Page No.
Date

LAB-1

1) Implement tic tac toe game, 2 players

Algorithm:

1) Initialize a 3x3 game board using list in Python and initialize all cells as empty
2) Write a function to display the board by printing the current state of the board.
3) Write a function to check for win, check all rows, columns and diagonals.

```
#row
for row in board
    if all(cell == player for cell in row)
        return True
#col
for col in range(3):
    if all(board[row][col] == player for row in range(3)):
        return True
#diagonal
if all(board[i][i] == player for i in range(3)) or
    all(board[i][2-i] == player for i in range(3)):
        return True
```

4) Check the current_player. If current player is human, call get_player_move(board, current_player) else call ai_move(board)
5) After every move check if the player wins. If there is no win condition, change the current player.
6) For every move, we check if the board is full. If it is If board is full, display the game is a tie.

2) Vacuum cleaner agent

```
def vacuum_cleaner_agent(location, status):
    x,y = location
    if status[x][y] == 'Dirty':
        return f"vacuum cleaner is at ({x}, {y}) and
        it is dirty. Cleaning"
    else:
        return f"The vacuum cleaner is at ({x}, {y}) and
        it is clean. Moving".
```

~~Status~~

```
Status = [['Dirty', 'Clean'], ['Dirty', 'Dirty']]
location = (0,0)
while True:
    action = vacuum_cleaner_agent(location, status)
    print(action)

    x,y = location
    if status[x][y] == 'Dirty':
        status[x][y] = 'Clean'.

    if Status[x][0] == 'Clean' and status[0][1] == 'Clean'
    and Status[1][0] == 'Clean' and status[1][1] = 'Clean':
        print ("All clean").
        break

    if y<1:
        location = (x, y+1)
    elif x<1
        location = (x+1, 0)
```

Output

VC is at (0,0); dirty. Cleaning

VC is at (0,1), clean. Moving

VC is at (1,0), dirty, cleaning

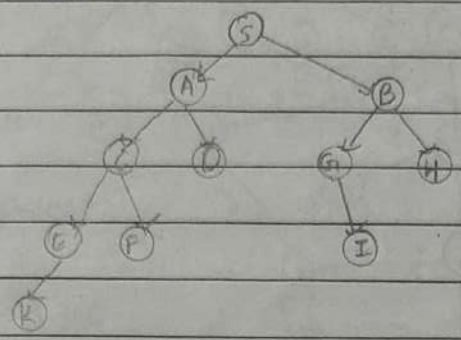
VC is at (1,1), dirty, clean

All clean

LAB-2

Solutions for the given search problems and pseudocode

1. Perform BFS

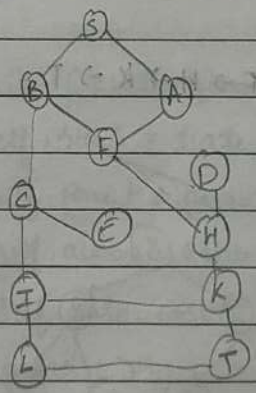


Frontier: S A B C D G H E F I K

Explored: S A B C D G H E ↓ goal reached

Backtracking: S → A → C → E

2. Performing DFS both tree search and graph search



Frontier:

~~S~~

~~B~~ A

~~C~~ F A

~~J~~ E F A

~~L~~ K E F A

T K E F A

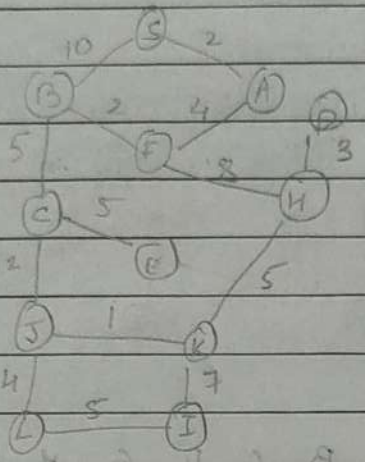
↓ goal reached

Explored: S B C J L

Path:

S → B → C → J → L

3. Perform Uniform cost search



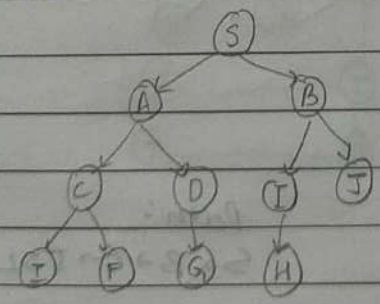
Frontier: S(0) A(2) B(10) F(6) H(14) C(15) K(19) D(17) J(17) E(20) L(21) T(26)

Explored: S(0) A(2) F(6) B(10) H(14) C(15) J(17) K(19) T(16)

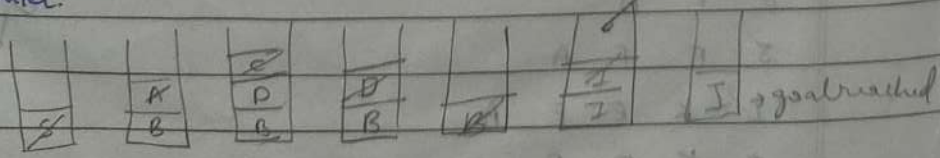
Total cost = 26

Path: S → A → F → H → K → T

4. Performing DLS



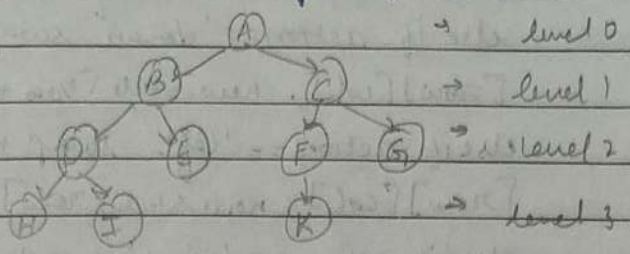
Frontier:



Explored: S A C D B I J

Path: S → B → J

5. Performing Iterative deepening search



Search process:

i) A

ii) A → B → C

iii) A → B → D → E → C

iv) A → B → D → E → C → F → G

Path: A → C → G

1) Solve 8-puzzle problem.

Pseudocode:

Class Node:

```
function_init_(state, parent, action, path_cost=0):
    set self.state = state
    set self.parent = parent
    set self.action = action
    set self.path_cost = path_cost
function expand():
    Create children
    set: row, col = find_blank()
    Create possible_actions
    if row > 0 then add 'up' to possible_actions
    if row < 2 then add 'down' to possible_actions
    if col > 0 then add 'left' to possible_actions
    if col < 2 then add 'right' to possible_actions
    for action in possible_actions:
        create new_state as a copy of set_state
        if action == 'up' then swap new_state[row][col]
```

with new_state[row-1][col]
else if action == 'down' swap new_state
[row][col], new_state[row+1][col]
else if action == 'left' swap new_state
[row][col], new_state[row][col-1]
else if action == 'right' swap new_state
[row][col], new_state[row][col+1]
append new Node(new_state, self, action,
self.path_cost+1) to children
return children
function find_blank():
for row from 0 to 2:
for col from 0 to 2:
if self.state[row][col] == 0 then
return row, col
function dept first_search(initial_state, goal_state):
set frontier = [Node(initial_state)]
set explored = empty set
while frontier is not empty:
set node = frontier.pop()
if node.state == goal_state then
return node
add tuple of node state to explored
for child in node.expand():
if tuple of child state not in explored
then append child to frontier
return none
function print_solution(node):
create path
while node is not none:
append (node.action, node.state) to path
set node = node.parent
reverse path.

```
for (action, state) in path:
    if action is not none then print "action:",
    action
    print state
    print " "
set initial_state = [[1,2,3], [0,4,6], [7,5,8]]
set goal_state = [[1,2,3], [4,5,6], [7,8,0]]
set solution = depth_first_search (initial_state,
goal_state)
if solution is not none then
    print "Sol found".
    call print_solution (solution)
else
    print ("Sol not found")
```

(2) Implement iterative deepening search algorithm

```
function iterative_deepening_search (initial_state, goal_state,
max_depth):
    for depth from 0 to max_depth:
        set results = depth_limited_search (initial_state,
        goal_state, depth)
        if result is not none then
            return result
    return none
function dept_limited_search (node, goal_state, limit):
    if node.state == goal_state then
        return node
    If node.depth >= limit then
        return none
    for each child in expand (node):
        set result = depth_limited_search (child,
        goal_state, limit)
```

if result is not none then
return result
return none
set initial_state, goal_state, max_depth
set solution = Iterative_deepening_search(initial_state, goal_state, max_depth)
if solution is not none then print solution
else print "no solution found"

LRUD

LAB. 04

Page No.
Date

13

i). 8 puzzle problem using heuristic approach (A^*)

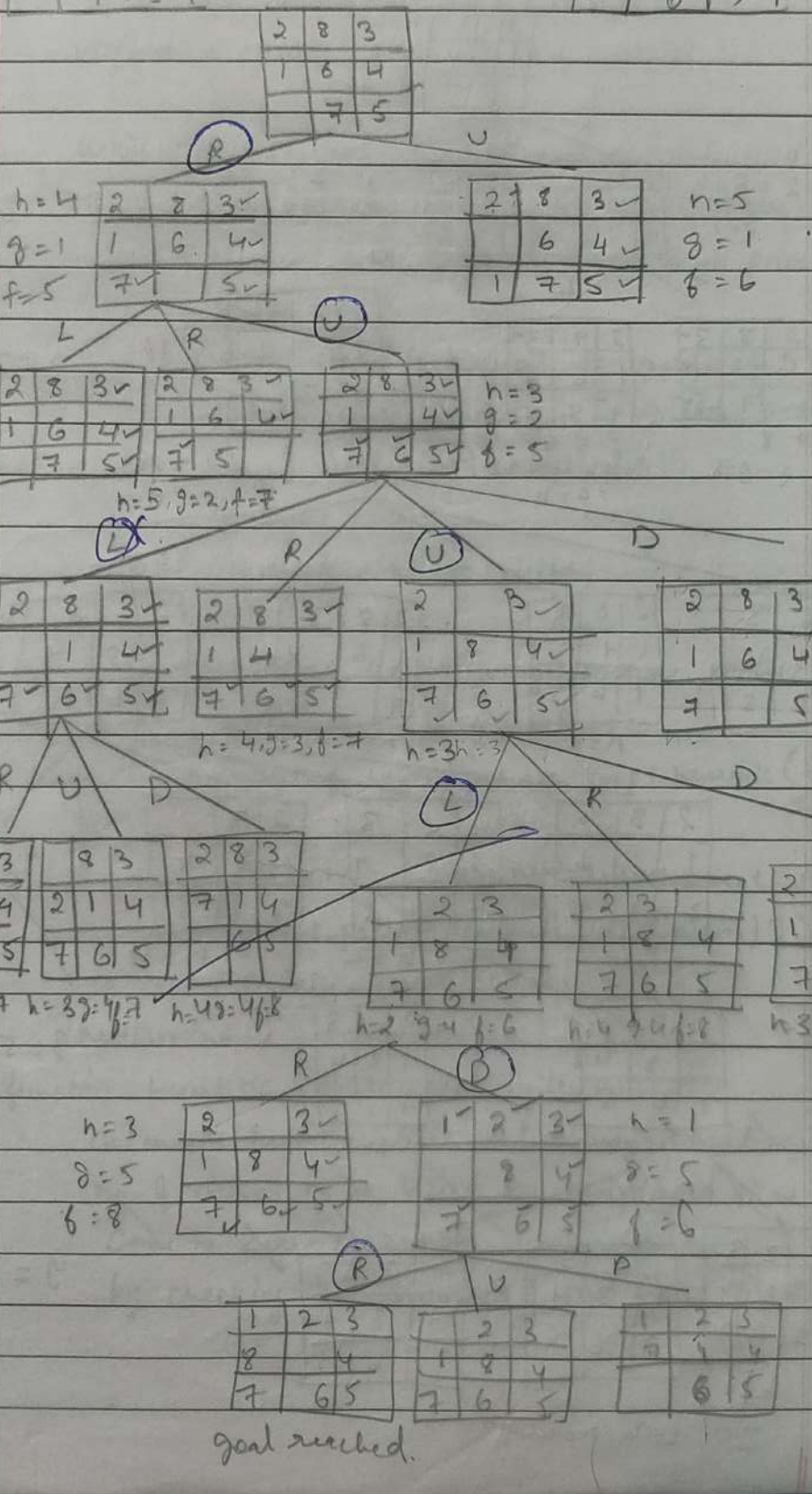
Initial State:

2	8	3
1	6	4
	7	5

$g=0$
$h=5$

Goal state

1	2	3
8		4
7	6	5



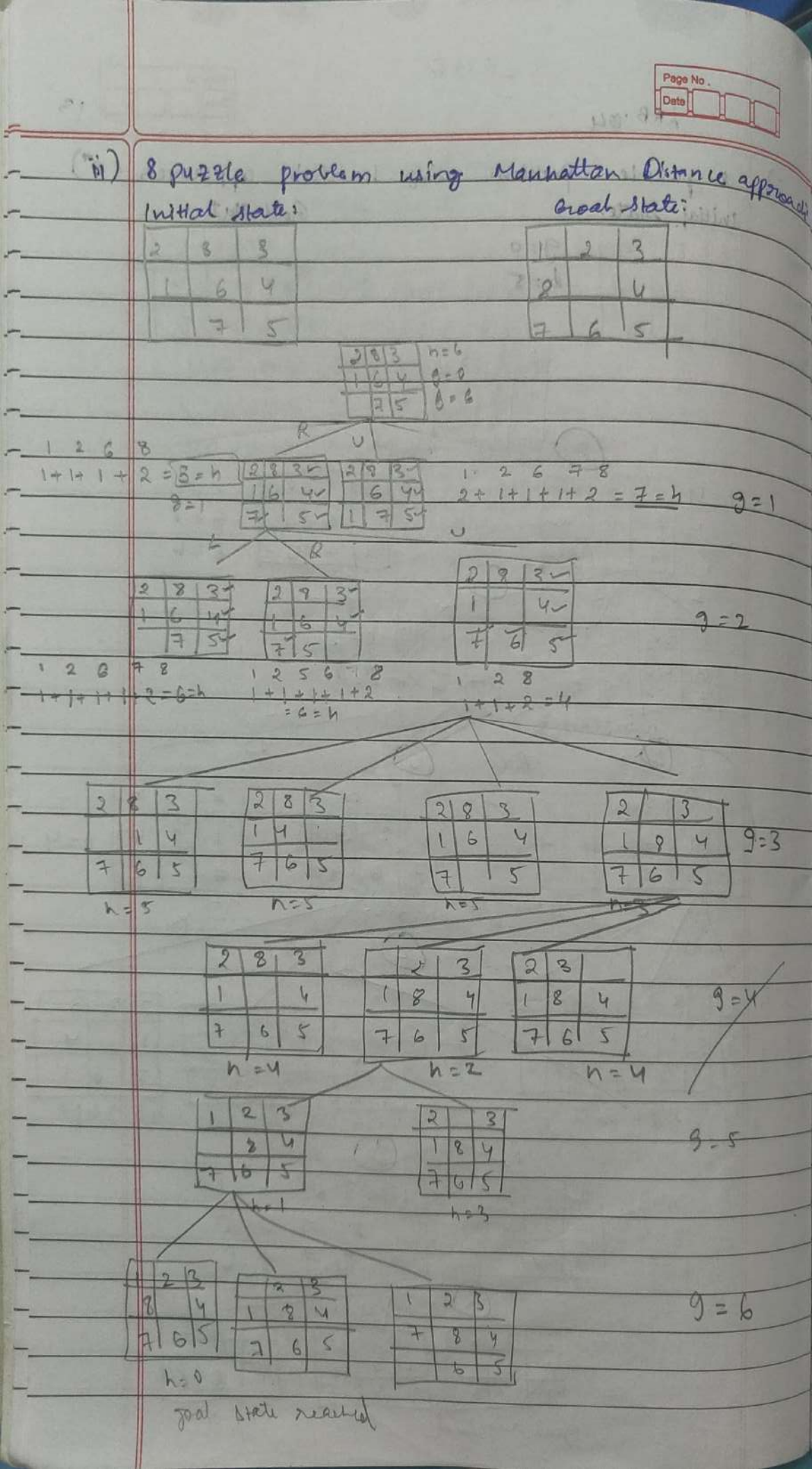
ii) 8 puzzle problem using Manhattan Distance approach

Initial State:

2	8	3
1	6	4
	7	5

Goal State:

1	2	3
8		4
7	6	5



i) Pseudocode for 8-puzzle using Heuristic approach.

```
function Solve_8puzzle(initial_state, goal_state):
    priority_queue = [(heuristic(initial_state, goal_state), 0,
                       initial_state, [])]
    visited = empty set

    while priority_queue is not empty:
        (f_cost, g_cost, current_state, current_path) =
            pop element with smallest f_cost from priority_queue

        if current_state is equal to goal_state:
            return current_path + [current_state]
        if current_state is in visited:
            continue
        add current_state to visited

        for next_state, action in get_possible_moves(current_state):
            new_g_cost = g_cost + 1
            new_f_cost = new_g_cost + heuristic(next_state,
                                                goal_state)
            Push(new_f_cost, new_g_cost, next_state,
                 current_path + [(current_state, action)]) into
                 priority_queue
    return None
function heuristic(state, goal_state):
    Calculate the heuristic value by summation of
    no. of moves to be taken by misplaced tiles to
    come to correct position
    by checking the number of mis place tiles.
```

function get_possible_moves(state):
find the position of 0 tile and
generate a list of possible moves [Up, Down, left, right) by swapping
blank the tile with its adjacent
Return list of [new_state, action] pairs.

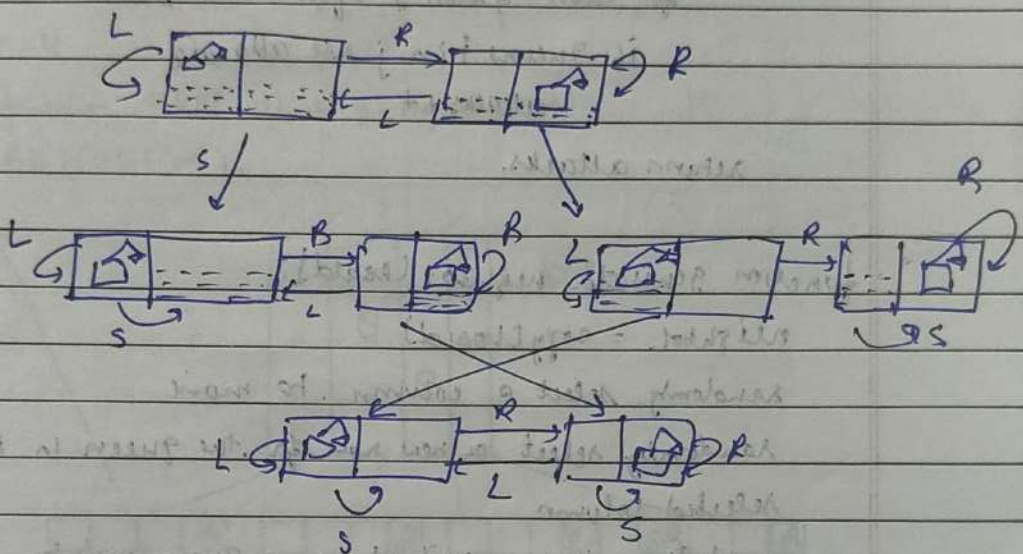
ii) pseudocode for 8 puzzle using Manhattan approach

function solve_8puzzle(initial_state, goal_state):
priority_queue = [(manhattan_distance(initial_state, goal_state), 0, initial_state, [])]
visited = empty set
while priority_queue is not empty:
(f_cost, g_cost, current_state, current_path) = pop smallest f_cost from queue
if current_state == goal_state: return current_path + [current_state]
if current_state in visited: continue
visited.add(current_state)
for next_state, action in get_moves(current_state):
new_g_cost = g_cost + 1
new_f_cost = new_g_cost + manhattan_dist (next_state, goal_state)
add (new_f_cost, new_g_cost, next_state, current_path + [(current_state, action)]) to queue
return None

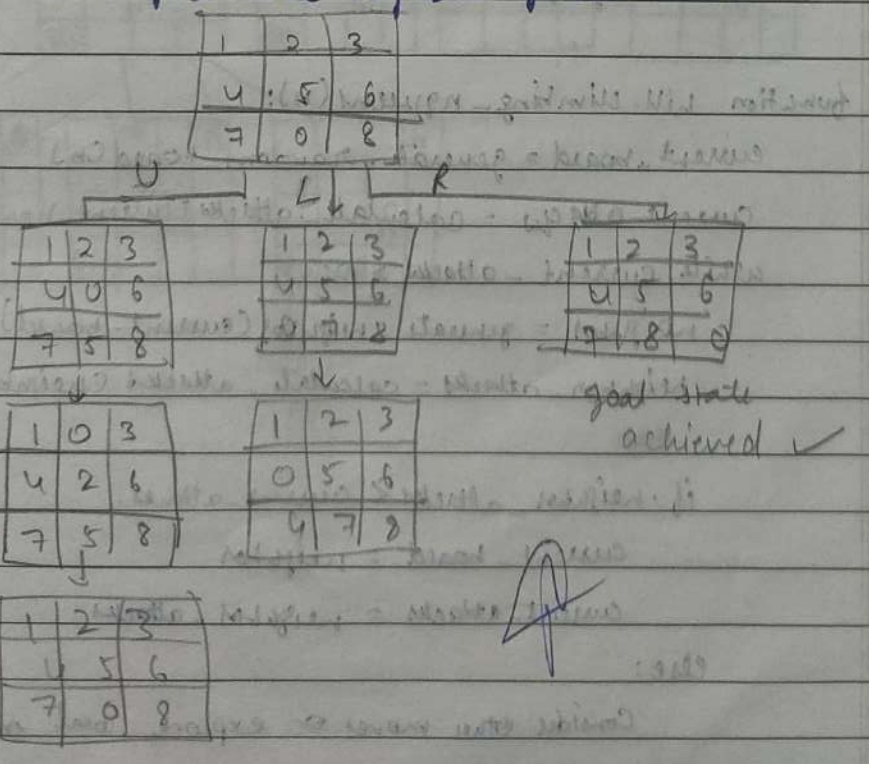
function manhattan_dist (state, goal_state):
distance = 0
for each tile in state:
if tile != 0:
distance += |goal_row - current_row| + |goal_col - current_col|
return distance.

function get_moves (state):
find blank tile position
generate possible moves (up, down, left, right) by swapping blank tile with adjacent tiles
return list of (new_state, action) pairs.

2) vacuum cleaner agent: (space tree)



3) State space tree - 8 puzzle problem (DFS):



LAB 5:

Implement Hill climbing search algorithm to solve N-queens problem:

Function to calculate the number of attacking pairs.

```
function calculate_attacks(board):
    attacks = 0
    for each queen i in board:
        for each queen j after i in board:
            if queens i in j are attacking:
                attacks++
    return attacks.
```

```
function generate_neighbor(board):
    neighbor = copy(board)
    randomly select a column to move
    randomly select a new row for the queen in the
    selected column
    update neighbor with the new queen position
    return neighbor
```

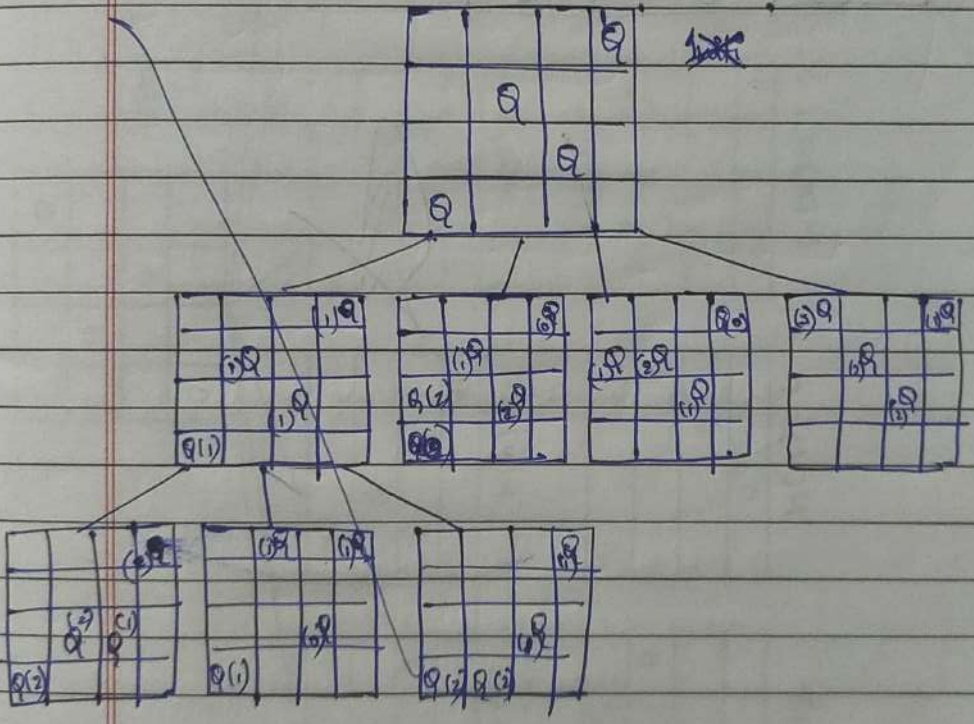
```
function hill_climbing_nqueens(n):
    current_board = generate_random_board(n)
    current_attacks = calculate_attacks(current_board)
    while current_attacks > 0:
        neighbor = generate_neighbor(current_board)
        neighbor_attacks = calculate_attacks(neighbor)

        if neighbor_attacks < current_attacks:
            current_board = neighbor
            current_attacks = neighbor_attacks
        else:
            Consider other moves or explore local minima
```

return current_board

function generate_random_board(n):
    board = new array of size n
    for each column in board:
        Place a queen in a random row in the column
    return board

n=4
solution = hill_climbing_nqueens(n)
Print(solution).



Initial board

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

Conflicts = 2

Final board

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

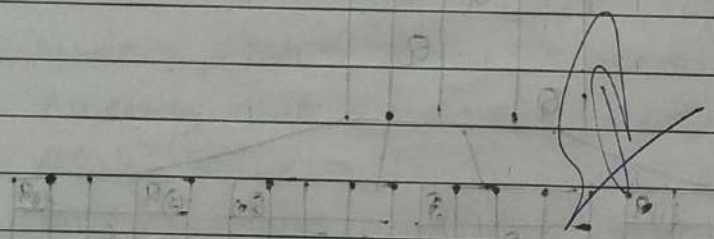
Conflicts = 2

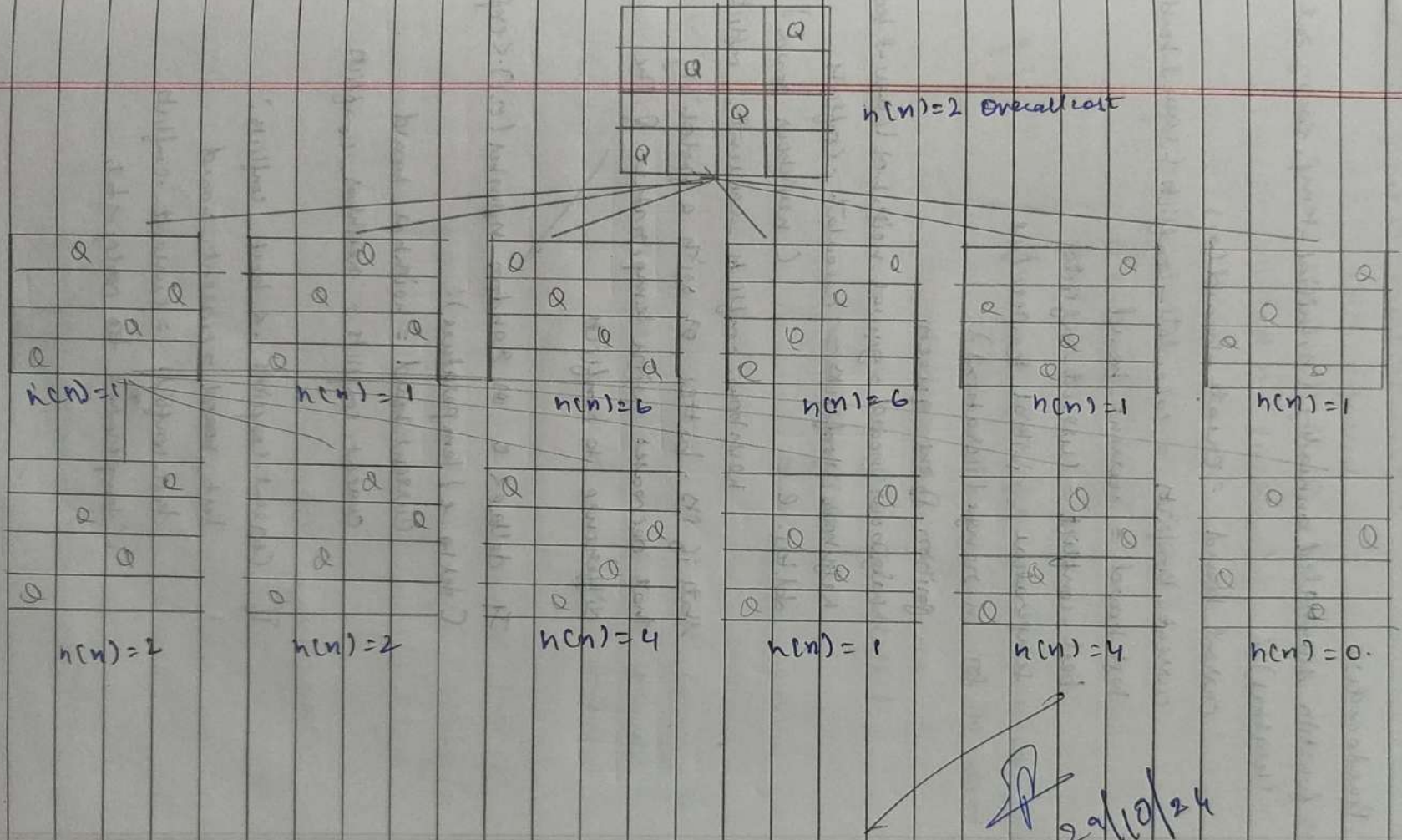
Solution : [3, 1, 2, 0]

Conflicts = 2

Current heuristic = 1 + 1 + 1 = 4

8	4	5	Q
4	Q	7	6
6	8	Q	4
Q	6	4	8





29/10/24

Page No.
Date

LAB-6

1. Analyse and Implement N Queen Problem using simulated annealing technique

Pseudocode:

function simulated annealing (n, initial_temp, cooling_rate, iterations):

current_board = create_board(n)

current_conflicts = calculate_conflicts (current_board)

best_board = current_board

best_conflict = current_conflicts

temperature = initial_temperature

for in range (iterations):

position of one queen

neighbour_board = general_neighbor (current_board)

neighbour_conflicts = calculate_conflicts

delta_e = (neighbour_board)

neighbour_conflicts - current_conflicts

state if it's better or with a probability that decreases with temperature of the difference to conflicts

If delta_e < 0 or random_number(0,1) < exp (delta_e / temperature):

Current_board = neighbor_board

current_conflicts = neighbor_conflicts

If Current_conflicts < best_conflicts:

best_board = current_board

best_conflicts = current_conflicts

temperature = to cooling state

return best_board, best_conflicts

function create_board(n):
with n queens
return a list of n random integers between
0 and n-1 (inclusive)

function calculate_conflicts (board):
n = length of board
conflicts = 0
for each pair of queens (i,j) where i<j:
if queens i and j are in the same row
or diagonal:
conflicts = conflicts + 1.
return conflicts.

2. Propositional Logic : Truth Table Enumeration.

Program:

```
import itertools
def evaluate_formula(formula, valuations):
    formula = formula.replace('p', str(valuation['p']))
    formula = formula.replace('q', str(valuation['q']))

    return eval(formula)

def extract_variables(formula):
    variables = set()
    for char() in formula:
        if char.isalpha():
            variables.add(char)
    return list(variables)

def generate_truth_table(KB, query):
    variables = extract_variable(KB) + extract_variable(query)
    variables = list(set(variables))

    print("Truth Table")
    print("|", join(variables + ["KB", "Query"]))
    Print("-"*(len(variables)*4+12))

    entails_query = True

for assignment in itertools.product([False, True], repeat
                                    = len(variables)):

    valuation = dict(zip(variables, assignment))
```

KB_truth = evaluate_formula (Query, valuation)

row = [str ('T' if valuation [var] else 'F') for var in variables]
row.append (str ('T' if query_truth else 'F'))

if KB_truth and not query_truth:
    entails_query = False

KB = input ("Enter KB")
query = input ("enter query").

generate_truth_table (KB, query)

OUTPUT:

Enter KB (eg: 'P and (P1 => Q1)': 'P1=Q
Enter the query: Q
Truth Table:

P1	Q1	KB	Query
F	T	F	F
F	T	T	T
T	F	F	F
T	T	T	T

KB entails query: False

LAB-7

First Order Logic:

1. "John is a human"

FOL:

$$H(John)$$

Explanation:

$H(x)$: x is human

John" is a constant representing an individual named john.

2. "Every human is mortal"

FOL:

$$\forall x(H(x) \Rightarrow M(x))$$

$H(x)$: x is a human

$M(x)$: x is mortal

$\forall x$: For all individual x, if x is a human then x is mortal

3. "John loves Mary"

$$L(John, Mary)$$

$L(x,y)$: x loves y

John and Mary are individuals.

4. "There is someone who loves Mary"

$$\exists x(L(x, Mary))$$

$L(x,y)$: x loves y

$\exists x$: There exists an individual x such that x loves Mary

5. All dogs are animals.

FOL

$$\forall x(D(x) \rightarrow A(x))$$

$D(x)$: x is a dog

$A(x)$: x is an animal

$\forall x$: For all individuals x, if x is dog $\rightarrow$ x is animal

6. "Some dogs are brown".

$\exists x (D(x) \wedge B(x))$

$D(x)$ is: x is a dog.

$B(x)$: x is brown

$\exists x$: There exists an individual x such that x is a dog an x is brown.

1) Let:

S: John is stupid

L: John is lazy

$\neg S$: John is not stupid

$\neg L$: John is not lazy

Either John is not stupid and he is lazy, or he is stupid

$(\neg S \wedge L) \vee S$

John is stupid. S

Therefore John is not lazy $\neg L$

a)

S	L	$\neg S$	$\neg L$	$(\neg S \wedge L)$	$(\neg S \wedge L) \vee S$	S	$\neg L$
T	T	F	F	F	T	T	F
T	F	F	F	F	T	T	T
F	T	T	T	T	T	F	F
F	F	T	F	F	F	F	T

$(P \rightarrow Q) \vee (\neg P \wedge R)$

2)

P	Q	R	$P \rightarrow Q$	$\neg P$	$\neg P \wedge R$	if P then Q else R
T	T	T	T	F	F	T
T	T	F	T	F	F	T
T	F	T	F	F	F	F
T	F	F	F	F	F	F
F	T	T	T	T	T	T
F	T	F	T	T	F	T
F	F	T	T	T	T	T
F	F	F	T	T	F	F

function translate_to_fol(sentence):
sentence = lowercase and trim(sentence)

if sentence contains "is a human":
return translate_is_a_human(sentence)
else if sentence contains "is mortal":
return translate_mortal(sentence)
else if sentence contains "loves":
return translate_loves(sentence)
else if sentence contains "every":
return trans_every(sentence)
else if sentence contains "exists" or "there is":
return trans_exist(sentence)
else if sentence contains "not all":
return trans_not_all(sentence)
else if sentence contains "if" and "then":
return trans_if_then(sentence)
else if sentence contains "and":
return trans_conjuction(sentence)
else
return: "Translation not available"

function translate_is_human(sentence):
if sentence matches the pattern "subject is a human"
sub = extracted subject
return "H(subject)"
else
return "Invalid"

function main():
while user input is not "exit":
get sentence from user
fol_translation = translate_to_fol(sentence)

main()

Lab-08

Implement unification in first order logic.

```
FUNCTION unify(expr1, expr2, substitutions={}):
    if expr1 == expr2:
        return substitutions

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

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

    if is_compound(expr1) AND is_compound(expr2):
        if expr1[0] != expr2[0] OR len(expr1[1:]) !=
                                    len(expr2[1:]):
            raise "Unification Error"
        for each pair (arg1, arg2) in zip(expr1[1:],
                                    expr2[1:]):
            substitutions = unify(arg1, arg2, substitutions)
        return substitutions
    raise "Unification Error"
function unify_variable(var, expr, substitutions):
    if var == expr:
        return true
    if is_compound(expr):
        for sub in expr:
            if occurs_check(var, sub, substitutions):
                return True
    if expr in substitutions:
        return occurs_check(var, substitutions[expr],
                            substitutions)
    return false
```

Page No.
Date

1. Unify the following $P(x,a,b)$ and $P(y,z,b)$

Substitution required: $y \to x$, $z \to a$

Unified predicate is $P(x,a,b)$

2. $\forall x P(x, f(y), P(z, f(a))$

Substitution required: $z \to x$, $y \to a$

unified predicate is $P(x, f(a))$

28/11/24

3. To prove

1. Ravi enjoys a wide variety of foods

FOL: $\forall x$ Food$(x) \to$ Enjoys (Ravi, x)

CNF: $\neg$ Food$[x] \vee$ Enjoys (Ravi, x)

2. Bananas are food

FOL: Food [Bananas)

CNF: Food [Bananas)

3. Pizza is food

Food (Pizza) $\to$ FOL(CNF)

4. A food is anything that anyone consumes and isn't harmed by

FOL $\forall x (\exists y (\text{consumes}(y,x) \wedge \neg \text{HarmedBy}(y,z)) \to \text{Food}(x)$

CNF $\exists y (\text{consume}(y,x) \wedge \neg \text{HarmedBy}(y,x)) \vee \text{Food}(x)$

$\neg$ Consumes(f(x), x) $\vee$ HarmedBy(f(x), x) $\vee$ Food(x)

5. Sam eats Idli and is still alive

FOL/CNF: Consumes(Sam, Idli) $\wedge \neg$ HarmedBy(Sam, Idli)

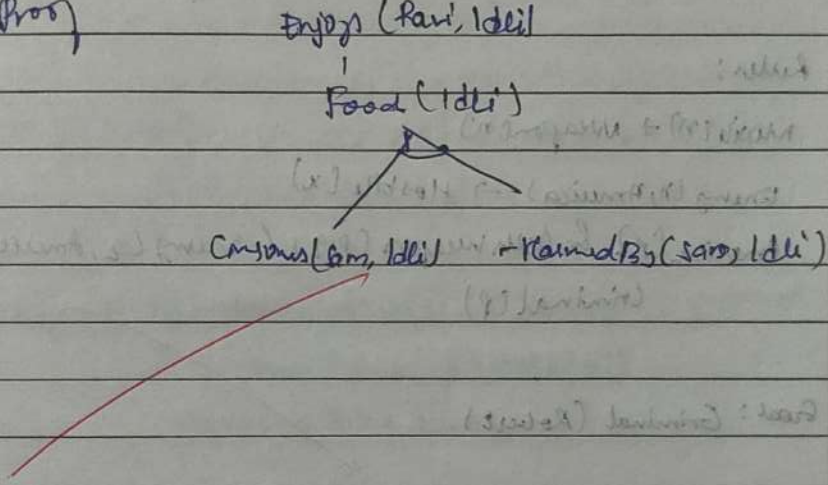
6. Bill eats everything Sam eats

FOL: $\forall x$ [Consumes(Sam, x)] $\rightarrow$ Consumes(Bill, x)

CNF: $\neg$ Consumes(Sam, x) $\vee$ Consumes(Bill, x)

Goal: Ravi likes Idli: Enjoys(Ravi, Idli)

Proof



Implementation of Forward chaining

Representation in FOL and generation of proof tree by forward chaining:

Facts:

American (Robert)

Enemy (A, America)

Owns (A, T1)

Missile (T1)

SellsWeapon (Robert, T1)

Rules:

Missile (x) → Weapon (x)

Enemy (x, America) → Hostile (x)

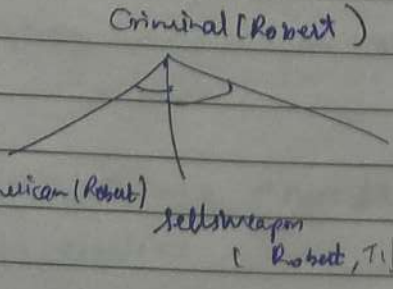
American (p) & SellsWeapon (p, q) & Enemy (q, America) → Criminal (p)

Goal: Criminal (Robert)

FOL:

$\forall p \forall q$ (American (p) ∧ SellsWeapon (p, q) ∧ Enemy (q, America) → Criminal (p))

→ Proof Tree



Thus: Criminal(Robert) is True.
Yes Robert is a Criminal.

(Implementation & Resolution:

→ Representation in FOL and generation of proof tree by resolution.

1. It's a crime for an American to sell weapons to hostile nations

$\forall p \forall q \forall r$ (American(p) $\wedge$ Weapon(q) $\wedge$ Sells(p, q, r)) : a
Hostile(r) $\Rightarrow$ Criminal(p)

- American(p) $\vee$ $\neg$ Weapon(q) $\vee$ $\neg$ Sells(p, q, r) $\vee$
$\neg$ Hostile(r) $\vee$ Criminal(p)

2. Country A has some missiles
$\exists x$ (Owns(A, x) $\wedge$ Missile(x))
Owns(x, T_1) $\wedge$ Missile(T_1)

3. All missiles were sold to country A by Robert
$\forall x$ (Missile(x) $\wedge$ Owns(A, x) $\Rightarrow$ Sells(Robert, x, A))
$\neg$ Missile(x) $\vee$ $\neg$ Owns(A, x) $\vee$ Sells(Robert, x, A)

4. Missiles are weapons
$\forall x$ (Missile(x) $\Rightarrow$ Weapon(x))
$\neg$ Missile(x) $\vee$ Weapon(x)

5. Enemy of America is Hostile
$\forall x$ (Enemy(x, America) $\Rightarrow$ Hostile(x))
$\neg$ Enemy(x, America) $\vee$ Hostile(x)

6. Robert is an American:

American(Robert)

7. Country A is an enemy of America:

Enemy(A, America)

Goal: Criminal(Robert)

Resolution: ¬Criminal(Robert)

3/12/24

Proof Tree:

¬American(p) ∨ ¬weapon(q) ∨ ¬sells(p, q, r) ∨ ¬Hostile(r) ∨ Criminal(p)

¬Criminal(Robert)

American(Robert)

¬Missile(q) ∨ ¬Owns(A, q) ∨ Sells(Robert, q, A)

American(p) ∨ ¬weapon(q) ∨ ¬sells(p, q, r) ∨ ¬Hostile(r)

¬Missile(q) ∨ ¬Owns(A, q) ∨ Missile(T1)

¬American(p) ∨ ¬weapon(q) ∨ ¬Hostile(r) ∨ ¬Missile(r) ∨ ¬Owns(A, r)

¬American(p) ∨ ¬weapon(q) ∨ ¬Hostile(r)

¬weapon(q) ∨ ¬Sells(p, q, r) ∨ ¬Hostile(r) ∨ ¬Missile(r) ∨ Weapon(r)

¬Sells(p, q, r) ∨ ¬Hostile(r) ∨ ¬Missile(r)

Missile(r)

Analyze alpha beta pruning method.

Function · alpha_beta (node, depth, alpha, beta, maximizing_player, path):

IF depth is 0 OR node is a terminal node THEN
RETURN · node's value, path

IF maximizing_player THEN
max_eval = negative infinity
Optimal_path = null
For each child of node DO
child_value, child_path = alpha_beta pruning (child, depth-1, alpha, beta, FALSE, path + child's name)
IF child_value > max_eval THEN
max_eval = child_value
Optimal_path = child_path
alpha = maximum (alpha, max_eval)
IF · beta <= alpha THEN
Break
return
else
min_eval = positive infinity
Optimal path = null
for each of node DO
child_value, Child_Path = alpha, beta pruning (child, depth-1, alpha, beta, TRUE, path + child's name)
IF · child_value < min_eval THEN
min_eval = child_value
Optimal_path = child_path
beta = minimum (beta, min_eval)

IF beta <= alpha THEN

BREAK

RETURN min_eval, optimal_path

maximizing_player = TRUE

initial_alpha = negative infinity

initial_beta = positive infinity

depth = 3

Optimal_value, optimal_path = alpha_beta(root node, depth, initial_beta, maximizing_player)

PRINT "The optimal value is" optimal value

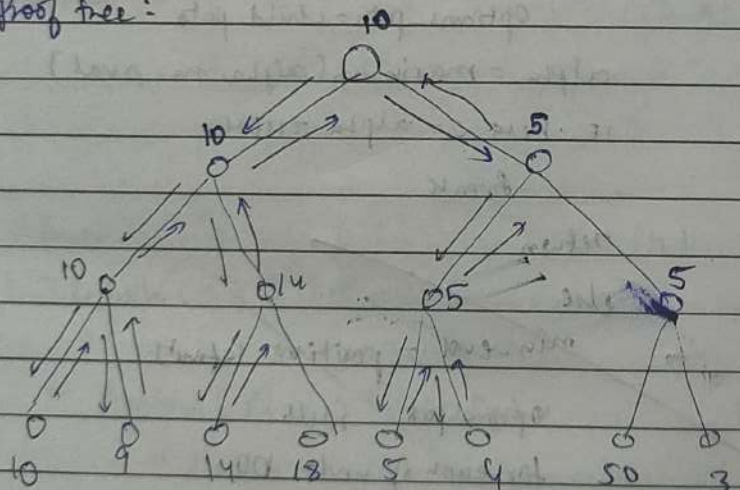
PRINT "The optimal path is", optimal path.

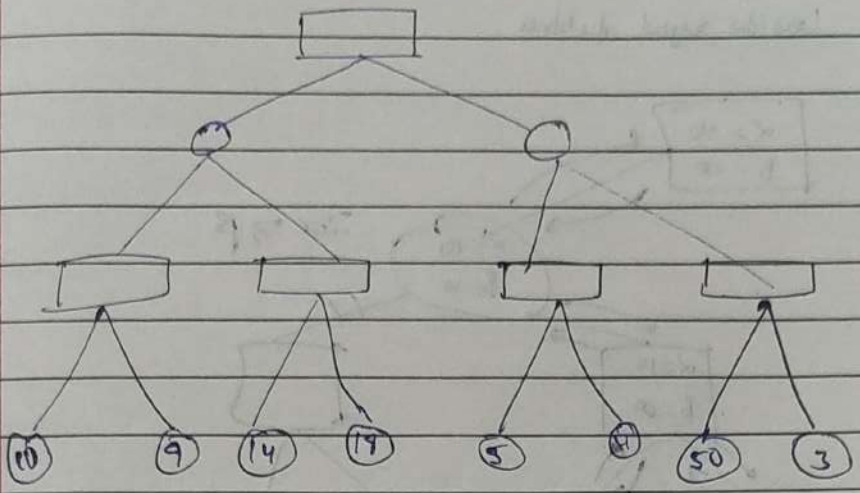
OUTPUT:

The optimal value is: 10

The optimal path is: A → B → D → H

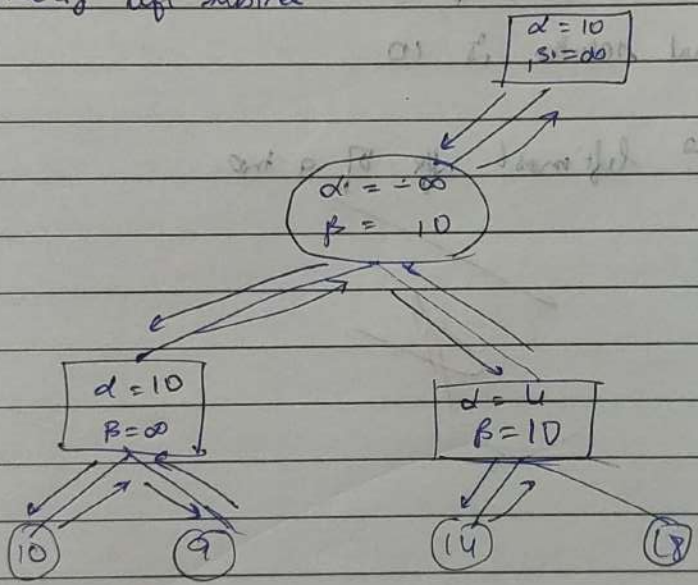
Proof tree:





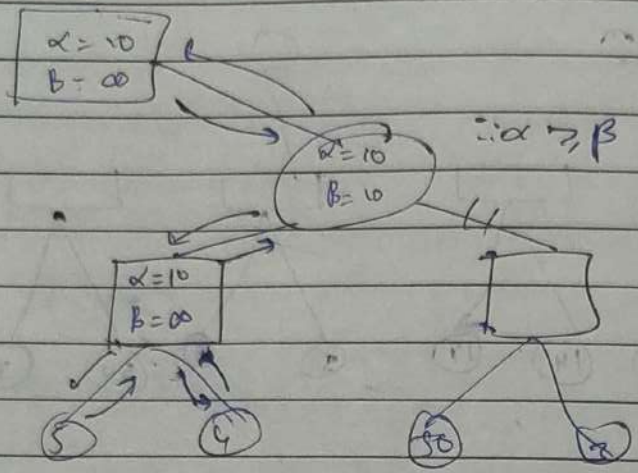
Applying DFS to left sub tree and initializing $\alpha = -\infty$, $\beta = \infty$

Considering left subtree



Date
17/12/24

Consider right subtree



Here $\alpha = 10, \beta = \infty$

Optimal solution is 10

Path → leftmost of a tree.