SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

18CSC305J - ARTIFICIAL INTELLIGENCE

REPORT SUBMITTED BY -

RAHUL GOEL

RA1911030010094

LAB 1 – Implementation of toy problems.

Aim: Implementation of toy problems

Problem Statement:

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).

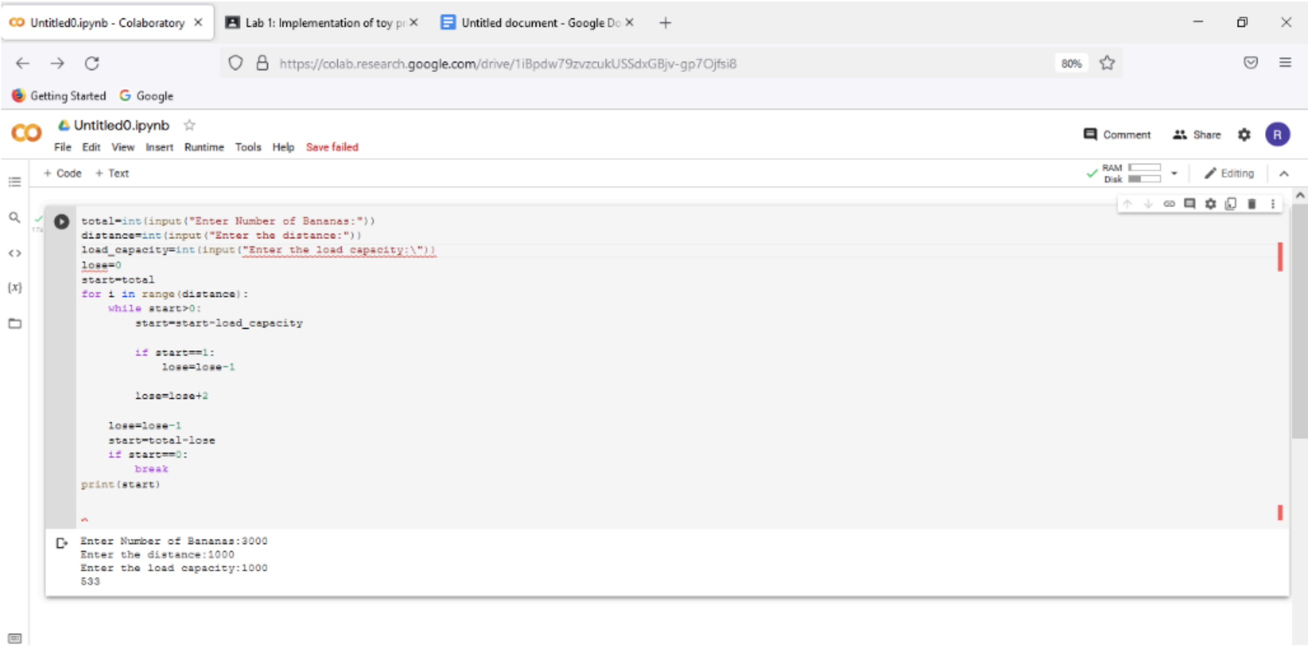
Code:

```
banana=int(input('Enter the total number of
bananas: ')) dist=int(input('Enter total distance to be
covered: '))
ip1 = banana-
dist ip2 =
banana-ip1
x=(banana-
ip1)/5 y=(ip1-
ip2)/3 z=ip2-x-
y max=ip2-z
print('maximum number of bananas camel can tranfer=',int(max))
```

Input:

3000
1000

Output:



Result: Hence the toy problem was implemented and the desired output was obtained.

LAB 2 - Developing agent programs for real world problems

AIM – Developing agent programs for real world problems by implementing graph coloring problem

Problem description:

Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color. This post will discuss a greedy algorithm for graph coloring and minimize the total number of colors used.

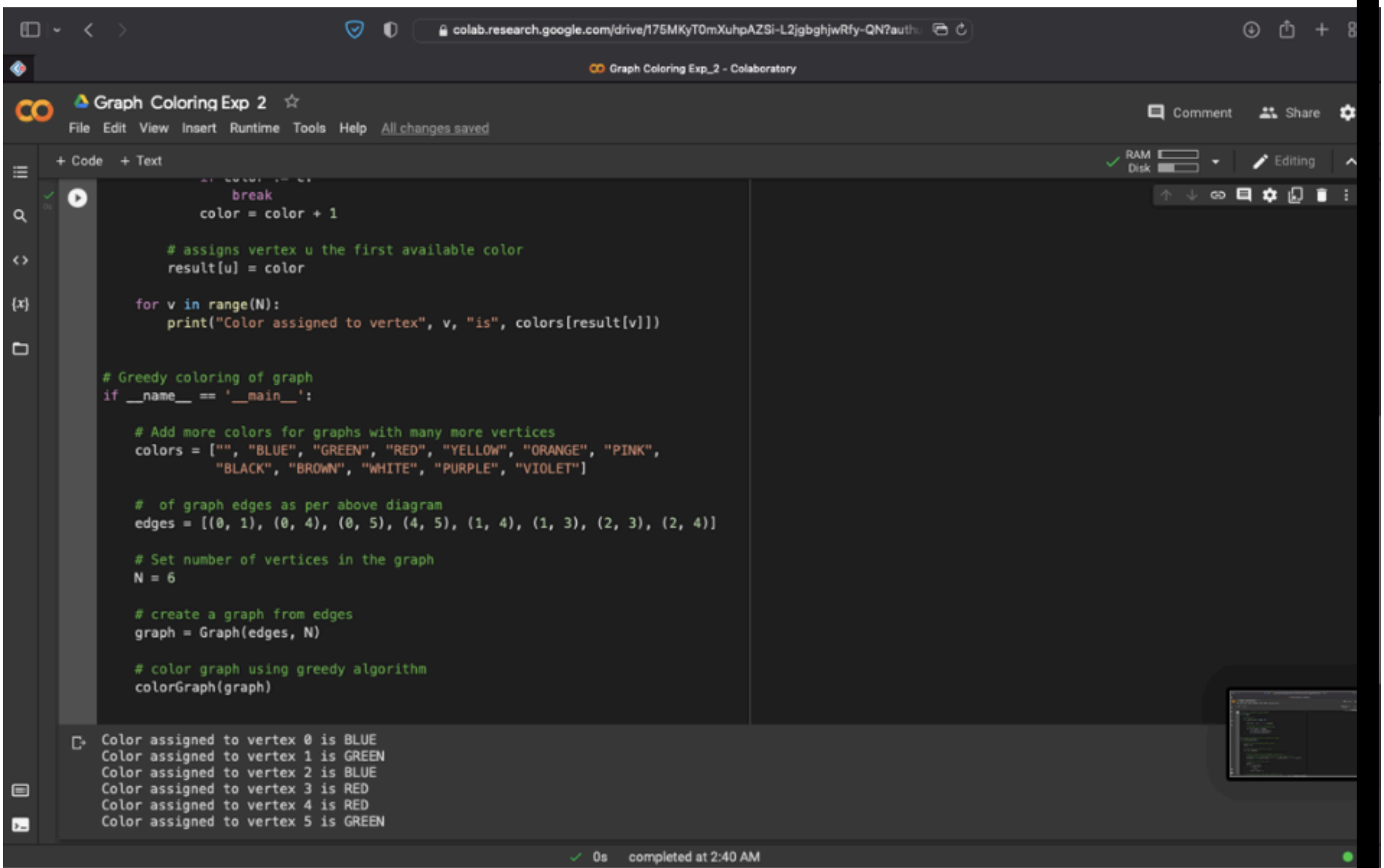
CODE:

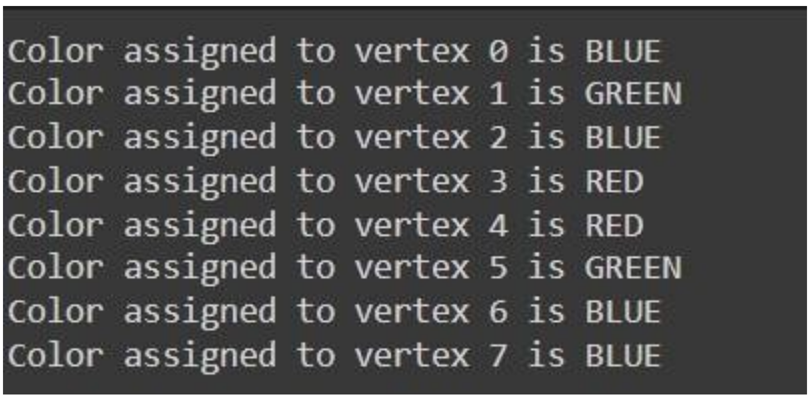
```
class Graph:
  def_init_(self, edges, n):
    self.adjList = [[]
for _ in range(n)]
     for (src, dest) in edges:
      self.adjList[src].append(dest)
      self.adjList[dest].append(src)
def colorGraph(graph, n):
   result = {}
      for u in range(n):
       assigned = set([result.get(i) for i in graph.adjList[u] if i in result])
         color = 1
      for c in assigned:
         if color != c:
             break
       color = color + 1
       result[u] = color
  for v in range(n):
     print(f'Color assigned to vertex {v} is{colors[result[v]]}')
if __name__== '_main_':
   colors = ['', 'BLUE', 'GREEN', 'RED', 'YELLOW', 'ORANGE', 'PINK',
           'BLACK', 'BROWN', 'WHITE', 'PURPLE', 'VOILET']
     edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]
n = 8
  graph = Graph(edges, n)
    colorGraph(graph, n)
```

OUTPUT:

Color assigned to vertex 0 is BLUE
Color assigned to vertex 1 is GREEN
Color assigned to vertex 2 is BLUE
Color assigned to vertex 3 is RED
Color assigned to vertex 4 is RED
Color assigned to vertex 5 is GREEN
Color assigned to vertex 6 is BLUE
Color assigned to vertex 7 is BLUE

SCREENSHOTS:





RESULT: Hence, Developing agent programs for real world problems was implemented using graph coloring problem.

LAB 3 - Constrain Satisfaction Problem

AIM: To implement Constraint satisfaction problem using python.

Problem Description:

In a CSP, we have a set of variables with known domains and a set of constraints that impose restrictions on the values those variables can take. Our task is to assign a value to each variable so that we fulfil all the constraints.

So, to formally define a CSP, we specify:

- the set of variables
- the set of their (finite or infinite) domains
- and the set of constraints , where each can involve any number of variables:

CODE:

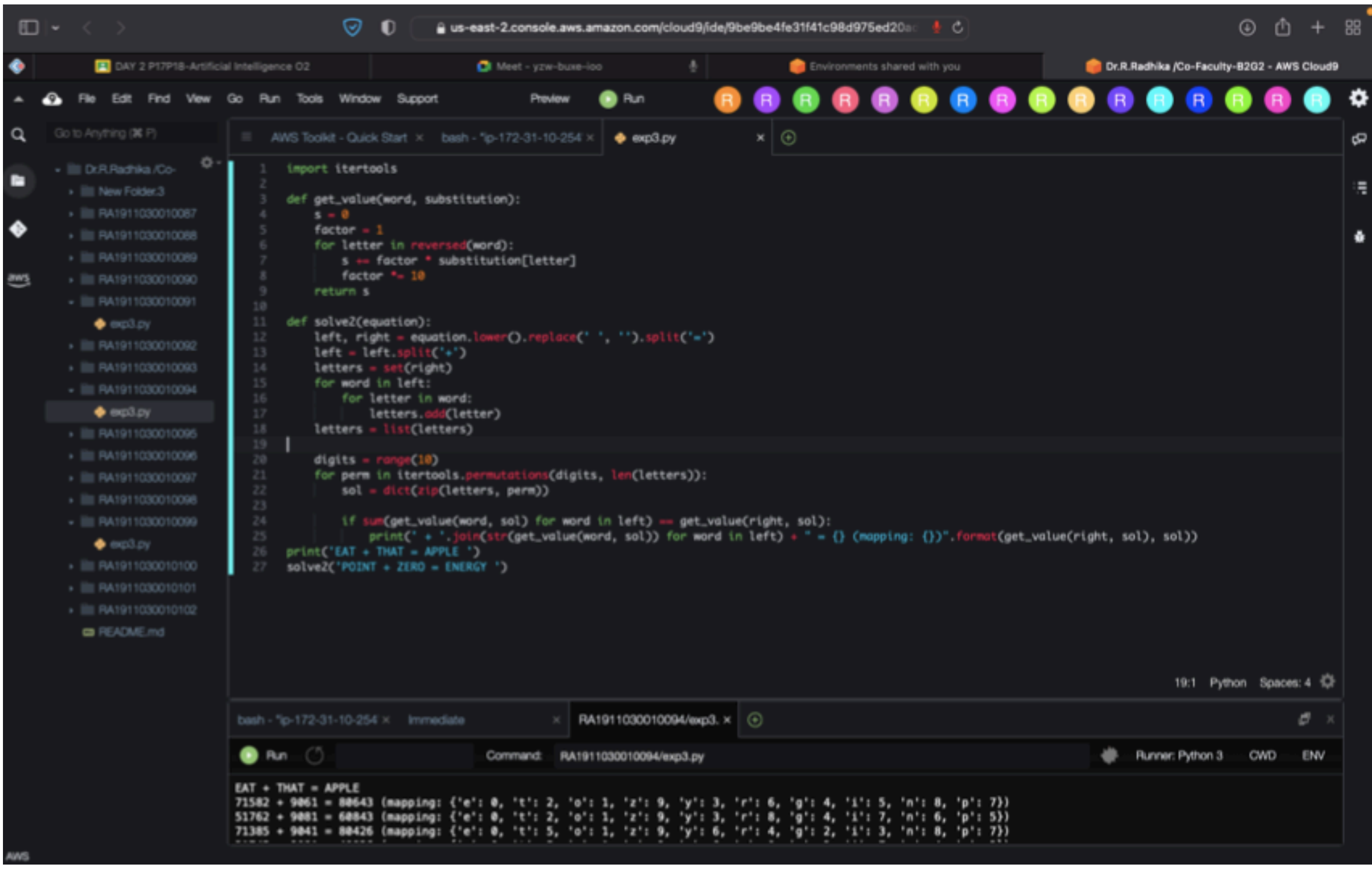
```
import itertools

def get_value(word,
substitution):     s = 0     factor =
1     for letter in
reversed(word):
        s += factor * substitution[letter]
factor *= 10
    return s

def solve2(equation):
    left, right = equation.lower().replace(' ',
'').split('=')     left = left.split('+')     letters =
set(right)     for word in left:         for letter in
word:             letters.add(letter)
    letters = list(letters)

    digits = range(10)     for perm in
itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))
```

```
        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
print(' + '.join(str(get_value(word, sol)) for word in left) + " = {}
(mapping: {})".format(get_value(right, sol), sol))



print('SEND + MORE = MONEY ')
solve2('SEND + MORE = MONEY ')
```

OUTPUT:

```
SEND + MORE = MONEY
2817 + 368 = 3185 (mapping: {'s': 2, 'n': 1, 'r': 6, 'e': 8, 'o': 3, 'd': 7, 'y': 5, 'm': 0})
2819 + 368 = 3187 (mapping: {'s': 2, 'n': 1, 'r': 6, 'e': 8, 'o': 3, 'd': 9, 'y': 7, 'm': 0})
3719 + 457 = 4176 (mapping: {'s': 3, 'n': 1, 'r': 5, 'e': 7, 'o': 4, 'd': 9, 'y': 6, 'm': 0})
3712 + 467 = 4179 (mapping: {'s': 3, 'n': 1, 'r': 6, 'e': 7, 'o': 4, 'd': 2, 'y': 9, 'm': 0})
3829 + 458 = 4287 (mapping: {'s': 3, 'n': 2, 'r': 5, 'e': 8, 'o': 4, 'd': 9, 'y': 7, 'm': 0})
3821 + 468 = 4289 (mapping: {'s': 3, 'n': 2, 'r': 6, 'e': 8, 'o': 4, 'd': 1, 'y': 9, 'm': 0})
5731 + 647 = 6378 (mapping: {'s': 5, 'n': 3, 'r': 4, 'e': 7, 'o': 6, 'd': 1, 'y': 8, 'm': 0})
5732 + 647 = 6379 (mapping: {'s': 5, 'n': 3, 'r': 4, 'e': 7, 'o': 6, 'd': 2, 'y': 9, 'm': 0})
5849 + 638 = 6487 (mapping: {'s': 5, 'n': 4, 'r': 3, 'e': 8, 'o': 6, 'd': 9, 'y': 7, 'm': 0})
6419 + 724 = 7143 (mapping: {'s': 6, 'n': 1, 'r': 2, 'e': 4, 'o': 7, 'd': 9, 'y': 3, 'm': 0})
6415 + 734 = 7149 (mapping: {'s': 6, 'n': 1, 'r': 3, 'e': 4, 'o': 7, 'd': 5, 'y': 9, 'm': 0})
6524 + 735 = 7259 (mapping: {'s': 6, 'n': 2, 'r': 3, 'e': 5, 'o': 7, 'd': 4, 'y': 9, 'm': 0})
6853 + 728 = 7581 (mapping: {'s': 6, 'n': 5, 'r': 2, 'e': 8, 'o': 7, 'd': 3, 'y': 1, 'm': 0})
6851 + 738 = 7589 (mapping: {'s': 6, 'n': 5, 'r': 3, 'e': 8, 'o': 7, 'd': 1, 'y': 9, 'm': 0})
7316 + 823 = 8139 (mapping: {'s': 7, 'n': 1, 'r': 2, 'e': 3, 'o': 8, 'd': 6, 'y': 9, 'm': 0})
7429 + 814 = 8243 (mapping: {'s': 7, 'n': 2, 'r': 1, 'e': 4, 'o': 8, 'd': 9, 'y': 3, 'm': 0})
7539 + 815 = 8354 (mapping: {'s': 7, 'n': 3, 'r': 1, 'e': 5, 'o': 8, 'd': 9, 'y': 4, 'm': 0})
7531 + 825 = 8356 (mapping: {'s': 7, 'n': 3, 'r': 2, 'e': 5, 'o': 8, 'd': 1, 'y': 6, 'm': 0})
7534 + 825 = 8359 (mapping: {'s': 7, 'n': 3, 'r': 2, 'e': 5, 'o': 8, 'd': 4, 'y': 9, 'm': 0})
7649 + 816 = 8465 (mapping: {'s': 7, 'n': 4, 'r': 1, 'e': 6, 'o': 8, 'd': 9, 'y': 5, 'm': 0})
7643 + 826 = 8469 (mapping: {'s': 7, 'n': 4, 'r': 2, 'e': 6, 'o': 8, 'd': 3, 'y': 9, 'm': 0})
8324 + 913 = 9237 (mapping: {'s': 8, 'n': 2, 'r': 1, 'e': 3, 'o': 9, 'd': 4, 'y': 7, 'm': 0})
8432 + 914 = 9346 (mapping: {'s': 8, 'n': 3, 'r': 1, 'e': 4, 'o': 9, 'd': 2, 'y': 6, 'm': 0})
8542 + 915 = 9457 (mapping: {'s': 8, 'n': 4, 'r': 1, 'e': 5, 'o': 9, 'd': 2, 'y': 7, 'm': 0})
9567 + 1085 = 10652 (mapping: {'s': 9, 'n': 6, 'r': 8, 'e': 5, 'o': 0, 'd': 7, 'y': 2, 'm': 1})
```

SCREENSHOTS:



Results:

Constraint Satisfaction problem has been successfully implemented.

LAB 4- Implementation and Analysis of BFS and DFS for an application

AIM – Implementation and analysis of BFS and DFS for an application.

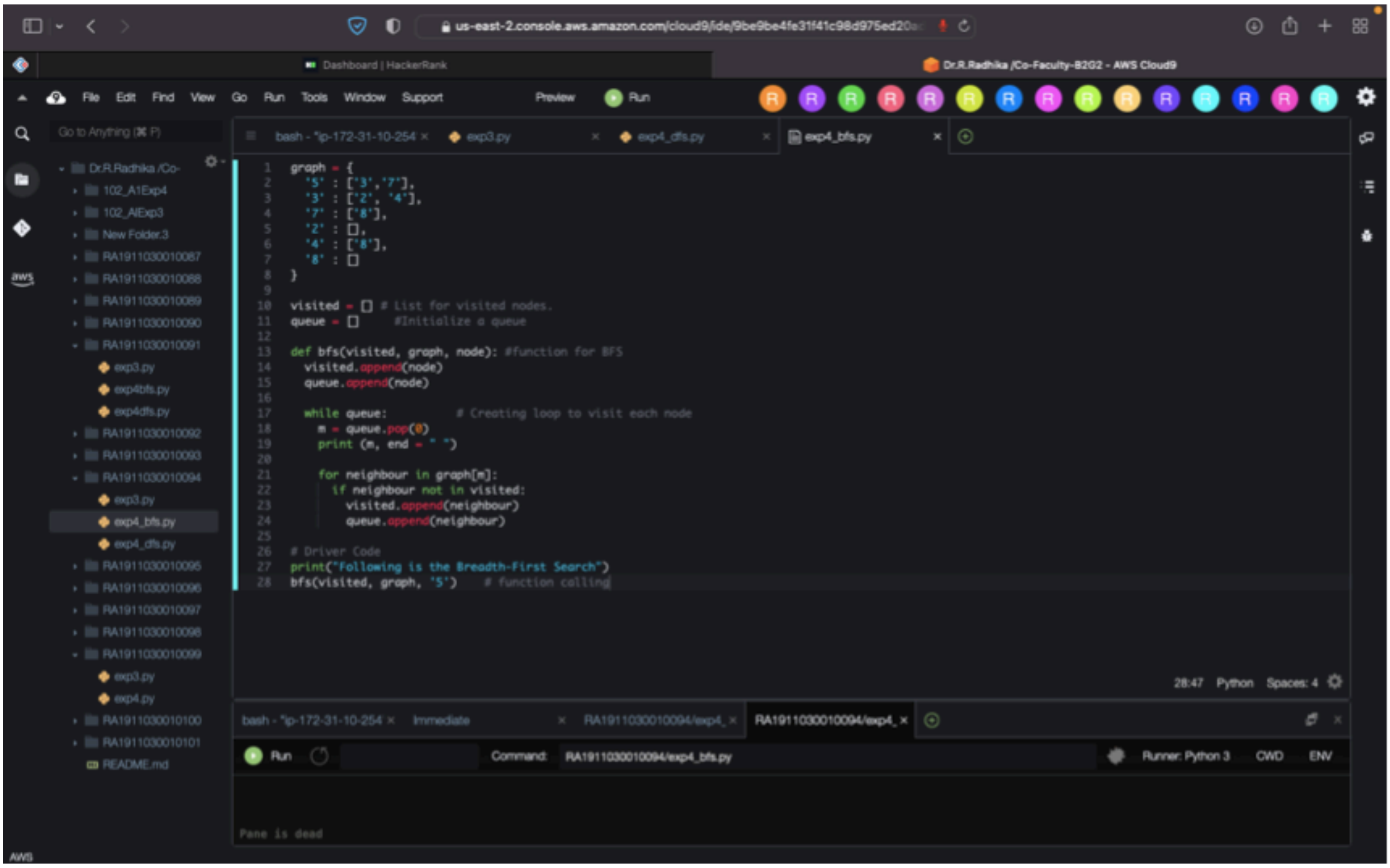
Problem Description of BFS:

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

BFS Breadth First Search Code:

```
graph =
{ '5' :
['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : [] } visited = []
queue = []      def
bfs(visited, graph,
node):
visited.append(node)
queue.append(node) while
queue:            m = queue.pop(0)
print (m, end = " ")      for
neighbour in graph[m]:
if neighbour not in visited:
visited.append(neighbour)
queue.append(neighbour)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

Screenshot:



Problem Description of DFS:

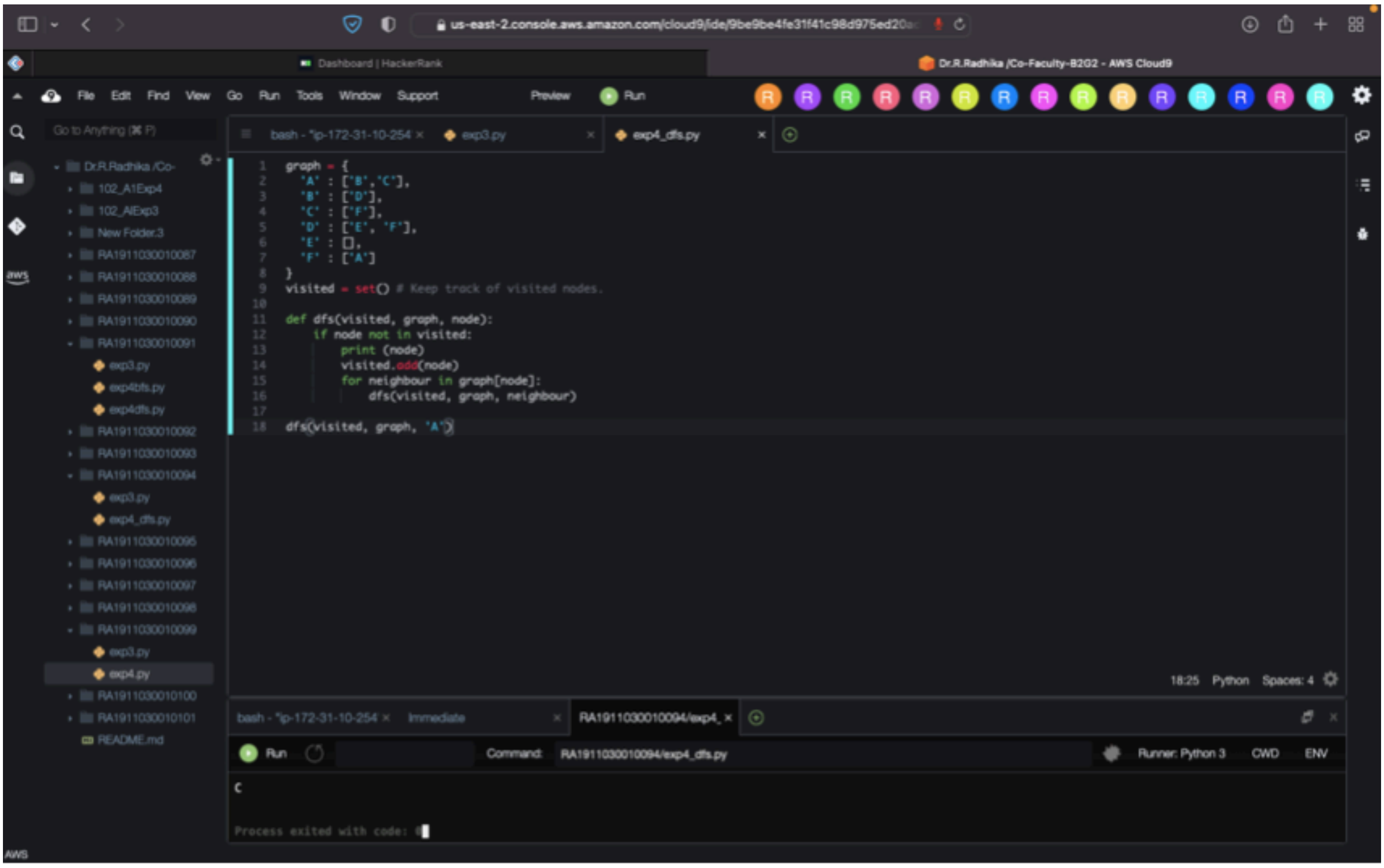
Depth First Search (DFS) is often used for traversing and searching a tree or graph data structure. The idea is to start at the root (in the case of a tree) or some arbitrary node (in the case of a graph) and explores each branch as far as possible before backtracking.

DFS – Depth First Search Code:

```
graph = {
 'A' : ['B','C'],
 'B' : ['D'],
 'C' : ['F'],
 'D' : ['E', 'F'],
 'E' : [],
 'F' : ['A']
}
visited = set() # Keep track of visited nodes.

def dfs(visited, graph, node):
if node not in visited:
print (node)
visited.add(node)        for
```

```
neighbour in graph[node]:
dfs(visited, graph, neighbour)

dfs(visited, graph, 'A')
```

Screenshot:



RESULT: Hence, BFS and DFS was implemented and analysed for an application.

LAB 5 – Developing Best First Search and A* Algorithm for real world problem

AIM: Implementation of Best First Search for an application

Problem Description for BFS:

Best first search is a traversal technique that decides which node is to be visited next by checking which node is the most promising one and then check it For this it uses an evaluation function to decide the traversal. This best first search technique of tree traversal comes under the category of heuristic search or informed search technique**.**

CODE:

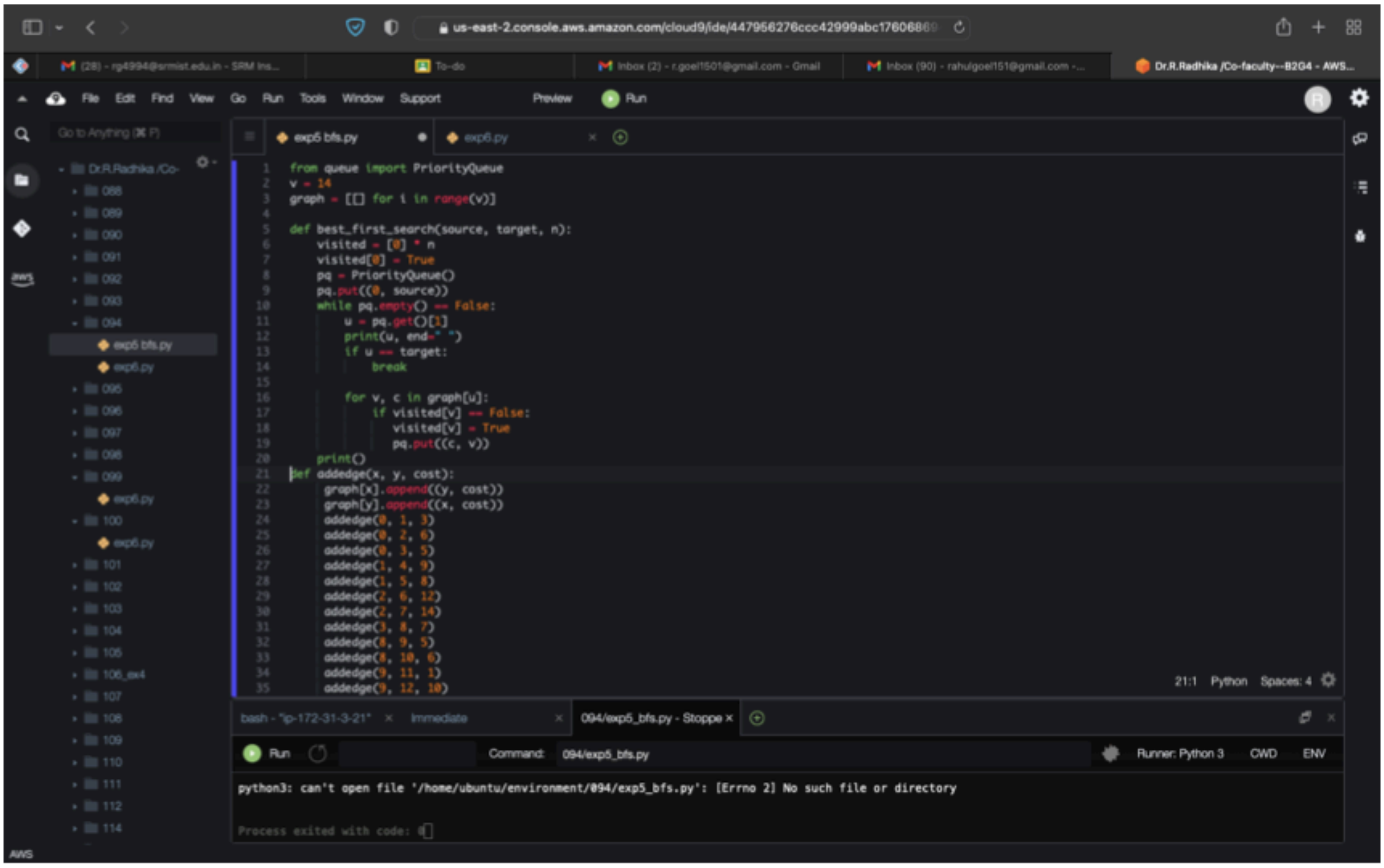
```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]

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))

addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
```

```
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)

source = 0
target = 9
best_first_search(source, target, v)
```

OUTPUT

0 1 3 2 8 9

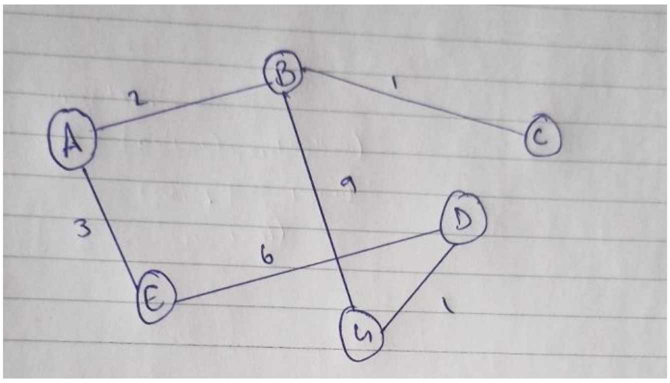
SCREENSHOT:



A* algorithm:

- open set is list of nodes which have been visited but neighbors haven’t all been inspected whereas closed set is list of nodes which have been visited but neighbors have been inspected.
- g contains current distances from start node to all other nodes.
- parents contains adjacency map of all nodes
- we find a node with the lowest value of f() - evaluation function
- if the current node is the stop_node then we begin reconstructing the path from it to the start_node
- if the current node isn’t in both open set and closed set add it to open set and note n as its parent
- otherwise, check if it's quicker to first visit n, then m and if it is, update parent data and g data and if the node was in the closed set, move it to open set
- remove n from the open set, and add it to closed set because all of his neighbors were inspected

GRAPH:



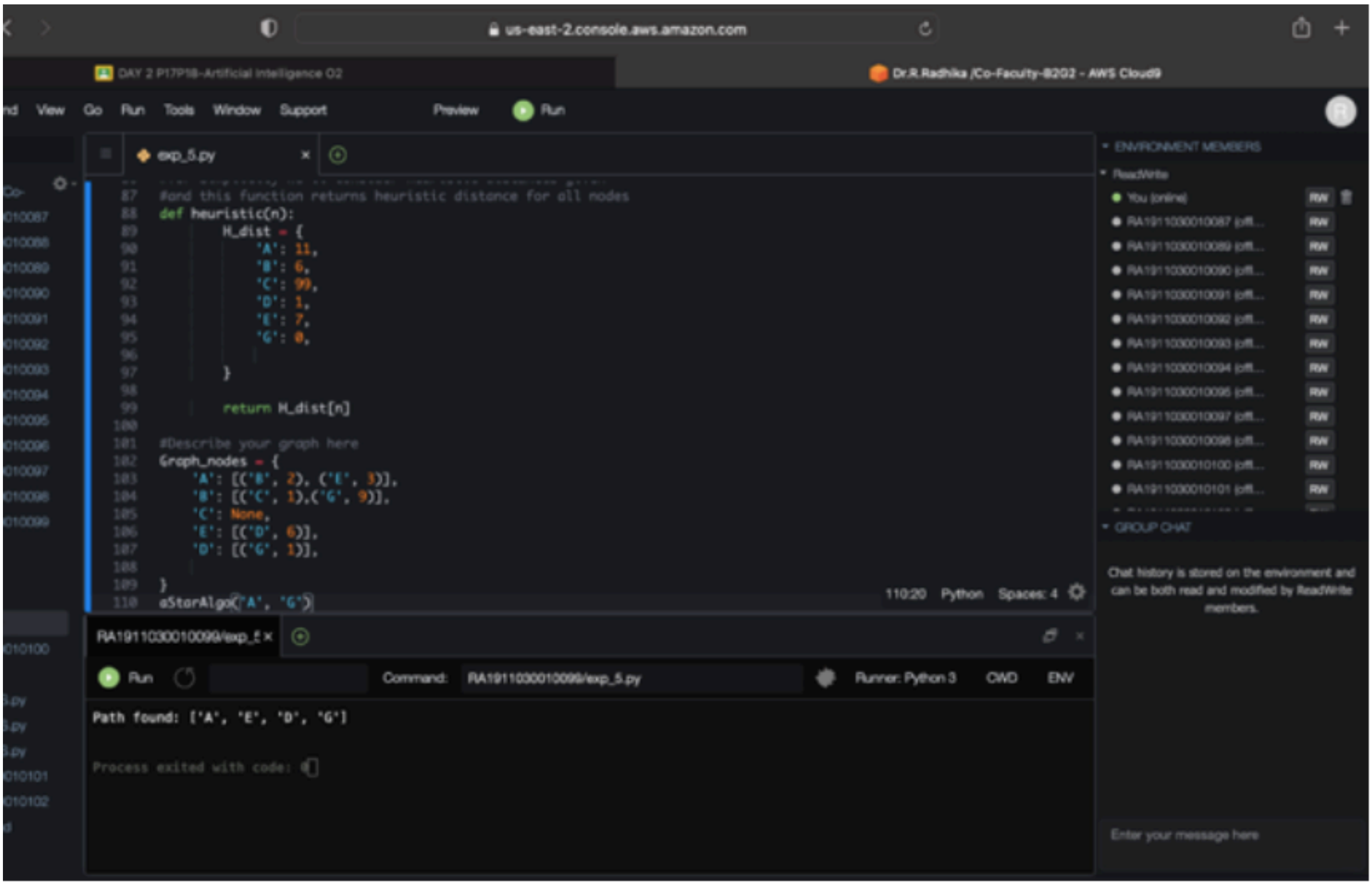
CODE:

```
def aStarAlgo(start_node, stop_node):
open_set = set(start_node)
closed_set = set()
g = {}
parents = {}
g[start_node] = 0
parents[start_node] = start_node
while len(open_set) > 0:
n = None
for v in open_set:
if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
```

```
n = v
if n == stop_node or Graph_nodes[n] == None:
pass
else:
for (m, weight) in get_neighbors(n):
if m not in open_set and m not in closed_set:
open_set.add(m)
parents[m] = n
g[m] = g[n] + weight
else:
if g[m] > g[n] + weight:
g[m] = g[n] + weight
parents[m] = n
if m in closed_set:
closed_set.remove(m)
open_set.add(m)
if n == None:
print('Path does not exist!')
return None
if n == stop_node:
path = []
while parents[n] != n:
path.append(n)
n = parents[n]
path.append(start_node)
path.reverse()
print('Path found: {}'.format(path))
return path
open_set.remove(n)
closed_set.add(n)
print('Path does not exist!')
return None
def get_neighbors(v):
if v in Graph_nodes:
return Graph_nodes[v]
else:
return None
def heuristic(n):
H_dist = {
'A': 11,
'B': 6,
'C': 99,
'D': 1,
```

```
'E': 7,
'G': 0,
}
return H_dist[n]
Graph_nodes =
{ 'A': [('B', 2), ('E',
3)],
'B': [('C', 1),('G', 9)],
'C': None,
'E': [('D', 6)],
'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

OUTPUT:



Result: Therefore, BFS and A* algorithm has been implemented successfully

LAB 6 - Minimax algorithm for an application

AIM: Implementation of minimax algorithm for Tic Tac Toe.

Problem Description:

- If the game is over, return the score from X's perspective.
- Otherwise get a list of new game states for every possible move
- Create a scores list
- For each of these states add the minimax result of that state to the scores list
- If it's X's turn, return the maximum score from the scores list
- If it's O's turn, return the minimum score from the scores list

CODE:

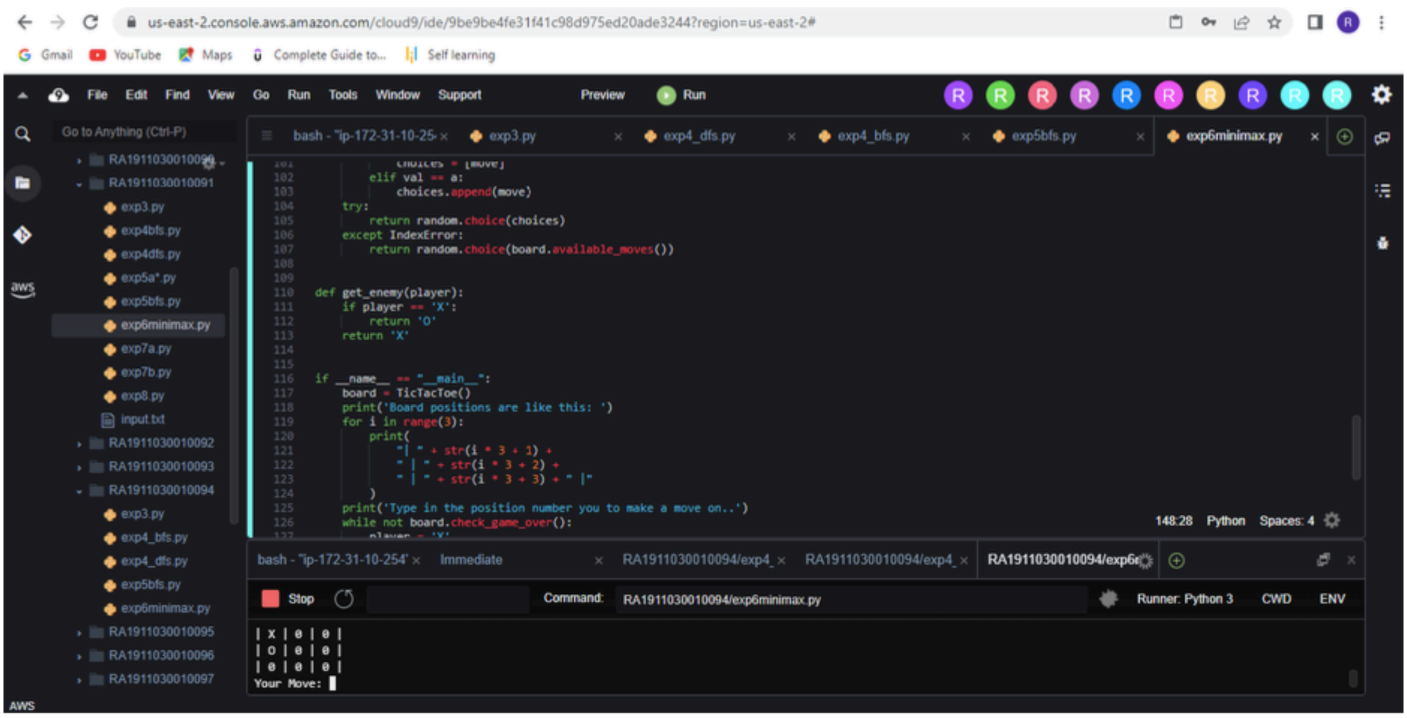
```
theBoard = {'1': ' ' , '2': ' ' , '3': ' ' ,
            '4': ' ' , '5': ' ' , '6': ' ' ,
'7': ' ' , '8': ' ' , '9': ' ' }
board_keys = []
for key in
theBoard:
    board_keys.append(key)
def printBoard(board):
    print(board['7'] + '|' + board['8'] + '|' + board['9'])
print('-----')
    print(board['4'] + '|' + board['5'] + '|' + board['6'])
print('-----')
    print(board['1'] + '|' + board['2'] + '|' +
board['3']) def game():     turn = 'X'     count = 0
for i in range(10):
        printBoard(theBoard)
        print("It's your turn " + turn + ". Move to which place?")
move = input()

        if theBoard[move] == ' ':
theBoard[move] = turn
            count += 1
else:
            print("That place is already filled.\n Move to which place?")
continue

        if count >= 5:            if theBoard['7'] ==
theBoard['8'] == theBoard['9'] != ' ':
                printBoard(theBoard)
```

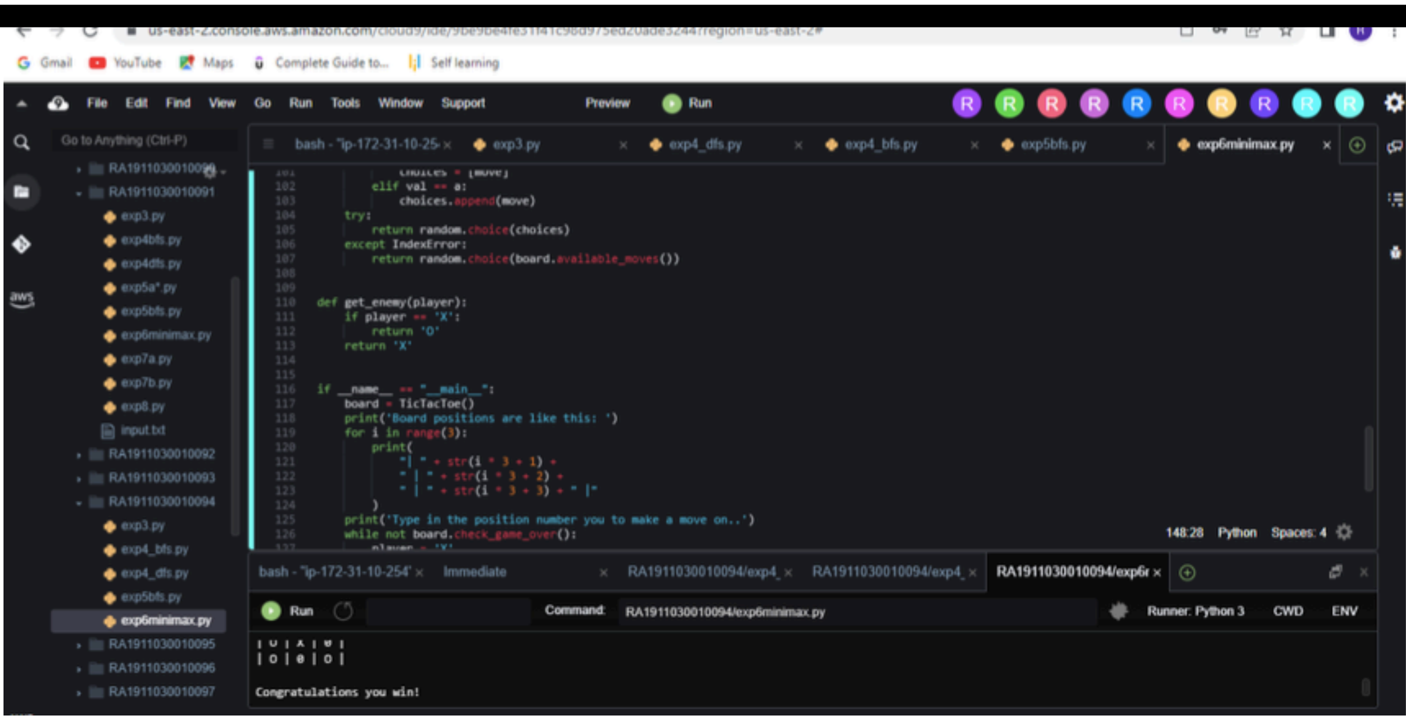
```
            print("\nGame Over.\n")
print(" ** " +turn + " won. **")                        break
elif theBoard['4'] == theBoard['5'] == theBoard['6'] != ' ':
            printBoard(theBoard)
            print("\nGame Over.\n")
print(" ** " +turn + " won. **")
break           elif theBoard['1'] ==
theBoard['2'] == theBoard['3'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
print(" ** " +turn + " won. **")
            break           elif theBoard['1'] == theBoard['4']
== theBoard['7'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
print(" ** " +turn + " won. **")
            break           elif theBoard['2'] == theBoard['5']
== theBoard['8'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
print(" ** " +turn + " won. **")
            break           elif theBoard['3'] == theBoard['6']
== theBoard['9'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
            print(" ** " +turn + " won. **")
            break           elif theBoard['7'] == theBoard['5']
== theBoard['3'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
print(" ** " +turn + " won. **")
            break           elif theBoard['1'] == theBoard['5']
== theBoard['9'] != ' ':
            printBoard(theBoard)
print("\nGame Over.\n")
            print(" ** " +turn + " won. **")
            break

      if count == 9:
         print("\nGame Over.\n")
print("It's a Tie!!")
```

if turn =='X':
turn = 'O' else:
turn = 'X'

restart = input("Do want to play Again?(y/
n)") if restart == "y" or restart ==
"Y":
for key in
board_keys:
theBoard[key] = " "
game() if __name__==
"__main__": game()

SCREENSHOTS:





RESULT: Hence, Minimax algorithm was implemented for Tic Tac Toe problem.

Exp7 -Unification and Resolution.

AIM: To implement unification and resolution algorithm.

PROCEDURE for Unification:

1) Initialize the substitution set to be empty.
2) Recursively unify atomic sentences:

- Check for Identical expression match.
- If one expression is a variable vi, and the other is a term ti which does not contain variable vi, then:
- Substitute ti / vi in the existing substitutions
- Add ti /vi 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:

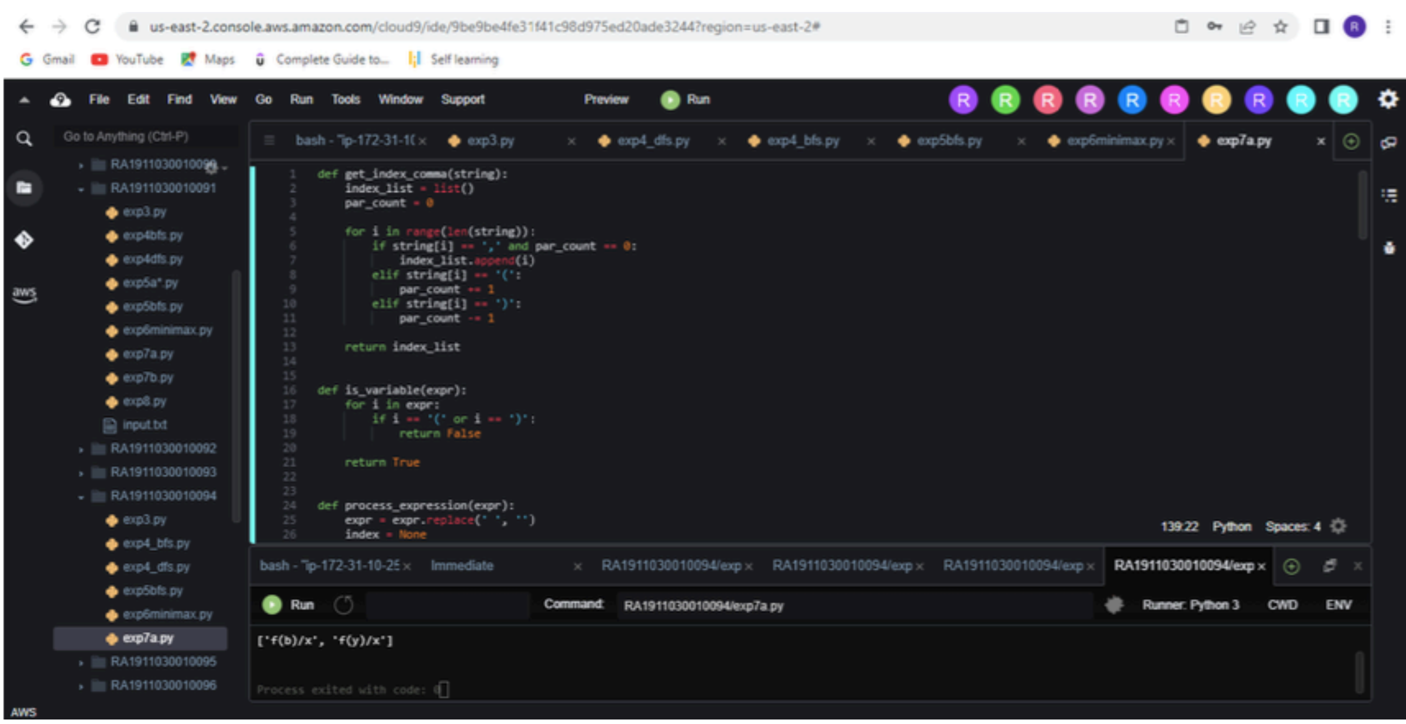
```
def get_index_comma(string):
    index_list = list()
    par_count = 0

    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)
```

SCREENSHOT:



PROCEDURE for Resolution:

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) 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(query
                  Predicate),
 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
```

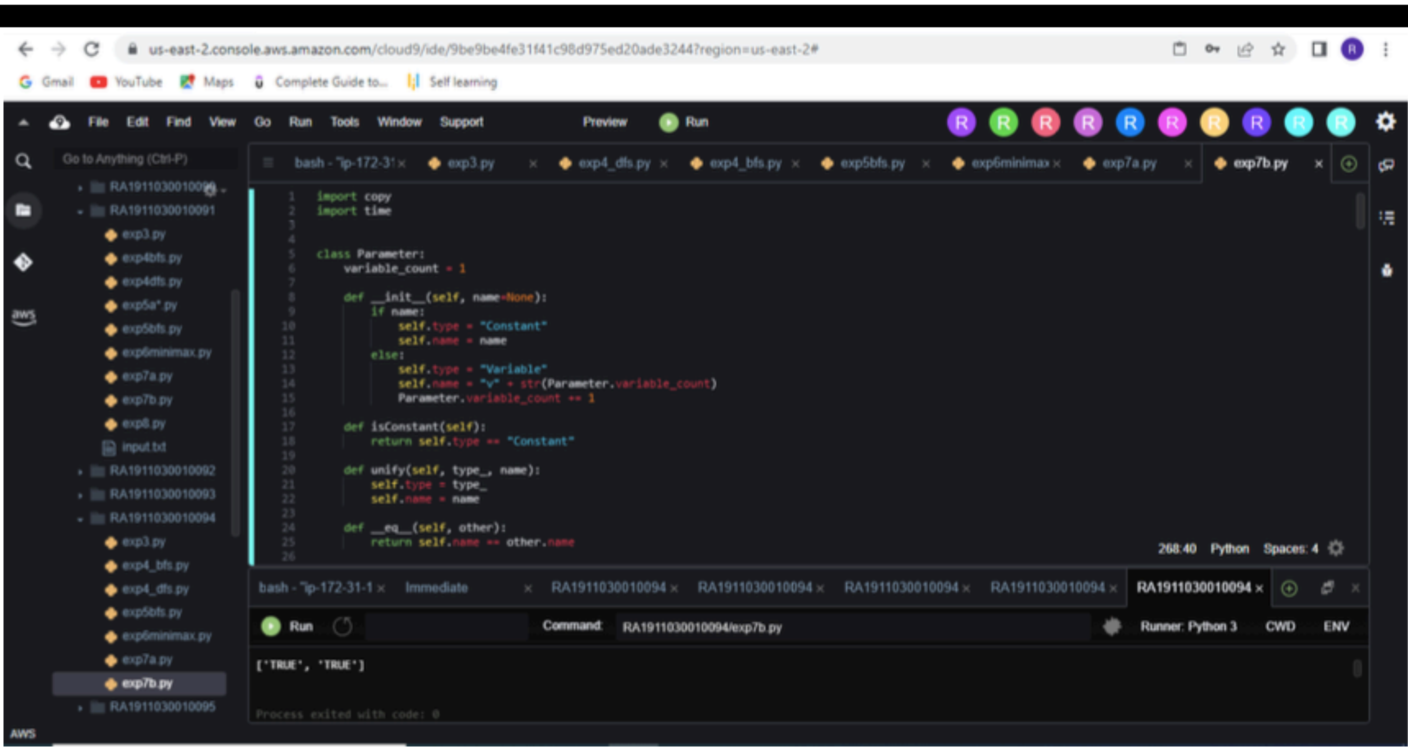
```
        return True


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
```

```
    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('/home/ubuntu/environment/RA1911030010091/input.txt')                    knowledgeBase =
KB(inputSentences_)     knowledgeBase.prepareKB()
    results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

INPUT.txt code:

```
2
Friends(Alice,Bob,Charlie,Diana)
Friends(Diana,Charlie,Bob,Alice)
2
Friends(a,b,c,d)
NotFriends(a,b,c,d)
```

Screenshot:



RESULT: Hence, Unification and Resolution were implemented.

LAB 8 - Implementation of knowledge representation schemes - use cases

AIM: To implement knowledge representation schemes.

ALGORTIHM:

- Create a knowledge base with identification rules.
- Create a question-and-answer knowledge.
- Ask question to user
- Use the inputs to the database
- If an animal is found print the guess.

CODE:

```
/* animal.pl animal
identification game.
   start with ?- go.     */ go :-
hypothesize(Animal),      write('I
guess that the animal is: '),
write(Animal),
     nl,
undo.
/* hypotheses to be tested */
hypothesize(cheetah) :- cheetah, !.
hypothesize(tiger)     :- tiger, !.
hypothesize(giraffe) :- giraffe, !.
hypothesize(zebra)     :- zebra, !.
hypothesize(ostrich) :- ostrich, !.
hypothesize(penguin) :- penguin, !.
hypothesize(albatross) :- albatross, !.
hypothesize(unknown).            /* no diagnosis */

/* animal identification rules */
cheetah :- mammal,
          carnivore,
```

```
        verify(has_tawny_color),
verify(has_dark_spots).
tiger :- mammal,
carnivore,
      verify(has_tawny_color),
verify(has_black_stripes). giraffe
:- ungulate,
verify(has_long_neck),
verify(has_long_legs).
zebra :- ungulate,
verify(has_black_stripes).

ostrich :- bird,
verify(does_not_fly),
verify(has_long_neck). penguin :-
bird,         verify(does_not_fly),
verify(swims),
verify(is_black_and_white).
albatross :- bird,
verify(appears_in_story_Ancient_Mariner),
verify(flys_well).
/* classification rules */
mammal    :- verify(has_hair),
 !. mammal   :-
verify(gives_milk). bird     :-
verify(has_feathers), !.
bird     :- verify(flys),
verify(lays_eggs). carnivore :-
verify(eats_meat), !. carnivore :-
verify(has_pointed_teeth),
verify(has_forward_eyes).
verify(has_claws),
Ungulate

:- mammal,
verify(has_hooves), !. ungulate :-
mammal,
```

```
verify(chews_cud).

/* how to ask questions */
ask(Question) :-
   write('Does the animal have the following
attribute:   '),   write(Question),   write('?   '),
read(Response),
   nl,
   ( (Response == yes ; Response == y)
     ->
      assert(yes(Question)) ;
assert(no(Question)), fail).

:- dynamic yes/1,no/1.
 /* How to verify something */
verify(S) :-
  (yes(S)
->
true ;
   (no(S)
->     fail ;
ask(S))).
/* undo all yes/no assertions
*/ undo :- retract(yes(_)),fail.
undo :- retract(no(_)),fail.
undo.
```

OUTPUT:

RESULT: Hence, knowledge representation schemes was implemented.

LAB 9 - Implementation of uncertain methods for an application

AIM: To implement uncertain methods for Monty Hall problem.

Problem Statement:

The Monty Hall problem is a counter-intuitive statistics puzzle:

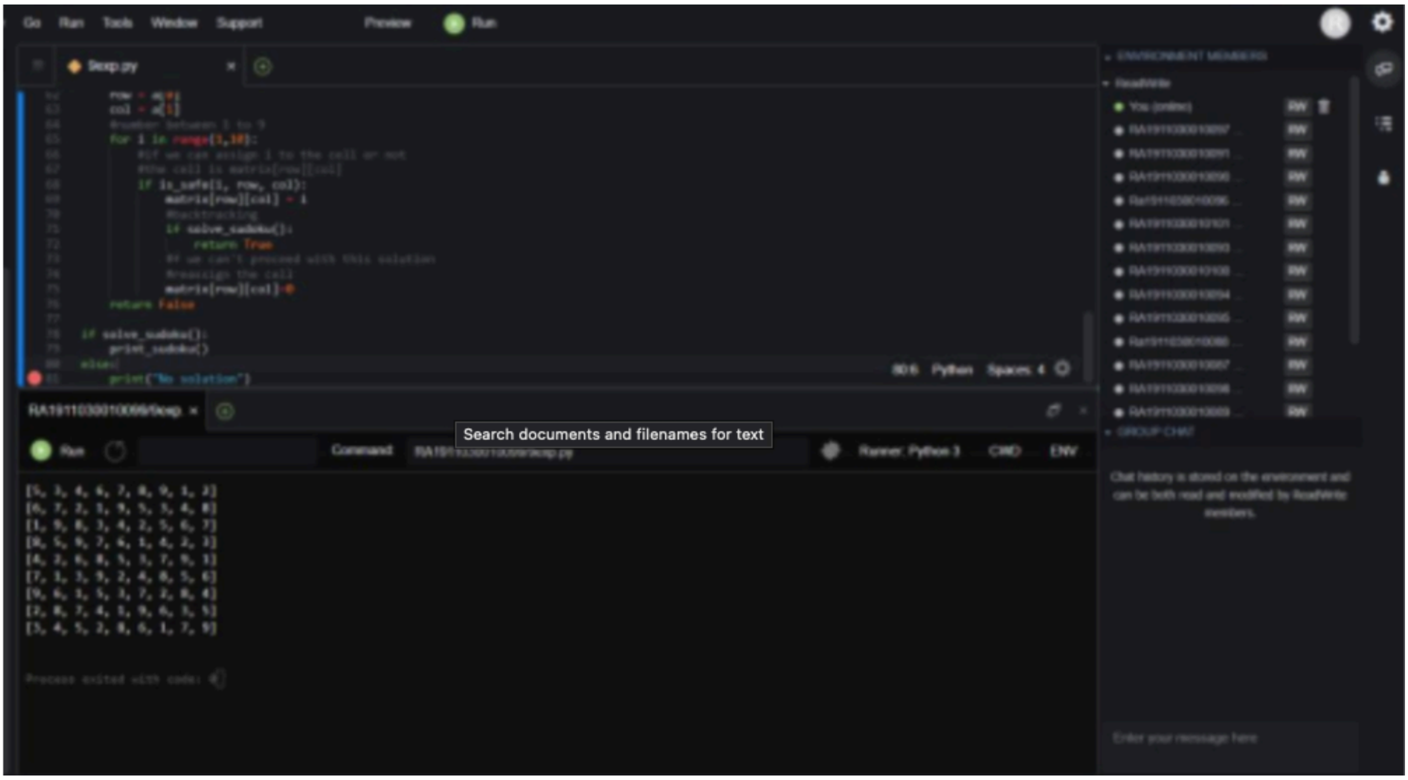
- There are 3 doors, behind which are two goats and a car.
- You pick a door (call it door A). You're hoping for the car of course.
- Monty Hall, the game show host, examines the other doors (B & C) and opens one with a goat. (If both doors have goats, he picks randomly.)

CODE:

```
import matplotlib.pyplot as plt
import seaborn; seaborn.set_style('whitegrid')
import numpy from
pomegranate import *
numpy.random.seed(0)
numpy.set_printoptions(suppress=True)
guest = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
prize = DiscreteDistribution({'A': 1./3, 'B': 1./3, 'C': 1./3})
monty = ConditionalProbabilityTable(
        [[ 'A', 'A', 'A', 0.0 ],
         [ 'A', 'A', 'B', 0.5
],        [ 'A', 'A', 'C',
0.5 ],        [ 'A', 'B',
'A', 0.0 ],
         [ 'A', 'B', 'B', 0.0
],        [ 'A', 'B', 'C',
1.0 ],
         [ 'A', 'C', 'A', 0.0 ],
         [ 'A', 'C', 'B', 1.0
],        [ 'A', 'C', 'C',
0.0 ],
         [ 'B', 'A', 'A', 0.0 ],
         [ 'B', 'A', 'B', 0.0
],        [ 'B', 'A', 'C',
1.0 ],        [ 'B', 'B',
'A', 0.5 ],
```

```
        [ 'B', 'B', 'B', 0.0
],         [ 'B', 'B', 'C',
0.5 ],
        [ 'B', 'C', 'A', 1.0 ],
        [ 'B', 'C', 'B', 0.0
],         [ 'B', 'C', 'C',
0.0 ],
        [ 'C', 'A', 'A', 0.0 ],
        [ 'C', 'A', 'B', 1.0
],         [ 'C', 'A', 'C',
0.0 ],         [ 'C', 'B',
'A', 1.0 ],
        [ 'C', 'B', 'B', 0.0
],         [ 'C', 'B', 'C',
0.0 ],
        [ 'C', 'C', 'A', 0.5 ],
        [ 'C', 'C', 'B', 0.5 ],
        [ 'C', 'C', 'C', 0.0 ]], [guest,
prize]) s1 = State(guest,
name="guest") s2 = State(prize,
name="prize")
s3 = State(monty, name="monty")
# Create the Bayesian network object with a useful name
model = BayesianNetwork("Monty Hall Problem")
model.add_states(s1, s2, s3)
model.add_edge(s1, s3)
model.add_edge(s2, s3)
model.bake()
model.probability([['A', 'B', 'C']])
model.probability([['A', 'B', 'C']])
print(model.predict_proba({}))
print(model.predict_proba([[None, None, None]]))
print(model.predict_proba([['A', None, None]]))
print(model.predict_proba([{'guest': 'A', 'monty': 'B'}]))
```

SCREENSHOTS



RESULT: Hence, the uncertain method for an application was implemented.

LAB 10 -Implementation of Block world Problem

AIM: To implement block world problem.

Problem Statement:
The blocks world is a planning domain in artificial intelligence. The algorithm is similar to a set of wooden blocks of various shapes and colors sitting on a table. The goal is to build one or more vertical stacks of blocks. Only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. Moreover, some kinds of blocks cannot have other blocks stacked on top of them

CODE:

```
class PREDICATE:
def_str_(self):
   pass  def
__repr_(self):
   pass  def
__eq_(self, other) :
   pass  def
__hash_(self):
   pass def get_action(self,
world_state):
   pass


class Operation:
def_str_(self):
   pass   def
__repr_(self):
   pass  def
__eq_(self, other) :
pass def
precondition(self):
     pass  def
 delete(self):
     pass  def
add(self):
   pass
```

```
class ON(PREDICATE):

  def_init_(self, X, Y):
    self.X = X
    self.Y = Y

  def_str_(self):
    return "ON({X},{Y})".format(X=self.X,Y=self.Y)

  def_repr_(self):
    return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def_hash_(self):
return hash(str(self))

  def get_action(self, world_state):
return StackOp(self.X,self.Y)

class ONTABLE(PREDICATE):

  def_init_(self, X):
    self.X = X

  def_str_(self):
    return "ONTABLE({X})".format(X=self.X)

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def_hash_(self):
return hash(str(self))

  def get_action(self, world_state):
    return PutdownOp(self.X)
```

```
class CLEAR(PREDICATE):

  def_init_(self, X):
    self.X = X

  def_str_(self):
    return "CLEAR({X})".format(X=self.X)
self.X = X

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def_hash_(self):
return hash(str(self))
def get_action(self,
world_state):     for
predicate in
world_state:       #If
Block is on another
block, unstack       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 block is on table, pick
up     if ONTABLE(X) in
world_state:
      return PickupOp(X)
    #If block is on another block,
unstack     else:       for predicate in
world_state:
        if isinstance(predicate,ON) and predicate.X==X:
          return UnstackOp(X,predicate.Y)

class ARMEMPTY(PREDICATE):

  def_init_(self):
    pass

  def_str_(self):
return "ARMEMPTY"

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def_hash_(self):
return hash(str(self))

  def get_action(self, world_state=[]):
    for predicate in world_state:
if isinstance(predicate,HOLDING):
return PutdownOp(predicate.X)
return None

class StackOp(Operation):

  def_init_(self, X, Y):
```

```
    self.X = X
    self.Y = Y

  def_str_(self):
    return "STACK({X},{Y})".format(X=self.X,Y=self.Y)

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def precondition(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def delete(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

  def add(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]

class UnstackOp(Operation):

  def_init_(self, X, Y):
    self.X = X
    self.Y = Y

  def_str_(self):
    return "UNSTACK({X},{Y})".format(X=self.X,Y=self.Y)

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def precondition(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) , CLEAR(self.X) ]

  def delete(self):
    return [ ARMEMPTY() , ON(self.X,self.Y) ]
```

```
  def add(self):
    return [ CLEAR(self.Y) , HOLDING(self.X) ]

class PickupOp(Operation):

  def_init_(self, X):
    self.X = X

  def_str_(self):
    return "PICKUP({X})".format(X=self.X)

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :
    return self.__dict___== other.__dict___and self.__class___==
other.__class___

  def precondition(self):
    return [ CLEAR(self.X) , ONTABLE(self.X) , ARMEMPTY() ]

  def delete(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

  def add(self):
    return [ HOLDING(self.X) ]

class PutdownOp(Operation):

  def_init_(self, X):
    self.X = X

  def_str_(self):
    return "PUTDOWN({X})".format(X=self.X)

  def_repr_(self):
return self._str_()

  def_eq_(self, other) :

    return   self.__dict___  ==   other.__dict___  and   self.__class___  ==   other.__class___      def

precondition(self):     return [ HOLDING(self.X) ]
```

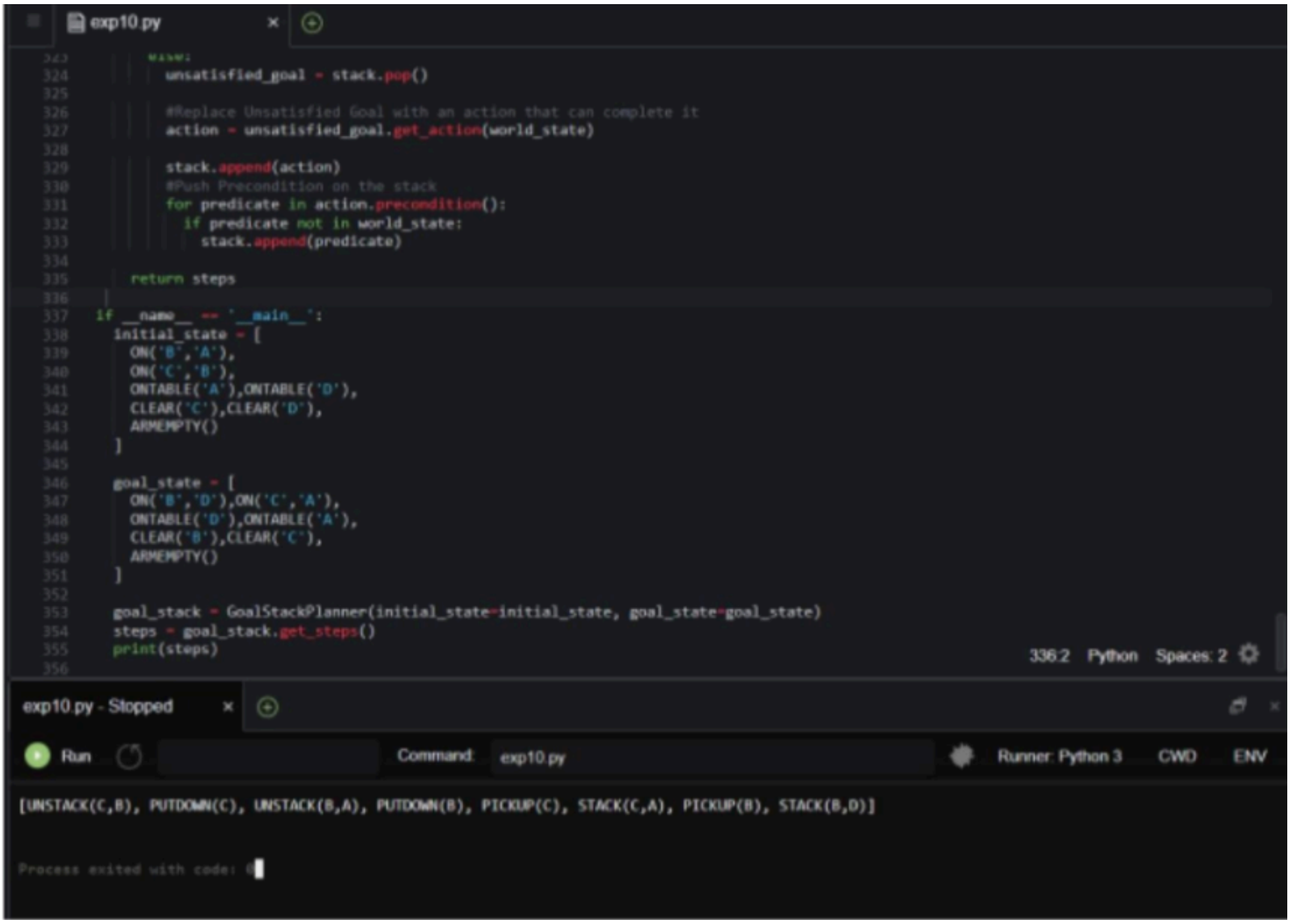
```
  def delete(self):
    return [ HOLDING(self.X) ]

  def add(self):
    return [ ARMEMPTY() , ONTABLE(self.X) ]

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

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

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

class GoalStackPlanner:

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

  def get_steps(self):

    #Store Steps
    steps = []

    #Program Stack
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)

     #If Stack Top is an action
elif isOperation(stack_top):

       #Peek the operation
       operation = stack[-1]

       all_preconditions_satisfied = True

       #Check if any precondition is unsatisfied and push it onto
program stack         for predicate in operation.delete():           if
predicate not in world_state:            all_preconditions_satisfied =
False            stack.append(predicate)

       #If all preconditions are satisfied, pop operation from stack and
execute it            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)
#If Stack Top is a single satisfied goal
elif stack_top in world_state:
       stack.pop()

     #If Stack Top is a single unsatisfied goal
else:
```

```
        unsatisfied_goal = stack.pop()
        #Replace Unsatisfied Goal with an action that can complete it
action = unsatisfied_goal.get_action(world_state)
stack.append(action)
        #Push Precondition on the stack
for predicate in action.precondition():
if predicate not in world_state:

stack.append(predicate)
return steps if __name___==
"_main_":
  initial_state = [
ON('B','A'),
    ON('C','B'),
    ONTABLE('A'),ONTABLE('D'),
    CLEAR('C'),CLEAR('D'),
    ARMEMPTY()
  ]
  goal_state =
    [ ON('B','D'),ON('C','A'),
    ONTABLE('D'),ONTABLE('A'),
    CLEAR('B'),CLEAR('C'),
    ARMEMPTY()
  ]
  goal_stack = GoalStackPlanner(initial_state=initial_state,
goal_state=goal_state) steps = goal_stack.get_steps()
print(steps)
```

OUTPUT:

[UNSTACK(C,B), PUTDOWN(C), UNSTACK(B,A), PUTDOWN(B), PICKUP(C), STACK(C,A), PICKUP(B), STACK(B,D)]



RESULT:

Hence, Block world problem was implemented.