



SRM INSTITUTE OF SCIENCE & TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

18CSC305J-ARTIFICIAL INTELLIGENCE

SEMESTER – 6

BATCH-2

REGISTRATION NUMBER	RA2011003011349
NAME	ABHAY GARG

INDEX

Ex No	DATE	Title	Page No	Marks
1	21-01-2021	Toy Problem:Selling movie tickets problem	3	10
2	29-01-2021	Graph Coloring Problem	7	8
3	05-02-2021	Constraint Satisfaction Problems	14	8
4	12-02-2021	Range Sum of Nodes in BST (DFS & BFS)	25	10
5	26-02-2021	Cut Off Trees for Golf Event (BFS & A*)	31	9
6	05-03-2021	Minimax Algorithm in Alpha-Beta Pruning	36	10
7	13-04-2021	Unification and Resolution Problem	40	10
10	12-03-2021	Implementation of block world Problem	56	10
8	21-04-2021	Rule Based Inference	69	
9	28-04-2021	Implementation of uncertain methods for an application	77	

LAB : 1

DATE :

TOY PROBLEM

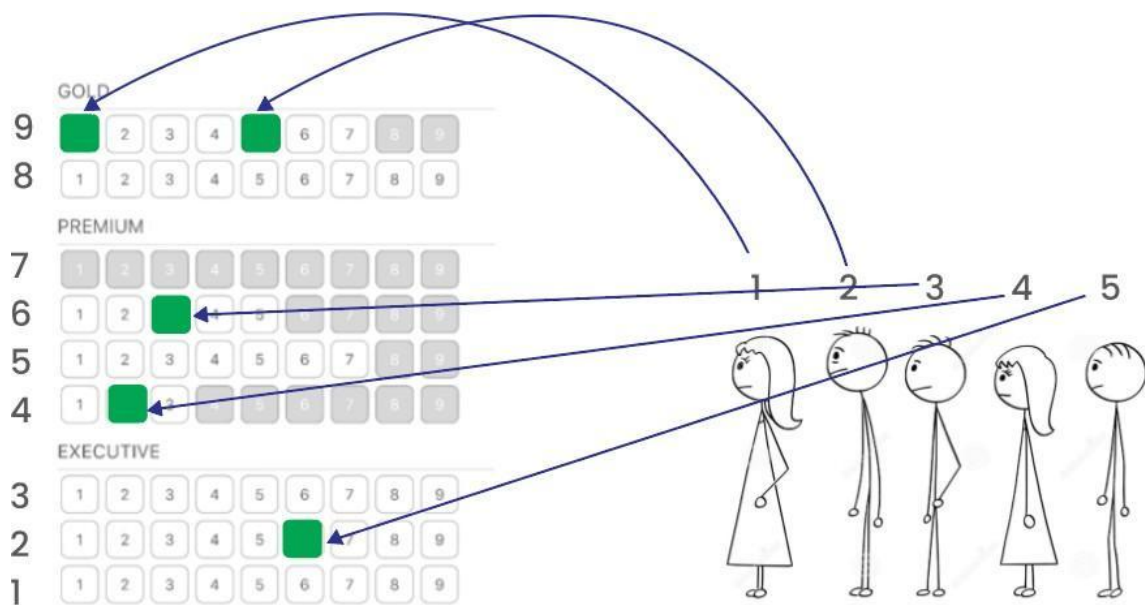
Problem Statement : Given an integer N and an array of seats[] where N is the number of people standing in a line to buy a movie ticket and seat[i] is the number of empty seats in the i th row of the movie theater. The task is to find the maximum amount a theater owner can make by selling movie tickets to N people. Price of a ticket is equal to the maximum number of empty seats among all the rows.

Algorithm :

1. Initialize queue q insert all seats array elements to the queue.
2. Tickets sold and the amount generated to be set to 0.
3. If tickets sold $< N$ (People in the queue) and q top > 0
4. Then remove top element from queue and update total amount
5. Repeat step 3 and 4 until tickets sold = number of people in the queue.

Optimization technique : This problem can be solved by using a priority queue that will store the count of empty seats for every row and the maximum among them will be available at the top.

1. Create an empty priority_queue q and traverse the seats[] array and insert all elements into the priority_queue.
2. Initialize two integer variable ticketSold = 0 and ans = 0 that will store the number of tickets sold and the total collection of the amount so far.
3. Now check while ticketSold $< N$ and $q.top() > 0$ then remove the top element from the priority_queue and update ans by adding top element of the priority queue. Also store this top value in a variable temp and insert temp - 1 back to the priority_queue.
4. Repeat these steps until all the people have been sold the tickets and print the final result.



Priority Queue

[2, 6, 9, 4, 9] ↗ Filling in Queue
5 3 2 4 1 ↖ Actual Priority

■ = Available Seats

Normal Queue

[1, 2, 3, 4, 5]

Simple FIFO approach

Tool : VS Code and Python 3.9.0

Programming code :

```
def maxAmount(M, N, seats):
```

```
    q = []
```

```
    for i in range(M):
```

```
        q.append(seats[i])
```

```
ticketSold = 0
```

```
ans = 0
```

```
q.sort(reverse = True)
```

```
while (ticketSold < N and q[0] > 0):  
    ans = ans + q[0]  
    temp = q[0]  
    q = q[1:]  
    q.append(temp - 1)  
    q.sort(reverse = True)  
    ticketSold += 1
```

```
return ans
```

```
if __name__ == '__main__':
```

```
    seats = []
```

```
    rows = int(input("Enter number of rows available : "))
```

```
    for i in range(0, rows):
```

```
        empty = int(input())
```

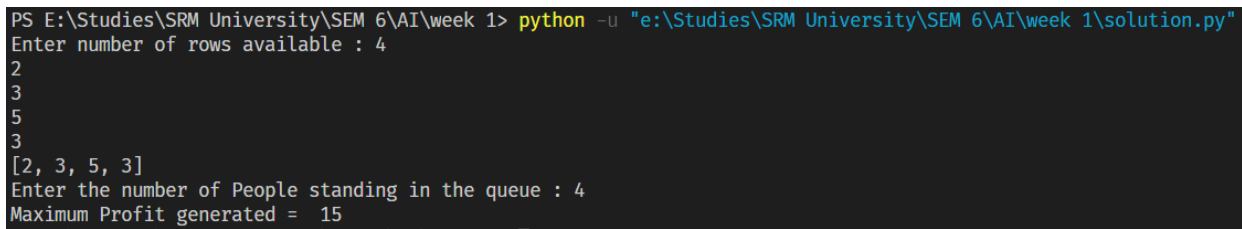
```
seats.append(empty)

print(seats)

M = len(seats)

N = int(input("Enter the number of People standing in the queue : "))
print("Maximum Profit generated = ", maxAmount(N, M, seats))
```

Output screen shots :



```
PS E:\Studies\SRM University\SEM 6\AI\week 1> python -u "e:\Studies\SRM University\SEM 6\AI\week 1\solution.py"
Enter number of rows available : 4
2
3
5
3
[2, 3, 5, 3]
Enter the number of People standing in the queue : 4
Maximum Profit generated = 15
```

Result : Successfully found out the maximum amount the theater owner can make by selling movie tickets to N people for a movie.

LAB : 2

DATE :

GRAPH COLORING PROBLEM

PROBLEM STATEMENT : Given a graph color its edges such that no two adjacent have the same color using minimum number of colors and return the Chromatic number.

ALGORITHM :

Initialize:

1. Color first vertex with first color.

Loop for remaining $V-1$ vertices.:

1. Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously colored vertices adjacent to it.
2. If all previously used colors appear on vertices adjacent to v , assign a new color to it.
3. Repeat the following for all edges.
4. Index of color used is the chromatic number.

OPTIMIZATION TECHNIQUE:

Graph coloring problem is to assign colors to certain elements of a graph subject to certain constraints.

Vertex coloring is the most common graph coloring problem. The problem is, given m colors, find a way of coloring the vertices of a graph such that no two adjacent vertices are colored using

the same color. The other graph coloring problems like Edge Coloring (No vertex is incident to two edges of same color) and Face Coloring (Geographical Map Coloring) can be transformed into vertex coloring.

Chromatic Number: The smallest number of colors needed to color a graph G is called its chromatic number. For example, the following can be colored at least 2 colors.

TOOLS : VS Code, Python 3.9.0

CODE - EDGE COLORING :

```
import matplotlib.pyplot as plt

import networkx as nx

from matplotlib.patches import Polygon

import numpy as np

G = nx.Graph()

colors = {0:"red", 1:"green", 2:"blue", 3:"yellow"}

G.add_nodes_from([1,2,3,4,5])

G.add_edges_from([(1,2), (1,3), (2,4), (3,5), (4,5)])

nodes = list(G.nodes)

edges = list(G.edges)

color_lists = []
```



```

color_of_edge = []

some_colors = ['red','green','blue','yellow']


for i in range(len(nodes) + 1):

    color_lists.append([])

    color_of_edge.append(-1)


def getSmallestColor(ls1,ls2): i
= 1
while(i in ls1 or i in ls2): i =
i + 1
return i


#iterate over edges i
= 0

for ed in edges:

    newColor = getSmallestColor(color_lists[ed[0]],color_lists[ed[1]])

    color_lists[ed[0]].append(newColor)

    color_lists[ed[1]].append(newColor)

    color_of_edge[i] = newColor i =
i + 1


# Makin graph again

```

```

G = nx.Graph()

for i in range(len(edges)):
    G.add_edge(edges[i][0],edges[i][1],color=some_colors[color_of_edge[i]-1])

colors = nx.get_edge_attributes(G,'color').values()

nx.draw(G, edge_color=colors, with_labels=True, width=2)

plt.show()

```

OUTPUT :

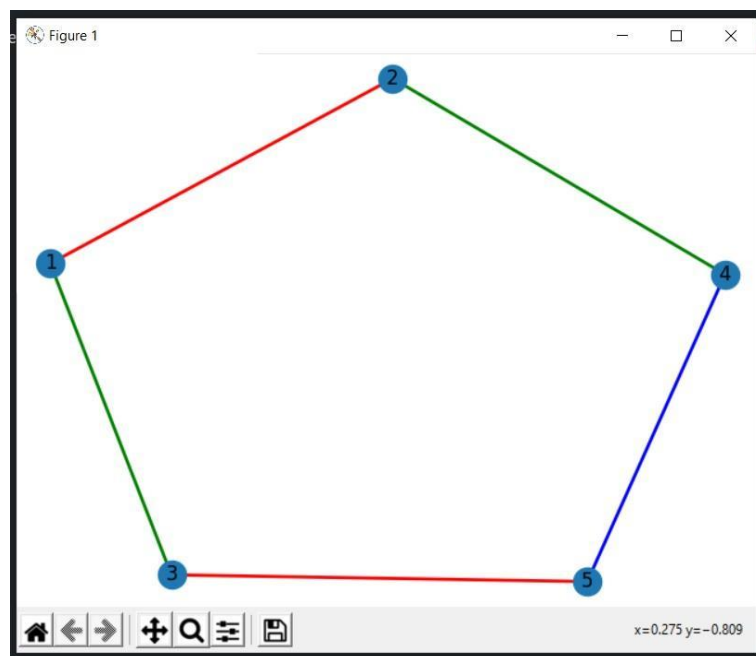


Fig: Edge Coloring

CODE - VERTEX COLORING :

```
import matplotlib.pyplot as plt
```

```
import networkx as nx
```

```
G = nx.Graph()
```

```
colors = {0:"red", 1:"green", 2:"blue"}
```

```
G.add_nodes_from([1,2,3,4,5])
```

```
G.add_edges_from([(1,2), (1,3), (2,4), (3,5), (4,5)])
```

```
d = nx.coloring.greedy_color(G, strategy = "largest_first")
```

```
node_colors = []
```

```
for i in sorted(d.keys()):
```

```
node_colors.append(colors[d[i]])
```

```
nx.draw(G, node_color = node_colors, with_labels = True, width = 5)
```

```
plt.show()
```

OUTPUT :

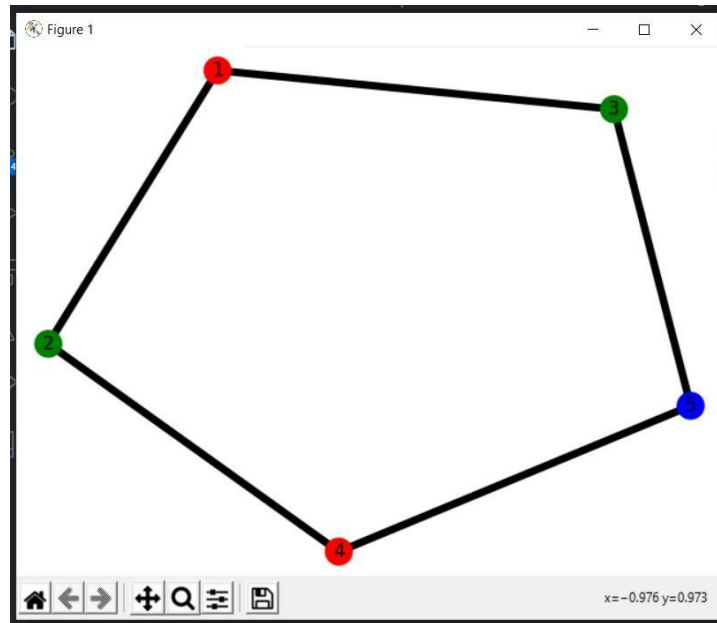


Fig : Vertex Coloring

CODE - FACE COLORING :

```
import networkx as nx

G = nx.Graph()

colors = {0:"red", 1:"green", 2:"blue", 3:"yellow"}

G.add_nodes_from([1,2,3,4,5])

G.add_edges_from([(1,2), (1,3), (2,4), (3,4), (4,5)])

nodes = list(G.nodes)

edges = list(G.edges)

some_colors = ['red','green','blue','yellow']

no_of_faces = len(edges)+2-len(nodes)-1

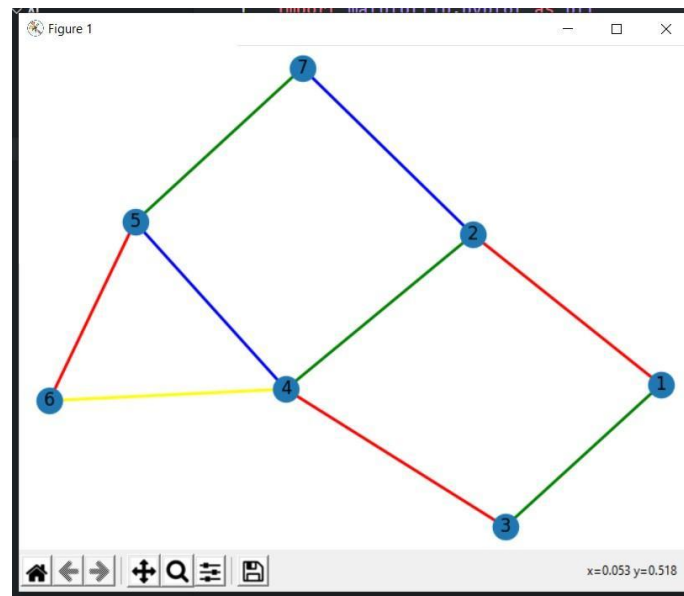
def regionColour(regions):
```

```

print("NO OF FACES : "+str(regions)) for i
in range(1,regions+1):
print("FACE    1    : "+some_colors[i%4])
regionColour(no_of_faces)

```

OUTPUT :



```

NO OF FACES : 3
FACE  1 : green
FACE  2 : blue
FACE  3 : yellow
PS E:\Studies\SRM University\SEM 6\AI> 

```

fig: Face Coloring

RESULT :

Edge, vertex and face coloring problem which are together known as graph coloring problem solved and visualized in an optimized way using greedy approach.

LAB : 3

DATE :

CONSTRAINT SATISFACTION PROBLEM

1) SEND + MORE = MONEY

5 4 3 2 1

S E N D

+ M O R E

c3 c2 c1

M O N E Y

1. From Column 5, $M=1$, since it is the only carry-over possible from the sum of 2 single digit numbers in column 4.
2. To produce a carry from column 4 to column 5 ' $S + M$ ' is at least 9 so ' $S=8 \text{ or } 9$ ' so ' $S+M=9 \text{ or } 10$ ' & so ' $O = 0 \text{ or } 1$ '. But ' $M=1$ ', so ' $O = 0$ '.
3. If there is carry from Column 3 to 4 then ' $E=9$ ' & so ' $N=0$ '.
But ' $O = 0$ ' so there is no carry & ' $S=9$ ' & ' $c3=0$ '.
4. If there is no carry from column 2 to 3 then ' $E=N$ ' which is impossible, therefore there is carry & ' $N=E+1$ ' & ' $c2=1$ '.
5. If there is carry from column 1 to 2 then ' $N+R=E \text{ mod } 10$ ' & ' $N=E+1$ ' so ' $E+1+R=E \text{ mod } 10$ ', so ' $R=9$ ' but ' $S=9$ ', so there must be carry from column 1 to 2. Therefore ' $c1=1$ ' & ' $R=8$ '.
6. To produce carry ' $c1=1$ ' from column 1 to 2, we must have ' $D+E=10+Y$ '

2. BASE + BALL = GAMES

Assuming numbers can't start with 0, G is 1 because two four-digit numbers can't sum to 20000 or more. $SE+LL=ES$ or $1ES$.

If it is ES , then LL must be a multiple of 9 because SE and ES are always congruent mod 9. But LL is a multiple of 11, so it would have to be 99, which is impossible.

So $SE+LL=1ES$. LL must be congruent to 100 mod 9. The only multiple of 11 that work is 55, so L is 5.

$SE+55=1ES$. This is possible when $E+5=S$. The possibilities for ES are 27, 38, or 49.

$BA+BA+1=1AM$. B must be at least 5 because $B+B$ (possibly +1 from a carry) is at least 10.

If A is less than 5, then $A+A+1$ does not carry, a and A must be even. Inversely, if A is greater than 5, it must be odd. The possibilities for A are 0, 2, 4, 7, or 9.

0 does not work because M would have to be 1.

2 and 7 don't work because M would have to be 5. 9

doesn't work because M would also have to be 9.

So A is 4, M is 9, and B is 7. This leaves 38 as the only possibility for ES . The

full equation is:

7483

+ 7455

14938

3. TWO + TWO = FOUR

$F = 1$ for carry over $T \geq 5$.

'O' can't be 0 as R will be 0. So T can't be 5 so let $T \geq 6$

If $T = 6$, $O = 2$ and $R = 4$ and $W + W = U$ for W can't be 1,2,6,4. $W < 4$ +s to

avoid carry over. W can't be 3 as U will be 6.

So $T = 7$, so, O can be 4 or 5 depending on whether or $W + W > 10$. If O is 4 then R

= 8. W can't be 1, 2. So $W = 3$

If $W = 3$ then $U = 6$ hence

Here is one + answer:

7 3 4

+ 7 3 4

- - - -

1 4 6 8

4. CROSS + ROADS= DANGER

Solution:

C5C4C3C2C1

CROSS

+ROADS

DANGER

Since it is already mentioned that the carry value of resultaint cannot be 0 then

let's presume that the carry value of D is 1

We know that the sum of two similar values is even, hence R will have an even

value

Hence $S+S=R$ So R is an even number for sure.

So the value of R can be $(0, 2, 4, 6, 8)$

Value of R cannot be 0 as two different values cannot be allotted the same

Digit, (if $S=10$ then their sum = 20 carry forward 2 , then the value of $R=0$)

which is not possible.

IF $S=1$:

Not possible since D has the same value. IF

$S=2$

Then $R=4$ which is possible Hence $S=2$ and $R=4$

$$C4+C+R=A+10$$

$$C4+C+4=A+10$$

$C4+C>5$ (Being the value of carry will be generated when the value of C is

greater than 5

$$C=9$$

$$C1+S+D=E$$

$$C1+2+1=E$$

$$\text{Therefore } E=3$$

$$C4+C+R=A+10$$

$$C4+9+4=A+10$$

Therefore $A=3$ but it cannot be possible as $E=3$

Now let's Consider $S+D+C1=E$

$$2+1+0=3$$

Therefore $E = 3$ making $C_2 = 0$ since $2+1=3$

Now let's consider the equation again:

$$C+R+C_4 = A+10$$

$$9+4+0 = A+10$$

$$13 = A+10$$

Therefore $A = 3$ but $E = 3$ So

A is not equal to 3

Again considering $R = 6$ So $S = 3$ $C_4 = 0$

$$C+R+C_4 = A+10$$

$$9+6+0 = A+10$$

$$15 = A+10$$

Therefore $A = 5$

And $S+D+C_1 = E$

$$3+1+0 = E \text{ therefore } E = 4 \text{ and } C_2 = 0$$

Now considering the equation $R+O+C_3 =$

N

$$6+0+C_3 = N$$

So $6+0+C_3 <$ or equal to 3

Let $C_3 = 1$

Then $O <$ or equal to 2

That is $O = 0, 1, 2$

Let $O = 2$

Again considering $R+O+C_3 = N$

$$6+2+1=N$$

Hence $N=9$ but $C=9$ so N cannot be equal to 9.

Now let $N=8$ and $C=0$

Let us consider equation

$$O+A+C_2=G$$

Therefore $G=7$

Hence $D=1$ $S=3$ $A=5$ $G=7$ $C=9$ $O=2$ $E=4$ $R=6$ $N=8$

And $C_1=0$ $C_2=0$ $C_3=0$ $C_4=0$ $C_5=1$

Now verifying the above values in the equation we get:

$$C_5C_4C_3C_2C_1$$

CROSS

96233

ROADS

62513

Shape

DANGER

158746

5. If $AA + BB = ABC$

Explanation:

AA

$BB + CC$

ABC

The digits are distinct and positive. Let's first focus on the value A, when we add three 2 digit numbers the most you get is in the 200's (ex: $AA + BB + CC = ABC$ u $99 + 88 + 77 = 264$). From this, we can tell that the largest value of A can be 2.

So Either $A = 1$ or $A = 2$.

Now focus on value B, let's take the unit digit of the given question: $A + B + C = C$ (units). This can happen only if $A + B = 0$ (in the units) u A and B add up to 10.

Two possibilities: $11 + 99 + CC = 19C$ u (1) or $22 + 88 + CC = 28C$ u (2)

Take equation (2), $110 + CC = 28C$

Focus on ten's place, $1 + C = 8$, here $C = 7$. Then $22 + 88 + 77 = 187$

Thus, Equation (2) is not possible.

From Equation (1), $11 + 99 + CC = 19C$ u $110 + CC = 19C$ u $1 + C = 9$, then $C = 8$.

$11 + 99 + 88 = 198$ u hence solved $A = 1$, $B = 9$ and $C = 8$

$+ B + C = 18$

6. NO + GUN + NO = HUNT

Solution:

$$\begin{array}{r}
 \text{N O} \\
 + \text{G U N N} \\
 \text{O} \\
 \hline
 \end{array}$$

$$\begin{array}{r}
 \text{H U N T} \\
 \hline
 \end{array}$$

Here $H = 1$, from the NUNN column we must have “carry 1,” so $G = 9$, $U = \text{zero}$.

Since we have “carry” zero or 1 or 2 from the ONOT column, correspondingly we

have $N + U = 10$ or 9 or 8 . But duplication is not allowed, so $N = 8$ with “carry 2”

from ONOT. Hence, $O + O = T + 20 - 8 = T + 12$. Testing for $T = 2, 4$ or 6 , we

find only $T = 2$ acceptable, $O = 7$. So we have $87 + 908 + 87 = 1082$.

$$\text{HUNT} = 1082$$

TOOLS : VS Code, Python 3.9.0

CODE :

```
def solutions():

# letters = ('s', 'e', 'n', 'd', 'm', 'o', 'r', 'y')

    all_solutions = list()

    for s in range(9, -1, -1):

    for e in range(9, -1, -1):

    for n in range(9, -1, -1):

    for d in range(9, -1, -1):

    for m in range(9, 0, -1):

    for o in range(9, -1, -1):

    for r in range(9, -1, -1):

    for y in range(9, -1, -1):

    if len(set([s, e, n, d, m, o, r, y])) == 8: send = 1000 * s + 100 * e +

        10 * n + d more = 1000 * m + 100 * o

        + 10 * r + e

    money = 10000 * m + 1000 * o + 100 * n + 10 * e + y

    if send + more == money: all_solutions.append((send, more, money))

    return all_solutions

print(solutions())
```

OUTPUT :

```
PS E:\Studies\SRM University\SEM 6\AI> python -u  
[(9567, 1085, 10652)]  
PS E:\Studies\SRM University\SEM 6\AI> 
```

RESULT :

The constraint satisfying problem $\text{SEND} + \text{MORE} = \text{MONEY}$ solved using the carry over technique and values for the alphabets obtained successfully.

Experiment No : 4

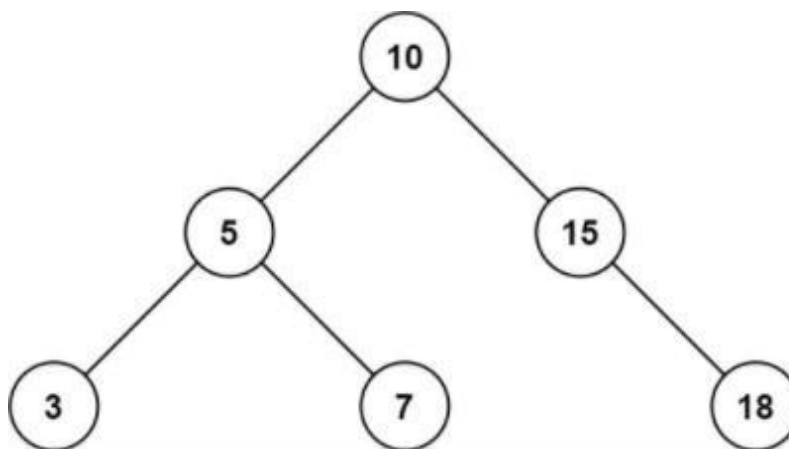
Date :

RANGE SUM OF BINARY SEARCH TREE (Implementation and Analysis of DFS and BFS for an application)

AIM : Given the root node of a binary search tree, return the sum of values of all nodes with a value in the range [low, high] using depth first and then breadth first search.

While :

- The number of nodes in the tree is in the range [1, 2 * 10⁴].
- 1 ≤ Node.val ≤ 105
- 1 ≤ low ≤ high ≤ 105
- All Node.value are unique.



Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32

ALGORITHM #1 DFS :

1. Traverse the tree using a depth first search.
2. Create a stack to store accessed nodes.
3. If `node.value` falls outside the range `[L, R]`
4. Then only the right branch could have nodes with value inside `[L, R]`.
5. If `Left <= node.value <= Right` then `Result[0] += node.value`
6. Else, recursively call the function until all nodes are visited.

ALGORITHM #2 BFS :

1. Traverse the tree using the breadth first search approach.
2. Maintain a queue and ptr to point toward the current node.
3. If `node == None` then continue.
4. If `Left <= node.value <= Right` the result `+= node.value`
5. If `Left > node.value` then `queue.append(node.right)`
6. If `R < node.value` then `queue.append(node.left)`
7. Repeat till all nodes are visited.

OPTIMIZATION TECHNIQUE:

Time Complexity: $O(N)$, where N is the number of nodes in the tree. Space Complexity: $O(N)$. For the recursive implementation, the recursion will consume additional space in the function call stack. In the worst case, the tree is of chain shape, and we will reach all the way down to the leaf node. For the iterative implementation, essentially we are doing a BFS (Breadth-First Search) traversal, where the stack will contain no more than two levels of the nodes. The maximal number of nodes in a binary tree is $N/2$.

Therefore, the maximal space needed for the stack would be $O(N)$.

TOOLS : VS Code and Python 3.9.0

DEPTH FIRST SEARCH (DFS) CODE :

#ITERATIVE APPROACH

```
class Solution(object):
```

```
    def rangeSumBST(self, root, L, R):
```

```
        def dfs(node):
```

```
            if node:
```

```
                if L <= node.val <= R:
```

```
                    self.ans += node.val
```

```
                if L < node.val:
```

```
                    dfs(node.left)
```

```
                if node.val < R:
```

```
                    dfs(node.right)
```

```
        self.ans = 0    dfs(root)
```

```
        return self.ans
```

RECURSIVE APPROACH

```
def rangeSumBST(root, L, R):
```

```
    ans = 0
```

```
    stack = [root]
```

```
    while stack:
```

```
        node = stack.pop()
```

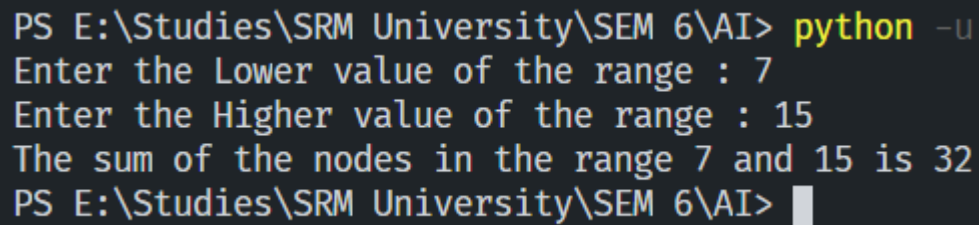
```
        if node:
```

```

    if L <= node.val <= R:
        ans += node.val
    if L < node.val:
        stack.append(node.left)
    if node.val < R:
        stack.append(node.right)
return ans

```

DEPTH FIRST SEARCH OUTPUT :



```

PS E:\Studies\SRM University\SEM 6\AI> python -u
Enter the Lower value of the range : 7
Enter the Higher value of the range : 15
The sum of the nodes in the range 7 and 15 is 32
PS E:\Studies\SRM University\SEM 6\AI>

```

BREADTH FIRST (BFS) CODE:

```

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution(object):
    def rangeSumBST(self, root, L, R):
        if root == None:
            return 0
        res = 0
        q = [root]
        while q:

```

```

next = []
for node in q:
    if L <= node.val <= R:
        res += node.val
    if node.left:
        next.append(node.left)
    if node.right:
        next.append(node.right)
q = next

return res

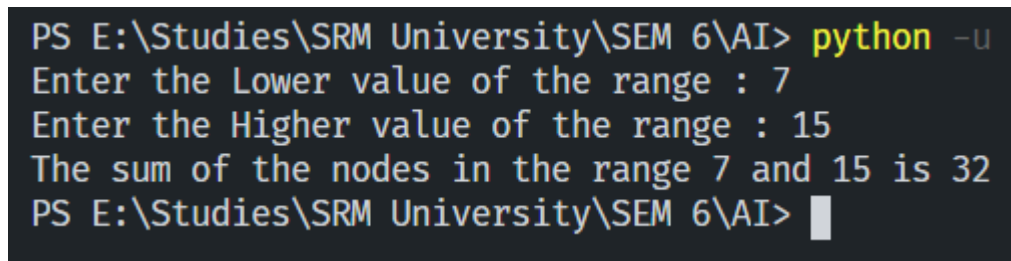
```

```

bst = TreeNode(10, 7, 15)
Solution().rangeSumBST(bst,10,7,15)

```

BREADTH FIRST OUTPUT:



```

PS E:\Studies\SRM University\SEM 6\AI> python -u
Enter the Lower value of the range : 7
Enter the Higher value of the range : 15
The sum of the nodes in the range 7 and 15 is 32
PS E:\Studies\SRM University\SEM 6\AI>

```

RESULT : Successfully found the sum of nodes in a binary search tree between any given range (min, max) using both depth first search and breadth first search approach.

Experiment No: 5

Date :

CUT OF TREE FOR GOLF EVENT (Implementation and Analysis of BFS and A* Search)

AIM : You are asked to cut off all the trees in a forest for a golf event. The forest is represented as an $m \times n$ matrix. In this matrix:

- 0 means the cell cannot be walked through.
- 1 represents an empty cell that can be walked through.
- A number greater than 1 represents a tree in a cell that can be walked through, and this number is the tree's height.

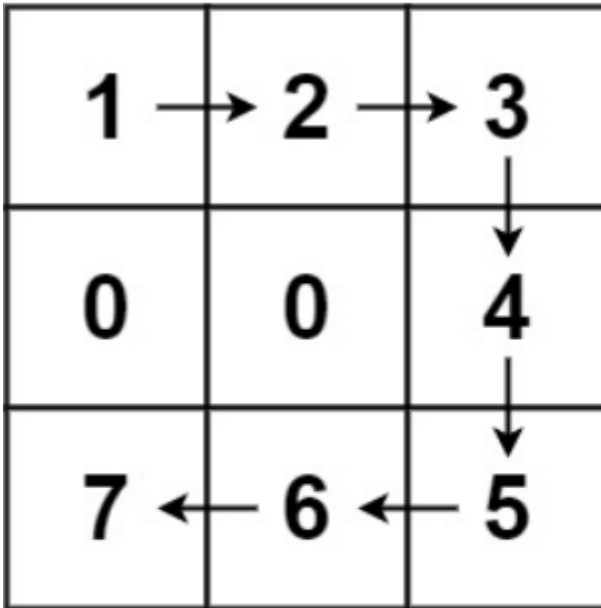
In one step, you can walk in any of the four directions: north, east, south, and west. If you are standing in a cell with a tree, you can choose whether to cut it off.

You must cut off the trees in order from shortest to tallest. When you cut off a tree, the value at its cell becomes 1 (an empty cell).

Starting from the point (0, 0), return the minimum steps you need to walk to cut off all the trees. If you cannot cut off all the trees, return -1.

You are guaranteed that no two trees have the same height, and there is at least one tree that needs to be cut off.

Example 1:



Input: forest = [[1,2,3],[0,0,4],[7,6,5]]

Output: 6

Explanation: Following the path above allows you to cut off the trees from shortest to tallest in 6 steps.

ALGORITHM #1 BFS :

1. Perform best-first-search, processing nodes (grid positions) in a queue.
2. Seen keeps track of nodes that have already been added to the queue at some point.
3. Those nodes will be already processed or are in the queue awaiting processing.
4. For each node that is next to be processed, look at it's neighbors. If they are in the forest (grid), they haven't been enqueued, and they aren't an obstacle, we will enqueue that neighbor.
5. Keep a side count of the distance travelled for each node. If the node we are processing is our destination 'target' (tr, tc), we'll return the answer.

ALGORITHM #1 A* SEARCH :

1. The A* star algorithm is another path-finding algorithm
2. For every node at position (r, c), have some estimated cost $\text{node.f} = \text{node.g} + \text{node.h}$
3. node.g is the actual distance from (sr, sc) to (r, c).
4. node.h is our heuristic (guess) of the distance from (r, c) to (tr, tc).
5. The taxicab distance, $\text{node.h} = \text{abs}(r - \text{tr}) + \text{abs}(c - \text{tc})$.
6. Keep a priority queue to decide what node to search in (expand) next.

OPTIMIZATION TECHNIQUE:

Frame the problem as providing some distance function $\text{dist}(\text{forest}, \text{sr}, \text{sc}, \text{tr}, \text{tc})$ that calculates the path distance from source (sr, sc) to target (tr, tc) through obstacles $\text{dist}[i][j] == 0$. (This distance function will return -1 if the path is impossible.)

What follows is code and complexity analysis that is common to all three approaches. After, the algorithms presented in our approaches will focus on only providing our dist function.

All three algorithms have similar worst case complexities, but in practice each successive algorithm presented performs faster on random data.

Time Complexity: $O((RC)^2)$ where there are R rows and C columns in the given forest. We walk to $R*C$ trees, and each walk could spend $O(R*C)$ time searching for the tree.

Space Complexity: $O(R*C)$, the maximum size of the data structures used.

TOOLS : Python 3.9.0 and VS Code.

CODE #1 BFS :

```
def bfs(forest, sr, sc, tr, tc):  
  
    R, C = len(forest), len(forest[0])  
  
    queue = collections.deque([(sr, sc, 0)])  
  
    seen = {(sr, sc)}  
  
    while queue:  
  
        r, c, d = queue.popleft()  
  
        if r == tr and c == tc:  
  
            return d  
  
        for nr, nc in ((r-1, c), (r+1, c), (r, c-1), (r, c+1)):  
  
            if (0 <= nr < R and 0 <= nc < C and  
  
                (nr, nc) not in seen and forest[nr][nc]):  
  
                seen.add((nr, nc))  
  
                queue.append((nr, nc, d+1))  
  
    return -1
```

OUTPUT #1 BFS :

```
PS E:\Studies\SRM University\SEM 6\AI> Input: forest = [[1,2,3],[0,0,4],[7,6,5]]  
>> Output: 6
```

CODE #2 A* SEARCH :

```
def astar(forest, sr, sc, tr, tc):  
  
    R, C = len(forest), len(forest[0])  
  
    heap = [(0, 0, sr, sc)]  
  
    cost = {(sr, sc): 0}  
  
    while heap:  
  
        f, g, r, c = heapq.heappop(heap)  
  
        if r == tr and c == tc: return g  
  
        for nr, nc in ((r-1,c), (r+1,c), (r,c-1), (r,c+1)):  
  
            if 0 <= nr < R and 0 <= nc < C and forest[nr][nc]:  
  
                ncost = g + 1 + abs(nr - tr) + abs(nc - tc)  
  
                if ncost < cost.get((nr, nc), 9999):  
  
                    cost[nr, nc] = ncost  
  
                    heapq.heappush(heap, (ncost, g+1, nr, nc))  
  
    return -1
```

OUTPUT #2 A* SEARCH :

```
PS E:\Studies\SRM University\SEM 6\AI> Input: forest = [[1,2,3],[0,0,4],[7,6,5]]  
>> Output: 6
```

RESULT : The cutting off of tree problem for a golf event successfully solved with 2 different approaches : Best first search and A* search algorithm.

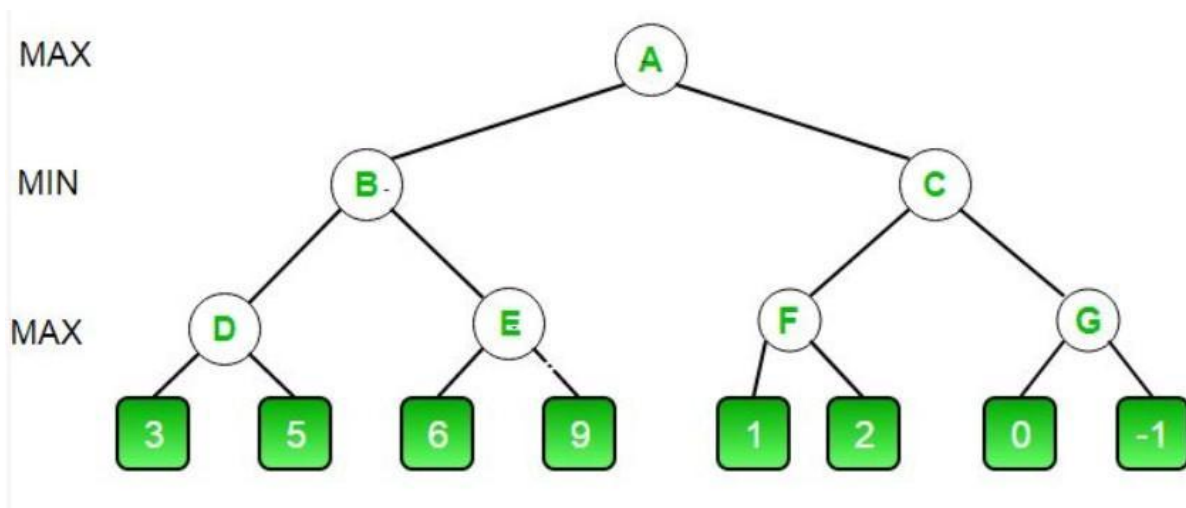
Experiment No: 6

Date :

MINIMAX ALGORITHM IN ALPHA BETA PRUNING

AIM : Developing a mini max algorithm for real world problems.

PROBLEM : Find the optimal value in the given tree of integer values in the most optimal way possible under the time complexity $O(B^D)$.



ALGORITHM MINIMAX APPROACH :

1. Start traversing the given tree in top to bottom manner.
2. If node is a leaf node then return the value of the node.
3. If isMaximizingPlayer exist then bestVal = -INFINITY
4. For each child node, value = minimax(node, depth+1, false, alpha, beta)
5. bestVal = max(bestVal, value) and alpha = max(alpha, bestVal)
6. If beta <= alpha then stop traversing and return bestVal
7. Else, bestVal = +INFINITY
8. For each child node, value = minimax(node, depth+1, true, alpha, beta)

9. $\text{bestVal} = \min(\text{bestVal}, \text{value})$ and $\text{beta} = \min(\text{beta}, \text{bestVal})$
10. if $\text{beta} \leq \text{alpha}$ the stop traversing and return bestVal

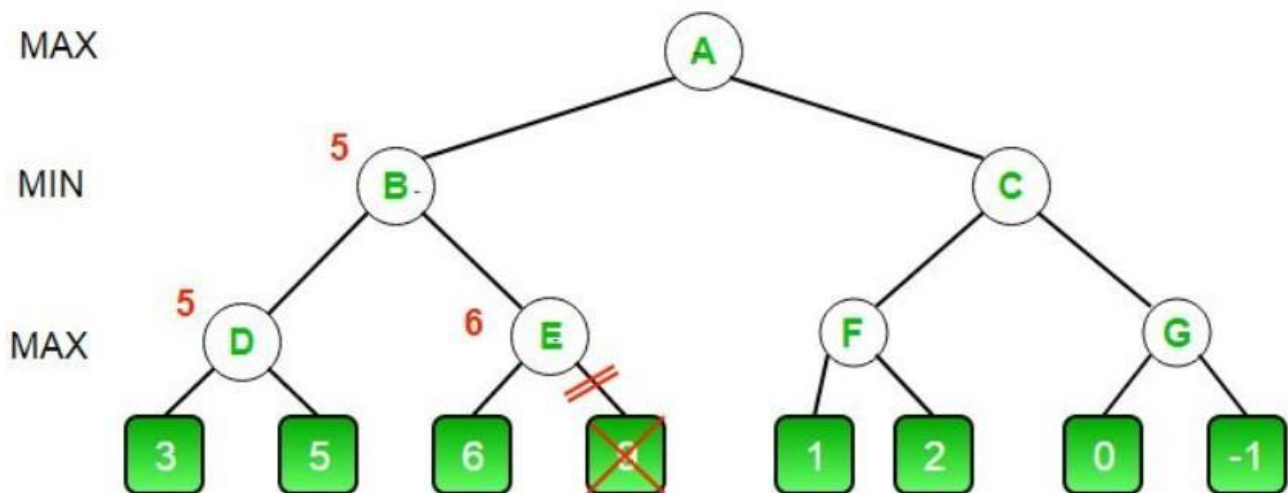
OPTIMIZATION TECHNIQUE :

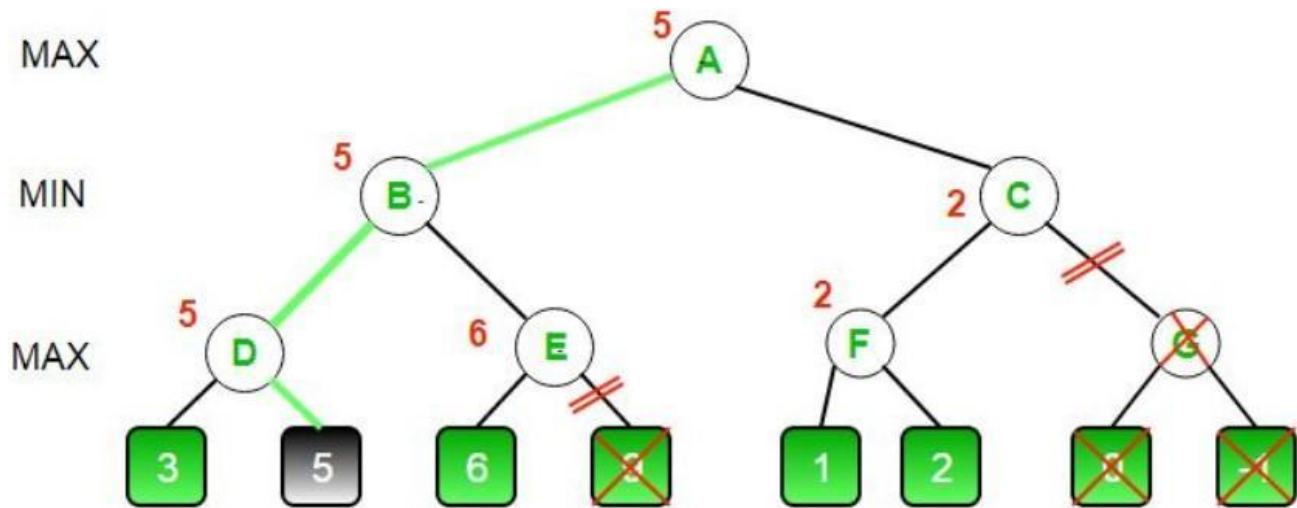
Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithms. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.

Alpha is the best value that the **maximizer** currently can guarantee at that level or above.

Beta is the best value that the **minimizer** currently can guarantee at that level or above.





CODE (MINIMAX ALGORITHM) :

MAX, MIN = 1000, -1000

```
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):
```

```
    if depth == 3:
        return values[nodeIndex]
```

```
    if maximizingPlayer:
```

```
        best = MIN
```

```
        for i in range(0, 2):
```

```
            val = minimax(depth + 1, nodeIndex * 2 + i,
                           False, values, alpha, beta)
```

```
            best = max(best, val)
```

```
            alpha = max(alpha, best)
```

```
        if beta <= alpha:
```

```
            break
```

```

        return best

    else:
        best = MAX
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                           True, values, alpha, beta)

            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break

        return best

if __name__ == "__main__":

    values = []
    for i in range(0, 8):

        x = int(input(f"Enter Value {i} : "))
        values.append(x)

    print ("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

```

OUTPUT :

```
PS E:\Studies\SRM University\SEM 6\AI> python
Enter Value 0 : 3
Enter Value 1 : 5
Enter Value 2 : 6
Enter Value 3 : 9
Enter Value 4 : 1
Enter Value 5 : 2
Enter Value 6 : 0
Enter Value 7 : -1
The optimal value is : 5
PS E:\Studies\SRM University\SEM 6\AI> █
```

RESULT : The Optimal value of the given tree successfully found using Minimax Algorithm with Alpha Beta Pruning in time complexity $O(B^D)$.

Experiment No: 7

Date :

IMPLEMENTATION OF UNIFICATION AND RESOLUTION

PROBLEM STATEMENT : Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

ALGORITHM :

1. function PL-RESOLUTION (KB, Q) returns true or false inputs: KB,
2. the knowledge base, group of sentences/facts in propositional logic
3. Q, the query, a sentence in propositional logic
4. clauses \rightarrow the set of clauses in the CNF representation of $KB \wedge Q$ new $\rightarrow \{ \}$
5. loop do for each C_i, C_j in clauses do
6. resolvents \rightarrow PL-RESOLVE (C_i, C_j)
7. if resolvents contains the empty clause the return true
8. new \rightarrow new union resolvents
9. if new is a subset of clauses then return false
10. clauses \rightarrow clauses union true

OPTIMIZATION TECHNIQUE:

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:

- There are no new clauses that can be added. An application of the resolution rule derives the empty clause
- An empty clause shows that the negation of the conclusion is a complete contradiction, hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic
2. Convert these statements into Conjunctive Normal Form
3. Negate the Conclusion (Proof by Contradiction)
4. Resolve using a Resolution Tree (Unification)

CODE UNIFICATION :

```
def get_index_comma(string):  
    index_list = list()  
    par_count = 0  
  
    for i in range(len(string)):  
        if string[i] == ',' and par_count == 0:  
            index_list.append(i)  
        elif string[i] == '(':  
            par_count += 1  
        elif string[i] == ')':  
            par_count -= 1  
  
    return index_list
```

```
def is_variable(expr):  
    for i in expr:  
        if i == '(' or i == ')':  
            return False  
  
    return True
```

```
def process_expression(expr):  
    expr = expr.replace(' ', '')  
    index = None
```

```

for i in range(len(expr)):
    if expr[i] == '(':
        index = i
        break
predicate_symbol = expr[:index]
expr = expr.replace(predicate_symbol, "")
expr = expr[1:len(expr) - 1]
arg_list = list()
indices = get_index_comma(expr)

if len(indices) == 0:
    arg_list.append(expr)
else:
    arg_list.append(expr[:indices[0]])
    for i, j in zip(indices, indices[1:]):
        arg_list.append(expr[i + 1:j])
    arg_list.append(expr[indices[len(indices) - 1] + 1:])

return predicate_symbol, arg_list

```

```

def get_arg_list(expr):
    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)

```

```

        for j in tmp:
            if j not in arg_list:
                arg_list.append(j)
            arg_list.remove(i)

    return arg_list

```

```

def check_occurs(var, expr):
    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False

```

```

def unify(expr1, expr2):

    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False

```

```

else:
    tmp = str(expr1) + '/' + str(expr2)
    return tmp
else:
    predicate_symbol_1, arg_list_1 = process_expression(expr1)
    predicate_symbol_2, arg_list_2 = process_expression(expr2)

    # Step 2
    if predicate_symbol_1 != predicate_symbol_2:
        return False
    # Step 3
    elif len(arg_list_1) != len(arg_list_2):
        return False
    else:
        # Step 4: Create substitution list
        sub_list = list()

        # Step 5:
        for i in range(len(arg_list_1)):
            tmp = unify(arg_list_1[i], arg_list_2[i])

            if not tmp:
                return False
            elif tmp == 'Null':
                pass
            else:
                if type(tmp) == list:
                    for j in tmp:
                        sub_list.append(j)
                else:
                    sub_list.append(tmp)

```

```
# Step 6
```

```
return sub_list
```

```
if __name__ == '__main__':
```

```
    # f1 = 'Q(a, g(x, a), f(y))'
```

```
    # f2 = 'Q(a, g(f(b), a), x)'
```

```
    f1 = input('f1 : ')
```

```
    f2 = input('f2 : ')
```

```
    result = unify(f1, f2)
```

```
    if not result:
```

```
        print("The process of Unification failed!")
```

```
    else:
```

```
        print("The process of Unification successful!")
```

```
        print(result)
```

OUTPUT UNIFICATION :

```
Vaishnavimoorthy:~/environment/RA1811028010049 $ python unification.py
f1 : 'Q(a, g(x, a), f(y))'
f2 : 'Q(a, g(f(b), a), x)'
The process of Unification successful!
['f(b)/x', 'f(y)/x']
Vaishnavimoorthy:~/environment/RA1811028010049 $
```

CODE RESOLUTION :

```
import copy
import time

class Parameter:
    variable_count = 1

    def __init__(self, name=None):
        if name:
            self.type = "Constant"
            self.name = name
        else:
            self.type = "Variable"
            self.name = "v" + str(Parameter.variable_count)
            Parameter.variable_count += 1

    def isConstant(self):
        return self.type == "Constant"

    def unify(self, type_, name):
        self.type = type_
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

    def __str__(self):
        return self.name

class Predicate:
```

```

def __init__(self, name, params):
    self.name = name
    self.params = params

def __eq__(self, other):
    return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def __str__(self):
    return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):
    return Predicate(negatePredicate(self.name), self.params)

```

```

class Sentence:

```

```

    sentence_count = 0

```

```

def __init__(self, string):
    self.sentence_index = Sentence.sentence_count
    Sentence.sentence_count += 1
    self.predicates = []
    self.variable_map = {}
    local = {}

```

```

    for predicate in string.split("|"):
        name = predicate[:predicate.find("(")]
        params = []

```

```

        for param in predicate[predicate.find("(") + 1: predicate.find(")"]].split(","):
            if param[0].islower():
                if param not in local: # Variable
                    local[param] = Parameter()

```

```

        self.variable_map[local[param].name] = local[param]
        new_param = local[param]
    else:
        new_param = Parameter(param)
        self.variable_map[param] = new_param

    params.append(new_param)

    self.predicates.append(Predicate(name, params))

def getPredicates(self):
    return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):
    return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate):
    self.predicates.remove(predicate)
    for key, val in self.variable_map.items():
        if not val:
            self.variable_map.pop(key)

def containsVariable(self):
    return any(not param.isConstant() for param in self.variable_map.values())

def __eq__(self, other):
    if len(self.predicates) == 1 and self.predicates[0] == other:
        return True
    return False

def __str__(self):
    return "".join([str(predicate) for predicate in self.predicates])

```



```

class KB:
    def __init__(self, inputSentences):
        self.inputSentences = [x.replace(" ", "") for x in inputSentences]
        self.sentences = []
        self.sentence_map = {}

    def prepareKB(self):
        self.convertSentencesToCNF()
        for sentence_string in self.inputSentences:
            sentence = Sentence(sentence_string)
            for predicate in sentence.getPredicates():
                self.sentence_map[predicate] = self.sentence_map.get(
                    predicate, []) + [sentence]

    def convertSentencesToCNF(self):
        for sentenceIdx in range(len(self.inputSentences)):
            # Do negation of the Premise and add them as literal
            if "=>" in self.inputSentences[sentenceIdx]:
                self.inputSentences[sentenceIdx] = negateAntecedent(
                    self.inputSentences[sentenceIdx])

    def askQueries(self, queryList):
        results = []

        for query in queryList:
            negatedQuery = Sentence(negatePredicate(query.replace(" ", "")))
            negatedPredicate = negatedQuery.predicates[0]
            prev_sentence_map = copy.deepcopy(self.sentence_map)
            self.sentence_map[negatedPredicate.name] = self.sentence_map.get(
                negatedPredicate.name, []) + [negatedQuery]

```

```

self.timeLimit = time.time() + 40

try:
    result = self.resolve([negatedPredicate], [
        False]*(len(self.inputSentences) + 1))
except:
    result = False

self.sentence_map = prev_sentence_map

if result:
    results.append("TRUE")
else:
    results.append("FALSE")

return results

def resolve(self, queryStack, visited, depth=0):
    if time.time() > self.timeLimit:
        raise Exception
    if queryStack:
        query = queryStack.pop(-1)
        negatedQuery = query.getNegatedPredicate()
        queryPredicateName = negatedQuery.name
        if queryPredicateName not in self.sentence_map:
            return False
        else:
            queryPredicate = negatedQuery
            for kb_sentence in self.sentence_map[queryPredicateName]:
                if not visited[kb_sentence.sentence_index]:
                    for kbPredicate in kb_sentence.findPredicates(queryPredicateName):

```

```

canUnify, substitution = performUnification(
    copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:
    newSentence = copy.deepcopy(kb_sentence)
    newSentence.removePredicate(kbPredicate)
    newQueryStack = copy.deepcopy(queryStack)

    if substitution:
        for old, new in substitution.items():
            if old in newSentence.variable_map:
                parameter = newSentence.variable_map[old]
                newSentence.variable_map.pop(old)
                parameter.unify(
                    "Variable" if new[0].islower() else "Constant", new)
                newSentence.variable_map[new] = parameter

        for predicate in newQueryStack:
            for index, param in enumerate(predicate.params):
                if param.name in substitution:
                    new = substitution[param.name]
                    predicate.params[index].unify(
                        "Variable" if new[0].islower() else "Constant", new)

        for predicate in newSentence.predicates:
            newQueryStack.append(predicate)

    new_visited = copy.deepcopy(visited)
    if kb_sentence.containsVariable() and len(kb_sentence.predicates) > 1:
        new_visited[kb_sentence.sentence_index] = True

    if self.resolve(newQueryStack, new_visited, depth + 1):

```

```
        return True
    return False
return True
```

```
def performUnification(queryPredicate, kbPredicate):
    substitution = {}
    if queryPredicate == kbPredicate:
        return True, {}
    else:
        for query, kb in zip(queryPredicate.params, kbPredicate.params):
            if query == kb:
                continue
            if kb.isConstant():
                if not query.isConstant():
                    if query.name not in substitution:
                        substitution[query.name] = kb.name
                    elif substitution[query.name] != kb.name:
                        return False, {}
                    query.unify("Constant", kb.name)
                else:
                    return False, {}
            else:
                if not query.isConstant():
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
                    elif substitution[kb.name] != query.name:
                        return False, {}
                    kb.unify("Variable", query.name)
                else:
                    if kb.name not in substitution:
                        substitution[kb.name] = query.name
```

```

        elif substitution[kb.name] != query.name:
            return False, {}
    return True, substitution

```

```

def negatePredicate(predicate):
    return predicate[1:] if predicate[0] == "~" else "~" + predicate

```

```

def negateAntecedent(sentence):
    antecedent = sentence[:sentence.find("=>")]
    premise = []

    for predicate in antecedent.split("&"):
        premise.append(negatePredicate(predicate))

    premise.append(sentence[sentence.find("=>") + 2:])
    return "|".join(premise)

```

```

def getInput(filename):
    with open(filename, "r") as file:
        noOfQueries = int(file.readline().strip())
        inputQueries = [file.readline().strip() for _ in range(noOfQueries)]
        noOfSentences = int(file.readline().strip())
        inputSentences = [file.readline().strip()
                           for _ in range(noOfSentences)]
    return inputQueries, inputSentences

```

```

def printOutput(filename, results):
    print(results)

```

```
with open(filename, "w") as file:
```

```
    for line in results:
```

```
        file.write(line)
```

```
        file.write("\n")
```

```
file.close()
```

```
if __name__ == '__main__':
```

```
    inputQueries_, inputSentences_ = getInput('input.txt')
```

```
    knowledgeBase = KB(inputSentences_)
```

```
    knowledgeBase.prepareKB()
```

```
    results_ = knowledgeBase.askQueries(inputQueries_)
```

```
    printOutput("output.txt", results_)
```

INPUT RESOLUTION :

6

A(Alice)

~A(Alice)

Z(Zig)

~Z(Zig)

G(Golf)

~G(Good)

10

$A(x) \Rightarrow B(x)$

$B(x) \Rightarrow C(x)$

$C(x) \Rightarrow D(x)$

$D(x) \Rightarrow E(x)$

$E(x) \Rightarrow A(x)$

G(g)

H(h)

I(i)

J(j)

X(x)

OUTPUT RESOLUTION :

```
Vaishnavimoorthy:~/environment $ cd RA1811028010049
Vaishnavimoorthy:~/environment/RA1811028010049 $ python resolution.py
['FALSE', 'FALSE', 'FALSE', 'FALSE', 'TRUE', 'FALSE']
Vaishnavimoorthy:~/environment/RA1811028010049 $
```

RESULT : Developed Unification and Resolution Algorithm in Python for solving logical problems.

Experiment No : 10

Date :

IMPLEMENTATION OF BLOCK WORLD PROBLEM

AIM : To implement the block world problem using correct artificial intelligence optimization techniques.

ALGORITHM :

1. Initialise a stack to store the blocks.
2. Make sure the stack is empty when HEAD NODE.NEXT = NULL
3. Read the pattern of blocks given label it START STATE
4. Compare the given pattern to the given final pattern label it GOAL STATE
5. Now start the movement of the blocks one by one either one on top or to the floor according to the need.
6. Keep recording these movements in the empty stack created by STACK.PUSH and STACK.POP methods.
7. Stop the block manipulation when goal state is reached.

OPTIMIZATION TECHNIQUE :

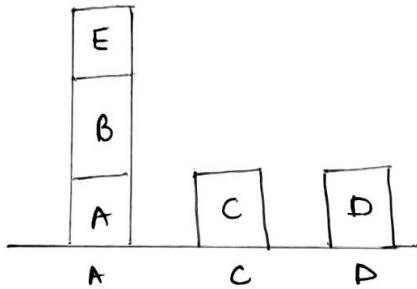
Here keeping track of movement of the block is the main problem, if we keep traversing the floor again and again after each move, our time complexity will be $O(n^2)$ which is exponentially higher than what is needed and should be avoided.

To solve this problem STACK data structure can be used, so whenever a movement is made the movement can be conveniently stored in the stack which will be initialized as empty which HEAD.NEXT = NULL.

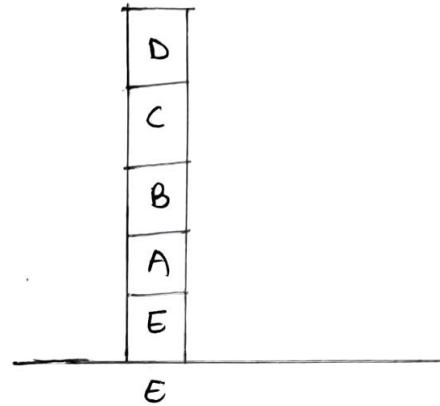
When the block is to be added to the sequence of blocks simply use STACK.PUSH() to make the movement. And when a block is supposed to be removed from the pattern of blocks STACK.POP() can be used to make that movement.

Implementing this will bring down the time complexity from $O(n)$ and worst case of $O(n^2)$ to $O(1)$ that is unit time which is a major optimization from exponential time.

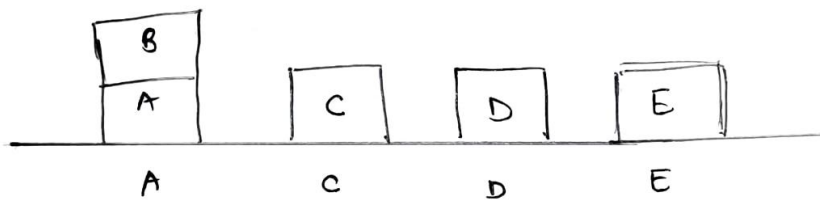
INITIAL STATE



FINAL STATE



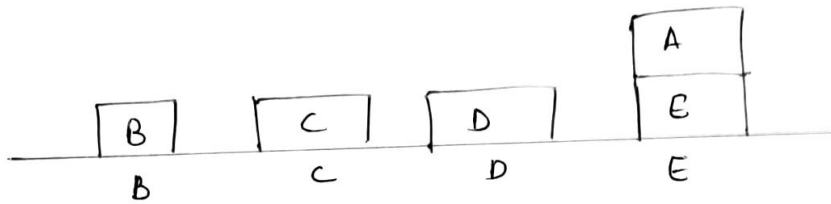
[I] UNSTACK E & PUTDOWN



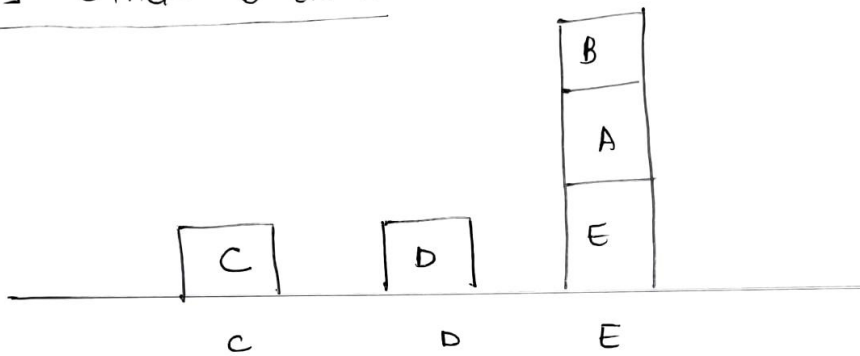
[II] UNSTACK B & PUTDOWN



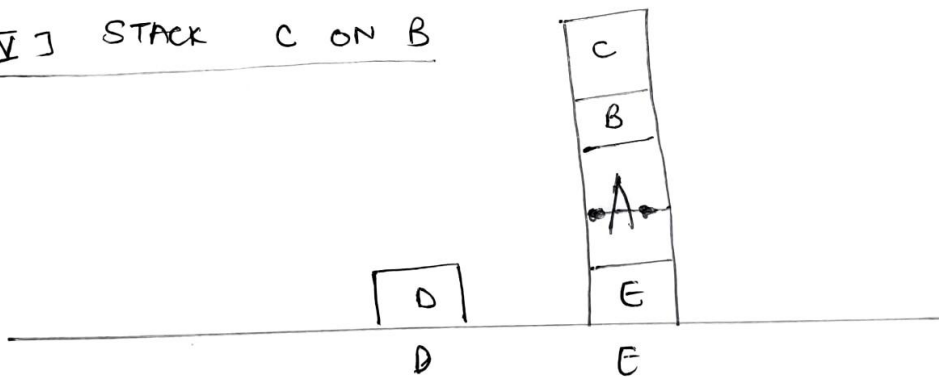
[III] STACK A ON E



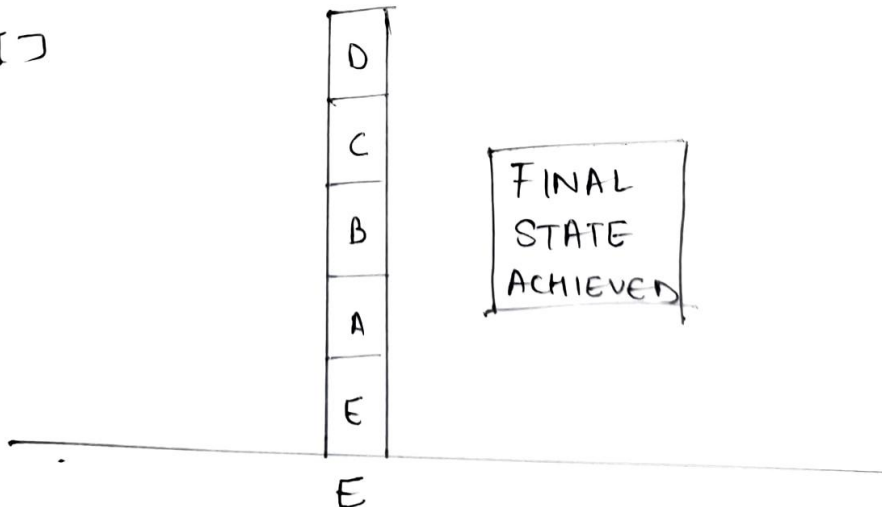
[IV] STACK B ON A



[V] STACK C ON B



[VI]



CODE :

```
class PREDICATE:
    def __str__(self):
        pass
    def __repr__(self):
        pass
    def __eq__(self, other) :
        pass
    def __hash__(self):
        pass
    def get_action(self, world_state):
        pass
```

```
class Operation:
    def __str__(self):
        pass
    def __repr__(self):
        pass
    def __eq__(self, other) :
        pass
    def precondition(self):
        pass
    def delete(self):
        pass
    def add(self):
        pass
```

```
class ON(PREDICATE):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
    def __str__(self):
```

```

    return "ON({X},{Y}).format(X=self.X,Y=self.Y)
def __repr__(self):
    return self.__str__()
def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
def __hash__(self):
    return hash(str(self))
def get_action(self, world_state):
    return StackOp(self.X,self.Y)

class ONTABLE(PREDICATE):
    def __init__(self, X):
        self.X = X
    def __str__(self):
        return "ONTABLE({X}).format(X=self.X)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def __hash__(self):
        return hash(str(self))
    def get_action(self, world_state):
        return PutdownOp(self.X)

class CLEAR(PREDICATE):
    def __init__(self, X):
        self.X = X
    def __str__(self):
        return "CLEAR({X}).format(X=self.X)
    def __repr__(self):
        return self.__str__()

```

```

def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
def __hash__(self):
    return hash(str(self))
def get_action(self, world_state):
    for predicate in world_state:
        if isinstance(predicate, ON) and predicate.Y==self.X:
            return UnstackOp(predicate.X, predicate.Y)
    return None

```

```

class HOLDING(PREDICATE):
    def __init__(self, X):
        self.X = X
    def __str__(self):
        return "HOLDING({X})".format(X=self.X)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def __hash__(self):
        return hash(str(self))
    def get_action(self, world_state):
        X = self.X
        if ONTABLE(X) in world_state:
            return PickupOp(X)
        else:
            for predicate in world_state:
                if isinstance(predicate, ON) and predicate.X==X:
                    return UnstackOp(X, predicate.Y)

```

```

class ARMEMPTY(PREDICATE):
    def __init__(self):

```

```

    pass
def __str__(self):
    return "ARMEMPTY"
def __repr__(self):
    return self.__str__()
def __eq__(self, other) :
    return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
def __hash__(self):
    return hash(str(self))
def get_action(self, world_state=[]):
    for predicate in world_state:
        if isinstance(predicate,HOLDING):
            return PutdownOp(predicate.X)
    return None

class StackOp(Operation):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
    def __str__(self):
        return "STACK({X},{Y})".format(X=self.X,Y=self.Y)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def precondition(self):
        return [ CLEAR(self.Y) , HOLDING(self.X) ]
    def delete(self):
        return [ CLEAR(self.Y) , HOLDING(self.X) ]
    def add(self):
        return [ ARMEMPTY() , ON(self.X,self.Y) ]

```

```

class UnstackOp(Operation):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y
    def __str__(self):
        return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def precondition(self):
        return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]
    def delete(self):
        return [ ARMEMPTY() , ON(self.X,self.Y) ]
    def add(self):
        return [ CLEAR(self.Y) , HOLDING(self.X) ]

class PickupOp(Operation):
    def __init__(self, X):
        self.X = X
    def __str__(self):
        return "PICKUP({X})".format(X=self.X)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def precondition(self):
        return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]
    def delete(self):
        return [ ARMEMPTY() , ONTABLE(self.X) ]
    def add(self):
        return [ HOLDING(self.X) ]

```

```

class PutdownOp(Operation):
    def __init__(self, X):
        self.X = X
    def __str__(self):
        return "PUTDOWN({X})".format(X=self.X)
    def __repr__(self):
        return self.__str__()
    def __eq__(self, other) :
        return self.__dict__ == other.__dict__ and self.__class__ == other.__class__
    def precondition(self):
        return [ HOLDING(self.X) ]
    def delete(self):
        return [ HOLDING(self.X) ]
    def add(self):
        return [ ARMEMPTY() , ONTABLE(self.X) ]

def isPredicate(obj):
    predicates = [ON, ONTABLE, CLEAR, HOLDING, ARMEMPTY]
    for predicate in predicates:
        if isinstance(obj,predicate):
            return True
    return False

def isOperation(obj):
    operations = [StackOp, UnstackOp, PickupOp, PutdownOp]
    for operation in operations:
        if isinstance(obj,operation):
            return True
    return False

```



```

def arm_status(world_state):
    for predicate in world_state:
        if isinstance(predicate, HOLDING):
            return predicate
    return ARMEMPTY()

```

```

class GoalStackPlanner:

```

```

    def __init__(self, initial_state, goal_state):
        self.initial_state = initial_state
        self.goal_state = goal_state

```

```

    def get_steps(self):
        steps = []
        stack = []

```

```

        #World State/Knowledge Base
        world_state = self.initial_state.copy()

```

```

        #Initially push the goal_state as compound goal onto the stack
        stack.append(self.goal_state.copy())

```

```

        #Repeat until the stack is empty
        while len(stack)!=0:

```

```

            #Get the top of the stack
            stack_top = stack[-1]

```

```

            #If Stack Top is Compound Goal, push its unsatisfied goals onto stack
            if type(stack_top) is list:
                compound_goal = stack.pop()

```

```

for goal in compound_goal:
    if goal not in world_state:
        stack.append(goal)

elif isOperation(stack_top):
    operation = stack[-1]
    all_preconditions_satisfied = True
    for predicate in operation.delete():
        if predicate not in world_state:
            all_preconditions_satisfied = False
            stack.append(predicate)

if all_preconditions_satisfied:

    stack.pop()
    steps.append(operation)

    for predicate in operation.delete():
        world_state.remove(predicate)
    for predicate in operation.add():
        world_state.append(predicate)

elif stack_top in world_state:
    stack.pop()
else:
    unsatisfied_goal = stack.pop()
    action = unsatisfied_goal.get_action(world_state)

    stack.append(action)
    for predicate in action.precondition():
        if predicate not in world_state:
            stack.append(predicate)

```

```
return steps
```

```
if __name__ == '__main__':
```

```
    initial_state = [  
        ON('B','A'),ON('E', 'B'),  
        ONTABLE('A'),ONTABLE('C'),ONTABLE('D'),  
        CLEAR('B'),CLEAR('C'),CLEAR('D'),CLEAR('E'),  
        ARMEMPTY()  
    ]
```

```
    goal_state = [  
        ON('B','D'),ON('D','C'), ON('C', 'A'),ON('A', 'E'),  
        ONTABLE('A'),  
        CLEAR('B'),CLEAR('C'), CLEAR('D'),CLEAR('E'),  
        ARMEMPTY()  
    ]
```

```
    goal_stack = GoalStackPlanner(initial_state=initial_state, goal_state=goal_state)
```

```
    steps = goal_stack.get_steps()
```

```
    print("UNSTACK(E,B)")
```

```
    print("PUTDOWN(E)")
```

```
    for i in steps:
```

```
        print(i)
```

OUTPUT :

```
PS E:\Studies\SRM University\SEM 6\AI> python
UNSTACK(E,B)
PUTDOWN(E)
UNSTACK(B,A)
PUTDOWN(B)
PICKUP(A)
STACK(A,E)
PICKUP(C)
STACK(C,A)
PICKUP(D)
STACK(D,C)
PICKUP(B)
STACK(B,D)
PS E:\Studies\SRM University\SEM 6\AI>
```

RESULT :

Block world problem successfully implemented using optimal artificial intelligence techniques under time complexity of $O(N^2)$ using a stack data structure for optimization.

Experiment No : 8**Date :****RULE BASED INFERENCE**

PROBLEM STATEMENT : Developing an optimized technique using an appropriate artificial intelligence algorithm to detect the animal.

ALGORITHM :

In artificial intelligence, we need intelligent computers which can create new logic from old logic or by evidence, so generating the conclusions from evidence and facts is termed as Inference. Inference rules are the templates for generating valid arguments. Inference rules are applied to derive proofs in artificial intelligence, and the proof is a sequence of the conclusion that leads to the desired goal.

In inference rules, the implication among all the connectives plays an important role. Following are some terminologies related to inference rules:

- Implication: It is one of the logical connectives which can be represented as $P \rightarrow Q$. It is a Boolean expression.
- Converse: The converse of implication, which means the right-hand side proposition goes to the left-hand side and vice-versa. It can be written as $Q \rightarrow P$.
- Contrapositive: The negation of converse is termed as contrapositive, and it can be represented as $\neg Q \rightarrow \neg P$.
- Inverse: The negation of implication is called inverse. It can be represented as $\neg P \rightarrow \neg Q$.

Identification of animal:

cheetah :- mammal,

carnivore,

verify(has_tawny_color),

verify(has_dark_spots).

tiger :- mammal,

carnivore,

verify(has_tawny_color),

verify(has_black_stripes).

giraffe :- ungulate,

verify(has_long_neck),

verify(has_long_legs).

zebra :- ungulate,

verify(has_black_stripes).

Classification rule :

mammal :- verify(has_hair), !.

mammal :- verify(gives_milk).

bird :- verify(has_feathers), !.

bird :- verify(flys),

verify(lays_eggs).

carnivore :- verify(eats_meat), !.

carnivore :- verify(has_pointed_teeth),

verify(has_claws),

verify(has_forward_eyes).

ungulate :- mammal,

verify(has_hooves), !.

ungulate :- mammal,

verify(chews_cud).

OPTIMIZATION TECHNIQUE :

Rule-based systems provide the computational mechanisms found in most expert systems. Knowledge is specified via facts and IF–THEN rules, and modus ponens is used as the underlying inference method to derive new conclusions from existing knowledge. These production rules in many cases allow a straightforward encoding of expertise about a particular domain, often as a situation–action pairs where the IF part of the rule specifies aspects of a situation leading to one or more actions as described in the THEN part. In principle, the rules and facts in a rule-based system can be translated into equivalent logical sentences. A combination of restrictions and additional constructs in their language, together with the tight integration between language and evaluation mechanism, offers substantial practical advantages for rule-based systems

- Propositional logic is also called Boolean logic as it works on 0 and 1.
- In propositional logic, we use symbolic variables to represent the logic, and we can use any symbol for a proposition, such A, B, C, P, Q, R, etc.
- Propositions can be either true or false, but it cannot be both.
- Propositional logic consists of an object, relations or function, and logical connectives.
- These connectives are also called logical operators.
- The propositions and connectives are the basic elements of propositional logic.
- Connectives can be said as a logical operator which connects two sentences.
- A proposition formula which is always true is called tautology, and it is also called a valid sentence.
- A proposition formula which is always false is called Contradiction.
- A proposition formula which has both true and false values is called
- Statements which are questions, commands, or opinions are not propositions such as "Where is Rohini", "How are you", "What is your name", are not propositions.

CODE :

```
import sys

def definiteNoun(s):
    s = s.lower().strip()
    if s in ['a', 'e', 'i', 'o', 'u', 'y']:
        return "an " + s
    else:
        return "a " + s

def removeArticle(s):
    "Remove the definite article 'a' or 'an' from a noun."
    s = s.lower().strip()
    if s[0:3] == "an ": return s[3:]
    if s[0:2] == "a ": return s[2:]
    return s

def makeQuestion(question, yes, no):
    return [question, yes, no]

def isQuestion(p):
    "Check if node is a question (with answers), or a plain answer."
    return type(p).__name__ == "list"

def askQuestion(question):
    print ("%s " % question,)
    return sys.stdin.readline().strip().lower()

def getAnswer(question):
    if isQuestion(question):
        return askQuestion(question[0])
    else:
```



```

    return askQuestion("Was the animal %s?" % definiteNoun(question))

def answeredYes(answer):
    if len(answer) > 0:
        return answer.lower()[0] == "y"
    return False

def gameOver(message):
    global tries
    print("")
    print ("%s" % message)
    print("")

def playAgain():
    return answeredYes(askQuestion("Play again?"))

def correctGuess(message):
    global tries
    gameOver(message)

    if playAgain():
        print("")
        tries = 0
        return Q
    else:
        sys.exit(0)

def nextQuestion(question, answer):
    global tries
    tries += 1

    if isQuestion(question):

```

```

    if answer:
        return question[1]
    else:
        return question[2]
else:
    if answer:
        return correctGuess("I knew it!")
    else:
        return makeNewQuestion(question)

def replaceAnswer(tree, find, replace):
    if not isQuestion(tree):
        if tree == find:
            return replace
        else:
            return tree
    else:
        return makeQuestion(tree[0],
            replaceAnswer(tree[1], find, replace),
            replaceAnswer(tree[2], find, replace))

def makeNewQuestion(wrongAnimal):
    global Q, tries

    correctAnimal = removeArticle(askQuestion("What did you think about?"))

    newQuestion = askQuestion("Enter a question that would distinguish %s from %s:"
        % (definiteNoun(correctAnimal), definiteNoun(wrongAnimal))).capitalize()

    yesAnswer = answeredYes(askQuestion("If I asked you this question " +
        "and you thought about %s, what would the correct answer be?" %
        definiteNoun(correctAnimal)))

```

```

# Create new question node
if yesAnswer:
    q = makeQuestion(newQuestion, correctAnimal, wrongAnimal)
else:
    q = makeQuestion(newQuestion, wrongAnimal, correctAnimal)
Q = replaceAnswer(Q, wrongAnimal, q)
tries = 0
return Q

```

```

def addNewQuestion(wrongAnimal, newques, correct):
    global Q
    q = makeQuestion(newques, correct, wrongAnimal)
    Q = replaceAnswer(Q, wrongAnimal, q)
    return Q

```

```

tries = 0
Q = (makeQuestion('Does it have fur?', 'Tiger', 'Penguin'))
q = addNewQuestion('Tiger', 'Does it have dark spots?', 'Leopard')
q = addNewQuestion('Leopard', 'Is it the fastest animal?', 'Cheetah')
q = addNewQuestion('Penguin', 'Can it fly?', 'Parrot')
q = Q

```

```

print ("Imagine an animal.")
print ("Answer YES or NO.")
print ("")

```

```

try:
    while True:
        ans = answeredYes(getAnswer(q))
        q = nextQuestion(q, ans)
except KeyboardInterrupt:

```

```
sys.exit(0)
```

except Exception:

```
sys.exit(1)
```

OUTPUTS:

```
Vaishnavimoorthy:~/environment/RA1811028010049 $ python exp8.py
Imagine an animal.
Answer YES or NO.

('Does it have fur? ',)
YES
('Does it have dark spots? ',)
NO
('Was the animal a tiger? ',)
NO
('What did you think about? ',)
Leopard
('Enter a question that would distinguish a leopard from a tiger: ',)
it is larger than tiger
('If I asked you this question and you thought about a leopard, what would the correct answer be? ',)
Leopard
('Does it have fur? ',)
YES
('Does it have dark spots? ',)
YES
('Is it the fastest animal? ',)
NO
('Was the animal a leopard? ',)
YES

I knew it!
```

```
Vaishnavimoorthy:~/environment/RA1811028010049 $ python exp8.py
Imagine an animal.
Answer YES or NO.

('Does it have fur? ',)
NO
('Can it fly? ',)
YES
('Was the animal a parrot? ',)
YES

I knew it!

('Play again? ',)
```

RESULT :

Animal Detection problem successfully implemented using optimal artificial intelligence techniques under time complexity of $O(n^2)$.

Experiment No : 9

Date

IMPLEMENTATION OF UNCERTAIN METHODS OF AN APPLICATION

Problem Statement:

To implement Fuzzy logic using matplotlib in python and find the graph of temperature, humidity and speed in different conditions.

Algorithm:

1. Locate the input, output, and state variables of the plane under consideration.
2. Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.
3. Obtain the membership function for each fuzzy subset.
4. Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the output of fuzzy subsets on the other side, thereby forming the rule base.
5. Choose appropriate scaling factors for the input and output variables for normalizing the variables between $[0, 1]$ and $[-1, 1]$ interval.
6. Carry out the fuzzification process.
7. Identify the output contributed from each rule using fuzzy approximate reasoning.
8. Combine the fuzzy outputs obtained from each rule.
9. Finally, apply defuzzification to form a crisp output.

Optimization Technique:

1. Decomposing the large-scale system into a collection of various subsystems.
2. Varying the plant dynamics slowly and linearizing the nonlinear plane dynamics about a set of operating points.
3. Organizing a set of state variables, control variables, or output features for the system under consideration.
4. Designing simple P, PD, PID controllers for the subsystems. Optimal controllers can also be designed.

Uncertainty In this problem : Fuzzy Logic - Temperature, Humidity and Speed.

CODE :

```
from fuzzy_system.fuzzy_variable_output import FuzzyOutputVariable
from fuzzy_system.fuzzy_variable_input import FuzzyInputVariable
# from fuzzy_system.fuzzy_variable import FuzzyVariable
from fuzzy_system.fuzzy_system import FuzzySystem

temp = FuzzyInputVariable('Temperature', 10, 40, 100)
temp.add_triangular('Cold', 10, 10, 25)
temp.add_triangular('Medium', 15, 25, 35)
temp.add_triangular('Hot', 25, 40, 40)

humidity = FuzzyInputVariable('Humidity', 20, 100, 100)
humidity.add_triangular('Wet', 20, 20, 60)
humidity.add_trapezoidal('Normal', 30, 50, 70, 90)
humidity.add_triangular('Dry', 60, 100, 100)

motor_speed = FuzzyOutputVariable('Speed', 0, 100, 100)
motor_speed.add_triangular('Slow', 0, 0, 50)
motor_speed.add_triangular('Moderate', 10, 50, 90)
motor_speed.add_triangular('Fast', 50, 100, 100)

system = FuzzySystem()
system.add_input_variable(temp)
system.add_input_variable(humidity)
system.add_output_variable(motor_speed)

system.add_rule(
    { 'Temperature': 'Cold',
      'Humidity': 'Wet' },
    { 'Speed': 'Slow' })

system.add_rule(
    { 'Temperature': 'Cold',
```

```
        'Humidity':'Normal' },  
        { 'Speed':'Slow'})
```

```
system.add_rule(  
    { 'Temperature':'Medium',  
      'Humidity':'Wet' },  
    { 'Speed':'Slow'})
```

```
system.add_rule(  
    { 'Temperature':'Medium',  
      'Humidity':'Normal' },  
    { 'Speed':'Moderate'})
```

```
system.add_rule(  
    { 'Temperature':'Cold',  
      'Humidity':'Dry' },  
    { 'Speed':'Moderate'})
```

```
system.add_rule(  
    { 'Temperature':'Hot',  
      'Humidity':'Wet' },  
    { 'Speed':'Moderate'})
```

```
system.add_rule(  
    { 'Temperature':'Hot',  
      'Humidity':'Normal' },  
    { 'Speed':'Fast'})
```

```
system.add_rule(  
    { 'Temperature':'Hot',  
      'Humidity':'Dry' },  
    { 'Speed':'Fast'})
```

```

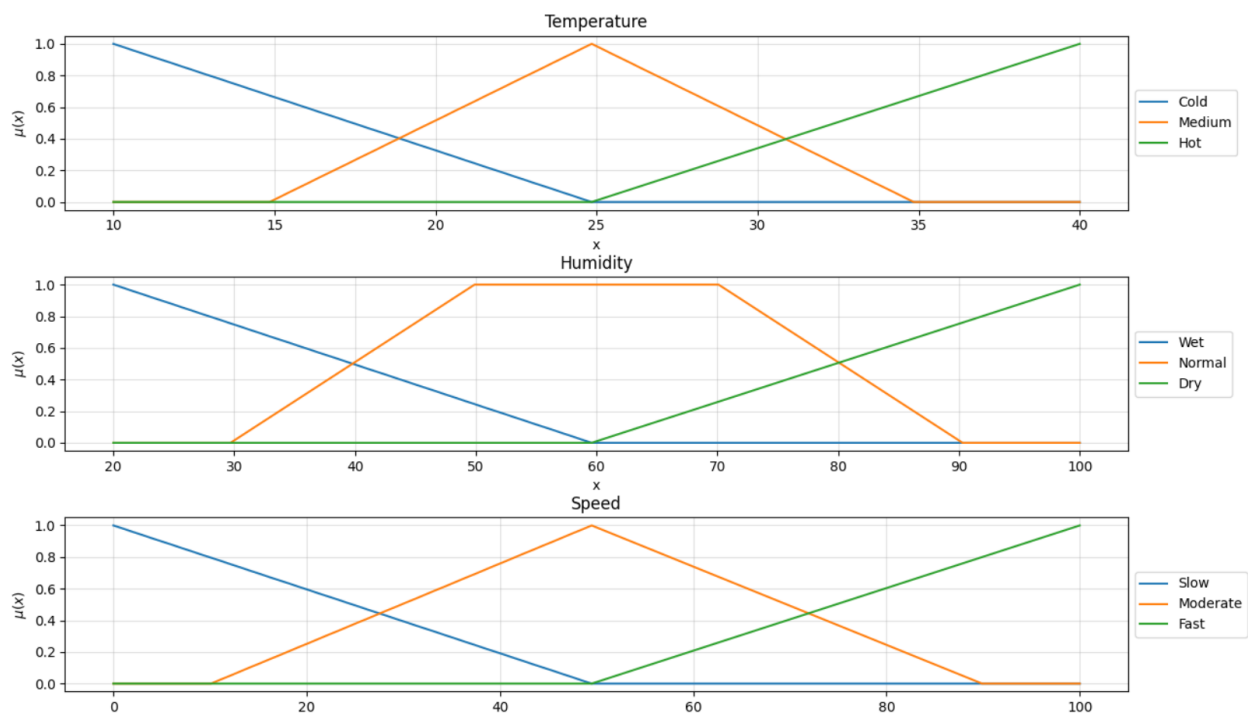
system.add_rule(
    { 'Temperature':'Medium',
      'Humidity':'Dry' },
    { 'Speed':'Fast'})

output = system.evaluate_output({
    'Temperature':18,
    'Humidity':60
})

print(output)
system.plot_system()

```

OUTPUT :



Result: We have successfully implemented fuzzy uncertainty problem using matplotlib and output is received.