Artificial Intelligence Lab Report

Submitted by

Ayush Aditya(1BM23CS057)
Batch: A3

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

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

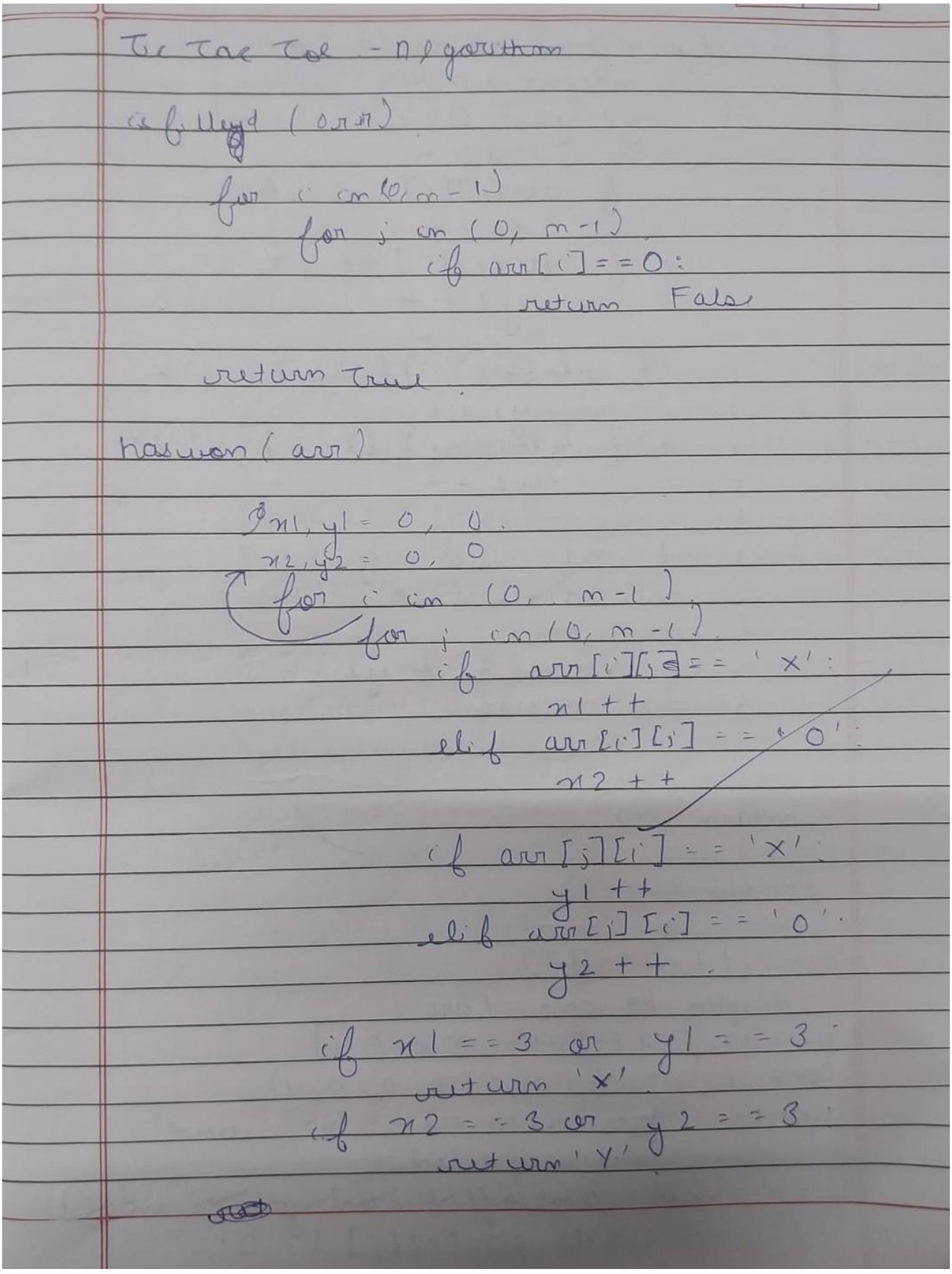
B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2025-2026

Table of contents

Program Number	Program Title	Page Number
1	Tic-Tac-Toe	3-10
2	Misplaced tiles – Manhattan distance	11-15
3	8-Puzzle IDDFS	16-20
4	8-Puzzle A*	21-27
5	Vacuum Cleaner	28-32
6	4 queens using hill climbing	33-36
7	4 queens using simulated annealing	37-41
8	Knowledge Base - Propositional Logic	42-47
9	Unification in First Order Logic	48-55
10	Resolution Program	56-60
11	Forward Chaining inference	61-65
12	Alpha Beta Pruning	66-70

Program 1 - Tic Tac toe

Algorithm



```
for i in range(0, n-1):
    x1, x2 = 0, 0
    y1, y2 = 0, 0
    if arr[i][i] == 'x':
        x1 ++
    elif arr[i][i] == 'y':
        y1 ++
    if arr[2-i][2-i] == 'x':
        y1 ++
    elif arr[2-i][2-i] == 'y':
        y2 ++
    if x1 == 3 or y1 == 3:
        return 'x'
    if x2 == 3 or y2 == 3:
        return 'y'
return 'No'
```

~~for (of ()~~

```
arr = [[0]*3]*3
display board(arr)
Im = Input("Enter X or O")
Comp i, j, i1, j1 = 0, 0, 0, 0
while haswon(arr) == No and
    1 is filled:
    x, y = input("Enter your position")
    while arr[i][j] != 0:
```

DATE :

```
j = Input("Enter y axis")
arr[i][j] = Im
if Im == 'x':
comp = ""
if Im == 'x':
    comp = 'O'
else
    comp = 'x'
while arr[i][j] != 0 and !isfilled(arr):
    i = random.randint(0, 2)
    j = randomrandint(0, 2)

    arr[i][j] = comp
    show(arr)
if hasWon(arr) == Im:
    display("Player has won")

if hasWon(arr) == comp:
    display("Player has lost")

if hasWon(arr) == 'No' and isfilled(arr):
    display("It was a draw")

show(arr)
for item in arr:
    for item1 in item
        if item1 == 0:
            print(-)
```

Pseudocode :

Vacuum cleaner

else

print (item1)

output :

O O O — — —
O O O — — —
O O O — — —

Enter x or O : XO

Enter x axis : O
Enter y axis : O

O — —
— — —
X — —

Enter x axis : O
Enter y axis : 1

O O —
— X —
X — —

Enter x axis : O
Enter y axis : 2

O O O
— X —
X X —

Tic Tac Toe output:

Enter X or O here : X

_ _ _
_ _ _
_ _ _

Enter x axis : 0
" y " : 0

X _ 0
_ _ _
_ _ _

Enter x axis : 1
Enter y " : 1

X _ 0
_ X _
0 _ _

Enter x axis : 2
Enter y axis : 2

X _ 0
_ X _
0 _ X

Player won.

Code

```
import random


def filled(arr):
    for i in range(len(arr)):
        for j in range(len(arr)):
            if arr[i][j] == 0:
                return False
    return True


def haswon(arr):
    # rows and cols
    for i in range(3):
        if arr[i][0] == arr[i][1] == arr[i][2] != 0:  # row
            return arr[i][0]
        if arr[0][i] == arr[1][i] == arr[2][i] != 0:  # col
            return arr[0][i]

    # diagonals
    if arr[0][0] == arr[1][1] == arr[2][2] != 0:
        return arr[0][0]
    if arr[0][2] == arr[1][1] == arr[2][0] != 0:
        return arr[0][2]

    return 'No'
def show(arr):
    for row in arr:
        for item in row:
            if item == 0:
                print('_', end=" ")
            else:
                print(item, end=" ")
        print()
```

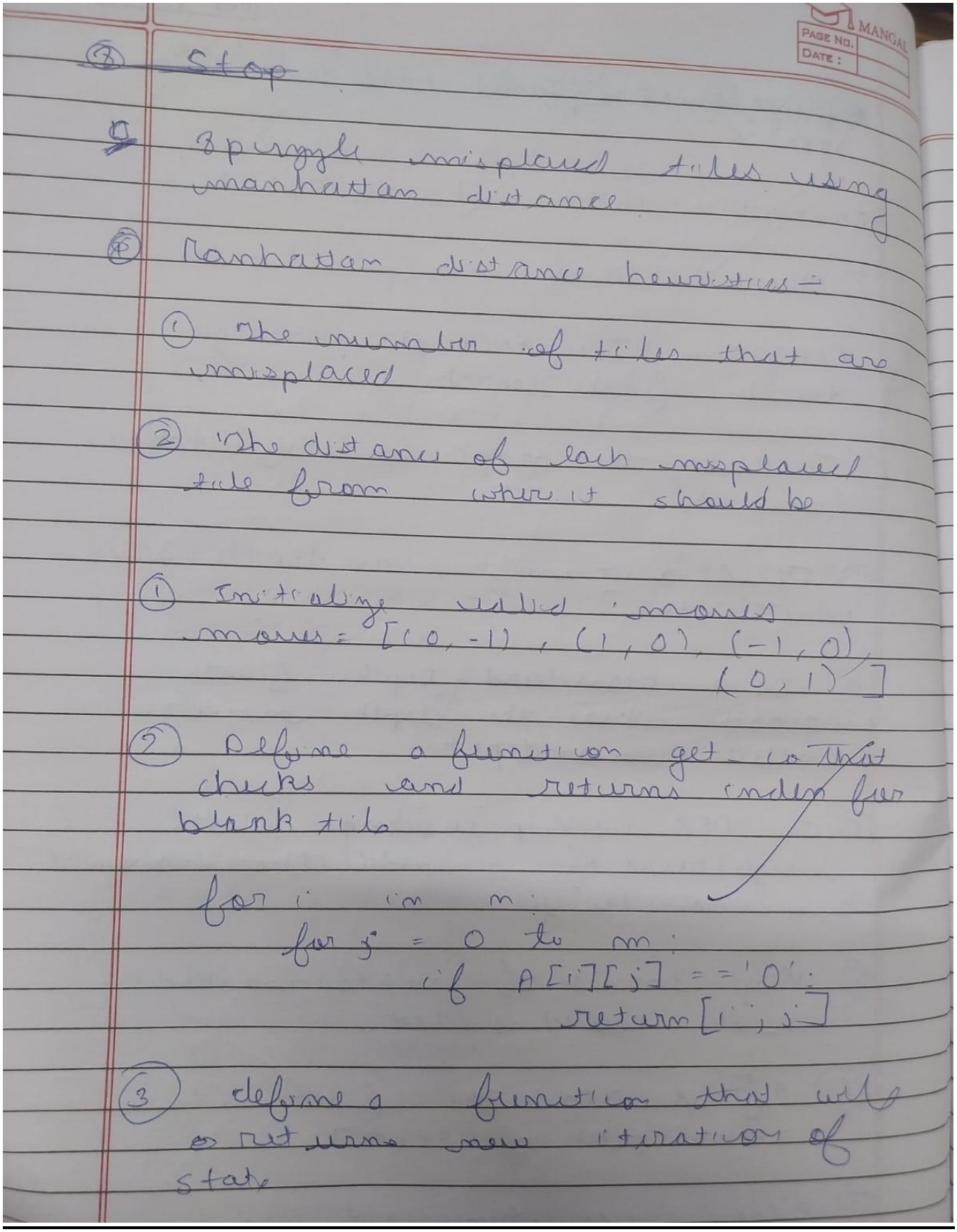
```
    print()
arra = [[0 for _ in range(3)] for _ in range(3)]
In = input("Enter X or 0 here : ").upper()
comp = '0' if In == 'X' else 'X'
show(arra)
while haswon(arra) == 'No' and not filled(arra):
    # Player move
    i = int(input("Enter x axis (0-2): "))
    j = int(input("Enter y axis (0-2): "))
    while arra[i][j] != 0:
        print("Cell already taken, try again.")
        i = int(input("Enter x axis (0-2): "))
        j = int(input("Enter y axis (0-2): "))

    arra[i][j] = In
    if haswon(arra) != 'No' or filled(arra):
        break
    i1, j1 = random.randint(0, 2), random.randint(0, 2)
    while arra[i1][j1] != 0:
        i1, j1 = random.randint(0, 2), random.randint(0, 2)
    arra[i1][j1] = comp

    show(arra)
winner = haswon(arra)
show(arra)
if winner == In:
    print("Player won ")
elif winner == comp:
    print("Player lost ")
else:
    print("It was a draw ")
```

Output Snapshot

```
Enter X or 0 here : X
_ _ _
_ _ _
_ _ _

Enter x axis (0-2): 0
Enter y axis (0-2): 0
X _ 0
_ _ _
_ _ _

Enter x axis (0-2): 1
Enter y axis (0-2): 1
X _ 0
_ X _
0 _ _

Enter x axis (0-2): 2
Enter y axis (0-2): 2
X _ 0
_ X _
0 _ X

Player won

Process finished with exit code 0
```

Program 2 – Misplaced Tiles Manhattan Distance

Algorithm



8. Stop

5. Sprygle misplaced tiles using manhattan distance

6. Manhattan distance heuristics =

① The number of tiles that are misplaced

② The distance of each misplaced tile from where it should be

① Initialize valid moves

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

② Define a function get_ which checks and returns index for blank tile

for i in n:
    for j = 0 to m:
        if A[i][j] == '0':
            return [i, j]

③ define a function that will return new iteration of state

MANGAL
PAGE NO.
DATE :

(4) new_state (state):
for all dx, dy
dx, dy ← moves
nx, ny = x+dx, y+dy
state[nx], state[ny] = state[ny], state[nx]

(4) Use manhattan distance to find out nearness of the current state to the final state

(5) If the final state is reached return path + [state]

(6) Else if man

(6) Do IDDFS iteration of new_states and new states to visited and do new_state operation on them

(7) If Define manhattan function as:

manhattan (state):

val = state[i][j] i<3, j<3
i = (val-1)//3
jq = (val-1) % 3
dist = (i - i) + (j - j)
return dist

(9) use manhattan distance to figure out nearness of current state to final state

g = g

(8) store states and f and g value in priority heap to figure out the nearest distance state

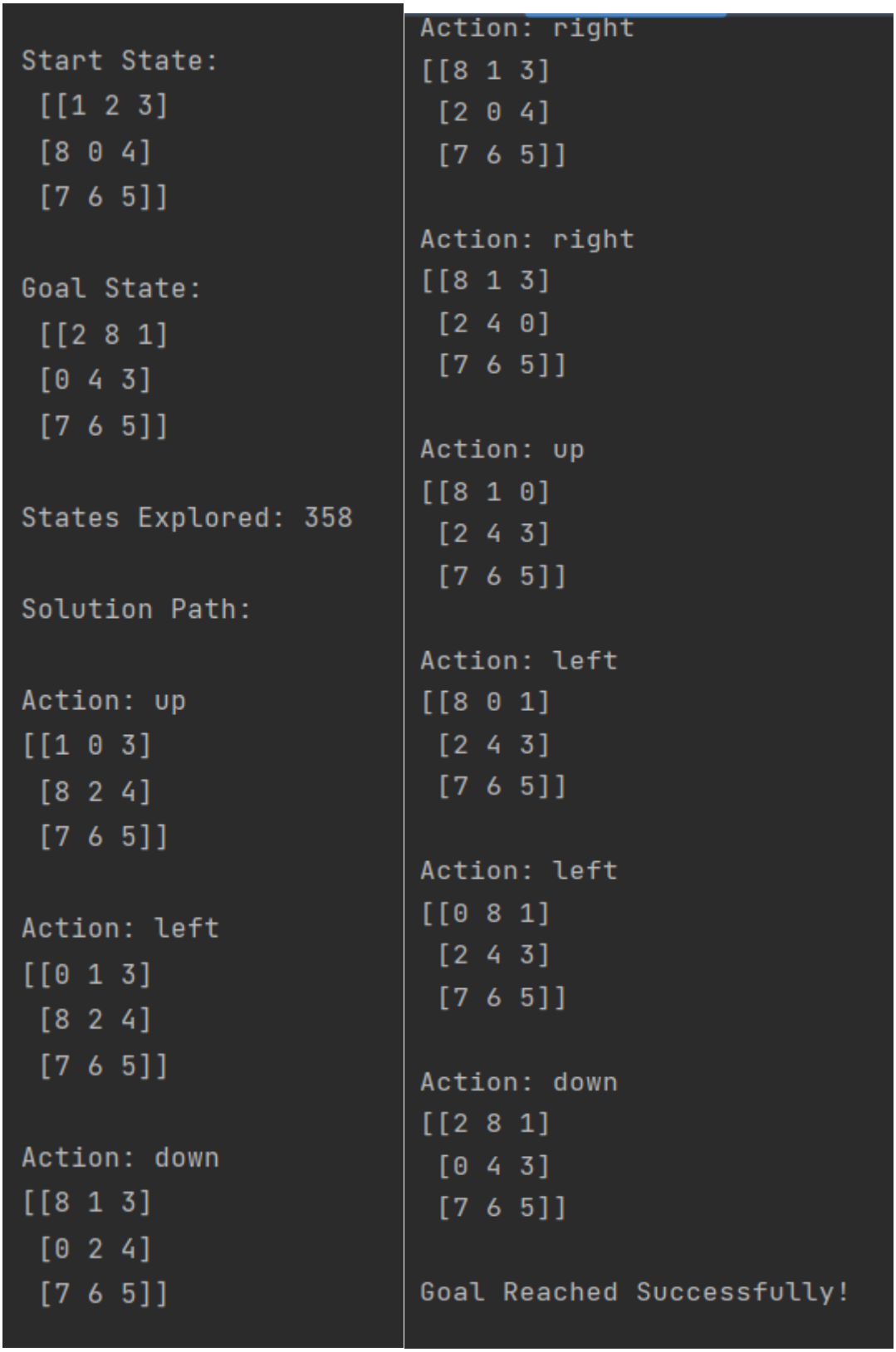
heap = [(f, g, state)]

(10) Continue until you reach ~~reach~~ max depth or final state is reached.

Code

```
def manhattan(state):
    dist = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                target_x = (val-1) // 3
                target_y = (val-1) % 3
                dist += abs(i - target_x) + abs(j - target_y)
    return dist

def ida_star(start):
    def search(path, g, bound):
        state = path[-1]
        f = g + manhattan(state)
        if f > bound:
            return f
        if is_goal(state):
            return "FOUND"
        min_threshold = float("inf")
        for neighbor in get_neighbors(state):
            if neighbor not in path:
                path.append(neighbor)
                result = search(path, g+1, bound)
                if result == "FOUND":
                    return "FOUND"
                if result < min_threshold:
                    min_threshold = result
                path.pop()
        return min_threshold
    bound = manhattan(start)
    path = [start]
    while True:
        result = search(path, 0, bound)
        if result == "FOUND":
            return path
        if result == float("inf"):
            return None
        bound = result
```

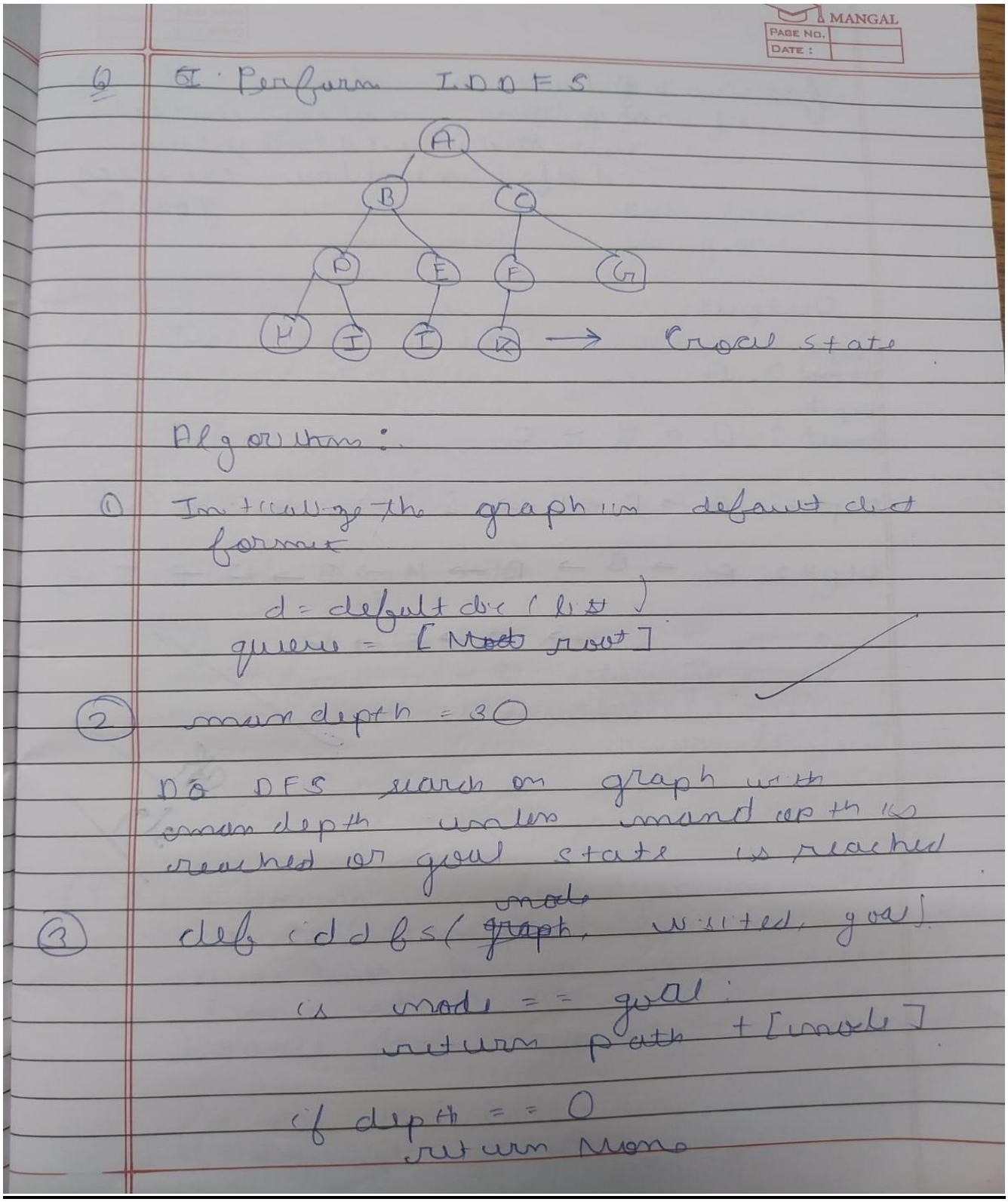
Output Snapshot



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

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

States Explored: 358

Solution Path:

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

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

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

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

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

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

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

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

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

Goal Reached Successfully!
```

Program 3 - 8 puzzle using IDDFS

Algorithm



for neighbour in d:
if neighbour not in visited
visited.add(neighbour)
id dfs(neighbour, visited, goal)

Output:

~~Depth~~ 0: A

Depth
~~level~~ 1: A → B → C

Depth 2: A → B → D → E → C → F → G

Depth 3: A → B → D → H → I → E → I
→ C → F → K

3/9

Code

```
import copy
inp=[[1,2,3],[4,-1,5],[6,7,8]]
out=[[1,2,3],[6,4,5],[-1,7,8]]
def move(temp, movement):
  if movement=="up":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=0:
            temp[i][j]=temp[i-1][j]
            temp[i-1][j]=-1
          return temp
  if movement=="down":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if i!=2:
            temp[i][j]=temp[i+1][j]
            temp[i+1][j]=-1
          return temp
  if movement=="left":
    for i in range(3):



      for j in range(3):
        if(temp[i][j]==-1):
          if j!=0:
            temp[i][j]=temp[i][j-1]
            temp[i][j-1]=-1
          return temp
  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=2:
            temp[i][j]=temp[i][j+1]
            temp[i][j+1]=-1
          return temp
def ids():
  global inp
```



```
  global out
```

```
  global flag
  for limit in range(100):
    print('LIMIT -> '+str(limit))
    stack=[]
    inpx=[inp,"none"]
    stack.append(inpx)
    level=0
    while(True):
      if len(stack)==0:
        break
      puzzle=stack.pop(0)
      if level<=limit:
        print(str(puzzle[1])+" --> "+str(puzzle[0]))
        if(puzzle[0]==out):
          print("Found")
          print('Path cost='+str(level))
          flag=True
          return
        else:
          level=level+1
          if(puzzle[1]!="down"):
            temp=copy.deepcopy(puzzle[0])
            up=move(temp, "up")



            if(up!=puzzle[0]):
              upx=[up,"up"]
              stack.insert(0, upx)
          if(puzzle[1]!="right"):
            temp=copy.deepcopy(puzzle[0])
            left=move(temp, "left")
            if(left!=puzzle[0]):
              leftx=[left,"left"]
              stack.insert(0, leftx)
          if(puzzle[1]!="up"):
            temp=copy.deepcopy(puzzle[0])
            down=move(temp, "down")
            if(down!=puzzle[0]):
              downx=[down,"down"]
              stack.insert(0, downx)



if(puzzle[1]!="left"):
temp=copy.deepcopy(puzzle[0])
```

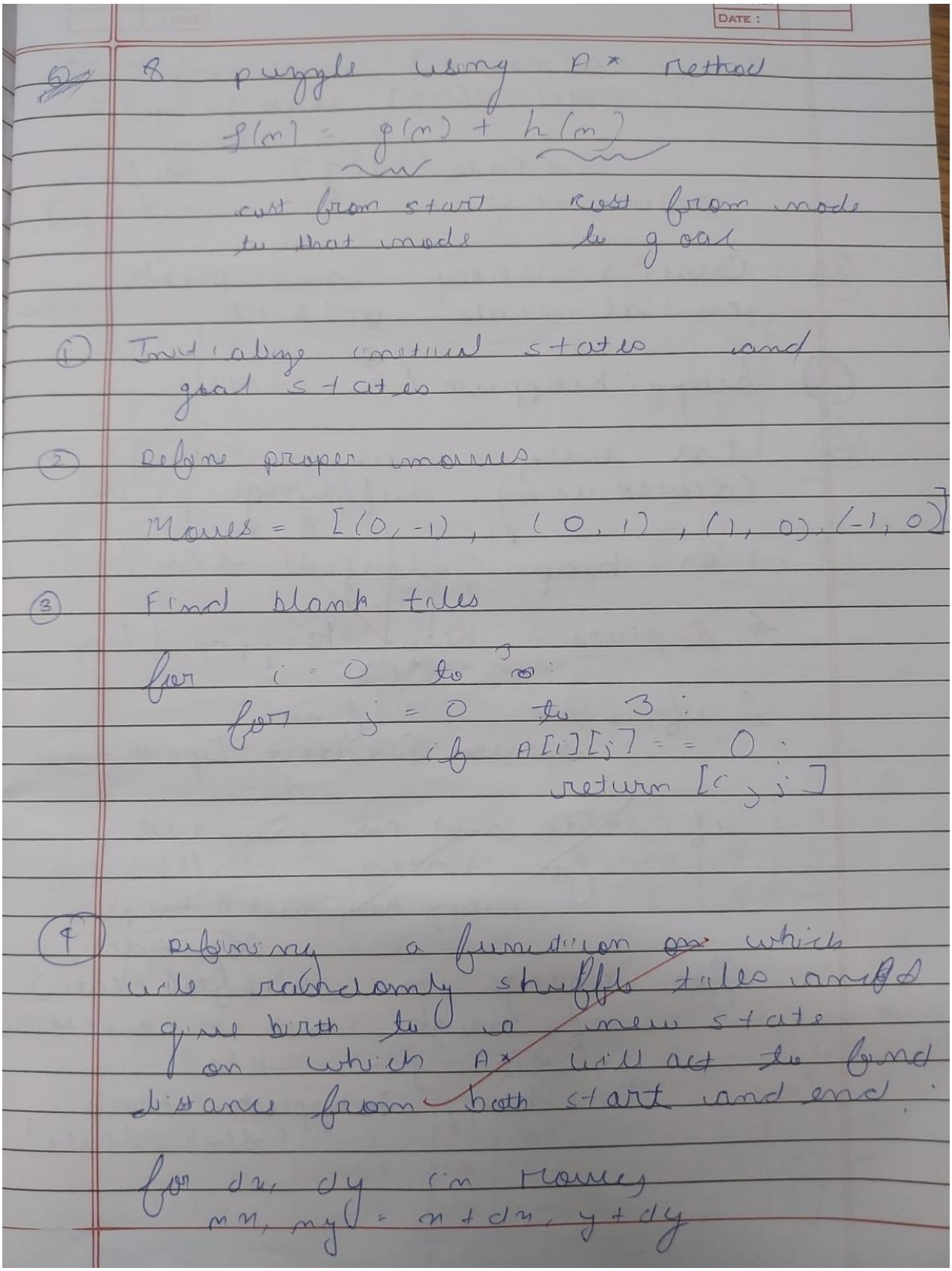
```
right=move(temp, "right")
if(right!=puzzle[0]):
rightx=[right,"right"]
stack.insert(0, rightx)
print('~~~~~~~~~~~~~ IDS ~~~~~~~~~~~~~')
ids()
```

Output Snapshot

```
~~~~~~ ITERATIVE DEEPENING SEARCH ~~~~~~

Trying depth limit: 1

Trying depth limit: 2

Goal Found at depth 2

Solution Path:
[1, 2, 3]
[4, -1, 5]
[6, 7, 8]

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

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

Program 04 - 8 Puzzle Using A*

Algorithm:



8 puzzle using A* method

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

cost from start to that node — cost from node to goal

1. Initialize initial states and goal states
2. Define proper moves

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

3. Find blank tiles

```
for i = 0 to 3:
    for j = 0 to 3:
        if A[i][j] == 0:
            return [i, j]
```

4. Defining a function which will randomly shuffle tiles and give birth to a new state on which A* will act to find distance from both start and end.

```
for dx, dy in Moves
    nx, ny = x + dx, y + dy
```

if $0 <= nx < 3$ and $0 <= ny < 3$:
state[nx][ry], state[x,y] =
state[x][y], state[nx][ny].
return state

(5) Create a heap and push initial state onto it.

(6) heapq.heappush(h, state)

(7) For every new state created using shuffling push it on heap alongside f(n)

f, state = heapq.heappop(heap)

if state == "final"
return (f, state + [path])

if state not in visited:
for state in shuffle(state):
heapq.heappush(heap, (f, state))
for neigh in shuffle(state):
if neigh not in visited:
f(n) = g(n) + h(n)
heapq.heappush(heap, (f(n), state))

MANGAL
PAGE NO.
DATE :

(7) When the final state is reached we have accomplished our goal.

Output :

Initial state

2	1	7
3	4	5
6	—	8

Final State

1	2	3
4	5	6
7	8	—

g = 6, h = 9

f = 9

2	1	7
3	4	5
6	—	8

h = 8

g = 2

f = 10

2	—	7
3	1	5
6	4	8

h = 7

g = 1,

f = 8

2	1	7
3	4	5
6	8	—

h = 6

g = 3

f = 9

2	1	7
3	5	—
6	4	8

f = 9

g = 5

h = 6

2	1	5
3	4	6
7	8	—

1	2	3
4	5	6
7	8	—

1	2	5
4	3	6
7	8	—

Final state
Reached

Code

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



```
   b = state.index(-1)
   d = []
```

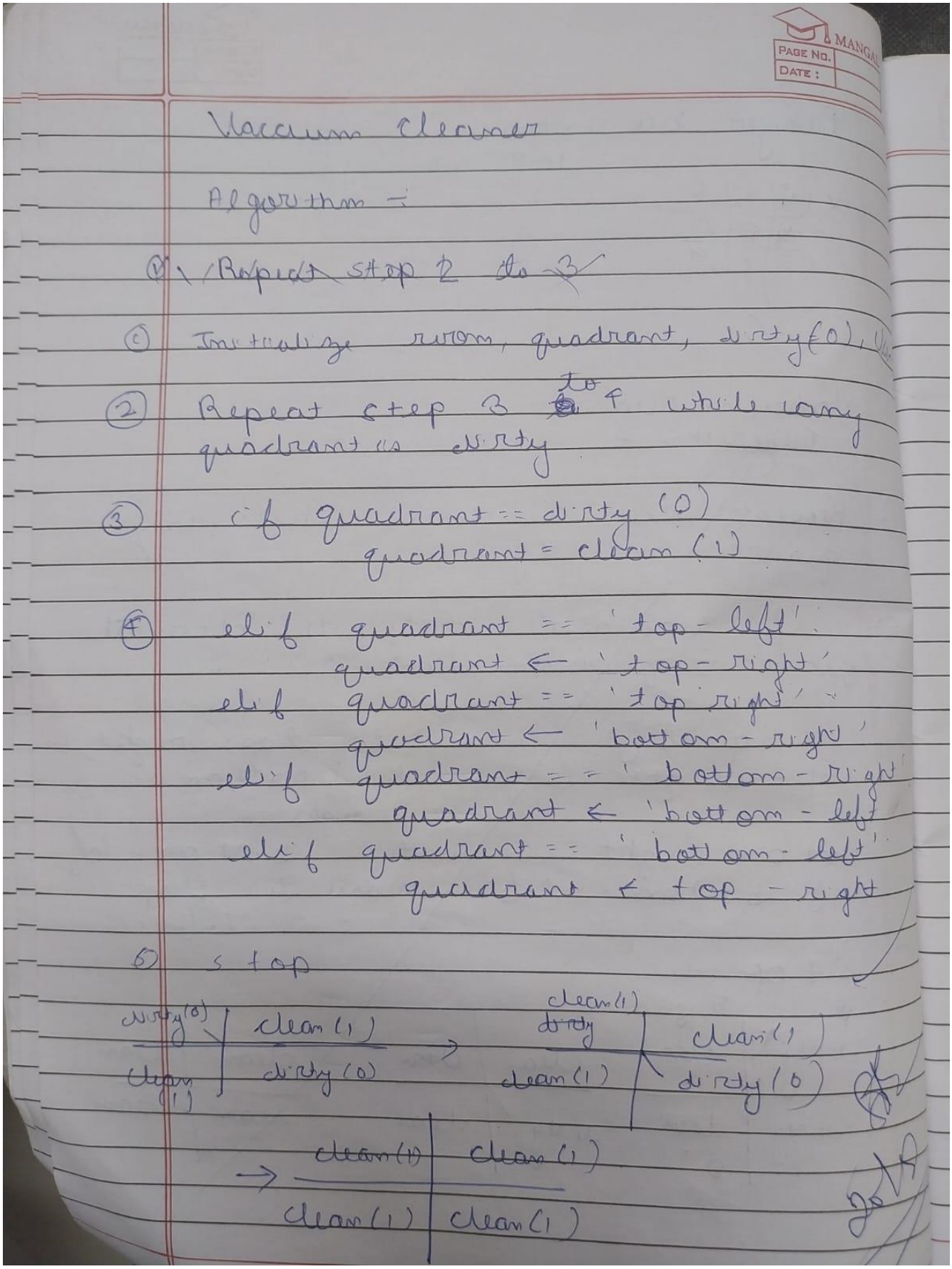
```
    if b - 3 in range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move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

Output Snapshot:

```
~~~~~~ ITERATIVE DEEPENING SEARCH ~~~~~~

Trying depth limit: 1

Trying depth limit: 2

Goal Found at depth 2

Solution Path:
[1, 2, 3]
[4, -1, 5]
[6, 7, 8]

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

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

Program 5 - Vacuum Cleaner

Algorithm



Vacuum cleaner

output :

Quadrant 1 has been cleaned.

current state of the room

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Quadrant 2 has been cleaned

Current state of the room

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Quadrant 3 has been cleaned

Current state of the room

$$\begin{bmatrix} 1 & 1 \\ 0 & 0 \end{bmatrix}$$

Quadrant 4 has been cleaned

current state of the room

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Final Room State :

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

All quadrants are clean !

Code

```
def vacuum_world():

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

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



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



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

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

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

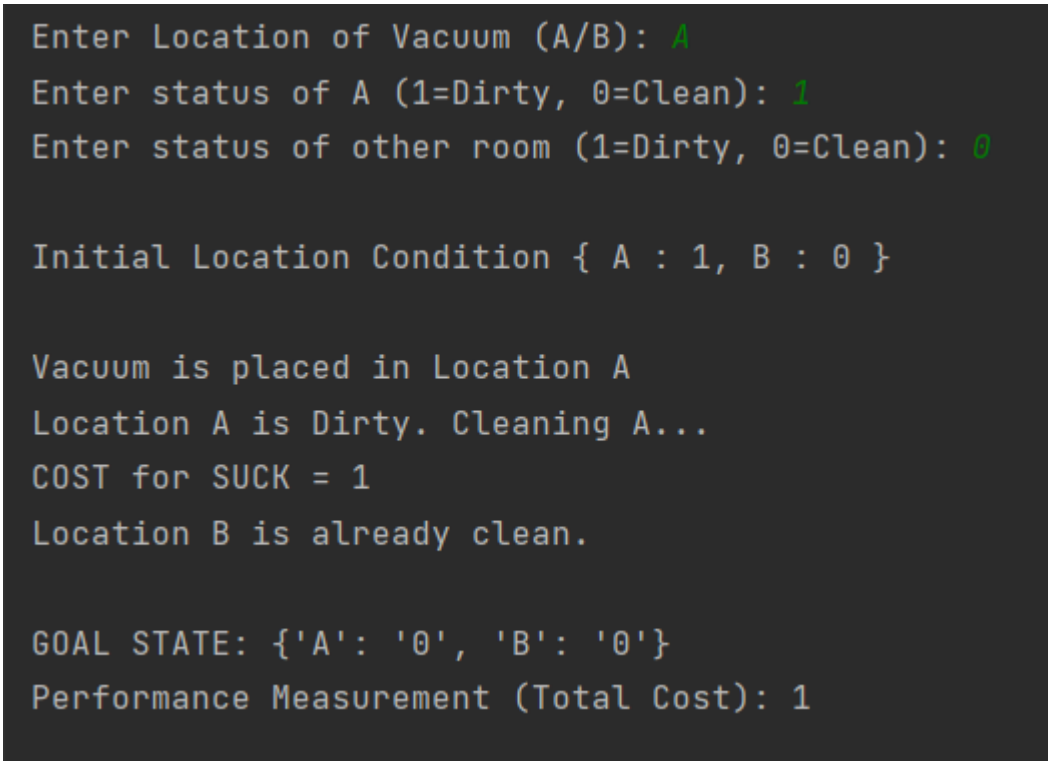
```
            cost += 1
            print("COST for moving RIGHT " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
      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

else:
```



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



        print("Location B has been Cleaned.")

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

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

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

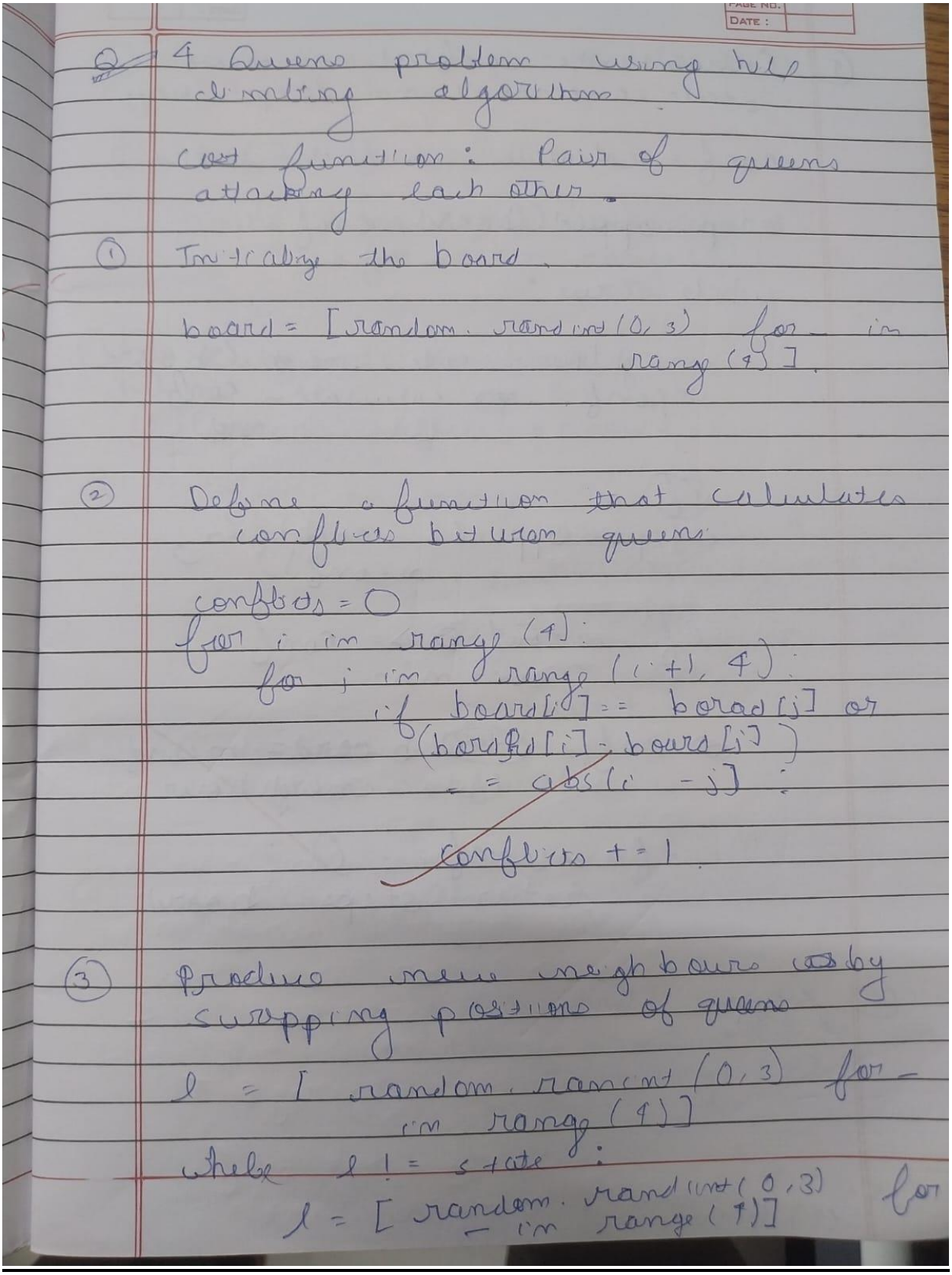
```
            cost += 1
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print("Location A is already clean.")

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

Output Snapshot



Program 6 queens using hill climbing

Algorithm



4 Queens problem using hill climbing algorithm.

cost function: Pair of queens attacking each other.

① Initializing the board.

board = [random.randint(0, 3) for _ in range(4)]

② Define a function that calculates conflicts between queens

conflicts = 0
for i in range(4):
    for j in range(i+1, 4):
        if board[i] == board[j] or
        (board[i] - board[j]) == abs(i - j):
            conflicts += 1

③ Produce new neighbours by swapping positions of queens

l = [random.randint(0, 3) for _ in range(4)]
while l != state:
    l = [random.randint(0, 3) for _ in range(4)]

(4) Calculate the conflicts in new state and add it in steps array

conf = calculate_conflicts(board)

steps.append((board, conf))

while True:

neighbour = calc_neigh(board)

nconf = calculate_conflicts(board)

if

steps.append((neighbour, nconf))

if nconf >= conf

continue

conf, board = nconf, neighbour

if nconf == 0:

return steps, board

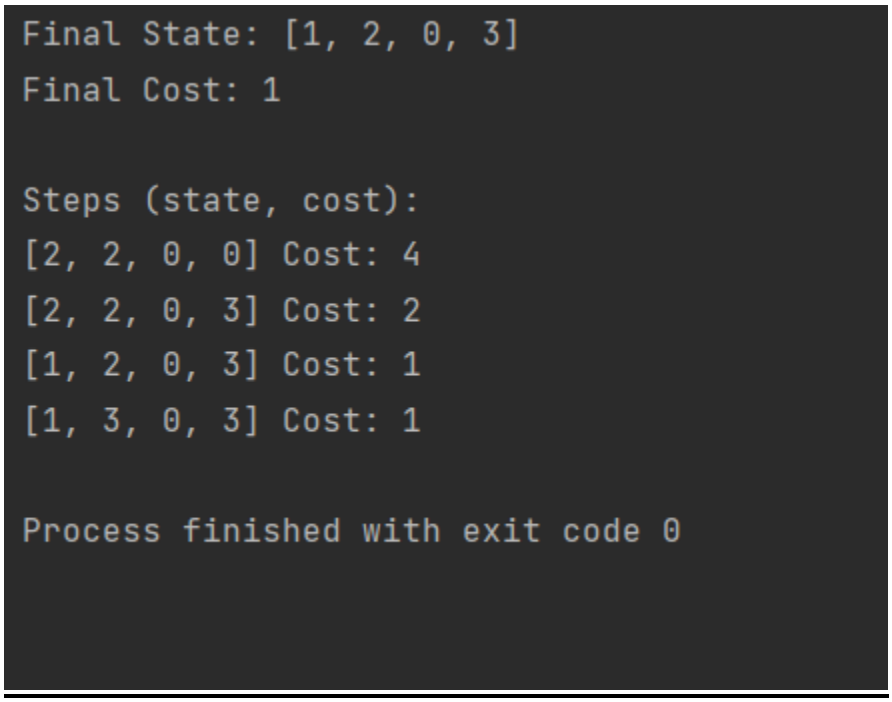
(5) Stop

Code

```
import random

def calculate_conflicts(state):
    """Heuristic: number of pairs of queens attacking each other."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def generate_neighbors(state):
    """Generate all neighbor states by moving one queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(n=4):
    # Initial random state
    state = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_conflicts(state)

    steps = []
    steps.append((state.copy(), current_cost))

    while True:
        neighbors = generate_neighbors(state)
        best_neighbor = min(neighbors, key=calculate_conflicts)
        best_cost = calculate_conflicts(best_neighbor)

        steps.append((best_neighbor.copy(), best_cost))
```

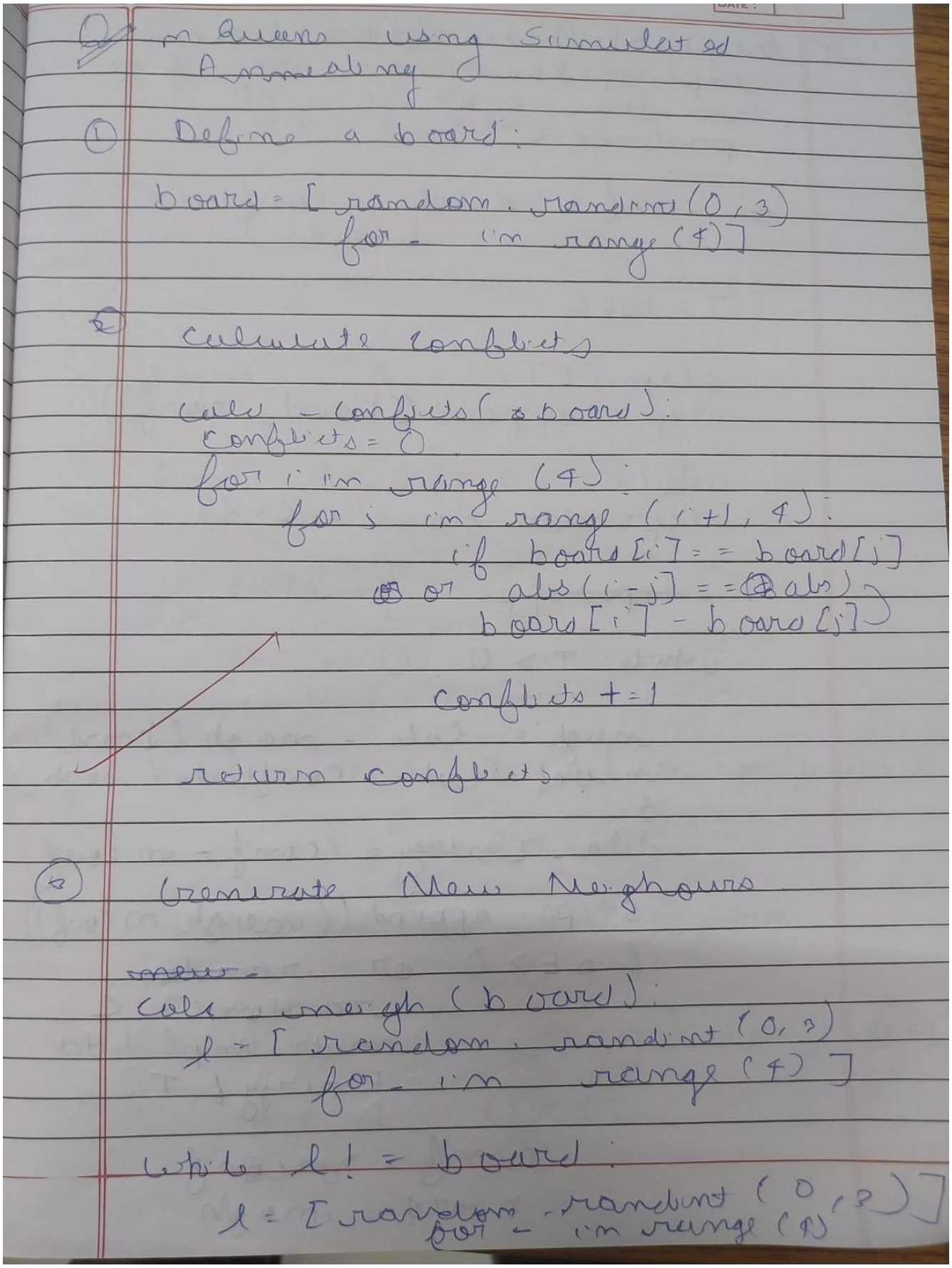
```
        if best_cost >= current_cost:  # Local optimum
            break
        state, current_cost = best_neighbor, best_cost

    return state, current_cost, steps

# Run for 4-Queens
solution, cost, steps = hill_climbing(4)

print("Final State:", solution)
print("Final Cost:", cost)
print("\nSteps (state, cost):")
for s, c in steps:
    print(s, "Cost:", c)
```

Output Snapshot



Program-07 4 queens using simulated annealing

Algorithm



n Queens using Simulated Annealing

① Define a board:

board = [random.randint(0, 3) for _ in range(4)]

② Calculate conflicts

calc_conflicts(board):
conflicts = 0
for i in range(4):
for j in range(i+1, 4):
if board[i] == board[j] or abs(i-j) == abs(board[i] - board[j]):
conflicts += 1
return conflicts

③ Generate New Neighbours

new_
calc_neigh(board):
l = [random.randint(0, 3) for _ in range(4)]
while l != board:
l = [random.randint(0, 3) for _ in range(4)]

(f) Generate new states and replace existing state if it has less $\Delta E > 0$ or prob $< e^{\Delta E/T}$

conf = calc_conflicts(boards)

T = 1000

steps = []
steps.append((board, conf))

step_down = 0.99

while T > 0

neigh = calc_neigh(board)
n_conf = calc_conflicts(neigh)
delta_Energy = (conf - n_conf)
steps.append((neigh, n_conf))
if $\Delta E > 0$ or random.random() < math.exp(delta_Energy / T)

conf = n_conf
board = neigh

if rconf == 0:
return steps, board

T* = step - down.

Output -

board = [0, 1, 2, 3],
conflicts = 6.

board = [0, 2, 1, 3],
conflicts = 1

~~delta~~ = ΔE = 6 - 1 = 5

ΔE > 0.
conflicts = 1

board = [0, 1, 3, 1] &, conflicts = 3

ΔE = 1 - 3 = -2

ΔE < 0 ⇒ No swap

board = [0, 2, 4, 1], conflicts = 0

ΔE = 1 - 0 = 0
& rconf = 0.

Code

```
import random
import math

# Define the size of the board (4x4)
N = 4


# Generate a random state (a random arrangement of queens on the board)
def random_state():
    return [random.randint(0, N - 1) for _ in range(N)]


# Calculate the number of conflicts (attacking pairs of queens)
def calculate_conflicts(state):
    conflicts = 0
    for i in range(N):
        for j in range(i + 1, N):
            if state[i] == state[j]:  # Same column
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):  # Same diagonal
                conflicts += 1
    return conflicts


# Perform a random move (a small change to the state)
def random_move(state):
    new_state = state[:]
    row = random.randint(0, N - 1)
    new_state[row] = random.randint(0, N - 1)
    return new_state


# Simulated Annealing algorithm
def simulated_annealing():
    current_state = random_state()
    current_temp = 1000  # Initial temperature
    min_temp = 1  # Minimum temperature to stop
    cooling_rate = 0.95  # Rate at which the temperature decreases

    # Loop until the temperature is low enough
    while current_temp > min_temp:
        # Calculate the current number of conflicts
        current_conflicts = calculate_conflicts(current_state)
```

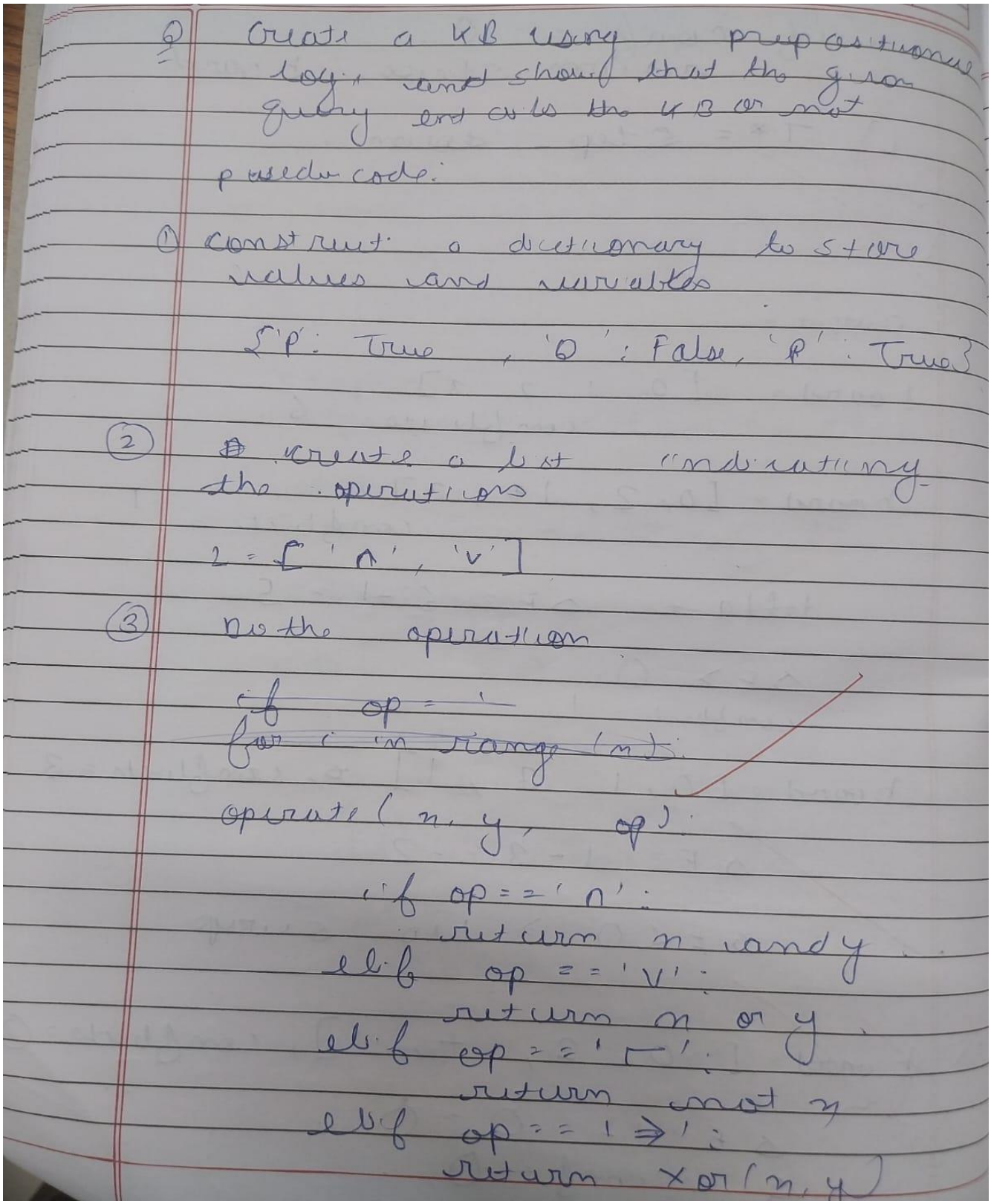
```
        # If no conflicts, we've found a solution
        if current_conflicts == 0:
            return current_state

        # Generate a new candidate by making a random move
        new_state = random_move(current_state)
        new_conflicts = calculate_conflicts(new_state)

        # If the new state is better or if we accept it probabilistically
        if new_conflicts < current_conflicts or random.random() < math.exp(
                (current_conflicts - new_conflicts) / current_temp):
            current_state = new_state

        # Cool down the temperature
        current_temp *= cooling_rate

    # Return the best state found
    return current_state


# Run the simulated annealing algorithm
solution = simulated_annealing()

# Display the solution
print("Solution: ", solution)
print("Conflicts: ", calculate_conflicts(solution))
```

OUTPUT

```
Solution:  [1, 3, 0, 2]
Conflicts:  0

Process finished with exit code 0
```

Program-08 Knowledge Base -Propositional Logic

Algorithm



Q Create a KB using propositional logic and show that the given query entails the KB or not

pseudo code:

① Construct a dictionary to store values and variables

{'P': True, 'Q': False, 'R': True}

② create a list indicating the operations

l = ['∧', 'v']

③ Do the operation

~~if op = '~'~~
~~for i in range (n):~~

operate (x, y, op):

if op == '∧':
    return x and y
elif op == 'v':
    return x or y
elif op == '¬':
    return not x
elif op == '⇒':
    return XOR (x, y)

else:
if x and y:
return True
elif x and not y:
return False
else:
return True

(f) Operate on the variables
ans = True
for i in range(1, n):
ans = ans and operate(d[i-1], d[i], l[i-1])

Entailment

KB ⊨ α

KB ⊨ α is true if α is true in every single model in which KB is true.

For eg

if x > 0 then x ≥ 0
(KB) (α)

MANGAL
PAGE NO.
DATE :

(6) Considering a KB that contains the following propositional logic sentences.

$Q \rightarrow P$

$P \rightarrow \neg Q$

$Q \vee R \Rightarrow KB$

(i) Construct truth table and show value of each sentence in KB. and indicate models in which the KB is true.

(ii) Does KB entail R ?

(iii) Does KB entail $R \rightarrow P$?

(iv) Does KB entail $Q \rightarrow R$?

~~Truth Table~~

Truth Table

P	Q	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$	$Q \rightarrow R$
F	F	F	T	T	F	T
F	F	T	T	T	T	T
F	T	F	F	T	T	F
T	T	F	T	F	T	F
F	T	T	F	T	T	T
T	F	F	T	T	F	T
T	F	T	T	T	T	T
T	T	T	T	F	T	T

R → P
T
F
T
T
F
T
T
T

(iii) Even though R is true, KB is also true, KB does not necessarily entail R

(i) Since P → P is not always true when K.B is true

$$K.B \not\models P \rightarrow P$$

(ii) Since Q → R is true always when KB is true

$$K.B \models Q \rightarrow R$$

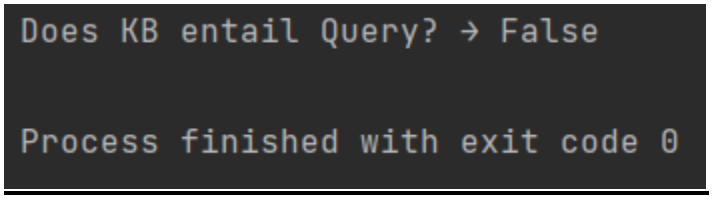
Satisfiability

Code

```
import itertools


# Evaluate a propositional logic sentence in a given model (dictionary of symbol → True/False)
def pl_true(expr, model):
    if isinstance(expr, str):  # atomic symbol
        return model[expr]
    op = expr[0]
    if op == 'not':
        return not pl_true(expr[1], model)
    elif op == 'and':
        return pl_true(expr[1], model) and pl_true(expr[2], model)
    elif op == 'or':
        return pl_true(expr[1], model) or pl_true(expr[2], model)
    elif op == 'implies':
        return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    elif op == 'iff':
        return pl_true(expr[1], model) == pl_true(expr[2], model)
    else:
        raise ValueError("Unknown operator: " + op)


# The truth table enumeration function
def tt_entails(kb, query, symbols):
    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if pl_true(kb, model):
            if not pl_true(query, model):
                return False  # Found a counterexample
    return True

# Example Knowledge Base and Query
# Example: (P ∧ Q) → R, and (P ∧ Q) is true, does R follow?


symbols = ['P', 'Q', 'R']

# Knowledge Base: (P ∧ Q) → R
kb = ('implies', ('and', 'P', 'Q'), 'R')
```

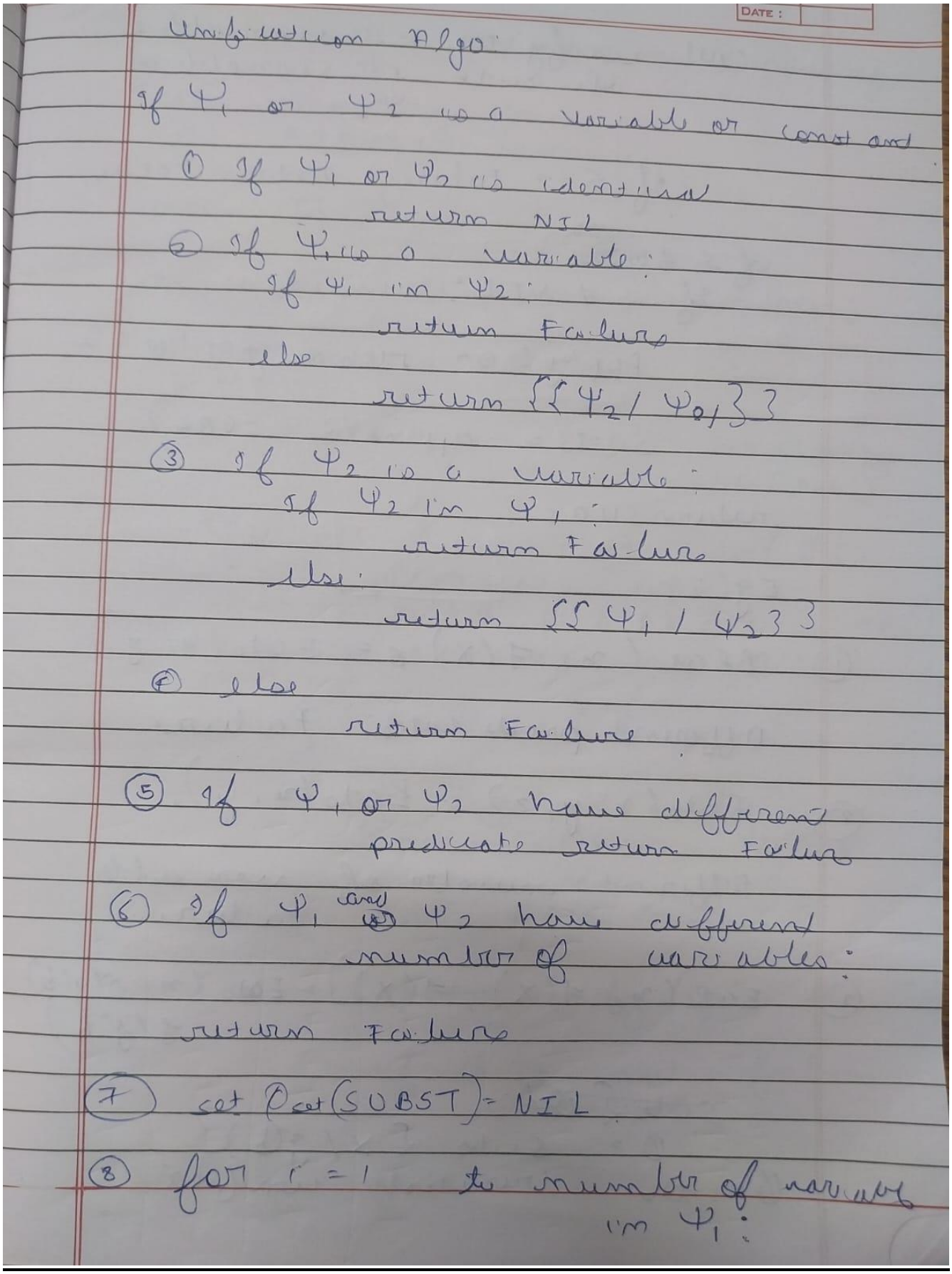
```
# Query: R
query = 'R'

# Check entailment
result = tt_entails(kb, query, symbols)

print("Does KB entail Query? →", result)
```

Output Snapshot



Program-09 Unification in first order logic

Algorithm



Unification Algo:

If Ψ_1 or Ψ_2 is a variable or const and

① If Ψ_1 or Ψ_2 is identical
return NIL

② If Ψ_1 is a variable:
If Ψ_1 in Ψ_2:
return Failure
else
return $\{\{\Psi_2 / \Psi_1\}\}$

③ If Ψ_2 is a variable:
If Ψ_2 in Ψ_1:
return Failure
else:
return $\{\{\Psi_1 / \Psi_2\}\}$

④ else
return Failure

⑤ If Ψ_1 or Ψ_2 have different predicate return Failure

⑥ If Ψ_1 and Ψ_2 have different number of variables:
return Failure

⑦ set Pcat(SUBST) = NIL

⑧ for i = 1 to number of variables in Ψ_1:

call unify on ith variable of ψ_1 with ith variable of ψ_2.

if S == Failure; return Failure

if S ≠ NIL:

Apply S on remaining ψ_1 or ψ_2

SUBS = append(S, SUBS)

return SUBS

Eg:

(1) BEats(x, F(x)) = Eats(x, y)

Different pred. names: Failure.

(2) Eats(x, y, z) = Eats(x, y).

Different number of variables: Failure.

(3) Eats(x, F(x), F(x)) = Eats(x, F(G(b)), F(G(b)))

subs{}.

$\emptyset$ = subs{x/G(b)}

$\emptyset$ is the unified unifier.

MANGAL
PAGE NO.
DATE :

① $P(f(x), g(y), y)$

$P(f(g(z)), g(f(a)), f(a))$

Find $\theta(M \& U)$

$\theta_1 = subs\{x / g(z)\}$

$P(f(g(z)), g(y), y)$

$\neq P(f(g(z)), g(f(a)), f(a))$

$\theta_2 = subs\{y / f(y)\}$

$\theta_3 = subs\{y / a\}$

$P(f(g(z)), g(f(a)), fa)$

$= P(f(g(z)), g(f(a)), f(a))$

θ_1, θ_2 and θ_3 are most general unifier

(1) Q(x, f(y), y)

Q(f(y), y)

⇒ Failure since they both have different pr.

⇒ Failure since they both have different predicates

(3) H(x, g(x))

H(g(y), g(g(z)))

(2) Q(x, f(x)), Q(f(y), y)

Failure since the same variable in same variable appears in different position

(3) H(x, g(x))

H(g(y), g(g(z)))

θ_1 = subs(x/g(z))

H(g(z), g(g(z))) and

H(g(y), g(g(z)))

$\theta_2 = subs(y/z)$

$H(g(z), g(g(z)))$ and

$H(g(z), g(g(z)))$.

θ_1 and θ_2 are MGU

(3) $G(x, f(x)), G(f(y), y)$

$\theta_1 = subs(x/y)$

$G(y, f(y))$ and $G(f(y), y)$

Since g_x want take value $f(y)$ and g at the same time the unification is a failure

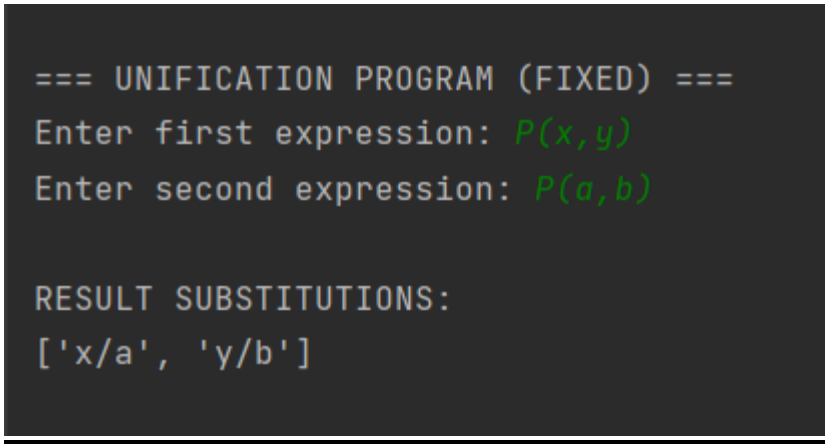
Code

```
import re

def getAttributes(expression):
    inside = expression[expression.index("(")+1 : expression.rindex(")")]
    return inside.split(",")

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(x):
    return len(x) == 1 and x.isupper()

def isVariable(x):
    return len(x) == 1 and x.islower()

def replaceAttributes(exp, old, new):
    predicate = getInitialPredicate(exp)
    attributes = getAttributes(exp)
    attributes = [new if x == old else x for x in attributes]
    return f"{predicate}({','.join(attributes)})"

def apply(exp, substitutions):
    for (new, old) in substitutions:
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    return var in exp

def getFirstPart(expression):
    return getAttributes(expression)[0]

def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)[1:]
    return f"{predicate}({','.join(attributes)})"

def unify(exp1, exp2):
    # Rule 1: identical
    if exp1 == exp2:
        return []

    # Rule 2: constants
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are different constants → FAIL")
            return []
```

```
        return []

    # Rule 3: variable cases
    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    # Rule 4: predicates must match
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Predicates do not match → FAIL")
        return []

    args1 = getAttributes(exp1)
    args2 = getAttributes(exp2)

    if len(args1) != len(args2):
        print("Argument lengths do not match → FAIL")
        return []

    # Recursive unification
    head1 = args1[0]
    head2 = args2[0]

    s1 = unify(head1, head2)
    if s1 == [] and head1 != head2:
        return []

    if len(args1) == 1:
        return s1

    t1 = apply(getRemainingPart(exp1), s1)
    t2 = apply(getRemainingPart(exp2), s1)

    s2 = unify(t1, t2)
    if s2 == [] and t1 != t2:
        return []

    return s1 + s2



if __name__ == "__main__":
    print("\n=== UNIFICATION PROGRAM (FIXED) ===")
    e1 = input("Enter first expression: ").strip()
    e2 = input("Enter second expression: ").strip()

    substitutions = unify(e1, e2)

    print("\nRESULT SUBSTITUTIONS:")
    if substitutions:
        print([f"{var}/{val}" for (val, var) in substitutions])
```

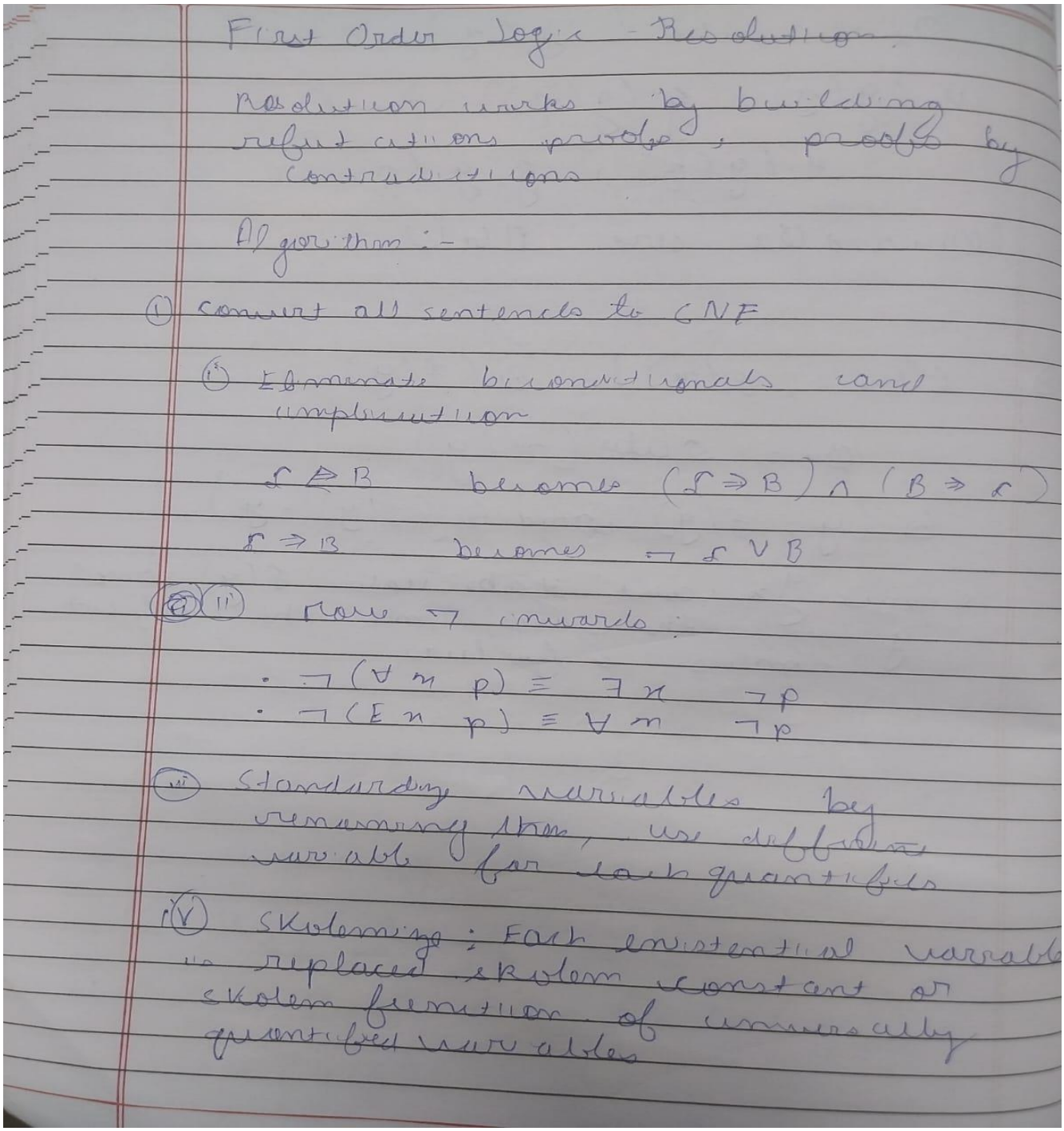
```
else:
    print("UNIFICATION FAILED")
```

Output Snapshot



Program-10 Resolution Program

Algorithm:



First Order Logic - Resolution

Resolution works by building refutations proofs, proofs by contradictions

Algorithm :-

(i) convert all sentences to CNF

(ii) Eliminate biconditionals and implication

$\alpha \Leftrightarrow \beta$ becomes $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

$\alpha \Rightarrow \beta$ becomes $\neg \alpha \vee \beta$

(iii) Move $\neg$ inwards :

- $\neg (\forall x \; p) \equiv \exists x \; \neg p$
- $\neg (\exists x \; p) \equiv \forall x \; \neg p$

(vi) Standardize variables by renaming them, use different variable for each quantifier

(v) Skolemize ; Each existential variable is replaced Skolem constant or Skolem function of universally quantified variables

MANGAL
PAGE NO.
DATE :

(v) Drop universal quantifiers.
∀x Person(x) becomes Person(x)

(vi) Distribute ∧ over ∨:
$(\alpha \wedge \beta) \vee \gamma = (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

(2) Negate conclusion S and convert result to CNF

(3) Add Negated conclusion S to the premise clauses

(4) Repeat steps (1) to (3) until contradiction or no progress is made:

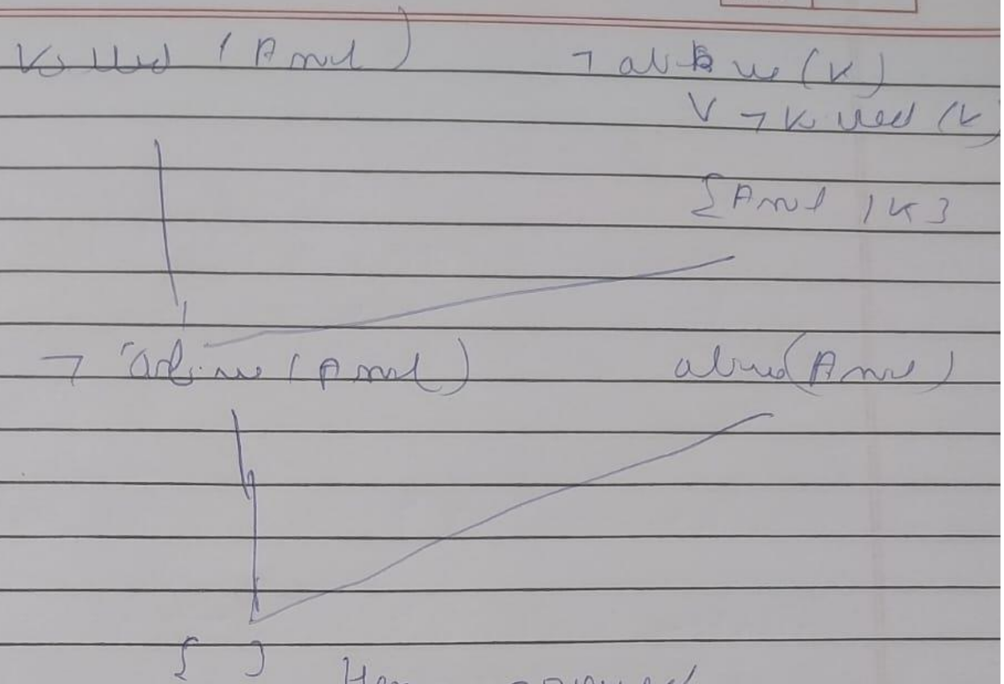
(i) select any 2 clauses

(ii) Resolve them

(iii) If empty clause is result, contradiction has been found

(iv) else add the resolvent to the premises.

(5) If step (4) succeeds, "conclusion has been proved".



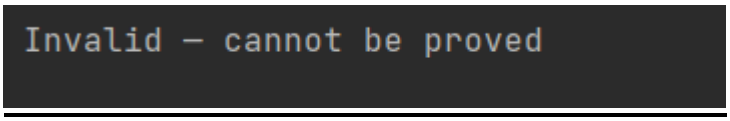
Hence proved

Code

```
from itertools import combinations

# Utility: negate a literal
def negate(literal):
    if literal.startswith('~'):
        return literal[1:]
    else:
        return '~' + literal

# Apply resolution on two clauses
def resolve(ci, cj):
    resolvents = set()
    for di in ci:
        for dj in cj:
            if di == negate(dj):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.add(frozenset(new_clause))
    return resolvents

def fol_resolution(KB, query):
    clauses = set(KB)
    clauses.add(frozenset([negate(query)]))  # add negation of query

    new = set()
    while True:
        pairs = list(combinations(clauses, 2))
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if frozenset() in resolvents:
                return True  # empty clause found → proved
            new = new.union(resolvents)

        if new.issubset(clauses):
            return False  # no new clauses → cannot prove
        clauses = clauses.union(new)

# ------------------------
# Knowledge Base in CNF
# ------------------------
KB = [
    frozenset(['~Food(x)', 'Likes(John,x)']),  # John likes all food
    frozenset(['Food(Apple)']),
    frozenset(['Food(Vegetables)']),
    frozenset(['~Eats(x,y)', 'Killed(x)', 'Food(y)']),
```

```
    frozenset(['Eats(Anil,Peanuts)']),
    frozenset(['Alive(Anil)']),
    frozenset(['~Eats(Anil,x)', 'Eats(Harry,x)']),
    frozenset(['~Alive(x)', '~Killed(x)']),
    frozenset(['Killed(x)', 'Alive(x)'])
]

query = 'Likes(John,Peanuts)'

# Run the resolution
if fol_resolution(KB, query):
    print("Proved:", query)
else:
    print("Invalid — cannot be proved")
```

OutputSnapshot



Program 11: Forward Chaining inference

MANGAL
PAGE NO.
DATE :

Forward Reasoning chaining

KB: Knowledge Base
f: Atomic sentence

new = f3

repeat until new is empty:

for rule in KB:

$(p1 \wedge p2 \wedge \ldots \Rightarrow q)$ = standardize variable(rule)

for each θ such that subs(θ, $p1 \wedge p2 \wedge p3 \ldots \wedge pn$) = subs($\theta$, $p1' \wedge p2' \wedge p3' \ldots \wedge pn'$)
for $pi' \wedge p2' \wedge p3'$ in KB:

$q' \leftarrow$ subs(θ, p)

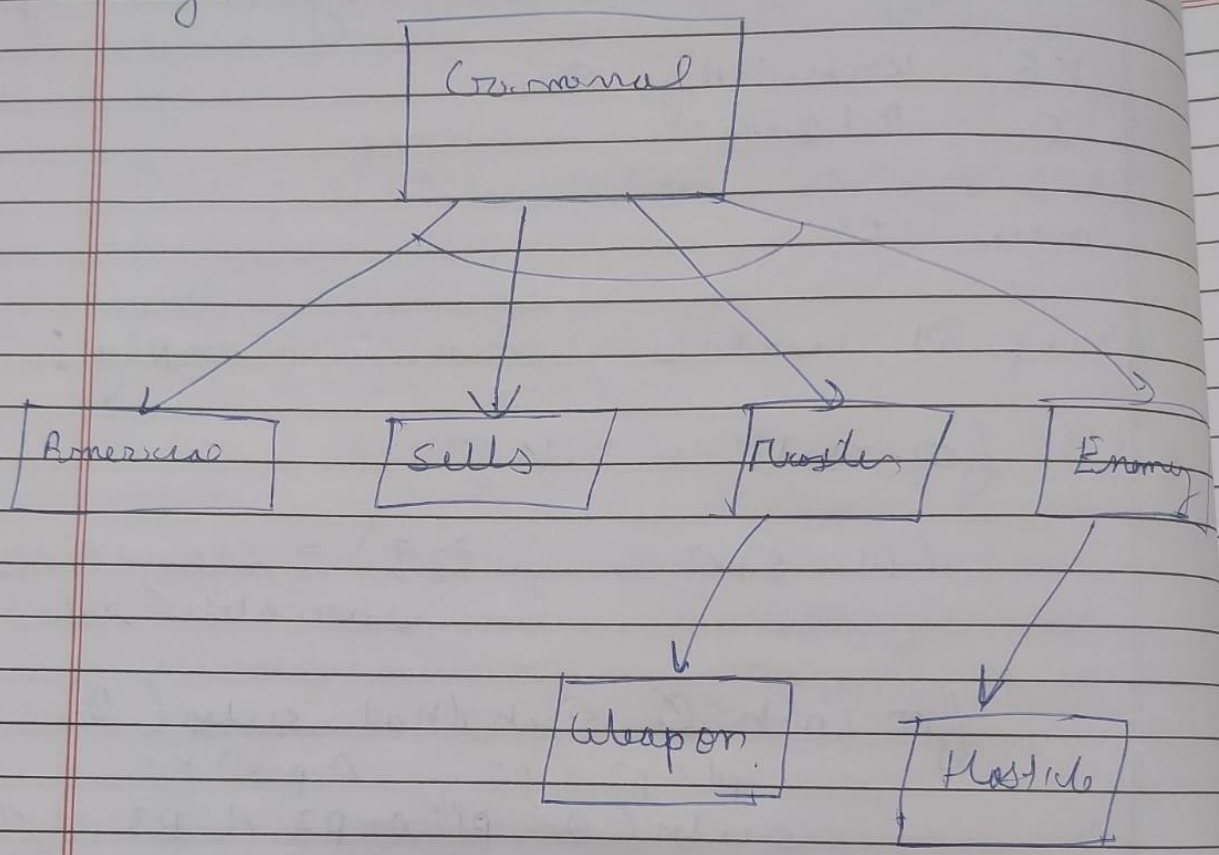
If q' does not unify with some sentence in KB or new:
Add q' to new

$\phi \leftarrow$ unify(q', f)

if ϕ is not failure return ϕ

return false

Eg :-



Code-

```
# Forward Chaining in First-Order Logic (FOL)
# Example: Prove "Criminal(Robert)"

# Knowledge Base (KB)
KB = [
    # Rule 1: It is a crime for an American to sell weapons to hostile nations
    {"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"], "then":
"Criminal(p)"},

    # Rule 2: Country A owns some missiles (T1)
    {"fact": "Owns(A, T1)"},
    {"fact": "Missile(T1)"},

    # Rule 3: Missiles are weapons
    {"if": ["Missile(x)"], "then": "Weapon(x)"},

    # Rule 4: All missiles owned by A were sold by Robert
    {"if": ["Missile(x)", "Owns(A, x)"], "then": "Sells(Robert, x, A)"},

    # Rule 5: Enemies of America are hostile
    {"if": ["Enemy(x, America)"], "then": "Hostile(x)"},

    # Rule 6: Robert is an American
    {"fact": "American(Robert)"},

    # Rule 7: Country A is an enemy of America
    {"fact": "Enemy(A, America)"}
]
# Function to extract all current known facts
def get_facts(kb):
    return {rule["fact"] for rule in kb if "fact" in rule}

# Function to perform variable substitution
def substitute(expr, var, val):
    return expr.replace(f"({var})", f"({val})").replace(f"{var},",
f"{val},").replace(f",{var})", f",{val})")

# Forward chaining algorithm
```

```
def forward_chain(kb, query):
    inferred = set()
    facts = get_facts(kb)
    new_inference = True

    print("Initial Facts:", facts)
    print("Goal:", query)
    print("\n--- Forward Chaining Process ---")
    while new_inference:
        new_inference = False
        for rule in kb:
            if "if" in rule:
                conditions = rule["if"]
                result = rule["then"]

                # Check if all conditions are satisfied
                satisfied = True
                temp_result = result
                substitution = {}

                for cond in conditions:
                    matched = False
                    for fact in facts:
                        if cond.split("(")[0] == fact.split("(")[0]:
                            # Try variable substitution
                            c_args = cond[cond.find("(") + 1:-1].split(",")
                            f_args = fact[fact.find("(") + 1:-1].split(",")

                            if len(c_args) == len(f_args):
                                for i in range(len(c_args)):
                                    if c_args[i].islower():
                                        substitution[c_args[i]] = f_args[i]
                                    elif c_args[i] != f_args[i]:
                                        break
                                else:
                                    matched = True
                                    break
                    if not matched:
                        satisfied = False
```

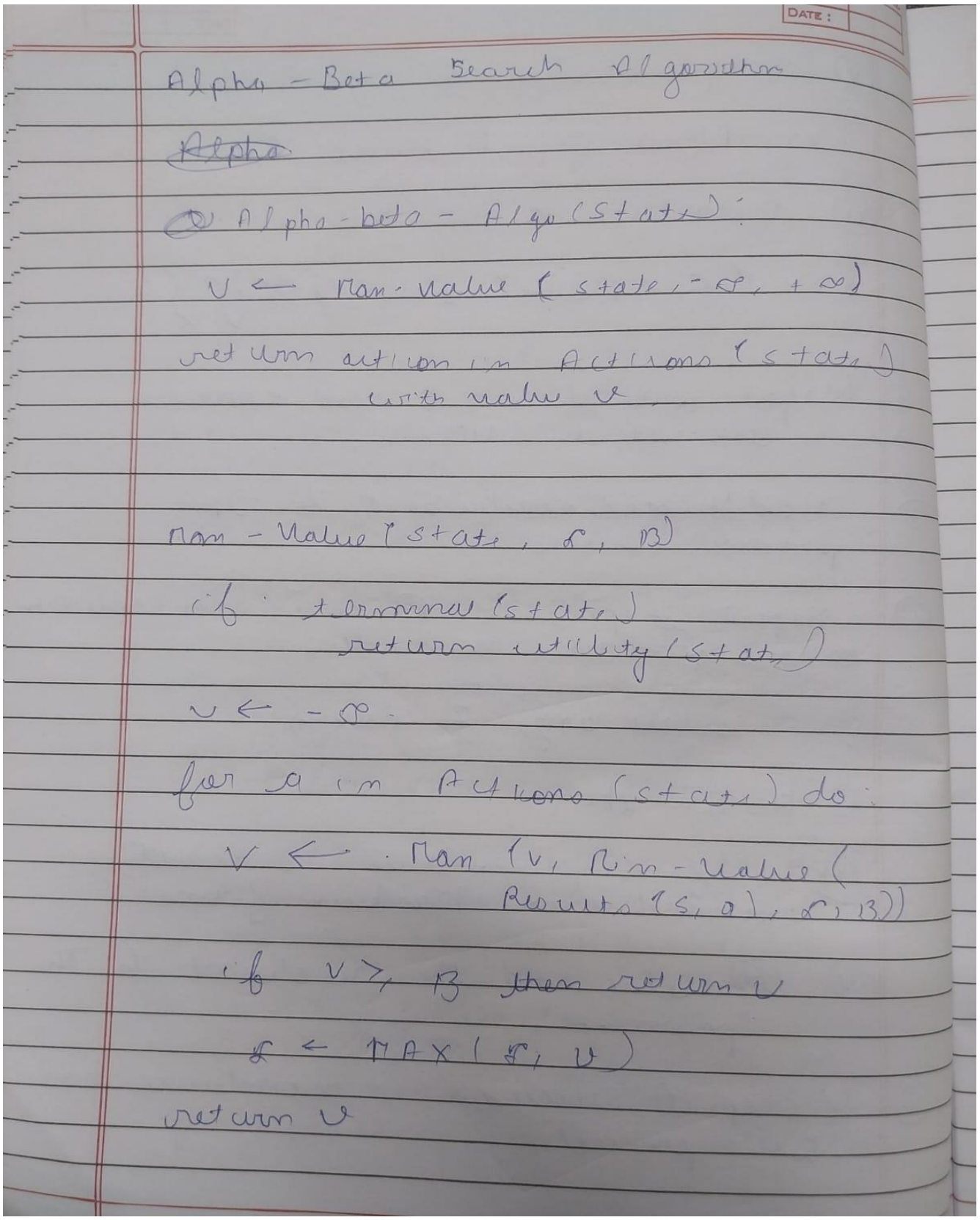
```
                break

        if satisfied:
            for var, val in substitution.items():
                temp_result = substitute(temp_result, var, val)

            if temp_result not in facts:
                facts.add(temp_result)
                inferred.add(temp_result)
                new_inference = True
                print(f"Inferred: {temp_result}")

            if temp_result == query:
                print("\n✅ Goal Reached!")
                return True

    print("\n❌ Goal Not Reached.")
    return False

# Main Execution
if __name__ == "__main__":
    query = "Criminal(Robert)"
    result = forward_chain(KB, query)
    print("\nFinal Result:", "Robert is a Criminal" if result else "Cannot prove Robert is Criminal")
```

Output-

```
Initial Facts: {'Missile(T1)', 'Owns(A, T1)', 'Enemy(A, America)', 'American(Robert)'}
Goal: Criminal(Robert)

--- Forward Chaining Process ---
Inferred: Weapon(T1)
Inferred: Sells(Robert, T1, A)
Inferred: Hostile(A)
Inferred: Criminal(Robert)

✅ Goal Reached!

Final Result: Robert is a Criminal
```

Program 12 :Alpha Beta Pruning

Algorithm-



function:

Max-Value (state, α, β)

if terminal (state):
    return utility (state)

$v \leftarrow +\infty$

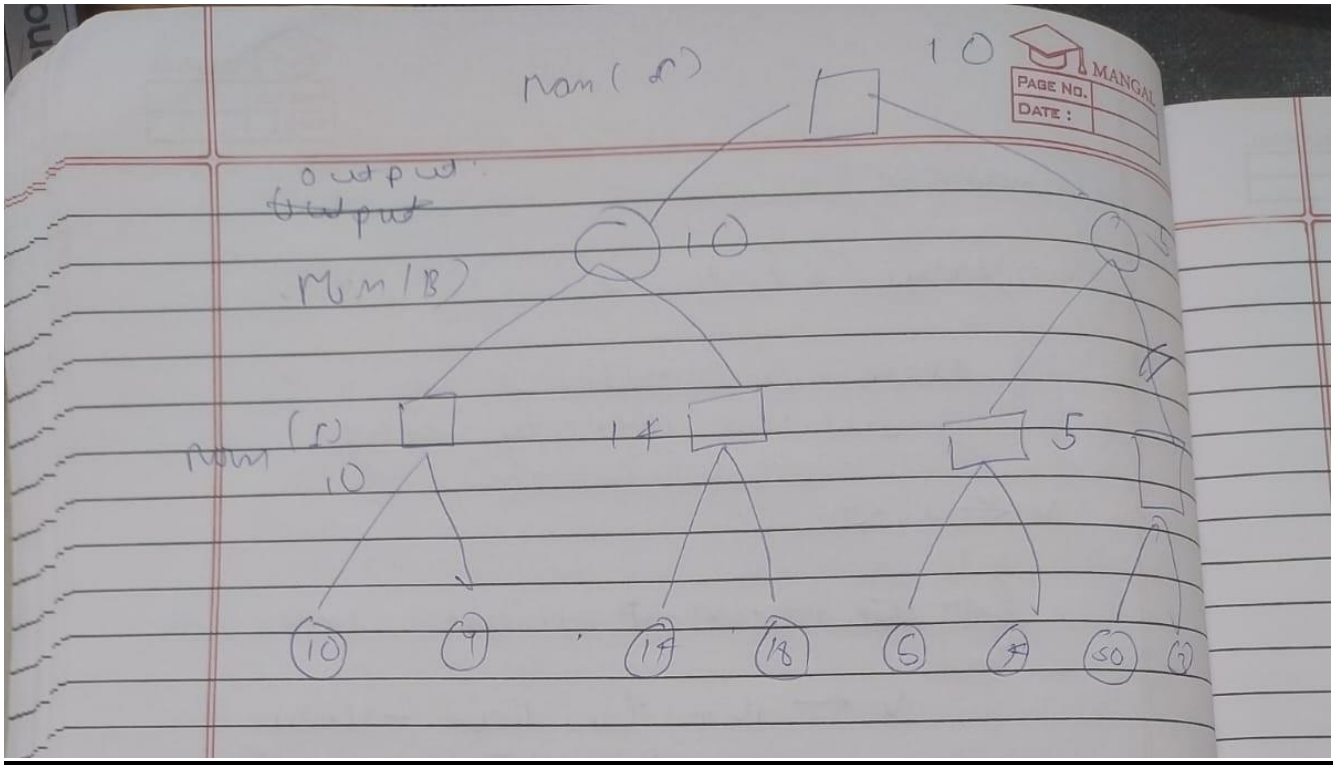
for a in Actions (state) do:

    $v \leftarrow \max(v, \text{Min-Value}($
    $\text{RESULT}(s, a), \alpha, \beta))$

    if $v \geq \beta$:
        return v

    $\alpha \leftarrow \max(\alpha, v)$

return v


max(a)
10
output
max(B)
10
max(a)
10
14
5
10
9
14
18
5
4
50
6

Code-

```
import math

# Alpha-Beta Pruning Function
def alpha_beta_pruning(depth, node_index, is_max, scores, alpha, beta, max_depth, path):
    # Base case: return leaf node value
    if depth == max_depth:
        path.append(node_index)
        return scores[node_index]

    if is_max:
        best = -math.inf
        for i in range(2):  # each node has 2 children
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, scores, alpha, beta, max_depth, path)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth} (MAX node): alpha={alpha}, beta={beta}")
                break  # Beta cut-off
        return best
    else:
        best = math.inf
        for i in range(2):
            val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, scores, alpha, beta, max_depth, path)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth} (MIN node): alpha={alpha}, beta={beta}")
                break  # Alpha cut-off
        return best


# ----------------------------
```

```
# Example tree for demonstration
# ----------------------------
# Binary tree leaves (depth = 3)
scores = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = int(math.log(len(scores), 2))

path = []

print("Leaf node values:", scores)
print("Applying Alpha-Beta Pruning...\n")

root_value = alpha_beta_pruning(0, 0, True, scores, -math.inf, math.inf, max_depth,
path)

print("\nResult:")
print(f"Value of the root node (best achievable for MAX): {root_value}")
print(f"Nodes explored (leaf indices): {path}")
```

Output -

```
Leaf node values: [3, 5, 6, 9, 1, 2, 0, -1]
Applying Alpha-Beta Pruning...

Pruned at depth 2 (MAX node): alpha=6, beta=5
Pruned at depth 1 (MIN node): alpha=5, beta=2

Result:
Value of the root node (best achievable for MAX): 5
Nodes explored (leaf indices): [0, 1, 2, 4, 5]
```