:
        for v, x in s.items():
            print(f"{v} = {to_str(x)}")

expr1 = "P(f(x), g(y), y)"
expr2 = "P(f(g(z)), g(f(a)), f(a))"

X = parse(expr1)
Y = parse(expr2)

subs = unify(X, Y)
show(subs)
```

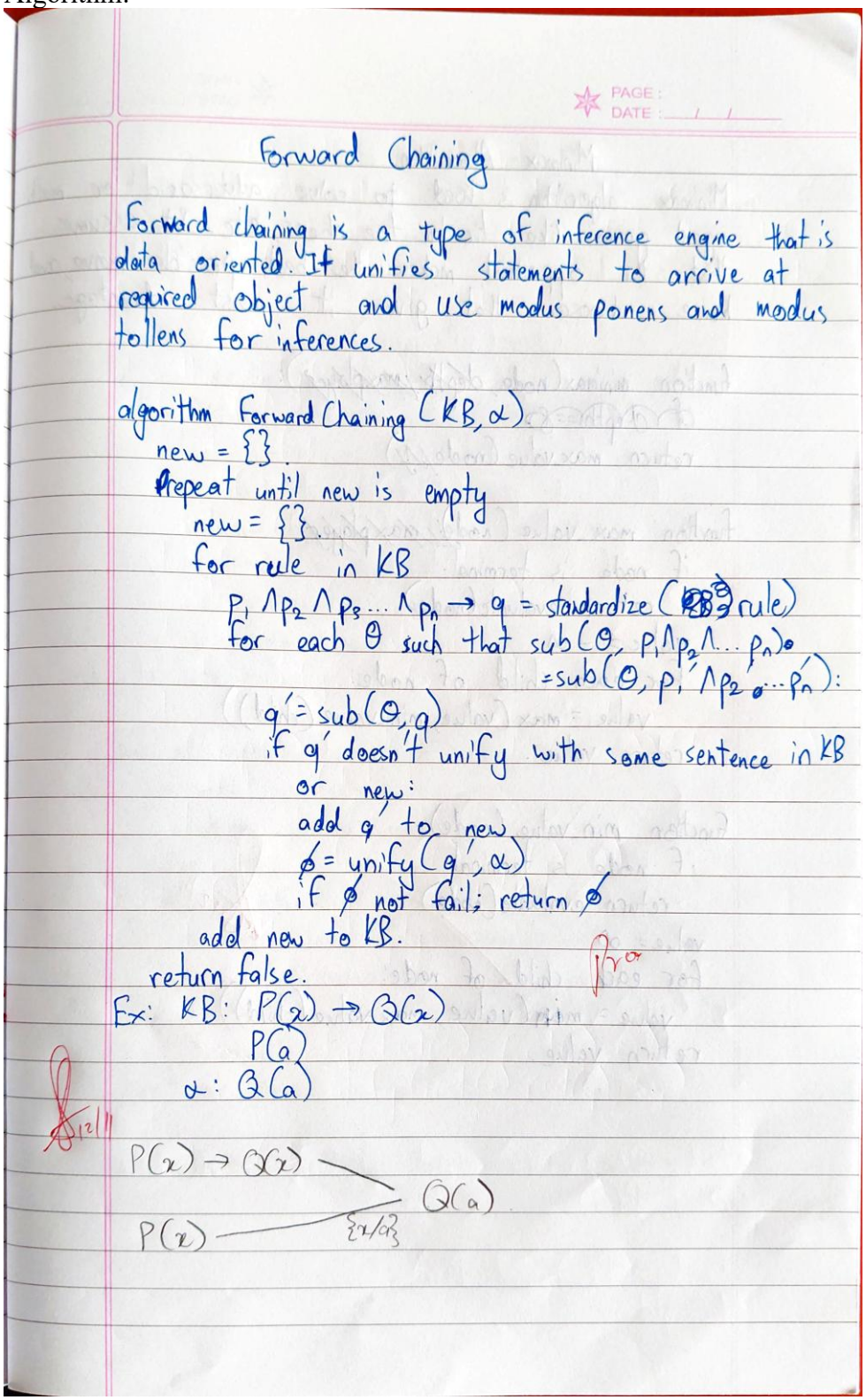
Output:

```
x = g(z)
y = f(a)
```

Program 8

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

Algorithm:



Forward Chaining

Forward chaining is a type of inference engine that is data oriented. It unifies statements to arrive at required object and use modus ponens and modus tollens for inferences.

algorithm Forward Chaining (KB, α)
  new = {}.
  Repeat until new is empty
    new = {}.
    for rule in KB
      $p_1 \wedge p_2 \wedge p_3 \ldots \wedge p_n \rightarrow q$ = standardize (rule)
      for each θ such that sub(θ, $p_1 \wedge p_2 \wedge \ldots p_n$) = sub(θ, $p_1' \wedge p_2' \ldots p_n'$):
        q' = sub(θ, q)
        if q' doesn't unify with some sentence in KB or new:
          add q' to new
          ϕ = unify(q', α)
          if ϕ not fail; return ϕ
    add new to KB.
  return false.

Ex: KB: $P(x) \rightarrow Q(x)$
    $P(a)$
    α: $Q(a)$

$P(x) \rightarrow Q(x)$
$P(x)$ — $\{x/a\}$ — $Q(a)$

Code:

```
import re

def is_variable(term):
    return term[0].islower()

def parse_predicate(expr):
    match = re.match(r"([A-Za-z_][A-Za-z0-9_]*)\(([^)]*)\)", expr)
    if match:
        pred, args = match.groups()
        args = [a.strip() for a in args.split(",")]
        return pred, args
    else:
        return expr, []

def unify(x, y, subs=None):
    if subs is None:
        subs = {}
    if x == y:
        return subs
    if isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            subs = unify(xi, yi, subs)
            if subs is None:
                return None
        return subs
    elif is_variable(x):
        return unify_var(x, y, subs)
    elif is_variable(y):
        return unify_var(y, x, subs)
    else:
        return None

def unify_var(var, x, subs):
    if var in subs:
        return unify(subs[var], x, subs)
    elif x in subs:
        return unify(var, subs[x], subs)
    else:
        new_subs = subs.copy()
        new_subs[var] = x
        return new_subs

def substitute(expr, subs):
    pred, args = parse_predicate(expr)
```

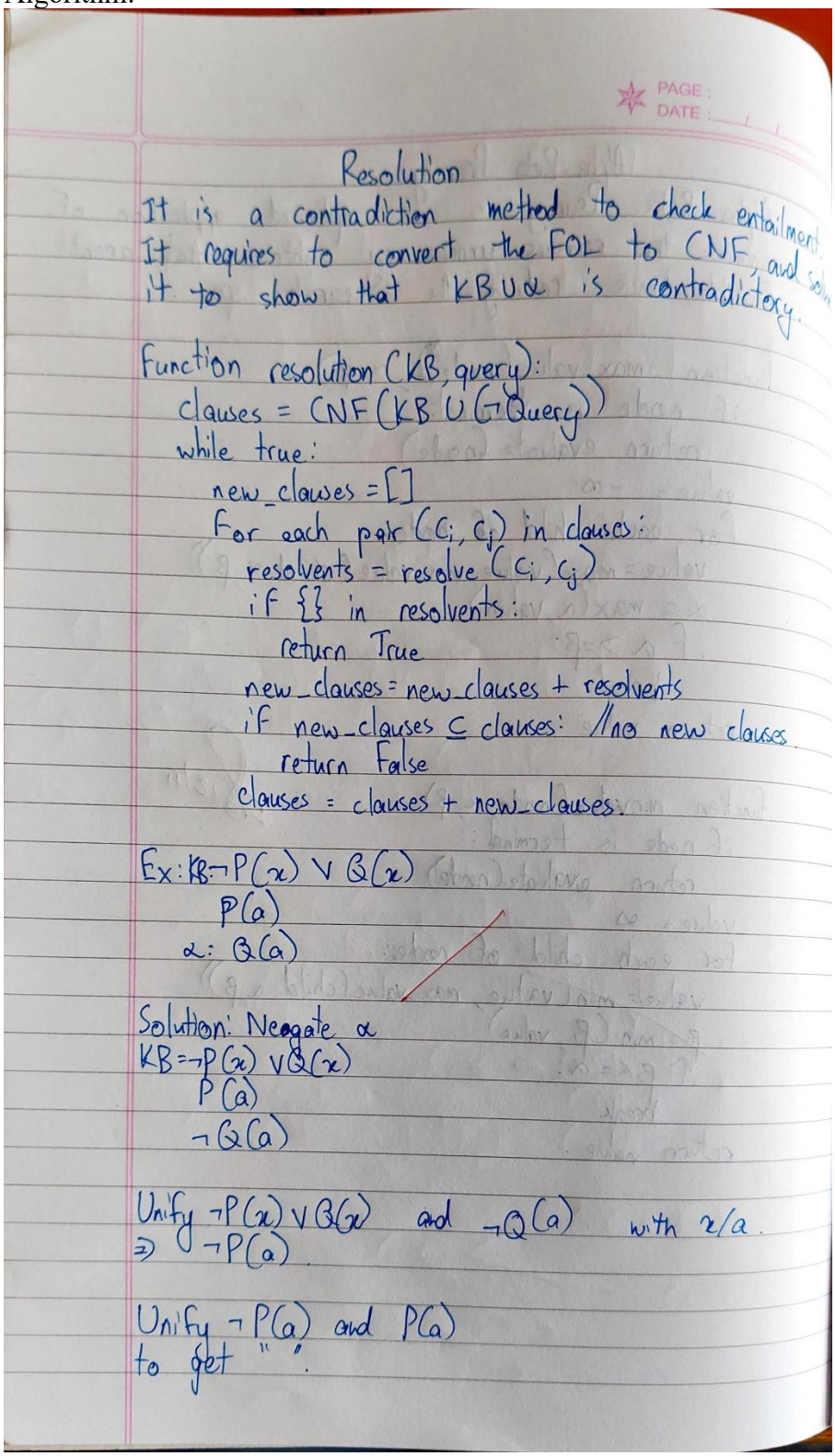
```
    new_args = [subs.get(a, a) for a in args]
    return f"{pred}({','.join(new_args)})"

def forward_chain(kb_rules, facts, query):
    inferred = set(facts)
    while True:
        new_inferred = set()
        for (premise, conclusion) in kb_rules:
            pred_p, args_p = parse_predicate(premise)
            pred_c, args_c = parse_predicate(conclusion)
            for fact in inferred:
                pred_f, args_f = parse_predicate(fact)
                if pred_p == pred_f:
                    subs = unify(args_p, args_f)
                    if subs is not None:
                        new_fact = substitute(conclusion, subs)
                        if new_fact not in inferred:
                            new_inferred.add(new_fact)
                            if new_fact == query:
                                return True
        if not new_inferred:
            return False
        inferred |= new_inferred

rules = [
    ("P(x)", "Q(x)")
]
facts = [
    "P(a)"
]
query = "Q(a)"

print("Entailed?", forward_chain(rules, facts, query))
```

Output:
Entailed? True

Program 9

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

Algorithm:



Resolution

It is a contradiction method to check entailment. It requires to convert the FOL to CNF, and then it to show that KB ∪ α is contradictory.

```
Function resolution(KB, query):
    clauses = CNF(KB ∪ ¬Query)
    while true:
        new_clauses = []
        For each pair (Ci, Cj) in clauses:
            resolvents = resolve(Ci, Cj)
            if {} in resolvents:
                return True
            new_clauses = new_clauses + resolvents
            if new_clauses ⊆ clauses: //no new clauses
                return False
            clauses = clauses + new_clauses.
```

Ex: KB: ¬P(x) ∨ Q(x)
P(a)
α: Q(a)

Solution: Negate α
KB: ¬P(x) ∨ Q(x)
P(a)
¬Q(a)

Unify ¬P(x) ∨ Q(x) and ¬Q(a) with x/a.
⇒ ¬P(a)

Unify ¬P(a) and P(a)
to get "".

PAGE :
DATE : / /

Due to blank, the KB is false. So, the new query is true.

12/11

Code:

```
import itertools
import re

def is_variable(term):
    return isinstance(term, str) and term[0].islower()

def parse_literal(literal):
    negated = literal.startswith("¬")
    if negated:
        literal = literal[1:]
    match = re.match(r"([A-Za-z_][A-Za-z0-9_]*)\(([^)]*)\)", literal)
    if match:
        pred, args = match.groups()
        args = [a.strip() for a in args.split(",")] if args.strip() else []
        return negated, pred, args
    else:
        return negated, literal, []

def unify(x, y, subs):
    if subs is None:
        return None
    elif x == y:
        return subs
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            subs = unify(xi, yi, subs)
            if subs is None:
                return None
        return subs
    elif is_variable(x):
        return unify_var(x, y, subs)
    elif is_variable(y):
        return unify_var(y, x, subs)
    else:
        return None

def unify_var(var, x, subs):
    if not isinstance(var, str):
        return None
    if var in subs:
        return unify(subs[var], x, subs)
    elif isinstance(x, str) and x in subs:
        return unify(var, subs[x], subs)
    elif occurs_check(var, x, subs):
```

```
        return None
    else:
        new_subs = subs.copy()
        new_subs[var] = x
        return new_subs

def occurs_check(var, x, subs):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, xi, subs) for xi in x)
    elif isinstance(x, str) and x in subs:
        return occurs_check(var, subs[x], subs)
    return False

def substitute(literal, subs):
    neg, pred, args = parse_literal(literal)
    new_args = [subs.get(a, a) for a in args]
    return ("¬" if neg else "") + f"{pred}({','.join(new_args)})" if args else literal

def resolve(ci, cj):
    resolvents = []
    for li in ci:
        for lj in cj:
            neg_i, pred_i, args_i = parse_literal(li)
            neg_j, pred_j, args_j = parse_literal(lj)
            if pred_i == pred_j and neg_i != neg_j:
                subs = unify(args_i, args_j, {})
                if subs is not None:
                    new_clause = set(substitute(l, subs) for l in (ci + cj))
                    new_clause.discard(substitute(li, subs))
                    new_clause.discard(substitute(lj, subs))
                    resolvents.append(tuple(sorted(new_clause)))
    return resolvents

def resolution(kb, query):
    clauses = kb + [(f"¬{query}",)]
    seen = set()
    while True:
        new = set()
        for (ci, cj) in itertools.combinations(clauses, 2):
            for resolvent in resolve(list(ci), list(cj)):
                if not resolvent:
                    return True
                new.add(resolvent)
        if new.issubset(set(clauses)) or new.issubset(seen):
            return False
```

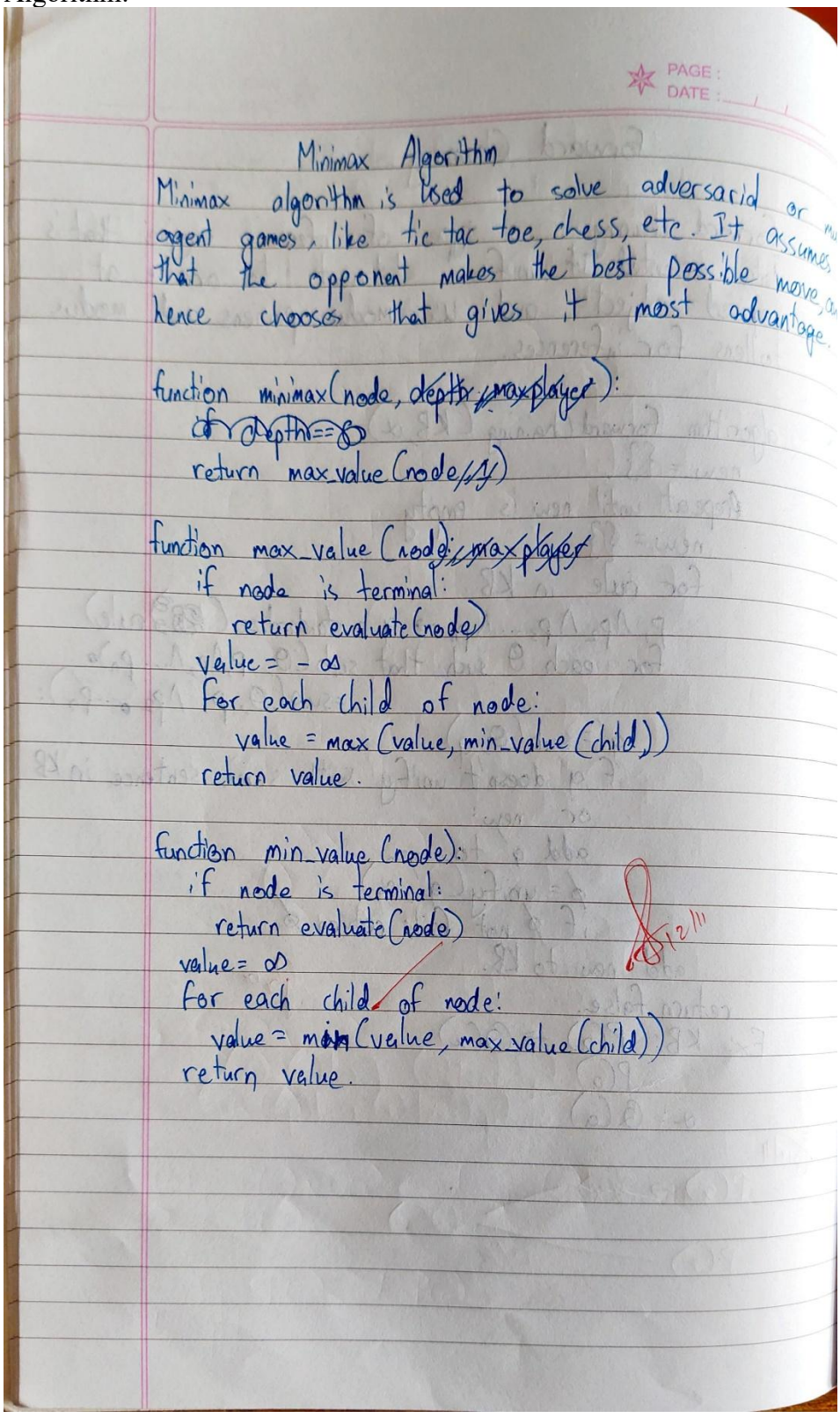
```
        clauses += list(new)
        seen |= new

KB = [
    ("¬P(x)", "Q(x)"),
    ("P(a)",)
]
query = "Q(a)"

print("Entailed?", resolution(KB, query))
```

Output:
Entailed? True

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Minimax Algorithm

Minimax algorithm is used to solve adversarial or multi agent games, like tic tac toe, chess, etc. It assumes that the opponent makes the best possible move, and hence chooses that gives it most advantage.

```
function minimax(node, depth, maxplayer):
    if depth==0
    return max_value(node, )

function max_value(node, maxplayer)
    if node is terminal:
        return evaluate(node)
    value = -∞
    for each child of node:
        value = max(value, min_value(child))
    return value.

function min_value(node):
    if node is terminal:
        return evaluate(node)
    value = ∞
    for each child of node:
        value = min(value, max_value(child))
    return value.
```

PAGE :
DATE : / /

Alpha-Beta Pruning

Alpha Beta Pruning is an optimised application of minimax that removes unnecessary subtrees that aren't feasible, hence reducing the search space.

```
function max_value(node, α, β)
  if node is terminal:
    return evaluate(node)
  value = -∞
  for each child of node:
    value = max(value, min_value(child, α, β))
    α = max(α, value)
    if α >= β:
      break
  return value

function minvalue(node, α, β)
  if node is terminal:
    return evaluate(node)
  value = ∞
  for each child of node:
    value = min(value, max_value(child, α, β))
    β = min(β, value)
    if β <= α:
      break
  return value.
```

12/11

Code:

```
import math

def print_board(board):
    print()
    for i in range(1, 10, 3):
        print(board[i], '|', board[i+1], '|', board[i+2])
    print()

def check_winner(board):
    win_positions = [
        (1,2,3), (4,5,6), (7,8,9),
        (1,4,7), (2,5,8), (3,6,9),
        (1,5,9), (3,5,7)
    ]
    for (x, y, z) in win_positions:
        if board[x] == board[y] == board[z] != ' ':
            return board[x]
    if all(board[i] != ' ' for i in range(1,10)):
        return 'Draw'
    return None

def alphabeta(board, depth, alpha, beta, is_maximizing):
    winner = check_winner(board)
    if winner == 'X':
        return 10 - depth
    elif winner == 'O':
        return depth - 10
    elif winner == 'Draw':
        return 0

    if is_maximizing:
        best_score = -math.inf
        for i in range(1, 10):
            if board[i] == ' ':
                board[i] = 'X'
                score = alphabeta(board, depth + 1, alpha, beta, False)
                board[i] = ' '
                best_score = max(best_score, score)
                alpha = max(alpha, score)
                if beta <= alpha:
                    break
        return best_score
    else:
        best_score = math.inf
        for i in range(1, 10):
            if board[i] == ' ':
```

```
            board[i] = 'O'
            score = alphabeta(board, depth + 1, alpha, beta, True)
            board[i] = ' '
            best_score = min(best_score, score)
            beta = min(beta, score)
            if beta <= alpha:
                break
        return best_score

def best_move(board):
    best_score = -math.inf
    move = -1
    for i in range(1, 10):
        if board[i] == ' ':
            board[i] = 'X'
            score = alphabeta(board, 0, -math.inf, math.inf, False)
            board[i] = ' '
            if score > best_score:
                best_score = score
                move = i
    return move

def play_game():
    board = [' '] * 10
    print("Tic Tac Toe — You (O) vs Computer (X)")
    print("Positions:")
    print("1 | 2 | 3\n4 | 5 | 6\n7 | 8 | 9")

    turn = input("\nDo you want to play first? (y/n): ").lower()
    human_first = (turn == 'y')

    while True:
        print_board(board)
        winner = check_winner(board)
        if winner:
            if winner == 'Draw':
                print("It's a draw!")
            else:
                print(f"{winner} wins!")
            break

        if (board.count(' ') % 2 == (0 if human_first else 1)):
            move = best_move(board)
            board[move] = 'X'
            print(f"Computer chooses position {move}")
        else:
            try:
```

```
            move = int(input("Enter your move (1-9): "))
            if 1 <= move <= 9 and board[move] == ' ':
                board[move] = 'O'
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Invalid input. Try again.")

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

Output:

```
Tic Tac Toe — You (O) vs Computer (X)
Positions:
1 | 2 | 3
4 | 5 | 6
7 | 8 | 9

Do you want to play first? (y/n): y

  |   |
  |   |
  |   |

Enter your move (1-9): 1

O |   |
  |   |
  |   |

Computer chooses position 5

O |   |
  | X |
  |   |

Enter your move (1-9): 9

O |   |
  | X |
  |   | O

Computer chooses position 2

O | X |
  | X |
  |   | O
```

```
Enter your move (1-9): 8

O | X |
  | X |
  | O | O

Computer chooses position 7

O | X |
  | X |
X | O | O

Enter your move (1-9): 3

O | X | O
  | X |
X | O | O

Computer chooses position 6

O | X | O
  | X | X
X | O | O

Enter your move (1-9): 4

O | X | O
O | X | X
X | O | O

It's a draw!
```