**AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY**

**AMITY UNIVERSITY UTTAR PRADESH**



**Artificial Intelligence Practical File**

**Submitted by:** Prateek Sharma A2305218146 7CSE 3X

**Submitted to:**

Mr. Deepak Gaur

# INDEX

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S.**  **No** | **Category of Assignment** | **Code** | **Prog. No.** | **Name of Program** | **Date of Allotment** | **Submission Date** | **Max Mark** | **Marks Obtained** | **Signature of Faculty** |
| **1.** | **Mandatory Experiment** | **LR (10)** | **1** | Basics of Python programming |  | 21-07-2021 | **1** |  |  |
| **2.** | **Mandatory Experiment** | **2** | Water-jug Problem by BFS |  | 28-07-2021 | **1** |  |  |
| **3.** | **Mandatory Experiment** | **3** | 1. Graph colouring Problem 2. Depth-first Search |  | 04-08-2021 | **1** |  |  |
| **4.** | **Mandatory Experiment** | **4** | 1. Best-first Search 2. A\* algorithm |  | 11-08-2021 | **1** |  |  |
| **5.** | **Mandatory Experiment** | **5** | Constraint satisfaction algorithm |  | 18-08-2021 | **1** |  |  |
| **6.** | **Mandatory Experiment** | **6** | Tic-Tac-Toe by Mini-max algorithm |  | 31-08-2021 | **1** |  |  |
| **7.** | **Open-ended Experiment** | **7** | 4-Queens Problem |  | 07-09-2021 | **1** |  |  |
| **8.** | **Mandatory Experiment** | **8** | Snakes and Ladders |  | 21-09-2021 | **1** |  |  |
| **9.** | **Mandatory Experiment** | **9** | Tokenization using NLTK |  | 28-09-2021 | **1** |  |  |
| **10.** | **Mandatory Experiment** | **10** | Knapsack Problem |  | 05-10-2021 | **1** |  |  |
| **11.** | **Mandatory Experiment** |  | **11** | Design an XOR truth table |  | 12-10-2021 | **1** |  |  |
| **12.** | **Performance** | **PR (10)** |  |  |  |  | **10** |  |  |
| **13.** | **Viva** | **Viva (5)** |  |  |  |  | **5** |  |  |

**EXPERIMENT - 1**

**PROGRAM 1**

**AIM:** WAP to implement Linear Search

**THEORY:**

Linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched.

**CODE**

def linearSearch(list1,key): for i in range(len(list1)):

if key == list1[i]:

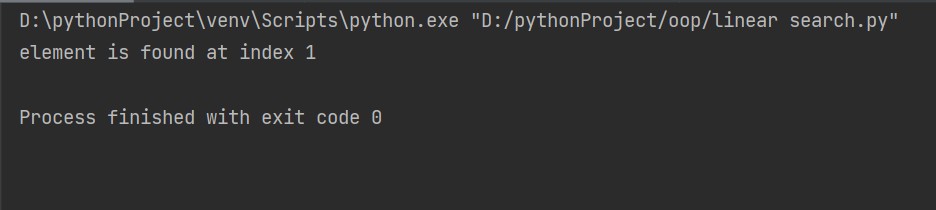
print('element is found at index',i) break

else:

print('element is not found')

list1 = [1,2,3,4,5]

linearSearch(list1,2)

**OUTPUT**

**PROGRAM 2**

**AIM:** WAP to implement Binary Search

**ALGO:**

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise, narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

**CODE**

1. **Iterative Method**

def binarysearch(list1,key): low = 0

high = len(list1)-1 found = False

while low<=high and not found: mid = (low+high)//2

if key == list1[mid]: found = True

elif key>list1[low]: low = mid+1

else:

high = mid-1 if found == True:

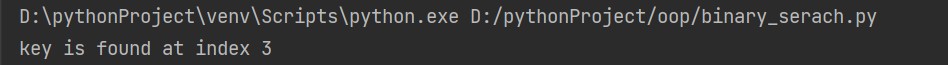
print('key is found at index',mid) else:

print('key is not found')

list1=[1,2,3,4,5]

list1.sort() binarysearch(list1,4)

**OUTPUT:**



1. **Recursive**

flag = False

def binarySearch(arr, low, high, x): if high >= low:

mid = (high + low) // 2 if arr[mid] == x:

return mid flag = True

elif arr[mid] > x:

return binarySearch(arr, low, mid - 1, x) else:

return binarySearch(arr, mid + 1, high, x)

else:

flag = False arr = [2, 3, 4, 10, 40]

arr.sort() key = 10

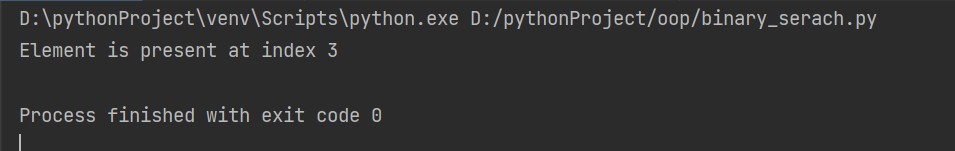
result = binarySearch(arr, 0, len(arr)-1, key)

if result:

print("Element is present at index", str(result)) else:

print("Element is not present")

Output:



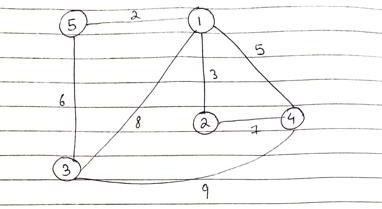
**PROGRAM 3**

**AIM:** WAP to implement Prims Algorithm

**ALGO:**

1. Create a set *mstSet* that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign key value as 0 for the first vertex so that it is picked first.
3. While mstSet doesn’t include all vertices
   1. Pick a vertex *u* which is not there in *mstSet* and has minimum key value.
   2. Include *u* tomstset.
   3. Update key value of all adjacent vertices of *u*. To update the key values, iterate through all adjacent vertices. For every adjacent vertex *v*, if weight of edge *u-v* is less than the previous key value of *v*, update the key value as weight of *u-v*

**GRAPH USED:**



|  |  |
| --- | --- |
| **CODE** |  |
| INF = 9999999 |  |
| V = 5 |  |
| G = [[0, 9, 75, | 0, 0], |
| [9, 0, 95, | 19, 42], |
| [75, 95, 0, 51, 66],  [0, 19, 51, 0, 31],  [0, 42, 66, 31, 0]] | |

selected = [0, 0, 0, 0, 0]

no\_edge = 0 selected[0] = True

print("Edge : Weight\n") while (no\_edge < V - 1):

minimum = INF x = 0

y = 0

for i in range(V): if selected[i]:

for j in range(V):

if ((not selected[j]) and G[i][j]):

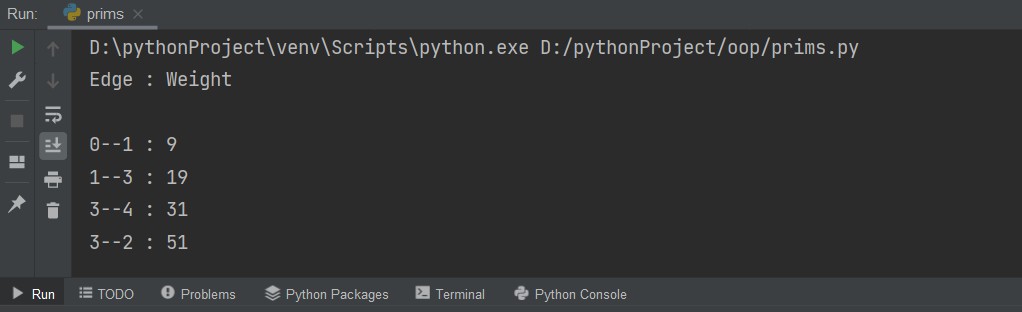
if minimum > G[i][j]: minimum = G[i][j] x = i

y = j

print(str(x) + "--" + str(y) + " : " + str(G[x][y])) selected[y] = True

no\_edge += 1

**OUTPUT**



**PROGRAM 4**

**AIM:** WAP to implement Kruskal’s Algorithm

**THEORY:**

Kruskal's algorithm finds a [minimum spanning forest](https://en.wikipedia.org/wiki/Minimum_spanning_tree) of an undirected [edge-](https://en.wikipedia.org/wiki/Weighted_graph) [weighted graph](https://en.wikipedia.org/wiki/Weighted_graph). If the graph is [connected](https://en.wikipedia.org/wiki/Connectivity_(graph_theory)), it finds a [minimum spanning tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree). It is a [greedy algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm) in [graph theory](https://en.wikipedia.org/wiki/Graph_theory) as in each step it adds the next lowest-weight edge that will not form a [cycle](https://en.wikipedia.org/wiki/Cycle_(graph_theory)) to the minimum spanning forest.

**Algorithm:**

### **Step 1:** Create a forest in such a way that each graph is a separate tree.

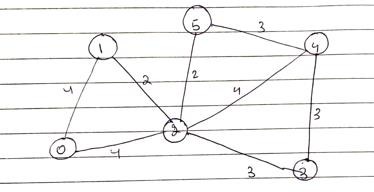
* **Step 2:** Create a priority queue Q that contains all the edges of the graph.
* **Step 3:** Repeat Steps 4 and 5 while Q is NOT EMPTY
* **Step 4:** Remove an edge from Q

### **Step 5:** IF the edge obtained in Step 4 connects two different trees, then Add it to the forest (for combining two trees into one tree). ELSE

Discard the edge

* **Step 6:** END

**GRAPH USED:**



**CODE**

class Graph:

def init (self, vertices): self.V = vertices self.graph = []

def add\_edge(self, u, v, w): self.graph.append([u, v, w])

def find(self, parent, i): if parent[i] == i:

return i

return self.find(parent, parent[i])

def apply\_union(self, parent, rank, x, y): xroot = self.find(parent, x)

yroot = self.find(parent, y) if rank[xroot] < rank[yroot]:

parent[xroot] = yroot

elif rank[xroot] > rank[yroot]: parent[yroot] = xroot

else:

parent[yroot] = xroot rank[xroot] += 1

def kruskal\_algo(self): result = []

i, e = 0, 0

self.graph = sorted(self.graph, key=lambda item: item[2]) parent = []

rank = []

for node in range(self.V): parent.append(node) rank.append(0)

while e < self.V - 1:

u, v, w = self.graph[i] i = i + 1

x = self.find(parent, u) y = self.find(parent, v) if x != y:

e = e + 1 result.append([u, v, w])

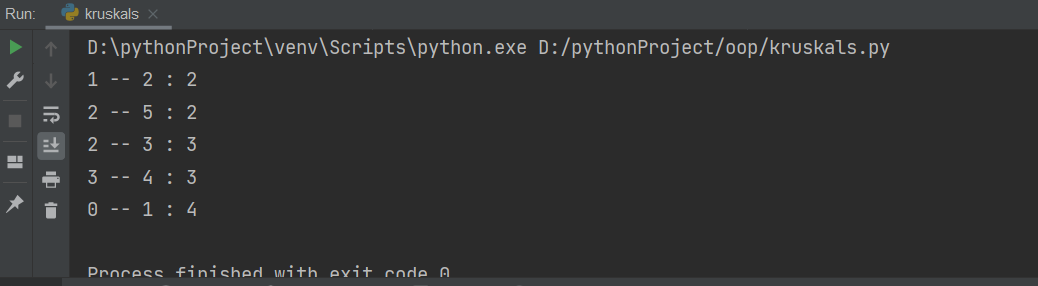
self.apply\_union(parent, rank, x, y) for u, v, weight in result:

print("%d -- %d : %d" % (u, v, weight))

g = Graph(6) g.add\_edge(0, 1, 4)

|  |  |  |
| --- | --- | --- |
| g.add\_edge(0, | 2, | 4) |
| g.add\_edge(1, | 2, | 2) |
| g.add\_edge(1, | 0, | 4) |
| g.add\_edge(2, | 0, | 4) |
| g.add\_edge(2, | 1, | 2) |
| g.add\_edge(2, | 3, | 3) |
| g.add\_edge(2, | 5, | 2) |
| g.add\_edge(2, | 4, | 4) |
| g.add\_edge(3, | 2, | 3) |
| g.add\_edge(3, | 4, | 3) |
| g.add\_edge(4, | 2, | 4) |
| g.add\_edge(4, | 3, | 3) |
| g.add\_edge(5, | 2, | 2) |
| g.add\_edge(5, 4,  g.kruskal\_algo() | | 3) |

## OUTPUT



**EXPERIMENT - 2**

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

**THEORY**: Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

Water Jug Problem:

You are given a m liter jug and a n liter 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 liters of water where d is less than n.

Production Rules used:

* First we initialize the state and pop off used state, if the state is already visited then continue.
* If the solution state is reached then ans=1, if no solution exists ans=0.
* If we have not reached the final state, we will start developing intermediate states to reach solution state.
* After filling jug 1 and jug 2, and we will check if specified state is possible or not, and accordingly we will empty or fill.
* At the end we will run the function with desired inputs.

**CODE**

from collections import deque def BFS(jug1, jug2, 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] > jug1 or u[1] > jug2 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]])

for i in range(len(path): print("(", path[i][0], " ",

path[i][1], ")")

break q.append([u[0], jug2])

q.append([jug1, u[1]])

for ap in range(max(jug1, jug2) + 1): c = u[0] + ap

d = u[1] - ap

if (c == jug1 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 == jug2): q.append([c, d])

q.append([jug1, 0])

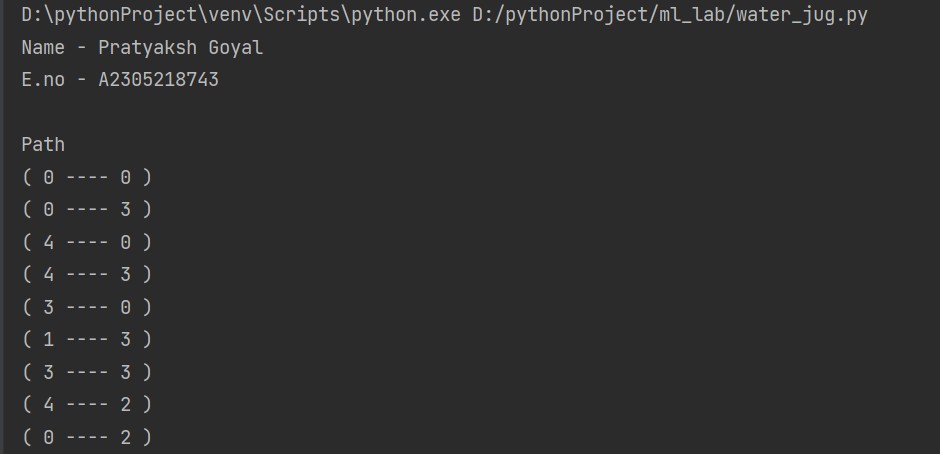
q.append([0, jug2]) if (not isSolvable):

print("No solution")

Jug1, Jug2, target = 4, 3, 2 print("Path")

BFS(Jug1, Jug2, target)

## OUTPUT



**EXPERIMENT - 3**

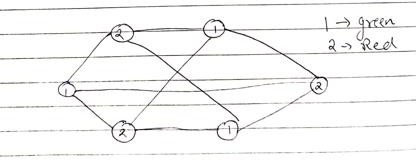
**AIM:** Implement Graph colouring problem using python.

**THEORY**:

Graph Colouring Problem:

Graph colouring problem is to assign colours to certain elements of a graph subject to certain constraints. Vertex colouring is the most common graph colouring problem. The problem is, given m colors, find a way of colouring the vertices of a graph such that no two adjacent vertices are coloured using same colour.

**GRAPH USED**:



**CODE**

class Graph:

def init (self, edges, N): self.adj = [[] for \_ in range(N)]

for (src, dest) in edges: self.adj[src].append(dest) self.adj[dest].append(src)

def colorGraph(graph): result = {}

for u in range(N):

assigned = set([result.get(i) for i in graph.adj[u] if i in

result])

color = 1

for c in assigned: if color != c:

break

color = color + 1 result[u] = color

for v in range(N):

print("Vertex", v, "color", colors[result[v]])

colors = ["", "GREEN", 'YELLOW', 'RED']

edges = [(0, 1), (0, 4), (0, 2), (0, 3), (1, 5), (1, 3), (2, 5),(2,

4), (3, 5), (4, 5)]

N = 6

graph = Graph(edges, N)

colorGraph(graph)

**OUTPUT**



**EXPERIMENT - 3(1)**

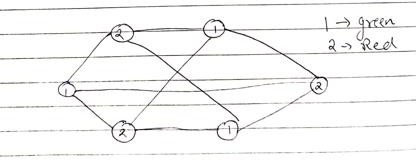
**AIM:** Write a program to implement DFS using Python

**THEORY**:

Depth-first Search (DFS):

Depth-first Search is a tree or graph traversal technique wherein upon reaching a node, the immediately next node traversed is the adjacent node to current node. Thus, the algorithm starts at the root node and explores as far as possible along each branch before backtracking.

**GRAPH USED**:



**CODE**

def dfs(visited, g, src): if visited[src]==0:

print(src, end =" ") visited[src]=1

for nbr in g[src]: dfs(visited, g, nbr)

def addEdge(g, x, y): g[x].append(y)

g[y].append(x) return g

g = [[] for i in range(6)] g = addEdge(g, 0, 1)

g = addEdge(g, 0, 3)

g = addEdge(g, 0, 5)

g = addEdge(g, 1, 2)

g = addEdge(g, 1, 4)

g = addEdge(g, 2, 3)

g = addEdge(g, 2, 5)

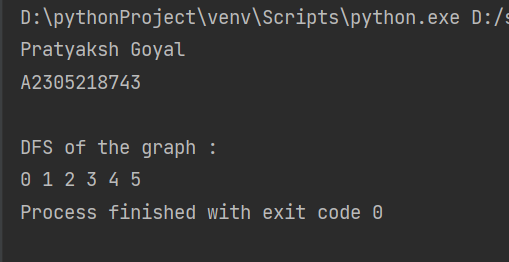
g = addEdge(g, 3, 4)

g = addEdge(g, 4, 5)

visited = [0] \* 6

print("DFS of the graph : ") dfs(visited,g,0)

**OUTPUT**



**EXPERIMENT - 4**

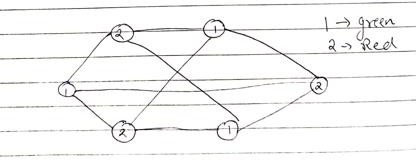
**AIM:** Write a program to implement Best First Search algorithm in python

**THEORY**:

Best First Search:

Best first search is an informed/heuristic method of traversal that decides which node is to be visited nest by checking which node is the most promising one. Graph traversal problem is implemented using priority queue.

**GRAPH USED**:



Algorithm-

Best-First-Search(Grah g, Node start)

* + 1. Create an empty PriorityQueue PriorityQueue pq;
    2. Insert "start" in pq. pq.insert(start)
    3. Until PriorityQueue is empty

u = PriorityQueue.DeleteMin If u is the goal

Exit Else

Foreach neighbor v of u If v "Unvisited"

Mark v "Visited" pq.insert(v)

Mark u "Examined"

End procedure

## CODE

from queue import PriorityQueue

def bestFirstSearch(g,src,dest,n,dist): vis = [0] \* n

vis[src] = 1 q=PriorityQueue() q.put((0, src))

while q.empty() == 0: node = q.get()[1] print(node,end=" ") if node == dest:

break

for nbr in g[node]: if vis[nbr] == 0:

vis[nbr] = 1 q.put((dist[nbr],nbr))

def addEdge(g, x, y): g[x].append(y)

g[y].append(x) return g

g = [[] for i in range(6)] g = addEdge(g, 0, 1)

g = addEdge(g, 0, 3)

g = addEdge(g, 0, 5)

g = addEdge(g, 1, 2)

g = addEdge(g, 1, 4)

g = addEdge(g, 2, 3)

g = addEdge(g, 2, 5)

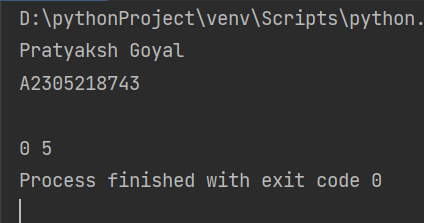
g = addEdge(g, 3, 4)

g = addEdge(g, 4, 5)

dist=[0,5,4,3,6,0]

bestFirstSearch(g,0,5,6,dist)

**OUTPUT**



**EXPERIMENT - 4(1)**

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

**THEORY**:

A\* Search:

A-star (also referred to as A\*) is one of the most successful search algorithms to find the shortest path between nodes or graphs. It is an informed search algorithm, as it uses information about path cost and also uses heuristics to find the solution.

Algorithm-

1: Firstly, Place the starting node into OPEN and find its f (n) value.

2: Then remove the node from OPEN, having the smallest f (n) value. If it is a goal node, then stop and return to success.

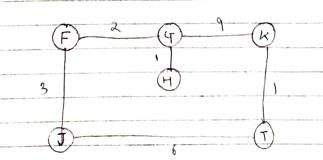
3: Else remove the node from OPEN, and find all its successors.

4: Find the f (n) value of all the successors, place them into OPEN, and place the removed node into CLOSE.

5: Goto Step-2.

6: Exit.

## GRAPH USED:



**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: g[m] = g[n] + weight parents[m] = n

if m in closed\_set: closed\_set.remove(m) open\_set.add(m)

if n == None:

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 From Source to Destination-->

{}'.format(path))

return path

open\_set.remove(n) closed\_set.add(n)

return None

def get\_neighbors(v):

if v in Graph\_nodes: return Graph\_nodes[v]

else:

return None

def heuristic(n): H\_dist = {

'F': 11,

'G': 6,

'H': 99,

'I': 1,

'J': 7,

'K': 0,

}

return H\_dist[n]

Graph\_nodes = {

'F': [('G', 2), ('J', 3)],

'G': [('H', 1), ('K', 9)],

'H': None,

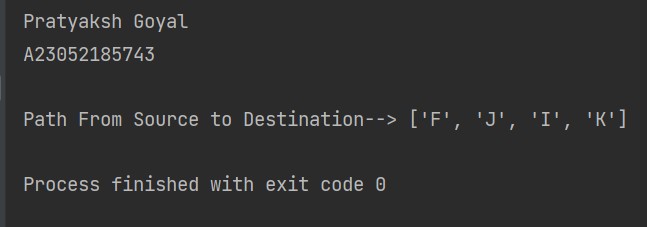
'J': [('I', 6)],

'I': [('K', 1)],

}

aStarAlgo('F', 'K')

## OUTPUT



**EXPERIMENT - 5**

**AIM:** Write a Program to implement Graph Colouring Problem Constraint Satisfaction Algorithm in python

**THEORY**:

Constraint Satisfaction Algorithm:

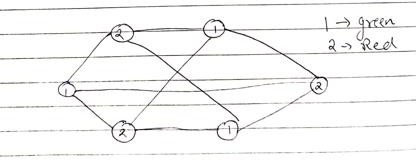
In artificial intelligence and operations research, constraint satisfaction is the process of finding a solution to a set of constraints that impose conditions that the variables must satisfy. A solution is therefore a set of values for the variables that satisfies all constraints—that is, a point in the feasible region.

Constraints used-

# Node 2 and 5 cannot have same color.

# Node 3 cannot be colored with color no. 0

**GRAPH USED**:



**CODE**

def printColor(g, V): used = [0] \* V ans = [-1] \* V ans[0] = 0;

for u in range(1, V): for i in g[u]:

if (ans[i] != -1): used[ans[i]] = 1

color = 0

while color < V:

if (used[color] == 0 and (u != 3 or ans[2] != color) and (u != 5 or color != 2)):

break color += 1

ans[u] = color for i in g[u]:

if (ans[i] != -1): used[ans[i]] = 0

maxColor = 0

for u in range(V):

print("Color of Vertex", u, " is", ans[u]) if maxColor < ans[u]:

maxColor = ans[u]

print('\nNo of colors used are ', maxColor + 1)

def addEdge(g, x, y): g[x].append(y)

g[y].append(x) return g

g = [[] for i in range(6)] g = addEdge(g, 0, 1)

g = addEdge(g, 0, 3)

g = addEdge(g, 0, 5)

g = addEdge(g, 1, 2)

g = addEdge(g, 1, 4)

g = addEdge(g, 2, 3)

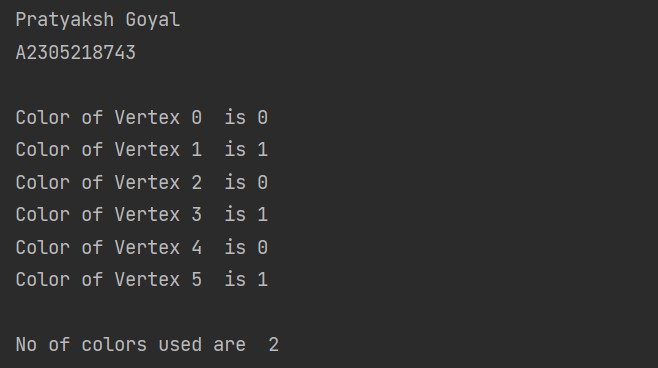
g = addEdge(g, 2, 5)

g = addEdge(g, 3, 4)

g = addEdge(g, 4, 5)

printColor(g, 6)

## OUTPUT



**EXPERIMENT - 6**

**AIM:** Write a program to implement Tic Tac Toe game problem. Use Min Max Algorithm.

**THEORY**:

Min-max Algorithm:

In artificial intelligence and operations research, Min-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally. It performs a depth-first search for the exploration of the complete game tree.

**CODE**

import numpy as np

from math import inf as infinity

# Set the Empty Board game\_state = [[' ', ' ', ' '],

[' ', ' ', ' '],

[' ', ' ', ' ']]

# Create the Two Players as 'X'/'O' players = ['X', 'O']

# Method for checking the correct move on Tic-Tac-Toe def play\_move(state, player, block\_num):

if state[int((block\_num - 1) / 3)][(block\_num - 1) % 3] is ' ': # Assign the player move on the current position of Tic-

Tac-Toe if condition is True

state[int((block\_num - 1) / 3)][(block\_num - 1) % 3] =

player

else:

block\_num = int(input("Block is not empty, ya blockhead!

Choose again: "))

play\_move(state, player, block\_num)

def copy\_game\_state(state):

new\_state = [[' ', ' ', ' '], [' ', ' ', ' '], [' ', ' ', ' ']]

for i in range(3): for j in range(3):

new\_state[i][j] = state[i][j] return new\_state

# Method to check the current state of the Tic-Tac-Toe def check\_current\_state(game\_state):

draw\_flag = 0

for i in range(3): for j in range(3):

if game\_state[i][j] is ' ': draw\_flag = 1

if draw\_flag is 0: return None, "Draw"

# Check horizontals

if (game\_state[0][0] == game\_state[0][1] == game\_state[0][2] != ' '):

return game\_state[0][0], "Done"

if (game\_state[1][0] == game\_state[1][1] == game\_state[1][2] != ' '):

return game\_state[1][0], "Done"

if (game\_state[2][0] == game\_state[2][1] == game\_state[2][2] != ' '):

return game\_state[2][0], "Done"

# Check verticals

if (game\_state[0][0] == game\_state[1][0] == game\_state[2][0] != ' '):

return game\_state[0][0], "Done"

if (game\_state[0][1] == game\_state[1][1] == game\_state[2][1] != ' '):

return game\_state[0][1], "Done"

if (game\_state[0][2] == game\_state[1][2] == game\_state[2][2] != ' '):

return game\_state[0][2], "Done"

# Check left diagonal

if (game\_state[0][0] == game\_state[1][1] == game\_state[2][2] != ' '):

return game\_state[1][1], "Done" # Check right diagonal

if (game\_state[0][2] == game\_state[1][1] == game\_state[2][0] != ' '):

return game\_state[1][1], "Done"

return None, "Not Done"

# Method to print the Tic-Tac-Toe Board def print\_board(game\_state):

print(' ')

print('| ' + str(game\_state[0][0]) + ' || ' + str(game\_state[0][1]) + ' || ' + str(game\_state[0][2]) + ' |')

print(' ')

print('| ' + str(game\_state[1][0]) + ' || ' + str(game\_state[1][1]) + ' || ' + str(game\_state[1][2]) + ' |')

print(' ')

print('| ' + str(game\_state[2][0]) + ' || ' + str(game\_state[2][1]) + ' || ' + str(game\_state[2][2]) + ' |')

print(' ')

# Method to implement the Minimax Algorithm def getBestMove(state, player):

winner\_loser, done = check\_current\_state(state)

# Check who won the game or if a draw occurred if done == "Done" and winner\_loser == 'O':

return 1

elif done == "Done" and winner\_loser == 'X': return -1

elif done == "Draw": return 0

moves = [] empty\_cells = [] for i in range(3):

for j in range(3):

if state[i][j] is ' ': empty\_cells.append(i \* 3 + (j + 1))

for empty\_cell in empty\_cells: move = {}

move['index'] = empty\_cell

new\_state = copy\_game\_state(state) play\_move(new\_state, player, empty\_cell)

# if player is computer if player == 'O':

# Make depth tree for human

result = getBestMove(new\_state, 'X') move['score'] = result

else:

# Make depth tree for computer result = getBestMove(new\_state, 'O') move['score'] = result

moves.append(move) best\_move = None

if None:

best = -infinity for move in moves:

if move['score'] > best: best = move['score']

best\_move = move['index']

else:

best = infinity for move in moves:

if move['score'] < best: best = move['score']

best\_move = move['index'] return best\_move

# Now PLaying the Tic-Tac-Toe Game play\_again = 'Y'

while play\_again == 'Y' or play\_again == 'y':

game\_state = [[' ', ' ', ' '],

[' ', ' ', ' '],

[' ', ' ', ' ']]

current\_state = "Not Done" print("\nNew Game!") print\_board(game\_state)

player\_choice = input("Choose which player goes first - X (You) or O(Computer): ")

winner = None

if player\_choice == 'X' or player\_choice == 'x': current\_player\_idx = 0

else:

current\_player\_idx = 1

while current\_state == "Not Done": # For Human Turn

if current\_player\_idx == 0:

block\_choice = int(input("Your turn please! Choose where to place (1 to 9): "))

play\_move(game\_state, players[current\_player\_idx], block\_choice)

# Computer Turn else:

block\_choice = getBestMove(game\_state, players[current\_player\_idx])

play\_move(game\_state, players[current\_player\_idx], block\_choice)

print("AI plays move: " + str(block\_choice)) print\_board(game\_state)

winner, current\_state = check\_current\_state(game\_state) if winner is not None:

print(str(winner) + " won!") else:

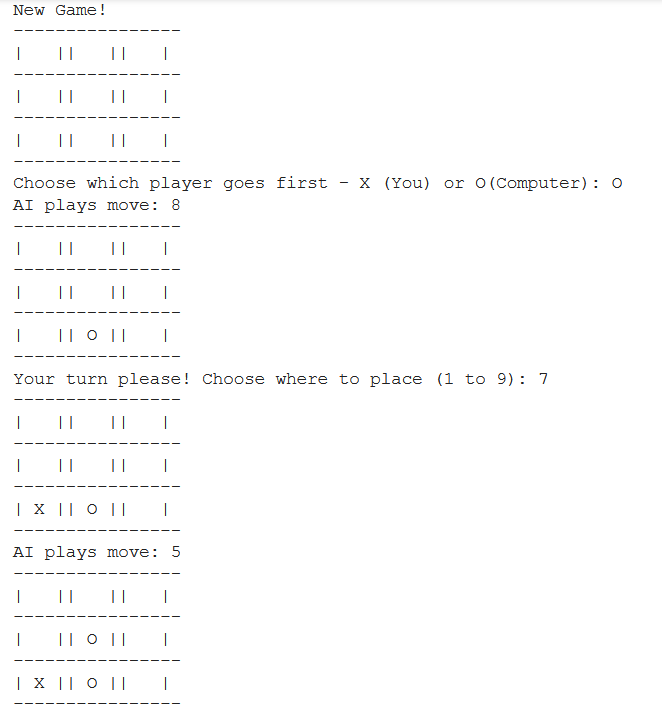
current\_player\_idx = (current\_player\_idx + 1) % 2

if current\_state is "Draw": print("Draw!")

play\_again = input('Wanna try again?(Y/N) : ') if play\_again == 'N':

print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')

## OUTPUT



**EXPERIMENT – 7**

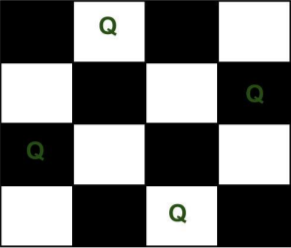
**AIM:** Write a program to implement N-Queens problem in Python for 4 Queens (on a 4 × 4 board)

**THEORY**:

N-Queens Problem:

N-Queens problem is a computer problem of placing *n* non-attacking queens on an *n* × *n* chessboard. In chess, a queen attacks any piece which is either vertically or horizontally in line with the queen, or on one of its diagonals.

This problem has a solution for all natural numbers *n* except for *n = 2* and *n = 3*.



**CODE:**

global N N = 4

def printSolution(board): for i in range(N):

for j in range(N): print(board[i][j], end=" ")

print()

def isSafe(board, row, col):

# Check this row on left side for i in range(col):

if board[row][i] == 1: return False

# Check upper diagonal on left side for i, j in zip(range(row, -1, -1),

range(col, -1, -1)): if board[i][j] == 1:

return False

# Check lower diagonal on left side for i, j in zip(range(row, N, 1),

range(col, -1, -1)): if board[i][j] == 1:

return False return True

def solveNQUtil(board, col):

# base case: If all queens are placed # then return true

if col >= N:

return True

# Consider this column and try placing # this queen in all rows one by one for i in range(N):

if isSafe(board, i, col):

# Place this queen in board[i][col] board[i][col] = 1

# recur to place rest of the queens

if solveNQUtil(board, col + 1) == True: return True

# If placing queen in board[i][col # doesn't lead to a solution, then # queen from board[i][col] board[i][col] = 0

# if the queen can not be placed in any row in # this colum col then return false

return False

# solveNQUtil() to returns false if queens

# cannot be placed, otherwise return true and # placement of queens in the form of 1s.

def solveNQ():

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

[0, 0, 0, 0],

[0, 0, 0, 0],

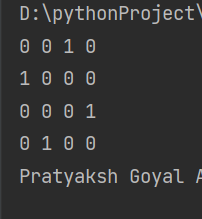
[0, 0, 0, 0]]

if solveNQUtil(board, 0) == False: print("Solution does not exist") return False

printSolution(board) return True

# Driver Code solveNQ()

## OUTPUT



**EXPERIMENT – 8**

**AIM:** Write a program to implement the Single-player game of Snakes and Ladders using Python

**CODE:**

import random

class snakesandladder(object):

def init (self, name, position): self.name = name

self.position = position

def dice(self): chances = 0

print(" Let the Games Begin

\n")

while self.position <= 100:

roll = random.choice([1, 2, 3, 4, 5, 6]) print('Dice rolls out: ', roll) self.position = roll + self.position

if self.position > 100:

self.position = self.position - roll if self.position == 100:

print('completed the game') break

print('Current position of ' + self.name + ' : ',

self.position, '\n')

chances += 1

print('Number of turns taken : ', chances)

Play = snakesandladder('XYZ', 0) Play.dice()

## OUTPUT

Text

Description automatically generated

**EXPERIMENT – 9**

**AIM:** Write a program to implement word tokenization and sentence tokenization of text on Python using the NLTK package

**THEORY**:

Tokenization:

Tokenization refers to the process of breaking unstructured text in natural language into chunks of information that can be considered as discrete elements (or tokens). The token occurrences in a document can be used directly as a vector representing that document.

Tokenization is the first step in any NLP pipeline. It has an important effect on the rest of your pipeline. By turning an unstructured string (text document) into a numerical data structure, it becomes usable for machine learning applications. They might be used in a machine learning pipeline as features that trigger complex decisions or behavior.

Tokenization can be done on the basis of words, sentences, punctuations, or any other meaningful delimiter.

NLTK:

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

**IMPLEMENTATION:**

Set-up & Installation:

To install NLTK run the following command on CMD Prompt. pip install nltk

Then download the requisite package from NLTK by running the following code on a cell in Jupyter Notebook.

import nltk nltk.download(‘punkt’)

1. **Word Tokenization:**

import nltk

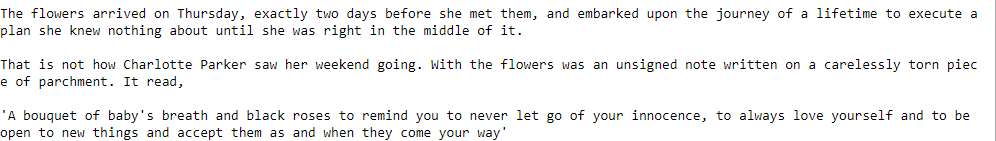
from nltk.tokenize import word\_tokenize

text = "The flowers arrived on Thursday, exactly two days before she met them, and embarked upon the journey of a lifetime to execute a plan she knew nothing about until she was right in the middle of it. \n\nThat is not how Charlotte Parker saw her weekend going. With the flowers was an unsigned note written on a carelessly torn piece of parchment. It read, \n\n'A bouquet of baby's breath and black roses to remind you to never let go of your innocence, to always love yourself and to be open to new things and accept them as and when they come your way'"

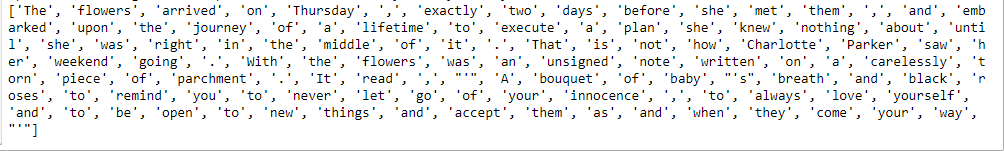
print(text) print(word\_tokenize(text))

**OUTPUT:**

Original Text:



Tokenized Text:



1. Sentence Tokenization:

import nltk

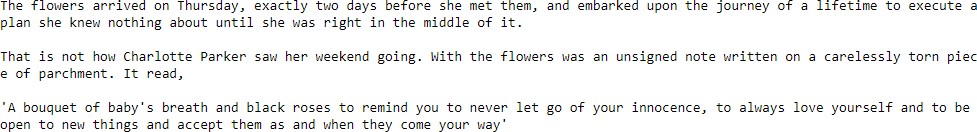
from nltk.tokenize import sent\_tokenize

text = "The flowers arrived on Thursday, exactly two days before she met them, and embarked upon the journey of a lifetime to execute a plan she knew nothing about until she was right in the middle of it. \n\nThat is not how Charlotte Parker saw her weekend going. With the flowers was an unsigned note written on a carelessly torn piece of parchment. It read, \n\n'A bouquet of baby's breath and black roses to remind you to never let go of your innocence, to always love yourself and to be open to new things and accept them as and when they come your way'"

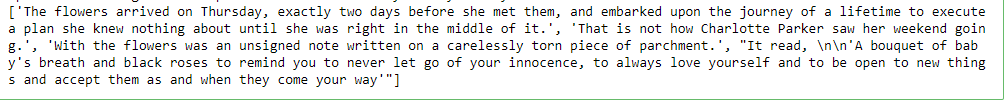
print(text) print(sent\_tokenize(text))

**OUTPUT:**

Original Text:



Tokenized Text:



**EXPERIMENT – 10**

**AIM:** Write a program to implement the Knapsack problem in Python

**THEORY**:

Knapsack Problem:

The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

**CODE:**

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

if (n == 0) or (W == 0):

return 0

if (wt[n] > W):

return knapSack(W, wt, val, n - 1)

else:

return max(

val[n] + knapSack(W - wt[n], wt, val, n - 1), knapSack(W, wt, val, n - 1)

)

n = int(input("Enter the number of items available: ")) val = []

wt = []

print("\nEnter value and weight of each item separately:-") for i in range(n):

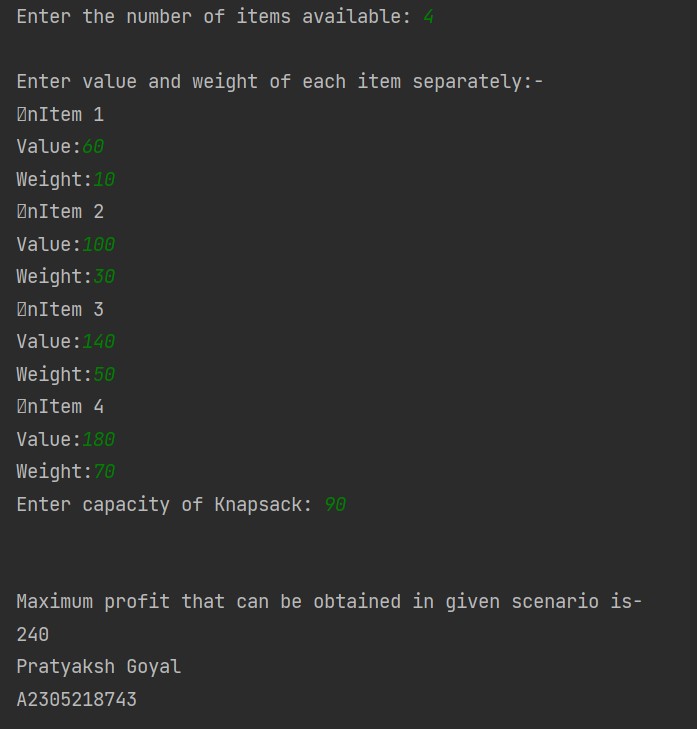
print("\anItem " + str(i + 1)) val.append(int(input("Value:")))

wt.append(int(input("Weight:")))

W = int(input("Enter capacity of Knapsack: ")) print("\n\nMaximum profit that can be obtained in given scenario is-")

print(knapSack(W, wt, val, n - 1))

**OUTPUT:**



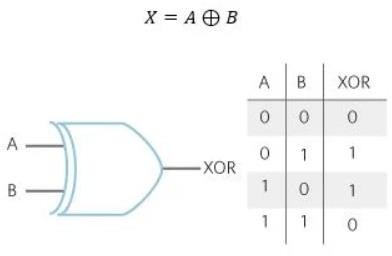
**EXPERIMENT – 11**

**AIM:** Write a program to design an XOR truth table using python.

**TOOL USED:** Jupyter Notebook

## THEORY:

XOR” an abbreviation for “Exclusively-OR.” The simplest XOR gate is a two- input digital circuit that outputs a logical “1” if the two input values differ, i.e., its output is a logical “1” if either of its inputs are 1, but not at the same time (exclusively). The symbol and truth table for an XOR is shown in Figure 1



**CODE:**

def XOR(a, b): if a != b:

return 1 else:

return 0

if name == ' main ':

print(" | XOR Truth Table | Result |")

print(" A = False, B = False | A XOR B =", XOR(False, False), " | ")

print(" A = False, B = True | A XOR B =", XOR(False, True), "

| ")

print(" A = True, B = False | A XOR B =", XOR(True, False), "

**OUTPUT:**

