The reliability of networks.

Richard James Kendall

17,890 words

April, 2005

Electronic Submission Copy

This report is submitted as part of the degree of Computer Science to the Board of Examiners in the Department of Computer Science, University of Durham.

Acknowledgements

Thanks to my supervisor Professor Hajo Broersma and to whomever was unlucky enough to have to proof read this document.

Abstract

This report documents a year long research project into the reliability and vulnerability of networks. The aim of this project was to create a tool to calculate the reliability of a given network and advise the user on changes that could be made to it in order to increase its reliability.

The project was successful in that a computer program was created which calculates the edge and vertex connectivity of a graph and can then augment the edge-connectivity of that graph. It also resulted in an API which can be used as a starting point from which other algorithms can be implemented.

Table of Contents

List of Figures

1 Introduction

Chapter Overview

This chapter is an introduction to the rest of the report. It provides an overview of this study of the reliability and vulnerability of networks. The main focus of this project is on the representation of a network as a graph and the methods by which we can reason about graphs to produce answers to the problems posed by network reliability and vulnerability. This chapter presents a high level view of a network and the structure that will be used to represent it: a graph. This chapter will also indicate to the reader what the rest of this report will contain.

1.1 What is a Network?

Simply put, a network is a collection of connected objects. Be it cities connected by roads, factory stations connected by conveyor belts or routers by the network links of the Internet. Ever since the most basic of networks have existed mathematicians have studied their properties in order to find the cheapest way of traversing them. The reasons for this are economic in nature, as if you can send your message along the shortest route through a computer network or drive along the quietest roads and therefore arrive at your destination sooner then you will save money and economics (the study of the efficient allocation of resources) is what drives our society.

As well as finding the quickest route across a given network, people have also wanted to know how likely is it that their message or parcel will reach its destination and under what circumstances the network may break down. This is the problem that this project is concerned with, given a particular network how reliable is its message delivery and how vulnerable is that network to failure.

Once some method of working out how reliable a network is has been defined it may be necessary to come up with some way to increase the reliability of that network. This may seem like a trivial case of adding new links (roads or network links), however, in the real world these items have a cost associated with them and so people may wish to increase the reliability of their network at a minimum cost.

1.2 Background to the Problem

To represent networks in a mathematical way this project is going to use a well researched area of discrete mathematics called Graph Theory to reason about and perform operations on networks.

Graph theory was first written about by a Swiss mathematician named Leonhard Euler (1707 – 1783) in a paper he published in 1736. Euler came up with the notion of graphs while he was studying the famous Konigsberg bridge problem. The city of Konigsberg (now called Kaliningrad) was built at the confluence of two rivers and this created two islands that were joined to each other and the rest of the city by seven bridges. The problem asks if a resident of Konigsberg can set out from his home and cross each bridge exactly once before returning home, the answer is no and graph theory gave Euler the answer why [OANG] (p1,2).

A graph is a collection of vertices and edges connecting those vertices. In modelling a network the vertices represent the objects to be connected, the computers or factory stations and the edges represent the links connecting them together, the network links or conveyor belts. Chapter below shows a pictorial representation of a simple graph with vertex set {a, b, c, d, e, f, g} and {ab, ac, bd, cd, df, de, fg, eg}. A formal definition of graphs and other relevant concepts will follow in Chapter Two.

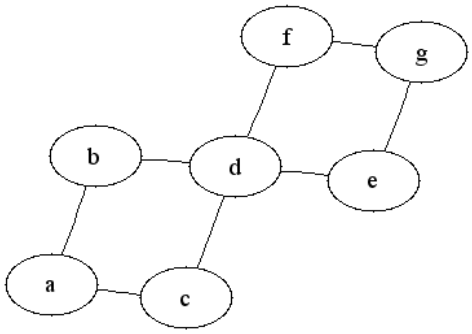


Figure 1.1: A simple graph.

Chapter Two will go into more detail about the structure of graphs and various properties of those graphs. This project is concerned with the reliability and vulnerability of networks and therefore of the graphs that are used to represent the networks. In order to measure the reliability of a given graph this project will make use of some key properties of graphs.

As networks and graphs lend themselves well to visual representation (as seen in Chapter) this project will aim to provide the results of analysis of the networks in a visual manner. Graph visualisation is the study of methods of algorithmically laying out graphs so that they have certain layout properties and look aesthetically pleasing. There has been a lot of research in this area.

As was mentioned previously, once the reliability of a given network has been established its operators may want to increase this is in a cost effective manner. This is the purview of field known as Graph Augmentation and as such this topic will be researched in order to develop some heuristics to perform this task.

1.3 Academic Context

This project is meant as a thorough survey of the existing research and documentation in the field of graph connectivity and other related questions. Its purpose is for the writer to find out about the aforementioned subject areas and attempt to implement algorithms to answer the questions posed by the problem of network reliability and vulnerability.

1.4 Project Objectives

This project is split into three sections known as the Basic, Intermediate and Advanced. A description of the deliverables for each level and what accomplishing these objectives will mean for the project follows this paragraph.

1.4.1 Basic Deliverables

- A framework for the implementation of graph connectivity algorithms
- The implementation of the connectivity algorithms

These are the preliminary tasks, completion of which will allow the project to move forward and produce a viable and useful tool looking at the reliability of networks. The framework will allow people to load graph definitions from the disk and find out how connected those graphs are (there will be more detail on graph connectivity in Chapter Two).

1.4.2 Intermediate Deliverables

- Output of the results of the connectivity algorithms in a visual manner
- Extending the framework to allow visual manipulation of graphs
- The implementation of algorithms to find the weak points in a given network and to display those results visually

Once these are complete the tool developed in the Basic section will be able to render the graphs in a visual fashion. The graphical user interface (GUI) will allow for vertices and edges to be added and removed from the graph and the results of the connectivity algorithms and the weak point finding algorithms will be displayed on the visual representation of the graph.

1.4.3 Advanced Deliverables

- Some experimental heuristics to add edges to a graph to increase its level of connectivity in a cost effective fashion.
- The implementation of algorithms that make use of those heuristics.
- Integrating the results of the above algorithms into the visual output mechanism.

Completion of these objectives will allow users of the tool to see not only the problems in their network structure but on the input of some simple edge-cost and vertex-cost data how they can increase the reliability to some prescribed level in a minimal (or close to minimal) cost fashion.

1.5 Project Plan

According to Ian Sommerville "Effective management of a software project depends on thoroughly planning the progress of the project" [ISSE] (p75). He goes on to say that this planning process can be broken down into:

1. *Project organisation* – how the development team is organised
2. *Risk analysis* – the possible risks and the probability that these risks occur
3. *Project requirements* – what does the project need in terms of hardware and software
4. *Work breakdown* – the deconstruction of the main task into smaller tasks
5. *Project schedule* – what has to be done and the order that it must be done plus the time allocated to each task
6. *Monitoring and reporting* – management reports and how the development should be monitored [ISSE] (p77).

As this is more an academic project rather than an excursion into software development much of this information on project planning is not relevant, however, some of it is and it will form the basis of this Project Plan. For example there is no team, just one developer and the computer hardware and software required is all provided by the University Information Technology Service (ITS). The final three points, however, are relevant to my project and following this text you will find each of the stages (Basic, Intermediate and Advanced) broken down into milestones with estimated time slots. As for monitoring and reporting that is taken care of by a weekly meeting with my supervisor and a project log which documents what I did and when I did it.

1.5.1 Plan for the Basic Deliverables

Task ID	Name	Length (Days)	Start	Finish	Preceding Tasks
1	Investigate graphs	10	01/10/04	10/10/04	~
2	Investigate data structures pertaining to graphs	10	01/10/04	10/10/04	~
3	Investigate connectivity algorithms	10	01/10/04	10/10/04	~
4	Create suitable data structures	2	11/10/04	13/10/04	1,2,3
5	Create a file format in which to store graph information	2	14/10/04	16/10/04	4
6	Implement the graph connectivity algorithms	10	17/10/04	06/11/04	5
7	Output the results of the algorithms to the command line	3	07/10/04	10/11/04	6

1.5.2 Basic Deliverables

Item	Date
Framework to implement algorithms	20/10/04
An implementation of the edge connectivity algorithms	10/11/04

1.5.3 Plan for Intermediate Deliverables

Task ID	Name	Length (Days)	Start	Finish	Preceding Tasks
8	Investigate graph visualisation techniques and tools	11	11/11/04	22/11/04	~
9	Design a GUI	10	11/11/04	20/11/04	5
10	Choose a method of visualising graphs and either implement it or integrate into the GUI	17	23/11/04	10/12/04	8

11	Investigate methods of obtaining the cut-set and edge-cut-sets of a graph.	10	02/01/05	12/01/05	~
12	Implement algorithms to find the cut-set and edge-cut-set of a graph	3	13/01/05	15/01/05	11

1.5.4 Intermediate Deliverables

Item	Date
A GUI	20/11/04
Output of algorithm results in a visual manner	10/12/04
The implementation of the algorithms to identify weak-spots in the network	15/01/05
An analysis of the running time of the connectivity algorithms	01/02/04

1.5.5 Plan for Advanced Deliverables

Task ID	Name	Length (Days)	Start	Finish	Preceding Tasks
13	Explore methods of increasing the reliability of a given network	6	02/02/05	08/02/05	~
14	Develop some experimental heuristics	5	09/02/05	12/02/05	13
15	Create and implement algorithms that make use of the heuristics	13	13/02/05	25/02/05	14
16	Integrate them into the GUI	14	26/02/05	10/03/05	9,15

1.5.6 Advanced Deliverables

Item	Date
Experimental heuristics to increase network reliability	12/02/05
An implementation of some algorithms that use those heuristics	25/02/05
The complete GUI with the new algorithms integrated.	10/03/05

1.6 Evaluation

There are many different levels on which a project can be judged to be a success or not, drawn from the realms of scientific experimentation to software engineering. Some of those measures are relevant here and some are not. Those from software engineering include:

3 Was it completed within the allocated time?
4 Was it completed within the set budget?
5 Does it meet all of its requirements?
6 Correctness of the solutions.

Examples of measures of success taken from scientific experimentation include:

1. Was the test fair?
2. Did the tests yield useful and meaningful results?
3. Can conclusions of weight be drawn from the results collected?

As this is a computing project to create a tool to analyse network structure and advise on changes that could be made measures of success can be drawn from both of these domains and while some are not relevant, such as budget constraints and conclusions of weight being drawn some are very relevant. As the project will include looking at the execution time of the algorithms it will be necessary to conduct the tests in an equitable manner so that the results are meaningful. Another measure will be how effective the developed program is at finding out the reliability of a network and advising on changes. Another measure of success will be the number of requirements that my project meets, in my case the requirements are the basic, intermediate and advanced deliverables.

1.6.1 Proposed Evaluation Methods

The project will be evaluated in Chapter Five using the following qualitative and quantitative methods:

- Number of delivered objectives
- Complexity of developed algorithms against the known best
- Actual running times versus the size of the input network
- How close the augmentations are to the optimal solutions

1.7 Report Outline

1.7.1 Chapter Two: Literature Survey

This chapter focuses on the theory behind the project and the tools that will be used in the implementation of the project objectives.

1.7.2 Chapter Three: Theory and Algorithms

This chapter shows how the theorems of graph theory that are covered in Chapter Two will be turned into working algorithms to answer the questions posed by the project.

1.7.3 Chapter Four: Design and Implementation

This chapter chronicles the implementation of the algorithms developed in Chapter Three; changes that were made to the conceptual designs of the algorithms are also included in this chapter.

1.7.4 Chapter Five: Results and Evaluation

This chapter looks at the correctness and quality of the algorithms that were developed in Chapter Three and implemented in Chapter Four.

1.7.5 Chapter Six: Conclusions

This chapter concludes the report with a brief statement showing what was achieved and what further work can be done.

2 Literature Survey

Chapter Overview

This chapter introduces the sources of information that have been used throughout this project and the theory behind the subject area. As there is no standard terminology within the field of graph theory, this chapter will also indicate to the reader what terminology they can expect to see in the rest of this report.

2.1 Research Topics

This literature survey is split into several strands which cover the theory behind the subject areas that concern this project. The strands are:

2 Basic graph theory: what a graph is, how it is represented and the principles and terminology that are associated with them.
3 Properties and Metrics: what about graphs can be used to answer the questions posed by network reliability and vulnerability and how these properties can be measured.
4 Computational Representation: data structures that can be used to store the graphs within a computers memory so that operations can be carried out on them.
5 Optimisation: how to increase the reliability and decrease the vulnerability of a given network.

2.2 Basic Graph Theory

A graph is a mathematical representation of a set of connected objects, for example the scenarios depicted in Figure 2.1 and Figure 2.2 can both be represented by the graph shown in Figure 2.3.

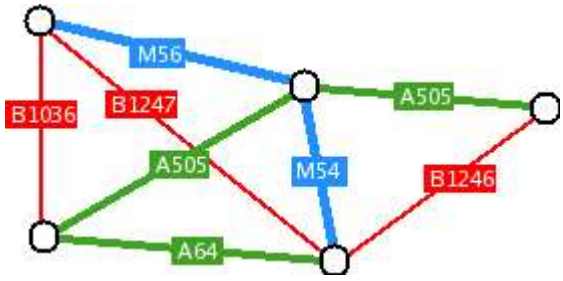


Figure 2.1: A road network[1]

[1] From Figure 1.1 pg 1 of [INGT]

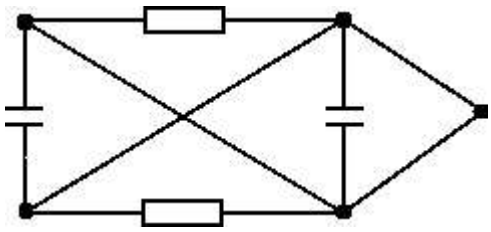


Figure 2.2: An electrical circuit[2]

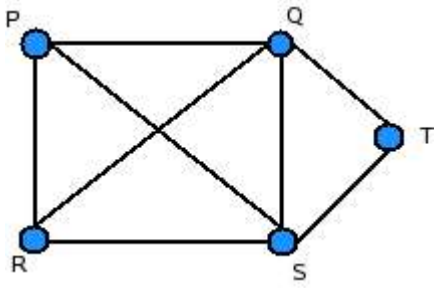


Figure 2.3: A graph[3]

The points in Figure 2.3 are called vertices [INGT] (p1) and the lines are called edges [INGT] (p1). As well as physical environments graphs can be used to model abstract situations such as scheduling problems in which the activities become vertices and any two vertices are connected by an edge if they cannot be carried out simultaneously.

More formally a graph G consists of a non-empty finite set V(G) of elements called vertices and a finite family E(G) of unordered pairs of elements from V(G) called edges. As this is a family rather than a set then the edges need not be distinct, i.e. it is possible to have more than one edge between two given vertices. An edge {v,w} is said to join the vertices v and w and is abbreviated to *vw*. For example the vertex set of the graph shown in Figure 2.3 is: {P, Q, R, S, T} and the edge family is {PR, PS, PQ, QT, QS, QR, SR, ST}.

Edges and vertices have certain properties which when put together make up properties of the graph itself. Some of the properties of edges are listed below:

[2] From Figure 1.2 pg 1 of [INGT]
[3] From Figure 1.3 pg 2 of [INGT]

- Direction – an edge can run from the source vertex to the destination vertex, from the destination vertex to the source vertex or both ways. A graph which contains any directed edges is called a digraph.
- Capacity or weight – an edge can be given a number usually called a capacity or weight that indicates something about the physical or situational link that the edge represents. In the example of a computer network the weight could mean simply the amount of information that link can carry per unit time or it could be calculated in a more complex way in order to take into account how busy the link gets or other such factors. These weights are usually integers or real numbers and can be represented formally as a total function $c : E(G) \rightarrow \mathbb{N}$.
- Flow – a weighted edge can similarly be given a flow which is the amount of the weight assigned to that edge that is being used at that moment in time. This again is a function that maps an integer or real number that is less than or equal to the value of c for that edge. Formally this can be represented by the following function $f : E(G) \rightarrow \mathbb{N}$ with the property that $f(e) \leq c(e) \forall e \in E(G)$. Flows also have to satisfy a number of other properties which will be covered in Chapter Three.

Vertices have the following properties:

- Degree – the total number of edges that are incident with a given vertex regardless of direction, formally this is called $deg(v \in V(G))$. In a digraph there are two sub versions of this property, known as:
    - In-Degree – the number of directed edges pointing inward that are incident with a given vertex.
    - Out-Degree – the number of directed edges pointing outward that are incident with a given vertex.

  Clearly the sum of the In-Degree and Out-Degree of a vertex v should be equal to $deg(v)$. A vertex with a degree of zero is called an isolated vertex and a vertex with a degree of one is called an end vertex.
- Labels – the vertices have an identifier (such as P, Q, … in Figure 2.3) and more often than not that is used to label the vertices but they can also have separate labels.
- Neighbourhood – the set of vertices to which the vertex v is directly connected (via an edge) formally this is called $N(v)$. $deg(v)$ should clearly be equal to $|N(v)|$. For directed graphs there is also an In-neighbourhood $\mathrm{IN}(v)$ the set of vertices with edges directed in to v and an Out-neighbourhood $\mathrm{ON}(v)$ the set of vertices with edges directed from v.

These are some of the properties of graphs, the ones listed here are the most useful:

- Min and Max Degree - $\delta(G)$ is the minimum degree of the graph, the size of the smallest degree of any vertex in G, $\Delta(G)$ is the maximum degree of the graph, the size of the largest degree of any vertex in G.
- Connectedness – if there is a path between every pair of vertices in the graph then that graph is referred to as connected, if not it is called disconnected.

A lot of graph theory has to do with 'walks' from vertex to vertex along the edges of a graph. A walk is a finite sequence of edges of the form $v_0v_1, v_1v_2, \ldots, v_{m-1}v_m$ or v_0 -> v_1 -> v_2 -> … v_m. In such a walk v_0 is called the initial vertex or source vertex and v_m is the final vertex or sink vertex [INGT] (p26). A walk in which every edge is distinct is called a trail, if all the vertices in a trail are distinct (apart from v_0 and v_m) then the trail is referred to as a path. If v_0 and v_m are the same then the path or trail is closed, a closed path is called a cycle [INGT] (p26,27).

2.3 Properties and Metrics

The following properties of graphs are relevant in the field of network reliability, they will be described in turn below:

- number of components in the graph,
- size of vertex cutset,
- size of edge cutset,
- edge connectivity,
- vertex connectivity,
- maximum flow.

2.3.1 Number of Components

A connected graph consists of one component, any graph can be partitioned into one or more pieces called components in which no vertices in component x are connected by an edge to any vertices in component y. This means that a graph G made of components C_1, C_2, …, C_n can be expressed as the union of those components $C_1 ? C_2 ? ... ? C_n$. This is shown in Figure 2.4.

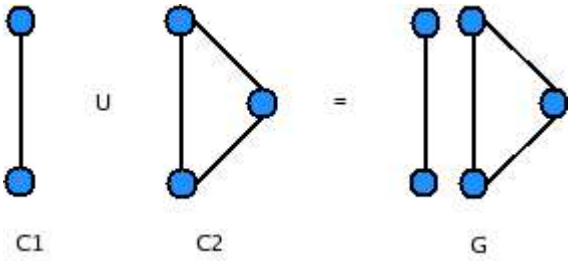


Figure 2.4: A union of components to make a graph[4]

Clearly the more components that make up a graph then the more likely it is that a communication will fail, this is a trivial measure of network reliability. The idea of the number of components into which a graph can be split is used in the next two more sophisticated measures of reliability, the only value that is acceptable for this measure is one.

2.3.2 Size of edge/vertex cutset

For any graph G there are subsets of E(G) called disconnecting sets [INGT] (p28), a disconnecting set is a set of edges whose removal disconnects G (separates G into two or more components). A stronger version of the disconnecting set is the cutset, a cutset is a disconnecting set no proper subset of which is a disconnecting set [INGT] (p28), in effect the smallest number of edges that need to be removed in order to disconnect G. These properties can also be applied to the vertices of G, V(G), the vertex equivalent of a disconnecting set is a separating set [INGT] (p29). In the case of vertex deletion all the edges incident with that vertex are also deleted (unlike the deletion of an edge which leaves the vertices intact). If a cutset of G contains only one edge then G is said to contain a 'bridge' [INGT] (p29) likewise if the smallest separating set of G contains only one vertex then that vertex is called a 'cut-vertex' [INGT] (p29).

The separating set and disconnecting set can be extended to a vw-disconnecting and a vw-seperating set. These are the edges and vertices that need to be removed from a graph G in order that the vertices v and w are in different components [INGT] (p122).

When these properties are mapped through to the real world scenario that the graph is representing it shows the tolerance that the network has to node and link failure, for example if it is a computer network that is being represented then the cutset shows how many links can fail before the network breaks down. The same is true of the smallest separating set as it shows how many servers or routers can break. Obviously the smaller the cutset or separating set the more vulnerable to failure that network is,

[4] From Figure 2.7 pg 10 of [INGT]

also if you have a network that contains bridges or cut-vertices then it is enormously vulnerable and this is something to be avoided.

2.3.3 Edge/Vertex Connectivity

These properties link back to those described in Section . A non-complete (some given vertex is not adjacent to every other vertex), connected graph G, G's connectivity $\kappa(G)$ is the size of the smallest separating set in G [INGT] (p29). Therefore $\kappa(G)$ is the minimum number of vertices that have to be removed from G in order to disconnect it. A graph G is called k-connected if $\kappa(G) \geq k$. Again if G is a connected graph then G's edge-connectivity $\lambda(G)$ is the size of the smallest cutset in G [INGT] (p29). Therefore $\lambda(G)$ is the minimum number of edges that have to be removed from G in order to disconnect it. A graph G is called k-edge-connected if $\lambda(G) \geq k$.

There are alternative definitions of k(-edge)-connectivity that are much more useful if one's goal is computing the connectivity values. They are centred on the definitions of vw-separating sets, vw-disconnecting sets, edge-disjoint paths and vertex-disjoint paths (described in Section). These alternative definitions make use of Menger's Theorem (1927) in its edge and vertex forms.

Menger's Theorem (1927):
The maximum number of vertex-disjoint paths connecting two distinct non-adjacent vertices v and w of a graph is equal to the minimum number of vertices in a vw-seperating set. [INGT] (Th 28.2 p124)

There is also an edge form of Menger's Theorem which was first proved by Ford and Fulkerson in 1955, the edge form is as follows:

The maximum number of edge-disjoint paths connecting two distinct vertices v and w of a connected graph is equal to the minimum number of edges in a vw-disconnecting set. [INGT] (Th 28.1 p122)

From the two theorems stated above one can deduce the following conditions for a graph to be k-edge-connected or k-connected.

Whitney's Second Theorem (1932):
A graph G is k-edge-connected if and only if any two distinct vertices of G are connected by at least k edge-disjoint paths. [INGT] (Co 28.3 p124)

Whitney's Second Theorem (1932) a:
A graph G with at least k+1 vertices is k-connected if and only if any two distinct vertices of G are connected by at least k vertex-disjoint paths. [INGT] (Co 28.4 p124)

The edge connectivity of a graph is also the same as the minimum of the maximum flow between any two vertices v and w if the capacity of all the edges is one. This equivalence means that standard maximum flow algorithms can be used to answer the edge connectivity problem.

2.4 Computational Representation

In order to perform calculations on graphs using a computer it is necessary to store the graph structure in the computer's memory. There are several data structures that can be used, each of these representations has virtues and at the same time makes some operations hard. Roughly the data structures can be organised into two classes: static and the more complicated dynamic or linked representations.

2.4.1 Static Representations

Static representations are simple data structures that in most cases use a large amount of space, the standard ones are:

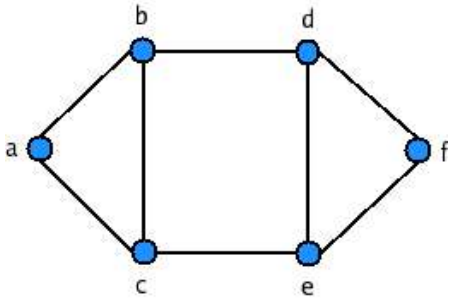
1. The adjacency matrix
2. The edge incidence matrix
3. The edge list.

2.4.1.1 The Adjacency Matrix

The adjacency matrix A of a graph G(V, E) is defined as the |V| x |V| matrix:

$$A(i,j)=\begin{cases}1 \text{ if } ij \in E(G)\\ 0 \text{ if } ij \notin E(G)\end{cases}$$

The adjacency matrix can completely represent a graph in $O(|V|^2)$ (for an explanation of Asymptotic Notation see Chapter Three of [INTA]). Using this representation determining if two vertices are adjacent takes $O(1)$ time [AGT] and determining what vertices are adjacent to a given vertex takes $O(|V|)$ time [AGT]. The adjacency matrix of a digraph is defined in exactly the same way except that the matrix for a digraph is asymmetric whereas for a graph it is symmetric. Figure 2.5 shows a graph along with the adjacency matrix that represents it.



	a	*b*	*c*	*d*	*e*	*f*
a	0	1	1	0	0	0
b	1	0	1	1	0	0
c	1	1	0	0	1	0
d	0	1	0	0	1	1
e	0	0	1	1	0	1
f	0	0	0	1	1	0

Figure 2.5: A graph and its Adjacency Matrix

There are extensions and optimisations that can be made to this representation, for example if the graph one wants to represent has weights on its edges then instead of placing a 1 at A(i,j) one could place the weight yielded by the function c(i,j). When this structure is used to represent an undirected graph the adjacency matrix is symmetric (as if $ij \in E(G)$ then $ji \in E(G)$), this means that you can halve the size of the data structure. That said the adjacency matrix is rarely used as it is hugely inefficient on all but the densest of graphs [GTA] (p74).

2.4.1.2 The Edge Incidence Matrix

The edge incidence matrix is similar to the adjacency matrix except that it marks which edges are adjacent or incident with a particular vertex. If G is an undirected graph with vertices $v_1,\ldots,v_p$ and edges $e_1,\ldots,e_q$ then the incidence matrix B of G is defined as the p x q matrix:

$$B(i,j) = \begin{cases} 1 \text{ if vertex } v_i \text{ is incident with edge } e_j \\ 0 \text{ otherwise} \end{cases}$$

2.4.1.3 The Edge List

The edge list structure for a graph G is a list of the edges in G, each edge is represented as a vertex pair. The pairs are unordered for undirected graphs and ordered for digraphs [AGT] (pg10). As there will be one entry per edge and it takes $\log(|V|)$ bits to uniquely identify one of |V| vertices it will take $O(|E(G)| * \log(|V(G)|))$ bits to store this structure in the memory of a computer. If the edges are stored in lexicographic order, vertices i and j can be tested for adjacency in $O(\log(|E|))$ time [AGT] (pg10). The set of vertices adjacent to a vertex v can be found in $O(\log(|E|) + \deg(v))$ time [AGT] (p10).

2.4.2 Dynamic Representations

Dynamic representations are memory efficient but use more complex data structures than static representations; they also facilitate dynamic changes to graphs [AGT] (p10). The adjacency list representation for a graph/digraph G(V,E) gives a list of adjacent vertices for each vertex v in V(G). An example of a graph and its adjacency list can be seen in Figure 2.6.

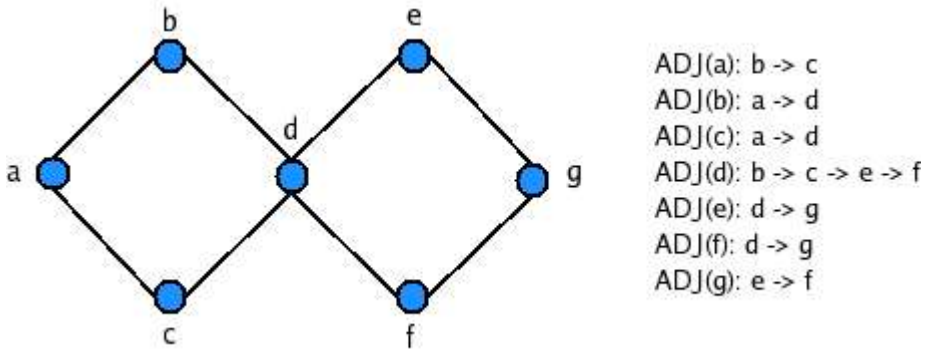


Figure 2.6: A graph and its Adjacency List

How the data structure works depends on how the vertices and edges are represented. The vertices can be represented in the following ways, which will be described in more detail in the coming paragraphs.

- A linear array,
- A linear linked list, or
- A pure linked structure.

2.4.2.1 A Linear Array

In this representation, each of the vertices has an element of a linear array which acts as the 'header' for the list of vertices that are incident with that vertex. This can be seen in Figure 2.7.

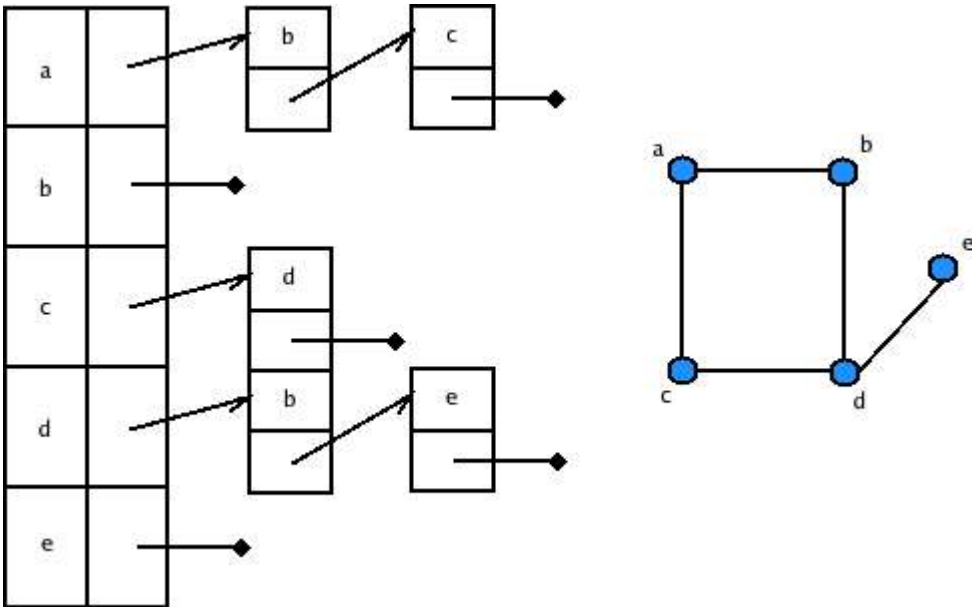


Figure 2.7: A Linear Array representation and the graph it represents.[5]

2.4.2.2 A Linear Linked List

This is similar to the Linear Array except that the 'headers' for each vertex are stored in a linked list. This is illustrated in Figure 2.8.

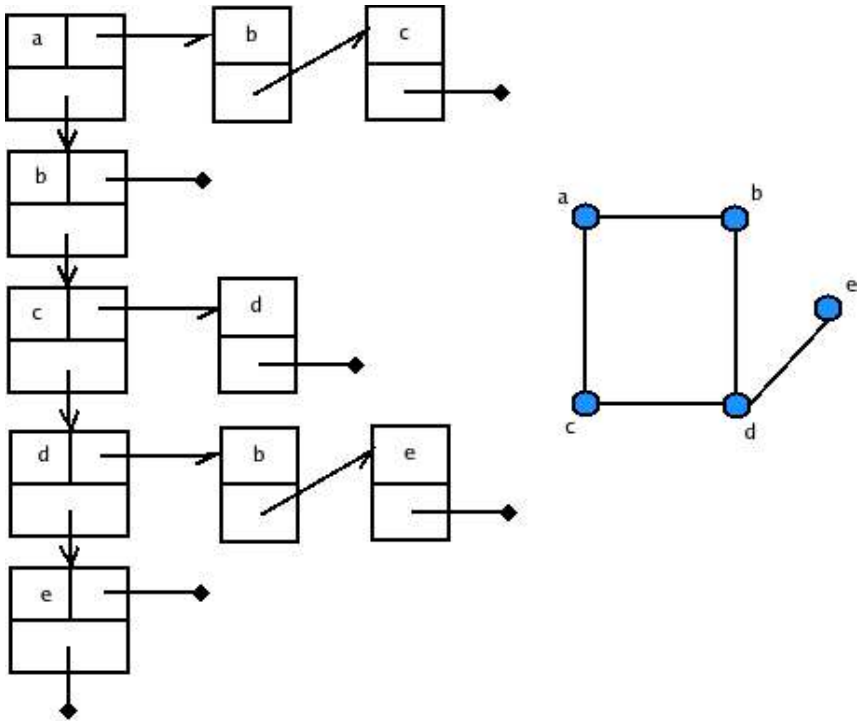


Figure 2.8: A Linked List representation and the graph it represents.[6]

[5] From Figure 1.10a pg 12 of [AGT]

[6] From Figure 1.10b pg 12 of [AGT]

2.4.2.3 A Pure Linked Structure

In the pure linked structure there is a special vertex called the entry vertex to the graph or digraph [AGT] (p11). There is a pointer that points to this vertex called the entry pointer and every vertex is reachable by traversing the structure from this initial place. If a vertex is not reachable through some combination of vertices from the entry pointer then a list of entry pointers is maintained [AGT] (p11). This list will be large enough such that every vertex in the graph can be accessed. This structure can be seen in Figure 2.9.

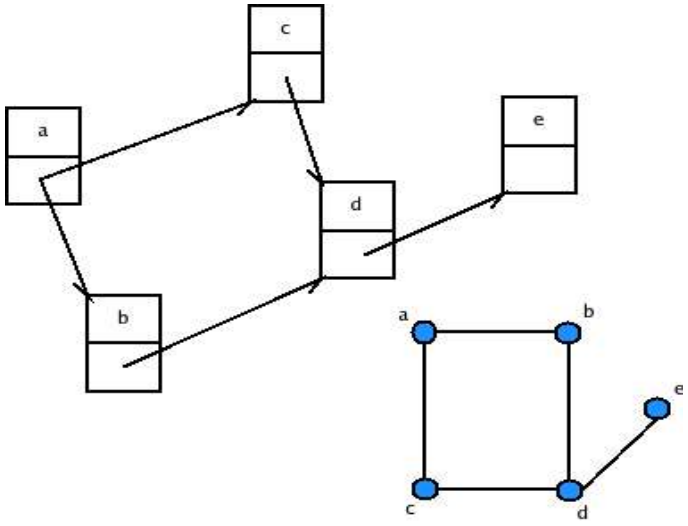


Figure 2.9: A pure linked representation[7]

The most memory efficient of the dynamic representations is the pure linked structure as it is the only one to retain just one copy of each vertex.

2.5 Graph Augmentation

Once measures have been assigned to a particular network model the owners or users of that network may wish to increase its reliability. As in the real world scenario that the graph represents, there are costs associated with adding new objects that the edges and vertices represent and this needs to be done in a minimal fashion. This is called Graph Augmentation and it is part of a large class of 'optimisation' problems associated with graphs. The advanced objectives of my project are centred around coming up with experimental heuristics for the augmentation of graphs to a certain level of connectivity and the implementation of algorithms that use these heuristics. For this reason this section will look into the techniques of augmentation that have already been detailed in various research papers. The main paper I shall be using is the survey "Augmenting Graphs to Meet Edge-Connectivity Requirements" by Andras Frank. This section of the document is going to present the ideas behind

[7] From Figure 1.10c pg 12 of [AGT]

augmentation in an informal fashion. Algorithms derived from these ideas will be shown in Chapter Three.

2.5.1 Ideas and Terminology

Augmenting a graph to a certain level of edge-connectivity means adding a certain number of edges. The object of these optimisation problems is to add a minimum number of edges.

This requires an alternative definition of k-edge-connectivity. An undirected connected graph G is k-edge-connected if $d(X_i) \geq k$ for all $\emptyset = X_i \subset V(G)$, i.e. the number of out-edges of every non-empty sub-partition of V(G) is at least k [AECR] (p29). This definition is used in the following theorem:

Cai & Sun's Theorem (1989):
An undirected graph G and an integer k, G can be made k-edge connected by adding γ new edges if and only if $\Sigma(k - d(X_i)) \leq 2\gamma$ holds for every sub-partition $\{X_1, X_2, ..., X_t\}$ of V(G) [AECR] (Thm 4.1 p34).

The augmentation process begins by adding a new vertex s to the vertex set of a graph G and enough edges from s to G such that $d(X_i) \geq k$ for all $\emptyset = X_i \subset V(G)$. This can be seen in the top part of Figure 2.10.

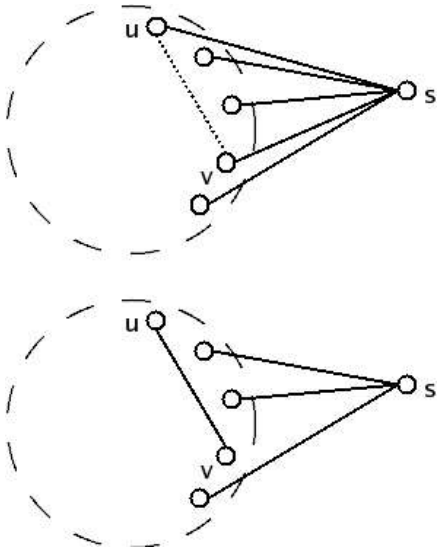


Figure 2.10: An illustration of the splitting lemma.

Then the 'splitting lemma' otherwise know as Lovász's Theorem is used to remove the edges between s and G and replace them with edges between the end vertices of the

edges in G. This is demonstrated in Figure 2.9 which in the top image shows the vertex 's' with edges to G, the edges su and sv can be removed by adding the edge uv as shown in the second image.

This augmenting technique works as a two-stage process:

1. Add a vertex s to the graph G and add enough edges such that the number of out-edges for every sub-partition of V(G) is at least k.
2. Repeatedly apply the splitting lemma such that the previous condition is not violated, until it can not be applied again

This process is always optimal, adding at most γ edges.

An example of how a tree can be augmented to be 3-edge-connected is shown in Figure 2.10 and Figure 2.11. Figure 2.10 shows the original tree after the vertex s has been added and connected by enough edges to satisfy $d(X_i) \geq k$ for all $\emptyset = X_i \subset V(G)$, Figure 2.11 shows the edges that are removed using the splitting lemma and the new edges that are added in their place. They are numbered showing the order in which they are removed and added. In this process the order in which the edges are replaced matters as in certain cases the method can get 'stuck', in these cases the method can back-track and take another route.

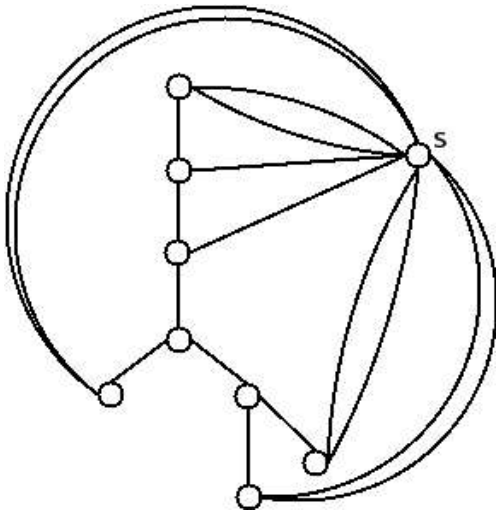


Figure 2.11: A tree after stage one of the augmentation technique.

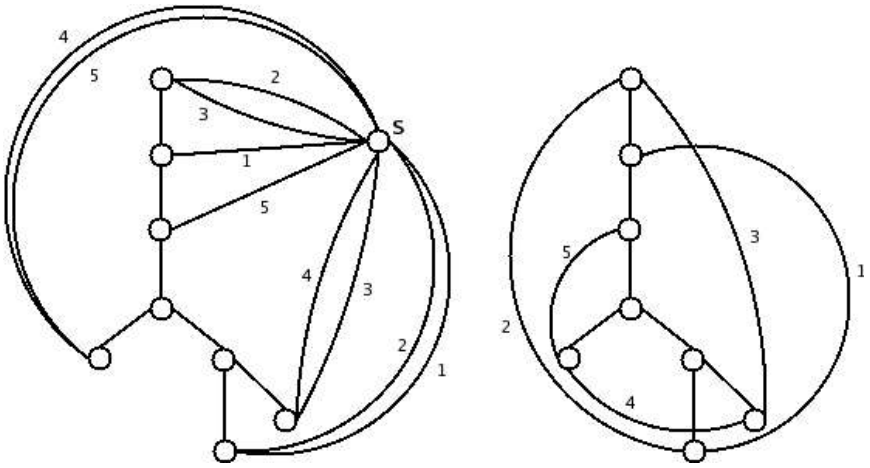


Figure 2.12: The tree after the splitting lemma (stage two) has been used.

The resulting graph (no longer a tree) is 3-edge-connected. As can be seen from Figure 2.11 the resulting graph can contain parallel edges and thus is a multi-graph. This usually happens in the case in which a tree is being augmented but when augmenting general graphs it happens less often.

3 Theory and Algorithms

Chapter Overview

This chapter is going to indicate to the reader how the theory that was covered in the previous chapter will be used to create working algorithms that will answer the questions posed by network reliability. Also it will build on that theory to look at some experimental heuristics for increasing the connectivity of a given graph to a predefined level.

3.1 Core Principles

There are several techniques that are crucial to the more complex graph manipulation algorithms that will be used. These algorithms and data structures will be covered in this section of the document.

3.1.1 The Linked Graph Structure

The linked representation that will be used to represent the graph in the memory of the computer will consist of three objects a Graph, Vertex and Edge object. They will be arranged as in Figure 3.1.

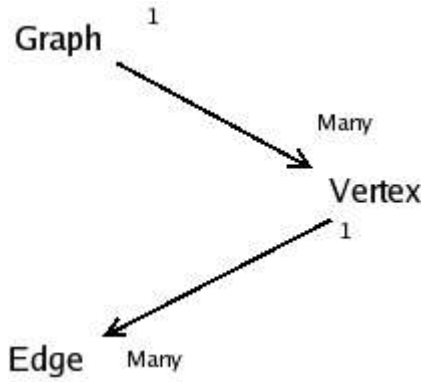


Figure 3.1: The object arrangement

The Graph object will maintain a list of Vertex objects and each of these Vertex objects will maintain a list of Edges, separated into in edges and out edges.

3.1.2 Tree Construction

One of the most important principles that makes up the core of several graph algorithms is tree construction, in general there are two tree styles that can be built: the breadth-first search tree and the depth-first search tree. Covered first here is the breath-first variant. Firstly a tree is a graph in which there is only one path between any two vertices, pictorially they are represented as in Figure 3.2.

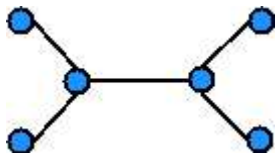


Figure 3.2: A tree

Breadth-first simply means that the tree is constructed in a layer first rather than a branch first manner. The pseudo code of the basic algorithm for producing a breadth first search (BFS) tree from a graph is shown in As well as . Used in this algorithm is a standard first in, first out queue (FIFO) with the following operations:

- `Push(Q, e)`: adds a new element, e, to the end if the queue Q
- `Pop(Q)`: returns the element at the beginning of the queue Q
- `Notempty(Q)`: returns true if the queue, Q, contains one or more items and false otherwise.

The operation of a queue is shown in Figure 3.3.

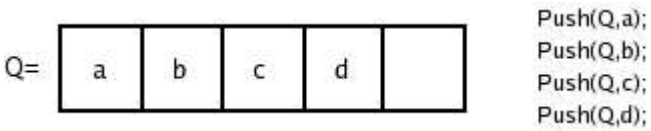


Figure 3.3: A queue

The queue is used to hold a list of the vertices that need to be visited and as such is known as the 'vertex worklist', initially it is populated with the starting vertex (also known as the root of the tree). The algorithm travels down each edge in the input graph as many times as the constant MAX allows (the reasons for which will become apparent later), as it travels down each edge it produces the tree structure in the variable called T.

```
1. BFS_Tree(G, s)
2.   MAX: Integer // max times one edge can be visited
3.   G: Graph // the input graph
```

```
4.   T: Tree // the tree
5.   W: Queue // vertex worklist
6.   counter: list // the edge visitation counter
7.
8.   push(W, s)
9.   add s as a root of T
10.  while(notempty(W)) do
11.       v = pop(W)
12.       p = next child on current layer of T
13.       for each child c of v
14.            if (counter[edge(W, c)] != MAX)
15.                 push(W, c)
16.                 increment(counter, edge(W,c))
17.                 add c as child to p
18.            end if
19.       end for
20.       move to next layer of T
21.  end while
22. End Procedure
```

Algorithm 3.1: The BFS tree construction algorithm[8]

As well as BFS trees there are depth-first search (DFS) trees, these are produced in the same way except that in the place of the Queue (line five) there as a Stack [AGT p152], a last in first out (or LIFO) structure. This allows the algorithm to explore the branches of the tree to a predefined depth (the MAX) constant. The operations on a stack are similar to those of a queue with two small differences:

- Push(S,e): adds the element e, to the top of the stack, S
- Pop(S): returns the element at the top of the stack, S.

You can see this demonstrated in Figure 3.4.

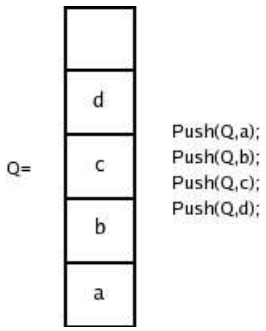


Figure 3.4: A stack

[8] Based on BFS(G,u) from page 154 of [AGT]

The use of the stack alters the algorithm by making it proceed down the most recently discovered vertex every time.

3.1.3 Tree Traversal

Once a search tree has been built it then needs to be looked at or traversed in some manner that yields useful results. There are three main ways to traverse a tree and they are in-order, pre-order and post-order.

In-order traversal is "processing all nodes of a tree in order of the left sub-tree, the root and then the right sub-tree and the root"[9]. During a pre-order traversal the root of the tree is processed followed by the sub-trees in the order that they are found, in a post-order traversal things happen the other way round, the sub-trees are processed follwed by the root of the tree. As these traversals are all recursive when a sub-tree is processed it is passed through the same mechanism. This can be seen in the pseudo code layout of an in-order traversal in Algorithm 3.2.

```
1.   In-order(tree)
2.        In-order(left_subtree)
3.        Process(root)
4.        In-order(right_subtree)
5.   End Procedure
```

Algorithm 3.2: In-order traversal pseudo code.

In-order traversal is not very useful in this arena as it only makes sense to use it with binary trees (trees in which each node may have no more than two children) and the BFS and DFS trees created from graphs are general trees (each node may have any number of children).

3.1.4 Path Finding

For many graph algorithms paths from vertex to vertex via the edges have to be found. They can be found by constructing a BFS tree and traversing the tree in someway to find how to get from the source vertex to the sink vertex. The algorithm used to create a BFS tree from a given input graph has already been presented and this just needs to be coupled with a tree traversal method in order to locate all the paths between two vertices.

[9] http://www.brpreiss.com/books/opus4/html/page260.html

3.2 Graph Algorithms

This section of the document details the main graph algorithms that do the complex and hard work, many of them use mechanisms and techniques laid out in the previous Core Principles section.

3.2.1 Maximum Flow

There are many techniques for finding the maximum possible flow from one vertex (the source) to another vertex (the sink) in a graph. The method that will be used by this project is known as the Ford Fulkerson method after the two mathematicians that devised it in 1956. The maximum flow between two vertices is the sum of the minimum available capacity on the paths between the source and the sink.

The idea of maximum flow only makes sense when one is working with graphs that have capacities and flows on the edges, the capacity function c(v,w) returns a real number that indicates the capacity of the edge connecting the vertices v and w in the graph. Similarly the function f(v, w) returns a real number indicating the flow (or the currently used capacity) on that edge. The following properties must hold:

- $f(v,w) \leqslant c(v,w) \forall v,w \in V(G)$ That is the flow on an edge must not exceed the capacity on that edge for all the edges in the graph.
- $\forall v \in V(G) \sum_{x \in \mathrm{IN}(v)} f(x,v) = \sum_{y \in ON(v)} f(v,y)$ The sum of the flows on the edges entering a given vertex must equal the flows of the edges leaving the vertex for all the vertices in G.

The Ford Fulkerson method begins by setting the flow on all the edges of the graph it is operating over to one, once this has been done it iterates over the graph looking for flow augmenting paths between the source and the sink node. Once no additional flow augmenting paths can be found it halts and reports the maximum available flow between the two vertices. A flow augmenting path is a path between the source and the sink on which the edges have available flow (i.e. c(v,w)-f(v,w) is +ve), a flow augmenting path is said to contribute to the flow between the source and sink. The maximum flow is the sum of the flows across the augmenting paths that have been found, the flow on a path is the value of the minimum flow value on an edge in that path. This method also relies on the idea of a residual network. A residual network of G, G_f is the vertex set of G, V(G) and a set of edges representing the edges in G, E (G) that can admit more flow. There is an additional capacity function defined for Gf, $c_f(v,w) = c(v,w) - f(v,w)$, this is the additional flow that the edge can admit. If the residual flow is zero then no more units of flow can be admitted and the edge is

said not to exist within the residual network. An example of a graph, and augmenting path and the residual network that results from this augmenting path being chosen can be seen in Figure 3.5.

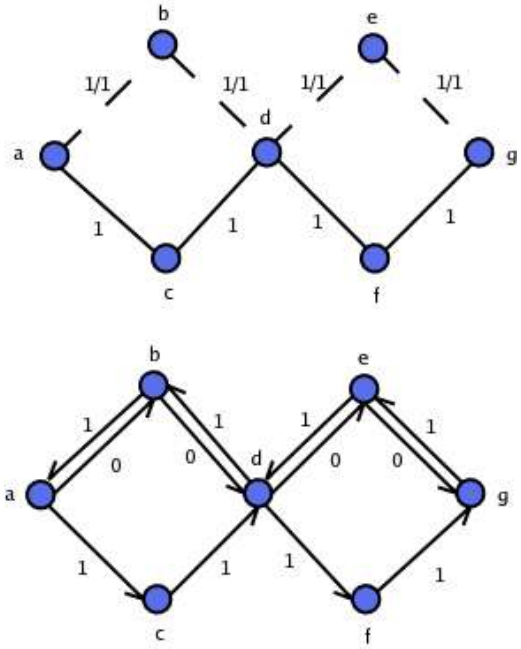


Figure 3.5: A graph, showing a FAP and the residual network.

A pseudo code version of the Ford Fulkerson method is shown in Algorithm 3.3.

```
1.  Ford-Fulkerson(G,s,t)
2.   f(v,w): flow across edge vw
3.   for every edge in E(G) (v,w)
4.        set f(v,w) & f(w,v) to 0
5.   end for
6.   while there is a flow augmenting path p in Gf
7.        set c(p) to be the min{cf(v,w) : (v,w) in p}
8.        for each edge (v,w) in p
9.             set f(v,w) to f(v,w) + c(p)
10.       end for
11.  end while
12.  report sum of flows
13. End Procedure
```

Algorithm 3.3: The basic Ford Fulkerson algorithm.

A crucial part of the implementation of the Ford Fulkerson method is the mechanism by which the flow augmenting paths are found. If they are found in a naive fashion then the algorithm may not terminate having found the maximum possible flow or it

may run for a vastly long time. As long as the paths are chosen using the Edmonds-Karp extension to Ford Fulkerson then it will terminate having found the maximum flow. By this extended method the augmenting paths are found by following the shortest available path at each stage of the computation [NoZa]. This uses the BFS method to search for and find the paths, which will find the shortest path at each stage of the computation and thus it will not get stuck.

3.2.2 Edge Connectivity

As was discussed in the previous chapter the edge connectivity of a given graph is the number of edges that need to be removed from a graph in order to disconnect it. By Menger's Theorem (1927) this can be thought of as the minimum of the sizes of the sets of edge disjoint sets connecting every pair of distinct vertices in the graph. There are two ways to calculate this value one makes use of the maximum flow technique and the other involves building a BFS tree rooted at each vertex in the graph, listing all the paths from one vertex to the others and then counting the unique paths. Both of these techniques will be explored here.

3.2.2.1 Edge Connectivity as Maximum Flow

In order to measure the number of edge disjoint paths between any two distinct vertices in a graph using a maximum flow algorithm the problem needs to be formulated such that the maximum flow over each path is exactly one. This way if there are three edge disjoint paths between the source and sink then the algorithm will return the number three. This can be achieved by setting the capacities of all the edges in the graph to one. This way by necessity the flow augmenting paths (FAP) that are found will be the same as the edge disjoint paths separating the two vertices. This is shown in Figure 3.6.

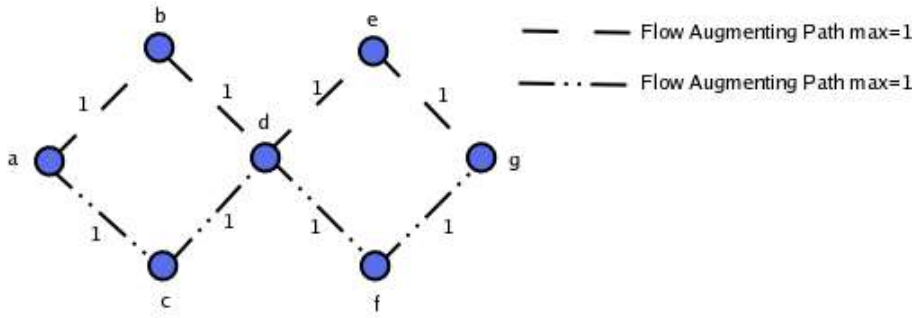


Figure 3.6: A graph showing FAPs from a->g.

Once the graph has been altered so that all the capacities are one then some algorithm needs to iterate over the vertices of the graph, running the maximum flow algorithm

between every pair of disjoint vertices and then output the smallest answer. A pseudo code sketch of this algorithm can be seen in Algorithm 3.4.

```
1.  Edge-Con-MF(G)
2.   min: int representing min value
3.   for each s in V(G)
4.        for each t in V(G)
5.             if (s is not t)
6.                  m = Ford-Fulkerson(G,s,t)
7.                  if (m < min)
8.                       min = m
9.                  end if
10.            end if
11.       end for
12.  end for
13. End Procedure
```

Algorithm 3.4: The maximum flow edge connectivity algorithm.

3.2.2.2 Edge Connectivity as Pure Search

In order to work out the number of edge disjoint paths between all the pairs of vertices by search a BFS tree has to be constructed rooted at every vertex of the graph. These trees are then traversed in a pre-order fashion to find all the possible paths to all the other vertices. These paths are then checked to see how many use distinct edges and this number is recorded, the minimum of all these values is the edge connectivity of the graph in question. A graph along with a BFS tree rooted at one of its vertices is shown in Figure 3.7. The pseudo code showing how this method should be implemented is documented in Algorithm 3.5.

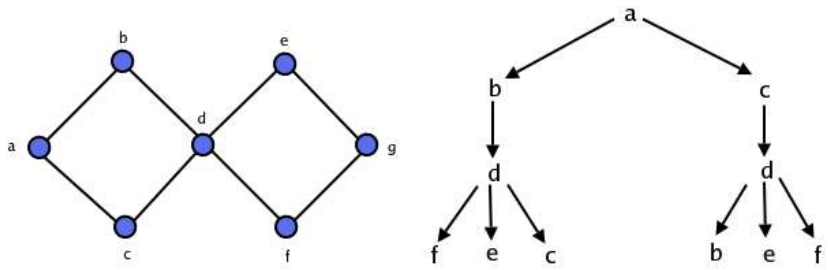


Figure 3.7: A graph and a partial BFS tree rooted at vertex 'a'.

```
1.  Edge-Con-Search(G)
2.   min: int representing min value
3.   for each s in V(G)
4.        T = BFS_Tree(G,s)
5.        total: int representing number of paths s -> t
6.        for each t in V(G)
```

```
7.              while p = Pre-Order(T,t)
8.                   increment total
9.              end while
10.        end for
11.        if (total < min)
12.             min = total
13.        end if
14.   end for
15. End Procedure
```

Algorithm 3.5: The search edge connectivity algorithm.

3.2.3 Vertex Connectivity

As was discussed in the previous chapter the vertex connectivity of a graph is the minimum number of vertices that once removed disconnect the graph. By Menger's Theorem (1927) this can be thought of as the minimum of the sizes of the smallest set of vertex disjoint paths connecting any two disjoint vertices in the graph. As with edge connectivity the vertex connectivity problem can be solved in two different ways using a computer, the first uses a maximum flow approach and the second uses a BFS tree. Both approaches will be discussed here.

3.2.3.1 Vertex Connectivity as Maximum Flow

It has already been shown that a maximum flow algorithm can be used to find the edge disjoint paths between the vertices of a graph. In order to calculate the number of vertex disjoint paths (as required by vertex connectivity) it is necessary to construct a graph in which there is an edge disjoint path for every vertex disjoint path and again set all the capacities to one before running the maximum flow algorithm over the graph.

A graph can be modified such that every vertex disjoint path is represented by an edge disjoint path in the following way:

- Replace every vertex in the graph by a vertex pair connected by a directed edge. The vertices in the pair are known as the 'receiver' and the 'transmitter'. The connecting edge is directed from the receiver to the transmitter.
- Iterate over the edges in the original graph, replace undirected edges with a pair of directed edges from the transmitter of the source vertex to the receiver of the sink and visa versa. Directed edges are replaced with a directed edge from the transmitter of the the source to the receiver of the sink.
- Set the capacities of all the edges to one.

A graph modified in this way is shown in Figure 3.8.

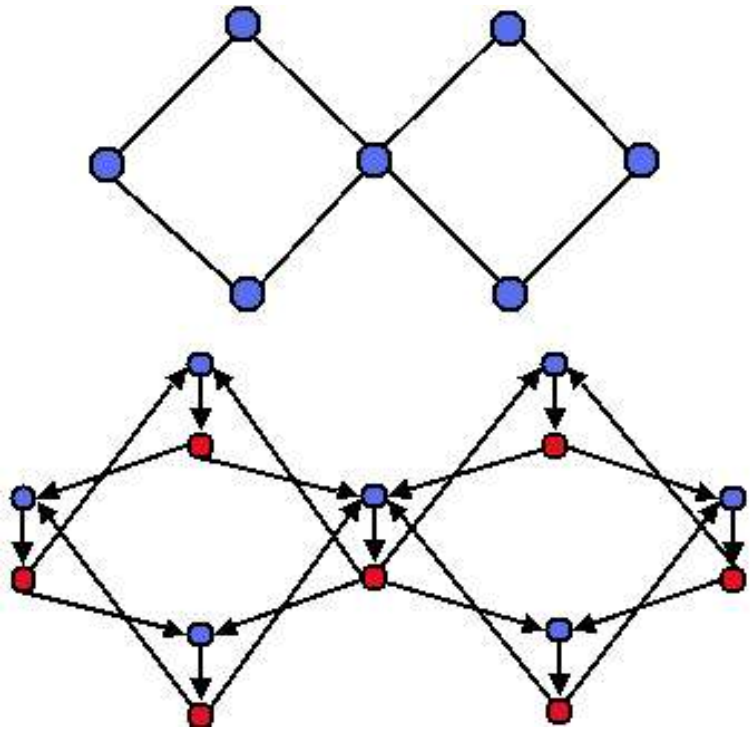


Figure 3.8: An undirected graph (above) and the version modified for maximum flow analysis.

Pseudo code showing the algorithm to determine the connectivity of a given graph is shown in Algorithm 3.6.

```
1.  Vertex-Con-MF(G)
2.   G': the new graph
3.   min: int representing the min value
4.   for each v in V(G)
5.        put vt & vr in V(G')
6.        put (vr,vt) in E(G')
7.   end for
8.   for each e in E(G)
9.        if e is undirected
10.            put (e.src+t,e.snk+r) in E(G')
11.            put (e.snk+t,e.src+r) in E(G')
12.       else
13.            put (e.src+t,e.snk+r) in E(G')
14.       end if
15.  end for
16.  set all capacities on E(G') to 1
17.  for each transmitter s in V(G')
18.       for each receiver t in V(G')
19.            if (s & t are not from the same v in V(G))
20.                 m = Ford-Fulkerson(G',s,t)
21.                 if (m < min)
22.                      m = min
```

```
23.                 end if
24.             end if
25.         end for
26.   end for
27. End Procedure
```

Algorithm 3.6: The maximum flow vertex connectivity algorithm.

3.2.3.2 Vertex Connectivity as Pure Search

Calculating the vertex connectivity of a graph by search is very similar to calculating the edge connectivity in the same way. A BFS tree has to be constructed rooted at every vertex in the graph, this forest of trees is then traversed (pre-order) in order to find all the paths from each of the root vertices to every other vertex in the graph. Once these paths have been found they are checked in order to see how many of them use unique vertices. The number of unique paths between each vertex and every other vertex is recorded and the minimum value is taken, this minimal value is the vertex connectivity of the graph being tested. A graph and its BFS tree can be seen in Figure 3.7 and the pseudo code for this method is shown in Algorithm 3.7.

```
1.  Vertex-Con-Search(G)
2.   min: int representing min value
3.   for each s in V(G)
4.        T = BFS_Tree(G,s)
5.        total: int representing number of paths s -> t
6.        for each t in V(G)
7.             while p = Pre-Order(T,t)
8.             if p is vertex unique
9.                  increment total
10.            end if
11.       end for
12.       if (total < min)
13.            min = total
14.       end if
15.  end for
16. End Procedure
```

Algorithm 3.7: The BFS based vertex connectivity algorithm.

3.2.4 Vertex/Edge Cut Sets

An obvious but interesting result from flow theory is that the minimum cut of a graph (the capacity across the smallest number of edges or vertices that have to be removed from the graph in order to disconnect it) is equal to the maximum flow. This is called the Max-Flow = Min-Cut theorem and it was proved by Feinstein and Shannon (and separately by Ford and Fulkerson) in 1956 [WIKI_MF]. This result provides a way of

algorithmically determining the minimal separating sets and cut sets of a given graph. Generally speaking there are two ways to approach this problem. The first is to build a list containing every combination of the edges or vertices and summing the capacity of each set, the set(s) that match the maximum flow would be identified as separating sets or cut sets. This however is a very inefficient way to go about determining what makes up these sets as it has a worst case running time of O((n-1)!). Much more efficient solutions to these problems can be found by using a greedy search algorithm based on a minimal packing routine (packing small items into larger boxes using the fewest boxes possible). If the maximum flow between two vertices v and w is already known then the minimum edge cut between these same two vertices will have that capacity and therefore it is just a case of going through edges on the flow augmenting paths and 'packing' their capacities into the maximum flow value. The same can be achieved for the separating sets by looking at the flows across the vertices and 'packing' those into the maximum flow. The pseudo code showing an outline of an algorithm to find a cut-set by greedy search can be seen in Algorithm 3.8, this can be easily changed to find a minimal separating set by looking at the vertices in the augmenting paths (other than s an t) and the flow across them.

```
1. Find-Cut-Set(G, s, t)
2.   max: max flow between s and t
3.   max = Ford-Fulkerson(G, s, t)
4.   cut: current cap of min edge cut
5.   cut-set: set of edges in Cut Set
6.   for each edge: e in FAP's between s and t
7.        if (max – cut is +ve)
8.             add e to cut-set
9.             cut += cf(e)
10.       fi
11.       if (max – cut == 0)
12.            break out of loop
13.       fi
14.  rof
15. End Procedure
```

Algorithm 3.8: Minimal packing cut set finder.

3.2.5 Minimum Spanning Tree

A minimum spanning tree (MST) is a subset of the edges of a graph that forms a tree which includes every vertex where the total weight of all the edges in the tree is minimal [WIKI_MST]. A graph and its corresponding MST is shown in Figure 3.4.

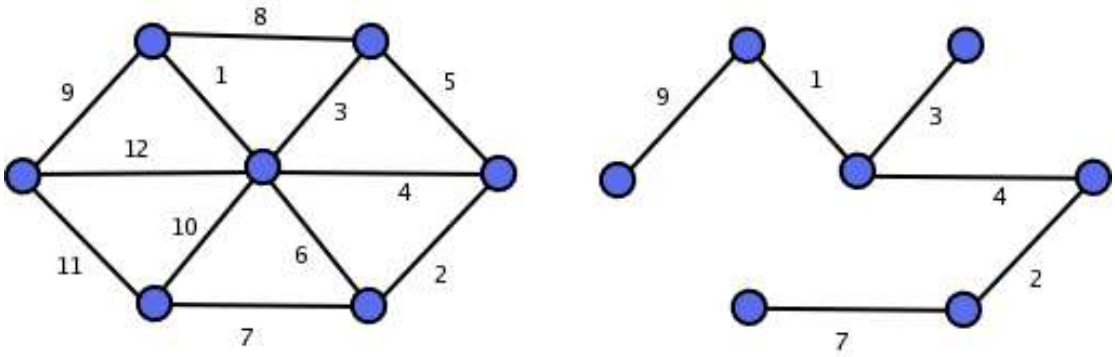


Figure 3.9: A graph and its MST[10]

There are several algorithms to calculate the MST of a given graph, the one looked a here is known as Kruskal's algorithm and it is an example of a greedy algorithm (takes the locally optimal choice at each step). Other algorithms that solve this problem include Prim's and Boruvka's however they will not be looked at here. Kruskal's algorithm produces an MST in the following way:

- It creates a forest F of trees in which each vertex in V(G) is in a separate tree.
- It creates a set S of the edges E(G) of the graph.
- While the set S is not empty,
    - It takes the edge with the smallest weight from S
    - If the edge connects two different trees in the forest F then it is added to the forest amalgamating the two connected trees into one tree
    - Otherwise the edge is discarded

Once the algorithm terminates providing its input was a connected graph the result will be a forest with only one component which is the MST for the input graph [WIKI_MST]. Pseudo code showing how the algorithm should be implemented can be seen in Algorithm 3.9.

```
1.  MST(G)
2.   F: forest of trees
3.   SE: list of sorted edges E(G), smallest cap first
4.   for each v in V(G)
5.        add a new tree rooted at v to F
6.   end for
7.   for each e in SE
8.        if e connects two distinct trees in F
9.             amalgamate trees into one tree
10.            remove original trees from F
11.            add new tree to F
```

[10]From Figs 4.5 & 4.6 pg 125 of [AGT]

```
12.       end if
13.       if F contains just one tree
14.            terminate for loop
15.       end if
16.  end for
17. End Procedure
```

Algorithm 3.9: Kruskal's algorithm to produce an MST

3.3 Graph Drawing

The visualisation of the graphs will be handled by an external software tool developed by AT&T Research Labs called GraphViz (http://www.graphviz.org). This tool consists of two programs for laying out graphs called 'dot' for digraphs and 'neato' for undirected graphs. They both read a common graph format and can output the resulting graph in image formats such as JPEG, GIF and PNG or markup languages like PostScript. A sample input file for dot or neato can be seen in Figure 3.3 and the graph resulting from this in Figure 3.2. The file format that is used by both of these programs is described in “Drawing graphs with dot” by Gansner, Koutsofios and North [DGWD].

```
digraph G {
     main -> parse -> execute;
     main -> init;
     main -> cleanup;
     execute -> make_string;
     execute -> printf;
     init -> make_string;
     main -> printf;
     execute -> compare;
}
```

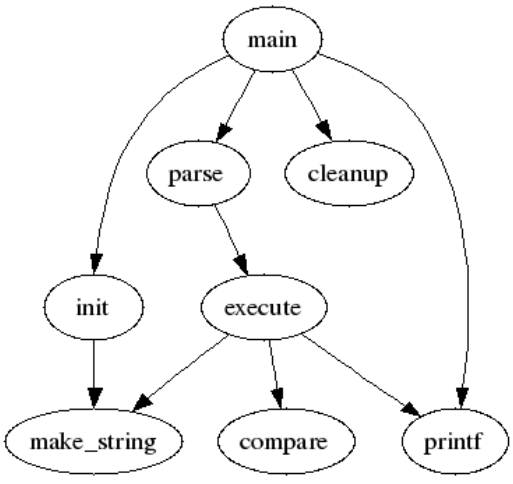


Figure 3.10: A small graph definition for dot[11].

Figure 3.11: The graph resulting from feeding Figure 3.10 through dot[12].

The GraphViz system also has support for more complicated graph formatting data including edge and vertex colouring, this will make it ideal for displaying the results of the algorithms in a visual fashion. It is simply a matter of producing an interface between the chosen development language and the dot and neato executables. The details of how dot and neato layout the graphs is of little consequence to this project, however if the reader wishes to find out they more are directed to [ATGD].

3.4 Graph Augmentation

The optimal number of edges that need to be added to a graph G in order to make it k-edge-connected was shown to be γ (by Cai and Sun's Theorem), this section of the document shows an algorithm derived from this theorem and the splitting lemma which aims to make a graph k-edge-connected where the current $\lambda(G)<k$. The first algorithm is shown in Algorithm 3.10, it prepares the target graph by adding a new vertex and enough edges to satisfy $d(X_i)\geq k$ for all $\emptyset = X_i \subset V(G)$. The graph will appear as in Figure 2.11 after this is complete. Algorithm 3.11 successively applies the splitting lemma until all the edges between s and the graph have been removed. Once this is complete the vertex s will be isolated and it can be removed from the graph, we are then left with a k-edge-connected graph which may contain parallel edges.

```
1.  Prepare-Graph(G, k)
2.   add a vertex s to G
3.   for every Xi such that ∅ = Xi ⊂ V(G)
4.        if d(Xi) < k
5.             add k – d(Xi) edges from Xi to s
6.        end if
7.   end for
8. End Procedure
```

Algorithm 3.10: Stage one of the augmentation technique

Each of the sub-partitions can be found by running a DFS algorithm rooted at every vertex and following the graph structure stopping at each new vertex found to count the number of out edges from that group (the new vertex and the previous vertices).

```
1.  Split-Graph(G, k)
```

[11]From Figure 1 pg 3 of [DGWD]

[12]From Figure 2 pg 3 of [DGWD]

```
2.   Stack: moves
3.   choose two vertices in G with edges to s
4.        apply the splitting lemma
5.        add action to the stack
6.        if the degree rule has been violated
7.             pop one move off the stack and try again
8.        end if
9.        if the method gets stuck
10.            pop moves off the stack until an
11.            alternative can be made.
12.       end if
13.  end loop
14. End Procedure
```

Algorithm 3.11: Stage two of the augmentation technique

3.5 Fulfilling the Project Objectives

Each of the topics that have been covered in this chapter answers either directly or indirectly one of the objectives of this project. Figure 3.1 shows which topic areas relate to each of the deliverables.

Level	***Objective***	***Algorithm or Technique***
Basic	Framework to implement algorithms	● Linked graph structure
	Connectivity algorithms	● 3.2.1 Maximum Flow ● 3.2.2.1, 3.2.2.2, 3.2.3.1 and 3.2.3.2 Edge/Vertex connectivity
Intermediate	GUI	● 3.3 Graph drawing
	Visual output of algorithm results	● 3.3 Graph drawing
	Weak-point analysis	3.2.4 MST ● 3.2.5 Cut-set finding
Advanced	Edge-connectivity augmentation	● 3.4 Graph augmentation
	Complete GUI	● 3.3 Graph drawing ● 3.4 Graph augmentation

Figure 3.12: Table showing objective traceability.

4 Design and Implementation

Chapter Overview

This chapter details how the theory that was researched and developed in Chapters Two and Three was implemented in a form executable on a computer. It will show the reader the structure of the program and how the disjoint pieces fit together to answer the original questions posed by this project.

4.1 Implementation Language

There are many computer programming languages and programming paradigms with different virtues and caveats. The programming language used to implement a problem has a great bearing on how the problem can be solved and what secondary problems are created along the way. For these reasons it is an important choice.

4.1.1 Possible Languages

There are many suitable language that the software for this project could be implemented in. The languages to be considered are:

- C++
- Haskell
- Java

4.1.1.1 C++

C++ is a fast object orientated language with compilers available for several platforms including (but not limited to) Windows, Linux and Unix. Although it is a high level language it makes use of pointers to memory cells and other low level features that can make programs written in the language hard to comprehend.

4.1.1.2 Haskell

Haskell is a functional programming language with features such as polymorphic typing and lazy evaluation [TCFP] (p10). Functional means that the entire program is one expression that is executed by evaluating that expression, the focus in this paradigm is centred on what gets computed rather than how it gets computed. Functional programming is a very different method from imperative languages such as C++ and Java and it usually yields code that is smaller and easier to understand.

4.1.1.3 Java

Java is an object orientated language built on a C type syntax, it has an extensive API (for more information see [JAPI]) of pre-built features that provides users with a large amount of functionality for little coding outlay. Particularly noteworthy is the GUI API which allows programmers to quickly and simply build usable interfaces and event driven programs.

4.1.2 Chosen Language

Java was chosen as the development language for several reasons, the most significant of which is the developers familiarity with the language as well as the ease of creating GUIs and interfacing with outside programs (like GraphViz).

4.2 Framework

The framework for the implementation of the graph algorithms has a structure as depicted in Figure 4.1. These diagrams have been laid out using the Unified Modelling Language (UML), for information on the UML see [UML].

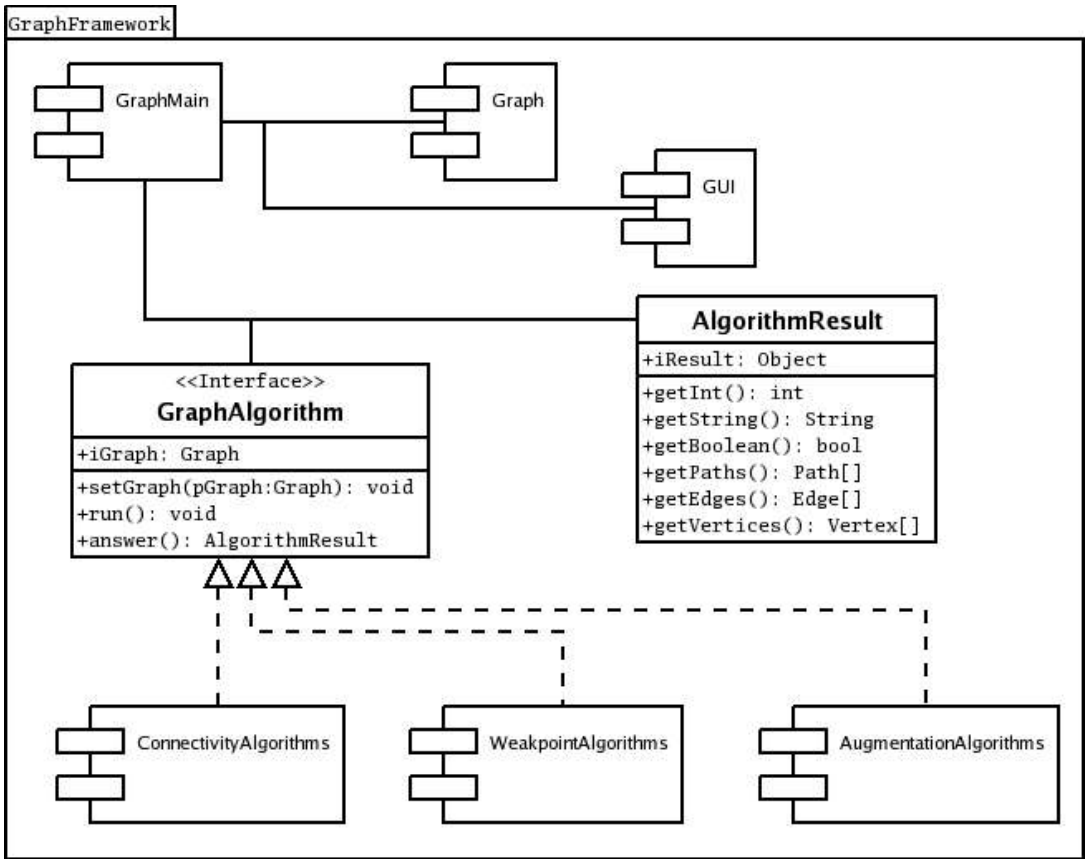


Figure 4.1: A UML diagram showing a high level overview of the system structure.

Each of the components in this diagram will be specified in further detail in the rest of the chapter.

4.2.1 Graph Component

This is responsible for loading a graph file from storage, interpreting that file and storing the graph (using the linked structure described in 3.1.1) and allowing access to the structure so that the rest of the program can function. The structure of this component is shown in Figure 4.2.

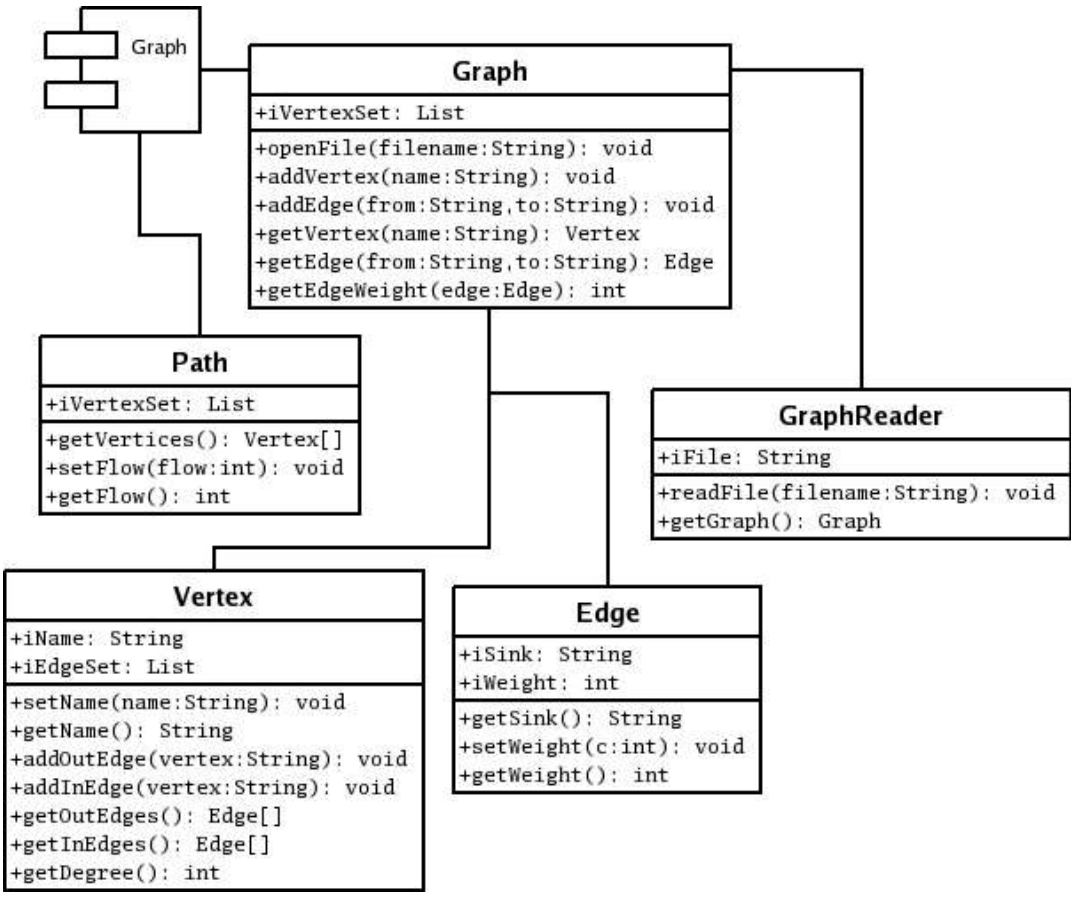


Figure 4.2: A UML diagram showing a high level overview of the Graph component.

4.2.1.1 Graph Class

This class maintains the vertex set and provides methods to access the graph vertex by vertex. Given a certain edge you can also find out what capacity for flow that edge has. The vertex set is kept in a hash table (provided by the java.util package) [JAPI]. A hash table is an array of linked lists, when an object is inserted into the table an integer is calculated from its key (in this case the name of the vertex). This integer is then modulo with the size of the array. The object is then inserted at this location in the array, if the slot is occupied (which will eventually happen) then list at that

location is searched, if the object already exists in that list then it is updated with the new object, if not it is appended to the end of the list [CJV2] (p136). Based on a good distribution of keys this makes the insertion and searching times favourable.

4.2.1.2 Vertex Class

The Vertex class is the object that is used to represent a single vertex in the graph a list of these objects is maintained in the iVertexSet in the Graph class. The class uses an ArrayList structure (provided by the java.util package) [JAPI] to store the edges incident with that vertex, this gives a list which can be expanded at runtime (unlike a static array Edge[]). The iEdgeSet array list contains Edge objects which themselves contain records saying what their sink vertex is, their direction and their weight.

4.2.1.3 Edge Class

Similar to the Vertex class, this object stores the information pertaining to a single edge. This information is stored in private variable fields and accessed using the methods the class provides. The stored information includes:

- `iSink` – the target vertex of the edge, accessed by `getSink()`
- `iWeight` – the integer capacity of the edge. accessed by `getWeight()` and `setWeight()`

4.2.1.4 Path Class

This class stores a set of edges that make up a path from one vertex (the source) to another vertex (the sink). This class is used to store the paths found by the maximum flow algorithms. In addition to storing the edges it also does some limited property testing which can inform the calling class if the path is a cycle or if it is an edge or vertex disjoint. The class also stores the maximum capacity of the path and the currently used level of flow.

4.2.1.5 GraphReader Class

This class uses reads in a graph definition file in the format described in section 4.2.1.5.1. It creates a new instance of the Graph class and populates its vertex set with the data held in the file. Once this process is complete the vertex set of this new graph replaces the vertex set of the Graph class that invoked the GraphReader.

4.2.1.5.1 File Format

The file format used to store the graph definitions is a simple ASCII (American Standard Code for Information Interchange) text system. The format has the grammar shown in Figure 4.3.

```
1. file = (statement newline)*
2. statement = vertex_stm | edge_stm
3. vertex_stm = vertex name
4. edge_stm = edge name name weight
5. vertex = “vertex”
6. edge = “edge”
7. name = [A..Z,a..z,0..9]+
8. weight = [1..9]+
```

Figure 4.3: The graph definition file grammar.

A sample graph definition file can be seen in Figure 4.4 along with a pictorial representation of the graph it defines in Figure 4.5.

```
1. vertex a
2. vertex b
3. vertex c
4. vertex d
5. edge a b 1
6. edge b c 1
7. edge c d 1
8. edge d a 1
```

Figure 4.4: A simple graph definition file.

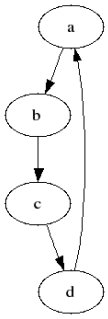


Figure 4.5: The graph Figure 4.4 corresponds to.

4.2.2 GraphAlgorithm Interface

This interface provides a template which all the graph algorithms conform to, the template is as follows, each implementing class provides:

- a void method: `setGraph(final Graph pGraph)`
  This gives the class the graph over which the algorithm it implements runs.
- a void method: `setParameters(final Hashtable pParams)`
  This gives the algorithm any starting data it might need, for example a maximum flow algorithm needs to know the source and sink vertex

- a void method: `run()`
  This runs the algorithm on the given graph and parameter set
- an AlgorithmResult method: `answer()`
  This returns an AlgorithmResult object containing the results of the algorithm.

More information about the AlgorithmResult class can be found in the next section.

4.2.3 AlgorithmResult Class

This class provides a way for a number of algorithms that may return different values ranging from a number or boolean to a set of paths a way of passing the data back to the calling class via a common return type. The class can store the following types of data:

- an Integer,
- a String,
- a Boolean,
- a set of Paths,
- a set of Edges,
- a Graph,
- or a set of Vertices.

An algorithm that returns an AlgorithmResult object simply calls the appropriate access function provided by the the class for the type of data it returns. When the calling function gets back from `answer()` it again calls the appropriate access function and retrieves the return result from the algorithm. If the program tries to retrieve a data slot that is empty then the access method throws an AlgorithmReturnTypeNotSupported exception.

4.3 GUI

4.3.1 GUI Component

The GUI component is responsible for creating the window interface that the program uses and reporting the results of the algorithms on a picture of the graph and in textual form. A high level overview of the GUI component's structure can be seen in Figure 4.6.

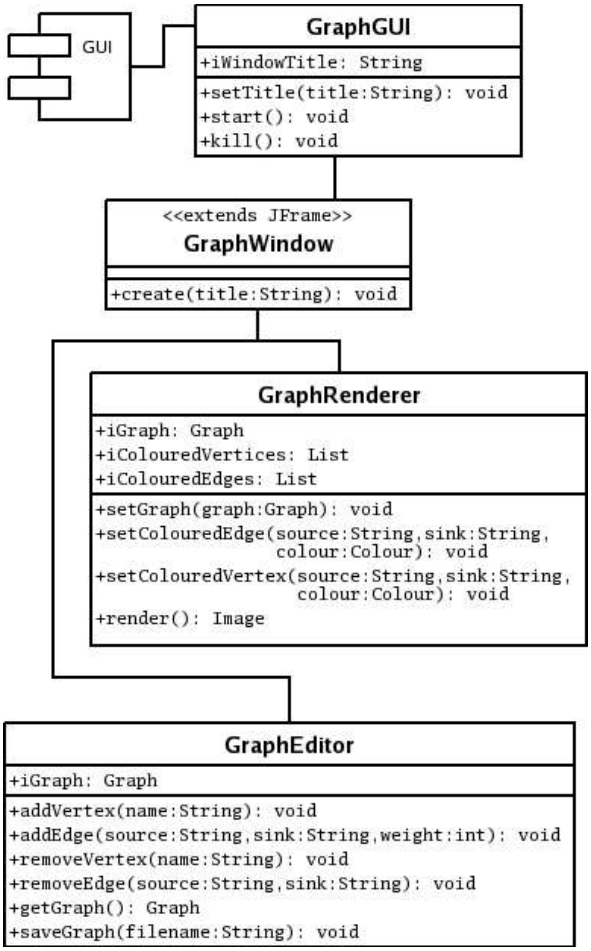


Figure 4.6: The UML showing a high level overview of the GUI component.

4.3.1.1 GraphGUI Class

This class ties together all the other GUI and logic objects. It provides an interface to load, edit and save graph definition files. Once the files are loaded connectivity and augmentation algorithms can be run on the graphs.

4.3.1.2 GraphWindow Class

This uses the Java Swing API (from the package javax.swing) [JAPI] to produce the interface through which users can interact with the program. The interface can be seen in Figure 4.7. The graph connectivity and augmentation algorithms can be accessed from the menu structure running across the top of the window, each of the menus can be seen in Figure 4.8. The results of the algorithms are shown in the text box to the left of the window and in some cases graphically in the large grey area on the right hand side. The graph can be edited using the controls to the bottom of the graph drawing area, these controls can be seen in more detail in Figure 4.10.

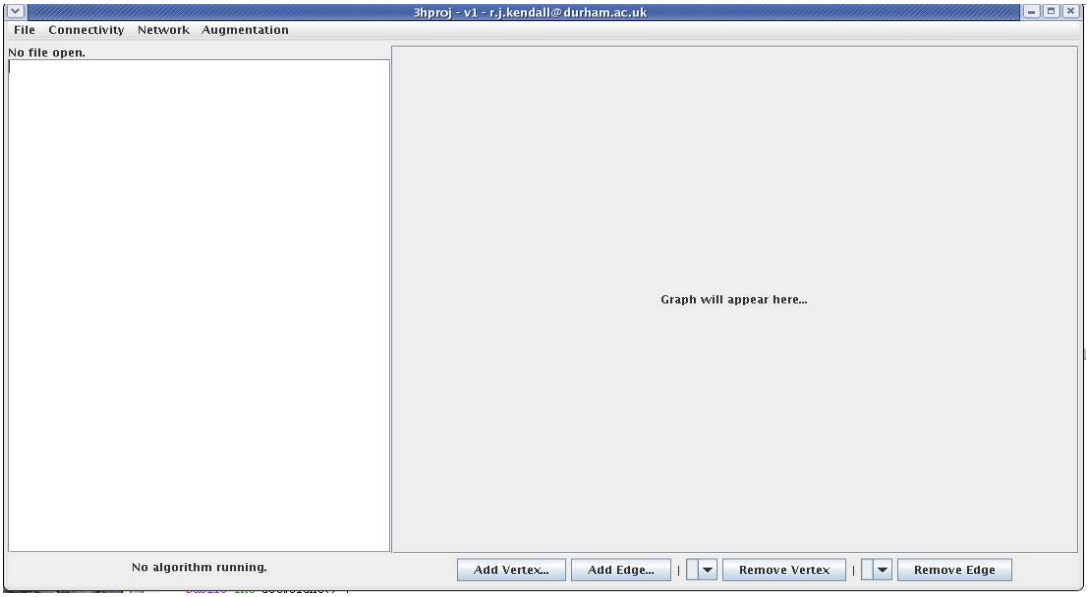


Figure 4.7: The main interface created by GraphWindow.

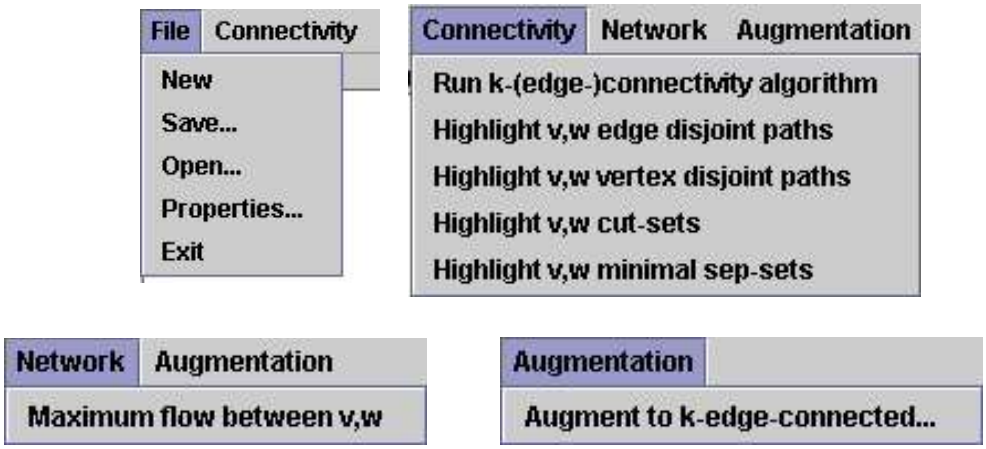


Figure 4.8: The GUI menus, used to access the functionality of the tool.

4.3.1.3 GraphRenderer Class

This class interfaces with the the GraphViz executable 'neato' (discussed in Chapter Three), renders the graph and then displays it in the graph panel on the main window (shown in Figure 4.7). Before neato can be run a dot format graph file has to be created for it to read. Any required formatting information (colour, size etcetera) is given to the class by GraphGUI after any algorithms have been run. Then once a call to render is made the following happens:

- The method runs through the edges and writes them to the file, if the edge has associated formatting data that is also written to the file.
- The method runs through the vertices and writes them to the file, again if the vertex has any associated formatting data that is also written to the file.

- The method then creates a Process (java.lang.Process) [JAPI] and executes:
  `$> neato -Tjpeg`
  This tells neato to output the graph in JPEG format to STDOUT, this output is then read by the method and read into an ImageIcon (java.awt) [JAPI] object which is then displayed in the graph display area.

The output from this procedure can be seen in Figure 4.9.

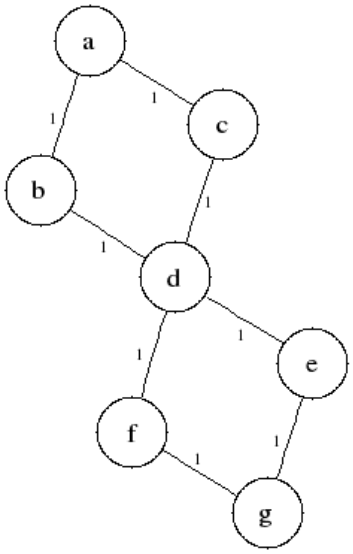


Figure 4.9: Sample output from the GraphRenderer class.

4.3.1.4 GraphEditor Class

This object provides the interface and the logic for editing the graph files, As the images displayed by the program are just static images generated 'on the fly' it is not possible to manipulate the graph by having users clicking the vertices and edges to edit them and so the controls shown in Figure 4.10 are provided for editing the graph.

Figure 4.10: The GraphEditor interface.

When a graph is loaded into the program, the drop-down boxes to the left of the 'Remove Vertex' and 'Remove Edge' buttons are populated with the vertices and edges in the graph. Removing an edge simply deletes that edge from the current graph, whereas, removing a vertex deletes that vertex and all the edges incident with that vertex. When the 'Add Vertex...' button is clicked it invokes the edge adding code which opens a window as seen in Figure 4.11 asking the user to input the name of the vertex, if a vertex with this name does not already exist it is added to the graph model

otherwise the user is informed of the conflict. A similar process occurs when an edge is added to the graph. The window opened by the clicking the 'Add Edge...' button can be seen in Figure 4.12.

Figure 4.11: The vertex name request dialog box.

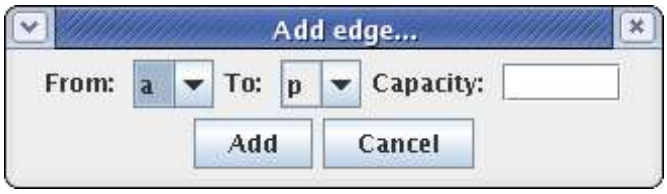


Figure 4.12: The add edge dialog box.

Changes that are made to the graph using the GraphEditor object can be saved using the 'Save Graph' option available in the 'File' menu provided by the GraphGUI object. It is also possible to create a graph from scratch using this system by adding vertices and edges to the empty graph.

4.4 Connectivity and Maximum Flow

4.4.1 Connectivity Algorithms

All the connectivity algorithms implement the GraphAlgorithm interface as described in 4.2.3. The structure of the connectivity algorithms is shown in Figure 4.13.

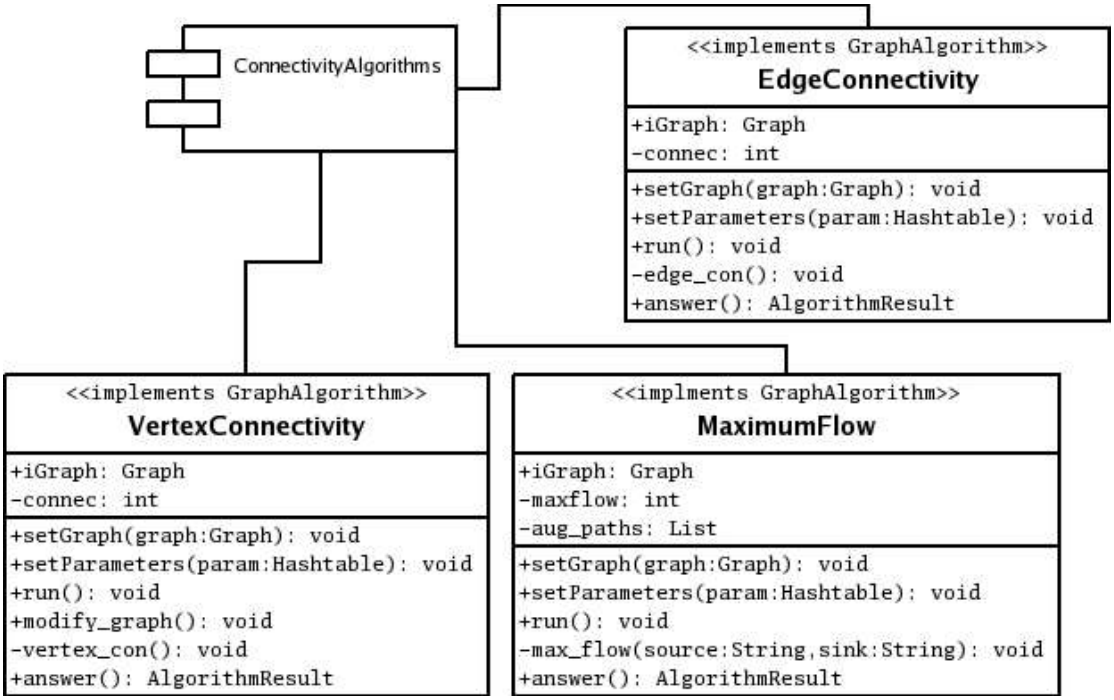


Figure 4.13: The high level overview of the connectivity algorithms.

4.4.1.1 MaximumFlow Class

This class calculates the maximum flow between a given source and sink vertex. The source and sink are given to the algorithm in the Hash table given to the setParameters method. The class uses the Ford-Fulkerson method with an Edmonds-Karp extension as described in Chapter Three. When the run method is called it calls the private method `max_flow()` which finds the flow augmenting paths between the source and sink by BFS and stores the paths found and the maximum flow value in aug_paths and maxflow respectively. The AlgorithmResult object made when `answer()` is called is given both the set of paths and the maximum flow value. Sample output from this algorithm can be seen in Figure 4.14.

```
Running Ford-Fulkerson maximum flow algorithm...
complete; time: 0 seconds.

Maximum flow from a to b: 4; flow augmenting paths:
        [*] [a, b]
        [*] [a, c, d, b]
        [*] [a, e, f, b]
        [*] [a, i, j, b]
```

Figure 4.14: Sample output from the Ford-Fulkerson implementation

4.4.1.2 VertexConnectivity Class

This class calculates the k-connectivity of the graph passed to it in the `setGraph()` method. It uses the maximum flow technique as described in 3.2.3.1 to determine the value of k, when the `run()` method is called the following occurs:

- `modify_graph()` is called to alter the graph so that it resembles the structure shown in 3.2.3.1 (i.e. all the vertex disjoint paths are represented by edge disjoint paths) also all the edge weights are set to one
- the maximum flow routine is run between every pair of distinct vertices and the smallest flow value is stored in the integer 'connec'

Once the `answer()` method is called an AlgorithmResult object containing the value in 'connec' is passed back to the calling method.

4.4.1.3 EdgeConnectivity

This class calculates the k-edge-connectivity of the graph passed to it in the `setGraph()` method. Like the VertexConnectivity class it uses a maximum flow technique to determine the value of k, when the `run()` method is called the following occurs:

- `modify_graph()` is called to alter the graph so that the weight on all the edges is one.
- the maximum flow routine is run between every pair of distinct vertices and the smallest flow value is stored in the integer 'connec'

The answer is passed back to the calling method in exactly the same way as the VertexConnectivity class.

```
Running k-(edge-)connectivity algorithm...
...complete; time: 0 seconds.

Results:
        Edge-connecivity: 4
        Connectivity: 4
```

Figure 4.15: Sample output from the Vertex/Edge connectivity implementation

4.4.2 Weak point Algorithms

In order to find the edges that need to be removed in order to separate the graph into two components such that a given vertex (the source) is separate from another given vertex (the sink) one can exploit the relationship between the maximum flow between

the source and sink and the capacity of a minimum cut i.e. that they are the same. The structure of the weak point algorithms component is shown in Figure 4.1.

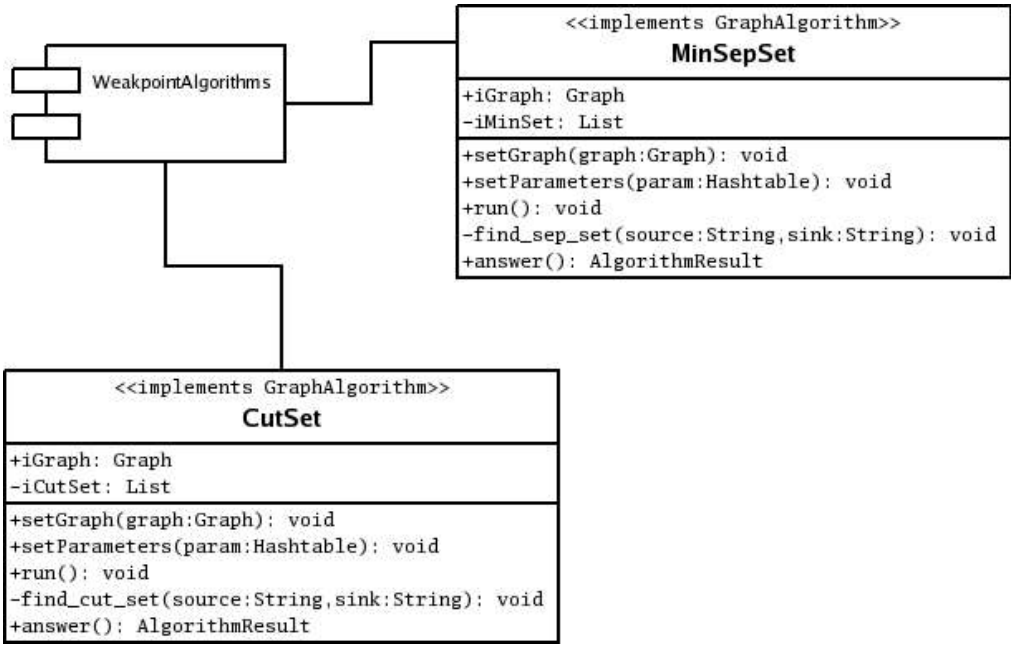


Figure 4.16: The high level overview of the weak point algorithms.

4.4.2.1 CutSet Class

This class determines the maximum flow between the source and sink vertices (given to it through the `setParameters()` method) and then it retrieves the augmenting paths and the maximum flow value from the AlgorithmResult. Once this has been done it iterates through the flow augmenting paths 'fitting' the largest edges into the flow until the remaining flow is zero. The edge set is given to the AlgorithmResult class in order to be passed back to the calling method when `answer()` is called.

4.4.2.2 MinSepSet Class

This class uses the maximum flow algorithm to find the maximum flow between the source and the sink vertex. Once the flow augmenting paths have been identified the algorithm iterates through the vertices on these paths until no more can be 'packed' into the maximum flow. This vertex set is passed back to the calling method in an AlgorithmResult object.

4.5 Optimisation Algorithms

4.5.1 Augmentation Algorithms

This set of algorithms is responsible for increasing the edge-connectivity of a given graph to a pre-defined level. The high level structure of this component can be seen in Figure 4.15.

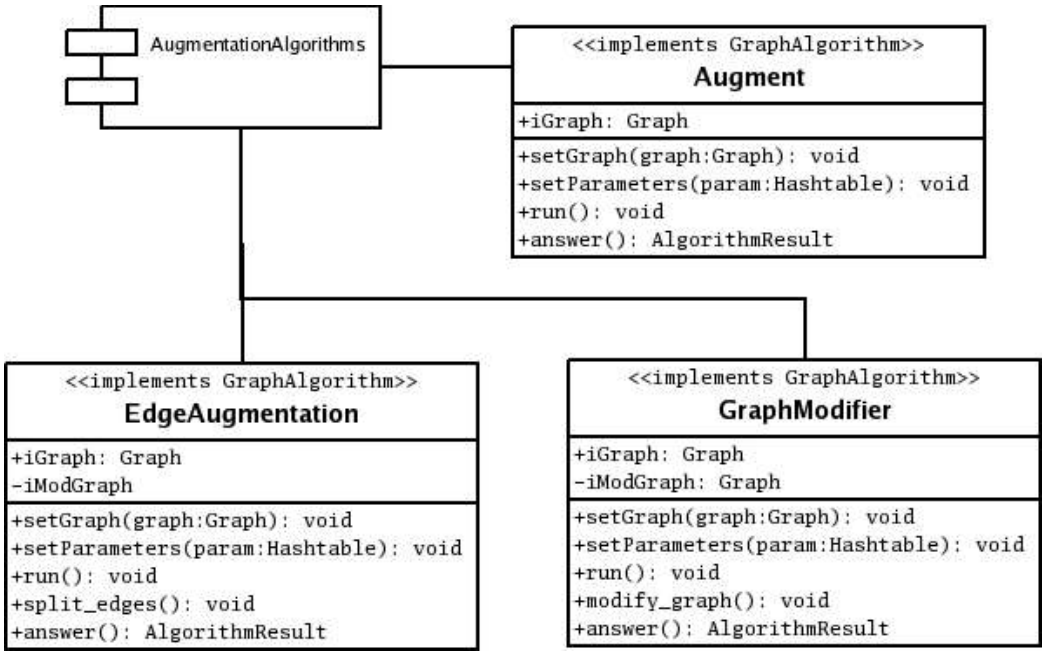


Figure 4.17: The high level overview of the augmentation algorithms.

4.5.1.1 Augment Class

This class knits together the two stages of the augmentation process, stage one is by the GraphModifier class and stage two is implemented by the EdgeAugmentation class. Once the run method is called the graph is passed to the GraphModifier which edits the graph as described in section 2.6.1. Once this is complete the graph is returned to the Augment class by way of an AlgorithmResult object, this graph is then passed to an EdgeAugmentation object which repeatedly applies the splitting lemma to the new graph until the vertex 's' is isolated and can be removed.

4.5.1.2 GraphModifier Class

This class takes the graph passed to it in the `setParameters()` method and adds a vertex 's' to it (using the `addVertex()` method in the Graph class). Once the vertex has been added enough edges are added from the original graph to s so that every sub-partition of the vertex set has at least k outgoing edges (the integer k is also passed into the class using the `setParameters()` method). The calling method can then retrieve the modified graph via the `answer()` method.

4.5.1.3 EdgeAugmentation Class

This class takes the modified graph and applies the splitting lemma to it until the new vertex 's' is isolated and can be removed from the graph. The vertices between which to add edges are chosen at random from each sub-partition X_i of V(G) with a

connection to s. It is possible that the algorithm will choose two vertices that 'trap it' and from there on it cannot eliminate all the edges incident with s. For this reason the choices that the algorithm makes are stored on a stack, if the algorithm reaches a point at which there are still edges incident with s but no more that can be removed with the splitting lemma. If this happens the algorithm can work backwards through the moves that it has made trying alternative routes. Once this is done the calling method (the Augment class, `run()` method) can access the new k-edge-connected graph through an AlgorithmResult object returned from the `answer()` method.

5 Results and Evaluation

Chapter Overview

This chapter will show the reader what was achieved during the implementation phase of this project, it will also attempt to show 'how good' the implementation is by devising several evaluation criteria and applying them to the solutions created for this project.

5.1 Results

Here are the results of the project implementation. The main issue in presenting these results is the correctness of the algorithms particularly of the ones which bring together a few algorithms, each of which is dependent on the output of another.

5.1.1 Connectivity Algorithms

In order to show the correctness of the connectivity algorithm several specially engineered test cases were passed through it and the output scrutinised to see if it was correct. The test cases were designed to be 'extreme' such as disconnected graphs to test that the algorithm produces the correct output. The graphs that were used as test cases are a disconnected graph and several complete (every vertex is adjacent to every other vertex).

5.1.1.1 Disconnected Graphs

The connectivity and edge-connectivity of a disconnected graph should be zero as at least one of the vertices in the graph is adjacent to no others. The algorithm returned the correct result of zero for both the edge connectivity and connectivity of such a graph.

5.1.1.2 Complete Graphs

Complete graphs on n vertices, known as K_n [PGT] have an edge-connectivity and a connectivity of n-1 [PGT]. The algorithm was tested using ten complete graphs on twenty to two hundred vertices inclusive at twenty vertex intervals, for every one of these graphs the algorithm returned the correct result of n-1.

5.1.2 Graphical User Interface (GUI)

The GUI was built as described in Chapter Four and integrated with the GraphViz drawing system. A selection of graphs drawn by the GUI can be seen in Figure 5.2, Figure 5.3 and Figure 5.4. More of these graphs can be seen in Appendix I. One

problem was encountered with the GraphViz system in that if the graph contains a large number of edges then the 'neato' system which draws nicely laid out and compact graphs fails and the less sophisticated 'dot' renderer has to be used. With large graphs this can sometimes lead to confusing images such as the one depicted in Figure 5.1. This however is a problem with the GraphViz program itself and it does not fall under the purview of this project to correct this problem. One 'work-around' was found during the implementation, this was to remove the capacity indicators from the edges however this produced graphs so tightly packed that they were almost impossible to read.

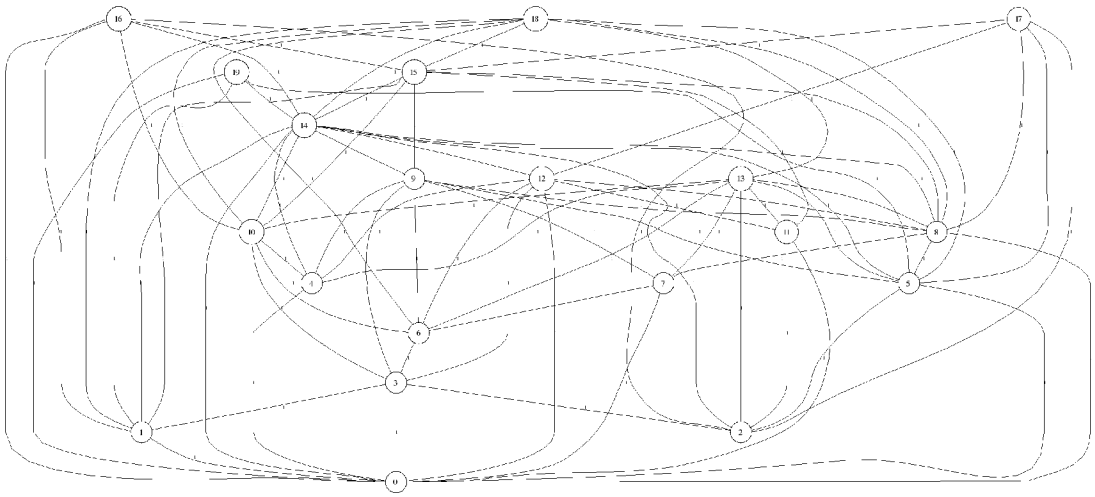


Figure 5.1: A random graph on 20 vertices as laid out by 'dot'.

5.1.3 Cut Set and Separating Set Algorithms

Showing the correctness of the cut and separating set finding algorithms can be done by creating graphs which always have certain edges in the cut sets and vertices in the minimal separating sets.

5.1.3.1 Separating Sets

The graph that was used to test the algorithm can be seen in Figure 5.2. Clearly any separating set between two distinct vertices of this graph must include the vertex 's' as it is adjacent to every other vertex in the graph. This is the case and all the results of the algorithm running on this graph can be seen in Appendix II.

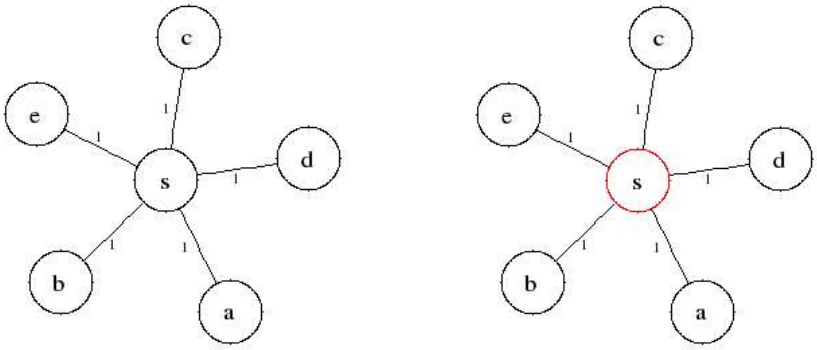


Figure 5.2: The cut-set test graph (left) and an a,b cut-set (right).

5.1.3.2 Cut Sets

The correctness of the cut-set algorithm can be shown with complete bipartite graphs. These are bipartite graphs in which there are n vertices in the first vertex subset and m vertices in the second vertex subset, for which the usual notation is $K_{n,m}$. The graph used for testing here is a $K_{3,3}$, it can be seen in Figure 5.3. Any cut-set between two vertices on each side of the graph is going to have the smaller of n and m edges, if the vertices are both on the n side then the number of edges will be n likewise for m. The implemented algorithm finds the correct cut-set of the correct size each time for both a $K_{3,3}$ and a $K_{5,3}$. The results are presented in Appendix II.

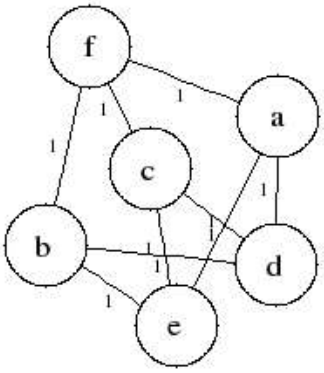


Figure 5.3: A $K_{3,3}$

5.1.4 Augmentation Algorithms

The results of the augmentation of a tree to a 3 edge-connected graph can be seen in Figure 5.4.

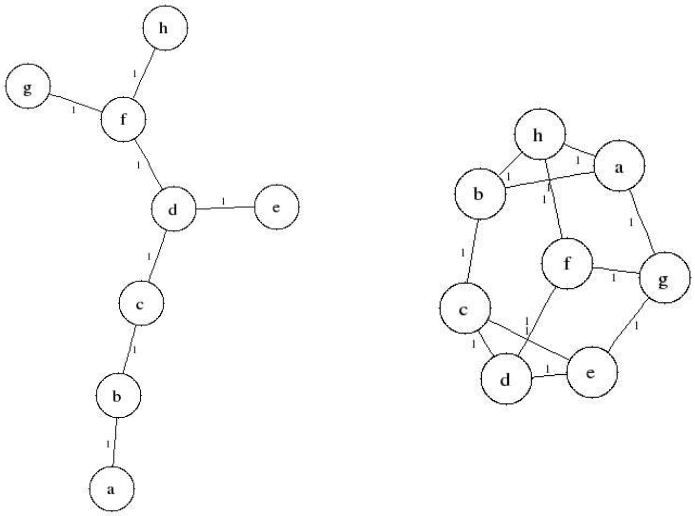


Figure 5.4: A 1 edge-connected graph and its counterpart augmented to be 3 edge-connected.

As the algorithm has been implemented in such a way that DFS is used to find the sub-partitions of the vertex set it has problems with input graphs which contain cycles and as trees are far from the usual input the algorithm takes this requires that any input graph be converted into a tree by way of Kruskal's algorithm. This graph is then augmented to the desired level of edge-connectivity and output.

It is difficult to prove the output from the algorithm correct, as it is a deterministic method which exactly determines the number of edges which need to be added and then adds those edges. Therefore there is no question of the quality of the output, just the correctness of the output. Each of the test graphs that have been run through the augmentation algorithms have be augmented correctly.

5.2 Evaluation

This section will look at the quality, feasibility and flexibility of the implemented solutions rather than just the question of validity answered by the first half of this chapter. This section will explore how quickly the algorithms find an answer and therefore how feasible it is that they be run on non-trivial inputs, how easily the algorithms may be adapted in order to answer subtly different questions and therefore how flexible they are and it will look at how robust they are and thus their quality. The graphical user interface will not be considered by this evaluation as it is not a central part of the project and is simply used to provide the results of the algorithms that have been implemented through the course of this project.

5.2.1 Execution Times

The execution times of the algorithms were determined by recording the time in milliseconds (using the System.currentTimeMillis() call [JAPI]) before and after the algorithm was run and then the difference between the two was the time the algorithm took to run. All the tests were carried out on a Intel Pentium 4 based PC with a CPU clock speed of 2.40Ghz. A small Java program was constructed to produce random test case graphs based on a number of vertices and a number of edges being input.

5.2.1.1 Maximum Flow Algorithm

Because the maximum flow algorithm forms the basis of the connectivity finding algorithms the running time of this method will be explored here first. The program that was introduced in the previous section was used to produce complete graphs (K_n) on twenty to two hundred vertices incrementing in steps of twenty vertices and then these graphs were fed into the algorithm and the maximum flow between the first and last vertices was calculated. This process was also performed for random graphs R_n (the edges are placed randomly between n vertices) on the same numbers vertices with n*4 edges. The running time for each of these inputs was recorded and can be seen plotted on a graph in Figure 5.5.

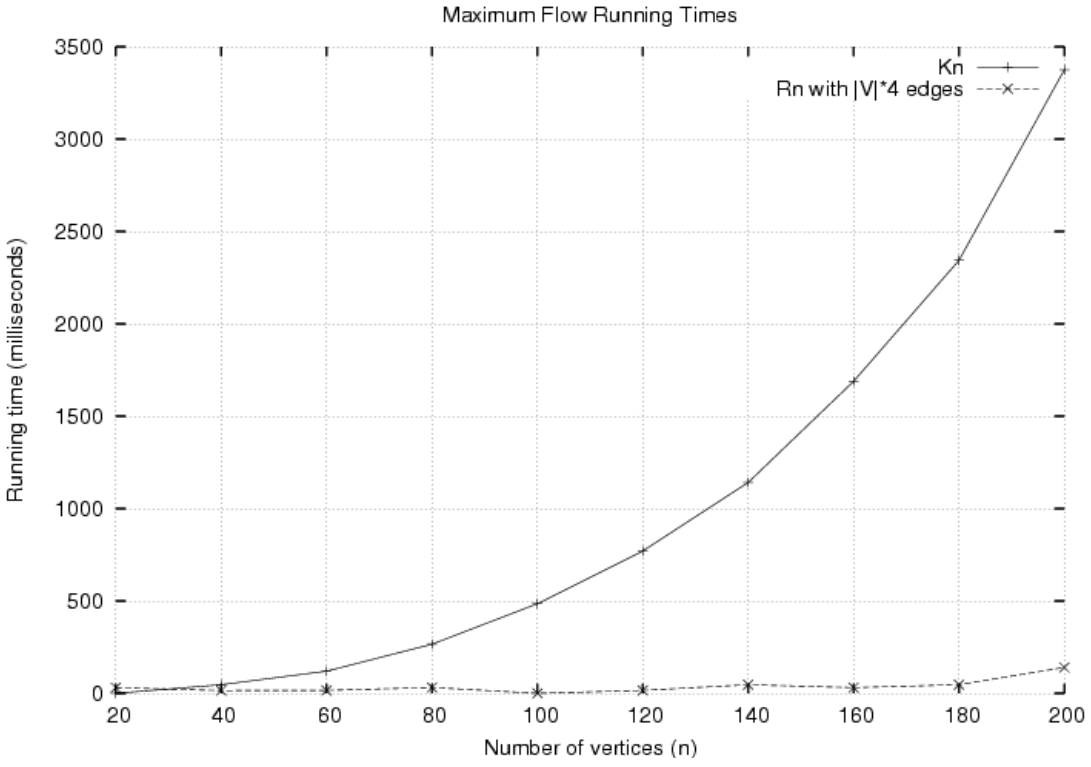


Figure 5.5: A graph showing the running times of the maximum flow algorithm.

The graph shows that the number of vertices and edges in a graph has little impact on the running time of the maximum flow method, rather the way they are organised dictates how long it will take until the maximum flow is found. The line representing the time taken by the algorithm on the R_n class has no relationship to the number of vertices, however, it does with the K_n's. Therefore the running time of the algorithm must be dependent on one of the factors where the R_n and K_n graphs differ: the number of edges and the connectivity:

- Number of edges in a K_n is n2-n
- Edge-connectivity of a K_n is n-1 and the connectivity is also n-1.

The particular variation of the Ford-Fulkerson method used by this program is called the Edmonds-Karp algorithm which works by always finding the shortest (in terms of number of edges) path when it is augmenting the flow, it does this by running a BFS search algorithm that takes $O(n)$ (where n = |V|) time to find an augmenting path (if one exists). Each time the flow is augmented one edge in the graph is saturated, as each edge can be saturated n/2 times during the execution of the algorithm there are n*m/2 (where m = |E|) edge saturations in total and this gives a running time for the algorithm of $O(n*m^2)$. This is an upper bound on the running time of the algorithm, however, practice shows that the edge-connectivity (bounded by the number of edges) plays a part in the running time of the algorithm. The more edge disjoint paths that exist between any two vertices the longer it will take the BFS method to find them all and the algorithm is pushed to its worst case running time. That effect can be seen in the results presented here (Figure 5.5) as the K_n class of graphs have high edge-connectivities (n-1) and calculating the maximum flow on these graphs took much longer than for their random counterparts on the same number of vertices.

So while it has been shown that complete graphs make the program exhibit the worst case behaviour of the Edmonds-Karp algorithm on general cases the method works reasonably well producing answers fairly quickly.

In terms of how good this is there were no algorithms located during the research for this project that can find the maximum flow in a time better than $O(n*m^2)$ that will work with all values of of n and m, however [AFD] describes a deterministic algorithm that runs in $O(n*m)$ time for all m > n + c (for some constant c).

5.2.1.2 Connectivity Algorithms

As with the maximum flow tests a similar program was used to generate several random test graphs on ten to one hundred vertices inclusive at ten vertex increments, they were produced with n*2 edges and n*4 edges. The connectivity algorithms were run on each of these graphs and their running times recorded. This data can be seen plotted on a graph in Figure 5.6. The graph shows that while the number of vertices in a graph affects the running time of the algorithm the number of edges is the overall deciding factor and that is due to the underlying algorithm that is being used to find the edge and vertex disjoint paths. The algorithm being used is the Edmonds-Karp maximum flow method which has a running time of $O(n*m^2)$, in the case of the edge connectivity it has to be run between every pair of distinct vertices which makes the running time of the edge-connectivity algorithm:

$$
\begin{aligned}
&= O(n^2 * (nm^2)) \\
&= O((nn^2) * (n^2 m^2)) \\
&= O(n^3 * n^2 m^2) \\
&= O(n^5 * m^2)
\end{aligned}
$$

In order to calculate the connectivity of the graph using the maximum flow technique the graph has to be modified as described in section 3.2.3.1, this increases the number of edges in the graph by n+2m, n for the edge connecting each receiver and transmitter and 2m for the two edges that replace each undirected edge in the graph. So each execution of the maximum flow algorithm will take $O(n*(n+3m)^2)$ which means the whole execution will take $O(n^5*(n+3m)^2)$ and in total calculating both the vertex and edge-connectivity will take:

$$
\begin{aligned}
&= O(n^5 * m^2) + O(n^5 * (n+3m)^2) \\
&= O(n^5 * m^2 + n^5 * (n+3m)^2) \\
&= O(n^5 * (n+3m)^2)
\end{aligned}
$$

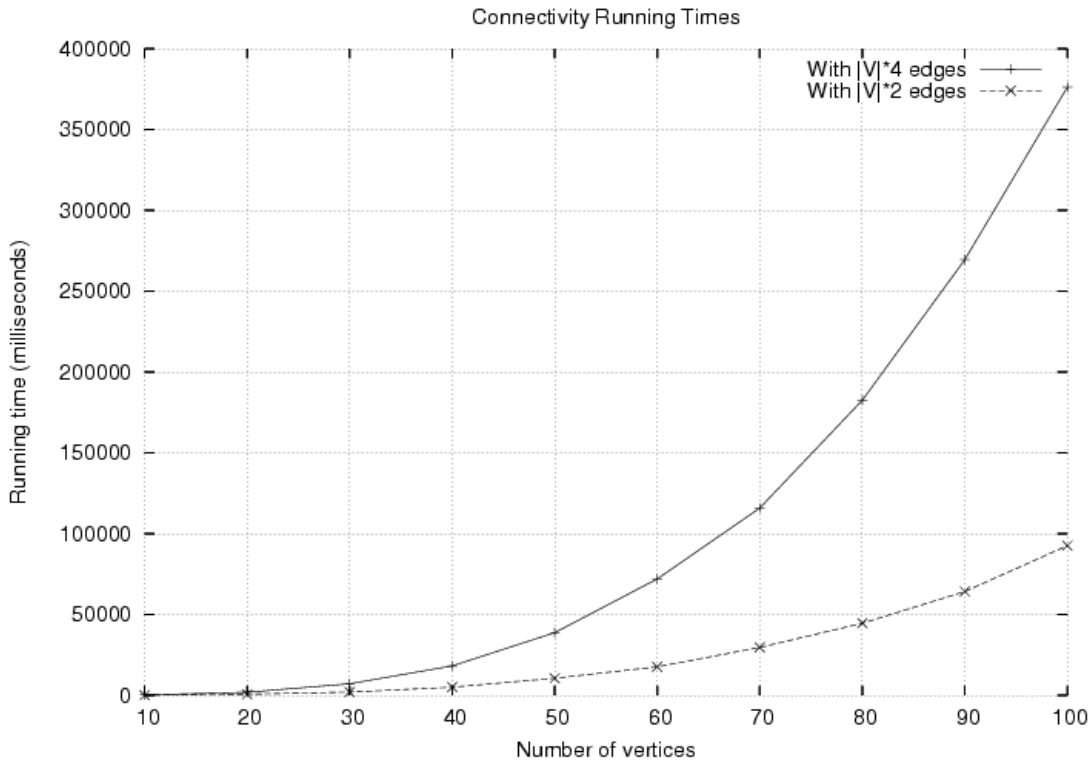


Figure 5.6: A graph showing the running times of the connectivity algorithms.

While theoretically this time complexity is considered viable (as it does not include any powers dependent on the input size) it is not really practical as when the input graphs reach any kind of useful size the algorithm takes a long time to run, for example with a graph of one hundred vertices and four hundred edges it took nearly four hundred seconds.

In comparison to other work this solution is very slow as there are algorithms that can find the edge connectivity of a graph in $O(nm)$ time, however, they are not flow based nor can they calculate the vertex connectivity by a similar method.

5.2.1.3 Augmentation Algorithms

The two algorithms that make up the augmentation mechanism, the graph modifier and the splitter (described in section 3.4) will be treated separately here.

The graph modifier adds a single vertex 's' to the graph and then enough edges from each sub-partition of V(G) such that each one of those sub-partitions has at least k outgoing edges. The algorithm is implemented in such a way that it finds the sub-partitions of the vertex set by DFS, an algorithm that runs in $O(n+m)$ time (n and m as defined as before). This algorithm is run rooted at each vertex in the graph and therefore the running time in total is $O(n*(n+m))=O(n^2+nm)=O(n^2)$.

The splitter then runs through the graph created by the modifier to replace pairs of edges between the original vertices and 's' with an edge between the two original vertices. As was shown in Chapter Two, Cai & Sun's theorem states that this will be no more than γ edges where $\Sigma(k-d(X_i))\leq 2\gamma$ holds for all the sub-partitions of V(G). Therefore as the method needs to loop γ times the optimal running time should be $\Omega(\gamma)$, however, this would only be the case if a perfect heuristic existed for choosing which edges to split at each stage. As it is a perfect heuristic was not discovered during this work and therefore the algorithm has to rely on a recursive backtrack technique when it gets stuck (i.e. there are still edges to split off but none that can be done without adding parallel edges). In the worst case therefore it is possible that every combination of 2 edges from the 2γ edges between V(G) and 's' has to be tried which could lead to a running time of:

$$O\left(\frac{2\gamma!}{(2\gamma-2)!}\right)=O(2\gamma!)$$

5.2.1.4 Remaining Algorithms

The remaining algorithms which find the cut and separating sets between two vertices in a graph are dependent upon one execution of the maximum flow algorithm. It has already been shown that the running time of the maximum flow method is $O(nm^2)$. Therefore as the algorithms run through each of the flow augmenting paths which is bounded by the number of edges the running time is $O(nm^2)+O(m)=O(nm^2)$.

5.2.2 Algorithm Flexibility

This section will assess how easy it is to adapt the implemented algorithms to perform different but related tasks to the one they were designed to perform. This is done based on the idea that code reuse is a common form of efficiency increase in development and the more of this code that can be re-used the better.

5.2.2.1 Algorithm Framework

The object-orientated design methodology was very much at the heart of the process which went into devising the code and structure behind this project. The system was designed so that it would take very little effort to add new algorithms and therefore functionality to the system, this was achieved by:

- Creating a template which all the algorithms followed
- Producing a common input interface so that the input data the algorithms needed could be accessed in the same way

- Producing a common output interface so that the output data from the algorithms could be extracted in the same way

This coupled with the abstraction of the graph drawing to a separate class allows for the addition of new algorithms with little or no code dealing with passing data around and allows the programmer to concentrate on the algorithm itself.

5.2.2.2 Maximum Flow Algorithm

In order to assess how flexible the implementation of this algorithm is this evaluation is going to look at how it can be modified to work with multiple sources and sinks. This can be done simply by adding two additional vertices to the graph, a source vertex adjacent to each of the vertices in the source group and a sink vertex again adjacent to each of the vertices in the sink group. If each of the edges incident with the two new vertices have their capacities set to that of the edge with the largest capacity in the graph then the maximum flow value will be correct. It is easy to prove that if the value of the largest capacity is c then the maximum flow between any two points of the graph is at most $c*|E|$. The manner in which the graph needs to be modified can be seen in Figure 5.7, and as both the connectivity and augmentation algorithms make extensive use of algorithmic graph modification it would not be at all hard to modify the maximum flow algorithm to add the two new vertices and the corresponding edges.

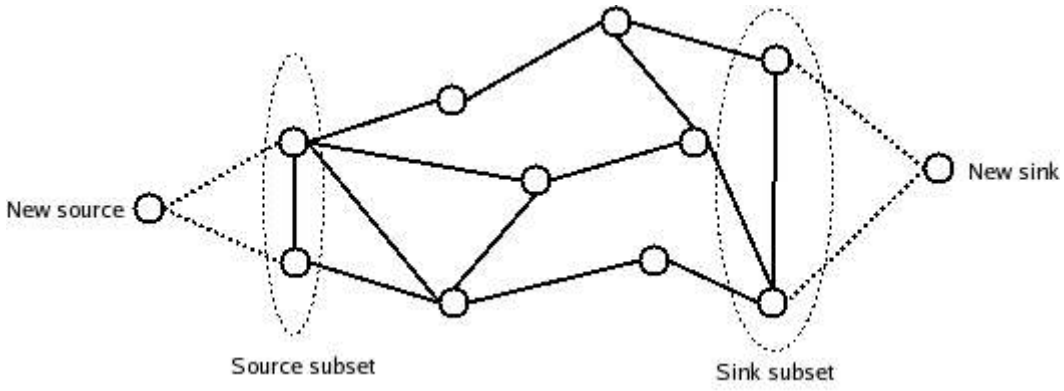


Figure 5.7: A diagram showing how a graph can be modified for multiple source and sink maximum flow.

5.2.2.3 Cut and Separating Set Algorithms

Currently the algorithms that find the cut and separating sets between two vertices a and b first find the flow augmenting paths working on the principle that one edge or vertex should be removed from each of the paths (one edge or vertex may be shared between more than one path). For any given pair of vertices there may be more than one cut or separating set, this is shown in Figure 5.8 and Figure 5.9.

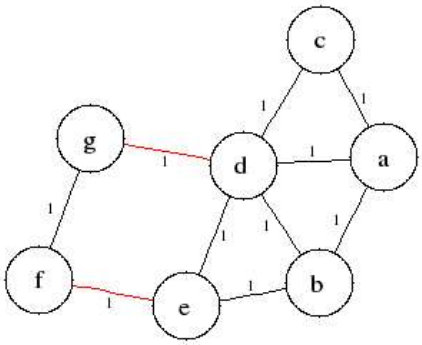


Figure 5.8: A graph and an a,g cut-set highlighted in red.

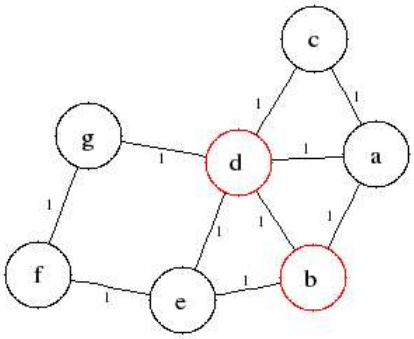


Figure 5.9: A graph and an a,g separating set higlighed in red.

The figures above show one possible a,g cut and seperating set but in fact there are two cut-sets and three seperating sets. As the algorithms purpose is to show any potential weaknesses in a given network structure it should be acknowledged that if one cut or separating set is augmented another one may be left behind. It would therefore it would be helpful to enumerate all the possible cut and seperating sets beween any two given vertices. This could be done with the current implementation by repeatedly running the algorithm, and when a cut or separating set is found recording the vertices or edges so they cannot be chosen again and then running the algorithm again. This would keep going until no more cut or separating sets were found.

As with the maximum flow algorithm it would also be useful to know which edges or vertices would have to be removed in order to disconnect a subset of vertices from another disjoint subset of vertices, this can be done by running the cut and seperating sets algorithms on the output from the maximum flow algorithm as modified in section 5.2.2.2.

5.2.2.4 Augmentation Algorithm

In many ways the algorithms that augment the edge-connectivity of a graph are the most inflexible of all those implemented for this project. As the initial stage uses DFS to locate all the sub-partitions of a graph it has a propensity to get stuck on any cycles that may exist in the input graph. The solution to this problem was to make a copy of the graph and run Kruskal's algorithm on that copy to produce a tree (which by definition contains no cycles) and then augment that to the level of edge-connectivity that has been requested. After this has been done only the new edges which were not present in the original graph are added to the graph. This increases the running time of the algorithm.

5.3 Overall Achievement

This section is broken down into the three strands: basic, intermediate and advanced that have run throughout this report. This will show what has been achieved by the project against the original objectives and deliverables laid out in the introductory chapter.

5.3.1 Basic Level

All of the basic objectives were met, they were:

- A framework in which graph algorithms could be implemented, supporting the loading and saving of graph definitions from a disk
- The implementation of algorithms to calculate the connectivity and edge-connectivity of a given input graph.

The evaluation has established that the connectivity algorithms run in polynomial time and while that is not ideal it is close to the optimum available algorithm to answer the same problem. It has also been shown that due to the modular design of the algorithm framework it is trivial to use this project as a starting base for other work in the area of computational graph theory.

5.3.2 Intermediate Level

All of the intermediate objectives were met, they were:

- The implementation of a graphical user interface to output the results of the graph algorithms in a visual manner
- The provision for the editing of graphs via the implemented graphical user interface

- The implementation of algorithms to find the weak spots, the cut and separating sets within input graphs.

The evaluation has shown that the algorithms which calculate the cut and separating sets work correctly and that they work again in polynomial time which is the optimal running time for a flow based algorithm of this type. Although there are non flow based algorithms which can calculate the cut sets in less time they cannot do the same for separating sets. Again is has been shown that these algorithms are flexible and can be modified to answer a range of related problems.

5.3.3 Advanced Level

The advanced objective was met, this was to provide a means by which to algorithmically increase the edge-connectivity of an input graph to a certain level. However as the evaluation has shown the algorithm is far from perfect and has problems with general graphs as well as running time. If it gets stuck on a 'run of bad luck' then it is feasible that the algorithm iterates through every possible combination of edges from the additional edges added in the first stage of augmentation and this could take a very long time.

6 Conclusion

Chapter Overview

This chapter sums up the work that has been done throughout and the findings of this project as well as indicating to the reader what further work may be done in this field.

6.1 Results of this Work

6.1.1 Achievements

This project has produced a working tool which can report the connectivity level of a graph. The graphs represent networks and therefore the tool is able to comment on the reliability of those networks as the project brief called for. This tool can also find the weak points in a network and advise the user on possible changes that could be made to the network in order to increase its reliability.

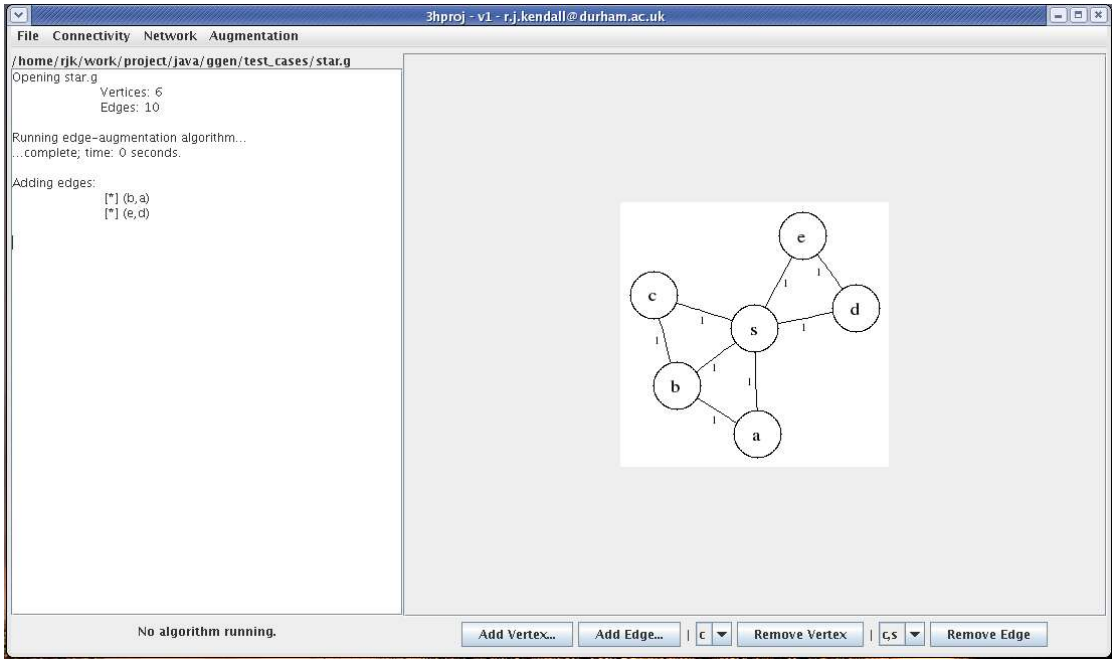


Figure 6.1: The tool after augmenting a five spoke star to be 2-edge-connected.

This tool has the following broad features:

- The ability to retrieve and save graph definitions from a disk
- A graph editor to add and remove edges and vertices from the graph

- A graph drawing system that can draw most small graphs in an aesthetically pleasing fashion
- The ability to calculate the connectivity and edge-connectivity of a graph
- It can show the minimum spanning tree of a given graph
- It can calculate the maximum flow between two vertices in a graph
- Given a graph and two distinct vertices in that graph it can show a cut set and a separating set between those vertices.
- The ability to augment the edge-connectivity of a graph to a pre-defined level while adding a minimum of parallel edges.

The program also has an application programming interface (API) structure which allows new algorithms to be implemented without worrying about the data structures and methods behind storing a graph in the computer's memory. The evaluation showed that the algorithms that have been implemented correctly calculate the connectivity and the cut and separating sets by looking at both common cases and extremal cases. However this is no concrete proof of correctness but it is adequate for this project. The evaluation also showed that while this project has produced a working edge-augmentation algorithm it can be quite slow due to the fact that no perfect heuristic was found for choosing edges to be split off.

6.1.2 Problems

There were a few problems encountered during this project and they are listed here:

- There is a problem with the graph drawing module of the program, when it is fed a graph which is too complex in terms of the number of vertices and edges it has. With these graphs it either outputs a messy graph or the graph renderer fails.
- The graph augmentation module can have problems and produce graphs which contain parallel edges. This should not happen if the requested edge-connectivity is less then n-1 where n is the number of vertices.

6.2 Possible Improvements

The following could be done to improve the tool created for this project:

- The graph editor could be changed so that it is more interactive with the graph diagram, this would mean that graphs can be edited by clicking the vertices and edges in the picture rather than by using the buttons along the bottom as in the current interface.

- The further abstraction of the GraphRenderer class so that more control can be taken over the currently used GraphViz system or another third party graph tool can be used.
- The extension of the API so that the main GraphGUI class does not need to be modified to add entries to the menus and collect information from the user.
- It could be re-implemented in a more efficient language such as C++ which can manipulate the data structures in a quicker fashion with pointer passing and other features.

6.3 Further Work

The augmentation of graphs to improve their edge and vertex connectivities is a topic into which there is a large amount of research particularly by The Egerváry Research Group (EGRES) on Combinatorial Optimisation led by András Frank. Further problems for which solutions could be integrated into the framework created for this project include:

- The implementation of routing algorithms such as interval routing and Dijkstra's algorithm to show how changes to a network structure will change the routing of goods or information across it.
- Algorithms to find which vertices can be removed from a graph while still preserving the edge-connectivity of that graph thus reducing the cost of that network in terms of hardware
- The use of the material contained in [EGRES] to improve the heuristic used to choose the edges to split off during the second phase of the augmentation algorithm.

6.4 Conclusion

This project has resulted in a useful tool that can be used to look at the connectivity properties of graphs in a practical way. This project was a success in the fact that all the objectives and deliverables that were set out in the introduction were met and the solution to the the main problems produced a re-usable graph framework that can be used as a starting point for further work in the subject.

7 Appendices

7.1 Appendix I

7.1.1 Graph Images

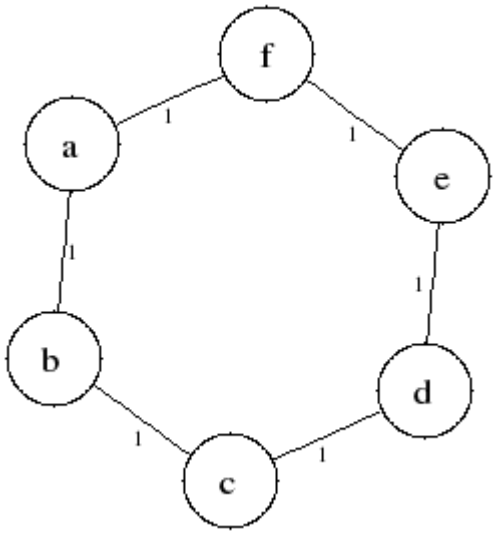


Figure 7.1: A C_6.

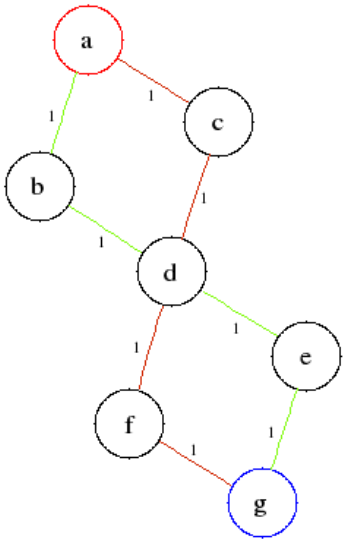


Figure 7.2: A graph with highlighted edges.

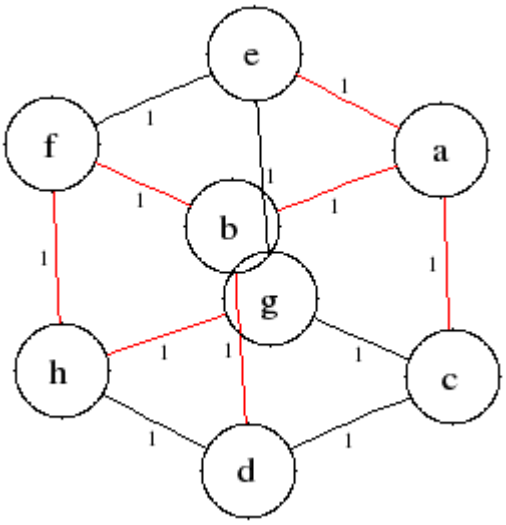


Figure 7.3: A Q_3 with highlighted MST.

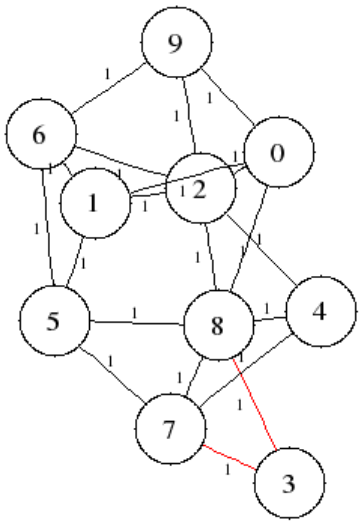


Figure 7.4: A messy graph and highlighted cut-set

7.2 Appendix II

7.2.1 Separating / Cut Set Results

Source	*Sink*	*Separating Set*	*Cut Set*
a	b	{s}	{as}
a	c	{s}	{as}
a	d	{s}	{as}
a	e	{s}	{as}
b	a	{s}	{bs}
b	c	{s}	{bs}
b	d	{s}	{bs}
b	e	{s}	{bs}
c	a	{s}	{cs}
c	b	{s}	{cs}
c	d	{s}	{cs}
c	e	{s}	{cs}
d	a	{s}	{ds}
d	b	{s}	{ds}
d	c	{s}	{ds}
d	e	{s}	{ds}
e	a	{s}	{es}
e	b	{s}	{es}
e	c	{s}	{es}
e	d	{s}	{es}

References

AECR: Andras Frank, Augmenting Graphs to meet Edge-Connectivity Requirements, 1992

AFD: V. King, S. Rao and R. Tarjan, A faster deterministic maximum flow algorithm, 1992

AGT: James A McHugh, Algorithmic Graph Theory, 1990, ISBN: 0-13-023615-2, Prentice Hall

CJV2: Cay S Horstmann and Gary Cornell, Core Java 2 Volume II - Advanced Features, 2002, ISBN: 0-13-092738-4, Prentice Hall

DGWD: Gansner, Koutsofios and North, Drawing Graphs with Dot, 1995

EGRES: Zoltan Szigeti, On partition constrained splitting off, 2004

GTA: Jonathan L Gross and Jay Yellen, Graph Theory and its Applications, 1999, ISBN: 0-84-933982-0, CRC Press

INGT: Robin J Wilson, Introduction to Graph Theory, 1996, ISBN: 0-582-24993-7, Prentice Hall

INTA: Thomas H Cormens, Introduction to Algorithms, 2001, ISBN: 0-26-253196-8, MIT Press

ISSE: Ian Sommerville, Software Engineering, 2001, ISBN: 0-201-39815-X, Addison Wesley

JAPI: The Java API 1.5.0, http://java.sun.com/j2se/1.5.0/docs/api, Last checked: 27/04/2005

OANG: Edward Minieka, Optimisation Algorithms for Networks and Graphs, 1978, ISBN: 0-8247-6642-3, Marcel Dekker Inc

PGT: Nora Hartsfield and Gerhard Ringel, Pearls in Graph Theory: A Comprehensive Introduction, 1990, ISBN: 0-12-328552-6, Academic Press

TCFP: Simon Thompson, Haskell: The Craft of Functional Programming, 1999, ISBN: 0-201-34275-8, Addison Wesley

UML: Joseph Schmuller, Teach Yourself UML, 2002, ISBN: 0-672-32238-2, Sams

WIKI_MF: Wikipedia, Ford Fulkerson Algorithm, http://en.wikipedia.org/Ford-Fulkerson_algorithm, Last checked: 27/04/2005

WIKI_MST: Wikipedia, Minimum Spanning Tree, http://en.wikipedia.org/wiki/Minimum_spanning_tree, Last checked: 27/04/2005