INDEX

Name : Manav Kumar Subject : AI Lab

Std. : Semester: 5 Div. : Section: 5C Roll No. : 1BM22CS348

School / College : BMSCE

Sl No.	Date	Title	Page No.	Teacher Sign/ Remarks
01.	01/10/24	Implement Tic-tac-toe game.	01-03	Nanaste. M. 15/10/2024
		Implement Vaccum cleaner agent		
02	08/10/24	8-Puzzle problem		
		Iterative deepening search algorithm.	09-11	Nanaste M. 15/10/2024
03.	15/10/24	A* algorithm for 8-puzzle.	12-15	Nanaste. M. 15/10/2024
04.	22/10/24	Hill climbing search for N-Queens's.	17-20	Nanaste. M. 29/10/2024
05.	29/10/24	8-queen's problem using simulated Annealing.	21.	Nanaste - M. 19/11/2024
06.	12/11/24.	Propositional Logic.	22-23.	
07.	19/11/24	English language to FOL.	24.	Nanaste. M. 3/12/2024
08	26/11/24	Unification in FOL.	25-26	
09.	26/11/24	FOL- forward Reasoning.	27-29	
10.	03/12/24	Resolution	30-32	
11.	17/12/24	Alpha-beta pruning	33-34	

Date: 01/10/2024:-

Q. Implement tic-tac toe game.

#). Program title:
Implement tic-tac toe game.

#). Algorithm:

```
function print_board(board):
    print board format

function check_winner(board):
    for each row in board:
        if all elements in row are same and not empty:
            return the element
    for each column:
        if all elements in column are same and not
            empty:
            return the element
    if diagonal checks yield same element and not empty
        return the element
    return None

function is_board_full(board):
    return True if no empty spaces, else False

function main():
    initialize board with empty spaces
    iteration = 0
    winner = None

    while winner is None:
        if iteration is even:
            print_board(board)
            get user input for row and column
```

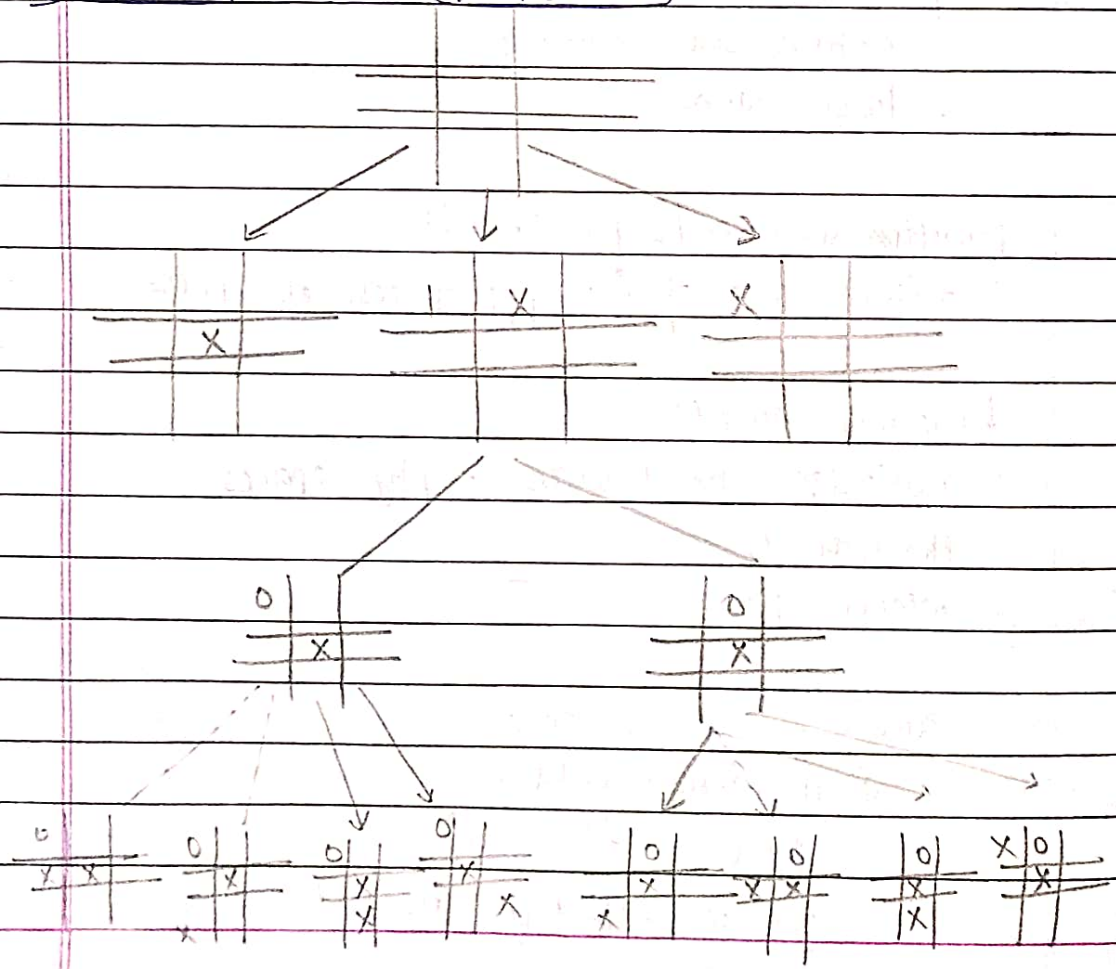
validate input
place 'O' in board
else:
randomly select empty position for 'X'.

iteration += 1
winner = check_winner(board)

if winner is not None:
print winner message
break

print game introduction
main().

State space tree (Tic Tac Toe):



Date: 01/10/2024

(#) Program title:
Implement Vacuum Cleaner Agent

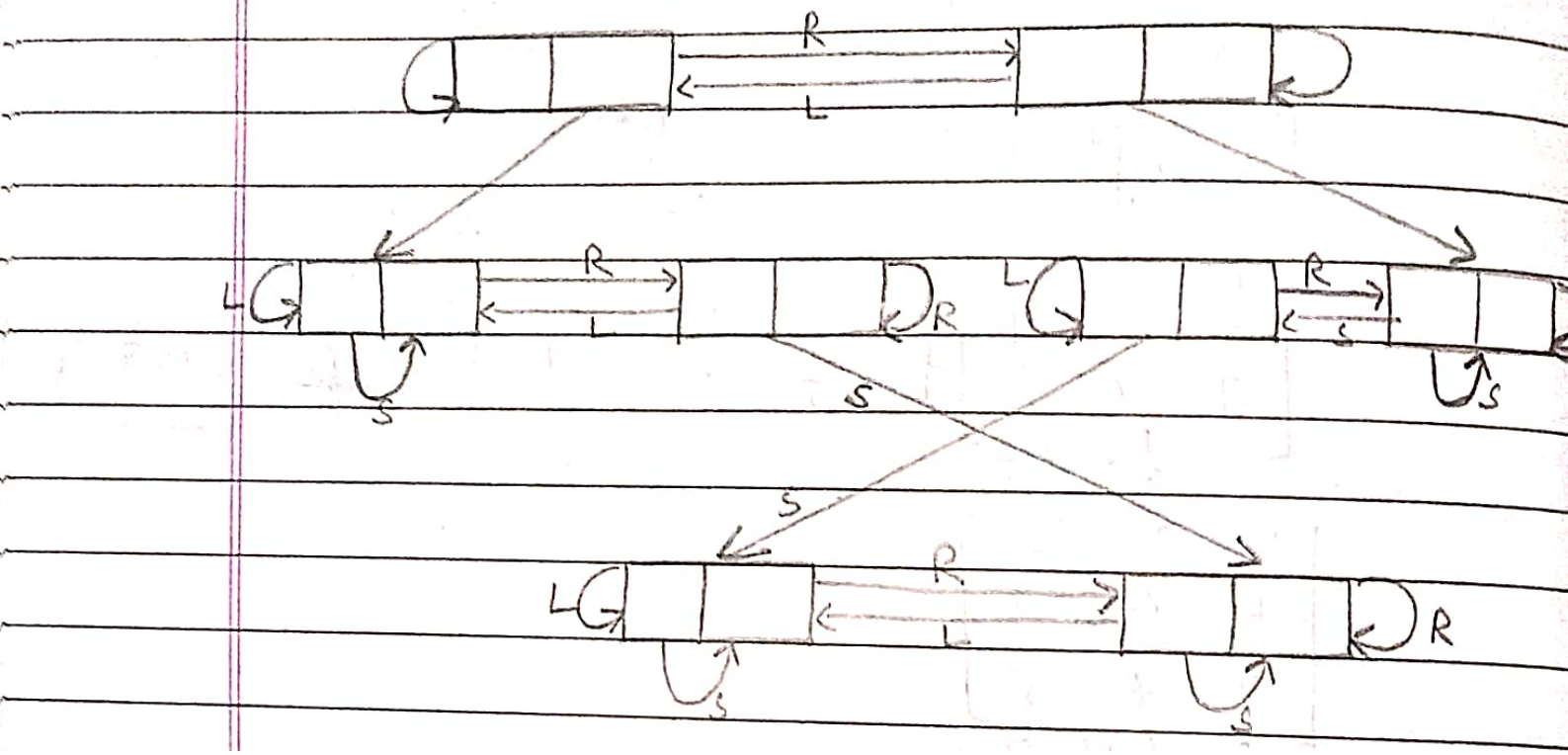
(#) Algorithm:

```
function VacuumCleanerAgent(environment):
    position = (0,0)
    cleaned_cells_count = 0

    while True:
        if environment[position] is dirty:
            clean(environment[position])
            cleaned_cells_count += 1
            print("cleaned position:", position)

        next_position = findNextDirty(environment)
        if next_position exists:
            position = next_position
        else:
            print("No more dirty cells found. Cleaning complete")
            break

function findNextDirty(environment):
    for each cell in environment:
        if cell is dirty:
            return cell's position.
    return None.
```

1/10/24

Mangal
Date
Page

#) State space Tree (Vaccum cleaner):



Date: 08/10/24

① Solve - 8 puzzle problem.

Pseudocode:

```
class Node:
    function - init - (state, parent, action, path-cost = 0):
        set self. state = state
        set self. parent = parent
        set self. actoin = action
        set self. path-cost = path-cost
    function expand():
        create children
        set row, col = find-blank()
        create possible-actions
        if row > 0 then add 'up' to possible-actions
        if row < 2 then add 'dooon' to possible-actions
        if col > 0 then add 'left' to possible-actions
        if col < 2 then add 'right' to possible-actions
        for action in possible-actions:
            create new-state as a copy of self.state
            if action == 'up' then swap new-state [row]
            [col] with new-state [row] [col]
            else if action == 'down' then swap
            new-state [row] [col] with new-state [row+1] [col]
            else if action == 'left' then swap new-state
            [row] [col] with new-state [row] [col-1]
            else if action == 'right' then swap
            new-state [row] [col] with new-state [row]
            [col+1]

            append new Node (new-state, self, action, self.path-cost+1)
            to children
        return children.
```

```
function find-blank():
  from row from 0 to 2:
    for col from 0 to 2:
      if self.state[row][col]==0 then
      return row, col

function depth-first-search(initial-state, goal-state):
set frontier = [Node(initial-state)]
  set explored= empty set
  while frontier is not empty:
    set node= frontier.pop()
    if node.state == goal-state then
    return node
  add tuple of node state to explored
  for child in node expand()%
    if tuple of child state not in
    explored then append child to frontier
  return none
  function print-solution(node):
    create path
    while node is not none:
      append (node action, node state)
      to path
      set node= node parent
      reverse path
      for (action, state) in path:
      if action is not none then print
      "action:", action
        print state
        print " "
```

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

(2) Implement iterative deepening search algorithm.

→
```
function iterative_deepening_search(initial_state, goal_state, max_depth):
    for depth from 0 to max_depth:
        set result = depth_limited_search(initial_state, goal_state, depth)
        if result is not none then
            return result
    return none
function depth_limited_search(node, goal_state, limit):
    if node.state == goal_state then
        return node
    if node.depth >= limit then
        return none
    for each child in expand(node):
        set result = depth_limited_search(child, goal_state, limit)
        if result is not none then
            return result
    return none
set initial_state, goal_state, max_depth
set solution = iterative_deepening_search(initial_state, goal_state, max_depth)
if solution is not none then print solution
else print "NO solution found".
```

(#). State space tree (8-puzzle problem):

Nandeesh M
22/10/2024

Date: 15/10/24:-

Q. A* algorithm for 8- Puzzle problem:

$f(n) = g(n) + h(n)$

Pseudocode & Algorithm:

To implement the A* algorithm for the 8-puzzle problem, we define two heuristics, h(n):

1. No. of Misplaced Tiles:
2. Manhattan Distance

The A* algorithm will use $f(n) = g(n) + h(n)$, where:

g(n) is the depth of the node.

h(n) is the heuristic, either misplaced tiles or Manhattan distance.

Algorithm:

1. Initialize:

→ Start with the initial state of the puzzle.

→ Use a priority queue to hold the nodes, ordered by $f(n) = g(n) + h(n)$.

→ Keep a set of visited nodes to avoid re-exploring nodes.

2. Expand Nodes:

→ While the priority queue is not empty.

→ Remove the node with the lowest f(n) from the queue.

→ If this node is the goal, return the solution.

→ Generate all possible child nodes (neighbouring states) by moving the blank tile.

For each child:

Calculate g(n) (increment depth by 1 from parent)

calculate h(n) using the chosen heuristic.

If the child has not been visited or if it has a better f(n) than before, add it to the queue.

3. Termination:

→ If the goal is found, return the path and total cost.

→ If no solution is found, return failure.

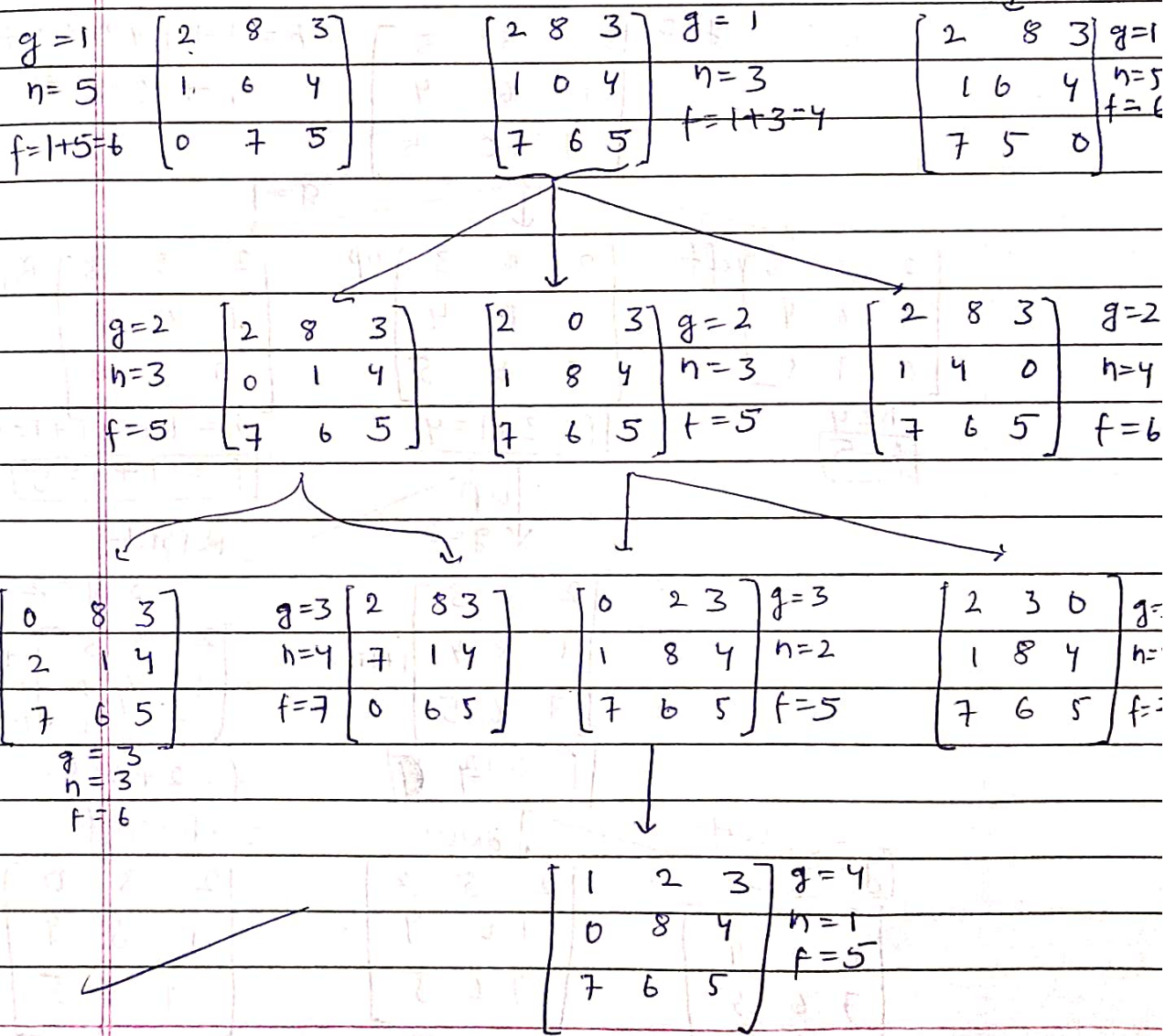
4. State space tree (misplaced tiles):

Final state:

1	2	3
8	0	4
7	6	5

Initial state:

2	8	3
1	6	4
7	0	5



1	2	3	g=4
0	8	4	h=1
7	6	5	f=5

1	2	3	g=5
8	0	4	h=0
7	6	5	f=5

1	2	3	g=5
7	8	4	h=2
0	6	5	f=7

Final state.

★. State space tree (Manhattan distance):

Initial state:

2	8	3
1	6	4
7	0	5

Final state:

1	2	3
8	0	4
7	6	5

g=0

2	8	3
1	6	4
7	0	5

h = 1+1+0+0+0+1+0+0+1 = 4

g=1

left

2	8	3
1	6	4
0	7	5

h=4

f=5

up

2	8	3
1	0	4
7	6	5

f = 3+1 = 4

Right

2	8	3
1	6	4
7	5	0

h = 1+1+1+1 = 4

f = 1+4 = 5

up

g=2

2	0	3
1	8	4
7	6	5

1+1+2

f = 2+4 = 6

Right

g=2

2	8	3
1	6	0
7	6	5

1+1+0+1+1+2

6

f = 2+6 = 8.

g=3

Left

0	2	3
1	8	4
7	6	5

1+1 = f = 3+2 = 5

down

2	8	3
1	0	4
7	6	5

f = 3+3 = 6.

Right

2	3	0
1	8	4
7	6	5

Mangal
Date
Page

$g = 4$

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

~~$f = 1 + 1 = 2$~~

~~$f = 4 + 2 = 6$~~ . . .

$g = 4$ | down

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

$n = 1$

~~$f = 4 + 1$~~ $= 5$

↓ Right

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

$n = 0$.

final stall.

Right

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

X.

~~$f =$~~ $4 + 4 = 8$.

Namrata. M.
15/10/2024

LAB-4

22/10/2024:

(#). Project title:-

Implement hill climbing search algorithm to solve N-Queens problem.

(##). Algorithm:-

function HILL-CLIMBING(problem) returns a state that is a local maximum

current ← MAKE-NODE(problem.INITIAL-STATE)

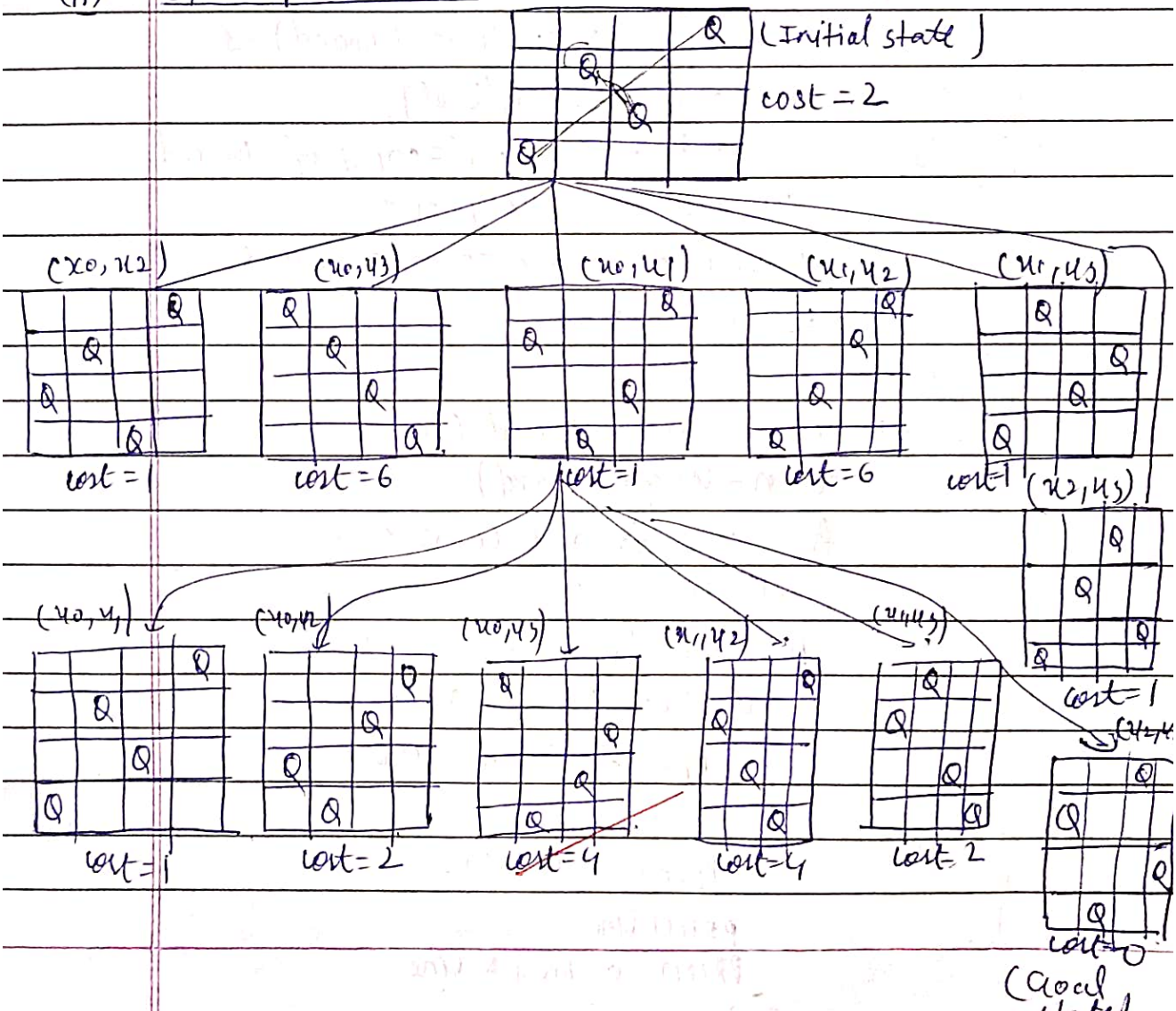
loop do

neighbour ← a highest-valued successor of current

if neighbour.VALUE < current.VALUE then return current.STATE

current ← neighbour.

(##). State space tree:-



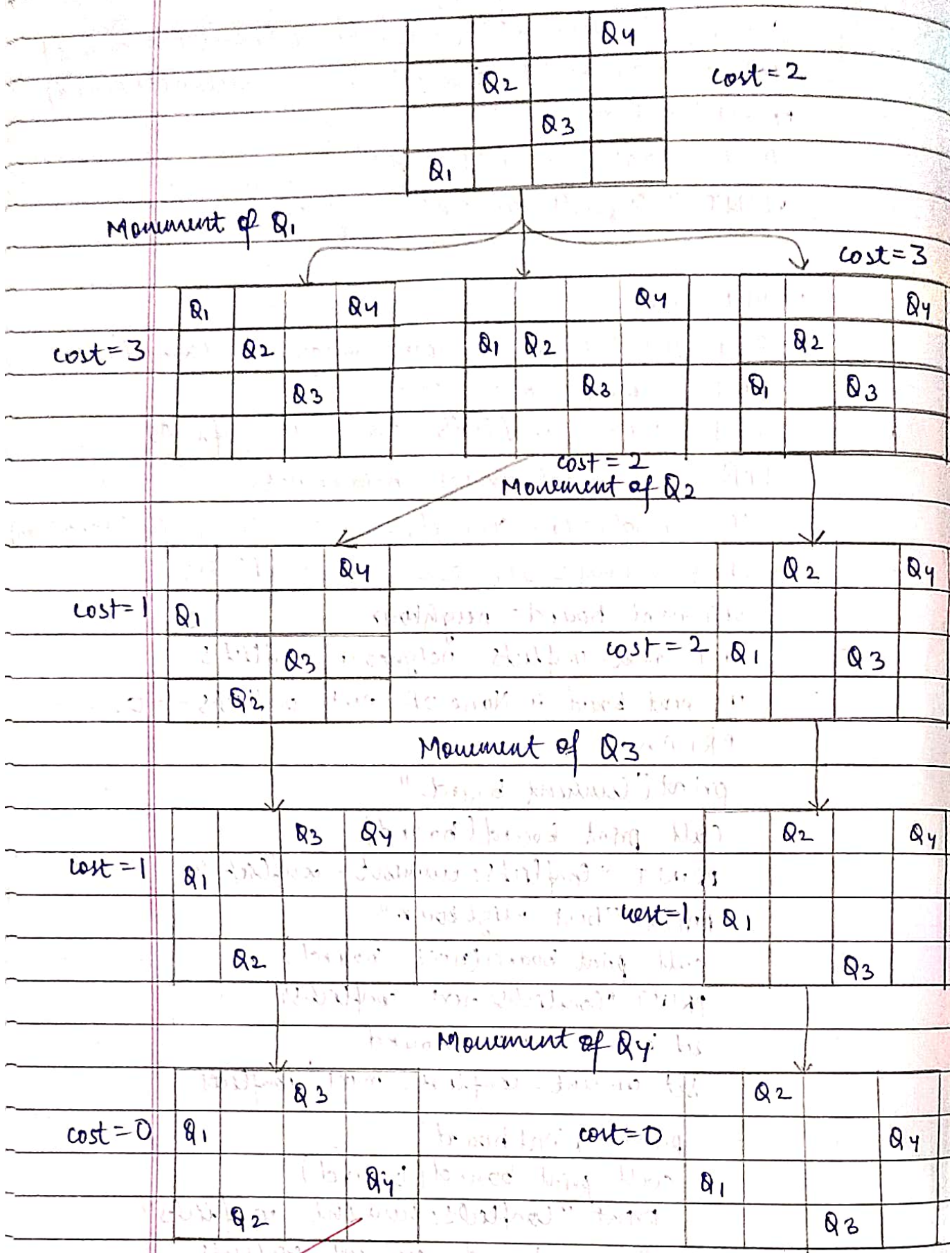
(##) ~~State space tree:~~

(##) **PSEUDOCODE:**

```
FUNCTION calculate-conflicts (board):
    Set n = length(board)
    Set conflicts = 0
    For i From 0 to n-1:
        For j from i+1 to n-1:
            if board[i] == board[j] OR
            ABS(board[i] - board[j]) == ABS (i-j):
                increment conflicts.
    Return conflicts

FUNCTION
generate_neighbours(board):
    SET neighbours = Empty list
    For col from 0 to length(board)-1:
    For row from 0 to length(board)-1:
        if row != board[col]:
            set new_board = copy of board
        Set new_board[col] = row
        Append new_board to neighbours
    Return neighbours.

FUNCTION print_board (board):
    Set n = length(board)
    for row from 0 to n-1:
    set line = " "
    For col from 0 to n-1:
        if board[col] == row:
            Append "Q" to line
        else
        Append "." to line.
        PRINT line
        PRINT a blank line
```

```
FUNCTION hill-climbing(n):
    Set board = list of n random integers (0 to n-1)
    set current-conflicts = calculate - conflicts(board)
    PRINT "Initial board:"
    CALL print-board(board)
    PRINT "conflicts: current-conflicts".

WHILE TRUE:
    SET neighbours = generate-neighbours(board)
    SET next-board = None
    SET next-conflicts = current-conflicts
    FOR each neighbour IN neighbours:
     SET neighbour-conflicts = calculate-conflicts(neighbour
     IF neighbour-conflicts < next-conflicts:
     SET next-board = neighbour
     SET next-conflicts = neighbour-conflicts
     IF next-board is None OR next-conflicts == 0:
      BREAK
      print("Current board:"
      Call print-board(board)
      PRINT "conflicts: current-conflicts",
      Print "Best neighbour:"
      call print-board(next-board)
      PRINT "conflicts: next-conflicts"
      set board = next-board
      set current-conflicts = next-conflicts
       print "final board"
       call print-board(board)
        print "conflicts: current-conflicts"
        Return board, current-conflicts
set n=4
set solution, conflicts = CALL
hill-climbing(n)
```

(##) State space tree:



Nameeta. M.
29/10/2024

LAB-5

Date: 23/11/2024

Simulated Annealing Algorithm:

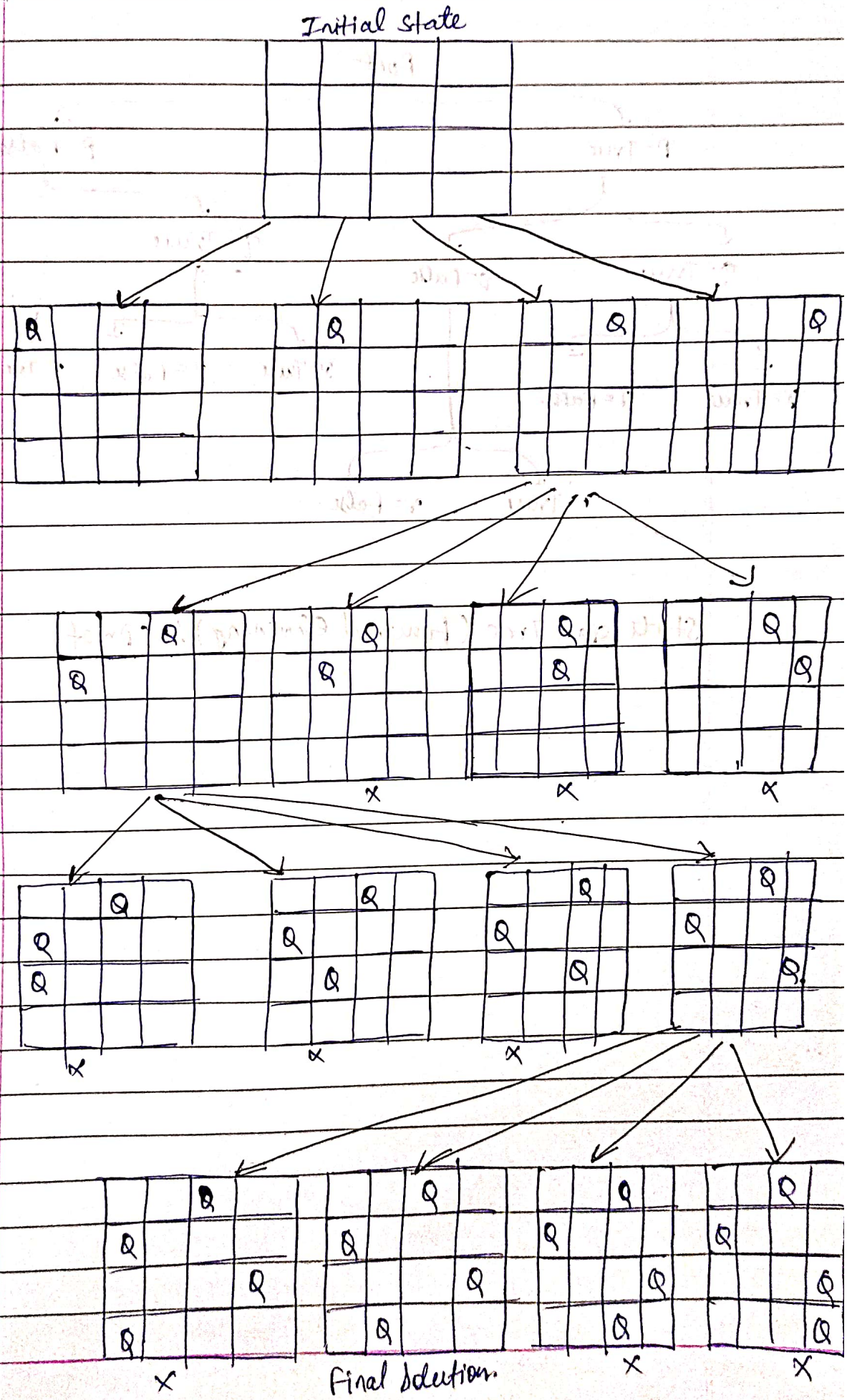
(#) Project title: 8 queen's problem using simulated Annealing Algorithm.

→ The algorithm can be decomposed in 4 simple steps:

1. Start at a random point x.
2. Choose a new point x_j on a neighbourhood $N(x)$.
3. Decide whether or not to move to the new point x_j. The decision will be made based on the probability function $P(x, x_j, T)$.
4. $P(x, x_j, T)$ is the function that will guide us on whether we move to the new point or not:

$$P(x, x_j, T) := \begin{cases} 1 & \text{if } F(x_j) \geq F(x) \\ e^{\frac{F(x_j) - F(x)}{T}} & \text{if } F(x_j) < F(x). \end{cases}$$

State space Tree: (Simulated Annealing)



Lab-6

Manual
Page 22

Date: 12/11/2024:

(##). Project Title: Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

1. Input: The user enters the knowledge base (KB) and the query (q).

2. Generate Truth combinations: Iterate through all combinations of truth assignments for the variables p, q and r (total 8 combinations).

3. Convert to postfix: For both the knowledge base and the query, convert the infix expressions to postfix notation.

4. Evaluate postfix Expressions: For each combination of truth values, evaluate both the knowledge base and the query.

5. Check Entailment: If there is any combination where knowledge base evaluates to True and the query evaluates to False, return False (indicating that the knowledge base does not entail the query). If no such combination is found, return True.

6. Output: Print whether the knowledge base entails the query.

Output:

Enter rule: p^ q^ r

Enter the query: P

Truth table Reference

KB	alpha
True	True
False	True
False	True
False	True
False	False
False	False
False	False
False	False

#Result: The knowledge base entails query.

Truth Space Table

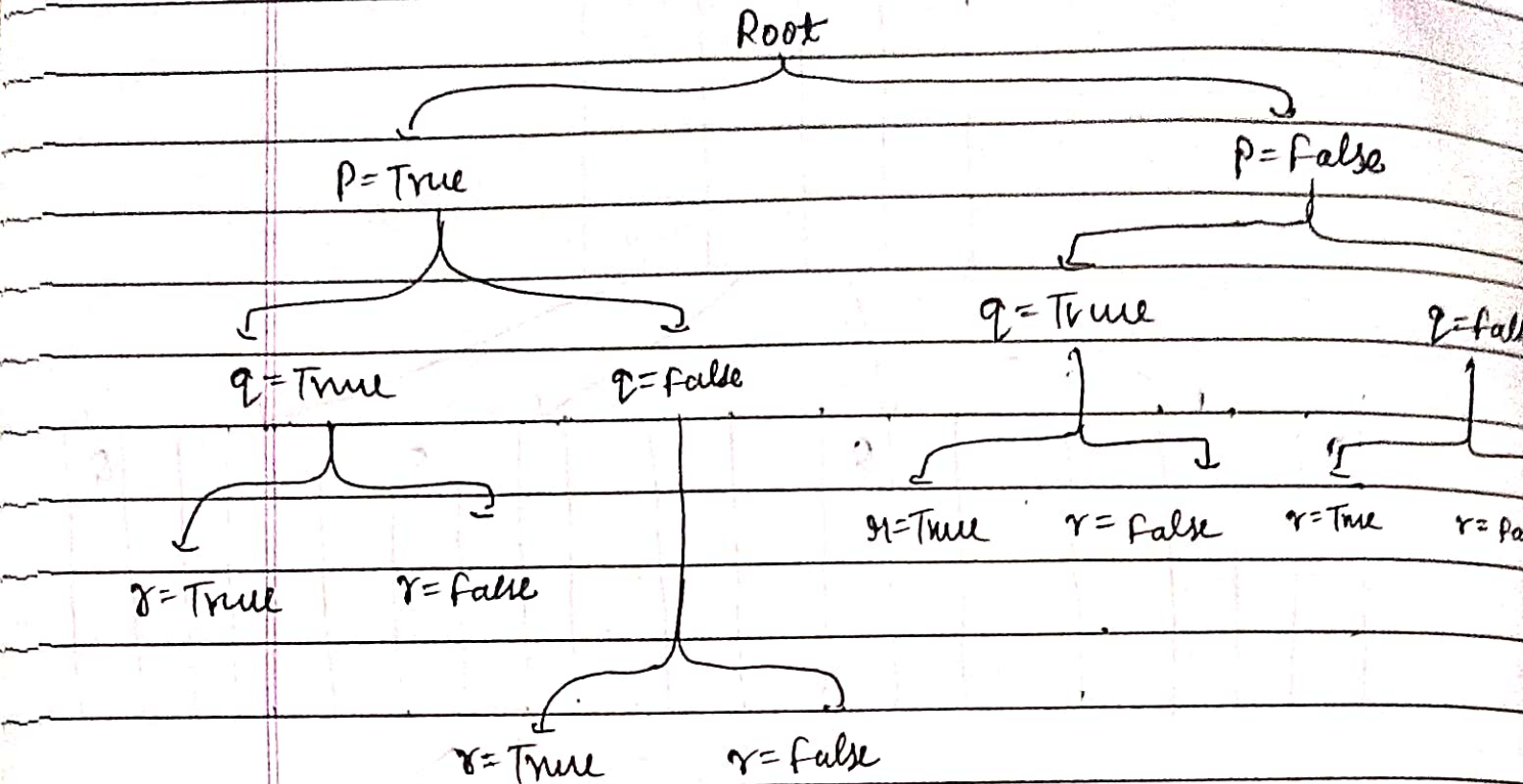
combination(P,q,R)	KB Evaluation	Query Evaluation	Entailment check
(T,T,T)	True	True	True
(T,T,F)	False	True	True
(T,F,T)	False	True	True
(T,F,F)	False	True	True
(F,T,T)	False	False	True
(F,T,F)	False	False	True
(F,F,T)	False	False	True
(F,F,F)	False	False	True.

Nandita.M
19/11/2024

#Result: The knowledge base does not entail query found KB: True and Query: False in state 2.

Mangal
Date
Page

State space Tree (propositional Logic)!



LAB-7

Manjal
Date
Page 24

Date: 19/11/2024.

Q. English language to FOL:

1. "John is a Human".
→ H(j)

2. "Every human is mortal".
→ ∀x(H(x) → M(x))

3. "John loves mary".
→ L(j, m)

4. "There is someone who loves mary".
→ ∃x(L(x, m))

5. "All dogs are animals."
→ ∀x(D(x) → A(x))

6. "Some dogs are brown".
→ ∃x(D(x) ∧ B(x))

Q. FOL to English:-

(a) ∀x(H(x) →

#) Algorithm:

1. Initialize predicates dictionary with natural phrase and FOL templates.
2. Initialize quantifiers dictionary with natural language quantifiers and FOL representation.
3. Convert sentence to lower case.
4. Split 'sentence' to 'words'.
5. Initialize 'fol' as an empty string.
6. For each 'quantifier' in 'quantifiers'
   - If quantifiers' is found in 'sentences':
     - Append its FOL representation to 'fol':
7. For each 'phrase' in 'predicates':
   - If 'phrase' is found in 'sentence':
     - split 'sentence' into 'parts' of 'phrases'.
     - Extract entities before & after the phrase.
     - Capitalize entities for proper noun.
     - Convert the predicate with entities mapped to FOL.
     - Break the loop.
8. If 'fol' is not empty return 'fol'.
9. Else, return "Could not map sentence to FOL".

#Output: There is no person who is both a bachelor & married.

FOL: not exists X(Bachelor(X) and married(X)).

LAB-7

Manjal
Date
Page 25

Date: 26/11/2024:

(##). Project title: Unification Algorithm for first-order logic.

→ Unification is used to match logical expressions by finding a substitution that makes them identical.

(##) Algorithm:-

Input: Two expressions E1 and E2

Output: A substitution θ or failure if E1 and E2 cannot be unified.

Step1: Initialize substitution:

start with an empty substitution θ = { }.

step2: Compare Expression:

Case 1: E1 and E2 are constants.

- If E1 = E2, return θ (no substitution)
- If E1 ≠ E2, return failure.

case 2: Either E1 or E2 is a variable

- Let X be a variable
  - If X ∉ θ (not yet substituted)
    - Add {X ↦ E} to θ, where 'E is the other term
    - Ensure no cyclic substitution (ex. X ≠ E1).
  - If X ∈ θ, recursively unify θ[X] with other term.

case 3: E1 and E2 are compound terms (eg f(x,y) & f(a,b))

- Decompose E1 and E2 into their functors and arguments;
  (ex. f(a,b) → functor: f, arguments: (a, b))
- If their functors differ, return failure.
- If their argument lengths differ, return failure.
- Otherwise, recursively unify each pair of arguments and accumulate substitutions.

Case 4: F1 or F2 is neither a variable nor a term.
- Return failure.

Step3: Apply substitutions
- After obtaining θ, apply it consistently to all variables in F1 and F2.

Step4: Repeat until fully unified or failure
- If θ is complete, return it as the result.
- If unification fails at any step, return failure.

LAB-8.

Date: 26/11/2024 :-

Page 27.

##. Project title :- Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

##. Forward Reasoning Algorithm:-

A conceptually straightforward, but very inefficient forward-chaining algorithm. On each iteration, iteration it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty
new ← { }
for each rule in KB do
$(p_1 \wedge \cdots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, $p_1 \wedge \cdots \wedge p_n$) = SUBST($\theta$, $p'_1 \wedge \cdots \wedge p'_n$)
for some $p'_1, \cdots p'_n$ in KB
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then

add q' to new
$\phi \leftarrow$ UNIFY (q', α)
if ϕ is not fail then return ϕ

add new to KB
return false

Given Case study:
As per the law, it is a crime for an American to sell weapons....

To prove: "Robert is a criminal".

Representation in FOL:

1. It is a crime for an American to sell weapons to hostile nations.
Let's say p, q and r are variables.
$American(p) \wedge weapon(q) \wedge Sells(p, q, r) \wedge Hostile(r) \Rightarrow Criminal(p)$.

2. Country A has some missiles.
$\exists x\ Owns(A, x) \wedge Missile(x)$
Existential Instantiation, introducing a new constant T1:
$Owns(A, T1)$
$Missile(T1)$

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

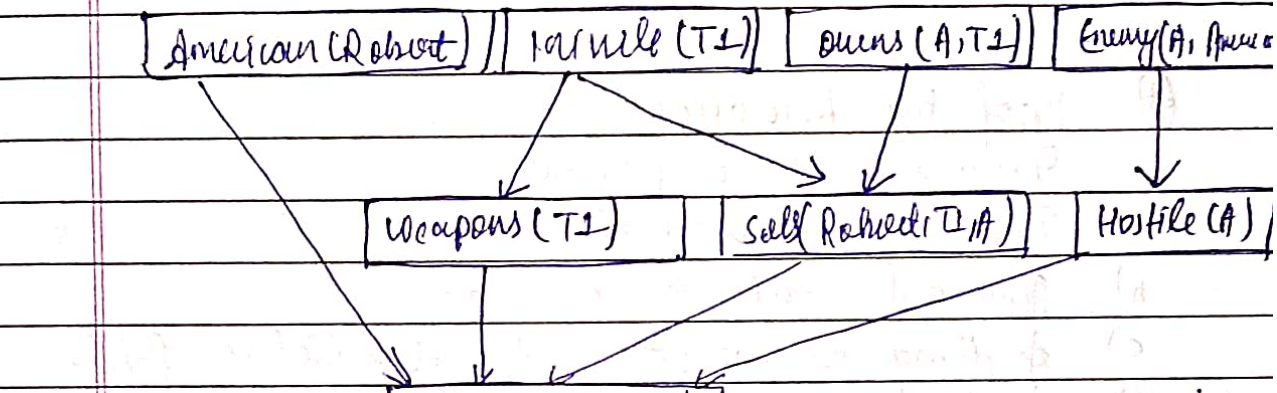
5. Enemy of America is known as hostile $\forall x$
Enemy $(x, \text{America}) \Rightarrow \text{Hostile}(x)$.

6. Robert is an American
American (Robert)

7. The country A, an enemy of America.
Enemy (A, America)

(##). Output:
It is 'criminal(Robert)' provable & True.

(##). Forward chaining proof:



American(p) $\wedge$ Weapon(q) $\wedge$
Sells (p, q, r) $\wedge$
Hostile(r)
$\Rightarrow$ Criminal (p).

Nandee. M.
3/12/2024

LAB-7

Mangal
Date
Page 30.

Date: 03/12/2024:

(#). Project title:- Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.

(#). Algorithm:-

1) Convert all sentences to CNF.
2) Negate conclusion S & convert result to CNF.
3) Add negated conclusion S to the premise clauses.
4) Repeat until contradiction or no progress is made.
   (a) Select 2 clauses (call them parent clauses)
   (b) Resolve them together, performing all required unification
   (c) If resolvent is the empty clause, a contradiction has been found.
   (d) If not, add resolvent to the premises. If we never stop, we have proved the conclusion

(#). Proof by Resolution:

Given the KB or premises

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

⇒ Representation in FOL:

a) $\forall x$: food(X) → likes(John, X)
b) food(Apple) ∧ food(vegetables)
c) $\forall x \forall y$: eats(X, y) ∧ ¬ killed(X) → food(Y)
d) eats(Anil, peanuts) ∧ alive(Anil)

c) $\forall x$: eats (Anil, x) → eats (Harry, x)
f) $\forall y$: ¬ killed(x) → alive(x)
g) $\forall x$: alive(x) → ¬ killed(x)
h) likes (John, peanuts)

→ Eliminate implications:

a) $\forall x \neg$ food (x) V likes (John, x)
c) $\forall x \forall y \neg [\text{eats}(x, y) \wedge \neg \text{killed}(x)] \vee \text{food}(y)$
e) $\forall x \neg$ eats (Anil, x) V eats (Harry, x)
f) $\forall x \neg [\neg$ killed (x) $]$ V alive (x)
g) $\forall x \neg$ alive (x) $\vee \neg$ killed (x)

→ Move negation (¬) inwards:

c) $\forall x y \neg$ eats $(x, y) \vee$ killed $(x) \vee$ food (y)
f) $\forall x$ killed $(x)] \vee$ alive (x)

→ Standardize variables:

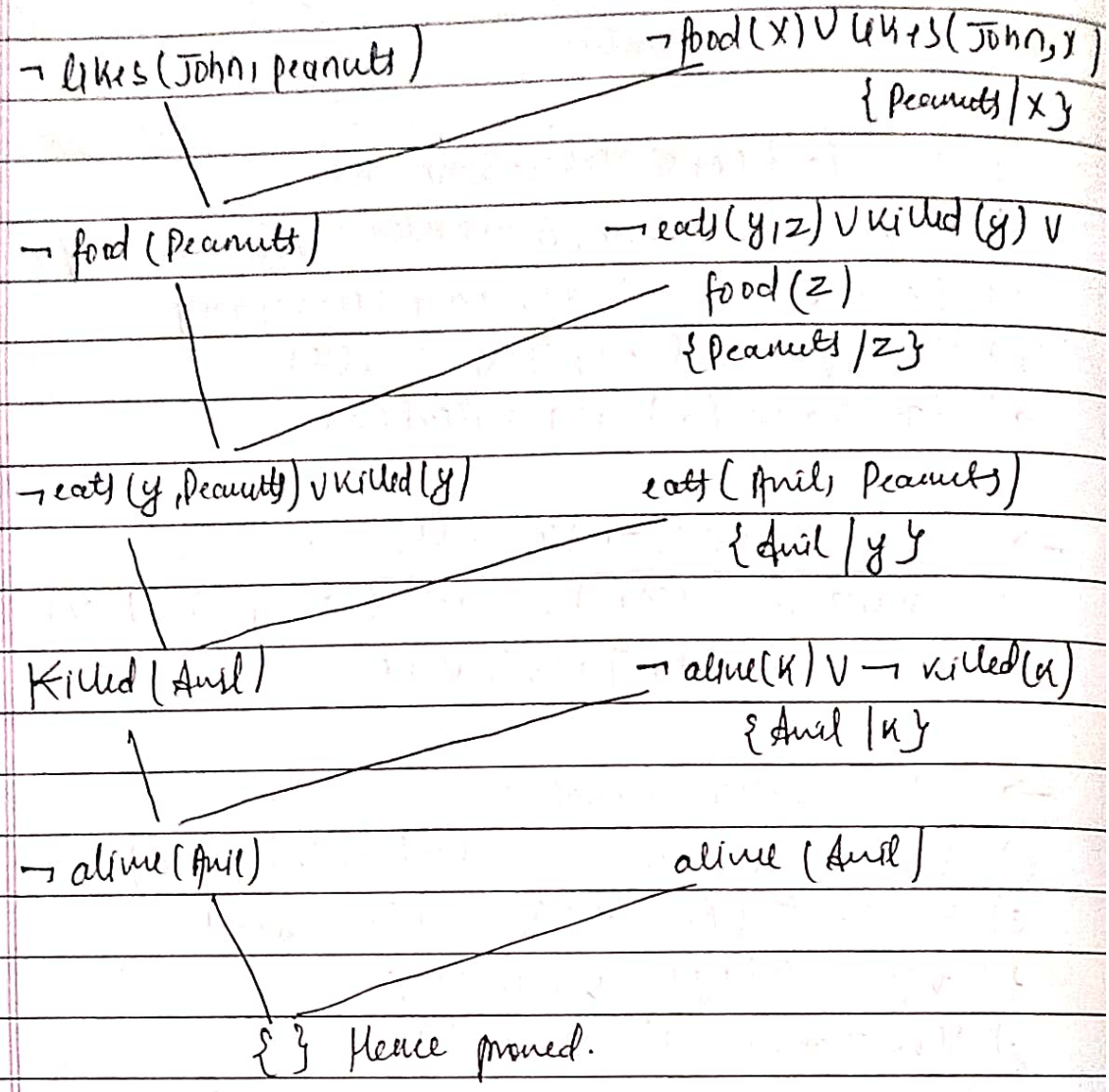
c) $\forall y \forall z$ eats $(y, z) \vee$ killed $(y) \vee$ food (z)
e) $\forall w \neg$ eats (Anil, w) V eats (Harry, w)
f) $\forall g$ killed $[g]] \vee$ alive (g)
g) $\forall k \neg$ alive $(k) \vee \neg$ killed (k)

→ Drop universal:

a) ¬ food (x) V likes (John, x)
b) food (Apple)
c) food (vegetables)
d) ¬ eats $(y, z) \vee$ killed $(y) \vee$ food (z)
e) eats (Anil, peanuts)
f) alive (Anil)
g) ¬ eats (Anil, w) V eats (Harry, w)

h) killed(g) V alive(g)
i) ¬ alive(k) V ¬ killed(k)

(##) Proof:



LAB-10

Mangal
Date
Page 33

Date: 17/12/2024

(#). Project title - Adversial search, Implement Alpha-Beta Pruning.

(#). Alpha Beta Pruning (cutoff) search Algorithm:-
A modified variant of the minmax method is alpha-beta pruning. It's a way for improving the minmax algorithm.
It involves two threshold parameters, Alpha and Beta, for future expansion.

(#). Algorithm:
↳ Alpha(α) - Beta (β) purposes to find the optimal path without looking at every node in the game tree.

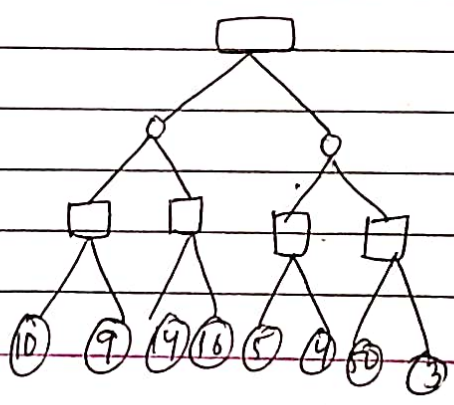
↳ Max contains Alpha(α) and MIN contains Beta(β) bound during the calculation.

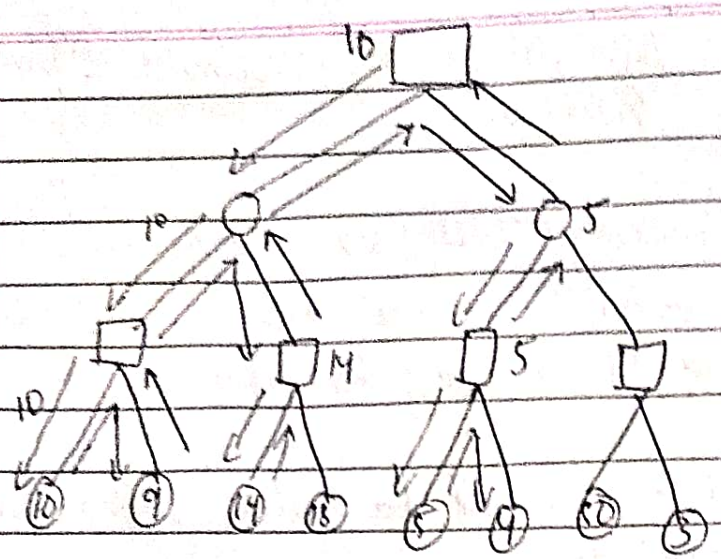
↳ In both MIN and MAX node, it returns when $\alpha \geq \beta$ which compares with its parent node only.

↳ Both minmax and Alpha(α) - Beta(β) cut-off give same path.

↳ Alpha(α) - Beta(β) gives optimal solution as it takes less time to get the value for the root node.

(#). Problem:




10
MAX[α]
10
5
MIN[β]
10
M
5
MAX[α]
10
9
14
13
5
4
12
5