ASS 1

Aim = Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

Breadth-first search (BFS) is an algorithm used for tree traversal on graphs or tree data structures.

BFS can be easily implemented using recursion and data structures like dictionaries and lists.

BFS Algorithm Applications

1. To build index by search index
2. For GPS navigation
3. Path finding algorithms
4. In Ford-Fulkerson algorithm to find maximum flow in a network
5. Cycle detection in an undirected graph

Code:

```
import itertools

def bfsalgo(childtree, openlist, closelist,goal):
    '''Implementation of BFS algo'''
    X=openlist[0]
    print("\n\n X = {}".format(X))
    closelist.append(X)
    for i in range(len(childtree)):

        if(X==childtree[i][0]):
            openlist.append(childtree[i][1:])

    openlist=list(itertools.chain(*openlist))
    del openlist[0]

    if(X!=goal):
        print("\n OPEN {}   CLOSE {}".format(openlist,closelist))

    if(X==goal):
        print('\n SUCCESS')
    elif(len(openlist)>0):
        bfsalgo(childtree, openlist, closelist,goal)
    else:
        print('\n\n FAILURE')
```

```
def createTree(treearr, treelength):
    '''Creating a child's child tree'''
    try:
        for i in range(treelength):
            childtree[i].append(treearr[i+1])
            checkchild=input("\n Does "+treearr[i+1]+" has any child 
node Press n for no :  ")
            if(checkchild=='n'):
                print()
            else:
                checkchildsibling=""
                while(checkchildsibling!='n'):
                    childname=input("\n Enter child node 
:  ")
                    childtree[i].append(childname)

                    checkchildsibling = input("\n Does "+treearr[i+1]+" 
has any other children Press n for no : ")
    except IndexError:
        pass

treearr=[]
root=input("Enter the root node :   ")

treearr.append(root)

checkchild=input("\n Does "+root+" has any child node Press n for no 
:  ")
if(checkchild=='n'):
    print(treearr)

else:
    checkchildsibling=""
    while(checkchildsibling!='n'):

        childname=input("\n Enter child node 
:  ")
        treearr.append(childname)

        checkchildsibling = input("\nDoes "+root+" has any other child 
Press n for no : ")

treelength=len(treearr)
childtree=[[] for x in range(treelength-1)]
createTree(treearr,treelength)
```

```
print("\n\n Tree successfully created root node wtih children\n")
print(treearr)

print("\n\n Children with their children and siblings \n")
print(childtree)

goal=input("\n Enter the goal node :     ")

openlist=[]
closelist=[]


X=treearr[0]
print("\n\n X = "+X)

openlist.append(treearr[1:])
openlist=list(itertools.chain(*openlist))
closelist.append(X)
print("\nOPEN {}    CLOSE {}".format(openlist,closelist))


bfsalgo(childtree,openlist,closelist,goal)
```

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Application of DFS Algorithm

1. For finding the path

2. To test if the graph is bipartite

3. For finding the strongly connected components of a graph

4. For detecting cycles in a graph

Code:

```
import itertools

def dfsalgo(childtree, openlist, closelist,goal):
    '''Implementation of DFS algo'''
    X=openlist[0]
    print("\n\n X = {}".format(X))
    closelist.append(X)
    for i in range(len(childtree)):
```

```
        if(X==childtree[i][0]):
            openlist.insert(0,childtree[i][1:])

    openlist=list(itertools.chain(*openlist))

    openlist.remove(X)

    if(X!=goal):
        print("\n OPEN {}   CLOSE {}".format(openlist,closelist))

    if(X==goal):
        print('\n SUCCESS')
    elif(len(openlist)>0):
        dfsalgo(childtree, openlist, closelist,goal)
    else:
        print('\n\n FAILURE')
def createTree(treearr, treelength):
    '''Creating a child's child tree'''
    try:
        for i in range(treelength):
            childtree[i].append(treearr[i+1])
            checkchild=input("\n Does "+treearr[i+1]+" has any child
node Press n for no :  ")
            if(checkchild=='n'):
                print()
            else:
                checkchildsibling=""
                while(checkchildsibling!='n'):
                    childname=input("\n Enter child node
:  ")
                    childtree[i].append(childname)

                    checkchildsibling = input("\n Does "+treearr[i+1]+"
has any other children Press n for no : ")
    except IndexError:
        pass

treearr=[]
root=input("Enter the root node :   ")

treearr.append(root)

checkchild=input("\n Does "+root+" has any child node Press n for no
:  ")
if(checkchild=='n'):
```

```
    print(treearr)
else:
    checkchildsibling=""
    while(checkchildsibling!='n'):
        childname=input("\n Enter child node :  ")
        treearr.append(childname)
        checkchildsibling = input("\nDoes "+root+" has any other child
Press n for no : ")
treelength=len(treearr)
childtree=[[] for x in range(treelength-1)]
createTree(treearr,treelength)
print("\n\n Tree successfully created root node wtih children\n")
print(treearr)
print("\n\n Children with their children and siblings \n")
print(childtree)
goal=input("\n Enter the goal node :    ")
openlist=[]
closelist=[]
X=treearr[0]
print("\n\n X = "+X)
openlist.append(treearr[1:])
openlist=list(itertools.chain(*openlist))
closelist.append(X)
print("\nOPEN {}    CLOSE {}".format(openlist,closelist))
dfsalgo(childtree,openlist,closelist,goal)
```

ASS 2

Aim = Implement A star Algorithm for any game search problem.

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. It is one which is the process of finding a path between multiple points, called "nodes". It enjoys widespread use due to its performance and accuracy. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, although other work has found A* to be superior to other approaches.

It is best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

$f(n) = g(n) + h(n)$, where

- $g(n)$ the cost (so far) to reach the node

• h(n) estimated cost to get from the node to the goal

• f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

ASS 3

Aim = Implement Greedy Search Algorithm for any of the following application: Prim's Minimal Spanning Tree Algorithm

A greedy algorithm is an algorithmic strategy that makes the best optimal choice at each small stage with the goal of this eventually leading to a globally optimum solution. This means that the algorithm picks the best solution at the moment without regard for consequences.

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which

• form a tree that includes every vertex

• has the minimum sum of weights among all the trees that can be formed from the graph

Prim's Algorithm Application

• Laying cables of electrical wiring

• In network designed

• To make protocols in network cycles

Code :

```
import sys


class Graph():

    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def printMST(self, parent):
        print("Edge\tWeight")
        for i in range(1, self.V):
            print(parent[i], "-", i, "\t", self.graph[i][parent[i]])

    def minKey(self, key, mstSet):
        min = sys.maxsize

        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
```

```
                min = key[v]
                min_index = v
        return min_index

    def primMST(self):
        key = [sys.maxsize]*self.V
        parent = [None]*self.V
        key[0] = 0
        mstSet = [False]*self.V

        parent[0] = -1

        for cout in range(self.V):
            u = self.minKey(key, mstSet)
            mstSet[u] = True
            for v in range(self.V):
                if self.graph[u][v] > 0 and mstSet[v] == False and key[v] >
self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)


g = Graph(5)
g.graph = []
n = int(input("Enter number of elemnts: "))
for i in range(0, n):
    ele = [int(input()), int(input()), int(
        input()), int(input()), int(input())]
    g.graph.append(ele)
print(g.graph)
g.primMST()
```

ASS 4

Aim= Implement the solution for a Constraint Satisfaction Problem using Branch and Bound and

Backtracking for n-queens problem or a graph coloring problem.

The N queens puzzle is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

In backtracking solution we backtrack when we hit a dead end. In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end.

Code for backtracking :

```
from typing import List

boardcount=0

def isboardok(chessboard:List,row:int,col:int):
    for c in range(col):
        if(chessboard[row][c]=='Q'):
            return False

    for r,c in zip(range(row-1,-1,-1), range(col-1,-1,-1)):
        if (chessboard[r][c]=='Q'):
            return False

    for r,c in zip(range(row+1,len(chessboard),1), range(col-1,-1,-1)):
        if(chessboard[r][c]=='Q'):
            return False
    return True

def displayboard(chessbaord:List):
    for row in chessbaord:
        print(row)
    print()

def placequeens(chessboard:List,col:int):
    global boardcount
    if(col>=len(chessboard)):
        boardcount+=1
        print("Board"+str(boardcount))
        print("=============================")
        displayboard(chessboard)
        print("==========================\n\n")
    else:
        for row in range(len(chessboard)):
            chessboard[row][col]='Q'
            if(isboardok(chessboard,row,col)==True):
                placequeens(chessboard,col+1)
            chessboard[row][col]='.'

chessboard=[]
N = int(input("Enter chessboard size:"))
for i in range(N):
    row=["."]*N
    chessboard.append(row)

placequeens(chessboard,0)
```

ASS 5

Aim = Develop an elementary chatbot for any suitable customer interaction application

Code:

```
pip install chatterbot
pip install chatterbot_corpus
import spacy
from spacy.cli.download import download
download(model="en_core_web_sm")
from chatterbot import ChatBot
from chatterbot.trainers import ListTrainer
chatbot = ChatBot('name')
trainer = ListTrainer(chatbot)
trainer.train([
        "Hi, can I help you?",
        "Sure, I'd like to book a flight to Iceland.",
        "Your Flight has been booked."
])
response = chatbot.get_response('I would like to book a flight.')
print(response)
while True:
        query=input()
        if query == 'exit':
                break
        ans =chatbot.get_response(query)
        print("Bot:",ans)
```

Cloud Computing

ASS 1

Aim = Amazon EC2

ASS 2

Aim = Installation and Configuration Google App Engine

Google App Engine (often referred to as GAE or simply App Engine) is a cloud computing platform as a service for developing and hosting web applications in Google-managed data centers. Applications are sandboxed and run across multiple servers. App Engine offers automatic scaling for web applications-as the number of requests increases for an application, App Engine automatically allocates more resources for the web application to handle the additional demand.

ASS 3

Aim = Create an Application in salesforce.com using Apex programming language.

Apex Classes

```
public class MyHelloWorld{

public static void applyDiscount(Book__c[] books)

{

        for(Books__c b:books)

                {b.Price__c*=0.9;}

}

}
```

Trigger

```
Trigger HelloWorldTrigger on Book__c(before insert)

{

Book__c[] books=Trigger.new;

MyHelloWorld.applyDiscount(books);

}
```

ASS 4

Aim = Design and develop custom Application (Mini Project) using Salesforce Cloud.