FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR THESIS

Study program/specialization: Bachelor's degree in engineering / Computer Science	The spring semester 2022 Open / Confidential
Authors: Markus Fenne Karlsen, Enes Ok	
Course coordinator: Reggie Davidrajuh Supervisor: Reggie Davidrajuh Co-supervisor: Rituka Jaiswal	
Thesis title: Application of Graph Algorithm in Smart Grid	
Credits (ECTS): 20	
Keywords: Graph Theory Graph Algorithm Steiner tree Minimum spanning tree	Number of pages: 22 + appendix/other: 5 Stavanger, 15. May 2022

Table of contents

Acknowledgements

We would like to thank and express our gratitude to our supervisors Reggie Davidrajuh and Rituka Jaiswal for their guidance and support. We also want to thank all of the teachers and classmates that have taught and encouraged us throughout the years of our education.

Abstract

Rituka Jaiswal and Reggie Davidrajuh developed a graph algorithm for finding Steiner trees in MATLAB language. The algorithm they made is called Jaiswal & Davidrajuh's algorithm. This project deals with realizing, implementing and analyzing that graph algorithm in Python language. Firstly, this project revolves around a study of Steiner trees and graph algorithms for finding minimumweight spanning trees. The second part of this bachelor thesis describes our implementation of Jaiswal & Davidrajuh's graph algorithm in Python language and tests it with examples.

Chapter 1

Introduction

1.1 Problem statement

Rituka Jaiswal and Reggie Davidrajuh developed a graph algorithm for Steiner trees in MATLAB language [1, 2, 3]. This project deals with realizing, implementing and analyzing that graph algorithm in Python language.

Firstly, this project revolves around a study of Prim's algorithm, Steiner trees and graph algorithms for finding minimumweight spanning trees. The second part of this bachelor thesis describes our implementation of Jaiswal & Davidrajuh's graph algorithm in Python language and how we tested it.

1.2 Background

We were tasked by Reggie Davidrajuh with researching graph algorithms and implementing Jaiswal & Davidrajuh's algorithm in Python language. We accepted the task and saw it as an oppurtunity to make use of and expand on our knowledge on graph algorithms and programming.

This paper and the work done on it, is derived from Jaiwal and Davidrujah's two papers [1, 2] and code [3].

When viewed from high above or on a map, a wind farm (a group wind turbines) can be represented as a graph, where each vertex in the group of vertices represents a single wind turbine and each edge in the graph represents a cable connection between two wind turbines. How can all wind turbines be connected in a way that yields the minimum length of cables? The application of Jaiswal & Davidrajuh's algorithm in Smart Grid solves this question.

Chapter 2

Theory

2.1 Definitions and notations

2.1.1 Graphs, Algorithms and Graph algorithms

A graph is a set of vertices (nodes) and edges. An edge is a connection between two vertices. Quite simply, a graph can be described as a group of dots with a set of lines connecting the dots. Vertices can represent a group of people, a set of electrical devices, wind turbines or intersections, to name a few. A connected graph is a graph where there are paths to all of the vertices in the graph. This means that there are no isolated vertices. In rare cases, a vertex has an edge that connects to itself and none of the graphs in this paper have such a case.

Edges are interconnections of two vertices, and can represent relations between people, wired/wireless connection between electrical devices, cables between wind turbines or roads connecting intersections, among other things.

Edges in a graph can be either weighted or not weighted. If an edge in a graph is weighted then there is a number affixed to the edge. That number represents how much it will cost to travel between the two vertices that

the edge connects. In real life, the numerical value affixed to the edge, the weight, can for example be a measurement of the length between two points or how much time it will take to travel from one city to another. If an edge in a graph is unweighted then there is no numerical value affixed to the weight, which signifies the travel cost between two vertices in question.

Edges in a graph can be either directed or undirected. If an edge is directed then it means that it is only possible to travel between two connected vertices in one direction. For example, vertices A and B are connected by a directed edge and it is only possible to travel from A to B and not from B to A. If an edge is undirected then it is possible to travel back and fourth between two vertices connected by an edge.

An algorithm is a set of instructions for solving a problem.

Graph algorithms are algorithms related to graphs, vertices and edges.

Graph algorithms can be used for finding out more about a graph, such as the shortest route between vertices A and B or finding a specific vertex. Commonly, graph algorithms involves going through (traversing) graphs and performing arithmetic operations with vertices and/or edges.

In programming, a graph can be represented and be made using a matrix. A matrix is a set amount of numbers positioned so that a grid is formed. A matrix can also be said to be a three dimensional array.

2.1.2 Minimum weight spanning tree

A tree is a graph that is acyclic (forms no cycles) and connected. If a group of vertices, in a given graph, and the edges between them form an enclosed polygon then the given graph is cyclic as the graph contains a cycle. If that is not the case then the graph is acyclic.

A minimum weight spanning tree connects all the vertices in a weighted and connected graph, and the sum of the weights of all of the edges of the tree yield the minimum possible weight.

The minimum weight spanning tree (MST) problem is a graph problem, where the problem is to find a minimum weight spanning tree of a given connected and weighted graph.

2.1.3 Steiner Tree

A Steiner tree is similar to an minimum weight spanning tree in that way they both trees connect certain vertices and have minimal amount of weight, however what makes a Steiner tree different is that a Steiner tree doesn't necessarily connect all vertices in a given graph. The Steiner tree problem in graph theory is a problem where a set of vertices, called terminal vertices, are to be connected in a way that yields a tree with minimal possible weight. The amount of terminal vertices in the Steiner tree problem is variable since the amount of terminal vertices is a subset of the total amount of vertices in a given graph. In many occurrences, in order to construct a Steiner tree of a given graph, additional vertices, that are not terminal vertices, need to be included in the Steiner tree. Thus connecting the terminal nodes and constructing the Steiner tree. These additional vertices in the Steiner tree, that are not terminal vertices, are called Steiner vertices.

The minimum weight spanning tree problem is a special case of the Steiner tree graph problem in which all vertices in the given graph are terminal vertices. Thus there will be none Steiner vertices in the tree that the solution of a minimum weight spanning tree problem. A MST is a special case of Steiner Tree. In an MST all vertices are terminals.

2.1.4 Prim's algorithm

One of the graph algorithms for finding the minimum weight spanning tree of a given graph is Prim's algorithm. In order to utilize Prim's algorithm on a given graph, the graph has to be undirected, weighted and connected. Prim's algorithm is a greedy algorithm. A greedy algorithm is an algorithm that solves a problem by choosing the most optimal answer at any time.

2.2 Related work

Reggie Davidrajuh and Rituka Jawal have written two papers on Steiner trees and an algorithm for Steiner tree problems. The first paper details their solution of a graph problem concerning wind farms [1]. The second paper details a simple algorithm for finding Steiner trees [2].

Chapter 3

Methodology

3.1 Planning

After having been assigned our thesis, we received the papers and computer program related to the Jaiswal-Davidrajuh algorithm [1, 2, 3].

In order to conduct our work on this thesis project in a proper way, we first had to consider and plan. The first step was to understand and analyse the aforementioned papers as well as the code. The second step was to consider our options when it comes to programming and how to implement the algorithm. The third step was to reflect and evaluate our own work in order to be critical.

3.2 Early process

In the early stages of our process, we focused on gaining knowledge instead of quickly producing a product. Therefore, we spent a reasonable amount of time researching as well as talking to our supervisor to better understand graph algorithms, especially for finding minimum weight spanning trees and Steiner trees. Chapter 2, in this paper detailing theory relevant for the thesis, is the result of the conduction of the first step of our plan.

3.2.1 Composition of the given computer program

As stated earlier, we received from Reggie Davidrajuh the computer program [3] that contained the implementation of the Jaiswal-Davidrajuh algorithm. The program was written in the MATLAB programming language. Understanding and analysing the code within this program was crucial to conducting our thesis project. To briefly explain the composition of the computer program: there is a file which acts as a hub for the entire program. That MATLAB-file is named ICECCME.m and is the file in which the Jaiswal-Davidrajuh algorithm is implemented by Reggie Davidrajuh and Rituka Jaiswal. The file calls upon five functions from other files, one after the other. The first function that gets called, creates a sample graph by using a matrix and then the graph gets returned and is assigned to a variable in ICECCME.m. The second function takes the aforementioned graph and uses a modified version of Prim's algorithm on it and then returns a minimum weight spanning tree. The third function just prints the output of the previous function. The fourth function takes the tree as an input, modifies it by eliminating Steiner vertices and then outputs the new tree. The fifth function that gets called, prints out the output from the last function, which is the Steiner tree for the given sample graph from the beginning of the program.

3.2.2 Choosing the technology

After understanding the relevant theory and the code, we began to look into our options on how to implement Jaiswal & Davidrajuh's algorithm in Python language and translate the MATLAB program into a Python program. We saw it the most appropriate to translate each code line, one by one, ourselves manually. Since we are going to be programming with graphs, creating graphs by utilizing matrixes will be useful. There are differences between MATLAB and Python when it comes to matrix functionality. MATLAB has matrix functionality that works well. whereas Python, in its base form (without any external libraries), does not have matrix functionality. Therefore we chose to use the external library for Python called NumPy. NumPy adds matrix functionality to Python. The goal of this paper is not to develop a Python program that adds matrix functionality. Sometimes there was not a necessarily direct way to translate a code line

seeing as MATLAB and the external Python library NumPy have different matrix functions. There are some matrix operation/manipulation functions that NumPy does not have, and vice versa.

3.3 Implementation

This section details our method of implementing the code.

3.3.1 Prim's algorithm

Start by importing the necessary libraries, in this case: numpy, sys, and our custom Graph class SampleGraph

Set the necessary variables as easy to reach local variables. In the given file a matrix of zeroes given by the number of terminals is defined, but this sees no use in the rest of the algorithm so we disregarded that line of code.

Set the variables m and n to be the size of the matrix. In Python this can be done in one function, but in Python we use the NumPy function size(). We do this along a specific axis, in which axis 0 represent rows and axis 1 represent columns.

Do a nested for-loop with the range from 0 to m in the outer loop and from 0 to n in the inner loop. In MATLAB these for-loops start at 1, but due to the indexing difference in Python we instead start at 0.

Then, if a given element of the graph A[i][j] is set to 0, we instead set it to infinite. In MATLAB there is a variable type named infinite, but this does not exist in Python. We cannot use NumPy's equivalent of infinite either, because this is a floating infinity type number, which cannot be converted into an integer. We instead set it to sys.maxsize, which is the highest possible integer Python allows for. This number is so unrealistically high that it is not going to cause problems for the algorithm.

After we have our array of small and high integers, we define some new

variables. We register the number of vertices, a starting list named tree vertices, and an empty array for our minimum spanning tree.

We then enter a while-loop, with the condition of: while the length of tree vertices is smaller than the number of vertices.

In this loop we first set a new variable named minWeight to infinite, again, in our version of the implementation of the algorithm we instead set it to sys.maxsize. Then, disregarding the MATLAB debugging code, we next enter a new for-loop, from 0 to the length of the tree vertices.

We next set a new variable named minU to be the value in the tree vertices list that corresponds with the index of the for-loop. We then use this variable as the row for the graph, and check for the smallest weight within the row. Afterwards, we extract the weight and ints corresponding to the column data. We now know the edge from the starting vertex minU with the smallest weight, and what other vertices it connects to. Thereafter, we check if this edge is smaller than minWeight, and if it does not already exist in the tree vertices list. If these conditions are met, we set minWeight to be the new weight, and set u and v variables to be their min counterparts.

In MATLAB it is possible to simply write "min(A(minU,:))" in order to get the minimum weight and minV variables, but in Python we have to write a custom function. We simply assign minWeight and minV to be equal to the first corresponding values within A[minU][0]. After we have a starting value, we can compare the other weights to see if they are smaller or not, and if they are then we replace the existing minwt and minV values. This is a greedy algorithm at work.

Next, we mark whatever edge we found to be the smallest as visited, meaning we set the vertices value to be sys.maxsize. Then, we append v to the tree vertices list, meaning we now have a new row we can check for a minimum weight in. The MATLAB implentation of this algorithm also sets vertices(v).pi = v, but this seems to be of no use, thus it is disregarded from the Python implementation.

Next, we extend the minimum spanning tree matrix with the values of u, v, and minWeight, with each set of variables forming a new row, meaning that the first column holds the u values, the second column holds the v

values, and the third column has their corresponding weights. In the Python implementation of this algorithm, things are a bit more complicated. This is because the first assignment has to set the structure of the matrix, and after this first assignment we can then stack the rest of the inputs under the first one. We also do a check to make sure the value of minWeight is not sys.maxsize, just in case some assignment issue occurs. This becomes more prevalent when working with randomly assigned adjacency matrix, as they may cause erratic or unpredictable behaviour.

This assignment continues until the while-loop is over, at which point the complete minimum spanning tree is returned. In the MATLAB implementation of the algorithm, this is done by assigning the MST to the input of the function V, however, in our Python implementation we instead create a new SampleGraph class containing the existing vertices names, and terminals, but now with a new graph. This is done for the sake of consistency, as well as making sure we can transfer the other properties of the existing graph into the future steps of the algorithm.

3.3.2 Steiner algorithm

The Steiner spanning tree algorithm starts in much the same fashion, by setting the input variables as local variables. This time around we assign the list of terminals, the list of vertices, the number of terminals, and an empty list for the terminal indices.

Then we enter a nested for-loop, from 0 to the number of terminals in the outer loop, and 0 to the length of nodes in the inner loop. We then assign t to the value of the terminals list that corresponds with the index of the outer loop, and n to the value of the vertices list that corresponds with the inner loop. Afterwards, we compare the two variables, and if they match, the index of the inner loop is appended to the terminal indices list. Essentially, what this means is that we are finding the vertex indexes that correspond with the given set of terminal vertices. These functions are pretty much the same in both Python and MATLAB.

Next, we make a variable for the edges of the minimum weight spanning tree. We also make a matrix of zeroes named A2, which is the same size as the matrix we originally started with. In the MATLAB implementation of the algorithm, this is done by taking the size of the original graph, which was automatically imported, but in our Python implementation of the algorithm we do not do so. Instead, we export the size of the graph from the prims_modified function, and import it as a variable. This makes it so that we do not have to import the entire matrix over, which would be unnecessary, because it is only used in this one circumstance.

The next step is filling in the values from the MST graph, into the new empty A2 graph. The first and second column of the MST graph being the coordinates, and the third column being the value to insert. It is done twice per value, because this matrix is symmetric, meaning for any i and j, the values A[i][j] and A[j][i] are always the equal to each other.

Now we have a graph that contains our extracted minimum spanning tree. The next step is to reduce it into a Steiner tree.

A variable called Iterations_Complete is set to false, we then make a while-loop that runs so long as Iterations_Complete remains false.

Inside the loop, we first set Iterations_Complete to True, then we enter a for-loop from 0 to the amount of vertices. We then check if the current index is not a terminal index, if it is not, we check how many connections the vertex has. If the number of connections is only one, this means that the vertex is an end vertex, in other words, having no nodes relying on it to be a part of the tree, we then remove the vertex from the tree, and set Iterations_Complete back to false.

This continues until there are no more vertices to remove, meaning we have completed the Steiner tree. In a similar manner, as in the Prim's algorithm, the our Python implementation returns an entire instance of the SampleGraph class.

3.3.3 Random Adjacency Matrix

This subsection details how we implemented a function that procedurally generates undirected, connected and weighted sample graphs by creating and using a random adjacency matrix. These graphs can be used to test our Python implementation of the Jaiswal & Davidrajuh's algorithm.

We start by importing a set of variables: Range(start,stop), Weight(start,stop), NodeCount,TerminalCount. Range is the amount of edges any given vertex can have, and Weight is the value of these edges. NodeCount is the total amount of vertices in the matrix, and this variable is used to make the NodeCount X NodeCount Matrix. TerminalCount is the amount of vertices that are to be assigned as terminals.

For example, rand_adjacency_matrix((1,3),(10,20),15,4), would output a random 15X15 matrix, with each vertex having anywhere from 1-3 edges, with a value from 10-20, and also 4 randomly assigned vertices as terminal vertices.

Firstly, we instantiate an empty matrix of the of the correct size, with the array elements having the type int. Then we create an empty dictionary named nodedict, and an integer nodesum set to 0. Then, in a for-loop from 0 to NodeCount, we first generate a random number using the the Range input values as boundaries. We then create a key-value pair, with the index of the for-loop being the key, and the random number being the value, we also increase the nodesum with the same value. Now we have a dictionary containing every vertex, and a corresponding random amount of edges to connect to it. Next we check if the nodesum is an odd or an even number, if its an even number we do not need to do anything else, if its an odd number however, we reduce the value of the first key value pair number that is not already at the lowest boundary by one. This is because any connection between two vertices require two of the values. If the number was odd the total amount of connections would not add up.

Next we create an empty set named connectedNodes, this set will contain the contain the vertices connected together in the form of tuples. A list could be used here instead of a set, but a set provides more clarity, because the contents of the set have to be unique. The contents are always going to be unique regardless of whether we use lists or sets, but because sets

have uniqueness as an implied characteristic it makes more sense for this application.

Afterwards, a while-loop with no specific condition is created. Inside the loop we first find the first highest value within the node dictionary, and its corresponding index. This will be one of the vertices used to make a connection in this loop. Then, a list is made named randrange, from 0 to Nodecount, and the index that we found was the highest is also removed from the list. Now we have a range of numbers to connect our first vertex to. Then, we check the connectedNodes set, to see if the first vertex already have any connections. If it does, we remove these vertices as well from the randrange list. We also check if any of the other already have reached their maximum amount of connections, and exclude those as well. Now we have a range of vertices that the first vertex has not already connected to, and that we are still allowed to connect to. Now we select a random number in the updated randrange list.

And so, we can add the first vertex and the second vertex to the graph of zeroes we made in the start with a value of -1. Currently the algorithm only produces output matrixes with positive integer values. If we wanted to include float number we would have to change the datatype of the array, which would make worse looking output data. And if we wanted to include negative numbers, we would have to set the value within the array to the highest value it can contain. During testing this would break the matrix, making it unable to function. This is presumably due to the highest values that could be assigned to the matrix is the datatype C long, which causes some overload if one inserts too many of them.

The next step is to add the two vertices as a tuple to the connectedNodes set. We then find the two vertices in the nodedict dictionary, and reduce their values by one. The loop now starts over from the top, until it is unable to put any variables into the randrange list, meaning all the vertices have now been properly connected.

The graph is now done. This means that the next step is to check whether or not the graph is connected. We must make sure that the graph is connected so that it has no isolated vertices and so that it is possible to perform find the MST/Steiner tree of the grapg. To check if the graph is connected, a separate function was made, which can be used to check the connectivity

of any matrix, not just this one. The function works as follows.

An empty node dictionary is made, then a nested for-loop is made from 0 to NodeCount in both the inner and outer loop and the indexes of the loop check the matrix to see if the corresponding edge has a value. If it does have a value, the index of the inner loop is added to a list in the node dictionary, and for the key in the outer loop. What this means is that every vertex has a list of its connected vertices.

We then assign a list named to "connected" to be the values within the first element of the node dictionary. We use this as a starting point to compare the other lists to. Next, a while-loop with no specific condition is started. We first increment a index variable, then we check the contents of the list in the dictionary that has the corresponding index. If one of the values within the list of the given key is in the "connected" list, then we extend the old list with the new one. We also set a lastentry variable, this is used to break the while-loop if no assignments have been made after a given set of rotations.

Next, the connected list is converted into a dictionary and then back into a list, this might seem unnecessary, but it prevents the size of the list from getting out of hand by removing duplicate elements, especially when the NodeCount becomes very high.

After this, we return False if the entire node dictionary has been gone through 2 entire times, meaning there is no possible way a connection could have been missed in the first iteration. We also return False if no connections have been made in one full loop. We then enter a for-loop from 0 to NodeCount, which checks whether every vertex has been connected or not, if they have then we can return True.

Now that we know whether or not all the nodes are connected, we come back to the rand_adjacency_matrix function. If we found that the matrix could not be connected, we start over from scratch by making the function call itself recursively. It tries this until it either finds a matrix that passes the check, or if 1000 recursive calls have been made, in which case we raise an error, and inform the user that the algorithm is unable to produce a proper result with the given variables and to try again with new ones.

If it passes the check, however, we can now assign values to all the vertices that we assigned to be -1. The reason why we do this here instead of further up in the function, is because it is a small optimisation in case we have to remake the algorithm multiple times. Next we set up a list of vertices, in which the elements are the string version of their vertex index. Then we randomly assign some of these vertices to be terminal vertices and that amount will be the input from the TerminalCount variable.

Lastly, we return the vertex list, graph, and terminal list in the form of the SampleGraph class.

Chapter 4

Results and discussion

4.0.1 Result Prim and Steiner

We were able to successfully translate the algorithms over from MATLAB into Python. To test this, we used the same sample input for both the MATLAB and the Python versions of the program, and we observed that they produced the exact same outputs.

4.0.2 Result Random Adjacency Matrix

We successfully were able to randomly generate new adjacency matrices using a set of input variables. We also observed breaking points in which the algorithm would not be able to generate any matrices. For example, the algorithm would not be able to generate an a graph if the number of nodes was 15, and the number of connections for each node was 1-2. In this case the algorithm would fail to produce a graph 1000 times in a row, and then throw an error. If however the amount of connections per node was increased to 1-3, it might fail a couple of times, but it would eventually find an algorithm with the required parameters to output. If this number was then increased to 2-4, it would produce a successful output every time.

```
Printing the edges that are scanned to make MST ...
Edge-1:  J-I    Wt: 5
Edge-2:  K-J    Wt: 6
Edge-3:  L-K    Wt: 6
Edge-4:  N-K    Wt: 8
Edge-5:  O-N    Wt: 4
Edge-6:  P-O    Wt: 4
Edge-7:  Q-K    Wt: 8
Edge-8:  M-L    Wt: 8
Total Weight : 49
     0     5     0     0     0     0     0     0     0
     5     0     6     0     0     0     0     0     0
     0     6     0     6     0     8     0     0     8
     0     0     6     0     8     0     0     0     0
     0     0     0     8     0     0     0     0     0
     0     0     8     0     0     0     4     0     0
     0     0     0     0     0     4     0     4     0
     0     0     0     0     0     0     4     0     0
     0     0     8     0     0     0     0     0     0

Printing the edges of Steiner tree ...
Edge-1:  I-J    Wt: 5
Edge-2:  J-K    Wt: 6
Edge-3:  K-L    Wt: 6
Edge-3:  K-N    Wt: 8
Edge-4:  L-M    Wt: 8
Total Weight : 33
     0     5     0     0     0     0     0     0     0
     5     0     6     0     0     0     0     0     0
     0     6     0     6     0     8     0     0     0
     0     0     6     0     8     0     0     0     0
     0     0     0     8     0     0     0     0     0
     0     0     8     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0
```

Figure 4.1: Output from the MATLAB version of the of the Prim and Steiner algorithms

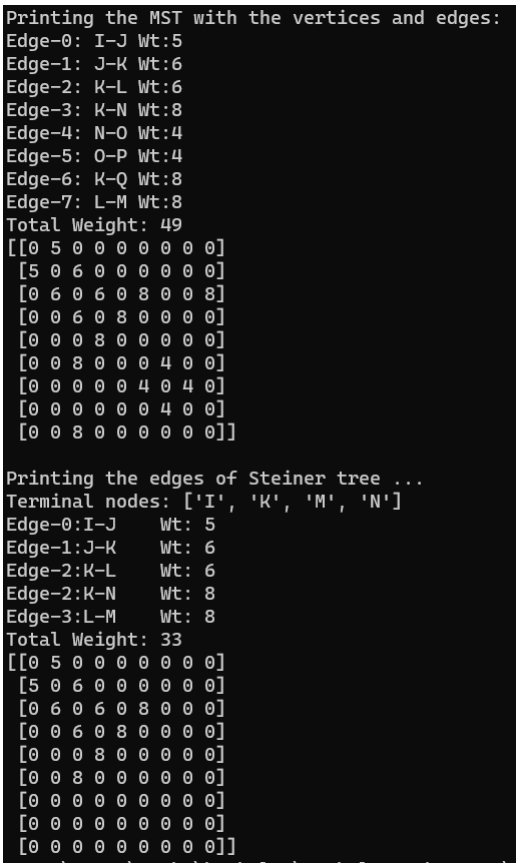

Figure 4.2: Output from the python version of the of the Prim and Steiner algorithms

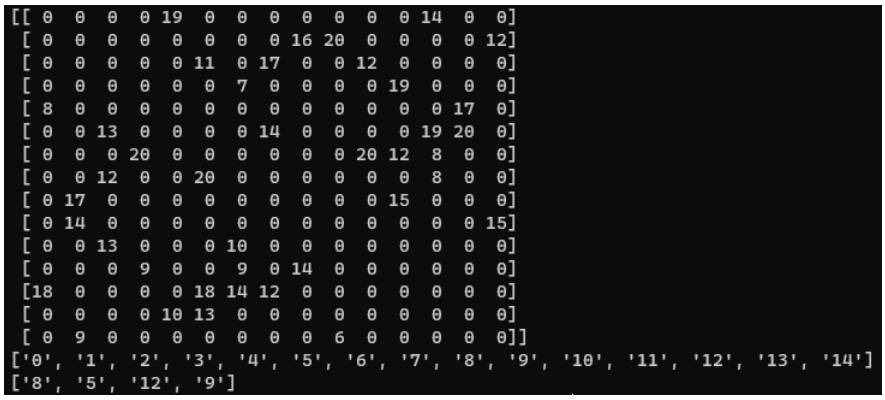

Figure 4.3: Output from the Random adjacency matrix algorithm with the parameters of Range(2,5) Weight(5,20),Nodecount=15, and TerminalCount=4

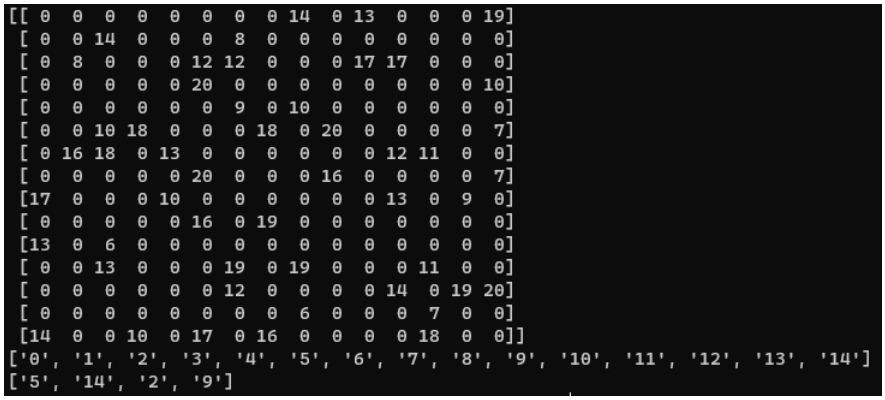

Figure 4.4: Different output using the same parameters

Chapter 5

Conclusion

To conclude, we successfully implemented Jaiswal & Davidrajuh's algorithm in Python language and tested it with many sample graphs. We also successfully made an algorithm for generating random adjacency matrices which created many sample graphs, and using these as inputs for Jaiswal & Davidrajuh's algorithm. This is about as far as we went with our work, primarily due to constructing the random adjacency matrix algorithm taking up a lot of time. We would have liked to also include data relating to the time complexity of the algorithms, as well as comparative performance with other algorithms for finding minimum weight spanning trees and Steiner trees. All in all, we are satisfied with the work that has been done, but we should have had better time management, as well as making better use of supervisor.

Bibliography

[1] R. Jaiswal and R. Davidrajuh, "Optimal design of wind farm collector system using a novel steiner spanning tree." https://davidrajuh.net/TEMP/NIK-2021.pdf, 2021. Accessed: 15.05.2022.

[2] R. Jaiswal and R. Davidrajuh, "A simple algorithm for finding steiner spanning trees." https://davidrajuh.net/TEMP/Simple_Algorithm.pdf, 2021. Accessed: 15.05.2022.

[3] R. Jaiswal and R. Davidrajuh, "Matlab implementation of the jaiswal-davidrajuh algorithm." https://davidrajuh.net/TEMP/New_Steiner_Algo.zip, 2021. Accessed: 15.05.2022.

Appendix A

Source Code

We used Git and GitHub to manage and store the source code for this project. All of the source code for this bachelor thesis project is on GitHub.

A.1 Source Code

https://github.com/MarkusFenneKarlsen/BachelorMarkusEnes

A.2 Compressed version of source code

https://github.com/MarkusFenneKarlsen/BachelorMarkusEnes/archive/refs/heads/main.zip