Computational Methods in Graph Connectivity

David L. Fairbairn

Department of Mathematics
Durham University
United Kingdom
April 2019

Abstract

Connectivity invariants aim to give a quantifiable measure of robustness to analyze the effect of failure on a network. This report introduces the notions of vertex and edge connectivity of a graph, giving explicit steps for their calculation. Whitney's Inequality relates these connectivity invariants and the minimal vertex degree. Fundamental theorems in connectivity, such as, Menger's Theorem, Whitney's Theorem and the duality between maximal flow and minimal capacity cuts will be discussed in-depth. A solution to the edge connectivity augmentation problem will be presented using the cactus representation. Construction of the cactus representation will be detailed in meticulous steps with a critical analysis of current literature. An introduction to the basic concepts of graph theory, graph representations, complexity theory and algorithmic problem solving are also included.

Acknowledgements

I extend my deepest thanks to Professor Norbert Peyerimhoff for supervising and providing guidance in this project. I have been lucky to have such a knowledgeable and earnest supervisor. I would also like to thank Dr. Daniele Dorigoni for being my academic advisor. His wise words and high-spirited approach to life, particularly mathematics, has given me continual encouragement and inspiration during my undergraduate studies. Finally, I am grateful to the organizers of the Supported Progression Scheme, especially Dr. Ostap Hryniv, for dedicating an immense amount of time and effort to inspire and enable prospective students from underprivileged backgrounds to attend Durham University.

Contents

1 Introduction **3**

2 Basic Concepts from Graph Theory **5**
2.1 Elements of Graphs . . . 6
2.1.1 Sub-graphs . . . 7
2.1.2 Walks, Trails, Paths & Circuits . . . 8
2.1.3 Cuts . . . 9
2.1.4 Degree Sequences . . . 9
2.2 Complete Graphs . . . 11
2.3 Trees & Forests . . . 11
2.3.1 Spanning Trees . . . 11
2.4 Representation of Graphs . . . 12
2.4.1 Edge List . . . 13
2.4.2 Adjacency List . . . 14
2.4.3 Incidence List . . . 14
2.4.4 Adjacency Matrix . . . 14

3 Algorithms and Complexity Theory **16**
3.1 Basic Algorithms . . . 16
3.1.1 Graphic Algorithm . . . 17
3.1.2 Minimum Spanning Tree Algorithms . . . 19
3.1.3 Prüfer Code . . . 22
3.1.4 Halting Problem . . . 25
3.2 Complexity Theory . . . 26
3.2.1 Types of Complexity . . . 26
3.2.2 Measuring Complexity . . . 26
3.2.3 Complexity of Minimum Spanning Tree Algorithms . . . 27

4 Flow Networks **28**
4.1 Maximum Flow . . . 29
4.2 Ford-Fulkerson Labelling Algorithm . . . 32
4.2.1 Cautionary Example . . . 35

5 Connectivity **37**
5.1 Vertex Connectivity . . . 37
5.2 Edge Connectivity . . . 38
5.3 Computing Edge Connectivity . . . 38
5.4 Computing Vertex Connectivity . . . 40

5.4.1 Menger's Theorem . . . 42
5.5 Whitney's Inequality . . . 44

6 The Edge Connectivity Augmentation Problem **48**
6.1 The Crossing Property . . . 48
6.2 Chain Representation . . . 50
6.3 Cactus Representation . . . 52
6.3.1 Constructing Cactus Representations . . . 53
6.4 Eulerian Tours . . . 57
6.5 Edge Connectivity Augmentation Algorithm . . . 59

7 Conclusion **61**

Appendices **65**

A Python Implementation **66**
A.1 Prerequisites . . . 66
A.2 Graphic Algorithm . . . 68
A.3 Prim's Algorithm . . . 70
A.4 Kruskal's Algorithm . . . 71
A.5 Prüfer Code . . . 72
A.6 Ford-Fulkerson Labelling Algorithm . . . 73
A.7 Edge Connectivity . . . 75
A.8 Vertex Connectivity . . . 76
A.9 Prescribed Connectivities Algorithm . . . 77

Chapter 1

Introduction

Graphs and networks are prolific in their description of numerous real world processes and systems. In this increasingly connected world, the ability to quantify and analyze these connections becomes a greater necessity as the demand and strain on systems tests their reliability. A fundamental question is to quantitatively measure the connectivity of these systems which describes their robustness and reliability. This report will familiarize the reader with various aspects of graph connectivity.

The content of this report is structured as follows:
An introduction to the basic concepts of graph theory and the notation used henceforth in this report is given in Chapter 2. This chapter will also discuss various graph representations. Since this report focuses mainly on an algorithmic approach to problem solving, Chapter 3 contains some basic algorithms in graph theory to make the reader familiar with relevant techniques. This is followed by a brief introduction into the fundamental notions of complexity theory.

After these two introductory chapters the attention shifts to the central topic of this report: graph connectivity. There are three natural problems which arise in this context: various notions of graph connectivity and their explicit calculation; relations between these different graph connectivity notions; the problem of improving graph connectivity in the most efficient manner.

To set the scene for explicit graph connectivity calculations, this report will introduce flow networks in Chapter 4. There is a natural duality between maximum flows and minimal capacity cuts, in solving the maximum flow problem, which is usually solved using the Ford-Fulkerson Method, this duality will give the foundation for calculating graph connectivities. These aspects are explained in detail in this chapter.

In Chapter 5 the basic notions of vertex and edge connectivity are introduced. Fundamental results related to the duality of vertex connectivity are Menger's Theorem and Whitney's Theorem. Whitney's Inequality provides a relation between these two connectivity invariants and the minimal vertex degree.

Chapter 6 is concerned with the problem of increasing the edge connectivity of a graph by one. An optimal solution for this problem is surprisingly difficult and requires the introduction of new concepts such as cactus representation. The literature (sometimes in Russian) on this topic is highly fragmented and difficult to follow. Moreover, some of the articles used in this report are ambiguous in their explanations and lack the required information to provide a complete and coherent solution, with some sources even being noted as incorrect by other authors. A main challenge in the preparation of this chapter was to overcome these difficulties with the literature. The aim of Chapter 6 is therefore to present the approach of one known solution in a comprehensive manner: presenting a step-by-step formulation of the algorithm alongside a

critical analysis of prior research.

A conclusion of this report is presented in the final Chapter 7. Python source code developed by the author is included in various appendices. Moreover, throughout this report algorithms are accompanied by in-text pseudo code for the reader's convenience.

Chapter 2

Basic Concepts from Graph Theory

Graph theory and topology both originate from a problem first posed by Sunday walkers in the city of Königsberg (now Kaliningrad) in the 1700's [36]. The city itself is split by the Pregel River into four distinct quarters: the north, the south and two islands in between (figure 2.1), with seven bridges connecting these quarters together. The exact problem asks, is it possible to walk across each of the seven bridges exactly once in a walk around the city of Königsberg, furthermore if such a walk exists, is it possible return to the quarter where the walk started. This question intrigued mathematicians at the time, spawning many attempts to solve the problem. Some mathematicians simply computed all possible walks across the bridges and deduced that it was not possible to complete such a walk around the city. Such numerical solutions displeased many mathematicians at the time, as many wanted a general solution for similar problems, rather than the specific solution to this problem.

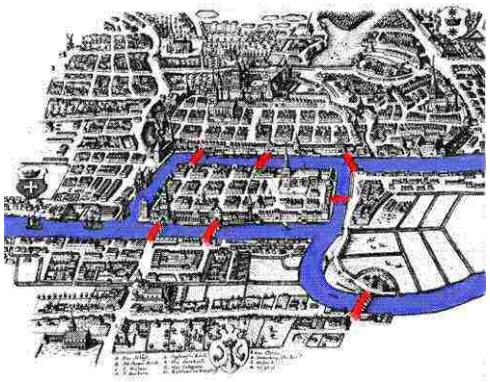


Figure 2.1: A plate of the 7 bridges of Königsberg stretching across the Pregel River. Source: [43].

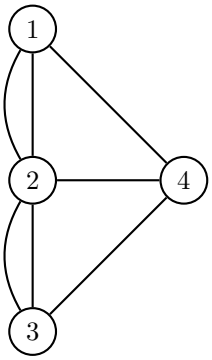


Figure 2.2: Graph of Königsberg.

Many attempts to find a general solution focused mainly on the bridges; whereas Leonard Euler, observed in 1735 [13] that the bridges alone are not the key to a general solution, the quarters also need to be taken into consideration. In particular, one should consider the number

of bridges connecting one quarter to one another. Euler reformulated the problem in an abstract way, considering each independent quarter as a uniquely labelled *vertex*, and each bridge as a unique *edge* (as in figure 2.2), hence laying the foundations for graph theory. Euler's solution to the Seven Bridges problem highlights an important tool for solving a later problem in this report and therefore will be presented in section 6.4.

2.1 Elements of Graphs

Euler laid the foundations of graph theory with the vertex and the edge. Combining a set of n vertices, $V = \{v_1, v_2, ..., v_n\}$, and set of m edges, $E = \{e_1, e_2, ..., e_m\}$, yields a finite *graph* $G = (V, E)$. The cardinality of the set V and E is denoted by $|V|$ and $|E|$.

Edges in G may be *undirected*, with e given by an unordered pair of vertices $(v_i, v_j) = (v_j, v_i)$, meaning that an edge joining two vertices, acts in both directions. Edges may also be *directed*, with e given by an ordered pair of vertices $(v_i, v_j) \neq (v_j, v_i)$, meaning that the edge has an origin v_i denoted e^-, and an endpoint v_j denoted e^+. Any graph whose edges are all directed, is called a *digraph* (Directed graph). Furthermore, a graph which has at least two identical edges $e_i = e_j$ for some $i \neq j$, is said to be a *multigraph*. If an edge in G connects a vertex to itself, then this edge is called a loop and G is also said to be a *multigraph*. A graph which is not a multigraph and has only undirected edges is said to be a *simple graph*. Within this report, the term "graph" refers to a finite undirected multigraph (which may be simple) unless stated otherwise.

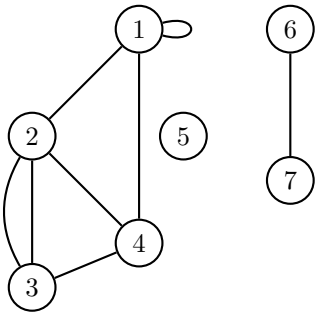


Figure 2.3: Undirected Graph.

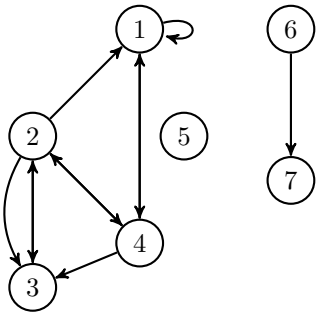


Figure 2.4: Directed Graph.

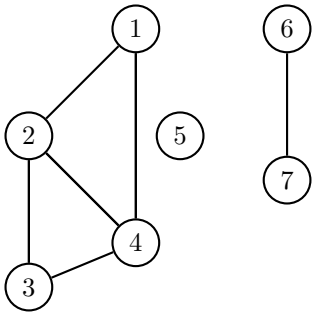


Figure 2.5: Simple Graph.

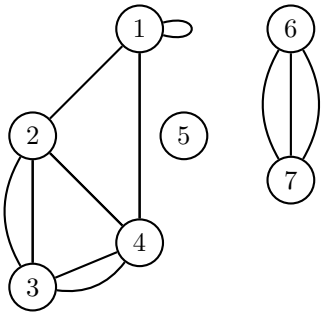


Figure 2.6: Multigraph.

Graphs often have other information encoded into both the vertices and edges. Edges often have length or weighting encoded to represent the distance or cost of transition between two

vertices. Any graph $G = (V, E)$ which has weight (equivalent to length) function $\omega : E \mapsto \mathbb{R}^+$ on the set of edges, is called a *weighted graph*.

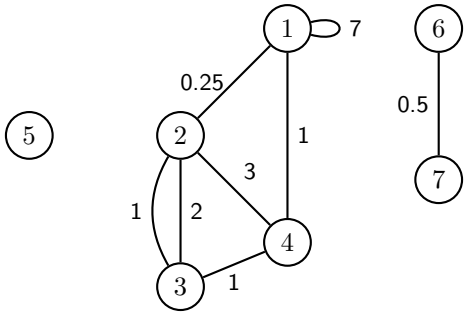


Figure 2.7: Weighted Graph.

2.1.1 Sub-graphs

The notation and content of this sub-section closely follow that of Jungnickel 2013 [25, p. 3-4].

When analysing a graph $G = (V, E)$, it is important to consider graphs contained within G. Such smaller graphs G' are called *subgraphs* of G, which are obtained by choosing a set $V' \subseteq V$ and taking the set of all edges $e \in E$ which have both their endpoints in V', denoted by $E|V'$. The graph $G' = (V', E|V')$ is called the *induced subgraph* of G. Another type of subgraph is formed by taking a subset of edges of an induced subgraph, $E' \subseteq E|V'$, then the graph $G' = (V', E')$ is simply called a subgraph of G. In addition a subgraph with $V' = V$ is said to be a *spanning subgraph* of G and G is a spanning induced subgraph of G.

In this report, subgraphs will often be formed by removing a subset of vertices or edges hence, let $G \setminus S_V$ denote the induced subgroup $G' = (V \setminus S_V, E|V \setminus S_V)$ on G, where $S_V \subseteq V$. Similarly $G \setminus S_E$ denotes the the spanning subgraph $G' = (V, E \setminus S_E)$, where $S_E \subseteq E$.

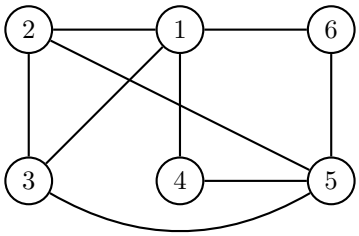


Figure 2.8: Simple graph $G = (V, E)$.

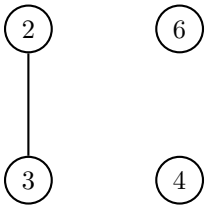


Figure 2.9: Induced subgraph $G \setminus \{1, 5\}$.

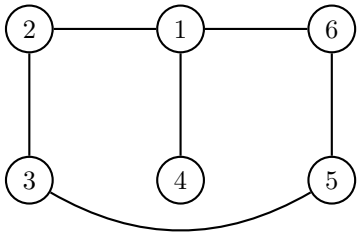


Figure 2.10: Spanning subgraph $G \setminus \{(1,3),(2,5),(4,5)\}$.

2.1.2 Walks, Trails, Paths & Circuits

Walks are a key tool in graph theory as they allow for movement from one vertex to another. In the Seven Bridges problem the idea of a walk motivated the problem, in particular, a walk over bridges (edges) to different quarters (vertices). A walk (as in Jungnickel 2013 [25, p. 5]) is defined as follows:

Definition 2.1.1. Given a graph $G = (V, E)$ and $\varepsilon = (v_0, e_1, v_1, e_2, v_2, ..., e_n, v_n)$, a sequence of edges $e_i \in E$ and vertices $v_i \in V$, such that $e_i = (v_{i-1}, v_i)$ for $i = \{1, ..., n\}$ then ε is a *walk*.

Walks define the most general form of movement within a graph. When further constraints are applied to a walk, more interesting movements can be defined. These movements are defined as given in Jungnickel 2013 [25, p. 5]. Given a walk ε, if $v_0 = v_n$ then ε is a *closed walk*. Given a walk ε if each edge $e_i \in \varepsilon$ is distinct then ε is a *trail*. If a trail has $v_0 = v_n$ then ε is a *closed trail*. Then a *path* is a trail with each v_j distinct. Furthermore a closed trail with $n > 2$ and all v_j distinct is called a *cycle* or *closed path*. A graph which contains no cycles is then called an *acyclic graph*.

A graph for which all pairs of vertices $v_i, v_j \in V$ have some path which starts at v_i and ends at v_j is called a *connected graph*. Conversely for some graphs it may not be possible to form such a path; in these cases v_j is said to be *unreachable* from v_i and the graph is called *disconnected*. A *connected component* of a graph $G = (V, E)$ is any induced connected subgraph $G \setminus S_V$, such that for any S'_V where $S_V \subset S'_V$ then $G \setminus S'_V$ is disconnected and $G \setminus S_V$ is a *maximally connected subgraph*. If a graph is connected then it is said to have one connected component (G itself). If a graph is disconnected, then the number of connected components is the number of maximally connected subgraphs of G (For example the graph in figure 2.9 has 3 connected components).

Lemma 2.1.1. *A connected simple graph $G = (V, E)$ with $|V| = n$, has $|E| \geq n - 1$.*

Proof. (Based on Jungnickel 2013 [25, p. 7]) Consider n = 1, an *isolated vertex*, which is connected. Now consider the case $n \geq 2$. Given a graph $G = (V, E)$, choose any vertex $v \in V$ and consider the subgraph $H = G \setminus v$. Now, H may be disconnected, with m connected components, consider each connected components Z_i, $1 \leq i \leq m$, with n_i vertices respectively. By induction, assume that each Z_i has at least $n_i - 1$ edges. Notice that also v must be connected to each Z_i in G by at least one edge. Therefore, G must contain at least $(n_1 - 1) + ... + (n_m - 1) + m = n - 1$ edges. □

Lemma 2.1.2. *An acyclic simple graph $G = (V, E)$ with $|V| = n$, has $|E| \leq n - 1$.*

Proof. (Based on Jungnickel 2013 [25, p. 7]) For $n = 1$ or $E = \emptyset$, trivially, no cycles exist. Now, consider the acyclic graph $G = (V, E)$ a spanning subgraph $G \setminus e$, for some arbitrary edge $e \in E$, has exactly one more connected component as no path can be found between the endpoints of e

in $G \setminus e$, otherwise G would contain a cycle. Then, $G \setminus e$, has m acyclic connected components Z_i, $1 \leq i \leq m$. By induction, assume that each Z_i has at most $n_i - 1$ edges. Therefore, G must contain at most $(n_1 - 1) + ... + (n_m - 1) = (n_1 + ... + n_m) - (m - 1) \leq n - 1$ edges. □

2.1.3 Cuts

A *cut* $C = (S, T)$ partitions a given graph $G = (V, E)$ by its vertices into two disjoint subsets, $S, T \subset V$. Each cut induces a *edge-cut* $S_C = \{e \in E : e = (i, j), i \in S, j \in T\}$, a set of edges such that one endpoint is in S and the other is in T.

Cuts and edge-cuts, are an interesting topic of discussion which will be explored later in this report, looking at problems such as the minimum cardinality of a set-cut on G.

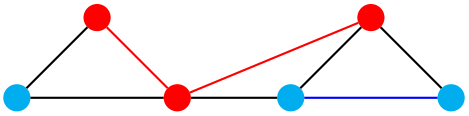
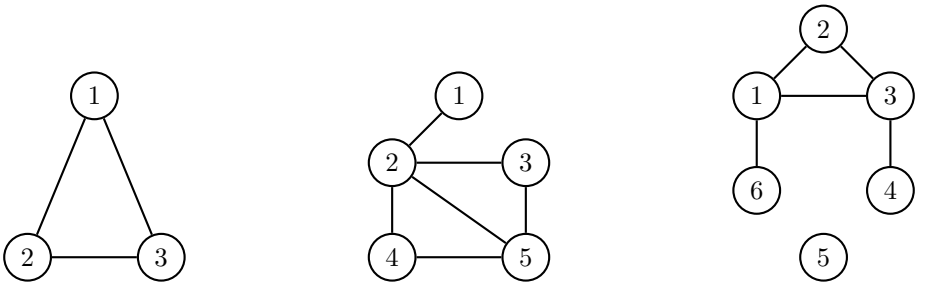


Figure 2.11: A cut $C = (S, T)$ on a simple graph, $s \in S$ (blue), $t \in T$ (red), edge-cut (black).

2.1.4 Degree Sequences

In an undirected graph $G = (V, E)$ the *degree* of a vertex $v \in V$, is the number of edges incident from v, denoted $\deg(v)$. The *degree sequence* of a graph G is a set D, comprised of the degree of each vertex such that the i^{th} entry in the sequence is the degree of the i^{th} vertex in V . That is,

$$D = \{d_i : d_i = deg(v_i), v_i \in V, i \in \{1, 2, ..., n\}\}. \tag{2.1}$$



$D = [2, 2, 2]$ $D = [1, 3, 2, 2, 3]$ $D = [3, 2, 3, 1, 0, 1]$

Figure 2.12: Examples of degree sequences.

Theorem 2.1.1. *For any undirected graph $G = (V, E)$ with degree sequence D,*

$$\sum_{d_i \in D} d_i = 2|E|. \tag{2.2}$$

Proof. Each edge $e \in E$ has two endpoints. Hence each e contributes twice in the summation, once for each endpoint. □

From theorem 2.1.1 it is possible to extrapolate some conditions which can be imposed on degree sequences for an arbitrary undirected graph. One such condition is given by Euler [13]:

Lemma 2.1.3. (Handshaking Lemma) *For a degree sequence D of a undirected graph $G = (V, E)$ has an even number of odd d_i.*

Proof. As, $\sum_{d_i \in D} d_i = 2|E|$, is even. Then, there must be an even number of odd d_i. □

When performing analysis on degree sequences, it is important to consider two important values: the minimal value of D, denoted $\delta(G) = \min(d_i \in D)$, and the maximal value of D, denoted $\Delta(G) = \max(d_i \in D)$. Using these values, a constraint on the degree sequence of simple graphs can be formulated by applying the pigeonhole principle,

Lemma 2.1.4. *Given a simple graph $G = (V, E)$ with $|V| \geq 2$ and degree sequence D. Then there are always at least two $d_i \in D$ with the same value.*

Proof. For a given simple graph $G = (V, E)$ the maximal possible value of $\Delta(G)$ is $|V| - 1$, the smallest possible value of $\delta(G)$ is 0, this gives $|V|$ possible values for each d_i. However, the values $\Delta(G) = |V| - 1$ and $\delta(G) = 0$ cannot occur simultaneously for a simple graph, this gives $|V| - 1$ possible values for each d_i but $|V|$ vertices each with a degree d_i. By the pigeon hole principle, there must be at least two d_i of the same value. □

Lemma 2.1.4 dictates that not all arbitrary degree sequences are realized by a simple graph. Hence, is it possible to find a set of conditions which can determine if, any arbitrary degree sequence if it is realized by a simple graph? These conditions solve a problem known as the graph realization problem, with any degree sequence satisfying these conditions called *graphic*. Such a set of conditions were first published in 1960 by Paul Erdős and Tibor Gallai [12] and later proved in 1972 by Frank Harary [19],

Theorem 2.1.2. (Erdős-Gallai Theorem) *Let $D = (d_1, ..., d_n)$ be an arbitrary degree sequence with $d_1 \geq d_2 \geq ... \geq d_n$. D is graphic if and only if the following conditions hold:*

$$\sum_{i=1}^{n} d_i \quad , \qquad \textit{is even}, \tag{2.3}$$

$$\sum_{i=1}^{k} d_i \quad \leq \quad k(k-1) + \sum_{i=k+1}^{n} \min(k, d_i)\,, \quad k = 1, 2, ..., n. \tag{2.4}$$

Proof. See Harary 1972 [19, p. 59] □

These conditions enable quick verification of whether a degree sequence is graphic. However, they do not provide any method of actually obtaining the simple graph which realizes this degree sequence. Such a method will be covered later in the report as an introduction to algorithms. Something of note is that the degree sequence is not always unique: two graphs may have different underlying structures but the same degree sequence.

2.2 Complete Graphs

The *complete graph* on n vertices, denoted K_n, is a simple graph, such that $|V| = n$ and each vertex has degree equal to $n - 1$. Hence, each vertex is adjacent to every other vertex and $|E| = n(n-1)/2$.

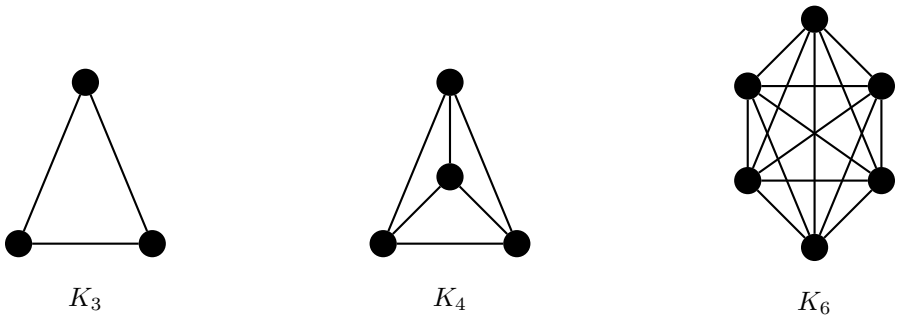


Figure 2.13: Examples of complete graphs.

2.3 Trees & Forests

An acyclic simple graph which is connected is called a *tree*. As a tree is acyclic, the removal of any edge will disconnect the tree into a disjoint union of trees, called a *forest*. A *leaf* of a tree is then any vertex with degree equal to one.

Lemma 2.3.1. *Given a tree $G_T = (V, E)$ with $|V| = n$ then, $|E| = n - 1$.*

Proof. G_T is a tree, therefore, G_T is connected and acyclic. By Lemma 2.1.1 and lemma 2.1.2, $n - 1 \leq |E| \leq n - 1$, which implies $|E| = n - 1$. □

Lemma 2.3.2. *A forest $G_F = (V, E)$ with $|V| = n$ and k connected components then, $|E| = n-k$.*

Proof. Each connected component has n_i vertices and therefore, $n_i - 1$ edges by lemma 2.3.1. Summing the number of edges over k connected components gives $\sum_k^i v_i - 1 = n - k$. □

2.3.1 Spanning Trees

Given an undirected graph, then a spanning subgraph whose structure is a tree is said to be a *spanning tree* (Figure 2.14). For a weighted graph, a spanning tree of this graph will then have an overall weight equal to the sum of all weights on its edges. Finding the spanning tree with the lowest total weighting across all edges can reduce the costs associated with the system and increase its efficiency. Such a spanning tree is called the *minimal spanning tree*. Formally, given a undirected weighted graph $G = (V, E)$ with weight function ω, a spanning tree $G_T = (V_T, E_T)$ of G is said to be a *minimal spanning tree* of G if $\sum_{e \in E_T} \omega(e)$ is minimal across all spanning trees of G. An example of a minimal spanning tree is presented in figure 2.14. Finding minimal spanning trees is not trivial and will be discussed later in this report as generating this structure requires computational methods.

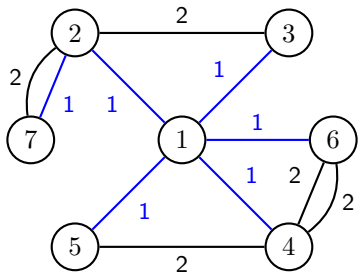


Figure 2.14: Weighted graph with spanning tree (blue).

Note: A tree structure is a requirement for systems such as the Internet as cycles cause feedback on data transmission, which is fatal to most computer networks. However, having a tree structure is not reliable as one disconnection (removed edge) causes the graph to become separated. Therefore, most systems do not limit themselves to a tree structure, but instead opt for a structure with more connections (and therefore contains cycles). But to avoid feedback from cycles, these systems only communicate via a spanning tree (See Spanning Tree Protocol (STP) [41] for exact implementation). In these systems, all vertices know the spanning tree and exclusively communicate via this tree. If one connection in the spanning tree is lost, the system can attempt to find another spanning tree to communicate through, hence creating a more reliable system.

2.4 Representation of Graphs

A graph may not be unique, especially when the labelling of vertices is irrelevant to the system. If two graphs have the same underlying structure and one can a labelling such that the two graphs are equivalent, then these graphs are said be isomorphic. That is, the graphs $G = (V, E)$ and $H = (V', E')$ are said to be isomorphic ($G \cong H$) if there exists a bijection, $\theta : V \mapsto V'$, such that $\theta(E) = E'$ where $\theta(E) = \{(\theta(i), \theta(j)) : (i, j) \in E\}$.

For many real world systems, such as transport networks, their graphs are best represented graphically (presented visually), so that the system is easier to read and understand. Graphic representations can be an effective method for conveying a lot of information in a condensed format but often take a lot of time to produce. As figures 2.15 and 2.16 demonstrate, the difficulty for determining whether a graph is a tree varies wildly between these different graphical representations. It can be very difficult to perform computational operations on a graphic representation, due to the time required for a computer to understand the graphs. In many situations, a computer will simply not be able to perform analysis on graphical representations. This section will therefore focus on introducing a variety of computational representations. Just as two graphic representations (e.g. figure 2.15 and figure 2.16) can determine properties at different efficiencies, so too can different computational representations of graphs. These representations will be used later in this report to solve problems using computational methods.

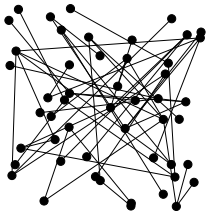


Figure 2.15: Tree $G = (V, E)$ with $|V| = 50$.

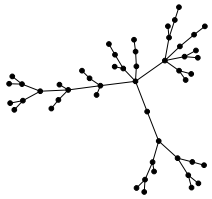


Figure 2.16: Tree $G = (V, E)$ with $|V| = 50$.

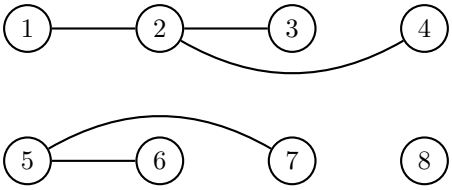


Figure 2.17: Simple graph G_1.

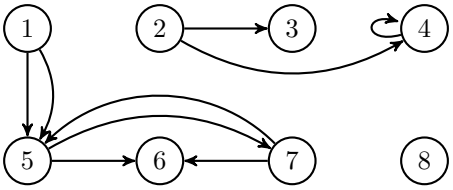


Figure 2.18: Directed multigraph G_2.

2.4.1 Edge List

An *edge list* representation of a graph $G = (V, E)$, is a list comprised of all ordered pairs of vertices (v_i, v_j) which form an edge in E. An edge list on its own is not guaranteed to encode all the data required to reconstruct a graph; (take for example figure 2.17 and figure 2.18) an isolated vertex will not appear in any element of the edge list. Hence, an edge list must also have attached the number of vertices in the graph, $|V|$, so that it can reconstruct isolated vertices. Thus an edge list for a directed multigraph $G = (V, E)$ on a vertex set V and edge set E (as in Jungnickel 2013 [25, p. 39]) is defined by:

1. $n = |V|$.
2. The list of its edges e_k, given as a sequence of ordered pairs $(v_i, v_j)_k \in E$.

A similar definition is given for a simple graph $G = (V, E)$. However the list of its edges e_k, is given by a sequence of unordered pairs $(v_i, v_j)_k \in E$.

The graphs G_1 and G_2 as in figure 2.17 and figure 2.18 can be represented respectively as an edge list as follows:

Example Edge Lists		
G_1	n Edge List	8 $\{(1,2),(2,3),(2,4),(5,6),(5,7)\}$
G_2	n Edge List	8 $\{(1,5),(1,5),(2,3),(2,4),(4,4),(5,6),(5,7),(7,5),(7,6)\}$

Storing a graph as an edge list, requires space for $2|E| + 1$ integers; n and the edge list consisting of $|E|$ pairs of integer. Edge lists will prove to be a useful representation later in this report. However, they do not allow for quick analysis of certain properties of vertices. Take for

example calculating the degree of a specific vertex; a program would have to read the whole edge list counting each time the desired vertex appeared. Hence, for more problems where the vertices take precedence, a more desirable representation should encode n so that isolated vertices are not ignored and be sorted by vertex.

2.4.2 Adjacency List

An *adjacency list* representation of a graph encodes $n = |V|$ by taking a list of n lists, for which the i^{th} list contains the set E_i such that some v_j is incident from v_i, that is, $E_i = \{v_j : (v_i, v_j) \in E\}$. Hence, an adjacency list of a directed multigraph G on a vertex set V (as in Jungnickel 2013 [25, p. 40]) is defined by:

1. $n = |V|$.
2. $E_1, ..., E_n$, where E_i is a list containing all vertices v_j for which the directed edge $e = (v_i, v_j) \in E$.

Storing a graph as an adjacency list, requires space for $|E|$ integers; one for each edge. The graphs G_1 and G_2 as in figure 2.17 and figure 2.18 can represented respectively as an adjacency list as follows:

Example Adjacency Lists	
G_1	$\{2\}, \{1, 3, 4\}, \{2\}, \{2\}, \{6, 7\}, \{5\}, \{5\}, \{\}$
G_2	$\{5, 5\}, \{3, 4\}, \{\}, \{4\}, \{6, 7\}, \{\}, \{5, 6\}, \{\}$

2.4.3 Incidence List

An *incidence list* representation of a graph takes the idea of the adjacency list but encodes more information into each of the $n = |V|$ lists, such as the weights or labels of the edge of a graph. An incidence list consists of n lists, with the i^{th} list containing the set A_i of ordered pairs (e', v_j), such that, $(v_i, v_j) \in E$ and (v_i, v_j) has edge label (or weight) e'. That is, an incidence list on a directed multigraph G on a vertex set V (as in Jungnickel 2013 [25, p. 39]) is defined by:

1. $n = |V|$.
2. $A_1, ..., A_n$, where A_i is a list of all ordered pairs (e', j) which are incident from vertex v_i and have endpoint vertex v_j with edge label e'.

Storing a graph as an incidence list with integer edge weights, requires space for $2|E|$ integers; two for each edge. The graphs G_1 and G_2 as in figure 2.17 and figure 2.18 can represented respectively as an adjacency list as follows:

Example Incidence Lists	
Figure 2.7	$\{(7, 1), (0.25, 2), (1, 4)\}, \{(0.25, 1), (1, 3), (2, 3), (3, 4)\},$ $\{(1, 2)(2, 2)(1, 4)\}, \{(1, 1)(3, 2)(1, 3)\}, \{\}, \{(0.5, 7)\}, \{(0.5, 6)\}$

2.4.4 Adjacency Matrix

The *adjacency matrix* A_G is a square matrix of size $n = |V|$, with each entry a_{ij} being equal to one if the edge $(v_i, v_j) \in E$ and zero if $(v_i, v_j) \notin E$. This report will only consider adjacency matrices of simple graphs. However, the definition of the adjacency matrix can also be extended to incorporate multigraphs (and weighted simple graphs) by defining each entry a_{ij} to be equal

to the number of edges $(v_i, v_j) \in E$ (or the weight of (v_i, v_j)). All adjacency matrices for simple graphs are symmetric.

Storing an adjacency matrix requires space for $|V|^2$ integers. This is the largest representation for simple graphs when compared to the edge list and adjacency list. However this representation allows for some very quick computation of some problems. Computing the degree sequence of a graph is trivial when using an adjacency matrix as the degree of vertex v_i is simply the entry a_{ii} of the matrix A_G^2 (Proof of which can be found in Jungnickel 2013 [25, p. 108]). Take for example G_1 as in figure 2.17, its adjacency matrix is given as follows:

$$\mathbf{A_{G_1}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2.5}$$

Chapter 3

Algorithms and Complexity Theory

3.1 Basic Algorithms

Within graph theory, the use of theorems and conjecture is not always enough to solve problems. The computational methods can be split into two main categories, non-deterministic methods such as Monte Carlo Simulation, and deterministic methods, such as Linear Programming. These methods are often employed to find solutions to problems which require more than simply a yes or no answer. This report will focus on the deterministic computational methods for solving problems through the use of algorithms as these allow for a more analytical approach to solving complex problems, using the solutions to previously studied problems. As presented in Jungnickel 2013 [25, p. 36], the techniques and steps taken to solve a problem is called an algorithm if the following criteria hold true:

1. Finite description - The technique only takes finite text to be explained.
2. Effectiveness - The technique and all steps involved are feasibly possible (and gives correct solution).
3. Termination - The technique is guaranteed to stop after a finite number of steps.
4. Determinism - Given the exact same input the technique will perform the exact same steps (no random steps).

These criteria hold if and only if each step in an algorithm satisfies the criteria of an algorithm. These criteria help to separate techniques which are inefficient. A finite description allows for good communication of the technique. Effectiveness ensures that techniques are possible and that the algorithm will always give the correct solution. Then termination, a very desirable property, ensures that any valid input will processed correctly without causing the algorithm to get stuck during calculation. Termination is not a trivial property to have or to prove (See the Halting problem later in this chapter). Therefore, all algorithms in this report will be constructed to be terminating, as will be seen later in this report constructing an algorithm to terminate for some inputs may not imply that the algorithm will always terminate for all inputs. Finally, determinism allows for analysis of the steps within an algorithm, an algorithm should take the same steps for the same input data every time.

3.1.1 Graphic Algorithm

As seen in the previous chapter with degree sequences, the Erdős-Gallai Theorem can very quickly determine whether a degree sequence is graphic. But this method does not give any indication of how to form a simple graph with such a graphic degree sequence. A shallow technique for solving this problem would be the following:

Step 1 If the Erdős-Gallai Theorem holds for D, then Go-to **Step 2**. Otherwise **Stop**; D is not graphic.

Step 2 Generate a random graph with $|V|$ equal to the cardinality of D, if the degree sequences match, then **Done**. Otherwise repeat **Step 2**.

This techinque will very quickly give a negative response (if D is not graphic), but if D is graphic, given enough time **Step 2** will yield a solution. This technique does not define an algorithm, although this technique will always find the correct answer it is not possible to say how many steps it will take to generate the desired graph, nor is it possible to place an upper bound on this time. Hence, it is unclear whether this technique will terminate and also fails the determinism condition because of the random process in **Step 2**.

One such algorithm for solving the graph realization problem is the Havel–Hakimi algorithm [28, p. 9-15]. This algorithm takes a different approach to solving this problem. First assuming that a degree sequence is graphic, then attempting to build a graph using this degree sequence, if that fails then, D is not graphic, but if it succeeds then not only is D graphic, the algorithm will have generated a graph which realizes D in the process.

To begin such an algorithm the degree sequence must be manipulated by considering a sequence $L = (l_0, ..., l_m)$ where $m = \delta(G)$, such that, each l_i contains a list of vertices with degree equal to i. That is,

$$m = \Delta(G), \tag{3.1}$$
$$L = \{l_i : l_i = \{v_j : \deg(v_j) = i, v_j \in V\}, i \in \{0, ..., m\}\}. \tag{3.2}$$

The graph will be built proactively throughout the algorithm, by adding edges to graph consisting of $n = |D|$ isolated vertices. As such the best representation to use in for this is an edge list. Therefore, a set E will be used to keep track of the edges in the final graph. A list of vertices will not need to be created as the algorithm will generate one.

The general idea for building the graph will be to consider the vertex with degree $\Delta(L)$ (maximal non-empty indexed $l_i \in L$), then attach $\Delta(L)$ edges (with no determined endpoint) to this vertex. In doing this, the vertex will need to be moved from $l_\Delta(L)$ to l_0 as it has its Δ edges. The $\Delta(L)$ edges with the undetermined endpoints must then join to the next $\Delta(L)$ disjoint vertices with highest non-zero degree. If there are not enough vertices, then D is not graphic, otherwise each edge should be added to E and the index of each vertex reduced in L by one. If all vertices are in l_0, then the graph has been generated and E and $V = l_0$ then act as the requirements for the edge list representation of a simple graph. Otherwise, this process can be repeated until the graph is either generated or found not to exist.

To keep up with these operations another sequence $L' = (l'_0, ..., l'_m)$, will be used to stop the possibility for two vertices sharing more than one edge. For any algorithm the initial value must be noted and for the sets E and L' these are as follows:

$$E = \emptyset \tag{3.3}$$
$$L' = \{l'_j : l'_j = \emptyset, j \in \{0, ..., m\}\}. \tag{3.4}$$

This algorithm can then be presented in more specific steps as follows:

Step 1 In L find the none-empty list with the largest index. Set $\Delta(L)$ equal to the index of this list. If $\Delta(L) = 0$ then; the algorithm has been found a graph, the edge set E and the set of vertex l_0 define an edge list of a simple graph. Otherwise set α equal to the vertex positioned at $l_{\Delta(L),0}$ (the first vertex in $l_{\Delta(L)}$), then remove $l_{\Delta(L),0}$ from $l_{\Delta(L)}$, add the vertex α to l_0. Go-to **Step 2**.

Step 2 In L, find the none-empty list with the largest index. Set q equal to the index of this list. If $q = 0$ then **Stop**; D does not define a simple graph. Otherwise set β equal to the vertex at $l_{q,0}$, add the edge (α, β) to E then set $\Delta(L) = \Delta(L) - 1$ and finally take $l_{q,0}$ remove it from l_q and add the vertex β to l'_{q-1}. Go-to **Step 3**.

Step 3 If $\Delta(L) = 0$, Go-to **Step 4**. Otherwise Go-to **Step 2**.

Step 4 For every $l'_j \in L'$, append each element of l'_j into the respective $l_j \in L$, $(l_j = l_j \cup l'_j)$. Set each $l'_j = \emptyset$, Go-to **Step 1**.

These steps are also presented here in pseudo code:

Algorithm 3.1.1 Graphic Algorithm - Input : Degree Sequence (D). Output : Edge set (E) and vertex set (V) if the degree sequence is graphic; False otherwise.

```
1:  procedure GRAPHICALGORITHM(D)
2:      m ← Δ(G)                                   ▷ Highest value in D.
3:      L ← [ [ ]_j : j ∈ {0, ..., m} ]            ▷ m + 1 empty lists.
4:      for d_i ∈ D do
5:          L_{d_i} ← L_{d_i} ∪ v_i
6:      E ← [ ]
7:      L' ← [ [ ]_j : j ∈ {0, m} ]
8:      while 1 ≠ 0 do                             ▷ Infinite loop - Step 1.
9:          p ← Largest j of none empty l_j ∈ L    ▷ p = Δ(L)
10:         if p = 0 then
11:             return E, l_0                      ▷ Algorithm successful.
12:         α ← l_{p,0}
13:         l_0 ← l_0 ∪ l_{p,0}
14:         l_p ← l_p \ l_{p,0}
15:         while p ≠ 0 do                         ▷ Step 3
16:             q ← Largest j of none empty l_j ∈ L ▷ Step 2.
17:             if q = 0 then
18:                 return False                   ▷ Algorithm failed.
19:             β ← l_{q,0}
20:             E ← E ∪ (α, β)
21:             p ← p − 1
22:             l'_{q−1} ← l'_{q−1} ∪ l_{q,0}
23:             l_q ← l_q \ l_{q,0}
24:         for j ∈ {0, ..., m} do                 ▷ Step 4.
25:             l_j ← l_j ∪ l'_j
26:             l'_j ← [ ]_j
```

In this report, any algorithm which has pseudo code is accompanied by an exact python implementation, by the author, in the appendix. For the Graphic Algorithm: See Appendix A.2.

3.1.2 Minimum Spanning Tree Algorithms

As discussed previously in this report, finding the minimum spanning tree of a system can both increase efficiency and reduce the costs involved in running a system. The first of such algorithms (Borůvka's Algorithm) to find the minimal spanning tree was discovered and published by Otakar Borůvka in 1926 [3] as a proposed solution to give the country of Moravia an efficient electricity network [4]. His algorithm requires all edge weightings to be distinct, hence, this algorithm will not be considered in this report, as a more general algorithm is required for modern systems.

The first of two notable algorithms is Prim's Algorithm (Jarník's Algorithm); first developed in 1930 by Vojtěch Jarník [24] and later republished by Robert Clay Prim [33] and Edsger Wybe Dijkstra [8] in 1957 and 1959 respectively. Prim's Algorithm focuses on building the minimal spanning tree progressively; starting the spanning tree from a single arbitrary vertex and joining the tree to a new vertex via the lowest cost edge from any vertex currently within the tree. In doing this, cycles are avoided as edges are only formed between new vertices and not between vertices already in the tree. That is, given a connected weighted undirected graph $G = (V, E)$ with $V = \{v_1, ..., v_n\}$ and weight function $\omega(v_i, v_j)$ for $(v_i, v_j) \in E$ then the minimal spanning tree $G_\mathrm{T} = (V_T, E_T)$ of G is determined by the following steps:

Step 1 Set $E_T = \emptyset$, $V_T = \{v_1\}$.

Step 2 Find $e \in E$ with minimal $\omega(u, w)$, for $u \in V_T and w \in V \setminus V_T$. If no such e exists, **Stop**; the minimal spanning tree $G_T = (V_T, E_T)$ has been found. Otherwise, add e to E_T and w to V_T, Go-to **Step 2**.

In pseudo code:

Algorithm 3.1.2 Prim's Algorithm - Input: Connected weighted undirected graph $G = (V, E)$ with weight function ω. Output: Minimal spanning tree $G_T = (V_T, E_T)$.

```
1:  procedure PRIMS(V, E, ω)
2:      E_T ← [ ]
3:      V_T ← [ V[0] ]                          ▷ First vertex in V.
4:      while True do                           ▷ Loops Forever.
5:          minedge ← ∞
6:          for u ∈ V_T do
7:              for v ∈ V \ V_T do
8:                  if (u, w) ∈ E AND  ω(u, v) < minedge then
9:                      u_T ← u
10:                     w_T ← w
11:                     minedge ← ω(u, w)
12:         if minedge = ∞ then                 ▷ No Edges found.
13:             return V_T, E_T                 ▷ Minimal spanning tree G_T = (V_T, E_T).
14:         E_T ← E_T ∪ [(u_T, w_T)]
15:         V_T ← V_T ∪ [w_T]
```

Python implementation: See Appendix A.3.
Proof of correctness: See Dieter Jungnickel [25, p. 114].

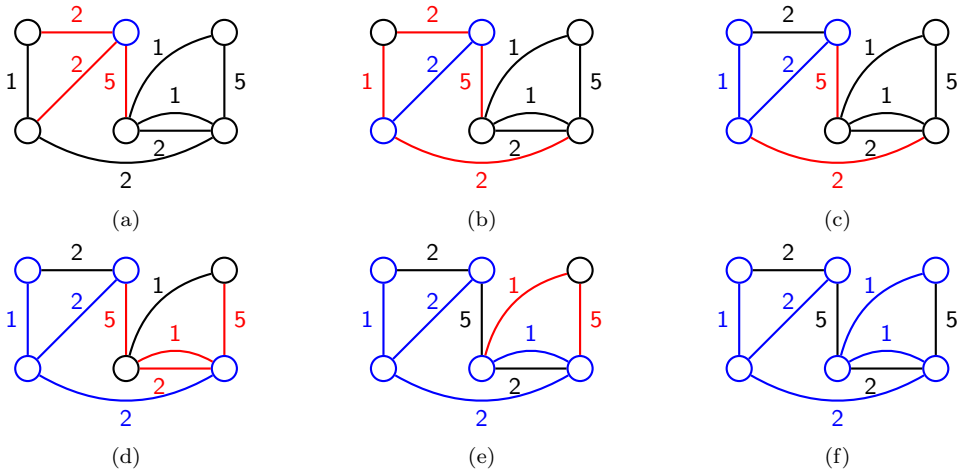


Figure 3.1: Prim's Algorithm - $G = (V, E)$ (black), $G_T = (V_T, E_T)$ (blue), edges under consideration (red).

Another method for finding a minimal spanning tree is Kruskal's Algorithm; first published by Joseph Bernard Kruskal in 1956 [29]. This algorithm finds the minimal spanning tree of $G = (V, E)$, by first considering a forest G_T with each vertex in G as a separate tree in this forest. These trees are then progressively joined together by selecting the smallest edge in G which joins two trees in G_T (an edge which does not form a cycle in G_T). Once G_T is a tree (and no longer a forest), then G_T is a minimal spanning tree of G. That is, given a connected weighted undirected graph $G = (V, E)$ with $V = \{v_1, ..., v_n\}$ and weight function $\omega(v_i, v_j)$ for $(v_i, v_j) \in E$ then the minimal spanning tree $G_T = (V_T, E_T)$ of G is be determined by the following steps:

Step 1 Set $E_T = \emptyset$, $V_T = \{v_1, ..., v_n\}$ and let the set of trees in G_T be given by, $T = [[v_1], ..., [v_n]]$ where $t_i \in T$ represents the vertices in the i^{th} tree.

Step 2 Find $e \in E$ with minimal $\omega(u, w)$, for $u \in t_i, w \in t_j, i \neq j$. If no such e exists, **Stop**; the minimal spanning tree $G_T = (V_T, E_T)$ has been found. Otherwise, add e to E_T, set $t_i = t_i \cup t_j$, then delete t_j, Go-to **Step 2**.

In pseudo code:

Algorithm 3.1.3 Kruskal's Algorithm - Input: Connected weighted undirected graph $G = (V, E)$ with weight function ω. Output: Minimal spanning tree $G_T = (V_T, E_T)$.

```
1:  procedure KRUSKALS(V, E, ω)
2:      E_T ← [ ]
3:      V_T ← V
4:      T ← [ [v_1], [v_2], ..., [v_n] ]
5:      E.sort(ω)                          ▷ Sorts edges by weight smallest to largest.
6:      while E ≠ ∅ do
7:          i ← {i : E[0][0] ∈ t_i}         ▷ E[0][0], the starting point of the first edge.
8:          j ← {j : E[0][1] ∈ t_j}         ▷ E[0][1], the endpoint point of the first edge.
9:          if i = j then                  ▷ If both endpoints are in the same tree.
10:             Delete E[0]
11:         else
12:             E_T ← E_T ∪ E[0]
13:             Delete E[0]
14:             t_i ← t_i ∪ t_j            ▷ E[0][0] ∈ t_i and E[0][1] ∈ t_j.
15:             Delete t_j
16:     return V_T, E_T                    ▷ Minimal spanning tree G_T = (V_T, E_T).
```

Python implementation: See Appendix A.4.

Proof of correctness: See Jungnickel 2013 [25, p. 116-117].

Using very different approaches, both of these algorithms succeed in finding a minimal spanning tree. One may ask which algorithm is best? It turns out that Kruskal's algorithm is more efficient than Prim's algorithm; this will be discussed later in this chapter once the idea of complexity has been introduced.

These algorithms highlight the importance of determinism; the minimum spanning tree may not be unique for a given graph, take for example figure 3.1 and figure 3.2. These algorithms give different minimal spanning trees not as a result of the differences in algorithm, but the arbitrary choices between similar valued edges which occurs in both algorithms. Arbitrary choices pose a problem in implementation; determinism dictates that the algorithm should always makes the same choice when presented with more than one optimal choice. This problem is often solved by considering a constant distinct value such as the label of a vertex.

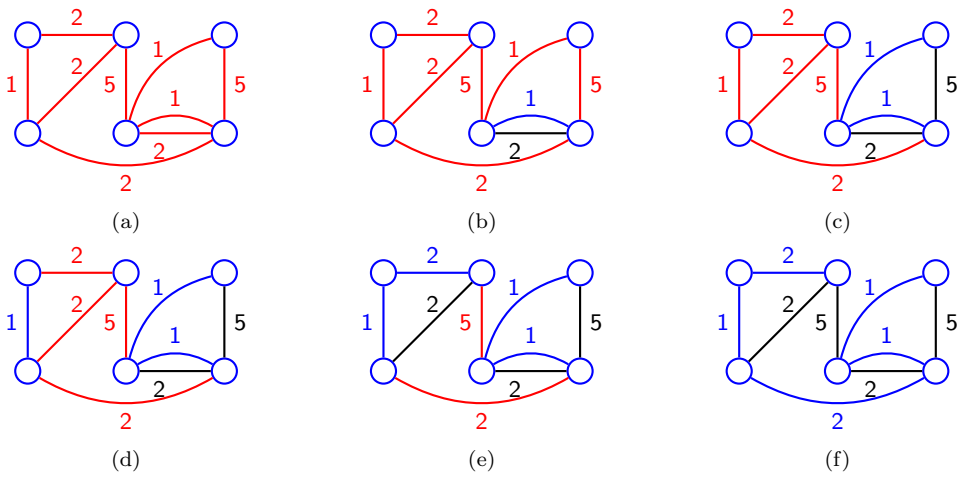


Figure 3.2: Kruskal's Algorithm - $G = (V, E)$ (black), $G_T = (V_T, E_T)$ (blue), edges under consideration (red).

All three algorithms (Borůvka's, Prim's and Kruskal's) are known as greedy algorithms; which takes the optimal choice at each step in an attempt to find the overall optimal way to solve a problem. Greedy algorithms are not always the best choice of algorithm; take for example figure 3.3, which presents the longest directed path problem. Starting the algorithm from vertex 1, as starting the algorithm from another vertex will yield a directed path with a lower number of edges, therefore a shorter directed path (as all weights are strictly positive); from here a greedy algorithm would choose to take the edge of length 6 to vertex 2 instead of the edge of length 1 to vertex 3 (as this is the most optimal choice at this step) and then choose the edge of length 5 to vertex 4; this gives the longest path a length of 11. This is the incorrect answer as taking the path $(1, 3, 5)$ yields a path of length 91. This demonstrates the need for more complex techniques for determining the optimal solution which will be introduced later in this report.

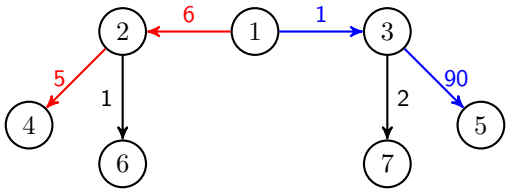


Figure 3.3: Longest directed path problem - Greedy algorithm (red), actual longest path (blue).

3.1.3 Prüfer Code

As previously discussed, the methods for representing graphs allow for storage and communication of graphs. When considering only trees, a representation such as an adjacency matrix has a lot of wasted space; resulting in more expensive storage and communication of the graph. If a graph is assumed to be a tree, then a more efficient representation to use is the *Prüfer code*. Discovered by Heinz Prüfer in 1918 [34], a Prüfer code is a list of $|V| - 2$ integers which encodes any given tree with $|V| > 2$.

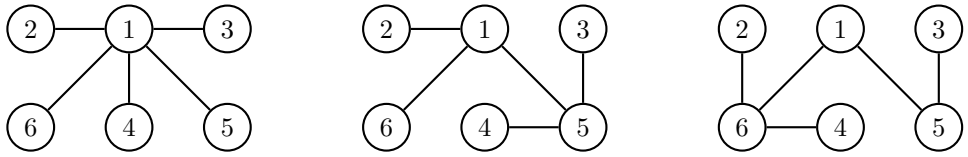


Figure 3.4: Example Prüfer codes.

A Prüfer code, denoted C, is formed for a given tree $G_T = (V_T, E_T)$ (assuming $|V| > 2$), with each vertex labelled by a distinct integer $v_i \in \{1, ..., n\}$. First consider the lowest valued leaf $v_i \in G_T$, find v_j adjacent to v_i; add v_j to the end of the Prüfer code C, then remove v_i from G_T. Continue this process of finding the lowest valued leaf v_i and appending v_j adjacent to v_i to C then removing v_i until only two vertices remain in G_T. That is,

Step 1 Set $C = \emptyset$.

Step 2 If $|V_T| = 2$, **Stop**; C is the desired Prüfer code. Otherwise; find $\min\{i : \deg(v_i \in V) = 1\}$ then find $\{j : v_j \text{ adjacent to } v_i\}$. Append j to the end of C then remove v_i from G_T. Go-to **Step 2**.

In pseudo code:

Algorithm 3.1.4 Prüfer Code - Input: A tree $G_T = (V_T, E_T)$ given as adjacency matrix A_{G_T} ($|V| > 2$). Output: Prüfer code (C).

```
1:  procedure PRUFERCODE(A_{G_T})
2:      A ← A_{G_T}
3:      D ← [A²_{i,i} : i ∈ {1, ..., |V_T|}]
4:      C ← [ ]
5:      while ∑D > 2 do
6:          for i ∈ {1, ..., |D|} do
7:              if d_i = 1 then
8:                  d_i = 0
9:                  for j ∈ {1, ..., |D|} do
10:                     if A_{i,j} = 1 then
11:                         C ← C ∪ j
12:                         d_i ← d_i − 1
13:                         A_{i,j} ← 0
14:                         A_{j,i} ← 0
15:                         Break
                    Break
16:     return C
```

Python implementation: See Appendix A.5.

Theorem 3.1.1. *For a given tree $G_T = (V_T, E_T)$ on n vertices, algorithm 3.1.4 will encode G_T into a unique Prüfer code of size $n - 2$.*

Proof. See Jungnickel 2013 [25, p. 11]. □

Similarly, given Prüfer code of size $n-2$ the steps of algorithm 3.1.4 can be performed in reverse to construct a tree on n vertices. That is,

Step 1 Set $n = |C| + 2$, $V_T = \{v_1, ..., v_n\}$, $i = 1$.

Step 2 Generate a degree sequence D, where the degree of each vertex is the number of times the vertex appears in C plus 1.

Step 3 If $i = n + 1$, Go-to **Step 4**. Otherwise, find the first lowest-numbered vertex, v_j, with degree equal to 1 in D, add the edge (v_j, c_i), $c_i \in C$ to the tree G_T, decrease the degrees of v_j and c_i by 1 in D, then increase i by 1, Go-to **Step 3**.

Step 4 Two vertices $u, v \in V$ with degree 1 will remain in D, add the edge (u, v) to the tree G_T.

In pseudo code:

Algorithm 3.1.5 Prüfer Decode - Input: Prüfer code (C). Output: A tree $G_T = (V_T, E_T)$ given as adjacency matrix (A).

```
1:  procedure PRUFERDECODE(C)
2:      n ← |C| + 2
3:      A ← 0_{n×n}                                  ▷ Zero matrix of size n.
4:      D ← [ d_i = 1 : i ∈ {1, ..., n} ]
5:      for i ∈ C do
6:          d_i ← d_i + 1
7:      for i ∈ C do
8:          for j ∈ {0, ..., n} do
9:              if d_j = 1 then
10:                 A_{i,j} ← 1
11:                 A_{j,i} ← 1
12:                 d_i ← d_i − 1
13:                 d_j ← d_j − 1
14:                 Break
15:     for j ∈ {0, ..., n} do
16:         if d_j = 1 then
17:             for i ∈ {j + 1, ..., n} do
18:                 if d_i = 1 then:
19:                     A_{i,j} ← 1
20:                     A_{j,i} ← 1
21:                     Break
22:             Break
23:     return A
```

Python implementation: See Appendix A.5.

Theorem 3.1.2. *Let $C = (c_1, ..., c_{n-2})$ be an ordered sequence with $c_i \in V_T = \{v_1, ..., v_n\}$. Then C is a Prüfer code which is decoded by algorithm 3.1.5 into a unique tree $G_T = (V_T, E_T)$ on n vertices.*

Proof. See Jungnickel 2013 [25, p. 11]. □

Theorem 3.1.3. *There exists a bijective mapping between the set of trees on n vertices and the set of Prüfer codes of size $n-2$, for all $n \in \mathbb{Z}$, $n \geq 2$.*

Proof. Theorem 3.1.1 implies that there is an injective mapping from the trees on n vertices to Prüfer codes of size $n-2$. Similarly theorem 3.1.2 implies there is an injective mapping from the Prüfer codes of size $n-2$ to trees on n vertices. Hence, there exists a bijective mapping between the set of trees on n vertices and the set of Prüfer codes of size $n-2$. □

Corollary 3.1.1. *The number of trees on n vertices is equal to the number of Prüfer codes of size $n-2$.*

Heinz Prüfer first used his code in 1918 [34] as an alternative proof to following formula; first discovered in 1860 by Carl Wilhelm Borchardt [1] and later expanded on by Arthur Cayley [5] in 1889 , from which the formula gets its name:

Theorem 3.1.4. *(Cayley's formula) The number of trees on n labelled vertices is n^{n-2}.*

Proof. The number of Prüfer codes of size $n-2$ on n vertices is n^{n-2} (n choices for $n-2$ elements). Then by corollary 3.1.1, the number of trees on n vertices is also n^{n-2}. □

A theoretical system which stores trees using a Prüfer code would need to decode them into a more standard representation so that standard operations and analysis can be performed. This adds to the complexity and expense of an algorithm. Therefore in the real world there are very limited practical applications of the Prüfer code, systems opting for a representations such as an edge list instead. This code does however highlight the usage of different representations to discover underlying properties of more general graphs. This idea of using non-standard representations, will prove to be a powerful tool when considering more complex problems later in this report.

3.1.4 Halting Problem

It may be intuitive to think that for any problem, given enough time, an algorithm (which may not yet be discovered) can find a solution to that problem. Unfortunately, it turns out that for some problems an algorithm simply cannot exist. One such problem is the halting problem, which asks, for a given arbitrary algorithm X and an input Y, whether X will terminate (halt) or continue to run forever with input Y. This decision problem was shown by Alan Turing [37] to be *undecidable*, that is, there is no general algorithm which leads to a correct yes or no answer.

Theorem 3.1.5. *The halting problem is undecidable.*

Proof. Assume that there exists an algorithm A, which solves the halting problem. The algorithm A takes two inputs, an algorithm X and an input Y, returning TRUE if the input Y causes X to halt, or FALSE if this input Y causes X to never terminate. Now consider the algorithm A' (figure 3.5); this algorithm takes the output of algorithm A and loops forever if A returns TRUE, otherwise A' will halt.

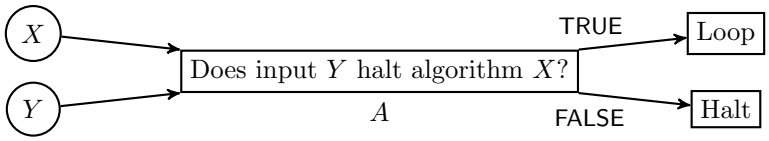


Figure 3.5: The algorithm A'.

By assumption algorithm A can determine if the algorithm A' will halt for any given input. However, consider $X = A'$ and $Y = A'$ as inputs to algorithm A'. If the input A' halts algorithm

A' then algorithm A will return TRUE, hence A' will loop forever contradicting the output of algorithm A. Similarly if input A' causes algorithm A' to loop forever then algorithm A will return FALSE, hence A' will halt contradicting the output of algorithm A. The algorithm A therefore cannot exist and the halting problem is undecidable. □

3.2 Complexity Theory

The notation and content of this section closely follow that of Jungnickel 2013 [25, p. 46-51].

As introduced by Jungnickel 2013 [25, p. 53], there are three main types of problem which this report will focus on within complexity theory, *decision problems*, *evaluation problems* and *optimisation problems*. Decision problems aim to give a yes or no answer to a problem, these are used in situations where knowing a solution exists, rather than the solution itself, is sufficient, much like deciding if a degree sequence is graphic. Evaluation problems ask for the value of an optimal solution without specifically requiring the exact solution to be obtained. Take for example obtaining the weight of a minimal spanning tree without actually obtaining the minimal spanning tree, it may not be obvious how this can be achieved however later in this report, max-min relations will give the means to relate problems for evaluation. Each evaluation problem gives rise to a decision problem, mainly, is the value of the evaluation problem equal to the value of the optimal solution? Finally, optimisation problems focus on finding the optimal solution to a problem, given a certain set of criteria. Each optimisation problem induces a decision problem, mainly, is there a more optimal solution? Finding a solution to an optimisation problem implies the solution to both the associated evaluation problem and decision problem. Later in this report, two main optimisation problems will be covered, mainly, the maximal flow problem and the edge connectivity augmentation problem.

3.2.1 Types of Complexity

This report will focus on two main types of complexity; the first, *space complexity*, is the measure of the storage cost of data during the operation of algorithm. The second, *time complexity*, is a measure of how many operations an algorithm must perform before it will terminate. For any given algorithm the time complexity will always be at least equal to the space complexity, as data must be written to memory before it can be accessed by an algorithm. The idea of space complexity has already been introduced for different representations of graphs. Some representations, such as adjacency matrices required a large array of size $|V|^2$ for storage, on the other hand an edge list only requires $2|E| + 1$ integers.

3.2.2 Measuring Complexity

Often it is the case that the complexity of an algorithm cannot be assigned an exact value. For some inputs the running time of an algorithm may be small, but for others it may not even terminate. Therefore, the growth in complexity of an algorithm can be estimated by considering another function $g : \mathbb{N} \mapsto \mathbb{R}^+$ which can then be related to the complexity of the algorithm $f(n)$ (as in Jungnickel 2013 [25, p. 47]) using the following notation,

$$f(n) = \mathcal{O}(g(n)), \text{ if } \exists\, c > 0 : f(n) \leq cg(n), \text{ for all } n \text{ sufficiently large}, \tag{3.5}$$

$$f(n) = \Omega(g(n)), \text{ if } \exists\, c > 0 : f(n) \geq cg(n), \text{ for all } n \text{ sufficiently large}, \tag{3.6}$$

$$f(n) = \Theta(g(n)), \text{ if } f(n) = \mathcal{O}(g(n)) \text{ and } f(n) = \Omega(g(n)). \tag{3.7}$$

It is said that f has rate of growth $g(n)$ if $f(n) = \Theta(g(n))$. Similarly f is said to have rate of growth at most g if $f(n) = \mathcal{O}(g(n))$ and f has rate of growth at least g if $f(n) = \Omega(g(n))$.

Given an algorithm A, with time or space complexity $\mathcal{O}(g(n))$, then A is said to have complexity $\mathcal{O}(g(n))$.

3.2.3 Complexity of Minimum Spanning Tree Algorithms

As seen with Prim's and Kruscal's algorithms for minimal spanning trees, a given problem can have multiple algorithms which realize a solution. Studying the complexity of these algorithms determines which algorithms will be most efficient for a given input. Consider the algorithms of Prim and Kruscal; the time complexity of these algorithms is the following,

Theorem 3.2.1. *Given a connected weighted graph $G = (V, E)$, Prim's Algorithm can find a minimal spanning tree of G with time complexity $\mathcal{O}(|V|^2)$.*

Proof. To build the edge set for a minimal spanning tree of G, **Step 2** in the algorithm must run $|V|$ times. During each iteration of **Step 2** at most $|V| - |V_T|$ comparisons of edge weights are required. Hence, the algorithm has time complexity $\mathcal{O}(|V|^2)$. □

Theorem 3.2.2. *Given a connected weighted graph $G = (V, E)$, Kruscal's Algorithm can find a minimal spanning tree of G with time complexity $\mathcal{O}(|E| \log(|E|))$.*

Proof. The edges of a given graph $G = (V, E)$ can be sorted by weight with time complexity $\mathcal{O}(E \log(E))$ using a merge sort algorithm [7]. From this sorted list, the minimal spanning tree can be constructed with time complexity $\mathcal{O}(E)$. Hence, the algorithm has time complexity $\mathcal{O}(|E| \log(|E|))$ □

By computing these complexities, it then follows that, in a worst case scenario Kruscal's Algorithm will be more efficient than Prim's Algorithm. That is not to say that Kruscal's algorithm will always yield a faster result, or that any implementation of Kruscal's and Prim's algorithm will run with time complexity $\mathcal{O}(|E| \log(|E|))$ and $\mathcal{O}(|V|^2)$ respectively. This all depends on the data structures used and which graph representation is given to the algorithm. For instance if an adjacency matrix is used, the space complexity of both algorithms is $\mathcal{O}(|V|^2)$, hence both algorithms will have time complexity $\mathcal{O}(|V|^2)$ as in both cases, the adjacency matrix must be written into the computers' memory before the algorithm can begin.

Chapter 4

Flow Networks

This chapter will examine a special type of digraph called a *flow network*. A flow network $N = (G, c, s, t)$ is defined by a digraph $G = (V, E)$ with vertex set V which contains a source s and a sink t. Each edge $e \in E$, must also have a *capacity*, given by a function $c : E \mapsto \mathbb{R}_0^+$ and *flow*, given by a function $f : E \mapsto \mathbb{R}$, which satisfies the following criteria (as given in Ibaraki and Nagamochi 2008 [23, p. 20-21]) for all $e \in E$,

$$0 \leq f(e) \leq c(e), \tag{4.1}$$

$$\sum_{e^+=v} f(e^-, e^+) - \sum_{e^-=v} f(e^-, e^+) \qquad \begin{cases} = 0 & v \neq s, t \in V, \\ \geq 0 & v = s, \\ \leq 0 & v = t. \end{cases} \tag{4.2}$$

As stated previously, e^- and e^+ denote the starting endpoint and final endpoint of the directed edge e, respectively.

Equation 4.1 is known as the *feasibility condition* of a flow network and equation 4.2 is known as the *flow conservation condition*; with any flow which satisfies these conditions is called a *feasible flow*. The feasibility condition dictates that the flow must be none-negative and cannot exceed the capacity of a given edge. The flow conservation condition dictates that flow is preserved for all vertices, except the source s, where flow is being introduced to the system, and the sink t, where the flow is being taken out of the system. When flow is measured in opposing direction to an edge this results in a negative net flow, that is,

$$f(e^-, e^+) = -f(e^+, e^-). \tag{4.3}$$

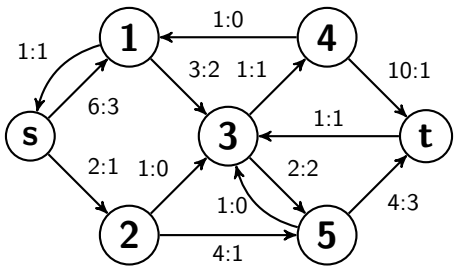


Figure 4.1: Flow network - Each edge labelled $c(e) : f(e)$.

Aside: In the real world, electricity networks follow Kirchhoff's Circuit Laws [17, p. 797]. The first of these laws, Kirchhoff's Current Law (Kirchhoff's Vertex Law), which says the sum of currents meeting at a point is zero. By considering all vertices, expect for the sink and the source, in a flow network, this is exactly the flow conservation condition.

4.1 Maximum Flow

Flow networks describe the movement of units from one destination to another. The natural question which follows is then what is the maximal flow. This section focuses on presenting a method to find the distribution of flow units which gives this maximum flow through a network. This distribution will later be exploited through the use of elegant max-min relations to solve seemingly unrelated problems in flow and connectivity. In particular, calculating the connectivity invariats in the next chapter.

A bound on the maximum flow can be established using the idea of a *cut* on a flow network. A cut (S,T) of a flow network $N=(G,c,s,t)$ is a partition of V such that $s\in S, t\in T$ with f a feasible flow on N. Cuts have capacity denoted $c(S,T)$ and flow, denoted $f(S,T)$, defined as,

$$c(S,T) = \sum_{e^-\in S, e^+\in T} c(e), \tag{4.4}$$

$$f(S,T) = \sum_{e^-\in S, e^+\in T} f(e) - \sum_{e^+\in S, e^-\in T} f(e). \tag{4.5}$$

Hence, the flow of a cut (S,T) is the net amount of flow moving from the set of vertices S to the set of vertices T. When considering a maximum flow, the distribution will not actively change hence, the amount of flow units into (and out of) the system will remain a constant value; this value is known as the *value of flow*, denoted $w(f)$. Given a flow network $N=(G,c,s,t)$ and f a feasible flow, then the value of flow, $w(f)$ is defined as follows,

$$w(f) = \sum_{e^-=s} f(e) - \sum_{e^+=s} f(e) = \sum_{e^+=t} f(e) - \sum_{e^-=t} f(e). \tag{4.6}$$

This definition for the value of flow, then implies that $w(f)=f(\{s\},V\setminus\{s\})=f(\{s\},V)$. In fact the following lemma shows the value of flow and the flow of a cut are one and the same.

Lemma 4.1.1. *For any cut (S,T) of a flow network N with feasible flow f, then $w(f)=f(S,T)$.*

Proof.

$$\begin{aligned} f(S,T) &= f(S,V)-f(S,S), \\ &= f(S,V), \\ &= f(\{s\},V)+f(S\setminus\{s\},V), \\ &= f(\{s\},V), \\ &= w(f). \end{aligned}$$

□

Therefore, the flow of any (S,T) cut is a constant. If a minimum upper bound for the flow of a (S,T) cut is found then this is the maximum possible value of flow. That is,

Theorem 4.1.1. *The value of any flow $w(f)$ is bounded above by the capacity of any cut $c(S,T)$.*

Proof.

$$\begin{aligned} w(f) &= f(S,T), \\ &= \sum_{e^- \in S, e^+ \in T} f(e), \\ &\leq \sum_{e^- \in S, e^+ \in T} c(e), \\ &= c(S,T). \end{aligned}$$

□

Note that the value of flow being equal to the capacity of a cut, $w(f) = c(S,T)$, is achievable if and only if each edge e with $e^- \in S$ and $e^+ \in T$ is *saturated* ($f(e) = c(e)$), whereas each edge e with $e^+ \in S$ and $e^- \in T$ is *void* ($f(e) = 0$). Another consequence of theorem 4.1.1 is that if there exists a flow such that $f(S,T) = c(S,T)$ for any cut (S,T) then this flow is maximal with value $w(f)$ and the cut (S,T) has minimal capacity. This consequence is known as the Max-Flow Min-Cut Theorem, which was proven by Lester Randolph Ford Jr. and Delbert Ray Fulkerson [16] in 1956. Formally,

Theorem 4.1.2. (Max-Flow Min-Cut Theorem). *The maximal value of a flow on a flow network N is equal to the minimal capacity of an (S,T) cut on N.*

Proof. Suppose that $w(f)$ is maximal with $w(f) = c(S,T)$, for some cut (S,T). If $c(S,T)$ was not minimal, then there exists some cut (S',T') for which $c(S',T') < c(S,T) = w(f)$, but this contradicts theorem 4.1.1, hence $c(S,T)$ must be minimal. Now consider $w(f)$ is maximal with $w(f) < c(S,T)$, for $c(S,T)$ minimal. But as $w(f) < c(S,T)$, there must be either some edges from S to T which are not saturated, or some edges from T to S which are not void. Therefore, $w(f)$ can be increased, but this is a contraction as $w(f)$ is maximal. □

This theorem then gives a shallow method for calculating the maximum flow; simply calculate the $\min(c(S,T))$ for all possible (S,T) cuts. This method does not give any indication of how a system with maximum flow would have its flow distributed in the network nor is it efficient.

To begin formulating an effective algorithm to find the exact distribution of a maximal flow network, consider a path P from s to t which can travel in any direction along a directed edge. If an edge e is in the direction of P, then e is called a *forward edge*, similarly if an edge e opposes the direction of P, then e is called a *backward edge*. Increasing the flow from s to t along such a path, works towards determining a maximum flow distribution.

Assuming that the source, s, has an infinite supply of flow units; their are three ways to increase the flow of units from s to t, along such a path P, while maintaining the feasibility condition. If P contains only forward edges, with all edges satisfying $f(e) < c(e)$, then flow along these edges can be increased by adding the smallest difference between the flow and the capacity of each edge in P, that is, $\min\{r(e) : r(e) = c(e) - f(e), e \in P\}$ ($r(e)$ is called the *residual capacity* of the edge e).

If P contains only backward edges, with all edges satisfying $f(e) > 0$, then the flow of this path can be increased by reducing the flow value of each edge by the minimum flow of any edge in P, that is, $\min\{f(e) : e \in P\}$.

If P is a combination of forward edges ($e \in P_{\text{forward}}$) and backward edges ($e \in P_{\text{backward}}$), with every forward edge satisfying $f(e) < c(e)$, and backward edge satisfying $f(e) > 0$, then increasing the flow can be done by applying a combination of both increasing flow in forward edges and reducing flow in backward edges by the minimum of the following quantities: $\min\{r(e) : e \in$

$P_{\text{forward}}\}$ and $\min\{f(e) : e \in P_{\text{backward}})\}$. Such paths for which flow can be increased are called *flow augmenting paths*. That is, given a flow network $N = (G, c, s, t)$ and feasible flow f, then a path P from vertex s to vertex t is said to be a *flow augmenting path* with respect to f, if $f(e) < c(e)$ holds for every forward edge $e \in P$ and $f(e) > 0$, for every backward edge $e \in P$.

Using these three methods for increasing flow, an augmented flow f^* is defined by,

$$\varphi(P) = \min\{\min\{r(e) : e \in P_{\text{forward}}\}, \min\{f(e) : e \in P_{\text{backward}}\}\}, \tag{4.7}$$

$$f^*(e) = \begin{cases} f(e), & \text{if } e \notin P, \\ f(e) + \varphi(P), & \text{if } e \in P_{\text{forward}}, \\ f(e) - \varphi(P), & \text{if } e \in P_{\text{backward}}. \end{cases} \tag{4.8}$$

Note: The quantity in equation 4.7 is denoted $\varphi(P)$ instead of $\delta(P)$, as is conventional in literature, to avoid confusion with $\delta(G)$.

Theorem 4.1.3. *The augmented flow f^*, maintains a feasible flow on N.*

Proof. See Kocay and Kreher 2013 [28, p. 166-167]. □

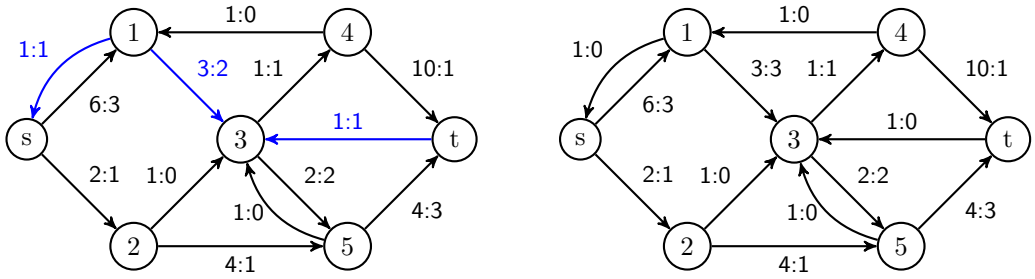


Figure 4.2: Initial flow distribution (left), augmenting path (blue) with $\varphi(P) = 1$. Augmented flow (right).

Theorem 4.1.4. (Augmenting Path Theorem). *A feasible flow f, on a flow network $N = (G, c, s, t)$ is maximal if and only if there are no augmenting paths with respect to f.*

Proof. (Based on Ford and Fulkerson 1956 [16]) Assume that f is a maximal flow. Suppose an augmenting path P exists. Then $\varphi(P) > 0$ by the definition of augmenting path and f^* can be applied to the flow. By theorem 4.1.3, f^* defines a feasible flow with value $w(f^*) = w(f) + \varphi(P) > w(f)$, this is a contradiction as $w(f)$ is assumed to be maximal. Now, assume there are no augmenting paths in the network N with feasible flow f. Then there must exist an (S, T) cut such that each forward edge in the set-cut is saturated and each backward edge is void. Then theorem 4.1.1 implies $w(f) = c(S, T)$ therefore f is maximal. □

Finding flow augmenting paths then applying f^* to the network, forms the basis of a method to find the maximal flow distribution; first presented in 1956 by Lester Randolph Ford Jr. and Delbert Ray Fulkerson [16]; the following steps determine the maximal flow for (integral and rational) flow networks:

Step 1 Set the flow to the *zero flow*, that is $f(e) = 0$ for all $e \in E$.

Step 2 Find an augmenting path P in N and update the flow by applying $f^*(e)$. If no augmenting path exists **Stop**; maximal flow has found. Otherwise Go-to **Step 2**.

This technique is referred to as the Ford-Fulkerson Method. Named a method and not an algorithm as the act of finding an augmenting path is not trivial. In the same 1956 paper [16], Ford and Fulkerson outline a more precise algorithm which details how an augmenting path may be found, mainly the Ford-Fulkerson Labelling Algorithm.

4.2 Ford-Fulkerson Labelling Algorithm

As will be demonstrated in the next section, this algorithm by Ford and Fulkerson is not guaranteed to terminate for all inputs. The following theorem ensures that as long as a flow network is integral or rational in its capacities, the proceeding algorithm will terminate,

Theorem 4.2.1. (Intergral Flow Theorem). *Let $N = (G, c, s, t)$ be a flow network where all capacities $c(e)$ are integers. Then there is a maximal flow on N such that all values $f(e)$ are integral.*

Proof. (Based on Ford and Fulkerson 1956 [16] and Jungnickel 2013 [25, p. 166]) First consider the zero flow, $f(e) = 0$ for all e; if this is the maximal flow then the value of flow is an integral flow on N, with value 0. Otherwise, by theorem 4.1.4, there exists an augmenting path P with respect to f. Then $\varphi(P)$ is a positive integer as all capacities are integer, therefore the flow $f \leftarrow f^*$, must also be integral with value $\varphi(P)$. If f is not maximal then augmenting paths can be found and value of flow increased in the same way. As the value of the flow is increased by $\varphi(P) > 0$ (a positive integer) and theorem 4.1.1 dictates the capacity of any cut is an upper bound on the value of the flow, then a finite number of steps achieves an integral flow f with no augmenting paths which by theorem 4.1.4, dictates that the flow f is maximal. □

Theorem 4.2.1 dictates that an integral maximal flow exists. However, a non-integral maximal flow may also exist for a given network. Theorem 4.2.1 also implies that a network with all rationals capacities has a maximal flow with all values $f(e)$ rational which can be found in finite steps. By considering scaling all capacities by a value $\gamma \in \mathbb{Q}$ such that $\gamma c(e)$ are integral, then by theorem 4.2.1 an integral maximal flow exists which can then be scaled by γ^{-1} to obtain a maximal flow for the original rational network. Theorem 4.2.1 supports the algorithm; if there were no integral solution the algorithm would never terminate under the integer operations of f^*.

The general idea of this algorithm is to form an augmenting path through the network, by testing routes to the sink using only edges which can have their flow increased. Starting with the zero flow, label the source with $(-, -, \infty)$, then look at all of the edges incident at the source to an unlabelled vertex (initially all other vertices will be unlabelled). If any forward edges are not saturated or backward edges void, then choose one of these edges to the connected vertex v_j. Label v_j with $(v_i, +/-, Rj)$ where v_i is the previous vertex, $+/-$ is the type of edge traversed ($+$: Forward, $-$: Backward) from v_i. Rj is then the minimum of R_i and the amount the flow can be increased along the traversed edge $\varphi(e)$. For example, the label for a vertex v_2, $(s, +, 5)$ means vertex v_2 is reached via vertex s along a forward edge with maximal possible flow increase 5. Repeat this process of labelling by moving along edges where flow can be increased to an unlabelled vertex until the sink is labelled. If at any stage no such edges exist from the current vertex, backtrack to the previous vertex. If the algorithm is at the source and cannot find any edges to unlabelled vertices which can have flow increased before labelling the sink, then the flow is maximal. Otherwise, once the sink is labelled, an augmenting path with $\varphi(P) = R_t$ can be formed in reverse by reading the label at the sink to obtain the previous vertex and the type of edge needed to traverse to the sink. The label of the previous vertex can then be read to obtain the next previous vertex and associated type of edge traversed. This process of reading

the labels and adding the edge to the augmenting path occurs until the algorithm gets back to the source. The flow can be increased along the augmenting path by applying $f \leftarrow f^*$. All the labels are then removed and the algorithm starts again by labelling the source with $(-, -, \infty)$; repeating until maximal flow is found. That is,

Step 1 Set $f(e) = 0$ for all e.

Step 2 Label s with $L_s = (-, -, \infty)$, (i.e. $L_s[0] = -, L_s[1] = -, L_s[2] = \infty$) . Let $v_i = s$.

Step 3 Find an unlabelled v_j, such that, the forward edge $e = (v_i, v_j) \in E$ has $r(e) = c(e) - f(e) > 0$ or the backward edge $e = (v_j, v_i) \in E$ has $r = f(e) > 0$. If no such v_j exists, Go-to **Step 4**. Otherwise; label v_j with $L_{v_j} = (v_i, +/-, \min(r, L_{v_i}[2]))$, giving $+$ if e is a forward edge and $-$ if e is a backward edge, set $v_i = v_j$ then Go-to **Step 5**.

Step 4 If $v_i = s$, **Stop**; the flow f on N is maximal. Otherwise; set $v_i = L_{v_i}[0]$, Go-to **Step 3**.

Step 5 If $v_i = t$, Go-to **Step 6**. Otherwise; Go-to **Step 3**.

Step 6 Set $P = [\,]$ and $\varphi(P) = L_t[1]$. Go-to **Step 7**.

Step 7 If $v_i = s$, apply $f(e) \leftarrow f^*(e)$, remove all labels, Go-to **Step 2**. Otherwise, if $L_{v_i}[1] = +$ then add the edge $(v_i, L_{v_i}[0])$ to P, else; $L_{v_i}[1] = -$ so add the edge $(L_{v_i}[0], v_i)$ to P. Set $v_i = L_{v_i}[0]$, Go-to **Step 7**.

Theorem 4.2.2. *In an integral flow network the Ford-Fulkerson Labelling Algorithm finds the the distribution of flow units which realizes a maximal value of flow f, with time complexity $\mathcal{O}(|E|f)$.*

Proof. An augmenting path is found in $\mathcal{O}(|E|)$ time complexity. For an integral flow network each augmenting path increases the flow of the network by at least 1. Therefore, the algorithm must find at most f augmenting paths, hence has time complexity $\mathcal{O}(|E|f)$. □

In pseudo code:

Algorithm 4.2.1 Ford Fulkerson Labelling Algorithm - Input: Directed graph $N = (V, E)$, capacity function (c), source node (s), sink node (t). Output: Flow distribution (f).

```
1:  procedure MaxFlow(N, c, s, t)
2:      for e ∈ E do f(e) ← 0                                   ▷ Zero flow.
3:      while True do                                           ▷ Loop forever.
4:          L ← [ [−, −, ∞]_s ]                                 ▷ Label s.
5:          i ← s
6:          while True do
7:              for v_j not labelled do
8:                  if (v_i, v_j) ∈ E and f(v_i, v_j) < c(v_i, v_j) then   ▷ Forward Edge.
9:                      labels.append([v_i, +, min(c(v_i, v_j) − f(v_i, v_j), L_{v_i}[2])]_{v_j})   ▷ Label v_j.
10:                     i ← j
11:                     break
12:                 else if (v_j, v_i) ∈ E and f(v_j, v_i) > 0 then   ▷ Backward Edge.
13:                     L.append([v_i, −, min(f(v_j, v_i), L_{v_i}[2])]_{v_j})   ▷ Label v_j.
14:                     i ← j
15:                     break
16:             if v_i = s then return f                        ▷ Flow is maximal
17:             if v_i = t then break                           ▷ Augmenting Path Found.
18:             v_i ← L_{v_i}[0]                                ▷ Label v_i.
19:         φ ← L_t[2]                                          ▷ Maximal flow increase.
20:         augPathForward ← [ ]                                ▷ Forward edges in path.
21:         augPathBackward ← [ ]                               ▷ Backward edges in path.
22:         while v_i ≠ s do                                    ▷ Generates the augmenting path.
23:             v_j ← L_{v_i}[0]
24:             if L_{v_i}[1] = + then
25:                 augPathForward.append((v_j, v_i))
26:             else
27:                 augPathBackward.append((v_i, v_j))
28:             v_i ← L_{v_i}[0]
29:         for e ∈ augPathForward do                           ▷ Augment forward edges
30:             f(e) ← f(e) + φ
31:         for e ∈ augPathBackward do                          ▷ Augment backward edges
32:             f(e) ← f(e) − φ
```

Python implementation: See Appendix A.6.

4.2.1 Cautionary Example

For integer and rational flow networks, the Ford-Fulkerson Method will always converge to the maximal flow distribution. However, when working with irrational flow networks, the Ford-Fulkerson Method requires some caution when choosing the augmenting path. This report will not focus on a maximal flow algorithm for an irrational flow network (for this the reader is advised to use the Edmonds-Karp Algorithm [10]), but instead will introduce an example presented by Uri Zwich in 1995 [40], where the Ford-Fulkerson Method breaks down.

Theorem 4.2.3. *The Ford-Fulkerson Method is not guaranteed to terminate for real valued capacities. Moreover, the Ford-Fulkerson Method may converge to a value of flow not equal to the maximal flow.*

Proof. (Based on Zwich 1995 [40]) Consider the flow network as presented in figure 4.3 with:

$$r = \frac{\sqrt{5}-1}{2}. \tag{4.9}$$

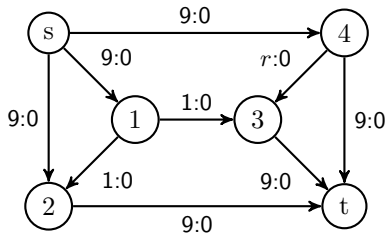
Then consider the sequence of augmenting paths as presented in the table below with integer valued $k \geq 2$:

Path Number	Path	$\varphi(P)$	$w(f)$
1	$[s,1,3,t]$	1	1
2	$[s,4,3,1,2,t]$	r	$1+r$
3	$[s,1,3,4,t]$	r	$1+2r$
4	$[s,4,3,1,2,t]$	r^2	$1+2r+r^2$
5	$[s,2,1,3,t]$	r^2	$1+2r+2r^2$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$4k-2$	$[s,4,3,1,2,t]$	r^{2k-1}	$1+2r+...+2r^{2k-2}+r^{2k-1}$
$4k-1$	$[s,1,3,4,t]$	r^{2k-1}	$1+2r+...+2r^{2k-1}$
$4k$	$[s,4,3,1,2,t]$	r^{2k}	$1+2r+...+2r^{2k-1}+r^{2k}$
$4k+1$	$[s,2,1,3,t]$	r^{2k}	$1+2r+...+2r^{2k}$

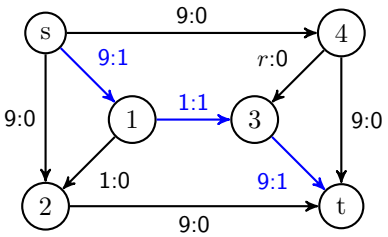
The maximum value of flow is equal to 19, realized by 9 units via [s,4,t], 9 units via [s,2,t] and 1 unit via [s,1,3,t]. However, taking suggested sequence of augmenting paths (which can always be taken [40]), then measuring the value of flow after the path $4k+1$ has been augmented, yields the value of flow equal to:

$$\begin{aligned} w(f) = 1 + 2\sum_{i=1}^{2k} r^i &\leq 1 + 2\sum_{i=1}^{\infty} r^i, \\ &= 3 + 2r, \\ &< 5. \end{aligned} \tag{4.10}$$

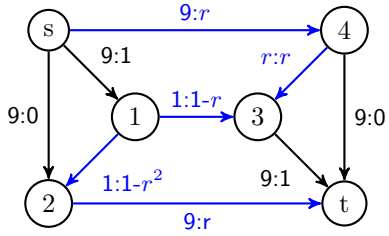
Therefore, this sequence of augmenting paths is infinite and does not converge to the maximal value of flow of the network. □



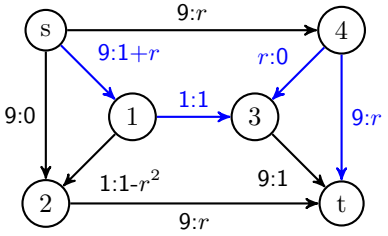
Initial Flow Network



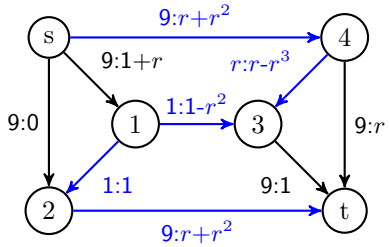
Path 1



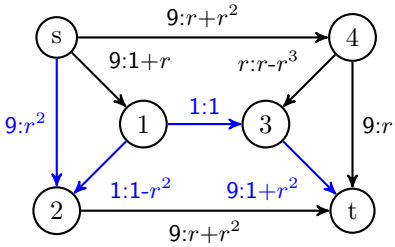
Path 2



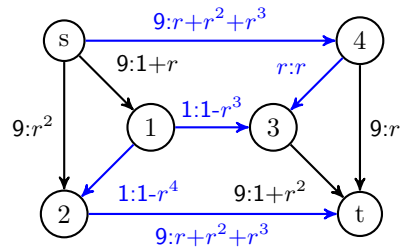
Path 3



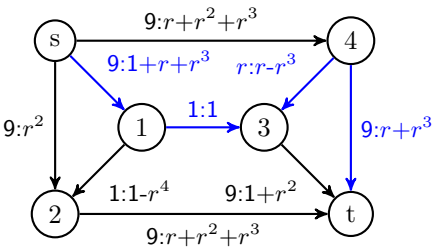
Path 4



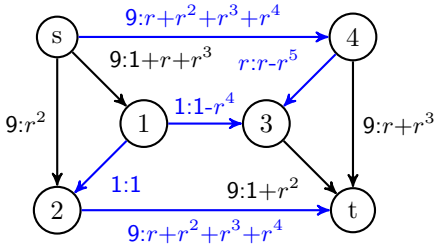
Path 5



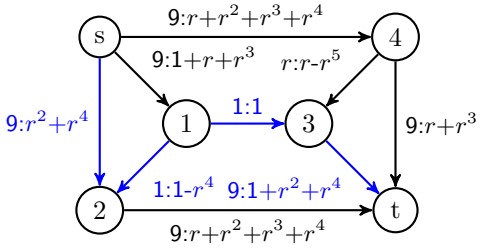
Path 6



Path 7



Path 8



Path 9

Figure 4.3: Example of the Ford-Fulkerson Method on an irrational flow network.

Chapter 5

Connectivity

Connectivity is a measure by which the robustness of a system (and its graph representation) can be quantified and studied for weaknesses. This chapter will introduce the basic notions of vertex and edge connectivity and give explicit steps in calculating these connectivity invaritants using flow networks. Fundamental results related to the duality of vertex connectivity through Menger's Theorem and Whitney's Theorem will also be studied in this chapter. Whitney's Inequality will then provides a relation between these two connectivity invariants and the minimal vertex degree. This chapter will also demonstrate how graphs with specified connectives can be constructed.

5.1 Vertex Connectivity

The first measure of connectivity looks at the robustness of graphs where vertices take precedence. Consider a graph $G = (V, E)$, then a *vertex-cut* on G is any set of vertices $S_V \subset V$ such that the induced sub-graph $G \setminus S_V$ is disconnected. It is not always the case that a vertex-cut exists, take for example the complete graph on n vertices, $K_n = (V, E)$. For any $S_V \subset V$ the induced sub-graph $K_n \setminus S_V$ is connected (note: $|S_V| < |V|$ hence, the null graph is not considered.). The *vertex connectivity* of a graph G, denoted $\kappa(G)$, is defined as the minimal cardinality of all vertex-cuts on G. If the set of vertex-cuts is the empty set, then vertex connectivity is defined as, $\kappa(G) = |V| - 1$, the smallest number of vertices whose removal forms an induced sub-graph with exactly one vertex.

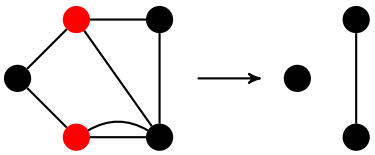


Figure 5.1: Vertex-cut (red) on G with $\kappa(G) = 2$.

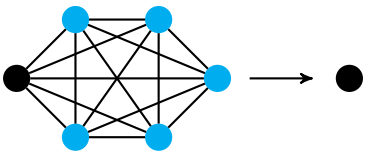


Figure 5.2: $\kappa(K_6) = 5$.

For a disconnected undirected graph G, $\kappa(G) = 0$. For any graph G with $\kappa(G) \geq 1$, then G is said to be 1-connected, likewise if G has $\kappa(G) \geq k$, then G is said to be k-connected.

Given a graph G, with $|V| \geq 2$, *specific vertex connectivity* of two non-adjacent vertices $s, t \in V$, denoted $\kappa(s, t)$, is defined as the smallest cardinal of a vertex-cut $S_V \subset V \setminus \{s, t\}$ for which the induced sub-graph $G \setminus S_V$ has s and t in different connected components (s is unreachable

from t). If s and t are adjacent vertices then $\kappa(s,t)$ is undefined; making s unreachable from t is impossible as a vertex cut S_V cannot remove the edge (s,t).

5.2 Edge Connectivity

Much like a vertex-cut, for a given graph $G = (V, E)$ an *edge-cut* is any set of edges $S_E \subseteq E$ such that the spanning sub-graph $G \setminus S_E$ is disconnected. If a graph G has exactly one vertex then no edge-cuts exists. The *edge connectivity* of a graph G, denoted $\lambda(G)$, is defined as the minimal cardinality of all edge-cuts on G. If the set of edge-cuts is the empty set, then edge connectivity is defined as, $\lambda(G) = 0$. If G is an undirected weighted graph, then the edge connectivity of G is defined as the minimal weight of all edge-cuts S_E on G.

Note: When considering the edge connectivity of weighted graphs this report will only consider integral weight functions.

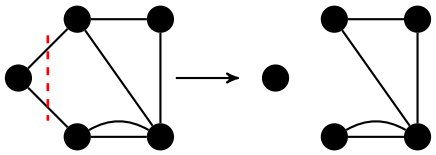


Figure 5.3: Edge-cut (red), $\lambda(G) = 2$.

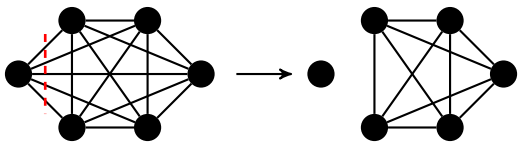


Figure 5.4: Edge-cut (red), $\lambda(G) = 5$.

For trivial graph or a disconnected undirected graph G, $\lambda(G) = 0$. For a graph G with $\lambda(G) \geq 1$, then G is said to be 1-edge connected, likewise if G has $\lambda(G) \geq k'$, then G is said to be k'-edge connected.

Given a graph G, with $|V| \geq 2$, the *specific edge connectivity* of two vertices $s, t \in V$, denoted $\lambda(s,t)$, is defined as the smallest cardinality of an edge-cut $S_E \subseteq E$ for which the spanning sub-graph $G \setminus S_E$ has s and t in different connected components (s is unreachable from t).

5.3 Computing Edge Connectivity

Earlier in this report, the Max-Flow Min-Cut Theorem gave a method for obtaining the minimal capacity of a cut between two vertices within a flow network. An algorithm to compute the edge connectivity of a graph takes advantage of this theorem by first transforming a given graph $G = (V, E)$ into a flow network $N = (G, c, s, t)$; replacing each undirected edge in G with two oppositely directed edges, both with capacity equal to one, this is called a *zero-one flow network*. In forming this zero-one flow network, an (S, T) cut in G which induces an edge-cut has cardinality k' if and only if $c(S, T) = k'$ in N. Therefore, a minimal capacity cut in N corresponds to a minimal edge-cut in G. Hence, the specific edge connectivity $\kappa(s,t)$, $s, t \in V$, can be computed using a maximal flow algorithm (such as algorithm 4.2.1), to determine the maximal flow distribution between s and t. Max-Flow Min-Cut Theorem then implies the value of flow is then equal to the minimal edge-cut which separates s from t in G.

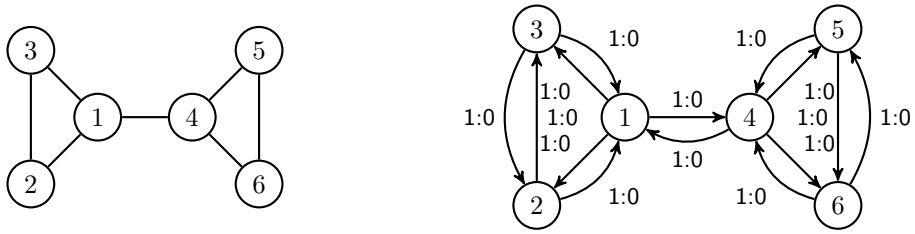


Figure 5.5: Undirected graph G, with corresponding zero-one flow network N.

Using this method for calculating specific edge connectivity, the edge connectivity of the entire graph can be obtained using the following theorem by Claus-Peter Schnorr [35],

Theorem 5.3.1. *Given a graph $G = (V, E)$ with vertex set $V = \{v_1, v_2, ..., v_n\}$. Then, with $v_{n+1} = v_1$:*

$$\lambda(G) = \min\{\lambda(v_i, v_{i+1}) : i = 1, .., n\}. \tag{5.1}$$

Proof. Consider the graph $G = (V, E)$, with vertices u and v such that $\lambda(G) = \lambda(u, v)$ with minimal (u, v) edge-cut X. Let S denote all vertices $w \in V$ for which a path from u to w exists and contains no edges in X. Similarly, let $T = V - S$. (S, T) is a cut of G with $u \in S$ and $v \in T$. For all $s \in S$ and $t \in T$, X is the minimal (s, t) edge-cut in G, that is $\lambda(G) = |X| = \lambda(s, t)$. Therefore, labelling each vertex, v_i for $i \in \{1, 2, ..., n\}$ with $v_{n+1} = v_1$, there must exist an i such that $v_i \in S$ and $v_{i+1} \in T$ with $\lambda(G) = \lambda(v_i, v_{i+1})$. □

Using theorem 5.3.1, an algorithm for computing edge connectivity follows,

Step 1 Given a connected graph $G = (V, E)$, form a directed graph $D = (V_D, E_D)$ with $V_D = V$ and $E_D = \{(u, v), (v, u) : (u, v) \in E\}$. Set $n = |V|$.

Step 2 Form a flow network $N = (D, c, s, t)$ with $c(e) = 1$, for all $e \in D$. Set $s = 1$, $t = n$.

Step 3 Set $\lambda =$ MaxFlow(N, c, s, t). Then, for each $s \in \{1, ..., n-1\}$, set $t = s + 1$ and $\lambda = \min(\lambda,$ MaxFlow$(N, c, s, t))$.

Step 4 The edge connectivity of G is then equal to λ.

In pseudo code:

Algorithm 5.3.1 Edge Connectivity - Input: Connected graph $G = (V, E)$. Output: Edge connectivity ($\lambda(G)$). Prerequisite: Algorithm to calculate maximal flow (MaxFlow(N,c,s,t)).

```
1: procedure LAMBDA(G)
2:     Convert G into directed graph D by replacing each e ∈ E with two directed edges.
3:     Convert D into a network N with c(e) = 1 for all e ∈ E_D.
4:     s ← 1
5:     t ← n
6:     λ ← MaxFlow(N, c, s, t)
7:     for s ← 1 to n − 1 do
8:         t ← s + 1
9:         λ ← min(λ , MaxFlow(N, c, s, t))
       return λ
```

Python implementation: See Appendix A.7.

The edge connectivity of an integral weighted graph $G = (V, E)$, with weight function $w(e)$ can be calculated by replacing each edge $e \in E$ with $w(e)$ edges; the resulting multigraph can then have its edge connectivity calculated by algorithm 5.3.1.

Theorem 5.3.2. *Given a connected graph $G = (V, E)$, with $s, t \in V$. Using the Ford-Fulkerson Labelling Algorithm, $\lambda(s, t)$ can be calculated with time complexity $\mathcal{O}(|E| \min\{deg(s), deg(t)\})$.*

Proof. The graph G can be transformed into the desired zero-one flow network with time complexity $\mathcal{O}(|E|)$. Within a zero-one flow network, the maximal value of flow f between between s and t is less than or equal to the minimal degree of s and t (in G). Hence, using the Ford-Fulkerson Labelling Algorithm to calculate the maximal value of flow (equal to $\lambda(s, t)$), can be done with time complexity $\mathcal{O}(|E|f) = \mathcal{O}(|E| \min\{\deg(s), \deg(t)\})$ (by theorem 4.2.2). □

Theorem 5.3.3. *Given a connected graph $G = (V, E)$ with maximal degree $\Delta(G)$. The edge connectivity $\lambda(G)$, can be calculated using the Ford-Fulkerson Labelling Algorithm with time complexity $\mathcal{O}(|V||E|\Delta(G))$.*

Proof. The graph G can be transformed into the desired zero-one flow network with time complexity $\mathcal{O}(|E|)$. Theorem 5.3.1 dictates that to calculate $\lambda(G)$, a calculation of $\lambda(s, t)$ must be performed $|V|$ times. The time complexity of the calculation of $\lambda(s, t)$ is at most $\mathcal{O}(|E|\Delta(G))$ hence, calculating $\lambda(G)$ has time complexity $\mathcal{O}(|V||E|\Delta(G))$. □

Note: As presented by Jungnickel 2013 [25, p. 260]; for alternative maximal flow algorithms the calculation of $\lambda(s, t)$ can be shown to run in time complexity $\mathcal{O}(|V|^{2/3}|E|)$ and the calculation of $\lambda(G)$ can be shown to run with time complexity $\mathcal{O}(|V||E|)$.

5.4 Computing Vertex Connectivity

In a similar way to edge connectivity; vertex connectivity is computed using a flow network with a specific structure. Once the flow network is in this structure, the Max-Flow Min-Cut Theorem can be applied to obtain the specific vertex connectivity, which can then be related to the vertex connectivity to the entire graph. This structure is more complex and is built around the idea of *internally (vertex) disjoint paths.*

Given an undirected graph $G = (V, E)$, with fixed $s, t \in V$, a set $K = \{p_1, p_2, ..., p_k\}$ of paths $p_i = (v_0, e_1, ..., e_n, v_n)$, with $v_0 = s$ and $v_n = t$, $s \neq t$. K is called a set of internally (vertex) disjoint paths, if for any two paths in K, they have no vertices in common, other than s and t. The maximal cardinallity of this set K will prove to be intrinsic to finding the vertex connectivity and in fact will be shown to be equal to the specific vertex connectivity of s and t.

One method for obtaining the maximal number internally disjoint paths inside of a graph is achieved by transforming the given graph into a flow network as presented in Ibaraki and Nagamochi 2008 [23, p 34-35]. Take a connected graph G, then form the zero-one flow network N, as introduced for edge connectivity; by replacing every undirected edge in G with two oppositely directed edges with capacity equal to one. Taking the maximal set of vertex disjoint paths K as a set of augmenting paths in N (with strictly forward edges) may not be a full set of augmenting paths which realize the maximal flow. Hence, the Max-Flow Min-Cut Theorem cannot be applied to find the cardinality of K in this case. The problem here is that each vertex in N may have more than one flow unit flowing through it in a maximal flow distribution. Therefore, if the flow through a vertex can be limited to one flow unit, then the maximal set of vertex disjoint paths will correspond to a set of augmenting paths which realize a maximal distribution. Then as each path increases flow by exactly one flow unit, the maximal flow will equal the cardinality

of K. Limiting the flow through a vertex in N can be achieved by splitting each vertex v in N into two vertices v' and v'', joined by a single directed edge from v' to v'' with $c(v', v'') = 1$. This edge is called the *limiting directed edge*. Each directed edge (u, v) in N is then replaced with the edge (u'', v') with $c(u'', v') = 1$. The resulting flow network has no edges of the form (v', v') or (v'', v'') hence, N^* is *bipartite*, meaning the set of vertices can be partitioned into two sets V' and V'', such that all edges have exactly one endpoint in each set. Formally (as in Nagamochi and Ibaraki 2008 [23, p 34-35]), given the zero-one flow network $N = (V, E, c, s, t)$ then, $N^* = (V' \cup V'', E' \cup E'', c, s, t)$ with $c(e) = 1$ for all e in N^* where,

$$V' = \{v' : v \in V\}, \tag{5.2}$$
$$V'' = \{v'' : v \in V\}, \tag{5.3}$$
$$E' = \{(u'', v') : (u, v) \in E\}, \tag{5.4}$$
$$E'' = \{(v', v'') : v \in V\}. \tag{5.5}$$

Then every path from s to t in G will have a corresponding path from s'' to t' in N^*. Similarly an internally disjoint path in G will therefore have a corresponding internally disjoint path in N^*. Then as a result of the limiting directed edge between v' and v''; all internally disjoint path from s'' to t' in N^* must be *edge disjoint* (that is, no two paths contain the same edge). Hence, the minimal capacity cut of N^*, obtained by computing the maximal flow of the network N^* between s'' and t' corresponds to the the maximal number of internally disjoint paths between s and t in G.

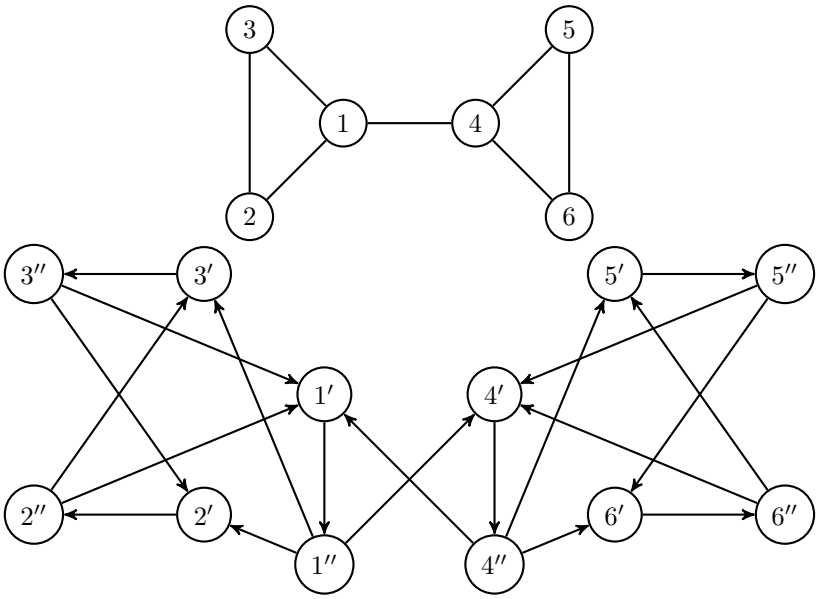


Figure 5.6: Undirected graph G (above) with corresponding N^* (below) with edge capacities equal to 1.

5.4.1 Menger's Theorem

For a connected graph $G = (V, E)$ the maximal number of internally disjoint paths between two non-adjacent vertices $s, t \in V$, is then related to the cardinality of a minimal vertex-cut using the following theorem discovered by Karl Menger [30] in 1927,

Theorem 5.4.1. (Menger's Theorem) *Let $G = (V, E)$ be graph, with non-adjacent vertices $s, t \in V$. The maximum number of internally disjoint paths from s to t is equal to the minimal cardinality of all vertex-cuts separating s and t.*

Proof. Let the $\kappa(s,t)$ be the minimal cardinality of a vertex-cut C separating s and t ($s, t \notin C$) and let $\alpha(s,t)$ denote cardinality of A, a set of internally disjoint paths from s to t. If $\kappa(s,t) = k$ and $\alpha(s,t) \geq k+1$, then at least $k+1$ s-t internally disjoint paths must pass through the vertex-cut C. As $\kappa(s,t) = k$, pigeonhole principle dictates, that at least one vertex in C must belong to at least two paths. However, this is a contradiction as the paths are no longer internally disjoint, hence $\alpha(s,t) \leq \kappa(s,t)$. Hence, if $\alpha(s,t) = \kappa(s,t)$ then $\alpha(s,t)$ is maximal.

This section of this proof is based on [42] aiming to prove by induction, that if $\kappa(s,t) \geq k$ and $\alpha(s,t)$ is maximal then $\alpha(s,t) \geq k$. First consider $\kappa(s,t) = 1$, there exists a vertex-cut C separating s and t. All paths from s to t must consist of at least one vertex from C, then as $|C| = 1$, there exists only one vertex disjoint path from s to t so, $\alpha(s,t) = 1$ (which is maximal). Then assume that $k \geq 1$ and if $\kappa(s,t) \geq k$ and $\alpha(s,t)$ is maximal then $\alpha(s,t) \geq k$. Now consider that $\kappa(s,t) \geq k+1$ and $\alpha(s,t)$ is maximal. The induction hypothesis states, there are k interally disjoint $s-t$ paths $P_1, P_2, ..P_k$. As $\kappa(s,t) \geq k+1$, the subsequent vertices to v in P_i (of which there are k) are not a vertex-cut separating s and t hence, there is an s-t path P whose initial edge is not in any P_i. Let x be the first vertex after s in P that belongs to some P_i. The choice of $P_1, .., P_k$ and x are such that the size of the path from x to t is minimized within in the induced subgraph $G \setminus \{s\}$. Then the subpath of P from s to x is P_{k+1}. If $x = t$ then $P_1, ..., P_{k+1}$ is the desired $k+1$ internally disjoint paths. Otherwise $x \neq t$, consider the induced subgraph $G \setminus \{x\}$. By the induction hypothesis, there are k internally disjoint s-t paths $Q_1, ..., Q_k$ in $G \setminus \{x\}$. Assume that these paths have been chosen so that a minimum number of edges not in any of the paths P_i. Let H be the graph consisting of the edges and vertices of the paths $Q_1, ..., Q_k$ and x (along with all edges incident from x to vertices in $Q_1, ..., Q_k$). Choose some P_j ($1 \leq j \leq k+1$), whose initial edge is not in H. Then let y be the first vertex in P_j after s which is in H. If $y = t$, then $Q_1, ..., Q_k, P_j$ are $k+1$ internally disjoint paths from s to t. Otherwise, $y \neq t$, then if $y = x$, let R be the shortest x-t path in $G \setminus \{s\}$. Let z be the first vertex in R that is in some Q_i. Then the size of the path in $G \setminus \{s\}$ from z to t is less than the distance from x to t. This contradicts the choice of $P_1, ..., P_k + 1$ to minimize the size of path from x to t. Hence, $y \neq x$. If y is on some Q_i, then the s-y subpath in Q_i has at least one edge not in any $P_1, ..., P_{k+1}$ otherwise two paths from $P_1, ..., P_{k+1}$ intersect at a vertex other than s,t or x. If the s-y path in Q_i is replaced by the s-y path in P_j, the choice of $Q_1, ...Q_k$, to minimize the number of edges not in P_i is then contradicted. Therefore, either $x = t$ or $y = t$. Therefore, $P_1, ..., P_k + 1$ or $Q_1, ..., Q_k, P_j$ are $k+1$ internally disjoint paths. As $\kappa(s,t) = 1$ implies $\alpha = 1$ then if $\kappa(s,t) \geq k$ and $\alpha(s,t)$ maximal implies $\alpha \geq k$ then $\kappa(s,t) \geq k+1$ and $\alpha(s,t)$ maximal implies $\alpha(s,t) \geq k+1$ for all positive integers k, by induction.

As $\alpha(s,t) \leq \kappa(s,t)$, then if $\kappa(s,t) = k$ and $\alpha(s,t)$ is maximal this implies $\kappa(s,t) \geq \alpha(s,t) \geq k$ then, $\kappa(s,t) = \alpha(s,t) = k$. □

Therefore, by calculating the maximal flow in N^* for any non-adjacent s'' and t', this will be equal to the number of internally disjoint paths from s to t (non-adjacent) in G which Menger's Theorem dictates is equal to the specific vertex connectivity of the s and t.

Menger's Theorem does not directly link the specific vertex connectivity and the vertex connectivity of the entire graph in the way theorem 5.3.1 did for edge connectivity, as it is unclear how the interaction between adjacent vertices contributes to the overall vertex connectivity. However, in 1932, Hassler Whitney [38, 39] showed the following relation between vertex connectivity and disjoint paths between all s and t,

Theorem 5.4.2. (Whitney's Theorem) *A graph $G = (V, E)$ is a k-connected graph if and only if there exists at least k internally vertex disjoint paths between all pairs of vertices $s, t \in V$.*

Proof. (Based on Jungnickel 2013 [25, p. 576 (7.1.7)]) Given a k-connected graph $G = (V, E)$. By Menger's Theorem (5.4.1), any two non-adjacent $s, t \in V$ are connected by k vertex disjoint paths. Now for adjacent $s, t \in V$, consider the subgraph obtained by removing the edge (s, t) from G, $H = (V, E \setminus (s, t))$. H is at least $(k-1)$-connected. By Menger's Theorem, s and t are connected by at least $k-1$ vertex disjoint paths in H, then including the edge (s, t), this gives at least k internally disjoint paths from s to t in G. Now for any given vertices $s, t \in V$, connected by at least k internally disjoint paths; G must be k-connected as at least k vertices must be removed to disconnect s and t. □

Corollary 5.4.1. *Let $G = (V, E)$ be a graph. If all pairs of vertices $s, t \in V$ are adjacent then $\kappa(G) = |V| - 1$, otherwise,*

$$\kappa(G) = \min\{\kappa(s, t) : \text{ non-adjacent } s, t \in V\}. \tag{5.6}$$

Although not required for calculating vertex connectivity, an important lemma which follows from of Whitney's Theorem relates the k-connectivity of a graph and the number of vertex disjoint cycles which have only two vertices in common.

Lemma 5.4.1. *Let $G = (V, E)$ be a k-connected graph. Then there are at least $\lfloor k/2 \rfloor$ cycles in G which have exactly two common vertices.*

Proof. As $G = (V, E)$ is k-connected, Whitney's Theorem implies that for any given $s, t \in V$ there exists at least k internally disjoint paths. Pairing together two internally disjoint paths then forms a cycle. Pairing all k internally disjoint paths between s and t gives $\lfloor k/2 \rfloor$ cycles in G. Each cycle has s and t in common and is otherwise vertex disjoint. □

Whitney's Theorem when paired with Menger's Theorem then gives the following steps for calculating the vertex connectivity of a given connected graph G,

Step 1 Given a connected graph $G = (V, E)$. Constuct the flow network $N^* = (V' \cup V'', E' \cup E'', c, s, t)$ as previously described, with $c(e) = 1$ for all $e \in E' \cup E''$. Set $s = 1$ and $t = 2$.

Step 2 Set $\kappa = |V| - 1$.

Step 3 If $s > \kappa$, **Stop**; the vertex connectivity of G is then equal to κ. Otherwise, Go-to **Step 4**.

Step 4 If s is adjacent to t in G, Go-to **Step 5**. Otherwise, set $\kappa = \min(\kappa, \text{MaxFlow}(N^*, c, s'', t'))$ then, Go-to **Step 5**.

Step 5 Set $t = t + 1$. If $t = |V| + 1$, then set $s = s + 1$ and $t = s + 1$. Go-to **Step 3**.

In pseudo code (by author in accordance with Kocay and Kreher 2013 [28, p. 182]):

Algorithm 5.4.1 Vertex Connectivity - Input: Connected graph $G = (V, E)$. Output: Vertex connectivity ($\kappa(G)$). Prerequisite: Algorithm to calculate maximal flow (MaxFlow(N,c,s,t)).

```
1:  procedure KAPPA(G)
2:      Convert G into the flow network N* as previously described.
3:      κ ← |V| − 1                                  ▷ Maximal vertex connectivity.
4:      s ← 0
5:      while s < κ do
6:          s ← s + 1
7:          for t ← s + 1 to n do
8:              if s ⋹ t then                        ▷ s not adjacent to t in G.
9:                  κ ← min(κ , MaxFlow(N*, c, s'', t'))
10:             if s > κ then return κ
        return κ
```

Python implementation: See Appendix A.8.

Theorem 5.4.3. *Given a connected graph $G = (V, E)$ with non-adjacent $s, t \in V$. The maximal number of vertex disjoint paths from s to t can be determined using the Ford-Fulkerson Labelling Algorithm with time complexity $\mathcal{O}(|E| \min\{deg(s), deg(t)\})$.*

Proof. The graph G can be transformed into the desired flow network N^* with complexity $\mathcal{O}(|E|)$. Within the flow network, the maximal value of flow f between between s and t is less than or equal to the minimal degree of s and t (in G). Hence, using the Ford-Fulkerson Labelling Algorithm to calculate the maximal value of flow (equal to the maximal number of disjoint paths between s and t), can be done with time complexity $\mathcal{O}(|E|f) = \mathcal{O}(|E| \min\{\deg(s), \deg(t)\})$ (by theorem 4.2.2). □

Theorem 5.4.4. *Given connected graph $G = (V, E)$ with maximal degree $\Delta(G)$. The vertex connectivity $\kappa(G)$, can be calculated using the Ford-Fulkerson Labelling Algorithm with time complexity $\mathcal{O}(|V|^2|E|\Delta(G))$.*

Proof. The graph G can be transformed into the desired zero-one flow network with complexity $\mathcal{O}(|E|)$. Corollary 5.4.1 dictates that to calculate $\kappa(G)$, a calculation of $\kappa(s, t)$ must be performed at most $|V|^2$ times. The time complexity of the calculation of $\kappa(s, t)$ is at most $\mathcal{O}(|E|\Delta(G))$ hence, calculating $\kappa(G)$ has time complexity $\mathcal{O}(|V|^2|E|\Delta(G))$. □

Note: As presented by Jungnickel 2013 [25, p. 212, 241], for alternative maximal flow algorithms, the calculation of $\kappa(s, t)$ can be shown to run in time complexity $\mathcal{O}(|V|^{1/2}|E|)$ and the calculation of $\kappa(G)$ can be shown to run with time complexity $\mathcal{O}(|V|^{1/2}|E|^2)$.

5.5 Whitney's Inequality

The vertex connectivity and edge connectivity of a graph share an intrinsic relation shown by Hassler Whitney in 1932 [38],

Theorem 5.5.1. (Whitney's Inequality) *Let $G = (V, E)$ be a non-trivial connected graph then,*

$$0 < \kappa(G) \leq \lambda(G) \leq \delta(G). \tag{5.7}$$

Proof. (Based on Kocay and Kreher 2013 [28, p. 120]) Let v_δ be a vertex such that $\deg(v_\delta) = \delta(G)$. Then by removing all edges incident from v_δ, v_δ will be isolated from the rest of the graph. Therefore, an edge-cut S_E consisting $\delta(G)$ edges incident with v_δ always exists. Hence $\lambda(G) \leq \delta(G)$.

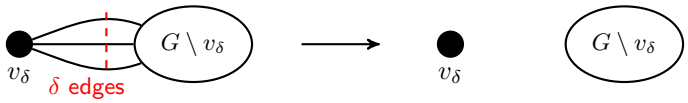


Figure 5.7: Isolating v_δ via an edge-cut of size δ.

Now let G be a graph with $\lambda(G) = 1$, then the removal of some edge $e = (u, v)$ disconnects G, but the removal of a single endpoint, u, will also disconnect G, implying $\kappa(G) \leq 1$. If G is a graph such that $\lambda(G) = k$ and a vertex-cut exists, then a minimal edge-cut of G consists of k edges. A vertex-cut with size k is formed by removing one of the endpoints of each of these edges. This implies that $\kappa(G)$ is at most equal to $\lambda(G)$. If G is such that no vertex-cut exists then $\kappa(s, t) = |V| - 1$, but if no vertex-cut exists, this implies that each vertex has an edge incident with every other vertex, or G is the isolated vertex, in both cases an edge-cut in G must be of cardinality at least $|V| - 1$. Hence $\kappa(G) \leq \lambda(G)$.

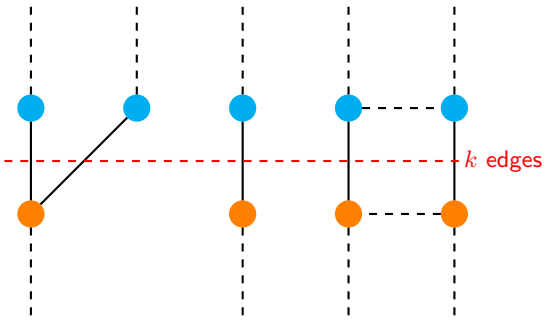


Figure 5.8: Minimal edge-cut with cardinality k (red) , vertex-cut with cardinality less than or equal to k (blue), vertex-cut with cardinality less than k (orange).

□

Whitney's Inequality shows a necessary condition for all connected graphs. Gary Chartrand and Frank Harary [6] proved in 1966 (published 1968), that any set of specified connectives which satisfy Whitney's Inequality are in fact realized by some connected graph. That is,

Theorem 5.5.2. *For all integers α, β, γ with $0 < \alpha \leq \beta \leq \gamma$, there exists a non-trivial connected graph G such that $\kappa(G) = \alpha, \lambda(G) = \beta, \delta(G) = \gamma$.*

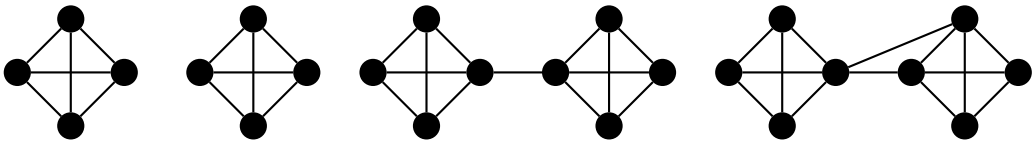
Proof. (Based on Chartrand and Harary 1968 [6]) A graph with specified connectivites satisfying Whitney's Inequality, can be constructed using the following steps,

Step 1 Take a graph G with two connected components G_1 and G_2, isomorphic to the complete graph on $\gamma + 1$ vertices $K_{\gamma+1}$.

Step 2 Then choose α distinct vertices $W_1 \subset V(G_1)$, and α distinct vertices $W_2 \subset V(G_2)$. Pair each elements of W_1 with an unpaired element in W_2, connect these paired elements via an edge.

Step 3 Connect $\beta - \alpha$ vertices from W_1 to G_2 via an edge.

Python implementation: See Appendix A.9.



Step 1 **Step 2** **Step 3**

Figure 5.9: Example construction - $\alpha = 1, \beta = 2, \gamma = 3$.

As $\delta(G_1) = \delta(G_2) = \gamma$ and $\alpha \leq \gamma$, not all vertices in G_1 can have an edge added, which implies that $\delta(G) = \gamma$. As $\kappa(G_1) = \kappa(G_2) = \lambda(G_1) = \lambda(G_1) = \gamma$ and G_1 is connect to G_2 via β edges from α vertices, removing these α vertices or removing these β edges will disconnect G, and as $0 < \alpha \leq \beta \leq \gamma$, this implies $\kappa(G) = \alpha$, $\lambda(G) = b$ and $\delta(G) = \gamma$.

□

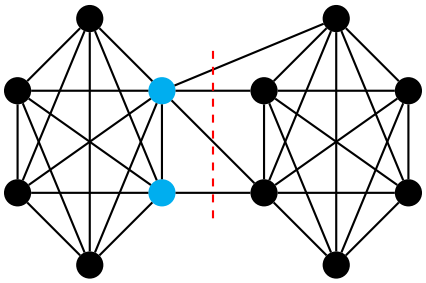


Figure 5.10: Graph G with $\kappa(G) = 2, \lambda(G) = 4$ and $\delta(G) = 5$. Minimum vertex cut (blue), minimum edge cut (red).

Theorem 5.5.3. *For a given simple graph, $G = (V, E)$ if $\delta(G) \geq \frac{|V|}{2}$ then, $\lambda(G) = \delta(G)$.*

Proof. (Based on Jungnickel 2013 [25, p. 589 (8.6.3)]) Consider the simple connected graph $G = (V, E)$, with cut (S, T) which induces a minimal edge cut C_E. Assume that $0 < x = |S| \leq |V|/2$. As G is simple induced subgraph $H = (S, E|S)$ has $|(E|S)| \leq x(x-1)/2$ (Maximal for H a complete graph) and the minimal number of edges incident from S in G is $x\delta(G)/2$. Hence, $|C_E| \geq (x\delta(G) - x(x-1))/2$. Then $f'(x) = -x + (\delta(G) + 1)/2$ and $f'(x) = 0$, implies $x = (\delta(G) + 1)/2$ is the only turning point of $f(x)$; if $\delta \geq |V|/2$ then, their are no turning points

for $x \in [1, |V|/2]$ then,

$$f(1) = \delta(G), \tag{5.8}$$

$$f'(1) = \frac{\delta(G) - 1}{2}, \tag{5.9}$$

$$f\left(\frac{|V|}{2}\right) = \frac{|V|}{2}\left(\delta(G) + 1 - \frac{|V|}{2}\right), \tag{5.10}$$

As $f'(1) > 0$, equations 5.8 and 5.10 are the local minima and local maxima respectively. Which implies, $\lambda(G) = |C_E| \geq f(x) \geq \delta(G)$ then, by Whitney's Inequality, $\lambda(G) = \delta(G)$. □

A graph G, is *k-maximally* connected if and only if $\kappa(G) = \delta(G) = k$. Similarly, G is called *k'-maximally-edge* connected if and only if $\lambda(G) = \delta(G) = k'$. Maximally connected graphs can be generated using the steps described in the proof of theorem 5.5.2. However, Gary Chartrand and Frank Harary [6] also gave a method to generate the smallest possible (by number of vertices) maximally connected simple graph.

Theorem 5.5.4. *The smallest simple graph G which is k-maximally connected is the complete graph K_{k+1} on $k+1$ vertices.*

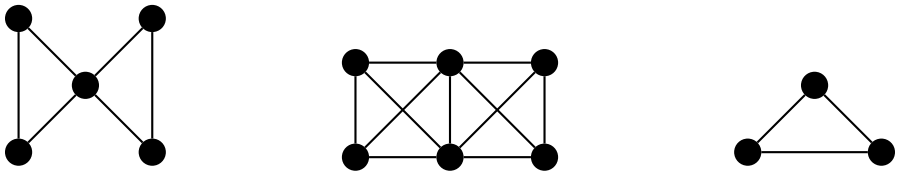
Proof. The complete graph K_{k+1} has $\kappa = k$ and $\delta = k$, then by Whitney's Inequality $\lambda = k$, hence K_{k+1} is k-maximally connected. As the graph is simple, if $|V|$ is smaller than $k+1$ then, $\delta < k$, hence K_{k+1} is the smallest k-maximally connected simple graph. □

Theorem 5.5.5. *The smallest k'-maximally-edge connected graph $G_{\alpha,k'}$ with $\kappa(G) = \alpha < \delta(G)$ and $\lambda(G) = \delta(G) = k'$ is constructed in the following steps,*

Step 1 *Take a graph G with three connected components H_1, H_2 and H_3, such that H_1 and H_2 are isomorphic to the complete graph on $k' - \alpha + 1$ vertices $K_{k'-\alpha+1}$ and H_3 is isomorphic to the complete graph on α vertices K_a.*

Step 2 *Add an undirected edge (u, v) for each pair $u \in V(H_1 \cup H_2)$ and $v \in V(H_3)$. The resulting graph, denoted $G_{\alpha,k'}$, has $\kappa(G_{\alpha,k'}) = \alpha$ and $\lambda(G_{\alpha,k'}) = \delta(G_{\alpha,k'}) = k'$.*

Proof. See Chartrand and Harary 1968 [6]. □



$G_{1,2}$ $G_{2,3}$ $G_{3,3}$

Figure 5.11: Examples of graphs $G_{\alpha,k'}$.

Chapter 6

The Edge Connectivity Augmentation Problem

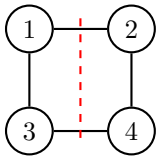
In many real world systems, actively increasing the edge connectivity, through the introduction of new (possibly parallel) edges, is a technique used to maintain the robustness of the system as connections become older and more likely to fail. The edge connectivity augmentation problem asks, what is the smallest set of edges, whose addition to a graph, increases its edge connectivity by $k \geq 1$. The first general algorithm to solve this problem was given by Toshimasa Watanabe and Akira Nakamura [31] in 1987. Their algorithm introduced a lot of new structures and ideas. Ideas which were taken and applied to a representation of a graph are known as a *cactus* by Dalit Naor, Dan Gusfield and Charles Martel [18] in 1997. This algorithm will be the focus of this chapter, in particular, using this algorithm to solve the edge connectivity augmentation problem when $k = 1$. Many of the tools required for this algorithm are not detailed in the original paper, with many being highly non-trivial and not well documented. This chapter will consolidate the tools required for this algorithm, presenting them alongside worked examples and giving exact detail, which is often omitted in literature regarding this problem.

6.1 The Crossing Property

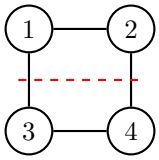
This section closely follows Naor and Vazirani 1991 [32, p. 274-275].

A minimal edge-cut splits the vertices of a graph into two disjoint sets A and $\bar{A} \coloneqq V \setminus A$. Therefore, the cut $(A, \bar{A})$ induces a minimal edge cut. A graph may have multiple minimal edge-cuts which split the graph into many differing sets of vertices. This motivates the question; is there any relationship between the distinct cuts $(A, \bar{A})$ and $(B, \bar{B})$, which both induce a minimal edge-cuts? Answering this question will reveal some of the underlying structure of minimal edge-cuts, which can then be exploited to compact the set of all minimal edge cuts into a linear graph representation (Cactus Representation) which will be shown later in this report. This structure stems from the *crossing property* of $(A, \bar{A})$ cuts in G. Mainly, two distinct cuts $(A, \bar{A})$ and $(B, \bar{B})$ are said to be *crossing cuts* if and only if the following hold true,

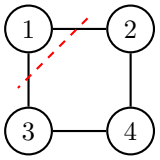
$$A \cap B \neq \emptyset, \tag{6.1}$$
$$\bar{A} \cap B \neq \emptyset, \tag{6.2}$$
$$A \cap \bar{B} \neq \emptyset, \tag{6.3}$$
$$\bar{A} \cap \bar{B} \neq \emptyset. \tag{6.4}$$



$(X, \bar{X}) = (\{1,3\}, \{2,4\})$



$(Y, \bar{Y}) = (\{1,2\}, \{3,4\})$



$(Z, \bar{Z}) = (\{1\}, \{2,3,4\})$

Figure 6.1: Cuts $(X, \bar{X})$, $(Y, \bar{Y})$ and $(Z, \bar{Z})$ on a graph G with $\lambda(G) = 2$.

That is, two cuts $(A, \bar{A})$ and $(B, \bar{B})$ are crossing cuts if and only if the sets A and $\bar{A}$ are not disjoint from the sets B and $\bar{B}$. Take for example figure 6.1, the cuts $(X, \bar{X})$ and $(Y, \bar{Y})$ are crossing cuts as $X \cap Y = \{1\}$, $\bar{X} \cap Y = \{2\}$, $X \cap \bar{Y} = \{3\}$ and $\bar{X} \cap \bar{Y} = \{4\}$. Similarly the cut $(Z, \bar{Z})$ is not a crossing cut with $(X, \bar{X})$ or $(Y, \bar{Y})$ as $Z \cap \bar{X} = Z \cap \bar{Y} = \emptyset$.

Let $d(A, B)$ denote the number of edges $(s, t) \in E$ (or sum of the weights of these edges for a weighted graph), for $s \in A$ and $t \in B$. Then the following lemma as given by Naor and Vazirani 1991 [32] applies,

Lemma 6.1.1. *Let $(A, \bar{A})$ and $(B, \bar{B})$ be crossing cuts which induce minimal edge-cuts in G. Then,*

$$\begin{aligned} d(A \cap B, \bar{A} \cap B) &= d(\bar{A} \cap B, \bar{A} \cap \bar{B}), \\ &= d(\bar{A} \cap \bar{B}, A \cap \bar{B}), \qquad (6.5) \\ &= d(A \cap \bar{B}, A \cap B) = \lambda(G)/2, \\ d(A \cap B, \bar{A} \cap \bar{B}) &= d(A \cap \bar{B}, \bar{A} \cap B) = 0. \qquad (6.6) \end{aligned}$$

Proof. See Dinitz *et al.* 1976 [9]. □

Taking the crossing cuts $(X, \bar{X})$ and $(Y, \bar{Y})$ as in figure 6.1, lemma 6.1.1 can be verified,

$$\begin{aligned} d(\{1\}, \{2\}) &= d(\{2\}, \{4\}), \\ &= d(\{4\}, \{3\}), \qquad (6.7) \\ &= d(\{4\}, \{1\}) = 1, \\ d(\{1\}, \{4\}) &= d(\{3\}, \{2\}) = 0. \qquad (6.8) \end{aligned}$$

Lemma 6.1.1 then gives the following important corollary,

Corollary 6.1.1. *For a graph or weighted graph G with integral weights. If $\lambda(G)$ is odd, then there are no crossing cuts in G.*

Proof. As the number of edges or weight of any cut is integral, $d(A, B)$ must also be integral. Hence, for a crossing cuts to exist equation 6.5 must also be integral. As $\lambda(G)$ is odd $\lambda(G)/2$ is not integral. This is a contradiction hence, no crossing cuts in G. □

Lemma 6.1.1 then gives the following lemma (as in Noar and Vazirani 1991 [32]),

Lemma 6.1.2. *If there are crossing cuts which induce minimal edge-cuts in G, then the set of vertices V can be partitioned into $k \geq 3$ disjoint subsets $V_1, ..., V_k$, that can be ordered on a cycle, this is called a circular partition if it has the following properties:*

1. *The number of edges between any two adjacent sets V_i and $V_{i+1(mod\ k)}$ on the cycle is exactly $\lambda(G)/2$.*
2. *The number of edges between any two non-adjacent sets of vertices on the cycle is zero.*
3. *For any $1 \leq a < b \leq k$, if $A = \cup_{i=a}^{b-1} V_i$, then the cut $(A, \bar{A})$ induces a minimal edge-cut, and if $(B, \bar{B})$ induces a minimal edge-cut and is is not of the same form as A, then $B \subset V_i$, for some i.*

Proof. See Dinitz *et al.* 1976 [9]. □

From lemma 6.1.2 a bound on the number of minimal edge-cuts can be obtained (as in Dinitz *et al.* 1976 [9]),

Theorem 6.1.1. *For a graph $G = (V, E)$, the number of minimal edge-cuts of G is at most equal to $\binom{|V|}{2}$.*

Proof. See Dinitz *et al.* 1976 [9]. □

6.2 Chain Representation

To form a compact representation of the minimal edge-cuts, first all minimal edge-cuts should be obtained. Mainly, let the cut $(A, \bar{A})$ induce a minimal edge-cut in a graph $G = (V, E)$, then let S_i, $1 \leq i < |V|$, denote set of $(A, \bar{A})$ cuts, such that $\{1, ..., i\} \subseteq A$ and $i + 1 \in \bar{A}$, that is,

$$S_i = \{(A, \bar{A}) | \{1, ..., i\} \subseteq A, \{i + 1\} \subseteq \bar{A}\}. \tag{6.9}$$

It is desirable that each S_i has no crossing cuts. If $\lambda(G)$ is odd then by corollary 6.1.1 there are no crossing cuts within S_i, but if $\lambda(G)$ is even then the following lemma is required (as in Noar and Vazirani 1991 [32]),

Lemma 6.2.1. *If the vertices of a graph $G = (V, E)$ are labelled such that for any $i \in \{1, ..., |V|\}$, the vertex v_{i+1} is adjacent to some v_j, $j \in \{1, ..., i\}$, then all cuts in S_i of G are non-crossing.*

Proof. See Noar and Vazirani 1991 [32]. □

Using lemma 6.2.1 to ensure each S_i has no crossing cuts then, $\cup_{i=1}^{|V|-1} S_i$ is the set of all cuts $(A, \bar{A})$ which induce a minimal edge-cut of G. The set of S_i however repeats a lot of elements, in fact if one takes any two cuts $(X, \bar{X}), (Y, \bar{Y}) \in S_i$ with $|X| \leq |Y|$ then $X \subseteq Y$ and $|X| = |Y|$ if and only if $X = Y$. As S_i is such that all cuts are non-crossing, then S_i be condensed into its *chain representation*, denoted C_i, a partition of V into $V_1, ...V_k$, $k = |S_i| + 1$ disjoint sets of vertices. C_i is then formed by taking the ordered set cuts $s_1, ..., s_{k-1} \in S_i$, such that $A_{s_1} \in s_1$ has minimal carnality of all $A \in S_i$ and $A_{s_i} \in s_i$ has minimal cardinality of all $A \in S_i \setminus \{s_1, ..., s_{i-1}\}$ for $1 < i < k$. Set $V_1 = A_{s_1}$, then $V_i = \{v : v \in A_{s_i}, v \notin V_{i-1} \cup ... \cup V_1\}$, for $1 < i \leq |S_i|$. Finally let $V_k = \{v : v \in V, v \notin V_{|S_i|} \cup V_{|S_i|-1} \cup ... \cup V_1\}$. As a result of this construction C_i has $\{1, ..., i\} \subseteq V_1$ and $i + 1 \in V_k$. Each C_i can then be represented as a path with with an edge

from V_j to V_{j+1} for $j \in \{1, .., |C_i| - 1\}$. Then each minimal edge-cut in this path corresponds to a minimal edge-cut in G.

As an example, take the graph in figure 6.2[1]; its set of S_i and corresponding chain representations are shown in figure 6.3.

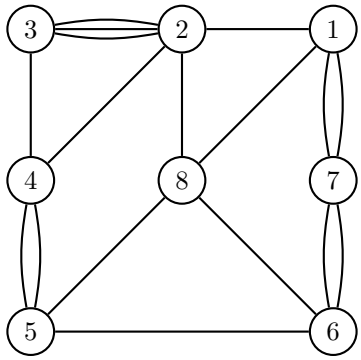


Figure 6.2: Graph $G = (V, E)$ with $\lambda(G) = 4$.

S_i	A
S_1	$\{1\}$ $\{1, 7\}$ $\{1, 7, 6\}$ $\{1, 7, 6, 8\}$ $\{1, 7, 6, 8, 5\}$ $\{1, 7, 6, 8, 5, 4\}$
S_2	$\{1, 2, 4, 5, 6, 7, 8\}$
S_3	$\{1, 2, 3, 6, 7, 8\}$ $\{1, 2, 3, 6, 7, 8, 5\}$
S_4	$\{1, 2, 3, 4, 6, 7, 8\}$
S_5	$\{1, 2, 3, 4, 5, 8\}$ $\{1, 2, 3, 4, 5, 8, 7\}$
S_6	$\{1, 2, 3, 4, 5, 6, 8\}$
S_7	$\{1, 2, 3, 4, 5, 6, 7\}$

C_i	Chain
C_1	$\{1\}, \{7\}, \{6\}, \{8\}, \{5\}, \{4\}, \{2, 3\}$
C_2	$\{1, 2, 4, 5, 6, 7, 8\}, \{3\}$
C_3	$\{1, 2, 3, 6, 7, 8\}, \{5\}, \{4\}$
C_4	$\{1, 2, 3, 4, 6, 7, 8\}, \{5\}$
C_5	$\{1, 2, 3, 4, 5, 8\}, \{7\}, \{6\}$
C_6	$\{1, 2, 3, 4, 5, 6, 8\}, \{7\}$
C_7	$\{1, 2, 3, 4, 5, 6, 7\}, \{8\}$

Figure 6.3: All S_i and corresponding C_i of the graph G as in figure 6.2.

[1]Figure 6.2 is equivalent to the graph given in Noar and Vazirani 1991 [32, Figure 1] however, this report includes implicit steps in forming the chain representation, cactus representation and augmenting the edge connectivity of this graph, which is omitted from Noar and Vazirani 1991 [32] and Noar *et al.* 1997 [18].

6.3 Cactus Representation

Each chain C_i of G can be merged into a single graph $\mathcal{H}(G)$ known as a *cactus*; established by Yefim Dinitz (previously Efim Dinitz), Alexander Karzanov and Michael Lomonosov [9] in their 1976 paper in Russian, later translated into English by Alexander Karzanov. This paper does not detail how to construct a cactus, merely defining the structure and noting some key properties. Although Dinitz *et al.* 1976 [9] has been attributed [15, 18, 32] with initially introduction of the cactus, as defined in this section. These graphs have been previously studied under the name of *Husimi trees* by Frank Harary and George Uhlenbeck [20] in 1953. The structure was named after Kodi Husimi; who did some particularly interesting work on these graphs (See Husimi 1950 [22]). Harary and Uhlenbeck 1953 [20] reserved the name "cactus" for a specific class of Husimi trees. Since then however, the term "Husimi tree" has come refer to graph with completely different structure, hence to avoid ambiguity the term cactus has been adopted as the general term.

Merging the chains C_i into a cactus will be discussed in the next section, but first this report will discuss the properties of a cactus and a specific set of cacti which will be required to solve the edge connectivity augmentation problem.

As presented in Naor and Vazirani 1991 [32] a cactus $\mathcal{H}(G)$, of a multigraph G, is a connected weighted graph, such that each vertex in G maps to exactly one node in $\mathcal{H}(G)$, with each node in $\mathcal{H}(G)$ corresponding to a subset (possibly empty) of vertices in G. Also, for each cut $(X, \bar{X})$ in $\mathcal{H}(G)$, the set of vertices A of G which map to the nodes $u \in X$, defines a cut $(A, \bar{A})$ on G. In particular, each $(X, \bar{X})$ which induces a minimal edge cut in the cactus $\mathcal{H}(G)$, corresponds to a cut $(A, \bar{A})$ which induces a minimal edge-cut in G. Hence, $\lambda(\mathcal{H}(G)) = \lambda(G)$ and $\mathcal{H}(G)$ compactly representing all minimal edge-cuts of G. An edge in the cactus $\mathcal{H}(G)$ is called a *cycle-edge* if the edge is contained within at least one cycle in $\mathcal{H}(G)$, otherwise it is called a *tree-edge*. Each cycle-edge must have weight $\lambda(G)/2$ and each tree-edge must have weight $\lambda(G)$. Hence, each minimal edge-cut is consists either of a single tree-edge or a pair cycle-edges from the same cycle. A node u is called a *leaf* if u has a single incident tree-edge or u has exactly two incident cycle-edges. The following theorem by Dinitz *et al.* 1976 [9] ensures that a cactus $\mathcal{H}(G)$ with these properties can always be found,

Theorem 6.3.1. *Every multigraph $G = (V, E)$ has a cactus $\mathcal{H}(G)$ on at most $4|V|$ vertices.*

Proof. See Dinitz *et al.* 1976 [9]. □

A cactus $\mathcal{H}(G)$ is not necessarily unique. To solve the edge augmentation problem, a more specified unique cactus structure is required.

A *2-way cut node* is any node whose removal disconnects the cactus into 2 connected components. Similarly a *3-way cut node* is any node whose removal disconnects the cactus into 3 connected components. An *empty node* is any node of a cactus $\mathcal{H}(G)$ which corresponds to an empty set of vertices. A trivial node in $\mathcal{H}(G)$ is any empty node u of a cycle Y, with degree greater than 2, whose removal of edges in Y causes the node u to be reachable only by empty nodes in $\mathcal{H}(G)$. Hence, as defined by Lisa Fleischer [15] a *canonical cactus* of graph is a cactus with no trivial nodes and no 3-way cut empty nodes.

Theorem 6.3.2. *Every multigraph $G = (V, E)$ has a unique canonical cactus.*

Proof. See Kameda and Nagamochi 1994 [26]. □

For the remainder of this report the term cactus will refer to a canonical cactus, denoted $\mathcal{H}(G)$. As shown by Tiko Kameda and Hiroshi Nagamochi [26], the (canonical) cactus of the graph then has the property that every circular partition of G corresponds to a cycle in $\mathcal{H}(G)$, and every cycle of $\mathcal{H}(G)$ represents a circular partition of G.

6.3.1 Constructing Cactus Representations

The first algorithm to constuct the cactus of a graph was detailed in 1986 by Alexander Karzanov and Evgeniy Timofeev [27], then refined by Dalit Naor and Vijay Vazirani [32] in 1991. The algorithm for cactus construction in Naor and Vijay Vazirani 1991 [32] is noted to be incorrect by Lisa Fleischer [15], the exact error pertains to the detail of **Step 4** (g) below. This detail is essential for introducing cycles into the cactus when merging chains together. A correct algorithm by Tiko Kameda and Hiroshi Nagamochi [26] as given by Fleischer 1999 [15] is presented as follows,

Step 1 For a given connected multigraph $G = (V, E)$ enumerate the vertices of G such that there is an edge connecting vertex $i + 1$ and some vertex in the set $\{1, ..., i\}$.

Step 2 For each positive integer $i < n$, compute the specific edge connectivity $\kappa(s, t)$ for all $s \in \{1, ..., i\}$ and $t = i+1$. If $\kappa(s, t) = \lambda(G)$ for some $s \in \{1, ..., i\}$, find the chain representation C_i of S_i. Otherwise, $C_i = \emptyset$.

Step 3 Set $\mathcal{H}(G_n)$ to the isolated node with label $\{1, ..., n\}$

Step 4 Set $t = n-1$. Merge the chains in reverse order from C_t to C_1 into a single cactus structure $\mathcal{H}(G)$ in the following steps:

(a) Let u be the vertex in $\mathcal{H}(G_{t+1})$ labelled $\{1, ..., t + 1\}$.

(b) $t = t - 1$.

(c) If $C_{t+1} = V_1, ..., V_k$, then remove u and all incident edges and introduce k new nodes $u_1, ..., u_k$ where $u_i = V_i \in C_{t+1}$, with undirected edges (u_j, u_{j+1}) for $1 \leq j < k$.

(d) For all tree-edges and cycle-edges (u, w) in $\mathcal{H}(G_{t+1})$, then let $W \neq \emptyset$ be the set of vertices in w, or if w is an empty node, the vertices reachable from w by some path of edges disjoint from a cycle containing (u, w). Find the subset V_j such that $W \subset V_j$ and connect w to u_j.

(e) Let $U = \{1, ..., t + 1\}$, set each $u_j = V_j \cap U$. All other mappings remain unchanged.

(f) For each empty node with degree ≤ 2. Contract an edge as in figure 6.4 or figure 6.5.

(g) For each empty node with degree equal to 3. If all three edges are tree-edges, then add a 3-cycle using the adjacent nodes, then remove the empty node (and all incident edges) as in figure 6.6. If exactly one edge of the empty node is a tree-edge, then contract the tree-edge as in figure 6.7. If all three edges of the empty node are cycle-edges then remove the empty node and all edges incident.

Theorem 6.3.3. *The canonical cactus of a graph can be constructed from the chain representation of a graph $G = (V, E)$ with time complexity $\mathcal{O}(|V|^2)$.*

Proof. See Fleischer 1999 [15]. □

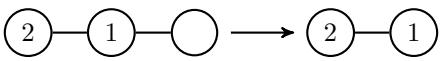


Figure 6.4: Contracting node 1 into the empty node.

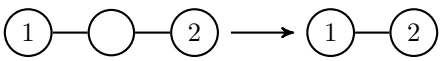


Figure 6.5: Contracting node 1 into the empty node.

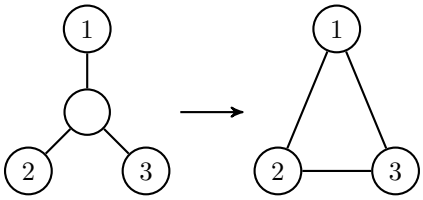


Figure 6.6: Remove the empty node then form the 3-cycle with nodes $1, 2, 3$.

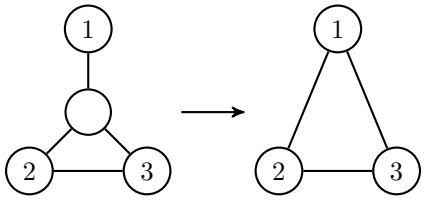


Figure 6.7: Contracting node 1 into the empty node.

This report will now present a step-by-step construction of the cactus for the graph in figure 6.2. Note that edge weights are omitted from figures; each tree-edge has weight $\lambda(G) = 4$, and each cycle-edge has weight $\lambda(G)/2 = 2$. The vertices of this graph have already been labelled in accordance with **Step 1**. **Step 2**: The chain representation of this graph has already been obtained as in figure 6.3. **Step 3**: $\mathcal{H}(G_8)$ as in figure 6.8. **Step 4**: Figure 6.9 shows how the chain C_7 is introduced into the cactus $\mathcal{H}(G_7)$.

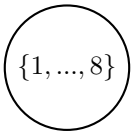


Figure 6.8: $\mathcal{H}(G_8)$

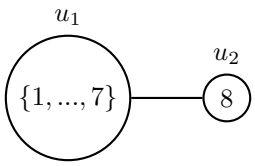


Figure 6.9: Merging C_7 into $\mathcal{H}(G_8)$.

Merging C_6 into $\mathcal{H}(G_7)$; **Step 4** (a): Set the node $\{1, ..., 7\} = u$. **Step 4** (c): Remove the edge $(8, u)$, then split u into 2 nodes, $u_1 = \{1, ..., 6, 8\}$, $u_2 = 7$ and join these nodes via an edge. **Step 4** (d): Join an edge from node 8 to u_1 (as $8 \in u_1$). **Step 4** (e): Set $u_1 = \{1, ..., 6\}$. u_2 is unaffected. **Step 4** (f) and **Step 4** (g) do not apply. This then gives $\mathcal{H}(G_6)$ as in figure 6.10.

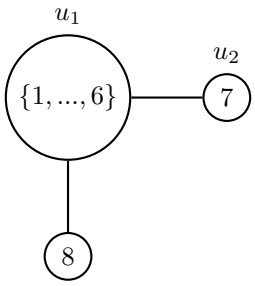


Figure 6.10: Merging C_6 into $\mathcal{H}(G_7)$.

Merging C_5 into $\mathcal{H}(G_6)$; **Step 4** (a): Set the node $\{1, ..., 6\} = u$. **Step 4** (c): Remove the edges $(7, u)$ and $(8, u)$, then split u into 3 nodes, $u_1 = \{1, ..., 5, 8\}$, $u_2 = 7$, $u_3 = 6$ then join the nodes u_1, u_2 via an edge, and u_2, u_3 via an edge. **Step 4** (d): Join an edge from node 8 to u_1 (as $8 \in u_1$) and an edge from node 7 to u_2 (as $7 \in u_2$). **Step 4** (e): Set $u_1 = \{1, .., 5\}$, $u_2 = \emptyset$. u_3 is unaffected. **Step 4** (f) does not apply. **Step 4** (g): Remove the empty node u_2 then form a 3-cycle with nodes $u_1, 6, 7$. This then gives $\mathcal{H}(G_5)$ as in figure 6.11.

Merging C_4 into $\mathcal{H}(G_5)$, as in figure 6.12, is similar to merging C_6 into $\mathcal{H}(G_7)$. Merging C_3

into $\mathcal{H}(G_4)$, as in figure 6.13, is similar to merging C_5 into $\mathcal{H}(G_6)$. Merging C_2 into $\mathcal{H}(G_3)$, as in figure 6.14, is then similar to merging C_6 into $\mathcal{H}(G_7)$.

Merging C_1 into $\mathcal{H}(G_2)$; **Step 4** (a) to **Step 4** (e) gives the graph as in figure 6.15. **Step 4** (f): Contract nodes 1 and 2 into empty node u_2 and u_6 respectively. **Step 4** (g): Remove the empty node u_4 then form the 3-cycle with nodes $8, u_3, u_5$. This then gives the cactus $\mathcal{H}(G)$ as in figure 6.16.

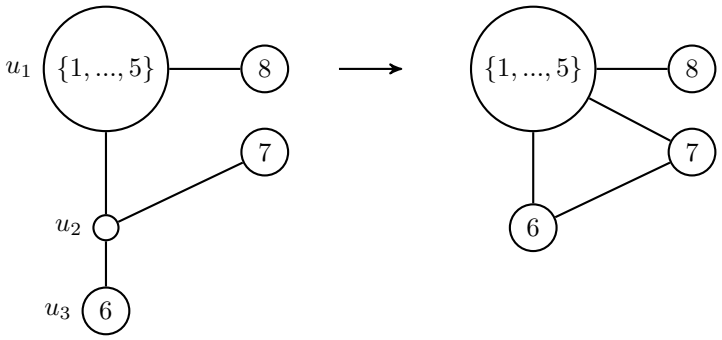


Figure 6.11: Merging C_5 into $\mathcal{H}(G_6)$. Removing the empty node u_2 and form a 3-cycle with nodes $u_1, 6, 7$.

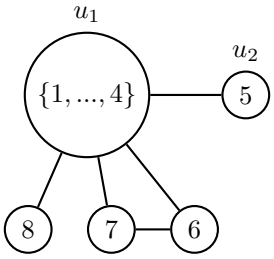


Figure 6.12: Merging C_4 into $\mathcal{H}(G_5)$.

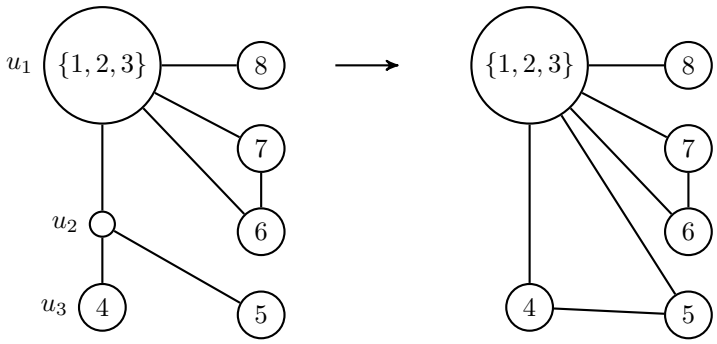


Figure 6.13: Merging C_3 into $\mathcal{H}(G_4)$. Removing the empty node u_2 and form a 3-cycle with nodes $u_1, 4, 5$.

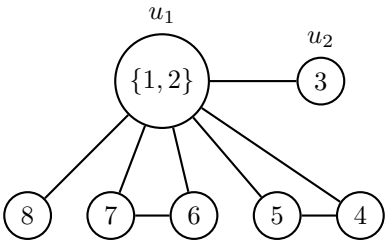


Figure 6.14: Merging C_2 into $\mathcal{H}(G_3)$.

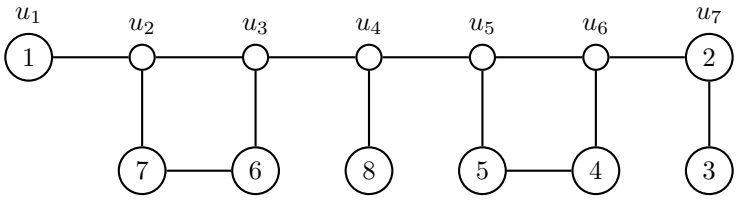


Figure 6.15: Merging C_1 into $\mathcal{H}(G_2)$ prior to **Step 4** (f) and **Step 4** (g).

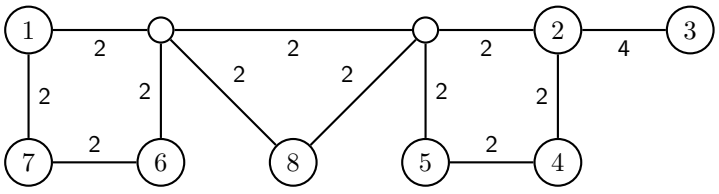


Figure 6.16: Canonical cactus $\mathcal{H}(G)$.

6.4 Eulerian Tours

Recall the problem of the Seven Bridges of Königsberg, from the beginning of this report. Does their exist a walk around the city, such that each bridge is crossed exactly one and if such a walk exists, is it possible to end the walk in the quarter where the walk stared. Solving this problem is the final tool required to solve the edge augmentation problem. This will provide an algorithm to find such a walk through a cactus like structure, which in-tern allows a systematic traversal of a cactus.

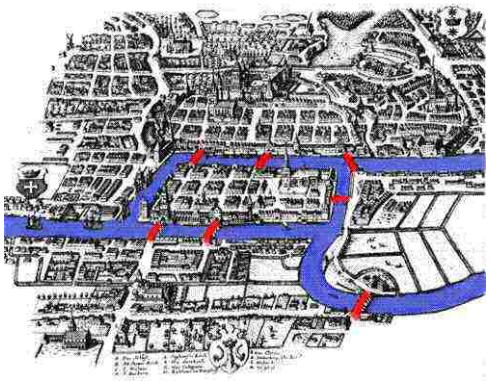


Figure 6.17: A plate of the 7 bridges of Königsberg stretching across the Pregel River. Source: [43].

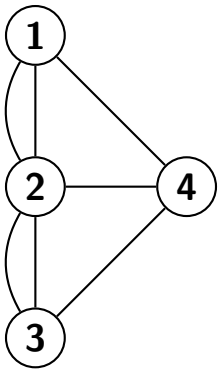


Figure 6.18: Graph of Königsberg.

Euler's original solution [13] to this problem transformed the city into a graph as in figure 6.18, with each quarter of the city being represented by a vertex and each bridge as an edge between the vertices. Euler noticed that in order for such a walk to exist, that if the starting vertex has even degree, then the walk must end on this vertex. Similarly, if the stating starting vertex has odd degree, then the walk must end on a different vertex of odd degree. This can be shown by considering the operation of entering and leaving the a given vertex, which will be assigned the values 0 and 1 respectively. Then a sequence of these operations on the starting vertex must always start with a 1 (always leave the vertex first) and alternate until all edges have been crossed. If the number of edges attached the starting vertex is even, then this sequence will terminate on a 0, hence the walk ends on this vertex. If the number of edges is odd, then this sequence ends with a 1, hence the walk has left the starting vertex and cannot return as all edges have been used. Furthermore, considering a vertex, not the starting vertex, its sequence must begin with a 0 (enter the vertex first) and alternate until all of its edges have been crossed. If the number of edges incident with this vertex is even, then this sequence will end on a 1, hence the walk has left the vertex and cannot return as all edges have been used. But if the number of edges is odd, then the sequence will end on a 0, hence the walk will end on this vertex. From these two observations Euler then stated without proof the following,

Theorem 6.4.1. *Given a connected graph $G = (V, E)$. There exists a trail containing every edge $e \in E$, if and only if, there exists exactly two or zero vertices of odd degree.*

Proof. See Hierholzer 1873 [21]. □

In honor of Euler, any trail which contains every edge of a graph exactly once, is called an *Eulerian trail*. Furthermore, if the final and first vertex of such a path is the same then it is said to be an *Eulerian tour*.

Corollary 6.4.1. *A graph $G = (V, E)$ contains an Eulerian tour if and only if all vertices $v \in V$ have even degree. In which case G is called Eulerian.*

In the context the seven bridges problem, four vertices have odd degree, which contradicts theorem 6.4.1. Therefore, no Eulerian trail or Eulerian tour exists over the seven bridges of Königsberg.

If a graph is Eulerian, then an Eulerian tour must exist. Such a tour can be found using the following algorithm by Carl Hierholzer 1873 [21],

Step 1 For a connected graph $G = (V, E)$, with even degree for all vertices . Set $i = 0$, then choose an arbitrary vertex v_0. Construct a closed trail $T_0 = (e_1, ..., e_k)$, by first choosing any arbitrary edge e_1 incident from v_0 to some $v \in V$, then continue to choose arbitrary edges e_j incident from the endpoint of e_{j-1} different from any previous edge $e_1, ..., e_{j-1}$ until returning to v_0.

Step 2 Choose a vertex $u \in V$ in the trail T_i incident with some edge not in the trail. If no such u_i exists, **Stop** T_i is an Eulerian tour. Otherwise, construct a trail $T_i' = (e_1', ..., e_k')$ by choosing a arbitrary edge e_1' not in T_i incident from u to some v, then continue to choose arbitrary edges e_j' not in T_i incident from the endpoint of e_{j-1}', different from any previous edge $e_1', ..., e_{j-1}'$ until returining to u.

Step 3 Form the next closed trail T_{i+1} by finding two edges e_j, e_{j+1} in T_i such that u is the endpoint of e_j and is the incident vertex of e_{j+1}, then add the trail T_i' in-between e_j and e_{j+1}. If $u_j = v_0$ then simply append T_i' to the end of T_i to construct T_{i+1}. Set $i = i + 1$, then Go-to **Step 2**.

Theorem 6.4.2. *Given an Eulerian graph $G = (V, E)$, an Eulerian tour can be found with time complexity $\mathcal{O}(|E|)$.*

Proof. See Hierholzer 1873 [21]. □

The algorithm to solve the edge connectivity augmentation problem requires a method to visit each node of a cactus systematically. An optimal solution is decided once every node has been assessed. This is unlike a greedy algorithm which chooses the optimal solution at each stage rather than assessing the entire graph. If a graph is connected and Eulerian, then an Eulerian tour systematically visits each vertex as it must cross every edge. However, a cactus is not necessarily Eulerian as a node may have odd degree. A cactus can be made Eulerian by replacing each tree-edge by a pair of parallel edges. Hence, an Eulerian tour can be found in the resulting graph $\mathcal{H}^*(G)$ using the algorithm above. This Eulerian tour then gives an ordered set of edges and nodes in $\mathcal{H}^*(G)$ which can be used to systematically visit each node in $\mathcal{H}(G)$.

6.5 Edge Connectivity Augmentation Algorithm

The cactus representation gives insight into the weaknesses of a graph. In particular, Naor *et al.* 1997 [18] showed that the leafs of a cactus can be used to place a lower bound on the optimal number of edges required to increase the edge connectivity of a graph by one.

Lemma 6.5.1. *For a connected multigraph $G = (V, E)$, if the cactus $\mathcal{H}(G)$ has k leaves, then at least $\lceil k/2 \rceil$ edges must be added to increase the edge connectivity of G by one.*

Proof. See Naor *et al.* 1997 [18]. □

In fact this lower bound can always be realized by the following algorithm by Naor *et al.* 1997 [18], which solves the edge connectivity augmentation problem for $k = 1$,

Step 1 Construct the (canonical) cactus $\mathcal{H}(G)$.

Step 2 Form the Eulerian graph $\mathcal{H}^*(G)$ from $\mathcal{H}(G)$, by splitting each tree-edge into two parallel edges. Find an Eulerian tour in $\mathcal{H}^*(G)$, enumerating the leaves of $\mathcal{H}(G)$, $u_1, ..., u_k$ in the order which the corresponding vertices in $\mathcal{H}^*(G)$ are first encountered in the Eulerian tour.

Step 3 Form the pairs $\{(U_i, U_{i+\lceil k/2 \rceil}) \,|\, 1 \leq i \leq \lfloor k/2 \rfloor\}$, where U_i is the set of vertices from G that map to the leaf u_i of $\mathcal{H}(G)$.

Step 4 For each pair $(U_i, U_{i+\lceil k/2 \rceil})$, $1 \leq i \leq \lfloor k/2 \rfloor$, pick an arbitrary vertex from U_i and an arbitrary vertex in $U_{i+\lceil k/2 \rceil}$ and connect them by an edge. If k is odd, then connect a vertex in $U_{\lceil k/2 \rceil}$ to a vertex in an arbitrary leaf U_j, $j \neq \lceil k/2 \rceil$.

Theorem 6.5.1. *Given a connected graph $G = (V, E)$, the edge connectivity can be increased by one with time complexity $\mathcal{O}(|V||E|)$.*

Proof. See Naor *et al.* 1997 [18]. □

Due to the arbitrary choices in this algorithm, an optimal solution to the edge connectivity augmentation problem is not necessarily unique. Applying this algorithm multiple times will increase the edge connectivity indefinitely however, it is noted by Naor *et al.* 1997 [18] that optimally increasing the edge connectivity by one k times, does not imply that this is an optimal solution for the edge connectivity augmentation problem when $k > 1$.

Returning to the example of the graph G, as in figure 6.2. The edge connectivity of G can be increased by one in the following steps. **Step 1**: The cactus $\mathcal{H}(G)$, as in figure 6.16, has already been constructed. **Step 2**: Split the tree-edge $(2, 3)$ into two parallel edges, then gives the Eulerian graph $\mathcal{H}^*(G)$, as in figure 6.19, with an Eulrian tour as shown in figure 6.20. The leaves of the cactus $\mathcal{H}(G)$ are then enumerated, as in figure 6.21, in the order they first appear in the Eulerian tour of $\mathcal{H}^*(G)$. Node 2 and the empty nodes are not leaves as they do not have degree equal to one or have exactly two incident cycle-edges.

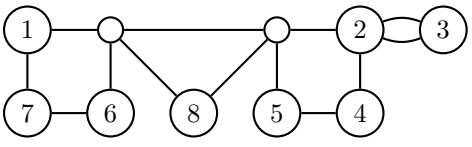


Figure 6.19: Eulerian graph $\mathcal{H}^*(G)$.

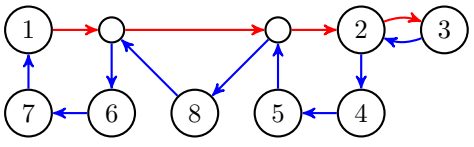


Figure 6.20: Eulerian tour in $\mathcal{H}^*(G)$.

U_i	U_1	U_2	U_3	U_4	U_5	U_6	U_7
Vertices	1	3	4	5	8	6	7

Figure 6.21: Set of vertices U_i of G which map to the leaf u_i.

Step 3: $k = 7$ hence, form the pairs (U_1, U_5), (U_2, U_6) and (U_3, U_7). **Step 4**: Using these pairs, choose an arbitrary vertex for each U_i, in each pair, gives the edges $(1, 8), (3, 6)$ and $(4, 7)$. As k is odd, U_4 has an arbitrary choice of which U_j $(j \neq 4)$ to be paired. Pairing U_4 with U_3 then gives the edge $(5, 4)$. Adding all of these edges to the graph G, optimally increases the edge connectivity of G by one, as in figure 6.22.

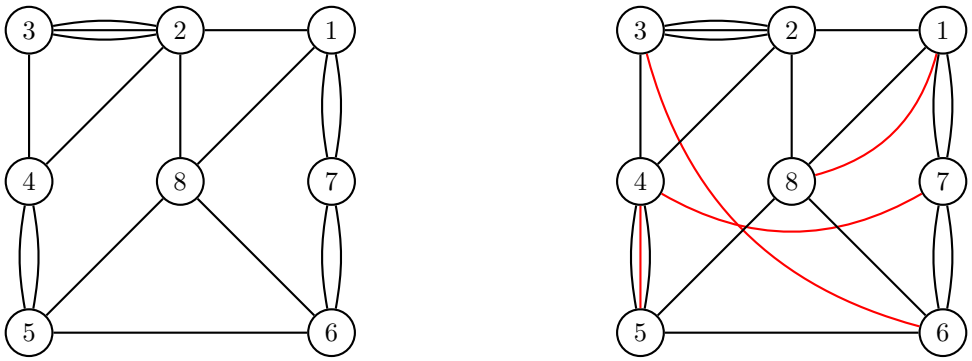


Figure 6.22: Graph G, with $\lambda(G) = 4$ (left). Optimal edge augmentation by one of G (right) with optimal edges highlighted in red.

Chapter 7

Conclusion

In studying the basic notions of vertex and edge connectivity this report has demonstrated some of the fundamental ideas for calculating, relating and increasing connectivity invariants. Flow networks were also established and shown to have a duality relationship to these connectivity invariants through Menger's Theorem, Whitney's Theorem and the Max-Flow Min-Cut Theorem. Then in solving the maximal flow problem through the Ford-Fulkerson Labelling Algorithm, these relations allowed the explicit calculation of both vertex and edge connectivity. Whitney's Inequality then gave us another relation between these connectivity invariants and minimal vertex degree, from which the idea of maximal connectivity and constructing graphs with specific connectivity stemmed. The problem of optimally increasing the edge connectivity by one was then solved by introducing the idea cactus representations.

Further to the topics discussed in this report; a structure similar to cactus representation, the *Gomory-Hu tree* [23, p. 46-50] is one of many structures required to solve the general edge augmentation problem. Each graph has a Gomory-Hu tree which encodes the minimal capacity cuts between two vertices of graph, into at least one of edges on the path joining these vertices in the tree. These trees then find a solution to the maximal flow problem with time complexity $\mathcal{O}(|V|\log(|V|))$ [23, p. 46].

There are also various other invariants measuring connectivity of graphs. One of these invariants is *Cheeger's constant* [2, p. 70] (or "edge expansion rate"), which is an isoperimetric invariant measuring the ratio between boundaries and interiors of all subsets of the network. This invariant is closely related to eigenvalues of the adjacency matrix and associated spectral gaps. This is a highly active research topic in the area of "Spectral Graph Theory" with various practical applications, such as the *page rank algorithm* [2, p. 59] used by Google or *spectral clustering* [2, p. 61] of networks using eigenvectors. Cheeger's constant also leads naturally to *expander graphs* [2, p. 70], a topic with various relations to deep mathematical theories like number theory (*Ramanujan graphs* [2, p. 68]), representation theory and geometric group theory (*Cayley graphs* [2, p. 94]).

Bibliography

[1] Borchardt, Carl W. *Über eine Interpolationsformel für eine Art Symmetrischer Functionen und über Deren Anwendung.* Math. Abh. der Akademie der Wissenschaften zu Berlin, pp.1-20, 1860.

[2] Brouwer, Andries E. Haemers, Willem H. *Spectra of Graphs.* Springer, New York, 2012.

[3] Borůvka, Otakar. *O jistém Problému Minimálním.* Práce mor. přírodověd. spol. v Brně III, **3**, pp.37-58, 1926.

[4] Borůvka, Otakar. *Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí.* Elektronický Obzor, **15**, pp.153-154, 1926.

[5] Cayley, Arthur. *A Theorem on Trees.* Quart. J. Pure Appl. Math. **23**, pp.376-378, 1889.

[6] Chartrand, Gary. Harary, Frank. *Graphs with Prescribed Connectivities.* Theory of Graphs : Proceedings of the Colloquium held in Tihany, Hungary, in September 1966. Academic Press New York, pp.61-63, 1968.

[7] Cormen, Thomas H. Leiserson, Charles E. Rivest, Ronald L. Stein, Clifford. *Introduction to Algorithms.* Third Edition, MIT Press and McGraw-Hill, 2009.

[8] Dijkstra, Edsger W. *A Note on Two Problems in Connexion with Graphs.* Numerische Mathematik, **1** (1), pp.269-271, 1959.

[9] Dinitz, Yefim (previously Efim). Karzanov, Alexander V. Lomonosov, Michael V. *On the Structure of a Family of Minimum Weighted Cuts in a Graph.* [In Russian] "Studies in Discrete Optimization" Nauka Publishers, pp.290-366, 1976. [English translation by Alexander V. Karzanov : `http://alexander-karzanov.net/ScannedOld/76_cactus_transl.pdf`]

[10] Edmonds, Jack. Karp, Richard. *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems.* Journal of the ACM, **19** (2), pp.248-264, 1972.

[11] Elias, Peter. Feinstein, Amiel. Shannon, Claude E. *Note on Maximum Flow Through a Network.* IRE Transactions on Information Theory. **2** (4), pp.117–119, 1956.

[12] Erdős, Paul. Gallai, Tibor. *Gráfok előírt fokszámú pontokkal.* Matematikai Lapok, **11**, pp.264–274, 1960.

[13] Euler, Leonhard. *Solutio problematis ad geometriam situs pertinentis.* Commentarii Academiae Scientiarum Petropolitanae, **8**, pp.128-140, 1741.

[14] Evans, James R. Minieka, Edward. *Optimization Algorithms for Networks and Graphs.* Second Edition, Marcel Dekker, 1992.

[15] Fleischer, Lisa. *Building Chain and Cactus Representations of All Minimum Cuts from Hao-Orlin in the Same Asymptotic Run Time.* Journal of Algorithms, Academic Press, **33** (1), pp.51-72, 1999.

[16] Ford, Lester R. Fulkerson, Delbert R. *Maximal Flow Through a Network.* Canadian Journal of Mathematics, **8**, pp.399–404, 1956.

[17] Edited by : Gross, Jonathan L. Yellen, Jay. Zhang, Ping. *Handbook of Graph Theory.* Second Edition, CRC Press, 2014.

[18] Gusfield, Dan. Martel, Charles. Naor, Dalit. *A Fast Algorithm for Optimally Increasing the Edge Connectivity.* SIAM J. Comput, **26** (4), pp.1139–1165, 1997.

[19] Harary, Frank. *Graph Theory.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1972.

[20] Harary, Frank. Uhlenbeck, George E. *On the Number of Husimi Trees, I.* Proceedings of the National Academy of Sciences, **39** (4), pp.315-322, 1972.

[21] Hierholzer, Carl. Wiener, Christian. *Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.* Mathematische Annalen, **6**, pp.30-32, 1873.

[22] Husimi, Kodi. *Note on Mayers' theory of cluster integrals.* Journal of Chemical Physics, **18** (5), pp.682-684, 1950.

[23] Ibaraki, Toshihide. Nagamochi, Hiroshi. *Algorithmic Aspects of Graph Connectivity.* Cambridge University Press, 2008.

[24] Jarník, Vojtěch. *O jistém Problému Minimálním.* Práce Moravské Přírodovědecké Společnosti, **6** (4), pp.57-63, 1930.

[25] Jungnickel, Dieter. *Graphs, Networks and Algorithms.* Fourth Edition, Springer, 2013.

[26] Kameda, Tiko. Nagamochi, Hiroshi. *Canonical Cactus Representation for Minimum Cuts.* Japan J. Industrial Applied Mathematics, **11**, pp.343-361, 1994.

[27] Karzanov, Alexander. Timofeev, Evgeniy. *Efficient Algorithm for Finding all Minimal Edge Cuts of a Nonoriented Graph.* Cybernetics and Systems Analysis, **22** (2), pp.156-162, 1986.

[28] Kocay, William. Kreher, Donald L. *Graphs, Algorithms, and Optimization.* First Indian Reprint, CRC Press, 2013

[29] Kruskal, Joseph Bernard Jr. *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem.* Proceedings of the American Mathematical Society, **7** (7), pp.48-50, 1956.

[30] Menger, Karl. *Zur allgemeinen Kurventheorie.* Fundamenta Mathematicae, **10.1**, pp.96–115, 1927.

[31] Nakamura, Akira. Watanabe, Toshimasa. *Edge-Connectivity Augmentation Problems.* Journal of Computer and System Sciences, **35** (1), pp.96-144, 1987.

[32] Naor, Dalit. Vazirani, Vijay V. *Representing and Enumerating Edge Connectivity Cuts in $\mathcal{RNC}$.* Algorithms and Data Structures. Lecture Notes in Computer Science, Springer, **519**, pp.273-285, 1991.

[33] Prim, Robert C. *Shortest Connection Networks and Some Generalizations.* Bell System Technical Journal, **36** (6), pp.1389-1401, 1957.

[34] Prüfer, Heinz. *Neuer Beweis eines Satzes über Permutationen.* Arch. Math. Phys, **27**, pp.742-744, 1918.

[35] Schnorr, Claus-Peter. *Bottlenecks and Edge Connectivity in Unsymmetrical Networks.* SIAM J. Comput. **8** (2), pp.265–274, 1979.

[36] Shields, Rob. *Cultural Topology: The Seven Bridges of Königsburg, 1736.* Theory, Culture & Society, **29** (4–5), pp.43–57. 2012.

[37] Turing, Alan M. *On Computable Numbers, with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society, **s2-42**, pp.230-265, 1937.

[38] Whitney, Hassler. *Congruent Graphs and the Connectivity of Graphs.* American Journal of Mathematics **54**. pp.150-168, The John Hopkins University Press, 1932.

[39] Whitney, Hassler. *Non-separable and Planar Graphs.* Proceedings of the National Academy of Sciences of the United States of America, **17** (2). pp.125-127, 1931.

[40] Zwick, Uri. *The Smallest Networks on which the Ford-Fulkerson Maximum Flow Procedure may Fail to Terminate.* Theoretical Computer Science, **148**, pp.165-170. 1995.

[41] ONLINE: *Understanding and Configuring Spanning Tree Protocol (STP) on Catalyst Switches.* Cisco, 2006. URL: `https://www.cisco.com/c/en/us/support/docs/lan-switching/spanning-tree-protocol/5234-5.html` , accessed 30-12-2018.

[42] ONLINE: *Theorems in Graph Theory Lecture Slides.* Higher School of Economics, 2015. URL: `https://cs.hse.ru/data/2015/04/16/1095668662/Lecture_10.pdf` , accessed 01-02-2019.

[43] ONLINE: *Maps of Königsberg Bridges.* MacTutor History of Mathematics Archive, March 1 2000. URL: `http://www-history.mcs.st-andrews.ac.uk/Extras/Konigsberg.html` , accessed 13-01-2019.

Appendices

Appendix A

Python Implementation

The following python implementations are the authors work.

A.1 Prerequisites

All algorithms require the following to be included:

```
import numpy as np
from numpy import linalg as LA
```

The following is used in some algorithms and should also be included:

```
def gensimpgraph(n = 2, p = 0.5):
    """Generate an adjacency matrix with probability of an edge p.
    n: Number of vertices (greater than 1).
    p: probability of an edge.
    """
    adjmatrix = np.zeros((n,n), dtype=int)
    if n > 1:
        for row in range(0,n-1):
            for column in range(row + 1, n):
                if np.random.random() < p:
                    adjmatrix[row][column] = int(1)
    adjmatrix = adjmatrix + adjmatrix.T
    return adjmatrix


def complete(n):
    """Returns adjacency matrix of complete graph on n vertices.
    n: Number of vertices.
    """
    return np.ones((n,n)) - np.identity(n)

def converttolist(adjmatrix):
    """Converts adjacency matrix to list of edges and list of vertices.
    adjmatrix: Adjacency matrix.
    """
    rows = adjmatrix.shape[0]
    columns = adjmatrix.shape[1] #Could reuse row as always square matrix
    adjlist = []
    vertexlist = []
    for row in range(0,rows):
        vertexlist.append(row)
```

```
        for column in range(row + 1,columns):
            if adjmatrix[row][column] == 1:
                adjlist.append([row,column])
    return adjlist, vertexlist

def convertfromadjlist(adjlist, vertexlist):
    """Converts list of vertices to adjacency matrix.
    adjlist: list of undirected edges.
    vertexlist: list of vertices.
    """
    rows = len(vertexlist) #Accounts for isolated vertices.
    adjmatrix = np.zeros((rows,rows), dtype=int)
    for edges in adjlist:
        adjmatrix[edges[0]][edges[1]] = 1
    adjmatrix = adjmatrix + adjmatrix.T
    return adjmatrix

def degree(adjmatrix):
    """Returns list of degrees
    adjmatrix: adjacency matrix.
    """
    squared = np.dot(adjmatrix,adjmatrix)
    return [squared[i][i] for i in range(adjmatrix.shape[0])]

def componentsviamatrix(adjmatrix):
    """Finds connected components using only adjacency matrix.
    """
    compmatrix = np.copy(adjmatrix)
    rows = compmatrix.shape[0]
    columns = compmatrix.shape[1]
    for i in range(rows):
        compmatrix[i][i] = i
    tracker = 0
    for row in range(0,rows -1):
        for column in range(row+tracker+1, columns):
            if compmatrix[row][column] == 1:
                compmatrix = swap(compmatrix, column, row+tracker+1)
                tracker = tracker + 1
        tracker = max(tracker - 1, 0)
    components = [[0]]
    for column in range(1,rows):
        for row in range(0, column):
            if compmatrix[row][column] == 1:
                components[-1].append(compmatrix[column][column])
                break
            elif row == column - 1:
                components.append([compmatrix[column][column]])
    return components

def swap(adjmatrix, vertexa, vertexb):
    """Swaps vertices in adjmatrix"""
    rows = adjmatrix.shape[0]
    swapmatrix = np.identity(rows, dtype=int)
    swapmatrix[vertexa][vertexa] = 0
    swapmatrix[vertexb][vertexb] = 0
    swapmatrix[vertexa][vertexb] = 1
    swapmatrix[vertexb][vertexa] = 1
    return np.dot(swapmatrix,np.dot(adjmatrix, swapmatrix))
```

A.2 Graphic Algorithm

```
def graphicalgorithm(degreeset):
    """Graphic Algorithm, returns False if not possible, otherwise returns
    adjacency matrix
     degreeset: List of n degrees of n vertices.
     """
    #Finds largest degree vertex.
    maxd = max(degreeset)
    #Makes a list of m lists, m = max(degreeset) + 1.
    degleft = [[] for i in range(maxd + 1)]
    #Creates an empty edge set.
    edgeset = []
    #Creates a store for used Vertices for use later in algorithm.
    used = []
    #Creates a memory store for later use.
    usedstore = []
    #Sets the number of empty lists from the final list = 0
    emptydeg = 0
    #A store for degree of "activate" vertex
    degreeplaceholder = 0
    #Creates a store for "activated" vertex number. (for use in populating the
    edge set)
    vertexplaceholder = -1
    #Places vertices into the list with index equal to its desired degree.
    for i in range(len(degreeset)):
        degleft[degreeset[i]].append(i)
    #Until the zero index list contains all the vertices, loop.
    while len(degleft[0]) != len(degreeset):
        #Finds the highest degree vertex (activated vertex), stores its degree (d)
     and its number (v).
        #Moves it from the highest degree list to the zero list
        degreeplaceholder = maxd - emptydeg
        vertexplaceholder = degleft[maxd - emptydeg][0]
        degleft[0].append(degleft[maxd - emptydeg][0])
        del degleft[maxd - emptydeg][0]
        #Loops until moved d vertices down 1 list.
        while degreeplaceholder != 0:
            #Checks to see if the current highest degree list is empty or the
    highest degree list is the 0 degree list.
            if len(degleft[maxd - emptydeg]) == 0 or maxd - emptydeg == 0:
                #Find the next none empty degree list
                while len(degleft[maxd - emptydeg]) == 0:
                    emptydeg +=1
                #If the algorithm is in the zero list, then cannot be done as
    their are still vertices to be moved but none can.
                if maxd - emptydeg == 0:
                    return False #Returns that not graphic
            #If we encounter something we have used before with the activated
    vertex, then delete it during this loop.
            while degleft[maxd - emptydeg][0] in used:
                #Stores the data on the removed vertex
                usedstore.append([degleft[maxd - emptydeg][0],maxd - emptydeg,
    emptydeg])
                del degleft[maxd - emptydeg][0]
                #Again does the check to find next none empty degree set
                if len(degleft[maxd - emptydeg]) == 0:
                    while len(degleft[maxd - emptydeg]) == 0:
                        emptydeg +=1
                    if maxd - emptydeg == 0:
                        return False #Returns that not graphic
```

```
            #Gets a vertex to be active with the activeted vertex, (highest degree unused at start of degree list)
            #Moves it down 1 degree list towards the 0 degree list, adds this to the used list but not the used stored memory
            activevertex = degleft[maxd - emptydeg][0]
            degleft[maxd - emptydeg - 1].append(activevertex)
            del degleft[maxd - emptydeg][0]
            #Reduces vertices needed by 1
            degreeplaceholder -= 1
            #Adds the activated and the active vertex to the edge list
            edgeset.append([vertexplaceholder, activevertex])
            used.append(activevertex)
        #Before next loop, re add all vertices which were deleted and reset the highest none empty degree list.
        for i in usedstore:
            degleft[i[1]].append(i[0])
            emptydeg = min(emptydeg, i[2])
        #Reset the used and usedstore lists
        used = []
        usedstore = []
        #Checks to see if current none empty degree set is empty. No need to check if at zero degree list here.
        if len(degleft[maxd - emptydeg]) == 0:
                while len(degleft[maxd - emptydeg]) == 0:
                    emptydeg +=1
    #Converts the edge set and a generated vertex set into adjmatrix and returns.
    return convertfromadjlist(edgeset, [i for i in range(len(degreeset))])

#Example
g = graphicalgorithm([1,1]) #Returns g = [[0,1],[1,0]].
n = graphicalgorithm([3,3,3]) #Returns n = False.
```

A.3 Prim's Algorithm

```
def prims(E, W, V):
    """Prims's Algorithm: Takes an Edge List E with Corresponding List of Weights W
        and set of Vertices V and returns false if minimal spanning tree does not exist
        or returns edge list, vertex list and corresponding list of weights
        of minimal spanning tree
        E: Edge list
        W: List of weights (E[i] has weight W[i])
        V: List of Vertices
    """
    tree = [V[0]] #Starting vertex
    edges = [[E[i], W[i]] for i in range(len(E))] #Combine edges and weights
    spanningedges = [] #List for edges
    weight = [] #List for final weights
    considered = [] #List of edges to be considered
    while (len(edges) != 0) and (len(spanningedges) != (len(V) - 1)):
        considered = []
        for i in range(len(edges)):
            #Goes through edges in reverse.
            if (edges[len(edges) -1-i][0][0] in tree) and (edges[len(edges) - 1-i][0][1] in tree):
                edges.pop(len(edges) -1-i) #Edge no longer needs to be considered for any tree.
            elif (edges[len(edges) -1-i][0][0] in tree) or (edges[len(edges) - 1-i][0][1] in tree):
                considered.append(edges[len(edges) -1-i]) #Edge will be considered for tree.
            else:
                pass
        if considered == []:
            return False #disconnected graph
        considered.sort(key=lambda x: x[1]) #sorts considered edges by weight
        tree.append(considered[0][0][0])
        tree.append(considered[0][0][1])
        spanningedges.append(considered[0][0]) #adds lowest weight edge
        weight.append(considered[0][1]) #adds weight
    if (len(spanningedges) != (len(V) - 1)):
        return False
    else:
        return spanningedges, V, weight

#Example
E = [[0,1],[1,2],[2,3],[1,3]]
W = [1,2,2,1]
V = [0,1,2,3]
prims(E,W,V) #Returns [[0, 1], [1, 3], [1, 2]], [0, 1, 2, 3], [1, 1, 2]
```

A.4 Kruskal's Algorithm

```
def kruskal(E, W, V):
    """Kruskal's Algorithm: Takes an Edge List E with Corresponding List of
    Weights W and set of Vertices V and returns false if minimal spanning tree
    does not exist or returns edge list, vertex list and corresponding list of
    weights of minimal spanning tree.
        E: Edge list
        W: List of weights (E[i] has weight W[i])
        V: List of Vertices
    """
    trees = [[i] for i in V] #Starts a tree at each vertex
    edges = [[E[i], W[i]] for i in range(len(E))] #combines edges and weight
    edges.sort(key=lambda x: x[1]) #Sorts edges by weight
    spanningedges = [] #List for edges
    weight = [] #List for final weights
    while (len(edges) != 0) and (len(spanningedges) != (len(V) - 1)):
        start = edges[0][0][0] #start of lowest weight edge
        end = edges[0][0][1] #end of lowest weight edge
        starttree = -1 #which tree does the start point belong
        endtree = -1 #which tree does the end point belong
        #Checks if in same tree
        for i in range(len(trees)):
            if start in trees[i]:
                if end in trees[i]:
                    edges.pop(0) #This edge would create a cycle
                    break
                else:
                    starttree = i #Found start point
            elif end in trees[i]:
                endtree = i #Found end point
            else:
                pass #Do nothing
            #If Both vertices found in different trees, add an edge between them.
            if (starttree != -1) and (endtree != -1):
                spanningedges.append([start, end])
                weight.append(edges[0][1])
                #join the trees together
                for i in trees[endtree]:
                    trees[starttree].append(i)
                trees.pop(endtree)
                edges.pop(0) #Edge no longer under consideration
                break
    if (len(spanningedges) != (len(V) - 1)):
        return False
    else:
        return spanningedges, V, weight

#Example
E = [[0,1],[1,2],[2,3],[1,3]]
W = [1,2,2,1]
V = [0,1,2,3]
kruskal(E,W,V) #Returns [[0, 1], [1, 3], [1, 2]], [0, 1, 2, 3], [1, 1, 2]
```

A.5 Prüfer Code

```
def prufercode(adjmatrix):
    """Takes in a connected tree and gives prufer code.
    adjmatrix: adjacency matrix corresponding to tree.
    """
    adj = np.copy(adjmatrix)
    degreeset = degree(adjmatrix)
    code = []
    while sum(degreeset) > 2:
        for i in range(len(degreeset)):
            if degreeset[i] == 1:
                degreeset[i] = 0
                for j in range(len(degreeset)):
                    if adj[i][j]==1:
                        code.append(j)
                        degreeset[j] -= 1
                        adj[i][j] = 0
                        adj[j][i] = 0
                        break
                break
    return code

def pruferdecode(code):
    """Given a prufer code, returns a graph as an adjacency matrix.
    code: prufer code given as tuple of n-2 integers.
    """
    v = len(code)+2
    adjmatrix = np.zeros((v,v), dtype = int)
    degreeset = [1 for i in range(v)]
    for i in code:
        degreeset[i] += 1
    for i in code:
        for j in range(v):
            if degreeset[j] == 1:
                adjmatrix[i][j] = 1
                adjmatrix[j][i] = 1
                degreeset[i] -= 1
                degreeset[j] -= 1
                break
    for j in range(v):
        if degreeset[j] == 1:
            for i in range(j+1,v):
                if degreeset[i] == 1:
                    adjmatrix[i][j] = 1
                    adjmatrix[j][i] = 1
                    break
            break
    return adjmatrix

#Example
prufercode(np.array([[0,1,1],[1,0,0],[1,0,0]])) #Returns [0]
pruferdecode([0]) #Returns array([[0, 1, 1], [1, 0, 0], [1, 0, 0]])
```

A.6 Ford-Fulkerson Labelling Algorithm

```
class networkgraph:
    def __init__(self, g, c, s, t):
        """Used for working with networks.
        g: List of ordered pairs of edges and vertices [[Edge list], [Vertex list]].
        c: Ordered list of capacity of each edge e in g[0] = [Edge List].
        s: Source Node s in G[1].
        t: Sink Node t in G[1].
        """
        self.g = g
        self.edges = self.g[0]
        self.vertices = self.g[1]
        self.c = c
        self.s = s
        self.t = t
        self.flowreset() #Zero Flow.

    def flowreset(self):
        """Resets flow in network to zero flow
        """
        self.f = [0 for i in self.g[0]] #Zero Flow.

    def maxflow(self):
        """Computes Max Flow"""
        self.fordfulkerson()
        value = 0
        for i in range(len(self.edges)):
            j = self.edges[i]
            if j[0] == self.s:
                value += self.f[i]
        return value

    def fordfulkerson(self):
        """ Applies ford fulkerson labelling algorithm until a maximal flow is obtained.
        """
        self.label = [["-", np.inf, self.s, "s"]] #[[start/end vertex, maximal increase in flow, end/start vertex, direction]]
        node = self.s #Starts at Source
        index = 0 #Looking at label 0
        labelled = [self.s]
        path = [self.t]
        pathsign = []
        while(self.t not in labelled):
            for i in range(0, len(self.edges)):
                #If forward edge and end of edge has not been labelled
                if (self.edges[i][0] == node and not(self.edges[i][1] in labelled)):
                    #If flow can actually be increased
                    if (self.c[i]-self.f[i] != 0):
                        value = min(self.c[i] - self.f[i], self.label[index][1])
                        self.label.append([node, value, self.edges[i][1] ,"+", i]) #Forward Edge
                        labelled.append(self.edges[i][1])
                #If backward edge and end of edge has not been labelelled
                elif (self.edges[i][1] == node and not(self.edges[i][0] in labelled)):
                    #If flow can actually be increased
                    if (self.f[i] != 0):
```

```
                        value = min(self.f[i], self.label[index][1])
                        self.label.append([node, value , self.edges[i][0] ,"-", i
    ]) #Backward Edge
                        labelled.append(self.edges[i][0])
                else:
                    pass
            #Looks at next oldest label
            index = index + 1

            #If next considered node is the sink t, do nothing
            if(self.label[index-1][2] == self.t):
                pass
            #If no more labels then no augmenting path
            elif (index + 1) > len(self.label):
                return False
            #Else more to search
            else:
                node = self.label[index][2]
        increase = self.label[-1][1]
        edges = []
        while path[-1] != self.s:
            find = path[-1]
            for i in self.label:
                if i[2] == find:
                    path.append(i[0])
                    edges.append(i[4])
                    pathsign.append(i[3])
                    break
        for i in range(len(edges)):
            if pathsign[i] == '+':
                self.f[edges[i]] += increase
            else:
                self.f[edges[i]] -= increase
        while self.fordfulkerson():
            pass
        return False

#Example
g = [[[0,1],[0,2],[2,3],[1,4],[3,4]],[0,1,2,3,4]]
network = networkgraph(g, [1 for i in gdash[0]], 0, 4) #All edges capacity 1.
   Source 0, Sink 4.
network.maxflow() #Returns 2
```

A.7 Edge Connectivity

```
def ctdn1 (adjmatrix):
    """ Convert simple graph (adjmatrix) to directed graph. (each undirected edge
    replaced with two directed edges) For Edge Connectivity.
    """
    rows = adjmatrix.shape[0]
    columns = adjmatrix.shape[1] #Could reuse row as always square matrix
    adjlist = []
    vertexlist = []
    for row in range(0,rows):
        vertexlist.append(row)
        for column in range(row + 1,columns):
            if adjmatrix[row][column] == 1:
                adjlist.append([row, column])
                adjlist.append([column, row])
    return adjlist, vertexlist

def edgeConnectivity(g):
    """ Computes Edge Connectivity lambda of a given simple graph.
    - Note: spelling lambda as lamda is intentional.
    g: Graph given by adjacency matrix.
    """
    if len(componentsviamatrix(g)) != 1:
        return 0
    n = g.shape[0]
    gdash = ctdn1(g)
    network = networkgraph(gdash, [1 for i in gdash[0]], 0, 1)
    network.s = 0
    network.t = n-1
    lamda = network.maxflow()
    for s in range(0,n-2):
        network.flowreset()
        network.s = s
        network.t = s+1
        mf = network.maxflow()
        lamda = min(mf, lamda) #lambda
    return lamda #lambda

#Example
edgeConnectivity(np.array([[0,1],[1,0]]) #Returns 1
```

A.8 Vertex Connectivity

```
def ctdn2 (adjmatrix):
    """ Convert simple graph to directed graph. For Vertex Connectivity.
    """
    adjlist1 , vertexlist1 = ctdn1(adjmatrix) #Converts simple to directed graph
    vdash = [i for i in vertexlist1]
    vdashdash = [(i + len(vdash)) for i in vertexlist1]
    edashdash = [[vdash[i] , vdashdash[i]] for i in range(len(vdash))]
    edash = [[vdashdash[edge[0]], vdash[edge[1]]] for edge in adjlist1]
    adjlist2 = [i for i in edash]
    for i in edashdash:
        adjlist2.append(i)
    vertexlist2 = [i for i in vdash]
    for i in vdashdash:
        vertexlist2.append(i)
    return adjlist2, vertexlist2

def vertexConnectivity(g):
    """ Computes vertex connectivity kappa of a given simple graph.
    g: Graph given by adjacency matrix.
    """
    if len(componentsviamatrix(g)) != 1:
        return 0
    n = g.shape[0]
    gdash = ctdn2(g)
    network = networkgraph(gdash, [1 for i in gdash[0]], 0, 1)
    vdash = gdash[1][:int(len(gdash[1])/2)]
    vdashdash = gdash[1][int(len(gdash[1])/2):]
    kappa = len(vdash) - 1
    s = -1
    while s < kappa:
        s = s + 1
        for t in range(s+1, len(vdash)):
            if not(([vdashdash[s],vdash[t]] in gdash[0]) or ([vdashdash[t],vdash[s
    ]] in gdash[0])):
                network.flowreset()
                network.s = vdashdash[s]
                network.t = vdash[t]
                m = network.maxflow()
                if m < kappa:
                    kappa = m
                if s > m:
                    return kappa
    return kappa

#Example
vertexConnectivity(np.array([[0,1],[1,0]]) #Returns 1
```

A.9 Prescribed Connectivities Algorithm

```
def genkld(kappa, lamda, delta):
    """Generates an adjacency matrix with kappa(G) = kappa, lambda(G) = lamda,
    delta(G) = delta.
       If kappa = 0 then lamda = 0.
       kappa: Vertex connectivity
       lamda: Edge connectivity
       delta: Lowest degree.
    """
    if kappa == 0 and lamda != 0: #kappa = 0 iff lamda = 0
        return False
    g1e, g1v = converttolist(complete(delta + 1)) #Generates a complete graph (
    edge list)
    g2v = [i + len(g1v) for i in range(delta + 1)] #Makes a copy of vertices and
    relabels
    g2e = [[edge[0] + len(g1v), edge[1] + len(g1v)] for edge in g1e] #Makes copy
    of edges and relabels
    kedges = [[g1v[i], g2v[i]] for i in range(kappa)] #Joins kappa edges between
    complete graphs
    ledges = []
    while len(ledges) != (lamda - kappa): #Joins lamda - kappa new edges between
    graphs
        for i in range(kappa):
            for j in range(delta - 1):
                if i != j:
                    ledges.append([g1v[i], g2v[j]])
                if len(ledges) == (lamda - kappa):
                    break
            if len(ledges) == (lamda - kappa):
                break
    #combine lists
    e = g1e + g2e + kedges + ledges
    v = g1v + g2v
    return convertfromadjlist(e, v) #converts lists to adjacency matrix

#Example
genkld(1,1,2) # Returns array([[0, 1, 1, 1, 0, 0], [1, 0, 1, 0, 0, 0], [1, 1, 0,
    0, 0, 0], [1, 0, 0, 0, 1, 1], [0, 0, 0, 1, 0, 1], [0, 0, 0, 1, 1, 0]])
```