UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

Practical 1

Author:
V Thalla

Student number:
16053525

February 15, 2018

DECLARATION OF ORIGINALITY

UNIVERSITY OF PRETORIA

The University of Pretoria places great emphasis upon integrity and ethical conduct in the preparation of all written work submitted for academic evaluation.

While academic staff teach you about referencing techniques and how to avoid plagiarism, you too have a responsibility in this regard. If you are at any stage uncertain as to what is required, you should speak to your lecturer before any written work is submitted.

You are guilty of plagiarism if you copy something from another author's work (e.g. a book, an article or a website) without acknowledging the source and pass it off as your own. In effect you are stealing something that belongs to someone else. This is not only the case when you copy work word-for-word (verbatim), but also when you submit someone else's work in a slightly altered form (paraphrase) or use a line of argument without acknowledging it. You are not allowed to use work previously produced by another student. You are also not allowed to let anybody copy your work with the intention of passing if off as his/her work.

Students who commit plagiarism will not be given any credit for plagiarised work. The matter may also be referred to the Disciplinary Committee (Students) for a ruling. Plagiarism is regarded as a serious contravention of the University's rules and can lead to expulsion from the University.

The declaration which follows must accompany all written work submitted while you are a student of the University of Pretoria. No written work will be accepted unless the declaration has been completed and attached.

Full names of student: ____________________

Student number: ____________________

Topic of work: ____________________)

Declaration

1. I understand what plagiarism is and am aware of the University's policy in this regard.
2. I declare that this assignment report is my own original work. Where other people's work has been used (either from a printed source, Internet or any other source), this has been properly acknowledged and referenced in accordance with departmental requirements.
3. I have not used work previously produced by another student or any other person to hand in as my own.
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

SIGNATURE: ____________________ DATE: ____________________

1 Introduction

The first practical of EAI320 introduces the Travelling Salesman Problem and poses the challenge of solving it in the context of a set of given locations.

The goal is to find the shortest path from the starting location through all locations once and back to the starting point. The point of finding this path can truly be appreciated when one imagines that the problem is being solved for an actual salesman that would like to make a round trip through the given locations back to where he started. He would most probably like to reduce the 'cost' of the trip by choosing the shortest route with 'cost' being the keyword.

This involves of the use of the GoogleMaps API to get the distances between each pair of locations.

2 Problem Statement

The practical aims to solve Travelling salesman problem(TSP). Five locations are given: 1. University of Pretoria, Pretoria (Starting location) 2. CSIR, Meiring Naude Road, Pretoria 3. Armscor, Delmas Road, Pretoria 4. Denel Dynamics, Nellmapius Drive, Centurion 5. Air Force Base Waterkloof, Centurion

The problem is divided into two tasks. The first task consists of creating the actual tree and the second task uses two different methods(Depth First search and Breadth First search) to find and compare the two solutions.

3 Methodoly

Task 1 requires the creation of a tree that essentially creates a hierarchical structure such that all permutations of the path that can be travelled are contained inside. For example, taking the given location, we would have the starting location(1) as the root. This root would have 4 children(2 - 5). Then each of these four children would have 3 children each (3, 4, 5), (2, 4, 5), (2, 3, 5) and (2, 3, 4). This pattern is repeated till the lowest level is reached where there is only one child. This is the recursive tree method. The use of such a memory-intensive tree like this seems illogical when a graph structure provides more efficiency and was, therefore, used. The graph used for this structure can be seen fig 1. However the tree implementation is also provided in the code.

The first step is the Node class. Let "Current" be the name of an instance of the Node class. It contains 3 fields: value(name of the location), and 2 arrays: children and distances. The children array stores references to the children nodes of Current. The distances array is a parallel array to the children array storing the distance from Current to each child node in the corresponding index. The Node also provides functionality to add a child to the children array and to find the index of a node in its children when given a value string to match.

The reason for the graph is very simple and can be explained in terms of memory

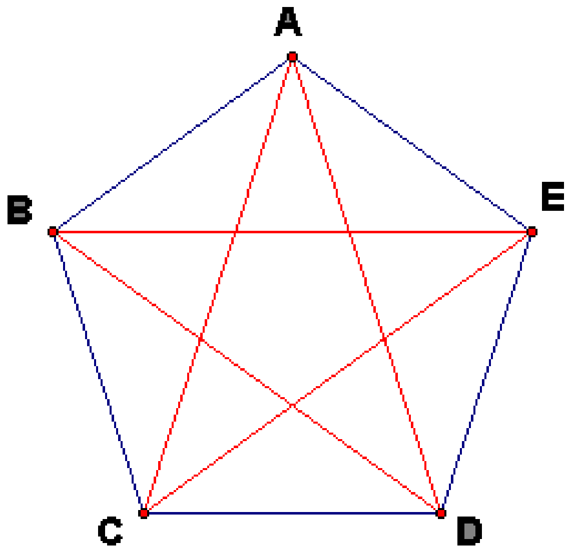

Figure 1.1: Graph Structure for TSP[1]

use. The graph structure keeps an array of all the nodes in the graph and interconnects them through an array of references mentioned as the children array earlier. On the other hand, the recursive tree needs the creation of a new node each time to add a child. This creates several duplicates in the tree and, therefore, wastes memory. Although the effects on memory will not be significant unless the tree is to be made for a number of nodes of a higher order.

The Tree contains creates the completely interconnected graph internally. This is configured by an 'insert' function and an array internal to the class called 'nodes'. 'nodes' contains all the nodes in the graph. The insert function creates a new node('Current') each time a new value is to be added. It then adds 'Current' as a child to all existing nodes and adds all the existing nodes as a child to 'Current'. Then 'Current' is added to the internal 'nodes' array Distances are set through the updateDistances() method which basically uses the googleMaps API to calculate the distance between each pair of locations and stores them in the distances of array of the node.

Task 2 required the use of a Depth First Search and Bread First Search to solve the TSP, counting the number of nodes visited along the way so as to see the difference in performance and efficiency in the corresponding solutions.

The Depth First search uses a recursive function to travel the several permutations of the path. The process starts at the root node and enters it's children and then the children's children and so on till a complete path is traversed(leaf node is reached) whose accumulated distance can be checked against the shortest recorded distance. This determines a new shortest distance if applicable, else a new path is tried. This process is spread across the entire graph.

The Breadth First Search uses a the recursive tree implementation as using the graph didn't work in the given time. It has a queue beginning from the root. The queue dequeues elements in First-in-First-Out order so the breadth of the graph is travelled on each level of the tree. So the first path is contains the root and all it's children. The second path contains the root, it's first child and all the first child's children. The third path path contains the root, it's second child and all the second child's children. This process works until the entire tree is traversed in this Breadth First manner. This Search Algorithm was adapted from a stack overflow answer to fit the solution in a clever way[2].

4 Results and Discussion

Criteria	DFS	BFS
Nodes visited	40	64
Shortest Distance(m)	60684	60684

Figure 1.2: DFS and BFS Results

```
CONSOLE RESULTS:
Best DFS path:
University of Pretoria, Pretoria
|
V
CSIR, Meiring Naude Road, Pretoria
|
V
Armscor, Delmas Road, Pretoria
|
V
Denel Dynamics, Nellmapius Drive, Centurion
|
V
Air Force Base Waterkloof, Centurion
|
V
University of Pretoria, Pretoria

Shortest DFS Distance is 60684m
DFS Path distance for checking: 60684m
Total nodes visited(DFS): 40

Shortest BFS Distance is 60684m
Total nodes visited(BFS): 64
```

Both the methods are appropriately implemented with an optimization such that they stop traversing a path if the accumulated distance already exceeds the shortest recorded distance. Another thing to note is that since the Breadth First search basically travels the same path almost 4 times, it's count will always be more than the Depth First search.

5 Conclusion

This TSP required the root to the starting and ending point with a completely interconnected path that allowed it to be traversed in multiple ways to reach the solution. This was executed through the Depth First Search and Breadth First Algorithms. Although it is not proof that it will be the winner in all cases, the Depth First search which seemed to be more suited to the problem at hand.

6 Bibliography

[1] https://sites.math.washington.edu/ king/coursedir/m444a02/class/11-18-penta-answers.html
[2] amit: https://stackoverflow.com/questions/29141501/how-to-implement-bfs-algorithm-for-pacman

7 Appendix

```python
"""
Created on Fri Feb 9 17:39:03 2018

@author: Vishal
Final Code for the Practical including Task 1 and Task 2
"""
import googlemaps
from queue import Queue

gmaps = googlemaps.Client(key = 'AIzaSyB8Sjq1wYpnC2mKqyheCNeeTIpZKNVKovI')

class Node(object): #needed for new style class
    #class will only hold a value which will be title and children nodes
    def __init__(self, value):
        self.value = value
        self.children = []
        self.distances = [] # will be parallel to children list
    """
    returns index of node(value) in self.children
    """
    def find(self, value):
        i = 0
        while (i != len(self.children) and self.children[i].value != value):
            i += 1
        return i

    def hasChild(self, value):
        if (self.find(value) == len(self.children)):
            return False
        return True

    #don't clutter the tree unnecessarily since there is no delete method
    def addChild(self, child):
        if (self.find(child.value) == len(self.children)): # not found
            self.children.append(child)

class Tree(object): #will be structured like a graph
    def __init__(self):
        self.nodes = []
        self.root = None
        self.updateFlag = False

    """
    checks if the given value node already exists in tree
    """
    def isDuplicate(self, value):
        i = 0
        while (i < len(self.nodes) and self.nodes[i].value != value):
```

```
        i += 1
    if (i == len(self.nodes)):
        return False
    else:
        return True
"""
inserts into completely connected graph
"""
def insert(self, value):
    #create a new root(Starting location) if needed
    if (self.root == None):
        self.root = Node(value)
        self.nodes.append(self.root)
        return
    """
    # if not root then each node in graph needs a link to newly added node
    # new node also needs a link to each existing node
    # new node then needs to be added to the existing nodes list
    # don't clutter graph with duplicates
    """
    if (not self.isDuplicate(value)):
        temp = Node(value)
        for i in range(len(self.nodes)):
            self.nodes[i].addChild(temp)
            temp.addChild(self.nodes[i])
        self.nodes.append(temp)
        self.updateFlag = True
"""
sets correct distances in each node
"""
def updateDistances(self):
    for i in range(0, len(self.nodes)):
        temp = []
        for j in range(0, len(self.nodes)):
            if (i == j):
                continue
            dist_dict = gmaps.distance_matrix(self.nodes[i].value,
                self.nodes[j].value)
            temp.insert(self.nodes[i].find(self.nodes[j].value),
                dist_dict["rows"][0]["elements"][0]["distance"]["value"])
        self.nodes[i].distances = temp
        #overwrite previous distances array
    self.updateFlag = False

"""
returns actual node object that has value
"""
def find(self, value):
    i = 0
    while (i != len(self.nodes) and self.nodes[i].value != value):
```

```
            i += 1
        if (i == len(self.nodes)):
            return None
        return self.nodes[i]

    """
    prints all children of node(value) with distance
    """
    def printChildren(self, node):
        print(node.value + '\n')
        for i in range (len(node.children)):
            print(node.children[i].value)
            if (node.distances):
                print(node.distances[i])
        print()

    """
    returns deep copy of tree
    """
    def clone(self):
        other = Tree()
        for i in range(len(self.nodes)):
            other.insert(self.nodes[i].value)
        return other

    """
    distance between 2 nodes
    """
    def getDist(self, n1, n2):
        if (self.updateFlag == True):
            self.updateDistances()
        if (n1.distances):
            return n1.distances[n1.find(n2.value)]
        else:
            graphN1 = self.find(n1.value)
            return graphN1.distances[graphN1.find(n2.value)]
#       else:#lazy hacking
#           dist_dict = gmaps.distance_matrix(self.nodes[i].value,
    self.nodes[j].value)
#           return dist_dict["rows"][0]["elements"][0]["distance"]["value"]

    """
    gives a deep copy of nodes references in given path p1
    """
    def copyPath(self, p1 = []):
        out = []
        for i in range(len(p1)):
            out.append(p1[i])
        return out
```

```
"""
prints the given path of nodes
"""
def printPath(self, path):
    for i in range(len(path)-1):
        print(path[i].value + "\n|\nV")
    print(path[len(path)-1].value + "\n")

"""
needs nodes in path to be ordered
returns total path distance given a path = array of nodes that are
 connected through children
"""
def getPathDistance(self, path):
    total = 0
    if (len(path) > 0):
        for i in range(len(path)-1):
            total += self.getDist(path[i], path[i+1])
    return total

"""
uses treeRec to make a recursive tree using the graph
"""
def toRecTree(self, node):
    if (node != None):
        tree_root = Node(node.value)
        if (not self.nodes[0].distances):
            self.updateDistances() # ensure values are stored
        for i in range(len(self.nodes)):
            if (self.nodes[i].value == tree_root.value):
                continue
            tree_root.addChild(Node(self.nodes[i].value))
            #tree_root.distances.append(node.distances[node.find(self.nodes[i].value)])

        return self.treeRec(tree_root)
    return None

def treeRec(self, node):
    if (len(node.children) > 1):#recursion end condition
        for child in node.children:
            #graphNode = self.find(child.value) # find value in self.nodes
                array
            for i in range(len(node.children)):
                if (node.children[i] == child):
                    continue
                child.addChild(Node(node.children[i].value))
                #child.distances.append(graphNode.distances[graphNode.find(node.children[
            self.treeRec(child)
    return node
```

```
"""
DepthFirst Search: (Task 2)
node is the starting point
requires the tree to be connected as in all nodes should connected with
 a children link somewhere
This can be optimised to STORE and use the shortest distance, the
 corresponding best path and node count but not in scope of practical
"""
def DepthFirstSearch(self, start):
    path = self.copyPath(self.nodes)
    shortSum = 0
    for i in range(len(path)-1):
        shortSum += self.getDist(path[i], path[i+1])

    shortSum += self.getDist(path[len(path)-1], start)
    self.DFSShortest = shortSum
    path.append(start)
    self.DFSPath = self.copyPath(path)
    #^ Just to set a reference to compare against and determine shortest
        route
    count = self.DFS(start, 0, [])
    print("Best DFS path: ")
    self.printPath(self.DFSPath)
    print("Shortest DFS Distance is " + str(self.DFSShortest)+ "m")
    print("DFS Path distance for checking: " +
        str(self.getPathDistance(self.DFSPath)) + "m")
    print("Total nodes visited(DFS): " + str(count))

"""
start is the starting node
total is the running total summed as path is traversed
path is the accumulated path so far
"""
#DFS works
def DFS(self, start, total, path= []):
    if (len(path) == 0):
        rest = [i for i in start.children]
        #this can be the only possibility
        path.append(start)
    else:
        rest = [i for i in path[len(path)-1].children if i not in path]
        #children of last node in path that are not already in path
    if (len(rest) == 0):
        return 0
    currSum = 0
    if (len(rest) == 1): # just using the problem and hacking it for the
        solution
        curr = path[len(path)-1]
```

```
        currSum += self.getDist(curr, rest[0])
        if (currSum + total < self.DFSShortest):
            currSum += self.getDist(rest[0], start) #using the problem
                specs to my advantage here
            if (currSum + total < self.DFSShortest):
                self.DFSShortest = currSum + total
                self.DFSPath = self.copyPath(path)
                self.DFSPath.extend([rest[0], start])
                #uncomment following two lines to see how best path is made
                #print("current shortest = " + str(self.DFSShortest))
                #self.printPath(self.DFSPath + "\n")
            #return 2 #only distance from last to start checked. start is
                not processed
        return 0
    count = 0
    for i in range(len(rest)):
        #remember to delete path elements in higher levels
        currSum = total + self.getDist(path[len(path)-1], rest[i])
        if (currSum < self.DFSShortest):
            path.append(rest[i])
            count += self.DFS(start, currSum, path) +1
            path.remove(rest[i])
        currSum = 0
    return count

"""
BreadthFirst Search: (Task 2)
node is the starting point
requires the tree to be connected as in all nodes should connected with
 a children link somewhere
This can be optimised to STORE and use the shortest distance, the
 corresponding best path and node count but not in scope of practical
"""
def BreadthFirstSearch(self, start):
    path = self.copyPath(self.nodes)
    shortSum = 0
    for i in range(len(path)-1):
        shortSum += self.getDist(path[i], path[i+1])

    shortSum += self.getDist(path[len(path)-1], start)
    self.BFSShortest = shortSum
    path.append(start)
    self.BFSPath = path
    #^ Just to set a reference to compare against and determine shortest
        route
    BFSstart = self.toRecTree(start)
    count = self.BFS(BFSstart)
    print("Shortest BFS Distance is " + str(self.BFSShortest)+ "m")
    print("Total nodes visited(BFS): " + str(count))
```

```
    #adapted from asnwer given by amit at
       https://stackoverflow.com/questions/29141501/how-to-implement-bfs-algorithm-for-pacm
    """
    begins from start and creates a queue to Bread first search and compare
     accumulated route distance to previously recorded shortest distance
     and set new shortest distance if found
    """
    def BFS(self, start):
        unvisited = Queue()
        unvisited.put(start)
        count = 1
        pathDistances = dict()
        pathDistances[start] = 0 # stores current distance in path for a node

        while(not unvisited.empty()):
            curr = unvisited.get()
            #distance = pathDistances[]
            currSum = pathDistances[curr]
            rest = [i for i in curr.children]
            if (currSum < self.BFSShortest):
                for i in range(len(rest)):
                    if (i == 0):
                        currSum += self.getDist(curr, rest[i])
                    else:
                        currSum += self.getDist(rest[i-1], rest[i] )
                    pathDistances[rest[i]] = pathDistances[curr] +
                        self.getDist(curr, rest[i])

                    count += 1
                    if (currSum < self.BFSShortest):
                        if (i == len(rest)-1 and currSum +
                            self.getDist(rest[i], start) < self.BFSShortest):
                            self.BFSShortest = currSum + self.getDist(rest[i],
                               start)
                    unvisited.put(rest[i])
        return count

tree = Tree()
tree.insert("University of Pretoria, Pretoria")
tree.insert("Denel Dynamics, Nellmapius Drive, Centurion")
tree.insert("CSIR, Meiring Naude Road, Pretoria")
tree.insert("Air Force Base Waterkloof, Centurion")
tree.insert("Armscor, Delmas Road, Pretoria")
tree.updateDistances()

tree.DepthFirstSearch(tree.root)
print()
tree.BreadthFirstSearch(tree.root)
```