INDEX

Name : P.S.ABISHEK

Subject : Artifial Intelligence

Std : IIIyr Sec : A Roll No 220701010

School : Raja Lakshmi Engineering College.

S.No.	Date	Title	Page No. marks	Teacher's Sign / Remarks
1	24/7/24	Sample python programs Using google Colab	9	
2)	7/8/24	Domain: Sentiment Analysis	10	
3)	4/9/24	N Queens - Problem.	10	
4)	4/9/24	DFS	10	
5)	11/9/24	A* algorithm.	10	
6)	18/9/24	AO* Algorithm.	10	
7)	25/9/24	Decesion Tree - gender	10	
8)	7/10/24	KMeans	10	
9)	9/10/24	Artificial neural network	10	
10)	16/10/24	Introducing Prolog	10	
12)	23/10/24	Prolog - Family Tree	10	
13)	23/10/24	Water Jug Problem.	10	

completed

Ex: 1

SAMPLE PYTHON PROGRAMS USING GOOGLE COLAB

① finding Average:

program:
```
def avg(a,b):
    c=(a+b)/2
    return c
a = int(input("enter numb"))
b = int(input("enter numb"))
average = avg(a,b)
print("Average is:", average)
```

O/P:
enter numb: 10
enter numb: 20
Average is: 15

② finding MAximum:

program:
```
def max(arr):
    i = arr[0]
    for x in arr
        if x > i:
            i = x
    return i
arr = [5,9,3,7,2,10]
print("maximum element:", max(arr))
```

O/P:
maximum element : 10

6) Palindrome program:

```
def is_palindrome(s):
    s = s.replace(" ", "").lower()
    return s == s[::-1]

input_str = str(input("enter string"))
print(f"{input_str} is a palindrome ? {is_palindrome(input_str)}")
```

Op: enter string: malayalam

malayalam is a palindrome : True

7) Factorial:

program:

```
def fact(n):
    if n==0 or n==1:
        return 1
    else:
        return n * fact(n-1)

n = int(input("enter number:"))
print(f"factorial of {n} is {fact(n)}")
```

Op:

enter numb: 5

factorial of 5 is 120

6) Is prime

Program:

```
def is_prime (num):
    if num <= 1:
        return false
    elif num <= 3:
        return True
    elif num % 2 == 0 or num % 3 == 0 or
         num % 5 == 0 or num % 7 == 0 or
         num % 9 == 0
        return false
    return True
num = int (input (" enter numb"))
print (f" num is {isprime} (num)")
```

O/P:

```
enter numb : 23
23 is True
```

7) Count character

program:

```
def count_char (s):
    char_ Count = {}
    for char in s:
        char Count [char] += 1
    else
        char_count [char] = 1
    return char_count
str1 = str (input ("enter string"))
print (f" string Count {Count_char (str)}")
```

O/P:

```
enter string : hello
Count : {h : 1 , e : 1 , l : 2 , o : 1}
```

8) Leap Year:

program:

```
def is_leap(year):
    if year % 4 == 0:
        if year % 100 == 0:
            if year % 400 == 0:
                return True
            else
                return False
        else
            return True
    else
        return False

year = int(input("year:"))
print(f"{year} is leap year: {is_leap(year)}")
```

O/P:

year: 2024

2024 is leap year: True.

9) Perfect Square:

program:

```
import math

def is_perfect_sq(num):
    sqrt_num = math.isqrt(num)
    return sqrt_num * sqrt_num == num
```

Reverse Digits:

```
def Reverse_digit (number):
    num_str = str(number)
    reversed_str = num_str[::-1]
    reversed_num = int (num.reversed_str)
    return reversed_num

num = int (input ("enter digit")
x = Reverse_digit (num)
print (" original numb: ", num)
print (" Reversed numb: ", ~~Reversed_num~~ x)
```

O/P: enter digit : 0987654321
Original : 0987654321
Reversed : 123456789

odd or Even:

```
def e_o(num):
    if num%2 == 0:
        print ("even").
    else
        print ("odd")

num = int (input ("enter numb"))
e_o(num)
```

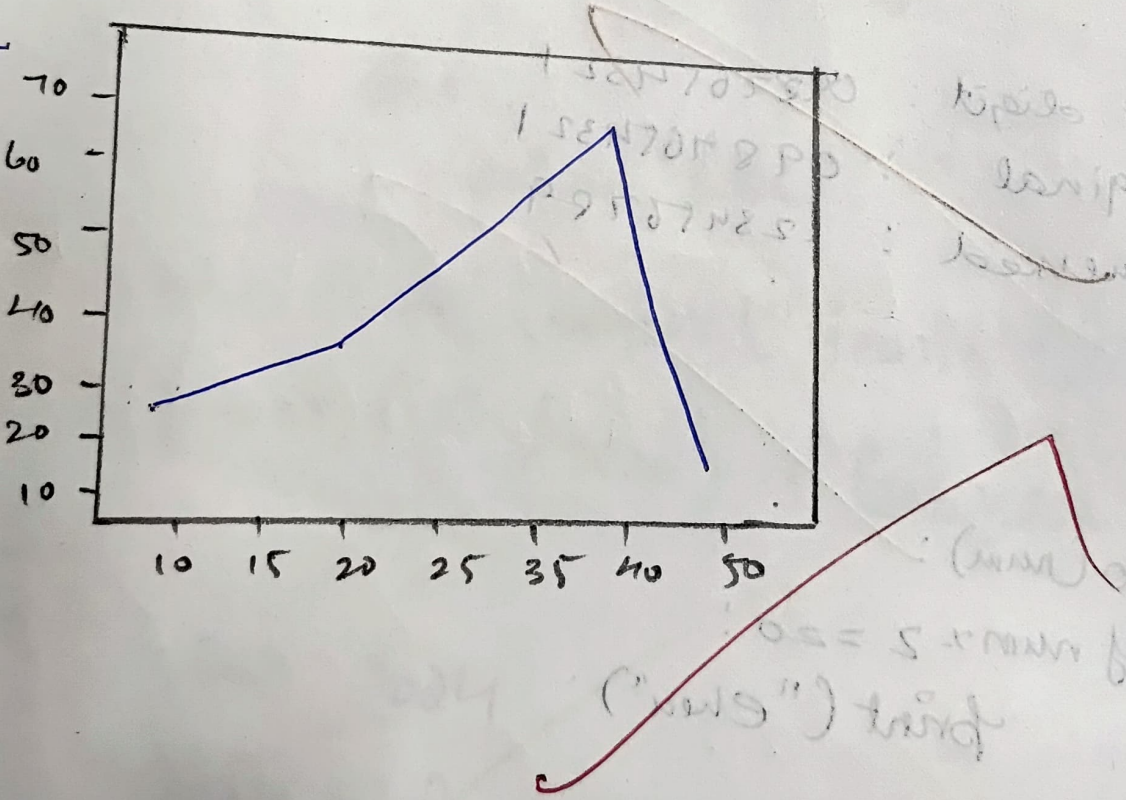
O/P: Enter numb: 23
odd

8) import matlib.pyplot as plt

```
x= [10,20,30,40,50]
y= [25,35 55,75,11]

plt.plot(x,y)
Plt.title('Simple line plot')
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.show().
```

Op:



N-Queens

Aim: To implement N queen problem.

Program:

```
def print_board(board):
    for row in board:
        print(' '.join('Q' if cell else '.' for cell in row))
    print()

def is_safe(board, row, col, N):
    for i in range(row):
        if board[i][col]:
            return False

    for i,j in zip(range(row,-1,-1), range(col,-1,-1)):
        if board[i][j]:
            return False

    for i,j in zip(range(row,-1,-1), range(col,N)):
        if board[i][j]:
            return False

    return True

def Solve_nqueens_util(board, row, N):
    if row >= N:
        print_board(board)
        return True

    res = False
    for col in range(N):
        if is_safe(board, row, col, N):
            board[row][col] = True
            res = Solve_nqueens_util(board, row+1, N)
                or board[row][col] = False

    return res
```

```
def Solve_nqueens(N):
    board = [[False]*N for _ in range(N)]
    if not Solve_nqueen_util(board, 0, N):
        print("Sol does not exist")

N = int(input("enter n"))
Solve_nqueens(N)
```

output:

```
enter n: 4
```

. Q . .
. . . Q
Q . . .
. . Q .

. . Q .
Q . . .
. . . Q
. Q . .

Result:

This n Queen problem Solved Successfully & executed.

DFS

Aim: To implement DFS using python.

Program:

```
def dfs_recursive(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start)
    for neighbour in graph[start]:
        if neighbour not in visited:
            dfs_recursive(graph, neighbour, visited)

graph = {'A':['B','C'], 'B':['A','D','E'], 'C':['A','F'], 'D':['B'],
         'E':['B','F'], 'F':['C','E']}

print("DFS Recursive")
dfs_recursive(graph, 'A')
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            print(vertex)
            visited.add(vertex)
            stack.extend(neighbour for neighbour in
                graph[vertex] if neighbour
                not in visited)
```

~~graph = {'~~

print("DFS Iterative"):
dfs_iterative(graph, 'A')

Op:

A B D E F C

Result:

Thus DFS implemented successfully.

11/9/2024

Aim: To implement A* algorithm.

program:

```
import heapq

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def astar(grid, start, end):
    open_list = []
    heapq.heappush(open_list, (0 + heuristic(start, end), 0, start))

    g_cost = {start: 0}
    came_from = {}
    directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

    while open_list:
        _, current_g, current = heapq.heappop(open_list)

        if current == end:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for dx, dy in directions:
            neighbour = (current[0] + dx, current[1] + dy)
            if (0 <= neighbour[0] < len(grid) and
                0 <= neighbour[1] < len(grid[0]) and
                grid[neighbour[0]][neighbour[1]] == 0):
                tentative_g = current_g + 1
                if neighbour not in g_cost or tentative_g < g_cost[neighbour]:
                    g_cost[neighbour] = tentative_g
                    g_cost[neighbour] = tentative_g + heuristic(neighbour, end)
```

```
            heapq.heappush(open_list, (f_cost, tentative_g, neighbour))
            came_from[neighbour] = current

    return None

grid = [ [0,0,0,0,0],
         [0,1,1,1,0],
         [0,0,0,1,0],
         [0,1,0,0,0],
         [0,0,0,0,0]
       ]

start = (0,0)
end = (4,4)

path = astar(grid, start, end)
print("path found", path)
```

O/P:

path found: [(0,0), (0,1), (0,2), (0,3), (0,4), (1,4), (2,4), (3,4), (4,4)]

Result:

Thus the A* algorithm implemented successfully

15/9/24

Aim: To implement AO* algorithm.

Program:

```
class graph:
    def __init__(self, graph, heuristic):
        self.graph = graph
        self.heuristic = heuristic
        self.solution = {}

    def ao_star(self, node):
        Print(f"Expanding: {node}")
        if node not in self.graph or not self.graph[node]:
            return

        children = self.graph[node]
        best_path = None
        min_cost = float('inf')

        for group in children:
            cost = sum(self.heuristic[child] for child in group)
            if cost < min_cost:
                min_cost = cost
                best_path = group

        self.solution[node] = best_path
        print(f"Best path for {node}: {best_path}
              with cost {min_cost}")

        for child in best_path:
            self.ao_star(child)

    def get_solution(self):
        return self.solution

Graph = {'A': [['B','C'], ['D']], 'B': [['E']], 'C': [['G']],
         'D': [['H']], 'E': [], 'G': [], 'H': []}
```

```
heuristic = {'A': 0, 'B': 1, 'C': 2, 'D': 4, 'E': 1, 'G': 3, 'H': 5}
graph_obj = Graph(graph, heuristic)
graph_obj.ao_star('A')
Solution = graph_obj.get_Solution()
print("Solution", Solution)
```

output:

```
Expanding: A
Best path for A: ['B', 'C']
Expanding: B
Best path for B: ['E']
Expanding: E
Expanding: C
Best path C: ['G']
Expanding: G
Solution: {A: [B, C], 'B': ['E'], 'C': ['G']}
```

Result:

Thus AO* algorithm implemented Successfully.

Implementation of decision Tree classification Techniques

Aim: To implement a decision tree classification technique for gender classification using python.

Program:

```
from sklearn.tree import DecisionTreeClassifier
import numpy as np

x = np.array ([[170,65,42],[180,75,44],[160,50,38],
               [175,70,43],[165,55,39],[180,80,45]])

y = np.array ([0,1,0,1,0,1])
clf = DecisionTreeClassifier()
clf.fit(x,y)

new_data = np.array([[172,68,43]])
prediction = clf.predict(new_data)
print ("predicted gender:", "Male" if prediction[0]
                                 ==1 else
                                 "female")
```

Output: Predicted gender : Male.

Result: Thus implementation of decision Tree classification implemented Successfully.

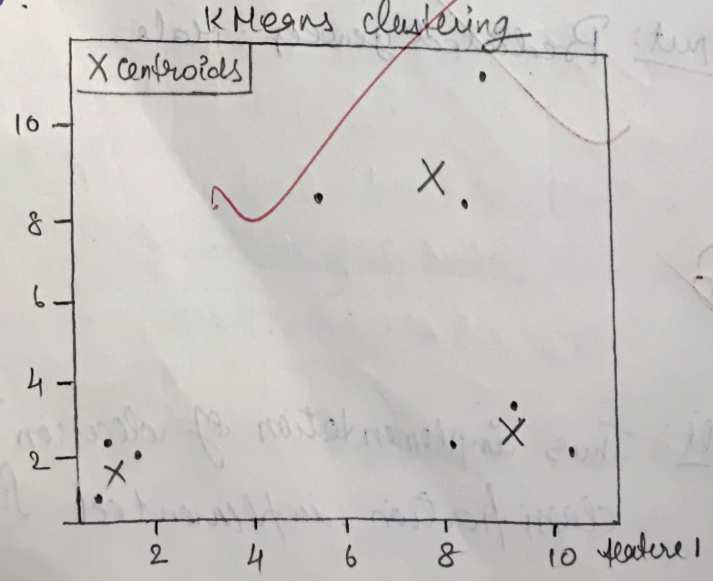
Implementation of Clustering Techniques K-Means

Aim: To implement a K Means clustering technique using python language.

Program:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans.
X = np.array ([[1,2],[1.5,1.8],[5,8],[8,8],[1,0.6],
               [9,11],[8,2],[10,2],[9,3]])
Kmeans = KMeans (n_clusters = 3)
Kmeans.fit(X)
Centroids = Kmeans.cluster_centers_
Labels = Kmeans.labels_
plt.scatter (X[:,0], X[:,1], c = labels, cmap = 'viridis', marker='o')
Plt.Scatter (centroids[:,0], centroids[:,1], c='red', s = 200,
             marker='x', label='centroids')
Plt.title('KMeans clustering')
plt.xlabel ('feature 1')
Plt.ylabel ('feature 2')
plt.legend()
plt.show().
```

Output:



Result: Thus implemented successfully & verified.

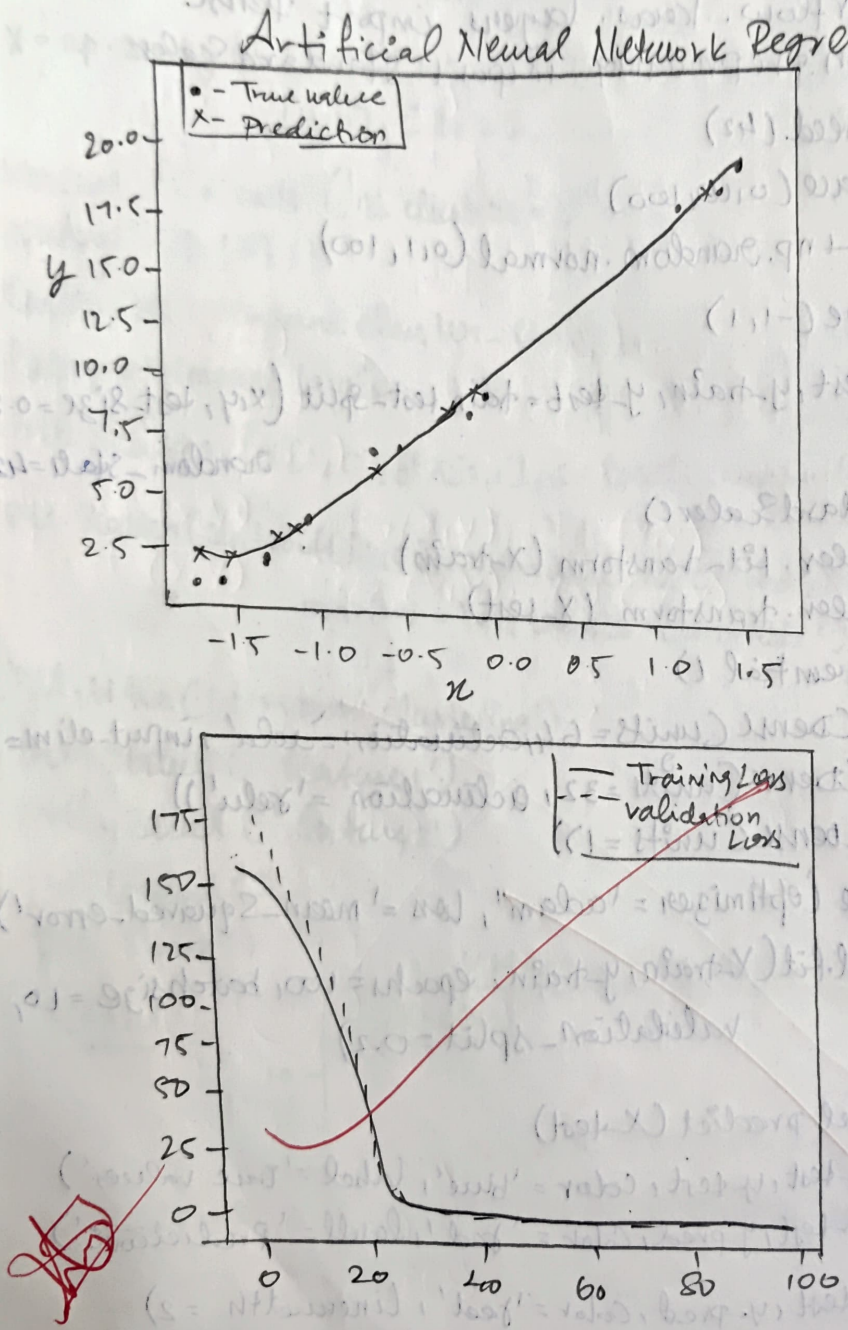
Aim: To implementing Artificial neutral for an application in Regression using python.

program:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from sklearn.preprocessing import StandardScaler
np.random.seed(42)
x = np.linspace(0,10,100)
y = 2*x+1+np.random.normal(0,1,100)
X = x.reshape(-1,1)
x_train, x_test, y_train, y_test = train_test_split(x,y, test_size=0.2,
                                   random_state=42)
Scaler = StandardScaler()
X_train = Scaler.fit_transform(X_train)
X_test = Scaler.transform(X_test)
model = Sequential()
model.add(Dense(units=64, activation='relu', input_dim=1))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(units=1))
model.compile(optimizer='adam', loss='mean_squared_error')
history = model.fit(X_train, y_train, epochs=100, batchsize=10,
                    validation_split=0.2)
y_pred = model.predict(X_test)
plt.scatter(X_test, y_test, color='blue', label='True values')
plt.scatter(X_test, y_pred, color='red', label='Predictions')
plt.plot(X_test, y_pred, color='red', linewidth=2)
plt.title('Artificial neutral Network Regression')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

```
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

Output:



Result:

Thus the artificial neural network for application in Regression implemented successfully.

Introducing To Prolog

Aim: To learn PROLOG terminologies and write basic program.

Terminologies:

1) **Atomic Terms:** Atomic terms are usually, string made up of lower and uppercase letters, digits and the underscore, starting with a lowercase letter.
ex: dog
abc_C_321

2) **Variable:** variable are string of letter, digits, underscore, starting with a capital letter or an underscore.
ex: Dog

3) **Compound Terms:** Compound terms are made up of a prolog atom and a number of agreement (PROLOG terms, i.e, atoms, number variable, or other Compound terms) enclosed in parenthesis & separated by Comma.
ex: is_bigger (elephant, X)

4) **Facts:** A fact is a predicate followed by a dot
eg: bigger_animal (whale)

5) **Rules:** A rule consist of a head (a predicate) & a body
eg: is_small (X, Y) :- is_bigger (Y, X)

Source Code:

KB1:

```
woman(mia)
woman(jody)
woman(yolanda)
playairguitar(jody)
party
Query1 :?- woman(mia)
Query2 :?- playairguitar(mia)
```

Output:

```
?- woman(mia)
true
? playairguitar(mia)
false.
```

KB2

happy(yolanda)
listen2music(mia)
listen2music(yolanda):-happy(yolanda)
playAirGuitar(mia):-listen2music(mia)
playAirGuitar(Yolanda):-listen2music(yolanda)

Output:

?-playAirGuitar(mia)
true
?-playAirGuitar(yolanda)
true

KB3:

likes(dan, Sally)
likes(Sally, dan)
likes(John, britney)
married(X,Y):-likes(X,Y), likes(Y,X)
friend(X,Y):-likes(X,Y), likes(Y,X)

Output:

?-likes(dan,X)
X=Sally
?-Married(dan,Sally)
true

KB4:

food(burger)
food(Sandwich)
food(pizza)
lunch(Sandwich)
dinner(pizza)
meal(X):-food(X)

Output:

?-food(pizza)
true
?-meal(X), lunch(X)
X=Sandwich
?-dinner(Sandwich)
false.

KBS:
```
owns(jack, car(bmw)).
owns(john, car(chevy))
owns(olivia, car(civic))
sedan(car(bmw))
sedan(car(civic))
truck(car(chevy))
```

Output:

```
?- owns(john, X)
X = car(chevy)
?- own(john, -)
true.
```

Result: Thus the experiment to learn about PROLOG and execute basic programs was done Sucessfully.

PROLOG - FAMILY TREE

Aim: To develop a family Tree program using PROLOG with all possible facts, rules, Queries.

Source Code:

Knowledge base:

```
/*FACTS ::*/
male(peter)
male(john)
male(chris)
male(kevin)

female(betty)
female(jeny)
female(lisa)
female(helen)

parentOf(chris,peter)
parentof(chris,betty)
parentof(helen,peter)
parentof(helen,betty)
parentof(kevin,lisa)
parentof(jeny,john)
parentof(jeny,helen)

/RULES*/
/*Son,parent
*Son,grandparent*/

father(X,Y):-male(Y),parentof(X,Y)
mother(X,Y):-female(Y),parentof(X,Y)
Grandfather(X,Y):-male(Y),parentof(X,Y),parentof(Z,Y)
Grandmother(X,Y):-female(Y),parentof(X,Z),parentof(Z,Y)
brother(X,Y):-male(Y),father(X,Z),father(Y,W),Z==W
brother(X,Y):-female(Y),father(X,Z),father(Y,W),Z==W
```

Output:

? parent of (kevin, x)
x = chris

? father (x, chris)
x = kevin

? sister (x, chris)
false.

Result:

Thus the PROLOG program to implement and execute family tree was Successfully Completely.

Depth First Search

WATER JUG PROBLEM

Program:

```
def dfs(x,y,target,visited,path,a,b):
    if b==target:
        Print("Solution found:")
        for step in path:
            Print(step)
        return True
    if (a,b) in visited:
        return False
    visited.add((a,b))
    path.append(f"({a},{b})")
    if dfs(x,y,target,visited,path,x,b):
        return True
    if dfs(x,y,target,visited,path,a,y):
        return True.
    if dfs(x,y,target,visited,path,0,b):
        return True
    if dfs(x,y,target,visited,path,a,0):
        return True
    if a+b <=y
        if dfs(x,y,target,visited,path,0,a+b):
        return True
    else:
        if dfs(x,y,target,visited,path,a-(y-b),y):
            return True
    if a+b <=x:
        if dfs(x,y,target,visited,path,a+b,0):
            return True.
```

```
        else:
            if dfs(x,y,target,visited,path,a,b-(x-a)):
                return True

        path.pop()
        return False

def water_jug_dfs():
    x,y = 3,4
    target = 2
    visited = set()
    path = []
        if not dfs(x,y,target,visited,path,0,0):
            print("not Solution exist")

water_jug_dfs().
```

Output:

Solution found:

(0,0)
(3,0)
(3,3)
(4,2)
(0,2)
(2,0).

Result:

Thus the water jug problem solved with depth first Search Successfully.