COLLEGE OF APPLIED BUSINESS

Gangahiti, Chabahil

Department of Computer Science and Technology

LAB REPORT OF

ARTIFICIAL INTELLIGENCE (CSC 261)

Submitted by:

Biplaw Chaudhary (19108)

BSc. CSIT 4th Semester

College of Applied Business

Department of Science and Technology

Submitted to:

Mr. Tekendra Nath Yogi

College of Applied Business

Department of Science and Technology

LAB-1

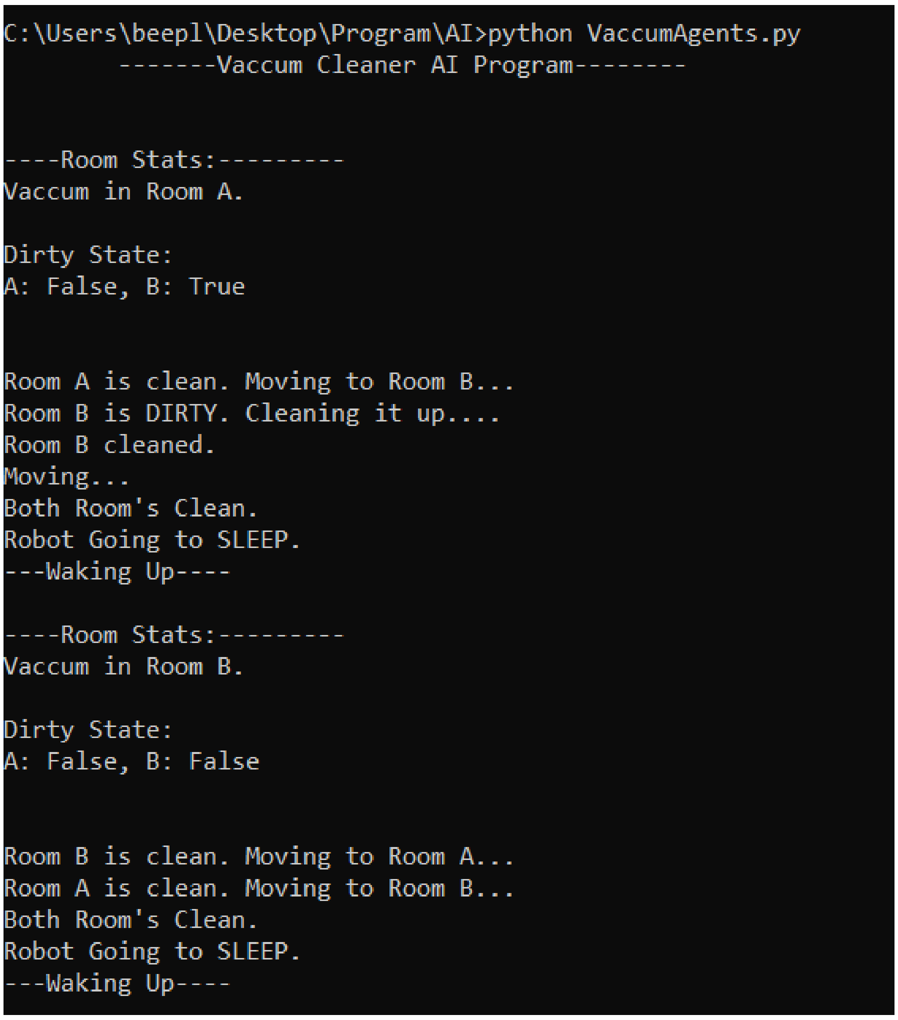
NAME

Write a basic python program to demonstrate Vacuum Cleaner problem.

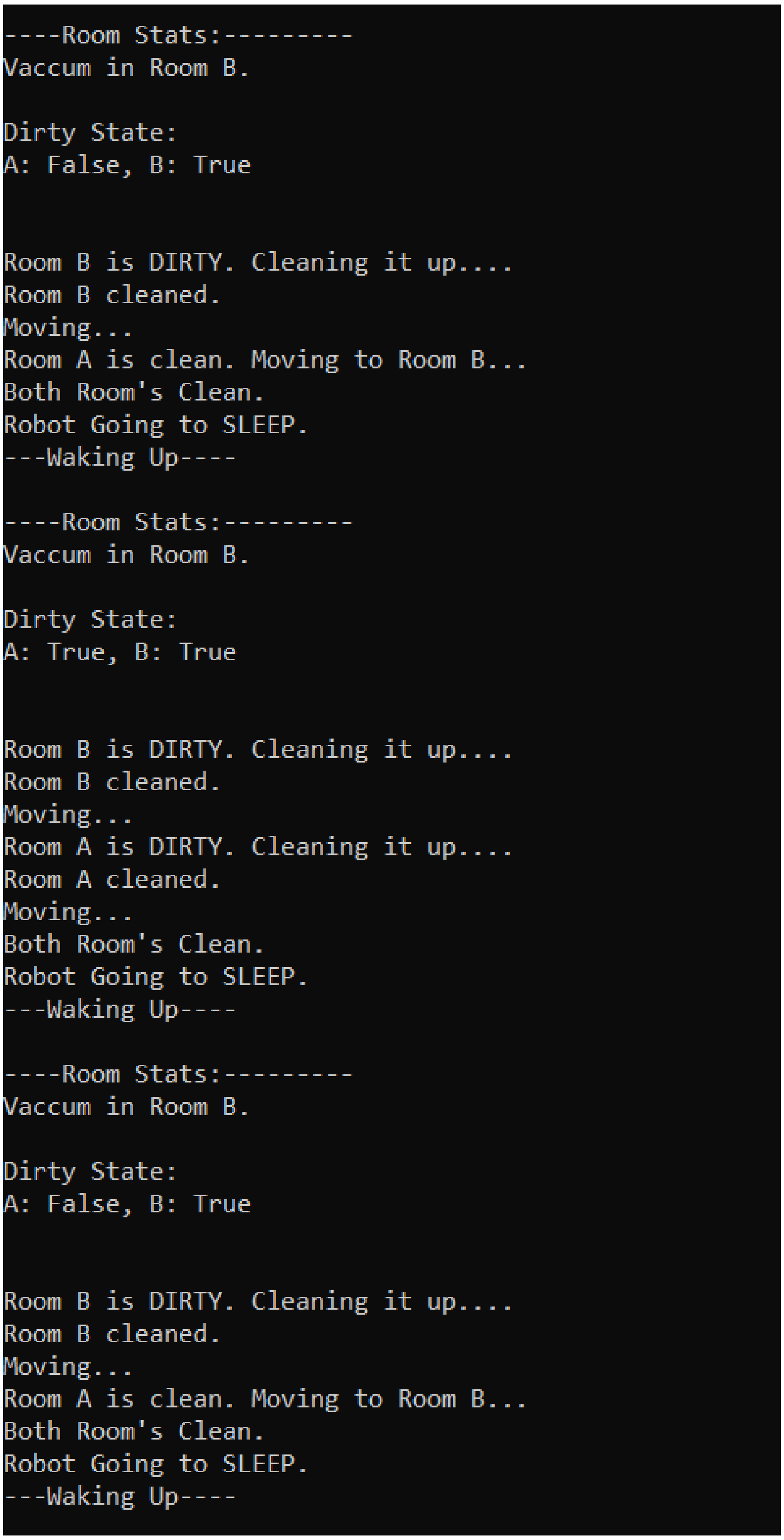
CODE

```
#Program to implement vaccum cleaner using different agents.
"""
Room A is 0
Room B is 1

"""

#---Imports---
from random import randint
from random import choice
from time import sleep

#----------------------------GLOBAL VARIABLES----------------------------------
#setting the room A and room B's state
env_dirty_state ={0:False, 1 :False } #inital setup for dictionary
#vaccum location
vaccum_location = 0

#------------------***-----FUNCTIONS-----****---------------------------------

#Create a random position for the vaccum cleaner at startup
def set_vaccum_loc():
    global vaccum_location
    #Setting the random initial vaccum location
    vaccum_location = randint(0,1)

#Function for generating random state for rooms
def rand_state_generation():
    global env_dirty_state
    #To create a random choice for dirty
    state=[True, False]
    #Setting the new choices
    env_dirty_state[0] = choice(state)  # Creating a random state for room A
    env_dirty_state[1] = choice(state) #Creating a random state for room B

#Function to clean the rooms
def clean_rooms(room):
    #Code to clean the room A
    if room==0:
        print("Room A is DIRTY. Cleaning it up....")
        sleep(3)
        env_dirty_state[room] = False
        print("Room A cleaned.\nMoving...")
        sleep(3)

    #Code to clean the room B
    elif room == 1:
        print("Room B is DIRTY. Cleaning it up....")
        sleep(3)
        env_dirty_state[room] = False
        print("Room B cleaned.\nMoving...")
        sleep(3)

#----------------------AGENTS-------------------------------------------------------------
#--The simple reflex agent--
def simple_reflex_agent():
    global env_dirty_state, vaccum_location
    name_of_room = {0: 'A', 1: 'B'}
    iter_count=0

    set_vaccum_loc()

    rand_state_generation()
    #List to see if both rooms are cleaned
    cleaned_room = 0

    #Simulation of sleeping
    sleep(2)

    #Simulates a continuous run of the cleaner machine.
    while True:

        #Creating a break condition for visualization
        if iter_count>10 and cleaned_room==2:
            print("Both room cleaned.")
            break

        iter_count+=1

        #Checking if both rooms are cleaned
        if cleaned_room == 2:
            cleaned_room = 0 #Set cleaned rooms to zero
            print("Both Room's Clean. ")
            print("Robot Going to SLEEP. ")
            sleep(5)
            print("---Waking Up----\n")
            #Generate new random location and data
            set_vaccum_loc()
            rand_state_generation()

        #Display the status of iteration, rooms
        if cleaned_room==0:
            print(f"----Room Stats:---------")
            print(f"Vaccum in Room {name_of_room[vaccum_location]}.\n")
            print(f"Dirty State: \nA: {env_dirty_state[0]}, B: {env_dirty_state[1]}\n\n")
            sleep(2)

        #If vaccum is in room A
        if vaccum_location==0:
            #If room A is dirty
            if env_dirty_state[vaccum_location]:
                clean_rooms(vaccum_location) #Clean the room
            #if room A is clean
            else:
                if cleaned_room !=2:
                    print("Room A is clean. Moving to Room B...")

            vaccum_location= 1  #Set the vaccum location to next room
              cleaned_room += 1  #Increment the cleaned room as Room A is now cleaned or is
clean.

        #If vaccum is in room B
        elif vaccum_location==1:
            #If room B is dirty
            if env_dirty_state[vaccum_location]:
                clean_rooms(vaccum_location)
            #if room B is clean
            else:
                if cleaned_room !=2:
                    print("Room B is clean. Moving to Room A...")
            vaccum_location= 0  #Set the vaccum location to next room
           cleaned_room +=1 #Increment the cleaned room as Room B is now cleaned or is clean.


#--------Creating a function like main----------
def main():
    print("\t-------Vaccum Cleaner AI Program--------\n\n")

    simple_reflex_agent()

#Run only if this program is not imported into another program.
if __name__ != "main":
    main()
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python VaccumAgents.py
        -------Vaccum Cleaner AI Program--------


----Room Stats:---------
Vaccum in Room A.

Dirty State:
A: False, B: True


Room A is clean. Moving to Room B...
Room B is DIRTY. Cleaning it up....
Room B cleaned.
Moving...
Both Room's Clean.
Robot Going to SLEEP.
---Waking Up----

----Room Stats:---------
Vaccum in Room B.

Dirty State:
A: False, B: False


Room B is clean. Moving to Room A...
Room A is clean. Moving to Room B...
Both Room's Clean.
Robot Going to SLEEP.
---Waking Up----
```



```
----Room Stats:---------
Vaccum in Room B.

Dirty State:
A: False, B: True


Room B is DIRTY. Cleaning it up....
Room B cleaned.
Moving...
Room A is clean. Moving to Room B...
Both Room's Clean.
Robot Going to SLEEP.
---Waking Up----

----Room Stats:---------
Vaccum in Room B.

Dirty State:
A: True, B: True


Room B is DIRTY. Cleaning it up....
Room B cleaned.
Moving...
Room A is DIRTY. Cleaning it up....
Room A cleaned.
Moving...
Both Room's Clean.
Robot Going to SLEEP.
---Waking Up----

----Room Stats:---------
Vaccum in Room B.

Dirty State:
A: False, B: True


Room B is DIRTY. Cleaning it up....
Room B cleaned.
Moving...
Room A is clean. Moving to Room B...
Both Room's Clean.
Robot Going to SLEEP.
---Waking Up----
```

LAB-2

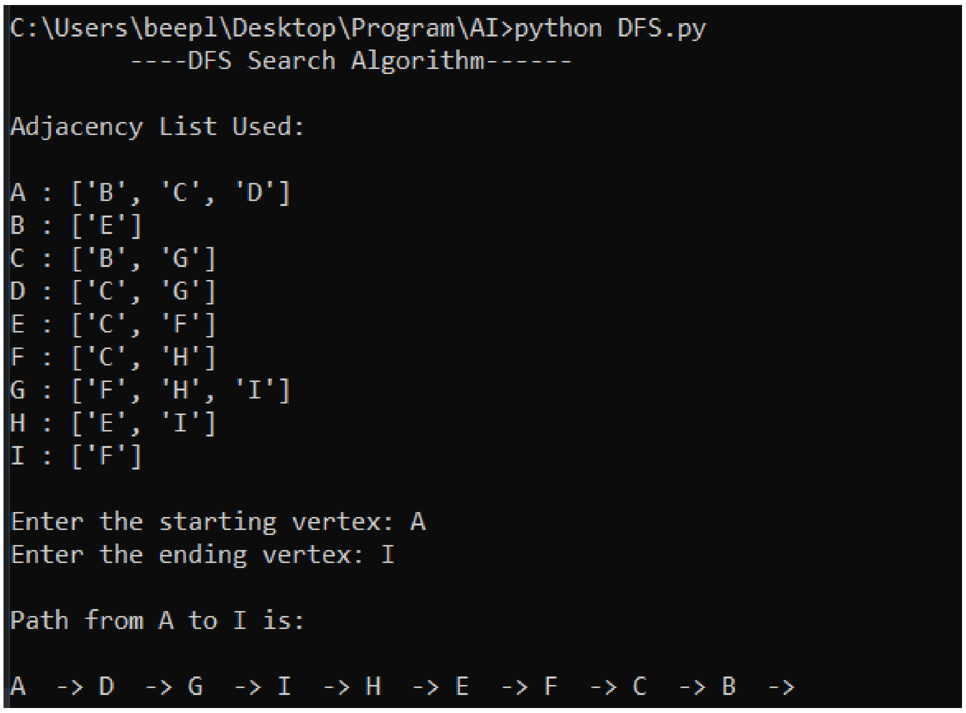
NAME

Write a basic program to implement Depth First Search Algorithm.

CODE

```
#Program to implement DFS search algorithm
#initalization of stack
stack=[]
final_path=[]

#Our graph represented in form of adjacency list
adj_list ={
    'A': ['B','C','D'],
    'B': ['E'],
    'C':['B','G'],
    'D': ['C','G'],
    'E': ['C','F'],
    'F':['C','H'],
    'G':['F','H','I'],
    'H':['E','I'],
    'I':['F']
}
'''
If status is 1 -> ready state
If status is 2 -> the node is enqueues and is said to be in waiting state.
If status is 3 -> then then node is dequeued.
'''
status={'A': 1, 'B': 1, 'C':1, 'D': 1, 'E': 1, 'F':1, 'G':1, 'H':1, 'I':1}

print("\t----DFS Search Algorithm------\n")

#Display the adjacency list that is being used.
print("Adjacency List Used: \n")
for key,value in adj_list.items():
    print(key,':', value)

start=input("\nEnter the starting vertex: ")
dest =input("Enter the ending vertex: ")

#Pushing the first node onto the stack
stack.append(start)
#Setting it's status to 2 as it now goes into ready state
status[start] = 2

while True:
    #Pop the node on the top of stack
    node = stack.pop()
    #Add the popped node on the final list
    final_path.append(node)
    #Set the status of popped node to 3
    status[node]=3

    #Get the adjacant nodes of the popped nodes from adjacant list
    adj_nodes = adj_list[node]
    #For each node that is adjacant to the popped node
    for node in adj_nodes:
        #If the node is in ready state or untouched:
        if status[node]==1:
            #Push that node onto the stack
            stack.append(node)
            #Set the status of that node to waiting (2)
            status[node] = 2

    #If the stack is empty then exit the loop
    if not stack:
        break

print(f"\nPath from {start} to {dest} is: \n",)
for each in final_path:
    print(each, ' -> ', end='')
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python DFS.py
        ----DFS Search Algorithm------

Adjacency List Used:

A : ['B', 'C', 'D']
B : ['E']
C : ['B', 'G']
D : ['C', 'G']
E : ['C', 'F']
F : ['C', 'H']
G : ['F', 'H', 'I']
H : ['E', 'I']
I : ['F']

Enter the starting vertex: A
Enter the ending vertex: I

Path from A to I is:

A  -> D  -> G  -> I  -> H  -> E  -> F  -> C  -> B  ->
```

LAB-3

NAME

Write a basic program to implement Breadth First Search Algorithm.

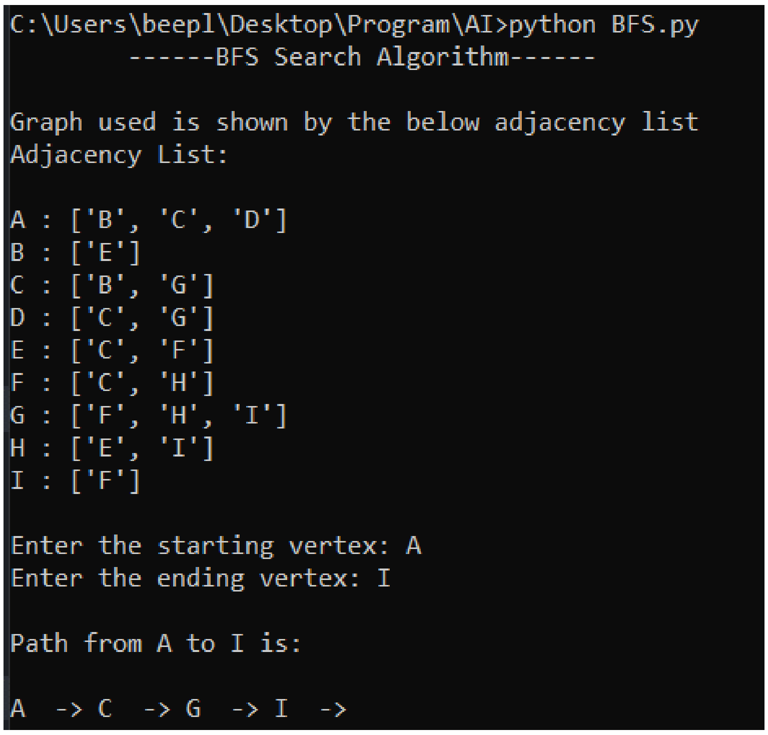
CODE

```
#Program to implement the BFS search algorithm
#initializing the queues
front = -1
rear=-1
#List to hold the nodes that are to be processed.
queue=[]
#List to keep track of the origin of each node
orig =[]

#Our graph represented in form of adjacency list
adj_list ={
    'A': ['B','C','D'],
    'B': ['E'],
    'C':['B','G'],
    'D': ['C','G'],
    'E': ['C','F'],
    'F':['C','H'],
    'G':['F','H','I'],
    'H':['E','I'],
    'I':['F']
}
'''
If status is 1 -> ready state
If status is 2 -> the node is enqueues and is said to be in waiting state.
If status is 3 -> then then node is dequeued.
'''
status={'A': 1, 'B': 1, 'C':1, 'D': 1, 'E': 1, 'F':1, 'G':1, 'H':1, 'I':1}

#-------------------------------
print("\t------BFS Search Algorithm------\n")

#Display the adjacency list that is being used.
print("Graph used is shown by the below adjacency list")
print("Adjacency List: \n")
for key,value in adj_list.items():
    print(key,':', value)

start=input("\nEnter the starting vertex: ")
dest =input("Enter the ending vertex: ")
```

```
#Enquiying the first node in the list
queue.append(start)
orig.append( '0')

front+=1
rear+=1

while True:
    #Dequeing the first node
    selected_node = queue[front]
    #Set it's status to 3
    status[selected_node] = 3
    front+=1

    #Selection of the adjacant nodes of the selected node
    adj_nodes = adj_list[selected_node]

   #For each node in the list of adjacant nodes
    for node in adj_nodes:
        #If the status of node is 1 i.e. the node is untouched then:
        if status[node]==1:
            #Enqueue the nodes and its origin in the queue
            queue.append(node)
            orig.append(selected_node)
            #Set the status of the node to 2 i.e. waiting state.
            status[node]=2
            #Increment the rear index
            rear+=1
    #If the rear item on the queue is our destination then break the loop
    if queue[rear]==dest:
        break

#-------Backtracking form destination using original list ----------------------
#List to store the final path results
final_path = list()

#This begins from the rearmost item in the queue
node = queue[rear] #Stores the nodes in our path
parent = orig[rear] #Stores the parent of the nodes in our path

#Append the destination node in the queue.
final_path.append(node)
]
```

```
#Continue adding in the list
while True:
    #Get the index of the parent from the queue.
    index = queue.index(parent)

    # Set the node in our path to that inat index
    node = queue[index]
    #Set the new parent
    parent= orig[index]
    #Add that node in the final path list
    final_path.append(node)

    #If we reach the starting node then break the loop
    if (parent == '0'):
        break

#Reverse the list so that it goes from source to destination
final_path = final_path[::-1]

#Displaying the result
print(f"\nPath from {start} to {dest} is: \n",)
for each in final_path:
    print(each, ' -> ', end='')
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python BFS.py
        ------BFS Search Algorithm------

Graph used is shown by the below adjacency list
Adjacency List:

A : ['B', 'C', 'D']
B : ['E']
C : ['B', 'G']
D : ['C', 'G']
E : ['C', 'F']
F : ['C', 'H']
G : ['F', 'H', 'I']
H : ['E', 'I']
I : ['F']

Enter the starting vertex: A
Enter the ending vertex: I

Path from A to I is:

A  -> C  -> G  -> I  ->
```

LAB-4

NAME

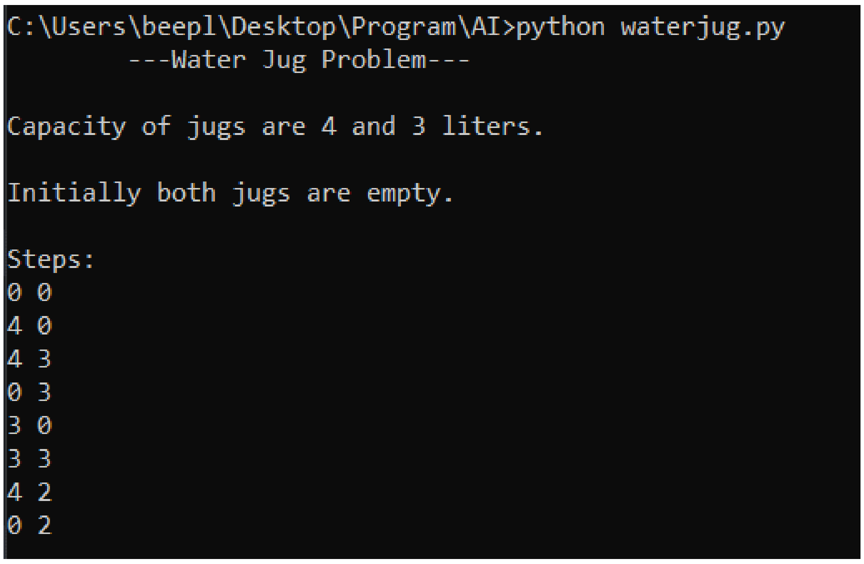
Write a program to implement Water Jug Problem.

CODE

```
#Program to implement the water jug problem in python
from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
processed = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJug(amt1, amt2):

    # Checks for our goal and
    # returns true if achieved.
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True

    # Checks if we have already visited the
    # combination or not. If not, then it proceeds further.
    if processed[(amt1, amt2)] == False:
        print(amt1, amt2)

        # Changes the boolean value of
        # the combination as it is visited.
        processed[(amt1, amt2)] = True

        # Check for all the 6 possibilities and
        # see if a solution is found in any one of them.
        return (waterJug(0, amt2) or
                waterJug(amt1, 0) or
                waterJug(jug1, amt2) or
                waterJug(amt1, jug2) or
```

```
            waterJug(amt1 + min(amt2, (jug1-amt1)),
            amt2 - min(amt2, (jug1-amt1))) or
            waterJug(amt1 - min(amt1, (jug2-amt2)),
            amt2 + min(amt1, (jug2-amt2))))

    # Return False if the combination is
    # already visited to avoid repetition otherwise
    # recursion will enter an infinite loop.
    else:
        return False

print("\t---Water Jug Problem---\n")
print("Capacity of jugs are 4 and 3 liters. \n")
print("Initially both jugs are empty.\n")

print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJug(0, 0)
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python waterjug.py
        ---Water Jug Problem---

Capacity of jugs are 4 and 3 liters.

Initially both jugs are empty.

Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

LAB-5

NAME

Write a program to demonstrate 8-puzzle problem with heuristics.

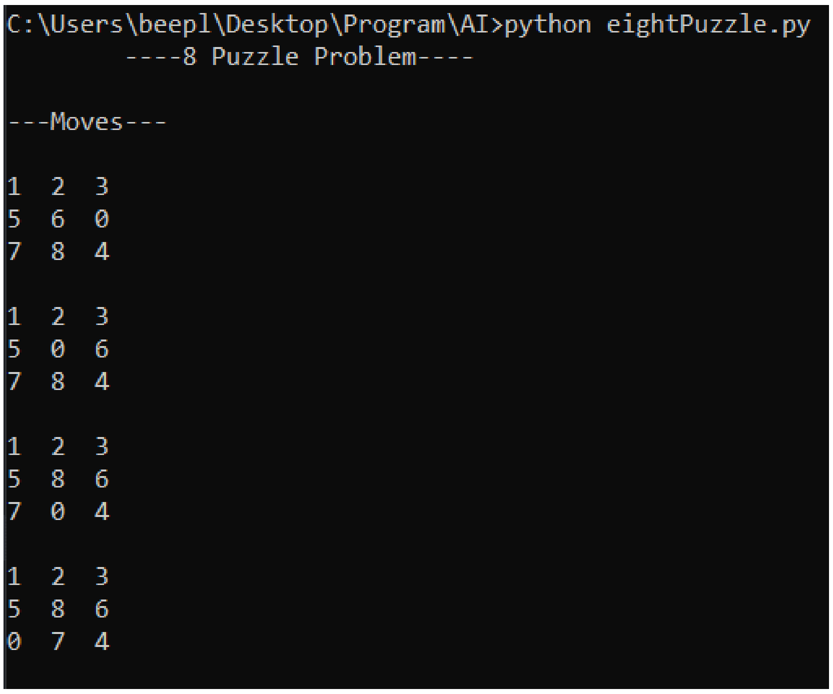
CODE

```
# Importing the 'copy' for deepcopy method
import copy

# Importing the heap methods from the python
# library for the Priority Queue
from heapq import heappush, heappop

# This particular var can be changed to transform
# the program from 8 puzzle(n=3) into 15
# puzzle(n=4) and so on ...
n = 3

# bottom, left, top, right
rows = [ 1, 0, -1, 0 ]
cols = [ 0, -1, 0, 1 ]

# creating a class for the Priority Queue
class priorityQueue:

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

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

    # funct to remove the element that is minimum, from the Priority Queue
    def pop(self):
        return heappop(self.heap)

    # funct to check if the Queue is empty or not
    def empty(self):
        if not self.heap:
            return True
        else:
            return False

# structure of the node
class nodes:
    def __init__(self, parent, mats, empty_tile_posi,
                costs, levels):

        # This will store the parent node to the current node And helps in tracing the
        # path when the solution is visible
        self.parent = parent

        # Useful for Storing the matrix
        self.mats = mats

        # useful for Storing the position where the
        # empty space tile is already existing in the matrix
        self.empty_tile_posi = empty_tile_posi

        # Store no. of misplaced tiles
        self.costs = costs

        # Store no. of moves so far
        self.levels = levels

    # This func is used in order to form the priority queue based on the costs var of objects
    def __lt__(self, nxt):
        return self.costs < nxt.costs

# method to calc. the no. of misplaced tiles, that is the no. of non-blank tiles not in their final
#posi
def calculateCosts(mats, final) -> int:

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

    return count

def newNodes(mats, empty_tile_posi, new_empty_tile_posi,
            levels, parent, final) -> nodes:

    # Copying data from the parent matrixes to the present matrixes
    new_mats = copy.deepcopy(mats)

    # Moving the tile by 1 position
    x1 = empty_tile_posi[0]
    y1 = empty_tile_posi[1]
    x2 = new_empty_tile_posi[0]
    y2 = new_empty_tile_posi[1]
    new_mats[x1][y1], new_mats[x2][y2] = new_mats[x2][y2], new_mats[x1][y1]

    # Setting the no. of misplaced tiles
    costs = calculateCosts(new_mats, final)

    new_nodes = nodes(parent, new_mats, new_empty_tile_posi,
                costs, levels)
    return new_nodes

# func to print the N by N matrix
def printMatsrix(mats):

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

        print()

# func to know if (x, y) is a valid or invalid
# matrix coordinates
def isSafe(x, y):

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

# Printing the path from the root node to the final node
def printPath(root):

    if root == None:
        return

    printPath(root.parent)
    printMatsrix(root.mats)
    print()

# method for solving N*N - 1 puzzle algo
# by utilizing the Branch and Bound technique. empty_tile_posi is
# the blank tile position initially.
def solve(initial, empty_tile_posi, final):

    # Creating a priority queue for storing the live
    # nodes of the search tree
    pq = priorityQueue()

    # Creating the root node
```

```
costs = calculateCosts(initial, final)
root = nodes(None, initial,
            empty_tile_posi, costs, 0)

# Adding root to the list of live nodes
pq.push(root)

# Discovering a live node with min. costs, and adding its children to the list of live
# nodes and finally deleting it from the list.
while not pq.empty():

    # Finding a live node with min. estimatsed costs and deleting it form the list of the
    # live nodes
    minimum = pq.pop()

    # If the min. is ans node
    if minimum.costs == 0:

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

    # Generating all feasible children
    for i in range(n):
        new_tile_posi = [
            minimum.empty_tile_posi[0] + rows[i],
            minimum.empty_tile_posi[1] + cols[i], ]

        if isSafe(new_tile_posi[0], new_tile_posi[1]):

            # Creating a child node
            child = newNodes(minimum.mats,
                            minimum.empty_tile_posi,
                            new_tile_posi,
                            minimum.levels + 1,
                            minimum, final,)

            # Adding the child to the list of live nodes
            pq.push(child)
```

```
# Main Code

# Initial configuration
# Value 0 is taken here as an empty space
initial = [ [ 1, 2, 3 ],
            [ 5, 6, 0 ],
            [ 7, 8, 4 ] ]

# Final configuration that can be solved
# Value 0 is taken as an empty space
final = [ [ 1, 2, 3 ],
        [ 5, 8, 6 ],
        [ 0, 7, 4 ] ]

# Blank tile coordinates in the
# initial configuration
empty_tile_posi = [ 1, 2 ]
# Method call for solving the puzzle
print("\t----8 Puzzle Problem----\n")
print("---Moves---\n")
solve(initial, empty_tile_posi, final)
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python eightPuzzle.py
        ----8 Puzzle Problem----

---Moves---

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
```

LAB-6

NAME

Write a program to demonstrate Hill Climbing Search Algorithm.

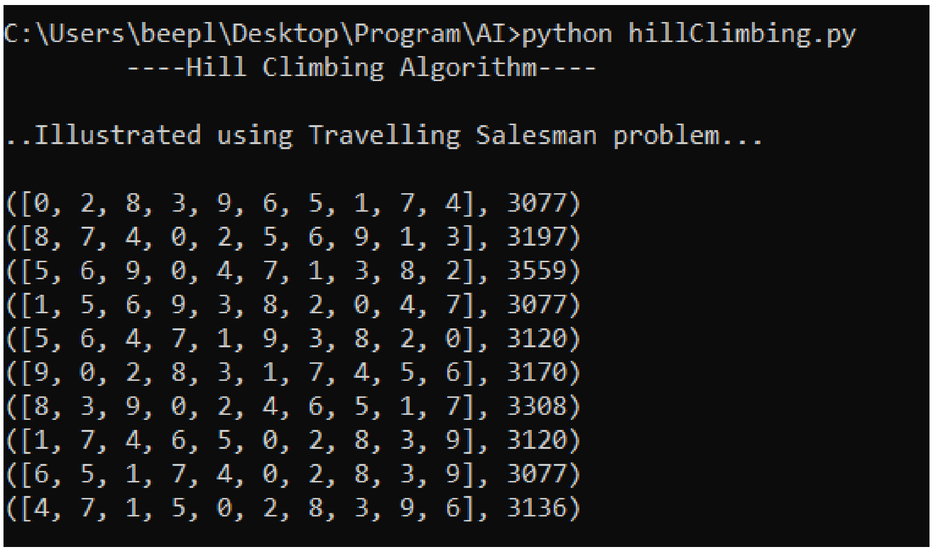
CODE

```
#Program to implement hill climbing algorithm.
from random import randint

#Random Solution generator
'''
For Hill climbing to work, it has to start with a random solution to our Travelling salesman
problem. first create a list of identifiers of all cities and from there on iteratively pick a city
from that list at random and add it to our solution.
'''
def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []

    for i in range(len(tsp)):
        randomCity = cities[randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)
    return solution

# function calculating the length of a route
'''
Since we want our Hill climber to find the shortest solution, we need a function calculating the
length of a specific solution.
'''
def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength

# function generating all neighbours of a solution
def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours
```

```
#function finding the best neighbour
def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength

def problemGenerator(nCities):
    tsp = []
    for i in range(nCities):
        distances = []
        for j in range(nCities):
            if j == i:
                distances.append(0)
            elif j < i:
                distances.append(tsp[j][i])
            else:
                distances.append(randint(10, 1000))
        tsp.append(distances)
    return tsp
```

```
def main():
    #Data for the Travelling Salesman Problem
    '''the distance from each city to itself is zero, and the distance from city A to
    city B is the same as the distance from city B to city A.
    Suppose we've four cities so, we get a list as:
    '''

    print("\t----Hill Climbing Algorithm----\n")
    print("..Illustrated using Travelling Salesman problem...\n")

    # print(hillClimbing(tsp))
    tsp = problemGenerator(10)
    for i in range(10):
        print(hillClimbing(tsp))


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

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python hillClimbing.py
        ----Hill Climbing Algorithm----

..Illustrated using Travelling Salesman problem...

([0, 2, 8, 3, 9, 6, 5, 1, 7, 4], 3077)
([8, 7, 4, 0, 2, 5, 6, 9, 1, 3], 3197)
([5, 6, 9, 0, 4, 7, 1, 3, 8, 2], 3559)
([1, 5, 6, 9, 3, 8, 2, 0, 4, 7], 3077)
([5, 6, 4, 7, 1, 9, 3, 8, 2, 0], 3120)
([9, 0, 2, 8, 3, 1, 7, 4, 5, 6], 3170)
([8, 3, 9, 0, 2, 4, 6, 5, 1, 7], 3308)
([1, 7, 4, 6, 5, 0, 2, 8, 3, 9], 3120)
([6, 5, 1, 7, 4, 0, 2, 8, 3, 9], 3077)
([4, 7, 1, 5, 0, 2, 8, 3, 9, 6], 3136)
```

LAB-7

NAME

Simulate a knowledge base for listed dataset and perform operations in Prolog as mentioned below.

Dataset

Tiger eats meat.
Cow eats vegetables.
Human eats meat.
Human eats vegetables.
Carnivorous eats meat.
Omnivorous eats vegetables and meat.

Logical Operations

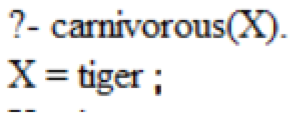
a) List out the animals that eats meat only.
b) List out the animals that easts vegetables only.
c) List out the animals that eats meat and vegetables.
d) List out the animals that eats meat.
e) List out the animals that eats vegetables.

CODE

```
eats(tiger,meat).
eats(cow, vegetables).
eats(human, meat).
eats(human, vegetables).
carnivorous(X) :- eats(X, meat).
omnivorous(X):- eats(X, meat), eats(X, vegetables).
hervivorous(X):- eats(X, vegetables).
```

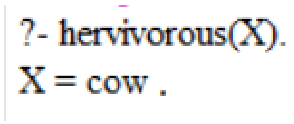
OUTPUT

a) List out the animals that eats meat only.



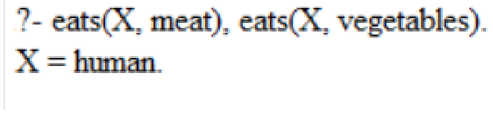
```
?- carnivorous(X).
X = tiger ;
```

b) List out the animals that easts vegetables only.



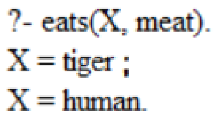
```
?- hervivorous(X).
X = cow .
```

c) List out the animals that eats meat and vegetables.



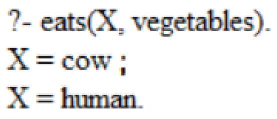
```
?- eats(X, meat), eats(X, vegetables).
X = human.
```

d) List out the animals that eats meat.



```
?- eats(X, meat).
X = tiger ;
X = human.
```

e) List out the animals that eats vegetables.



```
?- eats(X, vegetables).
X = cow ;
X = human.
```

LAB-8

NAME

Simulate a knowledge base for listed dataset and perform operations in Prolog as mentioned below.

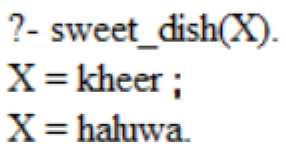
> ***Dataset***
>
> Kheer contains sugar.
> Haluwa contains sugar.
>
> Pickle contains salt.
> Daal contains salt.
>
> Sweet_Dish is a dish which contains sugar.
> Salt_Dish is a dish which contains salt.
>
> ***Logical Operations***
>
> a) List of sweet dishes.
> b) List of salt dishes.

CODE

```
contains(kheer, sugar).
contains(haluwa, sugar).
contains(pickle, salt).
contains(daal, salt).
sweet_dish(A) :- contains(A, sugar).
salt_dish(A) :- contains(A, salt).
```

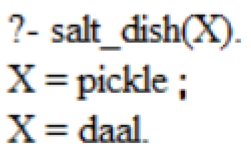
OUTPUT

a) List of sweet dishes.



```
?- sweet_dish(X).
X = kheer ;
X = haluwa.
```

b) List of salt dishes.



```
?- salt_dish(X).
X = pickle ;
X = daal.
```

LAB-9

NAME

Simulate a knowledge base for listed dataset and perform operations in Prolog as mentioned below.

Dataset

Banana tastes sweet.
Papaya tastes sweet.

Tamarind tastes sour.
Lapsi tastes sour.

Sweet_Fruit is a Fruit which tastes sweet .
Sour_Fruit is a Fruit which tastes sour.

Logical Operations

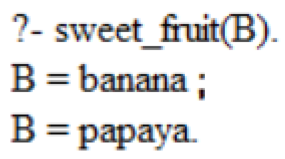
a) List of sweet fruits.
b) List of sour fruits.

CODE

```
tastes(banana, sweet).
tastes(papaya, sweet).
tastes(tamarind, sour).
tastes(lapsi, sour).
sweet_fruit(A) :- tastes(A, sweet).
sour_fruit(A) :- tastes(A, sour).
```

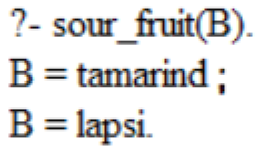
OUTPUT

a) List of sweet fruits.



```
?- sweet_fruit(B).
B = banana ;
B = papaya.
```

b) List of sour fruits.



```
?- sour_fruit(B).
B = tamarind ;
B = lapsi.
```

LAB-10

NAME

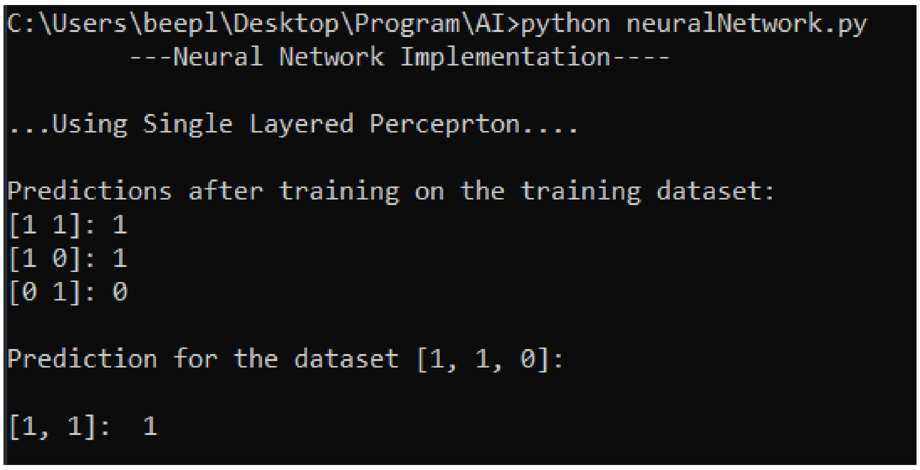
Write a program in a high level language to demonstrate a neural network to train a following dataset.

Inputs			Outputs
1	1	1	1
1	0	1	1
0	1	1	0
1	1	0	?

CODE

```
# Program to implement neural network for the given dataset
import numpy as np
from random import choice

print("\t---Neural Network Implementation----\n")
print("...Using Single Layered Perceprton....\n")
np.random.seed(2)

activation_function = lambda x:0 if x<0 else 1

training_data = [ (np.array([1,1,1]),1),
              (np.array([1,0,1]),1),
              (np.array([0,1,1]),0)
             ]

# model parameters
#learning rate is a configuration parameter that controls the amount that weights are updated,
learning_rate = 0.2
#the single step that takes to do forward and backward propagation.
training_steps = 100

# initialize weights
W = np.random.rand(3)

for i in range(training_steps):

    x, y = choice(training_data)

    #Compute dot Product between input (X) and weights matrix (W)
    l1 = np.dot(W, x)

    #Apply the activation function on the l1.
    #l1 is actually the output of our network.
    y_pred = activation_function(l1)
    error = y - y_pred
    '''
    model predicts the class correctly the error is equal to zero and no changes are done on the weights,
   but if it makes a mistake, then we can have a negative (decrease) or positive (increase) update on the weights.
    '''
    update = learning_rate * error * x
    W += update

# Output after training
print("Predictions after training on the training dataset:")
for x, _ in training_data:
    y_pred = np.dot(x, W)
    print("{}: {}".format(x[:2], activation_function(y_pred)))

print("\nPrediction for the dataset [1, 1, 0]:\n")
y_predict = np.dot([1, 1, 0], W)
print("[1, 1]: ", activation_function(y_predict))
```

OUTPUT



```
C:\Users\beepl\Desktop\Program\AI>python neuralNetwork.py
        ---Neural Network Implementation----

...Using Single Layered Perceprton....

Predictions after training on the training dataset:
[1 1]: 1
[1 0]: 1
[0 1]: 0

Prediction for the dataset [1, 1, 0]:

[1, 1]:  1
```