

# **18CSC305J – ARTIFICIAL INTELLIGENCE LABOATORY RECORD**

**ACADEMIC YEAR 2021 – 2022, EVEN SEMESTER,  
DEPRTMENT OF NETWORKING AND COMMUNICATION,  
SCHOOL OF COMPUTING.**

**NAME:** 1. NAVEENKUMAR S

2. HARSHITHA N

3. KAMBHAM HARIPRIYA

4. SAMHITA BBHARADWAJ

5. ARAVIND G

**REGISTRATION NUMBER:** 1. RA1911029010014

2. RA1911029010006

3. RA1911029010016

4. RA1911029010015

5. RA1911029010028

**YEAR/ SEMESTER:** 3<sup>RD</sup> YEAR, 6<sup>TH</sup> SEMESTER

**SECTION:** P1 C



**DEPARTMENT OF NETWORKING AND  
COMMUNICATION, SCHOOL OF COMPUTING, SRM  
INSTITUTE OF SCIENCE AND TECHNOLOGY,  
KATTANKULATHUR – 603203, KANCHEEPURAM  
DISTRICT.**

## TABLE OF CONTENTS

Exp. No.	Exp. Name	Date	Signature
1	Implementation of toy problems	24.01.2022	
2	Developing agent programs for real world problems	11.02.2022	
3	Implementation of constraints satisfaction problems	14.02.2022	
4	Implementation and Analysis of DFS and BFS for same application	25.02.2022	
5	Developing best first search and A* Algorithm for real world problem	04.03.2022	
6	Min Max Algorithm	11.03.2022	
7	Implementation of unification and resolution for real world problems	26.03.2022	
8	Monty Hall Problem	06.04.2022	
9	Machine Learning	22.04.2022	
10	NLP	22.04.2022	

# TOY PROBLEM

## SLIDING TILE PUZZLE

EXP NO: 1

DATE:

### AIM:

Write a program to solve the sliding puzzle(3 by 3 grid)problem using Branch and bound.

### PROBLEM DESCRIPTION:

It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state.

#### INITIAL CONFIGURATION:

1	2	3
5	6	
7	8	4

#### FINAL CONFIGURATION:

1	2	3
5	8	6
	7	4

### RULES OF SOLVING PUZZLE:

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile.

The empty space can only move in four directions (Movement of empty space)

1. Up
2. Down
3. Right or
4. Left

The empty space cannot move diagonally and can take only one step at a time.

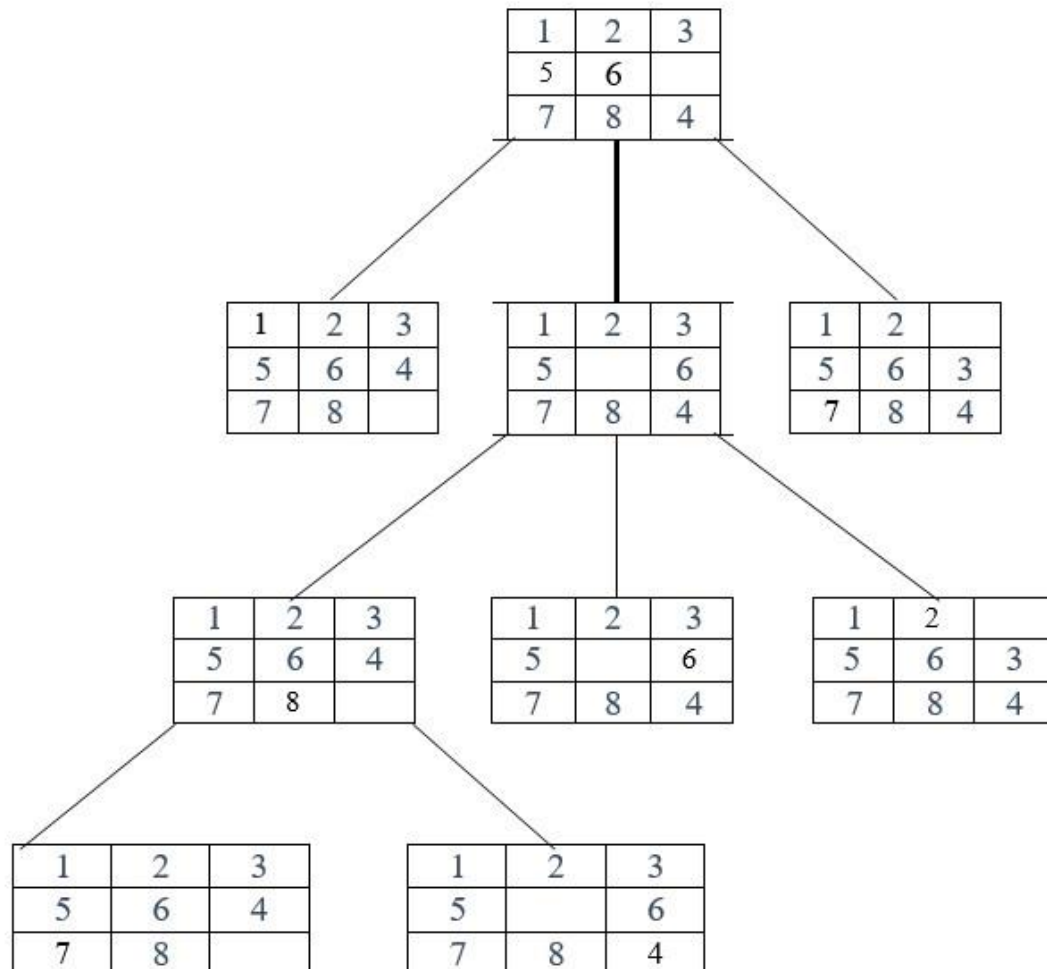
### IMPLEMENTATION:

Implemented using branch and bound.

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree.

An important advantage of branch-and-bound algorithms is that we can control the quality of the solution to be expected, even if it is not yet found. The cost of an optimal solution is only up to smaller than the cost of the best computed one.

#### STATE SPACE DIAGRAM:



#### PROGRAM CODE:

```
# Python3 program to print the path from root
# node to destination node for N*N-1 puzzle
# algorithm using Branch and Bound
# The solution assumes that instance of
# puzzle is solvable

# Importing copy for deepcopy function import
copy

# Importing the heap functions from python
```

```

# library for Priority Queue
from heapq import heappush, heappop

# This variable can be changed to change
# the program from 8 puzzle(n=3) to 15 #
# puzzle(n=4) to 24 puzzle(n=5)...
n = 3

# bottom, left, top, right row
= [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]

# A class for Priority Queue class
priorityQueue:

    # Constructor to initialize a
    # Priority Queue      def
    __init__(self):
        self.heap = []

    # Inserts a new key 'k'
    def push(self, k):
        heappush(self.heap, k)

    # Method to remove minimum
    # element            # from Priority Queue    def
    pop(self):
        return heappop(self.heap)

    # Method to know if the Queue is empty
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# Node structure class
node:

    def __init__(self, parent, mat, empty_tile_pos,
        cost, level):

        # Stores the parent node of the
        # current node helps in tracing
        # path when the answer is found
        self.parent = parent

```

```

        # Stores the matrix
        self.mat = mat

        # Stores the position at which the
        # empty space tile exists in the matrix
        self.empty_tile_pos = empty_tile_pos

        # Store the number of misplaced tiles
self.cost = cost

        # Stores the number of moves so far
        self.level = level

        # This method is defined so that the
        # priority queue is formed based on # the
        # cost variable of the objects def __lt__(self,
        nxt):
            return self.cost < nxt.cost

# Function to calculate the number of
# misplaced tiles ie. number of non-blank
# tiles not in their goal position def
calculateCost(mat, final) -> int:

        count = 0        for i in range(n):
for j in range(n):        if ((mat[i][j])
and                        (mat[i][j] != final[i][j])):
                            count += 1

        return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:

        # Copy data from parent matrix to current matrix
new_mat = copy.deepcopy(mat)

        # Move tile by 1 position x1
        = empty_tile_pos[0] y1 =
        empty_tile_pos[1] x2 =
        new_empty_tile_pos[0] y2 =
        new_empty_tile_pos[1]
        new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

        # Set number of misplaced tiles
        cost = calculateCost(new_mat, final)

        new_node = node(parent, new_mat, new_empty_tile_pos,

```

```

                                cost, level)

    return new_node

# Function to print the N x N matrix def
printMatrix(mat):

    for i in range(n):
        for j in range(n):
            print("%d " % (mat[i][j]), end = " ")

        print()

# Function to check if (x, y) is a valid
# matrix coordinate def
isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

# Print path from root node to destination node def
printPath(root):
    if root
    == None:
        return

    printPath(root.parent)
    printMatrix(root.mat)    print()

# Function to solve N*N - 1 puzzle algorithm
# using Branch and Bound. empty_tile_pos is
# the blank tile position in the initial state. def
solve(initial, empty_tile_pos, final):

    # Create a priority queue to store live
    # nodes of search tree pq
    = priorityQueue()

    # Create the root node cost =
    calculateCost(initial, final)
    root = node(None, initial,
                                empty_tile_pos, cost, 0)
    # Add root to list of live nodes pq.push(root)

    # Finds a live node with least cost,
    # add its children to list of live #
    # nodes and finally deletes it from
    # the list.
    while not pq.empty():

```

```

        # Find a live node with least estimated
        # cost and delete it form the list of
        # live nodes
        minimum = pq.pop()

        # If minimum is the answer node
        if minimum.cost == 0:

            # Print the path from root to
            # destination;
            printPath(minimum)
            return

        # Generate all possible children
        for i in range(n):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]
            if
isSafe(new_tile_pos[0], new_tile_pos[1]):

                # Create a child node
                child = newNode(minimum.mat,
                                minimum.empty_tile_pos,
                                new_tile_pos,
                                minimum.level + 1,
                                minimum, final,)

                # Add child to list of live nodes
                pq.push(child)

# Driver Code

# Initial configuration
# Value 0 is used for empty space initial
= [ [ 1, 2, 3 ],
      [ 5, 6, 0 ],
      [ 7, 8, 4 ] ]

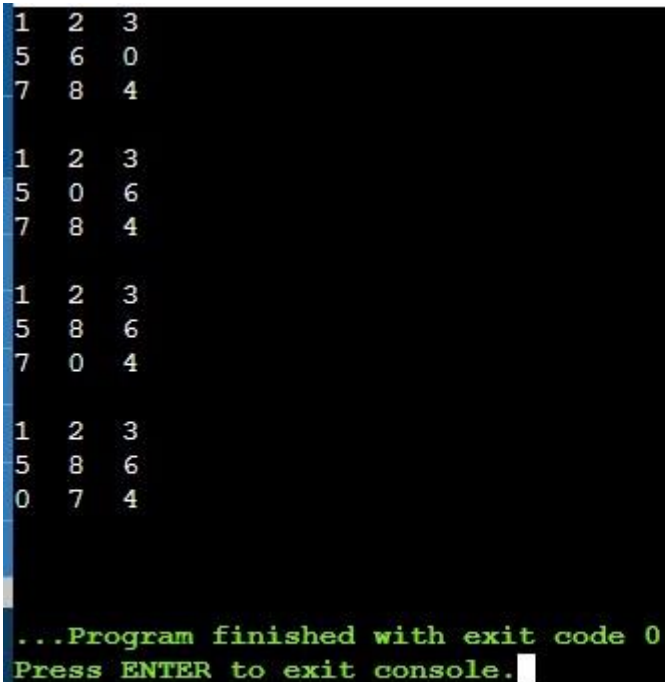
# Solvable Final configuration #
Value 0 is used for empty space
final = [ [ 1, 2, 3 ],
           [ 5, 8, 6 ],
           [ 0, 7, 4 ] ]

```



```
# Blank tile coordinates in #  
initial configuration  
empty_tile_pos = [ 1, 2 ]  
  
# Function call to solve the puzzle  
solve(initial, empty_tile_pos, final)  
  
# This code is contributed by Kevin Joshi
```

## OUTPUT:



```
1 2 3  
5 6 0  
7 8 4  
  
1 2 3  
5 0 6  
7 8 4  
  
1 2 3  
5 8 6  
7 0 4  
  
1 2 3  
5 8 6  
0 7 4  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

## RESULT:

Hence, The explanation about the sliding tile puzzle problem using branch and bound is stated and the output is presented and verified.

## FINDING NUMBER OF ISLANDS PROBLEM (AGENT PROBLEM)

EXP NO: 2

DATE :

### AIM:

Finding number of islands using python (using DFS )

### INTRODUCTION:

#### SIMPLE REFLEX AGENT PROBLEM:

Simple reflex agents ignore the rest of the percept history and act only on the basis of the current percept. Percept history is the history of all that an agent has perceived to date. The agent function is based on the condition-action rule. A condition-action rule is a rule that maps a state i.e, condition to an action. If the condition is true, then the action is taken, else not. This agent function only succeeds when the environment is fully observable. For simple reflex agents operating in partially observable environments, infinite loops are often unavoidable. It may be possible to escape from infinite loops if the agent can randomize its actions.

- Given a boolean 2D matrix, find the number of islands. A group of connected 1s forms an island. For example, the below matrix contains 5 islands.
- EXAMPLE: Input :  $\text{mat}[][] = \{\{1, 1, 0, 0, 0\},$   
 $\{0, 1, 0, 0, 1\},$   
 $\{1, 0, 0, 1, 1\},$   
 $\{0, 0, 0, 0, 0\},$   
 $\{1, 0, 1, 0, 1\}\}$

OUTPUT: 5

### PROBLEM STATEMENT:

Given a matrix of size M x N, where '1' represents land, while '0' represents water. The task is to return the number of islands present in the matrix. An **island** is a group of 1's surrounded either vertically or horizontally.

### Approach: DFS

The idea is to consider the given matrix as a graph, where each cell is a node of the given graph. Two nodes contain an edge if and only if there is a '1' either horizontally or vertically.

### EXPLANATION:

A cell in 2D matrix can be connected to 8 neighbours. So, unlike standard DFS(), where we recursively call for all adjacent vertices, here we can recursively call for 8 neighbours only. We keep track of the visited 1s so that they are not visited again.

### ALGORITHM:

1. Scan the matrix from (0,0) till (N, M).
2. If the current element is '1', start a **DFS**.
3. In the DFS traversal, mark every visited node.
4. Count the number of islands as the number nodes that trigger the DFS.
5. Return count.

### PROGRAM CODE:

```
class Graph:
    def __init__(self, row, col, g):
        self.ROW = row
        self.COL = col
        self.graph = g
    def isSafe(self, i, j, visited):
        return (i >= 0 and i < self.ROW and
                j >= 0 and j < self.COL and
                not visited[i][j] and self.graph[i][j])
    def DFS(self, i, j, visited):
        rowNbr = [-1, -1, -1, 0, 0, 1, 1, 1];
        colNbr = [-1, 0, 1, -1, 1, -1, 0, 1];
        visited[i][j] = True
```

```

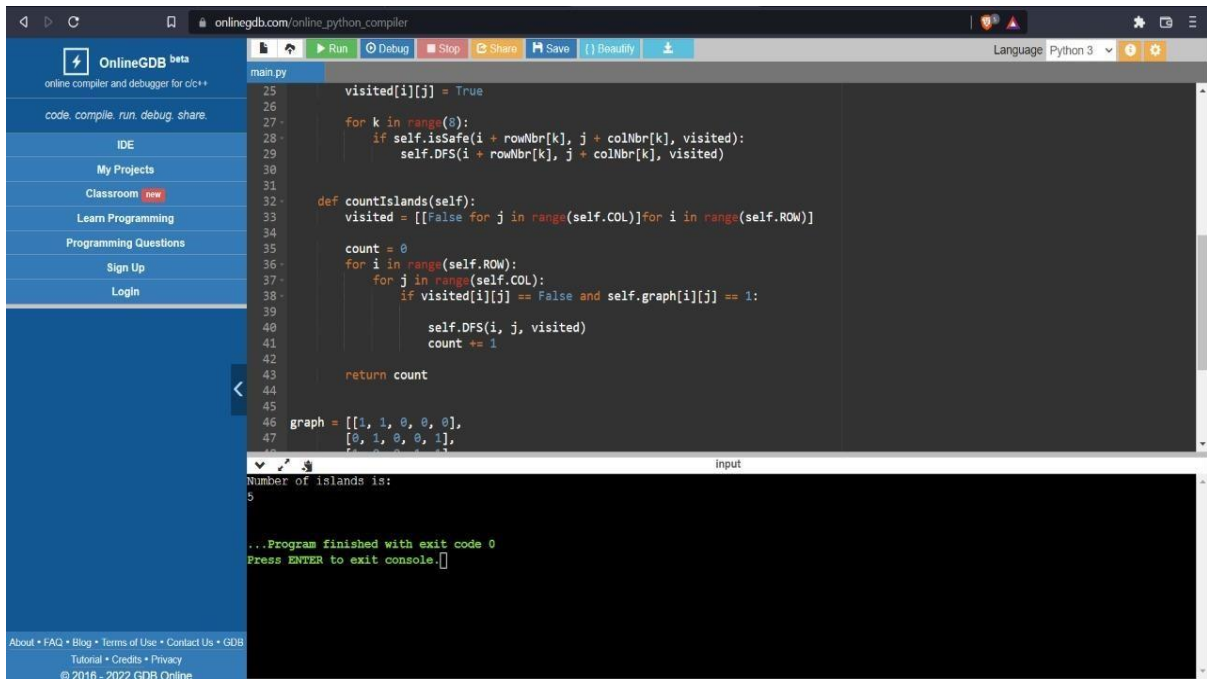
        for k in range(8):
            if self.isSafe(i +
rowNbr[k], j + colNbr[k], visited):
                self.DFS(i +
rowNbr[k], j + colNbr[k], visited)
            def countIslands(self):
                visited = [[False for j in range(self.COL)]for i in range(self.ROW)]
                count = 0
                for i in
range(self.ROW):
                    for j in
range(self.COL):
                        # If a cell with value 1 is not visited yet,
                        # then new island found
                        if visited[i][j] == False and self.graph[i][j] == 1:
                            self.DFS(i, j, visited)
                            count += 1

                return count

graph = [[1, 1, 0, 0, 0],
         [0, 1, 0, 0, 1],
         [1, 0, 0, 1, 1],
         [0, 0, 0, 0, 0],
         [1, 0, 1, 0, 1]]
row = len(graph) col =
len(graph[0]) g = Graph(row,
col, graph) print ("Number of
islands is:") print
(g.countIslands())

```

## OUTPUT:



The screenshot shows the OnlineGDB web interface. On the left is a blue sidebar with navigation links: OnlineGDB beta, code, compile, run, debug, share, IDE, My Projects, Classroom (new), Learn Programming, Programming Questions, Sign Up, and Login. The main area has a toolbar with Run, Debug, Stop, Share, Save, and Beautify buttons. The code editor contains a Python script for counting islands using DFS. The console at the bottom shows the output: 'Number of islands is: 5' and '...Program finished with exit code 0'. The graph data is defined as a 2D array.

```
main.py
25     visited[i][j] = True
26
27     for k in range(8):
28         if self.isSafe(i + rowNbr[k], j + colNbr[k], visited):
29             self.DFS(i + rowNbr[k], j + colNbr[k], visited)
30
31
32     def countIslands(self):
33         visited = [[False for j in range(self.COL)] for i in range(self.ROW)]
34
35         count = 0
36         for i in range(self.ROW):
37             for j in range(self.COL):
38                 if visited[i][j] == False and self.graph[i][j] == 1:
39                     self.DFS(i, j, visited)
40                     count += 1
41
42         return count
43
44
45
46 graph = [[1, 1, 0, 0, 0],
47           [0, 1, 0, 0, 1],
48           [0, 0, 0, 0, 0],
49           [0, 0, 0, 0, 0],
50           [0, 0, 0, 0, 0]]
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

input

Number of islands is:  
5

...Program finished with exit code 0  
Press ENTER to exit console.

## RESULT:

Hence number of islands are found using dfs in python.

## **CONSTRAINT SATISFACTION PROBLEM – GRAPH COLORING PROBLEM**

EXP NO: 3

DATE:

### **AIM:**

To solve a constraint satisfaction problem(Graph coloring) using Python.

### **INTRODUCTION:**

In a Constraint satisfaction problem, 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 fulfill all the constraints.

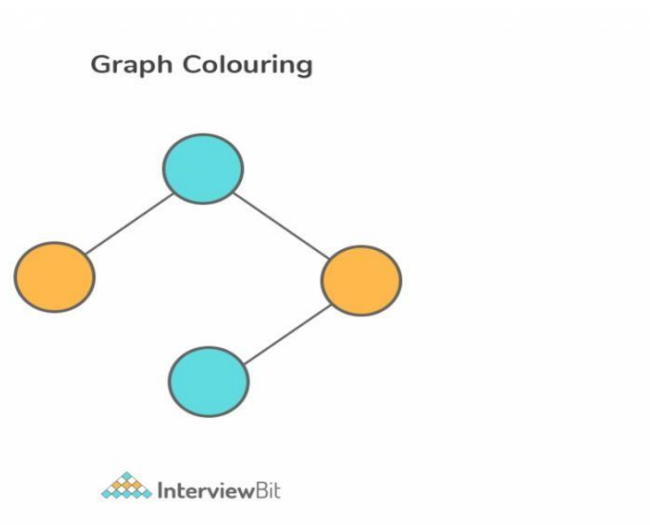
Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color.

### **PROBLEM STATEMENT:**

Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such that no two adjacent vertexes have the same color is called Graph Colouring.

This is also known as vertex coloring.

## EXAMPLE:



## ALGORITHM:

- Color first vertex with first color.
- Do following for remaining  $v-1$  vertices:
- Consider the currently picked vertex and color it with the lowest numbered color that has not been used on any previously coloured vertices adjacent to it.
- If all previously used colors on vertices adjacent to  $v$ , assign a new color to it.

## PROGRAM 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)
```

```

# Function to assign colors to vertices of a graph
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]]}')

# Greedy coloring of a graph
if __name__ == '__main__':

    # Add more colors for graphs with many more vertices
    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)]

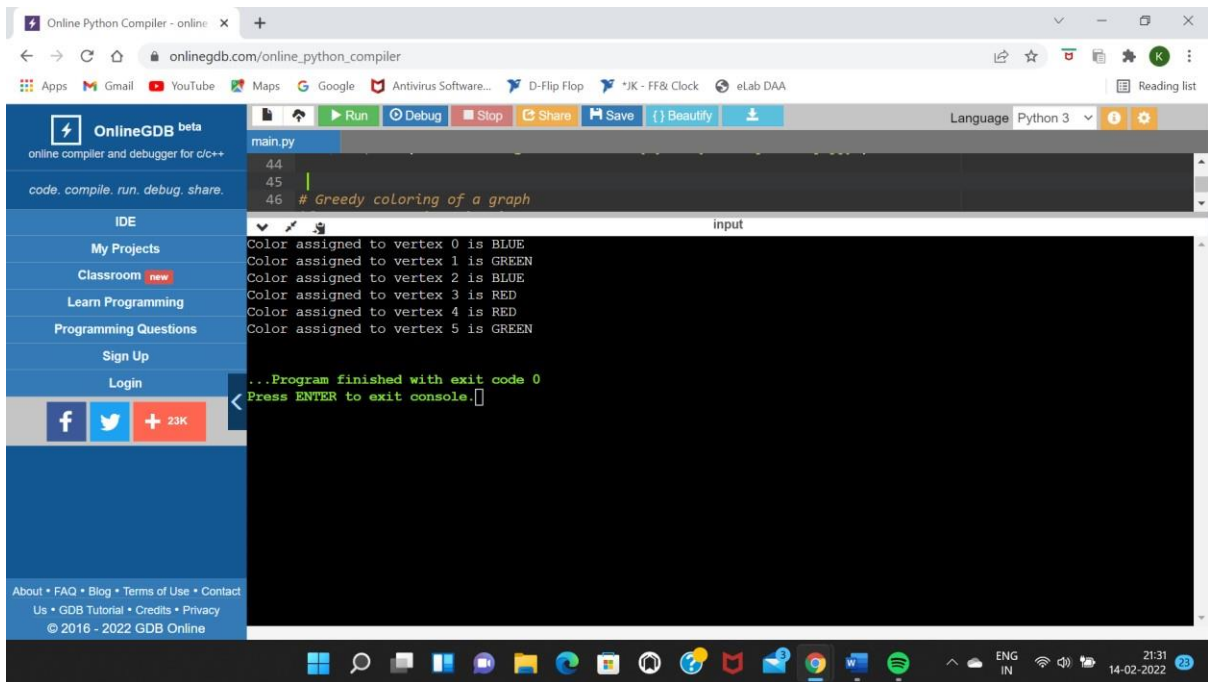
    n = 6

    graph = Graph(edges, n)
    colorGraph(graph, n)

```



## OUTPUT:



The screenshot shows the OnlineGDB web interface. The code editor contains a Python script named `main.py` with the following content:

```
44  
45  
46 # Greedy coloring of a graph
```

The output window displays the execution results:

```
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  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

The interface also includes a sidebar with navigation links (My Projects, Classroom, Learn Programming, etc.), a top toolbar with buttons (Run, Debug, Stop, etc.), and a footer with social media links and copyright information.

## RESULT:

Hence graph coloring is done using python code.

# DFS AND BFS - SUBSET SUM

EXP NO: 4

DATE:

## AIM:

Write a program to solve the BFS and DFS problem.

## INTRODUCTION:

BFS stands for Breadth First Search is a vertex based technique for finding a shortest path in graph. It uses a Queue data structure which follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

DFS stands for Depth First Search is a edge based technique. It uses the Stack data structure, performs two stages, first visited vertices are pushed into stack and second if there is no vertices then visited vertices are popped.

## PROBLEM STATEMENT:

Given a set of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

Example:

Input: set[] = {3, 34, 4, 12, 5, 2},

sum = 9

Output: True

There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2},

sum = 30

Output: False

There is no subset that add up to 30

## APPROACH:

### BACKTRACKING

We use backtracking approach. It is a recursive algorithm that uses brute force concept. The term backtracking implies that if the current solution is not suitable, then move a step back and try other solutions.

**PROGRAM CODE:** import sys

class Subset :

```

def printSum(self, result, front, tail) :
    print("[", end =
    "" ) i = front while
    (i < tail) :
        if (result[i] != sys.maxsize) :
            print(" ", result[i] , " ", end = "")
            i += 1
    print("]")

def subsetSum(self, arr, result, sum, size, current_sum, location) :
    if (location == -1) :
        return
    self.subsetSum(arr, result, sum, size, current_sum, location - 1)
result[location] = arr[location] if (current_sum + arr[location] == sum) :
    self.printSum(result, location, size)
    self.subsetSum(arr, result, sum, size, current_sum + arr[location], location - 1)
    result[location] = sys.maxsize

def findSubset(self, arr, size, sum) :
    if (size <= 0) :
        return
    result = [sys.maxsize] * (size)

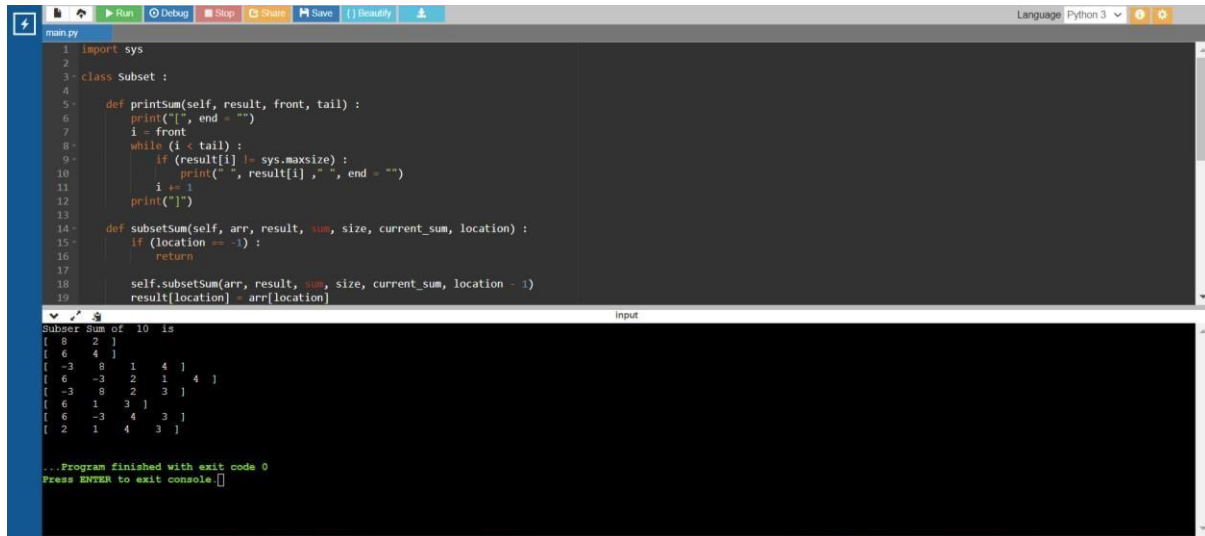
    print("Subser Sum of ", sum , " is ")
    self.subsetSum(arr, result, sum, size, 0, size - 1)

def main() :
    task = Subset() arr = [6,
    -3, 8, 2, 1, 4, 3] size =
    len(arr) sum = 10
    task.findSubset(arr, size, sum)

```

```
if __name__ == "__main__": main()
```

## OUTPUT:



The screenshot shows a Python IDE with a dark theme. The top toolbar includes icons for Run, Debug, Stop, Share, Save, and a Beautify button. The code editor displays a Python class named `Subset` with two methods: `printSum` and `subsetSum`. The `printSum` method prints the elements of an array from index `front` to `tail`. The `subsetSum` method recursively finds all subsets of an array that sum up to a given value `sum`. The output window shows the results of the program, displaying the sum of the array and the subsets that sum up to 10.

```
1 import sys
2
3 class Subset :
4
5     def printSum(self, result, front, tail) :
6         print("[", end = "")
7         i = front
8         while (i <= tail) :
9             if (result[i] != sys.maxsize) :
10                 print(" ", result[i], " ", end = "")
11                 i += 1
12             print("]")
13
14     def subsetSum(self, arr, result, sum, size, current_sum, location) :
15         if (location == -1) :
16             return
17         self.subsetSum(arr, result, sum, size, current_sum, location - 1)
18         result[location] = arr[location]
```

Subser Sum of 10 is

```
[ 8 2 ]
[ 6 4 ]
[ -3 8 1 4 ]
[ 6 -3 2 1 4 ]
[ -3 8 2 3 ]
[ 6 1 3 ]
[ 6 -3 4 3 ]
[ 2 1 4 3 ]

...Program finished with exit code 0
Press ENTER to exit console.
```

## RESULT:

Hence the Subset sum problem has been solved and verified.

# A\* ALGORITHM

EXP NO: 5

DATE:

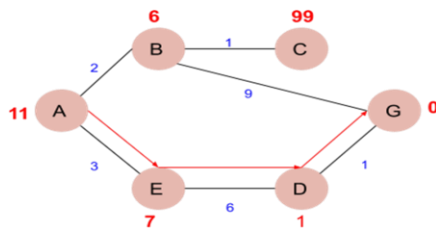
## AIM:

Write a program to solve Best first and A\* problem.

## INTRODUCTION:

The A\* Algorithm is a best-first search algorithm that finds the least cost path from an initial configuration to a final configuration. The most essential part of the A\* Algorithm is a good heuristic estimate function. This can improve the efficiency and performance of the algorithm. It is an extension of Dijkstra's algorithm.

## GRAPH:



## IMPLEMENTATION:

- we write a program in Python that can find the most cost-effective path by using the a-star algorithm.
- First, we create two sets, viz- open, and close. The open contains the nodes that have been visited but their neighbors are yet to be explored. On the other hand, close contains nodes that along with their neighbors have been visited.

## PROGRAM CODE:

```
def aStarAlgo(start_node, stop_node):
```

```
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    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:

```

Online Python Compiler - online
onlinegdb.com/online_python_compiler

main.py
83- return Graph_nodes[v]
84- else:
85-     return None
86- def heuristic(n):
87-     H_dist = {
88-         'A': 11,
89-         'B': 6,
90-         'C': 99,
91-         'D': 1,
92-         'E': 7,
93-         'G': 0,
94-     }
95-     return H_dist[n]
96-
97- Graph_nodes = {
98-     'A': [('B', 2), ('E', 3)],
99-     'B': [('C', 1), ('G', 9)],
100-     'C': None,
101-     'E': [('D', 6)],
102-     'D': [('G', 1)],
103- }
104- aStarAlgo('A', 'G')
105-
106- Path found: ['A', 'B', 'D', 'G']
107-
108- ... Program finished with exit code 0
109- Press ENTER to exit console.

```

## RESULT:

Hence the A\* algorithm problem has been solved and verified.

# MIN MAX ALGORITHM – TIC TAC TOE

EXP NO: 6

DATE:

## AIM:

Write a program to solve min max problem.

## INTRODUCTION:

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible. Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

## TIC TAC TOE:

Tic-tac-toe is a very popular game, so let's implement an automatic Tic-tac-toe game using Python. The game is automatically played by the program and hence, no user input is needed. Still, developing a automatic game will be lots of fun. Let's see how to do this. numpy and random Python libraries are used to build this game. Instead of asking the user to put a mark on the board, code randomly chooses a place on the board and put the mark. It will display the board after each turn unless a player wins. If the game gets draw, then it returns -1.

## EXPLANATION:

1. play\_game() is the main function, which performs following tasks :
2. Calls create\_board() to create a 9×9 board and initializes with 0.
3. For each player (1 or 2), calls the random\_place() function to randomly choose a location on board and mark that location with the player number, alternatively.
4. Print the board after each move. Evaluate the board after each move to check whether a row or column or a diagonal has the same player number.
5. If so, displays the winner name.
6. If after 9 moves, there are no winner then displays -1.

## PROGRAM 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

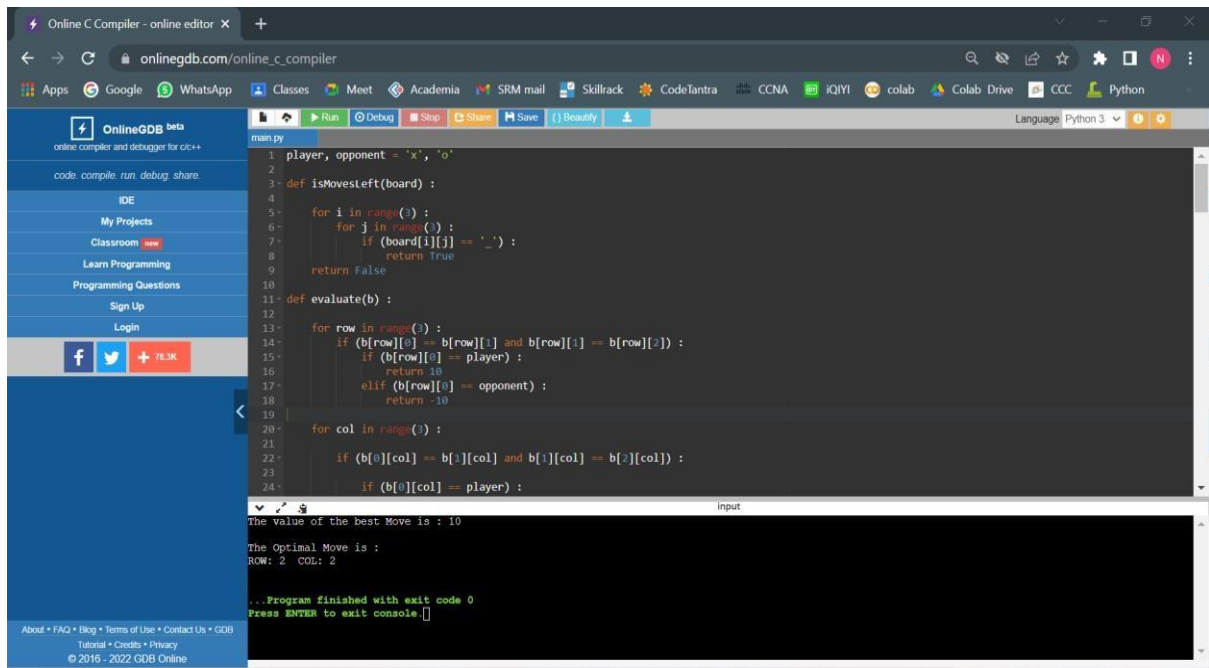
# Driver code board
= [      [ 'x', 'o', 'x'
],
        [ 'o', 'o', 'x' ],
        [ " ", ' ', '_' ]
]

bestMove = findBestMove(board)

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

```

## OUTPUT:



```
1 player, opponent = 'x', 'o'
2
3 def isMovesLeft(board) :
4
5     for i in range(3) :
6         for j in range(3) :
7             if (board[i][j] == '.' ) :
8                 return True
9     return False
10
11 def evaluate(b) :
12
13     for row in range(3) :
14         if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
15             if (b[row][0] == player) :
16                 return 10
17             elif (b[row][0] == opponent) :
18                 return -10
19
20     for col in range(3) :
21         if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :
22             if (b[0][col] == player) :
```

The Value of the best Move is : 10

The Optimal Move is :

ROW: 2 COL: 2

...Program finished with exit code 0

Press ENTER to exit console.

## RESULT:

Hence the tic tac toe has been solved and verified.

# IMPLEMENTATION OF UNIFICATION AND RESOLUTION

EXP NO: 7

DATE:

**AIM:** To implement unification and resolution of real world problems using python.

## PROBLEM STATEMENT:

- **Unification:** A unification problem is a finite set of equations. A solution or a unifier of such a problem is a substitution  $\theta$  such that for each pair  $t, u$  of the problem, the terms  $\theta t$  and  $\theta u$  have the same normal form.
- **Resolution:** Resolution method is an inference rule which is used in both Propositional as well as First-order Predicate Logic in different ways. This method is basically used for proving the satisfiability of a sentence.

## PROCEDURE:

### UNIFICATION:

Step. 1: If  $\Psi_1$  or  $\Psi_2$  is a variable or constant, then:

- If  $\Psi_1$  or  $\Psi_2$  are identical, then return NIL.
- Else if  $\Psi_1$  is a variable,
  - then if  $\Psi_1$  occurs in  $\Psi_2$ , then return FAILURE
  - Else return  $\{ (\Psi_2 / \Psi_1) \}$ .
- Else if  $\Psi_2$  is a variable,
  - If  $\Psi_2$  occurs in  $\Psi_1$  then return FAILURE,
  - Else return  $\{ (\Psi_1 / \Psi_2) \}$ .
- Else return FAILURE.

Step.2: If the initial Predicate symbol in  $\Psi_1$  and  $\Psi_2$  are not same, then return FAILURE.

Step. 3: IF  $\Psi_1$  and  $\Psi_2$  have a different number of arguments, then return FAILURE.

Step. 4: Set Substitution set(SUBST) to NIL.

Step. 5: For  $i=1$  to the number of elements in  $\Psi_1$ .

a) Call Unify function with the  $i$ th element of  $\Psi_1$  and  $i$ th element of  $\Psi_2$ , and put the result into S.

b) If S = failure then returns Failure

c) If  $S \neq \text{NIL}$  then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST= APPEND(S, SUBST).

Step.6: Return SUBST.

### CODE FOR UNIFICATION:

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

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

    return index_list
```

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

    return True
```

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

```

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

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

    return predicate_symbol, arg_list

```

```

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

```

```

    flag = True
while flag:
flag = False
    for i in arg_list:
if not is_variable(i):
    flag = True
    _, tmp = process_expression(i)
for j in tmp:
    if j not in arg_list:
arg_list.append(j)
    arg_list.remove(i)

    return arg_list

```

```

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

```

```

    return False

```

```

def unify(expr1, expr2):

```

```

    if is_variable(expr1) and is_variable(expr2):
if expr1 == expr2:        return 'Null'
    else:
        return False    elif is_variable(expr1) and
not is_variable(expr2):    if check_occurs(expr1,
expr2):
        return False
    else:
        tmp = str(expr2) + '/' + str(expr1)
        return tmp    elif not is_variable(expr1) and
is_variable(expr2):        if check_occurs(expr2,
expr1):
        return False
    else:
        tmp = str(expr1) + '/' + str(expr2)
        return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

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

        # Step 5:        for i in
range(len(arg_list_1)):
            tmp = unify(arg_list_1[i], arg_list_2[i])
            if not tmp:
return False                elif
tmp == 'Null':
                pass                else:
if type(tmp) == list:
    for j in tmp:
        sub_list.append(j)
    else:
        sub_list.append(tmp)

    # Step 6
    return sub_list

if __name__ == '__main__':

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

```



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

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

```
if not result:
```

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

```
else:
```

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

```
print(result)
```

## OUTPUT:

The screenshot shows a web browser window with the URL `onlinegdb.com/online_python_compiler`. The page displays the source code of a Python program for unification. The code defines a `unify` function and uses it to unify two queries, `f1` and `f2`. The output of the program is shown in the console, indicating that the unification process was successful and returning the substitution `{'f(b)/x', 'f(y)/x'}`.

```
main.py
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140

...Program finished with exit code 0
Press ENTER to exit console.
```

## CODE FOR RESOLUTION:

```
import sys
import re
```

```
queries = list()
kb = list() nq
= 0
ns = 0
```

```
def get_input():    fin =
"input.txt"    output_file =
"output.txt"    global
queries    global kb
global nq
    global ns
```

```
try:
```

```

        input_file = open(fin, 'r')
        lines = input_file.readlines()
    for index, line in enumerate(lines):
        if index
        == 0:
            nq =
            int(lines[index].strip("\n"))
            for i in
            range(1, nq + 1):
                queries.append(lines[i].strip("\n"))
            ns = int(lines[nq+1].strip("\n"))
            for i
            in range(nq + 2, nq + ns + 2):
                kb.append(lines[i].strip("\n"))
            break
    input_file.close()
    return queries, kb

```

```

    except IOError:
        fo = open(output_file,
        'w')
        fo.write("File not found:
        { }".format(fin))
        fo.close()
        sys.exit()

```

```

def parseKB(kb):
    negativeKB = dict()
    positiveKB = dict()

    for item in kb:
        data = item.split('|')
    for i in data:
        i = i.replace(' ', '')
        if i[0] == '~':
            b = i[1:]
            b = b.partition("(")[0]
        try:
            negativeKB[b].append(item)
        except KeyError:
            negativeKB[b] = [item]
    else:
        i = i.partition("(")[0]
    try:
        positiveKB[i].append(item)
    except KeyError:
        positiveKB[i] = [item]

    return negativeKB, positiveKB

```

```

def extract_constants(query):
    variable = re.search(r'\(((.*?))\)', query).group(1)
    return variable

```

```

def checkSentence(kb):
    if "|" in kb:
    return False
    const_list = re.search(r'((.*?))', kb).group(1)
    const = const_list.split(",")    for val in const:
    if val[0].isupper():
        continue
    else:
        return False
    return True

def unification(query,left_over,positiveKB,negativeKB,can_simplifyfy):
    #print("In unification")    if
    query[0] != '~':        tomatch =
    query.partition("(")[0]    try:
        value = negativeKB[tomatch]
    except KeyError:        return
    False

    for sentence in value:
    try:
        left_over_temp = left_over
        query_temp = query

        if sentence in can_simplifyfy:                ret1, l1
= remove(left_over_temp, sentence[1:])                ret2 =
1                l2 = ""                else:
        ret1, l1 = remove(left_over_temp, query_temp)
    ret2, l2 = remove(sentence, "~" + query_temp)        if
    ret1 == 0 or ret2 == 0:
        continue
    else:
        if l1 == "" and l2 != "":
        left_over_temp = l2                elif
        l2 == "" and l1 != "":
        left_over_temp = l1                elif
        l1 == "" and l2 == "":
        left_over_temp = ""                else:
        left_over_temp = l2 + " | " + l1

        if left_over_temp == "":
    return True                else:                if
    "|" in left_over_temp:                data =
    left_over_temp.split("|")                for i
    in data:                i = i.replace(" ", "")
    if

```

```

unification(i,left_over_temp,positiveKB,negativeKB,can_simplify):
return True
else:
break
else:
if
unification(left_over_temp,left_over_temp,positiveKB,negativeKB,can_simplify):
return True
else:
continue
except
RuntimeError as re:
if re.args[0] == 'maximum recursion depth
exceeded':
return False

return False
else:
tomatch =
query.partition("(")[0]
try:
value = positiveKB[tomatch[1:]]
except KeyError:
return False
for sentence in value:
try:
left_over_temp = left_over
query_temp =
query
if sentence in can_simplify:
ret_val1,
l1 = remove(left_over_temp, "~" + sentence)
ret_val2
= 1
l2 = ""
else:
ret_val1, l1 = remove(left_over_temp, query_temp)
ret_val2, l2 = remove(sentence, query_temp[1:])
if
ret_val1 == 0 or ret_val2 == 0:
continue
else:
if l1 == "" and l2 != "":
left_over_temp = l2
elif
l2 == "" and l1 != "":
left_over_temp = l1
elif
l1 == "" and l2 == "":
left_over_temp = ""
else:
left_over_temp = l2 + " | " + l1

if left_over_temp == "":
return True
else:
if "|" in left_over_temp:
data = left_over_temp.split("|")
for i in data:
i = i.replace(" ", "")
if unification(i,
left_over_temp, positiveKB, negativeKB, can_simplify):
return True
else:
break
else:
if unification(left_over_temp, left_over_temp, positiveKB, negativeKB,
can_simplify):
return True
else:
continue
except RuntimeError as re:
if
re.args[0] == 'maximum recursion depth exceeded':
return False
return False

```

```

def remove(k,query):
    __int, newq, news = substitution(k, query)
    if __int == 1:      if newq in news:
news1 = news.replace(newq, "")      else:
        start = news.find(query.partition("(")[0])
        end = news.find(')', start)
to_del = news[start:end + 1]      news1
= news.replace(to_del, "")      if " | | " in
news1:      news2 = news1.replace(" |
| ", " | ")      return 1,news2      elif
news1[:3] == " | ":      news2 =
news1[3:]      return 1,news2      elif
news1[-3:] == " | ":      news2 =
news1[:-3]      return 1,news2
else:
    return 1, news1
else:
    return 0,news

```

```

def substitution(sentence,query):
predicate = query.partition("(")[0]

    constant = extract_constants(query)
cons_list = constant.split(",")
    count = 0

    data = sentence.split("|")
    flag = 0    for i in data:
m = i.partition("(")[0]
m = m.replace(' ', "")    if
m == predicate:
        __vars = re.search(r'\((.*?)\)',i).group(1)
var_list = __vars.split(",")    for j in var_list:
if j[0].isupper() and cons_list[count][0].islower():
query = test(cons_list[count], query, j)    flag = 1
count += 1    elif j[0].islower() and
cons_list[count][0].isupper():
        sentence = test(j, sentence, cons_list[count])
flag = 1    count +=1    elif j[0].isupper()
and cons_list[count][0].isupper():    if j ==
cons_list[count]:    query = query
sentence = sentence
        flag = 1    else:    flag
= 0    break    count +=1
elif j[0].islower() and cons_list[count][0].islower():

```

```

        # print("both variables")
    if not (j == cons_list[count]):
    ## add code here
        sentence = test(j, sentence, cons_list[count])
        query = query
    flag = 1        else:
        sentence = sentence
    query = query        flag
    = 1        count += 1
    if flag == 1:
        break
    if flag == 0:
        return 0, query, sentence
    else:
        return 1, query, sentence


def test(word, to_replace, with_replace):    big_regex =
re.compile(r'\b%s\b' % r'\b\\b'.join(map(re.escape, word)))    a =
big_regex.sub(with_replace, to_replace)    return(a)


def main():    output_file = "output.txt"    fo =
open(output_file, 'w')    query_list, sentences =
get_input()    negativeKB, positiveKB =
parseKB(sentences)
    can_simplifyfy = []
    for a in sentences:        if
checkSentence(a):
        can_simplifyfy.append(a)


    for query in query_list:
    if query[0] == '~':
        new_query = query[1:]        if unification(new_query, new_query,
positiveKB, negativeKB, can_simplifyfy):
            fo.write("TRUE" + "\n")
    else:
        fo.write("FALSE" + "\n")


    else:
        new_query = "~" + query        if unification(new_query, new_query,
positiveKB, negativeKB, can_simplifyfy):
            fo.write("TRUE" + "\n")
    else:
        fo.write("FALSE" + "\n")
    fo.close()

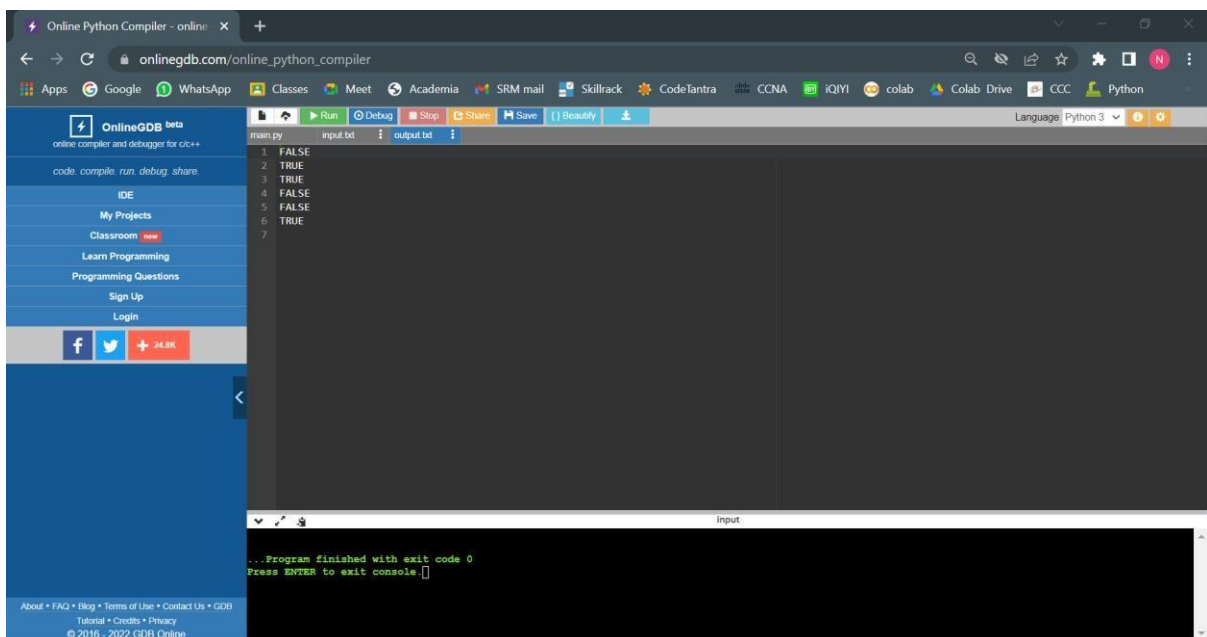
```

```
if __name__ == '__main__':  
    main()
```

## INPUT.TXT:

```
6  
F(Joe)  
H(John)  
~H(Alice)  
~H(John)  
G(Joe)  
G(Tom)  
14  
~F(x) | G(x)  
~G(x) | H(x)  
~H(x) | F(x)  
~R(x) | H(x)  
~A(x) | H(x)  
~D(x,y) | ~H(y)  
~B(x,y) | ~C(x,y) | A(x)  
B(John,Alice)  
B(John,Joe)  
~D(x,y) | ~Q(y) | C(x,y)  
D(John,Alice)  
Q(Joe)  
D(John,Joe)  
R(Tom)
```

## OUTPUT:



```
Online Python Compiler - online x +  
onlinegdb.com/online_python_compiler  
Apps Google WhatsApp Classes Meet Academia SRM mail Skillrack CodeLantra CCNA iQIYI colab Colab Drive CCC Python  
OnlineGDB beta  
online compiler and debugger for c/c++  
code compile run debug share  
IDE  
My Projects  
Classroom new  
Learn Programming  
Programming Questions  
Sign Up  
Login  
f t +24.8K  
About • FAQ • Blog • Terms of Use • Contact Us • GDB  
Tutorial • Credits • Privacy  
© 2016 - 2022 GDB Online  
main.py  
input.txt output.txt  
1 FALSE  
2 TRUE  
3 TRUE  
4 FALSE  
5 FALSE  
6 TRUE  
7  
...Program finished with exit code 0  
Press ENTER to exit console.
```

**RESULT:** The unification and resolution of real world problems were implemented and understood using Python.



## UNCERTAINTY– MONTY HALL PROBLEM

EXP NO: 8

DATE:

### AIM:

Write a program to solve Monty hall problem.

### INTRODUCTION:

The *Monty Hall problem* is a famous conundrum in probability which takes the form of a hypothetical game show. The contestant is presented with three doors; behind one is a car and behind each of the other two is a goat. The contestant picks a door and then the gameshow host opens a different door to reveal a goat. The host knows which door conceals the car. The contestant is then invited to switch to the other closed door or stick with their initial choice. The best for winning the car is to switch.

### EXPLANATION:

Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?

### PROBLEM STATEMENT:

The problem is stated as follows. Assume that a room is equipped with three doors. Behind two are goats, and behind the third is a shiny new car. You are asked to pick a door, and will win whatever is behind it. Let's say you pick door 1. Before the door is opened, however, someone who knows what's behind the doors (Monty Hall) opens one of the other two doors, revealing a goat, and asks you if you wish to change your selection to the third door (i.e., the door which neither you picked nor he opened). The Monty Hall problem is deciding whether you do.

The correct answer is that you do want to switch. If you do not switch, you have the expected  $1/3$  chance of winning the car, since no matter whether you initially picked the correct door, Monty will show you a door with a goat. But after Monty has eliminated one of the doors for you, you obviously do not improve your chances of winning to better than  $1/3$  by sticking with your original choice. If you now switch doors, however, there is a  $2/3$  chance you will win the car

### PROGRAM CODE:

```
import
random

def run_trial(switch_doors, ndoors=3):
    """
    Run a single trial of the Monty Hall problem, with or without switching
    after the gameshow host reveals a goat behind one of the unchosen doors.
    (switch_doors is True or False). The car is behind door number 1 and the
    gameshow host knows that.

    """

    # Pick a random door out of the ndoors available
    chosen_door = random.randint(1, ndoors)    if
switch_doors:      # Reveal a goat
    revealed_door = 3 if chosen_door==2 else 2
```

```

        # Make the switch by choosing any other door than the initially-
        # selected one and the one just opened to reveal a goat.
        available_doors = [dnum for dnum in range(1,ndoors+1)
                           if dnum not in (chosen_door, revealed_door)]
        chosen_door = random.choice(available_doors)

    # You win if you picked door number 1
    return chosen_door == 1

def run_trials(ntrials, switch_doors, ndoors=3):
    """
    Run ntrials iterations of the Monty Hall problem with ndoors doors, with
    and without switching (switch_doors = True or False). Returns the number
    of trials which resulted in winning the car by picking door number 1.

    """

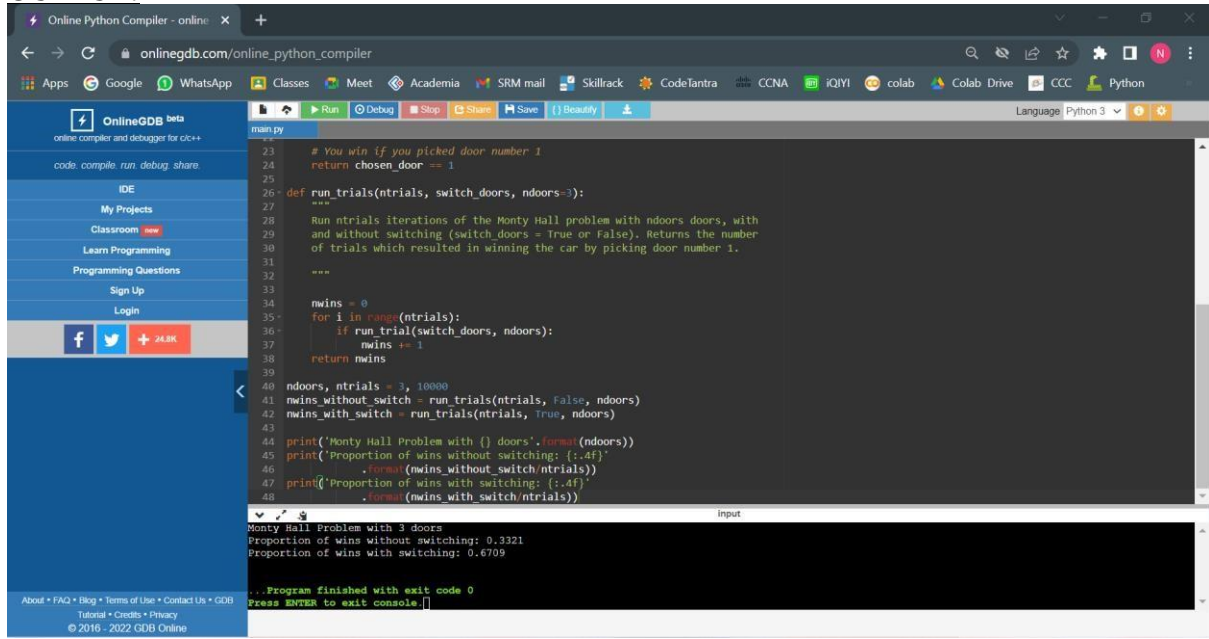
    nwins = 0
    for i in range(ntrials):
        if run_trial(switch_doors, ndoors):
            nwins += 1
    return nwins

ndoors, ntrials = 3, 10000
nwins_without_switch = run_trials(ntrials, False, ndoors)
nwins_with_switch = run_trials(ntrials, True, ndoors)

print('Monty Hall Problem with { } doors'.format(ndoors))
print('Proportion of wins without switching: {:.4f}'
      .format(nwins_without_switch/ntrials))
print('Proportion of wins with switching: {:.4f}'
      .format(nwins_with_switch/ntrials))

```

## OUTPUT:



The screenshot shows the OnlineGDB web interface. The left sidebar contains navigation links: OnlineGDB beta, code, compile, run, debug, share, IDE, My Projects, Classroom, Learn Programming, Programming Questions, Sign Up, and Login. The main editor displays a Python script for the Monty Hall problem. The script defines a function `run_trials` that simulates the game with a given number of trials and doors. It then runs the simulation 10,000 times for 3 doors, comparing the win rates with and without switching. The output in the console shows the results: 0.3321 for no switching and 0.6709 for switching.

```
main.py
23 # You win if you picked door number 1
24 return chosen_door == 1
25
26 def run_trials(ntrials, switch_doors, ndoors=3):
27     """
28     Run ntrials iterations of the Monty Hall problem with ndoors doors, with
29     and without switching (switch_doors = True or False). Returns the number
30     of trials which resulted in winning the car by picking door number 1.
31     """
32
33
34     mwins = 0
35     for i in range(ntrials):
36         if run_trial(switch_doors, ndoors):
37             mwins += 1
38     return mwins
39
40 ndoors, ntrials = 3, 10000
41 mwins_without_switch = run_trials(ntrials, False, ndoors)
42 mwins_with_switch = run_trials(ntrials, True, ndoors)
43
44 print('Monty Hall Problem with {} doors'.format(ndoors))
45 print('Proportion of wins without switching: {:.4f}'
46       .format(mwins_without_switch/ntrials))
47 print('Proportion of wins with switching: {:.4f}'
48       .format(mwins_with_switch/ntrials))
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

Monty Hall Problem with 3 doors  
Proportion of wins without switching: 0.3321  
Proportion of wins with switching: 0.6709

...Program finished with exit code 0  
Press ENTER to exit console.

## RESULT:

Hence the Monty hall problem has been solved and verified.

# LINEAR REGRESSION

EXP NO:9

DATE: 20-04-2022

## **AIM:**

To implement linear regression algorithm to predict the value of a dependent variable based on an independent variable.

## **DATASET USED:**

Boston house prices dataset.

This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University. The Boston house-price data has been used in many machine learning papers that address regression problems.

## **ALGORITHM:**

- Import some required libraries.
- Define the dataset.
- Plot the data points for better visualization.
- Calculate coefficient values :-
  - ⊕ Initialize the parameters.
  - ⊕ Predict the values of a dependent variable.
  - ⊕ Calculate error in prediction for all data point.
  - ⊕ Calculate cost of each step.
  - ⊕ Update values.
- Compute accuracy and error.

## **PROGRAM CODE:**

```
import numpy as np
import pandas as pd
```

```
#Visualization Libraries
```

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```
#To plot the graph embedded in the notebook
```

```
%matplotlib inline
from sklearn import datasets
from sklearn.linear_model import LinearRegression
```

```

from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.metrics import mean_squared_error
#loading the dataset directly from sklearn
boston = datasets.load_boston()
print(type(boston))
print('\n')
print(boston.keys())
print('\n')
print(boston.data.shape)
print('\n')
print(boston.feature_names)
print(boston.DESCR)
bos = pd.DataFrame(boston.data, columns = boston.feature_names)
bos['PRICE'] = boston.target

print(bos.head())
bos.isnull().sum()
print(bos.describe())
sns.set(rc={'figure.figsize':(11.7,8.27)})
plt.hist(bos['PRICE'], bins=30)
plt.xlabel("House prices in $1000")
plt.show()
#Created a dataframe without the price col, since we need to see the correlation
between the variables
bos_1 = pd.DataFrame(boston.data, columns = boston.feature_names)

correlation_matrix = bos_1.corr().round(2)
sns.heatmap(data=correlation_matrix, annot=True)
plt.figure(figsize=(20, 5))

features = ['LSTAT', 'RM']
target = bos['PRICE']

for i, col in enumerate(features):
    plt.subplot(1, len(features) , i+1)
    x = bos[col]
    y = target
    plt.scatter(x, y, marker='o')
    plt.title("Variation in House prices")
    plt.xlabel(col)

```

```

plt.ylabel("House prices in $1000")
X_rooms = bos.RM
y_price = bos.PRICE

X_rooms = np.array(X_rooms).reshape(-1,1)
y_price = np.array(y_price).reshape(-1,1)

print(X_rooms.shape)
print(y_price.shape)
X_train_1, X_test_1, Y_train_1, Y_test_1 = train_test_split(X_rooms, y_price,
test_size = 0.2, random_state=5)

print(X_train_1.shape)
print(X_test_1.shape)
print(Y_train_1.shape)
print(Y_test_1.shape)

#Model
reg_1 = LinearRegression()
#Fitting the model to the data
reg_1.fit(X_train_1, Y_train_1)

y_train_predict_1 = reg_1.predict(X_train_1)
rmse = (np.sqrt(mean_squared_error(Y_train_1, y_train_predict_1)))
r2 = round(reg_1.score(X_train_1, Y_train_1),2)

print("The model performance for training set")
print("-----")
print('RMSE is {}'.format(rmse))
print('R2 score is {}'.format(r2))
print("\n")

y_pred_1 = reg_1.predict(X_test_1)
rmse = (np.sqrt(mean_squared_error(Y_test_1, y_pred_1)))
r2 = round(reg_1.score(X_test_1, Y_test_1),2)

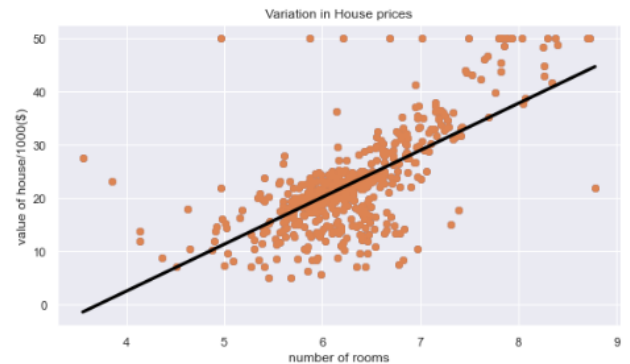
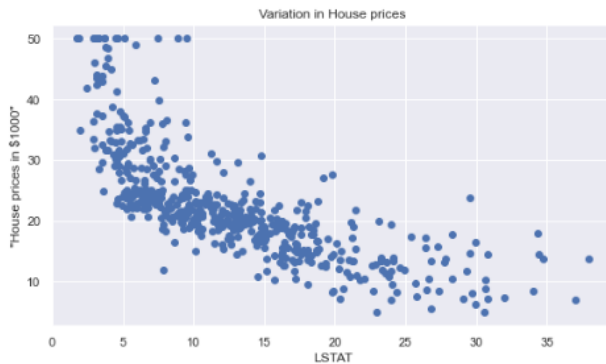
print("The model performance for training set")
print("-----")
print("Root Mean Squared Error: {}".format(rmse))

```

```
print("R^2: {}".format(r2))
print("\n")
```

```
prediction_space = np.linspace(min(X_rooms), max(X_rooms)).reshape(-1,1)
plt.scatter(X_rooms,y_price)
plt.plot(prediction_space, reg_1.predict(prediction_space), color = 'black',
linewidth = 3)
plt.ylabel('value of house/1000($)')
plt.xlabel('number of rooms')
plt.show()
```

## **OUTPUT:**



## **ACCURACY:**

R2 score is 0.43

$R^2$ : 0.69

```
\root{2}
The model performance for training set
-----
RMSE is 6.972277149440585
R2 score is 0.43
```

```
The model performance for training set
-----
Root Mean Squared Error: 4.895963186952216
R^2: 0.69
```

## **RESULT:**

Hence the task of linear regression with multiple variables has been performed with good accuracy.

# **SUPPORT VECTOR MACHINE (SVM)**

EXP NO:10

DATE: 20-04-2022

## **AIM:**

To write a program to implement support vector machine.

## **ALGORITHM:**

The goal of the SVM algorithm is to create the best line or decision boundary that can segregate n-dimensional space into classes so that we can easily put the new data point in the correct category in the future. This best decision boundary is called a hyper plane.

SVM chooses the extreme points/vectors that help in creating the hyper plane.

These extreme cases are called support vectors, and hence the algorithm is termed as Support Vector Machine.

## **PROCEDURE:**

- Remove blank rows if any.
- Change all the text to lower case. This is required as python interprets 'dog' and 'DOG' differently.
- Tokenization: In this each entry in the corpus will be broken into set of words.
- Tokenization: In this each entry in the corpus will be broken into set of words.
- Remove Stop words, Non-Numeric and perform Word Stemming/ Lemmenting.
- WordNetLemmatizer requires Pos tags to understand if the word is noun or verb or adjective etc. By default it is set to Noun.
- Declaring Empty List to store the words that follow the rules for this step.
- Initializing WordNetLemmatizer().
- pos\_tag function will provide the 'tag' i.e. if the word is Noun(N) or Verb(V) or something else.
- The final processed set of words for each iteration will be stored in 'text\_final'.
- Predict the labels on validation dataset.
- Use accuracy\_score function to get the accuracy.

## **PROGRAM CODE:**



```

import pandas as pd
import numpy as np
from nltk.tokenize import word_tokenize
from nltk import pos_tag
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.preprocessing import LabelEncoder
from collections import defaultdict
from nltk.corpus import wordnet as wn
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn import model_selection, naive_bayes, svm
from sklearn.metrics import accuracy_score

np.random.seed(500)
Corpus = pd.read_csv(r"https://raw.githubusercontent.com/Gunjitbedi/Text-
Classification/master/corpus.csv",encoding='latin-1')
Corpus['text'].dropna(inplace=True)
Corpus['text'] = [entry.lower() for entry in Corpus['text']]

print(Corpus['text'])

import nltk
nltk.download('punkt')
Corpus['text']= [word_tokenize(entry) for entry in Corpus['text']]

print(Corpus['text'])

import nltk
nltk.download('wordnet')

tag_map = defaultdict(lambda : wn.NOUN)
tag_map['J'] = wn.ADJ
tag_map['V'] = wn.VERB
tag_map['R'] = wn.ADV

import nltk
nltk.download('averaged_perceptron_tagger')

import nltk
nltk.download('stopwords')

```

```

for index,entry in enumerate(Corpus['text']):
    Final_words = []
    word_Lemmatized = WordNetLemmatizer()
    for word, tag in pos_tag(entry):
        if word not in stopwords.words('english') and word.isalpha():
            word_Final = word_Lemmatized.lemmatize(word,tag_map[tag[0]])
            Final_words.append(word_Final)
    Corpus.loc[index,'text_final'] = str(Final_words)

print(Corpus['text_final'])

Train_X, Test_X, Train_Y, Test_Y =
model_selection.train_test_split(Corpus['text_final'],Corpus['label'],test_size=0.3)

Encoder = LabelEncoder()
Train_Y = Encoder.fit_transform(Train_Y)
Test_Y = Encoder.fit_transform(Test_Y)

Tfidf_vect = TfidfVectorizer(max_features=5000)
Tfidf_vect.fit(Corpus['text_final'])
Train_X_Tfidf = Tfidf_vect.transform(Train_X)
Test_X_Tfidf = Tfidf_vect.transform(Test_X)

print(Tfidf_vect.vocabulary_)

print(Train_X_Tfidf)
SVM = svm.SVC(C=1.0, kernel='linear', degree=3, gamma='auto')
SVM.fit(Train_X_Tfidf,Train_Y)
predictions_SVM = SVM.predict(Test_X_Tfidf)
print("SVM Accuracy Score -> ",accuracy_score(predictions_SVM, Test_Y)*100)

```

## **OUTPUT:**

```
SVM Accuracy Score -> 84.6
```

## **RESULT:**

Support Vector Machine is trained and tested.