# Determining the Minimal Spanning Tree for a Network

*A dissertation submitted in partial fulfillment of the requirements for the degree of BACHELOR OF SCIENCE in Computer Science*

*in*

THE QUEEN'S UNIVERSITY OF BELFAST

*by*

*Dean* WELCH

April 27, 2018

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE**

**CSC3002 – COMPUTER SCIENCE PROJECT**

**Dissertation Cover Sheet**

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name: Dean Welch Student Number: 40109289

Project Title: Determining the Minimal Spanning Tree for a Network

Supervisor: Dr. Charles Gillan

## Declaration of Academic Integrity

Before signing the declaration below please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

**I declare that I have read both the University and the School of Electronics, Electrical Engineering and Computer Science guidelines on plagiarism - http://www.qub.ac.uk/schools/eeecs/Education/StudentStudyInformation/Plagiarism/ - and that the attached submission is my own original work. No part of it has been submitted for any other assignment and I have acknowledged in my notes and bibliography all written and electronic sources used.**

*Student's signature* Dean Welch *Date of submission 02/05/2018*

THE QUEEN’S UNIVERSITY OF BELFAST

# *Abstract*

EEECS

BACHELOR OF SCIENCE in Computer Science

**Determining the Minimal Spanning Tree for a Network**

by Dean WELCH

A network, in this case, is a series of interconnected nodes also known as a graph. Specifically we will be dealing with undirected weighted graphs where the edges have no direction and have a weight associated with them. The minimal spanning tree for a network or graph is the lowest weighted path required in order to connect every node in the network The final result of this project was a library written in C++ which could be incorporated into other projects to provide the functionality of determining the minimal spanning tree for a given network.

# *Acknowledgements*

I would like to take this opportunity thank my advisor for this project, Dr. Charles Gillan for their unique insight into the subject matter which greatly helped me in my work.

# Contents

# List of Figures

# List of Abbreviations

| | |
|---|---|
| **POC** | **P**roof **O**f **C**oncept |
| **RAM** | **R**andom **A**ccess **M**emory |
| **M(W)ST** | **M**inimum (**W**eight) **S**panning **T**ree |

# Chapter 1

# Introduction and Problem Specification

## 1.1 Background Information

### 1.1.1 Vertex/Node

A vertex, otherwise known as a node, is a fundamental component of a graph. [3]

### 1.1.2 Edge

An edge in a graph is a connection between two nodes within that graph. [4]

### 1.1.3 Connected Graphs

A connected graph is one wherein every node or vertex of the graph is connected such that any node can be reached from any other node simply by traversing the graph. If there are two nodes such that there is no path connecting these nodes then the graph is deemed to be unconnected. Only connected graphs have been considered here due to the nature of minimum spanning trees. [5]

### 1.1.4 Undirected Graphs

In an undirected graph the edges between nodes do not flow in a single direction. In other words it is possible to go from node X to node Y and node Y to node X along the same edge. This is opposed to a direction graph where it may be possible to travel from node X to node Y but not from node Y to node X. The most common algorithms for finding minimum spanning trees do so for undirected graphs which is the reason behind using them in this project. [6]

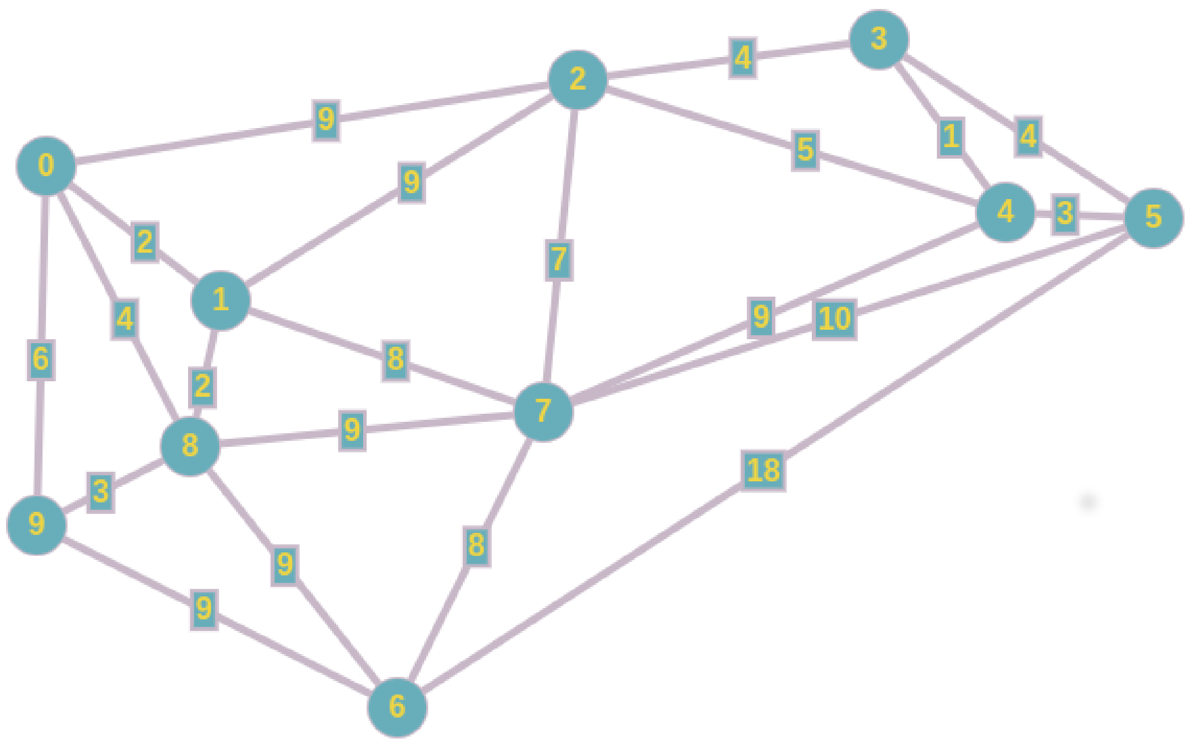


FIGURE 1.1: An Example of an undirected, weighted, connected graph.

### 1.1.5 Weighted Graphs

In order for a graph to be considered a weighted graph the edges connecting each of the nodes must have a number (weight) attached to it. This weight could represent any number of things such as length, cost, speed, etc. For minimum spanning trees weighted graphs are a necessity due to the fact that it is the weights that are used to determine minimum spanning trees. [7]

### 1.1.6 Minimum (Weight) Spanning Tree

A minimum (weight) spanning tree is a subset of all the edges in a graph where all nodes are connected such that the total weight of the edges is the lowest possible value. That is to say it must still be possible to reach every node from any other node in the graph whilst using the lowest weighted edges that do not introduce cycles into the tree. [8] See figure 1.2

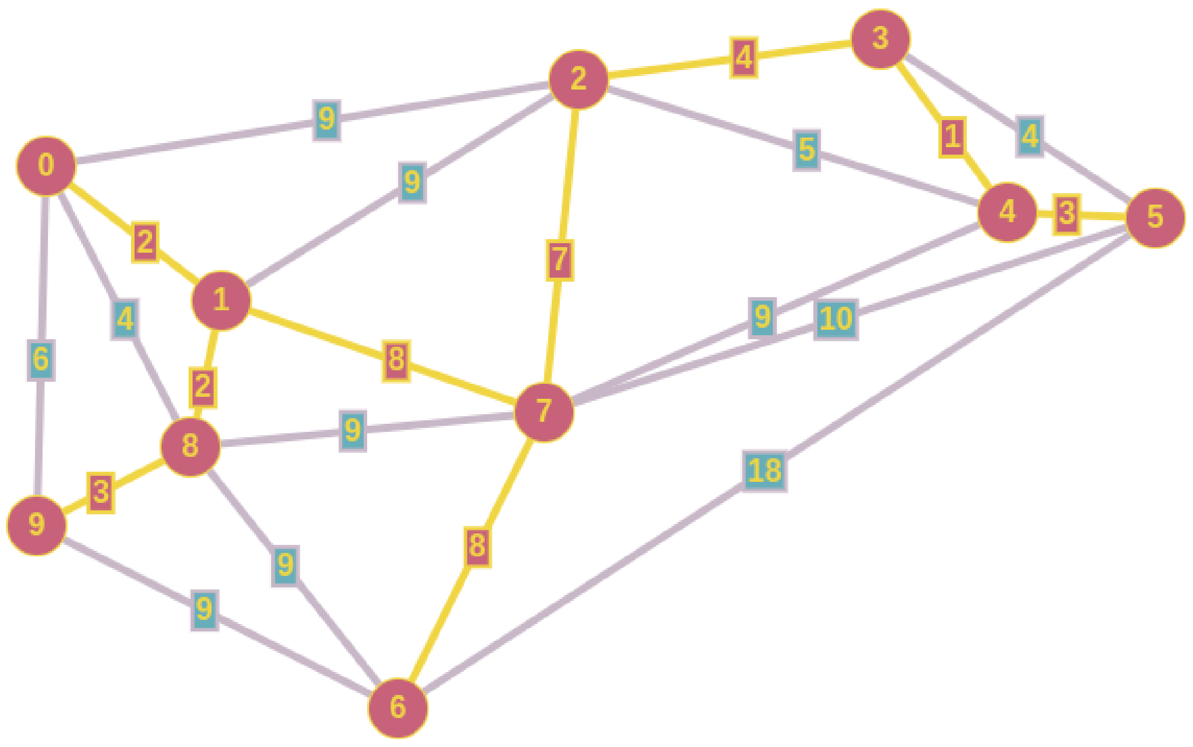


FIGURE 1.2: The highlighted MST of 1.1.

## 1.2 Context

Many applications use graphs as a main data structure to perform analysis on. These graphs can range from a computer network to a country's road infrastructure. In both these cases and in many others a minimum weight spanning tree can be incredibly useful for specific use cases. In the computer network example a minimum weight spanning tree or some variation thereof can be used to determine a single data path to follow between any two network nodes in order to eliminate transmission loops. In the case of a country's road infrastructure where the edges denote roads the weight could represent the cost of laying a sewer or telecommunications line. The minimum spanning tree would then connect all cities in the country together with the least amount of funding possible. This should clearly show the usefulness of minimum weight spanning trees and why it is an important problem to have solutions for.

## 1.3 Problem Description

The aim here is to determine the minimum weight spanning tree for a provided weighted undirected graph. In order to do this multiple decisions must be made first. Firstly a format or data structure which stores all necessary information about the graph must be created. The same is also true for how to display the end result, that being the minimum weight spanning tree. Additionally an algorithm for determining the minimum weight spanning tree must be evaluated and chosen. There are a range of algorithms which have been created which can determine the minimum weight spanning tree for a graph. Two of the more common algorithms have been expanded upon below as examples.

### 1.3.1 Prim's algorithm

Prim's algorithm is a greedy algorithm. This means that it will make the most optimal decision it can at each stage. [9] In a lot of cases greedy algorithms do not work to find the optimal solution when the full problem is taken into consideration but in the case of Prim's it always works to find the minimum weight spanning tree. Prim's algorithm begins by taking any starting vertex of the graph and connecting that to another vertex using the lowest weighted edge available to the initial vertex. Next, all the edges that are attached to one of the two connected nodes are considered and the lowest weighted one is chosen to connect the next vertex to the tree, taking care not to add a node to the tree that was added previously. This process of selecting the lowest weighted edge from the available set of currently connected nodes is continued until all nodes are connected to the tree. The result is a complete minimum spanning tree for the provided graph.

### 1.3.2 Kruskal's algorithm

Kruskal's algorithm, similarly to Prim's algorithm, is also greedy yet always ends up with the correct minimum weight spanning tree. Kruskal's algorithm works by taking the lowest weighted edge from the list of edges not currently in the tree and adding that edge to the tree. If adding an edge to the tree would result in a cycle that edge is discarded and the next lowest weighted edge is selected. This continues until every vertex in the graph has been included in the tree, this is the minimum weight spanning tree for the graph.

The reverse-delete algorithm is very similar to Kruskal's algorithm but in reverse. Instead of adding edges to a tree, you start off with a connected graph and remove the highest weighted edges possible without excluding a vertex from the final tree.

# Chapter 2

# System Requirements Specification

## 2.1 Requirements

- The system should be written in C++ (C++11) using Cmake (3.5).
- It must compile into a library able to be incorporated into other programs.
- It must run on Linux (Ideally on other major operating systems too).
- The library must provide a header file as an interface for all public functions and data structures.
- A data structure for the graph must be in the interface.
- A data structure to describe the edges in the minimum weight spanning tree also must be included in the interface.
- The interface must expose a function which can be called to determine the minimum weight spanning tree of the graph.
- The function exposed must also return a set of edges that describe the minimum weight spanning tree of the provided graph.
- The graph may be required in any format so long as it can accurately describe an undirected weighted graph.
- Unit tests are required for every function.
- The solver must be a class from which an object may be created to determine the minimum weight spanning tree for a single graph.

- Multiple instances of the solver may be created at one time to enable the ability to determine the minimum weight spanning tree for more than one graph simultaneously.
- Multiple methods/algorithms for determining the minimum weight spanning tree may be used provided there is an option to choose which method to use.
- There should be a proof of concept program provided which includes the library as a dependency and uses it to successfully determine the minimum weight spanning tree for a graph.

## 2.2 Assumptions

- The solver is only required to work for fully connected, undirected graphs with weights in an integer format.
- Any graph being provided to this system may be assumed to have no more nodes than the maximum integer used by C++ on a given 64 bit system.
- The maximum weight of any given edge may also be assumed to not exceed the maximum value stored by an integer in C++ on a given 64 bit system.

# Chapter 3

# Design

## 3.1 Architectural Description

This system was designed to be a library capable of being imported into a wide variety of applications that have a use case for finding the minimum weight spanning tree for a graph. In order to accomplish this interfaces in the form of headers were used as a means of abstraction to separate the logic of determining the minimum spanning tree from the calling program. Therefore it is intended that any program wishing to include will only have access to a single header file and the library file itself.

Within the program itself all .cpp files have an associated header which again is used as a method of abstraction only this time within the library itself. The header files are imported in whichever files needs access to the associated functions rather than importing the .cpp files itself.

## 3.2 Software System Design

In the library which has been created there are two core elements which provide the functionality to determine the minimum weight spanning tree of a graph. There are also additional C++ structs (A struct is simply a custom data structure used to store multiple members, possibly of varying data types, in order to keep related values packaged together in a convenient way) used to store relevant data in order to power the implemented algorithms.

There are two structs available in the "PrimSolver.h" header file. This file is intended as an interface for code external to the library to be aware of the necessary data structures and functions available to use in the program incorporating this library. The two available structs are the "CSRGraph" and "Edge" structs. The "CSRGraph" struct was created as a compact method of representing a connected undirected weighted graph. The CSR in the name stands for "Compact Sparse Rows" which just indicates the method of storing which of the vertices are connected to which other vertices without requiring the need for a pair of vertices for edge connection. The "Edge" struct was designed in order to provide a means of representing the final minimum weight spanning tree. An "Edge" struct provides the ability to show a link between two vertices in the graph, using an array of these it's then possible to have a complete description of the minimum weight spanning tree.

Also contained within the "PrimSolver.h" file is the "PrimSolver" class definition. This class definition provides all the information necessary for an external caller to be able to instantiate an instance of the class. The interface tells the external caller about variables stored within the object that are public and can be accessed along with a public function definition that can be called. Doing this provides a way of giving access to only the relevant components (from the reference point of someone using this library in their own program) of the solver. Since anyone using this library in their own program doesn't need to know the specifics of how the minimum weight spanning tree for the graph they've provided is determined all the interface provides them with is the ability to call the constructor of the class, the function to determine the minimum weight spanning tree (including its return type) and access to the graph that was provided in the constructor for which the minimum weight spanning tree is to be determined for. This results in other details being hidden as they are not necessary outside of the library itself.

The main logic for determining the minimum weight spanning tree for a given graph is contained within the "PrimSolver.cpp" file. It is here that the interface for the "PrimSolver" class is implemented. This class has been created such that any number can be instantiated without causing interference with other instances of the class. This is done by instantiating entirely new sets of variables in each object created rather than sharing any of these variables between instances. This means that with multiple non-interfering instances it's possible to run the algorithm multiple times simultaneously on different graphs.

Headers are also used as interfaces within the library itself as opposed to just being used as a way to access the library functions external from the library itself. An example can be seen with the "PrimSolver.cpp" file including the "FibonacciHeap.h" header file. The reason for doing this is much the same as previously. Using interfaces adds a layer of abstraction so the caller of the function does not need to have any knowledge of how the code it is calling works, it is provided all it needs by the interface it has access to without also needing the implementation.

# Chapter 4

# Implementation and Testing

## 4.1 Language/Development Environment

The system was developed on Linux, specifically Ubuntu 16.04 64-bit. The development language chosen was C++, the version used was C++11. GCC version 5.4.0 was used to compile the code and CMake version 3.5 was used as the build system.

C++ was the chosen development language for a number of reasons. The most obvious reason for its choice was speed. Since C++ is not a memory managed language like Java it moves a lot of that work that would usually be taken care of by the garbage collector to the developer, but the lack of garbage collector also results in a noticeable difference in execution time. So although it increases the development workload the trade off was deemed to be worth it as graphs could get to be very large in size to where a speed increase would mean shaving a large amount of time off of the execution which is doubly important in a speed critical application. Another reason for choosing C++ is its versatility. A C++ library can of course be included in any other C++ project but on top of that it can also be included in a C project or even in completely unrelated languages such as Python. Therefore choosing to do it in a language like C++ allowed for the library to have the potential to reach a much wider variety of applications including ones written in a different language.

Cmake was chosen as the build system because it works across multiple platforms without the need to make a separate makefile for each system you are building the project on.

The GCC (GNU Compiler Collection) was used as it is the standard free and open source C++ compiler available on Linux.

## 4.2 Libraries

No libraries were necessary to create the system however for testing purposes a testing framework library was included. Google Test [10] was the library chosen as the testing framework used to aid in the creation of unit tests. The suite was chosen for many reasons, importantly though it is available on a wide range of systems including Linux, Mac OS X and Windows. Additionally it makes the testing process a lot less painful with the availability of its test discovery feature and its wide range of included assertions.

## 4.3 Key Implementation Decisions

### 4.3.1 The Algorithm

The first key decision to be made was which algorithm to use for determining the minimum weight spanning tree for a graph. There are multiple established algorithms available to choose from, for example:

- Borůvka's algorithm
- Prim's algorithm
- Kruskal's algorithm
- Reverse-Delete algorithm

The Reverse-Delete algorithm is the reverse of Kruskal's algorithm and not very commonly used so it was decided to not use that one. The remaining algorithms all had the same time complexity of O(m log n) (exception being Prim's algorithm which can have a better time complexity of O(m + n log n) provided it can use a specific data structure). It was deemed that Borůvka's algorithm was less intuitive and more difficult to implement than either Prim's or Kruskal's algorithms, so it too was discounted. Basic versions of Prim's and Kruskal's algorithms were implemented in Java in an effort to better understand the algorithms to help determine which would be better suited for this system. It was decided that although both algorithms were suitable for the application and performed well that since Prim's algorithm had the ability to run at the better time complexity of O(m + n log n) it would be the initial algorithm chosen for this system.

**Algorithm Data Structure**

There are multiple data structures available that could be used to power Prim's algorithm. It's possible to implement Prim's algorithm without any special data structure and just iterating through all of the edge weights until you find the lowest weighted suitable edge but this obviously is inefficient as you have to check every edge. Instead a priority queue could be implemented. A priority queue is like a regular queue except that each entry in the queue has a priority associated with it with items of a higher priority being served before lower priority items. This provides a much better runtime than searching through each edge, meaning using a priority queue was the obvious choice. A priority queue is often implemented using a binary heap, this gives Prim's algorithm the previously mentioned runtime complexity of O(m log n). There exists another form of implementing priority queues with a Fibonacci heap. A Fibonacci heap reduces this runtime to O(m + n log n) where m is the number of edges and n the number of nodes, but has a much higher overhead than a standard binary heap. The Fibonacci heap is therefore only used in very specialised cases, one of those cases being in Prim's algorithm due to the relatively large number of priority changes of elements in the queue. Therefore it made sense to implement the priority queue using a Fibonacci heap in this case.

### 4.3.2 Graph Format

Consideration was also given to the format which would be used to represent the graph. A typical way of storing a graph would be to use a matrix. Essentially a 2D array would be needed where the indices would denote the nodes and the value at that position would indicate the weight that edge had with 0 indicating there was no connection between the two nodes. This method works but leaves a large memory footprint as you are storing unnecessary information such as where there are no edges which is not needed. Another better alternative would be to use a coordinate format. This is where the the two nodes and the weight of the edge would be stored for each edge in the graph. This removed the extraneous storage of connections that weren't actually included in the graph. The coordinate format however was not ideal as you still had each node possibly being stored multiple times. This lead to choosing a more efficient mechanism of an adjacency matrix, specifically using compressed sparse rows. With an adjacency matrix the index of an array is used to denote the vertices with the values in the array being the index of another array where you can find the other vertex in the pair completing the edge. Compressed sparse rows merely means that it is the rows of the matrix that are compressed rather than in compressed sparse columns where the columns are compressed, but in this case it makes no difference which method is chosen as undirected graphs are perfectly symmetrical in this regard.

## 4.4 Important Functions and Algorithms

### 4.4.1 Fibonacci Heap

A Fibonacci heap is the data structure we've chosen to implement a priority queue in order to power Prim's algorithm. The main benefit of a Fibonacci heap over other heaps (such as binary or binomial heaps) is that a number of important functions run in a constant amortised time O(1). This includes the insert, find minimum and decrease key functions. In comparison an insert on a binary heap in the worst case is O(log n) as is the decrease key function, although in both heaps the delete function also takes O(log n). In the case of an algorithm (such as Prim's) where there may be a large number of decrease key operations it makes sense to use a Fibonacci heap due to the lesser time complexity of the operation.

The Fibonacci heap implements a relatively clever method of getting these constant time operations. The heap itself is comprised of a list of trees, in this case implemented via a doubly linked list of the nodes at the root of each tree. The trees are then implemented as a root node which contains a reference to a child node (if applicable), these child nodes are also part of a doubly linked list for each node's children. Each of these trees in turn follows a rule where the children of every parent node each have a key that is of greater or equal value to the parent node's key thus ensuring that the minimum node is always at the root of one of these trees. A reference to the node with the lowest key is kept, this is how the minimum node is acquired in constant time. Since the only important factor is that child nodes have a greater key value than their parents this provides much greater flexibility to Fibonacci heaps than other heaps. This flexibility essentially allows for some heap operations to have some delayed effects. For example when inserting into a binary heap the heap needs to sort the node into the appropriate position immediately, however in a Fibonacci heap the node is inserted as a tree with a single node to the list of trees (updating the reference to the node with the lowest valued key if appropriate) and then it is only moved in the structure to a more appropriate position (to maintain the improved runtime) later on when for example a delete key operation is performed. This lazy execution allows for structural changes to the tree to stack up until they can be executed in bulk. [11, pg. 505-522]

The core functions required to use a Fibonacci heap are further described in more detail below:

### Insert Function

The insert function on a Fibonacci heap is relatively simple. The purpose of this function is to take a new node not currently contained within the heap and adding it to the heap in a manner that complies with the structure of a Fibonacci heap. The node that is being inserted is made into a tree containing only itself. This tree is then added to the list of trees in the heap. If the node has a lower key value than the current minimum node then the reference to the minimum node is altered to the newly inserted node.

### Extract Min Function

The Extract Min function is just a special case of the Delete Min function where the minimum node is returned as the result of the function as well as being removed from the heap itself. This is one of the more complex operations you can perform on a Fibonacci heap as it results in a reshuffling of the heap's structure in most cases. Initially a reference to the current minimum node is stored so it can be returned later then the minimum node is removed. The children of the minimum node are then each added to the list of root nodes as their own individual trees. The next step is to reduce the number of root nodes in an effort to have fewer nodes to search through in order to find the minimum node. To do this we find two root nodes of the same degree (degree being the number of child nodes a parent node has). Once we have two nodes of the same degree the node with a greater key value is made into the child node of the node with the lesser key value thus keeping the pattern of parent nodes having a lesser key value than their children, this also increases the parents degree by 1. This is repeated until all root nodes have unique degrees. With the now reduced number of root nodes they may now be searched through in an effort to find the node with the lowest key value which is not set to be the new minimum node.

### Decrease Key Function

The aim of the decrease key function is to take a node and set it's key value to some new lower value. Once the key value of the node has been decreased a check needs to be made in order to determine whether or not the new lower value has made this child node's key value lower than that of the parent node. If the key value is still greater than the parent node's key value then no further action needs to be taken. However, if the new key value is lower than that of the parent node's key value then the child node gets cut from the parent node and becomes the root node in a tree containing only itself and added to the list of trees. The parent is then marked (a marked node is one which as had a child node cut from it) and if it has been marked previously it is cut, becomes it's own tree in the tree list and it's parent is marked and so on until a node is reached that is either a root node or has not yet been marked. All nodes that become their own trees are also all set to be unmarked. Finally a check is made to see if this node's new key value qualifies it as being the new minimum node, if so the minimum node reference is updated.

### 4.4.2 Prim's Algorithm

Each node contained within the graph initially needs to be associated with a variable that will denote the lowest weighted edge currently connecting this node to the tree, this to begin with should be the largest number available to you (positive infinity or at the very least a number larger than the greatest weight in your graph). The reason we do this is to ensure that later when the lowest weight so far for that node is discovered there is no chance that the value already stored is less than the largest possible value available for the weight connecting a node to the graph. A corresponding variable is needed to denote which edge connects that node to the graph with the lowest weight, initialised for all nodes to a special value which indicates that there is not yet an edge connecting the node to the tree yet.

Next, an essentially random node is selected and added to the graph. For each edge that involves this node we must then set the lowest cost of the connecting node to the edge weight and set the node that corresponds to that edge to the node we selected at random. This provides us with the the initial starting position of our algorithm.

Select the next node with the lowest cost of addition to the graph, check that the node has not already been added, if it has do nothing otherwise add it to your graphs then for each edge attached to that node update the values of the cost of addition for the nodes connected by those edges if the weight of the edge is lower than the current lowest cost of adding that particular node. Ensure the node you've added is marked as added to the graph.

Once all nodes have been attached to the graph, by repeating the previous steps, you should now have a complete minimum weight spanning tree for the original graph provided. [11, pg. 634-636]

### 4.4.3 Kruskal's Algorithm

After the main functionality of the system had been completed using Prim's algorithm Kruskal's algorithm was next in line to be implemented into the system.

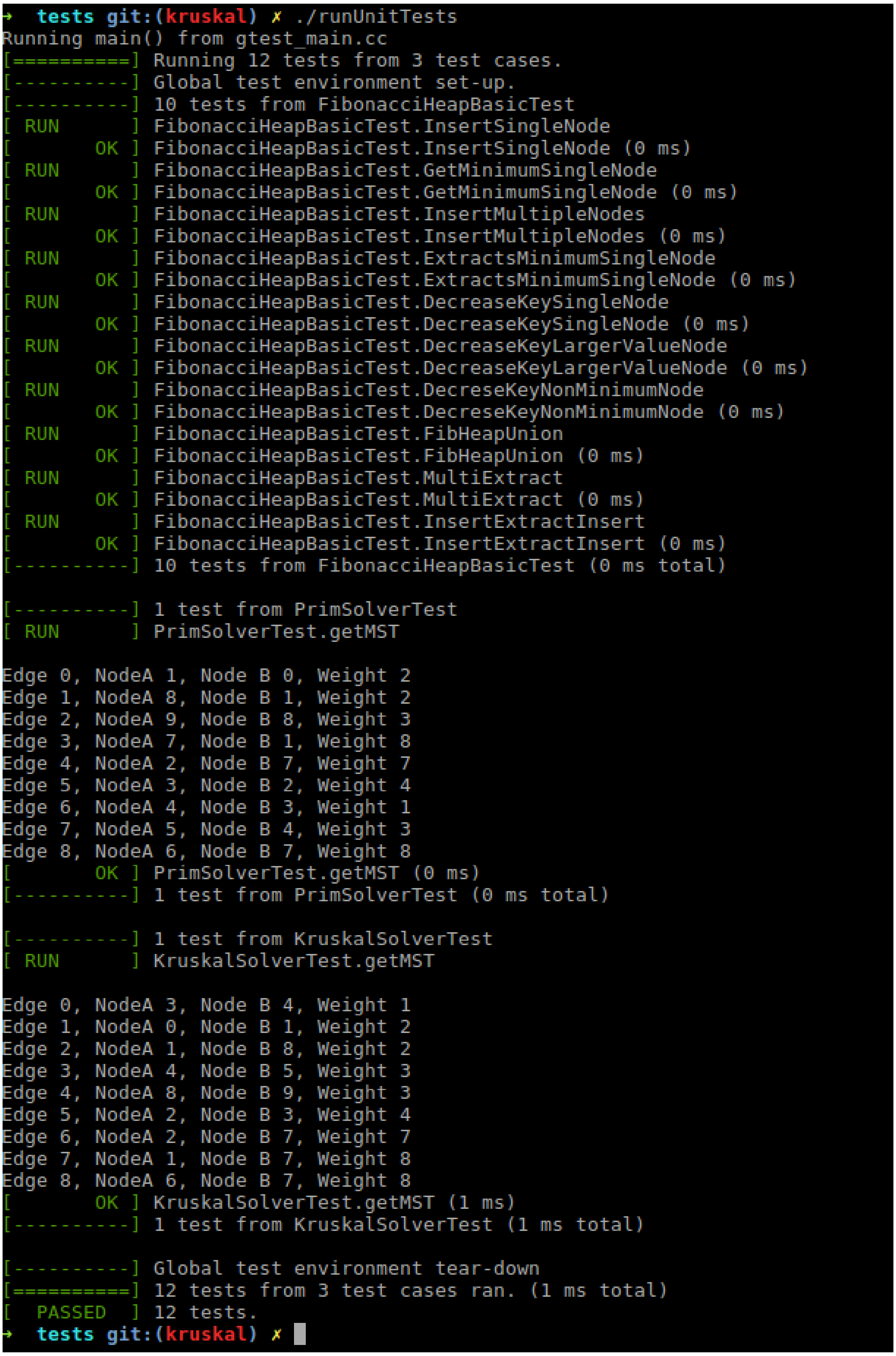
To begin with all the edges in the graph are added to the priority queue (in this case implemented as a Fibonacci heap). This provides us with the ability to select the edge in the graph with the next lowest weight.

The first edge is selected and added to the current representation of the MST. A note is made of the two nodes now currently in the tree. The next edge is then selected and a check is made. This check looks to see if the two nodes that make up this edge are already connected in the tree. If they are not the edge is added, otherwise this means that adding this edge would introduce a cycle to the tree so the edge is skipped over and the process is repeated until all nodes are connected. [11, pg. 631-633]

## 4.5 Test Approach

Testing was carried out throughout the entire process of creating this system. During the coding phase unit tests were added for functions as the functions themselves were completed, re-running all previous tests in order to check for regressions that may have been introduced by the new feature. Manual testing was also done periodically where the code was run on some sample data and the output verified providing no issues terminated the system early. With the use of Google Test the creation of unit tests was relatively quick and simple, with the included test discovery allowing for easy automation of the tests. The assertions provided by Google Test were used in order to assert that both the pre and post-conditions of the tests were to be as expected. The "SetUp" function in every test class was also made use of as a measure to set up each test identically therefore having reproducible results. Tests were also written in such a fashion to not be affected by any other tests in the suite, meaning that tests can be run in any order without the possibility of the results being affected.

The system has also been tested as a whole. To do this, the system was packaged up into a single library file and along with the appropriate header file for use as an interface was included in another separate project. This was done in an effort to prove that the system as a whole would work when incorporated into another project and also as an example for how someone may use this library.



```
➜  tests git:(kruskal) ✗ ./runUnitTests
Running main() from gtest_main.cc
[==========] Running 12 tests from 3 test cases.
[----------] Global test environment set-up.
[----------] 10 tests from FibonacciHeapBasicTest
[ RUN      ] FibonacciHeapBasicTest.InsertSingleNode
[       OK ] FibonacciHeapBasicTest.InsertSingleNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.GetMinimumSingleNode
[       OK ] FibonacciHeapBasicTest.GetMinimumSingleNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.InsertMultipleNodes
[       OK ] FibonacciHeapBasicTest.InsertMultipleNodes (0 ms)
[ RUN      ] FibonacciHeapBasicTest.ExtractsMinimumSingleNode
[       OK ] FibonacciHeapBasicTest.ExtractsMinimumSingleNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.DecreaseKeySingleNode
[       OK ] FibonacciHeapBasicTest.DecreaseKeySingleNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.DecreaseKeyLargerValueNode
[       OK ] FibonacciHeapBasicTest.DecreaseKeyLargerValueNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.DecreseKeyNonMinimumNode
[       OK ] FibonacciHeapBasicTest.DecreseKeyNonMinimumNode (0 ms)
[ RUN      ] FibonacciHeapBasicTest.FibHeapUnion
[       OK ] FibonacciHeapBasicTest.FibHeapUnion (0 ms)
[ RUN      ] FibonacciHeapBasicTest.MultiExtract
[       OK ] FibonacciHeapBasicTest.MultiExtract (0 ms)
[ RUN      ] FibonacciHeapBasicTest.InsertExtractInsert
[       OK ] FibonacciHeapBasicTest.InsertExtractInsert (0 ms)
[----------] 10 tests from FibonacciHeapBasicTest (0 ms total)

[----------] 1 test from PrimSolverTest
[ RUN      ] PrimSolverTest.getMST

Edge 0, NodeA 1, Node B 0, Weight 2
Edge 1, NodeA 8, Node B 1, Weight 2
Edge 2, NodeA 9, Node B 8, Weight 3
Edge 3, NodeA 7, Node B 1, Weight 8
Edge 4, NodeA 2, Node B 7, Weight 7
Edge 5, NodeA 3, Node B 2, Weight 4
Edge 6, NodeA 4, Node B 3, Weight 1
Edge 7, NodeA 5, Node B 4, Weight 3
Edge 8, NodeA 6, Node B 7, Weight 8
[       OK ] PrimSolverTest.getMST (0 ms)
[----------] 1 test from PrimSolverTest (0 ms total)

[----------] 1 test from KruskalSolverTest
[ RUN      ] KruskalSolverTest.getMST

Edge 0, NodeA 3, Node B 4, Weight 1
Edge 1, NodeA 0, Node B 1, Weight 2
Edge 2, NodeA 1, Node B 8, Weight 2
Edge 3, NodeA 4, Node B 5, Weight 3
Edge 4, NodeA 8, Node B 9, Weight 3
Edge 5, NodeA 2, Node B 3, Weight 4
Edge 6, NodeA 2, Node B 7, Weight 7
Edge 7, NodeA 1, Node B 7, Weight 8
Edge 8, NodeA 6, Node B 7, Weight 8
[       OK ] KruskalSolverTest.getMST (1 ms)
[----------] 1 test from KruskalSolverTest (1 ms total)

[----------] Global test environment tear-down
[==========] 12 tests from 3 test cases ran. (1 ms total)
[  PASSED  ] 12 tests.
➜  tests git:(kruskal) ✗ 
```

FIGURE 4.1: Evidence of tests running and their output.

# Chapter 5

# System Evaluation

## 5.1 Development Timeline

The whole system was in development from October 2017 to April 2018. Development consisted of 9 main stages outlined below.

### 5.1.1 Java POC

Initially a naive implementation of both Prim's and Kruskal's algorithms were implemented in Java in an effort to better understand the problem and two of the classic minimum weighted spanning tree algorithms. This version was not very versatile and had many hardcoded components to it. This version of the algorithms was used simply as a reference upon which to build the C++ version which was to be much more flexible, especially in terms of the size of graph it could handle.

### 5.1.2 Project Initialisation

To begin with the basic structure of the project needed to be created. This consisted of a folder structure intended to separate tests from the main source code along with the relevant CMake files required in order to compile and link the source code into a shared library which is required for the purposes of embedding the system into another project.

### 5.1.3 Fibonacci Heap

The next step in the development cycle was to fully implement a priority queue in the form of a Fibonacci heap. Both Prim's and Kruskal's algorithms benefit greatly from the use of a priority queue. This was not part of the development done is Java since Java provided access to a priority queue as a part of its standard library. The full process of creating the Fibonacci heap, including creating the appropriate data structures required as components of the heap, took about 6 weeks. Initial development here was slow to begin with mostly due to the lack of C++ or other low level language experience. Issues with pass by reference vs pass by value were somewhat common, along with general inexperience in dealing with pointers often leading to confusing errors adding large amounts of debug time.

### 5.1.4 Fibonacci Heap Unit Tests

The tests for the Fibonacci heap also took the same amount of development time (6 weeks) as the heap itself due to the tests being written essentially in parallel to the actual heap. Once a function of the heap was determined to be completed a test would be written in an effort to assert the correctness of the function. Often the test would initially show the function to segmentation fault, return garbage data or simply complete with the wrong answer/leave the heap in a bad state. This led to a back and forth of debugging and subsequently fixing the function in the heap and updating/adding the tests for that function as new issues appeared during the development process. Eventually the tests all passed whilst asserting the correct end result from each of the functions, this does not however mean that the Fibonacci heap is bug/error free as the tests can only prove the presence of a bug not the absence of any.

### 5.1.5 Prim's Algorithm

Once the Fibonacci heap had been implemented and tested it was time to begin work on Prim's algorithm. The task of implementing Prim's algorithm in C++ was made partially easier due to the previous experience and understanding of the algorithm gained by first producing it in Java. The use of C++ here again caused several issues similar to those discussed previously whilst working on the creation of the Fibonacci heap. However, at this point with more experience in the language and possibly more importantly in debugging the more common issues experience with C++ development, the time taken to find and fix any issues seemed to have improve. During the development of this phase several issues came up with regards to the Fibonacci heap that was integral to the functioning of Prim's algorithm. As each issue was discovered fixes were made to the Fibonacci heap along with the relevant unit tests. This process ended up taking about three weeks for the bulk of the work with small tweaks being made after the fact.

### 5.1.6 Prim's Algorithm Unit Tests

Once the algorithm had been completed the focus moved on to unit testing. Since the implementation here only involved a single function unit tests were not created as development progressed. the reason for this being the unlikelihood that a unit test created on a partially completed function would be of much use in the future and would likely need to be replaced. The testing process did turn up more of the same type of issues that were previously discovered in past unit tests. All discovered issues were fixed until the tests passed showing that for the sample data the correct minimum weight spanning tree could in fact be determined.

### 5.1.7 Library Creation and Embedding

With the core functionality now in place it was time to test that the system could be packaged successfully as a library and used as a 3rd party source in another project. To do this a release version of the library was built, this created the library file required by other programs to have access to the functionality provided by this system. After this another basic project was created which added the library file as a dependency in the CMake file as well as pointing to header file so the project had access to the necessary interfaces required to call the library functions. From this point it was trivial to call the library functions and attempt to compile the project ensuring that the linking of the library file was successful. The expand further on this project with the library embedded parsers were also written which would read in a file containing a particular format of storing a graph and run this through the library in an effort to produce the minimum weight spanning tree, as an example for how someone might use the system created. A further addition to this project was made once Kruskal's algorithm had become available in the library.

### 5.1.8 Kruskal's Algorithm

The process for implementing Kruskal's algorithm was very similar to that of Prim's algorithm. Using the Fibonacci heap with Kruskal's algorithm went a lot smoother likely to the fact that most issues would have been sorted out when implementing Prim's algorithm and that Kruskal's algorithm had a much more basic interaction with the heap. Since fewer issues were encountered during this process the time went was much reduced from Prim's taking roughly 2 weeks to complete.

### 5.1.9 Kruskal's Algorithm Unit Tests

A similar situation occurred here as with the unit tests for Prim's algorithm. With Kruskal's algorithm following the same pattern as Prim's with near identical interfaces there was once again only a single function to test. The tests for Kruskal's algorithm were also able to be made very similar to the Prim's tests due to the similarity in interfaces and the fact that both should produce identical minimum weight spanning trees for a graph (any graph with unique edge weights will always have a single minimum weight spanning tree).

## 5.2 System Capabilities

At the completion of this project the system is capable of:

- Being embedded in another program.
- Using Prim's algorithm to determine the minimum weight spanning tree of a graph.
- Using Kruskal's algorithm to determine the minimum weight spanning tree of a graph.
- Passing a full test suite.
- Running on Linux.

### 5.2.1 Trade Offs

To limit the scope of the project to a degree some compromises or trade offs had to be made.

One of the major trades offs made was limiting the weight of the edges of the graph to be integers only, many graphs use decimal values to represent the edge weights. This was done a measure of reducing the complexity of the code, coding time and the actual runtime of the system. Allowing multiple types to be used as the weight would likely require either templates or generics, along with integer processing generally being quicker than processing floating point numbers this was deemed to be a nice feature but unnecessary. If there was a graph with edge weights in decimal format it should be relatively simple for the program using this system to convert the decimal values into an integer in order to be able to use the system successfully.

Another feature which is lacking from this system is the ability to check the validity of graphs being passed into it. For example there is no check made to see if the graph is fully connected or not. A process for a check of this nature would involve generating the Laplacian matrix of the graph and analysing the eigenvectors from that matrix in order to determine the number of disconnected nodes. This was out of scope for the project and likely an unnecessary check in most cases wasting computational resources as the caller likely already knows whether or not their graph is fully connected and otherwise should be able to perform the check themselves before using the library.

### 5.2.2 Scaling

There are limits on how large a graph this system can process. The limits can vary depending on how the library was compiled and what the specification (namely RAM) of the machine it's running on is. The number of nodes will be limited to the maximum integer value allowed, the minimum this can be is 32767 with higher values possible depending on the system. The same holds true for the maximum allowed weight of an edge. The other possible limiting factor is the amount of RAM available on the system. With a large enough graph the memory footprint could easily use up multiple Gigabytes of a machines RAM, if the graph requires more RAM than is available then it is likely the program will crash. Other than the limits mentioned previously there should be no other factors that would affect the system provided you give it enough time to execute as the runtime does increase the more nodes and edges the graph contains.

# Chapter 6

# Conclusion

## 6.1 General Summary

The main goal of this project, which was to determine the minimum weight spanning tree for a graph, ended up being achieved. Not only was this accomplished but two different algorithms were successfully implemented, each independently capable of fulfilling that criteria.

In addition, many other criteria were able to be incorporated into the project:

- It has the ability to compile into a shared library.
- A POC program was created as an example for how to embed the library in your program and use it.
- An interface in the form of a header file is available for all reasons mentioned in the specification.
- Unit tests have been created for each function.
- The system runs on Linux.

## 6.2 Software Environment Evaluation

The language chosen (C++) turned out to have somewhat of a steep learning curve leading to increased development and testing time than was anticipated. Although this led to a larger than expected number of issues with the code these could be fixed as they were discovered whilst keeping the performance benefit of C++, this is opposed to choosing another language such as Java or Python which may provide a quicker more streamlined development process would likely be much slower than C++. Thus overall C++ was an appropriate language for the development of this system which may greatly benefit from the performance provided.

Using CMake as the build tool for this project had several benefits. It removed the need to manually compile and link the project files, it provided a simple way to include the latest version of the library used (Google Test) and it produced makefiles. This meant CMake had the effect of making the build process of the project a lot simpler and easier to understand.

## 6.3 Development Process

Beginning the process by developing a Java version of the solution proved to be a very valuable resource. Having this available provided a very nice reference for how to implement the algorithms rather than having to rely on pseudo code or a 3rd party implementation and porting those. The reduced development time for Java compared to C++ meant that there was little impact on time available for the project, not to mention the time spent was easily made up for in the reduction in time it meant for creating the C++ version.

In contrast however, the decision to create a custom Fibonacci heap ended up costing a lot of development time. A simpler priority queue would have been adequate for the purposes of this project or simply using a 3rd party library as the implementation of a priority queue was not set in the requirements.

## 6.4 Further Work

The system achieved all of it's major requirements but there is definitely still room for some further improvements to be made that could increase the quality of the system.

- Provide the capacity to have edge weights in a non-integer form.

  Having the capability to process graphs with decimal edge weights would increase the ease of use of the system. The need for a 3rd party developer to convert their edge weights into integers for compatibility with this system would be removed.

- The addition of a concurrent algorithm.

  If a concurrent algorithm could be implemented it would provide any users with the option to speed up the processing of their graph into a minimum weight spanning tree, providing they are running on a multi-core machine.

- Check the input graph is fully connected.

  Currently there is no process in the system which can check the validity of a graph. Adding an optional check (this process would be unnecessary if the user is aware of the connectedness of their graph) would make users aware of why their graph is not suitable for this application.

- Expand algorithms to determine the minimal spanning forest.

  If a graph has more than one group of fully connected nodes there is no single minimal weighted spanning tree. Instead what needs to be found is the minimal spanning forest which comprises of two or more minimum weight spanning trees. Prim's and Kruskal's algorithm are capable of doing this with some alteration.

# Appendix A

# How To Use The System

## A.1 Library Compilation

The library my be retrieved from here: https://gitlab.eeecs.qub.ac.uk/Dwelch/CSC3002-MST-Library [1]

To compile the system the minimum CMake version required is 3.5 and a C++11 compliant C compiler such as GCC version 5.4.0. The recommended Make version is GNU Make 4.1 as that is what the system has been built with during development.

To use CMake to prepare the system simply open up a terminal and run the command (internet access is required to download the Google Test framework):

```
cmake "path/to/project/folder"
```

This will generate everything needed to build the project in your current working directory.

Once the previous step has completed the project can be compiled. Using the same terminal you have open, in the same directory you ran the previous command run:

```
make
```

This uses the MakeFile generated by CMake to compile and link all necessary files in order to create the library file "libSolvingLibrary.a" which can now be found in the "src" directory within your current working directory.

## A.2 Running the Unit Tests

To run all the unit tests for the library open up a terminal window and navigate to the directory where you previously ran the CMake and Make commands. From here navigate to the "tests" directory and run the following command:

```
./runUnitTests
```

This will run through each unit test individually showing the results of the tests and a final summary indicating any successful or failed tests.

## A.3 Incorporating the Library into Another Project

An example of this being done can be found here: https://gitlab.eeecs.qub.ac.uk/Dwelch/MST-Program-Example [2]

Before compilation of this code some changes need made in order to link properly with the library you have previously compiled. Once you have downloaded the code you must edit the file found in

```
"<downloadLocation>/SolvingProgram/src"
```

called "CMakeLists.txt". Here you will find two lines that look like this:

```
set(LibraryBuildSrcLocation "/path/to/cmake/build/folder/src")
set(LibraryHeaderFolder "path/to/folder/containing/headers/src")
```

The part in quotes are the file paths for the two "src" folders needed. The first is the folder that contains the library file "libSolvingLibrary.a", located under the "src" directory where you ran the cmake command for the library. The other is the location of the "src" folder which contains the headers required to interface with the library. This can found at

```
"<downloadLocation>/SolvingProgram/src"
```

Once this has been done the code can be compiled. Compiling the example code follows the exact same instructions as for the library A.1. In order to run the project open up a terminal and navigate to your CMake build directory. From here navigate to the "src" folder and run the following command:

```
./SolvingProgram "path/to/matrix/file"
```

This will run the program with the parameter being the path to a graph stored in a coordinate format matrix market file. Examples of these are included with the code.

# Appendix B

# Meeting Minutes

QUEEN'S UNIVERSITY OF BELFAST
COMPUTER SCIENCE

## Minute of Project Supervision Meeting

| | | | |
|---|---|---|---|
| **Student Name:** | **40109289** | | |
| **Project Module Code:** | **CSC3002** | | |
| **Project Supervisor:** | **Dr. Charles Gillan** | | |
| **Meeting Number:** | **1** | **Date of Meeting:** | **11/10/2017** |

### Progress since last meeting, and decisions arrived at during meeting:

Progress since last meeting: No previous meeting but I had implemented Prims and Kruskals algorithms for finding the Minimum Spanning Tree (MST) of a graph to help better understand the problem.

Decision was made to write the system itself in C++ as a library which can be embedded in other projects.

### Action Points:

Implement a priority queue (Fibonacci heap) for Prim's algorithm

### Date of next meeting:

25/10/2017

**Agreed minute should be signed by the student and initialled by the supervisor.**

**Student's Signature:** ______________________ **Date:** __________

**Supervisor's Initials:** ______________________ **Date:** __________

### Supervisor's Comments:

QUEEN'S UNIVERSITY OF BELFAST
COMPUTER SCIENCE

# Minute of Project Supervision Meeting

| | | | |
|---|---|---|---|
| **Student Name:** | **Dean Welch** | | |
| **Project Module Code:** | **CSC3002** | | |
| **Project Supervisor:** | **Dr Charles Gillan** | | |
| **Meeting Number:** | **2** | **Date of Meeting:** | **25/10/2017** |

## Progress since last meeting, and decisions arrived at during meeting:

Progress was made in the development if a Fibonacci heap to be used to power Prim's algorithm, including setting up a testing environment with Google test

With the implementation of the Fibonacci heap taking longer than expected the decision to only work on a single method of determining the MST until the system was able to achieve that goal before progressing on to multiple algorithms.

## Action Points:

Complete work on the Fibonacci heap and move on to the implementation of Prim's algorithm.

## Date of next meeting:

8/11/2017

**Agreed minute should be signed by the student and initialled by the supervisor.**

**Student's Signature:** ______________________ **Date:** __________

**Supervisor's Initials:** ______________________ **Date:** __________

## Supervisor's Comments:

QUEEN'S UNIVERSITY OF BELFAST
COMPUTER SCIENCE

# Minute of Project Supervision Meeting

| | | | |
|---|---|---|---|
| **Student Name:** | **Dean Welch** | | |
| **Project Module Code:** | **CSC3002** | | |
| **Project Supervisor:** | **Dr. Charles Gillan** | | |
| **Meeting Number:** | **3** | **Date of Meeting:** | **8/11/2017** |

## Progress since last meeting, and decisions arrived at during meeting:

Fibonacci heap is nearing completion but still has some issues.

Work on Prim's algorithm has begun, beginning with what can be done without the Priority Queue provided by the Fibonacci heap

## Action Points:

Finish up the Fibonacci heap and fully implement Prim's algorithm including unit tests.

## Date of next meeting:

23/03/2018

**Agreed minute should be signed by the student and initialled by the supervisor.**

**Student's Signature:** ______________________ **Date:** ____________

**Supervisor's Initials:** ______________________ **Date:** ____________

## Supervisor's Comments:

QUEEN'S UNIVERSITY OF BELFAST
COMPUTER SCIENCE

## Minute of Project Supervision Meeting

| | | | |
|---|---|---|---|
| **Student Name:** | **Dean Welch** | | |
| **Project Module Code:** | **CSC3002** | | |
| **Project Supervisor:** | **Dr. Charles Gillan** | | |
| **Meeting Number:** | **4** | **Date of Meeting:** | **23/03/2018** |

### Progress since last meeting, and decisions arrived at during meeting:

Prim's algorithm has been completed and is fully functional.

Another code project has been created in which to include the library containing Prim's algorithm which reads in a file representing a graph and determines the MST.

### Action Points:

Implement Kruskal's algorithm in the library and provide the option to use that to determine the MST rather than Prim;s algorithm.

### Date of next meeting:

**Agreed minute should be signed by the student and initialled by the supervisor.**

**Student's Signature:** ______________________ **Date:** __________

**Supervisor's Initials:** ______________________ **Date:** __________

### Supervisor's Comments:

# Bibliography

[1] D. Welch, *Csc3002-mst-library*, Apr. 2018. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/Dwelch/CSC3002-MST-Library (visited on Apr. 13, 2018).

[2] D. Welch, *Mst-program-example*, Apr. 2018. [Online]. Available: https://gitlab.eeecs.qub.ac.uk/Dwelch/MST-Program-Example (visited on Apr. 13, 2018).

[3] P. E. Black, *Vertex*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Apr. 2004. [Online]. Available: https://www.nist.gov/dads/HTML/vertex.html (visited on Apr. 27, 2018).

[4] P. E. Black, *Edge*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Aug. 2017. [Online]. Available: https://www.nist.gov/dads/HTML/edge.html (visited on Apr. 27, 2018).

[5] P. E. Black, *Connected graph*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Apr. 2004. [Online]. Available: https://www.nist.gov/dads/HTML/connectedGraph.html (visited on Apr. 13, 2018).

[6] P. E. Black, *Undirected graph*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Aug. 2007. [Online]. Available: https://www.nist.gov/dads/HTML/undirectedGraph.html (visited on Apr. 13, 2018).

[7] P. E. Black, *Weighted graph*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Dec. 2003. [Online]. Available: https://www.nist.gov/dads/HTML/weightedGraph.html (visited on Apr. 13, 2018).

[8] J. L. Ganley, *Minimum spanning tree*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Sep. 2014. [Online]. Available: https://www.nist.gov/dads/HTML/minimumSpanningTree.html (visited on Apr. 13, 2018).

[9] P. E. Black, *Greedy algorithm*, in *Dictionary of Algorithms and Data Structures*, Vreda Pieterse and Paul E. Black eds. Feb. 2005. [Online]. Available: https://www.nist.gov/dads/HTML/greedyalgo.html (visited on Apr. 13, 2018).

[10] Google, *Google test*, Apr. 2018. [Online]. Available: https://github.com/google/googletest (visited on Apr. 13, 2018).

[11] T. Cormen, C. Leiserson, R. Clifford, and S. Rivest, *Introduction to algorithms, 3rd edition*, Sep. 2009. [Online]. Available: `http://ressources.unisciel.fr/algoprog/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf` (visited on Apr. 13, 2018).