**Lab File**

On

**Artificial Intelligence**

Submitted To

**Mr. Soumya Ranjan**

**Amity University Uttar Pradesh**



In partial fulfilment of the requirements for the award degree of

Bachelor of Technology

In

Artificial Intelligence

by

**Priyansh Jain**

**A023119820019**

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

AMITY UNIVERSITY UTTAR PRADESH

NOIDA (U.P)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.No** | **Category of Assignment** | **Code** | **Exp. No.** | **Name of Experiment** | **Date of Allotment of experiment** | **Dat of**  **Evaluation** | **Max.**  **Marks** | **Marks obtained** | **Signature of Faculty** |
| 1. | **Mandatory Experiment\*** | **LR (10)** | **1** |  |  |  |  |  |  |
| 2. | **Mandatory Experiment\*** | **2** |  |  |  |  |  |  |
| 3. | **Mandatory Experiment\*** | **3** |  |  |  |  |  |  |
| 4. | **Mandatory Experiment\*** | **4** |  |  |  |  |  |  |
| 5. | **Mandatory Experiment\*** | **5** |  |  |  |  |  |  |
| 6. | **Mandatory Experiment\*** | **6** |  |  |  |  |  |  |
| 7 | **Mandatory Experiment\*** | **7** |  |  |  |  |  |  |
| 8. | **Mandatory Experiment\*** | **8** |  |  |  |  |  |  |
| 9. | **Mandatory Experiment\*** | **9** |  |  |  |  |  |  |
| 10. | **Mandatory Experiment\*** | **10** |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
| 11. | **Viva** | **Viva (5)** |  |  |  |  |  |  |  |

INDEX

**Experiment Number: 1**

**AIM:** Write a program to implement A\* algorithm in python

**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

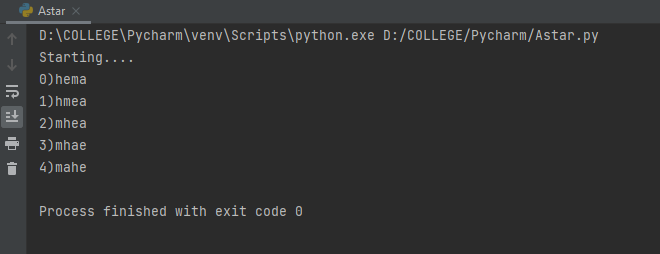
It is a searching algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A\* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal. It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem.

**CODE:**

from queue import PriorityQueue  
  
  
# Creating Base Class  
class State(object):  
 def \_\_init\_\_(self, value, parent, start=0, goal=0):  
 self.children = []  
 self.parent = parent  
 self.value = value  
 self.dist = 0  
 if parent:  
 self.start = parent.start  
 self.goal = parent.goal  
 self.path = parent.path[:]  
 self.path.append(value)  
  
 else:  
 self.path = [value]  
 self.start = start  
 self.goal = goal  
  
 def GetDistance(self):  
 pass  
  
 def CreateChildren(self):  
 pass  
  
  
# Creating subclass  
class State\_String(State):  
 def \_\_init\_\_(self, value, parent, start=0, goal=0):  
 super(State\_String, self).\_\_init\_\_(value, parent, start, goal)  
 self.dist = self.GetDistance()  
  
 def GetDistance(self):  
 if self.value == self.goal:  
 return 0  
 dist = 0  
 for i in range(len(self.goal)):  
 letter = self.goal[i]  
 dist += abs(i - self.value.index(letter))

return dist  
  
 def CreateChildren(self):  
 if not self.children:  
 for i in range(len(self.goal) - 1):  
 val = self.value  
 val = val[:i] + val[i + 1] + val[i] + val[i + 2:]  
 child = State\_String(val, self)  
 self.children.append(child)  
  
  
# Creating a class that hold the final magic  
class A\_Star\_Solver:  
 def \_\_init\_\_(self, start, goal):  
 self.path = []  
 self.vistedQueue = []  
 self.priorityQueue = PriorityQueue()  
 self.start = start  
 self.goal = goal  
  
 def Solve(self):  
 startState = State\_String(self.start, 0, self.start, self.goal)  
  
 count = 0  
 self.priorityQueue.put((0, count, startState))  
 while (not self.path and self.priorityQueue.qsize()):  
 closesetChild = self.priorityQueue.get()[2]  
 closesetChild.CreateChildren()  
 self.vistedQueue.append(closesetChild.value)  
 for child in closesetChild.children:  
 if child.value not in self.vistedQueue:  
 count += 1  
 if not child.dist:  
 self.path = child.path  
 break  
 self.priorityQueue.put((child.dist, count, child))  
 if not self.path:  
 print("Goal Of is not possible !" + self.goal)  
 return self.path  
  
  
# Calling all the existing stuffs  
if \_\_name\_\_ == "\_\_main\_\_":  
 start1 = "hema"  
 goal1 = "mahe"  
 print("Starting....")  
 a = A\_Star\_Solver(start1, goal1)  
 a.Solve()  
 for i in range(len(a.path)):  
 print("{0}){1}".format(i, a.path[i]))

**OUTPUT:**

****

**CONCLUSION:**

A\* algorithm has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 2**

**AIM:** Write a program to implement Single Player Game

**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

A single-player game is usually a game that can only be played by one person. It also include games which are played with computer set as the second player. Some examples of single-player game are:

* + - 1. Solitare
      2. Rock Paper Scissors
      3. Backgammon
      4. 21 Number Game

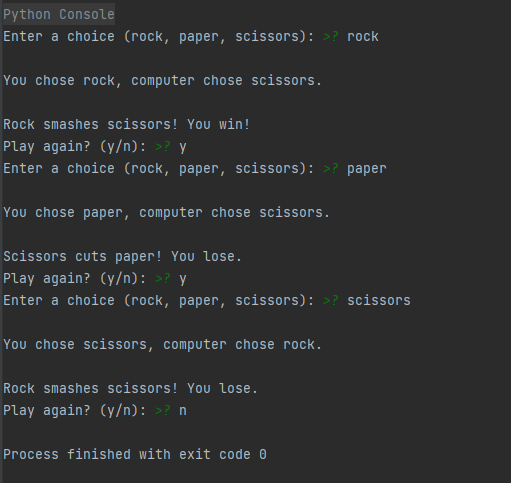
Rock Paper Scissors: Rock paper scissors is a hand game for two or more players. Participants say “rock, paper, scissors” and then simultaneously form their hands into the shape of a rock (a fist), a piece of paper (palm facing downward), or a pair of scissors (two fingers extended). The rules are straightforward:

* **Rock** smashes scissors.
* **Paper** covers rock.
* **Scissors** cut paper.

**CODE:**

import random  
  
while True:  
 user\_action = input("Enter a choice (rock, paper, scissors): ")  
 possible\_actions = ["rock", "paper", "scissors"]  
 computer\_action = random.choice(possible\_actions)  
 print(f"\nYou chose {user\_action}, computer chose {computer\_action}.\n")  
  
 if user\_action == computer\_action:  
 print(f"Both players selected {user\_action}. It's a tie!")  
 elif user\_action == "rock":  
 if computer\_action == "scissors":  
 print("Rock smashes scissors! You win!")  
 else:  
 print("Paper covers rock! You lose.")  
 elif user\_action == "paper":  
 if computer\_action == "rock":  
 print("Paper covers rock! You win!")  
 else:  
 print("Scissors cuts paper! You lose.")  
 elif user\_action == "scissors":  
 if computer\_action == "paper":  
 print("Scissors cuts paper! You win!")  
 else:  
 print("Rock smashes scissors! You lose.")  
  
 play\_again = input("Play again? (y/n): ")  
 if play\_again.lower() != "y":  
 break

**OUTPUT:**

****

**CONCLUSION:**

Single Player Game has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

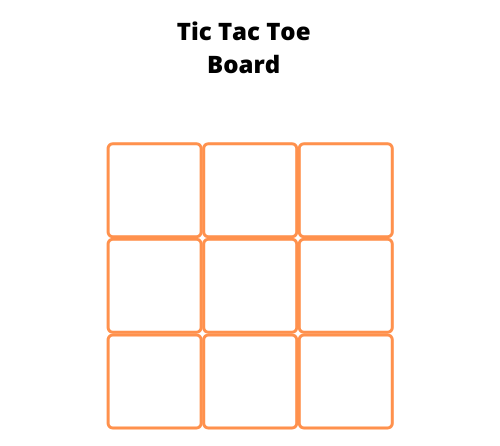
**Experiment Number: 3**

**AIM:** Write a program to implement Tic-Tac-Toe game problem

**LANGUAGE USED:** Python programming language

**THEORY:**

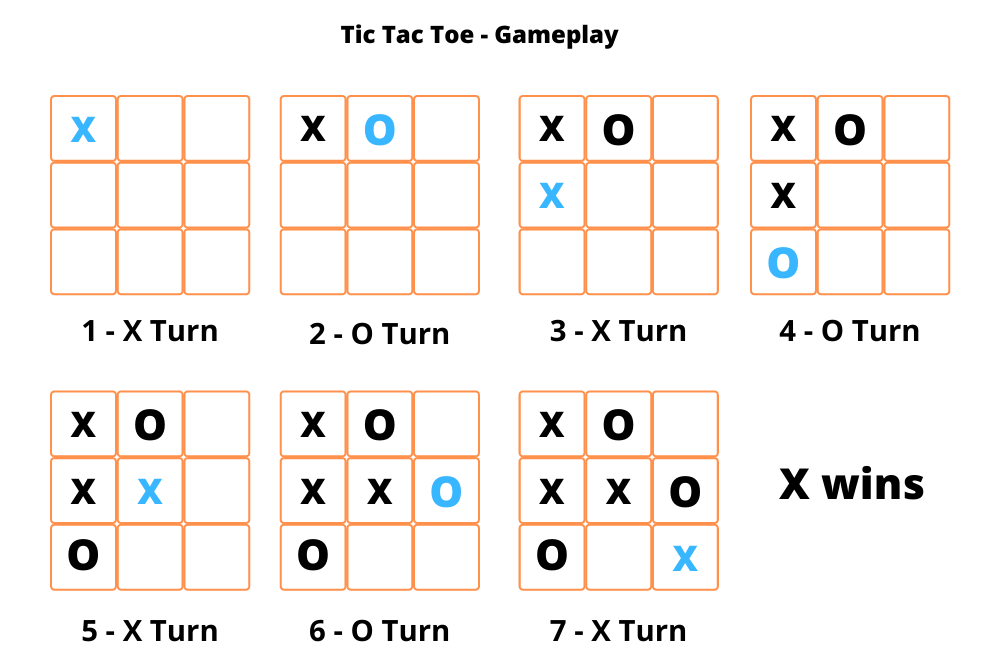
There will be two players in a game. Two signs represent each player. The general signs used in the game are **X**and **O**. Finally, there will be a board with **9**boxes. See the tic-tac-toe board visually.



The gameplay will be as follows.

* First, one user will place their sign in one of the available empty boxes.
* Next, the second user will place their sign in one of the available empty boxes.
* The goal of the players is to place their respective signs completely row-wise or column-wise, or diagonally.
* The game goes on until a player wins the game or it ended up in a draw by filling all boxes without a winning match.

SAMPLE GAMEPLAY:



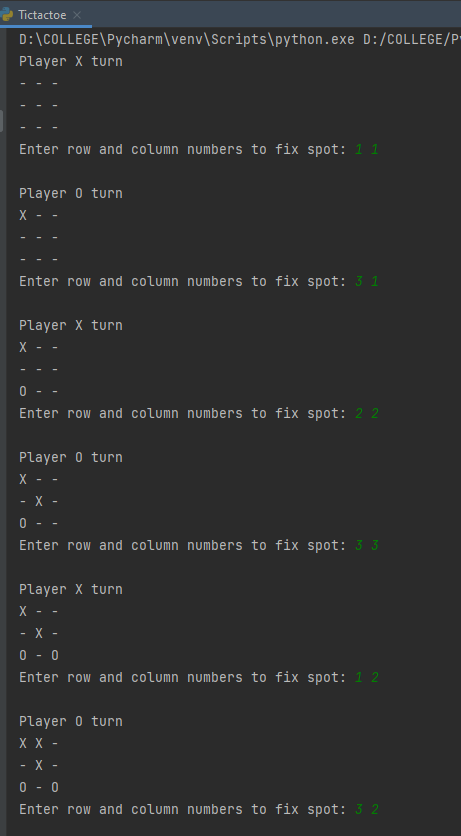
**CODE:**

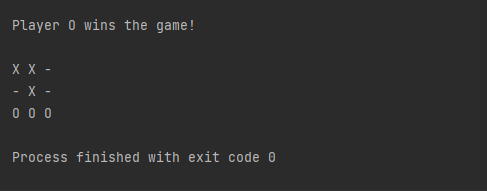
import random  
  
class TicTacToe:  
 def \_\_init\_\_(self):  
 self.board = []  
  
 def create\_board(self):  
 for i in range(3):  
 row = []  
 for j in range(3):  
 row.append('-')  
 self.board.append(row)  
  
 def get\_random\_first\_player(self):  
 return random.randint(0, 1)  
  
 def fix\_spot(self, row, col, player):  
 self.board[row][col] = player  
  
 def is\_player\_win(self, player):  
 win = None  
  
 n = len(self.board)  
 # checking rows  
 for i in range(n):  
 win = True  
 for j in range(n):  
 if self.board[i][j] != player:  
 win = False  
 break  
 if win:  
 return win  
 # checking columns  
 for i in range(n):  
 win = True  
 for j in range(n):  
 if self.board[j][i] != player:  
 win = False  
 break  
 if win:  
 return win  
 # checking diagonals  
 win = True  
 for i in range(n):  
 if self.board[i][i] != player:  
 win = False  
 break  
 if win:  
 return win  
  
 win = True  
 for i in range(n):  
 if self.board[i][n - 1 - i] != player:  
 win = False  
 break  
 if win:  
 return win  
 return False  
 for row in self.board:  
 for item in row:  
 if item == '-':  
 return False  
 return True

def is\_board\_filled(self):  
 for row in self.board:  
 for item in row:  
 if item == '-':  
 return False  
 return True  
  
 def swap\_player\_turn(self, player):  
 return 'X' if player == 'O' else 'O'  
  
 def show\_board(self):  
 for row in self.board:  
 for item in row:  
 print(item, end=" ")  
 print()  
  
 def start(self):  
 self.create\_board()  
  
 player = 'X' if self.get\_random\_first\_player() == 1 else 'O'  
 while True:  
 print(f"Player {player} turn")  
  
 self.show\_board()

# taking user input  
 row, col = list(  
 map(int, input("Enter row and column numbers to fix spot: ").split()))  
 print()  
 # fixing the spot  
 self.fix\_spot(row - 1, col - 1, player)  
  
 # checking whether current player is won or not  
 if self.is\_player\_win(player):  
 print(f"Player {player} wins the game!")  
 break  
 # checking whether the game is draw or not  
 if self.is\_board\_filled():  
 print("Match Draw!")  
 break  
 # swapping the turn  
 player = self.swap\_player\_turn(player)  
 # showing the final view of board  
 print()  
 self.show\_board()  
  
  
# starting the game  
tic\_tac\_toe = TicTacToe()  
tic\_tac\_toe.start()

**OUTPUT:**

****

****

**CONCLUSION:** Single Player Game has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 4**

**AIM:** Implement Brute force solution to the Knapsack problem in Python

**PLATFORM/TOOLS USED:**

Python programming language

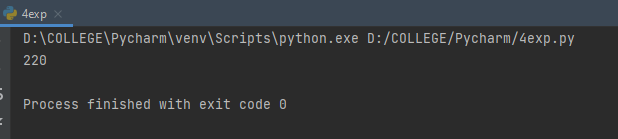
**THEORY:**

A problem from the optimization combinational related issues is known as a knapsack problem. This problem often arises in our daily life in resources allocations where the decisions have to be made from the set of items that are non-divisible tasks or projects which are placed under a fixed budget constraints or time constraint. In the **knapsack problem**, you need to pack a set of items, with given values and sizes (such as weights or volumes), into a container with a maximum capacity. If the total size of the items exceeds the capacity, you can't pack them all. In that case, the problem is to choose a subset of the items of maximum total value that will fit in the container.

**CODE:**

def knapSack(W, wt, val, n):  
 # initial conditions  
 if n == 0 or W == 0 :  
 return 0  
 # If weight is higher than capacity then it is not included  
 if (wt[n-1] > W):  
 return knapSack(W, wt, val, n-1)  
 # return either nth item being included or not  
 else:  
 return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),  
 knapSack(W, wt, val, n-1))  
# To test above function  
val = [60, 100, 120]  
wt = [10, 20, 30]  
W = 50  
n = len(val)  
print (knapSack(W, wt, val, n))

**OUTPUT:**

****

**CONCLUSION:**

Brute force implementation of Knapsack problem has been derived successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 5**

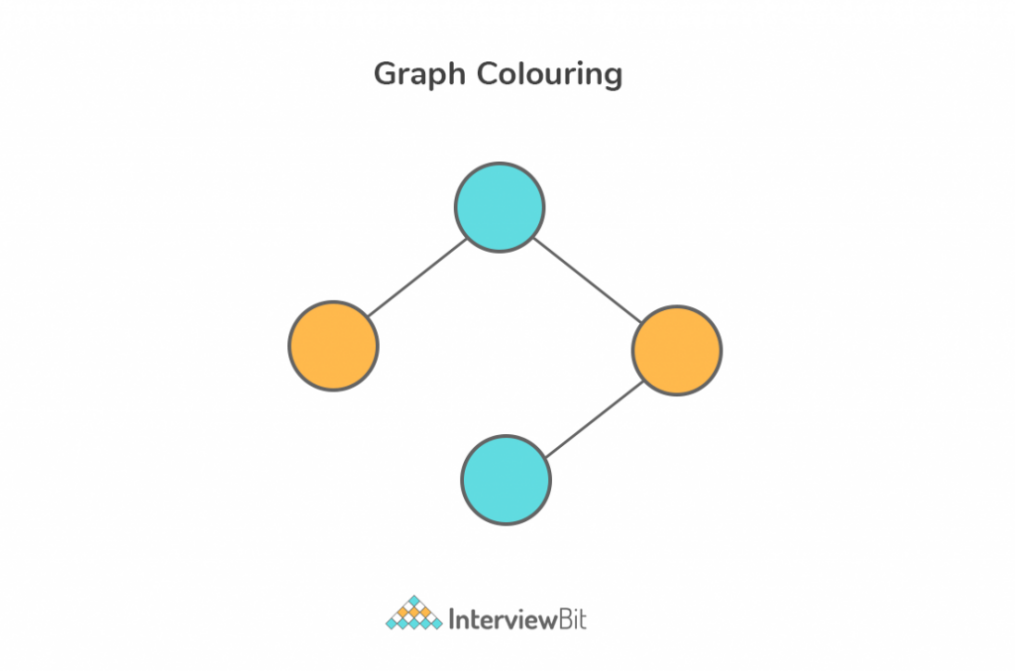
**AIM:** Implement Graph coloring problem using python

**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

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

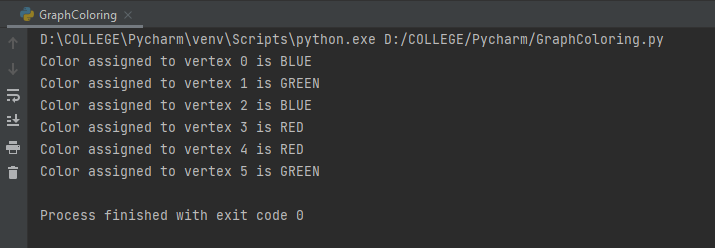


**CODE:**

class Graph:  
 def \_\_init\_\_(self, edges, n):  
 self.adjList = [[] for \_ in range(n)]  
  
 # add edges to the undirected graph  
 for (src, dest) in edges:  
 self.adjList[src].append(dest)  
 self.adjList[dest].append(src)  
  
# Function to assign colors to vertices of a graph  
def colorGraph(graph, n):  
 result = {}  
 # assign a color to vertex one by one  
 for u in range(n):  
 assigned = set([result.get(i) for i in graph.adjList[u] if i in result])  
 # check for the first free color  
 color = 1  
 for c in assigned:  
 if color != c:  
 break  
 color = color + 1  
 result[u] = color  
  
 for v in range(n):  
 print(f'Color assigned to vertex {v} is {colors[result[v]]}')

# Greedy coloring of a graph  
if \_\_name\_\_ == '\_\_main\_\_':  
 # Add more colors for graphs with many more vertices  
 colors = ['', 'BLUE', 'GREEN', 'RED', 'YELLOW', 'ORANGE', 'PINK',  
 'BLACK', 'BROWN', 'WHITE', 'PURPLE', 'VOILET']  
  
 # List of graph edges as per the above diagram  
 edges = [(0, 1), (0, 4), (0, 5), (4, 5), (1, 4), (1, 3), (2, 3), (2, 4)]  
  
 # total number of nodes in the graph (labelled from 0 to 5)  
 n = 6  
 # build a graph from the given edges  
 graph = Graph(edges, n)  
 # color graph using the greedy algorithm  
 colorGraph(graph, n)

**OUTPUT:**



**CONCLUSION:**

Graph Colouring problem has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 6**

**AIM:** Write a program to implement BFS for water jug problem using Python

**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

Water Jug Problem:

You are given an m litre jug and a n litre jug. Both the jugs are initially empty. The jugs don’t have markings to allow measuring smaller quantities. You have to use the jugs to measure d litres of water where d is less than n.

(X, Y) corresponds to a state where X refers to the amount of water in Jug1 and Y refers to the amount of water in Jug2.   
Determine the path from the initial state (xi, yi) to the final state (xf, yf), where (xi, yi) is (0, 0) which indicates both Jugs are initially empty and (xf, yf) indicates a state which could be (0, d) or (d, 0).

The operations you can perform are:

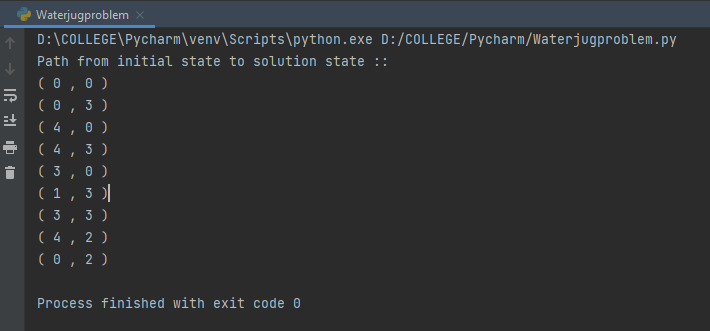
1. Empty a Jug, (X, Y)->(0, Y) Empty Jug 1
2. Fill a Jug, (0, 0)->(X, 0) Fill Jug 1
3. Pour water from one jug to the other until one of the jugs is either empty or full, (X, Y) -> (X-d, Y+d)

**CODE:**

from collections import deque  
  
def BFS(a, b, target):  
 m = {}  
 isSolvable = False  
 path = []  
  
 # Queue to maintain states  
 q = deque()  
  
 # Initialing with initial state  
 q.append((0, 0))  
  
 while (len(q) > 0):  
  
 # Current state  
 u = q.popleft()  
 # q.pop() #pop off used state  
 # If this state is already visited  
 if ((u[0], u[1]) in m):  
 continue  
  
 # Doesn't met jug constraints  
 if ((u[0] > a or u[1] > b or  
 u[0] < 0 or u[1] < 0)):  
 continue  
 path.append([u[0], u[1]])  
  
 # Marking current state as visited  
 m[(u[0], u[1])] = 1

# If we reach solution state, put ans=1  
 if (u[0] == target or u[1] == target):  
 isSolvable = True  
  
 if (u[0] == target):  
 if (u[1] != 0):  
 # Fill final state  
 path.append([u[0], 0])  
 else:  
 if (u[0] != 0):  
 # Fill final state  
 path.append([0, u[1]])  
  
 # Print the solution path  
 sz = len(path)  
 for i in range(sz):  
 print("(", path[i][0], ",",  
 path[i][1], ")")  
 break  
  
 # If we have not reached final state  
 # then, start developing intermediate  
 # states to reach solution state  
 q.append([u[0], b]) # Fill Jug2  
 q.append([a, u[1]]) # Fill Jug1  
  
 for ap in range(max(a, b) + 1):  
  
 # Pour amount ap from Jug2 to Jug1  
 c = u[0] + ap  
 d = u[1] - ap  
  
 # Check if this state is possible or not  
 if (c == a or (d == 0 and d >= 0)):  
 q.append([c, d])  
  
 # Pour amount ap from Jug 1 to Jug2  
 c = u[0] - ap  
 d = u[1] + ap  
  
 # Check if this state is possible or not  
 if ((c == 0 and c >= 0) or d == b):  
 q.append([c, d])  
  
 # Empty Jug2  
 q.append([a, 0])  
 # Empty Jug1  
 q.append([0, b])  
 # No, solution exists if ans=0  
 if (not isSolvable):  
 print("No solution")  
# Driver code  
if \_\_name\_\_ == '\_\_main\_\_':  
 Jug1, Jug2, target = 4, 3, 2  
 print("Path from initial state "  
 "to solution state ::")  
  
 BFS(Jug1, Jug2, target)

**OUTPUT:**

****

**CONCLUSION:**

Water Jug problem using BFS has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 7**

**AIM:** Write a program to implement DFS using Python

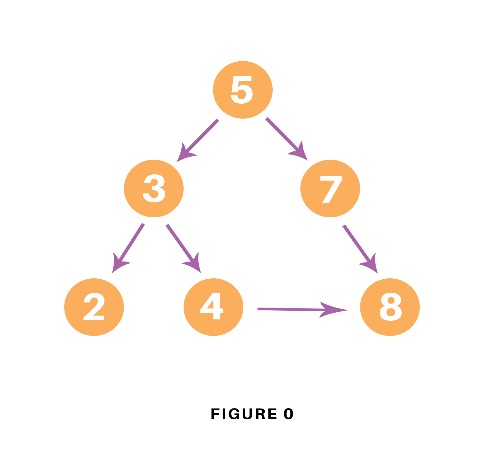
**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

**The Depth-First Search is a recursive algorithm that uses the concept of backtracking.** It involves thorough searches of all the nodes by going ahead if potential, else by backtracking. Here, the word backtrack means once you are moving forward and there are not any more nodes along the present path, you progress backward on an equivalent path to seek out nodes to traverse. All the nodes are progressing to be visited on the current path until all the unvisited nodes are traversed after which subsequent paths are going to be selected.

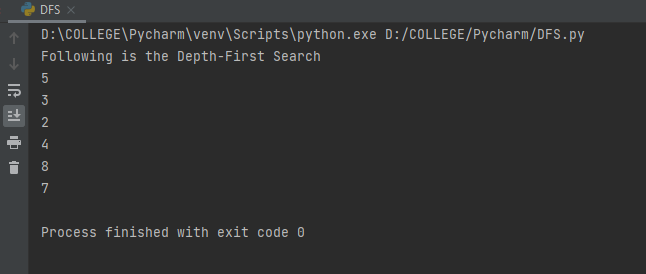
The sample that I have implemented with the code is as follows:



**CODE:**

# Using a Python dictionary to act as an adjacency list  
graph = {  
 '5' : ['3','7'],  
 '3' : ['2', '4'],  
 '7' : ['8'],  
 '2' : [],  
 '4' : ['8'],  
 '8' : []  
}  
visited = set() # Set to keep track of visited nodes of graph.  
  
def dfs(visited, graph, node): #function for dfs  
 if node not in visited:  
 print (node)  
 visited.add(node)  
 for neighbour in graph[node]:  
 dfs(visited, graph, neighbour)  
  
# Driver Code  
print("Following is the Depth-First Search")  
dfs(visited, graph, '5')

**OUTPUT:**

****

**CONCLUSION:**

DFS has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 8**

**AIM:** Tokenization of word and Sentences with the help of NLTK package

**PLATFORM/TOOLS USED:**

Python programming language

NLTK package

**THEORY:**

Tokenization is a common task in Natural Language Processing (NLP). It is a fundamental step in both traditional NLP methods like Count Vectorizer and Advanced Deep Learning-based architectures like Transformers.

Tokenization is a way of separating a piece of text into smaller units called tokens. Here, tokens can be either words, characters, or sub-words. Hence, tokenization can be broadly classified into 3-types – word, character, and sub-word (n-gram characters) tokenization.

### Word Tokenization: Word tokenization splits a piece of text into individual words based on a certain delimiter.

Character Tokenization: Character tokenization splits apiece of text into a set of characters.

Sub-word Tokenization: Sub-word tokenization splits the piece of text into sub-words.

**CODE:**

**Installation of NLTK package:**

****

****

**Code #1: Sentence Tokenization –** Splitting sentences in the paragraph

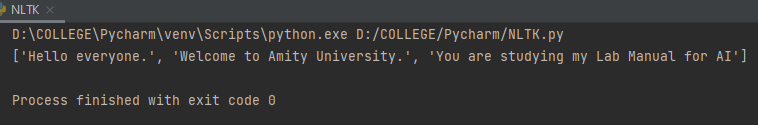
from nltk.tokenize import sent\_tokenize  
text = "Hello everyone. Welcome to Amity University. You are studying my Lab Manual for AI"  
print(sent\_tokenize(text))

**Code #2: Word Tokenization –** Splitting words in a sentence.

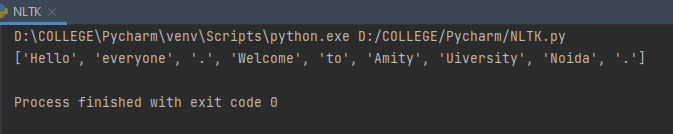
from nltk.tokenize import word\_tokenize  
text = "Hello everyone. Welcome to Amity Uiversity Noida."  
print(word\_tokenize(text))

**OUTPUT:**

**Code #1:**

****

**Code #2:**

****

**CONCLUSION:**

Tokenization of word and Sentences with the help of NLTK package has been implemented successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

**Experiment Number: 9**

**AIM:** Design an XOR truth table using Python

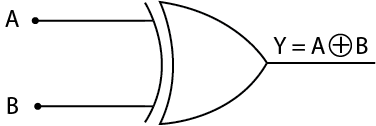
**PLATFORM/TOOLS USED:**

Python programming language

**THEORY:**

The Exclusive-OR Gate **/ XOR Gate** is a combination of all the three basic gates (NOT, AND, OR gates). It receives two or more input signals but produces only one output signal. It results in a low '0' if the input bit pattern contains an even number of high '1' signals. If there are an odd number of high '1' signals, it results in high.

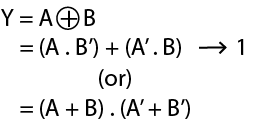
Exclusive-OR Symbol:



Truth Table for XOR Gate:

|  |  |  |
| --- | --- | --- |
| A | B | Y=A.B’ + A’.B |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

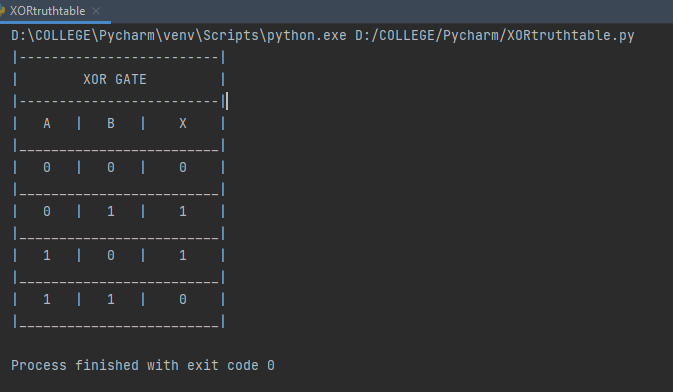
Boolean Expression:



**CODE:**

def XOR (a, b):  
 if a != b:  
 return 1  
 else:  
 return 0  
print("|-------------------------|")  
print("| XOR GATE |")  
print("|-------------------------|")  
print("| A | B | X |")   
print("|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|")  
print("| 0 | 0 |"," ",XOR(0,0) ," |")  
print("|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|")  
print("| 0 | 1 |"," ",XOR(0,1) ," |")  
print("|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|")  
print("| 1 | 0 |"," ",XOR(1,0) ," |")  
print("|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|")  
print("| 1 | 1 |"," ",XOR(1,1) ," |")  
print("|\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_|")

**OUTPUT:**

****

**CONCLUSION:**

Truth table for XOR gate has been constructed successfully.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |

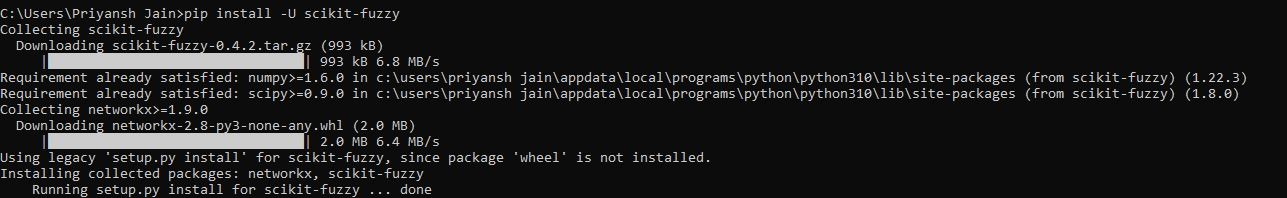
**Experiment Number: 10**

**AIM:** Study of SCIKIT fuzzy

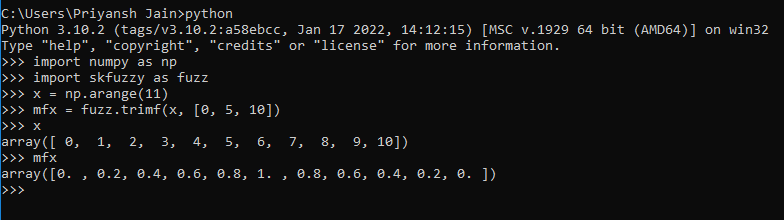
**THEORY:**

Scikit-Fuzzy is a collection of fuzzy logic algorithms intended for use in the SciPy Stack, written in the Python computing language. This SciKit is developed by the SciPy community.

scikit-fuzzy is an fuzzy logic Python package that works with numpy arrays. The package is imported as skfuzzy:

****

Within scikit-fuzzy, universe variables and fuzzy membership functions are represented by numpy arrays. Generation of membership functions is as simple as:

****

While most functions are available in the base namespace, the package is factored with a logical grouping of functions in submodules. The current capabilities of scikit-fuzzy include:

* **fuzz.membership:** Fuzzy membership function generation
* **fuzz.defuzzify:** Defuzzification algorithms to return crisp results from fuzzy sets.
* **fuzz.fuzzymath:** The core of scikit-fuzzy, containing the majority of the most common fuzzy logic operations.
* **fuzz.intervals:** Interval mathematics. The restricted Dong, Shah, & Wong (DSW) methods for fuzzy set math live here.
* **fuzz.image:** Limited fuzzy logic image processing operations.
* **fuzz.cluster:** Fuzzy c-means clustering.
* **fuzz.filters:** Fuzzy Inference Ruled by Else-action (FIRE) filters in 1D and 2D.

This package implements many useful tools for projects involving fuzzy logic, also known as grey logic.

Scikit-fuzzy is a robust set of foundational tools for problems involving fuzzy logic and fuzzy systems. This area has been a challenge for the scientific Python community, largely because the common first exposure to this topic is through the MATLAB® Fuzzy Logic Toolbox™.

The goals of scikit-fuzzy are to provide the community with a robust toolkit of independently developed and implemented fuzzy logic algorithms, filling a void in the capabilities of scientific and numerical Python, and to increase the attractiveness of scientific Python as a valid alternative to closed-source options. Scikit-fuzzy is structured similarly to scikit-learn and scikit-image, current source code is available on GitHub, and pull requests are welcome.

|  |  |  |  |
| --- | --- | --- | --- |
| CRITERIA | TOTAL MARKS | MARKS OBTAINED | COMMENTS |
| Concept (A) | 2 |  |  |
| Implementation (B) | 2 |  |  |
| Performance (C) | 2 |  |  |
| TOTAL | 6 |  |  |