

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY, KATTANKULATHUR**



**18CSC305J- ARTIFICIAL INTELLIGENCE**

**Semester- VI**

**Academic Year: 2021- 2022**

**Submitted by:**

**SANJUKTA SEN**

**RA1911028010039**

**Section- I2**

## **Experiment-1**

### **Implementation of Toy Problem using Python**

#### **Problem Statement:**

(Camel Banana problem) A person has 3000 bananas and a camel. The person wants to transport the maximum number of bananas to a destination which is 1000 KMs away, using only the camel as a mode of transportation. The camel cannot carry more than 1000 bananas at a time and eats a banana every km it travels. What is the maximum number of bananas that can be transferred to the destination using only camel (no other mode of transportation is allowed).

**Tool Used:** Google Collab

#### **Algorithm & Optimization technique:**

Let's see what we can infer from the question:

- We have a total of 3000 bananas.
- The destination is 1000KMs
- Only 1 mode of transport.
- Camel can carry a maximum of 1000 banana at a time.
- Camel eats a banana every km it travels.

With all these points, we can say that person won't be able to transfer any banana to the destination as the camel is going to eat all the banana on its way to the destination.

But the trick here is to have intermediate drop points, then, the camel can make several short trips in between.

Also, we try to maintain the number of bananas at each point to be multiple of 1000.

Let's have 2 drop points in between the source and destination.

With 3000 bananas at the source. 2000 at a first intermediate point and 1000 at 2nd intermediate point.

- To go from source to IP1 point camel has to take a total of 5 trips 3 forward and 2 backward. Since we have 3000 bananas to transport.
- The same way from IP1 to IP2 camel has to take a total of 3 trips, 2 forward and 1 backward. Since we have 2000 bananas to transport.

- At last from IP2 to a destination only 1 forward move.

Let's see the total number of bananas consumed at every point.

- From the source to IP1 its  $5x$  bananas, as the distance between the source and IP1 is  $x$  km and the camel had 5 trips.
- From IP1 to IP2 its  $3y$  bananas, as the distance between IP1 and IP2 is  $y$  km and the camel had 3 trips.
- From IP2 to destination its  $z$  bananas.

Sanjukta Sen  
RA1911028010039

Camel Banana Problem Solution

Source — IP1 — IP2 — Destination.  
3000                       $x$  km      2000       $y$  km      1000       $z$  km

~~Before~~ Calculating the distance between the points:

1.  $3000 - 5x = 2000$                        $\therefore x = 200$
2.  $2000 - 3y = 1000$        $y = 333.33$  but here the distance is also the number of bananas and it cannot be fraction so we take  $y = 333$  and at IP2 we have the number of bananas equal 1001, so its  $2000 - 3y = 1001$
3. So, the remaining distance to the market is  
 $1000 - x - y = z$  i.e.  $1000 - 200 - 333 = z = 467$

Now there are 1001 bananas at IP2.

$\therefore$  Remaining bananas are  $1001 - 467 = 534$

## Code:

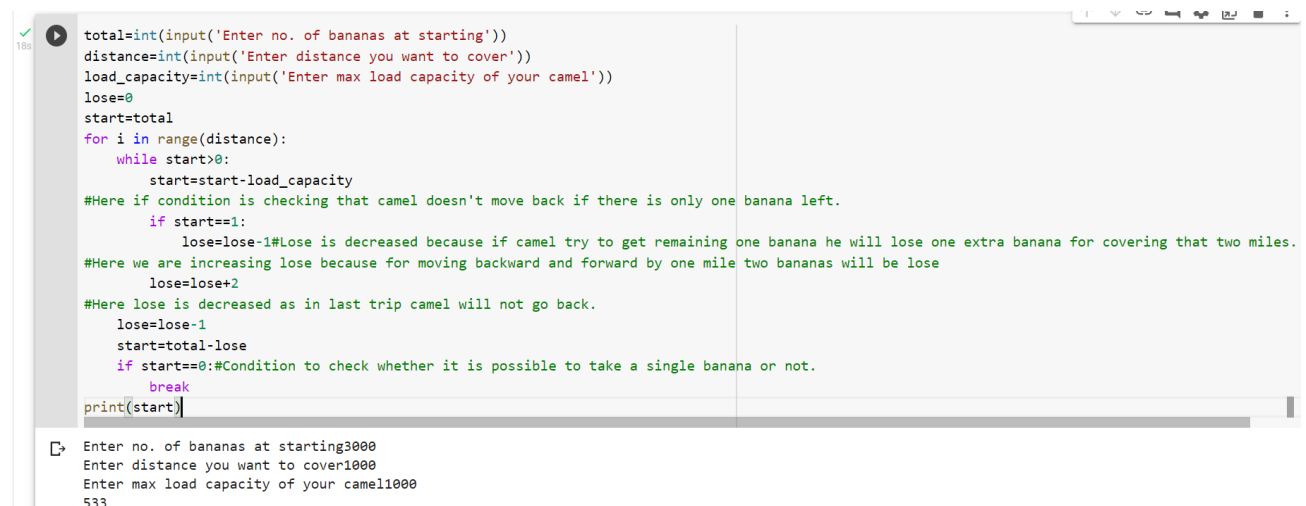
```
total=int(input('Enter no. of bananas at starting'))
distance=int(input('Enter distance you want to cover'))
load_capacity=int(input('Enter max load capacity of your camel'))
lose=0
start=total
for i in range(distance):
    while start>0:
        start=start-load_capacity
#Here if condition is checking that camel doesn't move back if there is
only one banana left.
    if start==1:
```

```

        lose=lose-1 #Lose is decreased because if camel try to get
remaining one banana he will lose one extra banana for covering that two
miles.
#Here we are increasing lose because for moving backward and forward by
one mile two bananas will be lose
        lose=lose+2
#Here lose is decreased as in last trip camel will not go back.
        lose=lose-1
        start=start-lose
        if start==0: #Condition to check whether it is possible to take a
single banana or not.
            break
print(start)

```

## Output:



```

total=int(input('Enter no. of bananas at starting'))
distance=int(input('Enter distance you want to cover'))
load_capacity=int(input('Enter max load capacity of your camel'))
lose=0
start=total
for i in range(distance):
    while start>0:
        start=start-load_capacity
        #Here if condition is checking that camel doesn't move back if there is only one banana left.
        if start==1:
            lose=lose-1#Lose is decreased because if camel try to get remaining one banana he will lose one extra banana for covering that two miles.
        #Here we are increasing lose because for moving backward and forward by one mile two bananas will be lose
            lose=lose+2
        #Here lose is decreased as in last trip camel will not go back.
            lose=lose-1
            start=start-lose
            if start==0:#Condition to check whether it is possible to take a single banana or not.
                break
print(start)

```

Enter no. of bananas at starting3000  
 Enter distance you want to cover1000  
 Enter max load capacity of your camel1000  
 533

```

Enter no. of bananas at starting3000
Enter distance you want to cover1000
Enter max load capacity of your camel1000
533

```

## Result:

Theoretically, the maximum number of bananas that can be transported using one camel, keeping all boundary conditions in mind is 534.

Experimentally, the maximum number of bananas that can be transported using one camel is 533.

Theoretical value  $\approx$  Experimental value

Therefore, we can say that the camel banana problem has been successfully implemented and the final number of bananas that can be transported is verified.

## **Experiment-2**

### **Developing agent programs for real world problem - Graph coloring problem**

#### **Problem Statement:**

Check whether the given graph is Bipartite or not.

**Tool Used:** Google Collab

#### **Algorithm:**

Algorithm to check if a graph is Bipartite:

One approach is to check whether the graph is 2-colorable or not using backtracking algorithm in coloring problem.

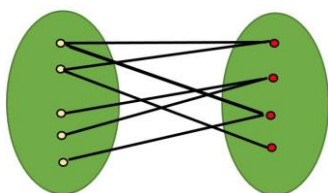
Following is a simple algorithm to find out whether a given graph is Bipartite or not using Breadth First Search (BFS).

Assign RED color to the source vertex (putting into set U).

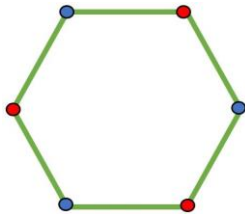
1. Color all the neighbors with BLUE color (putting into set V).
2. Color all neighbor's neighbor with RED color (putting into set U).
3. This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where  $m = 2$ .
4. While assigning colors, if we find a neighbor which is colored with same color as current vertex, then the graph cannot be colored with 2 vertices (or graph is not Bipartite)

#### **Optimization technique:**

A Bipartite Graph is a graph whose vertices can be divided into two independent sets, U and V such that every edge  $(u, v)$  either connects a vertex from U to V or a vertex from V to U. In other words, for every edge  $(u, v)$ , either  $u$  belongs to U and  $v$  to V, or  $u$  belongs to V and  $v$  to U. We can also say that there is no edge that connects vertices of same set.

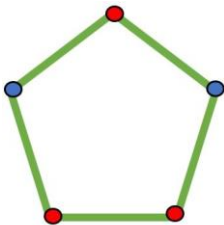


A bipartite graph is possible if the graph coloring is possible using two colors such that vertices in a set are colored with the same color. Note that it is possible to color a cycle graph with even cycle using two colors.



Cycle graph of length 6

It is not possible to color a cycle graph with odd cycle using two colors.



Cycle graph of length 5

Sanjukta Sen  
PA1911028010039

No. of vertex in the graph = 4

Adjacency matrix =

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

4x4

∴ graph found is

• let the colour used by A be black.

• In graph colouring, no two adjacent sides should have the same colour.

∴ Vertex B & C will have different colour.

• Vertex D can be coloured with black.

As only 2 colours were required to colour the graph

∴ The given graph is BIPARTITE.

Page No.

**Code:**

```
class Graph():
    def __init__(self, V):
        self.V = V
        self.graph = [[0 for column in range(V)] \
                       for row in range(V)]

    # This function returns true if graph G[V][V]
    # is Bipartite, else false
    def isBipartite(self, src):

        # Create a color array to store colors
        # assigned to all vertices. Vertex
        # number is used as index in this array.
        # The value '-1' of colorArr[i] is used to
        # indicate that no color is assigned to
        # vertex 'i'. The value 1 is used to indicate
        # first color is assigned and value 0
        # indicates second color is assigned.
        colorArr = [-1] * self.V

        # Assign first color to source
        colorArr[src] = 1

        # Create a queue (FIFO) of vertex numbers and
        # enqueue source vertex for BFS traversal
        queue = []
        queue.append(src)

        # Run while there are vertices in queue
        # (Similar to BFS)
```

```

while queue:

    u = queue.pop()

    # Return false if there is a self-loop
    if self.graph[u][u] == 1:
        return False;

    for v in range(self.V):

        # An edge from u to v exists and destination
        # v is not colored
        if self.graph[u][v] == 1 and colorArr[v] == -1:

            # Assign alternate color to this
            # adjacent v of u
            colorArr[v] = 1 - colorArr[u]
            queue.append(v)

        # An edge from u to v exists and destination
        # v is colored with same color as u
        elif self.graph[u][v] == 1 and colorArr[v] ==
colorArr[u]:

            return False

    # If we reach here, then all adjacent
    # vertices can be colored with alternate
    # color
    return True

# Driver program to test above function

```

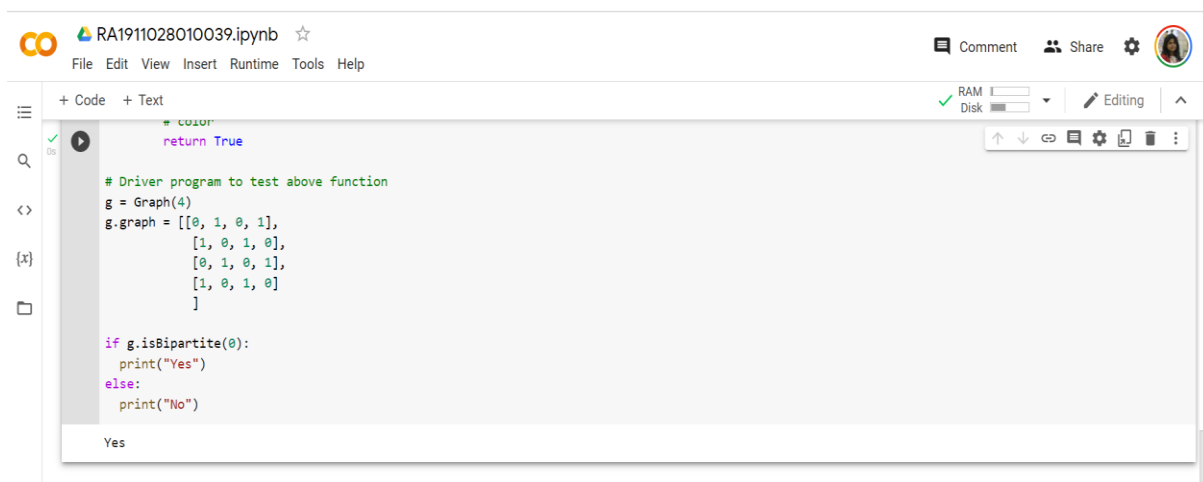


```
g = Graph(4)
g.graph = [[0, 1, 0, 1],
           [1, 0, 1, 0],
           [0, 1, 0, 1],
           [1, 0, 1, 0]
           ]

if g.isBipartite(0):
    print("Yes")
else:
    print("No")
```

## Output:

Yes



The screenshot shows a Jupyter Notebook interface with a file named 'RA1911028010039.ipynb'. The code cell contains the following Python code:

```
# Driver program to test above function
g = Graph(4)
g.graph = [[0, 1, 0, 1],
           [1, 0, 1, 0],
           [0, 1, 0, 1],
           [1, 0, 1, 0]
           ]

if g.isBipartite(0):
    print("Yes")
else:
    print("No")
```

The output of the code is 'Yes'.

## Result:

The given graph is Bipartite in nature as only 2 colors were used.

## **Experiment-3**

### **Implementation of Constraint Satisfaction Problem - Cryptarithmic puzzle**

#### **Problem Statement:**

To implement the constraint satisfaction problem using cryptarithmic puzzle.

**Tool Used:** Google Collab

#### **Algorithm & Optimization technique:**

Cryptarithmic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.

In **cryptarithmic problem**, the digits (0-9) get substituted by some possible alphabets or symbols. The task in cryptarithmic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

We can perform all the arithmetic operations on a given cryptarithmic problem.

**The rules or constraints on a cryptarithmic problem are as follows:**

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e.,  $2+2=4$ , nothing else.
- Digits should be from **0-9** only.
- There should be only one carry forward, while performing the addition operation on a problem.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

## Manual Procedure to Solve Cryptarithmic Problem:

Date \_\_\_\_/\_\_\_\_/\_\_\_\_

$SEND + MORE = MONEY$

$$\begin{array}{r} SEND \\ + MORE \\ \hline MONEY \end{array}$$

$S + M \rightarrow MO \rightarrow 9 + 1 = 10$

$S = 9 \quad M = 1 \quad O = 0$

$E + O \rightarrow N \rightarrow 5 + 0 = 5$   
1 carry

$N + R \rightarrow 6 + 8 = 14$   
but  $E = 5$ ,  $\Rightarrow$  wrong

$D + E \rightarrow Y \rightarrow 7 + 5 = 12$

All constraints

$S = 9$	$D = 7$	$R = 8$
$E = 5$	$M = 1$	$Y = 2$
$N = 6$	$O = 0$	

For,  $BASE + BALL = GAMES$

$$\begin{array}{r} BASE \\ + BALL \\ \hline GAMES \end{array}$$

All constraints

$B = 7$	$L = 5$
$A = 4$	$G = 1$
$S = 8$	$M = 9$
$E = 3$	

## Code:

```
import string
import itertools
inListNumsAsStringArray = [ ['BASE', 'BALL'],
                             ['SEND', 'MORE'] ]
inResultsArray = [ 'GAMES',
                   'MONEY' ]
```

```

inPossibleNumsAsStr = '0123456789'

# Input: (string, dictionary)
#   string 'AB'
#   key-
value pairs for characters and numbers. Ex: dictCharAndDigit = {'A': '1', 'B': '2'}
# Output: (Number)
#   12
def getNumberFromStringAndMappingInfo(inStr, inDictMapping):
    numAsStr = ''
    for ch in inStr:
        numAsStr = numAsStr + inDictMapping[ch]
    return int(numAsStr)

def solveCryptarithmicBruteForce(inListNumsAsString, inResultStr, inPossibleNumsAsStr):
    nonZeroLetters = []
    strFromStrList = ''
    for numStr in inListNumsAsString:
        nonZeroLetters.append(numStr[0])
        strFromStrList = strFromStrList + numStr
    nonZeroLetters.append(inResultStr[0])
    strFromStrList = strFromStrList + inResultStr
    uniqueStrs = ''.join(set(strFromStrList))
    for tup in itertools.permutations(inPossibleNumsAsStr, len(uniqueStrs)):
        dictCharAndDigit = {}
        for i in range(len(uniqueStrs)):
            dictCharAndDigit[uniqueStrs[i]] = tup[i]
        nonZeroLetterIsZero = False
        for letter in nonZeroLetters:
            if(dictCharAndDigit[letter] == '0'):
                nonZeroLetterIsZero = True
                break
        if(nonZeroLetterIsZero == True):
            continue
        result = getNumberFromStringAndMappingInfo(inResultStr, dictCharAndDigit)
        testResult = 0
        for numStr in inListNumsAsString:
            testResult = testResult + getNumberFromStringAndMappingInfo(numStr, dictCharAndDigit)

        if(testResult == result):
            strToPrint = ''
            for numStr in inListNumsAsString:

```

```

        strToPrint = strToPrint + numStr + '(' + str(getNumberFromStringAndMappingInfo(numStr, dictCharAndDigit)) + ')' + ' + '
        strToPrint = strToPrint[:-3]
        strToPrint = strToPrint + ' = ' + inResultStr + '(' + str(result) + ')'
        print(strToPrint)
        break

for i in range(len(inResultsArray)):
    solveCryptarithmicBruteForce(inListNumsAsStringArray[i], inResultsArray[i], inPossibleNumsAsStr)

```

## Output:



RA1911028010039 exp-3 ☆

File Edit View Insert Runtime Tools Help [All changes saved](#)

+ Code + Text

```

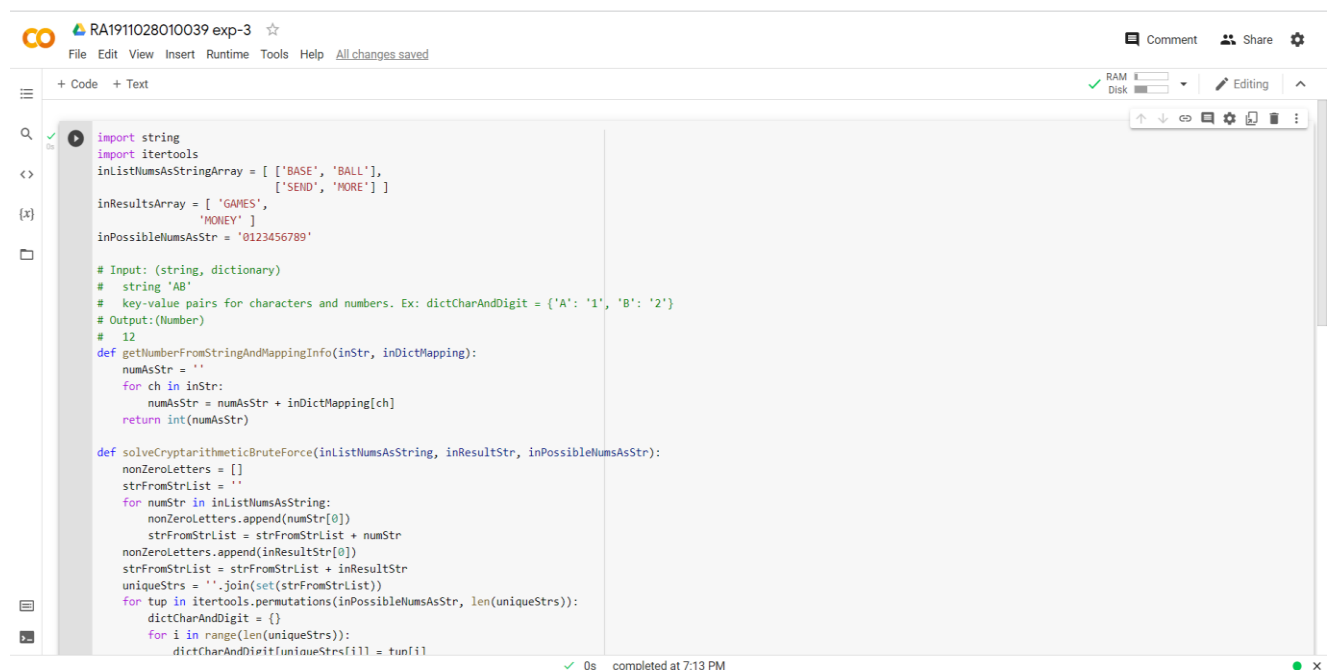
        strToPrint = strToPrint + numStr + '(' + str(getNumberFromStringAndMappingInfo(numStr, dictCharAndDigit)) + ')' + ' + '
        strToPrint = strToPrint[:-3]
        strToPrint = strToPrint + ' = ' + inResultStr + '(' + str(result) + ')'
        print(strToPrint)
        break

for i in range(len(inResultsArray)):
    solveCryptarithmicBruteForce(inListNumsAsStringArray[i], inResultsArray[i], inPossibleNumsAsStr)

```

BASE(7483) + BALL(7455) = GAMES(14938)  
 SEND(9567) + MORE(1085) = MONEY(10652)

## Screenshot:



RA1911028010039 exp-3 ☆

File Edit View Insert Runtime Tools Help [All changes saved](#)

+ Code + Text

```

import string
import itertools
inListNumsAsStringArray = [ ['BASE', 'BALL'],
                             ['SEND', 'MORE'] ]
inResultsArray = [ 'GAMES',
                   'MONEY' ]
inPossibleNumsAsStr = '0123456789'

# Input: (string, dictionary)
# string 'AB'
# key-value pairs for characters and numbers. Ex: dictCharAndDigit = {'A': '1', 'B': '2'}
# Output: (Number)
# 12
def getNumberFromStringAndMappingInfo(inStr, inDictMapping):
    numAsStr = ''
    for ch in inStr:
        numAsStr = numAsStr + inDictMapping[ch]
    return int(numAsStr)

def solveCryptarithmicBruteForce(inListNumsAsString, inResultStr, inPossibleNumsAsStr):
    nonZeroLetters = []
    strFromStrList = ''
    for numStr in inListNumsAsString:
        nonZeroLetters.append(numStr[0])
        strFromStrList = strFromStrList + numStr
    nonZeroLetters.append(inResultStr[0])
    strFromStrList = strFromStrList + inResultStr
    uniqueStrs = ''.join(set(strFromStrList))
    for tup in itertools.permutations(inPossibleNumsAsStr, len(uniqueStrs)):
        dictCharAndDigit = {}
        for i in range(len(uniqueStrs)):
            dictCharAndDigit[uniqueStrs[i]] = tup[i]

```

0s completed at 7:13 PM



```
dictCharAndDigit = {}
for i in range(len(uniqueStrs)):
    dictCharAndDigit[uniqueStrs[i]] = tup[i]
nonZeroLetterIsZero = False
for letter in nonZeroLetters:
    if(dictCharAndDigit[letter] == '0'):
        nonZeroLetterIsZero = True
        break
if(nonZeroLetterIsZero == True):
    continue
result = getNumberFromStringAndMappingInfo(inResultStr, dictCharAndDigit)
testResult = 0
for numStr in inListNumsAsString:
    testResult = testResult + getNumberFromStringAndMappingInfo(numStr, dictCharAndDigit)

if(testResult == result):
    strToPrint = ''
    for numStr in inListNumsAsString:
        strToPrint = strToPrint + numStr + '(' + str(getNumberFromStringAndMappingInfo(numStr, dictCharAndDigit)) + ')' + ' + ' + '
    strToPrint = strToPrint[:-3]
    strToPrint = strToPrint + ' = ' + inResultStr + '(' + str(result) + ')'
    print(strToPrint)
    break

for i in range(len(inResultsArray)):
    solveCryptarithmicBruteForce(inListNumsAsStringArray[i], inResultsArray[i], inPossibleNumsAsString)
```

BASE(7483) + BALL(7455) = GAMES(14938)  
SEND(9567) + MORE(1085) = MONEY(10652)

## Result:

Constraint Satisfaction Problem, ie, Cryptarithmic problem was implemented using Python 3.

## **Experiment 4:**

### **Implementation and Analysis of DFS and BFS for an application**

**Problem Statement:** Given a grid where each cell can take '#' or '.' Goal is to find the longest path that one can go through in this grid using DFS, given a binary tree find the depth of binary tree using BFS.

**Tool:** AWS

#### **Algorithm (DFS):**

1. Create a recursive function that takes the index of node and a visited array.
2. Mark the current node as visited and print the node.
3. Traverse all the adjacent and unmarked nodes and call the recursive function with index of adjacent node.

#### **Algorithm (BFS):**

1. Start with node and push that node into the queue
2. Pop n nodes from the queue and process it and push all its child nodes into the queue
3. Process all the nodes present in the queue in similar manner until queue is empty.

#### **Optimization Technique:**

The general approach of the given problem is through array traversal. To implement it through array traversal, the entire array will have to be traversed multiple times to check for neighbouring nodes making the program inefficient.

Using breadth first search algorithm or depth first search algorithm to work on this problem reduces the time complexity drastically as all the neighbouring nodes are assigned within traversing the array just once and then the islands or disconnected graphs can be found much quicker, thus, optimizing the code.

## Code

### (DFS):

```
from collections import defaultdict

# Class to represent a graph class Graph:

def __init__(self, vertices):
    self.graph = defaultdict(list) # dictionary containing adjacency List self.V = vertices # No.
    of vertices

# function to add an edge to graph def addEdge(self, u, v):

self.graph[u].append(v)

# A recursive function used by topologicalSort def topologicalSortUtil(self, v, visited,
stack):

# Mark the current node as visited. visited[v] = True

# Recur for all the vertices adjacent to this vertex for i in self.graph[v]:

if visited[i] == False:

self.topologicalSortUtil(i, visited, stack)

# Push current vertex to stack which stores result stack.append(v)

# The function to do Topological Sort. It uses recursive # topologicalSortUtil()
def topologicalSort(self):

# Mark all the vertices as not visited visited = [False]*self.V
stack = []

# Call the recursive helper function to store Topological # Sort starting from all vertices
one by one
for i in range(self.V):

if visited[i] == False: self.topologicalSortUtil(i, visited, stack)

# Print contents of the stack
print(stack[::-1]) # return list in reverse order

# Driver Code
g = Graph(6) g.addEdge(5, 2) g.addEdge(5, 0) g.addEdge(4, 0) g.addEdge(4, 1)
g.addEdge(2, 3) g.addEdge(3, 1)

print ("Following is a Topological Sort of the given graph")

# Function Call g.topologicalSort()
```



### Code (BFS):

```
from collections import defaultdict
# This class represents a directed graph # using adjacency matrix representation class
Graph:

def _init_(self, graph):
self.graph = graph # residual graph self. ROW = len(graph)
# self.COL = len(gr[0])

'''Returns true if there is a path from source 's' to sink 't' in residual graph. Also fills
parent[] to store the path '''

def BFS(self, s, t, parent):

# Mark all the vertices as not visited visited = [False]*(self.ROW)

# Create a queue for BFS queue = []

# Mark the source node as visited and enqueue it queue.append(s)
visited[s] = True

# Standard BFS Loop while queue:

# Dequeue a vertex from queue and print it u = queue.pop(0)

# Get all adjacent vertices of the dequeued vertex u # If a adjacent has not been visited,
then mark it
# visited and enqueue it
for ind, val in enumerate(self.graph[u]):

if visited[ind] == False and val > 0:
# If we find a connection to the sink node,

# then there is no point in BFS anymore
# We just have to set its parent and can return true queue.append(ind)
visited[ind] = True
parent[ind] = u
if ind == t:

return True

# We didn't reach sink in BFS starting # from source, so return false
return False

# Returns the maximum flow from s to t in the given graph def FordFulkerson(self, source,
sink):

# This array is filled by BFS and to store path parent = [-1]*(self.ROW)

max_flow = 0 # There is no flow initially
```

```
# Augment the flow while there is path from source to sink while self.BFS(source, sink,
parent) :
```

```
# Find minimum residual capacity of the edges along the # path filled by BFS. Or we can
say find the maximum flow # through the path found.
```

```
path_flow = float("Inf")
```

```
s = sink
```

```
while(s != source):
```

```
path_flow = min (path_flow, self.graph[parent[s]][s]) s = parent[s]
```

```
# Add path flow to overall flow max_flow += path_flow
```

```
# update residual capacities of the edges and reverse edges # along the path
```

```
v = sink
```

```
while(v != source):
```

```
u = parent[v] self.graph[u][v] -= path_flow self.graph[v][u] += path_flow v = parent[v]
```

```
return max_flow
```

```
# Create a graph given in the above diagram
```

```
graph = [[0, 16, 13, 0, 0, 0], [0, 0, 10, 12, 0, 0],
```

```
[0, 4, 0, 0, 14, 0],
```

```
[0, 0, 9, 0, 0, 20],
```

```
[0, 0, 0, 7, 0, 4], [0, 0, 0, 0, 0, 0]]
```

```
g = Graph(graph)
```

```
source = 0; sink = 5
```

```
print ("The maximum possible flow is %d " % g.FordFulkerson(source, sink))
```

## Time

### complexity:

BFS- $O(B^{d+1})$

DFS- $O(B^m)$

B=branching factor, d=depth of the tree, m=maximum depth

## Space

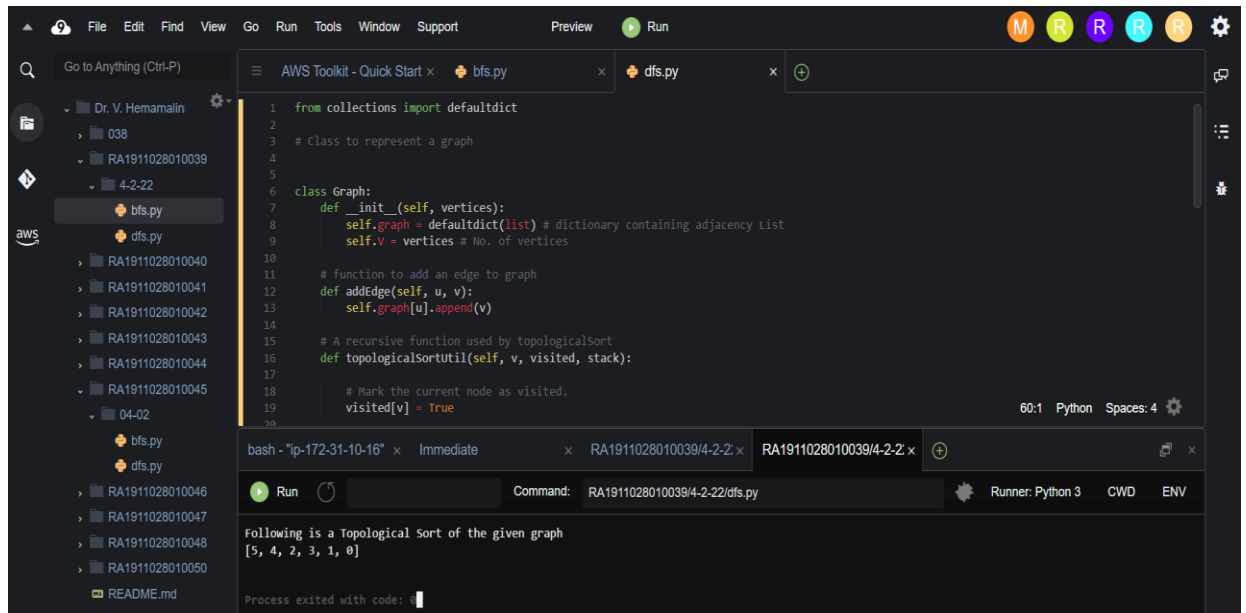
### complexity:

BFS-  $O(B^{d+1})$

DFS- $O(B^m)$

## Output:

### 1. DFS

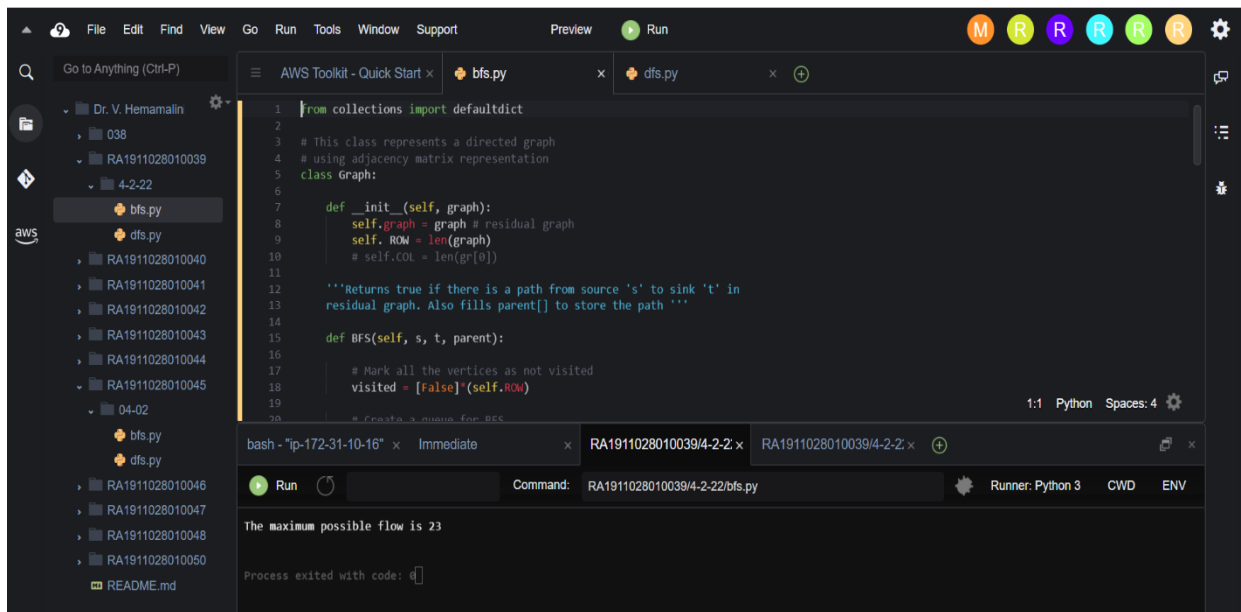


The screenshot shows the AWS Toolkit IDE with a Python file named `dfs.py` open. The code implements a Depth-First Search (DFS) algorithm. The output window displays the following text:

```
Following is a Topological Sort of the given graph
[5, 4, 2, 3, 1, 0]
```

The process exited with code: 0.

### 2.BFS



The screenshot shows the AWS Toolkit IDE with a Python file named `bfs.py` open. The code implements a Breadth-First Search (BFS) algorithm. The output window displays the following text:

```
The maximum possible flow is 23
```

The process exited with code: 0.

**Result:** The DFS and BFS algorithm was implemented using recursive calls and queue approaches in an application and solution was found.

## **Experiment-5**

### **Developing Best first search and A\* algorithm for real world problem**

#### **Problem Statement:**

To develop and implement A\* algorithm and Best first search for real world problems.

**Tool:** AWS

#### **A\* Algorithm-**

Given a MxN matrix where each element can either be 0 or 1. We need to find the shortest path between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

Expected time complexity is  $O(MN)$ .

#### **Best First Search-**

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So, both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So, the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

#### **Algorithm for A\*:**

1. We start from the source cell and calls BFS procedure.
2. We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.
3. We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.
4. We LOOP till queue is not empty
5. Dequeue front cell from the queue
6. Return if the destination coordinates have reached.
7. For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited.

**Algorithm for Best First Search:**

1) Create an empty PriorityQueue

    PriorityQueue pq;

2) Insert "start" in pq.

    pq.insert(start)

3) Until PriorityQueue is empty

    u = PriorityQueue.DeleteMin

    If u is the goal

        Exit

    Else

        Foreach neighbor v of u

            If v "Unvisited"

                Mark v "Visited"

                pq.insert(v)

        Mark u "Examined"

End procedure

**Code for A\*:**

```
import sys
```

```
# Check if it is possible to go to (x, y) from the current position. The
```

```
# function returns false if the cell is invalid, has value 0 or already visited
```

```
def isSafe(mat, visited, x, y):
```

```
    return 0 <= x < len(mat) and 0 <= y < len(mat[0]) and \
```

```
        not (mat[x][y] == 0 or visited[x][y])
```

```
# Find the shortest possible route in a matrix `mat` from source cell (i, j)
```

```
# to destination cell `dest`.
```

```
# `min_dist` stores the length of the longest path from source to a destination  
# found so far, and `dist` maintains the length of the path from a source cell to  
# the current cell (i, j).
```

```
def findShortestPath(mat, visited, i, j, dest, min_dist=sys.maxsize, dist=0):
```

```
    # if the destination is found, update `min_dist`
```

```
    if (i, j) == dest:
```

```
        return min(dist, min_dist)
```

```
    # set (i, j) cell as visited
```

```
    visited[i][j] = 1
```

```
    # go to the bottom cell
```

```
    if isSafe(mat, visited, i + 1, j):
```

```
        min_dist = findShortestPath(mat, visited, i + 1, j, dest, min_dist, dist + 1)
```

```
    # go to the right cell
```

```
    if isSafe(mat, visited, i, j + 1):
```

```
        min_dist = findShortestPath(mat, visited, i, j + 1, dest, min_dist, dist + 1)
```

```
    # go to the top cell
```

```
    if isSafe(mat, visited, i - 1, j):
```

```
        min_dist = findShortestPath(mat, visited, i - 1, j, dest, min_dist, dist + 1)
```

```
    # go to the left cell
```

```
    if isSafe(mat, visited, i, j - 1):
```

```
        min_dist = findShortestPath(mat, visited, i, j - 1, dest, min_dist, dist + 1)
```

```
    # backtrack: remove (i, j) from the visited matrix
```

```
visited[i][j] = 0
```

```
return min_dist
```

```
# Wrapper over findShortestPath() function
```

```
def findShortestPathLength(mat, src, dest):
```

```
    # get source cell (i, j)
```

```
    i, j = src
```

```
    # get destination cell (x, y)
```

```
    x, y = dest
```

```
    # base case
```

```
    if not mat or len(mat) == 0 or mat[i][j] == 0 or mat[x][y] == 0:
```

```
        return -1
```

```
    # `M × N` matrix
```

```
    (M, N) = (len(mat), len(mat[0]))
```

```
    # construct an `M × N` matrix to keep track of visited cells
```

```
    visited = [[False for _ in range(N)] for _ in range(M)]
```

```
    min_dist = findShortestPath(mat, visited, i, j, dest)
```

```
    if min_dist != sys.maxsize:
```

```
        return min_dist
```

```
    else:
```

```
        return -1
```

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

```
    mat = [  
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],  
        [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],  
        [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],  
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],  
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],  
        [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],  
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],  
        [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],  
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],  
        [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]  
    ]
```

```
    src = (0, 0)
```

```
    dest = (7, 5)
```

```
    min_dist = findShortestPathLength(mat, src, dest)
```

```
    if min_dist != -1:
```

```
        print("The shortest path from source to destination has length", min_dist)
```

```
    else:
```

```
        print("Destination cannot be reached from source")
```

### **Code for Best First Search:**

```
from queue import PriorityQueue
```

```
v = 5
```

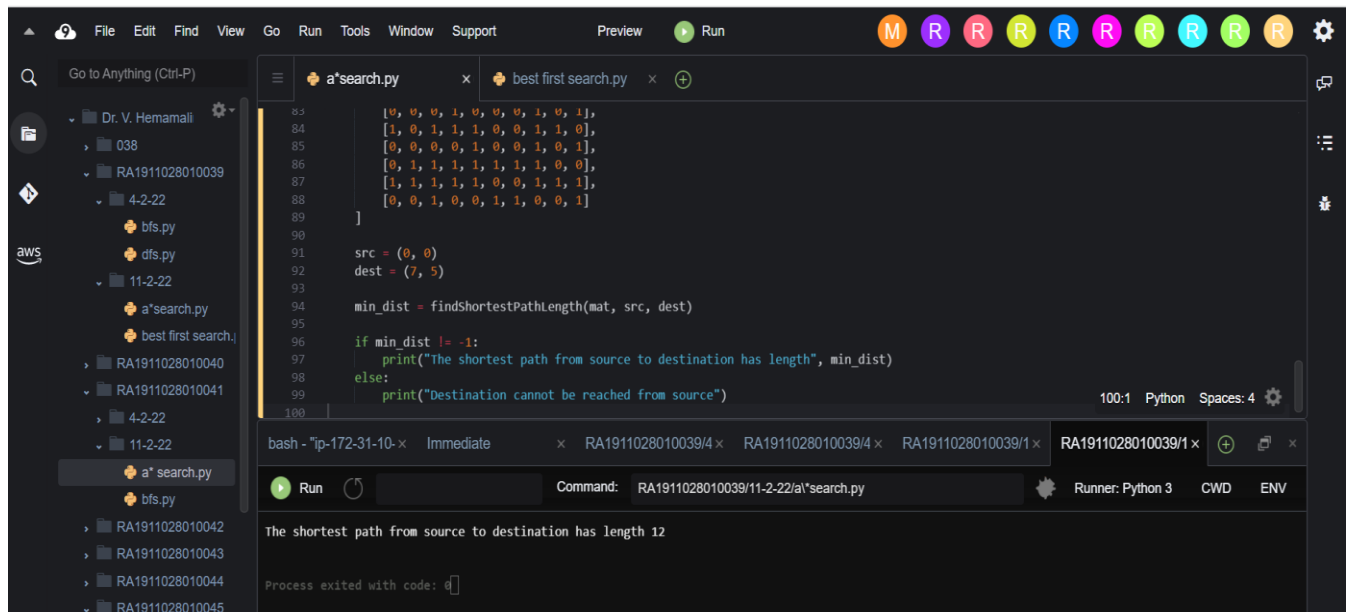


```

graph = [[] for i in range(v)]
def best_first_search(source, target, n):
    visited = [0] * n
    visited[0] = True
    pq = PriorityQueue()
    pq.put((0, source))
    while pq.empty() == False:
        u = pq.get()[1]
        print(u, end=" ")
        if u == target:
            break
        for v, c in graph[u]:
            if visited[v] == False:
                visited[v] = True
                pq.put((c, v))
    print()
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
adddedge(0, 1, 5)
adddedge(0, 2, 1)
adddedge(2, 3, 2)
adddedge(1, 4, 1)
adddedge(3, 4, 2)
source = 0
target = 4
best_first_search(source, target, v)

```

## Output for A\*:



The screenshot shows the AWS Cloud9 IDE interface. The left sidebar displays a file explorer with a project structure including folders like '038', 'RA1911028010039', '4-2-22', '11-2-22', and files like 'bfs.py', 'dfs.py', 'a\*search.py', and 'best first search.py'. The main editor window has two tabs: 'a\*search.py' and 'best first search.py'. The 'a\*search.py' tab is active, showing a Python script that defines a graph, sets source and destination nodes, and calls a function to find the shortest path. The output console at the bottom shows the command 'RA1911028010039/11-2-22/a\*search.py' being executed, resulting in the message 'The shortest path from source to destination has length 12'.

```
83     [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
84     [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
85     [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
86     [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
87     [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
88     [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
89 ]
90
91 src = (0, 0)
92 dest = (7, 5)
93
94 min_dist = findShortestPathLength(mat, src, dest)
95
96 if min_dist != -1:
97     print("The shortest path from source to destination has length", min_dist)
98 else:
99     print("Destination cannot be reached from source")
```

100:1 Python Spaces: 4

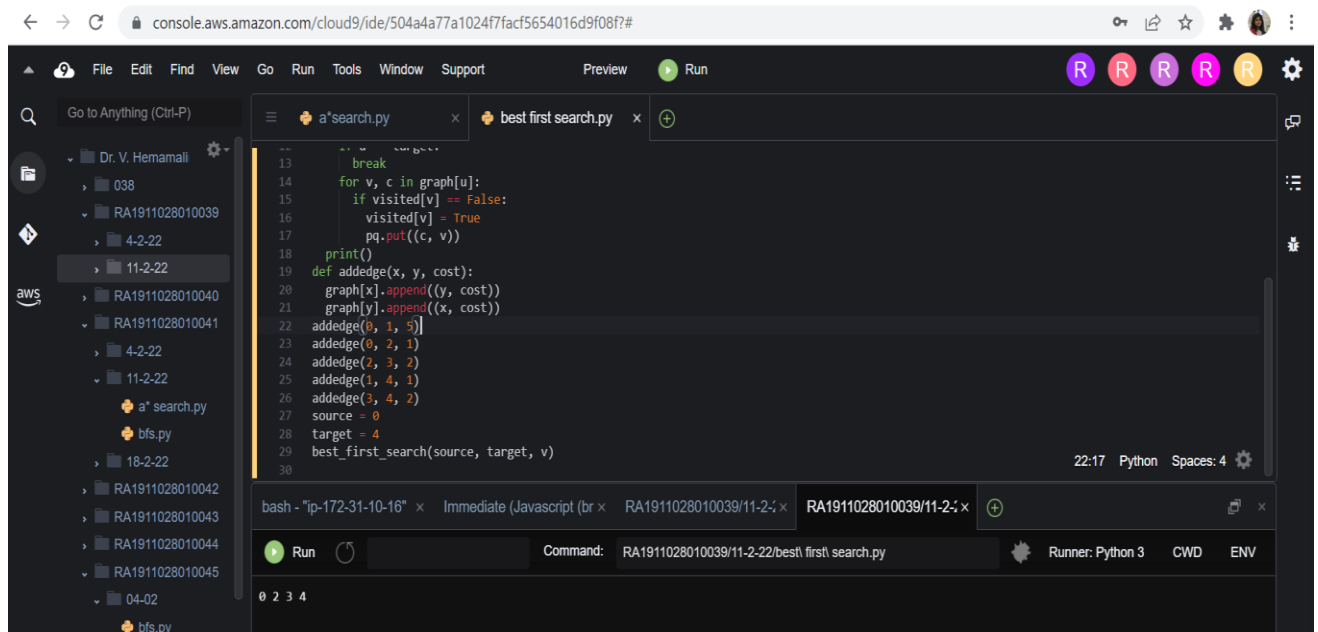
bash - "ip-172-31-10-16" x Immediate x RA1911028010039/4 x RA1911028010039/4 x RA1911028010039/1 x RA1911028010039/1 x

Run Command: RA1911028010039/11-2-22/a\*search.py Runner: Python 3 CWD ENV

The shortest path from source to destination has length 12

Process exited with code: 0

## Output for Best First Search:



The screenshot shows the AWS Cloud9 IDE interface. The left sidebar displays a file explorer with a project structure including folders like '038', 'RA1911028010039', '4-2-22', '11-2-22', and files like 'a\* search.py', 'bfs.py', and '18-2-22'. The main editor window has two tabs: 'a\*search.py' and 'best first search.py'. The 'best first search.py' tab is active, showing a Python script that defines a graph, sets source and target nodes, and calls a function to find the shortest path. The output console at the bottom shows the command 'RA1911028010039/11-2-22/best first search.py' being executed, resulting in the message '0 2 3 4'.

```
13     break
14     for v, c in graph[u]:
15         if visited[v] == False:
16             visited[v] = True
17             pq.put((c, v))
18     print()
19 def addedge(x, y, cost):
20     graph[x].append((y, cost))
21     graph[y].append((x, cost))
22 addedge(0, 1, 5)
23 addedge(0, 2, 1)
24 addedge(2, 3, 2)
25 addedge(1, 4, 1)
26 addedge(3, 4, 2)
27 source = 0
28 target = 4
29 best_first_search(source, target, v)
30
```

22:17 Python Spaces: 4

bash - "ip-172-31-10-16" x Immediate (Javascript (br x RA1911028010039/11-2-22 x RA1911028010039/11-2-22 x

Run Command: RA1911028010039/11-2-22/best first search.py Runner: Python 3 CWD ENV

0 2 3 4

## Result:

Both Best First Search and A\* algorithm for real world problems were successfully developed and implemented using Python 3.

## Experiment-6

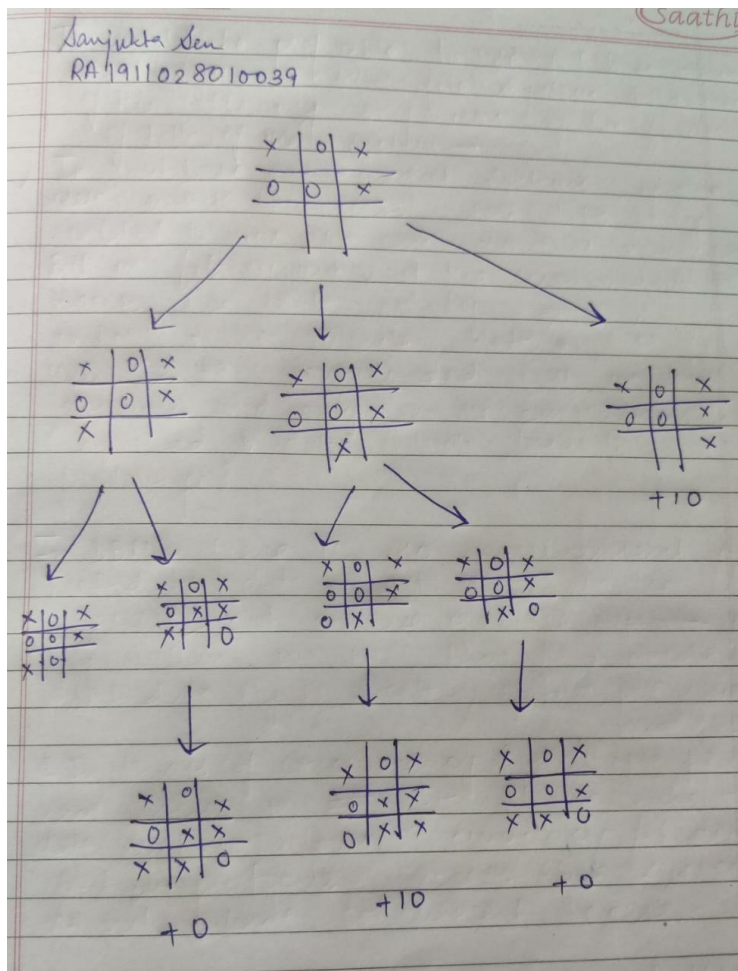
### Implementation of Min Max Algorithm For An Application

#### **Aim:**

To implement Min Max algorithm in Tic -Tac-Toe AI - Finding Optimal Move problem.

**Tool:** AWS

#### **Algorithm & Optimization technique:**



This image depicts all the possible paths that the game can take from the root board state. It is often called the Game Tree.

The 3 possible scenarios in the above example are :

- Left Move : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10;

- Middle Move : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0;
- Right Move : If X plays [2,2]. Then he will win the game. The value of this move is +10.

### Code:

# Python3 program to find the next optimal move for a player

player, opponent = 'x', 'o'

# This function returns true if there are moves

# remaining on the board. It returns false if

# there are no moves left to play.

def isMovesLeft(board) :

for i in range(3) :

for j in range(3) :

if (board[i][j] == '\_') :

return True

return False

# This is the evaluation function as discussed

# in the previous article ( <http://goo.gl/sJgv68> )

def evaluate(b) :

# Checking for Rows for X or O victory.

for row in range(3) :

if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :

if (b[row][0] == player) :

return 10

elif (b[row][0] == opponent) :

return -10

```
# Checking for Columns for X or O victory.
```

```
for col in range(3) :
```

```
    if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
```

```
        if (b[0][col] == player) :
```

```
            return 10
```

```
        elif (b[0][col] == opponent) :
```

```
            return -10
```

```
# Checking for Diagonals for X or O victory.
```

```
if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
```

```
    if (b[0][0] == player) :
```

```
        return 10
```

```
    elif (b[0][0] == opponent) :
```

```
        return -10
```

```
if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
```

```
    if (b[0][2] == player) :
```

```
        return 10
```

```
    elif (b[0][2] == opponent) :
```

```
        return -10
```

```
# Else if none of them have won then return 0
```

```
return 0
```

```
# This is the minimax function. It considers all
```

```
# the possible ways the game can go and returns
```

```

# the value of the board
def minimax(board, depth, isMax) :
    score = evaluate(board)

    # If Maximizer has won the game return his/her
    # evaluated score
    if (score == 10) :
        return score

    # If Minimizer has won the game return his/her
    # evaluated score
    if (score == -10) :
        return score

    # If there are no more moves and no winner then
    # it is a tie
    if (isMovesLeft(board) == False) :
        return 0

    # If this maximizer's move
    if (isMax) :
        best = -1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j]=='_') :

```

```

# Make the move
board[i][j] = player

# Call minimax recursively and choose
# the maximum value
best = max( best, minimax(board,
                           depth + 1,
                           not isMax) )

# Undo the move
board[i][j] = '_'
return best

# If this minimizer's move
else :
    best = 1000

# Traverse all cells
for i in range(3) :
    for j in range(3) :

        # Check if cell is empty
        if (board[i][j] == '_' ) :

            # Make the move
            board[i][j] = opponent

            # Call minimax recursively and choose
            # the minimum value
            best = min(best, minimax(board, depth + 1, not isMax))

```

```

        # Undo the move
        board[i][j] = '_'
    return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j] == '_' ) :

                # Make the move
                board[i][j] = player

                # compute evaluation function for this
                # move.
                moveVal = minimax(board, 0, False)

                # Undo the move
                board[i][j] = '_'

            # If the value of the current move is

```



```

        # more than the best value, then update

        # best/
    if (moveVal > bestVal) :
        bestMove = (i, j)
        bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove

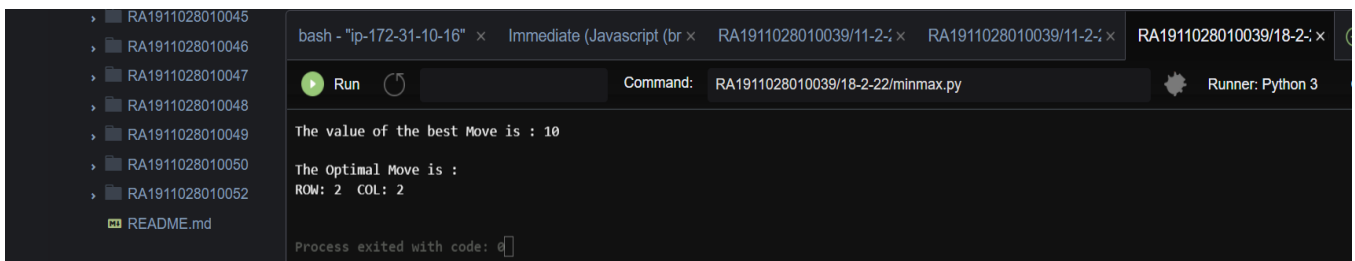
# Driver code
board = [
    ['o', 'x', 'o'],
    ['o', 'x', 'x'],
    ['_', '_', '_']
]

bestMove = findBestMove(board)

print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])

```

### Output:



The screenshot shows a code editor with a file explorer on the left and a terminal on the right. The terminal output is as follows:

```

The value of the best Move is : 10
The Optimal Move is :
ROW: 2 COL: 2
Process exited with code: 0

```

### Result:

Min Max algorithm for an application- Tic-Tac-Toe AI – Finding optimal move was successfully implemented and verified using Python 3.

## **Experiment-7**

### **7a) Implementation of Unification Algorithm**

#### **Problem Statement:**

To implement unification algorithm.

**Tool Used:** AWS

#### **Algorithm:**

1. Initialize the substitution set to be empty.
2. Recursively unify atomic sentences:
  - Check for Identical expression match.
  - If one expression is a variable  $v_i$ , and the other is a term  $t_i$  which does not contain variable  $v_i$ , then:
    - Substitute  $t_i / v_i$  in the existing substitutions
    - Add  $t_i / v_i$  to the substitution setlist.
    - If both the expressions are functions, then function name must be similar, and the number of arguments must be the same in both the expression.

**For each pair of the following atomic sentences find the most general unifier (If exist).**

#### **Code:**

```
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__':

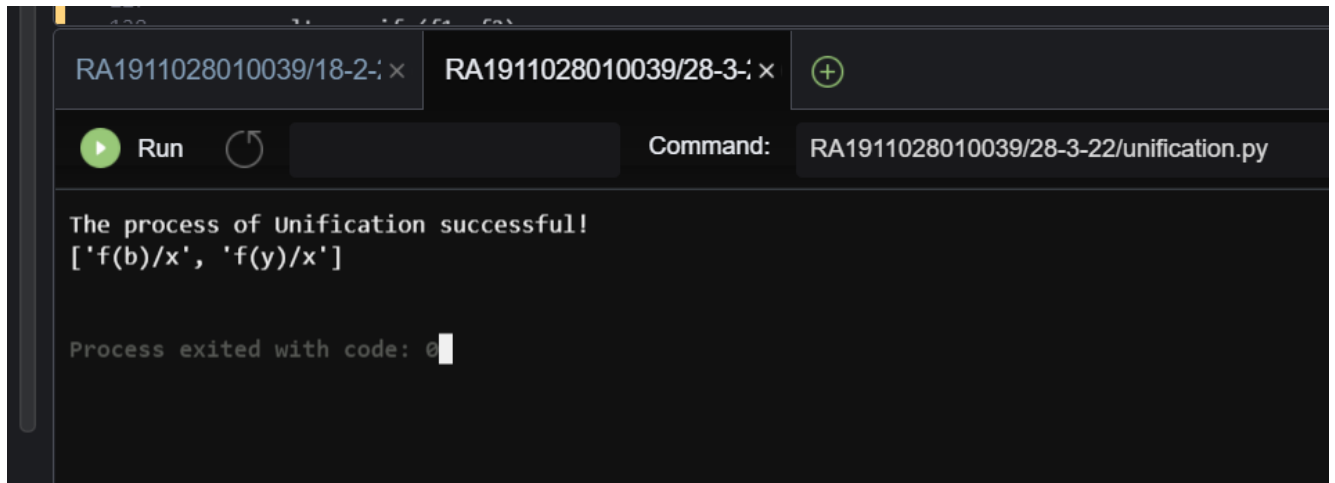
    f1 = 'Q(a, g(x, a), f(y))'
    f2 = 'Q(a, g(f(b), a), x)'
    # f1 = input('f1 : ')
    # f2 = input('f2 : ')

    result = unify(f1, f2)
    if not result:
        print('The process of Unification failed!')

```

```
else:  
    print('The process of Unification successful!')  
    print(result)
```

### Output:



```
RA1911028010039/18-2-; x RA1911028010039/28-3-; x  
Run Command: RA1911028010039/28-3-22/unification.py  
The process of Unification successful!  
['f(b)/x', 'f(y)/x']  
Process exited with code: 0
```

### Result:

Unification Algorithm was successfully implemented and verified using Python 3.

## 7b) Implementation of Resolution

### Problem Statement:

To implement resolution algorithm.

**Tool:** AWS

### Algorithm:

Resolution is used, if there are various statements are given, and we need to prove a conclusion of those statements. Unification is a key concept in proofs by resolutions. Resolution is a single inference rule which can efficiently operate on the **conjunctive normal form or clausal form**.

1. Conversion of facts into first-order logic.
2. Convert FOL statements into CNF

3. Negate the statement which needs to prove (proof by contradiction)
4. Draw resolution graph (unification).

**Code:**

```
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

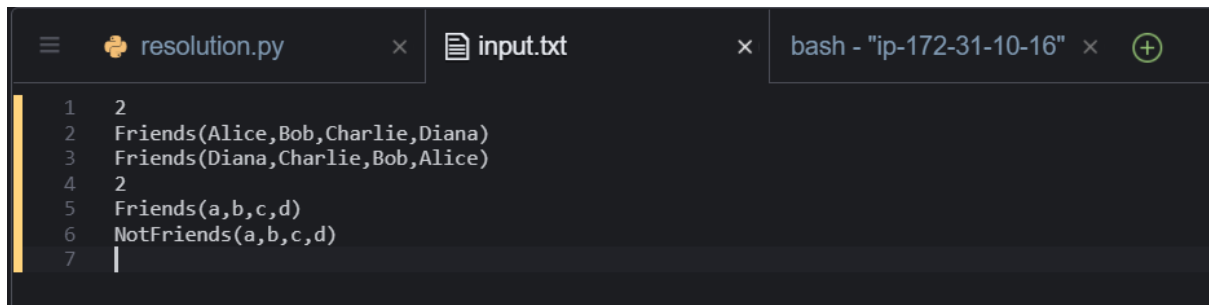
def printOutput(filename, results):
    print(results)
    with open(filename, "w") as file:
        for line in results:
            file.write(line)
            file.write("\n")
    file.close()

if __name__ == '__main__':
    inputQueries_, inputSentences_ = getInput('input.txt')
    knowledgeBase = KB(inputSentences_)
    knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
    printOutput("output.txt", results_)

```

**Input:**

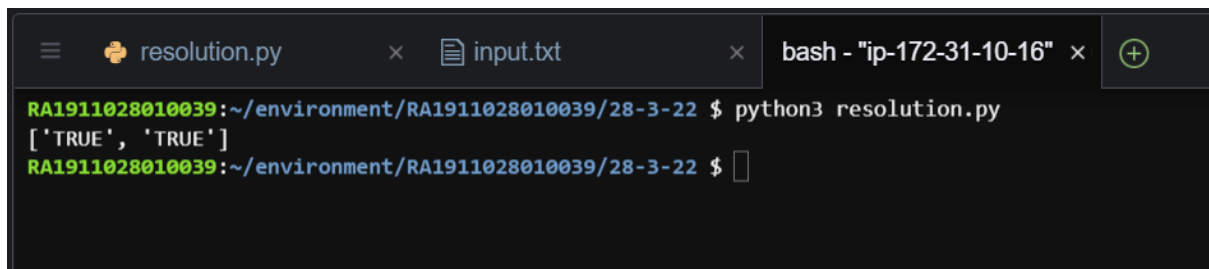




The screenshot shows a code editor with three tabs: 'resolution.py', 'input.txt', and 'bash - "ip-172-31-10-16"'. The 'resolution.py' tab is active, displaying the following code:

```
1 2
2 Friends(Alice,Bob,Charlie,Diana)
3 Friends(Diana,Charlie,Bob,Alice)
4 2
5 Friends(a,b,c,d)
6 NotFriends(a,b,c,d)
7 |
```

### Output:



The screenshot shows a terminal window with the same three tabs as the code editor. The 'bash - "ip-172-31-10-16"' tab is active, showing the command 'python3 resolution.py' being executed. The output is:

```
RA1911028010039:~/environment/RA1911028010039/28-3-22 $ python3 resolution.py
['TRUE', 'TRUE']
RA1911028010039:~/environment/RA1911028010039/28-3-22 $
```

### Result:

Resolution Algorithm was successfully implemented and verified using Python 3.