Observation Book

INDEX

Name Nithya Lakshmi. V Std Sec 5D

Roll No. Subject AI LAB School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	24/9/24	Tic tac toe	9	24/9
2.	1/10/24	Vaccum cleaner	10	
3.	8/10/24	8-Puzzle	10	
4.	15/10/24	IDDFS, A* algorithm 8-Puzzel	10	
5.	22/10/24	Simulated Annealing	10	
6.	29/10/24	A* search Algorithm Hill Climbing Algorithm	10	
7.	12/11/24	Entailment using literals	10	
8.	19/11/24	First order logic	10	
9.	26/11/24	Unification	10	
10.	3/12/24	Forward Chaining	10	

12/12

24/9/24 LAB-1

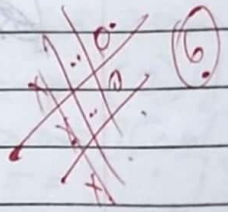
Date / /
Page

TIC TAC TOE

Algorithm

Step 1: import numpy, Random library
Step 2: Create a 3 × 3 matrix using 2-dimensional Array
Step 3: initialize the entire matrix as blank using for loop(-)hyphen
Step 4: Display a message for the user to either choose to begin or not
Step 5: Set patterns which is considered as win
Step 6: check the board if it matches any pattern after each move if yes then display win else after all blanks are filled Display tie.

Refer Algorithm



Code

```
import random

def initialize_board():
    return [[' ' for _ in range(3)]
            for _ in range(3)]

def display_board(board):
    for row in board:
        print('|'.join(row))
        print('-'*5)

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

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

    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 available_moves(board):
    return [(i,j) for i in range(3)
                  for j in range(3)
                  if board[i][j]==' ']

def check_two_in_a_row(board,
                       player):
    for row in range(3):
        if board[row].count(player)==2
           and board[row].
               count(' ')
               ==1:
            return row, board[row].index(' ')

    for col in range(3):
        if [board[row][col] for row in
            range(3)].count
            (player)==2:
            empty_index=[row for row
                in range(3) if
                board[row][col]
                ==' ']
            if empty_index:
                return empty_index[0], col

    if [board[i][i] for i in range(3)].
                count(player)==2:
        empty_index=[i for i in range(3)
                     if board[i][i]==' ']
        if empty_index:
            return empty_index[0],
                   empty_index
                   [0]
```

```
if [board[i][2-i] for i in range(3)].
                    count(player)
                    == 2:
    empty_index = [i for i in range(3)
                    if board[i]
                    [2-i]
                    == ' ']
    if empty_index:
        return empty_index[0], 2-empty
                    index
return None

def make_move(board, player, move):
    board[move[0]][move[1]] = player

def computer_move(board):

    move = check_two_in_a_row(board
                    'O')
    if move:
        make_move(board, 'O', move)
        return

    move = check available_moves(board
    if moves:
        move = random.choice(moves)
        make_move(board, 'O', move)

def user_move(board):
    while True:
        try:
            row = int(input("Enter
                    row(0-2):
                    "))
```

```
col = int(input("Enter column (0-2): "))
if board[row][col] == ' ':
    make_move(board, 'X', (row, col))
    return
else:
    print("That spot is already taken. Try again!")

except (ValueError, IndexError):
    print("Invalid input. Please enter number between 0 and 2")

def play_game():
    board = initialize_board()
    players = ['X', 'O']
    current_player = 0

for _ in range(9):
    display_board(board)
    if current_player == 0:
        user_move(board)
    else:
        computer_move(board)

winner = check_winner(board)
if winner:
    display_board(board)
    print(f"{players} {winner} wins!")
    return

current_player = 1 - current_player
```

```
display_board(board)
print("Its a draw!")

play_game()
```

output

```
  |   |
--|---|--
  |   |
--|---|--
  |   |

Enter row(0-2): 1
Enter column(0-2): 2

  |   |
--|---|--
  |   |X
--|---|--
  |   |

  |   |
--|---|--
  |   |X
--|---|--
  |0  |

Enter row(0-2): 0
Enter column(0-2): 0

X |   |
--|---|--
  |   |X
--|---|--
  |0  |

X |0  |
--|---|--
  |   |X
--|---|--
  |0  |
```

Enter row (0-2): 1
Enter column (0-2) : 1

```
 | | 
 |x|x
 |o|
```

```
x|o|
o|x|x
 |o|
```

Enter row (0-2) : 2
Enter column (0-2) : 2

```
x|o|
o|x|x
 |o|x
```

Player X wins!

24/9

LAB-2

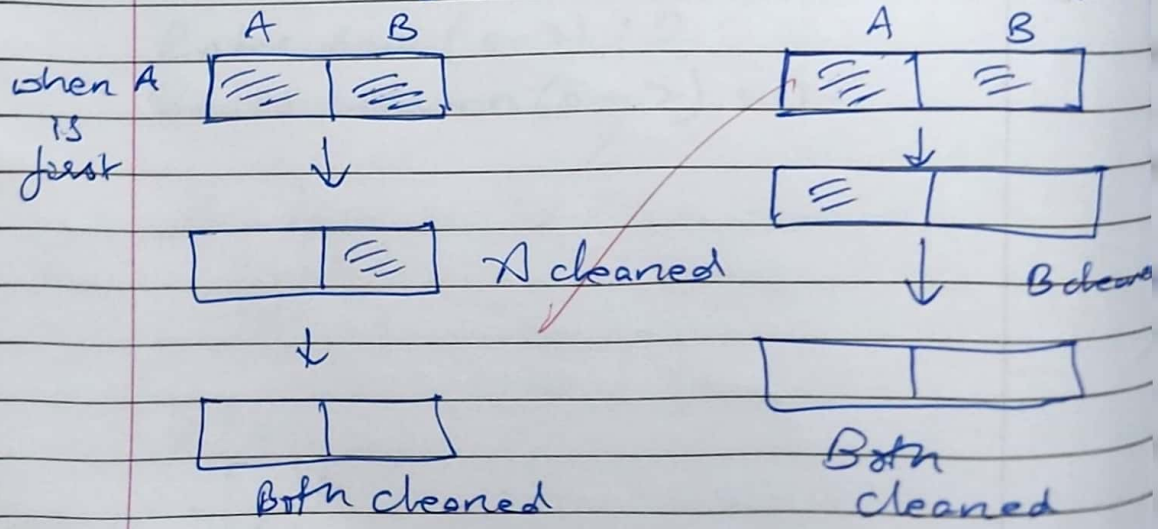
1/10/24

Vaccum Cleaner

1. Assign the no. of rooms from variables A & B to dirty

2. while (room A || B == dirty)
{
if (room A is dirty) {
if (vaccleaner is in A) {
clean A
dirty--;
else (goto B) }
if (cleaner in B) {
if (room B is dirty)
clean B
else
{ goto A }
}

if (A & B is clean)
exit



Date / /
Page

Percept Sequence

(A dirty), (A clean), (A left)
(A dirty), (A clean) (A left) B(dirty)
(A dirty), (A clean) (A left) (B dirty)
(B clean)
(A dirty), (A clean), (A left), (B dirty) (B clean)
(B right), (A clean)
(Exit)

Code

```
class VaccumCleaner:
def __init__(self, grid):
    self.grid = grid
    self.position = (0,0)

def clean(self):
 x, y = self.position
 if self.grid[x][y] == 1:
    printf" cleaning position {self.position
      self.grid[x][y] = 0                    ")
  else:
    print(f" Position {self.position} is already
                                    clean")

def move(self, direction):
    x, y = self.position
    if direction == 'up' and x > 0:
        self.position = (x-1, y)
    def direction == 'down' and x <
                                len(self.
                                  grid)
```

-1:

```
            self.position = (x+1, y)
        elif direction == 'left' and y >0:
            self.position = (x, y-1)
        elif direction == 'right' and y <
    len (self.grid[0])-1:
            self.position = (x, y+1)
        else:
            print("Move not possible")
        else:
            print("Move not
```

```
    def run(self):
        rows = len(self.grid)
        cols = len(self.grid[0])

        for i in range(rows):
            for j in range(cols):
                self.position = (i, j)
                self.clean()

        print("Final grid state:")
        for row in self.grid:
            print(row)

    def get_dirty_coordinates(rows, cols,
                              num_dirty_cells):
        dirty_cells = set()
        while len(dirty_cells) < num_dirty_
                                  cells:
            try:
                coords = input(f"Enter
                    coordinates for
                    dirty cell {len
                    (dirty_cells)
```

```
+1}) (format: row, col):
")
x, y = map(int, coord.split(','))
if 0 <= x < rows and 0 <= y < cols:
    dirty_cells.add((x, y))
else:
    print("Coordinates are out of bounds try again")
except ValueError:
    print("Invalid input. Please enter coordinates in the format: row, col")
return dirty_cells

rows = int(input("Enter the number of rows: "))
cols = int(input("Enter the number of columns: "))

num_dirty_cells = int(input("Enter the number of dirty cells:"))

if num_dirty_cells > rows * cols:
    print("Number of dirty cells exceeds total cells in the grid. Adjusting to maximum")
    num_dirty_cells = rows * cols

initial_grid = [[0 for _ in range(cols)] for _ in range(rows)]
```

```
dirty_coordinates = get_dirty_coordinates
                    (rows, cols, num_
                                 dirty_
                                 cells
for x, y in dirty_coordinates:
    initial_grid[x][y] = 1

vaccum = VacuumCleaner(initial_grid
print("Initial grid state:")
for row in initial_grid:
    print(row)

vaccum.run()
```

output

```
Enter rows : 2
Enter cols : 2
Enter dirty cells : 4
Enter coordinates for 1 : 0,0
Enter coordinates for 2 : 0,1
    "       "      for 3 : 1,1
    "       "      for 4 : 1,0

Initial grid
[1, 1]
[1, 1]

cleaning pos (0,0)
    "     " (0,1)
    "     " (1,0)
    "     " (1,1)
```

Final grid

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

8/10/24 LAB-3

Date / /
Page

8 puzzle Game

8 puzzle using DFS and Manhattan distance

1	7	4
5		2
3	8	6

1	2	3
4	5	6
7	8	

Original State

1. ~~Initialize~~ Set the initial state of the puzzle.
2. Create stack to store nodes to be explored.
3. Calculate the Manhattan distance for the initial state.
4. add the ~~ini~~ manhattan distance to the stack

DFS

```
Start_state = [. . . .]
goal_state = [. . . .]

stack = push(start_state)
visited_set = { }
moves = 0
if(i, j)
{
    visited_set = add(current_state)
    if(current state == goal_state)
        return moves
```

if (not in visited set)
{
left = f(i, j-1)
right = f(i, j+1)
up = f(i-1, j)
down = f(i+1, j)
}
}
print (moves)

The Manhattan distance is used as a heuristic to estimate the distance to the goal state.

Solve using DFS

Code

```
def manhattan(puzzle, goal):
    dist = 0
    for i in range(9):
        if puzzle[i] != 0:
            goal_idx = goal.index(puzzle[i])
            dist += abs(i // 3 - goal_idx // 3) + abs(i % 3 - goal_idx % 3)
    return dist

def dfs_manhattan(puzzle, goal, visited, path):
    if puzzle == goal:
        return path
    visited.add(tuple(puzzle))
```

```
idx = puzzle.index(0)
moves = [(1, 3), (-1, 3), (3, 1), (-3, 1)]
next_states = []
for move, cond in moves:
    new_idx = idx + move
    if 0 <= new_idx < 9 and (new_idx
                             // 3 == idx):
        new_puzzle = puzzle[:]
        new_puzzle[idx], new_puzzle
                         [new_idx] =
                         new_puzzle[new_
                         idx],
                         new_puzzle[idx]
        if tuple(new_puzzle) not in visited:
            next_states.append((new_puzzle,
                                manhattan(
                                new_puzzle,
                                goal)))
next_states.sort(key = lambda x: x[1])
for state, _ in next_states:
    res = dfs_manhattan(state, goal,
                        visited,
                        path +
                        [state])
    if res:
        return res
return None

def prettify(res):
    i = 0
    for j in range(3):
        for i in range(3):
            print(res[i], end = " ")
            i = i + 1
        print("\n")
```

start = [1, 2, 3, 4, 0, 5, 6, 7, 8]
goal = [0, 1, 2, 3, 4, 5, 6, 7, 8]
result = dfs_manhattan(start, goal, set(), [start])

```
for i in result:
    prettify(i)
    print("....")
```

output

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

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

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

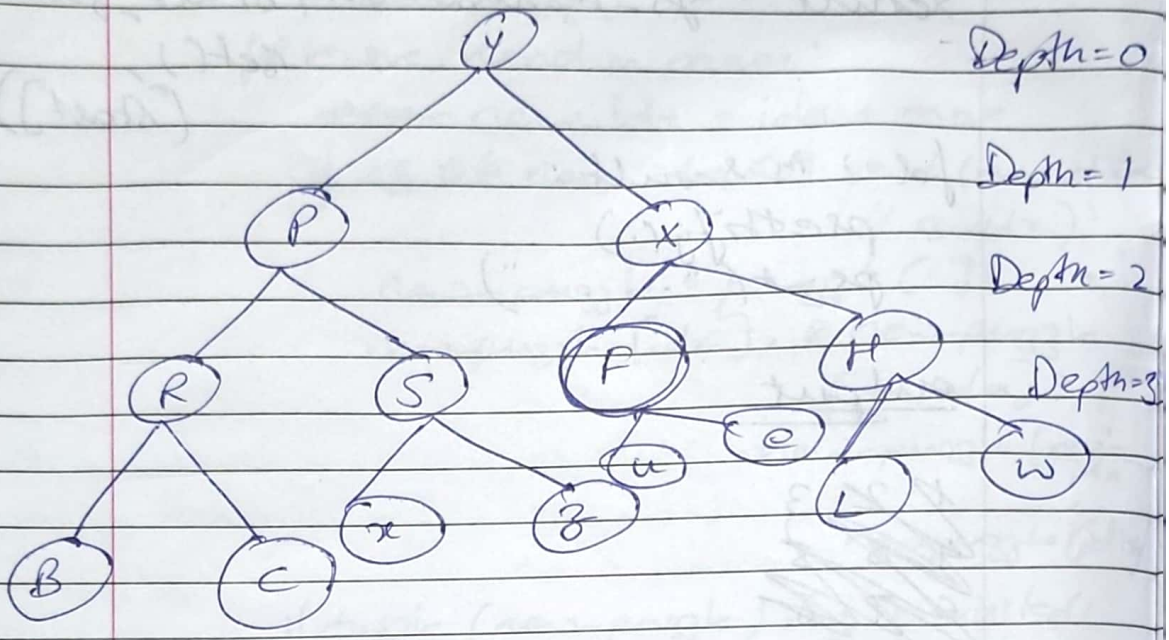
1 2 0
3 5 4
7 6 8
- - - -

0 1 2
3 4 5
6 7 8

15/10/24 LAB-4

Date __/__/__
Page

1) Iterative deepening search



I st Iteration

Depth = 0
visited nodes = Y

goal node not found

II nd iteration

Depth = 1
visited Nodes = Y P X

goal node not found

III rd iteration

Depth = 2
visited nodes = Y P R S X

END

goal node found

IV th iteration

Depth = 2
visited nodes = Y P R S X F

Date / /
Page

2) 8-Puzzle A* Algorithm

Initial State

1	2	3
8		4
7	6	5

goal state

2	8	1
	4	3
7	6	5

Heuristic values

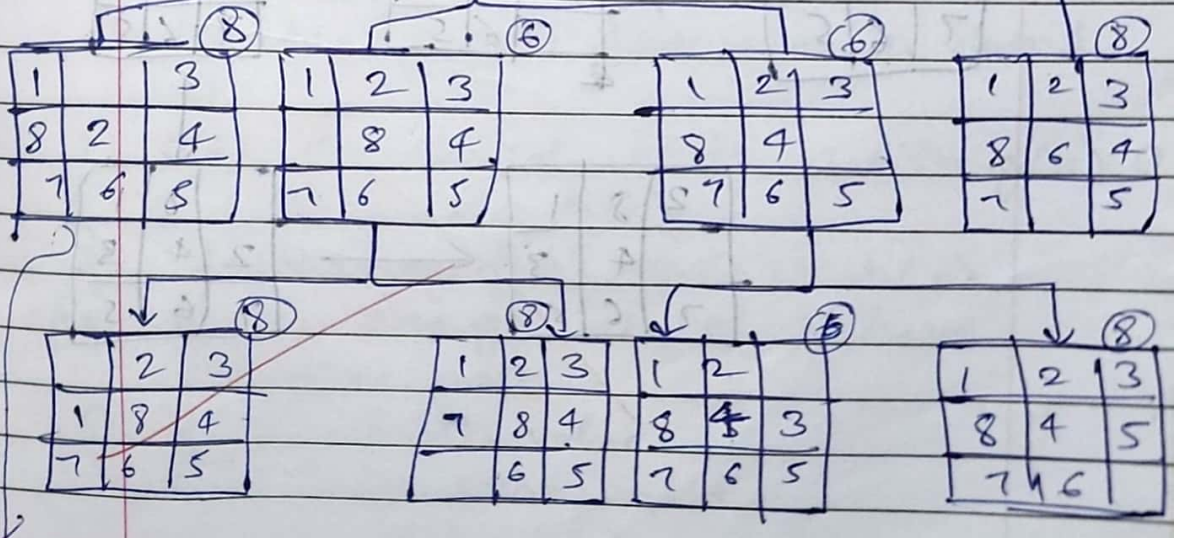
for 1 = 2	1 = 2
for 2 = 1	2 = 2
for 3 = 1	3 = 0
for 4 = 1	4 = 0
for 5 = 0	5 = 1
for 6 = 0	6 = 0
for 7 = 0	7 = 0
for 8 = 2	8 = 2
(7)	

2 2
1 2
2

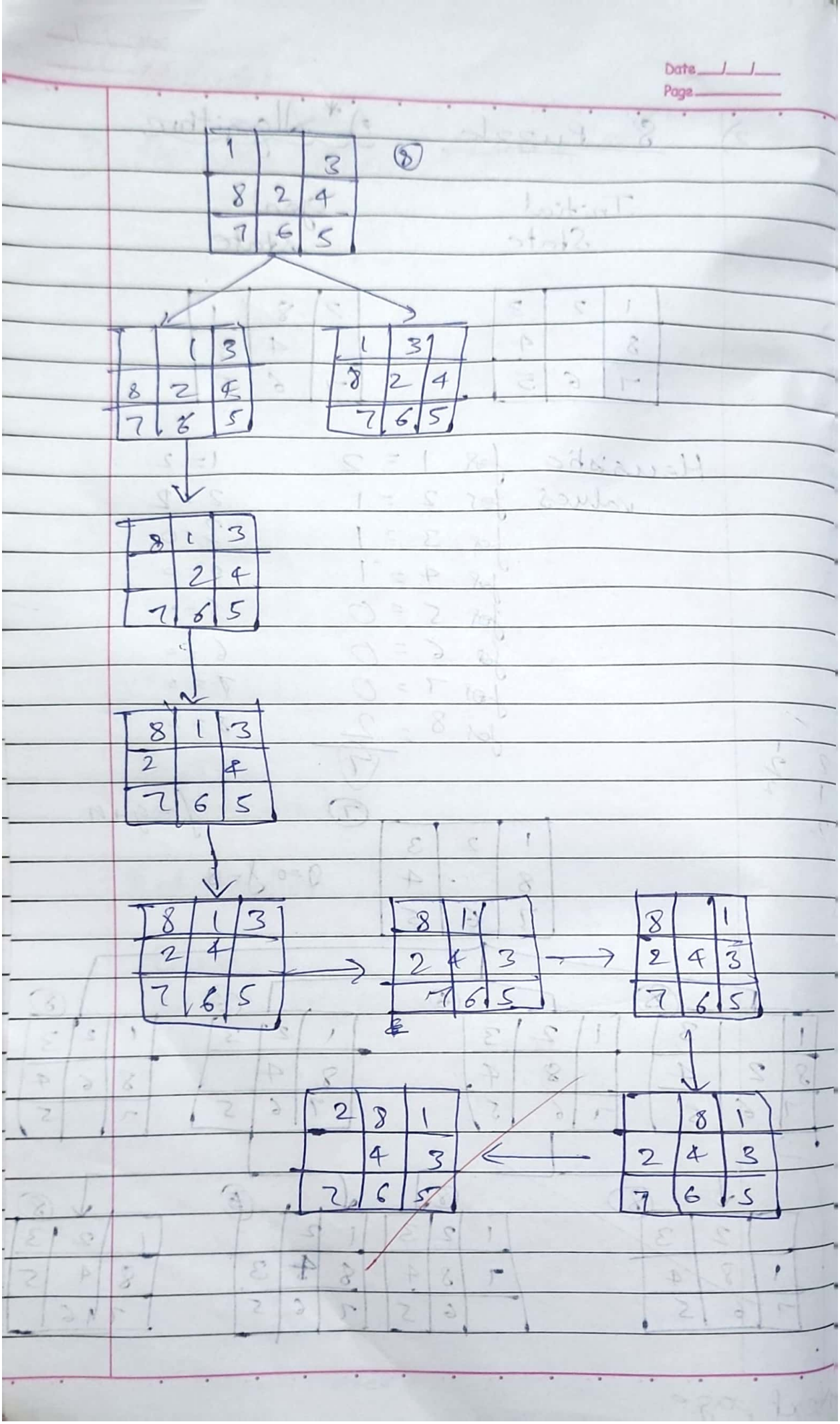
(7) f = g + h

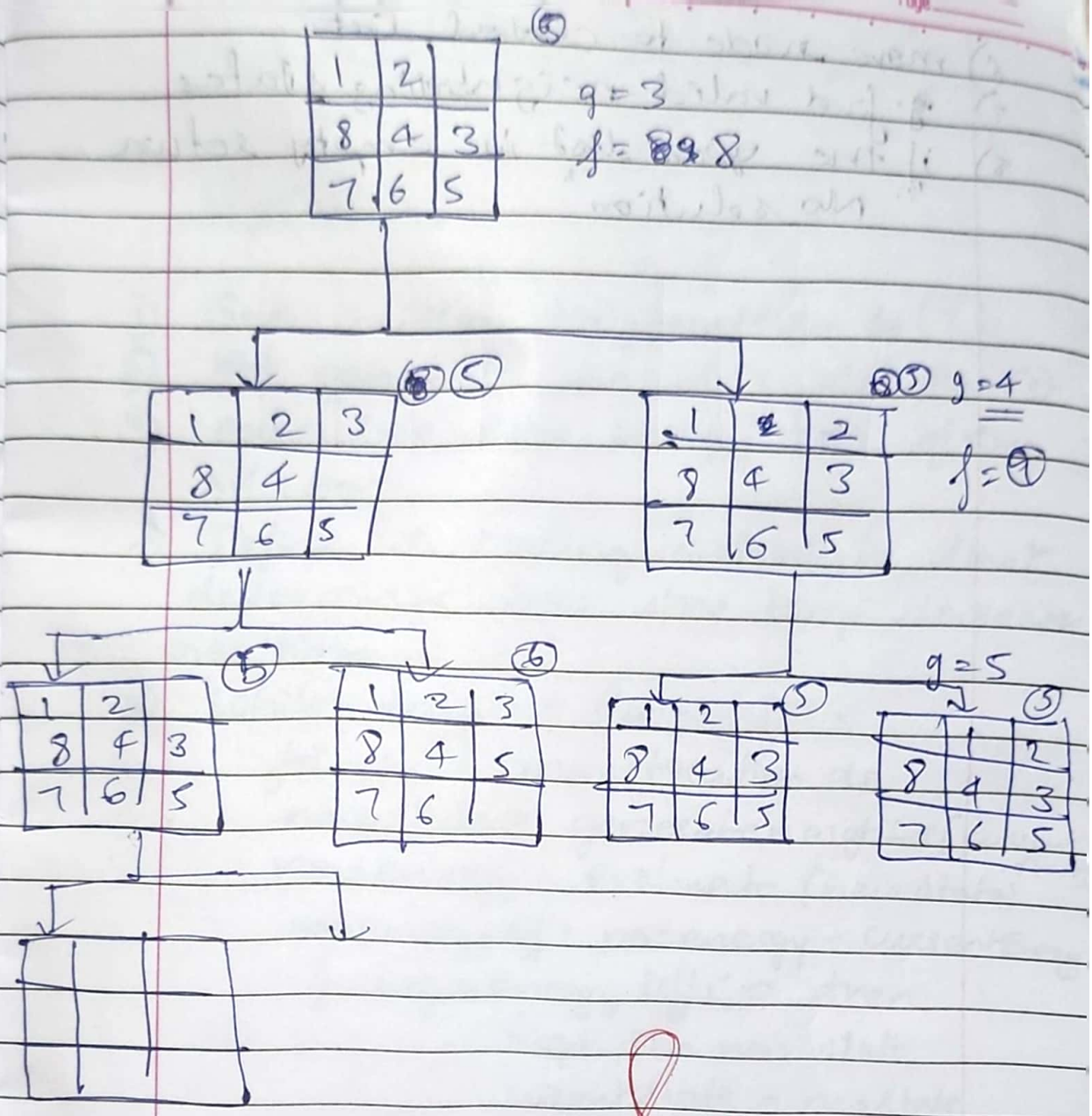
1	2	3
8		4
7	6	5

g = 0 f = 8



Next page





15/10

A* Algorithm

1) Create an open list and a closed list
2) Add the initial state with $g(n)=0$ and heuristic $h(n)$
3) Lowest of $f(n) = g(n) + h(n)$
4) If it's the goal state return solution path
5) $g(n)$ = depth value
   $h(n)$ = heuristic value
   $f(n) = g(n) + h(n)$

Date __/__/__
Page _____

6) move node to closed list
7) find valid neighboring states
8) if the open list is empty return "No solution"

Date / / Page

22/10/24 LAB-5

Simulated Annealing

Algorithm

1) Set initial temperature (T)
2) Set generate an initial solution (s)
3) calculate the energy (E) of the solution
4) Define a cooling schedule that determines how the temp decreases over time
5) While temp > 0 do
   for 1 to maxiteration do
   newstate = generateneighbor(current state)
   newenergy = Evaluate (newState)
   energy diff = newenergy - currentEnergy
   if energy Diff < 0 then
   Accept the new state
   currentstate = newstate
   currentEnergy = newenergy
   If currentEnergy < BestEnergy then
   1. bestState = currentState
   bestEnergy = currentEnergy
   temp = temp * coolingRate
   1. Accept worse.
6) Return BestState

22/10/24

Date __/__/__
Page _____

output

Enter the initial state (Starting point) = 10
Enter the initial temperature = 12
" " Cooling rate (b/w 0 to 1) = 0.2
" " number of iterations = 2.5
i2 : CS = 9.27, CC = 85.99 temp = 240
i2 : CS = 9.25, CC = 85.6, temp = 0.480

i 25 = CS = 3.5, CC = 12083, temp = 0.00
Best State : 3.58 Best energy = 12.83

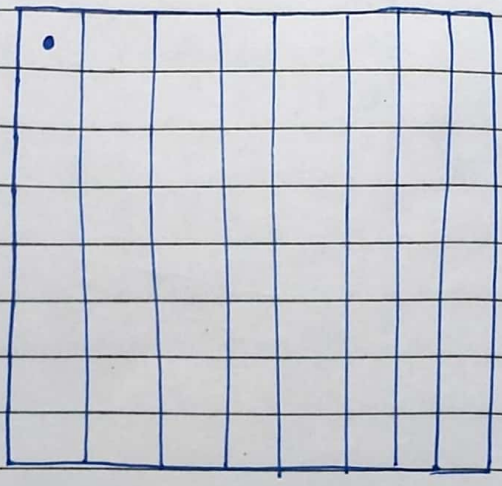
22/10/24

29/10/24 LAB-6

Date __/__/__
Page

A* Search Algorithm for 8 Queens

1) Start with an 8×8 empty matrix
2) Start adding the queens one by one
3) generate all possible states by placing a queen on the next empty row in a non-attacking column f=g+h
4) keep the list of boards set up that we need to explore
5) Calculate the score of each step
6) g is the no. of queens placed, h is no. of conflicts.
7) choose the best option, choose the step with lowest score to continue
8) Goal check() if setup has no conflicts and all the queens are placed then it is a solution

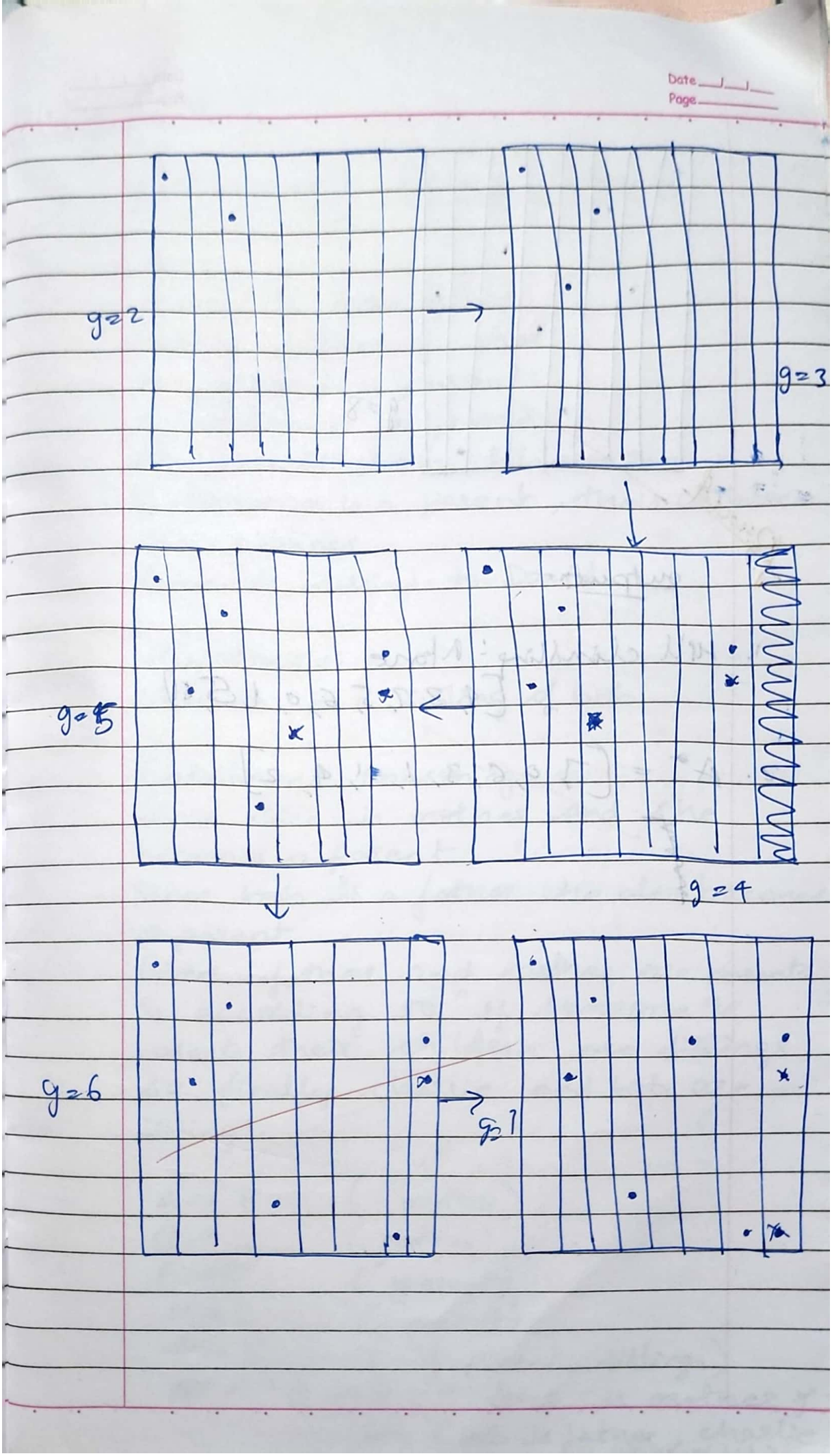


g=1

Date __/__/__
Page

Hill Climbing Algorithm for 8 queens

1. Start by placing the queen randomly
2. calculate heuristic for the current state
3. If the value is 0, then the solution is found.
4. generate the neighbouring state for each queen by moving it to every possible state and calculate the heuristic for each new state
5. choose the best neighbour, select the neighbour with lowest heuristic. If no neighbour state ~~includes~~ improves the current state, terminate
6. Repeat to find the goal solution


g=2
g=3
g=5
g=4
g=6
g=7

Output:-

1. Hill climbing: None
   - [4, 2, 7, 5, 6, 0, 1, 5, 1]

2. A* = [7, 0, 6, 3, 1, -1, 4, 2]

12/11/24 [LAB-7]

Entailment using literals

KB:
Alice is mom of bob
bob is father of charlie
A father is a parent
A mother is a parent
All parents have children
If someone is a parent, their children are siblings.
Alice is married to David

Hypothesis:
Charlie is a sibling of bob

Entailment Reasoning:
Since Alice is mother and she becomes a parent
Since bob is a father he also becomes a parent
Given, father and mother are parents & according to "If someone is parents their children are siblings, so finally charlie and bob are siblings.

$A \rightarrow B$ (mother)
$B \rightarrow C$ (father)
$F \rightarrow D$ (parent)
$M \rightarrow D$ (parent)
$P \rightarrow S$ (if parent, siblings)
$A \wedge B \rightarrow Q$ (if Alice is mother of bob & bob is father, charlie

Date / /
Page

check for entailment:

1. If A (Alice is mother of Bob) is true then B (Bob is the father of charlie) must also be true (A → B)

2. If B is true then C (Bob is a parent) must be true (F → P) and M (Alice is a parent) must also be true (M → P)

3. If both Alice & charlie are parents (i.e. M & F are true) than S (their children are siblings) must be true (P → S)

4. Since S is true, the hypothesis Q ("charlie is sibling of Bob") is true

Conclusion:
Using Propositional logic, we can conclude the hypothesis "Charlie is a sibling of Bob" is entailed by KB.

output

The hypothesis 'charlie is a sibling of Bob' is TRUE

19/11/24 LAB-8

First-order logic (FOL)

Scenario:

"All dogs are mamamals. Fido is a dog. Therefore, Fido is a mammal"

FOL Representation:

1. Define the predicates:

Dog(x) : x is a dog
Mammal(x) : x is a mammal

2. Define the constant:

fido : a specific dog

3. Represent the statements in FOL:

$\forall x$ (Dog(x) → Mammal(x))

// All dogs are mammals

Dog(fido) // fido is a dog

Conclusion:

Mammal(fido)

output

// fido is a mammal

19/11/24

26/11/24

LAB - 9

Unification

Algorithm

1. Check for a base case:
   If both expressions are identical, return an empty substitution.

2. Check for a variable case:
   If one expression is a variable.
   — If the variable is already bound to another term, recursively unify the bound term with the other expression.
   — Otherwise, create a new building between the variable and the other expression.

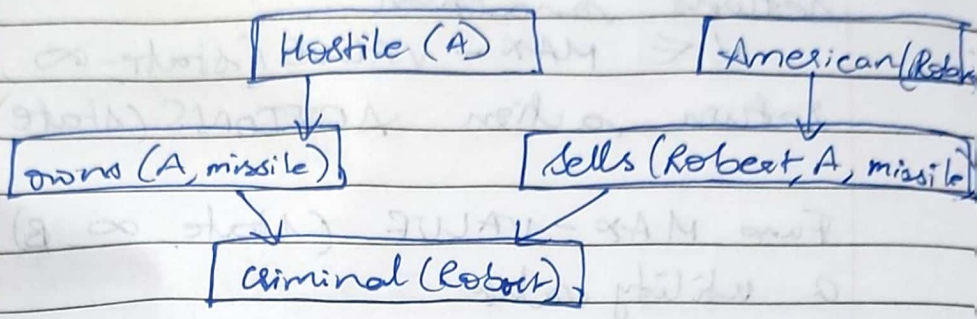
3. Check for a Constant Case:
   If both expressions are constants they must be identical. If not, fai

4. Check for a Complex term case:
   If the main functions or operations of both expressions are different fail.

3/12/24

FORWARD CHAINING

Proof:



⇒ American (a) ∧ Weapon (b) ∧ Sells ∧ Hostile (a, b, c) (d)

∀ a, owns (A, a) ∧ missile (a)

∀ a, Weapons (a) ∧ owns (A, a) ⇒ Sells (Rob, a, A)

Missile (a) ⇒ weapons (a)

∀a Enemy (a, America) ⇒ Hostile

American (Robert)

Enemy (A, America)

Criminal (Robert)

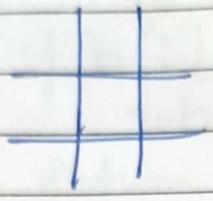
Date / /
Page

Alpha-beta Search

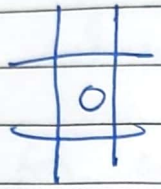
```
func, ALPHA-BETA-SEARCH (state)
returns an action
    v ← MAX-VALUE (state, -∞, +∞)
return an action ACTIONS (state) with v

Func MAX-VALUE (state, ∞, β) return
a utility value
  if TERMINAL-TEST (state) then
return UTILITY (state)
    v ← -∞
for each α in ACTION (state) do
  v ← MAX (α, v)
    return v

Func MIN-VALUE (state, α, β) return
α utility if TERMINAL-TEST (state)
then return UTILITY
    v ← +∞
for each α in ACTIONS (state) do
  v ← MIN (v, MAX-VALUE (RESULT (s, a),
  if v ≤ α then return v                α, β))
      β ← MIN (β, v)
      return v
```

output

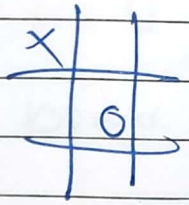
You are O, comp is X



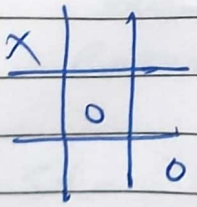
Enter move : 2, 2



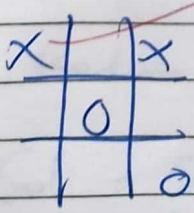
Comp played



Enter move : 3, 3



Comp played



Enter move : 1, 2

X	O	X
	O	
		O

Comp played

X	O	X
	O	
	X	O

Enter move : 3, 1

X	O	X
	O	
O	X	O

Comp played

X	O	X
X	O	
O	X	O

Enter move : 2, 3

X	O	X
X	O	O
O	X	O

Draw

N-Queens

```
def is_safe(board, row, col):
    i to 0, to row - 1:
    if b[i] = i wl & abs(b[i] - j
        = abs(i-
        return False
    Return True

def alpha_beta(board, row, α, β):
    if row == 8:
    return True

    for wl 0 to 7:
    is_safe(board, row, wt):
        board[row] = w
        if α-β(b, w wt+1, α, β):
            Return True
    board[row] = -1

alpha . max(alpha, row)
    if alpha S = beta:
    break
    Return False

Solve_8_queens():
Initialise(-1, -1, -1, -1, -1, -1, -1, -1)
if alpha_beta(board, 0, -∞, ∞):
    Return board
Return None

Sol = Solve_8_queens()
if sol is Not None:
    Print sol
    else No sol
```

Date __/__/__
Page _____

Output:

Q - - - - - - -
- - - - Q - - -
- - - - - - - Q
- - Q - - Q - -
- - - - - - - -
- Q - - - - Q -
- - - Q - - - -

3/12/24