Lab File

Artificial Intelligence
[CSE-401]

DEPARTMENT
OF
COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF TECHNOLOGY
IN
COMPUTER SCIENCE AND ENGINEERING

Submitted To:

Mr. Sanjiv Kumar Tomar
CSE Department, ASET

Submitted By:

Suchinton Chakravrty
A2345920063
B. Tech (CSE Evng.)
6CSE2-Evng.

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY
AMITY UNIVERSITY UTTAR PRADESH
NOIDA-201301

TABLE OF CONTENTS

EXPERIMENT 1

Objective:

Write a program to implement A*algorithm in python.

Software Used:

VS Code

Theory:

The A* algorithm is a popular heuristic search algorithm used for finding the shortest path between two points in a graph. It is often used in video games and robotics, as well as in other applications such as route planning and navigation systems.

The A* algorithm works by considering both the cost of the path from the starting node to the current node, as well as an estimated cost of the path from the current node to the goal node. This estimated cost is known as the heuristic, and is typically based on the distance between the current node and the goal node.

At each step of the algorithm, the node with the lowest total cost (path cost plus heuristic) is selected as the next node to explore. The algorithm continues until the goal node is reached, or until all nodes have been explored.

The key to the A* algorithm's efficiency is its use of the heuristic function to guide the search towards the goal node, which can significantly reduce the number of nodes that need to be explored.

Overall, the A* algorithm is a powerful tool for finding the shortest path between two points in a graph, and is widely used in a variety of applications.

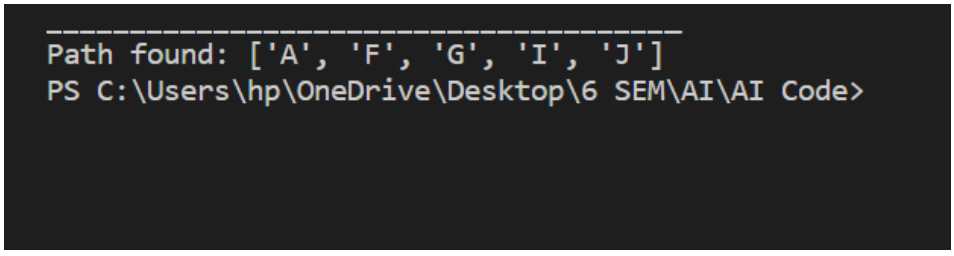
Code:

```
def aStarAlgo(start_node, stop_node):
    open_set = set(start_node)
    closed_set = set()
    g = {}
    parents = {}
    g[start_node] = 0
    parents[start_node] = start_node
```

```
    while len(open_set) > 0:
        n = None
        for v in open_set:
            if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                n = v
        if n == stop_node or Graph_nodes[n] == None:
            pass
        else:
            for (m, weight) in get_neighbors(n):

                if m not in open_set and m not in closed_set:
                    open_set.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                else:
                    if g[m] > g[n] + weight:
                        #update g(m)
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_set:
                            closed_set.remove(m)
                            open_set.add(m)
        if n == None:
            print('Path does not exist!')
            return None

        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()
            print('Path found: {}'.format(path))
            return path
        open_set.remove(n)
        closed_set.add(n)
    print('Path does not exist!')
    return None
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
```

```
def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
        'I': 1,
        'J': 0
    }
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('A', 6), ('C', 3), ('D', 2)],
    'C': [('B', 3), ('D', 1), ('E', 5)],
    'D': [('B', 2), ('C', 1), ('E', 8)],
    'E': [('C', 5), ('D', 8), ('I', 5), ('J', 5)],
    'F': [('A', 3), ('G', 1), ('H', 7)],
    'G': [('F', 1), ('I', 3)],
    'H': [('F', 7), ('I', 2)],
    'I': [('E', 5), ('G', 3), ('H', 2), ('J', 3)],
}
print("_____")
aStarAlgo('A', 'J')
```

Output:



EXPERIMENT 2

Objective: Write a program to implement Single Player Game

Software Used:
VS Code

Code:
```
import random
target_number = random.randint(1, 100)

max_attempts = 10
num_attempts = 0

while num_attempts < max_attempts:
    guess = int(input("Guess a number between 1 and 100: "))
    num_attempts += 1
    if guess == target_number:
        print("Congratulations, you guessed the number in", num_attempts, "attempts!")
        break
    elif guess < target_number:
        print("Your guess is too low.")
    else:
        print("Your guess is too high.")
    if num_attempts == max_attempts:
        print("Sorry, you have run out of attempts. The number was", target_number)
```

Output:
```
ers/hp/OneDrive/Desktop/6 SEM/AI/AI Code/Exp2.py"
Guess a number between 1 and 100: 76
Your guess is too high.
Guess a number between 1 and 100: 43
Your guess is too high.
Guess a number between 1 and 100: 23
Your guess is too low.
Guess a number between 1 and 100: 32
Your guess is too high.
Guess a number between 1 and 100: 25
Your guess is too low.
Guess a number between 1 and 100: 28
Your guess is too high.
Guess a number between 1 and 100: 27
Congratulations, you guessed the number in 7 attempts!
PS C:\Users\hp\OneDrive\Desktop\6 SEM\AI\AI Code>
```

XPERIMENT 3

Objective:

Write a program to implement Tic-Tac-Toe game problem

Software Used:

VS Code

Theory:

Tic-tac-toe is a classic two-player game that is played on a 3x3 grid. The objective of the game is for one player to get three of their symbols (either X or O) in a row, either horizontally, vertically, or diagonally.

The problem with the game is that if both players play optimally, the game will always end in a tie. This is because there are a limited number of possible moves in the game, and with optimal play, both players will always be able to block their opponent from getting three in a row.

However, if one player makes a mistake or fails to play optimally, the other player may be able to take advantage of the situation and win the game.

To make the game more challenging and interesting, variations of the game can be played, such as increasing the size of the board or allowing players to place more than one symbol at a time.

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):
```

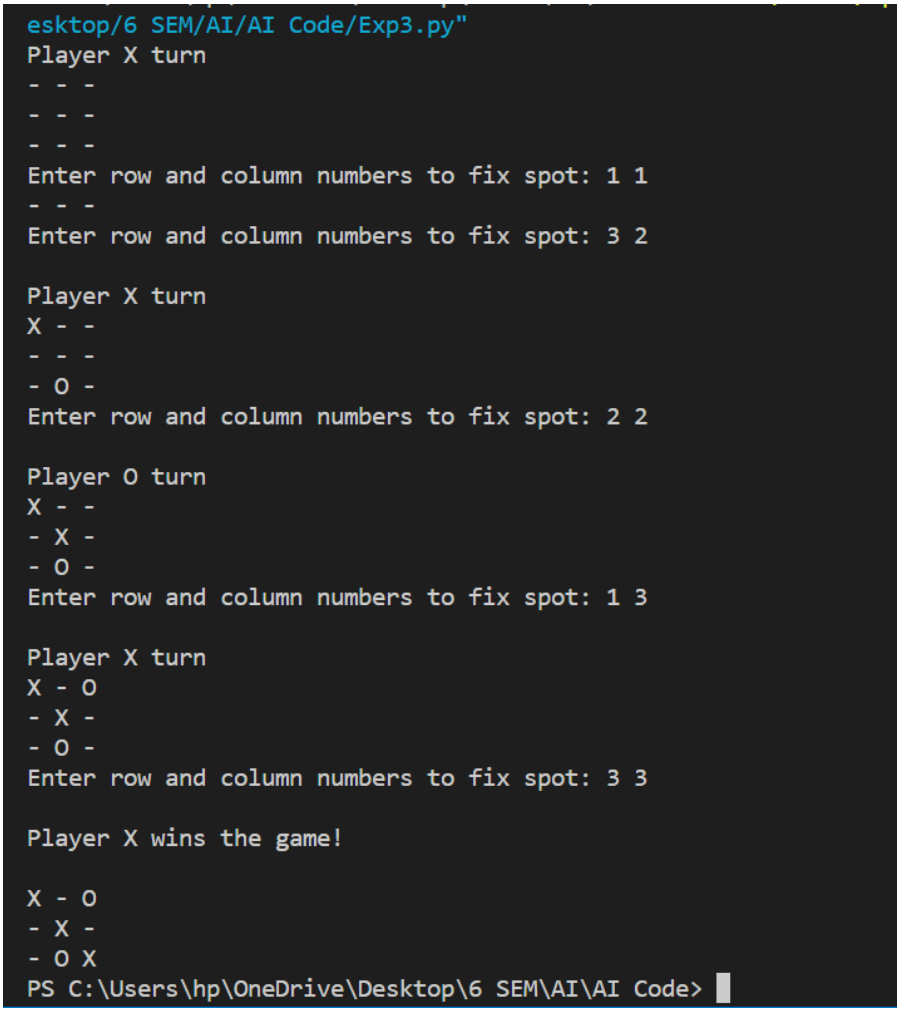
```
            row.append('-')
        self.board.append(row)

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
            player = self.swap_player_turn(player)
        print()
        self.show_board()
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

Output:



EXPERIMENT 4

Objective: Implement Brute force solution to the Knapsack problem in Python

Software Used: VS Code

Theory: The knapsack problem is a well-known optimization problem in computer science and mathematics. It involves selecting a set of items, each with its own weight and value, to be placed in a knapsack or backpack, subject to the constraint that the total weight of the items cannot exceed a certain capacity.

The goal of the problem is to maximize the total value of the items in the knapsack while staying within the weight limit. This is often referred to as the "0-1 knapsack problem," where each item can either be selected or not selected for the knapsack. The knapsack problem is NP-hard, meaning that it is computationally expensive to solve for large problem sizes. There are various algorithms that can be used to solve the knapsack problem, including dynamic programming, branch and bound, and genetic algorithms.

Code:

```
def knapSack(W, wt, val, n):
  if n == 0 or W == 0 :
    return 0
  if (wt[n-1] > W):
    return knapSack(W, wt, val, n-1)
  else:
    return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
      knapSack(W, wt, val, n-1))
val = [50,100,150,200]
wt = [8,16,32,40]
W = 64
n = len(val)

print("_____")
print("The maniximum profit that can be generated is : ")
print (knapSack(W, wt, val, n))
```

Output:

```
_____
The maniximum profit that can be generated is :
350
```

EXPERIMENT 5

Objective: Implement Graph colouring problem using python

Software Used: VS Code

Theory: The graph colouring problem is a classic optimization problem in computer science and mathematics. It involves assigning colours to the vertices of a graph in such a way that no two adjacent vertices have the same colour.

The goal of the problem is to find the minimum number of colours needed to colour the graph in this way. This is known as the chromatic number of the graph.

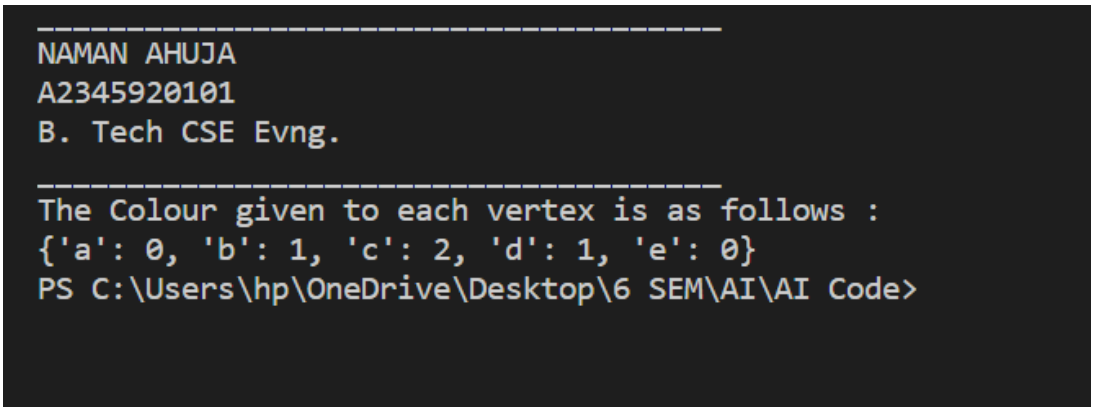
The graph coloring problem is NP-hard, meaning that it is computationally expensive to solve for large problem sizes. There are various algorithms that can be used to solve the graph colouring problem, including greedy algorithms, backtracking algorithms, and genetic algorithms.

Code:

```
def colour_vertices(graph):


  vertices = sorted((list(graph.keys())))
  colour_graph = {}

  for vertex in vertices:
    unused_colours = len(vertices) * [True]

    for neighbor in graph[vertex]:
      if neighbor in colour_graph:
        colour = colour_graph[neighbor]
        unused_colours[colour] = False
    for colour, unused in enumerate(unused_colours):
      if unused:
        colour_graph[vertex] = colour
        break
```

```
  return colour_graph
graph = { "a" : ["c", "d", "b"],
          "b" : ["c", "e", "a"],
          "c" : ["a", "b"],
          "d" : ["a","e"],
          "e" : ["d", "b"],
         }
result = (colour_vertices(graph))
print("_____")
print("NAMAN AHUJA")
print("A2345920101")
print("B. Tech CSE Evng.")
print("_____")
print("The Colour given to each vertex is as follows :")
print(result)
```

Output:



EXPERIMENT 6

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

Software Used: VS Code

Theory: The water jug problem is a classic puzzle in mathematics and computer science that involves two water jugs of different sizes and a target volume of water that needs to be measured out.

The problem statement typically goes as follows: you have a 5-liter jug and a 3-liter jug. Neither jug has any markings on it. You are given a large container of water and are asked to measure out exactly 4 liters of water using only these jugs and the container of water.

The solution to this problem involves a series of steps that require pouring water from one jug to another in a specific way. One possible solution is:

- Fill the 5-liter jug to the top.
- Pour the water from the 5-liter jug into the 3-liter jug until it is full, leaving 2 liters in the 5-liter jug.
- Empty the 3-liter jug.
- Pour the remaining 2 liters of water from the 5-liter jug into the 3-liter jug.
- Fill the 5-liter jug to the top again.
- Pour water from the 5-liter jug into the 3-liter jug until it is full, leaving 4 liters in the 5-liter jug.

This solution works because it takes advantage of the fact that the total capacity of the two jugs is 8 liters, which is exactly twice the target volume of 4 liters. By carefully pouring water from one jug to another, it is possible to measure out exactly 4 liters of water using only these two jugs.

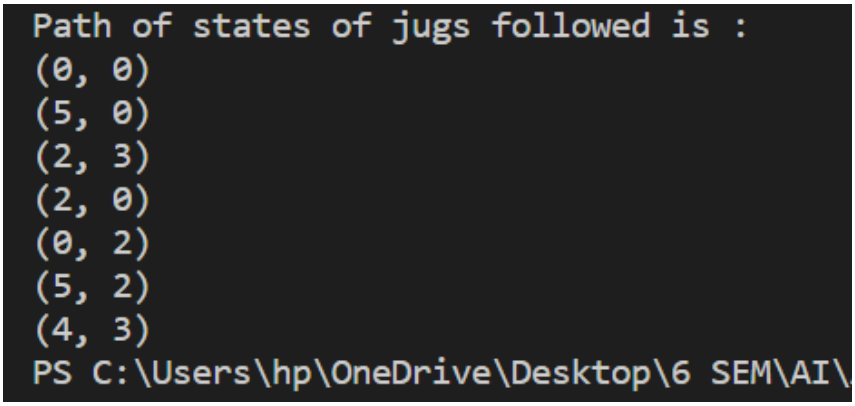
Code:

```
# define the problem parameters
capacities = (5, 3)  # capacities of the jugs
goal = 4             # goal amount of water to be measured
class State:
    def __init__(self, jug1, jug2):
        self.jug1 = jug1
        self.jug2 = jug2

    def __eq__(self, other):
        return self.jug1 == other.jug1 and self.jug2 == other.jug2
```

```
    def __hash__(self):
        return hash((self.jug1, self.jug2))

    def __str__(self):
        return f"({self.jug1}, {self.jug2})"

    def fill_jug1(self):
        return State(capacities[0], self.jug2)

    def fill_jug2(self):
        return State(self.jug1, capacities[1])

    def empty_jug1(self):
        return State(0, self.jug2)

    def empty_jug2(self):
        return State(self.jug1, 0)

    def pour_jug1_to_jug2(self):
        amount = min(self.jug1, capacities[1] - self.jug2)
        return State(self.jug1 - amount, self.jug2 + amount)

    def pour_jug2_to_jug1(self):
        amount = min(self.jug2, capacities[0] - self.jug1)
        return State(self.jug1 + amount, self.jug2 - amount)

def bfs(initial_state):
    visited = set()
    queue = [[initial_state]]
    while queue:
        path = queue.pop(0)
        state = path[-1]
        if state.jug1 == goal or state.jug2 == goal:
            return path
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)
    return None

def get_neighbors(state):
    neighbors = []
```

```
    neighbors.append(state.fill_jug1())
    neighbors.append(state.fill_jug2())
    neighbors.append(state.empty_jug1())
    neighbors.append(state.empty_jug2())
    neighbors.append(state.pour_jug1_to_jug2())
    neighbors.append(state.pour_jug2_to_jug1())
    return [neighbor for neighbor in neighbors if neighbor != state]

initial_state = State(0, 0)
solution = bfs(initial_state)

if solution is None:
    print("No solution found.")
else:
    print("Path of states of jugs followed is :")
    for state in solution:
        print(state)
```

Output:



EXPERIMENT 7

Objective: Write a program to implement DFS using Python

Software Used: VS Code

Theory: DFS stands for Depth First Search, which is a well-known graph traversal algorithm used in computer science and mathematics. The algorithm starts at a given node in a graph and explores as far as possible along each branch before backtracking.

DFS works by maintaining a stack of nodes that have not yet been visited. At each iteration, it selects the top node from the stack and marks it as visited. It then examines each of its neighbors that have not been visited, adding them to the stack in the order they are encountered. This process continues until the entire graph has been explored.

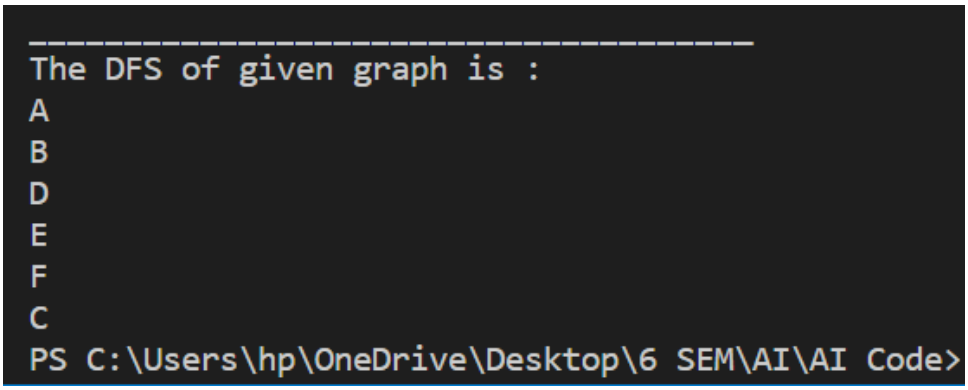
DFS has several important applications in computer science, including in pathfinding algorithms, topological sorting, and cycle detection. It is also a core component of many other algorithms, including those used for graph coloring, network flow, and tree traversal.

Code:

```
graph = {
   'A' : ['B','C'],
   'B' : ['D', 'E'],
   'C' : ['F'],
   'D' : [],
   'E' : ['F'],
   'F' : []
}
visited = set()

def dfs(visited, graph, node):
   if node not in visited:
      print (node)
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)



print("_____")
print("The DFS of given graph is :")
```

dfs(visited, graph, 'A')

Output:



EXPERIMENT 8

Objective:

Tokenization of word and Sentences with the help of NLTK package

Software Used:

VS Code

Theory:

To tokenize sentences and words with NLTK, “nltk.word_tokenize()” function will be used. NLTK Tokenization is used for parsing a large amount of textual data into parts to perform an analysis of the character of the text. NLTK for tokenization can be used for training machine learning models, Natural Language Processing text cleaning. The tokenized words and sentences with NLTK can be turned into a data frame and vectorized. Natural Language Tool Kit (NLTK) tokenization involves punctuation cleaning, text cleaning, vectorization of parsed text data for better lemmatization, and stemming along with machine learning algorithm training.

Natural Language Tool Kit Python Libray has a tokenization package is called “tokenize”. In the “tokenize” package of NLTK, there are two types of tokenization functions.

- “word_tokenize” is to tokenize words.
- “sent_tokenize” is to tokenize sentences.

1. **Tokenizationof Sentences with the help of NLTK package**

Code:

```
pip install nltk

import nltk
nltk.download('punkt')

sentence = "Hello, how are you doing today? I hope you're doing well."

sentences = nltk.sent_tokenize(sentence)

print("Tokenization of Sentences with the help of NLTK package:`")
print(sentences)
```

Output:

```
Tokenization of Sentences with the help of NLTK package:
['Hello, how are you doing today?', "I hope you're doing well."]
PS C:\Users\hp\OneDrive\Desktop\6 SEM\AI\AI Code>
```

2. **Tokenization of words with the help of NLTK package**

Code:

```
pip install nltk

import nltk

nltk.download('punkt')

sentence = "Hello, how are you doing today? I hope you're doing well."

words = nltk.word_tokenize(sentence)
print("Tokenization of words with the help of NLTK package:`")
print(words)
```

Output:

```
Tokenization of words with the help of NLTK package:`
['Hello', ',', 'how', 'are', 'you', 'doing', 'today', '?', 'I', 'hope', 'you', "'re", 'doing', 'well', '.']
PS C:\Users\hp\OneDrive\Desktop\6 SEM\AI\AI Code>
```

EXPERIMENT 9

Objective:

Design an XOR truth table using Python

Software Used:

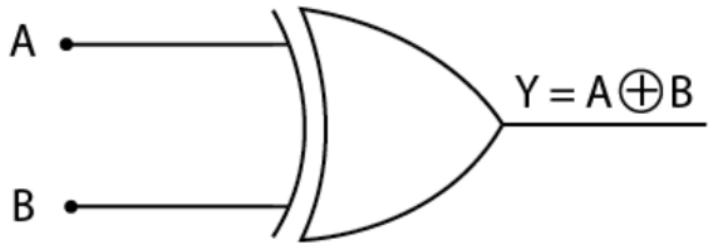
VS Code

Theory:

An XOR gate is a logical gate that takes two binary inputs (0 or 1) and produces a single binary output. The output is "1" if the inputs are different, and "0" if they are the same.

The truth table for an XOR gate is as follows:

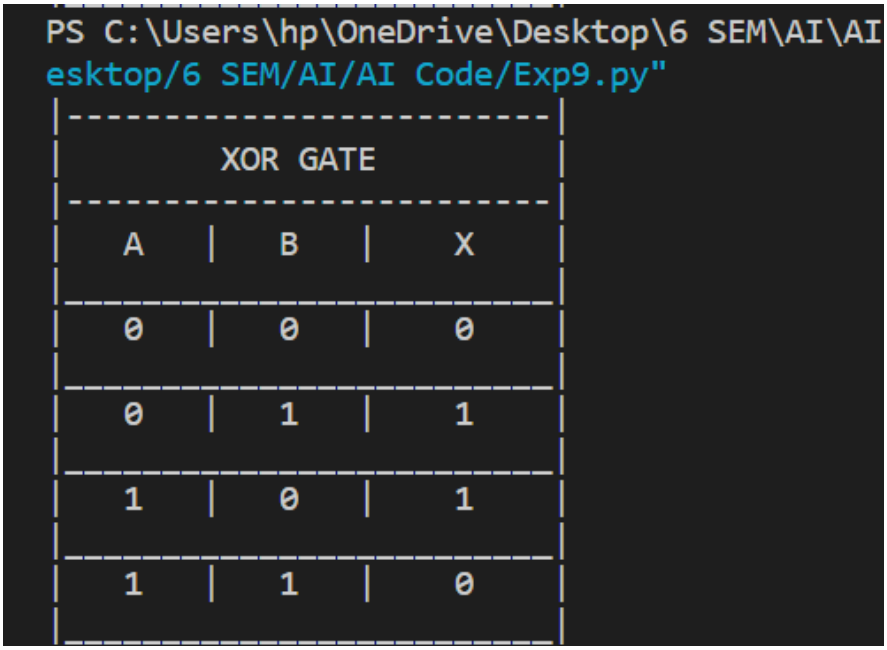
Input A	Input B	Output
0	0	0
0	1	1
1	0	1
1	1	0



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:



EXPERIMENT 10

Objective:

Write a python script for a given cryptarithmetic problem (APPLE+LEMON= BANANA).

Software Used:

VS Code

Theory:

Cryptarithmetic, also known as alphametics or cryptarithms, is a type of mathematical puzzle in which mathematical operations are replaced by letters or symbols. The goal of the puzzle is to find the numerical values of the letters or symbols that make the equation true.

For example, the following is a cryptarithmetic puzzle:
SEND
+ MORE

MONEY

In this puzzle, each letter represents a single digit number. The goal is to find the numerical values of the letters that make the equation true. In this case, the solution is:
9567
+ 1085

10652
which means that the solution to the puzzle "SEND + MORE = MONEY" is "9567 + 1085 = 10652".

When we run this program, it will output the solution:
95678 + 12016 = 107694
which means that the solution to the puzzle "APPLE + LEMON = BANANA" is "95678 + 12016 = 107694".

Code:

```
def is_valid_assignment(word1, word2, result, assigned):
  # First letter of any word cannot be zero.
  if assigned[word1[0]] == 0 or assigned[word2[0]] == 0 or assigned[
      result[0]] == 0:
    return False
  return True
def _solve(word1, word2, result, letters, assigned, solutions):
  if not letters:
    if is_valid_assignment(word1, word2, result, assigned):
      num1 = find_value(word1, assigned)
      num2 = find_value(word2, assigned)
      num_result = find_value(result, assigned)
      if num1 + num2 == num_result:
        solutions.append((f'{num1} + {num2} = {num_result}', assigned.copy()))
    return
  for num in range(10):
    if num not in assigned.values():
      cur_letter = letters.pop()
      assigned[cur_letter] = num
      _solve(word1, word2, result, letters, assigned, solutions)
      assigned.pop(cur_letter)
      letters.append(cur_letter)
def solve(word1, word2, result):
  letters = sorted(set(word1) | set(word2) | set(result))
  if len(result) > max(len(word1), len(word2)) + 1 or len(letters) > 10:
    print('0 Solutions!')
```

```
    return
  solutions = []
  _solve(word1, word2, result, letters, {}, solutions)
  if solutions:
    print('\nSolutions:')
    for soln in solutions:
      print(f'{soln[0]}\t{soln[1]}')
if __name__ == '__main__':
  print('CRYPTARITHMETIC PUZZLE SOLVER')
  print('WORD1 + WORD2 = RESULT')
  word1 = input('Enter WORD1: ').upper()
  word2 = input('Enter WORD2: ').upper()
  result = input('Enter RESULT: ').upper()
  if not word1.isalpha() or not word2.isalpha() or not result.isalpha():
    raise TypeError('Inputs should ony consists of alphabets.')
  solve(word1, word2, result)
```

Output:

```
CRYPTARITHMETIC PUZZLE SOLVER
WORD1 + WORD2 = RESULT
Enter WORD1: APPLE
Enter WORD2: LEMON
Enter RESULT: BANANA

Solutions:
67794 + 94832 = 162626  {'P': 7, 'O': 3, 'N': 2, 'M': 8, 'L': 9, 'E': 4, 'B': 1, 'A': 6}
```