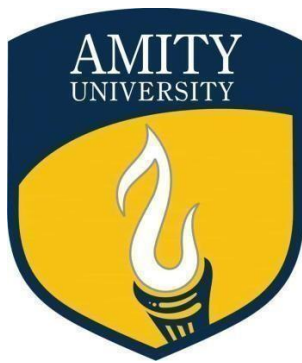


ARTIFICIAL INTELLIGENCE LAB FILE



AMITY UNIVERSITY

**DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING**

AMITY UNIVERSITY, NOIDA, UTTAR PRADESH

SUBMITTED BY

Priyansh Bhadauria

SUBMITTED TO

Dr Deepak Gaur

Enrollment No: A2305221430

6-CSE-4(Y)

INDEX

Sr No.	AIM of the Experiment	Date	Remark
1	Introduction to Python Programming Language.		
2	Write a Python script for implementation of vectors, matrix operations.		
3	Write a program to implement DFS and BFS algorithm.		
4	Create the program to implement BFS algorithm for 8- puzzle problem.		
5	Write a program to implement the water jug problem using BFS .		
6	Write a Python script to implement A* algorithm.		
7	Write a Python script to implement tic tac toe game problem.		
8	Write a Python script to implement graph coloring problem.		
9	Write the Python script for minmax algorithm.		
10	Implement brute force solution to the knapsack problem in Python.		
11	Write a program to implement DFS Algorithm for water jug problem.		
12	Tokenization of word and sentences with the help of Python.		

EXPERIMENT – 1

AIM – Introduction to python programming

SOFTWARE USED – VS code

THEORY -

Python is a high-level, versatile programming language known for its simplicity and readability. It supports multiple programming paradigms including procedural, object-oriented, and functional programming. Python's syntax emphasizes code readability and allows developers to express concepts in fewer lines of code compared to other languages.

Key features of Python include:

Interpreted: Python code is executed line by line by an interpreter, enabling rapid development and testing.

Dynamic Typing: Python uses dynamic typing, allowing variables to be assigned without specifying their type explicitly.

Rich Standard Library: Python comes with a comprehensive standard library, providing modules and packages for various tasks such as file I/O, networking, and data processing.

Extensive Third-Party Libraries: Python has a vast ecosystem of third-party libraries and frameworks for almost every conceivable task, including web development (Django, Flask), scientific computing (NumPy, SciPy), data analysis (Pandas), and machine learning (TensorFlow, PyTorch).

Platform Independence: Python code can run on various platforms including Windows, macOS, and Linux, making it highly portable.

Community Support: Python has a large and active community of developers who contribute to its growth and provide support through forums, mailing lists, and online resources.

Ease of Learning: Python's simple and intuitive syntax makes it an excellent choice for beginners learning to code.

Python is widely used in various fields such as web development, data science, artificial intelligence, automation, scientific computing, and more. Its popularity continues to grow due to its versatility, ease of use, and strong community support.

CODE-

Example: Hello, World! Program

```
print("Hello, World!")
```

Python uses indentation to define code blocks instead of braces.

For example, in the code below, the if statement's body is indented.

```
x = 10

if x > 5:

    print("x is greater than 5")

else:

    print("x is not greater than 5")

# Python supports dynamic typing, meaning you don't need to declare variable types explicitly.

# Variables are created when they are first assigned a value.

# For example:

message = "Hello, Python!" # message is a string

number = 10 # number is an integer

pi_value = 3.14 # pi_value is a float

# Python has various data types including integers, floats, strings, lists, tuples, dictionaries, etc.

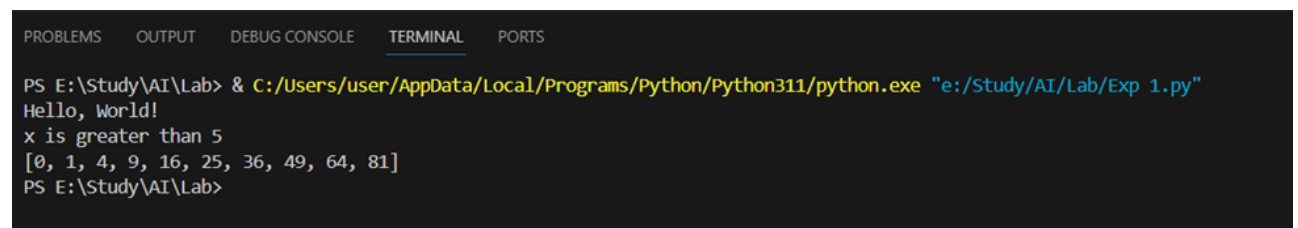
# Python also supports list comprehensions, which allow you to create lists using a more concise syntax.

# For example, to create a list of squares of numbers from 0 to 9:

squares = [x ** 2 for x in range(10)]

print(squares) # Output: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

OUTPUT-

A screenshot of a terminal window with a dark background. At the top, there are tabs labeled 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is active and underlined), and 'PORTS'. The terminal shows the command prompt 'PS E:\Study\AI\Lab>' followed by the command '& C:/Users/user/AppData/Local/Programs/Python/Python311/python.exe "e:/Study/AI/Lab/Exp 1.py"'. The output of the script is displayed line by line: 'Hello, World!', 'x is greater than 5', and '[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]'. The prompt returns to 'PS E:\Study\AI\Lab>'.

EXPERIMENT – 2

AIM – To write a Python script for implementation of vectors, matrix operations.

SOFTWARE USED – VS code

THEORY -

In Python, vectors and matrices can be implemented using lists, NumPy arrays, or other specialized libraries like SciPy.

Lists: In Python, lists can be used to represent vectors and matrices. For example, a vector can be represented as a one-dimensional list, while a matrix can be represented as a list of lists where each inner list represents a row. However, using lists for large-scale numerical computations may not be efficient due to Python's dynamic typing and lack of optimized numerical operations.

NumPy Arrays: NumPy is a powerful library for numerical computing in Python. It provides efficient implementations of arrays (vectors) and multidimensional arrays (matrices), along with a wide range of mathematical functions and operations optimized for performance. NumPy arrays are homogeneous and support element-wise operations, slicing, broadcasting, and advanced indexing.

SciPy Sparse Matrices: SciPy, built on top of NumPy, provides functionality for scientific and technical computing. It includes support for sparse matrices, which are useful for handling large datasets with many zero elements efficiently. Sparse matrices store only the non-zero elements along with their indices, saving memory and speeding up computations for certain operations.

CODE -

class Vector:

```
def __init__(self, components):
    self.components = components

def __str__(self):
    return str(self.components)

def __add__(self, other):
    if len(self.components) != len(other.components):
        raise ValueError("Vectors must have the same length for addition")
    result = [x + y for x, y in zip(self.components, other.components)]
    return Vector(result)

def __sub__(self, other):
    if len(self.components) != len(other.components):
```

```

        raise ValueError("Vectors must have the same length for subtraction")

    result = [x - y for x, y in zip(self.components, other.components)]

    return Vector(result)

def dot_product(self, other):
    if len(self.components) != len(other.components):
        raise ValueError("Vectors must have the same length for dot product")

    result = sum(x * y for x, y in zip(self.components, other.components))

    return result

class Matrix:
    def __init__(self, rows):
        self.rows = rows

        self.num_rows = len(rows)

        self.num_cols = len(rows[0])

    def __str__(self):
        return '\n'.join([' '.join(map(str, row)) for row in self.rows])

    def __add__(self, other):
        if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
            raise ValueError("Matrices must have the same dimensions for addition")

        result = [[self.rows[i][j] + other.rows[i][j] for j in range(self.num_cols)] for i in
range(self.num_rows)]

        return Matrix(result)

    def __sub__(self, other):
        if self.num_rows != other.num_rows or self.num_cols != other.num_cols:
            raise ValueError("Matrices must have the same dimensions for subtraction")

        result = [[self.rows[i][j] - other.rows[i][j] for j in range(self.num_cols)] for i in
range(self.num_rows)]

        return Matrix(result)

    def transpose(self):
        result = [[self.rows[j][i] for j in range(self.num_rows)] for i in range(self.num_cols)]

        return Matrix(result)

    def multiply(self, other):
        if self.num_cols != other.num_rows:

```

```
        raise ValueError("Number of columns in first matrix must be equal to number of rows in second matrix")
```

```
        result = [[sum(a * b for a, b in zip(row, col)) for col in other.transpose().rows] for row in self.rows]
```

```
        return Matrix(result)
```

```
# Example usage
```

```
v1 = Vector([1, 2, 3])
```

```
v2 = Vector([4, 5, 6])
```

```
print("Vector Addition:", v1 + v2)
```

```
print("Vector Subtraction:", v1 - v2)
```

```
print("Dot Product:", v1.dot_product(v2))
```

```
m1 = Matrix([[1, 2, 3], [4, 5, 6]])
```

```
m2 = Matrix([[7, 8], [9, 10], [11, 12]])
```

```
print("\nMatrix Addition:")
```

```
print(m1 + m1)
```

```
print("\nMatrix Subtraction:")
```

```
print(m1 - m1)
```

```
print("\nMatrix Transpose:")
```

```
print(m1.transpose())
```

```
print("\nMatrix Multiplication:")
```

```
print(m1.multiply(m2))
```

OUTPUT-

```
PS E:\Study\AI\Lab> & C:\Users\user\AppData\Local\Programs\Python\Python311\python.exe "e:/Study/AI/Lab/Exp 2.py"
Vector Addition: [5, 7, 9]
Vector Subtraction: [-3, -3, -3]
Dot Product: 32
```

```
Matrix Addition:
2 4 6
8 10 12
```

```
Matrix Subtraction:
0 0 0
0 0 0
```

```
Matrix Transpose:
1 4
2 5
3 6
```

```
Matrix Multiplication:
58 64
139 154
```

EXPERIMENT – 11

AIM - To write a program to implement DFS Algorithm for water jug problem.

SOFTWARE USED — VS Code

THEORY -

The water jug problem is a classic puzzle that involves using two jugs to measure a certain amount of water. Here's a short summary of implementing Depth-First Search (DFS) algorithm for solving the water jug problem in Python:

Define the State: Represent each state of the jugs (i.e., the amount of water in each jug) as a tuple (x, y), where x is the amount of water in the first jug and y is the amount of water in the second jug.

Define the Actions: Define the actions that can be taken in each state, such as filling a jug, emptying a jug, or pouring water from one jug to another.

Implement DFS: Write a recursive DFS function to explore the state space, trying different actions from each state until the goal state is reached or all possible states are explored.

Check Goal State: Check if the goal state (e.g., having a certain amount of water in one of the jugs) is reached in each step of the DFS traversal.

Backtracking: If the goal state is not reached, backtrack to the previous state and try other actions until all possibilities are exhausted.

Return Solution: Return the sequence of actions (e.g., a list of tuples representing the actions taken) that lead to the goal state.

CODE -

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 dfs(current_state, visited, jug1_capacity, jug2_capacity, target):
    if current_state.jug1 == target or current_state.jug2 == target:
        return True
    visited.add(current_state)
    # Fill jug1
    if current_state.jug1 < jug1_capacity:
        next_state = State(jug1_capacity, current_state.jug2)
        if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
            print(next_state)
            return True
    # Fill jug2
    if current_state.jug2 < jug2_capacity:
        next_state = State(current_state.jug1, jug2_capacity)
        if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
            print(next_state)
            return True
    # Empty jug1
    if current_state.jug1 > 0:
        next_state = State(0, current_state.jug2)
        if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
            print(next_state)
            return True
    # Empty jug2
    if current_state.jug2 > 0:
        next_state = State(current_state.jug1, 0)
        if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
            print(next_state)
            return True

```

```

# Pour from jug1 to jug2
if current_state.jug1 > 0 and current_state.jug2 < jug2_capacity:
    amount_to_pour = min(current_state.jug1, jug2_capacity - current_state.jug2)
    next_state = State(current_state.jug1 - amount_to_pour, current_state.jug2 + amount_to_pour)
    if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
        print(next_state)
        return True

# Pour from jug2 to jug1
if current_state.jug2 > 0 and current_state.jug1 < jug1_capacity:
    amount_to_pour = min(current_state.jug2, jug1_capacity - current_state.jug1)
    next_state = State(current_state.jug1 + amount_to_pour, current_state.jug2 - amount_to_pour)
    if next_state not in visited and dfs(next_state, visited, jug1_capacity, jug2_capacity, target):
        print(next_state)
        return True

return False

def water_jug_dfs(jug1_capacity, jug2_capacity, target):
    initial_state = State(0, 0)
    visited = set()
    if dfs(initial_state, visited, jug1_capacity, jug2_capacity, target):
        print(f"Target {target} reached!")
    else:
        print(f"Target {target} cannot be reached.")

# Example usage
water_jug_dfs(4, 2, 2)

```

OUTPUT -

```

PS E:\Study\AI\Lab> & C:\Users\user\AppData\Local\Programs\Python\Python311\python.exe "e:/Study/AI/Lab/Exp 11.py"
(4, 2)
(4, 0)
Target 2 reached!
PS E:\Study\AI\Lab> 

```

EXPERIMENT – 3

AIM - To write a program to implement DFS and BFS algorithm.

SOFTWARE USED — online python compiler

THEORY -

Breadth-First Search (BFS):

- BFS is a graph traversal algorithm that explores all the neighboring nodes at the present depth before moving on to nodes at the next level.
- It starts at the root node and explores all the neighbor nodes at the present depth before moving on to the nodes at the next depth level.
- BFS uses a queue data structure to keep track of the nodes to be visited. It explores nodes level by level, guaranteeing that it visits all nodes at a certain depth level before moving to deeper levels.
- BFS is typically used to find the shortest path between two nodes in an unweighted graph.

Depth-First Search (DFS):

- DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking.
- It starts at the root node and explores as far as possible along each branch before backtracking to the nearest unexplored node.
- DFS uses a stack data structure (or recursion) to keep track of the nodes to be visited. It explores nodes deeply before exploring the neighboring nodes.
- DFS does not guarantee finding the shortest path between two nodes. It can get stuck in deep branches and might not backtrack to explore other paths, which could potentially lead to longer paths.

CODE -

```
from collections import defaultdict
```

```
class Graph:
```

```

def __init__(self):
    self.graph = defaultdict(list)

def addEdge(self, u, v):
    self.graph[u].append(v)

def DFS(self, v):
    visited = set()
    stack = [v]
    while stack:
        s = stack.pop()
        if s not in visited:
            visited.add(s)
            print(s, end=' ')
            for neighbour in self.graph[s]:
                stack.append(neighbour)

def BFS(self, v):
    visited = set()
    queue = [v]
    while queue:
        s = queue.pop(0)
        if s not in visited:
            visited.add(s)
            print(s, end=' ')
            for neighbour in self.graph[s]:
                queue.append(neighbour)

```

```

g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)

```

```
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)
g.addEdge(3, 7)
g.addEdge(4, 8)
g.addEdge(5, 9)
g.addEdge(6, 10)
```

```
print("Following is Depth First Traversal")
```

```
g.DFS(0)
```

```
print("\nFollowing is Breadth First Traversal")
```

```
g.BFS(0)
```

OUTPUT

```
Shell Clear
Following is Depth First Traversal
0 2 6 10 5 9 1 4 8 3 7 |
Following is Breadth First Traversal
0 1 2 3 4 5 6 7 8 9 10 >
```

EXPERIMENT – 4

AIM - To Create the program to implement BFS algorithm for 8- puzzle problem.

SOFTWARE USED — google colab

THEORY -

Introduction

An instance of the n-puzzle game consists of a board holding n^2-1 distinct movable tiles, plus an empty space. The tiles are numbers from the set $1, \dots, n^2-1$. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, the blank space is going to be represented with the number 0. Given an initial state of the board, the combinatorial search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order $0, 1, \dots, n^2-1$.

The search space is the set of all possible states reachable from the initial state. The blank space may be swapped with a component in one of the four directions {'Up', 'Down', 'Left', 'Right'}, one move at a time.

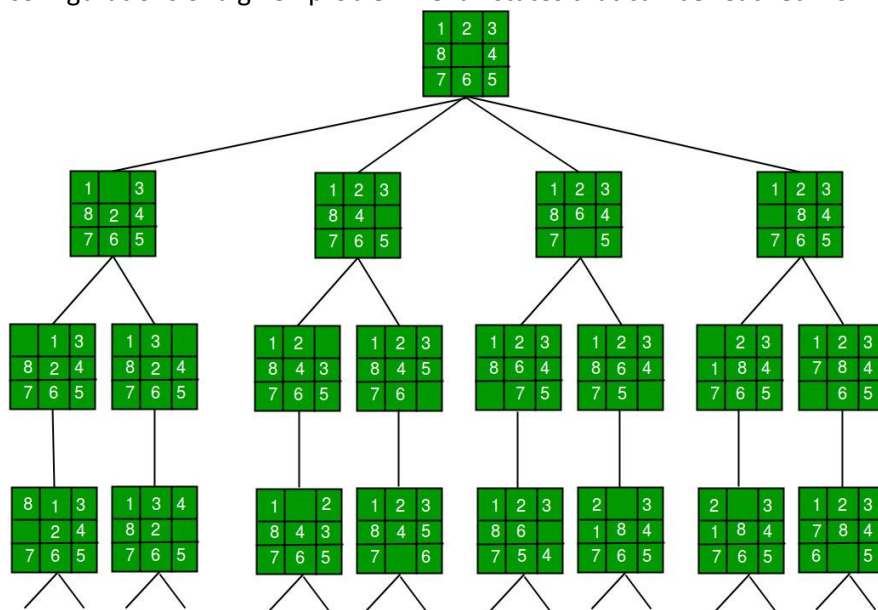
In this 8 puzzle problem is discussed.

Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

For example:

Initial configuration	Final configuration																		
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>6</td><td></td></tr><tr><td>7</td><td>8</td><td>4</td></tr></table>	1	2	3	5	6		7	8	4	<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>5</td><td>8</td><td>6</td></tr><tr><td></td><td>7</td><td>4</td></tr></table>	1	2	3	5	8	6		7	4
1	2	3																	
5	6																		
7	8	4																	
1	2	3																	
5	8	6																	
	7	4																	

Breadth First Search (BFS): We can perform a breadth-first search on state space (Set of all configurations of a given problem i.e. all states that can be reached from the initial state) tree.



Algorithm Review

The searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence in order to visit a node:

First, we remove a node from the frontier set.

Second, we check the state against the goal state to determine if a solution has been found.

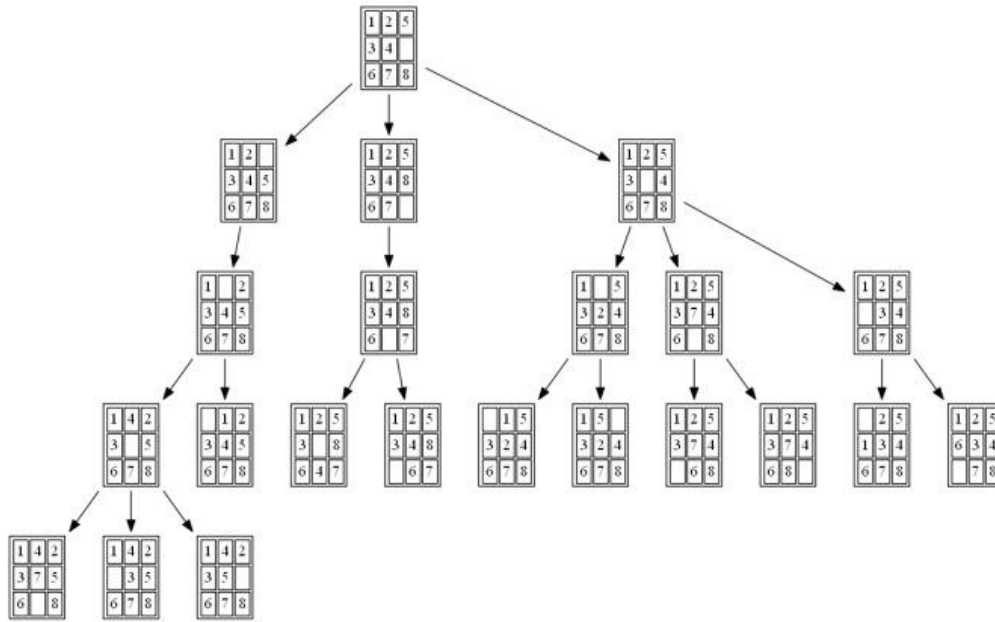
Finally, if the result of the check is negative, we then expand the node. To expand a given node, we generate successor nodes adjacent to the current node, and add them to the frontier set. Note that if these successor nodes are already in the frontier, or have already been visited, then they should not be added to the frontier again.

Example: Breadth-First Search

Initial State: 1,2,5,3,4,0,6,7,8

1	2	5
3	4	
6	7	8

The nodes expanded by BFS (also the nodes that are in the fringe / frontier of the queue) are shown in the following figure:



CODE-

```
from time import time
```

```
from queue import Queue
```

```
class Puzzle:
```

```
    goal_state=[1,2,3,8,0,4,7,6,5]
```

```
    num_of_instances=0
```

```
    def __init__(self,state,parent,action):
```

```
        self.parent=parent
```

```
        self.state=state
```

```
        self.action=action
```

```
        Puzzle.num_of_instances+=1
```

```
    def __str__(self):
```

```
        return str(self.state[0:3])+'\n'+str(self.state[3:6])+'\n'+str(self.state[6:9])
```

```
    def goal_test(self):
```

```
        if self.state == self.goal_state:
```

```
            return True
```

```
        return False
```



```
@staticmethod
```

```
def find_legal_actions(i,j):  
    legal_action = ['U', 'D', 'L', 'R']  
    if i == 0: # up is disable  
        legal_action.remove('U')  
    elif i == 2: # down is disable  
        legal_action.remove('D')  
    if j == 0:  
        legal_action.remove('L')  
    elif j == 2:  
        legal_action.remove('R')  
    return legal_action
```

```
def generate_child(self):  
    children=[]  
    x = self.state.index(0)  
    i = int(x / 3)  
    j = int(x % 3)  
    legal_actions=self.find_legal_actions(i,j)  
  
    for action in legal_actions:  
        new_state = self.state.copy()  
        if action is 'U':  
            new_state[x], new_state[x-3] = new_state[x-3], new_state[x]  
        elif action is 'D':  
            new_state[x], new_state[x+3] = new_state[x+3], new_state[x]  
        elif action is 'L':  
            new_state[x], new_state[x-1] = new_state[x-1], new_state[x]  
        elif action is 'R':  
            new_state[x], new_state[x+1] = new_state[x+1], new_state[x]
```

```
        children.append(Puzzle(new_state,self,action))
    return children
```

```
def find_solution(self):
    solution = []
    solution.append(self.action)
    path = self
    while path.parent != None:
        path = path.parent
        solution.append(path.action)
    solution = solution[:-1]
    solution.reverse()
    return solution
```

```
def breadth_first_search(initial_state):
    start_node = Puzzle(initial_state, None, None)
    if start_node.goal_test():
        return start_node.find_solution()
    q = Queue()
    q.put(start_node)
    explored=[]
    while not(q.empty()):
        node=q.get()
        explored.append(node.state)
        children=node.generate_child()
        for child in children:
            if child.state not in explored:
                if child.goal_test():
                    return child.find_solution()
                q.put(child)
    return
```

#Start executing the 8-puzzle with setting up the initial state

#Here we have considered 3 initial state intitalized using state variable

```
state=[[1, 3, 4,
```

```
      8, 6, 2,
```

```
      7, 0, 5],
```

```
      [2, 8, 1,
```

```
      0, 4, 3,
```

```
      7, 6, 5],
```

```
      [2, 8, 1,
```

```
      4, 6, 3,
```

```
      0, 7, 5]]
```

```
for i in range(0,3):
```

```
    Puzzle.num_of_instances=0
```

```
    t0=time()
```

```
    bfs=breathth_first_search(state[i])
```

```
    t1=time()-t0
```

```
    print('BFS:', bfs)
```

```
    print('space:',Puzzle.num_of_instances)
```

```
    print('time:',t1)
```

```
    print()
```

```
print('-----')
```

OUTPUT-

```
>>> <>:40: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:42: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:44: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:46: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:40: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:42: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:44: SyntaxWarning: "is" with a literal. Did you mean "=="?
>>> <>:46: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-2-d6a7185f00a3>:40: SyntaxWarning: "is" with a literal. Did you mean "=="?
    if action is 'U':
<ipython-input-2-d6a7185f00a3>:42: SyntaxWarning: "is" with a literal. Did you mean "=="?
    elif action is 'D':
<ipython-input-2-d6a7185f00a3>:44: SyntaxWarning: "is" with a literal. Did you mean "=="?
    elif action is 'L':
<ipython-input-2-d6a7185f00a3>:46: SyntaxWarning: "is" with a literal. Did you mean "=="?
    elif action is 'R':
```

```
BFS: ['U', 'R', 'U', 'L', 'D']
space: 66
time: 0.001922607421875

BFS: ['U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
space: 591
time: 0.009056568145751953

BFS: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
space: 2956
time: 0.06514430046081543

-----
```

EXPERIMENT – 5

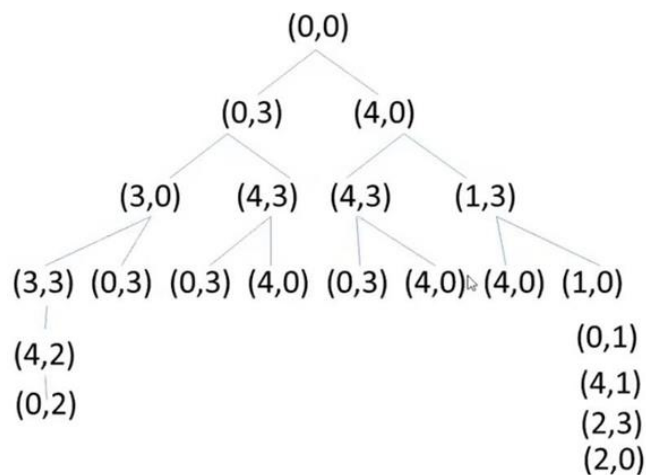
AIM - To Write a program to implement the water jug problem using BFS and DFS.

SOFTWARE USED — Python

THEORY -

Breadth-First Search (BFS) is a method for searching graphs or trees. Traversing the tree entails visiting every node. Breadth-First Search is a recursive method for searching all the nodes of a tree or graph. In Python, We can utilize data structures like lists or tuples to perform BFS. In trees and graphs, the breadth-first search is virtually identical. The only distinction is that the tree might have loops, which would enable us to revisit the same node.

Graph –



Algorithm

1. Initialise a queue to implement BFS.
2. Since, initially, both the jugs are empty, insert the state {0, 0} into the queue.
3. Perform the following state, till the queue becomes empty:
4. Pop out the first element of the queue.
5. If the value of popped element is equal to Z, return True.

6. Let X_left and Y_left be the amount of water left in the jugs respectively.
7. Now perform the fill operation:
 - a. If the value of $X_left < X$, insert $(\{X_left, Y\})$ into the hashmap, since this state hasn't been visited and some water can still be poured in the jug.
 - b. If the value of $Y_left < Y$, insert $(\{Y_left, X\})$ into the hashmap, since this state hasn't been visited and some water can still be poured in the jug.
8. Perform the empty operation:
 - a. If the state $(\{0, Y_left\})$ isn't visited, insert it into the hashmap, since we can empty any of the jugs.
 - b. Similarly, if the state $(\{X_left, 0\})$ isn't visited, insert it into the hashmap, since we can empty any of the jugs.
9. Perform the transfer of water operation:
 - a. $\min(\{X-X_left, Y\})$ can be poured from second jug to first jug. Therefore, in case – $\{X + \min(\{X-X_left, Y\}), Y - \min(\{X-X_left, Y\})\}$ isn't visited, put it into hashmap.
 - b. $\min(\{X_left, Y-Y_left\})$ can be poured from first jug to second jug. Therefore, in case – $\{X_left - \min(\{X_left, Y - X_left\}), Y + \min(\{X_left, Y - Y_left\})\}$ isn't visited, put it into hashmap.
10. Return False, since, it is not possible to measure Z litres.

CODE:-

```
from collections import deque
def Solution(a, b, target):
```

```
    m = {}
```

```
    isSolvable = False
    path = []
```

```
    q = deque()
    q.append((0, 0))
```

```
    while (len(q) > 0):
```

```
        u = q.popleft()
```

```
        if ((u[0], u[1]) in m):
```

```
            continue
```

```
        if ((u[0] > a or u[1] > b or
```

```

u[0] < 0 or u[1] < 0)):

continue path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target): isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(" + path[i][0] + ", ",

path[i][1] + ")")

break q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1): c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)): q.append([c, d])

c = u[0] - ap d = u[1] + ap

if ((c == 0 and c >= 0) or d == b): q.append([c, d])

q.append([a, 0])

q.append([0, b]) if (not isSolvable):

print("Solution not possible")

```

```
if __name__ == '__main__':  
  
    Jug1, Jug2, target = 4, 3, 2  
    print("Path from initial state "  
          "to solution state ::")  
  
    Solution(Jug1, Jug2, target)
```

OUTPUT:-

At the end of the experiment we can conclude that we were able to solve water jug problem using BFS successfully.

```
Path from initial state to solution state ::  
( 0 , 0 )  
( 0 , 3 )  
( 4 , 0 )  
( 4 , 3 )  
( 3 , 0 )  
( 1 , 3 )  
( 3 , 3 )  
( 4 , 2 )  
( 0 , 2 )  
  
...Program finished with exit code 0  
Press ENTER to exit console.□
```


EXPERIMENT – 6

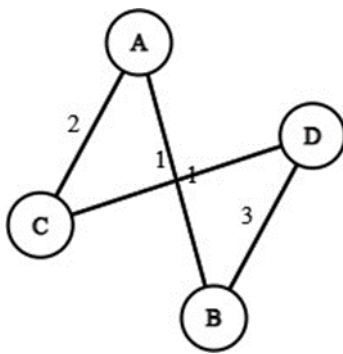
AIM - To Write a Python script to implement A* algorithm.

SOFTWARE USED – Python

THEORY -

A* Algorithm in Python or in general is basically an artificial intelligence problem used for the pathfinding (from point A to point B) and the Graph traversals. This algorithm is flexible and can be used in a wide range of contexts. The A* search algorithm uses the heuristic path cost, the starting point's cost, and the ending point.

Graph –



Algorithm –

1. Initialize the open list
 2. Initialize the closed list
- put the starting node on the open list (you can leave its f at zero)
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
 - II) else, compute both g and h for successor

```

        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

        III)if a node with the same position as successor is in the OPEN list which has a lower
f        than successor, skip this successor

        IV) 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)

```

CODE-

```

from collections import deque

class Graph:

    def __init__(self, adjac_lis): self.adjac_lis = adjac_lis

    def get_neighbors(self, v): return self.adjac_lis[v]

    def h(self, n): H = {
        'A': 1,
        'B': 1,
        'C': 1,
        'D': 1
    }
    return H[n]

    def a_star_algorithm(self, start, stop):
        open_lst = set([start])
        closed_lst = set([])
        poo = {}
        poo[start] = 0
        par = {}
        par[start] = start
        while len(open_lst) > 0:
            n = None
            for v in open_lst:

```

```

if n is None or poo[v] + self.h(v) < poo[n] + self.h(n): n = v;
if n is None:
    print('Path does not exist!') return None
if n == stop: reconst_path = [] while par[n] != n:
    reconst_path.append(n) n = par[n]
reconst_path.append(start) reconst_path.reverse()
print('Path found: {}'.format(reconst_path)) return reconst_path
for (m, weight) in self.get_neighbors(n):
    if m not in open_lst and m not in closed_lst: open_lst.add(m)
    par[m] = n
    poo[m] = poo[n] + weight

else:
    if poo[m] > poo[n] + weight: poo[m] = poo[n] + weight par[m] = n
    if m in closed_lst: closed_lst.remove(m) open_lst.add(m)
    open_lst.remove(n) closed_lst.add(n)
    print('Path does not exist!') return None

graph_data = {
    'A': [('B', 1), ('C', 2)],
    'B': [('A', 1), ('D', 3)],
    'C': [('A', 2), ('D', 1)],
    'D': [('B', 3), ('C', 1)]
}

graph = Graph(graph_data)
start_node = 'A' end_node = 'D'
graph.a_star_algorithm(start_node, end_node)

```

OUTPUT-

At the end of the practical we can conclude that we were able to implement A* algorithm successfully.

```
Path found: ['A', 'C', 'D']
```

```
...Program finished with exit code 0  
Press ENTER to exit console. 
```

EXPERIMENT – 9

AIM - To Write a Python script to implement MINMAX algorithm(tic tac toe game problem).

SOFTWARE USED — Python

THEORY -

Minimax is a game-theoretic approach used in decision making and game theory to find the optimal move for a player in a two-player game. In tic-tac-toe, the goal is to get three in a row either horizontally, vertically, or diagonally. The minimax algorithm can be used to determine the optimal move for a player, given the current state of the board. This is done by looking at all possible future moves and evaluating them using a heuristic score. The player then chooses the move with the highest score, as this is the most likely to lead to a win. In Python, this can be implemented using a recursive function which takes the current state of the board as an argument and returns the best move for the player.

CODE:-

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system

"""
An implementation of Minimax AI Algorithm in Tic Tac Toe,
using Python.
"""

HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
```

```

def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score

def wins(state, player):
    """
    This function tests if a specific player wins. Possibilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False

def game_over(state):
    """
    This function test if the human or computer wins
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)

def empty_cells(state):
    """
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):

```

```

        if cell == 0:
            cells.append([x, y])

    return cells

def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
    else:
        return False

def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False

def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score # max value

```

```

        else:
            if score[2] < best[2]:
                best = score # min value

    return best

def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')

def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '-----'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)

def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it chooses a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """

    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

```



```

    set_move(x, y, COMP)
    time.sleep(1)

def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = '' # X or O
    c_choice = '' # X or O
    first = '' # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')

```

```

        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Setting computer's choice
if h_choice == 'X':
    c_choice = 'O'
else:
    c_choice = 'X'

# Human may starts first
clean()
while first != 'Y' and first != 'N':
    try:
        first = input('First to start?[y/n]: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')

# Main loop of this game
while len(empty_cells(board)) > 0 and not game_over(board):
    if first == 'N':
        ai_turn(c_choice, h_choice)
        first = ''

    human_turn(c_choice, h_choice)
    ai_turn(c_choice, h_choice)

# Game over message
if wins(board, HUMAN):
    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)
    print('YOU WIN!')
elif wins(board, COMP):
    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)
    print('YOU LOSE!')
else:
    clean()
    render(board, c_choice, h_choice)
    print('DRAW!')

exit()

if __name__ == '__main__':
    main()

```

OUTPUT:-

Choose X or O
Chosen: x
First to start?[y/n]: y
Human turn [X]

```
-----  
|  |  |  |  
-----  
|  |  |  |  
-----  
|  |  |  |  
-----
```

Use numpad (1..9): 1
Computer turn [O]

```
-----  
| x |  |  |  
-----  
|  |  |  |  
-----  
|  |  |  |  
-----
```

Human turn [X]

```
-----  
| x |  |  |  
-----  
|  | O |  |  
-----  
|  |  |  |  
-----
```

Use numpad (1..9): 9
Computer turn [O]

```
-----  
| x |  |  |  
-----  
|  | O |  |  
-----  
|  |  | x |  
-----
```

Human turn [X]

```
-----  
| x | O |  |  
-----  
|  | O |  |  
-----  
|  |  | x |  
-----
```

Use numpad (1..9): 8
Computer turn [O]

```
-----  
| x | O |  |  
-----
```

```
|  || 0 ||  |
-----
```

```
|  || x || x |
-----
```

Human turn [X]

```
-----
| x || 0 ||  |
-----
```

```
|  || 0 ||  |
-----
```

```
| 0 || x || x |
-----
```

Use numpad (1..9): 3

Computer turn [O]

```
-----
| x || 0 || x |
-----
```

```
|  || 0 ||  |
-----
```

```
| 0 || x || x |
-----
```

Human turn [X]

```
-----
| x || 0 || x |
-----
```

```
|  || 0 || 0 |
-----
```

```
| 0 || x || x |
-----
```

Use numpad (1..9): 4

```
-----
| x || 0 || x |
-----
```

```
| x || 0 || 0 |
-----
```

```
| 0 || x || x |
-----
```

DRAW!

EXPERIMENT – 7

AIM - To Write a Python script to implement tic tac toe game problem(alpha beta pruning.).

SOFTWARE USED — Python

THEORY -

Alpha beta pruning is an optimization technique used in game theory to reduce the amount of time needed to make a decision. It is an extension of the minimax algorithm, which is used to determine the best move for a player in a two-player game. Alpha beta pruning works by analyzing the game tree and eliminating branches which cannot possibly lead to a win for either player. This reduces the number of possible moves that must be evaluated, and results in a more efficient decision-making process. In tic-tac-toe, alpha beta pruning can be used to reduce the time needed to determine the best move for a player by eliminating branches which cannot lead to a win.

CODE-

```
from random import choice
from math import inf

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

def Gameboard(board):
    chars = {1: 'X', -1: 'O', 0: ' '}
    for x in board:
        for y in x:
            ch = chars[y]
            print(f'| {ch} |', end='')
            print('\n' + '-----')
        print('=====')

def Clearboard(board):
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            board[x][y] = 0

def winningPlayer(board, player):
    conditions = [[board[0][0], board[0][1], board[0][2],
                    board[1][0], board[1][1], board[1][2],
                    board[2][0], board[2][1], board[2][2],
                    board[0][0], board[1][0], board[2][0],
                    board[0][1], board[1][1], board[2][1],
                    board[0][2], board[1][2], board[2][2],
                    board[0][0], board[1][1], board[2][2],
                    board[0][2], board[1][1], board[2][0]]

    if [player, player, player] in conditions:
        return True

    return False

def gameWon(board):
    return winningPlayer(board, 1) or winningPlayer(board, -1)
```

```

def printResult(board):
    if winningPlayer(board, 1):
        print('X has won! ' + '\n')

    elif winningPlayer(board, -1):
        print('O\'s have won! ' + '\n')

    else:
        print('Draw' + '\n')

def blanks(board):
    blank = []
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            if board[x][y] == 0:
                blank.append([x, y])

    return blank

def boardFull(board):
    if len(blanks(board)) == 0:
        return True
    return False

def setMove(board, x, y, player):
    board[x][y] = player

def playerMove(board):
    e = True
    moves = {1: [0, 0], 2: [0, 1], 3: [0, 2],
             4: [1, 0], 5: [1, 1], 6: [1, 2],
             7: [2, 0], 8: [2, 1], 9: [2, 2]}
    while e:
        try:
            move = int(input('Enter a number between 1-9: '))
            if move < 1 or move > 9:
                print('Invalid Move! Try again!')
            elif not (moves[move] in blanks(board)):
                print('Invalid Move! Try again!')
            else:
                setMove(board, moves[move][0], moves[move][1], 1)
                Gameboard(board)
                e = False
        except (KeyError, ValueError):
            print('Enter a number!')

def getScore(board):
    if winningPlayer(board, 1):
        return 10

    elif winningPlayer(board, -1):
        return -10

    else:
        return 0

def abminimax(board, depth, alpha, beta, player):
    row = -1

```

```

col = -1
if depth == 0 or gameWon(board):
    return [row, col, getScore(board)]

else:
    for cell in blanks(board):
        setMove(board, cell[0], cell[1], player)
        score = abminimax(board, depth - 1, alpha, beta, -player)
        if player == 1:
            # X is always the max player
            if score[2] > alpha:
                alpha = score[2]
                row = cell[0]
                col = cell[1]

        else:
            if score[2] < beta:
                beta = score[2]
                row = cell[0]
                col = cell[1]

        setMove(board, cell[0], cell[1], 0)
        if alpha >= beta:
            break

    if player == 1:
        return [row, col, alpha]

    else:
        return [row, col, beta]

def o_comp(board):
    if len(blanks(board)) == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
        setMove(board, x, y, -1)
        Gameboard(board)

    else:
        result = abminimax(board, len(blanks(board)), -inf, inf, -1)
        setMove(board, result[0], result[1], -1)
        Gameboard(board)

def x_comp(board):
    if len(blanks(board)) == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
        setMove(board, x, y, 1)
        Gameboard(board)

    else:
        result = abminimax(board, len(blanks(board)), -inf, inf, 1)
        setMove(board, result[0], result[1], 1)
        Gameboard(board)

def makeMove(board, player, mode):
    if mode == 1:
        if player == 1:

```

```

        playerMove(board)

    else:
        o_comp(board)
    else:
        if player == 1:
            o_comp(board)
        else:
            x_comp(board)

def pvc():
    while True:
        try:
            order = int(input('Enter to play 1st or 2nd: '))
            if not (order == 1 or order == 2):
                print('Please pick 1 or 2')
            else:
                break
        except(KeyError, ValueError):
            print('Enter a number')

    Clearboard(board)
    if order == 2:
        currentPlayer = -1
    else:
        currentPlayer = 1

    while not (boardFull(board) or gameWon(board)):
        makeMove(board, currentPlayer, 1)
        currentPlayer *= -1

    printResult(board)

# Driver Code
print("=====")
print("TIC-TAC-TOE using MINIMAX with ALPHA-BETA Pruning")
print("=====")
pvc()

```

OUTPUT-

```

from random import choice
from math import inf

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

def Gameboard(board):
    chars = {1: 'X', -1: 'O', 0: ' '}
    for x in board:
        for y in x:
            ch = chars[y]
            print(f'| {ch} |', end='')

```



```

        print('\n' + '-----')
    print('=====')

def Clearboard(board):
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            board[x][y] = 0

def winningPlayer(board, player):
    conditions = [[board[0][0], board[0][1], board[0][2]],
                  [board[1][0], board[1][1], board[1][2]],
                  [board[2][0], board[2][1], board[2][2]],
                  [board[0][0], board[1][0], board[2][0]],
                  [board[0][1], board[1][1], board[2][1]],
                  [board[0][2], board[1][2], board[2][2]],
                  [board[0][0], board[1][1], board[2][2]],
                  [board[0][2], board[1][1], board[2][0]]]

    if [player, player, player] in conditions:
        return True

    return False

def gameWon(board):
    return winningPlayer(board, 1) or winningPlayer(board, -1)

def printResult(board):
    if winningPlayer(board, 1):
        print('X has won! ' + '\n')

    elif winningPlayer(board, -1):
        print('O\'s have won! ' + '\n')

    else:
        print('Draw' + '\n')

def blanks(board):
    blank = []
    for x, row in enumerate(board):
        for y, col in enumerate(row):
            if board[x][y] == 0:
                blank.append([x, y])

    return blank

def boardFull(board):
    if len(blanks(board)) == 0:
        return True
    return False

def setMove(board, x, y, player):
    board[x][y] = player

def playerMove(board):
    e = True
    moves = {1: [0, 0], 2: [0, 1], 3: [0, 2],
              4: [1, 0], 5: [1, 1], 6: [1, 2],
              7: [2, 0], 8: [2, 1], 9: [2, 2]}

```

```

while e:
    try:
        move = int(input('Enter a number between 1-9: '))
        if move < 1 or move > 9:
            print('Invalid Move! Try again!')
        elif not (moves[move] in blanks(board)):
            print('Invalid Move! Try again!')
        else:
            setMove(board, moves[move][0], moves[move][1], 1)
            Gameboard(board)
            e = False
    except(KeyError, ValueError):
        print('Enter a number!')

def getScore(board):
    if winningPlayer(board, 1):
        return 10

    elif winningPlayer(board, -1):
        return -10

    else:
        return 0

def abminimax(board, depth, alpha, beta, player):
    row = -1
    col = -1
    if depth == 0 or gameWon(board):
        return [row, col, getScore(board)]

    else:
        for cell in blanks(board):
            setMove(board, cell[0], cell[1], player)
            score = abminimax(board, depth - 1, alpha, beta, -player)
            if player == 1:
                # X is always the max player
                if score[2] > alpha:
                    alpha = score[2]
                    row = cell[0]
                    col = cell[1]

            else:
                if score[2] < beta:
                    beta = score[2]
                    row = cell[0]
                    col = cell[1]

            setMove(board, cell[0], cell[1], 0)
            if alpha >= beta:
                break

        if player == 1:
            return [row, col, alpha]

        else:
            return [row, col, beta]

def o_comp(board):

```

```

if len(blanks(board)) == 9:
    x = choice([0, 1, 2])
    y = choice([0, 1, 2])
    setMove(board, x, y, -1)
    Gameboard(board)

else:
    result = abminimax(board, len(blanks(board)), -inf, inf, -1)
    setMove(board, result[0], result[1], -1)
    Gameboard(board)

def x_comp(board):
    if len(blanks(board)) == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
        setMove(board, x, y, 1)
        Gameboard(board)

    else:
        result = abminimax(board, len(blanks(board)), -inf, inf, 1)
        setMove(board, result[0], result[1], 1)
        Gameboard(board)

def makeMove(board, player, mode):
    if mode == 1:
        if player == 1:
            playerMove(board)

        else:
            o_comp(board)
    else:
        if player == 1:
            o_comp(board)
        else:
            x_comp(board)

def pvc():
    while True:
        try:
            order = int(input('Enter to play 1st or 2nd: '))
            if not (order == 1 or order == 2):
                print('Please pick 1 or 2')
            else:
                break
        except (KeyError, ValueError):
            print('Enter a number')

    Clearboard(board)
    if order == 2:
        currentPlayer = -1
    else:
        currentPlayer = 1

    while not (boardFull(board) or gameWon(board)):
        makeMove(board, currentPlayer, 1)
        currentPlayer *= -1

    printResult(board)

```

```
# Driver Code
print("=====")
print("TIC-TAC-TOE using MINIMAX with ALPHA-BETA Pruning")
print("=====")
pvc()
```

EXPERIMENT – 12

AIM - To do text pre-processing - Tokenization, Stop word removal, Stemming/Lemmatization, Parts of speech tagging, Rule Generation for Parsing

SOFTWARE USED — Python

THEORY -

Tokenization - Sentence tokenization refers to the process of separating the lines of a paragraph using a delimiter. Word tokenization means separating words of a sentence into its constituent words, example “She is a girl” will be tokenized to ‘She’, ‘is’, ‘a’, ‘girl’, ‘.’.

Stop word removal - This entails the process of discarding the words which do not incur significant meaning from a sentence so that more importance and weightage is imparted to the words which hold significant meaning, example words like ‘is’, ‘are’.

Stemming - Process of converting a word to its root form to find patterns and relations among different words, for example caring converted to car and cared converted to care.

Lemmatization - Process of converting a word to root or base form but this method, unlike stemming, takes care of meaning perseverance, for example caring is changed to care and not car.

Parts of speech tagging - This is the process of tagging words of a sentence into different parts of speech, example boy will be tagged as noun and dancing as a verb. They are used for syntax analysis and parsing in Natural Language Processing.

Source Code & Output

1. Tokenization

```
import nltk
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
text = "My name is John. I am 20 years old. I live in New York."
print(sent_tokenize(text))
print(word_tokenize(text))
word_tokens = word_tokenize(text)
```

```
['My name is John.', 'I am 20 years old.', 'I live in New York.']
['My', 'name', 'is', 'John', '.', 'I', 'am', '20', 'years', 'old',
 '.', 'I', 'live', 'in', 'New', 'York', '.']
```

2. Stop Word Removal

```
nlk.download('stopwords')
from nltk.corpus import stopwords
stop_words = set(stopwords.words('english'))
filtered_sentence = []
for w in word_tokens:
    if w not in stop_words:
        filtered_sentence.append(w)
print(filtered_sentence)
```

```
['My', 'name', 'John', '.', 'I', '20', 'years', 'old',
 '.', 'I', 'live', 'New', 'York', '.']
```

3. Lemmatization

```
nlk.download('wordnet')
from nltk.corpus import wordnet as wn
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
lemmatized_text = [lemmatizer.lemmatize(token) for token in word_tokens]
display(f"lemmatized nltk text: {lemmatized_text}")
```

```
lemmatized nltk text: ['My', 'name', 'is', 'John', '.', 'I', 'am',
 '20', 'year', 'old', '.', 'I', 'live', 'in', 'New', 'York', '.']
```

4. Stemming

```
from nltk.stem import PorterStemmer
porter = PorterStemmer()
stemmed_words = [porter.stem(token) for token in word_tokens]
display(f"stemmed nltk text: {stemmed_words}")
```

```
stemmed nltk text: ['my', 'name', 'is', 'john', '.',
 'i', 'am', '20', 'year', 'old', '.', 'i', 'live', 'in', 'new', 'york', '.']
```

5. Parts of speech tagging

```
nltk.download('averaged_perceptron_tagger')
tagged = nltk.pos_tag(word_tokens)
print(tagged)
```

```
[('My', 'PRP$'), ('name', 'NN'), ('is', 'VBZ'), ('John', 'NNP'),
('.', '.'), ('I', 'PRP'), ('am', 'VBP'), ('20', 'CD'), ('years', 'NNS'),
('old', 'JJ'), ('.', '.'), ('I', 'PRP'), ('live', 'VBP'), ('in', 'IN'),
('New', 'NNP'), ('York', 'NNP'), ('.', '.')]

```

6. Parse tree generation

```
import nltk
# Define the updated grammar
grammar = nltk.CFG.fromstring("""
    S -> NP VP
    NP -> 'There' | N P
    VP -> V V NP PP
    V -> 'have' | 'been' | 'saw'
    PP -> PRP V PRP
    P -> 'since'
    N -> 'centuries'
    DET -> 'a' | 'an' | 'the'
    ADJ -> 'old' | 'long'
    PRP -> 'I' | 'her'
""")
# Define the input sentence
sentence = "There have been centuries since I saw her"
# Create a parser
parser = nltk.ChartParser(grammar)
# Parse the sentence and print the parse tree
for tree in parser.parse(sentence.split()):
    print(tree)
```

```
(S
  (NP There)
  (VP
    (V have)
    (V been)
    (NP (N centuries) (P since))
    (PP (PRP I) (V saw) (PRP her))))

```

EXPERIMENT – 10

AIM - To Implement 0/1 knapsack using brute force in python.

SOFTWARE USED – Python

THEORY -

The 0/1 knapsack problem is a well-known optimization problem in computer science and operations research. The problem involves a knapsack with a limited capacity and a set of items, each with a weight and a value. The goal is to select a subset of the items that maximizes the total value while keeping the total weight within the capacity of the knapsack. The name "0/1" comes from the fact that each item can only be selected once (1) or not at all (0). The brute force approach to solving the 0/1 knapsack problem involves generating all possible subsets of the items and selecting the one that has the highest total value and meets the weight constraint. Since there are 2^n possible subsets (where n is the number of items), this approach becomes impractical for large problem instances. The following code uses a recursive implementation of the brute force approach to solve the 0/1 knapsack problem. The `knapSack` function recursively explores all possible subsets of the items, starting with the last item in the list and working backwards. At each step, the function checks whether the weight of the current item is within the remaining capacity of the knapsack. If so, the function recursively explores both the possibility of including and not including the current item, selecting the option that yields the higher total value. The function returns the maximum total value that can be obtained by selecting a subset of the items that fits within the knapsack, as well as the indices of the items in the selected subset. The final print statements display the maximum value and the list of selected items with their corresponding values and weights

Source Code

```
def knapSack(W, wt, val, n):  
    # Base Case
```



```

if n == 0 or W == 0:
    return 0, []

# If weight of nth item is more than Knapsack capacity W,
# then this item cannot be included
if (wt[n-1] > W):
    return knapSack(W, wt, val, n-1)

# Return the maximum of two cases: (1) nth item included (2) not included
without_val, without_items = knapSack(W, wt, val, n-1)
with_val, with_items = knapSack(W-wt[n-1], wt, val, n-1)
with_val += val[n-1]
with_items.append(n-1)

if with_val > without_val:
    return with_val, with_items
else:
    return without_val, without_items

val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)

max_val, items = knapSack(W, wt, val, n)

print("Original list of items:")
for i in range(n):
    print(" - Item", i+1, "with value", val[i], "and weight", wt[i])

print("\nMaximum value:", max_val)
print("Items selected:")
for i in items:
    print(" - Item", i+1, "with value", val[i], "and weight", wt[i])

```

Output

```

Original list of items:
- Item 1 with value 60 and weight 10
- Item 2 with value 100 and weight 20
- Item 3 with value 120 and weight 30

Maximum value: 220
Items selected:
- Item 2 with value 100 and weight 20
- Item 3 with value 120 and weight 30

```

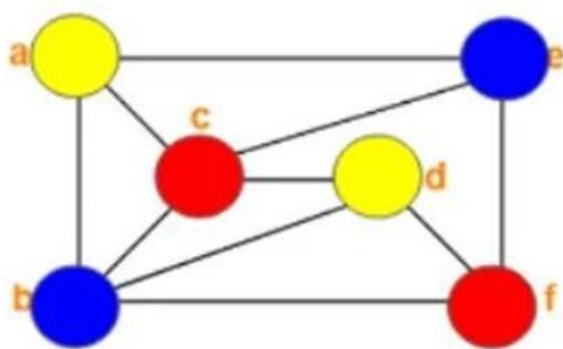
EXPERIMENT – 8

AIM - To Implement 0/1 knapsack using brute force in python.

SOFTWARE USED – Python

THEORY -

Graph:



Graph Coloring Problem:

In the graph coloring problem, we have a graph and m colors, we need to find a way to color the vertices of the graph using the m colors such that any two adjacent vertices are not having the same color. The graph coloring problem is also known as the vertex coloring problem.

Algorithm –

1. Initiate all the nodes.
2. Set the node for the first coloring, the priority is the node with the largest degree.
3. Choose the color candidate with the selection color function with no adjacent node having the same color.
4. Check the eligibility of the color, if it's able to save to the solution set.
5. Is the solution complete? Go to step 2 if not yet.

CODE-

```
G = [[ 0, 1, 1, 0, 1, 0],  
      [ 1, 0, 1, 1, 0, 1],  
      [ 1, 1, 0, 1, 1, 0],  
      [ 0, 1, 1, 0, 0, 1],
```

```

[ 1, 0, 1, 0, 0, 1],
[ 0, 1, 0, 1, 1, 0]]
node = "abcdef" t_={}
for i in range(len(G)): t_[node[i]] = i
degree=[]
for i in range(len(G)): degree.append(sum(G[i]))
colorDict = {}
for i in range(len(G)):
colorDict[node[i]]=["Blue","Red","Yellow","Green"]
sortedNode=[] indeks = []
for i in range(len(degree)):
_max = 0
j = 0
for j in range(len(degree)):
if j not in indeks:
if degree[j] > _max:
_max = degree[j] idx = j
indeks.append(idx) sortedNode.append(node[idx])
theSolution={}
for n in sortedNode:
setTheColor = colorDict[n] theSolution[n] = setTheColor[0] adjacentNode = G[t_[n]]
for j in range(len(adjacentNode)):
if adjacentNode[j]==1 and (setTheColor[0] in colorDict[node[j]]):
colorDict[node[j]].remove(setTheColor[0])
for t,w in sorted(theSolution.items()):
print("Node",t," = ",w)

```

OUTPUT-

At the end of the practical we can conclude that we were able to successfully color the given graph with nodes a,b,c,d,e,f.

```
Node a = Yellow  
Node b = Blue  
Node c = Red  
Node d = Yellow  
Node e = Blue  
Node f = Red
```

```
...Program finished with exit code 0  
Press ENTER to exit console.□
```