SRM INSTITUTE OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

18CSC305J-ARTIFICIAL INTELLIGENCE

SEMESTER – 6

BATCH-2

REGISTRATION NUMBER	RA1811028010049
NAME	SHUSHRUT KUMAR

B.Tech-CSE-CC, Third Year (Section: J2)

Faculty Incharge: Vaishnavi Moorthy, Asst Prof/Dept of CSE

Year 2020-2021

INDEX

LAB : 1

DATE : 21-01-2021

<div align="center">

TOY PROBLEM

</div>

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 ith 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.

GOLD

PREMIUM

EXECUTIVE
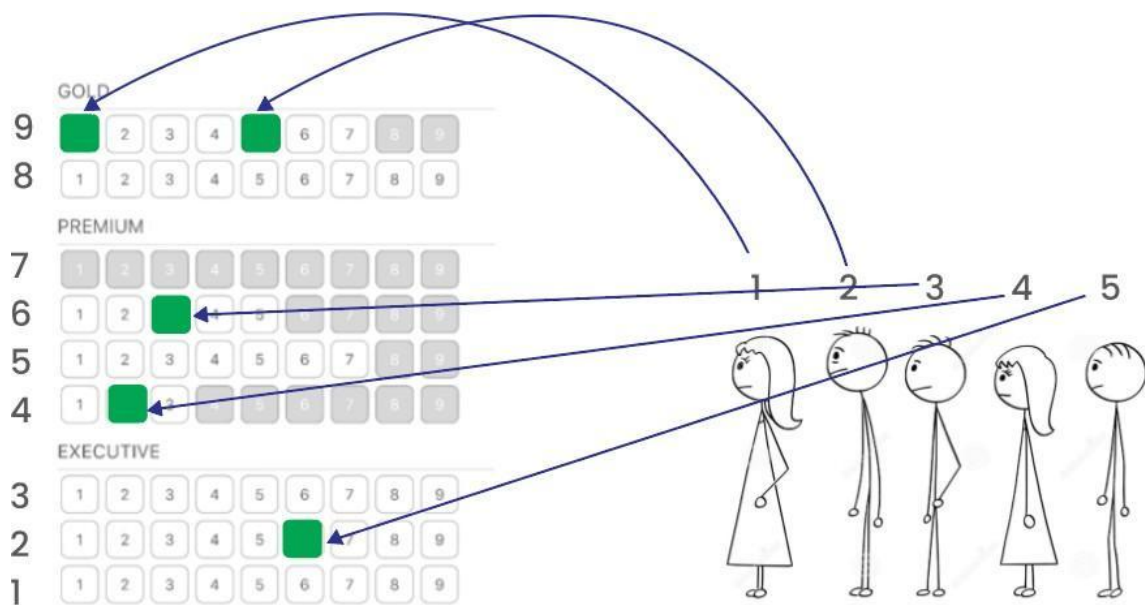
Priority Queue

$[2, 6, 9, 4, 9]$ — 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])
```

```python
        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())
```
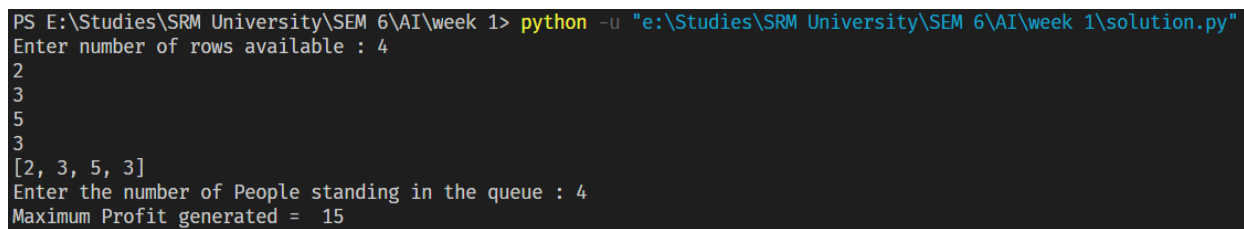
seats.append(empty)

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 : 29-01-2021

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 :

```python
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 = []
```

```python
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()
```
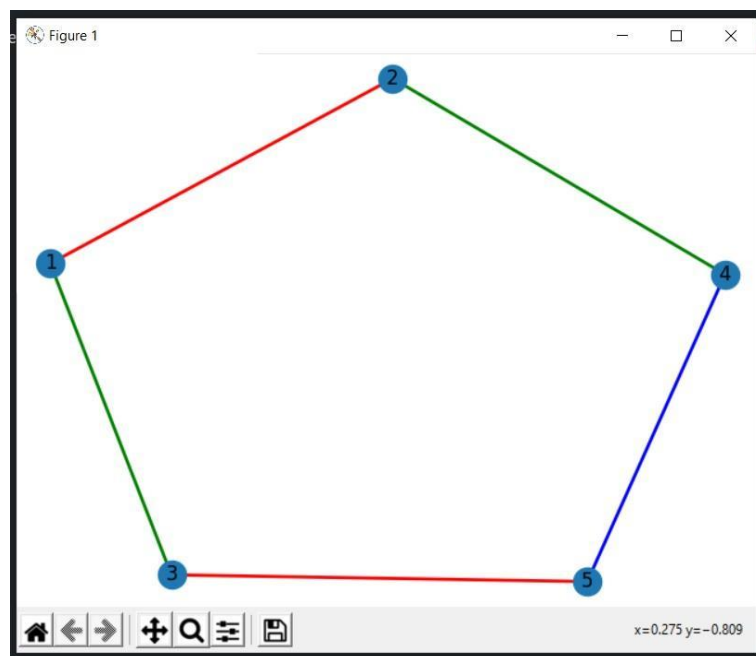
OUTPUT :

Fig: Edge Coloring

CODE - VERTEX COLORING :

```python
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()
```
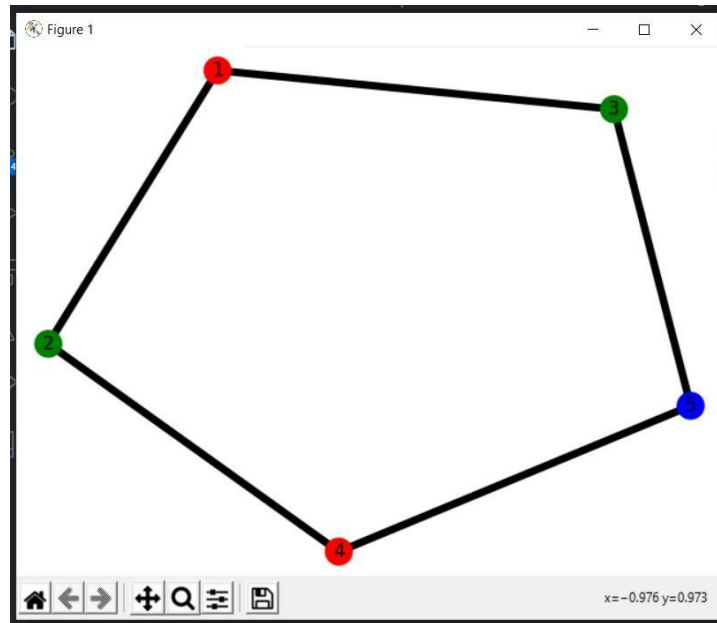
OUTPUT :

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):
```
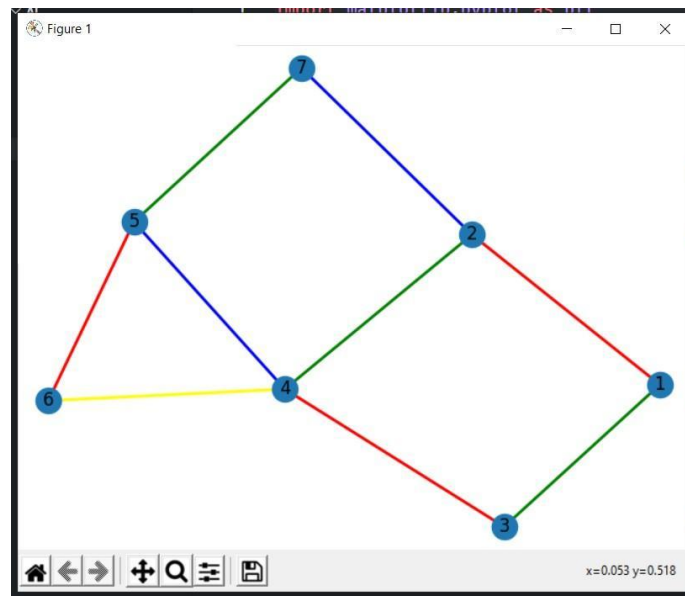
print("NO OF FACES : "+str(regions)) for i

in range(1,regions+1):

print("FACE   1   :   "+some_colors[i%4])

regionColour(no_of_faces)

OUTPUT :

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 : 05-02-2021

CONSTRAINT SATISFACTION PROBLEM

1) SEND + MORE = MONEY

5 4 3   2 1

S E N D

+   M O R E

c_3 c_2 c_1

  - - - - - - - - - - - - -

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=8or9' so 'S+M=9or10' & so 'O = 0 or 1'. But 'M=1', so 'O = 0'.

3. If there is c+rry from Column 3 to 4 then 'E=9' & so 'N=0'.

    But 'O = 0' so there is no carry & 'S=9' & 'c_3=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' & 'c_2=1'.

5. If there is carry from column 1 to 2 then 'N+R=E mod 10' &

    'N=E+1' so 'E+1+R=E mod 10', so 'R=9' but 'S=9', so there must be

    c+rry from column 1 to 2. Therefore 'c_1=1' & 'R=8'.

6. To produce carry 'c_1=1' from column 1 to 2, we must h+ve 'D+E=10+Y'

as Y cannot be 0/1 so D+E is at least 12. As D is at most 7 & E is

At least 5 (D cannot be 8 or 9 as it is already assigned). N is at most 7 &

'N=E+1' so 'E=5or6'.

7. If E were 6 & D+E at least 12 then D would be 7, but 'N=E+1' & N would

also be 7 which is impossible. Therefore 'E=5' & 'N=6'.

8. D+E is at least 12 for that we get 'D=7' & 'Y=2'.

SOLUTION:

```
 )      ;  ;  '

 - 1    )  ;  ;

   - - - - - - - - - -

1 0  6  5  2
```

VALUES:

S=9

E=5

N=6

D=7

M=1

O=0

R=8

Y=2

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 >= 5.

'O' can't be 0 as R will be 0. So T can't be 5 so let T>=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 b (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 v+lue of c+rry will be gener+ted when the v+lue of C is

greater th+n 5

C=   9

C1+S+D=   E

C1+2+1= E

Therefore   E=   3

C4+C+R=   A+10

C4+9+4= A+10

Therefore A= 3 but it c+nnot be possible +s E= 3

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

2+1+0= 3

Therefore E= 3 m+king C2= 0 since 2+1=3

Now let's consider the equation again:

C+R+C4= A+10

9+4+0= A+10

13= A+10

Therefore A= 3 but E= 3 So

A is not equ+l to 3

Ag+in considering R= 6 So S= 3 C4= 0

C+R+C4= A+10

9+6+0= A+10

15= A+10

Therefore   A=   5

And S+D+C1= E

3+1+0= E therefore E= 4 +nd C2= 0

Now considering the equ+tion R+O+C3=

N

6+0+C3= N

So 6+0+C3< or equ+l to 3

Let C3= 1

Then O< or equ+l to 2

Th+t is O= 0, 1, 2

Let O =2

Ag+in considering R+O+C3= N

6+2+1= N

Hence N= 9 but C= 9 so N c+nnot be equ+l to 9.

Now let N= 8 +nd C3= 0

Let us consider equ+tion

O+A+C2= G

Therefore G= 7

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

And C1= 0 C2= 0 C3= 0 C4= 0 C5= 1

Now verifying the above values in the equ+tion we get:

C5C4C3C2C1

CROSS

96233

ROADS

62513

Shape

DANGER

158746

5.   If AA + BB = ABC

Explanation:

A A

B B + C C

- - - - - - -

A B C

- - - - - - - -

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, Equ+tion (2) is not possible.

From Equ+tion (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 +nd C = 8 A

+ B + C = 18

6.   N O + G U N + N O = H U N T

Solution:

N O

+ G U N N

    O

————-

H U N T

————-

Here H = 1, from the NUNN column we must have "carry 1," so G = 9, U = zero.

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

h+ve 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 +accept+able, O = 7. So we h+ve 87 + 908 + 87 = 1082.

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 SEND + MORE = MONEY solved using the carry over technique and values for the alphabets obtained successfully.

Experiment No : 4

Date : 12-02-2021

<div align="center">

RANGE SUM OF BINARY SEARCH TREE

(Implementation and Analysis of DFS and BFS for an application)

</div>

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.
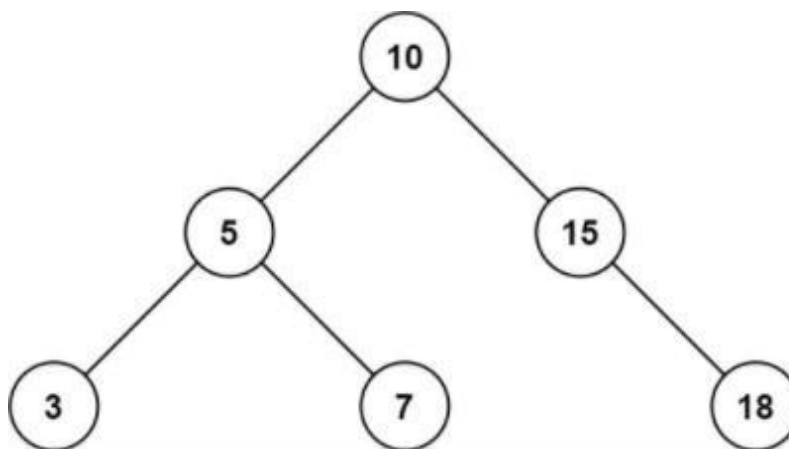

```
Input: root = [10,5,15,3,7,null,18], low = 7, high = 15
Output: 32
```

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

```python
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

```python
def rangeSumBST(root, L, R):
    ans = 0
    stack = [root]
    while stack:
        node = stack.pop()
        if node:
```
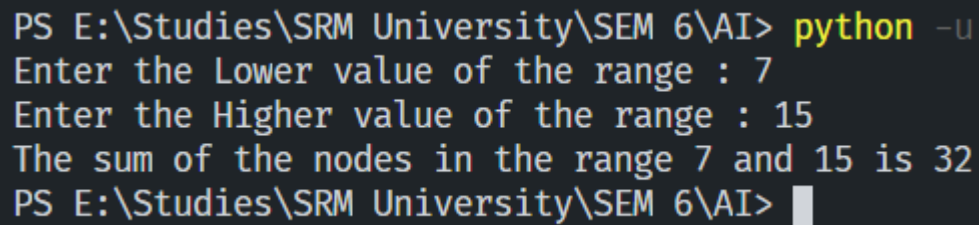
```
        if L <= node.val <= R:
            ans += node.val
        if L < node.val:
            stack.append(node.left)
        if node.val < R:
            stack.append(node.right)
    return ans
```

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)
```
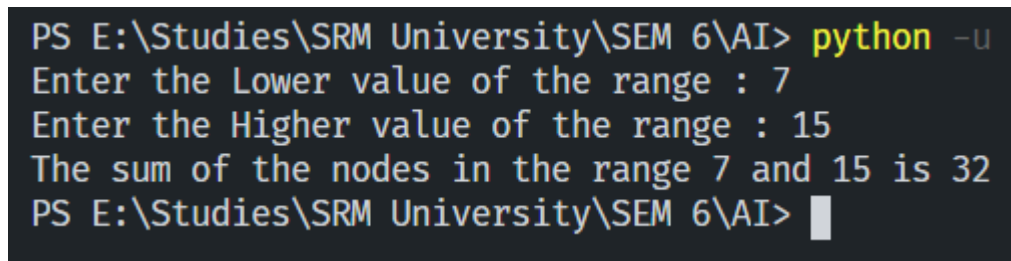
BREADTH FIRST OUTPUT:

```
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 : 26-02-2021

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 x 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:
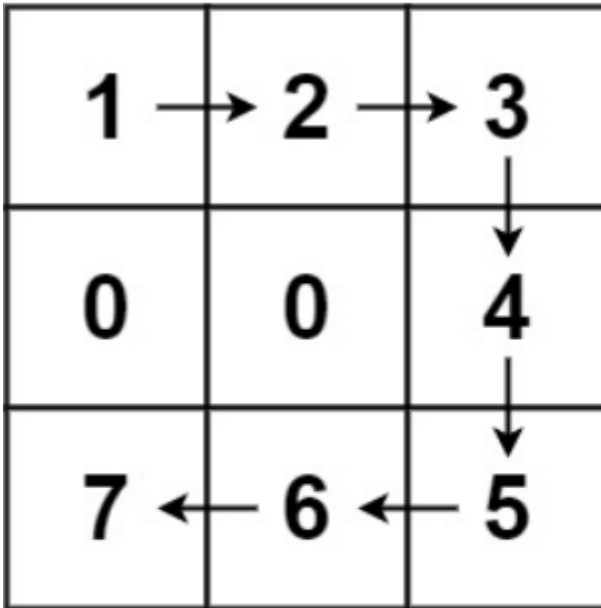

```
Input: forest = [[1,2,3],[0,0,4],[7,6,5]]
Output: 6
Explanation: Following the path above allows you to cut off
the trees from shortest to tallest in 6 steps.
```

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 node.f = node.g + 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, node.h = abs(r-tr) + abs(c-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 dist(forest, sr, sc, tr, tc) that calculates the path distance from source (sr, sc) to target (tr, tc) through obstacles 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 :

```python
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 :

```python
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.
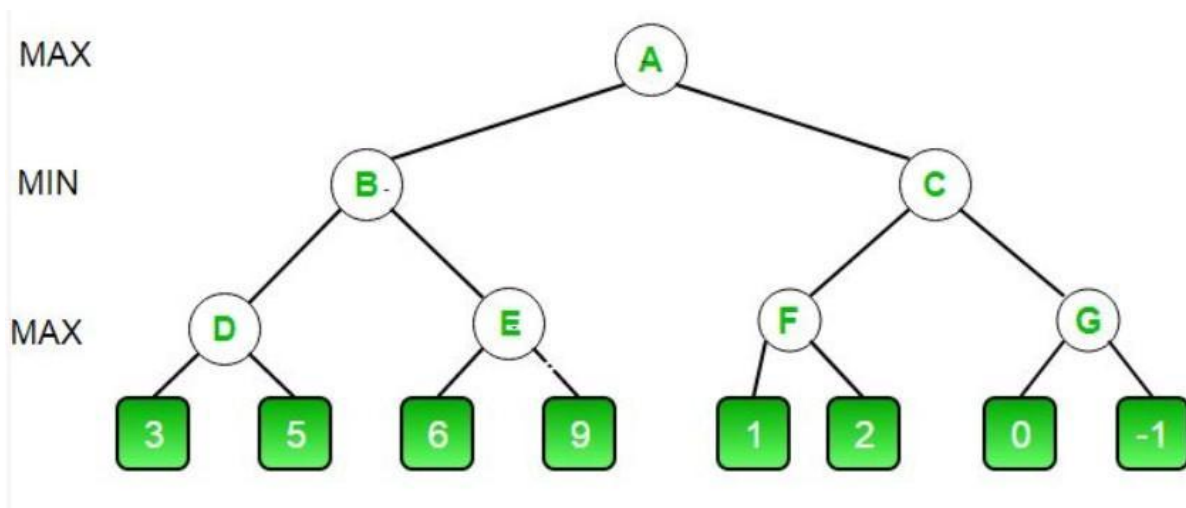
Experiment No: 6

Date : 26-02-2021

<p style="text-align:center">MINIMAX ALGORITHM IN ALPHA BETA PRUNING</p>

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.   bestVal = min(bestVal, value) and beta = min(beta, bestVal)

10.   if beta <= 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.
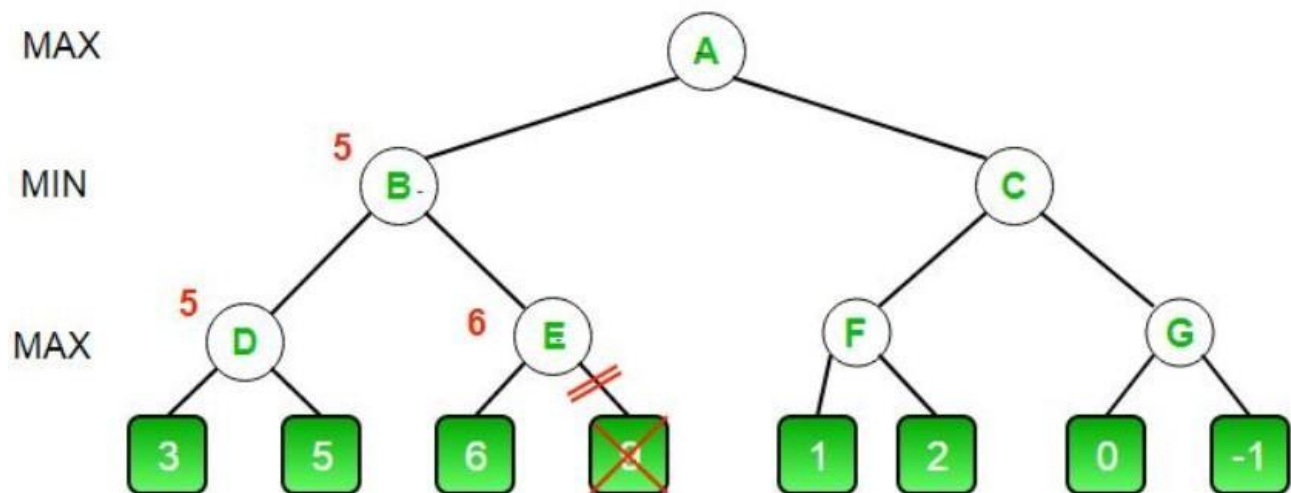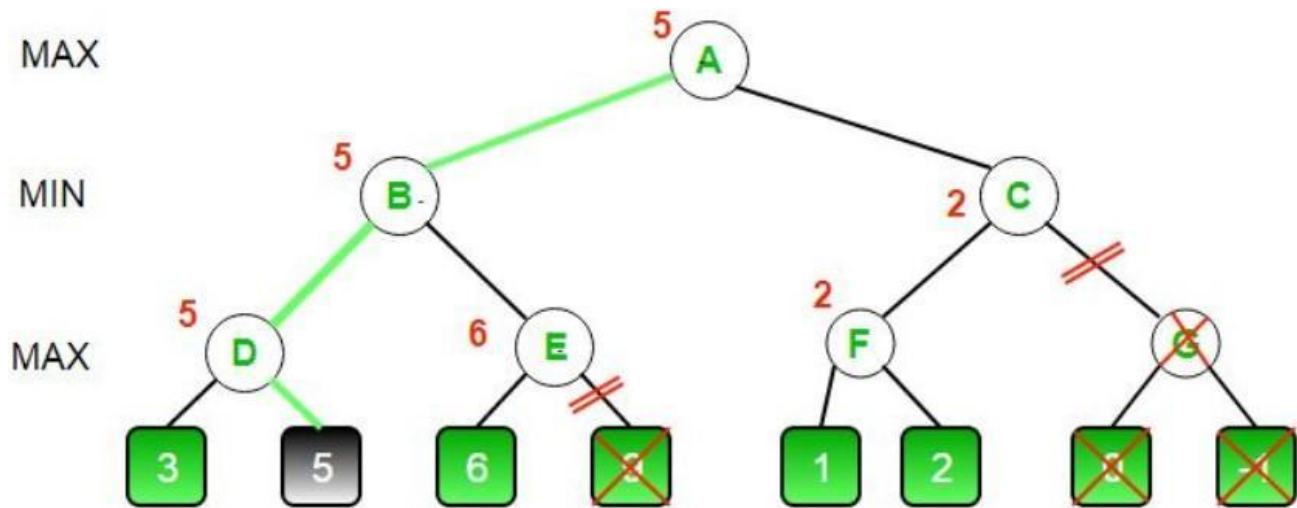
CODE (MINIMAX ALGORITHM) :

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
```

```python
            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 : 13-04-21

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 → the set of clauses in the CNF representation of KB ^ Q new → { }
5. loop do for each Ci, Cj in clauses do
6. resolvents → PL-RESOLVE (Ci, Cj)
7. if resolvents contains the empty clause the return true
8. new → new union resolvents
9. if new is a subset of clauses then return false
10. clauses → 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 a2. 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 clauseAn 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 :

```python
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
```

```python
    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)
```

```python
        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
```

```python
        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)
```

```python
        # 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 :

```python
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:
```

```python
    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()
```

```python
                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])
```

```python
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]
```

```python
        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):
```

```python
                    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
```

```python
        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) => B(x)
B(x) => C(x)
C(x) => D(x)
D(x) => E(x)
E(x) => 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 : 12-03-2021

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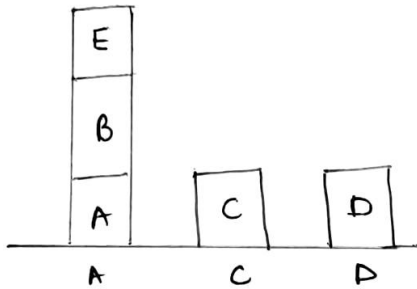
Implementing this will bring down the time complexity from $O(n)$ and worst case of $O(n^2)$ to $O(1)$ that is unit time which is a major optimization from exponential time.
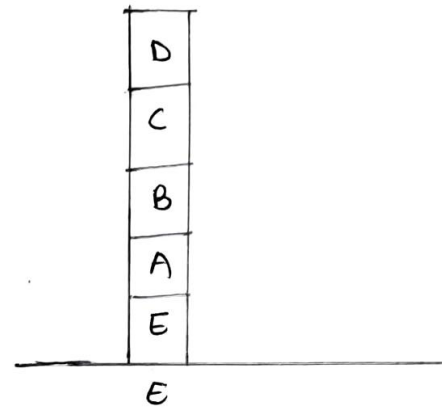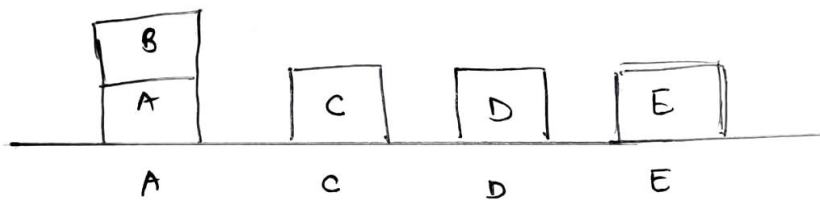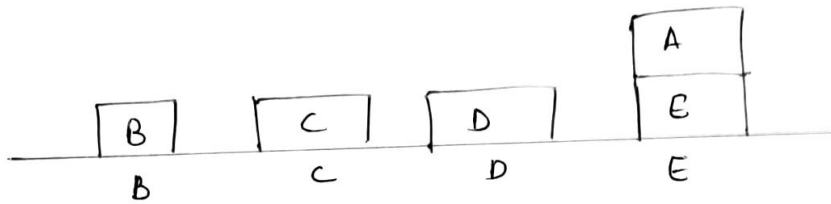
INITIAL STATE

FINAL STATE

[I] UNSTACK E & PUTDOWN

[II] UNSTACK B & PUTDOWN

[III] STACK A ON E

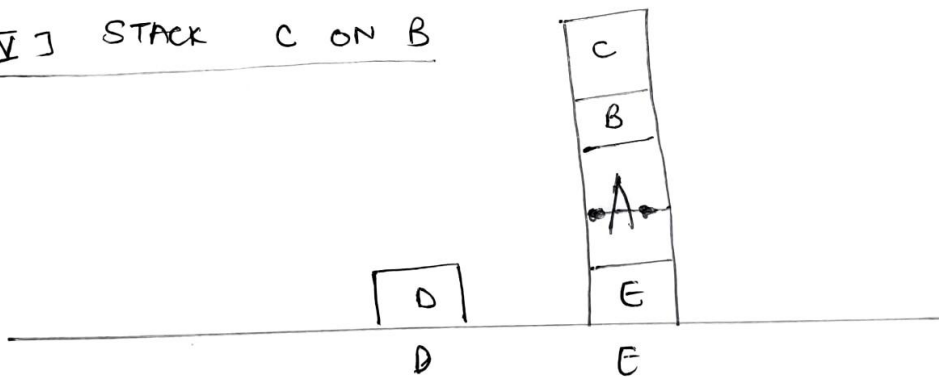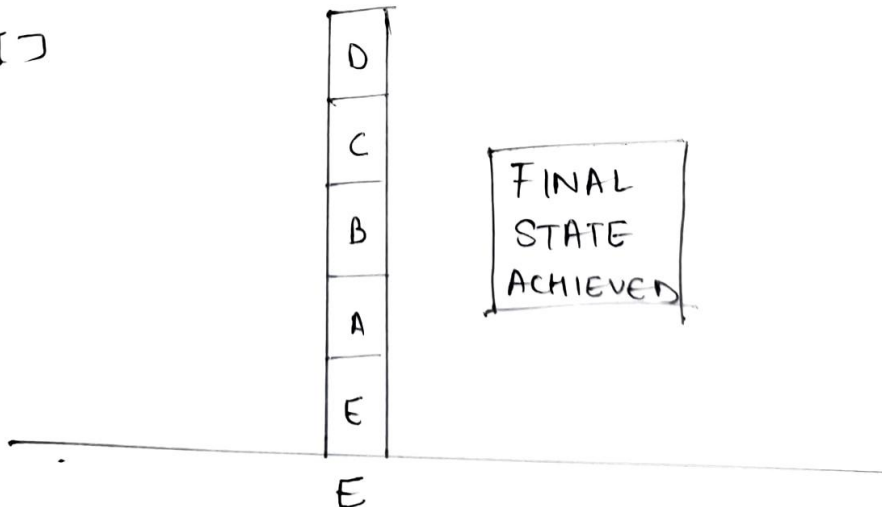B   C   D   E

[IV] STACK B ON A

C   D   E

[V] STACK C ON B

D   E

[VI]

FINAL
STATE
ACHIEVED

E
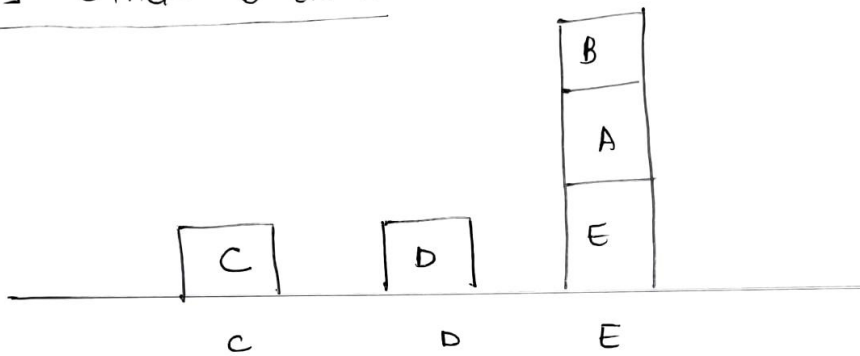
CODE :

```python
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):
```

```python
        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)
        self.X = X
    def __repr__(self):
        return self.__str__()
```

```python
    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):
```

```python
        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) ]
```

```python
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) ]
```

```python
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
```

```python
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)
```

```python
    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.