Bafna Gold
Date: Page:

24/9/24 LAB PGM-1 TIC-TAC-TOE

ALGORITHM:

1. Create a 3x3 matrix where each cell is empty with default value = -1
2. Take user input as 0 or X in one cell. if already filled ask to fill in other cell which is empty. board[row][col] empty
3. check if 3 0's or X's in horizontal, vertical or diagonally consecutive manner

```
if board[row][col] is empty:
    place the X
    end loop
else
    enter in empty cell
```

4. if row : (0,1,2) or (3,4,5) or (6,7,8)
   or column : (0,3,6), (1,4,7), (2,5,8)
   or diagonal : (0,4,8), (2,4,6)
   return True
   else
   return False
5. while true
   if board[row][col] is empty:
   Print O
   end loop
6. if random

3. After calling check function

If all values of row is 0
return 1

if all values of a col is 0
return 1

if [0][0] [1][1] & [2][2]
return 1

if [0][2] [1][1] & [2][0]
return 1

else return 0.

if the condition is true (1)
print user wins.

4. generate random()
row, col = random.choice [empty cells present]

Refer Algo

generate random row mod 3
generate random col mod 3.
if mat[i][j] != -1
return 1

else
generate random()
row, col = random.choice (empty cells)

5. then comp wins, print comp won

6. If all cells = filled.
Declare draw.

```
import random
def initialise_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]
                                    != ' ':

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_boo_win_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 in range(3) if board
                                [i][i]==' ']
    if empty_index:
        return empty_index[0], 2-empty_
                                index[0]

    return None

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

def computer_move(board):
    if move:
        make_move(board, 'O', move)
        return

move=check_two_in_a_row(board,'X')
```

```
make_move(board, 'O', move)
    return

moves = available(board)
if
    move = random.choice(moves)
    make_move(board, 'O', move)

def user_move(board):
    while True:
        try:
            row = int(input("Enter row"))
            col = int(input("col"))
            if board[row][col] == ' ':
                make_move(board, 'X', (row, col))
                return
            else:
                print("Not spot");
        except(ValueError, IndexError):
            print("Invalid")

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

for _ in range(9):
    display_board
    if current_player == 0:
        user_move(board)
    else:
        comp_move(board)
    win = check_win(board)
```

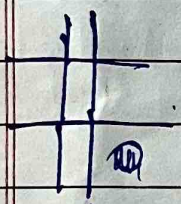
if winner
display_board (board)
print ("Player (winner) wins")
return

current_player = 1 - current_player

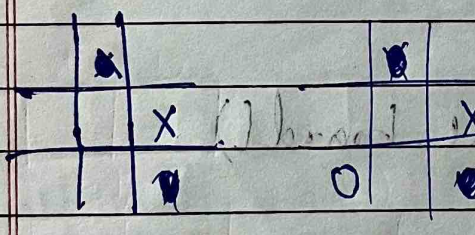
display_board (board)
print ("Draw")

play_game()

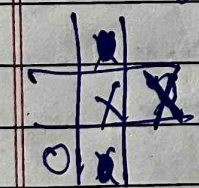
output:

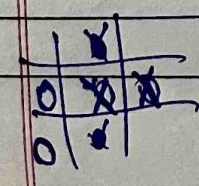


Enter row: 0
col: 2



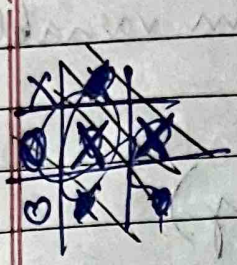
Enter row: 0
col: 1





Bafna Gold
Date: Page:

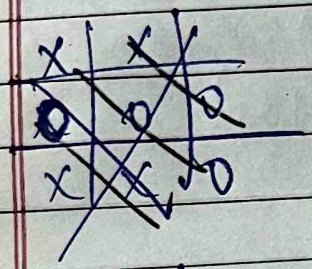
Enter row : 0
col : 0



X		
O	X	X
O		

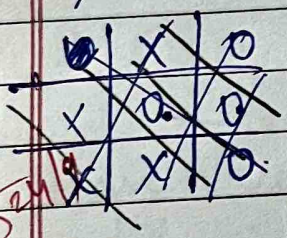
X		
O	X	X
O		O

Enter row : 0 2
col : 0 1



X		
O	X	X
O	X	O

X	O	
O	X	X
O	X	O



row : 0
col : 2

X	O	X
O	X	X
O	X	O

~~Straight draw~~ Its a draw.
fix

1/10/24

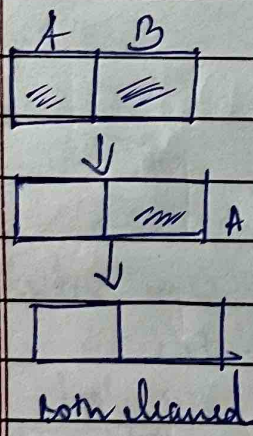
Vaccum cleaner

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

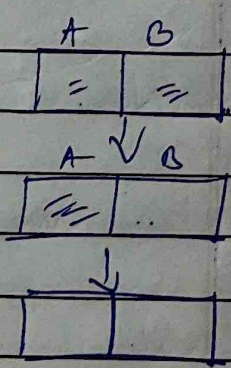
2. while (room A || B == dirty)
{
if (room A is dirty) {
if (vac cleaner is in A) {
clean A
dirty--;
else (goto B) {
if (cleaner is in B) {
if (room B is dirty)
clean B
else
goto A)
}

if (A & B is clean)
exit

when A is first



when B is first



Bafna Gold
Date: Page:

Percept Sequence:

	Action:
A(dirty) & [dirty]	clean A
A(clean)(A& dirty)	clean B go to A
A(dirty)(A& clean)	clean A go to B
A(clean) B(clean)	cleaned.

(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 vacuum cleaner:
  def __init__(self, grid):
    self.grid = grid
    self.position = (0,0)

def clean(self):
  x, y = self.position
  if self.grid[x][y] == 1:
    print("cleaning {self.pos}")
    self.grid[x][y] = 0
  else:
    print("Pos {self.pos} is clean")
def move(self, direction):
  x, y = self.pos
  if dir == up & x > 0:
    self.pos = (x-1, y)
```

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

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

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

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

def set_dirty_coordinates(rows, cols, num_dirty):
    dirty = set()
    while len(dirty) < num_dirty:
        coords = input("Enter coordinates")
        x, y = map(int, coords.split(','))
        if 0 <= x < rows and 0 <= y < cols:
            dirty.add((x, y))
        else:
            print("Out of bound")
```

```
rows: int(input("Enter rows"))
cols: int(input("Enter cols"))
num_dirty = int(input("Enter dirty cells"))

if dirty > rows * cols:
print("D.C exceeds")

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

dirty_coordinates = get_dirty_(rows, cols,
                               num_dirty)

for x, y in dirty_coordinates:
    initial_grid[x][y] = 1

vaccum = Vaccum_cleaner(initial_grid)
print("Initial grid")
  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
Enter coordinates for 3: 1,1
Enter coordinates for 4: 1,0
```

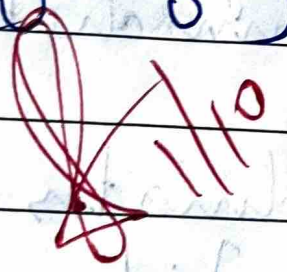
Initial grid

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

cleaning pos (0,0)
cleaning pos (0,1)
cleaning pos (1,0)
cleaning pos (1,1)

Final grid

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



LAB: 3 8. PUZZLE

Using DFS & Manhattan

Algorithm: Manhattan

1. Create 3×3 grid with an initial path & empty path.
2. Depending on the position of the no in which the no is located there are 3/2 possible moves.
3. If the place is already visited then move to the neighbouring place (up, left, right, left)
4. Make all possible moves and finally calculate the distance by subtracting the current position with the final position to achieve.
5. From all move select the minimum
6. Repeat until the puzzle completes.

DFS:

1. Push initial state into stack.
2. When stack is not empty
3. If element is not visited push onto stack
4. Do all possible moves (L, R, T, B).
5. While popping current node, if in final state end it or continue.
6. If not visited move (L, R, T, B)
7. Visited set will be added to stack.

Refer as update

check if visited:
pop, until current state.
make move.

else:
push into stack
exit if goal state is reached.

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

def get_neighbours(state):
    neighbours = []
    for i in range(3):
    for j in range(3):
    if state[i][j] == 0:
        if i > 0
neighbors.append(swap(state, i, j, i-1, j))
        if i < 2:
neighbors.append(swap(state, i, j, i+1, j))
    if j < 2:
    neighbors.append(swap(state, i, j, i, j+1))

def swap(state, i1, j1, i2, j2):
    new_state = [row[:] for row in state]
```

```
def dfs_with_manhattan(initial_state, goal_st,
                       visited=None):
    if visited is None:
        visited = set()

    if initial_st == goal_st:
        return [initial_st]

    visited.add(str(initial_state))

    neighbors = sorted(get_neighbors(initial_st),
        key=lambda x:
        manhattan_dist(x, goal_st))

    for neighbor in neighbors:
        if str(neighbor) not in visited:

            path = dfs_with_manh(neighb, goal_st,)
            return [initial_st] + path
    return None

initial_st = []
print("Initial st")
for i in range(3):
    row = input("Enter row{i+1}:
if sol:
    print("Sol")
    for state in sol:
        for row in state:
            print(row)
        print()
    else: print("No sol").
```

Output:

Enter initial:

1: 0 1 2

2: 5 6 3

3: 4 7 8

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

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

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

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

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

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

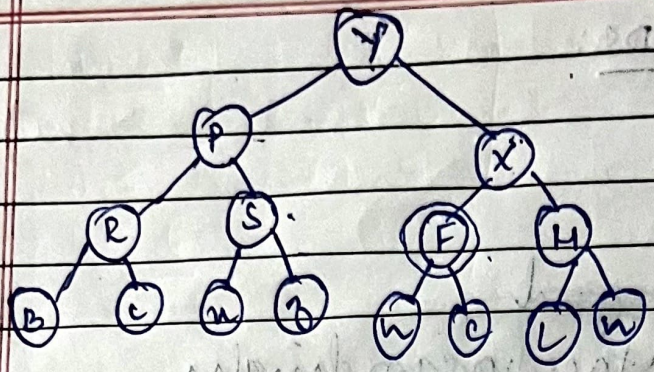
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 0 & 7 & 8 \end{bmatrix}$$

15.10.24 LAB: 4 IDS:

Algo:

1. ~~Create a 3x3 grid~~
2. Set a goal stage accordingly.
3. From the root node start searching and traverse the entire graph.
4. While searching, from root node, go to each child node, check if it is the goal state, exit.
5. Go level wise and check each node and search all the nodes in that level.
6. Start with depth as 0 & then increase the limit.
7. If the goal is found, exit if not inc the depth & continue

```
function IDFS (root, goal)
    for (depth = 0 to INT MAX)
        root = DLS( root, goal, depth)
            if result:
                return result
        return NULL
function DLS ( root, goal, depth)
    if depth = 0
        if root = goal
    else:   return root
    result = DLS (root, goal, depth - 1)
    if result:
        return result
    return NULL
```



I. iteration — At depth = 0 — Goal state
V.N = Y — Not found

II iteration — Depth = 1 — Goal state
visit = YPX — Not found

III iteration — Depth = 2
V.N = YPRSX

IV — Depth = 3 — Goal state.
V.N = YPRSXF — Found

Bafna Gold
Date: Page:

A* stas:

Algo:

1. Input the goal state as g
2. I/p initial state as i
3. We have to initialise g:=0.
4. Run until the goal state is reached.
5. When each move is made check if it is the same as the previous one.
6. If it is done, then skip that move or else continue.
7. f = g + no. of positions (displaced)
8. If goal is reached, return the value.
9. else, see the f values, & then choose min of f. & g++ (increment it)

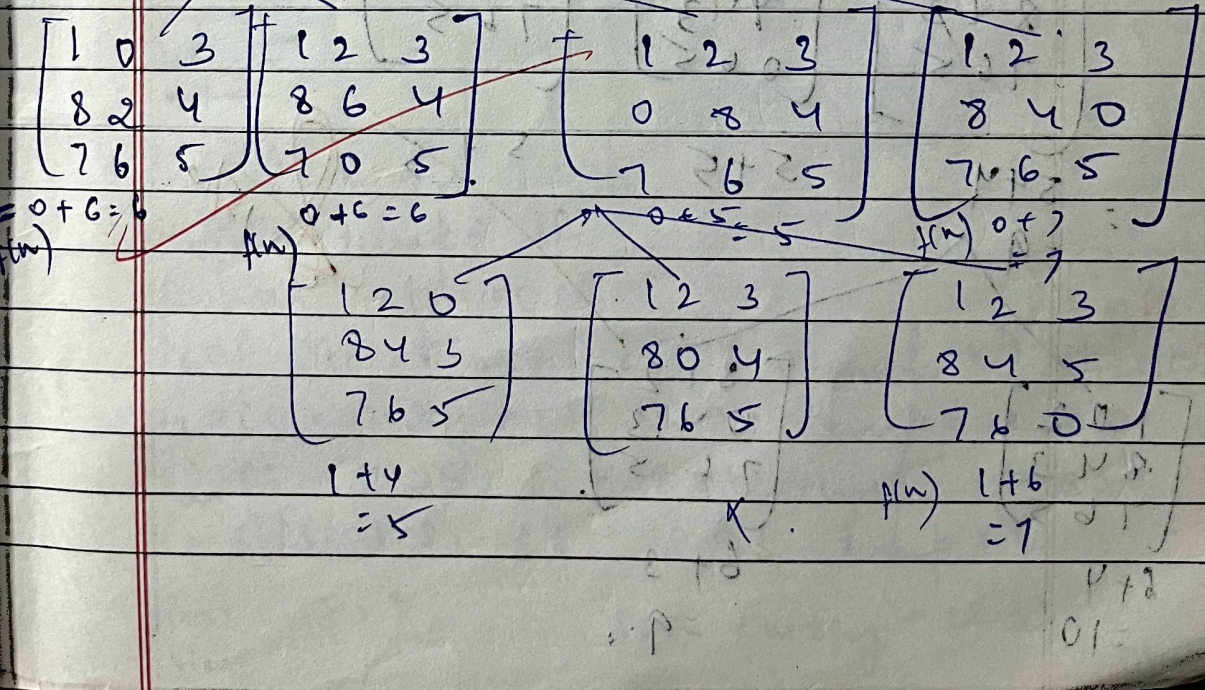
f(n) = g(n) + h(n) -> misplaced blocks

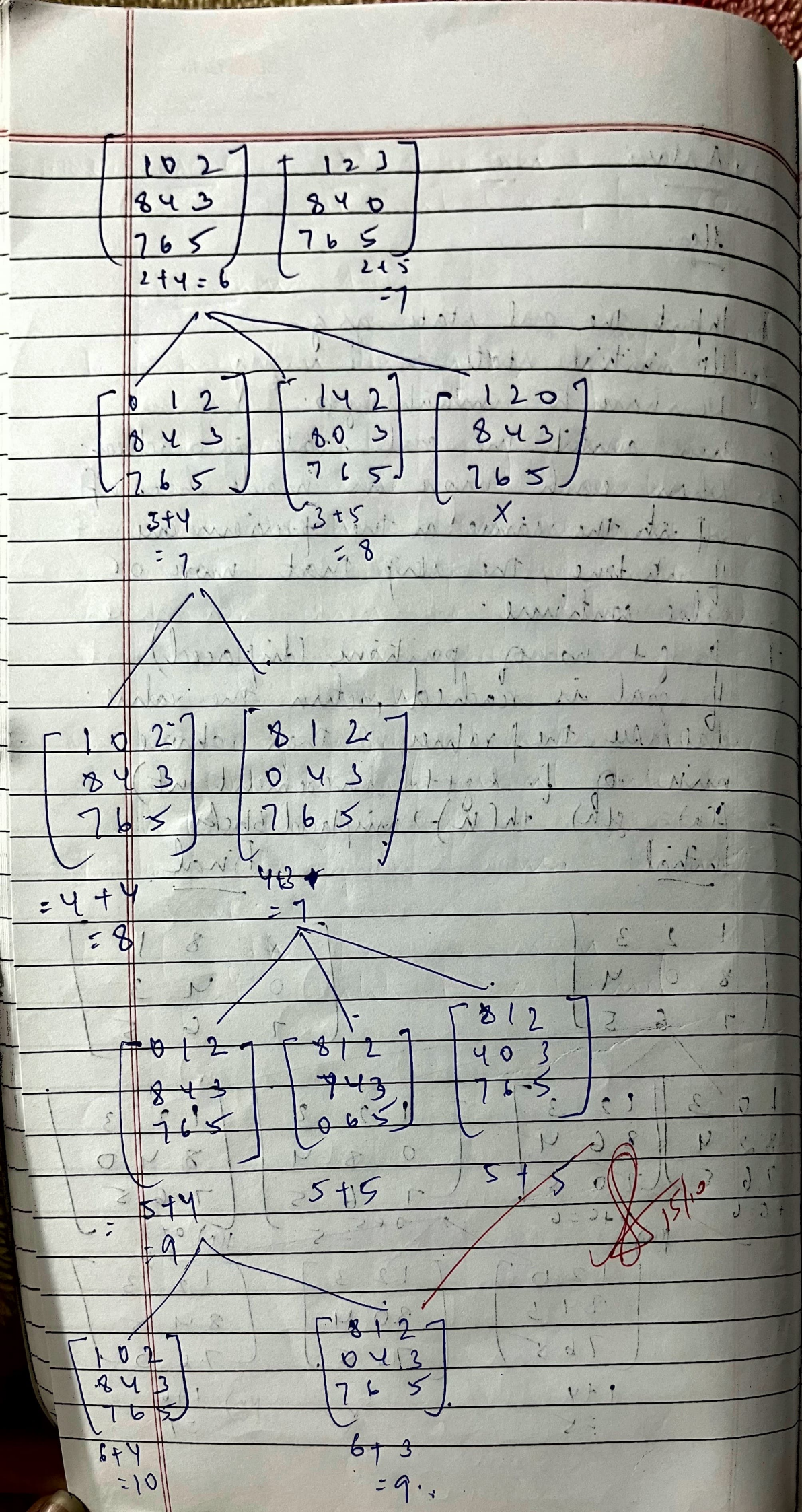
Initial

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

Final

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




1 0 2
8 4 3
7 6 5
2+4=6
1 2 3
8 4 0
7 6 5
2+5
=7
0 1 2
8 4 3
7 6 5
3+4
=7
1 4 2
8 0 3
7 6 5
3+5
=8
1 2 0
8 4 3
7 6 5
X
1 0 2
8 4 3
7 6 5
=4+4
=8
8 1 2
0 4 3
7 6 5
4+3
=7
0 1 2
8 4 3
7 6 5
5+4
=9
8 1 2
7 4 3
0 6 5
5+5
8 1 2
4 0 3
7 6 5
5+5
1 0 2
8 4 3
7 6 5
6+4
=10
8 1 2
0 4 3
7 6 5
6+3
=9

Bafna Gold
Date: Page:

22/10/24

SIMULATED ANNEALING

Algorithm:

1. Set an initial state
2. Initial temparature should also be chosen.
3. The cooling point should also be defined as the temparature decreases.
4. The process should be continued until the minimum temparature is reached.
5. The solution obtained should be modified to create another sol.
6. Evaluate the cost.
7. If this sol has low cost, then accept it.
8. If higher cost, then the prob = $e^{-\Delta E / T}$
9. Dec temp, according to the cooling pt.
10. When the criteria is met then stop there.

(temp > 0, iteration < max itera.)

objective func: $f(x) = x^2$

Implemented
22/10/24

Output

Initial sta = 10
Initial temp: 12
Cooling rate: 0.2
No of iterations = 25 (0 and 1)

Implemented
22/10/24

i1 = CS = 9.21, CE = 85.99 temp = 2.400
i2 = CS = 9.25, CE = 88.61, temp = 0.480
i25 = CS = 3.15, CE = 12.93, temp = 0

Best = 3.58
Hate

Best Energy = 12.83

29.10.24 LAB: 6 A* Algorithm & Hill Climb Racing for 8 Queens

Algorithm: A*

1. Initialization: Placing 1 queens randomly in empty board in the matrix.
2. Initialise the priority queue from the start node & its heuristics.
3. Dequeue the node from the lowest point value from the priority queue.
4. If h=0, return the possible solutions.
5. For each queen move it accordingly in its column
6. For every move it makes, calc the g(cost to reach current state) & h(existing queen)
7. The rest of the queens, add into queue.
8. f = g+h for new queen.
9. Repeat until sol is found.

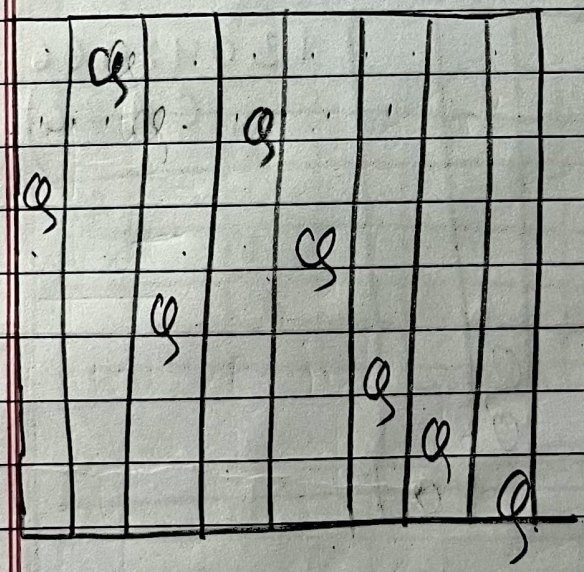
Bafna Gold
Date: Page:

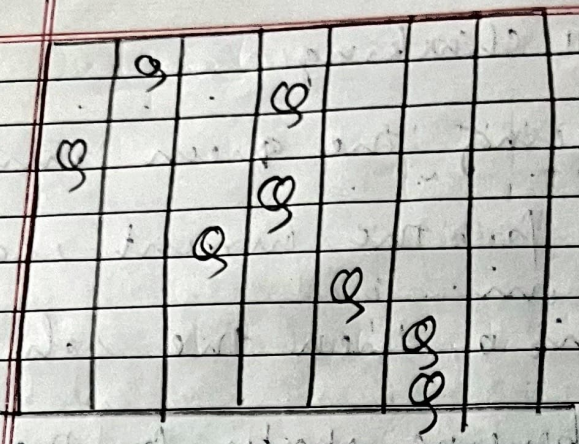
Algorithm : Hill Climbing.

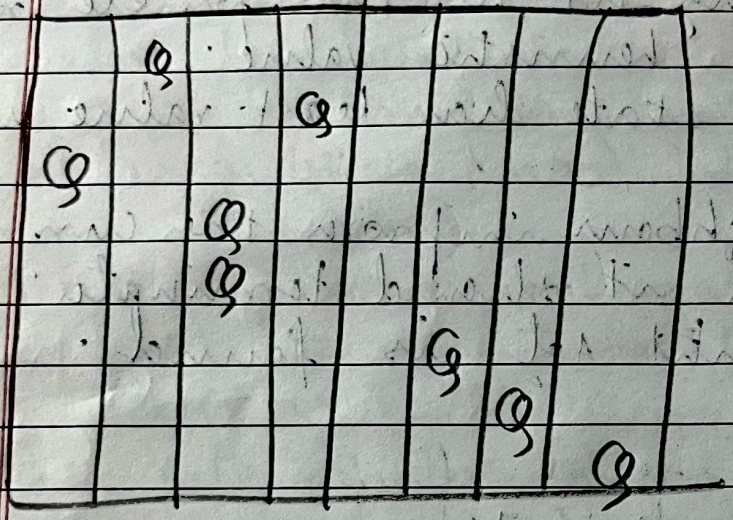
1. Start by placing the queen randomly ~~placed~~.
2. Calc heuristic for the current state which it is in.
3. If the value is 0, then the solution is found.
4. Find the neighbouring states for the queens by moving it to the possible row & calc the heuristic value.
5. Which ever state has least value select it.
6. If the neighbour improves the current state, then it should terminate.
7. Repeat until sol is found.

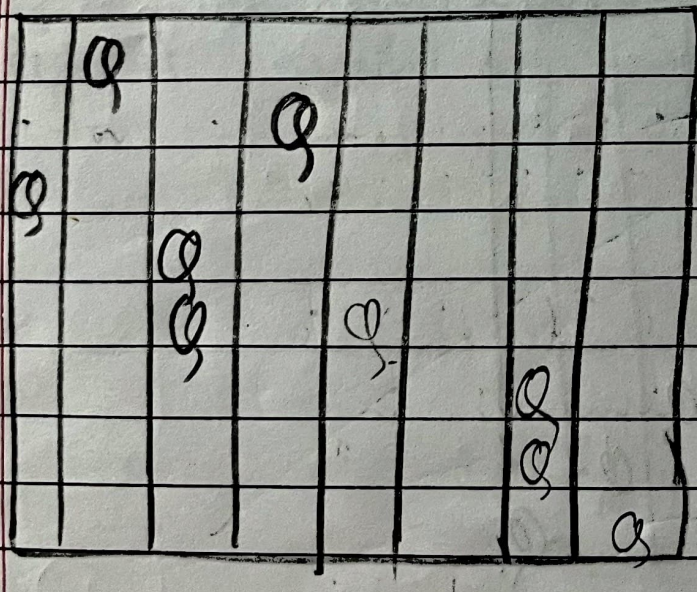
Proceed

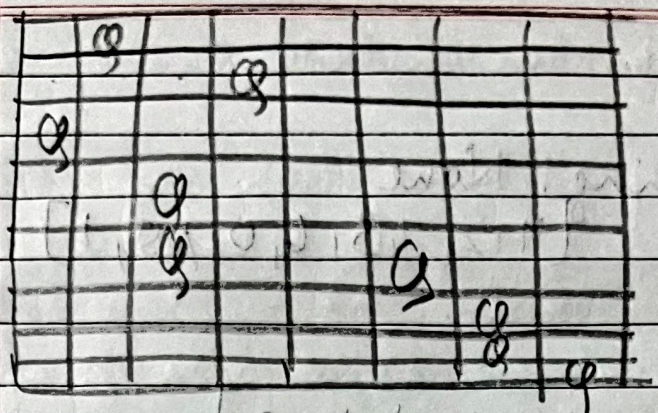
1 3 0 5 2 4 6 7



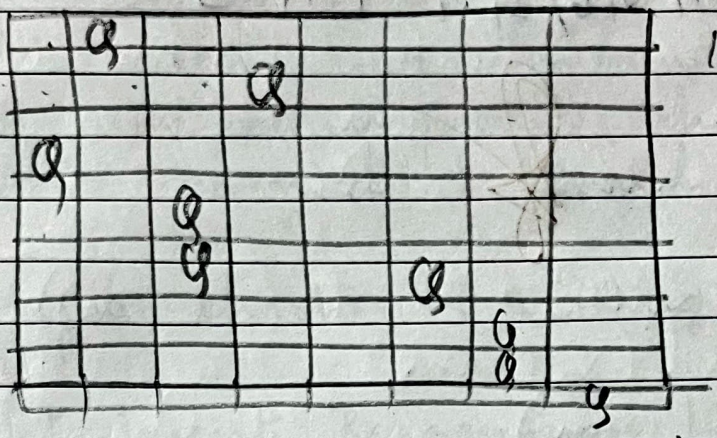
1) 
13052467 - (M-8)

2) 
13052466
(A-7)

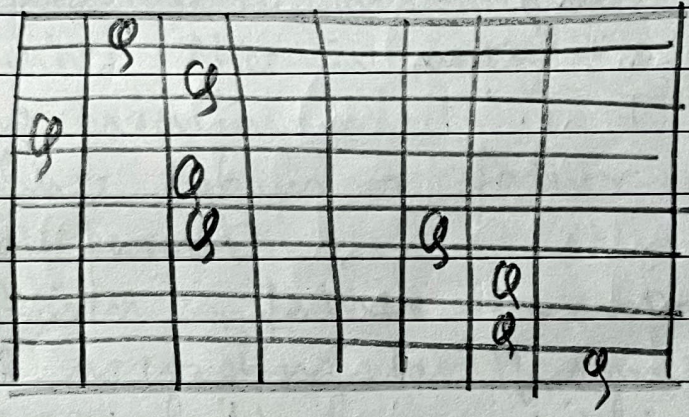
3) 
13042466
(A-6)

4. 

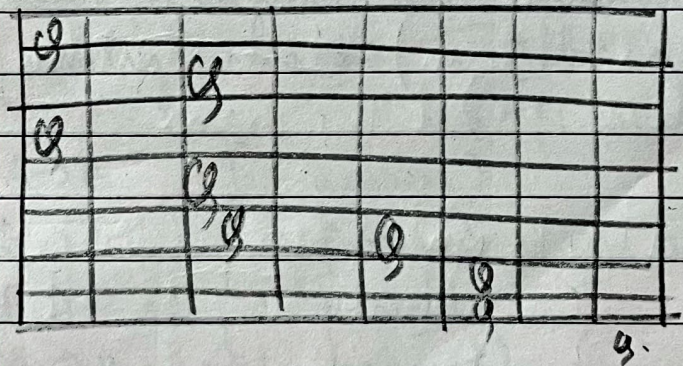
1 3 0 4 2 3 6

5. 

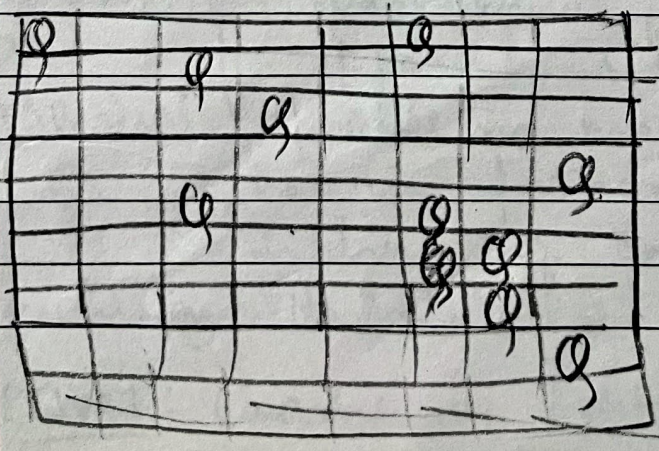
1 3 0 4 2 3 6

6. 

1 3 0 4 1 2 6 6

7. 

1 2 0 4 1 2 6 6

8. 

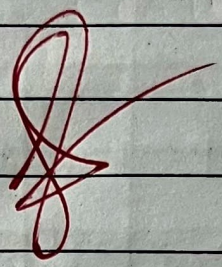
0 2 0 4 1 2 6 6

→ 0 2 0 3 1 2 6 6

Output:

1. Hill Climbing: None
[4, 2, 7, 3, 6, 0, 1, 5, 1]

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



12/11/24 CA5-6 Entailment using literals

KB:
Alice is mom of Bob
Bob is father of charlie
A father is parent
A mother is a parent
All parents have children
If someone is parent, children is sib
Alice married to David

Hypo: charlie is sibling of Bob.

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

A → B (mother)
B → C (father)
F → P (parent)
M → P (parent)
P → S (if parent, siblings)
A ^ B → Q (If Alice is mother of Bob & Bob is father, charlie is sibling of Bob.)
Conclusion: Its true

RESULT: charlie is sibling of Bob: True.

19/11/12

First order logic

Statement: If it is a holiday, then store is closed. Today is holiday so store is closed. ✗

Students have passed the exam, if any student who passed the exam receive the certificate, if John is a study & passed exam, he'll receive a certificate.

∀ x (Student(x) → Passed (x, Exam))
∀ x (Passed (x, Exam) → Receives Certificate(x))
Student (John)
Passed (John, Exam)
Receives Certificate (John)

Prof ∀ x (Student(x) → Passed (x, Exam))
tells us that if someone is student they passed the exam.

From Student (John) ∀ Passed (John, Exam)
so John has passed the exam.

∀ x (Passed (x, Exam) → Receives Certificate (x))
tells that if student has passed the exam, they receive certificate.

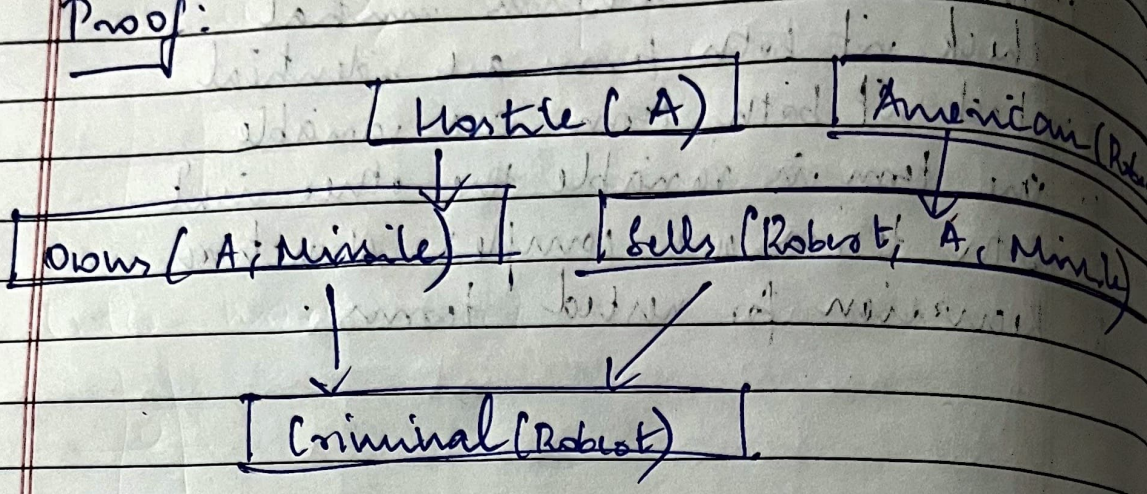
From Passed (John, Exam) ∀ → Receives Certificate (John)

⇒ Receives Certificate (John) is true.

19/11/12

LAB-8 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

Missile (a) ⇒ Weapons (a)

∀a Enemy (a, America) ⇒ Hostile

American (Robert)

Enemy (A, America)

Criminal (Robert)

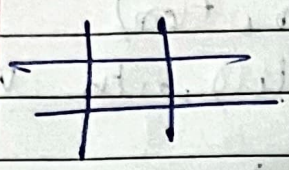
Bafna Gold
Date: Page:

Alpha-Beta Search

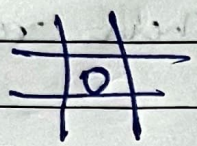
```
func ALPHA-BETA-SEARCH (state) returns an actio
    v ← MAX-VALUE (state, -∞, +∞)
    return action ACTIONS(state) with v

func MAX-VALUE (state, α, β) returns a
utility value
    if TERMINAL-TEST (state) then return
    UTILITY (state)
    v ← -∞
    for each a in ACTION (state) do
        v ← MAX (v, MIN-VALUE (RESULT(s,a)
        if v ≥ β then return v
        α ← MAX (α, v)
    return v

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

Output

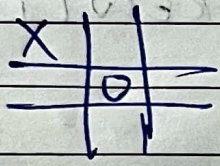
You are O, comp is X



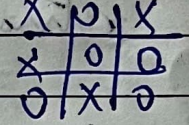
Enter move : 2,2



Comp played

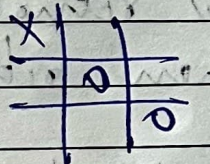


Enter move

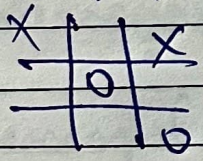


Draw

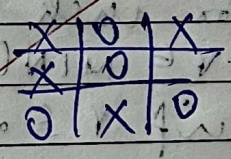
Enter move : 3,3



Comp played

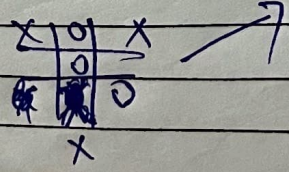


Comp played



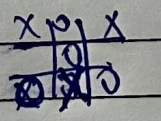
Enter move : 1,2

Comp played



Enter move

3,1

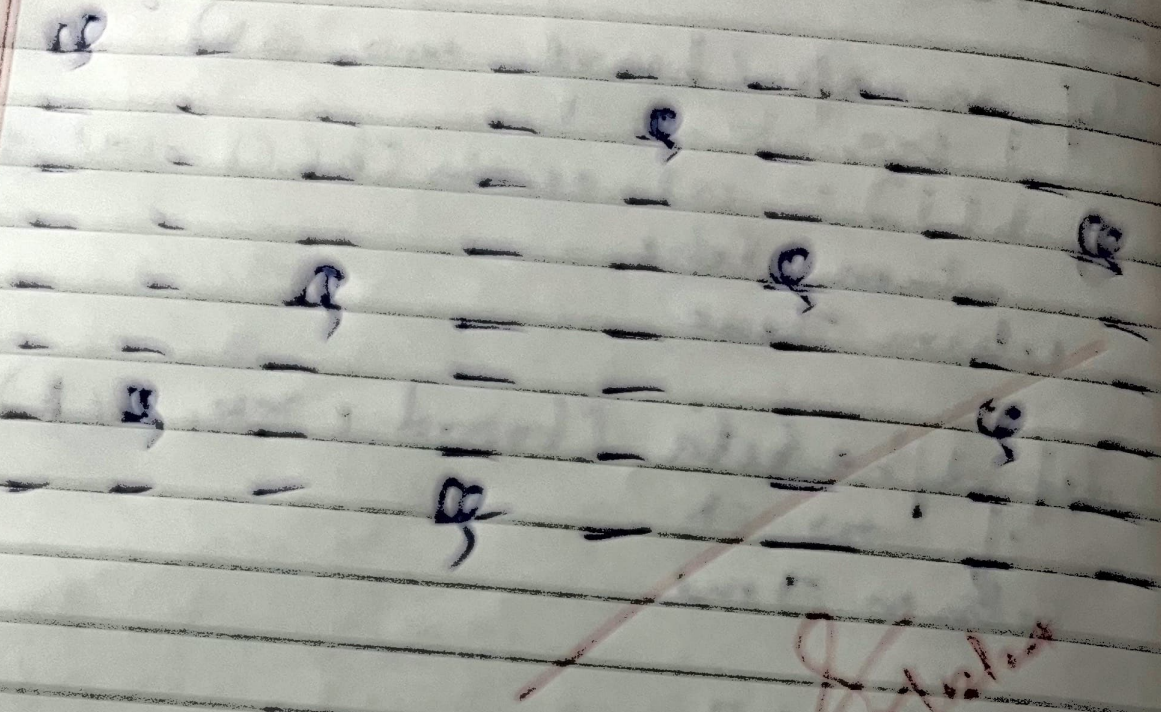


You won

N-Queens:

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def alphabeta(board, row, alpha, beta):
    if row == 8:
        return True

    for col in range(8):
        if is_safe(board, row, col):
            board[row] = col
            if alphabeta(board, row + 1, alpha, beta):
                return True
            board[row] = -1

    alpha = max(alpha, row)
    if alpha >= beta:
        break
    return False

solve_8queens():
    initialize [-1, -1, -1, -1, -1, -1, -1, -1]
    if alphabeta(board, 0, -∞, ∞):
        return board
    return None

sol = solve_8queens()
if sol is not None:
    print sol
else no sol:
```

Output:



Unification

1. Two terms t_1 & t_2 as initial
2. check wit both terms are identical
3. check wit both terms are variable
4. one term is variable the other isn't
5. Both terms are complex structures
6. Recursion for nested terms