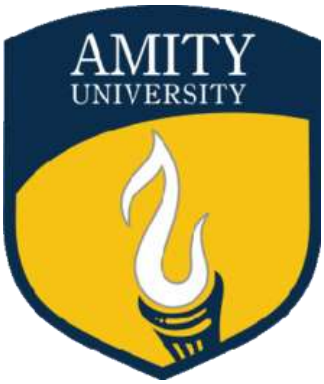
ARTIFICIAL INTELLIGENCE LAB
FILE
Submitted to

Amity University Uttar Pradesh



COURSE CODE- CSE 401
(B. TECH)

(Information Technology)

By

SHOBHIT HARIT
A2305319028
Under the guidance of

DR. DEEPTI MEHROTRA

DEPARTMENT OF INFORMATION AND TECHNOLOGY ENGINEERING
AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY AMITY UNIVERSITY UTTAR PRADESH

INDEX

SNO	PRACTICAL	DATE OF EXPERIMENT	DATE OF SUBMISSION	TEACHER'S SIGN
1	To implement graph in python.	19/01/2022	20/04/2022	
2	To implement BFS search algorithm	02/02/2022	20/04/2022	
3	To implement DFS search algorithm	02/02/2022	20/04/2022	
4	To implement Best First Search algorithm using priority queue	09/02/2022	20/04/2022	
5	To implement Water Jug Problem	16/02/2022	20/04/2022	
6	To implement A* algorithm for a given graph	02/03/2022	20/04/2022	
7	To implement Knapsack Problem using Hill Climbing Search	09/03/2022	20/04/2022	
8	To implement Tic-Tac-Toe Game	15/03/2022	20/04/2022	
9	To implement Natural Language Processing using Natural Language Toolkit (nltk)	30/03/2022	20/04/2022	
10	To implement Constraint Satisfaction Problem (Graph colouring)	06/04/2022	20/04/2022	

EXPERIMENT – 1

Date – 19/01/2022

Aim: To implement a graph

Language Used: Python

Code:

```
n=int(input("enter the number of nodes:"))
mat=[]
for i in range (n):
    row1 = []
    for j in range(n):
        element = int(input("Press 1 if connected or Press 0 if not connected"))
        row1.append(element)
    mat.append(row1)
print(mat)
checka = 0
checkb = 0
checkc = 0
checkd = 0
checke = 0
checkf = 0
checkg = 0
for i in range(n):
    for j in range(n):
        if(mat[i][j]==1):
            if(i==0):
                node1='a'
                checka=checka+1
            elif(i==1):
                node1='b'
                checkb=checkb+1
```

```
elif(i==2):
    node1='c'
    checkc=checkc+1
elif(i==3):
    node1='d'
    checkd=checkd+1
elif(i==4):
    node1='e'
    checke=checke+1
elif(i==5):
    node1='f'
    checkf=checkf+1
elif(i==6):
    node1='g'
    checkg=checkg+1
if(j==0):
    node2='a'
elif(j==1):
    node2='b'
elif(j==2):
    node2='c'
elif(j==3):
    node2='d'
elif(j==4):
    node2='e'
elif(j==5):
    node2='f'
elif(j==6):
    node2='g'
print(node1 + "->" + node2)
```

```
print("number of edges from a =" + str(checka))
print("number of edges from b =" + str(checkb))
print("number of edges from c =" + str(checkc))
print("number of edges from d =" + str(checkd))
print("number of edges from e =" + str(checke))
print("number of edges from f =" + str(checkf))
print("number of edges from g =" + str(checkg))
```

Output:

```
[[0, 1, 1, 0, 0, 1, 0], [1, 0, 1, 0, 1, 0, 1], [1, 1, 0, 1, 0, 0, 0], [0, 0, 1, 0, 1, 1, 0], [0, 1, 0, 1, 0, 1, 1], [1, 0, 0,
1, 1, 0, 0], [0, 1, 0, 0, 1, 0, 0]]
a->b
a->c
a->f
b->a
b->c
b->e
b->g
c->a
c->b
c->d
d->c
d->e
d->f
e->b
e->d
e->f
e->g
f->a
f->d
f->e
g->b
g->e
number of edges from a =3
number of edges from b =4
number of edges from c =3
number of edges from d =3
number of edges from e =4
number of edges from f =3
number of edges from g =2
```

EXPERIMENT – 2

Date – 02/02/2022

Aim: To implement BFS search algorithm.

Language Used: Python

Code:

```
graph = {
  'A' : ['O','S','C'],
  'B' : ['F', 'P'],
  'O' : ['A','S'],
  'S' : ['A','O','F','R'],
  'C' : ['A','R','P'],
  'R' : ['S','C','P'],
  'P' : ['C','R','B'],
  'F' : ['S','B']
}

vlist = []
qlist = []

def bfs(vlist, graph, node):
    vlist.append(node)
    qlist.append(node)

    while qlist:
        s = qlist.pop(0)
        print (s, end = " ")

        for neighbour in graph[s]:
            if neighbour not in vlist:
                vlist.append(neighbour)
```

qlist.append(neighbour)

bfs(vlist, graph, 'A')

Output:

```
                vlist.append(neighbour)
                qlist.append(neighbour)


bfs(vlist, graph, 'A')
```

```
A O S C F R P B
```

EXPERIMENT – 3

Date – 02/02/2022

Aim: To implement DFS search algorithm

Language Used: Python

Code:

```
graph = {
  'A' : ['O','S','C'],
  'B' : ['F', 'P'],
  'O' : ['A','S'],
  'S' : ['A','O','F','R'],
  'C' : ['A','R','P'],
  'R' : ['S','C','P'],
  'P' : ['C','R','B'],
  'F' : ['S','B']
}
vset = set()
def dfs(vset, graph, node):
    if node not in vset:
        print (node)
        vset.add(node)
        if node=='B':
            print ("Reached the goal State")
        for neighbour in graph[node]:
                dfs(vset, graph, neighbour)
dfs(vset, graph, 'A')
```

Output:

```
A
O
S
F
B
Reached the goal State
P
C
R
```

EXPERIMENT – 4

Date – 09/02/2022

Aim: To implement Best First Search algorithm using priority queue

Language Used: Python

Code:

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
	visited = [False] * n
	pq = PriorityQueue()
	pq.put((0, source))
	while pq.empty() == False:
		u = pq.get()[1]
		print(u, end=" ")
		if u == target:
			break
		for v, c in graph[u]:
			if visited[v] == False:
				visited[v] = True
				pq.put((c, v))
	print()
def addedge(x, y, cost):
	graph[x].append((y, cost))
	graph[y].append((x, cost))
def prGraph(adj, V):
	for v in range(V):
		print("vertex " + str(v), end = ' ')
		for x in adj[v]:
			print("-> " + str(x), end = '')
```

```
                print()
        print()
addedge(0, 1, 3)
addedge(0, 2, 5)
addedge(0, 3, 6)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 13
best_first_search(source, target, v)
prGraph(graph,v)
```

Output:

```
0 1 0 2 3 8 9 11 13
vertex 0 -> (1, 3)-> (2, 5)-> (3, 6)
vertex 1 -> (0, 3)-> (4, 9)-> (5, 8)
vertex 2 -> (0, 5)-> (6, 12)-> (7, 14)
vertex 3 -> (0, 6)-> (8, 7)
vertex 4 -> (1, 9)
vertex 5 -> (1, 8)
vertex 6 -> (2, 12)
vertex 7 -> (2, 14)
vertex 8 -> (3, 7)-> (9, 5)-> (10, 6)
vertex 9 -> (8, 5)-> (11, 1)-> (12, 10)-> (13, 2)
vertex 10 -> (8, 6)
vertex 11 -> (9, 1)
vertex 12 -> (9, 10)
vertex 13 -> (9, 2)
```

EXPERIMENT – 5

Date – 16/02/2022

Aim: To implement Water Jug Problem

Language Used: Python

Code:

```
from collections import deque
def BFS(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]) # Fill Jug2
                q.append([a, u[1]]) # Fill Jug1
                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 ("No solution")
if __name__ == '__main__':
        Jug1, Jug2, target = 4, 3, 2
        print("Path from initial state "
                "to solution state ::")
        BFS(Jug1, Jug2, target)
```

Output:

```
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 )
```

EXPERIMENT – 6

Date – 02/03/2022

Aim: To implement A* algorithm for a given diagram

Language Used: Python

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 = {
            'S': 20,
            'A': 10,
            'B': 8,
            'C': 7,
            'D': 6,
            'E': 4,
            'F': 2,
            'G': 0
        }
        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 == None or poo[v] + self.h(v) < poo[n] + self.h(n):
            n = v;
    if n == 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
adjac_lis = {
    'S': [('A', 2), ('B',6 ), ('C', 3)],
    'A': [('B', 2), ('G',10 )],
    'B': [('D', 4)],
    'C': [('D', 8), ('E',9 )],
    'D': [('F', 1)],
    'E': [('G', 3)],
    'F': [('G', 2)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('S','G')
```

Output:

```
Path found: ['S', 'A', 'G']
Out[6]: ['S', 'A', 'G']
```

EXPERIMENT – 7

Date – 09/03/2022

Aim: To implement Knapsack problem using Hill Climbing search

Language Used: Python

Code:

```
w = [40
,54,10,40,12]
p = [35,50,8,20,3]
M=100
x=[]
Knapsack = []
Tprofit = 0
perprofit = 0
for i in range(0,5):
    x.append([p[i]/w[i],i])
x.sort(reverse=True)
print("list in decending order of profit/weight ratio:")
print(x)
for i in x:
    for j in i:
        if type(j) == float:
            continue
        else:
            if M>0:
                if M>w[j]:
                    Knapsack.append(j)
                    Tprofit=Tprofit+p[j]
                    M=M-w[j]
                    print(j)
                else:
```

```
            Knapsack.append([j,M])
            perprofit=(p[j]/w[j])*M
            Tprofit=Tprofit+perprofit
            M=0
        else:
            continue
print("total profit:", Tprofit)
print("item number and its weight that are included in the Knapsack:", Knapsack)
```

Output:

```
list in decending order of profit/weight ratio:
[[0.9901960784313726, 0], [0.9259259259259259, 1], [0.8, 2], [0.5, 3], [0.25, 4]]
total profit: 99.01960784313727
item number and its weight that are included in the Knapsack: [[0, 100]]
```

EXPERIMENT – 8

Date – 15/03/2022

Aim: To implement Tic-Tac-Toe Game

Language Used: Python

Code:

```
def Validate(list):
    s=0
    if list[0]=="" or list[1]=="" or list[2]=="" or list[3]=="" or list[4]=="" or list[5]=="" or list[6]=="" or list[7]=="" or list[8]=="":
        if list[0]==list[1]==list[2]=='X' or list[3]==list[4]==list[5]=='X' or list[6]==list[7]==list[8]=='X' or list[0]==list[3]==list[6]=='X' or list[1]==list[4]==list[7]=='X' or list[2]==list[5]==list[8]=='X' or list[0]==list[4]==list[8]=='X' or list[2]==list[4]==list[6]=='X':
            s=1
            return s
        elif list[0]==list[1]==list[2]=='O' or list[3]==list[4]==list[5]=='O' or list[6]==list[7]==list[8]=='O' or list[0]==list[3]==list[6]=='O' or list[1]==list[4]==list[7]=='O' or list[2]==list[5]==list[8]=='O' or list[0]==list[4]==list[8]=='O' or list[2]==list[4]==list[6]=='O':
            s=2
            return s
        else:
            s=3
            return s
    else:
        if list[0]==list[1]==list[2]=='X' or list[3]==list[4]==list[5]=='X' or list[6]==list[7]==list[8]=='X' or list[0]==list[3]==list[6]=='X' or list[1]==list[4]==list[7]=='X' or list[2]==list[5]==list[8]=='X' or list[0]==list[4]==list[8]=='X' or list[2]==list[4]==list[6]=='X':
            s=4
            return s
        elif list[0]==list[1]==list[2]=='O' or list[3]==list[4]==list[5]=='O' or list[6]==list[7]==list[8]=='O' or list[0]==list[3]==list[6]=='O' or list[1]==list[4]==list[7]=='O'
```

```
or list[2]==list[5]==list[8]=='O' or list[0]==list[4]==list[8]=='O' or
list[2]==list[4]==list[6]=='O':
            s=5
            return s
        else:
            s=6
            return s
import numpy as np
list = [" "," "," "," "," "," "," "," "," "]
for i in range(9):
    if i<3:
        j=0
        list[i]= input("enter X, O or space for postions ({},{}): ".format(j,i))
    elif i>=3 and i<6:
        j=1
        list[i]= input("enter X, O or space for postions ({},{}): ".format(j,i-3))
    elif i>=6 and i<9:
        j=2
        list[i]= input("enter X, O or space for postions ({},{}): ".format(j,i-6))
situation=Validate(list)
ttt= np.array(list).reshape(3,3)
if situation==1 or situation==4:
    print("Player X won! ")
elif situation==2 or situation==5:
    print("Player O won! ")
elif situation==3:
    print("Match is not complete ")
elif situation==6:
    print("Match Draw ")
print(ttt)
```

Output:

```
enter X, O or space for postions (0,0): X
enter X, O or space for postions (0,1): X
enter X, O or space for postions (0,2): X
enter X, O or space for postions (1,0): O
enter X, O or space for postions (1,1): O
enter X, O or space for postions (1,2): X
enter X, O or space for postions (2,0): O
enter X, O or space for postions (2,1): X
enter X, O or space for postions (2,2): O
Player X won!
[['X' 'X' 'X']
 ['O' 'O' 'X']
 ['O' 'X' 'O']]
```

EXPERIMENT – 9

Date – 30/03/2022

Aim: To implement natural language processing using natural language toolkit (nltk)

Language Used: Python

Code:

Importing the nltk toolkit

```
import nltk
nltk.download("book")
```

Tokenizer

```
sentence="educator is educating the students in educational institute educator"
tokens = nltk.word_tokenize(sentence)
print(tokens)
tagged = nltk.pos_tag(tokens)
print(tagged)
```

Stemming

```
from nltk.stem import PorterStemmer
from nltk.tokenize import word_tokenize
ps = PorterStemmer()
for t in tokens:
    print(t, " : ", ps.stem(t))
```

Lemmatizer

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
for t in tokens:
    print(t, " : ",lemmatizer.lemmatize(t))
```

Word Count

```
wordCount=0
for t in tokens:
  for t1 in tokens:
    if t==t1:
      wordCount=wordCount+1

  print("the word " + t + " has a count = {}".format(wordCount))
  wordCount=0
```

Output:

i. Tokenizer

```
['Educator', 'is', 'educating', 'the', 'students', 'in', 'educational', 'institute', 'Educator']
[('Educator', 'NN'), ('is', 'VBZ'), ('educating', 'VBG'), ('the', 'DT'), ('students', 'NNS'), ('in', 'IN'), ('educational', 'J
J'), ('institute', 'NN'), ('Educator', 'NN')]
```

ii. Stemmer

```
Educator  :  educ
is  :  is
educating  :  educ
the  :  the
students  :  student
in  :  in
educational  :  educ
institute  :  institut
Educator  :  educ
```

iii. Lammatizer

```
Educator  :  Educator
is  :  is
educating  :  educating
the  :  the
students  :  student
in  :  in
educational  :  educational
institute  :  institute
Educator  :  Educator
```

iv. Word Count

```
the word Educator has a count = 2
the word is has a count = 1
the word educating has a count = 1
the word the has a count = 1
the word students has a count = 1
the word in has a count = 1
the word educational has a count = 1
the word institute has a count = 1
the word Educator has a count = 2
```

EXPERIMENT – 10

Date – 06/04/2022

Aim: To implement Constraint Satisfaction Problem (Graph colouring)

Language Used: Python

Code:

```
class Graph:
    def __init__(self, edges, n):
        self.adjList = [[] for _ in range(n)]
        for (src, dest) in edges:
            self.adjList[src].append(dest)
            self.adjList[dest].append(src)


def colorGraph(graph, n):
    result = {}
    for u in range(n):
        assigned = set([result.get(i) for i in graph.adjList[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(f'Color assigned to vertex {v} is {colors[result[v]]}')




if __name__ == '__main__':
    colors = ['', 'BLUE', 'GREEN', 'RED']
    edges = [(0, 1), (0, 2), (0, 4), (0, 5), (1, 3), (2, 3), (2, 6), (3, 4),(4,5)]
```

```
n = 7
graph = Graph(edges, n)
colorGraph(graph, n)
```

Output:

```
Color assigned to vertex 0 is BLUE
Color assigned to vertex 1 is GREEN
Color assigned to vertex 2 is GREEN
Color assigned to vertex 3 is BLUE
Color assigned to vertex 4 is GREEN
Color assigned to vertex 5 is RED
Color assigned to vertex 6 is BLUE
```