Practical 1

**a)** **Aim** : Write a program to implement depth first search algorithm.

**DEPTH FIRST SEARCH ALGORITHM**

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally print the nodes in the path.Depth First Search uses Stack data structure(LIFO).

**ALGORITHM**

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their

STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

**ADVANTAGES**

1.Depth First Search consumes very less memory space.

2.It will reach at the goal node in a less time period than BFS if it traverses in a right path.

3.It may find a solution without examining much of search because we may get the desired solution in the very first go.

**DISADVANTAGES**

1.It is possible that may states keep reoccurring. There is no guarantee of finding the goal node.

2.Sometimes the states may also enter into infinite loops.

**Code :**

graph1={

'A':['B','C','D'],

'B':['A','C'],

'C':['A','E'],

'D':['A'],

'E':['C']

}

print(" \n ")

def dfs(graph,node,visited):

if node not in visited:

visited.append(node)

for n in graph[node]:

dfs(graph,n,visited)

return visited

visited=dfs(graph1,'A',[])

print(visited)

**Output :**

**Conclusion :** We learned how to implement Depth First Search algorithm using Python Programming.

**b)** **Aim:** Write a program to implement breadth first search algorithm

**BREADTH FIRST SEARCH ALGORITHM**

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It uses the opposite strategy of depth-first search, which instead explores the node branch as far as possible before being forced to backtrack and expand other nodes.It uses the Queue Data Structure (FIFO).

**ALGORITHM**

Step 1: Place the root node inside the queue.

Step 2: If the queue is empty then stops and return failure.

Step 3: If the FRONT node of the queue is a goal node then stop and return success.

Step 4: Remove the FRONT node from the queue. Process it and find all its neighbours that are in ready state then place them inside the queue in any order.

Step 5: Go to Step 3.

Step 6: Exit.

**ADVANTAGES**

1.In this procedure at any way it will find the goal.

2.It does not follow a single unfruitful path for a long time. It finds the minimal solution in case of multiple paths.

**DISADVANTAGES**

1.BFS consumes large memory space. Its time complexity is more.

2.It has long pathways, when all paths to a destination are on approximately the same search depth.

**Code:**

graph={

'A':['B','C','D'],

'B':['A','C'],

'C':['A','E'],

'D':['A'],

'E':['C']

}

visited=[]

queue=[]

def bfs(visited,graph,node):

queue.append(node)

while queue:

s=queue.pop(0)

print('Poped element from the queue is:',s)

if s not in visited:

visited.append(s)

for neighbour in graph[s]:

print('neighbour of',s,'is',neighbour)

if neighbour not in visited:

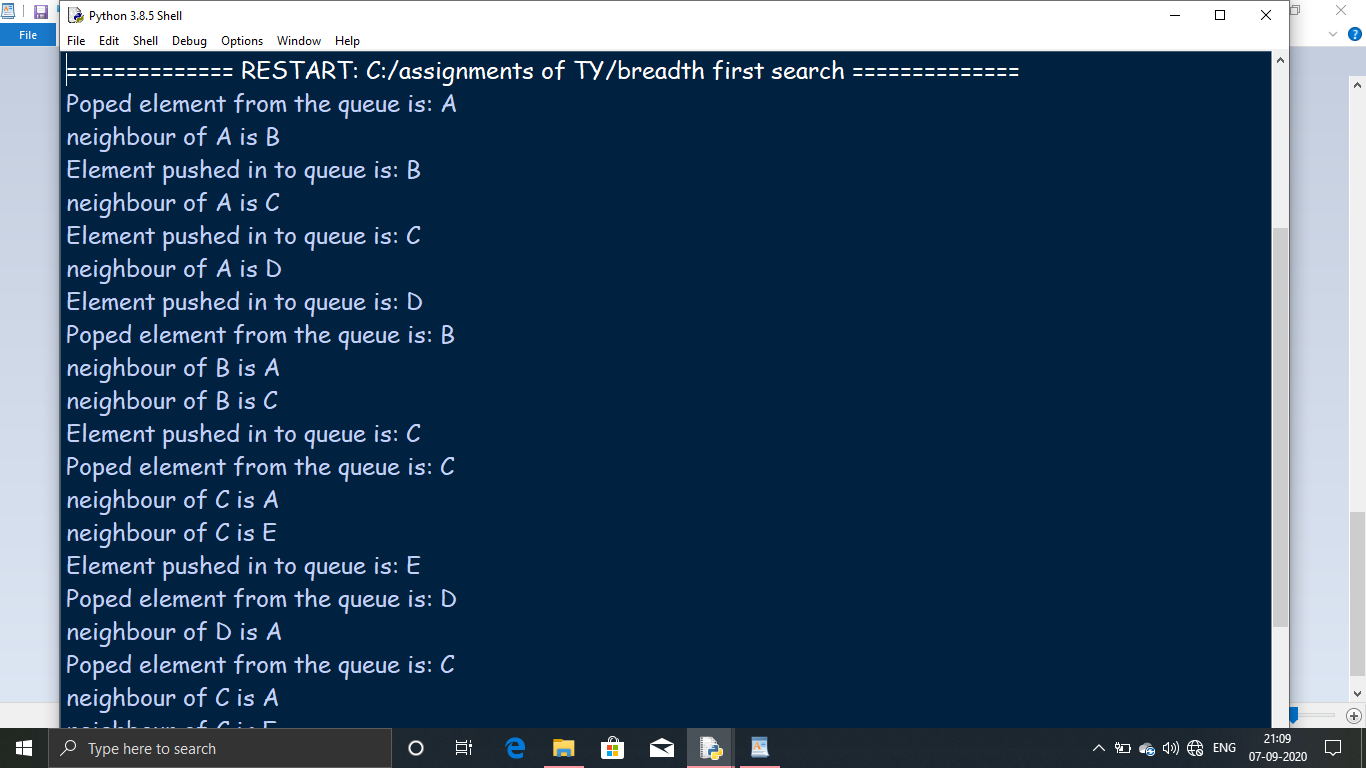
print("Element pushed in to queue is:",neighbour)

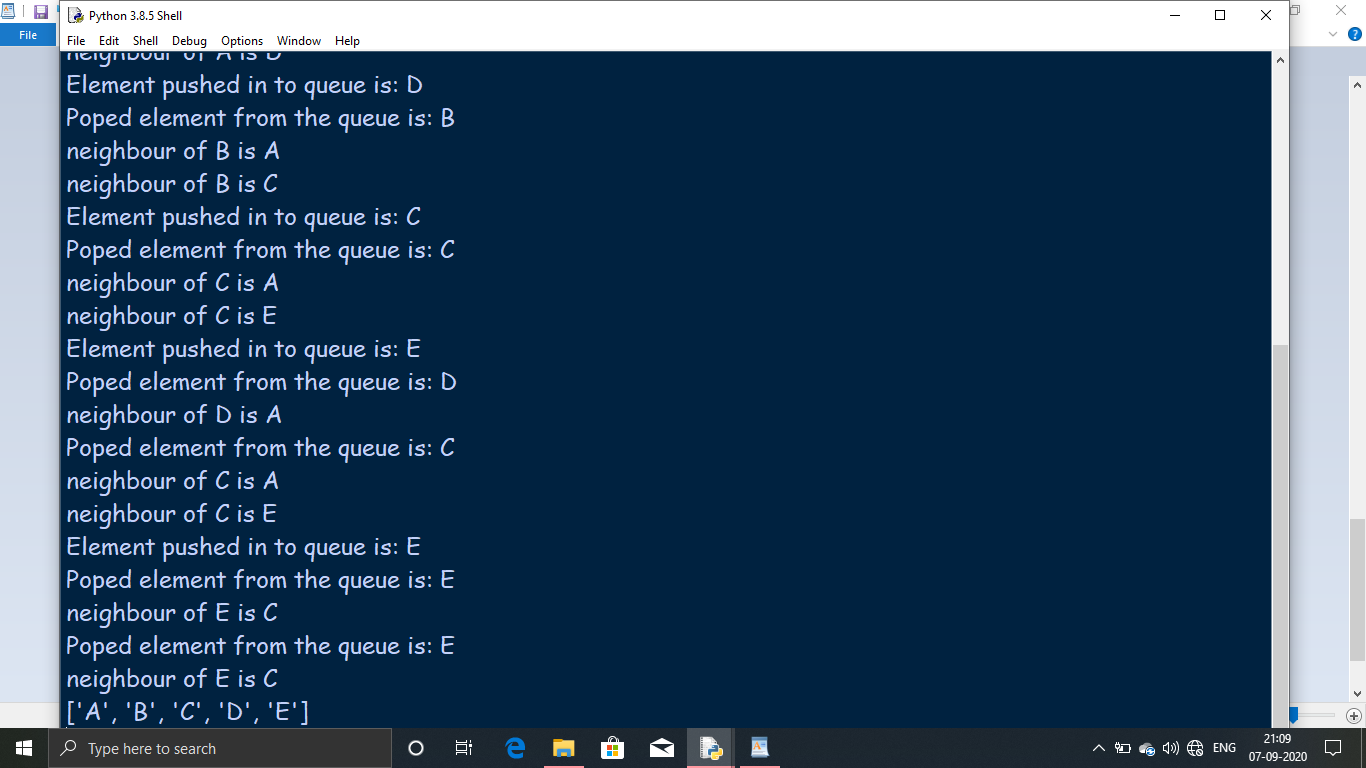
queue.append(neighbour)

bfs(visited,graph,'A')

print(visited)

**Output:**

****

****

**Conclusion :** We learned how to implement Breadth First Search Algorithm using Python Programming.

Practical 2

**a)** **Aim :** Write a program to simulate 4-Queen/N-Queen Problem.

**N-QUEEN PROBLEM**

his problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board. The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way. A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

**ALGORITHM**

Step 1: Start in the leftmost column

Step 2: If all queens are placed

return true

Step 3: Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

Step 4: If all rows have been tried and nothing worked,

return false to trigger backtracking.

As the below code uses backtracking technique lets have the advantages and disadvantages of using backtracking technique.

**ADVANTAGES**

1. The major advantage of the backtracking algorithm is the abillity to find and count all the possible solutions rather than just one while offering decent speed. In fact this is the reason it is so widely used.

2. Also one can easily produce a parallel version of the backtracking algorithm increasing speed several times just by starting multiple threads with different starting positions of the first queens.

**DISADVANTAGES**

1.Backtracking Approach is not efficient for solving strategic Problem.

2.The overall runtime of Backtracking Algorithm is normally slow .

3.To solve Large Problem Sometime it needs to take the help of other techniques like Branch and bound.

**Code :**

global n

n=8

def printnQ(bored):

for i in range(n):

for j in range(n):

print (bored[i][j],end=" ")

print(end="\n")

def isSafe(bored,r,c):

for i in range(c):

if bored[r][i]==1:

return False

for i,j in zip(range(r,-1,-1),

range(c,-1,-1)):

if bored[i][j]==1:

return False

for i,j in zip(range(r,n,1),

range(c,-1,-1)):

if bored[i][j]==1:

return False

return True

def solveNQUtil(bored,c):

if c>=n:

return True

for i in range(n):

if isSafe(bored,i,c):

bored[i][c]=1

if solveNQUtil(bored,c+1)==True:

return True

bored[i][c]=0

return False

def solveNQ():

bored=[

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0],

[0,0,0,0,0,0,0,0]

]

if solveNQUtil(bored,0)==False:

print ("Solution Doesn't Exists.")

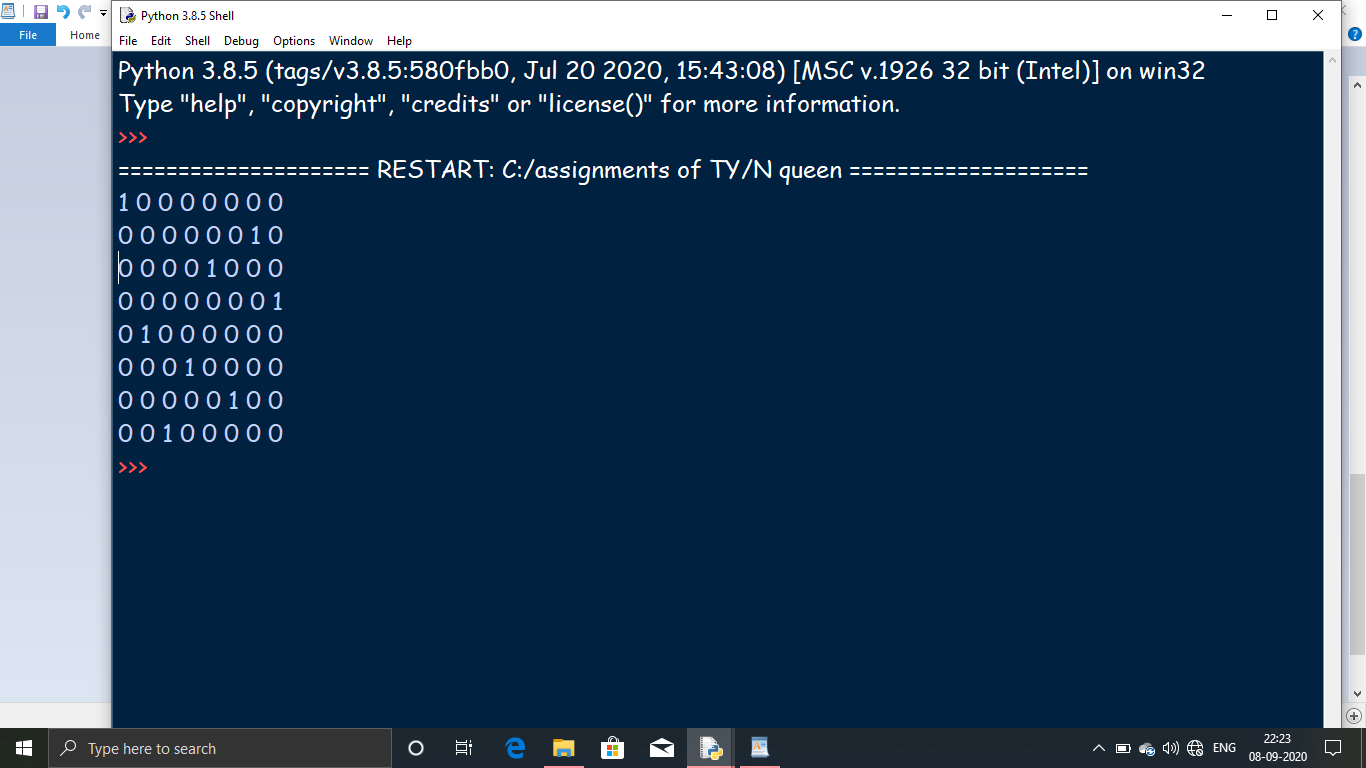
return False

printnQ(bored)

return True

solveNQ()

**OUTPUT**

****

**Conclusion :** We learned how to solve the n-queen problem using Python Programming.

**b)** **Aim:** Write a program to solve Tower of Hanoi Problem.

**TOWER OF HANOI PROBLEM**

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.

2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

3) No disk may be placed on top of a smaller disk.

**ALGORITHM**

Step 1: START

Step 2 :Procedure Hanoi(disk, source, dest, aux)

a) IF disk == 1, THEN

move disk from source to dest

b) ELSE

Hanoi(disk - 1, source, aux, dest) // Step 1

move disk from source to dest // Step 2

Hanoi(disk - 1, aux, dest, source) // Step 3

c) END IF

Step 3: END Procedure

Step 4: STOP

**Code:**

def toh(n,f,t,au):

if(n==1):

print ("Move for 1 disk is from rod "+f+" to rod "+t)

return

toh(n-1,f,au,t)

print ("Move for "+str(n)+" disk is from rod "+f+" to rod "+t)

toh(n-1,au,t,f)

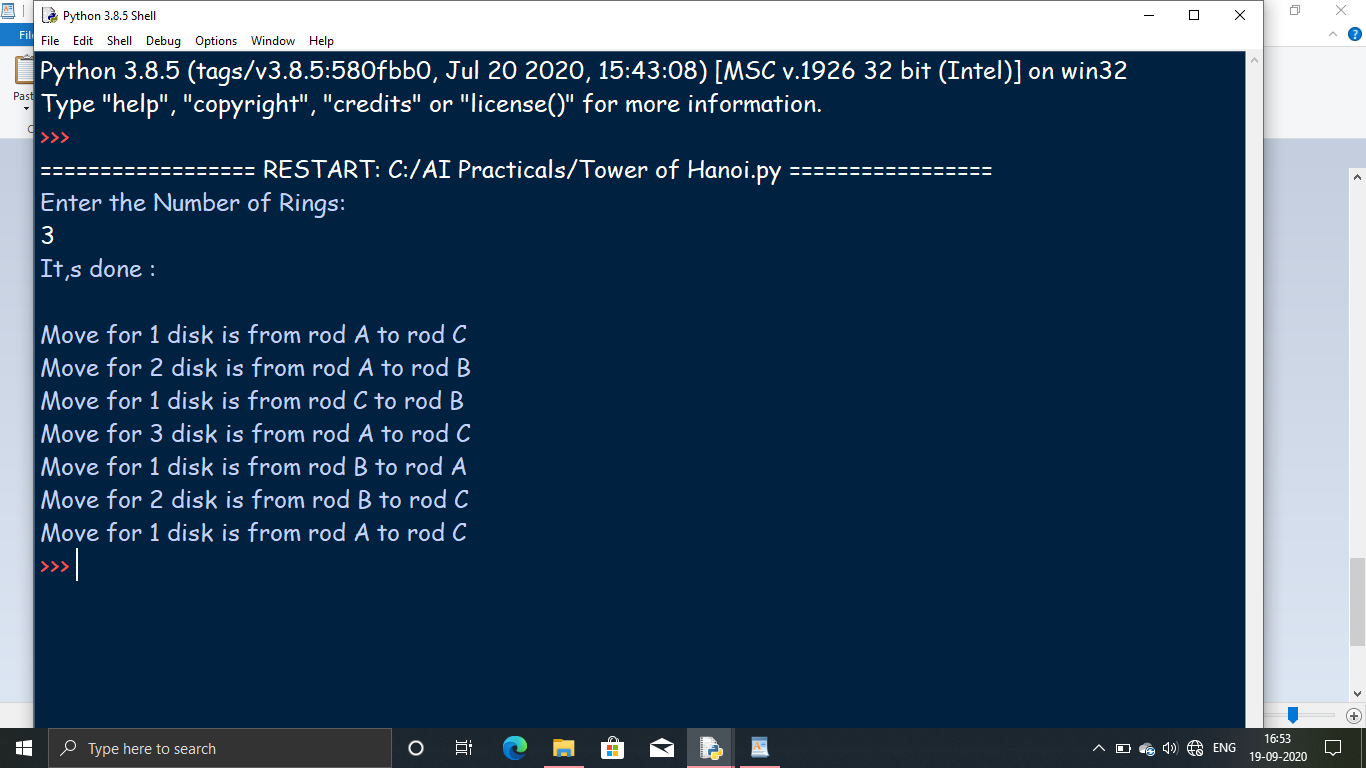
print ("Enter the Number of Rings:")

n=int(input())

print ("It,s done :\n")

toh(n,'A','C','B')

**OUTPUT**

****

**Conclusion :** We learned how to solve the Tower of Hanoi Problem using Python Programming.

Practical 3

**a)** **Aim:** Write a program to implement alpha beta search.

**ALPHA-BETA SEARCH**

Alpha–beta pruning is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree. It is an adversarial search algorithm used commonly for machine playing of two-player games (Tic-tac-toe, Chess, Go, etc.). It stops evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

**WORKING**

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

Step 3: Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

**ADVANTAGES**

1. Alpha-beta pruning plays a great role in reducing the number of nodes which are found out by minimax algorithm.

2. When one chance or option is found at the minimum, it stops assessing a move.

3. This method also helps to improve the search procedure in an effective way.

**DISADVANTAGES**

1. It does not solve all the problems associated with the original minimax algorithm.

2. Requires a set depth limit, as in most cases, it is not feasible to search the entire game tree.

3. Though designed to calculate the good move, it also calculates the values of all the legal moves.

**Code:**

MAX,MIN=1000,-1000

def minimax(depth,nodeIndex,maximizingPlayer,values,alpha,beta):

if depth==3:

return values[nodeIndex]

if maximizingPlayer:

best=MIN

for i in range(0,2):

val=minimax(depth+1,nodeIndex\*2+i,False,values,alpha,beta)

best=max(best,val)

alpha=max(alpha,best)

if beta<=alpha:

break

return best

else:

best=MAX

for i in range(0,2):

val=minimax(depth+1,nodeIndex\*2+i,True,values,alpha,beta)

best=min(best,val)

beta=min(beta,best)

if beta <= alpha:

break

return best

if \_\_name\_\_=="\_\_main\_\_":

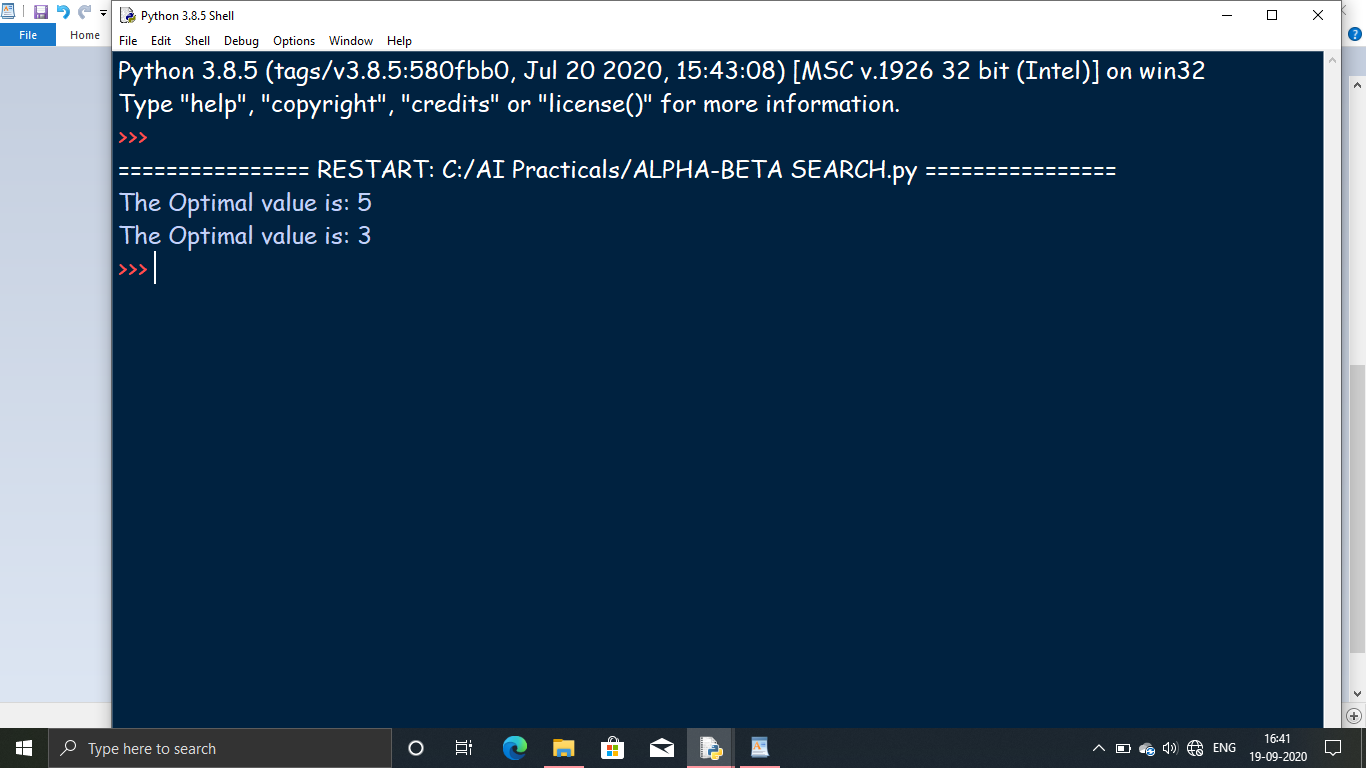
values=[3,5,6,9,1,2,0,-1]

print("The Optimal value is:",minimax(0,0,True,values,MIN,MAX))

values=[2,3,5,9,0,1,7,5]

print("The Optimal value is:",minimax(0,0,True,values,MIN,MAX))

**OUTPUT**

****

**Conclusion :** We learned how to implement alpha beta search and find the optimal solution using Python Programming.

**b)** **Aim:** Write a program for Hill Climbing Problem.

**HILL CLIMBING PROBLEM**

Hill Climbing is a heuristic search used for mathematical optimization problems in the field of Artificial Intelligence.

Given a large set of inputs and a good heuristic function, it tries to find a sufficiently good solution to the problem. This solution may not be the global optimal maximum.

In the above definition, mathematical optimization problems implies that hill-climbing solves the problems where we need to maximize or minimize a given real function by choosing values from the given inputs. Example-Travelling salesman problem where we need to minimize the distance traveled by the salesman.

‘Heuristic search’ means that this search algorithm may not find the optimal solution to the problem. However, it will give a good solution in reasonable time.

A heuristic function is a function that will rank all the possible alternatives at any branching step in search algorithm based on the available information. It helps the algorithm to select the best route out of possible routes.

**ALGORITHM**

Step 1 : Evaluate the initial state. If it is a goal state then stop and return success. Otherwise, make initial state as current state.

Step 2 : Loop until the solution state is found or there are no new operators present which can be applied to the current state.

a) Select a state that has not been yet applied to the current state and apply it to produce a new state.

b) Perform these to evaluate new state

i. If the current state is a goal state, then stop and return success.

ii. If it is better than the current state, then make it current state and proceed further.

iii. If it is not better than the current state, then continue in the loop until a solution is found.

Step 3 : Exit.

**ADVANTAGES**

1. Hill climbing technique is useful in job shop scheduling, automatic programming, circuit designing, and vehicle routing and portfolio management.

2. It is also helpful to solve pure optimization problems where the objective is to find the best state according to the objective function.

3. It requires much less conditions than other search techniques.

**DISADVANTAGES**

1. The question that remains on hill climbing search is whether this hill is the highest hill possible. Unfortunately without further extensive exploration, this question cannot be answered.

2. This technique works but as it uses local information that’s why it can be fooled.

3. The algorithm doesn’t maintain a search tree, so the current node data structure need only record the state and its objective function value. It assumes that local improvement will lead to global improvement.

**Code:**

import random

import string

def generate\_random\_solution(length=11):

return[random.choice(string.printable)for \_ in range(length)]

def evaluate(solution):

target=list("Hello World")

diff=0

for i in range(len(target)):

s=solution[i]

t=target[i]

diff+=abs(ord(s)-ord(t))

return diff

def mutate\_solution(solution):

index=random.randint(0,len(solution)-1)

solution[index]=random.choice(string.printable)

best=generate\_random\_solution()

best\_score=evaluate(best)

while True:

print('Best score so far',best\_score,'Solution',"".join(best))

if best\_score==0:

break

new\_solution=list(best)

mutate\_solution(new\_solution)

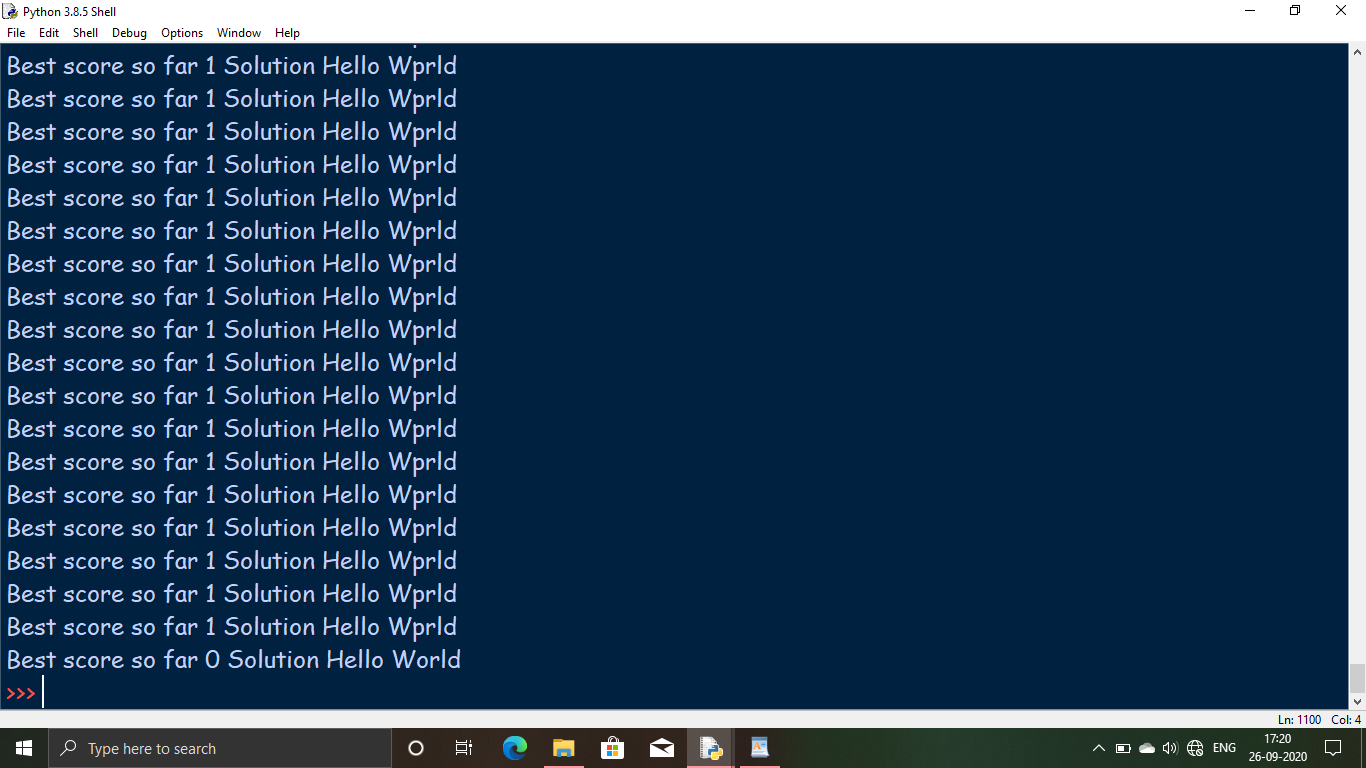
score=evaluate(new\_solution)

if evaluate(new\_solution)<best\_score:

best=new\_solution

best\_score=score

**OUTPUT**

****

**Conclusion :** We learned how to solve the hill climbing problem using Pyhton Programming.

Practical 4

**a)** **Aim :** Write a program to implement A\* algorithm.

**A\* ALGORITHM**

A\* (pronounced "A-star") is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its {\displaystyle O(b^{d})}O(b^d) space complexity, as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance,as well as memory-bounded approaches; however, A\* is still the best solution in many cases.Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968. It can be seen as an extension of Edsger Dijkstra's 1959 algorithm. A\* achieves better performance by using heuristics to guide its search.

**ALGORITHM**

Step 1: Initialize the open list

Step 2: Initialize the closed list

put the starting node on the open

list (you can leave its f at zero)

Step 3: while the open list is not empty

a) find the node with the least f on

the open list, call it "q"

b) pop q off the open list

c) generate q's 8 successors and set their

parents to q

d) for each successor

i) if successor is the goal, stop search

successor.g = q.g + distance between

successor and q

successor.h = distance from goal to

successor (This can be done using many

ways, we will discuss three heuristics-

Manhattan, Diagonal and Euclidean

Heuristics)

successor.f = successor.g + successor.h

ii) if a node with the same position as

successor is in the OPEN list which has a

lower f than successor, skip this successor

iii) if a node with the same position as

successor is in the CLOSED list which has

a lower f than successor, skip this successor

otherwise, add the node to the open list

end (for loop)

e) push q on the closed list

end (while loop)

**ADVANTAGES**

1. It is complete and optimal.

2. It is the best one from other techniques. It is used to solve very complex problems.

3. It is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A\*.

**DISADVANTAGES**

1. This algorithm is complete if the branching factor is finite and every action has fixed cost.

2. The speed execution of A\* search is highly dependant on the accuracy of the heuristic algorithm that is used to compute h (n).

3. It has complexity problems.

**Code :**

import numpy as np

class Node:

"""

A node class for A\* Pathfinding

'parent' is parent of the current Node

'position' is current position of the Node in the maze

'g' is cost from start to current Node

'h' is heuristic based estimated cost for current Node to end Node

'f' is total cost of present node i.e. : f = g + h

"""

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position # check if both are equal ... True iff self and other are the same object

#This function return the path of the search

def return\_path(current\_node,maze):

path = []

no\_rows, no\_columns = np.shape(maze) # get dimensions

# here we create the initialized result maze with -1 in every position

result = [[-1 for i in range(no\_columns)] for j in range(no\_rows)]

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

# Return reversed path as we need to show from start to end path

path = path[::-1]

start\_value = 0

# we update the path of start to end found by A-star search with every step incremented by 1

for i in range(len(path)):

result[path[i][0]][path[i][1]] = start\_value

start\_value += 1

return result

def search(maze, cost, start, end):

"""

Returns a list of tuples as a path from the given start to the given end in the given maze

:param maze:

:param cost

:param start:

:param end:

:return:

"""

# Create start and end node with initized values for g, h and f

start\_node = Node(None, tuple(start))

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = Node(None, tuple(end))

end\_node.g = end\_node.h = end\_node.f = 0

# Initialize both yet\_to\_visit and visited list

# in this list we will put all node that are yet\_to\_visit for exploration.

# From here we will find the lowest cost node to expand next

yet\_to\_visit\_list = []

# in this list we will put all node those already explored so that we don't explore it again

visited\_list = []

# Add the start node

yet\_to\_visit\_list.append(start\_node)

# Adding a stop condition. This is to avoid any infinite loop and stop

# execution after some reasonable number of steps

outer\_iterations = 0

max\_iterations = (len(maze) // 2) \*\* 10

# what squares do we search . serarch movement is left-right-top-bottom

#(4 movements) from every positon

move = [[-1, 0 ], # go up

[ 0, -1], # go left

[ 1, 0 ], # go down

[ 0, 1 ]] # go right

"""

1) We first get the current node by comparing all f cost and selecting the lowest cost node for further expansion

2) Check max iteration reached or not . Set a message and stop execution

3) Remove the selected node from yet\_to\_visit list and add this node to visited list

4) Perform Goal test and return the path else perform below steps

5) For selected node find out all children (use move to find children)

a) get the current postion for the selected node (this becomes parent node for the children)

b) check if a valid position exist (boundary will make few nodes invalid)

c) if any node is a wall then ignore that

d) add to valid children node list for the selected parent

For all the children node

a) if child in visited list then ignore it and try next node

b) calculate child node g, h and f values

c) if child in yet\_to\_visit list then ignore it

d) else move the child to yet\_to\_visit list

"""

#find maze has got how many rows and columns

no\_rows, no\_columns = np.shape(maze)

# Loop until you find the end

while len(yet\_to\_visit\_list) > 0:

# Every time any node is referred from yet\_to\_visit list, counter of limit operation incremented

outer\_iterations += 1

# Get the current node

current\_node = yet\_to\_visit\_list[0]

current\_index = 0

for index, item in enumerate(yet\_to\_visit\_list): # adds counter using enumerate

if item.f < current\_node.f:

current\_node = item

current\_index = index

# if we hit this point return the path such as it may be no solution or

# computation cost is too high

if outer\_iterations > max\_iterations:

print ("giving up on pathfinding too many iterations")

return return\_path(current\_node,maze)

# Pop current node out off yet\_to\_visit list, add to visited list

yet\_to\_visit\_list.pop(current\_index)

visited\_list.append(current\_node)

# test if goal is reached or not, if yes then return the path

if current\_node == end\_node:

return return\_path(current\_node,maze)

# Generate children from all adjacent squares

children = []

for new\_position in move:

# Get node position

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

# Make sure within range (check if within maze boundary)

if (node\_position[0] > (no\_rows - 1) or

node\_position[0] < 0 or

node\_position[1] > (no\_columns -1) or

node\_position[1] < 0):

continue

# Make sure walkable terrain

if maze[node\_position[0]][node\_position[1]] != 0:

continue

# Create new node

new\_node = Node(current\_node, node\_position)

# Append

children.append(new\_node)

# Loop through children

for child in children:

# Child is on the visited list (search entire visited list)

if len([visited\_child for visited\_child in visited\_list if visited\_child == child]) > 0:

continue

# Create the f, g, and h values

child.g = current\_node.g + cost

## Heuristic costs calculated here, this is using eucledian distance

child.h = (((child.position[0] - end\_node.position[0]) \*\* 2) +

((child.position[1] - end\_node.position[1]) \*\* 2))

child.f = child.g + child.h

# Child is already in the yet\_to\_visit list and g cost is already lower

if len([i for i in yet\_to\_visit\_list if child == i and child.g > i.g]) > 0:

continue

# Add the child to the yet\_to\_visit list

yet\_to\_visit\_list.append(child)

# \_\_name\_\_ == '\_\_main\_\_': # execute code only if the file was run directly, and not imported.

if \_\_name\_\_ == '\_\_main\_\_':

maze = [[0, 1, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0],

[0, 1, 0, 1, 0, 0],

[0, 1, 0, 0, 1, 0],

[0, 0, 0, 0, 1, 0]]

start = [0, 0] # starting position

end = [4,5] # ending position

cost = 1 # cost per movement

path = search(maze,cost, start, end)

print(path)

'''

print('\n'.join([''.join(["{:" ">3d}".format(item) for item in row])

for row in path]))

'''

no\_rows, no\_columns = np.shape(maze)

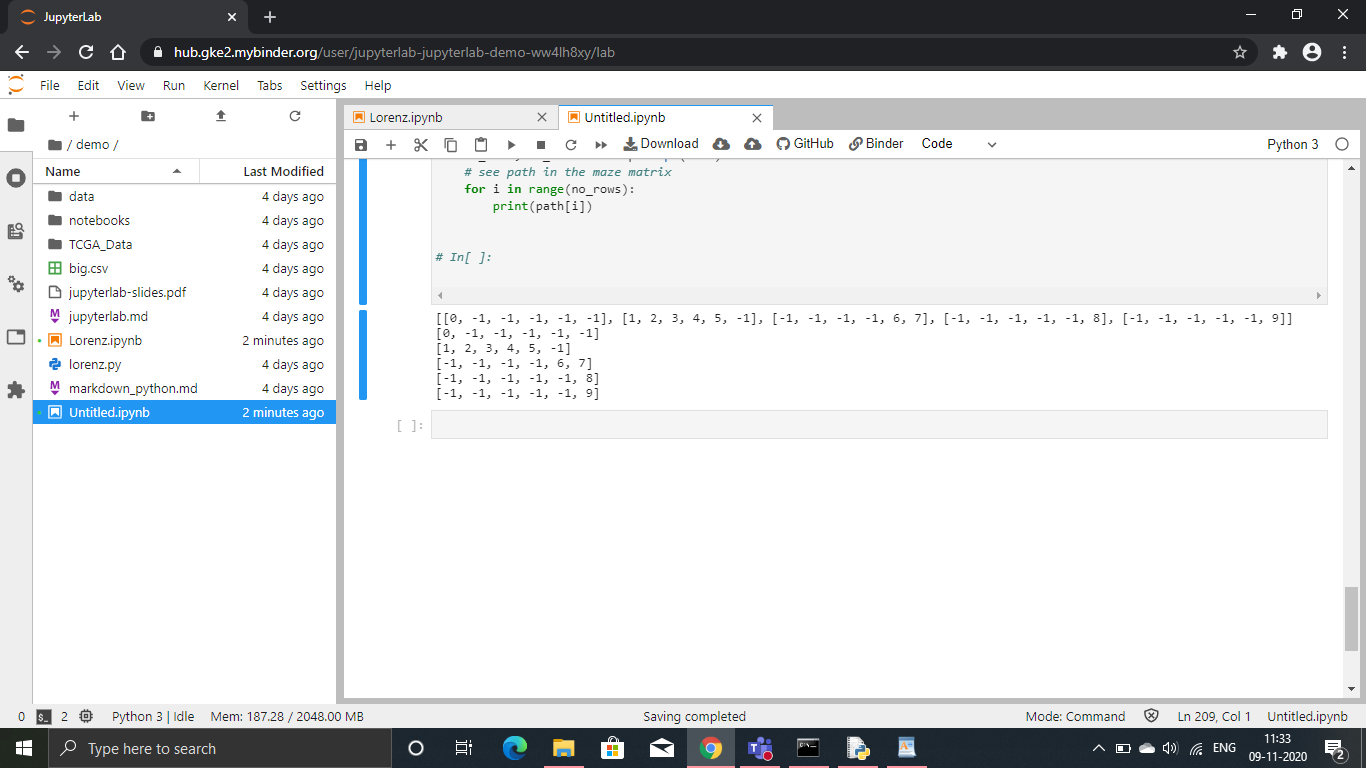
# see path in the maze matrix

for i in range(no\_rows):

print(path[i])

# In[ ]:

**OUTPUT**



**Conclusion :** We learned how to implement the A\* Algorithm using Python Programming.

Practical 5

**a)Aim** : Program to solve Water Jug Problem.

**WATER JUG PROBLEM**

In the water jug problem in Artificial Intelligence, we are provided with two jugs: one having the capacity to hold 3 gallons of water and the other has the capacity to hold 4 gallons of water. There is no other measuring equipment available and the jugs also do not have any kind of marking on them. So, the agent’s task here is to fill the 4-gallon jug with 2 gallons of water by using only these two jugs and no other material. Initially, both our jugs are empty.

**SOLUTION**

Step 1: Fill the n liter jug and empty it into m liter jug.

Step 2: Whenever the n liter jug becomes empty fill it.

Step 3: Whenever the m liter jug becomes full empty it.

Step 4: Repeat steps 1, 2 and 3 till either n liter jug or the m liter jug contains d liters of water.

**Code :**

ctr=0

def pour(jug1,jug2):

max1,max2,fill=5,7,4

global ctr

ctr=ctr+1

print("%d\t%d\t%d" %(++ctr,jug1,jug2))

if jug2 is fill:

return

elif jug2 is max2:

pour(0,jug1)

elif jug1!=0 and jug2 is 0:

pour(0,jug1)

elif jug1 is fill:

pour(jug1,0)

elif jug1<max1:

pour(max1,jug2)

elif jug1<(max2-jug2):

pour(0,(jug1+jug2))

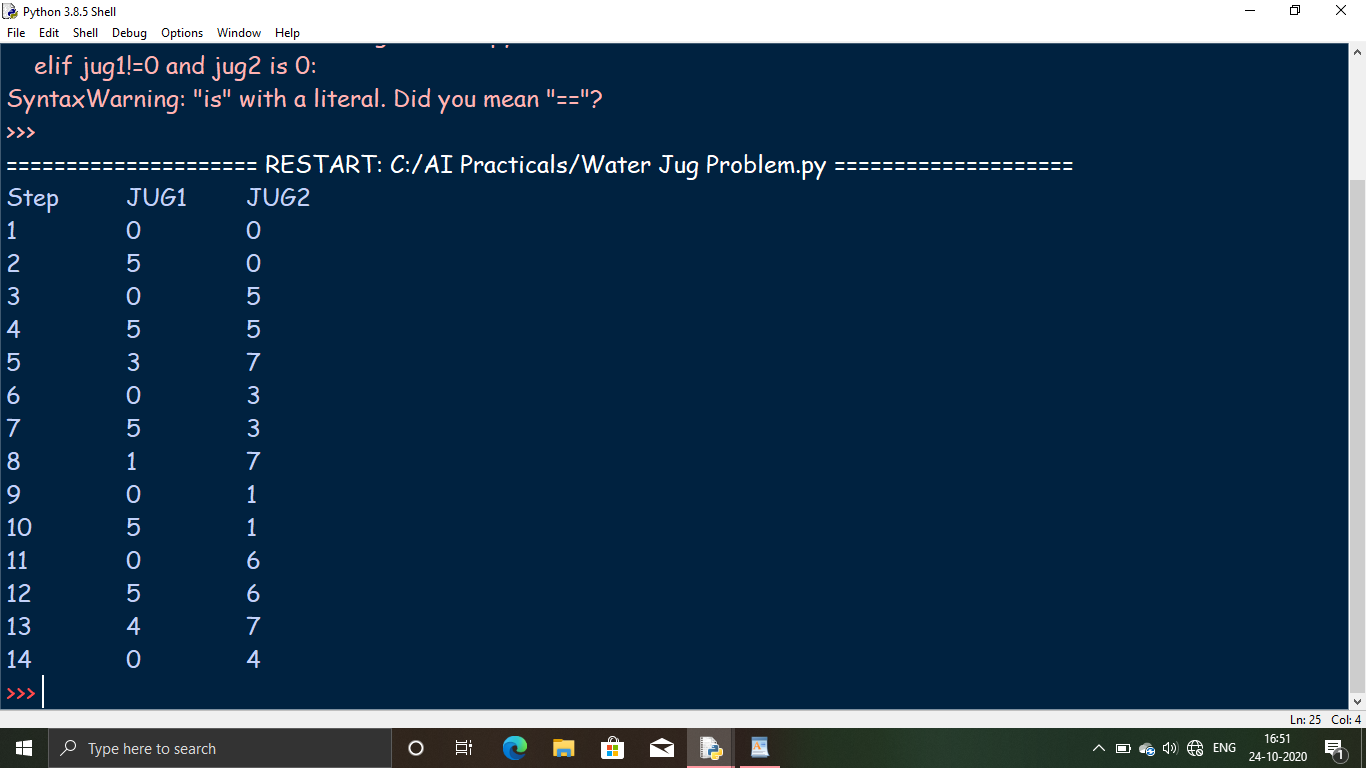
else:

pour(jug1-(max2-jug2),(max2-jug2)+jug2)

print("Step\tJUG1\tJUG2")

pour(0,0)

**OUTPUT**



**Conclusion :** We learned how to solve the water jug problem using Python Programming.

**b) Aim :** Design the simulation of Tic Tac Toe Game using min-max algorithm.

**TIC TAC TOE GAME**

Tic-tac-toe (American English), noughts and crosses (Commonwealth English), or Xs and Os, is a paper-and-pencil game for two players, X and O, who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row is the winner. It is a solved game with a forced draw assuming best play from both players.In order to win the game, a player must place three of their marks in a horizontal, vertical, or diagonal row.Because of the simplicity of tic-tac-toe, it is often used as a pedagogical tool for teaching the concepts of good sportsmanship and the branch of artificial intelligence that deals with the searching of game trees. It is straightforward to write a computer program to play tic-tac-toe perfectly or to enumerate the 765 essentially different positions (the state space complexity) or the 26,830 possible games up to rotations and reflections (the game tree complexity) on this space.[1] If played optimally by both players, the game always ends in a draw, making tic-tac-toe a futile game.

**ADVANTAGES**

1. Minimax algorithm is a beneficial problem-solving algorithm that helps perform a thorough assessment of the search space.

2. However, the most prominent advantage it offers is that it makes it possible to implement decision making in Artificial Intelligence, which has further given way to the development of new and smart machines, systems, and computers.

**DISADVANTAGES**

1. It has a huge branching factor, which makes the process of reaching the goal state slow.

2. Search and evaluation of unnecessary nodes or branches of the game tree degrades the overall performance and efficiency of the engine.

3. Both min and max players have lots of choices to decide from.

4. Exploring the entire tree is not possible as there is a restriction of time and space.

**Code :**

def ConstBoard(board):

print("Current State Of Board : \n\n");

for i in range (0,9):

if((i>0) and (i%3)==0):

print("\n");

if(board[i]==0):

print("- ",end=" ");

if (board[i]==1):

print("O ",end=" ");

if(board[i]==-1):

print("X ",end=" ");

print("\n\n");

#This function takes the user move as input and make the required changes on the board.

def User1Turn(board):

pos=input("Enter X's position from [1...9]: ");

pos=int(pos);

if(board[pos-1]!=0):

print("Wrong Move!!!");

exit(0) ;

board[pos-1]=-1;

def User2Turn(board):

pos=input("Enter O's position from [1...9]: ");

pos=int(pos);

if(board[pos-1]!=0):

print("Wrong Move!!!");

exit(0);

board[pos-1]=1;

#MinMax function.

def minimax(board,player):

x=analyzeboard(board);

if(x!=0):

return (x\*player);

pos=-1;

value=-2;

for i in range(0,9):

if(board[i]==0):

board[i]=player;

score=-minimax(board,(player\*-1));

if(score>value):

value=score;

pos=i;

board[i]=0;

if(pos==-1):

return 0;

return value;

#This function makes the computer's move using minmax algorithm.

def CompTurn(board):

pos=-1;

value=-2;

for i in range(0,9):

if(board[i]==0):

board[i]=1;

score=-minimax(board, -1);

board[i]=0;

if(score>value):

value=score;

pos=i;

board[pos]=1;

#This function is used to analyze a game.

def analyzeboard(board):

cb=[[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]];

for i in range(0,8):

if(board[cb[i][0]] != 0 and

board[cb[i][0]] == board[cb[i][1]] and

board[cb[i][0]] == board[cb[i][2]]):

return board[cb[i][2]];

return 0;

#Main Function.

def main():

choice=input("Enter 1 for single player, 2 for multiplayer: ");

choice=int(choice);

#The broad is considered in the form of a single dimentional array.

#One player moves 1 and other move -1.

board=[0,0,0,0,0,0,0,0,0];

if(choice==1):

print("Computer : O Vs. You : X");

player= input("Enter to play 1(st) or 2(nd) :");

player = int(player);

for i in range (0,9):

if(analyzeboard(board)!=0):

break;

if((i+player)%2==0):

CompTurn(board);

else:

ConstBoard(board);

User1Turn(board);

else:

for i in range (0,9):

if(analyzeboard(board)!=0):

break;

if((i)%2==0):

ConstBoard(board);

User1Turn(board);

else:

ConstBoard(board);

User2Turn(board);

x=analyzeboard(board);

if(x==0):

ConstBoard(board);

print("Draw!!!")

if(x==-1):

ConstBoard(board);

print("X Wins!!! Y Loose !!!")

if(x==1):

ConstBoard(board);

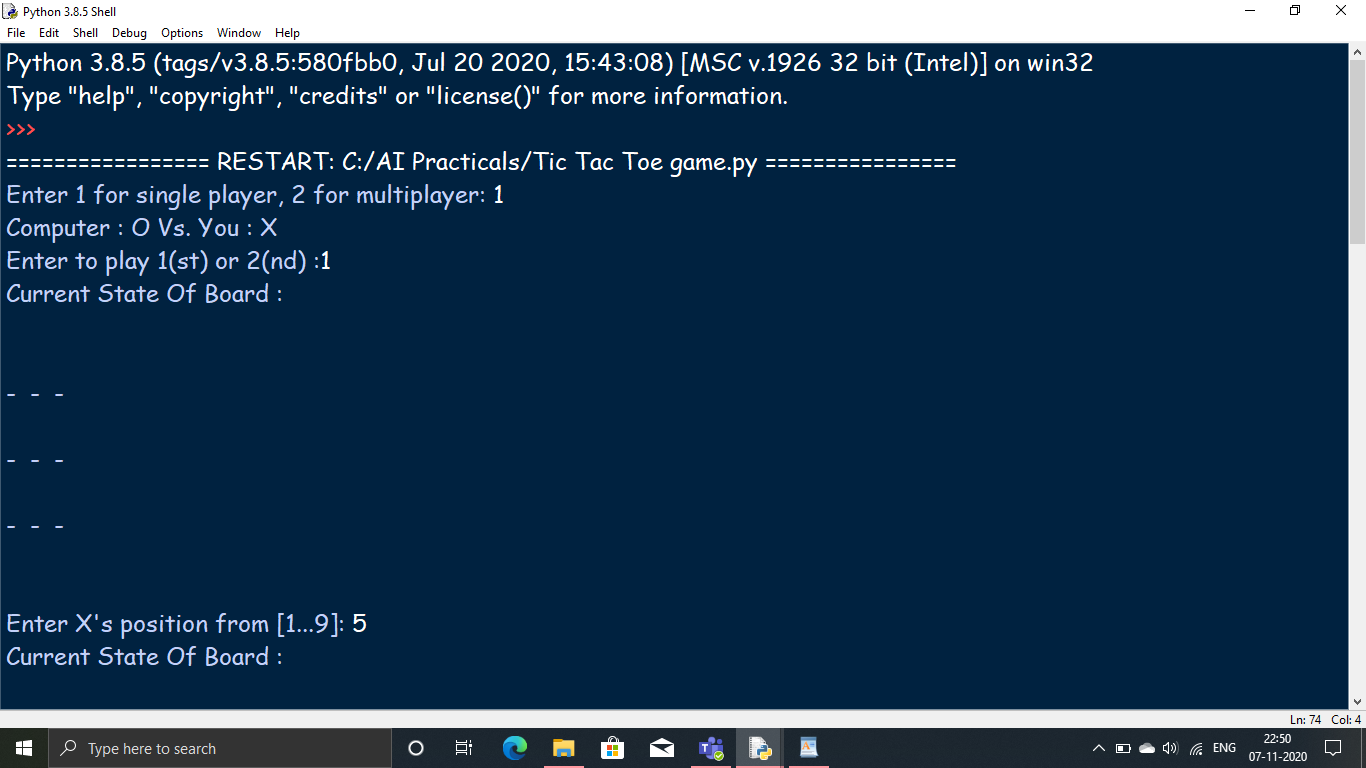
print("X Loose!!! O Wins !!!!")

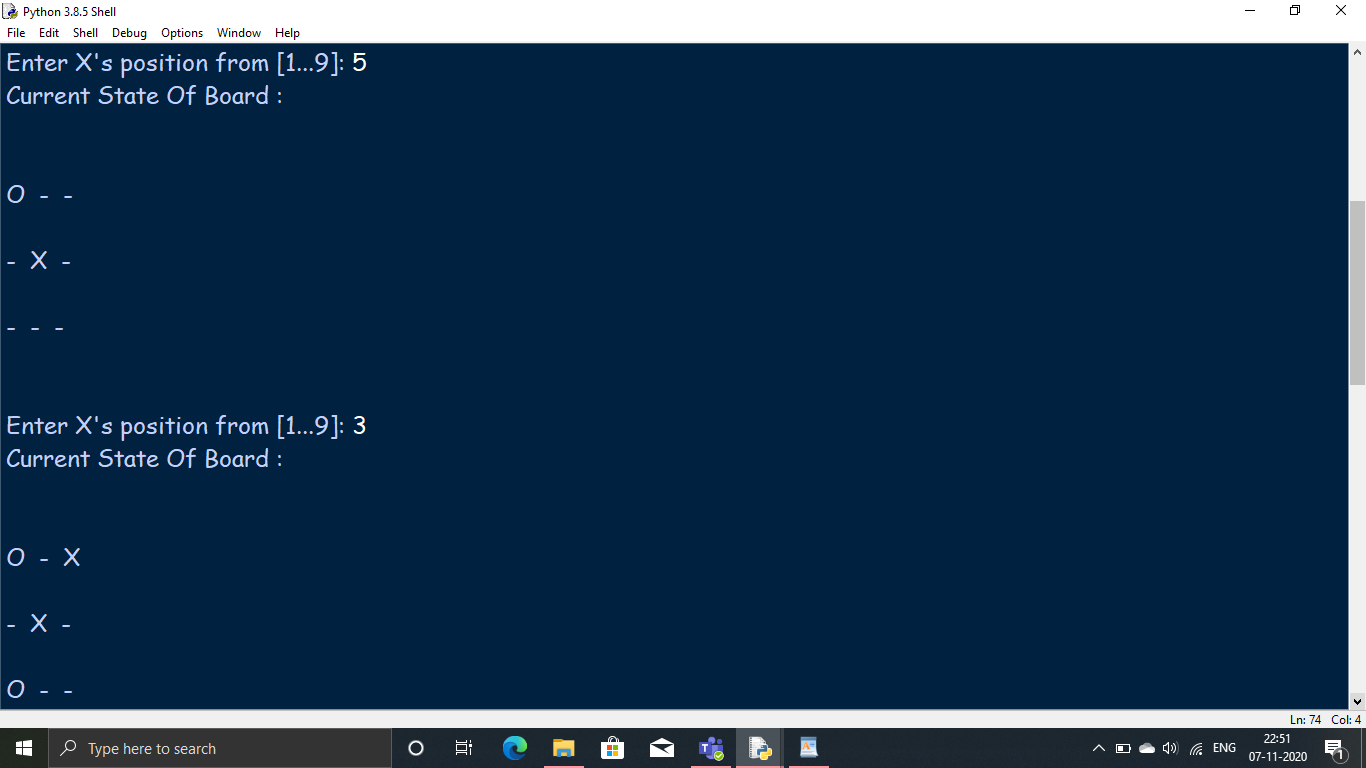
#---------------#

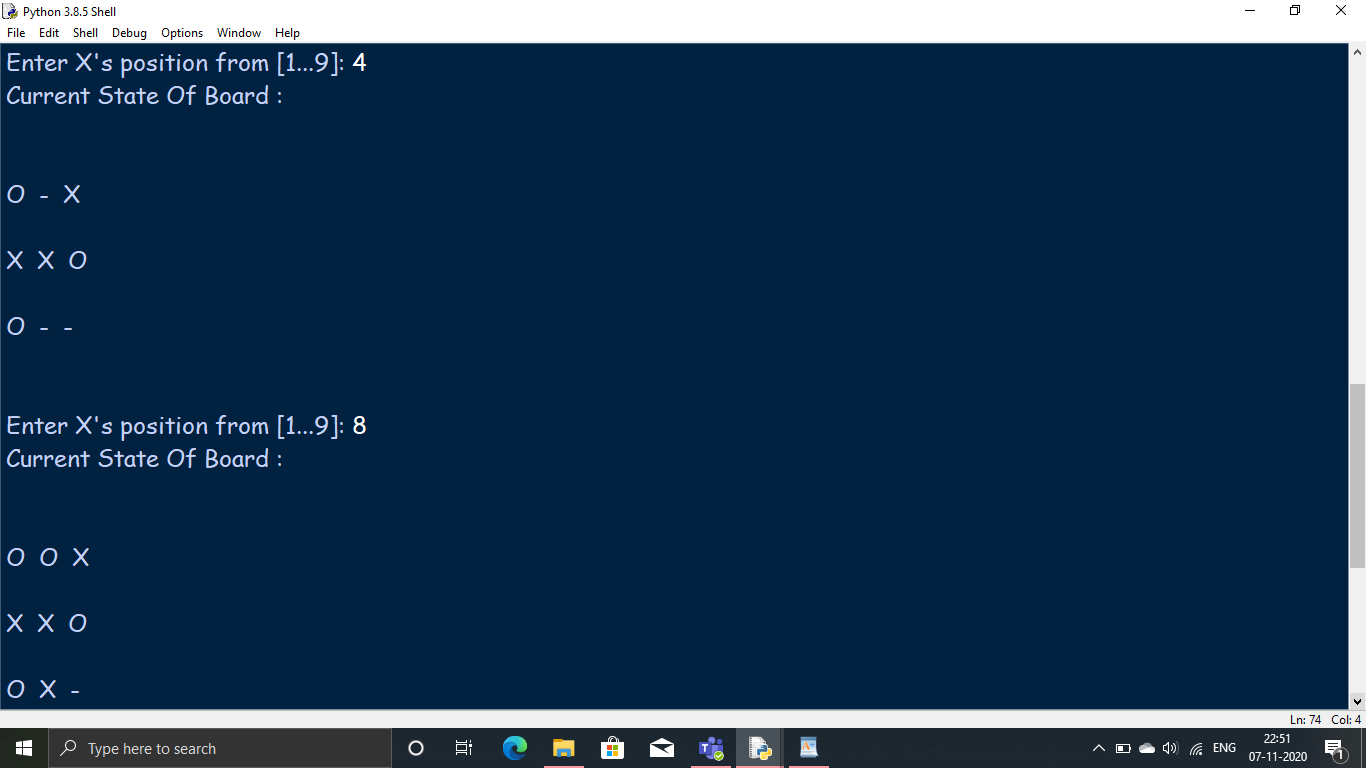
main()

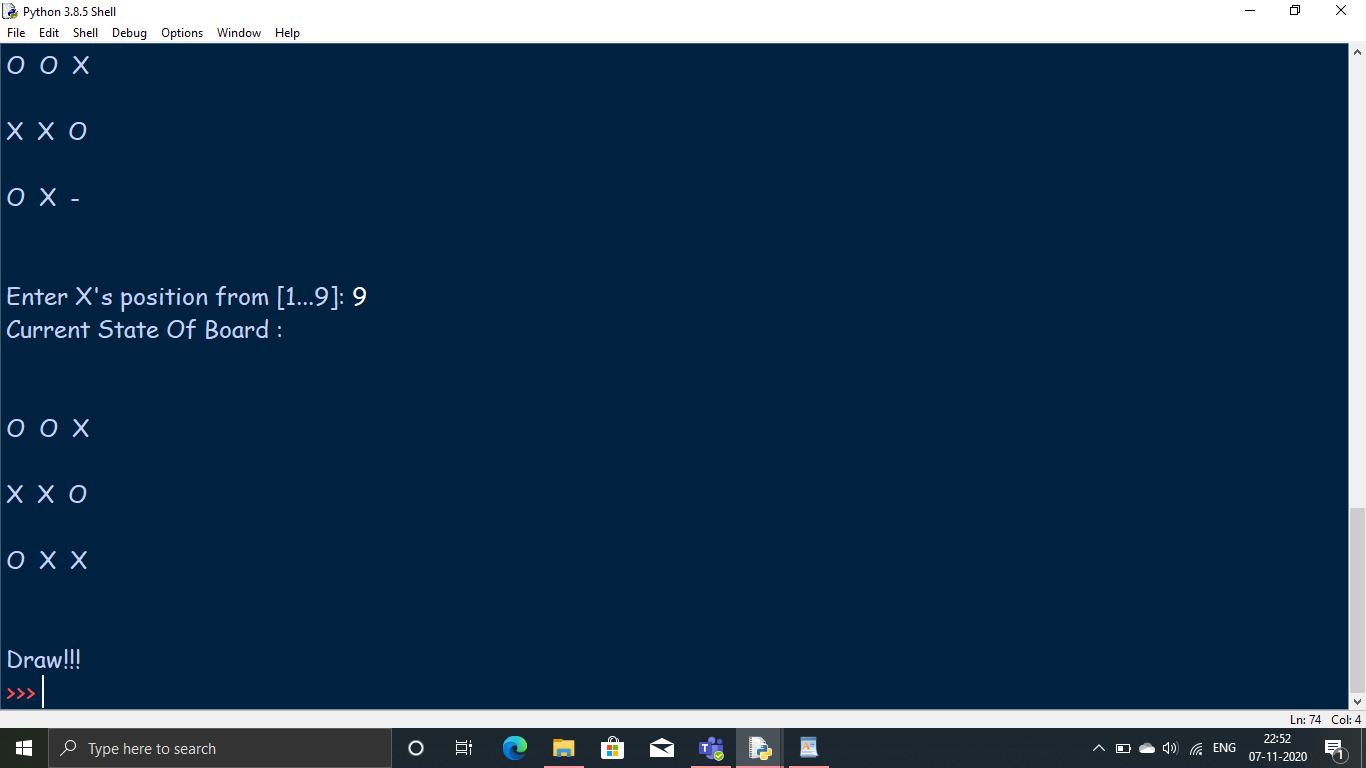
#---------------#

**OUTPUT**









**Conclusion :** We learned how to make the Tic Tac Toe game using Python Programming.

Practical 6

**a)** **Aim :** Write a program to solve Missionaries and Cannibals problem.

**MISSIONARIES AND CANNIBALS PROBLEM**

In the missionaries and cannibals problem, three missionaries and three cannibals must cross a river using a boat which can carry at most two people, under the constraint that, for both banks, if there are missionaries present on the bank, they cannot be outnumbered by cannibals (if they were, the cannibals would eat the missionaries). The boat cannot cross the river by itself with no people on board. And, in some variations, one of the cannibals has only one arm and cannot row.  
**SOLUTION**

First let us consider that both the missionaries (M) and cannibals(C) are on the same side of the river.

Left Right

Initially the positions are : 0M , 0C and 3M , 3C (B)

Now let’s send 2 Cannibals to left of bank : 0M , 2C (B) and 3M , 1C

Send one cannibal from left to right : 0M , 1C and 3M , 2C (B)

Now send the 2 remaining Cannibals to left : 0M , 3C (B) and 3M , 0C

Send 1 cannibal to the right : 0M , 2C and 3M , 1C (B)

Now send 2 missionaries to the left : 2M , 2C (B) and 1M . 1C

Send 1 missionary and 1 cannibal to right : 1M , 1C and 2M , 2C (B)

Send 2 missionaries to left : 3M , 1C (B) and 0M , 2C

Send 1 cannibal to right : 3M , 0C and 0M , 3C (B)

Send 2 cannibals to left : 3M , 2C (B) and 0M , 1C

Send 1 cannibal to right : 3M , 1C and 0M , 2C (B)’

Send 2 cannibals to left : 3M , 3C (B) and 0M , 0C

Here (B) shows the position of the boat after the action is performed.

**Code :**

class State():

def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):

self.cannibalLeft = cannibalLeft

self.missionaryLeft = missionaryLeft

self.boat = boat

self.cannibalRight = cannibalRight

self.missionaryRight = missionaryRight

self.parent = None

def is\_goal(self):

if self.cannibalLeft == 0 and self.missionaryLeft == 0:

return True

else:

return False

def is\_valid(self):

if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \

and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \

and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \

and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):

return True

else:

return False

def \_\_eq\_\_(self, other):

return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \

and self.boat == other.boat and self.cannibalRight == other.cannibalRight \

and self.missionaryRight == other.missionaryRight

def \_\_hash\_\_(self):

return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))

def successors(cur\_state):

children = [];

if cur\_state.boat == 'left':

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 2)

## Two missionaries cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 2, cur\_state.missionaryRight)

## Two cannibals cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)

## One missionary and one cannibal cross left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',

cur\_state.cannibalRight, cur\_state.missionaryRight + 1)

## One missionary crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',

cur\_state.cannibalRight + 1, cur\_state.missionaryRight)

## One cannibal crosses left to right.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

else:

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 2)

## Two missionaries cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 2, cur\_state.missionaryRight)

## Two cannibals cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)

## One missionary and one cannibal cross right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',

cur\_state.cannibalRight, cur\_state.missionaryRight - 1)

## One missionary crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',

cur\_state.cannibalRight - 1, cur\_state.missionaryRight)

## One cannibal crosses right to left.

if new\_state.is\_valid():

new\_state.parent = cur\_state

children.append(new\_state)

return children

def breadth\_first\_search():

initial\_state = State(3,3,'left',0,0)

if initial\_state.is\_goal():

return initial\_state

frontier = list()

explored = set()

frontier.append(initial\_state)

while frontier:

state = frontier.pop(0)

if state.is\_goal():

return state

explored.add(state)

children = successors(state)

for child in children:

if (child not in explored) or (child not in frontier):

frontier.append(child)

return None

def print\_solution(solution):

path = []

path.append(solution)

parent = solution.parent

while parent:

path.append(parent)

parent = parent.parent

for t in range(len(path)):

state = path[len(path) - t - 1]

print( "(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \

+ "," + state.boat + "," + str(state.cannibalRight) + "," + \

str(state.missionaryRight) + ")")

def main():

solution = breadth\_first\_search()

print ("Missionaries and Cannibals solution:")

print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")

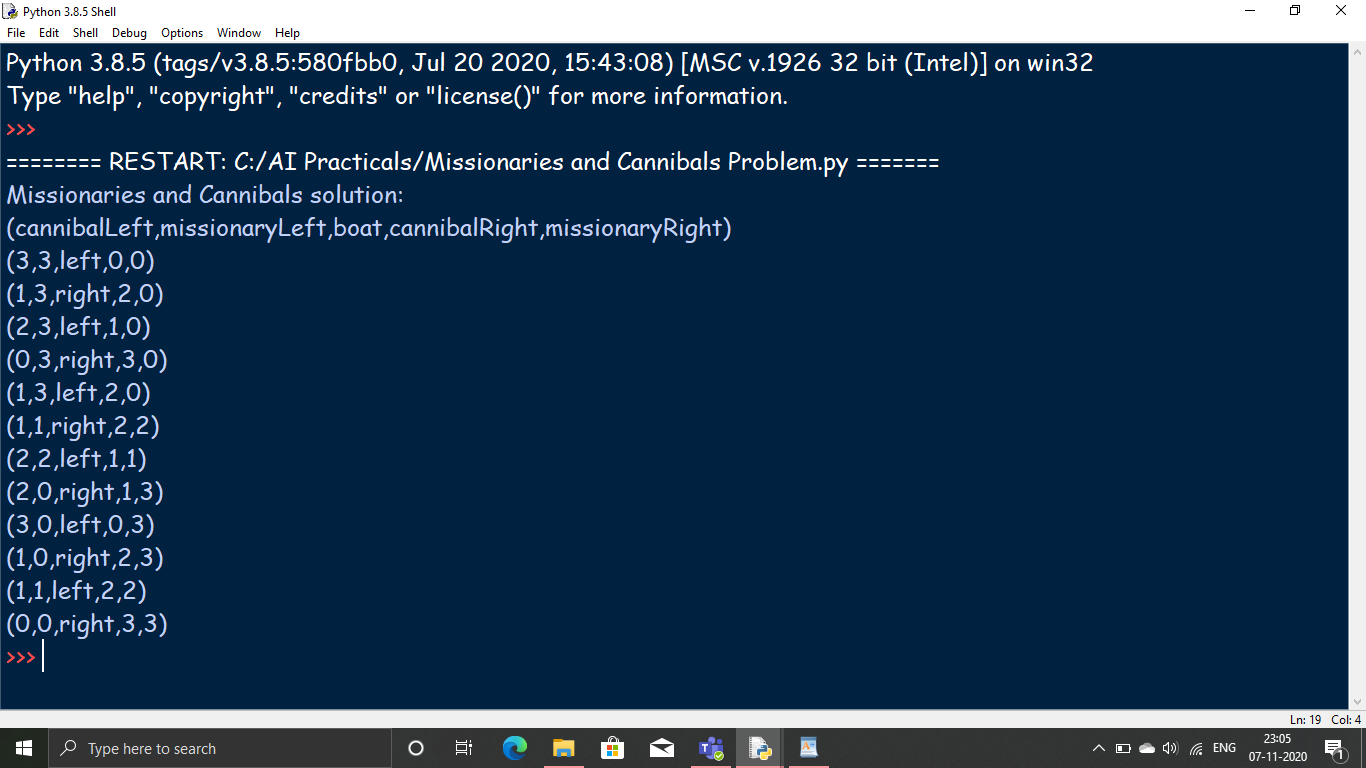
print\_solution(solution)

# if called from the command line, call main()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT**



**Conclusion :** We learned how to solve the missionaries and cannibals probkem using Python Prohramming.

**b)** **Aim :** Design an application to simulate number puzzle problem.

**NUMBER PUZZLE PROBLEM**

The problem. The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order. You are permitted to slide blocks horizontally or vertically into the blank square.

**RULES TO SOLVE**

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)

Up

Down

Right or

Left

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

**Code :**

# 6-b- 8 puzzle problem

# use pip install simpleai before executing the program

# from \_\_future\_\_ import print\_function

from simpleai.search import astar, SearchProblem

# from simpleai.search.viewers import WebViewer

GOAL = '''1-2-3

4-5-6

7-8-e'''

INITIAL = '''4-1-2

7-e-3

8-5-6'''

def list\_to\_string(list\_):

return '\n'.join(['-'.join(row) for row in list\_])

def string\_to\_list(string\_):

return [row.split('-') for row in string\_.split('\n')]

def find\_location(rows, element\_to\_find):

'''Find the location of a piece in the puzzle.

Returns a tuple: row, column'''

for ir, row in enumerate(rows):

for ic, element in enumerate(row):

if element == element\_to\_find:

return ir, ic

# we create a cache for the goal position of each piece, so we don't have to

# recalculate them every time

goal\_positions = {}

rows\_goal = string\_to\_list(GOAL)

for number in '12345678e':

goal\_positions[number] = find\_location(rows\_goal, number)

class EigthPuzzleProblem(SearchProblem):

def actions(self, state):

'''Returns a list of the pieces we can move to the empty space.'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

actions = []

if row\_e > 0:

actions.append(rows[row\_e - 1][col\_e])

if row\_e < 2:

actions.append(rows[row\_e + 1][col\_e])

if col\_e > 0:

actions.append(rows[row\_e][col\_e - 1])

if col\_e < 2:

actions.append(rows[row\_e][col\_e + 1])

return actions

def result(self, state, action):

'''Return the resulting state after moving a piece to the empty space.

(the "action" parameter contains the piece to move)

'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

row\_n, col\_n = find\_location(rows, action)

rows[row\_e][col\_e], rows[row\_n][col\_n] = rows[row\_n][col\_n], rows[row\_e][col\_e] # swaping using tupple

return list\_to\_string(rows)

def is\_goal(self, state):

'''Returns true if a state is the goal state.'''

return state == GOAL

def cost(self, state1, action, state2):

'''Returns the cost of performing an action. No useful on this problem, i

but needed.

'''

return 1

def heuristic(self, state):

'''Returns an \*estimation\* of the distance from a state to the goal.

We are using the manhattan distance.

'''

rows = string\_to\_list(state)

distance = 0

for number in '12345678e':

row\_n, col\_n = find\_location(rows, number)

row\_n\_goal, col\_n\_goal = goal\_positions[number]

distance += abs(row\_n - row\_n\_goal) + abs(col\_n - col\_n\_goal)

return distance

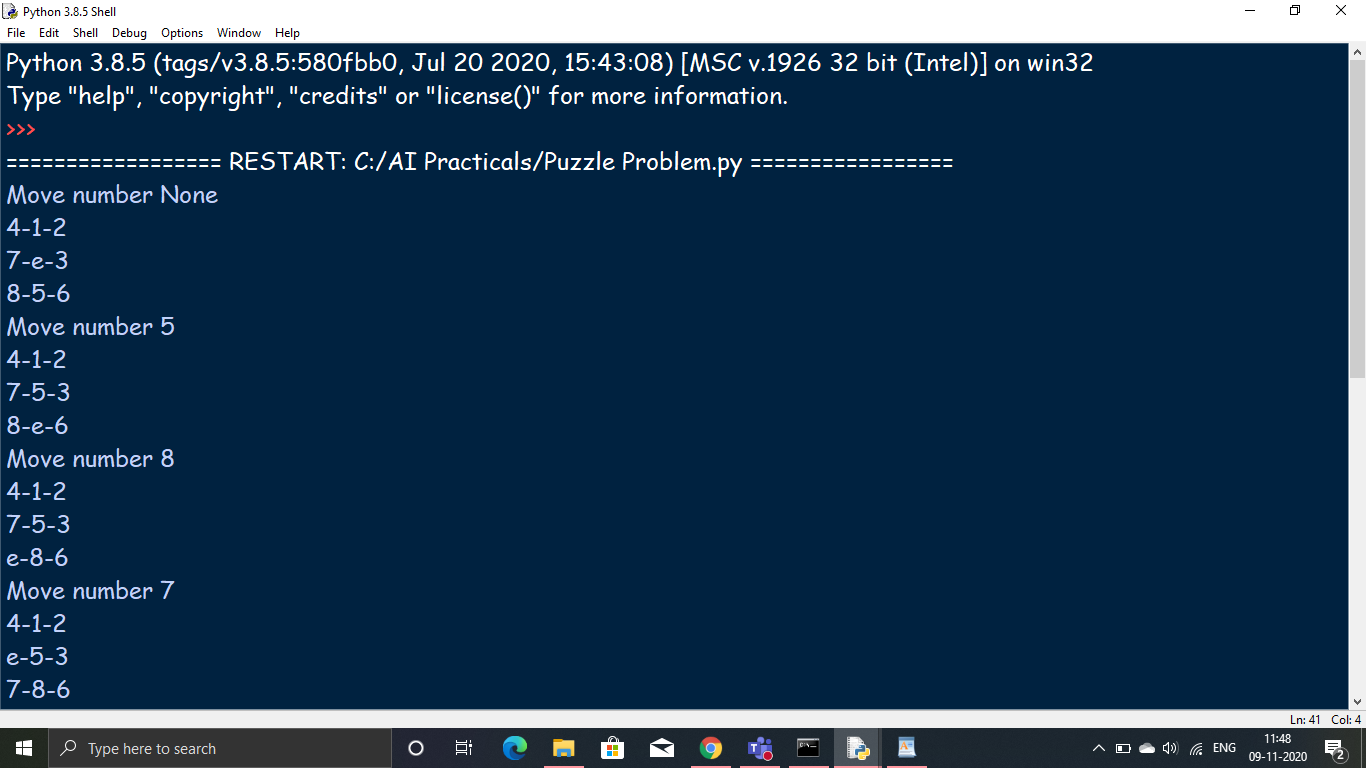
result = astar(EigthPuzzleProblem(INITIAL))

for action, state in result.path():

print('Move number', action)

print(state)

**OUTPUT**





**Conclusion :** We learned how to solve N-puzzle problem using python programming.

Practical 7

**a)** **Aim :** Write a program to shuffle deck of cards.

**SHUFFLING DECK OF CARDS**

Given a deck of cards(52 cards), the task is to shuffle them.

ALGORITHM

Step 1: First, fill the array with the values in order.

Step 2: Go through the array and exchange each element

with the randomly chosen element in the range

from itself to the end.

**Code :**

import itertools, random

# make a deck of cards

deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

'''

The product() function in itertools module to create a deck of cards.

This function performs the Cartesian product of the two sequences.

The two sequences are numbers from 1 to 13 and the four suits.

So, altogether we have 13 \* 4 = 52 items in the deck with each card as a tuple

'''

# shuffle the cards

# we shuffle it using the function shuffle() in random module.

random.shuffle(deck)

print("The cards sequences")

for i in range(len(deck)):

print("card",i,deck[i])

# draw five cards

# we draw the first five cards and display it to the user.

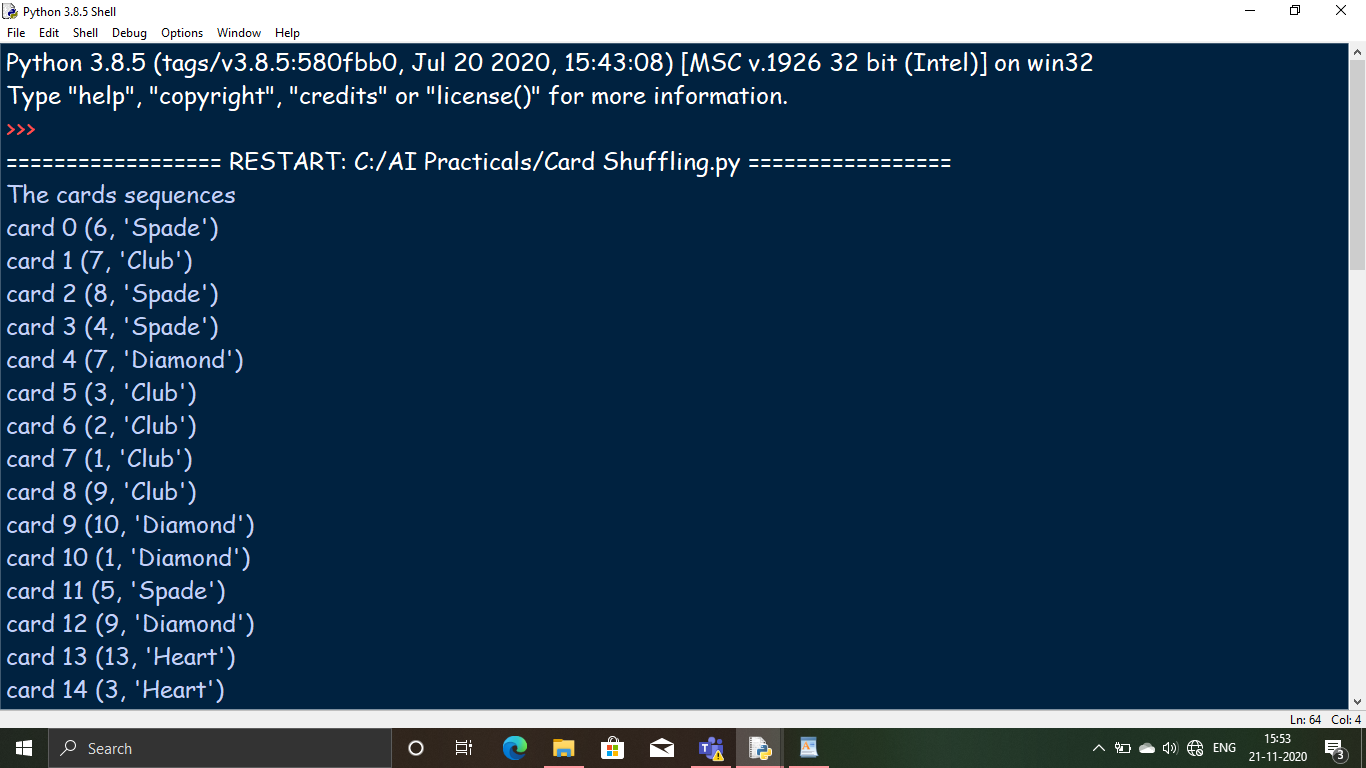
#We will get different output each time you run this program as shown in our two outputs.

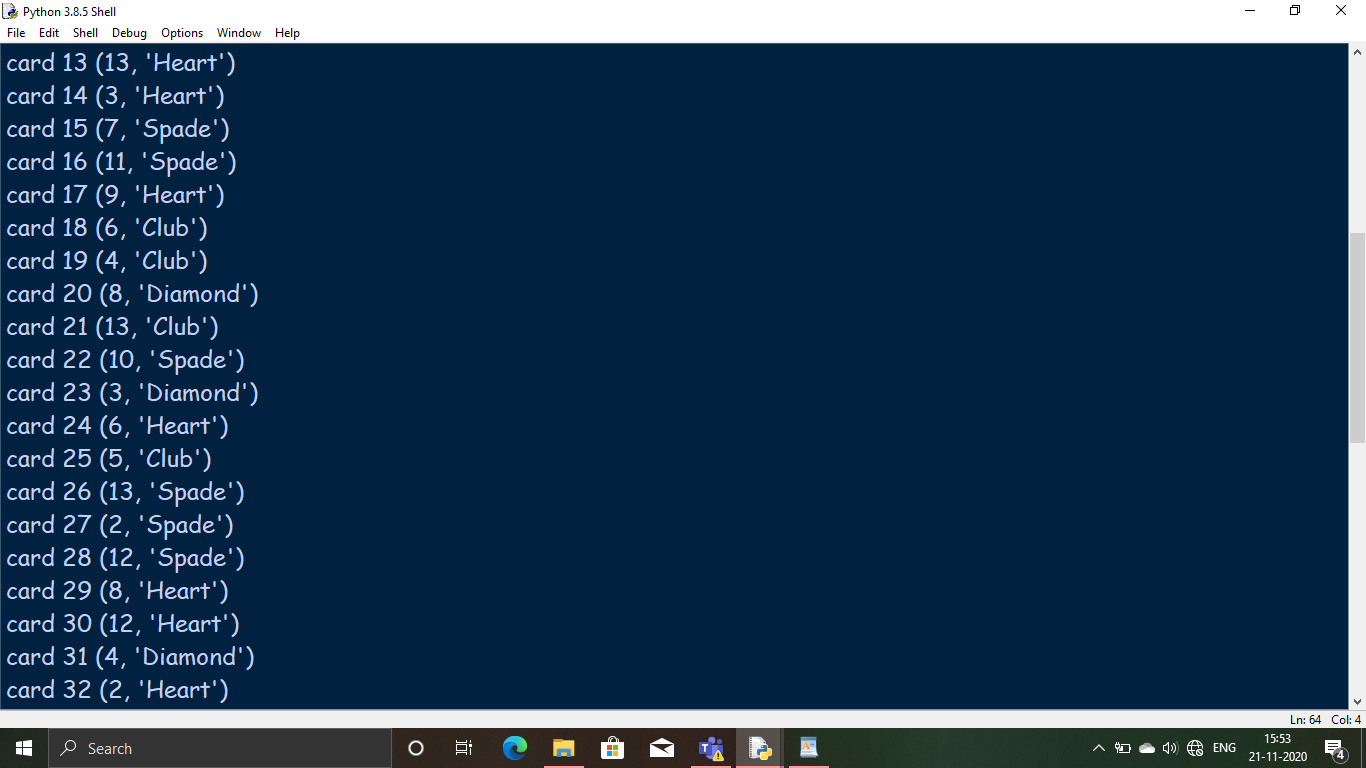
print("You have got after shuffling:")

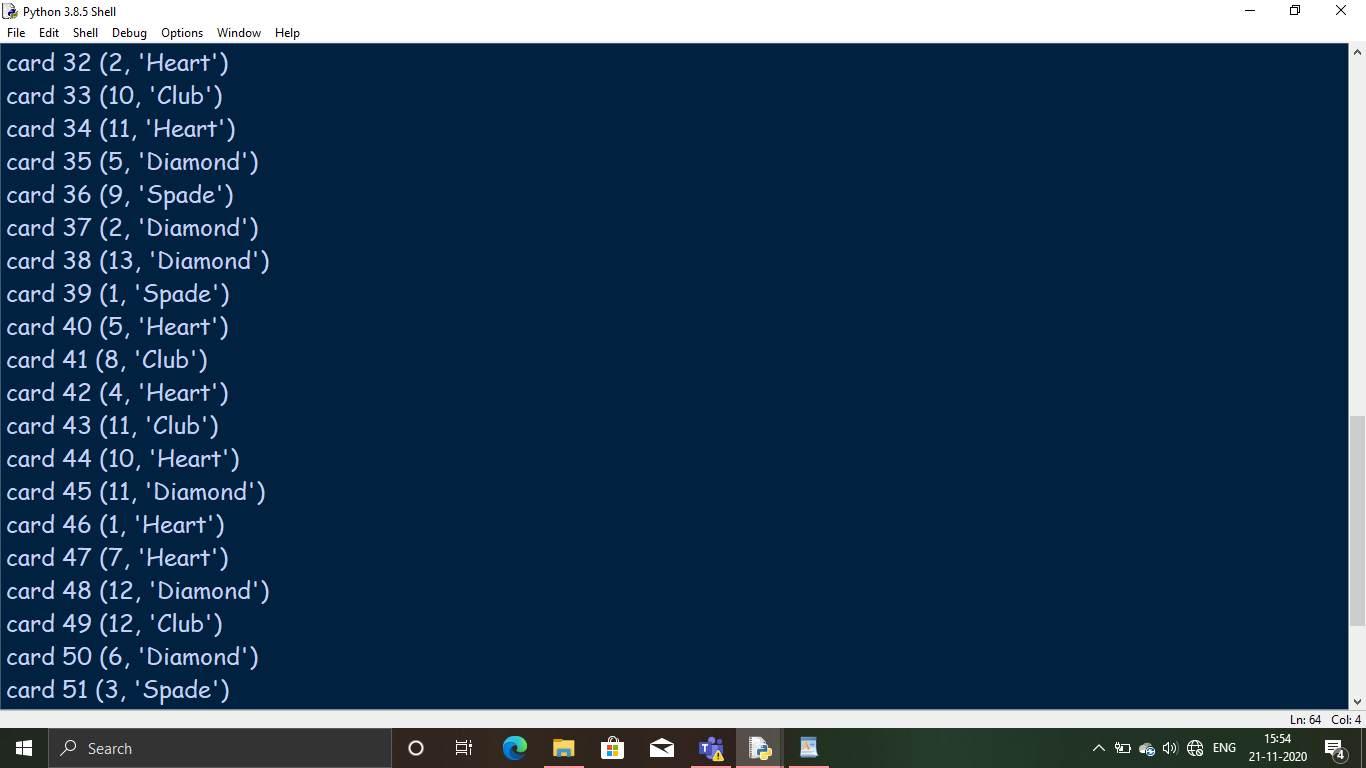
for i in range(5):

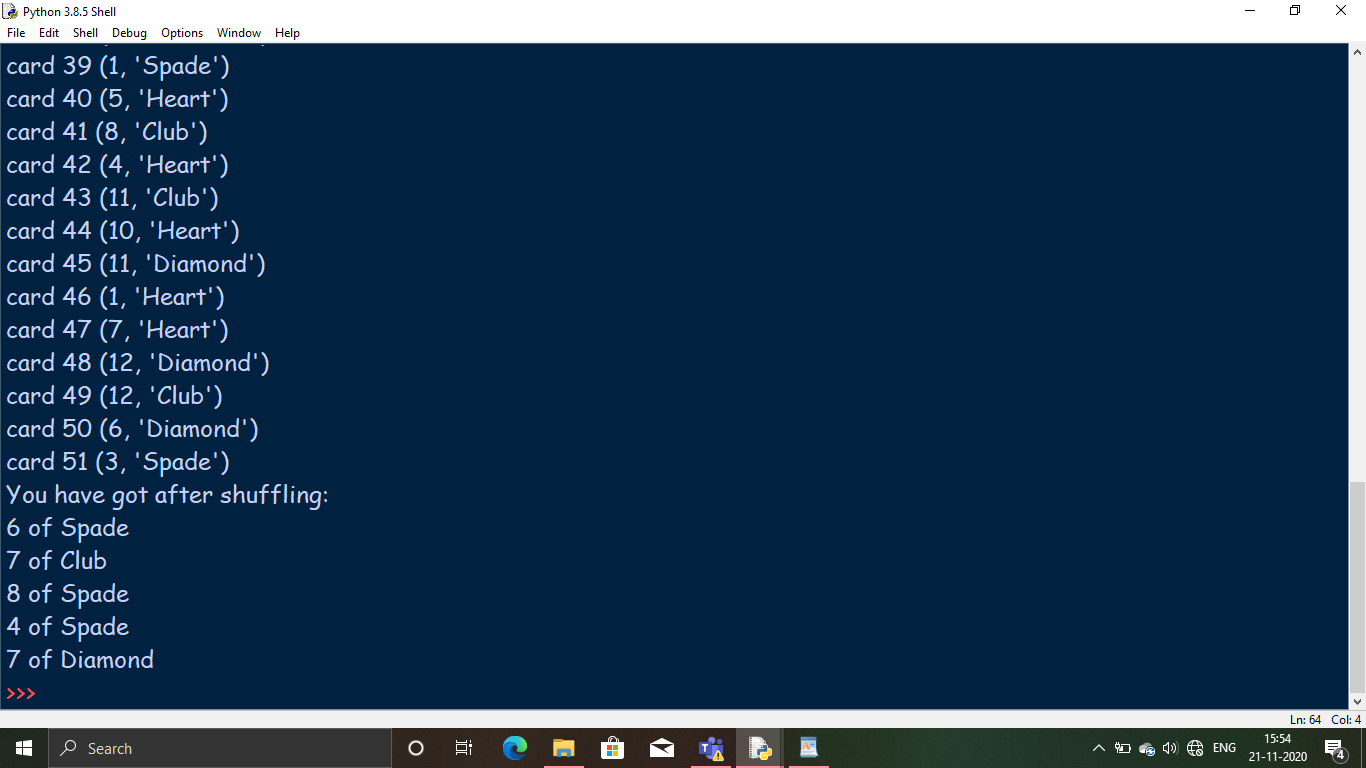
print(deck[i][0], "of", deck[i][1])

**OUTPUT**









**Conclusion :** We learned how to shuffle a deck of cards using Python Programming.

**b)** **Aim :** Solve the travelling salesman problem using artificial intelligence technique.

**TRAVELLING SALESMAN PROBLEM**

The travelling salesman problem (also called the traveling salesperson problem[1] or TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in theoretical computer science and operations research.

**NAIVE SOLUTION**

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate cost of every permutation and keep track of minimum cost permutation.

4) Return the permutation with minimum cost.

**Code :**

import random

import copy

# This class represent a state

class State:

# Create a new state

def \_\_init\_\_(self, route:[], distance:int=0):

self.route = route

self.distance = distance

# Compare states

def \_\_eq\_\_(self, other):

for i in range(len(self.route)):

if(self.route[i] != other.route[i]):

return False

return True

# Sort states

def \_\_lt\_\_(self, other):

return self.distance < other.distance

# Print a state

def \_\_repr\_\_(self):

return ('({0},{1})\n'.format(self.route, self.distance))

# Create a shallow copy

def copy(self):

return State(self.route, self.distance)

# Create a deep copy

def deepcopy(self):

return State(copy.deepcopy(self.route), copy.deepcopy(self.distance))

# Update distance

def update\_distance(self, matrix, home):

# Reset distance

self.distance = 0

# Keep track of departing city

from\_index = home

# Loop all cities in the current route

for i in range(len(self.route)):

self.distance += matrix[from\_index][self.route[i]]

from\_index = self.route[i]

# Add the distance back to home

self.distance += matrix[from\_index][home]

# This class represent a city (used when we need to delete cities)

class City:

# Create a new city

def \_\_init\_\_(self, index:int, distance:int):

self.index = index

self.distance = distance

# Sort cities

def \_\_lt\_\_(self, other):

return self.distance < other.distance

# Get best solution by distance

def get\_best\_solution\_by\_distance(matrix:[], home:int):

# Variables

route = []

from\_index = home

length = len(matrix) - 1

# Loop until route is complete

while len(route) < length:

# Get a matrix row

row = matrix[from\_index]

# Create a list with cities

cities = {}

for i in range(len(row)):

cities[i] = City(i, row[i])

# Remove cities that already is assigned to the route

del cities[home]

for i in route:

del cities[i]

# Sort cities

sorted = list(cities.values())

sorted.sort()

# Add the city with the shortest distance

from\_index = sorted[0].index

route.append(from\_index)

# Create a new state and update the distance

state = State(route)

state.update\_distance(matrix, home)

# Return a state

return state

# Create a population

def create\_population(matrix:[], home:int, city\_indexes:[], size:int):

# Create a gene pool

gene\_pool = city\_indexes.copy()

# Remove the home city

gene\_pool.pop(home)

# Create a population

population = []

for i in range(size):

# Shuffle the gene pool at random

random.shuffle(gene\_pool)

# Create a new state and update the distance

state = State(gene\_pool[:])

state.update\_distance(matrix, home)

# Add an individual to the population

population.append(state)

# Return a population

return population

# Ordered crossover (TSP)

def crossover(matrix:[], home:int, parents:[]):

# Copy parents

parent\_1 = parents[0].deepcopy()#clonning

parent\_2 = parents[1].deepcopy()

# Child gene parts

part\_1 = []

part\_2 = []

# Select the genes to copy from parents

a = int(random.random() \* len(parent\_1.route))

b = int(random.random() \* len(parent\_2.route))

start\_gene = min(a, b)

end\_gene = max(a, b)

# Get genes from parent 1

for i in range(start\_gene, end\_gene):

part\_1.append(parent\_1.route[i])

# Get genes from parent 2

part\_2 = [int(x) for x in parent\_2.route if x not in part\_1]

# Create a child

state = State(part\_1 + part\_2)

state.update\_distance(matrix, home)

# Return a child

return state

# Mutate a solution

def mutate(matrix:[], home:int, state:State, mutation\_rate:float=0.01):

# Create a copy of the state

mutated\_state = state.deepcopy()

# Loop all the states in a route

for i in range(len(mutated\_state.route)):

# Check if we should do a mutation

if(random.random() < mutation\_rate):

# Swap two cities

j = int(random.random() \* len(state.route))

city\_1 = mutated\_state.route[i]

city\_2 = mutated\_state.route[j]

mutated\_state.route[i] = city\_2

mutated\_state.route[j] = city\_1

# Update the distance

mutated\_state.update\_distance(matrix, home)

# Return a mutated state

return mutated\_state

# A genetic algorithm

def genetic\_algorithm(matrix:[], home:int, population:[], keep:int, mutation\_rate:float, generations:int):

# Loop to create new generations

for i in range(generations):

# Sort the population to get the fittest individuals at the beginning

population.sort()

# Generate parents

parents = []

for j in range(1, len(population)):

parents.append((population[j-1], population[j]))

# Generate childrens (breed) with crossover

children = []

for partners in parents:

children.append(crossover(matrix, home, partners))

# Mutate children

for j in range(len(children)):

children[j] = mutate(matrix, home, children[j], mutation\_rate)

# Keep the fittest n from the population

population = population[:keep]

# Add children to the population

population.extend(children)

# Sort the population

population.sort()

# Return the best state

return population[0]

# The main entry point for this module

def main():

# Cities to travel

cities = ['New York', 'Los Angeles', 'Chicago', 'Minneapolis', 'Denver', 'Dallas', 'Seattle', 'Boston', 'San Francisco', 'St. Louis', 'Houston', 'Phoenix', 'Salt Lake City']

city\_indexes = [0,1,2,3,4,5,6,7,8,9,10,11,12]

# Index of start location

home = 2 # Chicago

# Distances in miles between cities, same indexes (i, j) as in the cities array

matrix = [[0, 2451, 713, 1018, 1631, 1374, 2408, 213, 2571, 875, 1420, 2145, 1972],

[2451, 0, 1745, 1524, 831, 1240, 959, 2596, 403, 1589, 1374, 357, 579],

[713, 1745, 0, 355, 920, 803, 1737, 851, 1858, 262, 940, 1453, 1260],

[1018, 1524, 355, 0, 700, 862, 1395, 1123, 1584, 466, 1056, 1280, 987],

[1631, 831, 920, 700, 0, 663, 1021, 1769, 949, 796, 879, 586, 371],

[1374, 1240, 803, 862, 663, 0, 1681, 1551, 1765, 547, 225, 887, 999],

[2408, 959, 1737, 1395, 1021, 1681, 0, 2493, 678, 1724, 1891, 1114, 701],

[213, 2596, 851, 1123, 1769, 1551, 2493, 0, 2699, 1038, 1605, 2300, 2099],

[2571, 403, 1858, 1584, 949, 1765, 678, 2699, 0, 1744, 1645, 653, 600],

[875, 1589, 262, 466, 796, 547, 1724, 1038, 1744, 0, 679, 1272, 1162],

[1420, 1374, 940, 1056, 879, 225, 1891, 1605, 1645, 679, 0, 1017, 1200],

[2145, 357, 1453, 1280, 586, 887, 1114, 2300, 653, 1272, 1017, 0, 504],

[1972, 579, 1260, 987, 371, 999, 701, 2099, 600, 1162, 1200, 504, 0]]

# Get the best route by distance

state = get\_best\_solution\_by\_distance(matrix, home)

print('-- Best solution by distance --')

print(cities[home], end='')

for i in range(0, len(state.route)):

print(' -> ' + cities[state.route[i]], end='')

print(' -> ' + cities[home], end='')

print('\n\nTotal distance: {0} miles'.format(state.distance))

print()

# Run genetic search to find a better solution

population = create\_population(matrix, home, city\_indexes, 100)

state = genetic\_algorithm(matrix, home, population, 20, 0.01, 100)

print('-- Genetic algorithm solution --')

print(cities[home], end='')

for i in range(0, len(state.route)):

print(' -> ' + cities[state.route[i]], end='')

print(' -> ' + cities[home], end='')

print('\n\nTotal distance: {0} miles'.format(state.distance))

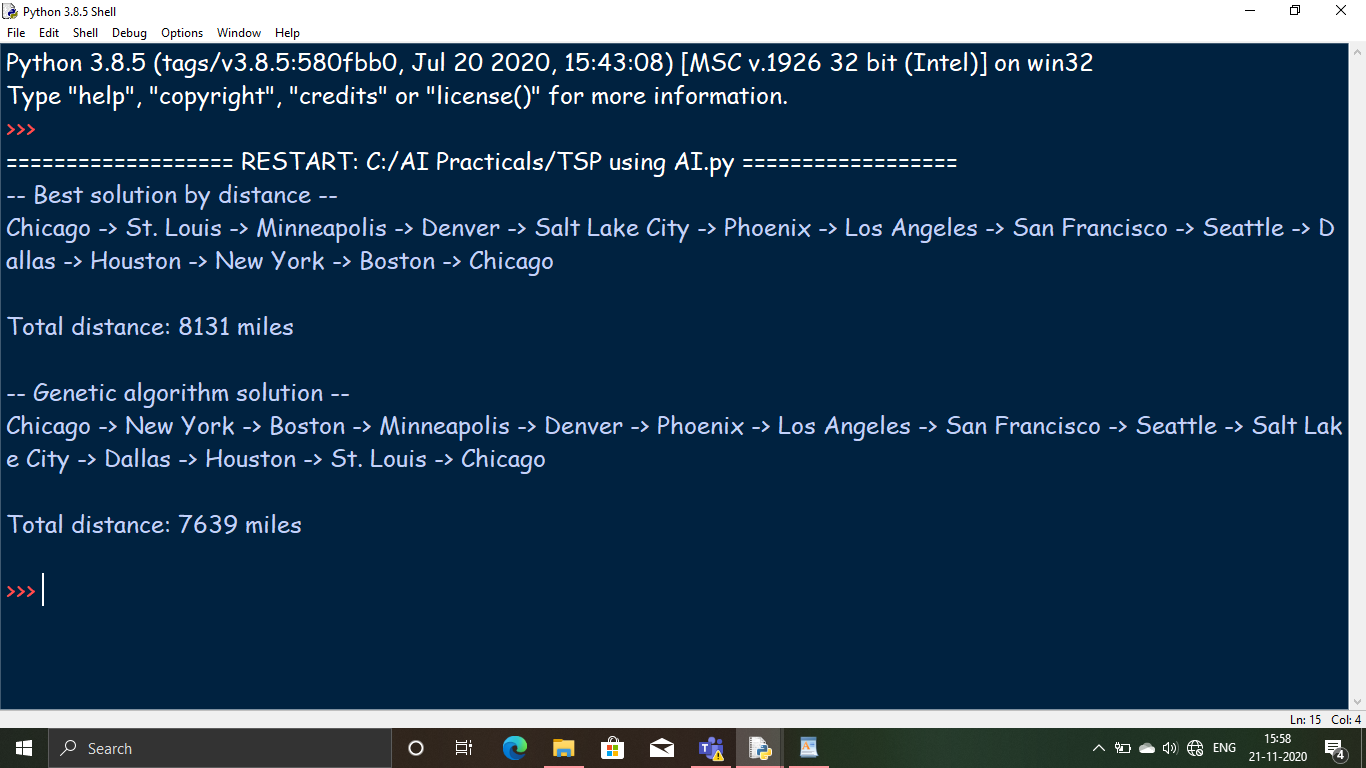
print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

# the result will take time for GA solution

**OUTPUT**

****

**Conclusion :** We learned how to implement the Travelling Salesman Problem using Python Programming.

Practical 8

**b)** **Aim :** Solve constraint satisfaction Problem.

**CONSTRAINT SATISFACTION PROBLEM**  
Constraint satisfaction problems (CSPs) are mathematical questions defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many seemingly unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. Constraint Programming (CP) is the field of research that specifically focuses on tackling with this kind of problems.

**Code :**

def constraint\_func(names, values):

return values[0] != values[1]

if \_\_name\_\_=='\_\_main\_\_':

# Specify the variables

names = ('Mark', 'Julia', 'Steve', 'Amanda', 'Brian',

'Joanne', 'Derek', 'Allan', 'Michelle', 'Kelly','Chris')

# Define the possible colors

colors = dict((name, ['red', 'green', 'blue', 'gray']) for name in names)

# Define the constraints

constraints = [

(('Mark', 'Julia'), constraint\_func),

(('Mark', 'Steve'), constraint\_func),

(('Julia', 'Steve'), constraint\_func),

(('Julia', 'Amanda'), constraint\_func),

(('Julia', 'Derek'), constraint\_func),

(('Julia', 'Brian'), constraint\_func),

(('Steve', 'Amanda'), constraint\_func),

(('Steve', 'Allan'), constraint\_func),

(('Steve', 'Michelle'), constraint\_func),

(('Amanda', 'Michelle'), constraint\_func),

(('Amanda', 'Joanne'), constraint\_func),

(('Amanda', 'Derek'), constraint\_func),

(('Brian', 'Derek'), constraint\_func),

(('Brian', 'Kelly'), constraint\_func),

(('Joanne', 'Michelle'), constraint\_func),

(('Joanne', 'Amanda'), constraint\_func),

(('Joanne', 'Derek'), constraint\_func),

(('Joanne', 'Chris'), constraint\_func),

(('Derek', 'Kelly'), constraint\_func),

(('Derek', 'Brian'), constraint\_func),

(('Derek', 'Julia'), constraint\_func),

(('Derek', 'Amanda'), constraint\_func),

(('Derek', 'Michelle'), constraint\_func),

(('Derek', 'Chris'), constraint\_func),

]

# Solve the problem

problem = CspProblem(names, colors, constraints)

# Print the solution

output = backtrack(problem)

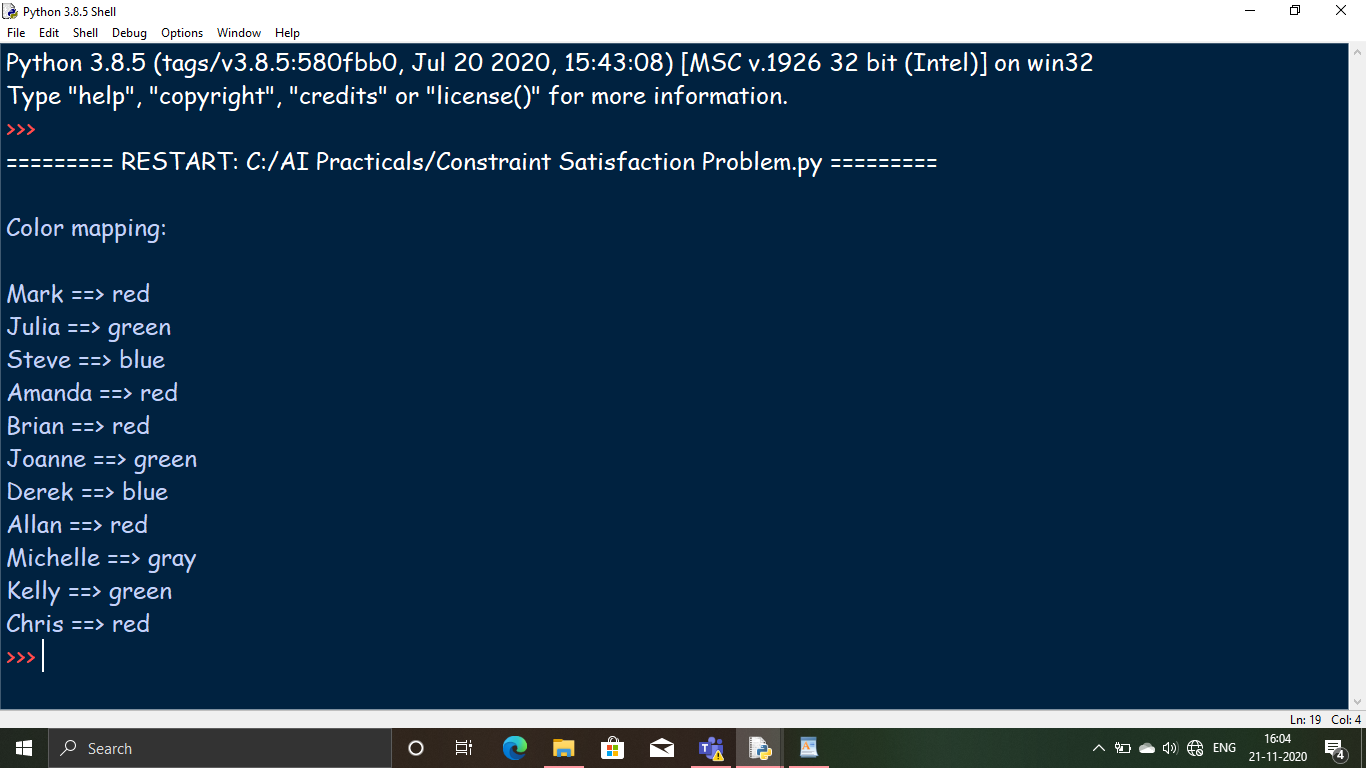
print('\nColor mapping:\n')

for k, v in output.items():

print(k, '==>', v)

# In[ ]:

**OUTPUT**

****

**Conclusion :** We learned how to solve the constraint satisfaction problem using Python Programming.

Practical 9

**a)** **Aim :** Derive te expressions based on Associative Law.

**ASSOCIATIVE LAW**

Associative law, in mathematics, either of two laws relating to number operations of addition and multiplication, stated symbolically: a + (b + c) = (a + b) + c, and a(bc) = (ab)c; that is, the terms or factors may be associated in any way desired. While associativity holds for ordinary arithmetic with real or imaginary numbers, there are certain applications—such as nonassociative algebras—in which it does not hold.

**Code :**

student(raj).

student(rahul).

hscstudent(raj).

sscstudent(raj).

sscstudent(rahul).

nondetermhscstudent(X).

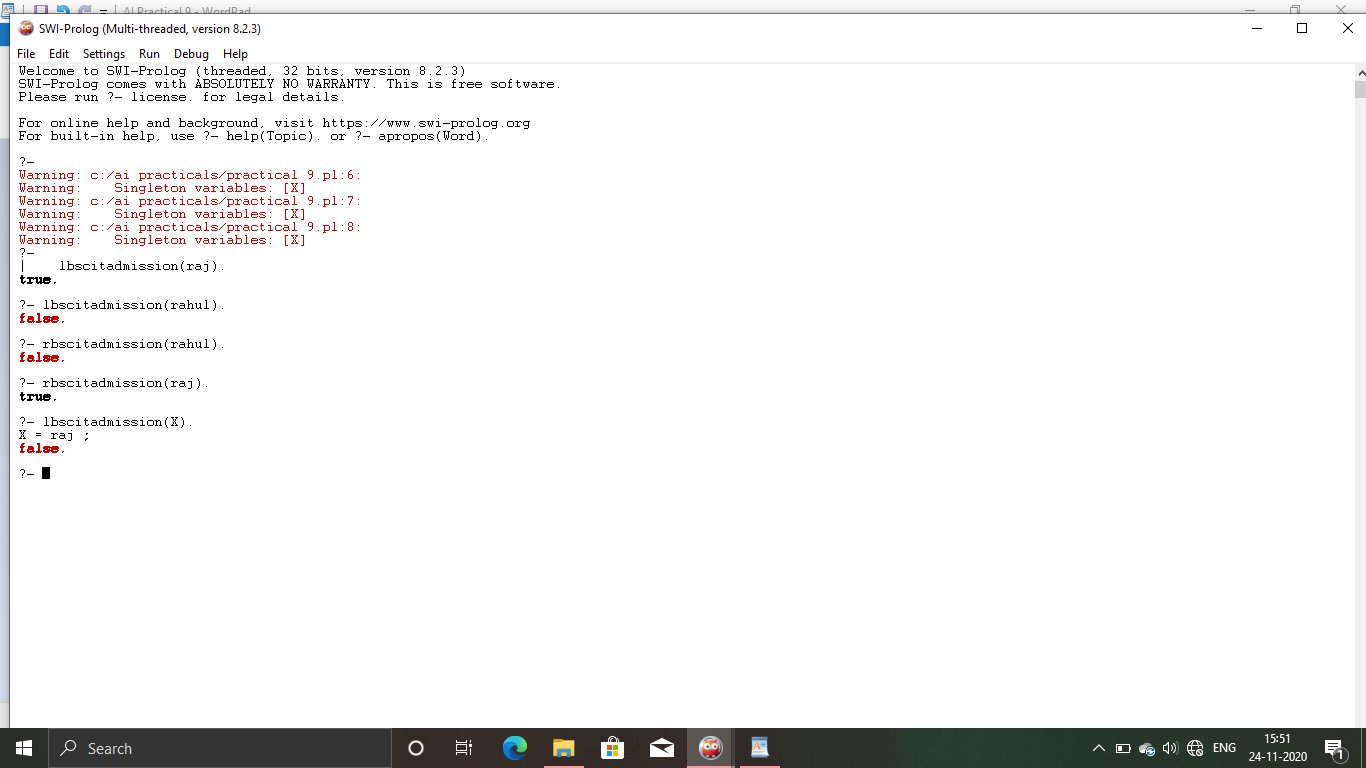
nondetermsscstudent(X).

nondetermstudent(X).

lbscitadmission(X):-((student(X),hscstudent(X)),sscstudent(X)).

rbscitadmission(X):-(student(X),(hscstudent(X),sscstudent(X))).

**OUTPUT**



**Conclusion :** We learned how to express Associative Law using Prolog Application.

**b)** **Aim :** Derive the expressions based on Distributive Law.

**DISTRIBUTIVE LAW**

In mathematics, the distributive property of binary operations generalizes the distributive law from Boolean algebra and elementary algebra. In propositional logic, distribution refers to two valid rules of replacement. The rules allow one to reformulate conjunctions and disjunctions within logical proofs.

**Code :**

male(raj).

male(aayush).

doctor(raj).

engineer(aayush).

nondetermmale(X).

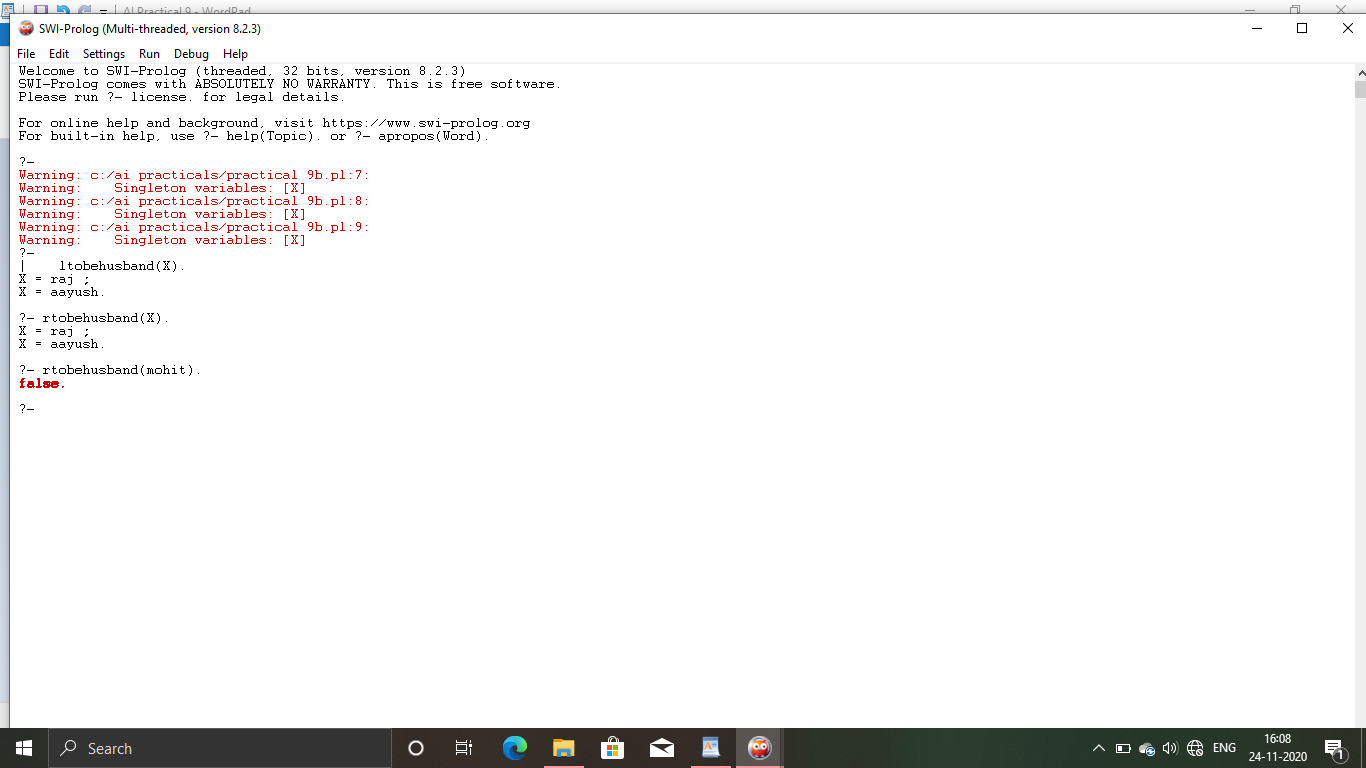
nondetermengineer(X).

nondetermdoctor(X).

ltobehusband(X):-(male(X),doctor(X);engineer(X)).

rtobehusband(X):-((male(X),doctor(X));(male(X),engineer(X))).

**OUTPUT**

****

**Conclusion :** We learned how to express Distributive Law using Prolog Application.

Practical 10

**a)** **Aim :** Write a program to derive the Predicate.

(For e.g: Sachin is batsman, batsman is cricketer)->Sachin is cricketer.

**PREDICATE**

The predicate of a sentence is the part that modifies the subject in some way. Because the subject is the person, place, or thing that a sentence is about, the predicate must contain a verb explaining what the subject does and can also include a modifier.

**Code :**

batsman(sachin,batsman).

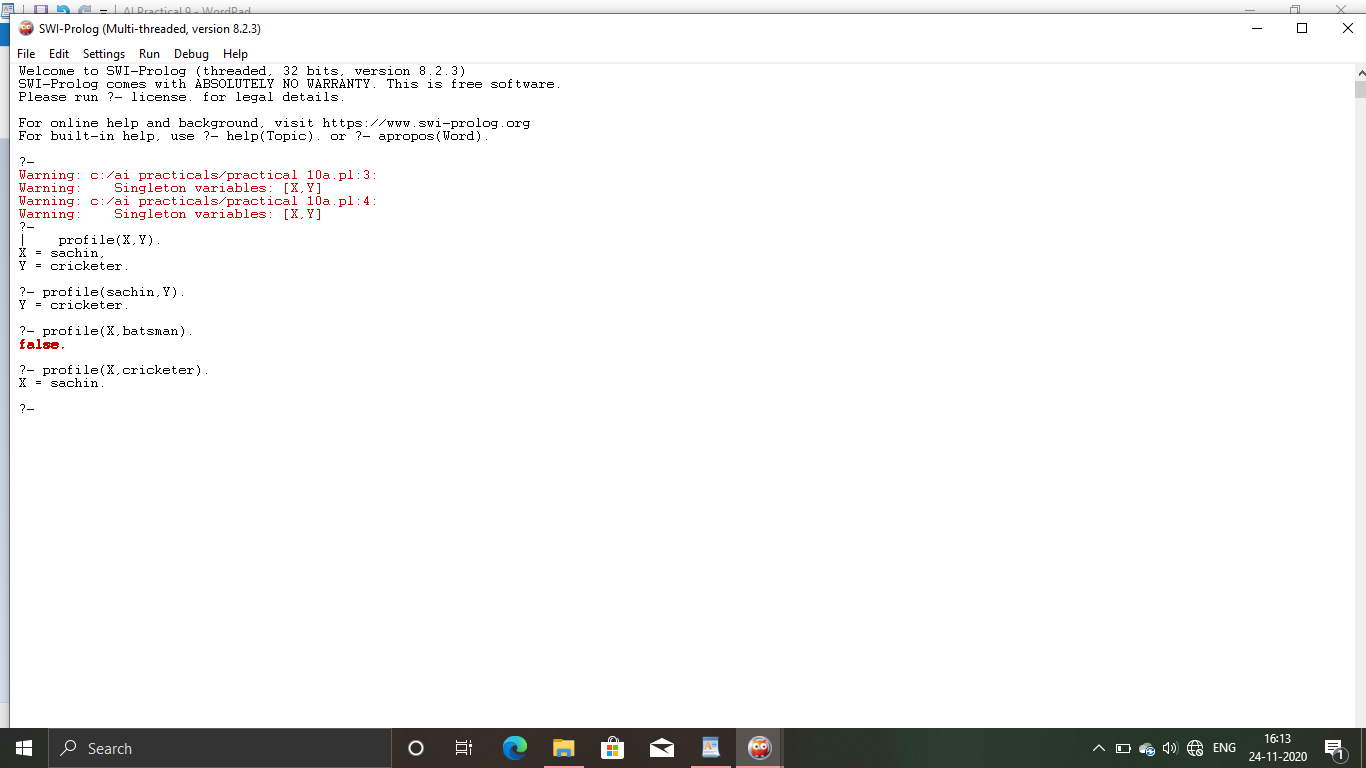
cricketer(batsman,cricketer).

nondetermbatsman(X,Y).

nondetermcricketer(X,Y).

profile(X,Y):-batsman(X,Z),cricketer(Z,Y).

**OUTPUT**

****

**Conclusion :** We learned how to derive predicate using Prolog Application.

**b)** **Aim :** Write a program which contains three predicates :male,female, parent. Make rules for following family relations: father,mother,grandfather,grandmother,brother,sister,uncle,aunt,nephew and neice,cousin.

**Code :**

male(shankar).

male(ulhas).

male(satish).

male(saurabh).

male(prashant).

female(umabai).

female(mrunal).

female(sadhna).

female(swati).

parent(shankar,umabai,ulhas).

parent(shankar,umabai,satish).

parent(ulhas,mrunal,prashant).

parent(satish,sadhna,saurabh).

parent(satish,sadhna,swati).

brother(ulhas,satish).

brother(satish,ulhas).

brother(prashant,saurabh).

brother(saurabh,prashant).

sister(swati,saurabh).

sister(swati,prashant).

father(X,Y):-parent(X,Z,Y).

mother(X,Y):-parent(Z,X,Y).

son(X,Y,Z):-male(X),father(Y,X),mother(Z,X).

daughter(X,Y,Z):-female(X),father(Y,X),mother(Z,X).

wife(X,Y):-female(X),parent(Y,X,Z).

grandfather(X,Y):-male(X),father(X,Z),father(Z,Y).

grandmother(X,Y):-female(X),mother(X,Z),father(Z,Y).

uncle(X,Y):-male(X),((father(Z,Y),father(A,Z),father(A,X));(mother(Z,Y),father(A,Z),father(A,X))).

aunt(X,Y):-wife(X,Z),uncle(Z,Y).

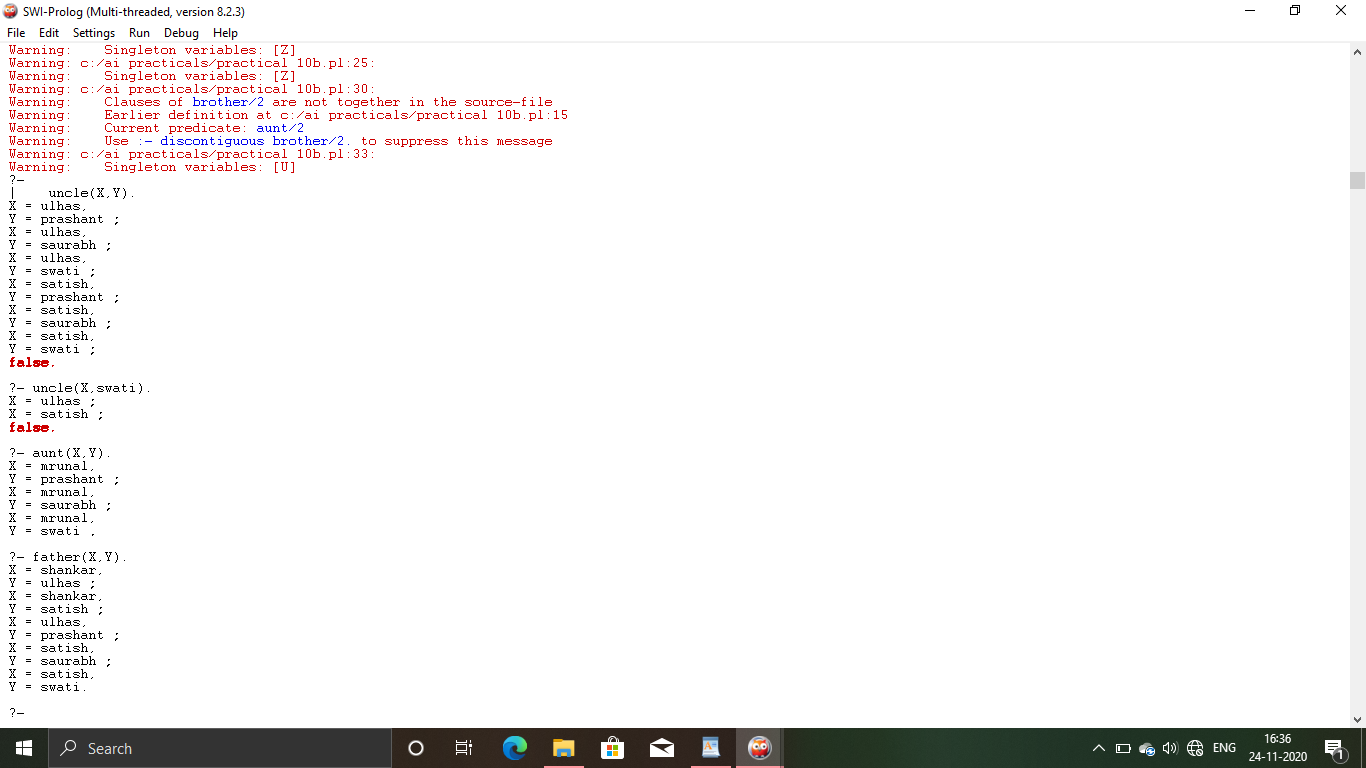
brother(X,Y):-male(X),father(Z,X),father(Z,Y).

cousin(X,Y):-father(Z,X),brother(Z,W),father(W,Y).

ancestor(X,Y,Z):-parent(X,Y,Z).

ancestor(X,Y,Z):-parent(X,Y,W),ancestor(W,U,Z).

**OUTPUT**

****

**Conclusion :** We learned how to derive predicates using Prolog Application.