INDEX

Name Dipesh Sah.

Standard 5 **Section** 'B' **Roll No.** 092

Subject Artificial Intelligence Lab.

SL No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	01-Oct 2024	Vaccum cleaner; Tic-Tac-Toe.	01 - 07	8/10
02.	8-Oct 2024	8-puzzle problem - DFS & BFS	8 - 12	S. Pradeep 8/10/24 10/10
03.	15-Oct 2024	A* search for 8-puzzle with Misplaced Tiles & Manhatten Distance	13 - 16	S. Pradeep 14/10/24 10/10
04.	22-Oct 2024	Hill Climbing Search Algorithm.	17 - 19	S. Pradeep 22/10 10/10
05.	29-Oct 2024	Stimulated Annealing to solve N-Queens Problem.	20 - 25-26	10/10 S. Pradeep
06.	12-Nov 2024	Propositional Logic.	21-22	S. Pradeep 12/11/24 10/10
07.	19-Nov 2024	Unification in First Order Logic	23 - 24	10/10 S. Pradeep
08.	26-Nov 2024	Forward Reasoning in FOL	27 - 28	10/10 S. Pradeep 26/11/24
09.	03-Dec 2024	FOL Resolution	29 - 33	10/10 S. Pradeep 3/12/24
10.	17-Dec 2024	Alpha-Beta Pruning.	34 - 35	

Lab - 1

1.) Vaccum Cleaner

Algorithm :-

1. Initialize goal state :
   - Set goal_state = {'A':'0', 'B':'0'}
     (0 = clean, 1 = dirty)
   - Initialize cost = 0.

2. Input the vaccum location :
   - Get vaccum's location :
     location_input = 'A' or 'B'.
   - Get status of the current room :
     status_input = '0' or '1'.
   - Get status of the other room :
     status_input_complement = '0' or '1'.

3. Vaccum in Room A :
   - If location_input == 'A' :
     - If status_input == '1' (A is dirty) :
       - Clean A, set goal_state ['A'] = '0', increment cost.
     - If status_input_complement == '1' (B is dirty) :
       - Move to B, clean it.
       - update goal_state ['B'] = '0'.
       - increment cost.

4. Vaccum in Room B :
   - If location_input == 'B' :
     - If status_input == '1' (B is dirty) :

- Clean B, set goal_state ['B'] = '0',
- increment cost.

- If status_input_complement == '1' (A is dirty):
  - Move to A, clean it
  - update goal_state ['A'] = '0'.
  - increment cost.

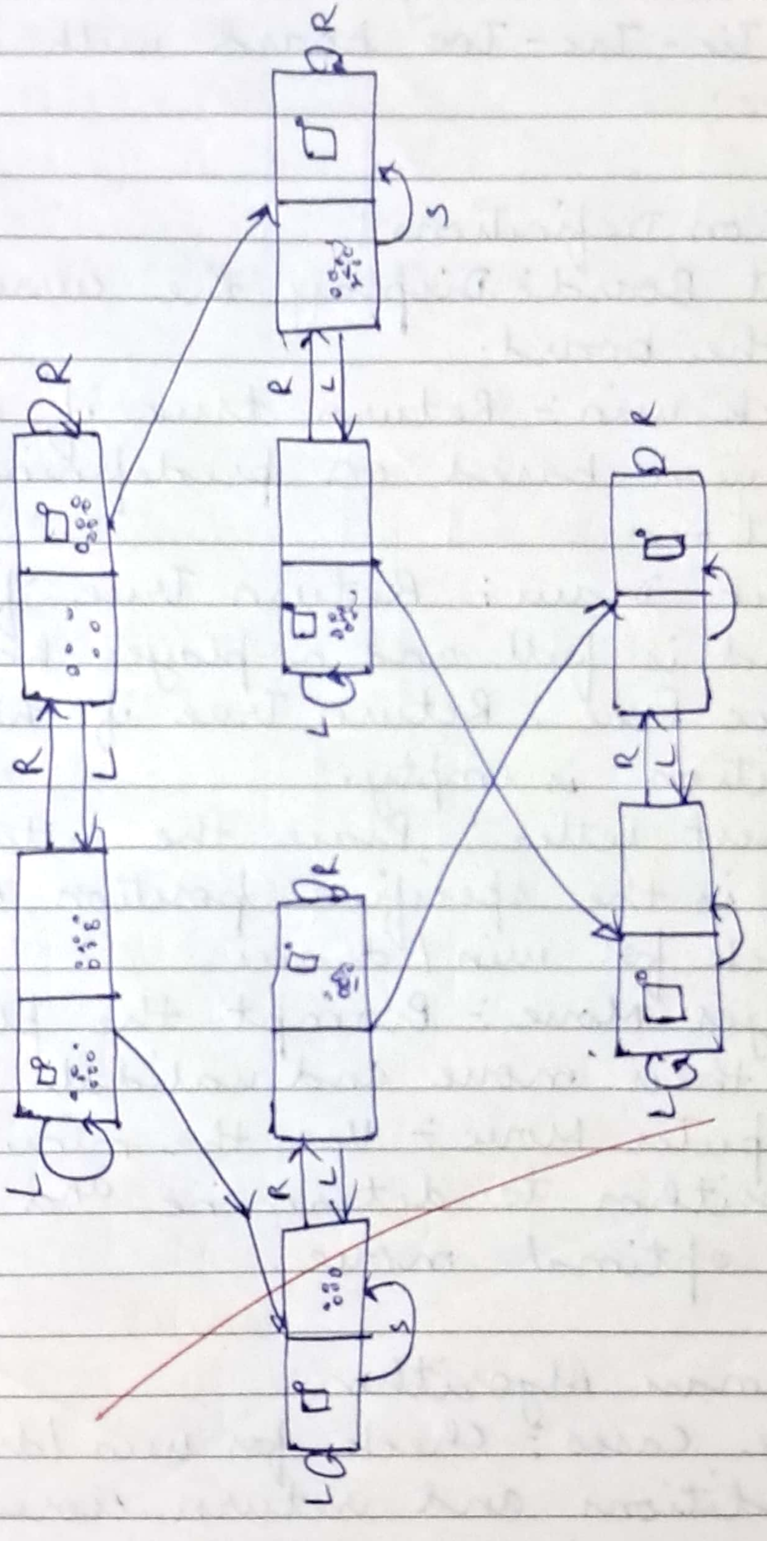
5. Output:
   - print goal_state and cost as the performance measure.

Output:- Initial condition : {'A': '1', 'B': '1'}
Vaccum is placed in location. A.
Location A is dirty.
Cost for Cleaning A: 1
Location A has been cleaned.

Location B is Dirty.
Moving right to location B.
Cost for moving Right : 2
Cost for cleaning : 3
Location B has been cleaned.

GOAL State:-
{'A': '0', 'B': '0'}
Performance Measurement : 3

State space Tree



2) Tic-Tac-Toe.

Algorithm:

1. Initialize the board.
Create a dictionary (or list) to represent the Tic-Tac-Toe board with empty spaces.

2. Function Definitions:
- Print Board: Display the current state of the board.
- Check win: Return true if any player has won based on predefining wining conditions
- Check Draw: Return True if the board is full and no player has won.
- Space Free: Return True if the selected position is empty.
- Insert letter: Place the letter ('X' or 'O') in the specified position and check for win / draw.
- Player Move: Prompt the player for their move and validate input.
- Computer Move: Use the minimax algorithm to determine and make the optimal move.

3. Minimax Algorithm:
- Base Cases: Check for win / draw conditions and return scores based on the outcome.
- Recursion: For maximizing (computer) and minimizing (player), evaluate

possible moves and return the best score.

4. Main Game Loop.
• While the game is ongoing:
  • If no win/draw, call computer Move.
  • Check for win/draw conditions.
  • If no win/draw, call Player move.
  • Check for win/draw conditions.

5. End Game:
• Display the final board and announce the result (win/draw).

Pseudocode :-

```
Initialize board
While true:
    If not checkWin() && not checkDraw();
        compMove ()
    If checkWin() or checkDraw();
        break;
    If not checkWin() && not checkDraw();
        playerMove ()
    If checkWin() or checkDraw():
        break

Display final board.
Announce result.
```

Output:-

X		

Enter position for O(1-9) : 3

X		O

X		O
X		

Enter position for O (1-9) : 7.

X		O
X		
O		

X		O
X	X	
O		

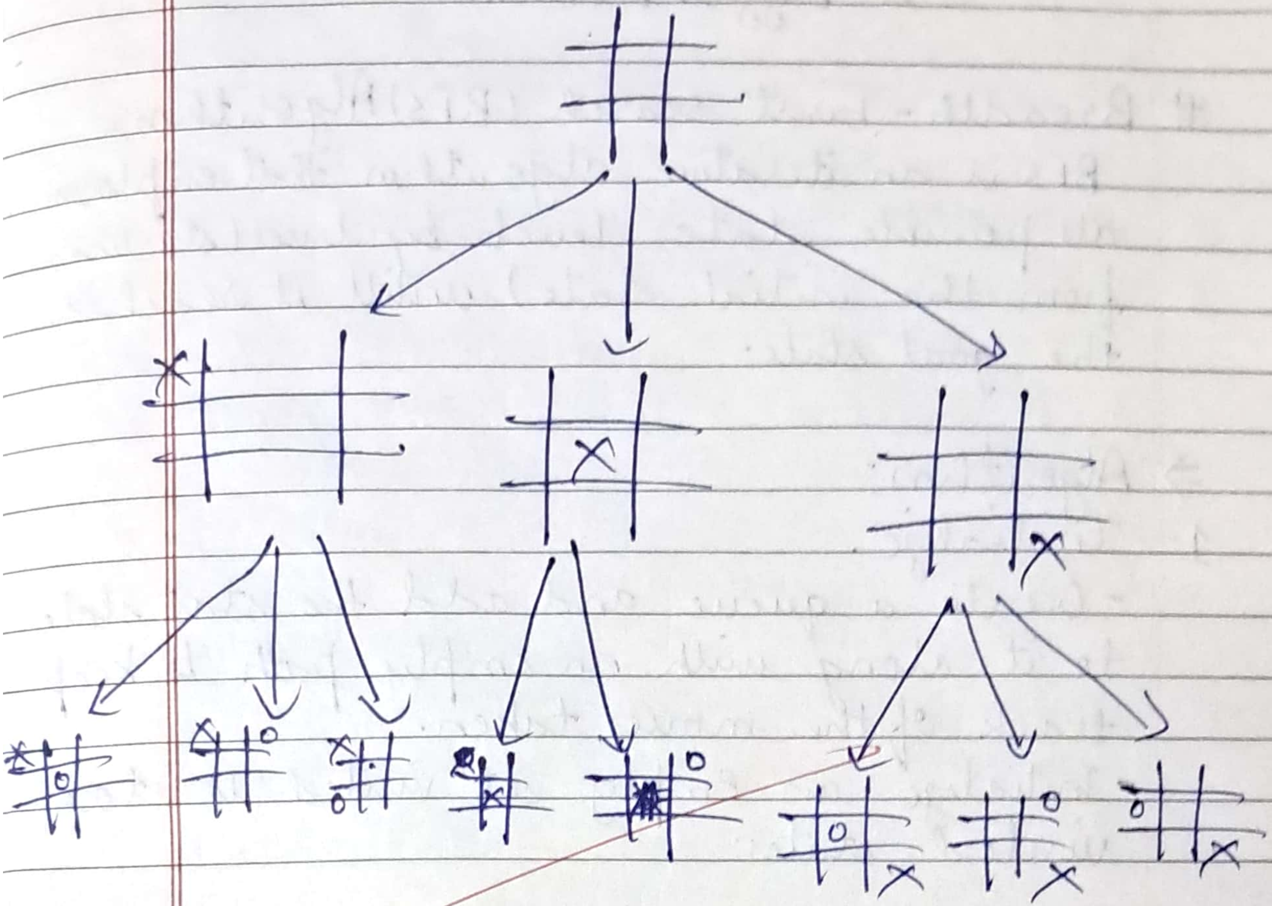
Enter position for O(1-9) : 6

X		O
X	X	O
O		

X		O
X	X	O
O		X

Bot wins !

• State Space Tree :



1/10/24

Lab-2
8-Puzzle Problem.

Breadth-First Search (BFS) Algorithm.

BFS is an iterative algorithm that explores all possible states level by level (starting from the initial state) until it reaches the goal state.

⇒ Algorithm:-

1. Initialize.
   - Create a queue and add the start state to it, along with an empty path to keep track of the moves taken.
   - Initialize an empty set 'visited' to store visited states.

2. While queue is not empty:
   - dequeue the front element from the queue (current state and the path.).

3. Check goal.
   - If the current state is the goal state, return the path (solution).

4. Mark current State as visited.
   - add the current state to 'visited set'.

5. Generate successors:
   - For each valid move from the current state:
     - apply the move to get a new state.
     - If the new state has not been visited:

- add the new state and the updated path to the queue.

6. Return Failure:
- If the queue is empty and no solution has been found, return failure (no solution).

⇒ Output:
Initial state:
[1, 2, 3
4, 0, 6
7, 5, 8]

Solving using BFS:
Exploring state in BFS:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8],

Exploring state in BFS:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

⋮

- BFS solution: ['down', 'right']

Exploring state in DFS:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

8-puzzle problem using Depth-First Search Algorithm

DFS explores a branch as far as possible before backtracking when it reaches a dead end (or goal).

⇒ Algorithm:

1- Initialize:
- Create a stack and add the start state to it, along with an empty path to keep track of the moves taken.
- Initialize an empty set 'visited' to store visited states.

2- While stack is not empty:
- pop the top element from the stack (the current state and the path).

3. Check goal:
- If the current state is the goal, return the path (solution).

4. Mark current state as visited:
- add the current state to the 'visited set'.

5. Generate successors:
- For each valid move from the current state:
  - apply the move to get a new state.
  - If the new state has not been visited:
    - push the new state and the updated path to the stack.

6. Return failure:
- If the stack is empty and no solution has been found, return failure (no solution exists.)

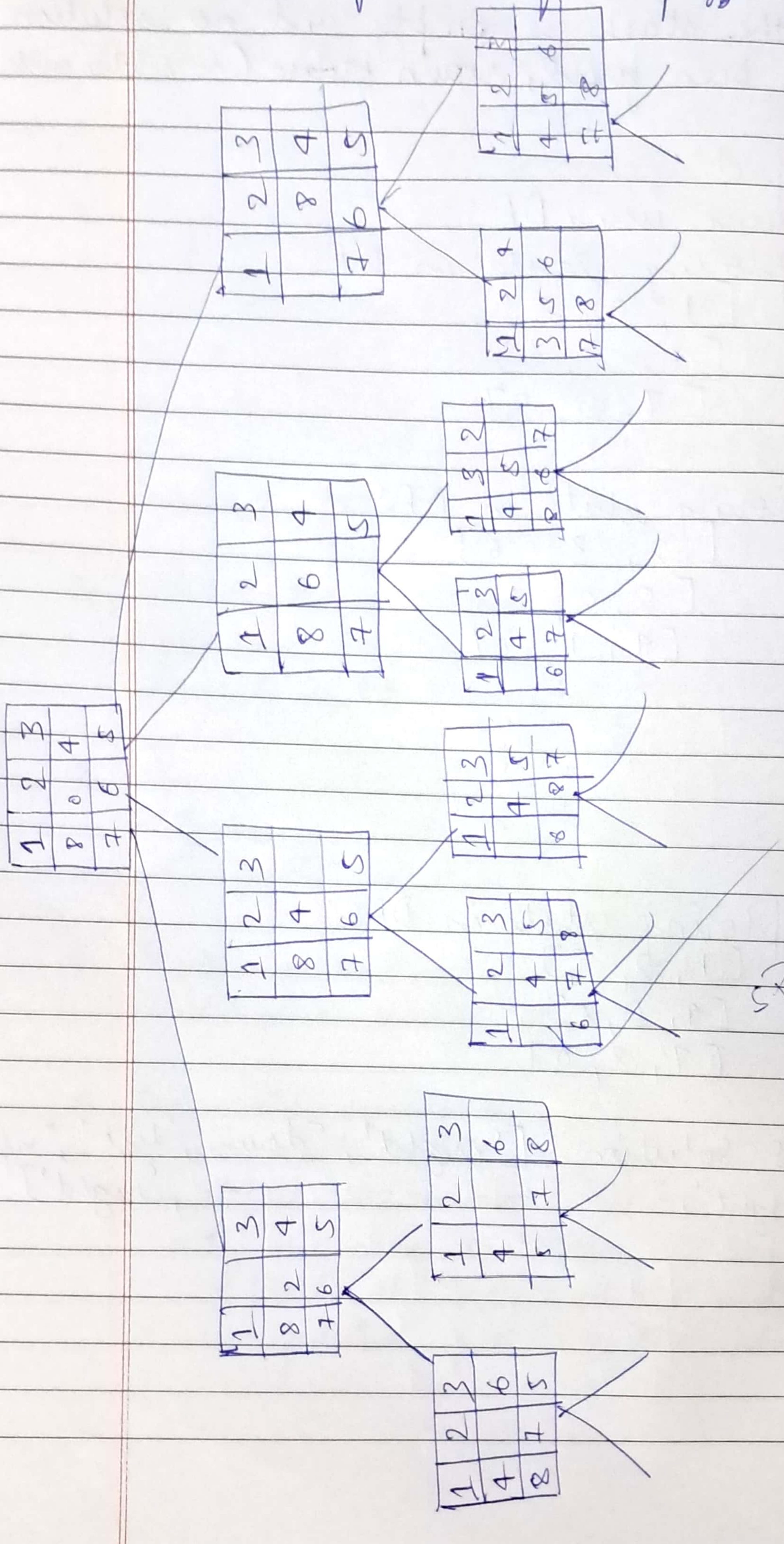
⇒ Output:
Solving using DFS:
Exploring state in DFS:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]
⋮

Exploring state in DFS:
[2, 3, 6]
[8, 0, 5]
[4, 1, 7]
⋮

Exploring state in DFS:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

DFS solution: ['right', 'down', 'left', 'up', 'right', ----------, 'right']

State Space Tree for 8-puzzle problem.



S. Pradeep
8/10/24

Lab-3

Algorithm 1: A* search for 8-puzzle with Misplaced Tiles Heuristic.

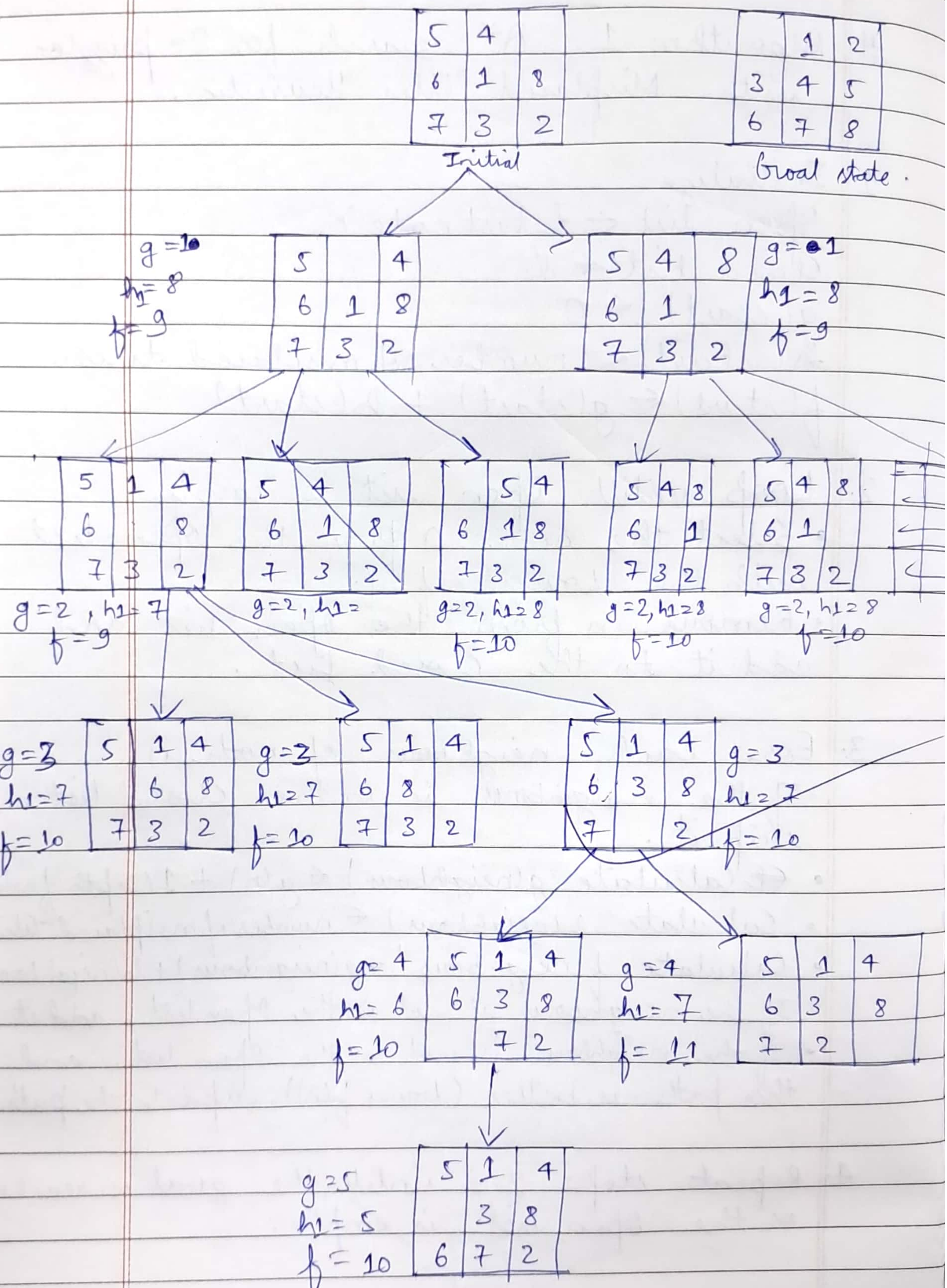
1. Initialize
Open list ← {start node}
Closed list ← ∅
g(start) ← 0
h(start) ← number of misplaced tiles.
f(start) ← g(start) + h(start)

2. Loop untill Open list is empty
- Select the node n from the open list with the lowest f(n).
- Remove n from the open list and add it to the closed list.

3. For each neighbour of node n.
- If the neighbour is in the closed list, skip it.
- ~~Ct~~ Calculate g(neighbour) ← g(n) + 1 (Depth of node)
- Calculate h(neighbour) ← number of misplaced tiles
- Calculate f(neighbour) ← g(neighbour) + h(neighbour)
- If the neighbour is not in the open list, add it.
- If the neighbour is not in the open list, and this path is better (lower f(n)), update the path.

4. Repeat steps 2-3 untill the goal is reached or the open list is empty.

State Space Tree.



Algorithm 2: A* Search for 8-puzzle with Manhattan Distance heuristic.

1. Initialize
open list ← {start node}
closed list ← ϕ
g (start) ← 0
h(start) ← Manhattan distance
f(start) ← g(start) + h(start)

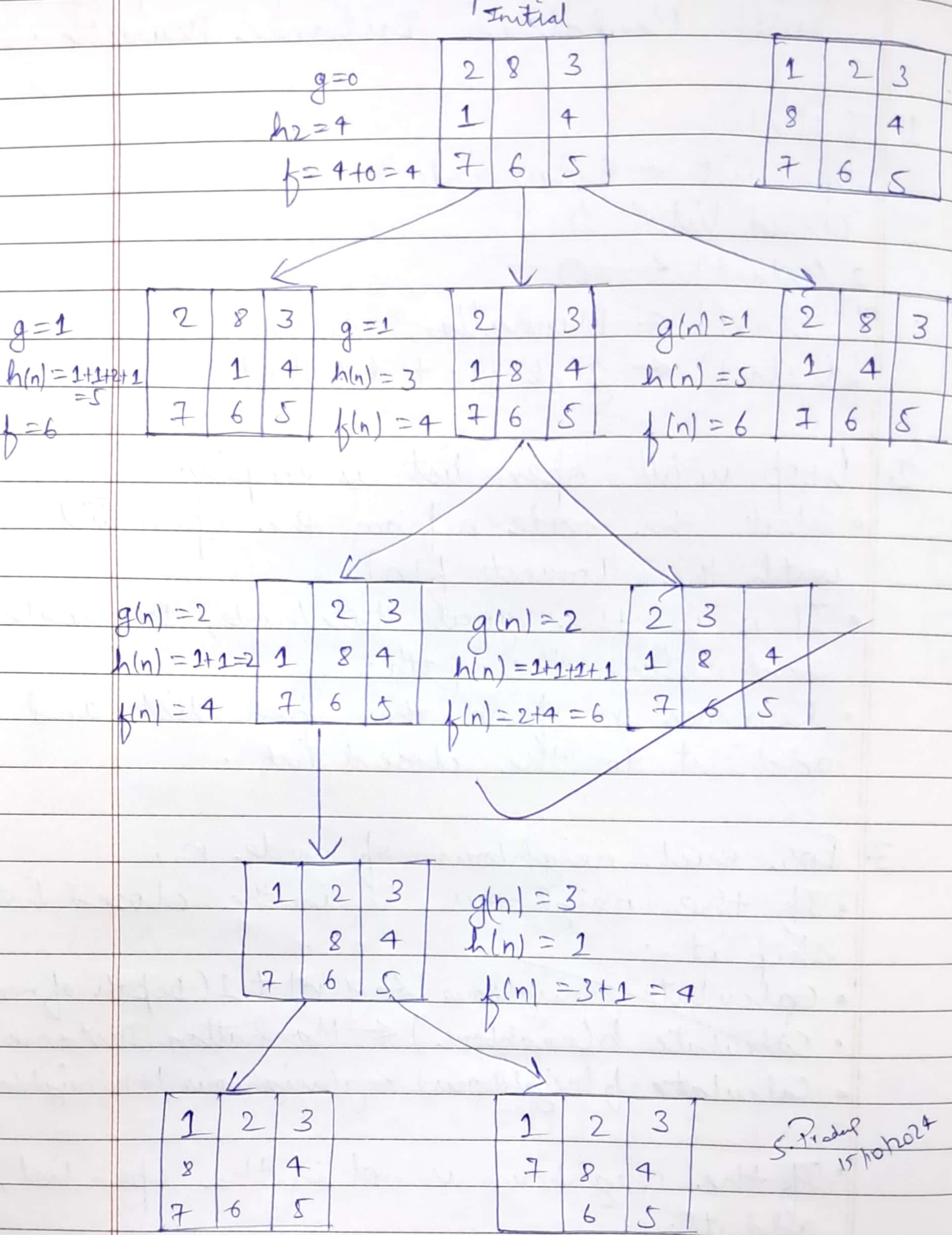
2. Loop until open list is empty.
- Select the node n from the open list with the lowest f(n)
- If n is the goal state/node, terminate and return the path.
- Remove n from the open list and add it to the closed list.

3. For each neighbour of node n,
- If the neighbour is in the closed list, skip it.
- Calculate g(neighbour) ← g(n) + 1 (Depth of node)
- Calculate h(neighbour) ← Manhattan Distance.
- Calculate f(neighbour) ← g(neighbour) + h(neighbour)

- If the neighbour is not in the open list, add it.
- If the neighbour is in the open list and this path is better (lower (f(n)), update the path.

4. Repeat steps 2-3 until the goal is reached or the open list is empty.

State Space Tree



Final state
goal reached.

S. Pradeep
15/10/2024

Lab - 4

Algorithm for :- Implement Hill Climbing Search algorithm to solve N-Queens problem.

1. Generate a random initial configuration of N queens on the board with one queen per column.

2. Evaluate the cost (heuristic cost) of the current configuration. This cost is the number of pairs of queens that are attacking each other.

3. Generate neighbours by moving each queen within its column to every other row and calculate the cost for each neighbour.

4. Select the neighbour with the lowest cost. If no neighbour has a lower cost than the current state, terminate. (reached a local or global optimum).

5. If a neighbour has a lower cost, move to that neighbour and repeat the process.

6. If stuck at a local optimum, optionally restart with a new random configuration (random restart hill climbing).

7. Stop when the goal (zero conflict) is reached.

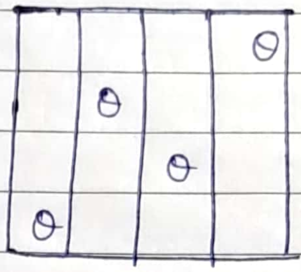
Pseudocode:

```
function hill_climbing_n_queens(initial state):
    current_state = initial_state

    while true:
        current_heuristic = calculate_heuristic (current)

        if current_heuristic == 0:
            return current_state

        neighbours = generate_neighbours (current_state)
        best_neighbour = None
        best_heuristic = infinity

        for each neighbour in neighbours:
            heuristic = calculate_heuristic (neighbour)
            if heuristic < best_heuristic
                best_heuristic = heuristic
                best_neighbour = neighbour

        if best heuristic >= current_heuristic:
            break // No better neighbour found.

        current_state = best_neighbour

    return None // No solutions found.
```

State Space Tree

Step. 0:

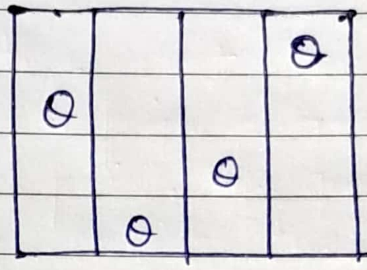
initial state



$x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0$

cost = 2

Step. 1 :- Move to neighbour

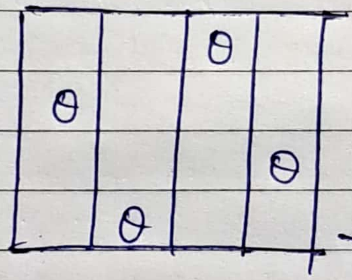


$x_0 = 3, \ x_1 = 0$

$x_2 = 2, \ x_3 = 1$

cost = 1

Step. 2 : Move to better neighbour.



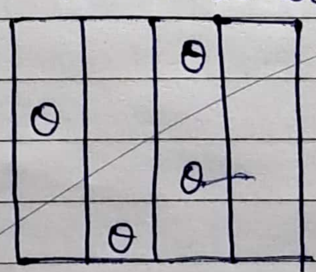
$x_0 = 2, \ x_1 = 0$

$x_2 = 3, \ x_3 = 1$

cost = 0

→ Final state (Optimal)

Step. 3 : Reached local minimum already, so no better solution can be found.



$x_0 = 2, \ x_1 = 0$

$x_2 = 2, \ x_3 = 1$

Final cost = 0

S.Pradep
22/10/24

Lab-5 (continued)
Stimulated Annealing for N-Queens.

Algorithm :-

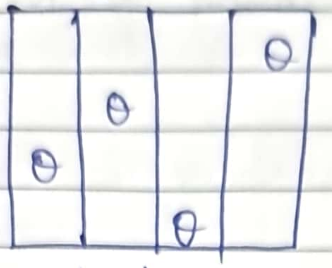
1. Initialization
   - Start with a random configuration of queens on board.
   - Calculate initial energy.
   - Set the initial temperature T_0 and final temperature T_f.

2. Annealing Loop:
   - Repeat with $T < T_f$:
     - Generate neighbouring configuration.
     - Calculate the energy of new config.
     - If new config has lower energy, accept it.
     - ~~If new config has higher energy, ΔE is change in energy.~~
   - Reduce temp T using cooling rate 'α' ; $T = T \times \alpha$.

3. Termination :
   - Stop when T reaches T_f, or a solution with zero conflicts is found.

4. Output :
   - If a valid solution (zero conflicts) is found, then return solution.
   - Otherwise, return the best config found.

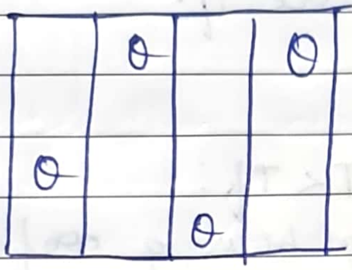
State-Space Tree.



no. of queens = 4
initial set = {2, 1, 3, 0}
$T = 100$
$\alpha = 0.95$

cost = 1

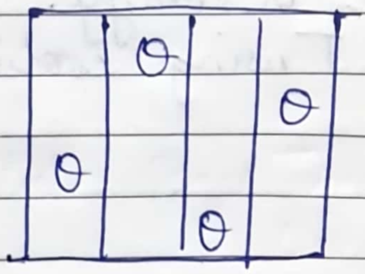
- Generate a neighbour:
  Move queen 2 from row1 to row0.



new state = {2, 0, 3, 0}
cost = 1
Δ cost = 1-1 = 0

$n = e - T_{cost} = e - 0 = e$
$T = T \times 0.95 = 95$

- Generate neighbour.



new state = {2, 0, 3, 1}
cost = 0

since cost = 0, stop.

∴ accept solution, thus final state.

Lab-6
Propositional Logic.

1. Initialize Truth Value Combinations.
   - Generate all possible True/False combinations for the propositional variables.

2. Define Operator Priority.
   - Set priorities for '~' highest, 'v' lowest.

3. User Input.
   - Prompt the user for the knowledge Base (KB) and Query (Q) expressions.

4. Convert Expressions to Postfix.
   - Convert both KB and Q from infix to postfix notation.
   - Add operands directly to the postfix expressions.
   - Use a stack to handle parenthesis and operator precedence.

5. Evaluate Postfix Expressions.
   For each truth value combination:
   - Evaluate KB and evaluate Q using a stack-based postfix evaluation.
   - Push operand values from the current combination onto the stack.
   - Apply operators using stack operations (e.g: '~' negates the top, '∧' and 'v' operate on the two values.

∧ AND   Rule $p \wedge q$
∨ OR   Query - q

6. Check Entailment.
- If there's any combination where KB is True and Q is false, return False.
- If no such combination exists, return True.

7. Output Result.
- Print "The Knowledge Base entails the query", if entailment holds; otherwise, print "The knowledge Base does not entail the query".

#

State	p	q	r	KB evaluation	Query	KB→Query
1.	True	True	True	True	True	True
2.	True	True	False	True	False	False
3.	True	False	True	False	True	True
4.	True	False	False	False	False	True
5.	False	~~False~~ True	True	False	True	True
6.	False	True	False	False	False	True
7.	False	False	True	False	True	True
8.	False	False	False	False	False	True

Result:) The query doesn't entail the query as there is a True-False relation which entails to False.

S. Pradeep
12/11/24

Lab - 7
Unification in First Order Logic.

Algorithm.

1. If Ψ_1 or Ψ_2 is a variable or constant:
   a) If Ψ_1 or Ψ_2 are identical:
      then return NIL.
   b) Else if Ψ_1 is a variable,
      i) then if Ψ_1 occurs in Ψ_2, then
         return Failure.
      ii) else return $\{(\Psi_2/\Psi_1)\}$.
   c) Else if Ψ_2 is a variable,
      i) If Ψ_2 occurs in Ψ_1, then
         return Failure.
      ii) Else return $\{(\Psi_1/\Psi_2)\}$
   d) Else return ~~$\{(\Psi_1/\Psi_2)\}$~~ Failure.

2. If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure.
3. If Ψ_1 and Ψ_2 have a different number of arguements, then return Failure.
4. Set Substitution set(SUBS) to NIL.

5. For i = 1 to the number of elements in Ψ_1:
   a) Call Unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.
   b.) If S = failure then return Failure.
   c.) If S ≠ NIL, then do.
      i) Apply S to the remainder of both L1 & L2.
      ii) SUBST = APPEND (S, SUBST).
6. Return SUBST.

Output:

Case 1: Basic Unification.
Enter the first expression (Ψ_1) : ['p', 'x', ['f', 'y']]
Enter the second expression (Ψ_2) : ['p', ['f', 'a'], 'z']
Output:
Unification Result: [[['f', 'a'], 'x'], ['z', 'y']]

Case. 2: Failure Due to Mismatched Predicates
Enter the first expression (Ψ_1): ['p', 'x', 'y']
Enter the second expression (Ψ_2): ['q', 'x', 'y']
Output:
Unification Result: FAILURE.

Case. 3: Substitution Chains.
Enter the first expression (Ψ_1) : ['p', 'x', 'y']
Enter the second expression (Ψ_2): ['p', 'a', 'b'].
Output:
Unification Results: [['a', 'x'], ['b', 'y']]

S. Pradeep 19/11/24

Lab-8
Forward Reasoning in FOL

Algorithm:

1. Initialization:
   - Begin with a given knowledge base KB containing a set of first-order definite clauses.
   - Define the query q, an atomic sentence to be proven.
   - Initialize new as an empty set to track newly derived facts.

2. Iteration:
   - Loop untill no new facts can be inferred.
   - For each rule r in the knowledge base:-

   i) Standardize Variables:
      - Replace all variables in r with unique variables to avoid clashes with existing variables in the KB.

   ii) Check Premises:-
      - Find substitutions θ such that all premises $P_1, P_2, \ldots, P_n$ of the rule hold in the current KB.

   iii) Derive Conclusions:-
      - Apply θ to the conclusion Q to infer a new fact 'Q''.

   iv) Avoid Redundancy:-
      - Ensure Q' is not already in KB or new before adding it to new.

v) Unify with Query:
- Check if θ' unifies with q. If so, return the substitution θ.

3. Update Knowledge Base:
- Add all new facts from new to the KB.

4. Termination:
- If no new facts can be inferred and the query is not resolved, return False.

Ques.) As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all the missiles were sold to it by Robert, who is an American citizen.

Prove that Robert is a criminal.

Output:) Robert is a criminal.

26/11/24

State Space Tree

Criminal (Robert)

American (Robert) ∧ Sold (Robert, Missiles, A) ∧ Hostile (A)

American (Robert)   Sold (Robert, Missiles, A)

Hostile (A)

Lab-9
FOL Resolution.

Algorithm:

1. Convert all sentences to CNF.

a) Eliminate biconditionals & implications:
- Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
- Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \Rightarrow \beta$.

b) Move $\neg$ inwards:
- $\neg(\forall x\, p) \equiv \exists x\, \neg p$.
- $\neg(\exists x\, p) \equiv \forall x\, \neg p$.
- $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$.
- $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$.
- $\neg\neg\alpha \equiv \alpha$

c) Standardize variables apart by renaming them: each quantifier should use a different variable.

d) Skolemize: each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variables.
- For instance, $\exists x\, Rich(x)$ becomes $Rich(G1)$ is a new skolem constant.
- "Everyone has a heart" $\forall x\, Person(x) \Rightarrow \exists y\, Heart(y) \wedge Has(x,y)$ becomes $\forall x\, Person(x) \Rightarrow Heart(H(x)) \wedge Has(x, H(x))$, where H is a new symbol (Skolem function).

e) Drop universal quantifiers
- For instance, $\forall x$ Person (x) becomes Person(x).

f) Distribute $\wedge$ over $\vee$:
- $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$.

2. Negate conclusion S and convert the result to CNF

3. Add the negated query to the set of premises (negated conclusion S).

4. Repeat untill contradiction or no progress is made:
   a) Select 2 parent clauses (any 2 clauses).
   b) Resolve them together, performing all required unifications.
   c) If resolvent is empty clause, a contradiction has been found.
   (i.e., S followed the pemise).
   d) If not, add resolvent to premises.

When succeeded in step 4, we have proved the conclusion.

Proof by Resolution :

Given KB or premise :

a) John likes all kinds of food.
b) Apple and vegetables are food.
c) Anything anyone eats and not killed is food.
d) Anil eats peanuts and still alive.
e) Marry eats everything that Anil eats.
f) Anyone who is alive implies not killed.
g) Anyone who is not killed implies alive.

To prove :

h) John likes peanuts.

I) First Order Logic :

a) $\forall x : food(x) \rightarrow likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d) $eats(Anil, peanuts) \wedge alive(Anil)$.
e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f) $\forall x : \neg killed(x) \rightarrow alive(x)$
g) $\forall x : alive(x) \rightarrow \neg killed(x)$
h) $likes(John, Peanuts)$.

II) Eliminate implications :

a) $\forall x ; \neg food(x) \vee likes(John, x)$
b) $food(Apple) \wedge food(vegetables)$
c) $\forall x \forall y \rightarrow [eats(x, y) \wedge \neg killed(x)] \vee food(y)$.

d.) eats (Anil, Peanuts) $\land$ alive (Anil)
e.) $\forall x \neg$ eats (Anil, x) $\lor$ eats (Harry, x)
f.) $\forall x \neg$ [$\neg$ killed (x)] $\lor$ alive (x)
g.) $\forall x \neg$ alive (x) $\lor \neg$ killed (x)
h.) likes (John, peanuts).

III.) Standardize variables:
c.) $\forall y \forall z \neg$ eats (y, z) $\lor$ killed (y) $\lor$ food (z)
e.) $\forall w \neg$ eats (Anil, w) $\lor$ eats (Harry, w)
f.) $\forall g$ killed (y) $\lor$ alive (y)
g.) $\forall k \neg$ alive (k) $\lor \neg$ killed (k)

IV.) Drop universal quantifiers.
a.) $\neg$ food (x) $\lor$ likes (John, x)
b.) food (Apple)
c.) food (Vegetables)
d.) $\neg$ eats (y, z) $\lor$ killed (y)
e.) eats (Anil, Peanuts)
f.) Alive (Anil)
g.) $\neg$ eats (Anil, w) $\lor$ eats (Harry, w)
h.) killed (y) $\lor$ alive (y)
i.) $\neg$ alive (k) $\lor \neg$ killed (k)
j.) likes (John, Peanuts).

Output: John likes Peanuts.

¬ likes(John, Peanuts)   ¬ food(x) ∨ likes(John,x)

¬ food(Peanuts)   ¬ eats(y,z) ∨ killed(y) ∨ food(z)

¬ eats(y, Peanuts) ∨ killed(y)   eats(Anil, peanuts)

Killed(Anil)   ¬ alive (k) ∨ ¬ killed(k)

¬ alive   alive (Anil)

{}

Hence, proved.

S. Pradeep
3/12/24

Lab-10
Alpha-Beta Pruning Algorithm.

Algorithm:

$\alpha \leftarrow -\infty$ // Initialize α to negative infinity.
$\beta \leftarrow +\infty$ // Initialize β to $+\infty$
$V \leftarrow$ max_value (state, α, β)
return the action a in ACTIONS (state) with value V.

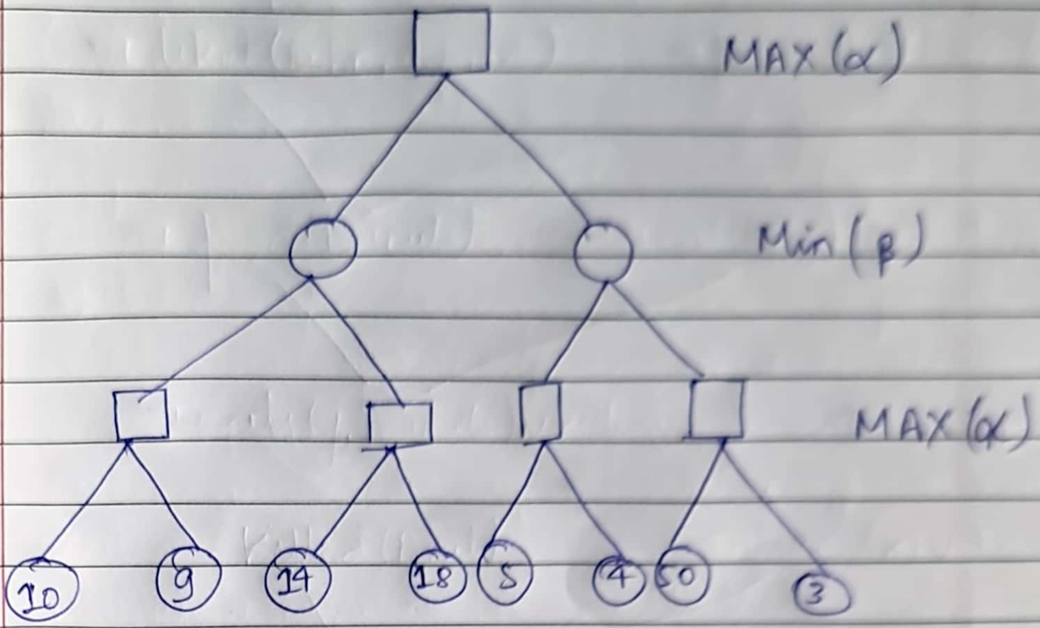
Max-value function (state, α, β)
  if TERMINALTEST (State):
    return UTILITY (state)
  $V \leftarrow -\infty$

  for each action a in ACTIONS(State):
    $V \leftarrow$ MAX (V, MIN-VALUE (RESULT (state, a), α, β))

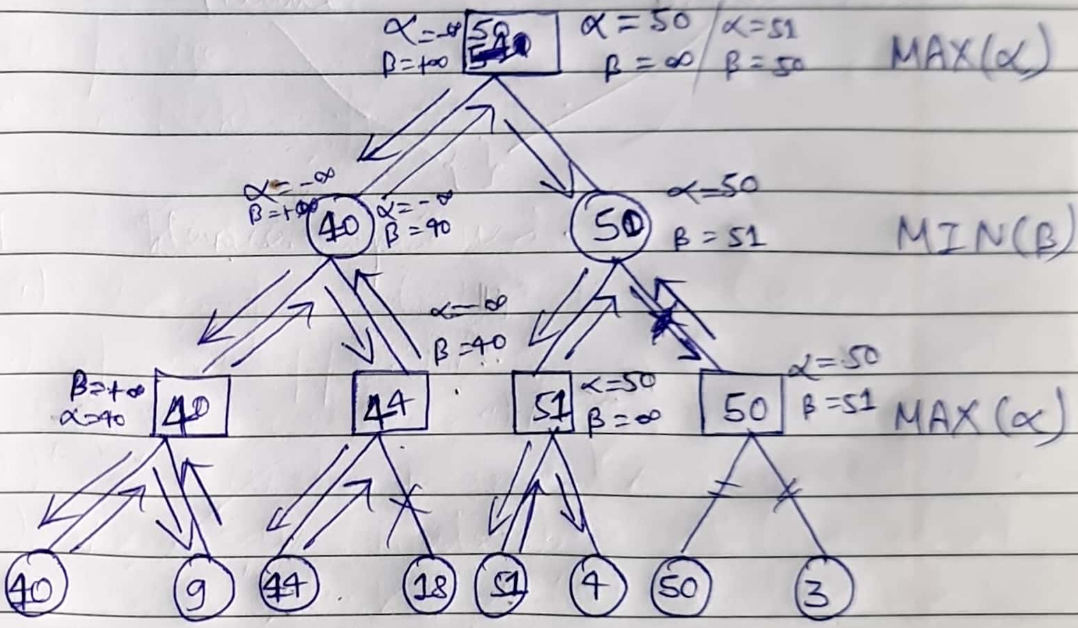
  if $V \geq \beta$
    return V
  $\alpha \leftarrow$ MAX (α, v)
return V

function Min-Value (state, α, β).
  if TERMINAL-TEST (state)
    return UTILITY (state)
  $V \leftarrow +\infty$
for each action a in ACTIONS (state) :
  $V \leftarrow$ MIN (V, MAX-Value (RESULT (state, a), α, β))
If $V \leq \alpha$ then return V
  $\beta \leftarrow$ Min (β, V)
return V

Problem



Solution:



Output is
Enter the value of leaf nodes:
10 9 14 18 5 4 50 3

Optimal value : 10