INDEX

Name Rohit. R. Gandhi

Standard I Section D Roll No. 147

Subject AI LAB

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1	24/09/24	TIC-TAC-TOE		24/9/24 (10)
2	1/10/24	Vacume Cleaner		1/10/24 (10)
3	8/10/24	8 Puzzle		8/10/24 (10)
4	15/10/24	8 Puzzle / IDF		15/10/24 (10)
5	22/10/24	Annealing Algorithm		(9)
6	29/10/24	8 Queens		(10)
7	12/10/24	Entailments		19/11/24 (10)
8	19/11/24	Literals		19/11/24 (10)
9	3/12/24	FOL forward chaining		(10)
	3/12/24	TIC-TAC-TOE - Min Max		3/12/24
10	3/12/24	8-Queens - Alpha-beta pruning		

Bafna Gold
Date: Page:

24/8/24

Tic - Tac - Toe

1) Step 1:- Make an array of 9 blocks:-

2) step 2:- Let CPU or Human make any random move one by one

3) Step 3:- for second step choose the nearest free box from the last entered value:

4) step 4:- check for computer:-
if any (n mod 3) == same values place the 3rd value from the CPU to the next position. [for row]

5) step 5:- transpose the matrix of array values and then check for the columns using step 4

6) step 6:- if all the n mod 3 matches connect the points and declare that player as winner.

7) step 7:- check for diagonal elements
if [block 1 = 5 = 9] mark win
if [block 3 = 5 = 7] mark win

8) step 8:- Stop

24/9/24

Program

```
board = [str(i) for i in range(1,10)]

def display_board():
    print(f" {board[0]} | {board[1]} |
                {board[2]} ")
    print("---+---+---")
    print(f" {board[3]} | {board[4]} |
                {board[5]} ")
    print("---+---+---")
    print(f" {board[6]} | {board[7]} |
                {board[8]} ")

def check_winner(player):
    win_combinations = [
        [0,1,2]
        [3,4,5]
        [6,7,8]
        [0,3,6]
        [1,4,7]
        [2,5,8]
        [0,4,8]
        [2,4,6]
    ]
    for combination in win_combinations:
        if all(board[i] == player for i in
               combination):
            return True
    return False
```

```
def user_move():
    while True:
        try:
            position = int(input("Enter your move (1-9): "))
            if position in range(1,10) &
                break
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Please enter a valid no.")

def computer_move():

    empty_cells = [i for i in range(1,10) if
                   board[i-1] not in ["X","O"]]
    if not empty_cells;


def play_game():
    current_player = random.choice["user","comp"]
    print(f"{current_player.capitalize()} plays
          first!")

    for i in range(9):
        display_board()

        if current_player == 'user':
            user_move()
        if check_winner("X")
            display_board()
            print("User wins!")
            return
```

```
else:
    computer_move()

    display_board()
    print("It's a tie")

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

Output :-

User plays first

Enter your move (1-9) : 5

```
1  2  3
4  X  6
7  8  9
```

Computers move : 8

```
1  2  3
4  X  6
7  O  9
```

Enter your move : 1

```
X  2  3
4  X  6
7  O  9
```

Computers move : 9

```
X  2  3
4  X  6
7  O  O
```

Bafna Gold
Date: Page:

Enter your move (1-9): 7

x 2 3
4 x 6
x o o

Computer move: 4

x 2 3
o x 6
x o o

Enter your move: 3

x 2 x
o x 6
x o o

player x wins //

Dm 24/9/w

1/10/24 Vacuum Cleaner & Agent

+ Algorithm

step 1:- Initialize Grid [rows] [columns] with with "clean and Dirty".

set vacume position to [0,0]
set cleaned cells to 0

step 2: Display the grid status

While all cells are not "clean"
If current cell is "dirty":
set current cell to "clean"
and increment clean cells to +1

step 3:- Move a vacume cleaner to next position (Right, Down, left, up).

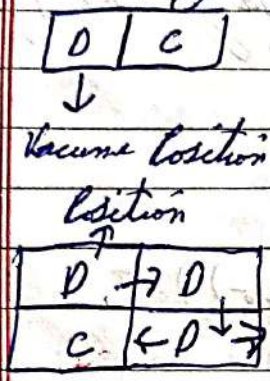
step 4: check the grid status if all the cells are "clean".
if clean:
set the vacume position to [0,0]
exit the program.

step 5:- if all the grid are not clean
go to step 2.

step 6:- if vacume initial position is [x,x] set

to (0,0).

+ Positioning program:-



→ check if: initial cell is clean is not clean and move to right. (0,2).

→ check if:
cell is clean & dirty:
if dirty:
set to "clean"
move at right:
check if clean or dirty
if dirty:
set to clean:
if right = NULL
go left
check: cells:
if dirty:
set to clean:
and move left:

Proceed

Program :-

```
def initialize_grid(m, m) [rows, columns
    return [["dirty" for i in range(m)] for i in
            range(m)]

def print_grid(grid):
    for row in grid:
        print(" ", join(row))
        print()

def is_grid_clean(grid)
    for row in self.grid
        print(" ", join(row)
    print()

def is_grid_clean(self):
    return all("dirty" not in row for row in
        self.grid)

def clean(self):
    for i in range(self.m)
        for j in range(self.m)
            self.vacuum(i,j)

    print("All cells are clean")

def vacuum(self, i, j)
    print(f"vacuum at position [{i}, {j}]")
    if [i][j] == "dirty":
        print(f"clean cell [{i}, {j}]")
            self.grid[i][j] = 'clean'
```

```
if __name__ == "__main__":
    m, n = 1, 2 / 2, 2
    vacuum_cleaner = VacuumCleaner(m, n)
    print("Initial Grid:")
    vacuum_cleaner.print_grid()
    vacuum_cleaner.print_grid()
    vacuum_cleaner.clean()
```

output:
Dirty / Dirty

Vacuum at position (0,0)
clean cell (0,0)
clean / Dirty

Vacuum at position (0,1)
clean cell (0,1)
clean / clean

All cells are clean!

11/10/24

8-Puzzles (DFS)

+ Algorithm :

```
def DFS_8Puzzle (initalize_state, goal_st
stack = [(initial_state, 0)]
visited = set()

while stack :

    current_state, depth = stack.pop()
    if current_state == goal_state
        return goal state achieved

def find_blank_tile (state):
    return state.index (0)

def swap (state, i, j)
    new_state = state [:]
    new state [i], new state [j] = merge
    return new state

def get_neighbours (state)
    neighbour = []
    index = find_blank_tile (state)
    row, col = index 113

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

in moves:
new_row, new_col = row + move(0), col + move(1)

if 0 <= new_row < 3 and 0 <= new_col

new_state = swap(state, index, new_state)
neighbours.append(new_state)
neighbours

if current_state == goal_state
    return path + (current_state)

for neighbour in get_neighbours
    if (tuple neighbour) not in visited:
        stack.append((neighbour, path + [current]))

if solution != No_Solution:
    print(puzzle(state))
else
    print(solution)

Exp:-

1	3	8
4	6	5
2	7	X

1	2	3
4	5	6
7	8	

Proved

1	3	8
4	6	5
2	X	7

→

1	3	8
4	X	5
2	6	7

→

1	3	8
X	2	5
6	7	4

→

1	2	8
7	6	5
4	7	

Manhattan Distance

Algo:-

1) Initialize the search:

Define initial and final state

Create a priority queue for the A* algorithm and set to track visited state.

Push the initial states into priority queue with movement cost as 0.

2 Calculate Manhattan Distance: [f(n) = g(n) + h(n)]

for a given state calculate the sum of manhattan distance of all tiles.

Pop the value from the lowest priority queue

If the goal path is reached return the solution path.

Or generate all possible valid moves.

4) Backtrack and looping loops.
Keep track of visited state to avoid cycle.

5) Final results:- If the priority queue is empty no solution is found.
if the goal is reached return solution path.

1	2	3
4	5	6
7	8	X

1	X	3
4	2	5
6	7	8

1	2	3
4	X	3
6	7	8

1	2	3
4	5	6
7	8	X

Backtracking

2
3
4
5
6
7
3
4
5

Program:

```
from collection import deque

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

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

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i, goal_j = divmod(state[i][j]-1, 3)
                distance += abs(i-goal_i) + abs(j-goal_j)
    return distance

def is_goal_state(state):
    return state == goal_state

def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
initial_state = [
    [4,1,3],
    [7,2,6],
    [5,8,0]]
```

```
if path:
    print("Solution found")
    for state in path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found")
```

output:-

$$\begin{bmatrix} 4,1,3 \\ 7,2,6 \\ 5,8,0 \end{bmatrix} \rightarrow \begin{bmatrix} 4,1,3 \\ 7,2,6 \\ 5,0,8 \end{bmatrix} \rightarrow \begin{bmatrix} 4,1,3 \\ 7,2,6 \\ 0,5,8 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 4,1,3 \\ 0,2,6 \\ 7,5,8 \end{bmatrix} \rightarrow \begin{bmatrix} 0,1,3 \\ 4,2,6 \\ 7,5,8 \end{bmatrix} \rightarrow \begin{bmatrix} 1,0,3 \\ 4,2,6 \\ 7,5,8 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 1,2,3 \\ 4,5,6 \\ 7,0,8 \end{bmatrix} \rightarrow \begin{bmatrix} 1,2,3 \\ 4,5,6 \\ 7,8,0 \end{bmatrix}$$

No of moves taken to place each value:

Value 1: 9 moves

Value 2: 9 moves

Value 3: 7 moves

Value 4: 9 moves

Value 5: 3 moves

Value 6: 8 moves

Value 7: 6 moves

Value 8: 4 moves

15/10/24

LAB-4

+ 8 puzzle using A*

1	8	2				2	8	1
3	X	4				X	4	3
7	6	5				7	6	5

Initial state Final state

+ Algorithm:

+ Step 1:- create a 9 valued of 3×3 tuple.

step 2:- mark the empty step as 0.

step 3:- open list and closed list [creation] open list will store all the possible current moves and closed list will store all the moves that are not available.

step 4:- find the 0 position calculate f(n) and g(n) for all the possible moves.

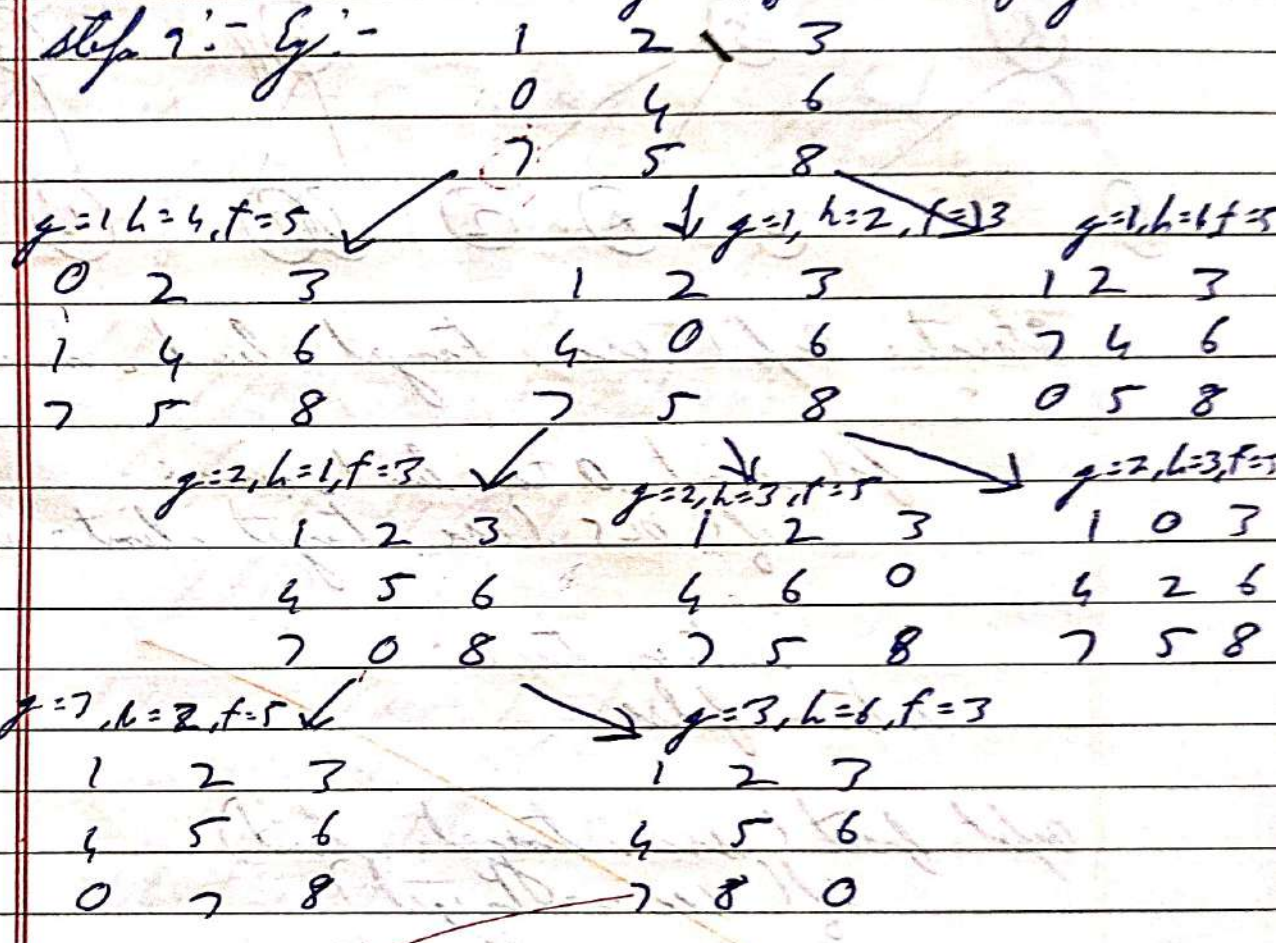
step 5:- calculate the initial state terms using f(n) + g(n) = T(n)

step 6:- choose the least T(n) and perform the available moves.

step 7:- go to step # 3 again and calculate the next position and add the data in open list() and close list()

step 8:- perform these operations till we go to the final

step 9:- Eg:-

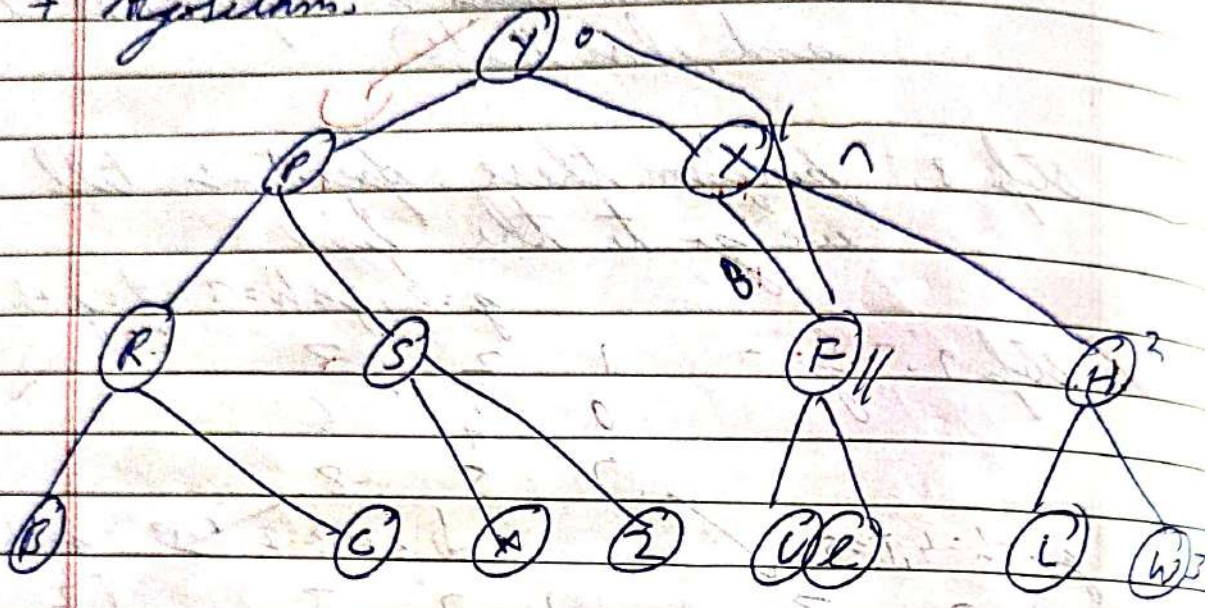


final destination

1	2	3
4	5	6
7	8	0

IDS

→ Algorithm:-



Iterative dep (src , target , limit_max)
{
for limit 0 to limit_max :
if(DLS .(src , target , limit :: true)
return true.
return false.

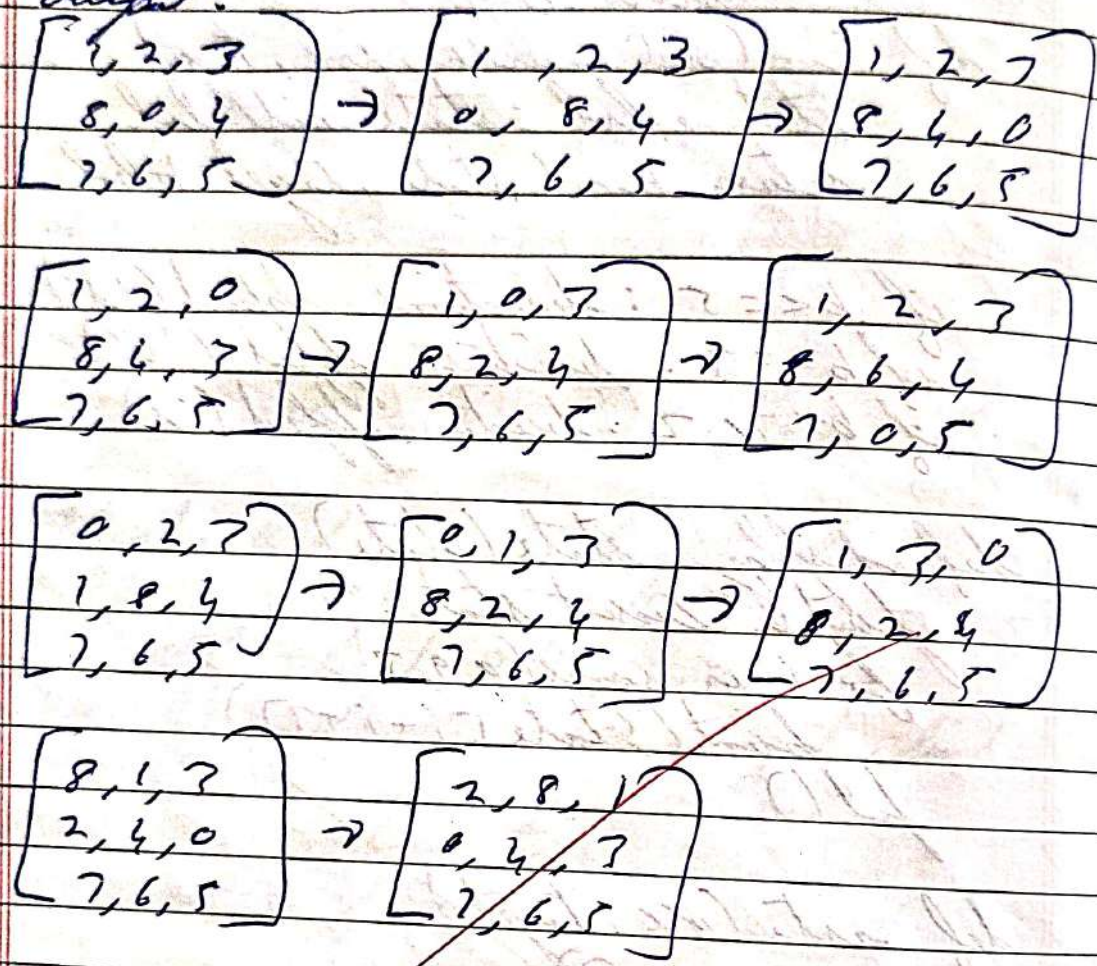
dept-first (src , target , limit)
if(src == target)
return true
if (limit <= 0) // if max depth reached
return false
for each i adjacent of src
if dept-first (i, target , limit-1)
return false.

8 Puzzle

```
def h_n(state, target):
    return sum(1 for x,y in zip(state, target) if x!=y)
def f_n(state_with_lvl, lvl, target):
    state, lvl = state_with_lvl
    return h_n(state, target) + lvl

if b<=5: directions.append('d')
if b>=3: directions.append('u')
if b%3!=2: directions.append('r')

def display_state(state):
    print("current state:")
    for i in range(0,9,3):
        print(state[i:i+3])
    print()

def astar(src, target):
    src = [src, 0]
    visited_states = []
    iterations = 0
    while src:
        iterations += 1
        current = min(src, key = lambda x: f_n(x, target))
        src.remove(current)
        display_state(current[0])

    if current[0] == target:
        return f"found with {iterations} iterations"
    return "Not found"
```

src = [1, 2, 3, 8, 4, 7, 6, 5]
target = [2, 8, 1, 0, 4, 7, 7, 6, 5]
fit (astar (src, target))

output:



Found 40 iterations.

IDF

```
def iterative_deepening_search(graph, start, goal):
    def dfptl(node, depth):
        if node == goal:
            return [node]
        elif depth > 0:
            for child in graph.get(node, []):
                result = dfptl(child, goal, depth-1)
                return [node] + result
    depth = 0
    while True:
        result = depth_limited(start, goal, depth)
        if result is not None:
            return result
        depth += 1
def get_user_input_graph():
    graph = {}
def get_user_input_graph():
    for _ in range(num_edges):
        node1, node2 = input().split()
        if node1 in graph:
        else:
            graph[node1].append(node2)
        else:
            graph[node2] = [node1]

def main():
```

if path:
    print("found") -> join(path[::-1])
else:
    print("No path found")
if name == "__main__":
    main()

output:
Ent no of edges: 14
Ent each Edge
x p
y x
p s
p s
x f
x h
s b
s c
s v
s z
f v
f s
h i
h w

Ent the starting node = y
ending node: f
Path found: y -> x -> f //

15/10/24

22/10/24

Annealing Algorithm

Algorithm :-

+ Step 1 :- Import math and random

+ step 2 :- Define the objective function
  Eg :- $f(x) = (x-3)^2$

+ step 3 :- Set the current solution to the initial solution :- [the current solution is generated randomly]

+ step 4 :- Simulated Annealing function
  set the initial state, temperature cooling rate & no of iterations as input.

+ step 5 :- Temperature function
  the algorithm iterates to a fixed no of time.
  In iterations new state is generated by making a small changes

+ step 6 :- If the new solution is accepted and is low then it is accepted or it is temporarily stored in the storage

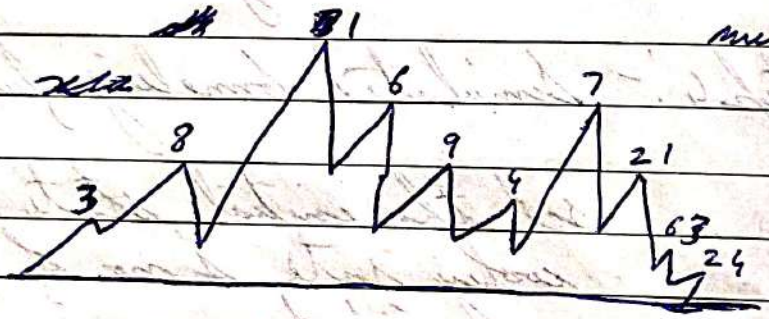
+ step 7 :- as each iteration is done

reduce the temperature according to the cooling rate.

step 2:- As the temperature is reduced to the lowest the iterations will reduced.

step 3:- When the cooling rate is the lowest stop iteration and give the last answer.

Eg:- $f(x) = (x+3)^3$



Temperature = 100
cooling rate = 10
x is generated randomly first (y = 1)
iterations ()
every iteration the initial value is compared with the next generated value if its the best solution its stored in the best solution variable.
Temperature = 100 - 10 // (Temperature - cooling rate)

The loop is executed till the temp = 0 // and the last solution is considered as the best solution.

Proved

Bafna Gold
Date: Page:

Program

```
import math
import random

def objective_function(x):
    return 10 * len(x) + sum([(xi**2 - 10*
    math.cos(2 * math.pi * xi)) for
    xi in x])
```

def Formula = if (f(n) < f(n)) accept n
otherwise accept n with

$$p = \exp\left(\frac{-f(n) - f(n)}{T}\right)$$

```
def get_neighbors(x, step_size=0.1):
    neighbors = x[:]
    index = random.randint(0, len(x)-1)
    neighbor[index] += random.uniform(-step_
        size, step_size)
    return neighbors.

for i in range(n_iterations):
    t = temp / float(i+1)
    candidate = get_neighbors(current, step_
        size)
    candidate_eval = objective(candidate)
    if candidate_eval < best_eval or random
        .random() < math.exp(current_eval/t):

if i % 100 == 0:
    print("i" iteration [i], Temp [t:3f], best
```

state]

Evaluation: {best_val:.5f}")

return best, best_val, scores.

bound = [(-5.0, 5.0) for _ in range(2)]
n_iterations = 1000
step_size = 0.1
temp = 10

best, score, scores = (step_size)

print('best solution: {best}')

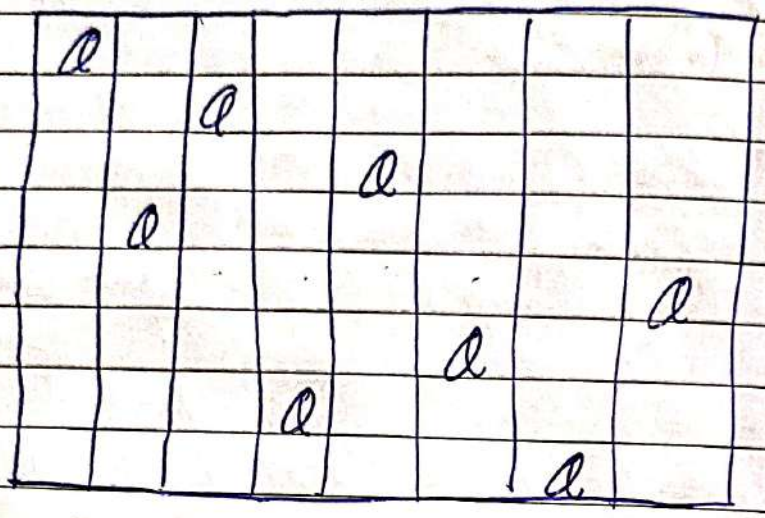
Output:-

Iteration 0: Temperature 10.000, Best Eval: 49.67956
Iteration 100: Temperature 0.099 Best Eval = 17.92548
Iteration 200: Temperature 0.050 Best Eval = 17.92070
Iteration 300: Temperature 0.025 Best Eval = 17.92070
Iteration 400: Temperature 0.020 Best Eval = 17.90770
Iteration 500: Temperature 0.020 Best Eval = 17.90770
Best Score: 17.909695940499816

22/10/24

29/10/24

LAB-#6 (8 queens)



Hill climbing search for 8-queens :-

```
function hill_climb():

    current_state = random_initial_state()
    current_cost = evaluate_state(current_st

    while current_cost != 0:

        neighbours = generate_neighbours(current_stat

        best_neighbour = None
        best_cost = current_cost

        # evaluate each neighbour and find the one
        with least attacking pairs.

        for neighbour in neighbours:
            neighbour_cost = eval_state(neighbour)
            if neighbour_cost < neighbour_cost
                best_cost = neighbour
```

Bafna Gold
Date: Page:

```
// Move to best neighbour.
current_state = best_neighbour
current_cost = best_cost

return current_state //

// evaluate states (state):
    attacking_pairs = 0
for i = 0 to len(state) - 1:
    for j = i+1 to len(state) - 1:
        if state[i] == state[j] or
            abs(state[i] - state[j]) == abs(i - j)
            attacking_pairs += 1
return attacking_pairs
```

A*

```
// Initialization
open_list = []
closed_list = set()

start_state = random_initial_state()
g_start = 0
h_start = eval_heuristic(start_state)
f_start = g_start + h_start
open_list.push((f_start, g_start, start_state))

Main loop //
while open_list is not empty:
    f_current, g_current = open_list.pop()

if (eval_heuristic(current_state) == 0:
```

return current_state

No of queens placed ↑ No of queens to be placed

Find neighbors : [f(n) = g(n) + h(n)]

↓ Heuristic value

g_neighbor = g - count(+)

h_neighbor = find_heuristic(neighbor)

f_neighbor = g_neighbor + h_neighbor

open_list.push(f_neighbor, g_neighbor, neighbor)

Outputs :- A*

Q + + + + + + +
+ + + + + + Q +
+ + + + Q + + +
+ + + + + + + Q
+ Q + + + + + +
+ + + Q + + + +
+ + + + + Q + +
+ + Q + + + + +

Bill Shirley :-

+ Q + + + + + +
+ + + + + + + Q
+ + + + + Q + +
+ + Q + + + + +
+ + + + + + Q +
+ + + Q + + + +
Q + + + + + + +
+ + + + Q + + +

No solution found

12/10/24

Bafna Gold
Date: Page:

LAB-7

Entailments - Literals

Problem:-

Knowledge base:

1) Alice is the mother of Bob
2) Bob is the father of Charlie
3) A father is a ~~person~~ parent
4) A mother is a parent
5) All parents have children
6) If someone is a parent their children are siblings.
7) Alice is married to David.

step by step solution

1) Hypothesis: "charlie is a sibling of bob"

2) Knowledge process:-

→ considering statement 1 & 4, we can conclude that alice is a parent of bob.

→ considering statement 2 & 3 we can conclude that Bob is a parent of charlie

→ considering statement 6, we say that if children are siblings then they must have a common parent.

→ considering statement 1 & 2, We can say

that bob & charlie don't have a same parent

⇒ Therefore charlie & bob are not siblings.

3) Conclusion:

The hypothesis "charlie is a sibling of bob." is not Entailed by the knowledge base.

+ Literals Knowledge processing :-

+1) Alice is mother of bob
M(Alice, bob)

2) Bob is father of charlie:
F(Bob, charlie)

3) A father is a parent, A mother is a parent
P(Alice) and P(Bob)

4) All parents have children
$\exists y\, C(Alice, y)$ and $\exists y\, C(Bob, y)$

5) If someone is parent their children is sibling
C(Alice, Bob) & C(Bob, charlie)

6) A Conclusion:
Entailment S(Bob, charlie) is true.

Program:-

```
class Family_tree:
    def __init__(self):
        self.father_of = {'Bob': 'Alice'}
        self.mother_of = {'charlie': 'bob'}
        self.married_to = {('Alice', 'David')}

    def prove_all(self):
        print("1. Alice is the mother of Bob:", "Bob"
              in self.mother_of and self.mother_
              of['Bob'] == 'Alice')

        print("2. Bob is the father of charlie:",
              "charlie" in self.father_of &
              self.father_of['charlie'] == 'Bob')

        print("3. A father is a parent:", Bob" in
              self.father_of)

        print("5. All parents have children:")

        print("6. If someone is a parent:")

        print("7. If Alice is married to David) is
              self.married_to of ('David', 'Alice') is
              self.married_to)

main()
family_tree = family_tree()
```

family_tree.pson - all()

Output :-

1) Alice is mother of Bob : True
2) Bob is father of Charlie : True
3) A father is present : True
4) A mother is present : True
5) All parents have children : True
6) Bob & Charlie are siblings : True
7) Alice is married to Brad : True

Conclusion: Charlie is sibling of Bob : True //

19/11/24

10/11/24

Lab - 8

First order logic and Unification

1) ~~John loves mary & Alice gives Bob a book~~

2) ~~Socrates teaches~~

1) Every student gives a book to someone
2) John teaches philosophy to every student
3) There is a person who teaches philosophy to a student
4) Alice gives a pen to every teacher
5) Anyone who teaches philosophy loves a philosopher.

+ Predicates:
Gives (x, y, z) : x gives y to z.
Teaches (x, y, z) : x teaches y to z.
Loves (x, y) : x loves y
Student (x)
Teacher (x)
Philosopher (x)

First-order-logic :-

1) $\forall x (\text{student}(x) \rightarrow \exists y (\text{Gives}(x, \text{Book}, y)))$

2) $\forall x (\text{student}(x) \rightarrow \text{Teaches}(\text{John}, \text{Philosophy}, x))$

3) $\text{Teaches}(x, \text{Philosophy}, y, y) \wedge \text{student}(y))$

4) $\forall x (Teaches(x) \rightarrow Loves(Alice, x))$

5) $\forall x (Teaches(x, Philosophy, y) \rightarrow (Philosopher(x))$

* Unification

1) $\forall x (student(x) \rightarrow \exists y (Loves(x, Rob, y))$
$\exists x \exists y (Teaches(x, Philosophy, y) \wedge student(y))$

$\forall x (student(x) \rightarrow \exists y (Loves(x, Rob, y)))$

$= student(x, Rob, y)$

2) $\exists x \exists y (Teacher(x, philosophy, y) \wedge student(y))$

$\rightarrow Teaches(student, Philosophy, Y)$

Proceed

Output :-
Original Predicates:
Loves(Alice, Rob, Rob)
Loved(Rob, Rob, Alice)

Unification successful with substitutions
$Y \rightarrow Rob$

19/11/24

Bafna Gold
Date: Page:

7/12/24

Week 7

Solve FOL using Forward chaining

Problem:- As per the law, it is crime for an American to sell weapons, to hostile nations, is enemy of America, has some missiles & all the missiles were sold to it by Robert who is American citizen.

Prove:- "Robert is criminal."

Facts:-

1) Country A is an enemy of America.
2) Country A has missiles.
3) Robert sold missiles to country A.
4) Robert is an American citizen.

Representation in FOL:-

1) It is a crime for an American to sell weapons to hostile nations.

+ $American(p) \wedge Weapons(q) \wedge Sells(p,q,s) \wedge Hostile(s) \Rightarrow Criminal(p)$

2) Country A has some missiles:
$\exists x \; Owns(A,x) \wedge Missiles(x)$

3) All of the missiles were sold to country A by Robert

$\forall x$ Missile $(x) \wedge$ owns $(A, x) \Rightarrow$ Sells (Robert, x, A)

4) Missiles are weapons

Missile $(x) \Rightarrow$ weapon (x)

5) Enemy of America is known as hostile

$\forall x$ Enemy $(x,$ America$) \Rightarrow$ Hostile (x)

6) Robert is american

American (Robert)

7) The country A, an enemy of America

Enemy (A, America)

Observation :-

From 7 & 5

We consider country A is hostile to america Enemy (A, America) $\Rightarrow$ Hostile (A)

from 6 & 3

We identify that robert is american & sold missiles to country (A) & owns (R) Weapon

$\therefore$ for 1 we consider Robert is criminal

Robert (criminal) ~~0 R ?~~

Output :-

Robert is a criminal //

Bafna Gold
Date: Page:

3/12/24

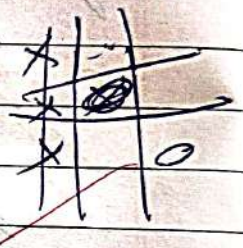
LAB-10

TIC-TAC-TOE using Min-Max

```
def evaluate(board)
    row = 3
    col = 3
    grid = row + col

def min_max(board, depth, player)
    i = 0
for i in 1 to 100
    if i%2==0
        Ai
        min_max()
    if else
        Ai2
        min_max()

score = evaluate(board)
if score == 10 or score == -10:
    return score

if maximizing player(AI)
    best = -float('inf')
    for j in range(3)
        for j in range(3):
            if board[i][j] == '_':
            if board[i][j] == 'x':
                if board[i][j] == 'o':
return best
```

```
def find best move(board):
    best_val = -float('inf')
    best_move = (-1, -1)

    for i in range(3):
        for j in range(3):
            if board[i][j] == '_':
                move_val = min_max(board, 0, False)

                if move_val > best_value:
                    move value = best_val
    return best move.
```

-10
0
10



output :-

Current board

```
X O X
O O X
_ _ X
```

The best move is (3,3)

Bafna Gold
Date: Page:

1/12/24

LAB-10

8- Queens by alpha-beta pruning

```
def grid():
    size = 8*8

def is safe(self, size, row, col)

    for i in range(col):
        if board[row][i] == 1:
            return false

    for i, j in zip(range(row, self, size),
        range(col, -1, -1)):
        if board[i][j] == 1;
            return false.
    return false.

alpha_beta_search(self.board col. alpha, beta
    maximizing_player)
    if col >= self.size : # if all queens are
        return e,                    placed

    else max_player:
        best_board = None
    for row in range(self, size)
        board[row][col] = 1
    if val_score > max_val:
        return max_val

main():
```

def print_board(self, board):

Output :-

. . . . . a . . .
. . . . a .
a . . . . . .
. . a . . . . . .
. . . . a
. . . . . a . . .
. . . a . . . . .

Ground

Air