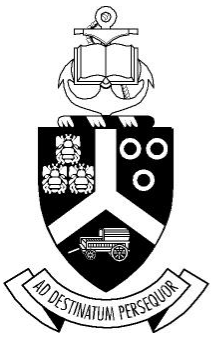

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

EAI 320 - Practical 1 Report

Author:
Mohamed Ameen Omar

Student number:
u16055323

February 15, 2018

Contents

List of Figures

List of Tables

1 Introduction

During the first practical for EAI 320 students were tasked to find a solution to the Travelling Salesman problem. The Travelling Salesman problem is a classic problem in computer science wherein one is given a set of locations or nodes and distances or weights between each pair. One should find the shortest possible route that visits every location exactly once and returns to the original location.

The applications of the Travelling Salesman problem are extremely vast and wide, including, astronomy, navigation, planning and DNA sequencing. Thus, investigating and finding the most effective an optimal search algorithm that solves the problem, will be beneficial to a wide range of disciplines not just Computer Science.

A class of searching algorithms called Brute Force algorithms includes a Depth First Search and a Breadth First search. These searching algorithms take different approaches to find a solution, our task is to investigate the most effective search algorithm.

2 Problem Definition

Students were given a list of five locations and were required to make use of the Google Maps Distance Matrix API as well as the googlemaps Python libraries to compute the distances between each pair of locations.
The locations given were:

1. University of Pretoria, Pretoria (Start)
2. CSIR, Meiring Naude Road, Pretoria
3. Armscor, Delmas Road, Pretoria
4. Denel Dynamics, Nellmapius Drive, Centurion
5. Air Force Base Waterkloof, Centurion

We were also tasked with implementing our own search tree data structure in python. Each branch in the tree represented a possible route and we needed to search every route in order to determine the shortest. Thus, the tree needed to be scalable and dynamically created, since the number of nodes in the tree would explode as the number of locations increased and each location should be added seamlessly to the tree and should not be hard coded.

The scalability of the tree needed to be taken into account, since the number of branches in the tree is proportional to the number of locations and the number of nodes in the tree would grow exponentially as the number of locations increased. Thus, if the tree is not optimized to handle such a large number of nodes, the amount of processing power and time needed to just construct the tree would be immense and thus not practical to be used for a real-world problem as opposed to a theoretical one.

The Tree also needed to be able to be printed out in order for us to verify the structure of the tree and ensure that the actual implementation was as designed.

Both, the Depth First Search and the Breadth First Search algorithms needed to be implemented by students in order to solve the Travelling Salesman problem. The Search techniques needed to be tailored to the tree implemented by students since the class variables and member functions would differ from one implementation to another. The effectiveness of these algorithms needed to be evaluated, thus the number of nodes evaluated or visited by each algorithm needed to be stored.

3 Tree Structure Implementation and Solution

My implementation of a search tree was a variation of a B - Tree wherein each node represented a particular location and each of the nodes children represented the different paths one could travel from the current location. To aid in my implementation I created separate node and tree classes, both of which can be found in Appendix A.

In my implementation I have taken advantage of the fact that when building the tree from the second level or second generation, the siblings of every node at a particular level, become that node's children in the next level, thus I have implemented "helper" functions in my node class to aid with this as described below.

My node class has several variables and helper functions to assist in the construction of my tree and to find the shortest path. Including the name of the location, the parent, a list of the nodes children, a Boolean to determine if it has been visited, the distance from the nodes parent and the total distance travelled in the path so far. I make use of multiple member functions such as the addChild(), getSiblings() and addSiblingsAsChildren() functions which adds a node as child to the current node, returns the siblings of the current node and adds the siblings of the current node as children to the current node.

My tree's constructor takes in the name of the starting location in order to begin constructing the tree. The starting location represents the root or head of the tree Additional functions were implemented in order to take in either a list of locations or a single location and add it to the tree. As the tree is built using a recursive function, I have ensured that the last node or the leaf node in each branch (path) represents the starting location again, thus ensuring that the complete path of possible travel is represented in the tree and these leaf nodes contain the total distance for their entire respective paths in their totalDistance class variable. I have also created a new node for each location in very path/branch, thus ensuring that although nodes may have the same name in the tree, they represent locations visited at different times and in a different sequence. This allowed me to ensure that data belonging to one location in a particular path, is not confused with data belonging to the location with the same name, but appearing in a different branch.

In order to neatly print my final built Tree, I have made use of the anytree python library which aided in providing a solution to the problem of achieving a graphical representation of the tree. My tree structure can be seen in figure 2.

4 Brute Force Search Algorithms

Brute Force Search Algorithms are extremely expensive search and relatively simple algorithms used to find a solution. They are termed as expensive due to the fact that they search almost every single node until a solution is found or there are no results found, thus performing a Brute Force search consumes a lot of computing resources.

4.1 Depth First Search

In a Depth First Search, the nodes in a single branch are first searched before backtracking to another branch until the entire tree has been searched. In my implementation, I have made use of a Python list to act as a Stack data structure in order to perform the search which allowed me to avoid making use of a recursive function to find the result. When a node is evaluated, it is marked as visited aiding in ensuring that no location in a particular path is evaluated more than once. The total distance of the path from the starting location, in our case, The University of Pretoria, through all four other locations and back to the starting location, as well as the names of the locations in the path is stored. In order to save time and processing, when a node is being evaluated, a check is done to see whether the current shortest distance found is less than the distance from the head to the current node. If this is true, the rest of the nodes in the path are ignored since they will not turn out to be our result. In doing this, I have managed to save time and resources by stopping some nodes from being checked. At the end of the function, once the entire tree has been searched, the total number of nodes evaluated, the shortest distance value as well as the shortest distance location names are printed on the screen for the user to see the result.

4.2 Breadth First Search

In a Breadth First Search, the tree is searched level by level for a result. Due to this nature, there is no way of avoiding certain nodes from being evaluated due the fact that the end of all possible paths is found in the leaf nodes and the algorithm has no way of determining that it has searched all the leaf nodes until the entire tree is searched. I have made use of the queue Python library to perform a Breadth First Search of the tree. Just as in DFS, once the result has been found (the entire tree searched) the total number of nodes evaluated, the shortest distance value as well as the shortest distance path are all printed on the screen for the user to see the result.

5 Results

	Breadth First Search	Depth First Search
Number of Node visits	89	59
Shortest Distance	60684	60684

Table 1: Results of DFS and BFS for Travelling Salesman Problem

Below is an output snippet of my program. It shows the shortest path found by both searches, the total distance of the path as well as the total number of nodes that were evaluated by each search algorithm in its pursuit to find the solution.

```
1 ---------- DFS ----------
Shortest Path is: ['University of Pretoria , Pretoria', 'CSIR , Meiring
    Naude Road , Pretoria', 'Armscor, Delmas Road, Pretoria', 'Denel
    Dynamics, Nellmapius Drive, Centurion', 'Air Force Base Waterkloof,
    Centurion', 'University of Pretoria , Pretoria']
It's Distance is: 60684
Number of nodes visited by DFS is: 59

6 ---------- BFS ----------
Shortest Path is: ['University of Pretoria , Pretoria', 'CSIR , Meiring
    Naude Road , Pretoria', 'Armscor, Delmas Road, Pretoria', 'Denel
    Dynamics, Nellmapius Drive, Centurion', 'Air Force Base Waterkloof,
    Centurion', 'University of Pretoria , Pretoria']
It's Distance is: 60684
Number of nodes visited by BFS is: 89
```

Figure 1: Program Output snippet

As we can see from figure 1, for the given problem the Depth First Search algorithm was the most optimal since it needed to visit less nodes in order to find the solution. Both search techniques found that the shortest path and thus the solution to this particular Travelling Salesman problem was:

1. University of Pretoria, Pretoria (START)
2. CSIR, Meiring Naude Road, Pretoria
3. Armscor, Delmas Road, Pretoria
4. Denel Dynamics, Nellmapius Drive, Centurion
5. Air Force Base Waterkloof, Centurion
6. University of Pretoria, Pretoria (END)

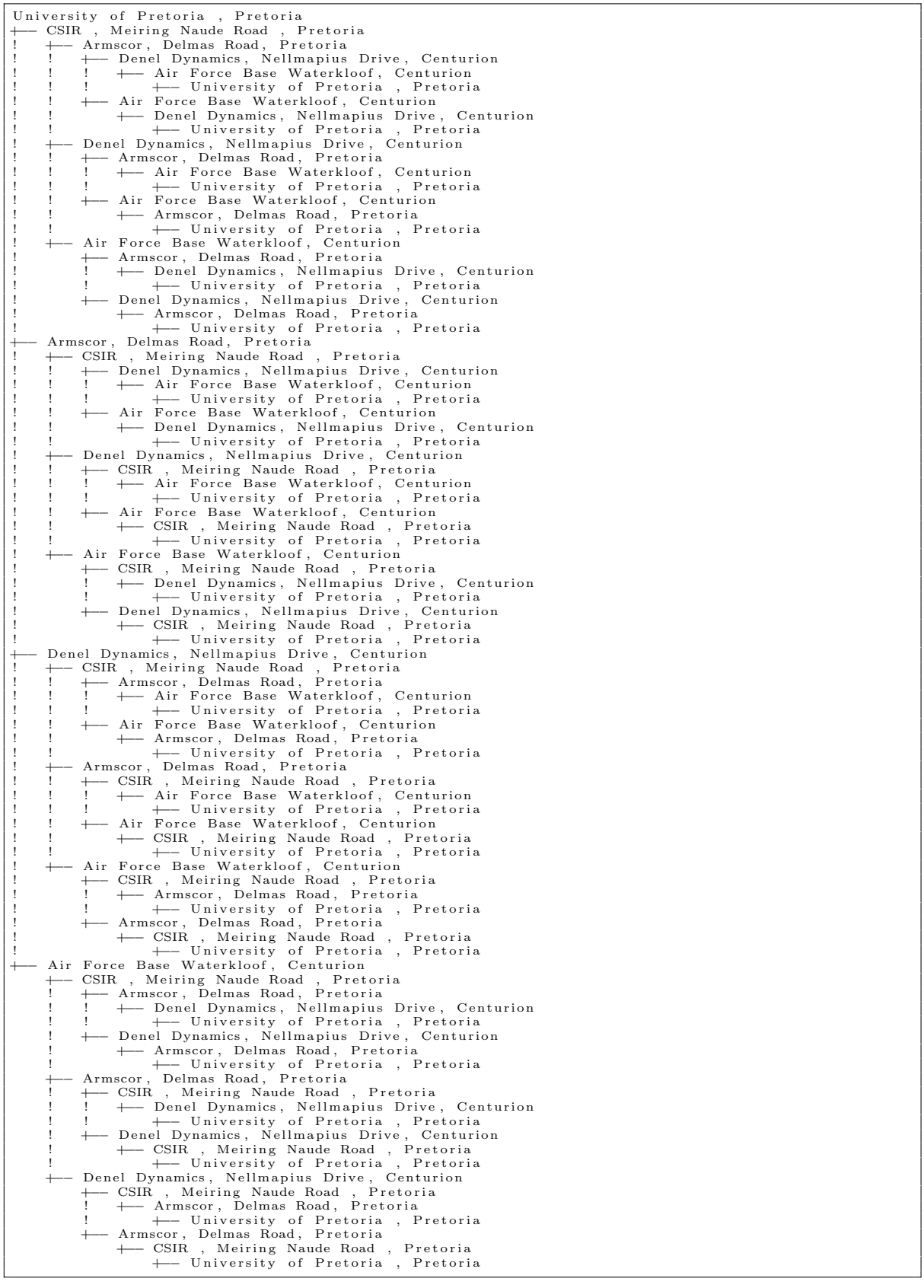

Figure 2: Console Print of the Tree Data Structure

6 Discussion

From figure 1, we can see that both the DFS and BFS managed to find the solution to the TSP given. Both searches found the same result and thus we can say with a large amount of certainty that the solution found is correct.

Both search algorithms were effective since they both found a solution to the Travelling Salesman Problem. The degree of effectiveness varied between the Depth First Search and the Breadth First Search due to the number of nodes needed to be evaluated in finding a solution and this in extension the amount of processing needed to find a solution.

The Tree was made up of a total of 89 nodes, the DFS only needed to evaluate 59 nodes, whereas the BFS needed to evaluate the 89 nodes. Clearly the Depth First Search was more efficient needing to evaluate 20 nodes less than the Breadth first search, thus saving both time and power in the process of finding the solution. The inefficiency of the BFS, was due to the fact that the algorithm needed to evaluate every single node in the tree and the efficiency of the DFS came from the fact that certain groups of nodes could be ignored from evaluation once a path was found.

This result cannot be used as conclusive evidence that DFS is always more efficient than BFS, since the order of the insertion of the locations, plays a huge role in the structure of the tree and in turn a role in the branch wherein the shortest path resides. If the locations or nodes are placed in such a way that the path lengths are sorted in decreasing order from left to right, the algorithm would need to evaluate every single branch (path) of the tree and thus every single node in order to find the solution. This would then result in DFS and BFS having to evaluate the same number of nodes and therefore have the same efficiency. The only conclusive statement we can make is that the DFS is as efficient and can be more efficient than the BFS depending on the structure of the tree. Thus, both search algorithms are effective in finding a solution, but the DFS technique, with the exception of certain cases, would be more efficient and optimal in finding the solution.

Since both these search techniques are expensive, it is recommended that a more efficient search algorithm be investigated and designed in order to attain optimal or more rather more efficient results.

7 Conclusion

Both the DFS and BFS were able to find the solution to the TSP. The efficiency of the DFS depends on the structure of the tree and is at worst as efficient as the BFS and at best more efficient than the BFS in finding the solution. Thus, we can say that if a Brute Force searching technique is to be used, the Depth First Search algorithm is the most ideal due to the fact that it can skip certain nodes in the tree from being evaluated, which would save both time and resources, since as the number of locations increase, the number of nodes added to the tree can grow at a gigantic rate. It is not recommended to use any of these expensive "Brute Force" searching techniques as they would be much too inefficient for larger trees with more locations. Smarter and faster search techniques, such as the A* Search algorithm, should be researched and tested.

8 Appendix A : Practical Code

```
#Mohamed Ameen Omar
#u16055323
#EAI 320 - Practical 1
#2018
import googlemaps
import queue
from anytree import Node, RenderTree
gmaps = googlemaps.Client(key='AIzaSyD5Jjcpu_zl38yBDijOSELow5eZhGq6aic')
tuks = "University of Pretoria , Pretoria"
csir = "CSIR , Meiring Naude Road , Pretoria"
airforce = "Air Force Base Waterkloof, Centurion" #West Gate, Hans Strydom Drive, Lyttelton, Centurion, 0157
denel = "Denel Dynamics, Nellmapius Drive, Centurion"
armscor = "Armscor, Delmas Road, Pretoria"
nameOfCities = [csir,armscor,denel,airforce] #list of the locations besides the starting point
```

Figure 3: Program Code 1 of 8 [Python]

```python
class node:
    def __init__(self, name, distance = 0, totalDistance = 0, parent = None, progeny = []):
        self.distance = distance #distance from parent to node
        self.name = name
        self.progeny = [] #list with all node's children
        self.totalDistance = totalDistance #totalDistance of the "Family" upto and inlcuding current node
        self.parent = parent
        self.visited = False
        self.anyTreeNode = None
    #adds the location passed in as a child for the current node
    def addChild(self, name):
        if(self.progeny != []):
            for x in range(0,len(self.progeny)):
                if(name == self.progeny[x].name):
                    return
        dist = gmaps.distance_matrix(self.name,name)['rows'][0]['elements'][0]['distance']['value']
        totalDist = self.totalDistance + dist
        tempNode = node(name,dist,totalDist,self)
        #tempNode.parent = self
        self.progeny.append(tempNode)
    def isLeaf(self):
        if(self.progeny == []):
            return True
        return False
```

Figure 4: Program Code 2 of 8 [Python]

```
#returns a list of the names of the current node's siblings
    def getSiblings(self):
        if(self.parent == None or self.parent.childHasSiblings() == False)
    :
            return [""]
        siblings = []
        for x in range(0, len(self.parent.progeny)):
            if(self.parent.progeny[x].name != self.name):
                siblings.append(self.parent.progeny[x].name)
        return siblings
    #checks if child has siblings
    def childHasSiblings(self):
        if (self.progeny == []):
            return False
        if(self.progeny == [""]):
            return False
        if(len(self.progeny) < 2):
            return False
        return True
    # Adds the siblings of current node as the children of current node,
    returns False if no siblings
    def addSiblingsAsChildren(self):
        siblings = self.getSiblings()
        if(siblings == [""] or siblings == []):
            return False
        for x in range(0, len(siblings)):
            self.addChild(siblings[x])
        return True
```

Figure 5: Program Code 3 of 8 [Python]

```python
class tree:
    def __init__(self,name):
        self.head = node(name)
    #recusive
    def resetVisited(self):
        tempNode = self.head
        pseudoStack = [self.head]
        while(pseudoStack != []):
            tempNode = pseudoStack.pop()
            tempNode.visited = False
            if(tempNode.progeny == []):
                continue
            for x in range(0,len(tempNode.progeny)):
                pseudoStack.append(tempNode.progeny[x])
    #adds the first level
    def addProgenyToHead(self, progeny = [""]):
        if(progeny == [""]):
            return
        for x in range(0,len(progeny)):
            self.head.addChild(progeny[x])
    #recusive function to build the tree once the "base" level/level 2 has been built
    def sortNodeChildren(self, tempNode):
        if(tempNode.childHasSiblings() == False):
            if(len(tempNode.progeny) == 1):
                tempNode = tempNode.progeny[0].addChild(self.head.name) #adds starting point as the ending point for the current path
            return
        for x in range (0, len(tempNode.progeny)):
            tempNode.progeny[x].addSiblingsAsChildren()
            self.sortNodeChildren(tempNode.progeny[x])
    #adds locations to tree
    def addAllLocations(self, locations = [""]):
        if(locations == [""]):
            return
        self.addProgenyToHead(locations)
        if(self.head.progeny == []):
            return
        self.sortNodeChildren(self.head)
    #add a single location to the tree
    def insertSingleLocation(self, myString):
        if(self.head.progeny == []):
            thisString = [myString]
            self.addAllLocations(thisString)
            return
        thisString = []
        for x in range(0, len(self.head.progeny)):
            thisString.append(self.head.progeny[x].name)
        thisString.append(myString)
        self.head = node(self.head.name)
        self.addAllLocations(thisString)
```

Figure 6: Program Code 4 of 8 [Python]

```
#insert an additional list of locations to the tree
    def insertMultipleLocations(self, myString):
        if(self.head.progeny == []):
            thisString = [myString]
            self.addAllLocations(thisString)
            return
        thisString = []
        for x in range(0, len(self.head.progeny)):
            thisString.append(self.head.progeny[x].name)
        for x in range(0, len(myString)):
            thisString.append(myString[x])
        self.head = node(self.head.name)
        self.addAllLocations(thisString)

    #makes use of anyTree in  order to print the tree
    def printTree(self):
        if(self.head == None):
            print("No Tree has been constructed yet")
            return
        print("Printing tree\n")
        self.resetVisited()
        tempNode = self.head
        myQueue = queue.Queue()
        myQueue.put(tempNode)
        while(myQueue.empty() == False):
            tempNode = myQueue.get()
            if(tempNode.visited == True):
                continue
            tempNode.visited = True
            if(tempNode == self.head):
                self.head.anyTreeNode = Node(tempNode.name)
            else:
                tempNode.anyTreeNode = Node(tempNode.name, tempNode.parent
    .anyTreeNode)
            for x in range(0,len(tempNode.progeny)):
                if(tempNode.progeny[x].visited == False):
                    myQueue.put(tempNode.progeny[x])
        self.resetVisited()
        for pre, fill, x in RenderTree(self.head.anyTreeNode):
            print("%s%s  " % (pre, x.name))
```

Figure 7: Program Code 5 of 8 [Python]

```
#returns a list with the route of the current path with the last node in
    the path passed in
def getPathFromEnd(tempNode):
    currentShortestPath = []
    while(tempNode != None):
        currentShortestPath.reverse()
        currentShortestPath.append(tempNode.name)
        currentShortestPath.reverse()
        tempNode = tempNode.parent
    return currentShortestPath
############################################ DFS
    ###################################################################
#DFS for shortest path for a tree
def DFS(myTree):
    myTree.resetVisited()
    tempNode = myTree.head
    numNodes = 0
    currentShortestDistance = -1
    currentShortestPath = []
    pseudoStack = []
    pseudoStack.append(tempNode)
    while(pseudoStack != []):
        tempNode= pseudoStack.pop()
        if(tempNode.visited == True):
            continue
        if( ( (currentShortestDistance <= tempNode.totalDistance) and
    currentShortestDistance != -1) or tempNode.visited == True): #if the
    path is longer or node already visited
            continue
        numNodes = numNodes + 1
        tempNode.visited = True
        if(tempNode.progeny == []): #lastNode
            if( (currentShortestDistance > tempNode.totalDistance) or
    currentShortestDistance == -1):
                currentShortestDistance = tempNode.totalDistance
                currentShortestPath = getPathFromEnd(tempNode)
        if(tempNode.progeny != []):
            x = len(tempNode.progeny)
            for x in range(len(tempNode.progeny)-1, -1,-1):
                if(tempNode.progeny[x].visited != True):
                    pseudoStack.append(tempNode.progeny[x])
    print("Shortest Path is:", currentShortestPath)
    print("It's Distance is:", currentShortestDistance)
    print("Number of nodes visited by DFS is:",numNodes)
    myTree.resetVisited()
```

Figure 8: Program Code 6 of 8 [Python]

```
################################ BFS
    ##############################################
def BFS(myTree):
    myTree.resetVisited()
    tempNode = myTree.head
    myQueue = queue.Queue()
    myQueue.put(tempNode)
    numNodes = 0
    path = []
    currentShortestDistance = -1
    while(myQueue.empty() == False):
        tempNode = myQueue.get()
        if(tempNode.visited == True):
            continue
        tempNode.visited = True
        numNodes = numNodes+1
        if(tempNode.progeny == []):
            if(currentShortestDistance == -1 or currentShortestDistance >
    tempNode.totalDistance):
                currentShortestDistance = tempNode.totalDistance
                path = getPathFromEnd(tempNode)
        for x in range(0,len(tempNode.progeny)):
            if(tempNode.progeny[x].visited == False):
                myQueue.put(tempNode.progeny[x])
    print("Shortest Path is:", path)
    print("It's Distance is:", currentShortestDistance)
    print("Number of nodes visited by BFS is:",numNodes)
    myTree.resetVisited()
    return
```

Figure 9: Program Code 7 of 8 [Python]

```
################################################ main program
    ##########################################################################
myTree = tree(tuks)
print("Building Search Tree\n")
myTree.addAllLocations(nameOfCities)
print("Search Tree is Fully Built\n")
myTree.resetVisited()
myTree.printTree()
print("")
print("---------------------------------- DFS ----------------------------------")
DFS(myTree)
print("")
print("---------------------------------- BFS ----------------------------------")
BFS(myTree)
```

Figure 10: Program Code 8 of 8 [Python]